Proceedings of the 2006 Winter Simulation Conference
L. F. Perrone, F. P. Wieland, J. Liu, B. G. Lawson, D. M. Nicol, and R. M. Fujimoto, eds.

PARALLEL AND DISTRIBUTED SIMULATION: TRADITIONAL TECHNIQUES AND RECENT ADVANCES

Kalyan S. Perumalla

Oak Ridge National Laboratory
One Bethel Valley Road
Oak Ridge, Tennessee 37831-6085, U.S.A.

ABSTRACT

This tutorial on parallel and distributed simulation systems reviews some of the traditional synchronization techniques and presents some recent advances.

1 INTRODUCTION

Parallel and distributed simulation finds relevance in many applications, including civilian applications such as telecommunication networks, physical system simulations and distributed multi-player gaming, and non-civilian applications such as battlefield simulations and emergency event training exercises, to name only a few. It deals with ways of using multiple processors in a single simulation. Achieving correctness of parallel execution requires synchronization across processors. Parallel simulation, at its core, is concerned with accurately synchronizing simulations that run on multiple inter-connected processors. All processors together serve to collectively simulate an integrated set of application models. The simulation is partitioned spatially (or temporally), and the partitions are mapped to processors. Although multiple processors can be employed to execute multiple separate single-processor simulation runs in parallel, here we focus on parallel simulation methods in which all processors are together used to execute a single simulation run, be it an integrated set of simulators or a single monolithic parallel system.

To be meaningful, the results produced by a parallel simulation run must ideally match those that could be produced by an equivalent sequential simulation run. To achieve this match, parallel execution must be properly synchronized to preserve the right orderings and dependencies during computation of simulation state across processors. One of the challenges in this synchronization is in minimizing the runtime execution overheads (memory, computation and communication) incurred during parallel execution. It is thus important to keep the overhead within acceptable levels, in order for the parallel execution to deliver sufficient value above and beyond sequential simulation. A large amount of research has been focused on reducing this overhead by devising efficient methodologies, algorithms and implementations for synchronization in parallel and distributed simulation.

In this vein, parallel and distributed simulation techniques have been well studied in the past two to three decades. The literature is now sizable, and several comprehensive books and survey articles, such as (Banks, et al. 1996, Fujimoto 1990, Fujimoto 1993, 2000), already serve well to document traditional techniques in parallel and distributed simulation. Recent emergence of new application demands, techniques and hardware platforms are resulting in enhancements to traditional techniques and formulation of newer techniques. These have been appearing at various places in the literature and together represent newer advances in parallel and distributed simulation. Here we present a brief overview of traditional techniques followed by a presentation of some of the recent advances.

The structure of the paper is as follows. Section 2 presents quick overview of basic concepts and classical techniques. Section 3 serves as a practical case study that describes the parallel time synchronization functionality provided by the IEEE standard called the High Level Architecture. Section 4 documents some of the recent developments. Section 5 discusses the issues and challenges raised by the expanding set of hardware platform types on which parallel simulation is now being performed. Finally, a summary in Section 6 recapitulates the material.

2 CLASSICAL TECHNIQUES

Parallel and distributed simulation approaches can be broadly categorized into spatial parallel and time parallel schemes. In spatial parallel schemes, the application is partitioned along spatial dimensions underlying the application's models (e.g., three dimensional grid cells in physical system simulation, or computers and network routers in Internet simulations). Time parallel schemes partition the simulation along its time dimension (e.g., regular time intervals along the simulation time axis).

Spatial decomposition is by far the most commonly used parallel simulation scheme. In this scheme, applica-

tion models are partitioned into logical processes (LPs). Each LP contains its own individual state variables, and interactions among LPs are only via exchange of time-stamped events. The simulation progresses via execution of these events in temporal order. The temporal ordering is either maintained at every instant during simulation, or is achieved in an asymptotic manner (i.e., the system guarantees eventual convergence to overall temporal order). Here we focus only on spatial parallel schemes. Further information on time parallel schemes can be found in (Fujimoto 2000) and other places.

2.1 Basic Concepts

Let us first examine some basic concepts that underlie any synchronized parallel/distributed simulation, followed by an overview of basic synchronization approaches.

2.1.1 Notions of Time

In simulations, there are generally three distinct notions of time. The first is the *physical time*, which is the time in the physical system that is being modeled (e.g., 10-11pm on January 1990). The second is the *simulation time*, which is a representation of the physical time for the purposes of simulation (e.g., number of seconds since 10pm of January 1990, represented in floating point values in the range [0..3600] corresponding to the simulated time period of the physical time). Finally, the *wallclock time* is the elapsed real time during execution of the simulation, as measured by a hardware clock (e.g., number of milliseconds of computer time during execution). For each, the notions of *time axis* and *time instant* can be defined – time axis is the totally ordered set of time instants along the corresponding timeline. In particular, for simulation time, the time axis is common across all processors, and a processor's simulation time is its current time instant along the time axis up to which the processor has advanced its simulation.

2.1.2 Execution Pacing

In general, there is a one-to-one mapping from physical time to simulation time. In contrast, there may or may not exist a specific relationship between simulation time and wallclock time. The mode of simulation execution determines this particular relationship. In an as-fast-as-possible execution, the simulation time is advanced as fast as computing speed can allow, unrelated to wallclock time. In real-time execution, on the other hand, advances in simulation time are performed in lockstep with wallclock time – one unit of simulation time is advanced exactly in one same unit of wallclock time. A variation of real-time execution is scaled real-time execution, in which simulation time period is some constant factor times an equivalent wallclock time period.

Synchronization algorithms are required to provide correct execution, avoiding undesirable effects such as deadlocks, live-locks and termination problems. What more, in analytic simulations, which are executed in an as-fast-as-possible fashion, an important system goal is to minimize the overheads of synchronization such that the simulation completes as faster than real-time as possible. This adds the need for delivering rapid simulation progress, on top of correctness of parallel operation.

2.1.3 Events and Event Orderings

An event is an indication of an update to simulation system state at a specific simulation time instant. Thus each event specifies a timestamp. When events are exchanged among processors, their delivery at the receiving processors needs to be carefully coordinated at runtime. In general, multiple different types of delivery ordering systems can be defined. Two commonly used orderings are (1) receive-order (2) timestamp-order. Other types (Lamport 1978), such as causal order (Lee, et al. 2001), could also be useful in certain cases, but they are not as commonly used.

In receive-ordered delivery (RO), events from other processors are delivered to the receiving processors as and when the events arrive at the receiving processor. In contrast, in timestamp-ordered delivery (TSO), events are guaranteed to be delivered in non-decreasing order of their timestamps. Typically, since RO delivers events right away, RO events incur lower delivery delay/latency from the moment they are sent by a processor to the moment the destination processor(s) receives them. TSO events on the other hand undergo runtime checks and buffering until their non-decreasing timestamp order can be ascertained and guaranteed, and hence TSO events incur relatively higher latency. However, a significant difference arises with respect to modeling accuracy afforded by RO and TSO. RO cannot always preserve "before and after" relationships, while TSO does guarantee preservation of such relationships. Similarly, with TSO, all processors see the exact same ordering of events, whereas with RO, identical ordering among events cannot be guaranteed across processors. Overall parallel execution can be made repeatable with TSO from one execution to the next, while RO cannot ensure such repeatability.

2.1.4 Timestamp-Ordered (TSO) Processing

The rationale behind timestamp-ordered processing is that it permits the models to be accurately simulated, such that events are processed in the same order as their corresponding actions in the physical system. To enable such processing order, a simple local rule to follow is that a processor whose simulation time is at T should not receive events with timestamps less than T. Hence, advances of a processor's current time have to be coordinated and controlled

carefully to prevent events appearing in processor's "past." This requirement gives rise to different synchronization approaches, and consequently, different algorithms.

2.2 Basic Synchronization Approaches

In parallel simulation, broadly four approaches are commonly used: conservative, optimistic, relaxed, and combined synchronization.

Conservative: This approach always ensures safe timestamp-ordered processing of simulation events within each LP (Chandy and Misra 1978, 1981). In other words, an LP does not execute an event until it can guarantee that no event with a smaller timestamp will later be received by that LP. However, runtime performance is critically dependent on a priori determination of an application property called *lookahead*, which is roughly dependent on the degree to which the computation can predict future interactions with other processes in the absence of global information. In a lookahead-based approach, events that are beyond the next lookahead window are blocked until the window advances later, sufficiently far to cover those events. Typically the lookahead property is very hard to extract in complex applications, as it tends to be implicitly defined in the source code interdependencies. The appeal of this approach however is that it is easier to implement than the optimistic approach (described next) if such a lookahead value can be specified by the application.

Optimistic: This approach avoids "blocked waiting" by optimistically processing the events beyond the lookahead window (Jefferson 1985). When some events are later detected to have been processed in incorrect order, the system invokes compensation code such as state restoration or reverse computation (described later). A key issue introduced by large-scale platforms is the increased delay of inter-processor communication. Optimistic synchronization offers the potential for greater resilience to delays in the sense that computations may progress despite the delay in generation/delivery of certain events. Since blocking is not used, the lookahead value is not as important, and could even be specified to be zero without greatly affecting the runtime performance. While this approach eliminates the problem of lookahead extraction, it has a different challenge – namely, support for compensating code. Traditional optimistic methods rely on state saving or other techniques to enable rolling back to a previous state in case an event arrives in the "past".

Relaxed synchronization: This approach relaxes the constraint that events be strictly processed in time stamp order (e.g., see (Fujimoto 1999, Rao, et al. 1998)). For example, it might be deemed acceptable to process two events out of order if their time stamps are "close enough." This approach offers the potential of providing a simplified approach to synchronization, but without the lookahead constraints that plague conservative execution. A key challenge with this approach is determining the extent that ordering constraints can be relaxed without compromising the validity of the simulation results. An additional challenge lies in ensuring that the execution of the simulation is repeatable. Repeatability means multiple executions of the same simulation with the same inputs are guaranteed to yield the same numeric results from one execution to the next. This property may not be preserved with relaxed execution because events within each process may be processed in a different order from one execution to the next unless special care is taken.

Combined synchronization: This approach combines elements of the previous three. For example, sometimes it might help to have some parts of the application execute optimistically ahead (e.g., parts for which lookahead is low or hard to extract), while other parts execute conservatively (e.g., parts for which lookahead is large, or for which compensation code is difficult to generate) (e.g., see (Jha and Bagrodia 1994, Perumalla 2005, Rajaei, et al. 1993)). In such cases, a combination of synchronization techniques can be appropriate. A practical application of such a combined synchronization approach is the High Level Architecture, described in greater detail in Section 3.

3 CASE STUDY: THE HIGH LEVEL ARCHITECTURE

The US Department of Defense (DoD) High Level Architecture (2000), heretofore referred to as the HLA, includes support for time-synchronized parallel/distributed simulations, built on fundamental concepts of parallel and distributed simulation (PADS). In the HLA, an integrated execution of simulations is called a federation. Individual simulators participating in a federation are called federates. Federates can be of different types: pure software simulators such as computer generated forces, human-in-the-loop simulators such as virtual simulators, or live components such as instrumented weapon systems. In this section, we will use the terms "federate" and "processor" interchangeably.

As mentioned earlier, a significant amount of literature exists in the PADS research community, which has explored issues in time synchronized simulations. The time synchronization module of the HLA, called the Time Management (TM) has, in large part, been built on insights from PADS research. Thus, the fundamental concepts in HLA TM are common with those in PADS.

3.1.1 Interoperability Challenge

The HLA's TM services address two important components: (1) overall event processing order by each federate (2) synchronized event delivery to each federate.

While enabling event processing order and synchronized event delivery, all in a single encompassing standard

framework, the HLA needs to accommodate a large variety of individual types of simulators. In general, there is a plethora of different simulator types – event-stepped vs. time-stepped, sequential vs. parallel, real-time vs. as-fast-as-possible, conservative vs. optimistic, etc. An HLA federation might include any combination of any of these simulator types. Moreover, the exact combination of the types is not always made known *a priori* to the HLA RTI, and hence the interface as well as the implementation must be sufficiently general to accommodate any/all of them. The HLA TM interface does an amazing job of accommodating any arbitrary combinations of, and any number of instances of, different types of simulators, all in one core, seamless interface.

3.1.2 Synchronization Services

The HLA Time Management module provides a clear interface that each federate must invoke in order to synchronize with other processors. The three most commonly used services are: Time Advance Request (TAR), Next Event Request (NER) and Flush Queue Request (FQR).

A federate undertaking fixed time increments in simulation time can use TAR(T) to unconditionally advance its simulation time to T. Events from other federates that arrive with timestamps less than T are all delivered to this federate before the runtime permits the federate to advance to time T. Time-stepped parallel simulations typically use this service to coordinate their time steps across processors.

A federate operating under a discrete event paradigm invokes the NER(T) primitive to conditionally advance its simulation time to T. If the runtime discovers that other events with timestamps less than T are generated by other processors, the earliest of those events are delivered to the federate, and time is advanced only to their timestamp. This service is most commonly used by conservative parallel discrete event simulators.

A federate equipped to execute its events in optimistic mode (i.e., ahead of receiving guarantees of correctness) can invoke FQR(T) to force the runtime to release any and all events that it currently has, irrespective of their timestamps. The runtime utilizes the supplied timestamp T to compute absolute global time advances.

3.1.3 Computing LBTS

A fundamental role of a TM implementation is in computing a quantity known as Lower Bound on incoming Time Stamps (LBTS). At each federate, the LBTS value specifies a guarantee on the least timestamp on any future incoming event. In other words, no event will ever arrive at that federate with a timestamp smaller than LBTS. Once this global value is known, it is rather straightforward to locally serve TM requests, such as TAR, NER and FQR.

In order to compute the LBTS value at each federate, a distributed algorithm is required that exchanges messages to coordinate the LBTS computation without deadlocks, live-locks or undue performance degradation. Several such algorithms have been proposed in PADS literature – see for example (Perumalla and Fujimoto 2001). A close cousin to the LBTS computation is Global Virtual Time (GVT) computation in optimistic simulation (Bellenot 1990). Another closely related work in general distributed processing is that of distributed "flush barrier" algorithms (Ahuja 1990). Analogous to these algorithms, several variants exist for LBTS computation.

One such algorithm is based on global asynchronous distributed reductions. In this algorithm, the minimum local (conditional) guarantee on timestamps of events that could be generated is taken at each federate, and a global reduction algorithm is used to find the minimum of all the local minima. This can be performed fairly quickly and scalably, in $\log(N_p)$ steps, where N_p is the number of federates, using a butterfly pattern of communication (Brooks 1986). Assuming there are no events in transit across federates, the minimum of the minima gives a tight lower bound on LBTS.

3.1.4 Transient Messages

What if there are some events that are in transit in the network while the global minimum of local minima is being computed? This is called the transient event problem, in which some events could become potentially unaccounted for if they are not considered into the global algorithm. There exist different schemes by which transient events can be accounted for, albeit at the cost of either additional messages being sent/received and/or additional time spent blocking while waiting for all transient events to reach their destinations. A popular one is called the Mattern's algorithm (Mattern 1993) in which distributed consistent cuts are used to mark and recognize events belonging to distributed different snapshots.

For conservative parallel simulations, it is clear that the larger the lookahead, the fewer the number of LBTS computations that need to be performed, because of increased concurrency enabled by the larger lookahead.

3.1.5 Serving Synchronization Requests

The RTI internally maintains a priority queue of TSO events, ordered by their timestamps. When a federate invokes TAR(T), the RTI first examines if LBTS is greater than T. If so, the request is trivially satisfied – the RTI delivers all events from its TSO queue whose timestamps are less than or equal to T, and then issues a TAG(T). If T is greater than LBTS, then the RTI initiates a new distributed LBTS computation (if one is not already in progress). The lesser of T and minimum timestamp in TSO queue is used

as this federate's contribution in the LBTS computation. The operation is similar for NER(T) invocations as well, except that the TAG time could be smaller than T if events with timestamps earlier than T are delivered.

3.2 Other HLA TM Services

In addition to supporting basic integration of conservative federates, the HLA TM services include primitives to integrate federates that use advanced simulation methods, such as event retractions and optimistic event execution.

3.2.1 Event Retractions

In simulations, models are sometimes written to un-schedule previously generated events. For example, although a move event is scheduled on an entity at T, it might have to be retracted later if the entity gets destroyed after the event is scheduled but prior to T. Such event retractions are called user-level retractions. Typically, user-level retractions are enabled as follows. When an event is scheduled, the system returns a handle to that event. Later, if and when that event needs to be retracted, a retract primitive is invoked to which the event handle is given. The system then un-schedules that event. The HLA RTI provides such a framework using event handles and retraction primitive. Interestingly, the same service is also used for "system-level" retraction in optimistic simulations, as described next.

3.2.2 Optimistic Event Execution

As mentioned previously, the HLA supports conservative federates as well as optimistic federates, as well as their arbitrary combinations. Optimistic federates differ from their conservative counterparts in that they do not discard events after processing them. Instead they keep the events around, and also maintain copies of simulation states before modifying them as part of event processing. Since optimistic federates do not rely on lookahead, they execute their events without blocking for safety. In particular, they use the FQR(T) service of the RTI to force the RTI to deliver events from its TSO queue even if LBTS has not progressed past T. The difference between FQR and NER is that FQR does not guarantee that it has delivered all events with timestamp less than T. Thus, the federate will have to rollback its computation if/when it later receives events whose timestamp is less than T. There are two main parts to such rollback: (1) undo local computation by restoring the state prior to erroneous event processing (2) undo all events erroneously sent to other federates. The first part is typically federate-specific, and hence the HLA does not provide a standard service for it. The second part is realized by using the event retraction service described previously. When an optimistic federate receives a retraction request, it performs an event annihilation procedure canceling the original event.

Note that the HLA RTI shields conservative federates from optimistic events by holding on to optimistic events in RTI TSO queues until such a time that LBTS sweeps past their timestamps. If the optimistically scheduled events happen to get retracted by their sending federates, those events will get annihilated within the RTI's TSO queues without ever getting delivered to the (conservative) destination federate.

4 RECENT ADVANCES

While parallel and distributed simulation as an area has been well studied, some advances are being made recently along multiple fronts. These advances are driven by new challenges raised by high-performance needs of applications such as large-scale telecommunication network simulation (e.g., Internet-scale TCP/IP simulations), hardware/VLSI system simulations. Some of these advances are examined next.

4.1 Mixed Synchronization

When large application scenarios are considered, it often is the case that the models are heterogeneous in nature, with varying amounts of computational loads across spatial dimensions and varying levels of modeling. For example, simulations of the Earth's magnetosphere inherently contain models of varying granularity and synchronization requirements (Karimabadi, et al. 2005).

In contrast to the heterogeneity needs, the traditional method of prevalence in building parallel simulation systems is to build a system specifically for one synchronization method (e.g., one conservative algorithm, or one optimistic variant). This tradition has two fallouts. First, additions to the underlying framework involve major overhauls. Secondly, modelers need to either determine and stick to one mechanism, or re-code their models to switch to a new mechanism.

A set of systems are being constructed to alleviate this problem when the application needs to accommodate multiple synchronization types and be resilient to future changes to parts of its models. An early system that provided this type of interface is the Maisie language (Bagrodia and Liao 1994), in which entities (equivalent to LPs) are capable of choosing and/or changing their own conservative/optimistic modes dynamically at runtime. Later, the HLA has been designed to accommodate multiple simulator types, including optimistic and conservative ones, although this has been leveraged mostly in the context of coarse-grained simulators, and is not readily amenable to high-performance execution of fine-grained simulations. More recently, the μsik system (Perumalla 2005) was developed that provides a fine-grained simulation en-

vironment for mixed mode synchronization across shared memory and distributed memory platforms.

4.2 Lookahead Extraction

In purely conservative parallel simulations, the single most important parameter is the lookahead value. A small lookahead can have a large detrimental effect on overall runtime performance. Thus lookahead extraction has been a subject of intense study in PADS literature.

Recently many important analytic applications have been developed using conservative synchronization, largely in the network simulation context. In these applications, the models are prohibitively complex to apply optimistic simulation techniques but are also very tightly-coupled, exhibiting very small lookahead between LPs. Wireless network simulation is one such application (e.g., predicting the performance of IEEE 802.11 Wi-Fi networks). Static lookahead in such applications is typically very small, given by the time for electromagnetic signal propagation (at close to speed of light) over short distances. When coupled with the need to use such simulations in real-time context (for network emulation), the gain from improving the lookahead even by a small amount can be quite significant.

It is precisely in this context that recent efforts have focused on extracting as much lookahead information as possible from the models. Moving from purely static estimates of lookahead, these new techniques examine the LP inter-dependency structures dynamically at simulation runtime, and manage to extract values of lookahead that are higher than those estimated statically. Two such efforts are presented in (Zhou, et al. 2004) and (Liu and Nicol 2002).

4.3 Critical Channel Traversal

Another method of improving performance of conservative simulations focuses on scheduling alternatives. In the most common event scheduling approach, events from the future event list are executed in an earliest-timestamp-first manner. While being very simple to implement, such a scheme is less than optimal when caching and other effects are considered. Newer scheduling algorithms such as Critical Channel Traversal are designed for exposing and taking advantage of inherent locality of event execution in conservative simulations. Benefits include better cache performance due to immediate reuse of event buffers across causal chain of events and improved spatial locality due to reflection of LP inter-dependence on event processing patterns. The gains from such sophisticated event scheduling approaches have been shown to be significant, with very high event rates achieved even on fine-grained applications such as ATM network simulations. LPs that are "closely related" with respect to event communication are grouped together as "tasks". Tasks are then used as the primary scheduling units, and LPs within a task form secondary scheduling units. When LPs within a task interact among themselves more heavily than with other LPs outside that task, locality is enhanced and event scheduling overheads are decreased. Other beneficial effects include automatic realization of dynamic load balancing on shared-memory multiprocessor platforms, because processors operate on a dynamic pool of active tasks rather than with a traditional static assignment of LPs to processors.

While CCT is useful for enhancing performance, it does require the application be amenable to grouping of LPs into tasks, and is specific to conservative (lookahead-based) parallel simulations.

4.4 Optimizations for Efficient Rollback

While lookahead extraction is the bane of conservative parallel simulations, rollback support mechanism is its counterpart in optimistic parallel simulations. Efficient rollback techniques thus play an important role in the making or breaking of optimistic schemes. For a long time, checkpointing-based approaches were the dominant form of enabling rolling back computations. However, these imposed undue amount of overheads, making optimistic simulation schemes unappealing to many applications. An alternative scheme called reverse computation has been recently developed to alleviate this problem, as described next.

4.4.1 State Saving for Rollback

One of the common approaches for rolling back incorrect computations is based on checkpointing, also called state-saving. Checkpointing methods make a copy of the state variables of LPs before the variables are overwritten by optimistic event computations. If and when those event computations have to be negated (due to arrival of a remote event with an earlier timestamp, or due to withdrawal/cancellation of an event by a remote processor), state variable values are fetched from the checkpoint log and overwritten, thereby restoring the LP state to their correct values corresponding to the fault point.

The most commonly used state-saving technique is copy state-saving (CSS), in which a copy of the entire state is made before an event is executed. In CSS, the state is saved every time an event is processed. A variation of copy state-saving is called periodic state saving (PSS). PSS is a generalized technique in which state is saved only periodically, say, every p^{th} event, instead of every event as is done with CSS. This implies that some events save state before processing, and others do not. The former set of events can be rolled back easily by restoring the state to the saved values. The latter set of events needs special treatment, since they do not have saved state. The state restoration for these events is achieved by starting with a past processed

event that does have saved state, and then re-executing the sequence of events from that past event to the event just before the rolled back event.

Incremental state-saving (ISS) is an alternative state-saving technique in which only the modified portions are saved just before modification. Modifications to state are logged as pairs of address-value pairs, and stored in a log array for each LP. Optimistic simulators can permit CSS, PSS and ISS to be used simultaneously together in the same application, although most applications tend to use them mutually exclusively, because either the LP states are uniformly too large (warranting ISS) or uniformly small (warranting CSS) or in between (warranting PSS). A detailed treatment of compiler-based techniques to optimize the appropriate state saving operations is given in (Gomes 1996).

As it turns out, state saving techniques imposes two types of penalties: (1) a memory copy overhead during forward computation (2) memory consumption beyond that of conservative execution. The degree of severity varies with the particular variant of state saving, but they carry across all variants. The first penalty reduces the efficiency of parallel execution (e.g., a four-processor optimistic simulation running at the same speed as that of a uniprocessor conservative simulation). The second type of penalty introduces not only undue memory overheads for fine-grained applications (fine grained events, by definition, execute rapidly, and therefore can produce a long state vector log equally rapidly), but also pollute the processor cache, thereby reducing forward computation speed.

4.4.2 Reverse Computation

Alternatives to checkpointing solutions evaded the community until recently a reverse computation (RC) approach was proposed. In this approach, values of state variables are not saved for rollback support. Instead, as and when rollback is initiated, a perfect inverse of event computation is invoked that serves to recover the modified state values to their original values. In other words, the overwritten state values are reconstructed by executing the forward code backwards.

The RC approach has been successfully applied in different application domains, such as telecommunication network simulations (Carothers, et al. 1999, Garrett, et al. 2003) and physical system simulations (Tang, et al. 2005). Among optimistic simulation systems that currently support RC are ROSS (Carothers, et al. 2002) and μsik (Perumalla 2005).

Note that PSS and RC can be seen as duals of each other. Figure 1 depicts the relation between PSS and RC using a generic snapshot of execution at which rollback is initiated. The rightmost vertical bar represents the latest optimistically executed event. The light vertical bar in the middle represents the point to which computation needs to be rolled back. The left-most dark vertical line indicates the most recent periodic checkpoint before the rollback point. If RC is used, reverse execution is invoked on all events from left to middle. If PSS is used, coast forwarding computation (event re-execution to regenerate state) is invoked from left to middle. Clearly, the efficiency of RC or PSS depends on expected distance of left or right points, respectively, from the rollback point.

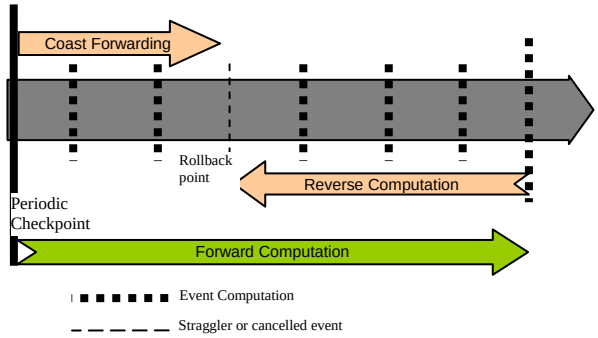

Figure 1: Comparison of Rollback Operation Using Reverse Computation vs. Periodic State Saving

However, a crucial difference between RC and PSS is that the state size is independent of the computing platform for RC, whereas it depends on sizes of data types in PSS. Also, PSS has the drawback that not all events from time points earlier than GVT/LBTS can be reclaimed for reuse (i.e., cannot be fossil-collected). Those events that fall between the periodic checkpoint time and GVT/LBTS need to be retained for coast-forwarding (Perumalla 1999).

4.5 Intra-Processor Lookahead

In large-scale application scenarios, each processor typically hosts many LPs. For example, in Internet simulations, each processor simulates thousands of network nodes. Traditional conservative parallel simulation maps network links crossing processor boundaries to in/out channels in null-message algorithms, and uses the minimum among all transmission delays as the lookahead values on all those out channels, irrespective of delays added by internal network nodes. However, time advances can be improved significantly if the additional topology information can be incorporated into the null message algorithm. The concept of "internal lookahead", illustrated in Figure 2, is defined as the minimum among all shortest paths between all pairs of in- and out-channels. This is easily computed as the sum of transmission delays along the shortest paths from all in channels to all out channels, which can add up to much more than lookahead along one transmission link. Adding this "internal lookahead" to null message timestamps significantly improves concurrency of the conservative simulation, delivering time advances in increments larger than one lookahead. The internal lookahead feature is supported in newer systems such as libSynk (Perumalla 2004).

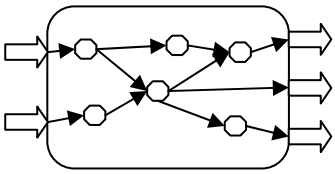

Figure 2: Illustration of Internal Lookahead

4.6 PADS Applied to *Sequential* Execution

Interestingly, even though parallel/distributed simulation techniques are developed specifically for execution on parallel computing platforms, they are in fact being discovered to be useful even in a sequential processing (uniprocessor) setting. When properly applied, the concepts developed for parallel execution can also be used to improve the speed of sequential simulation as an efficient alternative to traditional execution based on a central event list.

4.6.1.1 Conservative Execution on 1 Processor

The beneficial effects of parallel/distributed simulation techniques for sequential execution arise primarily due to (a) lower priority queue overheads (b) better caching effects (Curry, et al. 2005, Simmonds, et al. 2002). In a central event list implementation, events are organized into one priority queue (PQ), whereas in a parallel implementation, they are organized into multiple PQs. If N_e is the size of PQ encountered on average by each event, each event insertion/deletion incurs $O(\log(N_e))$ time overhead (assuming a general-purpose PQ implementation). On the other hand, if parallel simulation of the same application organizes the simulation as N_p logical processes, then, with algorithms such as critical channel traversal (Xiao, et al. 1999), each insertion/deletion of an event incurs priority queue sizes that are smaller than N_e/N_p. This makes the amortized event overhead less than $O(\log(N_e/N_p) + \log(N_p))$, where $\log(N_e/N_p)$ is the time to enqueue events into event priority queues and $\log(N_p)$ is the time taken to order the LPs by their earliest event timestamps. Moreover, reuse of memory buffers induces good caching effects which results in significant increase in performance. Such reuse of buffers for events processes immediately after one another naturally occurs for a chain of causally related events that fall within the same safe-time window.

4.6.1.2 Optimistic Execution on 1 Processor

Similar to application of conservative parallel simulation to sequential simulation, there are two different ways in which sequential simulation can benefit from optimistic parallel simulation methods: aggregate event processing and pipe-lined event processing.

In aggregate event processing, two or more events in the future event list are scheduled together (invoked with one "aggregate event handler"), with the expectation that no intermediate event computation arises between them to break their aggregate. In other words, the runtime behaves as though the events are scheduled as one event, rather than as multiple events. Rollback is initiated on the aggregate computation if violation of this assumption is detected at runtime (e.g., when a new event is scheduled by one of the events in the aggregate with timestamp intervening the aggregate's range of timestamps). The advantage of this approach is two-fold: event scheduling costs are reduced and compiler optimizations (such as common subexpression elimination) are realized on the aggregate event handler.

In pipe-lined event processing, newly generated events are executed at the tail of event handler of the generating event, bypassing the event queue, thereby reducing event queuing overheads and improving locality. Rollback is initiated on the tail-executed events if their time order is detected to be violated by later events.

4.6.1.3 Supercomputer-based Parallel Simulations

Another recent advance is in the realm of large-scale parallel computing platforms. Applications such as Internet simulations are serving to motivate the use of high-end computing for parallel discrete event simulation. With nascent interest in utilizing supercomputing facilities for such simulations, a few leading efforts have focused on porting, testing and improving traditional techniques on supercomputers. The state of the art in this area has now pushed the capabilities to one thousand processors and beyond.

Some of these include the demonstration of the PHOLD benchmark executed on 1,033 processors on the Pittsburgh Supercomputing Center using the DSIM Time Warp system (Chen and Szymanski 2005), and the RC-based PHOLD execution on 1,024 processors using the µsik system (Perumalla 2005) at the San Diego Supercomputing Center. The latter includes conservative, optimistic and mixed-mode simulations. Figure 3 shows processing time per event on PHOLD with increasing number of LPs and message population (MSG), with the largest recorded configuration containing 1 million LPs and 1 billion messages active at all times during the simulation. The amortized overhead of less than 20 microseconds per event brings it within the reach of even the most demanding fine grained simulation.

Conservative simulations have pushed the scalability limits to 512 processors on physical system simulations (Perumalla, et al. 2006) and to 1,536 processors (Fujimoto, et al. 2003) on network simulations. Additional refinements were shown using a variant of null message algorithm in (Park, et al. 2004).

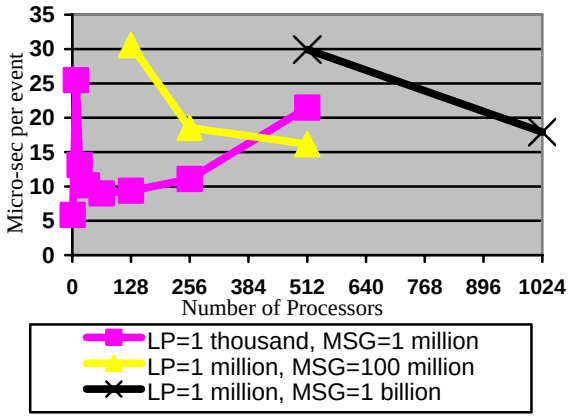

Figure 3: Runtime Performance Results from Optimistic Simulation of the PHOLD Benchmark on a Supercomputer

4.7 Approximate Time

Another exciting development in relatively recent past is the articulation of a new concept called Approximate Time. In this world model, the traditional notion of absoluteness of timestamps is relaxed into a range, rather than a point, on the simulation time axis. The range notion hinges on the realization that models typically have implicit fuzziness in the timestamps they generate (e.g., variation in the time taken for a battle tank to travel a given distance). This implicit uncertainty in timestamps can be exploited to relax the tight lookahead-based dependencies across processors, and consequently the parallel simulation speed can be improved. Systems that exploit temporal uncertainty are documented in (Fujimoto 1999, Loper 2002, Loper and Fujimoto 2004).

4.8 Fault Tolerance and Security

With the increase in the number of processors/federates participating in a large integrated simulation, the probability increases that one or more of the processors fail during simulation. Transparent techniques to sustain the progress of the simulation despite such crashes is a subject of recent interest. Schemes to incorporate such fault tolerance and/or continued optimistic execution are now being explored (Santoro and Quaglia 2006) (Chen, et al. 2006).

Also, secure exchange of information across processors without compromising confidentiality of information contained in models is an important facet of interoperable integrated simulations (e.g., preventing undesirable leaking of model information in joint exercises by different nations). This concern for security of information exchange during simulation raises new challenges, such as runtime overhead for encrypted communication, which are beginning to be addressed only recently.

5 EXPANDING SIMULATION PLATFORMS

Another exciting dimension of new advances in parallel simulation is in the use of novel computing platforms. With the advent of new computing hardware such as network co-processors and general-purpose graphical processing units, parallel simulation techniques are being revisited. Here we document some recent efforts in this light.

5.1 General-Purpose Graphical Processing Units

Traditionally, graphics cards for workstations and personal computers have been designed to handle intensive graphics operations to enable high speed rendering of complex objects and scenes. More recently, their programmability has reached a point to make them suitable for more general-purpose computation (Owens, et al. 2005, Pharr and Fernando 2005). As illustrated in Figure 4, certain applications have been shown to execute much faster on GPGPUs than on CPUs (Owens, et al. 2005). Texture data are input to a fragment processors (FPs), which "render" the results of their computation to target textures.

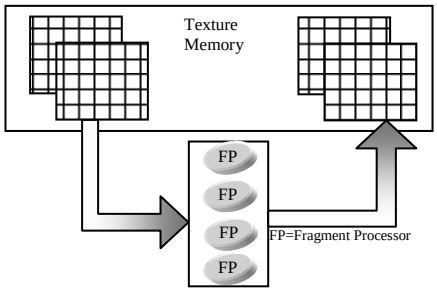

Figure 4: Simplified Schematic of GPGPU Operation

Since GPGPUs lack efficient random write access to memory (texture) locations, the traditional discrete event loop cannot be employed. Instead, algorithms such as the one shown in Figure 5 are more suitable. Computing the step $dt_{min}=min(dt_i)$, for example, is efficient on GPGPUs, taking only logarithmic time.

```
While not end of simulation
  /*Find next update times for all LPs*/
  For all (i): dt_i = compute_dt(i)
  /*Find minimum among all update times*/
  dt_min = min(dt_i) of all (i)
  /*Advance current simulation time*/
  t_now += dt_min
  /*Advance all LPs to current time*/
  For all (i): compute_state(i, dt_min)
```

Figure 5: A Variant of the Typical Sequential Discrete Event Algorithm Modified to Suit Execution on GPGPUs

While time-stepped approaches have been studied in simulations using GPGPUs, not enough work has focused

on discrete event simulation (DES) on GPGPU. Some recent work has shown the potential of GPGPUs for discrete event simulation (Perumalla 2006), showing four-fold speedup of GPGPU-based execution over that of regular processor.

5.2 Network Co-Processors and Hybrid Processors

The programmability of network co-processors is also being exploited to speed up certain operations. One of the uses is in off-loading state saving and other checkpointing functionality to programmable Myrinet network switches (Quaglia and Santoro 2003, Santoro and Fujimoto 2004). This helps free up the main processor to continue with main computation, minimizing rollback overhead. Another interesting use of co-processor is in fast computation of LBTS values in optimistic simulation (Rosu, et al. 1997), exploiting the low latency access to network operations by programs executing on the network card.

5.3 Other Architectures

Other architectures include Web-based execution, such as used in Extensible modeling and simulation framework (Moves-Institute 2006), and Internet/grid-based architectures, as used in the master-worker paradigm of (Park and Fujimoto 2006). The Cell processor from IBM is another new architecture that is poised to be suitable for high-performance parallel simulation, similar to the GPGPUs.

Tera- and Peta-Scale Computing Platforms such as the Blue Gene/L and Cray XT3 impose unprecedented scalability challenges for which it is unclear if traditional techniques will be adequate. For example, it is unclear if fully peer-to-peer architecture of traditional parallel simulation is appropriate in the presence of frequent node failures. Concerns such as fault tolerance and security are poised to become critical on such large-scale platforms. Similarly, recent multi-core processor architectures are yet to be fully explored. For example, key problems such as contention for I/O and communication among cores on a single system need to be resolved, possibly via staggered execution.

6 SUMMARY

This tutorial presented an overview of parallel and distributed simulation systems, their traditional synchronization approaches and a case study using the HLA standard interface and implementation. Recent advances, such as scalability to supercomputing platforms and novel rollback techniques have been presented. The interaction of parallel simulation with newly emerging hardware architectures is outlined.

The future outlook seems to warrant focus on needs from larger scale simulation scenarios (e.g., large-scale human behavior modeling & simulation, and large traffic simulations), to be achieved on high-end computing platforms. There is also interest on high-performance simulations on low-end platforms such as using the multi-core architectures, GPGPUs and other co-processor-based systems. However, practical challenges remain to be explored, including: (a) wide spectrum of network latencies (b) highly dynamic participation by processors (c) semantics and implementations of always-on presence for large simulations.

ACKNOWLEDGEMENTS

This paper has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the U.S. Department of Energy. Accordingly, the United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes.

REFERENCES

"IEEE Std. 1516. 2000. High Level Architecture," in *Institute of Electrical and Electronic Engineers*.

Ahuja, M. 1990. Flush Primitives for Asynchronous Distributed Systems. *Information Processing Letters*5-12..

Bagrodia, R. and W.-T. Liao. 1994. Maisie: A Language for the Design of Efficient Discrete-Event Simulations. *IEEE Transactions on Software Engineering*, vol. 20(4), pp. 225-238.

Banks, J., J. S. Carson II, and B. L. Nelson. 1996. *Discrete-Event System Simulation*. Upper Saddle River, N.J.: Prentice Hall.

Bellenot, S. 1990. Global Virtual Time Algorithms," in *Proceedings of the SCS Multiconference on Distributed Simulation*: Society for Computer Simulation, pp. 122-127.

Brooks, D. E. 1986. The Butterfly Barrier. *The International Journal of Parallel Programming*, vol. 14295-307.

Carothers, C., D. Bauer and S. Pearce. 2002. ROSS: A High-Performance, Low Memory, Modular Time Warp System. *Journal of Parallel and Distributed Computing*, vol. 62(11), pp. 1648-1669.

Carothers, C., K. S. Perumalla and R. M. Fujimoto. 1999. Efficient Optimistic Parallel Simulations using Reverse Computation. *ACM Transactions on Modeling and Computer Simulation*, vol. 9(3), pp. 224-253.

Chandy, K. M. and J. Misra. 1978. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Transactions on Software Engineering*, vol. SE-5(5), pp. 440-452.

Chandy, K. M. and J. Misra. 1981. Asynchronous Distributed Simulation via a Sequence of Parallel Computations. *Communications of the ACM*, vol. 24(4), pp. 198-205.

Chen, D. and B. K. Szymanski. 2005. DSIM: Scaling Time Warp to 1,033 Processors. In *Proceedings of the 2005 Winter Simulation Conference*. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Chen, D., S. J. Turner and W. Cai. 2006. A Framework for Robust HLA-based Distributed Simulations. In *Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation*.

Curry, R., C. Kiddle, R. Simmonds and B. Unger. 2005. Sequential Performance of Asynchronous Conservative PDES Algorithms. In *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*.

Fujimoto, R. M. 1990. Parallel Discrete Event Simulation. *Communications of the ACM*, vol. 33(10), pp. 30-53, 1990.

Fujimoto, R. M. 1993. Parallel Discrete Event Simulation: Will the Field Survive? *ORSA Journal on Computing*, vol. 5(3), pp. 213-230.

Fujimoto, R. M. 1999. Exploiting Temporal Uncertainty in Parallel and Distributed Simulations. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*.

Fujimoto, R. M. 2000. *Parallel and Distributed Simulation Systems*: Wiley Interscience.

Fujimoto, R. M., K. S. Perumalla, A. Park, H. Wu, M. Ammar, and G. F. Riley. 2003. Large-Scale Network Simulation - How Big? How Fast? In *Proceedings of the 11th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*.

Garrett, Y., D. C. Christopher and K. Shivkumar. 2003. Large-Scale TCP Models Using Optimistic Parallel Simulation. In *Proceedings of the 17th Workshop on Parallel and Distributed Simulation.*

Gomes, F. 1996. Compiler Techniques for State Saving in Parallel Discrete Event Simulation, thesis, University of Calgary, Canada.

Jefferson, D. 1985. Virtual Time. *ACM Transactions on Programming Languages and Systems*, vol. 7(3), pp. 404-425.

Jha, V. and R. Bagrodia. 1994.A unified framework for conservative and optimistic distributed simulation. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*..

Karimabadi, H., J. Driscoll, Y. Omelchenko, K. S. Perumalla, R. M. Fujimoto, and N. Omidi. 2005. Parallel Discrete Event Simulation of Grid-based Models: Asynchronous Electromagnetic Hybrid Code. *International Conference on Applied Parallel Computing*, 2005.

Lamport, L. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, vol. 21(7), pp. 558-565.

Lee, B.-S., W. Cai and J. Zhou. 2001. A Causality Based Time Management Mechanism for Federated Simulations. In *Proceedings of the 15th Workshop on Parallel and Distributed Simulation*, pp. 83-90.

Liu, J. and D. Nicol. 2002. Lookahead Revisited in Wireless Network Simulations. In *Proceedings of the 16th Workshop on Parallel and Distributed Simulation*.

Loper, M. 2002. Approximate Time and Temporal Uncertainty in Parallel and Distributed Simulation, thesis, Georgia Institute of Technology.

Loper, M. and R. Fujimoto. 2004. A Case Study in Exploiting Temporal Uncertainty in Parallel Simulations. *International Conference on Parallel Processing*, 2004.

Mattern, F. 1993. Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation. *Journal of Parallel and Distributed Computing*, vol. 18(4), pp. 423-434.

Extensible Modeling and Simulation Framework (XMSF). 2006. Moves-Institute. Available online via <http://www.movesinstitute.org/xmsf/xmsf.html>.

Owens, J. D., D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, and T. J. Purcell. 2005. A Survey of General-Purpose Computation on Graphics Hardware. *Eurographics 2005*.

Park, A. and R. Fujimoto. 2006. Aurora: An Approach to High Throughput Parallel Simulation. In *Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation*.

Park, A., R. M. Fujimoto and K. S. Perumalla. 2004. Conservative Synchronization of Large-scale Network Simulations. In *Proceedings of the 18th Workshop on Parallel and Distributed Simulation*.

Perumalla, K. S. 1999. Techniques for Efficient Parallel Simulation and their Application to Large-scale Telecommunication Network Models, thesis, Georgia Institute of Technology.

libSynk, Perumalla, K. S. 2004. Available online via <http://www.cc.gatech.edu/computing/pads/kalyan/libsynk.htm>.

Perumalla, K. S. 2005. µsik - A Micro-Kernel for Parallel/Distributed Simulation Systems. In *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*.

Perumalla, K. S. 2006. Discrete Event Execution Alternatives on General Purpose Graphical Processing Units (GPGPUs). In *Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation*.

Perumalla, K. S. and R. M. Fujimoto. 2001. Virtual Time Synchronization over Unreliable Network Transport.

In *Proceedings of the 15th Workshop on Parallel and Distributed Simulation*.

Perumalla, K. S., R. M. Fujimoto and H. Karimabadi. 2006. Scalable Simulation of Electro-magnetic Hybrid Codes. In *Proceedings of the 6th International Conference on Computational Science*.

Pharr, M. and R. Fernando. 2005. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*: Addison Wesley Professional.

Quaglia, F. and A. Santoro. 2003. Non-blocking Checkpointing for Optimistic Parallel Simulation. *IEEE Transactions on Parallel and Distributed Systems*, vol. 14(6), pp. 593-610.

Rajaei, H., R. Ayani and L.-E. Thorelli. 1993. The Local Time Warp Approach to Parallel Simulation. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*.

Rao, D. M., N. V. Thondugulam, R. Radhakrishnan and P. A. Wilsey. 1998. Unsynchronized Parallel Discrete Event Simulation. In *Proceedings of the 1998 Winter Simulation Conference*, pp. 1563-1570. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Rosu, M., K. Schwan and R. M. Fujimoto. 1997. Supporting Parallel Applications on Clusters of Workstations: The Intelligent Network Interface Approach. *IEEE Symposium on High Performance Distributed Computing*, 1997.

Santoro, A. and R. M. Fujimoto. 2004. Off-Loading Data Distribution Management to Network Processors in HLA-Based Distributed Simulations. *Distributed Simulations and Real-Time Applications*, 2004.

Santoro, A. and F. Quaglia. 2006. Transparent Optimistic Synchronization in HLA via Time Management Converter. In *Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation*.

Simmonds, R., C. Kiddle and B. Unger. 2002. Addressing Blocking and Scalability in Critical Channel Traversing. In *Proceedings of the 16th Workshop on Parallel and Distributed Simulation*.

Tang, Y., K. S. Perumalla, R. M. Fujimoto, H. Karimabadi, J. Driscoll, and Y. Omelchenko. 2005. Optimistic Parallel Discrete Event Simulations of Physical Systems using Reverse Computation. In *Proceedings of the19th Workshop on Principles of Advanced and Distributed Simulation*.

Xiao, Z., B. Unger, R. Simmonds and J. Cleary. 1999. Scheduling Critical Channels in Conservative Parallel Discrete Event Simulation. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*.

Zhou, J., Z. Ji, M. Takai and R. Bagrodia. 2004. MAYA: Integrating hybrid network modeling to the physical world *ACM Transactions on Modeling and Computer Simulation*, vol. 14(2), pp. 149-169.

AUTHOR BIOGRAPHIES

KALYAN S. PERUMALLA is a senior researcher in the Computational Sciences and Engineering Division at the Oak Ridge National Laboratory. He also holds an adjunct faculty appointment with the College of Computing, Georgia Tech. He received a Ph.D. in Computer Science from Georgia Tech (1999). Dr. Perumalla has over 10 years of research and development experience in the area of parallel and distributed simulation systems, including high-performance runtime infrastructures and large-scale simulation, and has published widely on these topics. He co-developed the Federated Simulations Development Kit (FDK), a widely-disseminated high-performance runtime infrastructure for HLA-like distributed simulator federations. He has also built several additional research prototype systems and tools (e.g., for distributed debugging, network modeling, interoperable simulations and parallel optimization), most of which are in use by researchers worldwide. He has served as co-PI on multiple federally-funded projects on scalable parallel/distributed discrete event simulation systems. His e-mail is <perumallaks@ornl.gov> and his Web address is <http://www.ornl.gov/~2ip>.