Proceedings of the 2019 Winter Simulation Conference
N. Mustafee, K.-H.G. Bae, S. Lazarova-Molnar, M. Rabe, C. Szabo, P. Haas, and Y.-J. Son, eds.

TIME WARP SIMULATION ON MULTI-CORE PLATFORMS

Philip A. Wilsey

Department of Electrical Engineering and Computer Science
University of Cincinnati
PO Box 210030
Cincinnati, OH 45221-0030, USA

ABSTRACT

Parallel Discrete Event Simulation (PDES) is concerned with the parallel and distributed execution of discrete event simulation models. This tutorial reviews parallel computing and the properties of Discrete Event Simulation (DES) models and then examines the construction of PDES solutions that use the Time Warp Synchronization Mechanism. A review of the general challenges to building high-performance Time Warp Synchronized PDES on multi-core processors and clusters composed of multi-core processors is presented. These challenges include considerations of the general quantitative metrics exhibited by DES simulation models as well as a review of the impact that small overheads and serial portions of a program can have on the potential peak performance of a parallel solution. Finally, directions for future opportunities with heterogeneous computing, domain specific hardware solutions, and edge computing will be explored.

1 INTRODUCTION

Parallel computing hardware has become ubiquitious. Outside of inexpensive processors designed for low-cost embedded work, virtually all processors today are released with multiple-cores and in many cases also containing simultaneous multi-threading (Hennessy and Patterson 2019). Parallel Discrete Event Simulation (PDES) attempts to harness these parallel computing platforms. While simulation run time is often a motivating objective for PDES, numerous other motivating factors for PDES also exist. For example, PDES enables the exploitation of the multiplicity of resources available from a collection of computers, it also facilitates the integration (federation) of distinct simulation models, it facilitates the scaling to (and study of) very large simulation models, and so on (Fujimoto 1990; Fujimoto 2000). Traditionally this execution is on multi-core/many-core processors, big-iron parallel computing hardware (*e.g.,* IBM Blue Gene systems) or Beowulf clusters. However, emerging hardware solutions such as GPGPUs (Perumalla 2006), FPGAs (Rahman et al. 2019), Domain Specific Architectures (DSAs) (Hennessy and Patterson 2019), and, more generally, heterogeneous computing platforms provide exciting opportunities for PDES.

Discrete Event Simulation is a good candidate application for parallelization as it is easily decomposed into a collection of discrete event simulation functions that can concurrently process events. In many cases, the simulation models are quite large with ample potential parallelism. The design of a concurrent execution of a DES Model requires that the simulation engine implement some type of synchronization mechanism that coordinates the concurrent execution to ensure correct execution. There are numerous synchronization mechanisms including organized as either centralized or distributed solutions (Fujimoto 1990; Fujimoto 2000). The distributed synchronization mechanism are generally classified as either *conservative* (Bryant 1979; Chandy and Misra 1979; Chandy and Misra 1981) or *optimistic* (Jefferson 1985; Fujimoto 1990). This tutorial is concerned with implementing an optimistically synchronized PDES using the Time Warp Mechanism (Jefferson 1985). The particular focus in this tutorial will be building and deploying a high-performance solution that operates effectively on multi-core processors and clusters containing multi-core processors.

The remainder of this paper is organized as follows. Section 2 contains a background discussion on parallel simulation and outlines how a discrete event simulation is commonly organized for parallelization. Section 3 discusses the challenges of parallelism as well as reviewing some of the significant properties of discrete event simulation that challenge the attainment of speedup. This section also discusses some of the opportunities that discrete event simulation contribute for effective parallelization. Finally, this section also discusses some of the parallel hardware and programming structures that can contribute to better or worse parallel performance for the fine-grained application that PDES is. Section 4 reviews the main components of a Time Warp simulation and discusses the various sub-algorithms for each. Section 5 discusses some of the work already performed with GPGPUs, FPGAs and Domain Specific Architectures (DSAs) for PDES. Finally, Section 6 contains some concluding remarks. Some data and graphs in this tutorial are taken from (Child and Wilsey 2012; Alt and Wilsey 2014; Crawford et al. 2017).

2 BACKGROUND

A DES model captures the behavior of a real or planned system as discrete actions or transitions through a simulated time period. These discrete changes in the modeled system can occur at distinct and irregular simulated time stamps. In general, a physical system is organized as a collection of Physical Processes (PPs) that are each modeled by a *Logical Process* (LP) in the DES model. The LPs generally contain state variables that collectively record the state of of the physical system. State transition triggers are captured by *events* that have a time stamp denoting when that state transition is planned to occur. Thus, for example, a DES model of a logic circuit could represent the logic gates as LPs and the signal changes on the wires between the gates as events. Simulation of a DES model then orders the events by their time stamps and uses the LPs to effect the transitions defined by the events. Termination occurs whenever some termination condition is satisfied.

Conceptually the LPs of a DES model can all be considered independent discrete event simulation engines that maintain a local simulation time and execute events denoting transitions for that LP. This concurrent execution requires synchronization of the LPs to ensure that the correct orderly execution of all events in the system occurs (Lamport 1978). Initially some central synchronization system was used to provide synchronization (Fujimoto 1990). In one embodiment, this central system would operate something like a clock broadcasting advances of a global synchronizing time boundary that the parallel LPs would execute against. In another embodiment, the central system would hold the pool of events to be executed and in a step-by-step manner distribute a next set of events that could all be executed simultaneously (while preserving their causal order). As with most systems built around a centralized control unit, the handshaking and control enforces strict causal orders that inhibits maximal parallelism. These shortcomings lead to the development of distributed mechanism to synchronize parallel simulation.

Distributed synchronization mechanisms for parallel simulation are generally classified as being either *conservative* (Bryant 1979; Chandy and Misra 1979; Chandy and Misra 1981) or *optimistic* (Jefferson 1985; Chandy and Sherman 1989). Conservatively synchronized PDES implements a distributed synchronization protocol that *strictly maintains the causal order of events throughout their parallel execution*. In this case each LP processes events only when it is certain that the event is the next to be processed and no other event with an earlier timestamp will arrive for processing. This requires that all events generated and sent by an LP are sent in their timestamp order and that the message passing subsystem deliver and process messages in a first-in-first-out (FIFO) order. Each LP then will monitor the FIFO queue from *all* of the LPs that send event messages to it and will process the event with the smallest timestamp. If the FIFO queue from any sending LP is empty, the receiving LP must block until another event message arrives on that queue. This can lead to deadlock. The deadlock can be broken by having the receiving LP send back a *null* message to the LP with an empty FIFO queue to request an updated time boundary for any new event messages that it might send to the receiving LP. This updated time boundary is generally computed with the assistance of a *lookahead* value $\mathscr{L}$. The lookahead value permits an LP to communicate a guarantee of how much time will occur before another event will be sent to the receiving LP. That is, if the sending LP

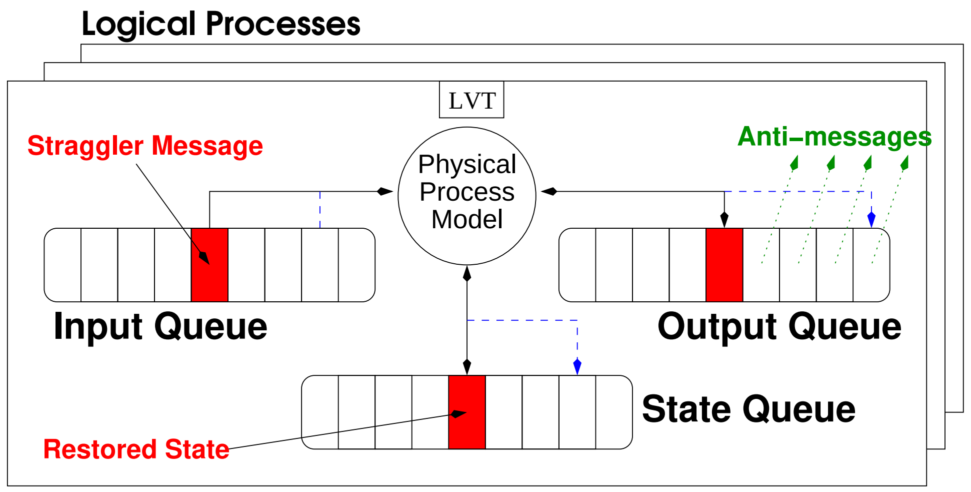

Figure 1: System view of a Time Warp Simulation.

is at simulation time $\mathscr{T}$ and its lookahead value is $\mathscr{L}$, then any event message sent by the sending LP will have a timestamp of at least $\mathscr{T}+\mathscr{L}$. The above conservative synchronization is commonly described as the Chandy/Misra/Bryant (CMB) algorithm and frequently conservative synchronization is used synonymously with CMB, there are other conservative mechanisms that will guarantee events are processed in their causal order. Some examples of other conservative mechanisms are: Bounded Lag (Lubachevsky 1988), YAWNS (Nicol 1993) and Breathing Time Buckets (Steinman 1991).

In contrast, optimistically synchronized PDES *does not strictly enforce the causal order of events* and instead builds some mechanism to repair the system whenever a causal violation occurs. The optimistic nature of these methods permit the parallel simulation engines to process events aggressively without the overheads associated with an enforcement of the strict causal order. Optimistic methods assume that the causal orders are generally satisfied and then implement mechanisms to recover whenever a causal violation is discovered. Obviously these methods work well when events can be generated and communicated to the receiving LP before that LP can advance to the timestamp of the incoming events. When this happens the optimistic methods benefit from the removal of the synchronization overhead. However, when causal violations occur, there is a performance penalty for its restoration. Probably the most commonly used and studied optimistic synchronization method is the Time Warp Mechanism that was introduced by (Jefferson 1985). Effective implementation of a Time Warp synchronized parallel simulation engine designed for high-performance execution on a multi-core processor and cluster is the main topic of this tutorial. A more detailed description of the Time Warp Mechanism is presented in the next section.

Both conservative and optimistic synchronization methods have been and are widely researched. While there has historically been a friendly rivalry between researchers of the two camps (Fujimoto et al. 2017), neither approach has been shown to be consistently superior to the other and work continues to advance developments and optimizations to each.

2.1 PDES with Time Warp

The *Time Warp* mechanism is an implementation of the *Virtual Time* paradigm proposed by Jefferson (Jefferson 1985). In a Time Warp simulation, the system under investigation is decomposed into a set of LPs that operate as asynchronously communicating discrete-event simulators (Jefferson 1985). An illustration of a Time Warp synchronized simulation is shown in Figure 1. This figure depicts a collection of LPs show as a card deck style stack. Each LP in the stack operates concurrently, processing the input events in their timestamp order and communicating with other LPs by exchanging timestamped events as they are generated. Each LP maintains its own simulation clock called the Local Virtual Time (LVT), which is the timestamp of the event last processed by that LP. To correctly simulate a physical system, the LPs must process the input events in their timestamp order. An LP continues to receive and process events from its input event queue until there are no more unprocessed events or until an event arrives whose timestamp is lower than the LVT of that LP. Such an out-of-order message is called a *straggler event* (or simply *straggler*). When a straggler arrives at an LP, the LP suspends event processing and performs a *rollback*. A *rollback* consists of restoring the state of the LP to the exact point where the straggler should have been processed (had it arrived in order), and canceling the side-effects of the premature lookahead computation done by the LP. In order to facilitate a rollback, each LP maintains an *Input Queue* which is the event queue of the LP that contains all the processed and unprocessed events, a *State Queue* that contains snapshots of the LP's state after each event has been processed (assuming that the state of the process is checkpointed every event), and an *Output Queue* that contains exact negative copies (called *anti-messages*) of the messages sent out by the LP. Figure 1 illustrates rollback of the queues (shown in red) with the blue dashed lines showing the premature advance of the simulation that occurred prior to the arrival of the straggler.

The purpose of an anti-message is to curtail the spread of the erroneous computation by annihilating the corresponding positive message (henceforth, the terms event and messages will be used interchangably to denote a timestamped event) from the input queue of the receiving process. Whenever a positive message meets its corresponding anti-message they *annihilate* each other, leaving no trace that either ever existed. If the annihilated positive message has not yet been processed, then the LP receiving the anti-message simply discards the two. If the message has been processed, then the LP receiving the anti-message must first be rolled back to undo the effect of processing the incorrect message. This rollback, may, in turn, cause other anti-messages to be sent out. Recursively repeating the above procedure allows all the effects of the erroneous computation to be canceled.

Out-of-order messages are the primary cause of causality errors in a Time Warp simulator. Since different LPs reside on different processors and since there is no synchronization, it is possible that there are LPs which may be working in the simulation time future of other LPs. If an LP residing on a processor in the past sends a message to an LP residing on a processor in the future, the object receiving that message may have already moved past the message's receive-time, in which case the work done by the object before the arrival of straggler may be in error (Reiher et al. 1990). These erroneous messages can have cascading effects. One out-of-order message can result in the generation of other out-of-order messages containing erroneous data, which in turn can generate many more, causing the error to spread across the system. For the simulation to proceed along the correct path, the damage done by these erroneous messages must be undone.

Cancellation is the mechanism used to remove incorrectly sent output messages from the distributed computation. Depending on the sending time of the anti-messages, two methods exist for canceling incorrect messages in a Time Warp simulator, namely *Aggressive Cancellation* (AC) and *Lazy Cancellation* (LC) (Fujimoto 1990; Fujimoto 2000). In AC, anti-messages are dispatched to the receiving processes immediately upon the arrival of a straggler. AC works under the assumption that an out-of-order event always produces incorrect output. In LC, the LP waits until it is sure that the messages that it had sent out during the erroneous lookahead will not also be sent by the proper computation. Anti-messages are sent for only those output messages which are generated differently after the rollback.

3 CHALLENGES AND OPPORTUNITIES FOR PDES

DES Models provide a number of challenges and opportunities to the construction of high-performance PDES. On the one hand, they are generally easily decomposable so that a significant number of concurrent LPs can be defined. It is not difficult to find simulation models containing 10-100K LPs and those of 1M or more are not uncommon. That said, the execution run time of a single event on one of these LPs is generally very small, on the order of $10\mu s$ or less. While one of the principle ideas of the Time Warp mechanism is to remove the overhead/synchronization costs of maintaining the strict causal orders of all events as they

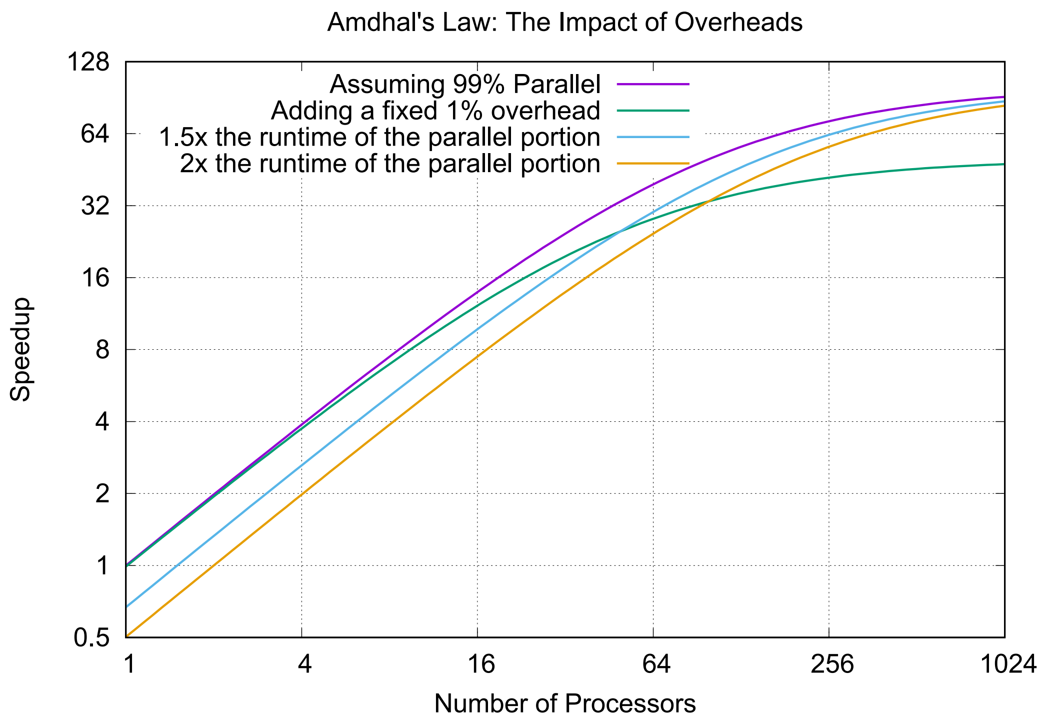

Figure 2: Using Amdhal's Law (Hennessy and Patterson 2019) to illustrate the impacts that small fixed and variable overheads have on the potential for strong scaling speedups.

are executed, this relaxation is made possible only with some additional costs such as state savings (or at least the implementation of reverse computation (Carothers et al. 1999)), fossil collection, and global time management. The Time Warp algorithm also complicates mechanisms to observe global conditions of the simulation which can lead to additional costs to detect, for example, termination conditions (Chandy and Lamport 1985). The remainder of this section has two objectives. The first is to examine the above issues in more detail. The second is to examine some aspects of parallel programming and hardware mechanisms that are significant to achieving performant PDES implementations.

3.1 Small Overheads Matter

While the promise of parallelism is attractive and seems like it should provide speedups that are nearly linear in the amount of parallelism, gaining scalable speedup from parallelism is challenging. Minor overheads can easily destroy any potential speedup. The application level challenges to achieving effective speedup comes in two forms, namely: (i) the fraction of the original program that is not parallelizable, and (ii) the overheads added to the sequential components when they are parallelized (*e.g,* both in terms of additional instructions: locks and such, as well as contention for shared resources). While the sequential portion may seem insignificant, even a 1% sequential component has significant impact at scale.

The impact that each of the above characterized overheads has on potential speedup is illustrated in Figure 2. The curves in the legend (top to bottom) are computed to show: (i) ideal parallelism with no overheads, (ii) a fixed added overhead of 1% of the original computation added back to the result, (iii) an overhead of $1.5x$ added to each parallel task (that is each parallel task executes 1.5 times the sequential component it replaces), and (iv) an overhead of $2x$ added to each parallel task. Thus, for example, the fixed cost might denote the added cost to initiate the parallelism and the variable cost denoting the synchronization or other additional overheads. Note these curves assume an ideal problem that is decomposed into equal

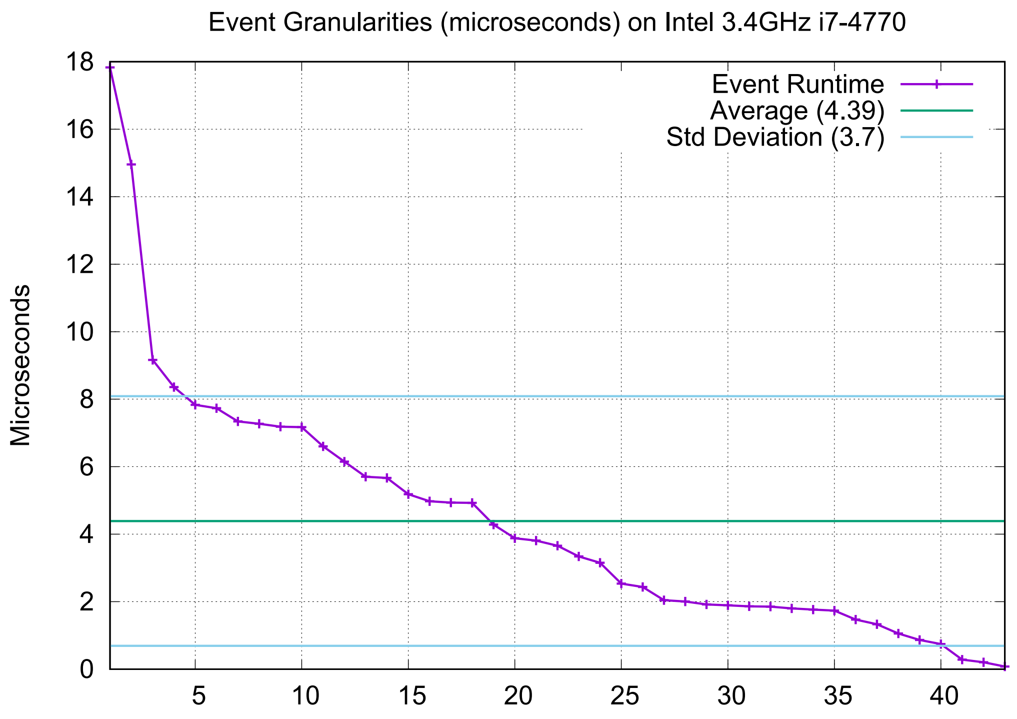

Figure 3: Event execution run time for a variety of simulation models from the WARPED2 and ROSS simulation repositories. Run time data captured with simulation kernels configured for sequential execution.

sized compute units without constraints or limits. The curves show that even in the idealized case with no overheads the speedup the sequential portion limits total speedup to 99 — *the sequential component matters*. Equally interesting is that even a small fixed cost added to the sequential portion is more punishing than a variable cost added to each parallel portion. Thus, while it is important to mange cost additions to the parallel elements, it is perhaps even more important to address and minimize any remaining sequential portion of the solution.

3.2 Properties of DES Models

Discrete Event Simulation models (DES Models) appear to be excellent candidates for parallelization and in many cases, this is true. They generally have a large number of components that are easily parallelized, they are often scaled to very large sizes (sometimes exceeding the computational and/or memory capacities a single compute node), and they can have very long execution run times. That said, they also have some considerations that challenge the attainment of speedup from PDES execution. In particular, the processing grainularities of discrete event simulation models is generally quite small. Figure 3 shows the run time costs to process an event for 9 distinct simulation models in 43 different configurations/sizes (sometimes configuration size can have significant impact on event granularity). These models range in size from 100s of LPs to 1M LPs. The data shows that the average event execution time is 3.7μs on a 3.4GHz Intel CPU which translates to approximately 15K instructions per event. Thus, one must be careful to limit expansion of the code required to execute events, especially when something as simple as an atomic instruction has a run time cost that is 10–30 times longer than a standard integer instruction. Acquiring locks and lock contention can easily add significant overhead to process an event.

In contrast to the small event granularity, many DES Models have a very large number of LPs that can be executed in parallel and in many cases, these LPs communicate most of their generated events to a

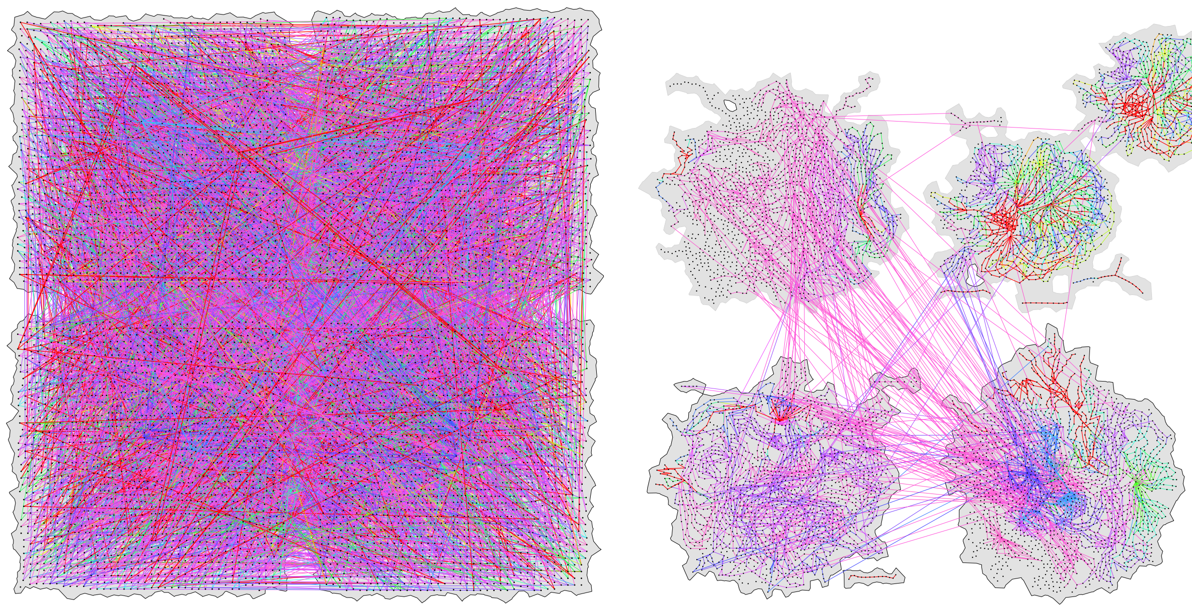

Figure 4: Heatmap of events communicated between LPs in two different partitions of a simulation model. The left image is from a random partitioning. The right image is from a profile guided partitioning.

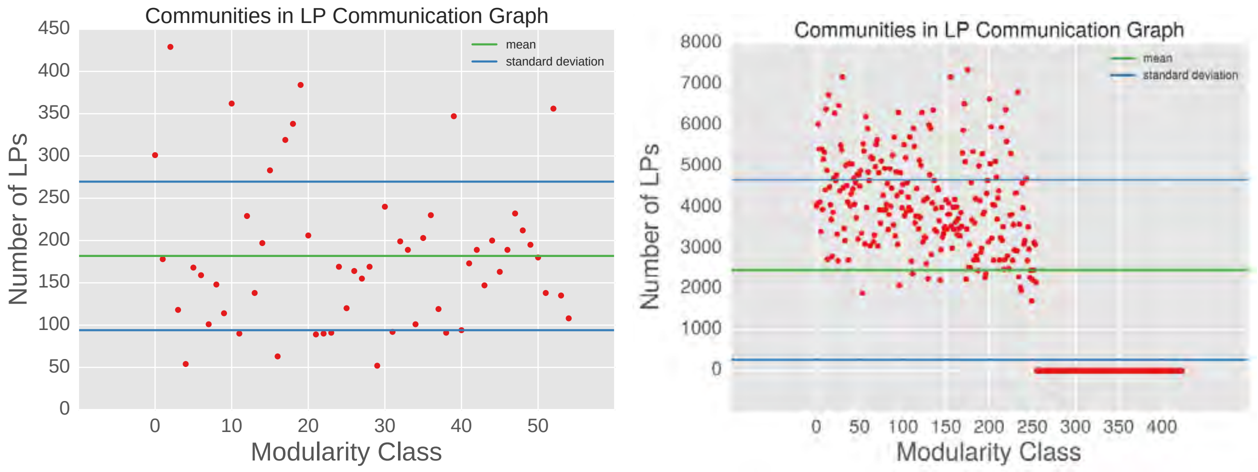

Figure 5: Examples of the Communities found in two simulation models taken from (Crawford et al. 2017). The model data on the left is epidemic model from the WARPED2 code base (Weber 2016) and the model data on the right is the PCS model from the ROSS code base (Carothers et al. 2000). The x-axis simply enumerates the found communities; the y-axis records the number of LPs in each community.

small community of other LPs (Crawford et al. 2017). Thus, it is generally possible to organize the LPs into semi-distinct partitions for parallel execution. The impact of this is visually illustrated by heatmaps of event communications between LPs shown in Figure 4 (taken from (Alt and Wilsey 2014)). The left image is a random partitioning whereas the right image is a partition that is created from profile data captured in a pre-simulation step. This is more extensively examined in (Crawford et al. 2017). More specifically, that work examines the *modularity* of the event communication graph between the LPs. The modularity measure how well sub-networks (or *communities*) compose a large network. This study showed that many simulation models had many (40-250) subnetworks of more frequent communicating LPs. An example of two of these graphs are shown in Figure 5. These modularity results can be used for partitioning and, as will be shown later, efficient partitioning is critical to achieving high-performance run time with minimal rollback.

3.3 Parallel Programming Primitives

Transactional memory (TM) is a concurrency control mechanism that attempts to eliminate the static sequential execution of a critical section by dynamically determining when accesses to shared resources can be executed concurrently (Rajwar and Goodman 2001). Transactional memory operates on the same principles as database transactions (Harris et al. 2010). The processor atomically commits *all* memory operations of a successful transaction or discards *all* memory operations if the transaction should fail (a collision to the updates by the multiple threads occurs). In order for a transaction to execute successfully, it must be executed in isolation, *i.e.,* without conflicting with other transactions/threads memory operations. This is the key principle that allows transactional memory to expose untapped concurrency in multi-threaded applications. There are both hardware and software solutions to provide transactional memory to the user.

The concept of transactional memory is appealing as it simplifies locks and potentially side-steps some of the contention issues. Transactional memory works best when concurrent threads access distinct regions of a shared data structure. Unfortunately, the critical shared resource of PDES is the pool of pending events which can be a highly contended resource in a multi-threaded environment. As a result, transactional memory does not significantly improve PDES performance (Hay and Wilsey 2015).

4 TIME WARP

This section addresses implementation strategies for the significant components needed for the implementation of a Time Warp synchronized PDES solution. By far the most significant of these is the organization of the pending event set and the binding of event processing threads to LPs. An effective implementation of the pending event set can result in a nearly 100% commit rate of events in a single SMP processing node. In clusters, the secondary challenge is message latency and its impact on the late delivery of event messages communicated between the various cluster nodes.

Overall, the our studies with WARPED2 (Gupta and Wilsey 2017; Gupta 2018) have shown that the best organization of threading is to organize the threads on each multi-core processing node into *worker threads* that strictly execute events (including rollback and state savings) and a single *manager thread* that maintains the Time Warp housekeeping functions such as GVT and termination. In general one worker thread per hardware processing thread delivers the best performance, although scaling drops off significantly once the number of threads exceeds the physical core count (entering the shared SMT thread space). Oddly enough, preserving one hardware thread for O/S operation (and binding the O/S to that hardware thread) does not noticeably impact performance. Likewise, experiments with the WARPED2 kernel show the operation of the manager thread is sufficiently lightweight and non-intrusive to the operation of the worker threads that is does not appear to require a separate hardware thread.

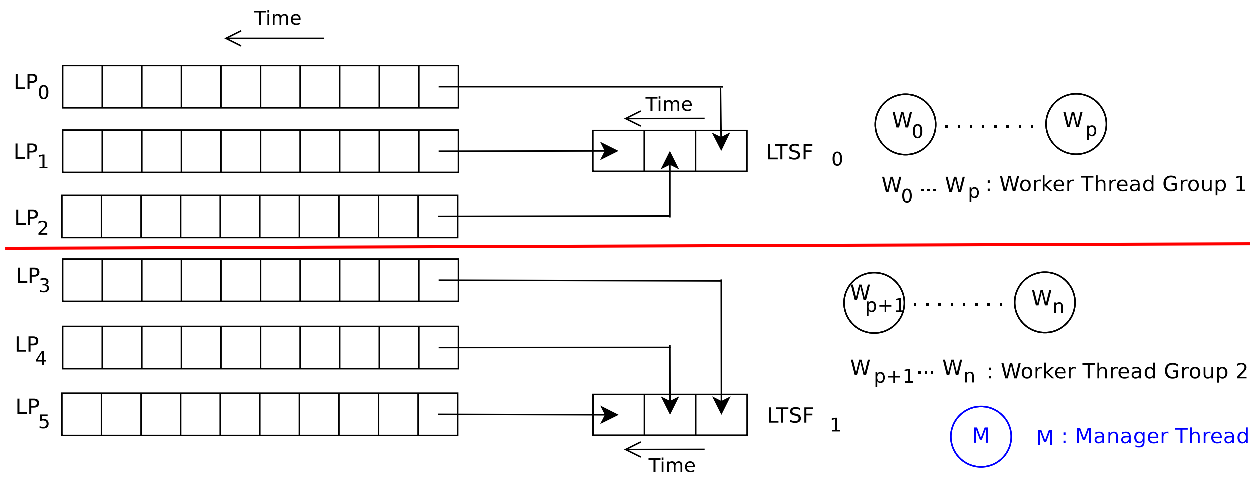
Time
LP0
LP1
LP2
LTSF 0
W0
Wp
W0 ... Wp : Worker Thread Group 1
LP3
LP4
LP5
LTSF 1
Wp+1
Wn
Wp+1 ... Wn : Worker Thread Group 2
M
M : Manager Thread

Figure 6: Organizing the LPs, Threads, and Pending Event set.

4.1 Logical Processes, Threads, and the Pending Event Set

The original WARPED Time Warp simulation kernel was designed for high-performance execution on a cluster of single node processors. However, the widespread emergence of multi-core processors motivated a need to explore the addition of threads to the solution. Two separate studies (Miller 2010; Dickman et al. 2013) gave considerable insights into a general design solution that has formed the basis for the WARPED2 design solution (Weber 2016). The crucial elements of the general solution are depicted in Figure 6. In this design, the LPs maintain their local simulation time progress as well as their input, output, and state event queues (only the input queues are shown). The pool of LPs are assigned to a single *Lowest TimeStamped First* (LTSF) schedule queue. The input queues and LTSF queues are all shared data structures with mutex locks controlling their access. As outlined below, this organization has been extensively studied to explore and contrast the performance of different software architectures and data structures.

Contention to the shared data structure holding the pending event set can significantly impede performance of a PDES solution (Dickman et al. 2013; Gupta and Wilsey 2017; Gupta 2018). Explorations on the pending event set design with the WARPED (Miller 2010; Dickman et al. 2013) and WARPED2 (Gupta and Wilsey 2014; Gupta 2018) have evolved a design solution that results a software architecture and programming solution that is able to commit nearly 100% of events without rollback for many different DES model types (Gupta 2018). These studies included explorations with transactional memory (Hay and Wilsey 2015), lock-free data structures (Gupta and Wilsey 2014), partitioning of LPs to separate schedule queues (Dickman et al. 2013; Gupta 2018), scheduling event groups and bags (Gupta and Wilsey 2017; Gupta 2018) have all been conducted. Likewise, experiments various with data structures such as the Calendar Queue (Brown 1988), Ladder Queue (Tang et al. 2005), Splay Tree (Sleator and Tarjan 1985) or the C++ STL MultiSet have all conducted (Gupta 2018). In all of these experiments, one consistent result emerges, namely that instantiating one worker threads per hardware thread and partitioning the LPs to a separate LTSF queue for each worker thread will consistently deliver the best performance. While the experimental results also favor the Ladder Queue, the performance benefits are less significant or consistent and in many cases, the MultiSet or Splay tree deliver good performance as well.

The solution with WARPED2 uses profile-guided partitioning to distributed the work evenly among the worker threads. While this is highly effective, it may prove impractical for very large simulation models that of a size where the modularity analysis cannot be performed or where the profile data for even a small pre-simulation run grow very large. To address this issue, the ROSS (Carothers et al. 2000) kernel provides that capability for a model-based partitioning solution. An alternate is to use some unstructured (or weakly structured) partitioning solution and rely on load balancing to balance the workload. As described below, dynamic load balancing is not unknown to the PDES community.

Reiher and Jefferson implemented dynamic load balancing for the Time Warp Operating System (TWOS) (Reiher and Jefferson 1990). Processes are migrated based on the measurement of a metric called *effective utilization* which is intended as a replacement for the raw CPU utilization metric often employed in conventional load balancing schemes. Effective utilization is the fraction of work being done by an LP that will not be rolled back, and it is estimated by tracking the CPU cycles spent by an LP doing work that is not rolled back. LPs with low effective utilization are paired with LPs with high effective utilization at regular intervals, and some amount of load is transferred.

Schlagenhaft *et al* migrates clusters of simulation objects between simulators (Schlagenhaft et al. 1995). A metric called *Integrated Virtual Time* (IVT) is used to gather information about the virtual time progress of simulation processes. The rate of change of two IVTs with respect to real time is used as a measure of load. The load is balanced by moving topology information, states, and events to the new simulator.

Sarkar and Das determine the degree of load imbalance of LPs by calculating the rate of virtual time progression with respect to real time (Sarkar and Das 1997). Each LP then determines whether it is over(under) loaded by checking if its LVT is under(over) the min(max) estimate for GVT. Each under(over) loaded LP finds a neighbor that is over(under) loaded, and transfers some amount of load such that the LVT of the two LPs is more balanced.

These load balancing approaches all use global information to determine a source and destination of the load migration. The work presented in this thesis is different in that no load migration occurs, but global load information is needed to be able to match pairs of LPs for over and under-clocking. The author of this thesis adopts a rollback-based load information metric as opposed to the virtual time based approaches of (Schlagenhaft et al. 1995; Sarkar and Das 1997), as it more accurately conveys the amount of real work being done by each LP (Reiher and Jefferson 1990; Palaniswamy and Wilsey 1996).

4.2 Messages, Anti-Messages, and Cancellation

On a single node shared memory the communication and canceling of events is generally done by direct insertion and removal of events in the pending event set. One important and useful optimization for cancellation is to implement *direct cancellation* (Fujimoto 1989). Under direct cancellation when an event is generated, a pointer to its location in the input queue of the receiving LP is stored in the sending LPs output queue (Figure 1). This permits immediate and direct to the shared memory location where an event can be removed; anti-messages to LPs in other processing nodes must, however, still be sent as a communication to the remote location. However, it is also possible to send a "block" anti-messages to remote sites that denotes a range of timestamps for which events should be canceled. Finally, the decision as to when to generate the anti-messages must be discussed. There are three options for this, namely: *aggressive cancellation*, *lazy cancellation*, or *dynamic cancellation* (Fujimoto 1990; Fujimoto 2000). Aggressive cancellation sends the anti-message immediately on rollback; lazy cancellation sends an anti-message only when re-processing after rollback determines that the original, prematurely sent, message is incorrect; and dynamic cancellation is a configuration where the cancellation policy (aggressive/lazy) is determined by each LP at run time based on its past behavior. The chief problem is that neither cancellation strategy is clearly superior to the other for all applications (Reiher et al. 1990). In general, however, the WARPED2 solution favors an aggressive cancellation policy. This decision is made for two primary reasons. First, the code is compact and adds less overhead to the event processing (in lazy cancellation new and old output events must be compared whenever event re-processing occurs after a straggler event). Second, the general solution of WARPED2 usually experiences few rollbacks on a single node and, in a cluster setting, the prospect of experiencing cascading rollbacks (Tay et al. 1998) with the long latency in some message passing frameworks motivations an aggressive stance when it comes to message cancellation.

4.3 Maintaining LP State

Mechanisms to support rollback of an LP state must be implemented in a Time Warp solution. The three key solutions to address this issue are: (i) reverse computation (Carothers et al. 1999; Carothers et al. 2000), (ii) incremental state savings (Bauer and Sporrer 1993), and (iii) full copy state savings (Fleischmann and Wilsey 1995). Full copy state savings will make a complete copy of the LP state that can be restored on a rollback (Fujimoto 1990). A complete copy can be made after each event processed by the LP or after some number of events are processed (called periodic state savings). When periodic state savings is performed, it is necessary to implement a coast-forward phase where the events between the saved state and the straggler event must be reprocessed to generate the correct state for the straggler. During this coast forward time, any re-generated events are suppressed. Incremental state savings saves only the subparts of the state that are changed when an event is processed. On rollback, the simulator must rebuild the state stepwise using the incremental state information. Reverse computation operates by defining an inverse function for each LP step backward through each undone event. In cases where the LP function does not have an inverse, incremental state savings is performed. Reverse computation works very well whenever the LPs have inverse functions. If the LPs do not have inverse functions, reverse computation reduces to incremental state savings. Depending on the size of the state and the amount it changes with each event processed will help dictate the choice between incremental and periodic state savings. Periodic state savings can provide great benefit if the saving period can be lengthened so that the time spent saving is less than incremental. That said, if rollbacks are frequent, then the coasting forward costs can accumulate under periodic checkpointing. In the WARPED2 kernel, a periodic state saving implementation is used. This is due to the relatively low rollback frequency that the WARPED2 kernel incurs with most simulation models.

4.4 GVT, Fossil Collection, and Termination

Global Virtual Time (GVT) records the global progress of the entire simulation. GVT is used whenever some global understanding of the simulated system progress is required. For example, the ability to commit output file writes is governed by GVT; likewise, the ability to perform *fossil collection* to reclaim saved state and event information that is no longer necessary to sustain rollback is guided by GVT. GVT can be computed exactly by stopping the simulation temporarily to determine its value (Bauer et al. 2005). Alternatively, GVT can be estimated by an on-line algorithm that operates simultaneously with normal event processing (Mattern 1993; Tomlinson and Garg 1993). While many algorithms have been developed to compute/estimate GVT, unless memory is at a premium, generally any lightweight algorithm will suffice. The frequency and speed of GVT estimation can be adjusted by prioritizing (or de-prioritizing) the estimation algorithm. If highly accurate GVT is needed due to memory considerations, a synchronous method can be triggered by memory availability. Alternatively, the *optimistic fossil collection* (Young and Wilsey 1996; Young et al. 1998) algorithm has a tight lower memory bound; although it introduces the notion of catastrophic failure of the simulation that can be recovered by file based checkpointing. The WARPED2 system using a lightweight variant of Mattern's algorithm (Mattern 1993).

5 HETEROGENEOUS PLATFORMS

Recently there has been an increasing effort to build, deploy, and exploit heterogeneous parallel computing platforms. While work with GPGPUs has been popular for a number of years, more recent directions have been to integrate even more exotic plugin cards containing FPGAs, custom ASICs, and tightly coupled general purpose parallel hardware (*e.g.*, Knights Landing). GPGPUs and FPGAs for PDES have been studied by a number of investigators (Perumalla 2006; Rahman et al. 2019). Likewise PDES experiences with parallel hardware PCIe daughter cards have been reported (Williams et al. 2017). Many of these studies report speedups under certain constrained situations. The primary challenges to speedups lie with limitations of memory or slow data transfer rates over the PCIe backplane. However, new hardware with larger memories and next generation PCIe standards are currently emerging. The 4.0 PCIe standard doubles the unidirectional bandwidth to 32GB/s and 5.0 doubles it again to 63GB/s. These higher transport speeds coupled with larger onboard memories for these plugin cards may well make heterogeneous solutions highly effective. Finally, new directions with *Domain Specific Architectures* deployed as PCIe plugin cards are receiving considerable attention by the architecture and machine learning communities (Hennessy and Patterson 2019). While these platforms are targeting machine learning at least one (the Microsoft Catapult) is a general purpose FPGA/NIC card that has custom programming capabilities. The future holds much promise for heterogeneous PDES computing.

6 CONCLUSIONS

Parallel and distributed discrete-event simulation is an important research area that is ripe for the effective exploitation of parallel heterogeneous computing hardware. That said, the effective exploitation of the hardware still requires careful design considerations with a disciplined focus on the management of overheads (both fixed and variable). On a single node multi-core platform, a Time Warp PDES solution can be developed with minimal lock contention and will a nearly zero rollbacks. The depolyment on muti-core cluster platforms must also contend with message latency issues between nodes. Fortunately opportunities for higher performance/lower-latency solutions my lie with domain specific architecture solutions. As these hardware platforms become faster, with larger memories, and more unique capabilities additional research will need to be performed to better understand how to exploit these systems.

ACKNOWLEDGMENTS

This material is based upon work supported by the AFOSR under award No FA9550–15–1–0384.

REFERENCES

Alt, A., and P. A. Wilsey. 2014. “Profile Driven Partitioning of Parallel Simulation Models”. In *Proceedings of the 2014 Winter Simulation Conference*, edited by A. Tolk, S. Y. Diallo, I. O. Ryzhov, L. Yilmaz, S. Buckley, and J. A. Miller, 2750–2761. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Bauer, D., G. Yaun, C. D. Carothers, M. Yuksel, and S. Kalyanaraman. 2005. “Seven-O’Clock: A New Distributed GVT Algorithm Using Network Atomic Operations”. In *Proceedings of the Workshop on Parallel and Distributed Simulation*, PADS ’05, 39–48. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Bauer, H., and C. Sporrer. 1993. “Reducing Rollback Overhead in Time-Warp Based Distributed Simulation with Optimized Incremental State Saving”. In *Proceedings of the 26th Annual Simulation Symposium*, 12–20. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Brown, R. 1988. “Calendar Queues: A Fast O(1) Priority Queue Implementation for the Simulation Event Set Problem”. *Communications of the ACM* 31(10):1220–1227.

Bryant, R. E. 1979. “Simulation on a Distributed System”. In *Proceedings of the 1st International Conference on Distributed Computing Systems*, 544–552. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Carothers, C. D., D. Bauer, and S. Pearce. 2000. “ROSS: A High-performance, Low Memory, Modular Time Warp System”. In *Proceedings of the Fourteenth Workshop on Parallel and Distributed Simulation*, PADS ’00, 53–60. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Carothers, C. D., K. S. Perumalla, and R. M. Fujimoto. 1999. “Efficient Optimistic Parallel Simulations Using Reverse Computation”. *ACM Transactions on Modeling and Computer Simulation* 9(3):224–253.

Chandy, K. M., and L. Lamport. 1985. “Distributed Snapshots: Determining Global States of Distributed Systems”. *ACM Transactions on Computer Systems* 3(1):63–75.

Chandy, K. M., and J. Misra. 1979. “Distributed Simulation: A Case Study in Design and Verification of Distributed Programs”. *IEEE Transactions on Software Engineering* 5(5):440–452.

Chandy, K. M., and J. Misra. 1981. “Asynchronous Distributed Simulation via a Sequence of Parallel Computations”. *Communications of the ACM* 24(11):198–206.

Chandy, K. M., and R. Sherman. 1989. “Space-Time and Simulation”. In *Proceedings of the SCS Multiconference on Distributed Simulation*, Volume 21, 53–57. San Diego, CA: Society for Computer Simulation.

Child, R., and P. A. Wilsey. 2012. “Using DVFS to Optimize Time Warp Simulations”. In *Proceedings of the 2012 Winter Simulation Conference*, 1–12. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Crawford, P., S. J. Eidenbenz, P. D. Barnes Jr., and P. A. Wilsey. 2017. “Some Properties of Communication Behaviors in Discrete-Event Simulation Models”. In *Proceedings of the 2017 Winter Simulation Conference*, edited by W. K. V. Chan, A. D’Ambrogio, G. Zacharewicz, N. Mustafee, G. Wainer, , and E. Page, 1025–1036. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Dickman, T., S. Gupta, and P. A. Wilsey. 2013. “Event Pool Structures for PDES on Many-Core Beowulf Clusters”. In *Proceedings of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM PADS ’13, 103–114. New York, NY, USA: Association of Computing Machinery.

Fleischmann, J., and P. A. Wilsey. 1995. “Comparative Analysis of Periodic State Saving Techniques in Time Warp Simulators”. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation (PADS 95)*, 50–58. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Fujimoto, R. 1990. “Parallel Discrete Event Simulation”. *Communications of the ACM* 33(10):30–53.

Fujimoto, R. M. 1989. “Time Warp on a Shared Memory Multiprocessor”. *Transactions of Society for Computer Simulation* 6(3):211–239.

Fujimoto, R. M. 2000. *Parallel and Distribution Simulation Systems*. New York, NY, USA: John Wiley & Sons, Inc.

Fujimoto, R. M., R. Bagrodia, R. E. Bryant, K. M. Chandy, D. Jefferson, J. Misra, D. Nicol, and B. Unger. 2017. “Parallel Discrete Event Simulation: The Making of a Field”. In *Proceedings of the 2017 Winter Simulation Conference*, edited by W. K. V. Chan, A. D’Ambrogio, G. Zacharewicz, N. Mustafee, G. Wainer, , and E. Page, 262–291. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Gupta, S. 2018. *Pending Event Set Management in Parallel Discrete Event Simulation*. Ph. D. thesis, Department of Electrical Engineering and Computer Science, University of Cincinnati, Cincinnati, Ohio. http://rave.ohiolink.edu/etdc/view?acc_num=ucin1535701778479768, accessed September 2019.

Gupta, S., and P. A. Wilsey. 2014. “Lock-free Pending Event Set Management in Time Warp”. In *Proceedings of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, PADS 2014, 15–26. New York, NY, USA: Association of Computing Machinery.

Gupta, S., and P. A. Wilsey. 2017. “Quantitative Driven Optimization of a Time Warp Kernel”. In *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, PADS 17, 27–38. New York, NY, USA: Association of Computing Machinery.

Harris, T., J. R. Laurus, and R. Rajwar. 2010. “Transactional Memory, 2nd ed.”. *Synthesis Lectures on Computer Architecture* 5(1):1–263.

Hay, J., and P. A. Wilsey. 2015. “Experiments with Hardware-Based Transactional Memory in Parallel Simulation”. In *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, edited by S. J. E. Taylor, N. Mustafee, and Y.-J. Son, 75–86. New York, NY, USA: Association of Computing Machinery.

Hennessy, J. L., and D. A. Patterson. 2019. *Computer Architecture: A Quantitative Approach*. 6th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Jefferson, D. 1985. “Virtual Time”. *ACM Transactions on Programming Languages and Systems* 7(3):405–425.

Lamport, L. 1978. “Time, Clocks, and the Ordering of Events in a Distributed System”. *Communications of ACM* 21(7):558–565.

Lubachevsky, B. D. 1988. “Bounded Lag Distributed Discrete Event Simulation”. In *Proceedings of the 1988 SCS Multiconference on Distributed Simulation*, 183–191. San Diego, CA: Society for Computer Simulation.

Mattern, F. 1993. “Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation”. *Journal of Parallel and Distributed Computing* 18(4):423–434.

Miller, R. J. 2010. “Optimistic Parallel Discrete Event Simulation on a Beowulf Cluster of Multi-core Machines”. Master’s thesis, Department of Electrical Engineering and Computer Science, University of Cincinnati, Cincinnati, Ohio. http://rave.ohiolink.edu/etdc/view?acc_num=ucin1282322836, accessed September 2019.

Nicol, D. M. 1993. “The Cost of Conservative Synchronization in Parallel Discrete Event Simulations”. *Journal of the ACM* 40(2):304–333.

Palaniswamy, A., and P. A. Wilsey. 1996. “Parameterized Time Warp: An Integrated Adaptive Solution to Optimistic PDES”. *Journal of Parallel and Distributed Computing* 37(2):134–145.

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*, PADS ’06, 74–81. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Rahman, S., N. Abu-Ghazaleh, and W. Najjar. 2019. “PDES-A: Accelerators for Parallel Discrete Event Simulation Implemented on FPGAs”. *ACM Transactions on Modeling and Computer Simulation* 29(2):12:1–12:25.

Rajwar, R., and J. R. Goodman. 2001. “Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution”. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-34)*, 294–305.

Reiher, P. L., R. M. Fujimoto, S. Bellenot, and D. Jefferson. 1990. “Cancellation Strategies in Optimistic Execution Systems”. In *Proceedings of the SCS Multiconference on Distributed Simulation*, Volume 22, 112–121. San Diego, CA: Society for Computer Simulation.

Reiher, P. L., and D. Jefferson. 1990. “Dynamic load management in the Time Warp Operating System”. *Transactions of the Society for Computer Simulation* 7(2):91–120.

Sarkar, F., and S. K. Das. 1997. “Design and Implementation of Dynamic Load Balancing Algorithms for Rollback Reduction in Optimistic PDES”. In *Proceedings of the 5th International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, MASCOTS ’97, 26–31. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc. Computer Society.

Schlagenhaft, R., M. Ruhwandl, C. Sporrer, and H. Bauer. 1995. “Dynamic Load Balancing of a Multi-Cluster Simulator on a Network of Workstations”. In *Proceedings of the Ninth Workshop on Parallel and Distributed Simulation*, PADS ’95, 175–180. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc. Computer Society.

Sleator, D., and R. Tarjan. 1985. “Self Adjusting Binary Search Trees”. *Journal of the ACM* 32(3):652–686.

Steinman, J. S. 1991. “SPEEDES: A Unified Approach to Parallel Simulation”. In *6th Workshop on Parallel and Distributed Simulation*, 75–84. San Diego, CA: Society for Computer Simulation.

Tang, W. T., R. S. M. Goh, and I. L.-J. Thng. 2005. “Ladder Queue: An O(1) Priority Queue Structure for Large-Scale Discrete Event Simulation”. *ACM Transactions on Modeling and Computer Simulation* 15(3):175–204.

Tay, S. C., Y. M. Teo, and R. Ayani. 1998. “Performance Analysis of Time Warp Simulation with Cascading Rollbacks”. In *Proceedings of the Twelfth Workshop on Parallel and Distributed Simulation*, PADS ’98, 30–37.

Tomlinson, A. I., and V. K. Garg. 1993. “An Algorithm for Minimally Latent Global Virtual Time”. In *Proc of the 7th Workshop on Parallel and Distributed Simulation (PADS)*, 35–42. San Diego, CA: Society for Computer Simulation.

Weber, D. 2016. “Time Warp Simulation on Multi-core Processors and Clusters”. Master’s thesis, University of Cincinnati, Cincinnati, OH.

Williams, B., D. Ponomarev, N. Abu-Ghazaleh, and P. A. Wilsey. 2017. “Performance Characterization of Parallel Discrete Event Simulation on Knights Landing Processor”. In *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS ’17, 121–132. New York, NY, USA: Association of Computing Machinery.

Young, C. H., N. B. Abu-Ghazaleh, and P. A. Wilsey. 1998. “OFC: A Distributed Fossil-Collection Algorithm for Time-Warp”. In *12th International Symposium on Distributed Computing, DISC’98 (formerly WDAG)*.

Young, C. H., and P. A. Wilsey. 1996. “A Distributed Method to Bound Rollback Lengths for Fossil Collection in Time Warp Simulators”. *Information Processing Letters* 59(4):191–196.

AUTHOR BIOGRAPHIES

PHILIP A. WILSEY is a Professor in the Department of Electrical Engineering and Computer Science at the University of Cincinnati. His research interests are in *High Performance Computing* with applications to *Parallel Discrete-Event Simulation* and *Data Science*. He also has interests in Privacy Preserving Data Mining, Embedded Systems, and Point-of-Care medical devices. In the field of *Parallel and Distributed Simulation* (PDES), he is currently studying the design of solutions for the pending event set problem for high performance parallel and distributed simulation on single node and clusters of multi-core and many-core processors. Finally, he is pursuing

studies to extract profile data from discrete event simulation models to obtain quantitative data on their run time characteristics. The principle objective of these studies is to support quantitative based optimization of PDES simulation kernels. His email address is wilseypa@gmail.com, his web pages are at: http://secs.ceas.uc.edu/~paw, and his research software is released from git repositories at: http://github.com/wilseypa.