# Parallel Discrete Event Simulation for Multi-Core Systems: Analysis and Optimization

Jingjing Wang, *Student Member, IEEE*, Deepak Jagtap, *Student Member, IEEE*, Nael Abu-Ghazaleh, *Member, IEEE*, and Dmitry Ponomarev, *Member, IEEE*

**Abstract**—Parallel Discrete Event Simulation (PDES) can substantially improve the performance and capacity of simulation, allowing the study of larger, more detailed models, in less time. PDES is a fine-grained parallel application whose performance and scalability is limited by communication latencies. Traditionally, PDES simulation kernels use message passing; often these simulators are written for distributed environments, and shared memory is used to optimize message passing among processes on the same machine. In this paper, we develop, characterize and optimize a thread-based version of a PDES simulator on three representative multi-core platforms. The multi-threaded implementation eliminates multiple message copying and significantly minimizes synchronization delays. We study the performance of the simulator on three hardware platforms: an Intel Core i7 machine, and a 48-core AMD Opteron Magny-Cours system, and a 64-core Tilera TilePro64. We discover that the three platforms encounter substantially different bottlenecks because of their different architectures. We identify these bottlenecks and propose mechanisms to overcome them. Our results show that multi-threaded implementation improves the performance over an MPI-based version by up to a factor of 3 on the Core i7, 1.4 on the AMD Magny-Cours, and 2.8 on the Tilera Tile64.

**Index Terms**—PDES, multi-threaded, optimistic simulation, multi-core systems, optimization

## 1 Introduction

Discrete Event Simulation (DES) is a type of simulation used to study systems where the changes of state are discrete. It is widely used in system evaluation and analysis in many areas including computer and telecommunication systems, biological networks, military war gaming, online games, and operational management. The increasing demands of simulation models challenge the capabilities of sequential simulators. Parallel Discrete Event Simulation (PDES) exploits the natural parallelism present in simulation models to substantially improve the performance and capacity of DES simulators.

The high communication overhead and latency limit the performance of PDES, especially when running on a cluster environment [1]. Several approaches have been proposed to reduce communication overheads [2], [3], [4]. However, PDES remains highly constrained by the high cost of communication.

The emergence of multi-core architectures and their expected evolution into many-cores present an opportunity for PDES and similar fine-grained applications. The low communication latency and tight memory integration among the cores on a multi-core chip substantially reduce the communication cost improving the performance and scalability of communication bound applications. However, most existing PDES simulation kernels such as WarpIV [5], GTW [6], and ROSS [7], have been created for cluster environments and have not been optimized to work in multi-core settings.

In this paper, we report on our experiences in optimizing a PDES simulation kernel, the Rensselaer's Optimistic Simulation System (ROSS) [7], for multi-core platforms. Specifically, we re-implement the process-based simulator as a multi-threaded model, which we call ROSS-MT, to take advantage of the tight integration among cores on the same chip. This allows us to substantially reduce communication latency by passing events directly from one thread to another. We evaluate the performance of the multi-threaded ROSS on two primary multi-core platforms: an Intel Core i7, and an AMD Magny-Cours 48-core machine. In addition, we use a 64-tile Tilera platform as an alternative architecture.

We discover a number of performance bottlenecks, especially on the 48-core machine, and propose optimizations to reduce their effect. First, we show that the MPI barrier synchronization does not scale due to lock contention. Instead, the optimized `pthread_barrier` implementation should be used. Second, we show that the standard implementation of memory allocation is not aware of the non-uniform memory latency present on some multi-core architectures. We propose and evaluate several policies that are aware of these effects. Finally, we show that there is substantial contention for the incoming event queues, and present a distributed implementation that significantly reduces this contention. Together, with these optimizations, the multi-threaded ROSS outperforms the baseline distribution of ROSS by up to a factor of 3 on the Intel Core i7, and a factor of 1.4 on the 48-core AMD Magny-Cours system, and a factor of 2.8 on the Tilera platform.

- *The authors are with the Department of Computer Science, Binghamton University, Binghamton, NY 13902 USA. E-mail: {jwang36, djagtap1, nael, dima}@cs.binghamton.edu.*

*Manuscript received 19 Dec. 2012; revised 27 June 2013; accepted 1 July 2013. Date of publication 4 Aug. 2013; date of current version 16 May 2014. Recommended for acceptance by M.E. Acacio.*
*For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.*
*Digital Object Identifier no. 10.1109/TPDS.2013.193*

This paper significantly expands a previously published article in the International Parallel and Distributed Processing Symposium (IPDPS 2012) [8]. First, we explore the design space of the locking mechanism around the critical event queue, implementing several new organizations that differ in the locking primitives and lock distribution. In addition, we identify a significant memory leak problem that occurs due to the interaction of the NUMA optimization present in [8] with the operating system NUMA allocation policy. We develop and explore several solutions to this problem. We also present alternative NUMA optimization based on allocating space on intermediate cores. Moreover, we analyze the performance of the simulator and the proposed optimizations on a real model of a Personal Communication System. Finally, we evaluate the performance of ROSS-MT on a 64-tile Tilera platform.

The remainder of this paper is organized as follows. Section 2 provides background information on PDES in general, and the multi-core platforms used in our experiments. Section 3 provides details of the solutions proposed in this paper. Performance bottlenecks and our solutions to them are described in Section 4. Section 5 overviews the experimental setup, while Section 6 presents the performance evaluation of the ROSS-MT simulator and the proposed optimizations. In Section 7 we review the related work. Finally, Section 8 offers our concluding remarks. Associated with the paper is a supplementary document which is available in the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2012.193 that includes more background information, details regarding the experimental set up, and expanded experiments and analysis.

## 2 Background

In this section, we first briefly describe parallel discrete event simulation and the ROSS simulator [7], and overview two multi-core platforms used in our experiments.

### 2.1 Parallel Discrete Event Simulation

In parallel discrete-event simulation (PDES), a model is partitioned across a group of Processing Elements (PEs) that communicate by exchanging time-stamped event messages [9], [10]. Each event carries a time-stamp which determines the simulation time at which it is scheduled to occur. Each PE processes its events in time-stamp order to ensure causality. However, without synchronization, it is possible for an event generated from one PE to arrive at another PE after its scheduled processing time has passed; such a *straggler* event indicates a causality error. Two primary synchronization algorithms are widely used in PDES to enforce correct causality: conservative and optimistic synchronization. Conservative simulation requires PEs to coordinate to guarantee that no causality errors can occur. In contrast, in optimistic simulation no explicit synchronization is enforced during simulation. However, if a causality error is detected, the simulation is rolled back to a time before the straggler event, and messages are sent to cancel any erroneously sent events after that time. More details about conservative and optimistic simulation are in Section 1 of the supplementary material available online.

In this paper, we use the ROSS [7] PDES simulator. In the optimistic mode, ROSS leverages efficient reverse computation where, instead of check-pointing, reverse computation code is associated with every event to reverse its effect to restore the state during rollbacks.

### 2.2 Multi-Core Architectures

We study three multi-core platforms, with significantly different architecture and memory organizations. The first is a 4-core Intel Core i7 processor. Each core supports two Simultaneous Multi-threaded (SMT) thread contexts. The cores have private 32 KB L1 and 256 KB L2 caches but share an 8 MB L3 cache. The second platform we use is a 48-core AMD Magny-Cours machine. There are four CPU chips on the memory bus, each holding 12 cores. The chips are connected using AMD proprietary Hyper-transport 3.0 links. On each chip, the cores are located on two separate dies, with each die holding 6 cores. Each core has a private 64 KB L1 and 512 KB L2 caches, and shares 6 MB L3 cache with other cores on the same die. A specialized interconnect is used to connect the caches across dies. The cores have non-uniform memory access (NUMA) to different regions in memory and experience non-uniform latencies on cache hits to the L3 cache depending on whether the cache line is in the L3 cache of the same die or a remote die. The third platform is the Tilera TilePro64, an many-core architecture with 64 identical tiled cores. Tilera features low latency and high bandwidth communication fabric interconnecting the cores. More details about the Tilera machine are presented in Section 2 of the supplementary document available online.

## 3 Multi-Threaded ROSS: Design Overview

In ROSS, communication occurs for three primary purposes: 1) Exchange of event messages; 2) exchange of anti-messages, cancelling earlier messages sent erroneously; and 3) for Global Virtual Time (GVT) computation which is used to commit events, and garbage collect unneeded event checkpoint information. It is essential for communication latency to be low for all three of those functions. Otherwise, rollbacks occur more frequently, are more expensive and more difficult to contain, and GVT computation overhead becomes high, delaying event commitment and increasing the simulator memory usage.

### 3.1 Communication Mechanism in the MPI-Based ROSS

MPI is used for communication in the baseline ROSS simulator (ROSS-MPI). Fig. 1 shows event message communication mechanism in ROSS-MPI. Each PE maintains a queue of outgoing remote events. When a PE sends a message to another remote PE, an event message is first queued into the *Output Queue* (*Outq*). Events are later dequeued from the Outq and sent to the appropriate destination process asynchronously based on receiver buffer availability. *Posted sends* and *Posted receives* buffers are used for asynchronous message passing. Once the event message is successfully received at the destination process, it is enqueued into the event queue at the receiver side. The event queue is a priority queue maintained by the

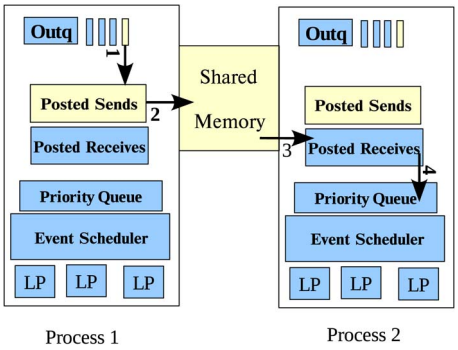



Fig. 1. MPI-based ROSS communication mechanism.

scheduler to keep the events in time-order. The scheduler dequeues events from the event queue for processing.

### 3.2 The Design of ROSS-MT

In ROSS-MT, we use threads instead of processes as seen in Fig. 2. Because the threads share the same address space, we use an input queue for each thread containing all remote events from other threads (PEs). No buffering is needed and thus the *Posted sends* and *Posted receives* buffers are eliminated. During communication the sender keeps each message, so that in case of rollbacks, cancellation messages can be generated. A copy of each message is then created, and a pointer to this message copy is inserted in the input event queue of the destination thread. The receiver thread dequeues events from the input queue and inserts them into the primary event queue for processing.

### 3.3 Performance of Communication Primitives

Since PDES is a communication-bound application, the performance is substantially impacted by the performance of the communication primitives. Thus, to set the context for later results, we study the performance of the communication primitives under both message passing and multi-threaded communication for both the Intel Core i7 and the AMD 48-core platforms.

First, we perform ping-pong message exchange latency tests for both message passing and multi-threaded communication on both platforms. More precisely, the experiment consists of two MPI processes for message passing communication, or two threads for multi-threaded communication, with each pinned to a different core. For the Magny-Cours platform, we selected two cores on the same

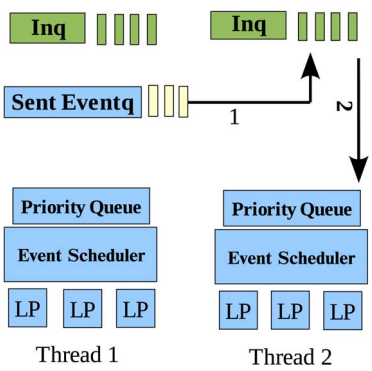



Fig. 2. Multi-threaded ROSS communication mechanism.

TABLE 1
Number of Exchanged Messages per Second on Intel Core i7 Machine

| Communication | MPI | Multi-threaded |
|---|---|---|
| inter-core | 3703704 | 12500000 |

die (intra-die), two cores on different dies but on the same chip (inter-die), and two cores on different chips (inter-chip) respectively. We evaluate the performance of communication primitives by measuring the message sending rate (count of exchanged messages per second). In addition, we fixed the message size at 128 bytes, which is the event message size used by ROSS.

Tables 1 and 2 show the performance of corresponding experiments on the Intel Core i7 machine, and AMD 48-core platform respectively. We discover that the multi-threaded implementation outperforms the MPI-based version: the message rate of multi-threaded version exceeds that of MPI-based version by a factor of 3.4 on the Core i7, and 1.8 on the Magny-Cours machine. This performance improvement occurs because two memory copying operations are performed through shared memory for each message in the MPI-based communication, incurring significant overhead. On the other hand, these operations are eliminated in the multi-threaded implementation.

## 4 Performance Bottlenecks and Optimizations

In the next set of experiments, we use the *Phold* benchmark [11] to compare the performance of the baseline multi-threaded implementation to the MPI implementation, as shown in Fig. 3. *Phold* is the most widely used benchmark for performance evaluation of PDES systems [12], [13]. The model starts with a number of objects that have events. Event execution sends a message to another object (picked uniformly among all the objects in the simulation). The message causes this object in turn to later send another event message to a third object. Thus, the number of events in the simulation remains constant. *Phold* can be configured to control the percentage of event messages that are remote. More details about *Phold* and the experimental setup are presented in Section 5. While the Core i7 results show substantial performance improvements with multi-threading, surprisingly, the Magny-Cours results show considerable slowdown. Thus, the section identifies the performance bottlenecks in the multi-threaded implementation, and provides optimizations to address them.

### 4.1 Efficient Barrier Synchronization

Barrier synchronization and all-reduce communication primitives are key components of the GVT computation.

TABLE 2
Number of Exchanged Messages per Second on AMD 48-Core Machine

| Communication | MPI | Multi-threaded |
|---|---|---|
| intra-die | 980392 | 1785714 |
| inter-die | 833333 | 1492537 |
| inter-chip | 806452 | 1470588 |

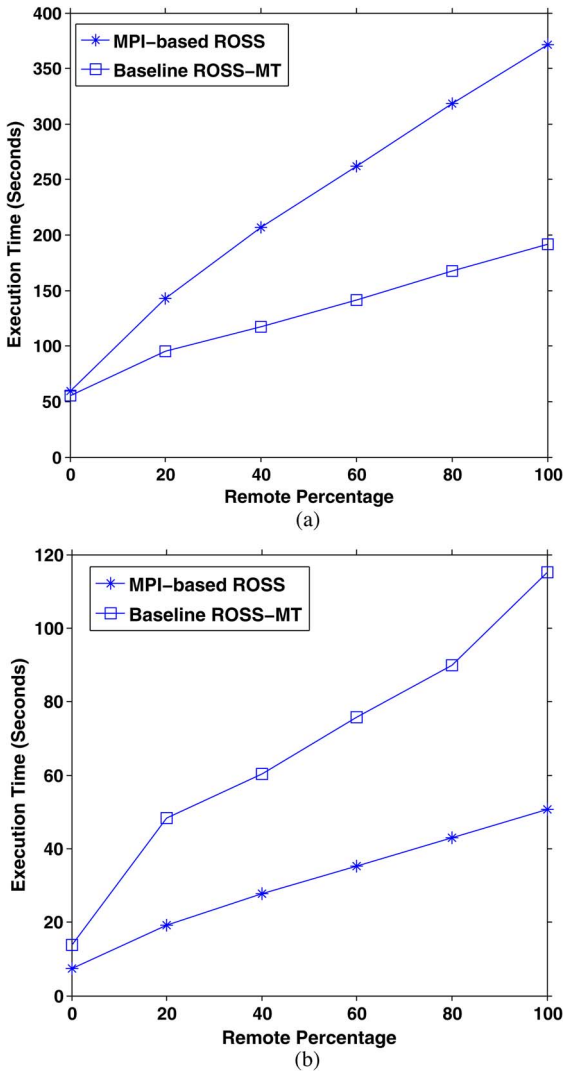



Fig. 3. Performance of baseline ROSS-MT vs. ROSS using MPI. (a) Core i7. (b) Magny-Cours.

ROSS-MT uses its own library for barrier synchronization and all-reduce operation. In the baseline version of the multi-threaded implementation, we used condition variables and pthread_mutex for implementing these operations. We found that the use of condition variables can result in high overheads. We addressed this problem by using the `pthread_barrier` implementation instead of condition variables.

### 4.2 NUMA-Aware Free Memory Management

ROSS implements application-level free memory management to avoid unnecessary use of the memory allocation library. The ROSS implementation places the memory of an event message after it is consumed in a free memory pool. This memory is then used for future message events. Suppose that a message is generated from PE 1 to PE 2. The message is allocated by PE 1 from its closest memory region (the Operating System NUMA option enforces that). Once the message is consumed by PE 2, it is returned to the memory pool for PE 2. In the future, if PE 2 needs to send an event to another PE, say PE 3, it picks the memory region that was allocated by PE 1, which is remote for both PE 2 and PE 3, leading to high access latencies.

To address this problem, we propose splitting the free memory pool to keep track of the allocation source. When PE 2 needs memory space for an event, it uses the free memory pool for the destination PE to ensure NUMA friendly behavior. If there is no available memory for the destination PE, the memory region is picked from the free queue of the sender PE itself. As a result, every event message is local to either the source or the destination. In addition, we implemented a Last In First Out (LIFO) approach to message allocation to improve cache reuse. We call this allocator the Base NUMA allocator (BNA).

While BNA is effective, it suffers from the following problem. If the communication activity is not balanced between any given pair of PEs, a subtle memory leak occurs. Consider a chain topology with 3 PEs such that PE 1 communicates to PE 2, PE 2 communicates to PE 3, and PE 3 communicates back to PE 1. PE 1 receives messages from PE 3, and queues these messages in the free memory queue for PE 3. However, it never communicate to PE 3, so these messages just accumulate in this queue. PE 1 never receives messages from PE 2, so it has no free message in the memory pool for PE 2 to use, and is forced to allocate memory from its own free-memory pool. Eventually, this memory-pool is consumed, and no memory is available to communicate to PE 2. The same pattern repeats at PE 2 and PE 3. The simulation eventually enters livelock as it computes GVT and reclaims checkpoint memory, discovering that none is available. Thus, the NUMA optimization combined with the OS allocation policy leads to catastrophic failure of the simulation.

To address this problem, we monitor the size of each memory pool during simulation. If the memory pool grows beyond a threshold, we return this memory to the originating PE's free-memory pool (free queue). This operation requires simple pointer manipulation to connect the free-memory linked-list to the originating PE's free-memory list. We call this BNA policy *Free Queue* (BNA-FQ). If the value of the threshold used for memory return is too small, we return memory frequently, incurring overheads. On the other hand, if the threshold is large, the memory leak may still occur. Empirically, we find that there is a relatively wide range of this threshold that performs well on both platforms. In our experiments, we used a threshold of 50, 000 events in the memory pool.

BNA-FQ has the following disadvantage: each PE must acquire a lock when it accesses its own free memory pool, introducing significant overhead. To reduce the lock overhead, we use an intermediate staging buffer where the events are moved to first. When a local PE memory pool is exhausted, the PE picks up the events placed for it in this buffer. Thus, contention (and locking) occurs only when accessing the staging buffer, which is an infrequently occurring operation. At the same time, the frequently accessed local free-memory pool remains private, and requires no locking. We call this BNA allocator *Staging Buffer* (BNA-SB).

Finally, we implemented another NUMA-aware policy that allocates event memory from a memory bank between the sender and the receiver to reduce and balance the NUMA overhead; we call this allocator Delay Sensitive NUMA (DSN). We implemented DSN on the 48-core

platform which has 8 memory banks. Each PE has a preference list for each destination NUMA node. The list keeps the selection preference for the memory bank to allocate events. The list is created based on the NUMA node distance matrix on the 48-core machine. The first memory bank in the list has the minimum total distance between itself and, both sender and receiver nodes. The other banks in the list are sorted accordingly. The lookup of the memory bank starts from the front of the list. If the selected memory bank is completely in use, the next memory bank in the list is selected. In order to prevent memory leaks, the memory bank for the sender node is selected only if the memory banks for other nodes are empty.

### 4.3 Distributed Locking for the Input Queue

By allowing the sender threads to directly access the input queue of the receiving threads, we eliminate the need for a buffer copy to an intermediate message queue. However, each input queue may now be accessed by any of the sender threads, as well as the receiving thread (i.e., all threads in the simulation). This gives rise to high contention on the lock to access the input queue. To reduce this contention, we split the input queue into private queues, one for each possible sender. The contention for the queue is reduced from all threads, to only two threads, the sender and the receiver. To reduce the locking overhead, we used the pthread spin lock.

### 4.4 Lock Implementation Tradeoffs

To reduce the locking overhead, we attempted to use reader-writer locks in the fully distributed case. The sender threads are readers (they can all access the distributed queue at the same time since none of them accesses the same queue) and the receiver thread is a writer (it gets exclusive access to the queues). In this way, only one lock needs to be acquired by the receiver. In this design, there is no contention between the different sender threads since each goes to a different queue. However, there is a contention between the sender threads and the receiver thread since they may be accessing the same queue at the same time. To combine the advantages of both pure distributed locking and reader-writer locking, we also developed a hybrid locking approach which requires a few locks, and introduces small lock contention. In this design, the receiver has 8 reader-writer locks, each competed by 6 sender threads on the 48-core platform.

## 5 Experimental Setup and Benchmark

We use the *Phold* benchmark [11], for most of the experiments. Before the execution of the simulation, the free-memory pool is set to be of size 150,000 events at each PE; this pool is reused during the simulation. In addition to remote percentage, the number of initial events per object (we selected 1 for this parameter) can also impact the performance as it increases the number of events in the simulation.

Some of our experiments also use a Personal Communication System (PCS) model [14] in order to demonstrate that the trends hold when considering real simulation models. The PCS model simulates realistic hand-off patterns in PCS networks. In this model, an event simulates a mobile phone call being generated at a cell phone tower, and later sent to another tower. A new phone call may be generated at the end of the previous one. At the end of the simulation, the model collects performance statistics such as how often calls were blocked because of limited tower capacity.

We used both Intel Core i7 and 48-core AMD Magny-Cours as our primary platforms, to evaluate performance of multi-threaded ROSS against MPI based ROSS (ROSS-MPI). In addition, some experiments were performed on a 64-tile Tilera TilePro64 platform (see Section 4 of the supplementary document available online). More details about the configurations on each multi-core are in Section 2 of the supplementary document available online. In the next section, we evaluate the *Phold* performance on these three different architectures.

## 6 Performance Evaluation

In this section, we present an experimental evaluation of ROSS-MT and the different optimizations we proposed. Before we present these results, we first evaluate the performance of ROSS-MT with different lock strategies and different NUMA memory management policies, to identify the most efficient implementations. Once we identify these implementations, we use them in the remainder of the experiments. For all results, each point represents an average of 10 separate runs, which we verified was sufficient to bound the 95 percent confidence interval to be within 2 percent of the average.

### 6.1 Lock Overhead of ROSS-MT

Fig. 4a shows the execution time of ROSS-MT when using the distributed locking on the Magny-Cours platform. Fig. 4b shows the corresponding simulation efficiency. Efficiency is calculated as the ratio of committed events to the total events executed. With a single queue, high lock contention occurs, leading to starvation and low efficiency of the simulation. We observed that a thread fails to acquire a lock for a long period of time, causing substantial delays until it manages to send its event, causing a long rollback once this late event is received. As we increase the number of queues, the lock contention is reduced and the efficiency increases. Increasing the queues further eventually results in slightly lower performance, as no significant contention is encountered, while the receiver is burdened with having to acquire a higher number of locks to check all the different queues.

Fig. 5 shows the performance of ROSS-MT with different locking strategies under different remote communication percentages. It is clear that ROSS-MT with spin-lock performs better than both the mutex and the reader-writer lock. As the remote percentage increases, the version with reader-writer lock performs worse than the other two variations, because the writers experience starvation.

### 6.2 Evaluation of NUMA Policies

The default memory policy on the 48-core NUMA machine is first-touch, where a page of data is allocated in the

memory of the core that first accesses it [15]. This policy may cause a page to be returned to the free pool used by a different node than the node it is pinned to. A better solution for memory allocation is to ensure that the pages are allocated on a specific NUMA node by using the NUMA library APIs. Fig. 6 shows the performance of ROSS-MT with different memory allocation policies. The specific NUMA node policy achieves up to 10 percent performance improvement compared to first-touch policy. We use this NUMA-aware memory allocation policy in our later experiments.

Fig. 7 shows the performance of 48-way simulation under different NUMA policies on the Magny-Cours machine. In this experiment, we compare four different implementations of ROSS-MT: the version without NUMA optimization, BNA-FQ, BNA-SB, and DSN. Recall that both BNA-FQ and BNA-SB employ the same BNA allocation described in Section 4, but use different approaches to return memory in the case of asymmetric communication (solving the memory leak problem we identified in Section 4). In particular, when a PE's memory pool grows beyond a threshold, BNA-FQ allows the PE to return the memory to the originating PE's free memory pool. On the other hand, BNA-SB returns the memory of freed events to an intermediate staging buffer. A PE later picks up the events placed for it in the buffer when its memory pool is exhausted. Finally, DSN uses a different NUMA allocation policy where each PE maintains a preference list for each destination NUMA node.

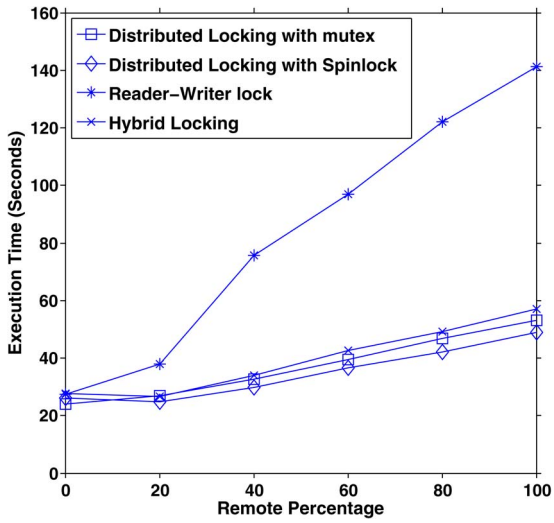
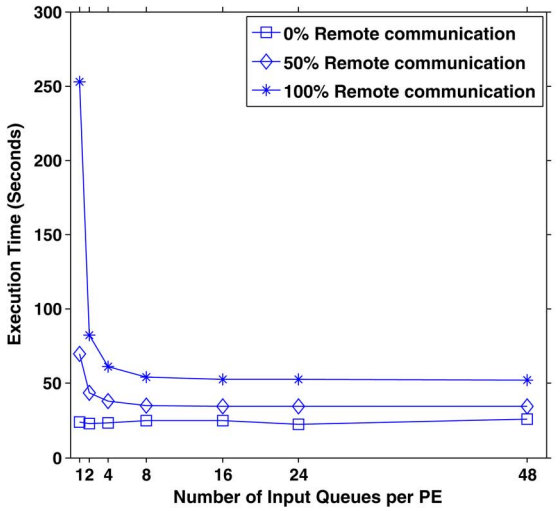
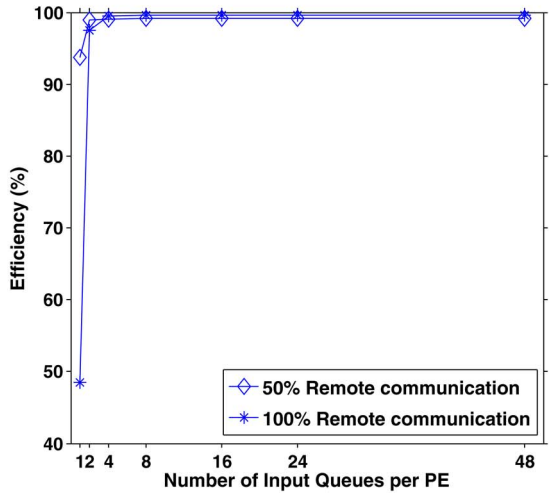



Fig. 5. Impact of locking organization: Magny-Cours.




(a)




(b)

Fig. 4. Impact of distributed locking: Magny-Cours. (a) Execution time. (b) Efficiency.

Fig. 7 shows that BNA-FQ performs poorly in comparison to both BNA-SB and DSN, and even worse than the implementation with no NUMA optimization. After analysis, we discovered that BNA-FQ suffers high overheads because of lock contention during the access to the free-memory pools. This contention is dramatically reduced by the staging buffer used in BNA-SB. BNA-SB performs marginally better than DSN (which allocates memory halfway between the sender and receiver); the NUMA latency between caches in our 48-core platform is not that different, providing little opportunity for DSN to improve performance. We believe DSN may become more beneficial on other architectures with more heterogeneous latencies

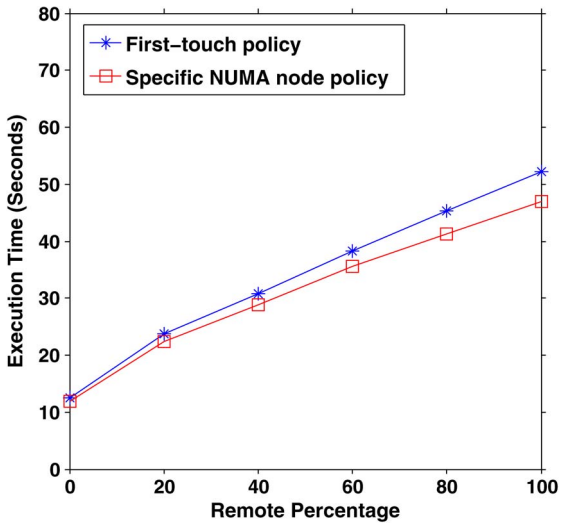



Fig. 6. Impact of memory allocation policies: Magny-Cours.

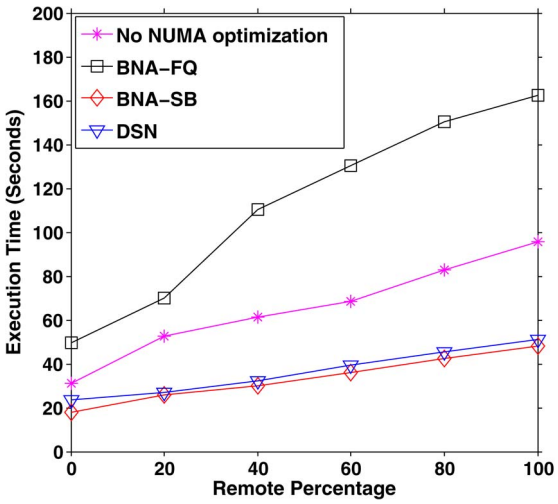



Fig. 7. Impact of NUMA policies: Magny-Cours.

at the cache level. We use BNA-SB for ROSS-MT in the remaining experiments.

### 6.3 Impact of the Optimizations

We implemented the optimizations discussed in the previous section (Efficient barrier synchronization, NUMA aware memory pool management, distributed input queue, and lock optimization). Figs. 8a, 8b, 8c show the performance improvement obtained from each of the optimizations in isolation and combined on the Core i7. We consider a 2-, 4-, and 8-way simulation, while keeping the number of objects per thread the same. A number of interesting observations can be made. For two nodes, as the number of remote messages increases, the optimizations harm performance. Distributing the locking on the queue is not beneficial since the degree of contention is not significant, but the overhead is increased. Moreover, NUMA issues are not important either since each memory element is local to either of the two threads. Finally, lock contention issues are minor in the barrier implementation. It is interesting to see some gain initially, but that is likely due to the LIFO strategy introduced as part of the NUMA optimization; other optimizations introduce overhead without benefit for a two-thread simulation. As the number of threads is increased, the optimizations start to become useful. The optimized ROSS-MT achieves up to 50 percent improvement relative to the baseline ROSS-MT.

Figs. 9a, 9b, and 9c show the impact of the optimizations for 4, 16 and 48 thread scenarios respectively on the Magny-Cours. Since the bottlenecks were most severe for this machine, the optimizations yield substantial improvement in performance (over 150 percent for 48 threads). The impact of the barrier optimization increases with the degree of parallelism, and reduces slightly with the increase in event communication (recall that the barrier optimization affects GVT computation but not event communication). We also study the impact of the optimizations on the Magny-Cours when the percentage of remote communication is fixed. Because of space limitations, we present these results in Section 3 of the supplementary material available online.

In the next experiment, we evaluate the performance of ROSS-MPI and the optimized ROSS-MT on three platforms, using the classical *Phold* model, as shown in Fig. 10. In Fig. 10a, we show results on the Core i7 machine. In particular, the experiments were executed on 8 hardware threads. It is clear that the multi-threaded implementation is substantially faster than the MPI version on this platform. Fig. 10b shows the same comparison for the Magny-Cours platform with 48 cores. ROSS-MT also outperforms the MPI version, although the gap is substantially smaller. Fig. 10c shows the performance of ROSS-MPI and ROSS-MT on the Tilera machine. The simulation model consists of 56000 objects equally distributed across 56 PEs. In addition, the simulation time was set to 1000 for Tilera. Clearly, the performance of ROSS-MT exceeds that of ROSS-MPI by a factor of up to 2.8 on the Tilera platform. More detailed evaluation of the simulator on the Tilera machine is presented in Section 4 of the supplementary material available online.

### 6.4 Other Benchmarks

In our previous experiments, we used the *Phold* benchmark to evaluate the performance of both ROSS-MPI and ROSS-MT. In this study, we consider another two benchmarks:

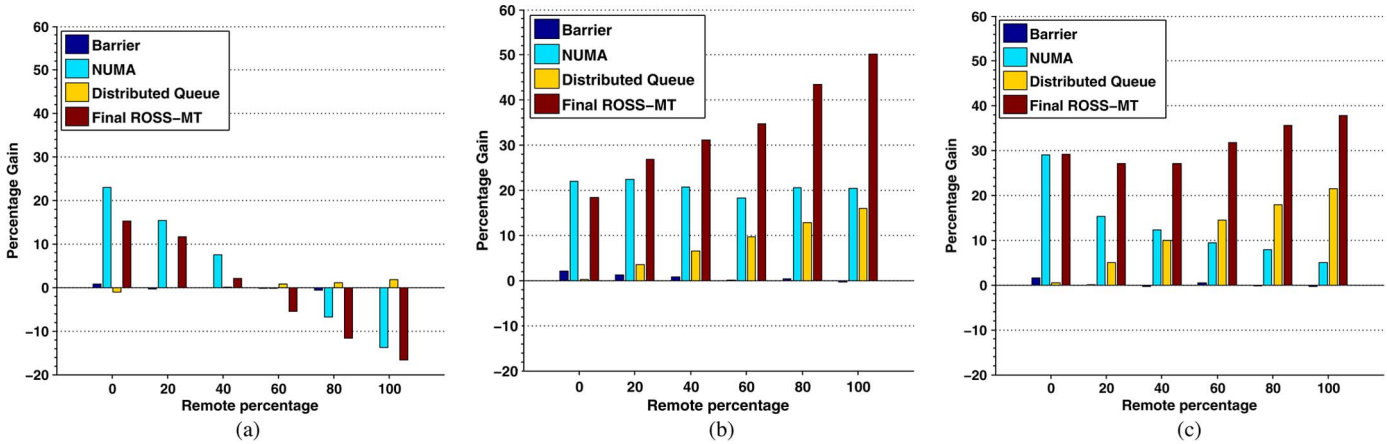



Fig. 8. ROSS-MT on the Intel core i7. (a) 2 nodes. (b) 4 nodes. (c) 8 nodes.

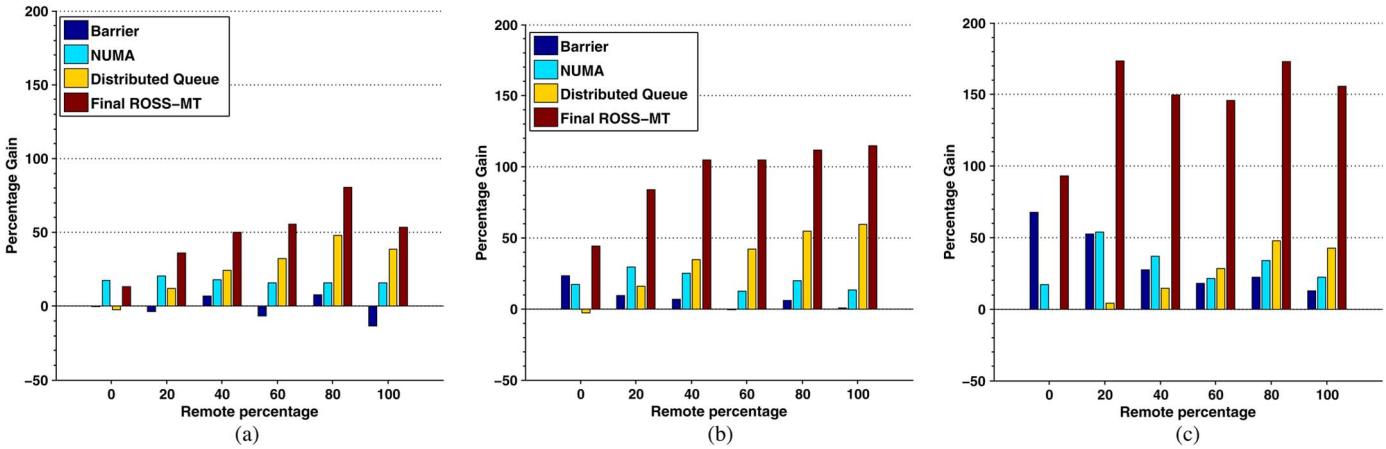



Fig. 9. Magny-Cours performance with increased parallelism. (a) 4 nodes. (b) 16 nodes. (c) 48 nodes.

Personal Communication Services (PCS) simulation model [14], and hierarchical *Phold* model [16]. Both of these benchmarks mimic the behaviors of real-world systems.

In the PCS model, the total number of cell phone towers (LPs) is fixed at 57600. Figs. 11a and 11b show the PCS model performance on the Core i7 and Magny-Cours machines respectively. The sequential simulation run-time was 1106 seconds on the Core i7 machine, and was 247 seconds on the Magny-Cours machine. We discover that the performance improvement of ROSS-MT over ROSS-MPI in the PCS model is smaller than that in the *Phold* model. This is because that PCS model has more computation and less remote communication (< 10 percent) than *Phold*.

Hierarchical *Phold* model groups objects in a hierarchical communication structure, compared to the random selection of communication targets in the classical *Phold*. The communication frequencies among groups of objects are determined by a Pareto distribution [16]. This model exhibits similar object communication graphs to those present in some real-world systems such as 3 M-ncs, an Internet-scale network simulation model [17]. In our experiment, the hierarchical *Phold* model consists of 8160 LPs distributed equally among PEs. Fig. 12 shows the performance of hierarchical *Phold* model for both ROSS-MPI and ROSS-MT on the 48-core platform. Clearly, ROSS-MT achieves a better performance than ROSS-MPI. For example, at the case of 48 nodes, the performance of ROSS-MT exceeds that of ROSS-MPI by a factor of 1.4.

## 7 Related Work

PDES is difficult to parallelize because of its fine-grained nature, and dynamic dependency patterns which vary with the model being simulated [9], making it substantially different from typical parallel applications. In this study, we focus on optimizations specific to PDES.

### 7.1 Optimizing the Communication Cost for PDES

Bahulkar *et al.* [16] proposed a static partitioning approach to reduce the communication overhead of PDES, by placing the most heavily communicating objects on the same processor before the simulation starts. Similarly, dynamic partitioning have been proposed to repartition the simulation to recover dynamic behavior changes of the simulation model for both conservative (e.g., [18]) and optimistic (e.g., [3]) synchronization protocols. Chetlur *et al.* [4] proposed the use of message aggregation, where multiple event messages are combined in a single communication message, to amortize the overheads associated with communication across multiple messages. Mattern developed a

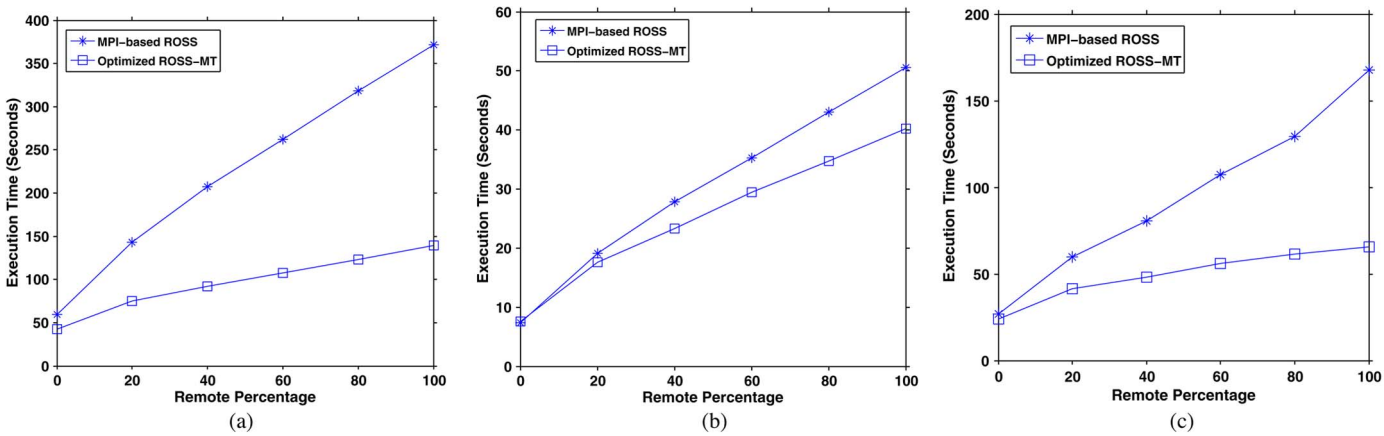



Fig. 10. Optimized ROSS-MT vs. ROSS-MPI. (a) Intel core i7. (b) AMD Magny-Cours. (c) Tilera.

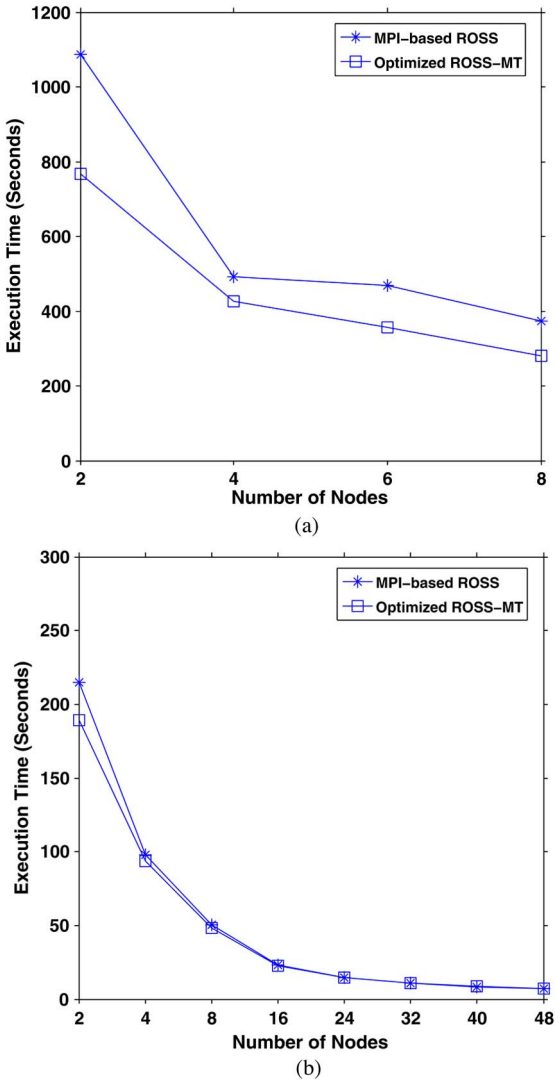



Fig. 11. Performance of PCS model. (a) Core i7. (b) Magny-Cours.

non-blocking GVT algorithm which allows event processing to proceed concurrently with GVT computation, allowing the cost of that expensive operation, which includes global communication among the PEs, to be hidden [19]. Noronha *et al.* used a programmable network card to optimize event communication and GVT computation [20].

### 7.2 MPI on Shared Memory Architectures

This work replaces MPI communication with efficient shared memory primitives. Thus, it is important to understand the advantages of this model over optimizing MPI operation for shared memory. Typically, MPI implementations require two memory copy operations to copy the message from the sender to the shared memory segment and then back to the receiver. Flow control is supported using pipes to ensure that the message is received correctly. In addition, implementing the semantics of message passing requires MPI to use multiple system calls for each send and receive. The message is piped (using a system call) through the shared memory segment to avoid a case where the message size exceeds the available space.

Some researchers have proposed different approaches to improve the performance of MPI for shared memory architectures. Graham *et al.* [21] optimized collective operations of MPI, where each process can directly access other processes data out of their shared memory buffers. Hoefler *et al.* [22] leveraged remote memory access interface of MPI-3.0 to support the direct memory access for one-sided communication. However, double copies are required for point-to-point communication. Goglin *et al.* [23] support efficient intra-node MPI communication for large messages, by using kernel-assisted direct copies between processes. However, for small messages (such as those used in PDES), they observe that the standard two-copy implementation performs better.

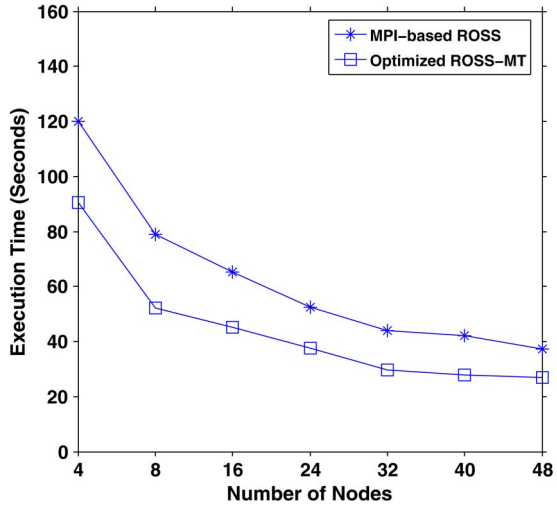



Fig. 12. Hierarchical phold model performance on the AMD Magny-Cours.

### 7.3 Multi-Threaded PDES

To improve PDES performance on multi-cores, some researchers have designed multi-threaded PDES engines. Chen *et al.* [24] proposed a multi-threaded PDES implementation that uses a global event scheduling mechanism. Although such an organization is helpful for load balancing, it sacrifices locality and introduces overheads for scheduling through a centralized queue. Vitali *et al.* [25] proposed a different multi-threaded PDES simulator that employed a load-sharing scheme. This simulator differs from ROSS-MT because multiple threads may be assigned to the same PE. An interesting direction of future work is to comparatively evaluate these different organizations of the multi-threaded simulation engine. Wang *et al.* study the performance of PDES on clusters of multi-cores, discovering that inter-machine latency dominates, but that solutions can be developed to hide its very high cost [26].

## 8 Conclusion and Future Work

This paper presented experiences in building a multi-threaded PDES simulator optimized for representative state-of-the-art multi-core machines. We used the ROSS PDES simulator, and modified it from a process-based model to a thread-based model. Although the implementation showed significant performance benefits on the Core i7 platform, it showed surprisingly poor performance on the AMD Magny-Cours.

We studied the reasons for this poor performance, and identified three bottlenecks. First, the barrier and all-reduce primitives used in GVT computation were implemented in an inefficient way using condition variables and broadcasts. We replaced this implementation with one that uses the `pthread_barrier` mechanism, which uses atomic instructions for efficiency. The second performance problem occurred due to the NUMA nature of the Magny-Cours platform. We proposed and evaluated a number of policies that are sensitive to the NUMA nature of the platform. The best solution results in guaranteed liveness, and incurs no performance overhead when it is not needed. The third bottleneck was due to the lock contention on the input queue. We resolved this issue by splitting the queues to reduce contention. We also studied the tradeoff between different lock implementations, exploring the use of mutex locks, spin-locks and reader-writer locks. The optimizations resulted in substantial improvement in performance; optimized ROSS-MT outperforms the MPI-based version by a factor of up to 3 on the Core i7 platform, and up to 1.4 on the Magny-Cours, and up to 2.8 on the Tilera.

In our future work, we plan to explore some lock-free alternatives to reduce coordination overheads. Moreover, we plan to explore some adaptive mechanisms to automatically enable and configure the different optimizations to match the architecture and model behavior.

## ACKNOWLEDGMENT

This material is based on research sponsored by Air Force Research Laboratory under agreement number FA8750-11-2-0004. The authors also gratefully acknowledge support from the National Science Foundation grants CNS-0916323 and CNS-0958501.

## REFERENCES

[1] J. Wang, D. Ponomarev, and N. Abu-Ghazaleh, ''Performance Analysis of Multithreaded PDES on a Cluster of Multicores,'' in *Proc. ACM/IEEE/SCS Int. Workshop PADS*, 2012, pp. 93-95.

[2] L. Li and C. Tropper, ''A Design-Driven Partitioning Algorithm for Distributed Verilog Simulation,'' in *Proc. 20th Int. Workshop PADS*, 2007, pp. 211-218.

[3] P. Peschlow, T. Honecker, and P. Martini, ''A Flexible Dynamic Partitioning Algorithm for Optimistic Distributed Simulation,'' in *Proc. 20th Int. Workshop PADS*, 2007, pp. 219-228.

[4] M. Chetlur, N. Abu-Ghazaleh, R. Radhakrishnan, and P.A. Wilsey, ''Optimizing Communication in Time-Warp Simulators,'' in *Proc. 12th Workshop Parallel Distrib. Simul. Soc. Comput. Simul.*, May 1998, pp. 64-71.

[5] J. Steinman, *The Warpiv Parallel Simulation Kernel Version 1.5.2*, 2008, WarpIV Technologies. [Online]. Available: http://www.warpiv.com/

[6] S. Das, R. Fujimoto, K. Panesar, D. Allison, and M. Hybinette, ''GTW: A Time Warp System for Shared Memory Multiprocessors,'' in *Proc. Winter Simul. Conf.*, J.D. Tew, S. Manivannan, D.A. Sadowski, and A.F. Seila, Eds., Dec. 1994, pp. 1332-1339.

[7] C. Carothers, D. Bauer, and S. Pearce, ''ROSS: A High-Performance, Low Memory, Modular Time Warp System,'' in *Proc. 11th Workshop PADS*, 2000, pp. 53-60.

[8] D. Jagtap, N. Abu-Ghazaleh, and D. Ponomarev, ''Optimization of Parallel Discrete Event Simulator for Multi-Core Systems,'' in *Proc. IPDPS*, 2012, pp. 520-531.

[9] R. Fujimoto, ''Parallel Discrete Event Simulation,'' *Commun. ACM*, vol. 33, no. 10, pp. 30-53, Oct. 1990.

[10] A.J. Park and R.M. Fujimoto, ''Efficient Master/Worker Parallel Discrete Event Simulation on Metacomputing Systems,'' *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 5, pp. 873-880, May 2012.

[11] R. Fujimoto, ''Performance of Time Warp under Synthetic Workloads,'' in *Proc. SCS Multiconf. Distrib. Simul.*, Jan. 1990, vol. 22, no. 1, pp. 23-28.

[12] K. Perumalla, ''Scaling Time Warp-Based Discrete Event Execution to $10^4$ Processors on a Blue Gene Supercomputer,'' in *Proc. ACM Conf. CF*, 2007, pp. 69-76.

[13] D. Bauer, C. Carothers, and A. Holder, ''Scalable Time Warp on Blue Gene Supercomputer,'' in *Proc. ACM/IEEE/SCS Workshop PADS*, 2009, pp. 35-44.

[14] C. Carothers, R. Fujimoto, and Y. Lin, ''A Case Study in Simulating PCS Networks Using Time Warp,'' in *Proc. Workshop PADS*, June 1995, pp. 87-94.

[15] C. McCurdy and J. Vetter, ''Memphis: Finding and Fixing Numa-Related Performance Problems on Multi-Core Platforms,'' in *Proc. IEEE ISPASS*, Mar. 2010, pp. 87-96.

[16] K. Bahulkar, J. Wang, N. Abu-Ghazaleh, and D. Ponomarev, ''Partitioning on Dynamic Behavior for Parallel Discrete Event Simulation,'' in *Proc. ACM/IEEE/SCS Int. Workshop PADS*, 2012, pp. 221-230.

[17] B. Hou, Y. Yao, and S. Peng, ''Empirical Study on Entity Interaction Graph of Large-Scale Parallel Simulations,'' in *Proc. ACM/IEEE/SCS Workshop PADS*, 2011, pp. 1-6.

[18] A. Boukerche and S. Das, ''Dynamic Load Balancing Strategies for Conservative Parallel Simulation,'' in *Proc. 11th Workshop PADS*, 1997, pp. 20-28.

[19] F. Mattern, ''Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation,'' *J. Parallel Distrib. Comput.*, vol. 18, no. 4, pp. 423-434, Aug. 1993.

[20] R. Noronha and N.B. Abu-Ghazaleh, ''Active NIC Optimization for Time Warp,'' in *Proc. IPDPS*, 2002, pp. 39-46.

[21] R.L. Graham and G. Shipman, ''MPI Support for Multi-Core Architectures: Optimized Shared Memory Collectives,'' in *Proc. 15th Eur. PVM/MPI Users' Group Meet. Recent Adv. Parallel Virtual Mach. Message Passing Interf.*, 2008, pp. 130-140.

[22] T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B. Barrett, R. Brightwell, W. Gropp, V. Kale, and R. Thakur, ''Leveraging MPI's One-Sided Communication Interface for Shared-Memory Programming,'' in *Proc. 19th Eur. Conf. Recent Adv. Message Passing Interf.*, 2012, pp. 132-141.

[23] B. Goglin and S. Moreaud, ''KNEM: A Generic and Scalable Kernel-Assisted Intra-Node MPI Communication Framework,'' *J. Parallel Distrib. Comput.*, vol. 73, no. 2, pp. 176-188, Feb. 2013.

[24] L. Chen, Y. Lu, Y. Yao, S. Peng, and L. Wu, ''A Well-Balanced Time Warp System on Multi-Core Environments,'' in *Proc. PADS*, 2011, pp. 1-9.

[25] R. Vitali, A. Pellegrini, and F. Quaglia, ''Towards Symmetric Multi-Threaded Optimistic Simulation Kernels,'' in *Proc. PADS*, 2012, pp. 211-220.

[26] J. Wang, K. Bahulkar, D. Ponomarev, and N. Abu-Ghazaleh, ''Can PDES Scale in Environments with Heterogeneous Delays?'' in *Proc. ACM SIGSIM Conf. Principles Adv. Discrete Simul.*, 2013, pp. 35-46.

**Jingjing Wang** is pursuing the PhD degree in the Department of Computer Science at SUNY Binghamton, Binghamton, NY, USA. His research interests are in the areas of parallel discrete event simulation on multi-core systems. He is a Student Member of the IEEE.

**Deepak Jagtap** is pursuing the PhD degree in the Department of Computer Science at SUNY Binghamton, Binghamton, NY, USA. His research interests are in the areas of parallel discrete event simulation on multi-core systems. He is a Student Member of the IEEE.

**Nael Abu-Ghazaleh** received the PhD degree from the University of Cincinnati, Cincinnati, OH, USA, in 1997. He is an Associate Professor in the Department of Computer Science at SUNY Binghamton, Binghamton, NY, USA. His research areas are wireless and sensor networks, system security, and parallel discrete event simulation. He is a member of the IEEE.

**Dmitry Ponomarev** received the PhD degree from SUNY Binghamton, Binghamton, NY, USA, in 2003. He is an Associate Professor in the Department of Computer Science at SUNY Binghamton. His research areas are computer architecture, security, and parallel discrete event simulation. He is a member of the IEEE.