Parallel Discrete-Event Simulation

Jason Liu
School of Computing and Information Sciences
Florida International University
Miami, Florida 33199
liux@cis.fiu.edu

February 9, 2009

Abstract

Parallel discrete-event simulation (PDES), simply referred to as parallel simulation, is concerned with the execution of discrete-event simulation on parallel computers. PDES has been recognized as a challenging research field bridging between modeling and simulation, and high-performance computing. By exploiting the potential parallelism in a simulation model, PDES can overcome the limitations imposed by sequential simulation both in the execution time and the memory space, and therefore demonstrate as a viable technique for solving large-scale complex models. In this article, we provide a brief overview of the current state of PDES, identify its fundamental challenges, and discuss existing principal solutions resulted from three decades of intensive research in this field. Further, we report specific research advances in high-performance modeling and simulation of large-scale computer networks as an exemplar of typical PDES applications.

1 Simulation and Parallelization Methods

1.1 Simulation with Discrete Events

Simulation—the practice of mimicking the operations of systems over time—is one of the most important and widely used techniques in operations research. These systems are abstracted as models in the form of mathematical or logical relationships. In simulation, one uses computers to evaluate the models numerically in a controlled environment, where data are gathered and used to estimate the behavior of the target systems. Simulation is effective for prototyping new system designs, as well as providing insight to the

true characteristics of existing systems. Simulation is particularly indispensable for studying large-scale and complex systems, which can be otherwise intractable to closed-form mathematical or analytical solutions. Due to the practical nature of the design process where simplified assumptions fly in the face of required complexities, simulation is an irrefutable choice for both system design and evaluation.

A simulation model specifies the state evolution over time. The target system can be viewed either as a *continuous* system, where state changes continuously with respect to time, as a *discrete* system, where state is modified only at specific points in time, or as a *hybrid* system, which consists of both continuous and discrete elements. In simulation, time progression is described by the so-called "time advancement function", which specifies two classes of simulation methods: time-driven and event-driven. In a *time-driven* (or *time-stepping*) simulation, time is measured at small intervals giving the impression that the system evolves continuously over time. As such, it is naturally more suitable for simulating continuous systems. In an *event-driven* (or *discrete-event*) simulation, time leaps through distinct points in time, which we call *events*. Consequently, event-driven simulation is more appropriate for simulating discrete systems. Note that one can combine both types of simulations, for example, in a computer network simulation, using discrete events to represent detail network transactions, such as sending and receiving packets, and using continuous simulation to capture the fluid dynamics of overall network traffic [34].

This article mainly focuses on discrete-event simulations and efficient techniques to parallelize them. It is necessary to first understand what constitutes a *sequential* discrete-event simulation before we proceed to review the parallelization methods. A discrete-event simulation maintains a data structure called the *event-list*, which is basically a priority queue that sorts events according to the time at which they are scheduled to happen in the simulated future. A clock variable T is used to denote the current time in simulation. At the heart of the program is a loop; the simulator repeatedly removes an event with the smallest timestamp from the event-list, sets the clock variable T to the timestamp of this event, and processes the event. Processing an event typically changes the state of the model and may generate more future events to be inserted into the event-list. The loop continues until the simulation termination condition is met, for example, when the event-list becomes empty or when the simulation clock has reached a designated simulation completion time.

1.2 Parallelization Methods

Parallel discrete-event simulation (PDES), also known simply as *parallel simulation*, is an important research area cross-cutting between modeling and simulation, and high-performance computing. PDES is concerned with executing a single discrete-event simulation program on parallel computers, which can be shared-memory multiprocessors, distributed-memory clusters of interconnected computers, or a combination of both. By exploiting the potential parallelism in the model, PDES can overcome the limitations imposed by sequential simulation in both the execution time and the memory space. As such, PDES can bring substantial benefit to time-critical simulations, and simulations of large-scale systems that demand an immense amount of computing resources.

The simplest form of parallel simulation is called *replicated trials* [23], which executes multiple instances of a sequential simulation program concurrently on parallel computers. This approach has the obvious advantage of simplicity and it can expedite the exploration of a large parameter space. The disadvantage is that each replicated trial does not provide any speedup and cannot overcome the memory limit due to sequential execution. To address the latter problem, Hybinette and Fujimoto [25] introduced a cloning method as an efficient parallel computation technique to allow simultaneous exploration of different simulation branches resulted from alternative decisions made in simulation.

Another form of parallel simulation is to assign different functions of a simulation program, such as random number generation and event handling, to separate processors. This method is called *functional decomposition* [11]. The main problem is the lack of ample parallelism. Also, the tight coupling of the simulation functions creates an excessive demand for communication and synchronization among the parallel components, which can easily defeat the parallelization effort.

More generally, one can view simulation as a set of state variables that evolve over time. Chandy and Sherman [10] presented a space-time view of simulation, where each event can be characterized by a temporal coordinate, indicated by the the timestamp of the event, and a spatial coordinate, indicated by the location of the state variables affected by the event. Accordingly, the state space of a discrete-event simulation can be perceived as consisting of a continuous time axis and a discrete space axis; the objective of the simulation is therefore to compute the value at each point in the space-time continuum. This space-time view provides a high-level unifying concept for parallel simulation, where one can divide the space-time graph

into regions of arbitrary shape and assign them to separate processors for parallel processing. For example, Bagrodia, Chandy, and Liao [3] presented a distributed algorithm using fixed-point computations.

A special case of the space-time view is the *time-parallel* approach, which is based on temporal decomposition of the time-space continuum. Time-parallel simulation divides the space-time graph along the time axis into non-overlapping time intervals, and assign them to different processors for parallel processing. Due to the obvious dependency issue, that is, the initial state of a time interval must match the final state of the preceding time interval, the efficiency of this approach relies heavily on the model's ability of either rapidly computing the initial state or achieving fast convergence under relaxation. For this reason, only a limited number of cases using time-parallel simulation exist in the literature. Successful examples include trace-driven cache simulations [22, 46], queuing network and Petri net simulations [1, 31], and road traffic simulations [28].

Orthogonal to the time-parallel approach, *space-parallel* simulation is based on data decomposition, where the target system is divided into a collection of subsystems, each simulated by a *logical process (LP)*. Each LP maintains its own simulation clock and event-list, and is only capable of processing events pertaining to the subsystem to which it is assigned. These LPs can be assigned to different processors and executed concurrently. Communications between the LPs take place exclusively by exchanging timestamped events. Space-parallel simulation is in general more robust than the other parallelization approaches, mainly because data decomposition is naturally applicable to most models. For this reason, we focus on space-parallel simulation for the rest of this article.

2 Synchronization Algorithms

In a discrete-event simulation, events need be processed in a non-decreasing timestamp order, because an event with a smaller timestamp has the potential to modify the state of the system and thereby affect events that happen later. This is what we call the *causality constraint*. Provided that simultaneous events—events with the same timestamps—are sorted deterministically and consistently using certain tie-breaking rules, the causality constraint implies a total ordering of events. In parallel simulation, the global event-list in sequential simulation is replaced by a set of event-lists; each LP maintains its own simulation clock and a separate event-list that contains events that can only affect the state of the corresponding LP. Since each LP pro-

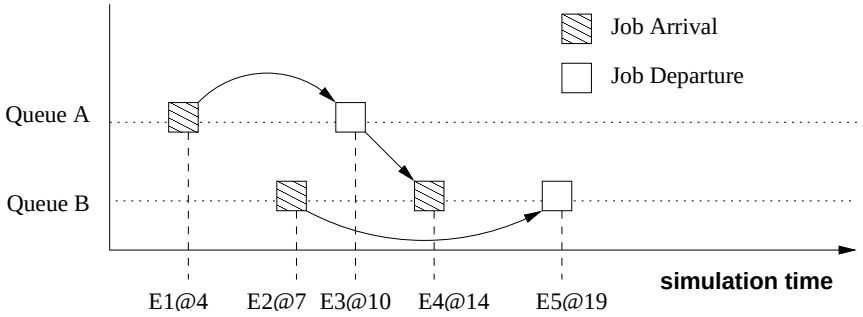

Figure 1: Simulation of Two Queues

cesses events on its own event-list in timestamp ordering—a property also known as the *local causality constraint*—the total ordering maintained by the original sequential discrete-event model is replaced by a partial ordering similar to Lamport's "happens before" relationship [29]. The fundamental challenge is therefore associated with the difficulty of preserving the local causality constraint at each LP without the use of a global simulation clock.

We use a simple example to illustrate this problem (Figure 1). Suppose there are two interconnected queues, A and B, with jobs in both queues waiting to be serviced. After a job is serviced at one queue, it joins the other queue after a certain delay. There are two types of simulation events: one representing the job arrival and the other representing the job departure. The two queues are simulated by two LPs running on two separate processors. Upon processing the "job departure" event at one LP, a "job arrival" event will be sent from the LP to the other LP. Figure 1 shows that a job enters Queue A at time 4 (represented by $E1$), departs at 10 ($E3$), and then travels to Queue B at 14 ($E4$), while another job enters Queue B at time 7 ($E2$) and leaves at 19 ($E5$). Since the two queues are simulated on separate processors, it is possible that the LP handling Queue B gets to process $E5$ immediately after processing $E2$, while the other LP is still in the middle of processing $E1$ at Queue A. That is, it is probable that an event (in this case, $E4$) could later arrive from the other LP carrying a timestamp smaller than the LP's current simulation clock. To avoid potential causality errors, some synchronization mechanism needs to be in place. Specifically, an LP must be able to make the right decision whether to process the smallest-timestamped event on its local event-list.

The distinction on how local causality constraint is enforced underscores the difference between two major parallel simulation methods. *Conservative parallel simulation* eliminates the possibility of any causality errors; that is,

an LP must be blocked from processing the next event in its event-list until it is sure that it will not cause out-of-order event execution due to future events from other LPs. In contrast, *optimistic parallel simulation* allows events to be processed out of order. Once a causality error is detected, the offending LP must be rolled back and recover from such an error. In order for the simulation to roll back from possible erroneous computations, state saving and recovery mechanisms must be provided.

3 Conservative Synchronization

3.1 The CMB Algorithm

The first parallel simulation synchronization protocol is the *CMB algorithm* proposed independently by Chandy and Misra [7], and Bryant [5]. In CMB, LPs are connected via directional links, through which events are transferred from one LP to another in chronological order (with nondecreasing timestamps). Events are enqueued at the receiving LPs—there is one input queue for each incoming link at an LP. Also, each input queue is associated with a clock variable, set to be either the timestamp of the first event in the queue, or, if the queue is empty, the timestamp of the last processed event (or zero initially). Each LP maintains a loop: at each iteration, the LP selects an input queue with smallest clock value and processes the event at the beginning of the input queue; if the input queue is empty, the LP blocks until an event arrives at the queue and then continues at the next iteration.

Since LPs block on empty input queues (with the smallest clock value), deadlock may happens once a waiting cycle is formed, in which case no progress will be made even if there are events in other input queues. The CMB algorithm uses *null messages* to avoid this pathological situation. A null message does not represent any real activities in the model; it carries only a timestamp and is regarded as a guarantee from the sending LP that it won't send events in the future with timestamps smaller than the timestamp of the null message. Upon receiving a null message, an LP can advance the clock associated with the input queue. The LP can further propagate the time advancement to its successor LPs, possibly by sending out more null messages. Consequently, no waiting cycle is formed and deadlock is avoided.

It is important to note that the use of null messages is not the only way to prevent deadlocks. Alternatively, one can allow deadlock to happen, and subsequently detect and recover from deadlock situations [8]. Deadlock recovery is based on the observation that events with the smallest timestamp

in the system can always be processed safely. In cases where deadlocks happen more frequently, however, this may result in a significant amount of sequential execution that can adversely affect the overall performance.

3.2 Exploiting Lookahead

The CMB algorithm introduced an important concept regarding local event processing. Each LP must determine the lower bound on the timestamps (LBTS) of future events to arrive from other LPs. LBTS is actually the upper bound up to which the LP can safely process its local events and advance its simulation clock without introducing causality errors. CMB uses a small time increment ahead the LP's current simulation clock as the timestamp of the null messages it sends to the successor LPs. This is the essential idea behind the concept of *lookahead*, which is defined as the amount of simulation time that an LP can predict into the simulated future (and therefore allow other LPs to safely advance their simulation clocks). In other words, lookahead is the inherent asynchrony in the simulation model—with a positive lookahead, an LP can process events within a certain range independent of its neighbors. Extensive performance studies (e.g., [14, 15, 51]) confirmed the positive correlation between lookahead and parallel performance of conservative algorithms.

Exploiting lookahead takes on two directions. One direction focuses on extracting lookahead from model characteristics. Nicol [45] provided a classification of lookahead based on different levels of knowledge that can be extracted from the model. Several models exhibit good lookahead properties. A notable example is the simulation of first-come-first-serve (FCFS) stochastic queuing networks [43]. By pre-sampling the job service time and determining branch destination at the time when a job enters the queue, one can improve the lookahead tremendously. Another example is the simulation of continuous-time Markov chains (CTMC) [21, 40, 41]. The method exploits the mathematical structure of CTMC models (more specifically, the uniformization property) that allows the pre-selection of synchronization points ahead of time. Consequently, the synchronization complexity therefore can be reduced significantly.

The other direction focuses lookahead extrapolation for general applications. Chandy and Sherman's *conditional event* approach [9] takes into account two types of events: *definite* events, which are bound to happen regardless of the earlier events, and *conditional* events, which can be canceled because of earlier events. This algorithm requires each LP determine the lower bound on the timestamp of future events it will send to each of its

neighbor LPs on the condition that its earliest local conditional event will not be canceled. This lower bound can be calculated as the timestamp of the earliest conditional event on the local event-list plus lookahead. Since the earliest conditional event in the system is actually a definite event, the algorithm uses a global min-reduction to find out the true lower bound and use it as the LBTS for all LPs to process their local events.

The topology of the model—how LPs are connected with each other—plays an important role in the lookahead computation. Lubachevsky's *bounded lag* algorithm [37] takes advantage of the minimum propagation delay between logical processes and the so-called opaque period within which the state of an LP is not affected by other LPs due to the model's non-preemptive behavior. The algorithm introduces a time interval B, called the *lag*, using which the algorithm computes the "sphere of influence", which is the set of LPs that can possibly affect a given LP within B units of simulation time. These are the LPs that need to be considered to determine whether events on the given LP with timestamps between T (the current simulation time) and $T + B$ can be safely processed. Ayani's *distance-between-objects* algorithm [2] also exploits the distance between the LPs; it uses a shortest-path algorithm to determine the LP's LBTS. The *time-of-next-event* algorithm proposed by Groselj and Tropper [20] also explores the LP topology; the algorithm considers situations where multiple LPs reside on a single processor and allows efficient computation of the greatest lower bound of all LPs on a processor. Nicol [44] established a scalability analysis based on a window-based synchronization algorithm. The algorithm is similar to the conditional-event approach in that it also uses a global min-reduction to determine the time of next synchronization point.

In general, LP-based conservative synchronization methods can be viewed as a scheduling problem where LPs are selected to run on parallel processors. Xiao et al. [62] introduced an asynchronous multi-level LP-scheduling algorithm, called the *critical channel traversing* (CCT) algorithm, to accommodate simulation models with small computational granularity. CCT schedules groups of LPs among the processors using a shared data structure to reduce unnecessary low-level synchronization overheads. Nicol and Liu [42] proposed a composite synchronization scheme combining both synchronous and asynchronous scheduling algorithms to avoid the potential performance pitfalls from either synchronous or asynchronous mechanism used alone.

4 Optimistic Synchronization

Optimistic synchronization enforces the local causality constraint differently from its conservative counterpart: an LP is allowed to process events that arrive in the simulated past, as long as the simulation is able to detect such causality errors, rewind the simulation clock, and roll back the erroneous computations.

4.1 Time Warp

It is safe to say by and large that Jefferson's *Time Warp* algorithm [26] is the most far-reaching parallel synchronization protocol. The protocol was first conceived to be offering a general solution to all parallel simulation problems, although the overhead associated with many aspects of the protocol's execution eventually caught up. As a result, the protocol set forth an era of intensive research in the PDES field.

In Time Warp, each LP saves the events received from other LPs in the input queue, and those sent to other LPs in the output queue. Also, the state variables are saved in the state queue each time before an event is processed. When an event arrives with a timestamp smaller than the current simulation clock (called the *straggler* event), the LP must be rolled back to the saved state immediately before the timestamp of the straggler event. All actions that the LP might have affected on other LPs later than the timestamp of the straggler event must also be canceled. This can be achieved by sending *anti-messages* corresponding to the original messages that are stored at the output queue. Upon receiving an anti-message, the LP will annihilate the corresponding message in the input queue. If the anti-message carries a timestamp smaller than the LP's current simulation clock (and thus become a straggler event), the LP will also be rolled back accordingly.

Simple and elegant as it appears to be, the algorithm must address the important issue of memory consumption: as the simulation progresses, the size of the input, output, and state queues could increase without bound. Furthermore, since rollback does not apply to irrevocable operations, such as I/O, the algorithm must also determine when these irrevocable operations can be executed. The *global virtual time* (GVT) is defined as the minimum timestamp of all events and messages (including both positive messages and anti-messages) in the system at a given wall-clock time. One can show that the system will never roll back to a time earlier than GVT. As such, GVT can be viewed as a measure of progress, where messages or state vectors

earlier than GVT can be reclaimed (in a process called *fossil collection*). Irrevocable operations that happen before GVT can also be committed.

Time Warp needs to overcome several practical problems pertaining to its performance: state saving inevitably introduces memory overhead; rollback allows more complexities, where state needs to be restored, and anti-messages need to be sent, which can possibly cause further rollbacks rippling through the entire system. These problems have prompted a lot of interesting research, some of which we describe next.

4.2 State Saving, GVT and Rollback

State saving is necessary to undo modifications to the state variables caused by erroneous computations. There are three types of state-saving techniques: *copy state saving* makes a copy of all state variables each time an event is processed; *incremental state saving* saves only those state variables modified as a result of processing an event; and *infrequent state saving* adjusts the interval between checkpoints and reduces the frequency of state saving. Incremental state saving can be made automatic, by overloading the assignment operators in certain object-oriented programming languages [55], by using specific hardware support [16], or by directly modifying the executable files [61]. To enable infrequent state saving, the rollback mechanism must be able to handle situations where an LP may be rolled back to a state not immediately before the straggler event. In this case, the LP must *coast forward*, re-processing events until the desired state is reached. For this reason infrequent state saving must be applied judiciously. Lin et al. [33] studied the trade-off between the state-saving interval length and the rollback cost, and suggested an analytically optimal solution.

GVT computation is a major source of overhead for Time Warp. The task of computing GVT amounts to capturing a consistent snapshot of the state of a distributed system. According to Samadi [56], there are two major issues: the *transient message* problem and the *simultaneous reporting* problem. A transient message is a message that has been sent but not yet received; either the sender or the receiver LP must account for the message in the GVT computation, or inconsistency would happen. Simultaneous reporting is also related to the possible miscount of transient messages; it is, however, prompted by the difference in the ordering of events perceived by different LPs. Samadi's algorithm [56] employs a message acknowledgment scheme to solve the transient message problem and adopts a barrier synchronization method to solve the simultaneous reporting problem. To avoid the expensive barrier synchronization, Preiss's algorithm [49] arranges the

LPs in a ring and uses a token traversal approach to compute GVT. Mattern's algorithm [38] uses two broadcasts to define separate "cuts" and uses messages with "colors" to distinguish those spanning across the cuts.

Optimizations can also be applied to reduce the overhead from rollbacks. Gafni [19] proposed *lazy cancellation* to avoid canceling messages that are more likely to be re-created after rollbacks. The algorithm does not immediately send anti-messages during a rollback; instead, they are sent only when the same positive messages are not re-created. West [60] proposed another method called *lazy re-evaluation* to avoid re-processing the events, which can be achieved by comparing the state vectors. Both methods require additional time for comparing messages or state vectors, which could cause more erroneous computation to be spread further.

One interesting alternative to the state-saving approach is *reverse computation*, proposed by Carothers, Perumalla, and Fujimoto [6]. Rather than saving the state to enable rollback, reverse computation performs the inverses of the individual operations that correspond to the normal event processing (i.e., by executing the code backwards) to get the system back to an earlier state. This method alleviates the state-saving cost from the forwarding computation path and transfers the cost to the reverse computation path. It can also be viewed as trading the memory space (which would otherwise be consumed by state saving) with the execution time (for the more expensive rollback). Significant improvements over traditional state-saving methods have been reported both in run time and memory utilization for fine-grain models, including queuing network models. One potential problem for reverse computation is that not all operations are reversible, in which case the method is degenerated to the traditional state-saving approach. The increased rollback cost may also get erroneous computations to be propagated to more LPs possibly causing unstable behavior.

4.3 Curbing Optimism

In practice, "pure" Time Warp usually does not provide an acceptable performance. Many alternative solutions have been proposed that effectively limit optimistic execution to circumvent various performance hazards.

Lin and Preiss [32] first introduced the concept of *storage optimality*. A storage optimal parallel simulator can complete the execution of any model using no more than the memory space required to complete a sequential execution multiplied by some constant factor. To achieve storage optimality, a necessary condition is that the parallel simulator should be able to reclaim the memory space beyond the sequential execution requirement, even if it

implies restraint in the speculative execution. Several algorithms run on shared-memory multiprocessors qualify for the storage optimality, which include Jefferson's *cancel-back* algorithm [27], and Lin and Preiss's *artificial rollback* algorithm [32]. Preiss and Loucks's *prune-back* algorithm [50] is designed for distributed-memory architectures; however, it fails to meet the storage optimality requirement.

A number of optimistic parallel simulation algorithms are proposed to curb optimism. For example, the *moving time window* approach [57] uses a window to control how far an LP may get ahead of other LPs. The effectiveness of this algorithm obviously depends on the size of the moving time window. Reynolds [52] introduced a classification method for optimistic synchronization protocols based on two attributes: *aggressiveness* and *risk*. Aggressiveness specifies that events on an LP can be processed out of timestamp order, where risk suggests that messages generated by aggressive event processing are allowed to be sent to other LPs. For example, the SRADS protocol by Dickens and Reynolds [13] is aggressive, but allows no risk: it permits only local rollbacks; messages are not sent unless the LP is sure that the messages will not be rolled back. As an extension, the *breathing time bucket* approach, proposed by Steinman [58], introduced a quantity called *event horizon*. An LP's local event horizon is the smallest timestamp of any new messages that are generated as a result of processing local events. The global event horizon can be computed as the minimum of the local event horizons among all LPs. The algorithm allows only messages with a sending time prior to the global event horizon to be sent out. By doing that, it can be shown that these messages will never be rolled back.

Many have tried to compare conservative and optimistic simulations; yet it comes at no surprise that there is no conclusive answer to which is a better approach. This is mainly because parallel simulation performance largely depends on the characteristics of the simulation model. On the one hand, conservative synchronization, due to its low operational overhead and small memory footprint, can usually achieve good performance as long as good lookahead can be extracted from the model. On the other hand, the optimistic approach can exploit the full parallelism in the model through speculative execution. However, optimistic execution requires state saving, GVT computation, fossil collection and rollbacks, which could bring signification design complexity and memory overhead.

5 High-Performance Modeling and Simulation of Large-Scale Networks

PDES has been applied in many areas, such as military applications (including war games and training exercises), on-line gaming, business operations, manufacturing, logistics and distribution, transportation, computer systems and computer networks. In this section, we review the latest development in high-performance modeling and simulation of large-scale computer networks as an example of successful PDES applications.

Internet is described by Paxson and Floyd [47] as "an immense moving target": its size, heterogeneity, and rapid change pose a daunting challenge for accurately simulating its complex behavior. Over the years, researchers have developed several parallel simulators that can model networks at unprecedented scales. For example, Riley, Fujimoto, and Ammar [53] used a federated approach to parallelize the popular *ns-2* simulator [4]. The original sequential discrete-event simulation engine is "refurbished" with parallel functions to allow time-synchronized event distribution. There are also unadulterated efforts in developing parallel network simulations from scratch. One early effort is the Telecommunications Description Language (TeD), proposed by Perumalla, Ogielski, and Fujimoto [48], which allows parallel execution of large network models with an optimistically synchronized parallel simulation engine.

The Scalable Simulation Framework (SSF) was later developed as a common Application Programming Interface (API) for parallel simulations of large-scale telecommunication systems [12]. SSF is different from TeD in that it does not require a separate programming language; SSF network models are written in common programming languages (Java or C++) using simple simulation constructs that support transparent conservative parallel execution on different platforms. A similar effort is the Georgia Tech Network Simulator (GTNetS) [54], which is a full-fledged simulation environment embellished with a rich set of network protocol implementations. The GTNetS effort has recently evolved into *ns-3* [24], which is a renewed effort for open community development of network simulations. Both SSF and GTNetS are developed based on conservative synchronization protocols. ROSSNet [63], on the other hand, is a large-scale network simulator using an optimistic approach. It offers a compact and light-weight implementation framework that reduces the amount of state required to simulate large-scale network models. ROSSNet uses reverse computation to enable fast optimistic execution on commodity multi-processor systems.

Two important application areas using high-performance network simulation have shown very promising results: on-line simulation and real-time simulation. *On-line network simulation* refers to the use of network simulation as an integrated service for real-time network management with the goal of improving parameter tuning, network planning, network monitoring, and network traffic engineering (e.g., [59, 64]). Parallel simulation allows detailed performance analysis faster than real time and thus can be used to predict, control and fine-tune network parameters in real time. *Real-time network simulation* refers to simulation of large-scale networks in real time so that the virtual network can interact with real implementations of network protocols, network services, and distributed applications. Real-time simulation allows network immersion, where a simulated network is made indistinguishable from a physical network in terms of conducting network traffic between real applications. It provides the necessary realism, scalability, and flexibility for experimental networking research [35].

6 Additional Readings

This article is but scratching the surface of the exciting field of parallel discrete-event simulation. Due to space limitations, we are not able to provide an in-depth study of the problems and solutions. There are many excellent surveys of the parallel discrete-event simulation field (e.g., [17, 30, 36, 39]). Fujimoto's book on PDES [18] provides an in-depth review of the intriguing problems and solutions in the area.

References

[1] H. Ammar and S. Deng. Time Warp simulation using time scale decomposition. *ACM Transactions on Modeling and Computer Simulation*, 2(2):158–177, April 1992.

[2] R. Ayani. A parallel simulation scheme based on the distance between objects. *Proceedings of the 1989 SCS Multiconference on Distributed Simulation*, 21(2):113–118, March 1989.

[3] R. Bagrodia, K. M. Chandy, and W. T. Liao. A unifying framework for distributed simulation. *ACM Transactions on Modeling and Computer Simulation*, 1(4):348–385, October 1991.

[4] L. Breslau, D. Estrin, K. Fall, S. Floyd, J. Heidemann, A. Helmy, P. Huang, S. McCanne, K. Varadhan, Y. Xu, and H. Yu. Advances in network simulation. *IEEE Computer*, 33(5):59–67, May 2000.

[5] R. E. Bryant. Simulation of packet communication architecture computer systems. Technical Report MIT-LCS-TR-188, MIT, 1977.

[6] C. D. Carothers, K. S. Perumalla, and R. M. Fujimoto. Efficient optimistic parallel simulations using reverse computation. *ACM Transactions on Modeling and Computer Simulation*, 9(3):224–253, July 1999.

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

[8] K. M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 24(4):198–205, April 1981.

[9] K. M. Chandy and R. Sherman. The conditional event approach to distributed simulation. *Proceedings of the 1989 SCS Multiconference on Distributed Simulation*, 21(2):93–99, March 1989.

[10] K. M. Chandy and R. Sherman. Space-time and simulation. *Proceedings of the 1989 SCS Multiconference on Distributed Simulation*, 21(2):53–57, March 1989.

[11] John C. Comfort. The simulation of a master-slave event set processor. *Simulation*, 42(3):117–124, March 1984.

[12] J. H. Cowie, D. M. Nicol, and A. T. Ogielski. Modeling the global Internet. *Computing in Science & Engineering*, 1(1):42–50, January 1999.

[13] P. M. Dickens and P. F. Reynolds, Jr. SRADS with local rollback. *Proceedings of the 1990 SCS Multiconference on Distributed Simulation*, 22(1):161–164, January 1990.

[14] R. M. Fujimoto. Lookahead in parallel discrete event simulation. *Proceedings of the 1988 International Conference on Parallel Processing*, pages 34–41, August 1988.

[15] R. M. Fujimoto. Performance measurements of distributed simulation strategies. *Transactions of the Society for Computer Simulation*, 6(2):89–132, April 1989.

[16] R. M. Fujimoto. The virtual time machine. *Proceedings of the 1st Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'89)*, pages 199–208, June 1989.

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

[18] R. M. Fujimoto. *Parallel and distributed simulation systems.* John Wiley & Sons, New York, 2000.

[19] A. Gafni. Rollback mechanisms for optimistic distributed simulation systems. *Proceedings of the 1988 SCS Multiconference on Distributed Simulation*, 19(3):61–67, July 1988.

[20] B. Groselj and C. Tropper. The time of next event algorithm. *Proceedings of the 1988 SCS Multiconference on Distributed Simulation*, 19(3):25–29, July 1988.

[21] P. Heidelberger and D. Nicol. Conservative parallel simulation of continuous time Markov chains using uniformization. *IEEE Transactions on Parallel and Distributed Systems*, 4(8):906–921, August 1993.

[22] P. Heidelberger and H. S. Stone. Parallel trace-driven cache simulation by time partitioning. *Proceedings of the 1990 Winter Simulation Conference (WSC'90)*, pages 734–737, December 1990.

[23] Shane G. Henderson. Input model uncertainty: why do we care and what should we do about it? In *Proceedings of the 35th Conference on Winter Simulation (WSC'03)*, volume 1, pages 90–100, December 2003.

[24] Thomas R. Henderson, Sumit Roy, Sally Floyd, and George Riley. ns-3 project goals. In *Proceeding of the 2006 Workshop on ns-2: the IP network simulator (WNS2)*, October 2006.

[25] Maria Hybinette and Richard Fujimoto. Cloning parallel simulations. *ACM Transactions on Modeling and Computer Simulation*, 11(4):378–407, October 2001.

[26] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–245, July 1985.

[27] D. R. Jefferson. Virtual time II: Storage management in distributed simulation. *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, pages 75–89, August 1990.

[28] T. Kiesling and J. Luthi. Towards time-parallel road traffic simulation. In *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation (PADS 2005)*, pages 7–15, June 2005.

[29] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[30] Y. B. Lin and P. Fishwick. Asynchronous parallel discrete event simulation. *IEEE Transactions on Systems, Man, and Cybernetics*, 26(4):397–412, July 1996.

[31] Y. B. Lin and E. D. Lazowska. A time-division algorithm for parallel simulation. *ACM Transactions on Modeling and Computer Simulation*, 1(1):73–83, January 1991.

[32] Y. B. Lin and B. R. Preiss. Optimal memory management for Time Warp parallel simulation. *ACM Transactions on Modeling and Computer Simulation*, 1(4):283–307, October 1991.

[33] Y. B. Lin, B. R. Preiss, W. M. Loucks, and E. D. Lazowska. Selecting the checkpoint interval in Time Warp simulation. *Proceedings of the 7th Workshop on Parallel and Distributed Simulation (PADS'93)*, pages 3–10, May 1993.

[34] Jason Liu. Packet-level integration of fluid TCP models in real-time network simulation. In *Proceedings of the 2006 Winter Simulation Conference (WSC'06)*, pages 2162–2169, December 2006.

[35] Jason Liu. A primer for real-time simulation of large-scale networks. In *Proceedings of the 41st Annual Simulation Symposium (ANSS'08)*, pages 307–323, April 2008.

[36] Y. H. Low, C. C. Lim, W. Cai, S. Y. Huang, W. J. Hsu, S. Jain, and S. J. Turner. Survey of languages and runtime libraries for parallel discrete-event simulation. *Simulation and Transactions of the Society for Computer Simulation (SCS), Joint Special Issue on Parallel and Distributed Simulation*, 72(3):170–186, March 1999.

[37] B. D. Lubachevsky. Bounded lag distributed discrete event simulation. *Proceedings of the 1988 SCS Multiconference on Distributed Simulation*, 19(3):183–191, July 1988.

[38] F. Mattern. Efficient distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 18(4):423–434, August 1993.

[39] D. Nicol and R. Fujimoto. Parallel simulation today. *Annals of Operations Research*, 53:249–286, December 1994.

[40] D. Nicol and P. Heidelberger. Optimistic parallel simulation of continuous time Markov chains using uniformization. *Journal of Parallel and Distributed Computing*, 18(4):395–410, August 1993.

[41] D. Nicol and P. Heidelberger. A comparative study of parallel algorithms for simulating continuous time Markov chains. *ACM Transactions on Modeling and Computer Simulation*, 5(4):326–354, October 1995.

[42] D. Nicol and J. Liu. Composite synchronization in parallel discrete-event simulation. *IEEE Transactions on Parallel and Distributed Systems*, 13(5):433–446, May 2002.

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

[44] D. M. Nicol. The cost of conservative synchronization in parallel discrete event simulations. *Journal of the ACM*, 40(2):304–333, April 1993.

[45] D. M. Nicol. Principles of conservative parallel simulation. *Proceedings of the 1996 Winter Simulation Conference (WSC'96)*, pages 128–135, December 1996.

[46] D. M. Nicol, A. G. Greenberg, and B. D. Lubachevsky. Massively parallel algorithms for trace-driven cache simulations. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):849–858, August 1994.

[47] V. Paxson and S. Floyd. Why we don't know how to simulate the Internet. *Proceedings of the 1997 Winter Simulation Conference (WSC'97)*, pages 1037–1044, December 1997.

[48] K. Perumalla, A. Ogielski, and R. Fujimoto. TeD—a language for modeling telecommunication networks. *ACM SIGMETRICS Performance Evaluation Review*, 25(4):4–11, March 1998.

[49] B. R. Preiss. The Yaddes distributed discrete event simulation specification language and execution environments. *Proceedings of the 1989 SCS Multiconference on Distributed Simulation*, 21(2):139–44, March 1989.

[50] B. R. Preiss and W. M. Loucks. Memory management techniques for Time Warp on a distributed memory machine. *Proceedings of the 9th Workshop on Parallel and Distributed Simulation (PADS'95)*, pages 30–39, June 1995.

[51] D. A. Reed, A. D. Malony, and B. D. McCredie. Parallel discrete event simulation using shared memory. *IEEE Transactions on Software Engineering*, 14(4):541–53, April 1988.

[52] P. F. Reynolds, Jr. A spectrum of options for parallel simulation. *Proceedings of the 1988 Winter Simulation Conference (WSC'88)*, pages 325–332, December 1988.

[53] G. F. Riley, R. M. Fujimoto, and M. H. Ammar. A generic framework for parallelization of network simulations. *Proceedings of the 7th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'99)*, pages 128–135, October 1999.

[54] George F. Riley. The Georgia Tech network simulator. In *Proceedings of the ACM SIGCOMM Workshop on Models, Methods and Tools for Reproducible Network Research (MoMeTools'03)*, pages 5–12, August 2003.

[55] R. Rönngren, M. Liljenstam, J. Montagnat, and R. Ayani. Transparent incremental state saving in Time Warp parallel discrete event simulation. *Proceedings of the 10th Workshop on Parallel and Distributed Simulation (PADS'96)*, pages 70–77, May 1996.

[56] B. Samadi. *Distributed simulation, algorithms and performance analysis*. PhD thesis, Department of Computer Science, UCLA, 1985.

[57] L. M. Sokol, D. P. Briscoe, and A. P. Wieland. MTW: A strategy for scheduling discrete simulation events for concurrent execution. *Proceedings of the 1988 SCS Multiconference on Distributed Simulation*, 19(3):34–42, July 1988.

[58] J. S. Steinman. Breathing Time Warp. *Proceedings of the 7th Workshop on Parallel and Distributed Simulation (PADS'93)*, pages 109–118, May 1993.

[59] Boleslaw K. Szymanski, Adnan Saifee, Anand Sastry, Yu Liu, and Kiran Madnani. Genesis: a system for large-scale parallel network simulation. In *Proceedings of the 16th Workshop on Parallel and Distributed Simulation (PADS 2002)*, pages 89–96, May 2002.

[60] D. West. Optimizing Time Warp: Lazy rollback and lazy re-evaluation. Master's thesis, Department of Computer Science, University of Calgary, January 1988.

[61] D. West and K. Panesar. Automatic incremental state saving. *Proceedings of the 10th Workshop on Parallel and Distributed Simulation (PADS'96)*, pages 78–85, May 1996.

[62] Z. Xiao, B. Unger, R. Simmonds, and J. Cleary. Scheduling critical channels in conservative parallel discrete event simulation. *Proceedings of the 13th Workshop on Parallel and Distributed Simulation (PADS'99)*, pages 20–28, May 1999.

[63] Garrett R. Yaun, David Bauer, Harshad L. Bhutada, Christopher D. Carothers, Murat Yuksel, and Shivkumar Kalyanaraman. Large-scale network simulation techniques: examples of TCP and OSPF models. *ACM SIGCOMM Computer Communication Review*, 33(3):27–41, July 2003.

[64] Tao Ye, Hema T. Kaur, Shivkumar Kalyanaraman, and Murat Yuksel. Large-scale network parameter configuration using an on-line simulation framework. *IEEE/ACM Transactions on Networking*, 16(4):777–790, August 2008.