Composite Synchronization in Parallel Discrete-Event Simulation

David M. Nicol and Jason Liu

Abstract—This paper considers a technique for composing global (barrier-style) and local (channel scanning) synchronization protocols within a single parallel discrete-event simulation. Composition is attractive because it allows one to tailor the synchronization mechanism to the model being simulated. We first motivate the problem by showing the large performance gap that can be introduced by a mismatch of model and synchronization method. Our solution calls for each channel between submodels to be classified as synchronous or asynchronous. We mathematically formulate the problem of optimally classifying channels and show that, in principle, the optimal classification can be obtained in time proportional to $\max\{C \times \log C, V \times N\}$, where C is the number of channels, V the number of unique minimal delays on those channels, and N is the number of submodels. We then demonstrate an implementation which finds an optimal solution at runtime and consider its performance on network topologies, including one of the global internet at the autonomous system level. We find that the automated method effectively determines channel assignments that maximize performance.

Index Terms—Synchronization, simulation, parallel processing, performance, optimization.

1 Introduction

It has long been recognized that discrete-event simulation is a challenging and important application area for parallel and distributed processing; its literature dates back to the late 1970's. Most of this is coached in terminology of "logical processes," "channels," "messages," and "lookahead." A logical process (also known now as a "timeline" to emphasize that all model entities reflected in this process use a common event-list) is a submodel that may run concurrently with other submodels; timelines communicate exclusively through time-stamped messages passed over channels. Synchronization is a fundamental issue. If we think of a timeline advancing its simulation clock and modifying its state with every event it executes, we must be concerned with the possibility of it receiving a message when its clock is z and the message should affect its state at time $y < z$. So-called "conservative" techniques ensure that no such straggler message ever arrives. Such techniques generally require *lookahead* on channels; a channel has lookahead x if a message sent over that channel at time y never affects the recipient before time $x + y$.

Parallel discrete-event simulation has proven successful in several application areas, most notably in aviation control [23], Markov chain simulation [15], architectural simulation [20], [4], [5], and telecommunications [3]. Nevertheless, every success involves some tuning of synchronization protocol to the model. This is one of several reasons why parallel discrete event simulation is viewed by many as a domain for experts only.

We have been working to make high-performance parallel discrete-event simulation more generally accessible with the development of the *Scalable Simulation Framework* (SSF) (see, www.ssfnet.org). SSF provides a simple API for Java and for C++; a number of independent SSF simulation libraries have been developed to support it. Like many other parallel simulation systems, SSF models are written in Java or C++ code that make calls to a kernel library. The library's API was designed from the start with parallel execution in mind, so that critical lookahead can be automatically extracted and exploited. The kernel described in this paper derives from the Dartmouth implementation of the C++ API, known as **DaSSF**. Like other SSF implementations, the principle application area for **DaSSF** has been communication networks. The present work is part of our on-going efforts to optimize DaSSF performance, while maintaining or increasing its flexibility.

The literature on conservative synchronization of parallel discrete-event simulations describes synchronous approaches based on barriers (e.g., [1], [12], [13]) and asynchronous approaches which govern a submodel's advance as a function of the advance of other submodels which may affect it (e.g., [2]). Synchronous approaches exploit the computational efficiency of global parallel operations like barriers and min-reductions and are valued for their simplicity and scalability. The simplicity frequently comes at the cost of overly pessimistic assumptions about connectivity, e.g., that any timeline can at any instant create an event that might affect every other timeline. On some simulation models, this can thwart exploitation of parallelism. In contrast, an asynchronous approach focuses its attention only on timeline interactions that the topology indicates can occur, but is subject to significant overhead costs on timelines that are highly connected.

We can view a user's description of a model as a directed graph of entities, connected by links. Entities communicate by message-passing, over the links. Each link is marked with a minimal latency time for any message sent over it.

• The authors are with Computer Science Department, Dartmouth College, 6211 Sudikoff Laboratory, Hanover, NH 03755.
E-mail: {nicol, jasonliu}@cs.dartmouth.edu.

Manuscript received 17 Jan. 2001; accepted 20 Apr. 2001.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 113495.

Both synchronous and asynchronous approaches rely on lookahead. Latencies of individual messages across a link may vary, so long as each is at least as large as the minimum, assumed to be known at the time the simulation is initialized. We will be concerned with an aggregated view of the model, where sets of entities are organized into *timelines*; simulation activity of all entities on a timeline is serialized. A channel from timeline t_i to t_j is an aggregated representation of all links from entities aligned on t_i to entities aligned on t_j. The channel is marked with the smallest minimal latency among all links it represents.

From the synchronization protocol's point of view, the model is a graph whose nodes are timelines and whose directed weighted edges are channels. A barrier-based approach is sensitive to the minimum incoming edge weight in the entire timeline graph, whereas an asynchronous approach is sensitive to average node in- and out-degrees. A composite approach tries to avoid these sensitivities by expending synchronization effort only where (and when) needed. Channels with large latency can be handled synchronously, in bulk, while channels with low latency are handled asynchronously. In our approach, the set of channels is partitioned into a synchronous set and an asynchronous set. The simulation synchronizes globally every m_s units of simulation, where m_s is the smallest minimal latency among all channels in the synchronous set. Between barriers, every timeline uses an asynchronous approach *only on its asynchronous channels*. The key insight is that all overhead associated with synchronous channels is captured entirely in the barrier and that overhead depends solely on m_s. Tractable solution to the optimal channel partitioning problem relies upon this observation.

In this paper, we more explicitly describe this approach. We then construct a model of synchronization overhead costs and use it to formulate the problem of assigning channels to be synchronous or asynchronous as a mathematical optimization problem. We show that the problem can be solved in time proportional to $\max\{C \times \log C, V \times N\}$, where C is the number of channels, V is the number of unique minimal delay values on those channels, and N is the number of timelines. We then use both synthetic and observed topologies of the global internet to illustrate the performance advantages of our approach in the **DaSSF** system.

Variations of composing global and asynchronous styles have been considered in more specific contexts. In one of these, at a global synchronization point, timelines construct "appointment" schedules with other timelines. Here, an appointment is a simulation time at which two timelines agree to coordinate. Synchronization then proceeds in accordance with the appointment schedules (a timeline that might receive a message an appointment does not advance beyond that appointment time before its appointment partner reaches that point). A simulation's ability to construct an appointment schedule depends very much on the model being simulated. In [15], the appointments are derived from mathematical properties of Markov chains and, in [5], they are derived from observation of directly executed computer application code. Another compositional approach is motivated by parallel machines comprised of networked clusters of shared-memory-multiprocessors (SMP) [9]. An asynchronous algorithm coordinates all timelines within an SMP node and a globally synchronous algorithm coordinates different SMP nodes. The solution we develop in this paper is in some ways similar in spirit to the first variation, but is applicable in much more general contexts. The UPS system [16] comes close to some of the issues we address here. UPS had the capability of synchronizing different portions of the model differently, but never cast the problem as an optimization, nor could it have explored different assignments at runtime.

Other efforts that combine synchronization approaches include [19], where authors introduce a scheme that combines a conservative time window synchronization with Time Warp. Intracluster activities are synchronized optimistically while intercluster synchronization is handled conservatively. This approach and similar others [21], [22] combine conservatism and optimism in an effort to overcome the limitations of either one of them used alone.

The remainder of this paper is organized as follows: Section 2 gives some background needed to appreciate the optimization problem we consider and then Section 3 looks at empirical results that motivate our problem. Section 4 describes the composite synchronization method in detail, while Section 5 develops and solves the optimal channel assignment problem. We look at empirical results in Section 6 and present our conclusions in Section 7.

2 Background

Much of the literature considers problems arising from the use of a parallel or distributed computing system to execute a discrete-event simulation. Temporal synchronization problems have always attracted significant attention. Consider the possibility of a timeline simulating up to time z and then receiving a message with time-stamp $y < z$. The timeline's state at z may be incorrect, as its computation did not incorporate the information contained in the newly received message. "Conservative" synchronization techniques prohibit this situation from ever arising; a timeline cannot advance to time z before the synchronization protocol can ensure that no further message with a time-stamp smaller than z will be received. "Optimistic" techniques recover from temporal errors after they occur. For the case at hand, the timeline rolls back to its state at time y and continues forward again, canceling all messages it formerly sent in its previous execution over interval $[y, z]$. Comprehensive surveys of various synchronization techniques include [7], [17], [14], [6], [11].

In this paper, we combine two particular conservative styles. The synchronous style uses barrier synchronization. If all timelines are synchronized at time z and it is known that z' is a lower bound on the earliest time a future message generated by *any* timeline might affect *any other* timeline, the timelines are free to simulate up to time z' without further coordination among themselves. Upon synchronizing at z', they exchange messages generated by the last execution burst, identify the next synchronization time z'', and continue as before. In its simplest form, the simulation identifies the least minimal latency m between any pair of timelines and synchronizes globally every m units of simulation time.

In an asynchronous style, a timeline bases its decision to simulate forward entirely on predicted future behavior of timelines that may affect it. While the approach we develop applies to many asynchronous protocols, for concreteness, we will discuss it in the context of a particular asynchronous protocol, based on Critical Channel Traversal [24]. We describe our modification to this protocol now in more detail. Let $\mathcal{T}$ be the set of all timelines and $\mathcal{C}$ be the set of all channels between them. For each $t_i \in \mathcal{T}$, we define $clock_i$ to be the clock value of t_i. A timeline's clock indicates the position of its state in simulation time. For each $c \in \mathcal{C}$, we define $l(c)$ to be the minimal latency of c; this means that, if timeline t_i executes an event that can affect t_j's state through channel c, then at least $l(c)$ units of simulation time pass before t_j's state is affected. A classic example of this is latency on a communication channel—the recipient's state is not affected until the message actually arrives. We associate a *channel time* $z(c)$ with each channel c, maintained by the channel's source timeline to be equal to the sum of the source's clock with $l(c)$. A timeline observing $z(c)$ knows that it already has all information from the source that could affect its state up to time $z(c)$. Timeline t_i marks the channel from timeline t_j as *critical* if t_i cannot advance until t_j advances the channel time. Each timeline t_i has a *critical counter* x_i reflecting the number of channels whose times must advance before t_i is able to advance again. Finally, we let $I(t_i)$ be the set of in-channels for t_i, those from which t_i reads events. We let $O(t_i)$ be the set of out-channels for t_i, those over which t_i writes events.

Algorithm 1 Timeline Logic in CCT Algorithm.

```
1. Compute the safe time $h_i$ :
       $$h_i = \min_{c \in I(t_i)} \{z(c)\}.$$
2. Accept all new events sent to $t_i$ from other timelines and merge them into the event list.
3. Execute events until the timeline's event-list is empty, or the time-stamp of the least-time event is as large as $h_i$.
4. Set $clock_i = h_i$.
5. For every $c \in O(t_i)$, set $z(c) = clock_i + l(c)$. If $c$ is marked as critical, then clear the mark and decrement $x_j$, where $c \in I(t_j)$. If $x_j = 0$ after the decrement, then schedule $t_j$ to run again.
6. For every $c \in I(t_i)$, if $z(c) = h_i$, then mark $c$ as critical and increment $x_i$. If no channels are marked by this loop, return to Step 1.
7. Suspend.
```

When a scheduled timeline runs, it executes the logic expressed in Algorithm 1. In Step 1, the least channel time h_i among all incoming channels gives a lower bound on the time at which *any* future event may be delivered to the timeline from another. The timeline therefore has already received all events necessary for it to simulate up to time h_i. It does so in Step 3 and then in Step 4 advances the timeline's clock to h_i because the timeline's state after processing the last message is what it must be just prior to time h_i. It is important to note that the timeline might not execute any events at all in Step 3, but still its clock advances to h_i. In Step 5, the timeline can update channel times because it has increased its clock time. Any timeline that is blocked waiting only for this increase is scheduled to run again. In Step 6, the timeline reassesses its safe time. If it has increased, the timeline is free to execute further and returns to Step 1; otherwise, the channels holding back the timeline are marked as critical, the critical count is set appropriately, and the timeline suspends. It is worth pointing out here that the overheads of Steps 1, 5, and 6 are proportional to the number of channels involved. This fact figures large in the problem addressed by this paper.

In our discussions, it will be important to remember that, while a timeline may be executed independently on one processor, a processor's workload may be composed of many timelines. Each processor runs a scheduler to select the next timeline to run. Throughout this paper, the scheduling policy used by our implementation is to choose the timeline whose current clock is least. There are often performance advantages to partitioning a model into more timelines than processors, as it gives more flexibility in load-balancing (both static and dynamic) and in latency hiding. Timelines are threads (our simulation engine is process-oriented) and all the reasons for using threads, in general parallel computations, apply here also.

3 Motivating Example

Use of the wrong synchronization method for a model can have deleterious effects on performance, as can be shown empirically, using the **DaSSF** parallel simulator (see www.cs.dartmouth.edu/research/DaSSF). **DaSSF** can run fully asynchronously or fully synchronously. Comparison of these modes motivates composite synchronization.

Our example is a synthetic simulation model comprised of 1,024 nodes, (numbered 0 to 1,023), with two parameters. To maximally expose the overhead costs of synchronization, we first consider "nonaggregate" experiments where each node is treated by the simulation as an independent timeline (in a subsequent experiment, we will aggregate nodes into timelines). Parameter n governs connectivity—node i may send messages to n other nodes, those numbered $(i+1) \bmod 1024$ through $(i+n/2) \bmod 1024$ and those numbered $(i-n/2) \bmod 1024$ through $(i-1) \bmod 1024$. Each such connection has a minimum latency delay, nominally, 1 unit of simulation time. The edges from node 0 to node 1023 and from 1023 to 0 are allowed to have a "small" minimal latency, m, the second parameter. In this model, the actual latency experienced by every message is 1, even over channels with small minimal latencies.

The experiments we conducted use $n = 2, 8, 16, 32$ and $m = 0.001, 0.01, 0.1, 1.0$.

Every node maintains a pointer to a circular list of its out-channels. When it receives an event, it sends that event to the out-channel presently pointed to and moves the pointer up one position. The whole simulation is started at time zero, with every node i sending an event to the first out-channel in its list.

We illustrate the sensitivity of both protocols to node degree by examining performance as n changes. Fig. 1 plots the aggregate rate of message delivery (in wallclock time)

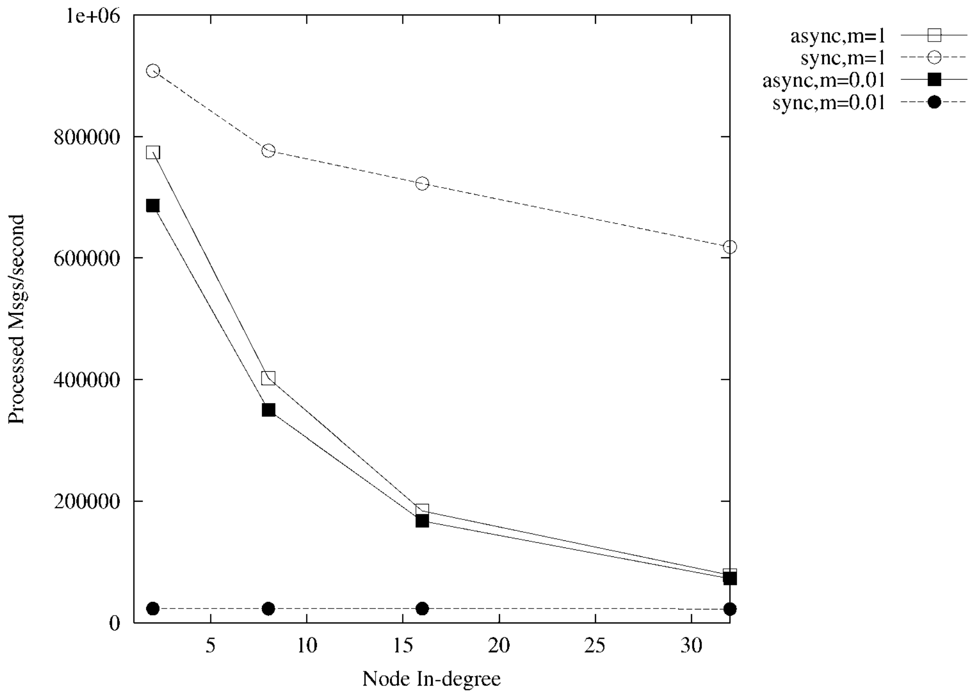

Fig. 1. Sensitivity to node degree, nonaggregate case.

when we run this model on eight processors of a Sun Enterprise 6500 server. Rates for both synchronous and asynchronous protocols are shown as n varies from 2 to 32 and for minimal latency (m) values of 1 and 0.01. For $m = 0.01$, the synchronous protocol is insensitive to node degree; for $m = 1$ performance decreases with increasing degree because of increased cache invalidations caused by the larger model size. The asynchronous protocol is very sensitive to node degree, as we expect. It is important to note that, on this model, the synchronous protocol is much more sensitive to small latency than is the asynchronous protocol. This fact creates the interesting situation that the choice of protocol for optimal performance depends very much on node degree and minimal latency. Either protocol is at some point in topology space at a severe performance disadvantage to the other.

It is also instructive to look at performance as the minimal latency value varies. Fig. 2 shows that both protocols have some sensitivity to it. This comes as no surprise; in this model, every message's real latency is 1, so that small minimal latency creates substantial synchronization overhead. The synchronous protocol is much more sensitive to minimal latency. It is worth noting that, even on this large model, a single cycle of edges with small minimal delay can significantly hold back the execution of the synchronous algorithm. Once again, the crucial point is that each protocol significantly dominates the performance of the other in some portion of topology space. As with the earlier data, we see here that asynchronous performance is very much better than synchronous performance when the node degree is small and the minimal latency is significantly smaller than the average latency. On the other hand, the synchronous protocol's performance is vastly better when the minimum latency is closer to the average latency and the node degree is high.

This set of data highlights overhead costs, by partitioning the model into as many timelines as possible. We can reduce the contribution of synchronization overhead to overall running time by using many fewer timelines. This comes at the cost of giving up flexibility in dynamic load-balancing and latency hiding, but, in this example, serves to show how performance is affected by protocol in models with a more realistic mixture of workload and overhead. We aggregate the nodes into 32 timelines of 32 nodes each, modularly, so that all nodes with the same remainder mod 32 are in the same timeline. This ensures that timelines have the same in- and out-degree as they do in the nonaggregated model and ensures that the "short edge" is exposed between timelines. Figs. 3 and 4 illustrate the same experiments as before, under this aggregation. As expected, there is a smaller difference in performance between synchronous and asynchronous and a greater overall level of performance. Nevertheless, we see again that, in some areas of topology space, the asynchronous method is significantly better than the synchronous method and, in other areas, the synchronous is significantly better.

While the model is artificial, it illustrates very clearly the dangers of committing to one style of synchronization for all models. It also suggests a direction of inquiry. Observe that the asynchronous model does comparatively better in contexts where the minimum latency is small relative to the average and the node degree is small. The synchronous model does comparatively better in contexts where the minimal latency is closer to the average and the node degree is high. The obvious question is whether, given a model, we can determine which synchronization approach achieves the best performance. The not-so-obvious question (but a better one, it turns out) is whether we can, in a single

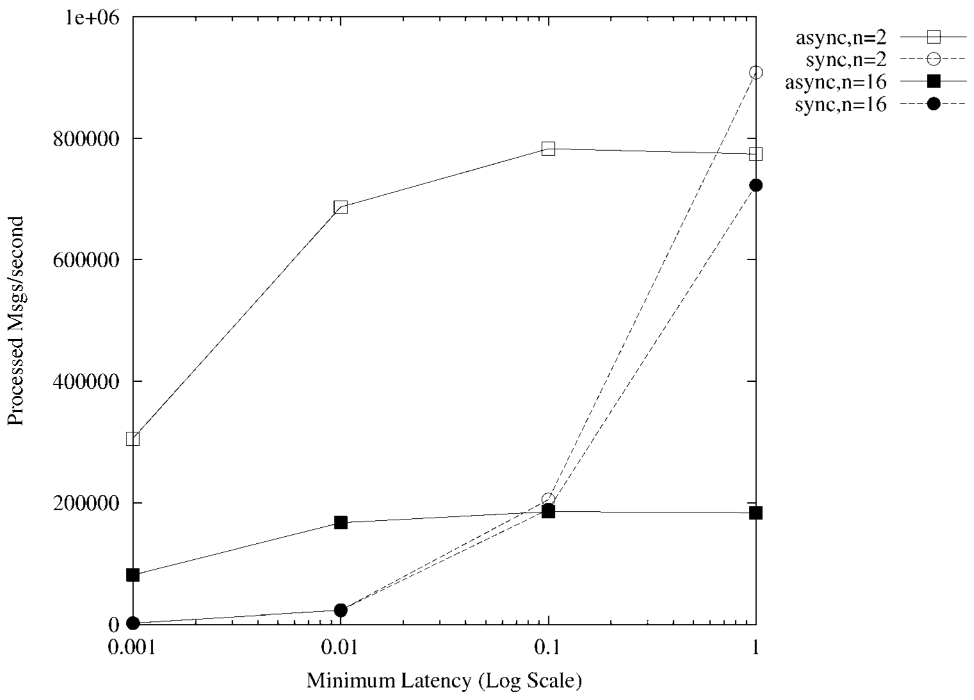

Fig. 2. Sensitivity to minimal latency, nonaggregate case.

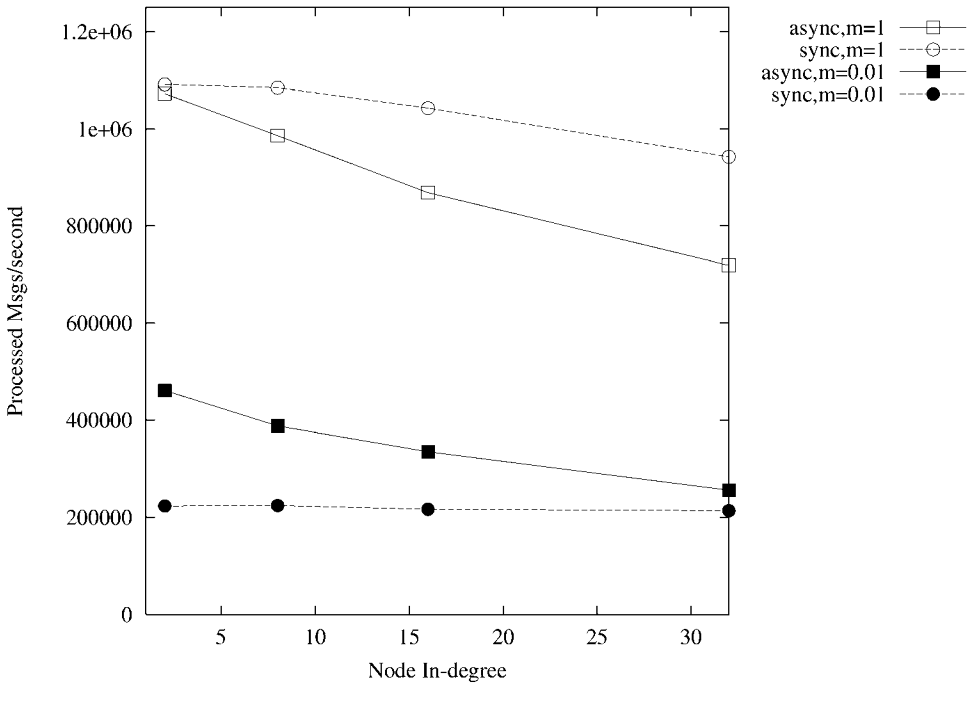

Fig. 3. Sensitivity to node degree, aggregate case.

model, combine synchronous and asynchronous approaches in a way that avoids the pitfalls of "pure" approaches. We can, by classifying each channel as synchronous or asynchronous. A global synchronization window is established as a function of the minimal latencies on synchronous channels and within a window the simulation progresses asynchronously interacting solely through the asynchronous channels. We call the synchronization method that combines synchronous and asynchronous treatment of channels *composite synchronization*.

4 Composite Synchronization

The basic idea behind composite synchronization is simple—every channel is classified as synchronous or asynchronous. The smallest minimum latency among all synchronous channels defines the width of a global synchronization window. When a timeline is scheduled to run, it computes its safe-time as the minimum of the next global synchronization time and the minimum channel time among all of its asynchronous in-channels. It executes as many events as it can up to its safe time and then updates

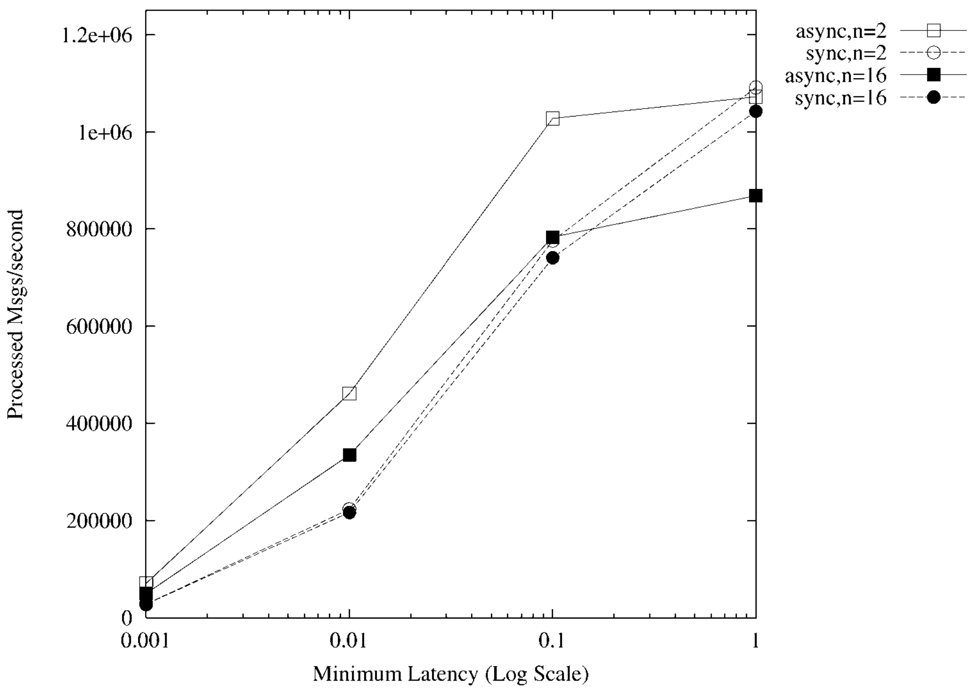

Fig. 4. Sensitivity to minimal latency, aggregate case.

the channel times on all of its asynchronous out-channels. If its safe time is identically the next global synchronization time, the timeline suspends itself. Otherwise, the logic proceeds just as the CCT algorithm described in Section 2, except that only asynchronous channels are involved. Timelines do not mark synchronous channels as critical and, consequently, do not need to check synchronous channels for critical flags.

Some additional notions are needed to describe this approach. Recall that $\mathcal{T}$ is the set of all timelines, that $\mathcal{C}$ is the set of all channels between them, and that, for every $c \in \mathcal{C}$, $l(c)$ denotes the minimal latency of messages sent across c. A function $\mathcal{P} : \mathcal{C} \rightarrow \{s, a\}$ partitions $\mathcal{C}$ into a synchronous set (those channels mapped to s) and an asynchronous set (those mapped to a). For every timeline t_i, we let $I_s(t_i, \mathcal{P})$ and $O_s(t_i, \mathcal{P})$ be the set of its in-channels (respectively, out-channels) marked as synchronous by $\mathcal{P}$ and, similarly, let $I_a(t_i, \mathcal{P})$ and $O_a(t_i, \mathcal{P})$ denote its asynchronous in-channels and out-channels. Timelines without any in-channels at all are considered to have a pseudo-in-channel in the synchronous class, with as large a minimal delay as may be desirable. This imposes necessary flow-control and keeps us from having to consider such timelines as special cases in our algorithm description and proof of correctness.

Recalling that $l(c)$ is the minimal latency on channel c, we define $m_s(\mathcal{P})$ to be the minimum value of $l(c)$ among all channels c marked as synchronous by $\mathcal{P}$. Whenever a timeline runs, it has access to variable G, which holds the simulation time of the next synchronous barrier. After a barrier, G advances by precisely $m_s(\mathcal{P})$. Thus, no event sent across a synchronous channel can arrive in the same synchronization window as that in which it is sent.

Algorithm 2 gives pseudocode of the logic used by a timeline t_i when it executes. Initially, every timeline is scheduled to run (and has clock time zero). The safe-time calculation is different in that scans are made only of the asynchronous channels; value G stands in for the channel times of all synchronous channels. Step 5 differs from its CCT counter-part in that only the channel times of asynchronous channels are updated and only those channels are scanned for critical flags. Likewise, Step 7 differs from its CCT counter-part in scanning and marking only asynchronous channels. Unlike the earlier CCT algorithm, in this one a timeline may suspend (at Step 6), without marking any channels as critical. During every synchronization window, every timeline ultimately suspends through Step 6; eventually, there will be no runnable timelines. Processors globally synchronize on this condition; once no processor has any runnable timelines, the schedulers increase G (by at least $m_s(\mathcal{P})$, possibly more, depending on an analysis of known future events), put every timeline on the runnable list, and continue on to the next global synchronization.

Algorithm 2 Timeline Logic in Composite CCT Algorithm.

```
1. Compute the safe time $h_i$ :
       $$h_i = \min\{G, \min_{c \in I_a(t_i, \mathcal{P})} \{z(c)\}\}.$$
2. Accept all new events sent to $t_i$ from other timelines
   and merge them into the event-list.
3. Execute events until the timeline's event-list is
   empty, or the time-stamp of the least-time event is
   as large as $h_i$.
4. Set $clock_i = h_i$.
5. For every $c \in O_a(t_i, \mathcal{P})$, set $z(c) = clock_i + l(c)$. If $c$ is
   furthermore marked as critical and $c \in I_a(t_j, \mathcal{P})$, then
```

clear the mark and decrement x_j. If $x_j = 0$ after the decrement, then put t_j in the runnable timeline queue.
6. If $h_i = G$, then suspend.
7. For every $c \in I_a(t_i, \mathcal{P})$, if $z(c) = h_i$, then mark c as critical and increment x_i. If no channels are marked by this loop, return to Step 1.
8. Suspend.

Access to shared variables (x_i and critical markers) may be managed using locks, or through lock-free mechanisms as described in [10]. (The implementation described in this paper uses locks.) We do need to be concerned about deadlock because suspension through Step 6 alters the logic of the CCT algorithm. Correctness and freedom of deadlock of this algorithm can be established by induction on the number of barrier points. We argue that, for every $k = 0, 1, 2, \ldots,$ the simulation is correct and free from deadlock at the kth barrier. The base case of $k = 0$ is trivially satisfied. For the induction hypothesis, we suppose the simulation is correct and free from deadlock at the $(k-1)$st barrier. Following the barrier, every timeline is put on its processor's runnable list. We claim that, at the point the kth barrier is reached, every timeline t_i has suspended itself through Step 6. For the sake of contradiction, suppose not. This means deadlock has developed among timelines on asynchronous channels, implying that there exists a cycle of timelines on channels marked as critical, with the same common channel time. Among all such cycles, choose one whose channel time is least, say $y < G$. Every cycle in the timeline graph must have at least one channel with nonzero minimal latency, so let t_i be the source on that channel and t_j be the destination. Observe that $clock_i < clock_j = y$, which means that the CCT algorithm has failed to schedule t_i to scan its own in-channels, for such a scan must allow t_i to increase $clock_i$ at least to y and clear the critical flag on the channel to t_j. However, the simulation behavior up to time y is purely that of CCT (on asynchronous channels) and the correctness of CCT been proven elsewhere. This establishes the contradiction and completes the induction.

5 Optimization Problem

We can now formulate the channel partitioning optimization problem. We first construct an analytic model of synchronization overhead that focuses on channel-specific overhead. The objective function adds together overhead from all processors, as a function of channel partition; we seek the partition that minimizes this cost function. This is only a heuristic in that we cannot guarantee that overall performance is optimized by this objective function. The overall performance is affected by other factors and it is sometimes possible to achieve better overall performance by expending more rather than less time in synchronization logic, if by doing so better parallelism is revealed.

We suppose there is an execution cost C_{scan} of interacting with a channel—either reading its channel time or setting its channel time. We suppose there is a cost C_{sync} of detecting when every scheduler's runnable timeline list is empty and a cost C_{sched} of putting a timeline on the runnable queue. Recalling that $m_s(\mathcal{P})$ denotes the minimal latency among all synchronous channels and assuming that the global window advances only by $m_s(\mathcal{P})$ each window, we see that the *rate* (per unit simulation time) at which the overhead related to executing barrier logic accumulates is $C_{sync}/m_s(\mathcal{P})$. This term accounts for all barrier costs on all processors.

Algorithm 2 shows us that the costs associated with scanning asynchronous channels lie in two scans of $I_a(t_i, \mathcal{P})$ and one scan of $O_a(t_i, \mathcal{P})$ (assuming that "return to Step 1" in Step 7 is not exercised). Let $f_i(\mathcal{P})$ be the asymptotic average increase in t_i's clock value for each execution of this code. Then, the asymptotic rate (per unit simulation time) at which asynchronous overhead associated with t_i accumulates is

$$\frac{C_{sched} + C_{scan}(2|I_a(t_i, \mathcal{P})| + |O_a(t_i, \mathcal{P})|)}{f_i(\mathcal{P})}.$$

Here, we lump the cost of t_i being scheduled to run with the scanning overheads it suffers when it runs and exclude the scheduling costs in Step 5 because these are accounted for by the timelines that run as a result. Adding synchronous and asynchronous overhead rates, we obtain the overall overhead rate (summed over all processors) as (with $\alpha = 2$)

$$R(\mathcal{P}, \alpha) = \frac{C_{sync}}{m_s(\mathcal{P})} + \sum_{t_i \in \mathcal{T}} \left(\frac{C_{sched} + C_{scan}(\alpha|I_a(t_i, \mathcal{P})| + |O_a(t_i, \mathcal{P})|)}{f_i(\mathcal{P})} \right). \tag{1}$$

We parameterize this function by α to allow it to encompass another algorithm (to be described later) for which $\alpha = 1$. The optimization problem now is to find that partition $\mathcal{P}_{opt}$ which minimizes (1).

Two definitions help us describe conditions when this optimization problem is tractable.

Definition 1. *Let $\mathcal{P}$ and $\mathcal{P}'$ be two partitions. We say that $\mathcal{P}'$ is* weakly more synchronous *than $\mathcal{P}$ if $m_s(\mathcal{P}) = m_s(\mathcal{P}')$ and for every $t_i \in \mathcal{T}$, $I_s(t_i, \mathcal{P}) \subseteq I_s(t_i, \mathcal{P}')$.*

Intuitively, $\mathcal{P}'$ is weakly more synchronous than $\mathcal{P}$ if one can construct it from $\mathcal{P}$ by changing some channel assignments to synchronous, but without decreasing the minimum latency among synchronous channels.

The main theoretical result depends on a property of the synchronization strategy, captured by the following definition.

Definition 2. *Let $\mathcal{S}$ be a synchronization algorithm for parallel simulation. We say that $\mathcal{S}$ is* WS-deterministic *if, whenever partition $\mathcal{P}'$ is weakly more synchronous than $\mathcal{P}$, then for each timeline t_i and every $k = 1, 2, \ldots,$ the kth execution of t_i advances $clock_i$ under $\mathcal{P}'$ exactly as far as does the kth execution of t_i under $\mathcal{P}$.*

Note that this definition is not tied to Algorithm 2. We will say more about this definition after we use it in the lemma that is at the heart of the main result.

Lemma 1. *Let $\mathcal{P}$ and $\mathcal{P}'$ be two partitions such that $\mathcal{P}'$ is weakly more synchronous than $\mathcal{P}$ and let synchronization algorithm $\mathcal{S}$*

be WS-deterministic and have overhead costs modeled by (1). Then, under $\mathcal{S}$ for any α, $R(\mathcal{P}', \alpha) \leq R(\mathcal{P}, \alpha)$.

Proof. Consider (1). By WS-determinism, $f_i(\mathcal{P}') = f_i(\mathcal{P})$ for every timeline t_i. By weak synchrony, $|I_a(t_i, \mathcal{P}')| \leq |I_a(t_i, \mathcal{P})|$ and $|O_a(t_i, \mathcal{P}')| \leq |O_a(t_i, \mathcal{P})|$ for every timeline t_i. Therefore, under $\mathcal{P}'$, for every t_i, the term representing its cost in the sum over timelines $\mathcal{T}$ is no larger than that term is under $\mathcal{P}$. Therefore, $R(\mathcal{P}', \alpha) \leq R(\mathcal{P}, \alpha)$. □

The lemma does all the work needed for the main result, which states that the optimal partition is a *threshold* policy—there exists a threshold T such that all channels with minimum latency less than T are asynchronous and all channels with minimum latency T or greater are synchronous.

Theorem 1. *Choose any channel $c \in \mathcal{C}$ and consider the constraint that all synchronous channels have a minimal delay of at least $l(c)$. Under these constraints and a WS-deterministic synchronization algorithm $\mathcal{S}$ whose costs are modeled by (1), the partition $\mathcal{P}_{opt}(l(c))$ that minimizes (1) categorizes every channel c' with $l(c') \geq l(c)$ as synchronous and all others as asynchronous.*

Proof. Partition $\mathcal{P}_{opt}(l(c))$ is weakly more synchronous than any other partition $\mathcal{P}$ with $m_s(\mathcal{P}) = l(c)$. Then, by Lemma 1, for any α, $R(\mathcal{P}_{opt}(l(c)), \alpha) \leq R(\mathcal{P}, \alpha)$. □

It is evident that the real force behind the result is the assumed WS-deterministic properties of $\mathcal{S}$. We can show though that assumption of this property is not vacuous. But, first, we note that Algorithm 2 is **not** WS-deterministic because a necessary (but insufficient) condition for WS-determinism is that a timeline advance its clock on the *k*th execution by the same amount, *every time the same simulation is run*. The behavior of Algorithm 2 depends on timing. To see this, just imagine a "producer" timeline and a "consumer" timeline, with a single directed link from producer to consumer, with a message produced every 10 units of simulation time. In one run of the simulation under Algorithm 2, the consumer could start by computing a safe time of 10; in another run where the consumer's execution is a little bit delayed, it could start by computing a safe time of 40 because the producer has computed farther ahead.

However, WS-deterministic synchronization algorithms do exist, as we can show. Consider a modification to Algorithm 2 (similar to that described in [18]) that limits the extent of an advance of a timeline t_i to be the minimum incoming asynchronous channel delay,

$$d(t_i, \mathcal{P}) = \min_{c \in I_a(t_i, \mathcal{P})} \{l(c)\}.$$

We require here that any channel c between timelines has $l(c) > 0$. A timeline t_i will be on the runnable list only if it is safe to advance $d(t_i, \mathcal{P})$ units of simulation time; therefore, an initial scan of input channels is unnecessary. At the end of a burst, the timeline updates channel times as before. It also scans its asynchronous predecessors to determine those which will need to advance in order for the timeline to execute again and marks its inchannels with those timelines as being critical.

Algorithm 3 Timeline Logic in Fixed Advance Algorithm.

```
1. Compute the safe time $h_i$ :
       $h_i = \min\{G, clock_i + d(t_i, \mathcal{P})\}.$
2. Accept all new events sent to $t_i$ from other timelines and merge them into the event-list.
3. Execute events until the timeline's event-list is empty, or the time-stamp of the least-time event is as large as $h_i$.
4. Set $clock_i = h_i$.
5. For every $c \in O_a(t_i, \mathcal{P})$, set $z(c) = clock_i + l(c)$. If $c$ is furthermore marked as critical and $c \in I_a(t_j, \mathcal{P})$, then clear the mark and decrement $x_j$. If $x_j = 0$ after the decrement, then put $t_j$ on the runnable timeline list.
6. If $h_i = G$, then suspend.
7. For every $c \in I_a(t_i, \mathcal{P})$, if $z(c) < clock_i + d(t_i, \mathcal{P})$, then mark $c$ as critical and increment $x_i$.
8. Suspend.
```

It is not difficult to see that this "Fixed Advance" algorithm is WS-deterministic because weak synchrony does not affect window size, nor does it affect a timeline's smallest minimal channel delay on incoming channels. Nor is it difficult to prove correctness and freedom from deadlock. However, in practice, we use Algorithm 2 rather than Algorithm 3 because the former tends to have longer execution bursts (and, hence, less overhead due to context switching). Nevertheless, Algorithm 3 is better when debugging because a timeline's behavior is absolutely deterministic (although the ordering of timeline executions may not be). The overhead cost function for Algorithm 3 under partition $\mathcal{P}$ is just $R(\mathcal{P}, 1)$, so that Theorem 1 holds.

In summary, Algorithm 3 provides an existence proof for the following result.

Theorem 2. *There exists a WS-deterministic parallel simulation algorithm.*

Having now shown that Theorem 1 is not vacuous, it can be used to identify the optimal partition. We simply compute the overhead cost for each conditionally optimal partition that it identifies and select the partition with least cost. To do this though, we must *predict* synchronization overhead and, hence, we must quantify the constants and find closed form expression for the values $f_i(\mathcal{P})$. In principle, determining the constants can be done empirically (e.g., see [8]). In the case of Algorithm 3, it is easy to compute $f_i(\mathcal{P})$ as the window size divided by the number of times the timeline is scheduled in the window:

$$f_i(\mathcal{P}) = \frac{m_s(\mathcal{P})}{\left\lceil \frac{m_s(\mathcal{P})}{d(t_i, \mathcal{P})} \right\rceil}.$$

A straightforward way of identifying the optimal partition starts by sorting all the channels into increasing order of minimum latency. We start with the partition $\mathcal{P}_s$ that makes every channel synchronous and compute the

$f_i(\mathcal{P}_s)$ values (for Algorithm 3, $f_i(\mathcal{P}_s) = m_s(\mathcal{P}_s)$ for every i.) For every timeline t_i, we initialize value $v_i = 0$ and compute

$$R(\mathcal{P}_s, \alpha) = \frac{C_{sync}}{m_s(\mathcal{P}_s)} + \sum_{t_i \in \mathcal{T}} \left(\frac{C_{sched} + C_{scan}\alpha v_i}{f_i(\mathcal{P}_s)} \right).$$

We store this value, then remove all channels with the least minimal latency from the sorted list. For each channel removed, we add 1 to v_i where t_i is the channel's source and add 1 to v_j where t_j is the channel's destination. We recompute the f_is for the new partition in time proportional to N, the number of timelines, and then calculate the cost of the new partition using the new v_i and f_i values, as above. Computing all values of f_i dominates. Continuing on in this fashion, we compute the cost of every conditionally optimal partition; the cost of the solution where all channels are asynchronous is obtained by setting the synchronous component of the cost function to zero. Having computed the cost of every solution that can be optimal, we identify the least cost solution. The initial sorting step requires order $C \times \log C$ time and each analysis of a partition requires order N time. There being $V + 1$ partitions to consider, (recall that V is the number of unique channel delays) the overall solution time cost is therefore proportional to $\max\{C \times \log C, V \times N\}$.

This relatively low complexity is of theoretical interest, but, in practice, quantifying the overhead cost function is delicate and machine dependent. However, the theoretical result points the way to an practical approach. Rather than predict synchronization cost as a function of threshold channel value, we can at runtime dynamically change channel assignments, *measure* which threshold value delivers peak performance, and lock down on that value. We explore this technique in the following section.

6 Experiments

We have implemented the composite synchronization algorithm in the **DaSSF** simulation system.[1] While the optimization result of the previous section is interesting and provides a sound theoretical basis for composite synchronization, our implementation automatically searches at runtime for the conditionally optimal channel assignment whose measured behavior appears to optimize performance. It does not engage in a static preanalysis startup cost.

When a model is loaded at initialization, the set of unique minimal delays on channels between timelines is discovered and then is sorted in increasing order. This list provides a set of thresholds; following Theorem 1, we will use each value w on the list to construct partition $\mathcal{P}_{opt}(w)$ and measure its performance. After working through each possible partition and the end-cases of fully synchronous and fully asynchronous partitions, we revert to the partition whose observed performance was best. We assess the quality of a partition by measuring the length of wallclock time required to simulate $2 \times d_{\max}$ units of simulation time, where $d_{\max}$ is the largest minimal delay among all channels.

1. See www.cs.dartmouth.edu/research/DaSSF.

Our implementation takes a brute force approach to reassigning channels and computes sets $I_a(t_i, \mathcal{P})$ and $O_a(t_i, \mathcal{P})$ from scratch every time partition $\mathcal{P}$ is changed, therefore, the cost of reassignment is linear in the size of the simulation model. With more sophisticated programming and precomputed data structures, this overhead could certainly be reduced. Our implementation simplistically tests every possible conditionally optimal partition. We easily imagine optimizations that limit the number of partitions tried, or that use a nonexhaustive search algorithm to find the best partition. In the examples we study in this paper, the number of unique values of minimum channel delays is small; indeed, in our experience with models of communication networks, the number of unique channel latencies tends not to be large. In the experiments we run, the time spent working through possible assignments is small relative to the length of the simulation run, on the order of 5 percent of the time or less. If a model had many different channel latencies, we could easily approximate the optimal assignment by "binning" channel latencies into a manageable number and then search over the minimum latency in each bin. One bin might have latencies from 0.1ms to 0.5ms, the next from 0.5ms to 0.75ms, and so on.

The first set of experiments reexamines the model we explored in Section 3 on the same architecture (eight processors of a Sun Enterprise 6500 multiprocessor). We present the data differently in that we take the performance of the fully synchronous protocol as a baseline and compute the "speedup" of other methods relative to that: the ratio of the message rate under the fully asynchronous protocol to that, of the fully synchronous protocol and, likewise, the ratio of the message rate under the automated composite approach to the fully synchronous protocol. Fig. 5 illustrates the same data as did Fig. 1 in these terms, including the performance of the composite protocol. Note that the y-axis is logarithmic. The results show that composite synchronization does exactly what we had hope for. In the case where all channels have the same latency ($m = 1$), it finds the fully synchronous assignment to be optimal, chooses it, and enjoys its performance benefit over the asynchronous protocol when the in-degree is large. However, the real benefit of the method is observed when $m = 0.01$. In that case, the asynchronous method does significantly better than the synchronous method, by factors of 32 to 3.5, depending on the node degree. But, here, the composite protocol performs well. It identifies as optimal the partition that treats the small latency channels asynchronously and all the rest synchronously. By this choice, it retains the insensitivity to node degree characteristic of the synchronous protocol, remaining significantly faster than *both* synchronous and asynchronous protocols for larger node degrees.

Fig. 6 likewise reexamines the data of Fig. 2 and includes composite protocol performance. In the case of in-degree $n = 2$, the comparative performance of the composite and synchronous protocols is close; here again, the composite solution makes the small latency channels asynchronous and the other channels synchronous. The small node degree

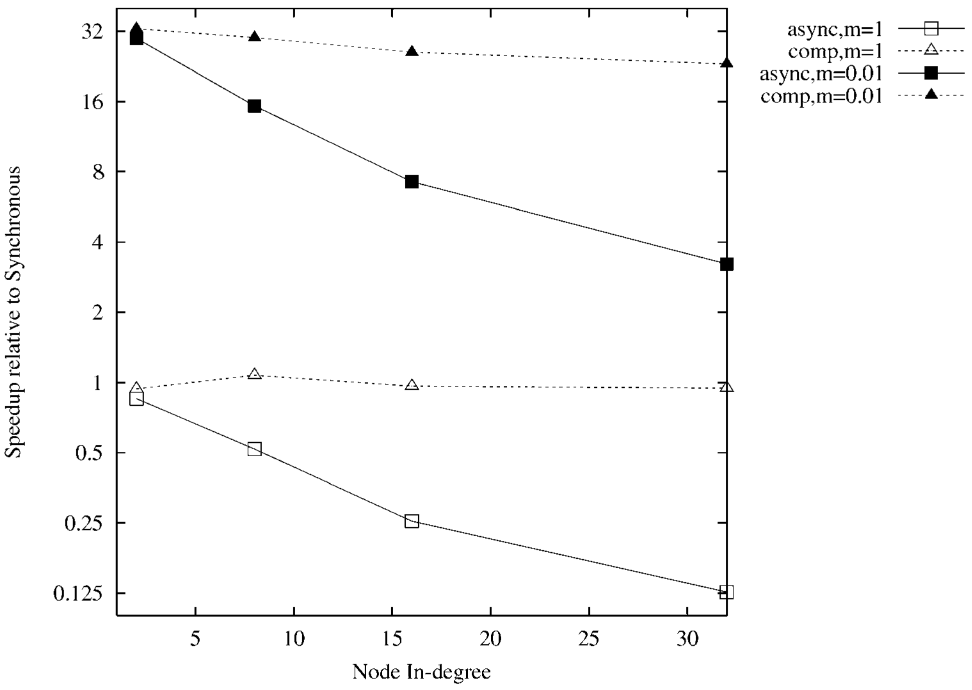

Fig. 5. Sensitivity of composite synchronization to node degree, nonaggregate case.

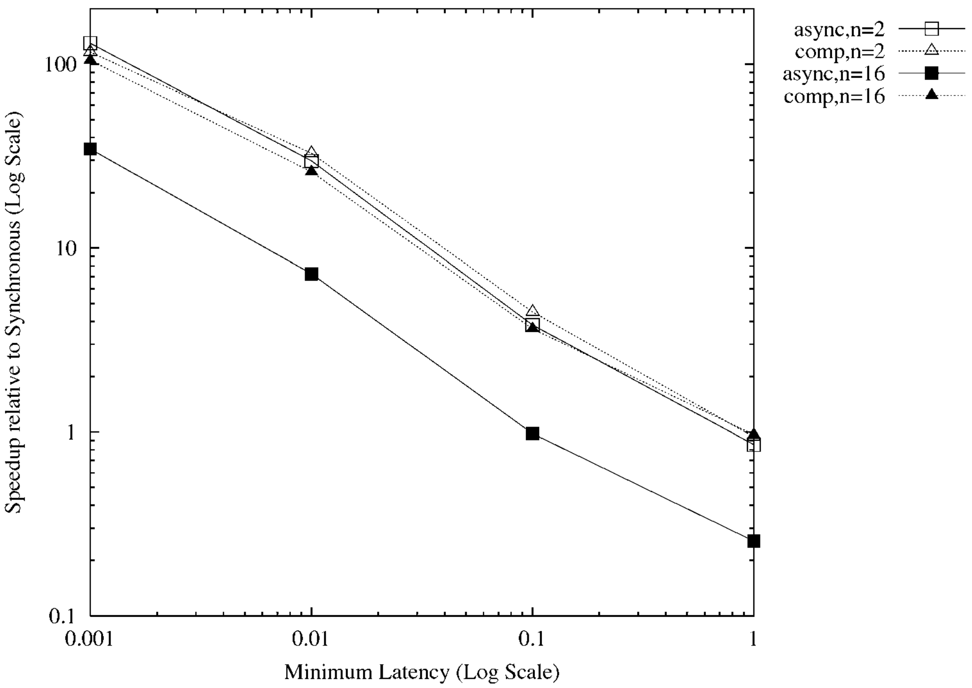

Fig. 6. Sensitivity of composite synchronization to minimal latency, nonaggregate case.

means the asynchronous protocol does not suffer large scanning overheads. The situation is very different though when the in-degree is 16. For very small minimum latencies, the asynchronous protocol is better than synchronous by significant factors, but becomes significantly worse than synchronous for larger minimal latencies. The composite protocol is 100 times faster than synchronous for the smallest minimum latency configuration and maintains its dominance over both protocols throughout the entire curve. Again, we see that the method automatically "tracks" the performance sweet spot.

Figs. 7 and 8 provide the same analysis for the data from the aggregated model with 32 timelines. While the relative performance comparison with synchronous is less dramatic, the composite method finds the correct operation point and (with the asynchronous method) achieves a level of performance that, for small minimal latencies, is twice that of the synchronous protocol.

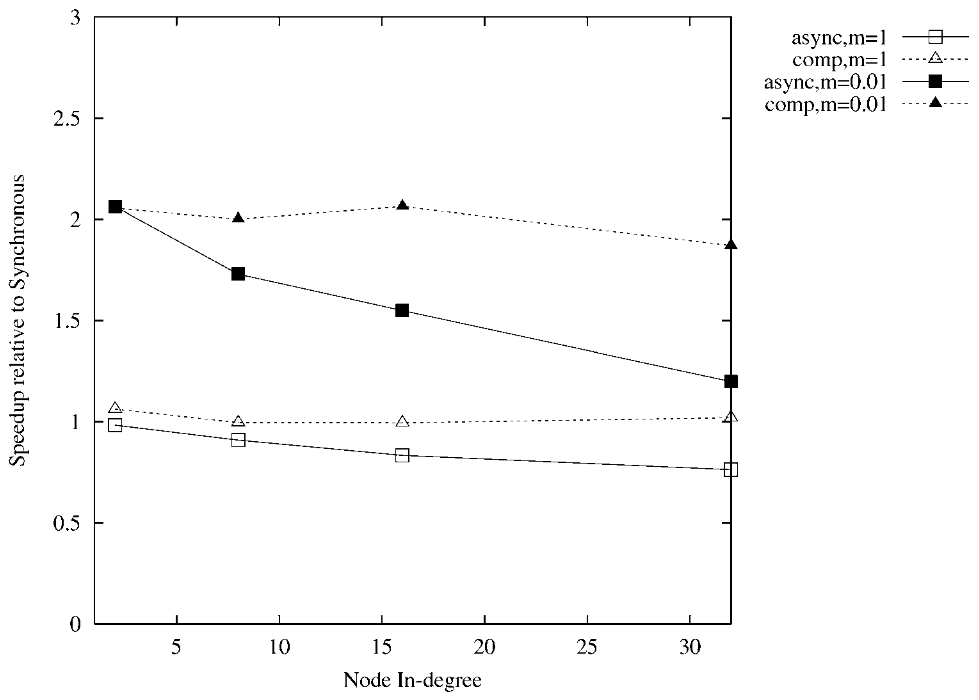

Fig. 7. Sensitivity of composite synchronization to node degree, aggregate case.

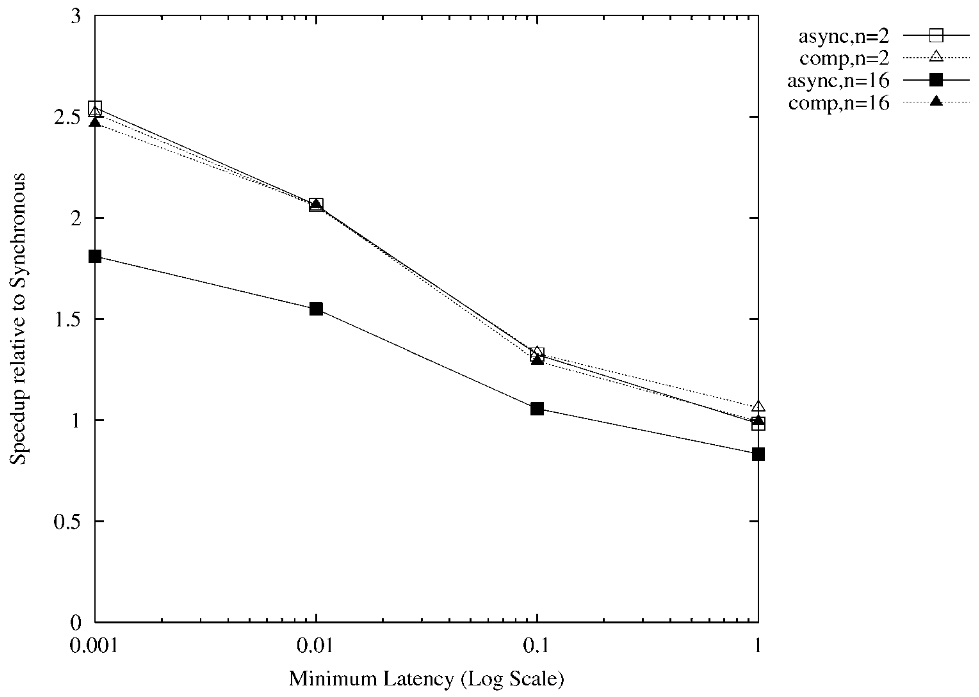

Fig. 8. Sensitivity of composite synchronization to minimal latency, aggregate case.

Next, we consider a network example that is part of the **SSFNet** (see www.ssfnet.org) distribution, a body of networking protocol models developed to run under SSF. For this model (and the autonomous system model to follow), essentially the same program is run as was just studied, with one difference. From now on a channel's minimum latency is identical to its actual latency, as is characteristic of network models. Our earlier distinction between minimal and actual latency was useful in identifying protocol sensitivity to differences between minimal and actual latency.

The **SSFNet** model loosely reflects hierarchical structure found in a midsize network, e.g., for a campus or small corporation. It has 646 nodes and 1,826 edges. The table below gives the histogram of node degree.

degree	4	6	8	10	12	14	16	18	20
nodes	456	48	39	41	36	20	2	3	1

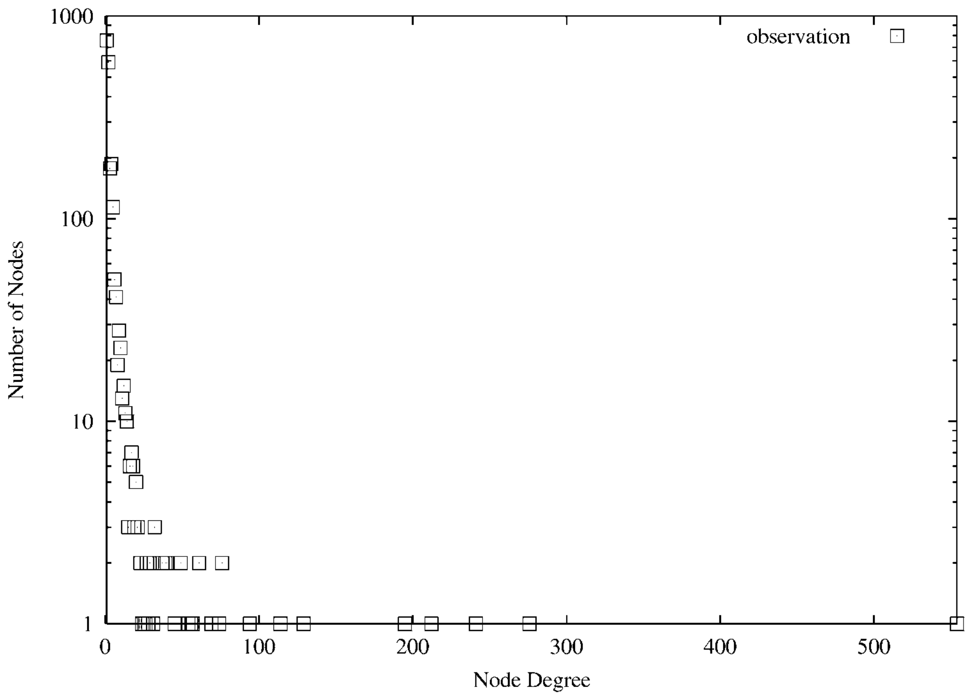

Fig. 9. Histogram of node degree, Internet AS graph.

The network has three unique channel latencies, which we denote L_0, L_1, and L_2. The table below characterizes the frequency distribution of channel latencies at a node, as a function of the node degree.

degree	L_0	L_1	L_2
4	1.0	0.0	0.0
6	0.26	0.67	0.07
8	0.49	0.5	0.01
10	0.37	0.58	0.05
12	0.57	0.28	0.15
14	0.84	0.14	0.02
16	0.5	0.12	0.38
18	0.44	0.11	0.45
20	0.4	0.1	0.5

The network's statistical data tells us the network is relatively sparse—2/3 of the nodes have degree 4. It also tells us that the L_0 weighted channels predominate. The model distributed in **SSFNet** uses the assignment $L_0 = 0.001$, $L_1 = 0.005$, and $L_2 = 0.033$. With these settings, using eight processors of the Sun Enterprise 6500 server and the assumption of one timeline per node, the fully asynchronous protocol is 3.5 times *slower* than the fully synchronous protocol; the composite protocol locks-in on the fully synchronous protocol and has the same performance as the synchronous protocol. We then did experiments that made L_0 smaller. This changed the ratio of asynchronous performance to synchronous performance, but did not change the dominance of the synchronous protocol nor the the action of the composite protocol to lock-in on the fully synchronous partition. This is intuitive because the synchronous protocol is more efficient on nodes of degree 4 than is the asynchronous protocol and most of the edges attach to nodes of degree 4. The story changes, though, if we consider giving the L_0 edges large latencies and the L_2 edges small latencies, e.g., $L_0 = 0.1$ and $L_2 = 0.00033$. While this assignment does not reflect latencies we had expect in the graph under consideration, it does serve to further illustrate the sensitivity of synchronization costs to topology. The asynchronous protocol is 2.5 times faster than the synchronous protocol and the composite protocol (which locks-in making L_2 channels asynchronous and the rest synchronous) is best of all, at 3.4 times faster than the synchronous protocol.

For our last example, we turn to the Internet itself. URL www.moat.nlanr.net/AS gives access to stored histories of the always changing peering relationships between Autonomous Systems (AS) in the Global Internet. We get a real graph by considering each AS as a node, with an edge shared with an AS with which it peers (i.e., between two AS's that exchange traffic with each other).

Fig. 9 gives a histogram of the AS graph in late spring of 2000. The graph has 2,107 nodes and 9,360 edges. The extremity of node degree is striking. Nearly 1/2 of the nodes have one (undirected) edge, over 3/4 have just one or two edges. Node degrees become very large. This is actually a poor example for parallel processing because one node (representing AS **uunet**) connects to nearly 1/2 of all other nodes and, as a result, becomes an "attractor" for messages in the traffic generation pattern we use in our examples; most of the workload ends up hitting one node. The main interest to us is that the graph illustrates the heavy-tailed connectivity one finds in the Internet.

We use this real graph to motivate a synthetic one which is modeled on its characteristics. We imagine that the single highly connected AS is built using 128 routers, which are fully connected. We futhermore imagine that each of these

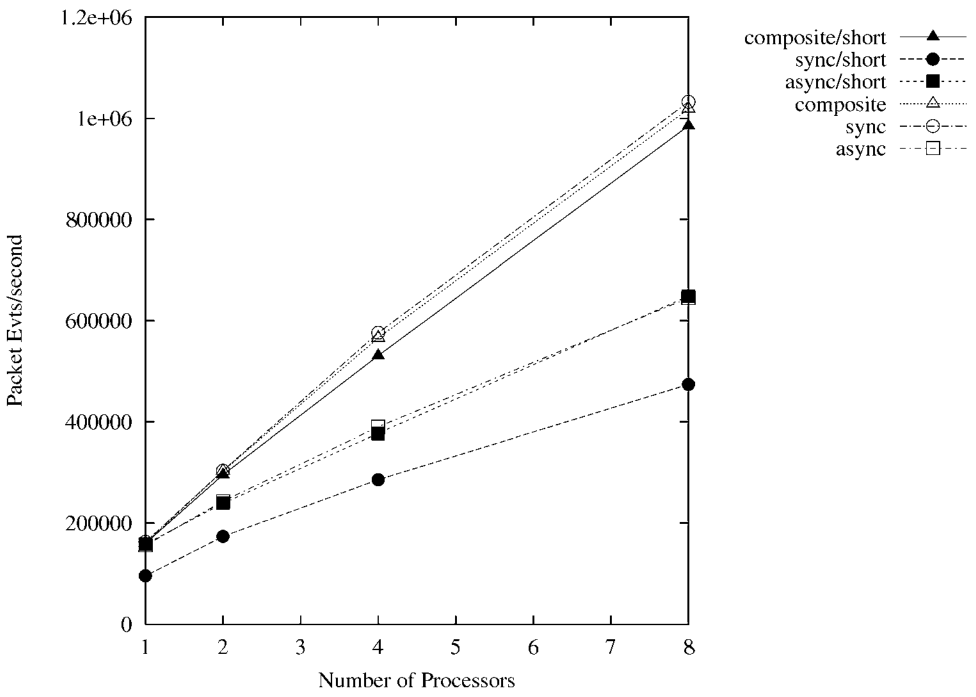

Fig. 10. Performance on synthetic AS topology graph.

routers has four satellite hosts, which are connected only to that router. This gives us a graph with 640 nodes and 17,280 edges. Five hundred and twelve nodes have edge degree 1,128 nodes have edge degree 133. We take the latency between a satellite and its router to be 10ms; latencies between routers in the fully connected component are taken to be 100ms. We run traffic generation and TCP protocol models on each host. The model generates high traffic load, by having each host initiate a new bulk transfer after its previous one has terminated. TCP session endpoints are chosen uniformly at random and the routing within the modeled AS is shortest path.

We evaluate the composite algorithm in this scenario by introducing two links in the AS which may be "short," 0.1ms. Fig. 10 illustrates the results. Performance using fully synchronous, fully asynchronous, and composite protocols are shown, in the case where there are no short edges in the AS and in the case where they do exist. Performance is plotted in terms of aggregate "packet events" per second—a packet event is the occurence of a packet being generated, being placed on a communication channel, or being received by either a router or host. This is a loose measure, as not all packet events have the same workload; packets whose receipt triggers TCP protocol calculations require more work than packets which are simply placed on a communication channel. However, it does give a measure that is independent of the simulation kernel of how much work is being performed. The graph plots the packet event rate as a function of the number of processors used.

The curves marked by filled plot points correspond to the graph with short edges. As expected, the asynchronous algorithm is virtually unaffected by the short edges. It achieves a better level of performance than the synchronous protocol. However, the composite method does significantly better than either of these—its large global window avoids the pitfall of the short edge latency, while exploiting sparseness of the asynchronous (short edge) channel topology. The open plot points correspond to the graph without short edges. Here, we find that the synchronous and composite algorithms have nearly identical performance across the range of processors and that this performance exceeds that of the asynchronous protocol by almost a factor of 2. The composite algorithm's performance is virtually the same on both graphs and, in both cases, is very close to the best performance shown. It delivers nearly one million packet events per second using eight processors and contributes to speedups of 1.84 using two processors, 3.31 using four processors, and 6.14 using eight processors. Here, again, we see the significant risk of not using a composite technique and the significant benefit of doing so.

7 Conclusions

Synchronization protocols for parallel discrete-event simulation can generally be categorized as "synchronous" or "asynchronous." Synchronous approaches are simple and scalable, but vulnerable to models where the worst-case lookahead is much smaller than that of the average case. Asynchronous methods are more finely tuned to actual lookaheads, but are vulnerable to models with high connectivity. Because these differences are model dependent, one style or the other cannot be best for all cases.

We explore the idea of combining synchronous and asynchronous approaches in an attempt to use one method in parts of the model where the other method is weak. By classifying each channel as asynchronous or synchronous, we can limit the effect of high connectivity by making most of a node's channels synchronous; we can limit the effect of unusually low lookahead by making channels with low lookahead asynchronous. We show that, under reasonable assumptions, we can formulate the channel assignment problem as a mathematical optimization problem and show that the optimal classification has a threshold structure—there exists threshold T such that all channels with minimal latency less than T are classified as asynchronous and all channels with minimal latency T or greater are classified as synchronous.

There are practical problems associated with quantifying the constants implicit in the optimization problem. However, we can still use the main theoretical result that says the optimal policy has threshold structure. Rather than *predict* performance under various conditionally optimal assignments, our implementation *measures* performance and, at runtime, dynamically alters channel classification to search for the one which optimizes performance. We have implemented this in the **DaSSF** parallel simulation engine and considered its performance on three networking models. The first is almost entirely balanced, useful for highlighting the substantive model-dependent performance differences that occur between asynchronous and synchronous approaches. Another is a model of a medium-scale network, the last is a synthetic network modeled after observed connectivity patterns in the Internet. In all these cases, we see that the proposed method either identifies the better of the two synchronization endpoints, or identifies a true mixture whose performance exceeds either "pure" approach.

Composite synchronization is just one of several problems that must be solved together if automated parallelization of simulations is to be a reality. However, it is clear that it is a solution that parallel simulation kernels cannot afford to ignore.

Acknowledgments

This research is supported in part by DARPA contract N66001-00-C-8069, US National Science Foundation (NSF) Grant ANI-98 08964, NSF Grant EIA-98-02068, and US Department of Justice contract 2000-CX-K001.

References

[1] R. Ayani, "A Parallel Simulation Scheme Based on Distances between Objects," *Distributed Simulation 1989,* pp. 113-118, 1989.

[2] K.M. Chandy and J. Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Trans. on Software Eng.,* vol. 5, no. 5, pp. 440-452, Sept. 1979.

[3] J. Cowie, A. Ogielski, and D. Nicol, "Modeling the Global Internet," *Computing in Science and Eng.,* vol. 1, no. 1, p. 42-50, Jan./Feb. 1999.

[4] H. Davis, S. Goldschmidt, and J. Hennessy, "Multiprocessor Simulation and Tracing Using Tango," *Proc. 1991 Int'l Conf. Parallel Processing,* pp. II99-II107, Aug. 1991.

[5] P. Dickens, P. Heidelberger, and D. Nicol, "Parallelized Direct Execution Simulation of Message Passing Programs," *IEEE Trans. Parallel and Distributed Systems,* vol 7, no. 10, pp. 1090-1105, Oct. 1996.

[6] A. Ferscha, "Parallel and Distributed Simulation of Discrete Event Systems," *Parallel and Distributed Computing Handbook,* 1995.

[7] R.M. Fujimoto, "Parallel Discrete Event Simulation," *Comm. ACM,* vol. 33, no. 10, pp. 30-53, Oct. 1990.

[8] B. Premore, J. Liu, D. Nicol, and A. Poplawski, "Performance Prediction of a Parallel Simulator," *Proc. 1999 Parallel and Distributed Simulation Conf.,* 1999.

[9] J. Liu and D. Nicol, "Learning Not to Share," *Proc. 2000 Parallel and Distributed Simulation Conf.,* May 2001.

[10] J. Liu, K. Tan, and D. Nicol, "Lock-Free Scheduling of Logical Processes in Parallel Discrete-Event Simulation," *Proc. 2000 Parallel and Distributed Simulation Conf.,* May 2001.

[11] Y.-H. Low, C.-C. Lim, W. Cai, S.-Y. Huang, W.-J. Hsu, S. Jain, and S. Turner, "Survey of Languages and Runtime Libraries for Parallel Discrete-Event Simulation," *Simulation,* vol. 72, no. 3, pp. 170-186, Mar. 1999.

[12] B.D. Lubachevsky, "Efficient Distributed Event-Driven Simulations of Multiple-Loop Networks," *Comm. ACM,* vol. 32, no. 1, pp. 111-123, 1989.

[13] D. Nicol, "The Cost of Conservative Synchronization in Parallel Discrete-Event Simulations," *J. ACM,* vol. 40, no. 2, pp. 304-333, Apr. 1993.

[14] D. Nicol, "Principles of Conservative Synchronization," *Proc. 1996 Winter Simulation Conf.,* pp. 128-135, Dec. 1996.

[15] D. Nicol and P. Heidelberger, "A Comparative Studey of Parallel Algorithms for Simulating Continuous Time Markov Chains," *ACM Trans. Modeling and Computer Simulation,* vol. 5, no. 4, pp. 326-354, Oct. 1995.

[16] D. Nicol and P. Heidelberger, "Parallel Execution for Serial Simulators," *ACM Trans. Modeling and Computer Simulation,* vol. 6, no. 3, pp. 210-242, July 1996.

[17] D.M. Nicol and R.M. Fujimoto, "Parallel Simulation Today," *Annals of Operations Research,* vol. 53, pp. 249-286, Dec. 1994.

[18] A. Poplawski and D. Nicol, "An Investigation of Out-of-Core Parallel Discrete-Event Simulation," *Proc. 1999 Winter Simulation Conf.,* pp. 524-530, 1999.

[19] H. Rajaei, R. Ayani, and L.-E. Thorelli, "The Local Time Warp Approach to Parallel Simulation," *Proc. 1993 Workshop Parallel and Distributed Simulation,* pp. 119-26, 1993.

[20] S. Reinhardt, M. Hill, J. Larus, A. Lebeck, J. Lewis, and D. Wood, "The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers," *Proc. 1993 ACM SIGMETRICS Conf.,* pp. 48-60, May 1993.

[21] L.M. Sokol, D.P. Briscoe, and A.P. Wieland, "MTW: A Strategy for Scheduling Discrete Simulation Events for Concurrent Execution," *Proc. SCS Multiconf. Distributed Simulation,* pp. 34-42, 1988.

[22] S.J. Turner and M.Q. Xu, "Performance Evaluation of the Bounded Time Warp Algorithm," *Proc. Sixth Workshop Parallel and Distributed Simulation (PADS92),* pp. 117-26, 1992.

[23] F. Wieland, "The Detailed Policy Assessment Tool (DPAT)," *Proc. 1997 Spring INFORMS Conf.,* May 1997.

[24] Z. Xiao, B. Unger, R. Simmonds, and J. Cleary, "Scheduling Critical Channels in Conservative Parallel Discrete Event Simulation," *Proc. 13th Workshop Parallel and Distributed Simulation,* pp. 20-28, 1999.

David M. Nicol received the PhD degree in computer science from the University of Virginia, Charlottesville, in 1985, and taught at the College of William and Mary until 1996. He is currently a professor and chair of the Department of Computer Science at Dartmouth College. He is coauthor of the *Discrete Event Systems Simulation*, third edition (Bands, Carson, and Nelson), and of more than 120 technical articles. Since 1996, he has served as editor-in-chief of *ACM Transactions on Modeling and Computer Simulation*. His research interests are in parallel simulation, performance evaluation, networking, and computer security.

Jason Liu received the BA degree in computer science from Beijing Polytechnic University in China in 1993. He graduated form the College of Williaim and Mary in 2000 with the MS degree in computer science. He is a PhD student in the Computer Science Department at Dartmouth College. His research focuses on parallel discrete-event simulation, performance modeling, simulation of computer systems, and communication networks. He is also interested in large-scale simulation of wireless communicaiton systems.