2012 IEEE 26th International Parallel and Distributed Processing Symposium

Optimization of Parallel Discrete Event Simulator for Multi-core Systems

Deepak Jagtap, Nael Abu-Ghazaleh and Dmitry Ponomarev
Computer Science Department
State University of New York at Binghamton
{djagtap1,nael,dima@cs.binghamton.edu}

***Abstract*—Parallel Discrete Event Simulation (PDES) can substantially improve performance and capacity of simulation, allowing the study of larger, more detailed models, in shorter times. PDES is a fine-grained parallel application whose performance and scalability are limited by communication latencies. Traditionally, PDES simulation kernels use processes that communicate using message passing; shared memory is used to optimize message passing for processes running on the same machine. We report on our experiences in implementing a thread-based version of the ROSS simulator. The multithreaded implementation eliminates multiple message copying and significantly minimizes synchronization delays. We study the performance of the simulator on two hardware platforms: a Core i7 machine and a 48-core AMD Opteron Magny-Cours system. We identify performance bottlenecks and propose and evaluate mechanisms to overcome them. Results show that multithreaded implementation improves performance over the MPI version by up to a factor of 3 for the Core i7 machine and 1.2 on Magny-cours for 48-way simulation.**

I. Introduction

Discrete Event Simulation (DES) is a type of simulation used to study systems where the changes of state are discrete; for example, it is widely used in the simulation of computer and telecommunication systems, biological networks, and war simulation. 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, allowing the simulation of larger, more detailed models in shorter times.

A. Parallel Discrete Event Simulation

In PDES, a simulation model is partitioned across a number of processes (called *Processing Elements*, or PEs). Each PE processes events in simulation time order, sending messages to remote PEs if future events are generated to them. These messages must be processed in correct simulation time to maintain causality. This global time synchronization can be supported in two ways: (1) *conservative simulation* requires PEs to coordinate to guarantee that no causality errors can occur (i.e., simulation time does not progress beyond a point until all events that occur prior to that point are received and processed); and (2) *optimistic simulation:* no explicit synchronization is enforced between PEs. However, if an event is received late (it has a simulation time earlier than the current simulation time), the simulation is restored (*rolled-back*) to a time before the event time, possibly sending messages to cancel any erroneously sent event after that time, and restarted. Conservative simulation requires frequent communication, even when no dependencies are present. On the other hand, optimistic simulation can hide the latency of communication by allowing PEs to process speculatively; however, it remains sensitive to communication latency, and incurs the overheads associated with checkpointing and rollbacks.

PDES is difficult to parallelize effectively because of its fine-grained communication behavior and the complex underlying dependency pattern present in most models. Researchers have explored reducing the impact of message latency in a number of ways [7], [28], [31], [20]. Model partitioning [18] and dynamic object migration [25] attempt to localize important dependencies, reducing the frequency of remote communication. Throttling attempts to avoid excessive rollbacks by limiting the simulation from speculating aggressively [29]. However, PDES remains highly constrained by the high cost of communication.

B. PDES on Multi-core Architectures

The emergence of multi-core architectures and their expected evolution into many-cores presents an exciting opportunity to 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 and have significant impact on the scalability of PDES simulations. However, most existing PDES simulation kernels have been created for cluster environments and have not been optimized to work in multi-core settings.

1530-2075/12 $26.00 © 2012 IEEE
DOI 10.1109/IPDPS.2012.55

In this paper, we report on our experiences in optimizing a PDES simulation kernel, the Rensselaer's Optimistic Simulation System (ROSS) [5], for multicore platforms. Specifically, we reimplement the process based simulator as a multi-threaded simulator, 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 profile the performance of the multithreaded ROSS on two multicore platforms: an Intel core i7, and an AMD Magny-cours 48-core machine.

We discover a number of performance bottlenecks, especially on the 48-core machine, and propose optimizations to improve their performance. First, we show that the MPI barrier synchronization does not scale due to lock contention, and use the optimized `pthread_barrier` implementation instead. 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 and develop message allocation strategies 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 or 3 on the Intel core i7 and 1.2 on 48-core AMD Opteron Magny-cours.

The remainder of this paper is organized as follows. Section II provides background information on PDES in general, and the ROSS simulator in particular. Section III provide details of the solution we are proposing. Analysis of the baseline implementation helped us identify a number of bottlenecks; we discuss these and propose solutions to them in Section IV. In Section V, we first present our experimental setup and then the performance evaluation of the ROSS-MT simulator and the proposed optimizations. In section VI we review related work. Finally, Section VII will provide the conclusion of the study.

II. Background

In this section, we first briefly describe parallel discrete event simulation and the ROSS simulator [5], and overview a typical multi-core cluster organization.

A. Optimistic Parallel Discrete Event Simulation

A parallel discrete-event simulator (PDES) is organized as a collection of Processing Elements (PEs) that communicate by exchanging time-stamped event messages [12], [17]. Each PE processes its events in time stamp order (to ensure causality). PDES simulators differ in the synchronization algorithm used to ensure correct event ordering among events on different PEs. The PEs in conservative simulators exchange messages to upgrade each other of their progress and guarantee correctness. Alternatively, optimistic simulation may be used where PEs process events with no explicit synchronization occurring among them. Causality is preserved among different processes by exchanging time-stamped event messages and using rollback upon receiving a message with a time in the past. Thus, the state of the simulation must be saved to allow rollbacks when a causality breach is detected.

The progress of the simulation (the Global Virtual Time, or GVT) is computed as the minimum of the timestamps of all PEs as well as messages in transit. GVT is used to garbage collect state information, commit output events and often to adapt configuration parameters of the simulation. When a rollback occurs, the state of the simulation is restored to a valid state before the rollback time. Any messages erroneously sent to other PEs must be cancelled by sending anti-messages. Cascading rollbacks can occur when a rollback at one node causes a sequence of rollbacks at other nodes as it sends out it's anti-messages. Cascading rollbacks significantly harm performance. More frequent communication, or higher communication latencies can lead to rollbacks and cascading rollbacks and delays in computing GVT resulting in larger memory footprint, and slower overall execution and so communication frequency and latency play a major role in determining the performance of simulation [7];

In our experiments, we use the ROSS [5] simulator. ROSS is an optimistically synchronized simulator. It has the option of implementing reverse computation where, instead of storing state information, code is stored to undo events in case of rollbacks. If no reverse computation code is provided, ROSS uses state saving instead.

B. Multi-core Architectures

In our experiments we use two multicore platforms with significantly different architecture and memory organizations. The first is a 4-core Intel core i7 processor (Figure 1). Each core supports two Simultaneous Multi-threaded (SMT) thread contexts. The cores have private L1 and L2 caches but share an L3 cache. The second platform we use is a 48-core AMD Magny-cours machine [8]. As shown in Figure 2, there are four CPU chips on the memory bus, each holding 12 cores. The cores on a chip are on two separate dies, with each die holding 6 cores. The cores have

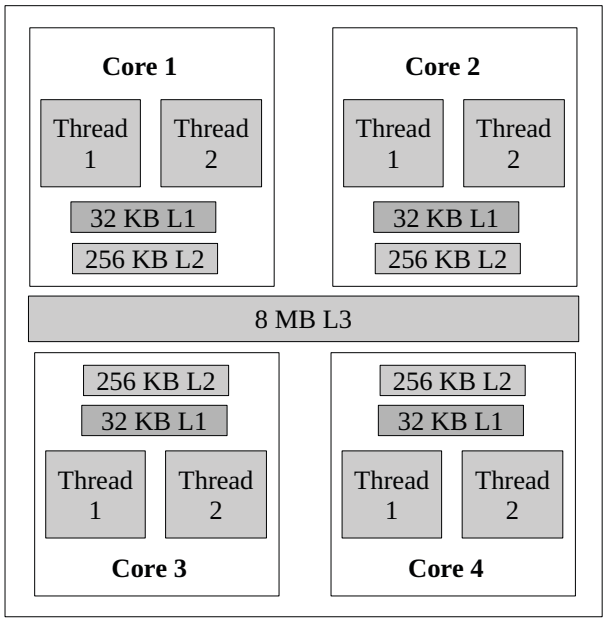


Figure 1: Architecture of the Intel Core-i7

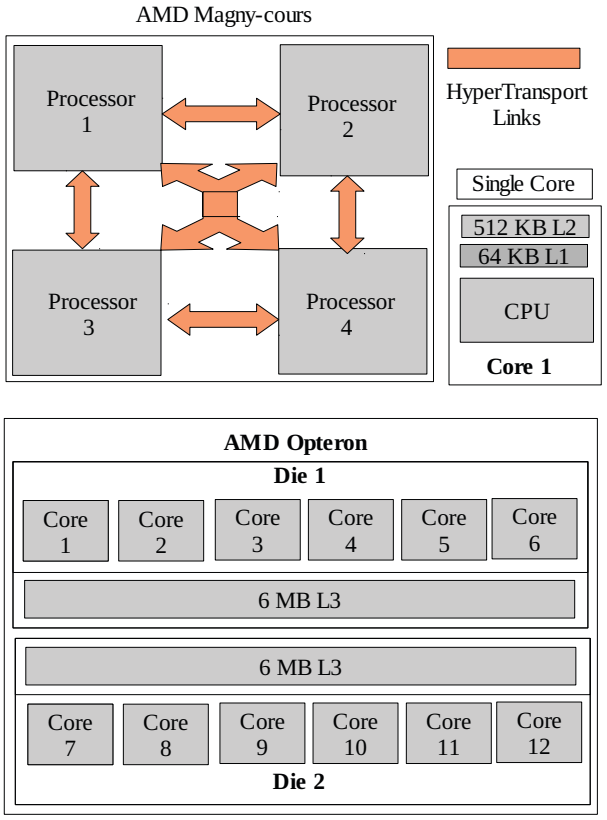


Figure 2: Architecture of the AMD Magny-cours

private L1 and L2 caches, and share the L3 level of the cache. A specialized interconnect is used to connect the caches across dies. The cores have non-uniform memory access 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. More details about the specifics of the two machines are presented in Section V.

III. Multi-Threaded ROSS: Design Overview

In this section we overview the components of the simulation kernel that require the use of communication to show the impact of the communication cost on the simulation. We show how communication support is implemented in the baseline MPI-based ROSS simulator. We then overview the baseline threaded implementation of ROSS (ROSS-MT).

A. ROSS Simulation Loop

Communication occurs in the ROSS simulator 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 computation which is used to commit events, and garbage collect unneeded state and 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 very high.

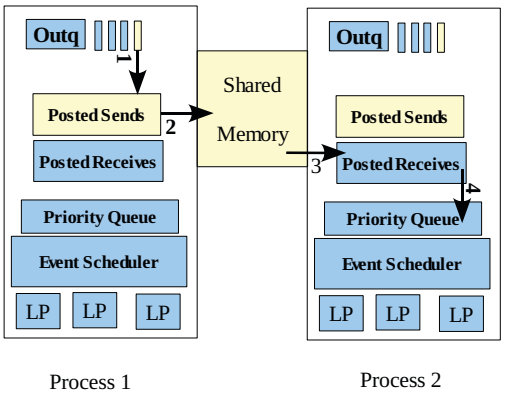


Figure 3: MPI-based Message Passing Mechanism

Event message communication in the MPI version of ROSS works as shown in Figure 3. 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 in to the *Output Queue (Outq)*. Events are later dequeued from Outq and sent to 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 queued in to priority queue at the receiver side, while the sender marks the message as successfully sent. The event queue is a priority queue maintained by the scheduler to keep the events in time-order. The scheduler dequeues events from priority queue and processes them one by one. Due to the need to compute Global Virtual Time, the state of messages in transit must be tracked. Keeping track of message state enables the appropriate steps to be taken during a rollback as well.

B. ROSS-MT

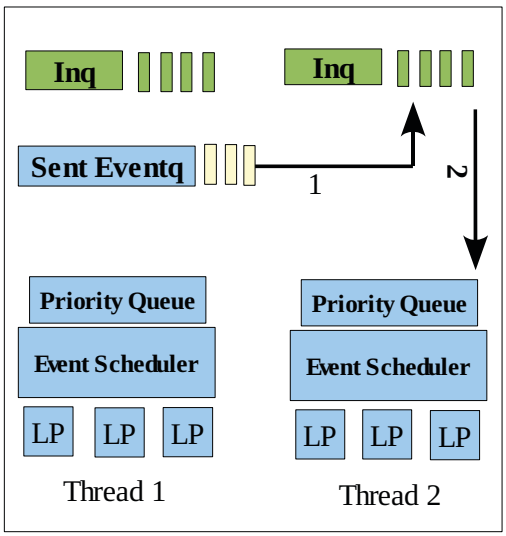


Figure 4: Mutlithreaded ROSS Message Passing Mechanism

In ROSS-MT we use threads instead of processes as seen in Figure 4. Because the threads share the same memory image, there is no need to use explicit message passing between them. Thus, instead of using a separate input and output queue for each thread, we only use an input queue for each thread containing all remote events from other threads (PEs). No buffering is needed and thus Posted send and Posted receive buffers are eliminated. A thread is associated with its own memory manager, scheduler and free event queue for fossil collection.

Communication occurs by inserting a pointer of the message copy in the input event queue of the destination thread. The sender keeps a copy of each message sent so that in case of rollbacks, cancellation messages can be sent. The receiver thread dequeues events from the input queue and inserts them into the event priority queue for processing. Thus we completely avoid synchronization delays present in MPI based ROSS implementation. We use this two stage insertion to avoid lock contention on the main event queue.

IV. Performance Bottlenecks and Optimizations

Figure 5 shows the performance of the basic multithreaded implementation in comparison to the MPI implementation for both the Intel Core i7 (Figure 5(a)) and the AMD Magny-cours (Figure 5(b)) platforms. For these results, we used the clustered Phold benchmarks, which allows us to control the percentage of event messages that are remote; Clustered Phold is described in more detail in the next section. While the core i7 results show substantial performance improvements with multithreading, surprisingly, the Magny-cours results show significant slowdown.

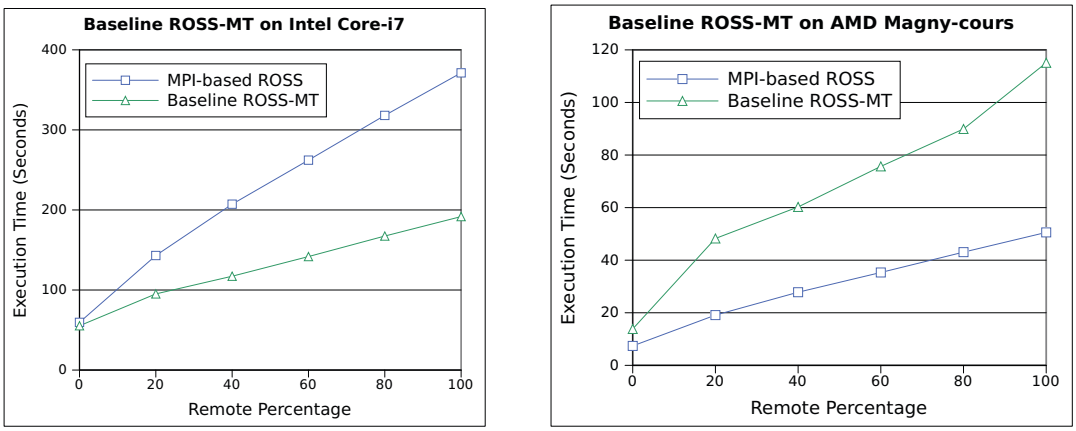
The two machines have substantially different architectures, especially with respect to the memory organization. Moreover, the Magny-cours machine has substantially higher parallelism (48-cores) than the core i7 (4 cores/8 threads). We profiled the ROSS-MT execution behavior, which allowed us to identify a number of bottlenecks. In this section, we describe three of these bottlenecks and describe optimizations to address them.

A. Efficient Barrier Synchronization

Barrier synchronization and all-reduce communication primitives are key operations for GVT computation. ROSS-MT uses its own library for barrier synchronization and all-reduce operation. In the baseline version of multithreaded implementation we used condition variables and pthread_mutex for implementing these operations. Profiling results showed very high overheads due to the use of condition variables. We optimized this library by using pthread_barrier construct instead of condition variables.

B. NUMA-aware free memory management

ROSS implements its own free memory management to avoid unecessary use of the memory allocation library. The ROSS implementation returns the memory of an event message after it is consumed to 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 OS 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



(a) Core i7 (b) Magny-Cours

Figure 5: Performance of Baseline ROSS-MT vs. ROSS using MPI

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 latency.

To address this issue, we split 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 receiving PE to ensure NUMA friendly behavior. In addition, we implemented a Last In First Out (LIFO) approach to message allocation. Thus, the most recently freed message is used from each free memory sub-pool. This policy improves cache reuse.

C. Distributed Locking for the Input Queue

By allowing sending threads to directly access the input queue of 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 sending 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.

In the next section we evaluate how well the optimizations work individually and in combination. The impact of the optimizations is different for the two platforms due to the differences in their memory organization. Additionally, the larger number of possible threads on the Magny-cours platform exacerbates lock contention issues.

We note that the current implementation exploits shared memory to optimize only the message communication aspects of the simulator. There are additional opportunities for optimization that arise due to direct access to other thread's space that we plan to implement in the future. For example, direct cancellation can be used to optimize rollbacks by removing unexecuted erroneous messages directly (instead of using anti-messages) [10]. In the future, we will explore mechanisms to share a single copy of the message instead of creating a copy for rollback purposes; we have preliminary results for this optimization.

V. Performance Evaluation of ROSS-MT

In this section, we present a performance evaluation of ROSS-MT, including the three proposed optimizations. First, we discuss the evaluation environment, and the simulation benchmark that we use.

A. Experimental Setup and Benchmark

To capture a wide range of application characteristics we developed a synthetic, controllable benchmark that is a variant of the classical *Phold* benchmark. *Phold* is the most widely used for performance evaluation of PDES systems. 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. While *Phold* has a number of drawbacks: its perfectly load balanced, with an flat dependency pattern, it is valuable for the characterizing the performance of the communication behavior of the simulator.

In particular, we modified *Phold* to allow control of the target probability for the events to allow us to control the percentage of events that are generated local to a

core, to another core on the same machine, or remotely to a core on a different machine. This benchmark is similar to that used by Perumalla [24] and Bauer et al [2] in recent scalability studies of PDES on the IBM Blue Gene.

We evaluated performance of multi-threaded ROSS against MPI based ROSS on two hardware platforms.

1) A 4-core (8 thread) Intel Core i7-860 processor with 8 GB memory and Debian 6.0.2 with Linux version 3.0.0-1. Each core has a private 32KB L1 data and instruction cache and private L2 256KB cache. 8 MB L3 cache is shared among all the cores. We use classic PHOLD as benchmark application with configuration of 1000 LPs per PE, one PE per MPI node and total 8 MPI nodes. Simulation model thus consists of 8000 LPs distributed equally on 8 PEs and a PE is pinned to one hyperthread. We selected efficient settings for GVT computation period to balance fossil collection overhead with rollbacks as per the experiment [2]. We use 16 KPs per PE and a GVT interval 256 for Intel Core-i7 and 512 for AMD Opteron with batch size 8.
2) In order to verify the scalability of multithreaded ROSS on upcoming many-core platforms we studied multithreaded ROSS behavior on an AMD Opteron 6100 (Magny-cours) 48 core machine (4 chips with 12 cores each) [8]. The chips are connected using Hyper-transport 3.0 links. Each chip consists of 2 dies and each die has 6 cores, with a 6 MB L3 cache shared among the cores on each die. Each core has private 64KB L1 and 512 KB L2 caches. The Hyper-transport links are cache coherent, thus 4P configuration provides 48 core shared memory environment. The server is running Ubuntu 10.10 with Linux version 2.6.35-30-server and has 64GB memory.

B. ROSS-MT Performance Analysis

We use the Clustered Phold model with 1000 objects per core. We fix the GVT interval at 512 (as recommended by prior evaluation studies of ROSS [2] and confirmed by our own experiments). Execution time is measured at different remote percentage for fixed batch size. We observed that batch size of 8 is optimal for both multi-threaded ROSS and MPI-based ROSS.

We implemented the three optimizations discussed in the previous section (Barrier optimization, NUMA aware memory pool management, and distributed input queue). Figure 6 shows the performance improvement obtained from each of the optimizations in isolation and combined on the core i7. We consider a 2-way, 4-way and 8-way simulation, while keeping the number of objects per thread the same.

A number of observations stand out. For two nodes, as the number of remote messages increase, the optimizations are actually counter productive. Queue distribution is not beneficial since the contention degree is not reduced, 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 are likely to introduce overhead without benefit for a two thread simulation. As the number of threads is increased, the optimizations start to become useful. The baseline barrier implementation seems efficient up to 8 nodes; the optimized implementation does not result in significant improvement in performance. The optimizations seem to interact positively as their combined effect is higher than the linear sum of their isolated effects; up to 50% improvement relative to the baseline ROSS-MT is achieved.

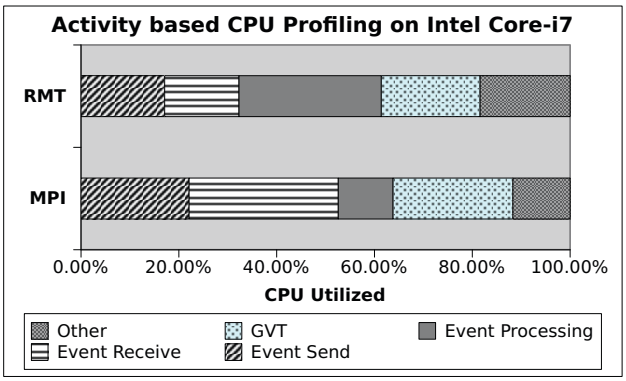


Figure 7: Execution Time Breakdown – Core i7

Figure 7 shows the breakdown of the execution time among the various stages of the simulation. We note that with ROSS-MT, significantly smaller percentage of time is spent in event send and receive (30% compared to 50% for MPI). Less time is also spent in GVT computation (18% compared to 25%). Moreover, the percentage of time spent in event processing is more than doubled.

Figure 8 shows the impact of the optimizations for the Magny-cours machine, for 4, 16 and 48 thread scenarios. Since the bottlenecks were most severe for this machine, the optimizations yield substantial improvement in performance (over 150% for 48 threads). The impact of the barrier optimization increases with

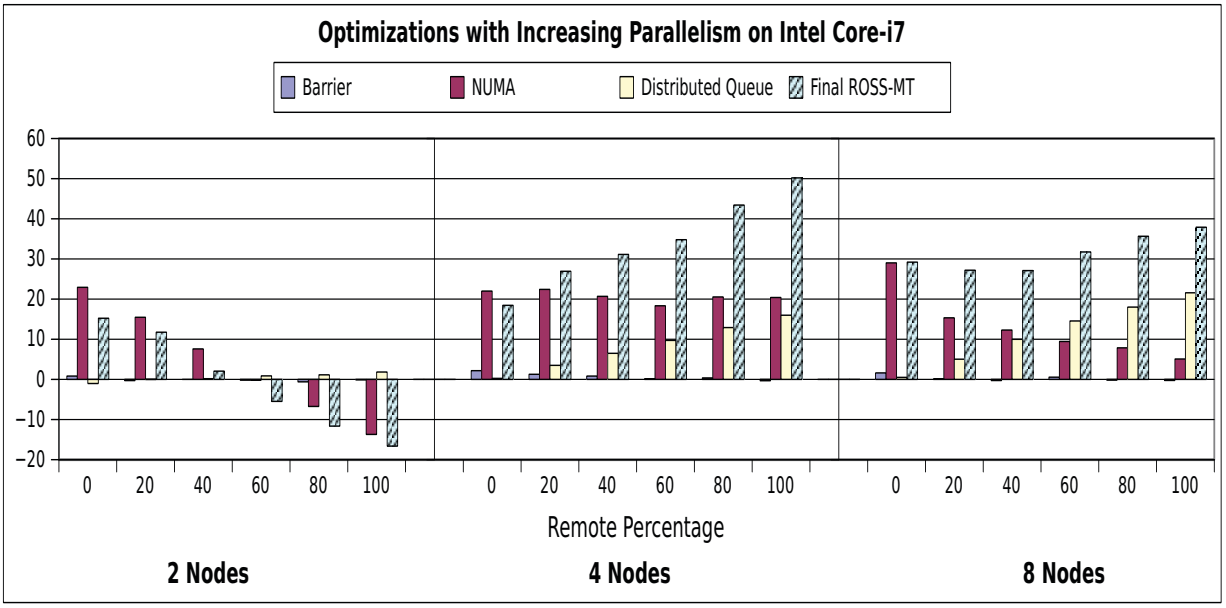


Figure 6: Optimizing ROSS-MT on Intel core-i7

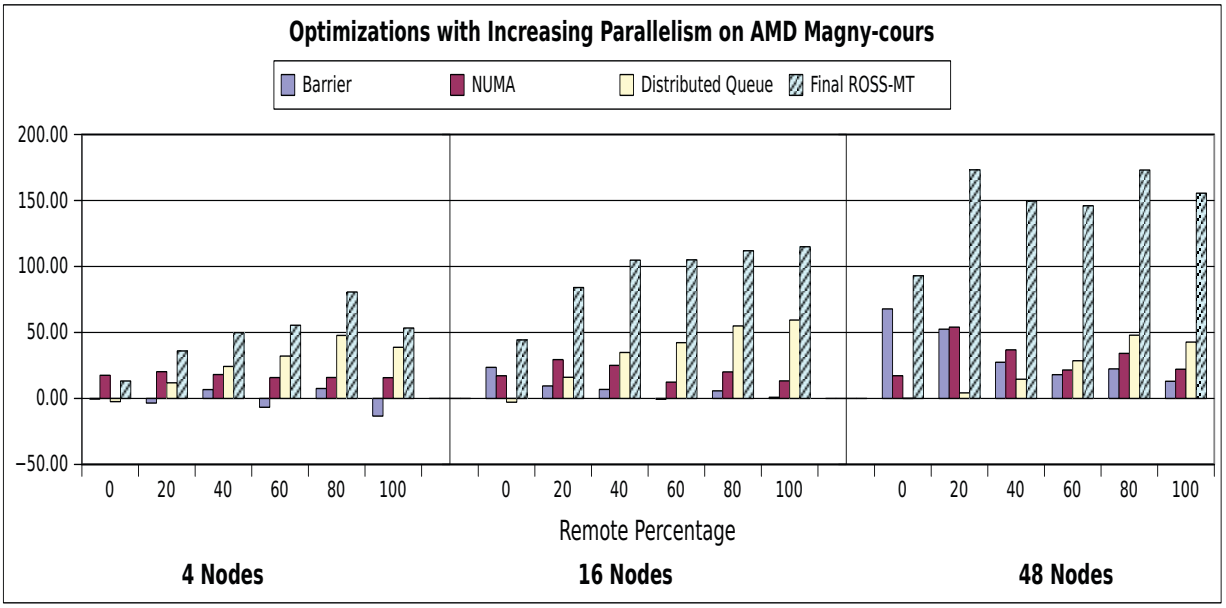


Figure 8: Magny-cours performance for different degrees of parallelism

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). Like the core i7, the queue distribution benefit increases as contention on the queue increases: both with increasing the degree of parallelism and increasing the remote event percentages.

In Figure 9 we show another snapshot of the Magny-cours machine performance. In this case, we fix the percentage of remote communication and show the performance improvement for varying degrees of parallelism. In general, the performance benefit of the individual optimizations increases with the degree of parallelism.

The results show that the Barrier and NUMA optimizations signifcantly improve performance when remote communication is infrequent. In such models, the LIFO strategy in the NUMA optimization likely leads to

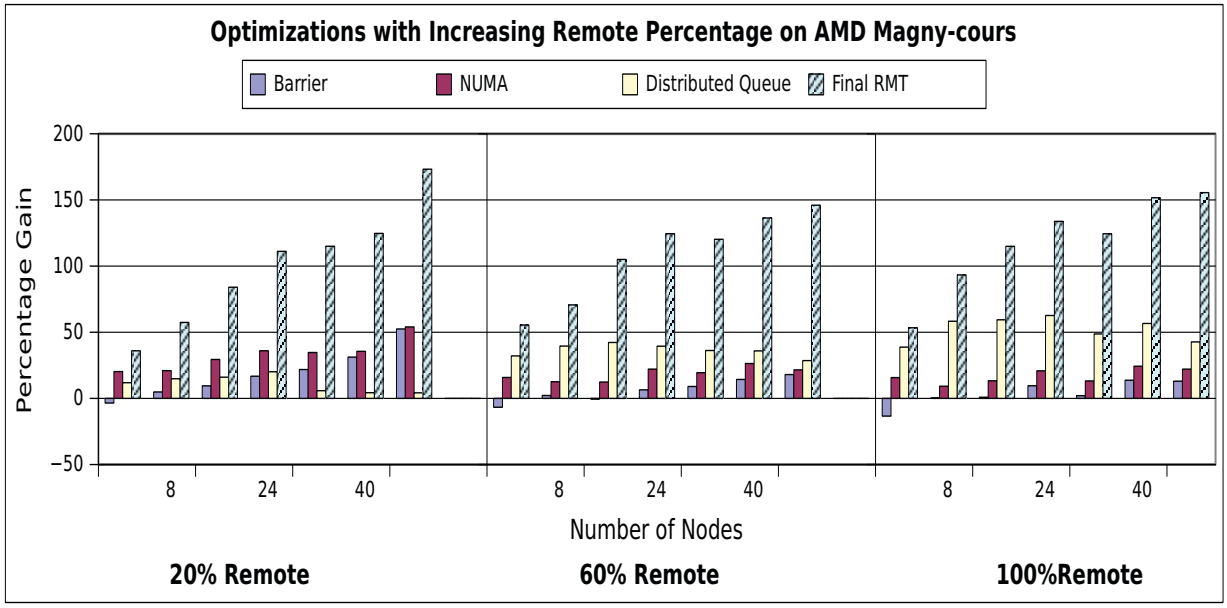


Figure 9: Magny-cours Performance as a function of Remote Events

much better cache locality explaining the better performance. As can be seen in Figure 7, GVT computation consumes a substantial portion of execution time in the 20% remote case; the barrier optimization impacts GVT computation and significantly improves performance. Finally, the distributed queue optimization increases in impact as the percentage of remote communication increases, increasing the perssure on the input queue. Note that in these figures, we show the speedup (rather than reduction in run-time); this explains why their combined effect is high since it is bound by the product rather than the sum of the speedup from the individual optimizations.

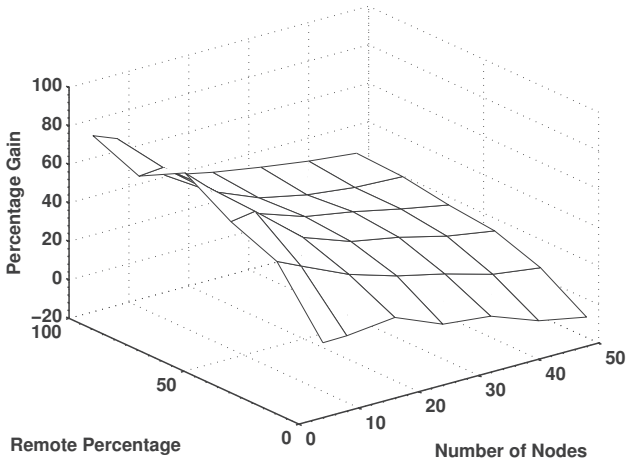


Figure 10: Performance Improvement Relative to MPI–Magny-cours

Figure 10 shows the performance improvement for the Magny-cours ROSS-MT with all optimizations relative to the baseline ROSS with MPI. The performance improvement increases as the percentage of remote events increases (increasing the use of the communication subsystem and the optimizations). However, in general, the benefit is less pronounced as the degree of parallelism is increased. We believe that with some effort, we can track down additional lock contention issues and address them.

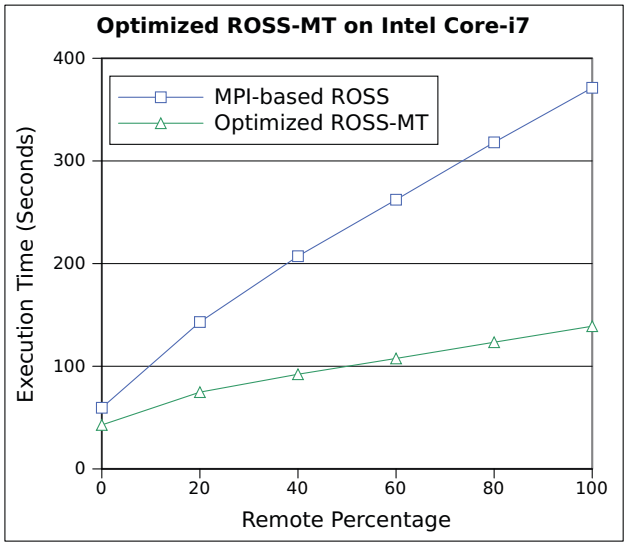


Figure 11: Execution time of the optimized ROSS-MT on Intel core-i7

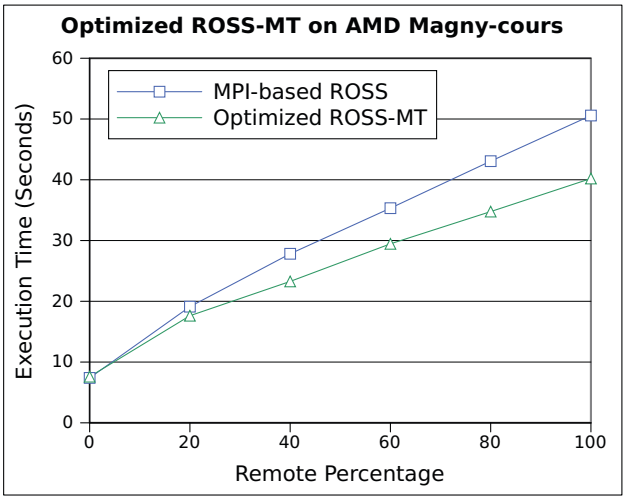


Figure 12: Execution time of the optimized ROSS-MT on AMD Magny-cours

We show the impact of the optimizations on run time on the core i7 machine with 8 cores and event message size of 8 bytes in Figure 11. Its clear that the multi-threaded implementation is substantially faster than the MPI version on this platform. Figure 12 shows the same comparison for the Magny-cours platform with 48 cores (same message size). ROSS-MT also outperforms the MPI version, although the gap is substantially closer.

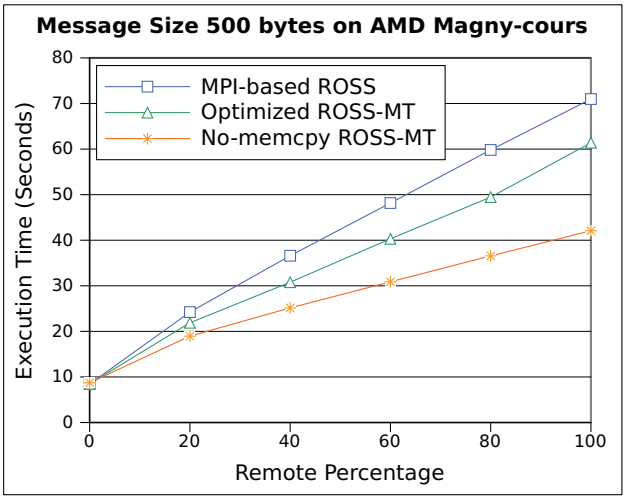


Figure 13

In the next experiment, we study the impact of bigger message sizes on the performance of the simulator. The message size is a key factor impacting communication cost; not only is transmission cost increased, but buffer copies and checksum operations increase in cost with the size of the message. We studied the impact of message size on event rate of MPI based ROSS and ROSS-MT. Figure 13 shows the performance of ROSS-MT on the AMD Magny-cours with message size 500. The multi-threaded implementation does a message copy into the input queue to keep a separate copy in case of rollback. However, this additional copy can be avoided by keeping pointers and tracking sharing to the message. We implemented a preliminary version of this optimization (results shown in Figure 12); substantial reduction in execution time are observed, making the ROSS-MT as much as 65% faster than the MPI implementation. We believe that there remains room for ROSS-MT optimization.

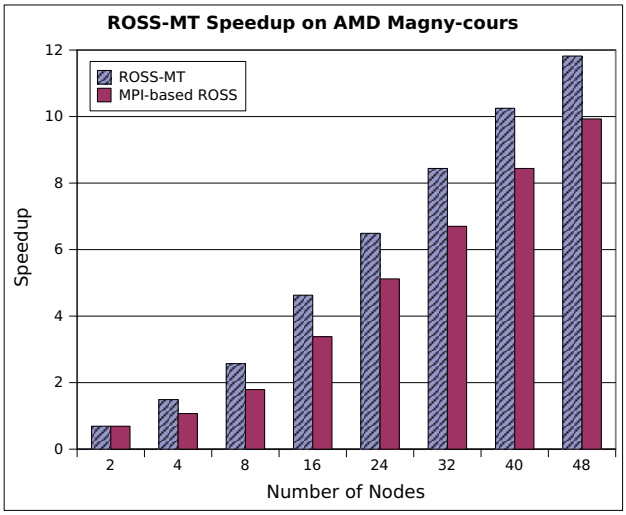


Figure 14: Speedup for AMD Magny-Cours

Finally, Figure 14 shows the speedup that ROSS-MT and MPI ROSS relative to sequential execution as the number of cores is increased with 100% remote communication. Speedup increases linearly up to the full 48-cores for both implementations where ROSS-MT achieves speedup close to 12, while the MPI version achieves a speedup of 10.

In summary, we showed the performance of ROSS-MT on two different multi-core platforms. The large difference in behavior can be explained by the following differences in their architecture.

1) Even though the number of cores is high on the Magny-cours each core is individually less than the Intel core i7 cores.
2) Although NUMA remote cache access issues have been improved by the NUMA optimization and LIFO free memory strategy, NUMA issues are still present and significant remote cache accesses occur in the current multithreaded ROSS implementation.

3) Further, only L3 is shared among the 6 cores on a die. On core i7, all the cores share L3, and two hyperthreads on the same core share L1 and L2.

VI. Related Work

In the general high performance computing community, it is well known that communication is a common performance bottleneck, especially for fine-grained parallel applications [27]. As a result, managing the impact of communication is a recurring focus of the parallel processing community. One of the approaches to reduce communication latency is to improve the network performance. For clusters, high performance networks [19] and networking abstractions, communication libraries and system software implementations (e.g., [3], [9], [16]) have been proposed. In a multi-core environment, the design of the on-chip interconnect remains an open research problem [23]; most existing designs use the on-chip interconnect to implement indirect communication through shared caches.

At the application/algorithm levels, there are general techniques for optimizing parallel applications such as overlapping computation and communication and reducing lock contention) [27], [1]. Efficient partitioning is necessary to reduce remote communication [30], [18]. However, most efforts in implementing parallel applications discover that application insights and awareness of the architecture are necessary to optimize the parallel implementation [26], [6], [11]. Parallel Discrete Event Simulation is difficult to parallelize because of its fine-grained nature, and complex and dynamic dependency pattern [12], making it substantially different from typical parallel applications. Thus, we focus on optimizations specific to PDES, rather than other applications or parallel processing in general.

A. Optimizing Communication for PDES

Previous works have demonstrated the importance of partitioning to reduce the communication frequency in PDES (e.g., [18]). Similarly, dynamic partitioning and workload rebalancing mechanisms have been proposed to repartition the simulation to recover dynamic behavior changes of the simulation model for both conservative (e.g., [4]) and optimistic (e.g., [25]) synchronization protocols.

Chetlur et al [7] 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. Sharma et al [31] explored optimizing the polling frequency to check for the presence of event messages. Rajasekaran et al [28] explored using a single anti-message with the earliest rollback timestamp to inform receiver PEs of rollbacks instead of sending individual anti-messages for each remote event to be cancelled. Mattern developed a 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 [20]. Fujimoto et al designed the rollback chip, which implements wolf-calls (very fast notification of all PEs of the occurrence of a rollback) [15]. Wolf-calls substantially limit cascading rollbacks that occur due to communication latency. Noronha et al used a programmable network card to optimize event communication and GVT computation [22].

B. Shared Memory PDES

Fujimotos GTW simulator is one of the first shared memory optimistic PDES implementations. It exploits shared memory for efficient message communication. GTW also implemented optimizations such as direct cancellation, which allows an LP to cancel out erroneously sent remote events directly, eliminating the need for anti-messages [10]. Similarly, in shared memory, messages can simply be written into a buffer and become visible to all processors. The exploitation of such features allows the GVT computation algorithm to be implemented through a single round of inter-processor communication (as opposed to at least two rounds required for message passing programming model) with minimal number of shared variables and data structures. Fujimoto and Hybinette also describe an efficient *on-the-fly* fossil collection algorithm to enable fast reclamation of memory [14]. They also explore efficient buffer management algorithms for shared memory environments [13].

While some of the schemes developed for shared memory, which were developed and evaluated on Symmetric Multi-processor machines, can apply for many-cores, the tighter coupling of processing elements and shorter communication delays for accessing shared caches requires at least a careful reconsideration of these approaches. Our paper reports on experiences in optimizing a PDES simulator on emerging multicore systems.

Our work is targeted towards emerging multi-core and many-core architectures. Current examples of these architectures commonly employ chips with multiple-cores with on chip memory controllers such as the AMD Opteron Magny-cours. [21] provides insights into NUMA related performance issues on such multi-core platforms and also discusses commonly employed

solutions to these problems. Our multi-threaded implementation can be much more beneficial on these platforms provided the design is NUMA-aware. Some of the techniques presented in [21] were incorporated in our work such as use of first-touch policy for memory allocations.

VII. Concluding Remarks and Future Work

In this paper, we presented our experiences in building a multi-threaded PDES simulator optimized to representative state-of-the-art multi-core machines. We used the ROSS PDES simulator, and reimplemented it from a process based model to a thread based model. Performance evaluation of the base implementation showed signficant performance benefits on core i7, but 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 locks and broadcasts. We replaced it with an implementation that is based on pthread_barrier, which uses native machine instructions for implementing barriers, significantly improving the performance of GVT. The second problem was due to free memory management, which was not sensitive to the NUMA nature of the Magny-cours platform. We addressed this problem by splitting the free memory pool to keep track of the memory origin for future allocation. The third bottleneck was due to the lock contention on the input queue. We resolved this issue by splitting the queues to reduce contention from all threads to only two threads for each queue.

The optimizations resulted in subtantial improvement in performance; optimized ROSS-MT outperforms the MPI version by a factor of up to 3 on core i7 and up to 1.2 on Magny-cours. We also explore a preliminary implementation of a no-copy version of our communication primitives, which significantly improves performance especially when message lengths are large.

Our future work includes integrating ROSS-MT to allow operation on a cluster of multi-cores. We also plan to look at additional algorithmic optimizations that are possible due to the tight integration available in a multi-threaded implementation.

VIII. Acknowledgements

This material is based on research sponsored by Air Force Research Laboratory under agreement number FA8750-11-2-0004 and by National Science Foundation grants CNS-0916323 and CNS-0958501. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies and endorsements, either expressed or implied, of Air Force Research Laboratory, National Science Foundation, or the U.S. Government.

References

[1] G. Andrews. *Foundations of Multithreaded, Parallel and Distributed Programming*. Addison Wesley, 2000.

[2] D. Bauer, C. Carothers, and A. Holder. Scalable time warp on bluegene supercomputer. In *Proc. of the ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation (PADS)*, 2009.

[3] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet – a Gigabit-per-second Local-Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[4] A. Boukerche and S. Das. Dynamic load balancing strategies for conservative parallel simulation. In *Proc. 11th Workshop on Parallel and Distributed Simulation (PADS)*, pages 32–37, 1997.

[5] C. Carothers, D. Bauer, and S. Pearce. ROSS: A high-performance, low memory, modular time warp system. In *Proc of the 11th Workshop on Parallel and Distributed Simulation (PADS)*, 2000.

[6] A. Chandramowlishwaran, S. Williams, L. Oliker, I. Lashuk, G. Biros, and R. Vuduc. Optimizing and tuning the fast multipole method for state-of-the-art multicore architectures. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2010.

[7] M. Chetlur, N. Abu-Ghazaleh, R. Radhakrishnan, and P. A. Wilsey. Optimizing communication in Time-Warp simulators. In *12th Workshop on Parallel and Distributed Simulation*, pages 64–71. Society for Computer Simulation, May 1998.

[8] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache hierarchy and memory subsystem of the amd opteron processor. *IEEE Micro*, 30(2):16–29, 2010.

[9] L. Dagum and R. Menon. OpenMP: an industry standard API for shared memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[10] S. Das, R. Fujimoto, K. Panesar, D. Allison, and M. Hybinette. GTW: a Time Warp system for shared memory multiprocessors. In J. D. Tew, S. Manivannan, D. A. Sadowski, and A. F. Seila, editors, *Proceedings of the 1994 Winter Simulation Conference*, pages 1332–1339, December 1994.

[11] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore arhcitectures. In *Proc. of ACM/IEEE Confernece on Supercomputing*, 2008.

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

[13] R. Fujimoto and K. Panesar. Buffer management in shared-memory Time Warp system. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation (PADS 95)*, pages 149–156, June 1995.

[14] R. M. Fujimoto and M. Hybinette. Computing global virtual time in shared-memory multiprocessors. *ACM Transactions on Modeling and Computer Simulation*, 7(4):425–446, 1997.

[15] R. M. Fujimoto, J. Tsai, and G. C. Gopalakrishnan. Design and evaluation of the rollback chip: Special purpose hardware for Time Warp. *IEEE Transactions on Computers*, 41(1):68–82, January 1992.

[16] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, MA, 1994.

[17] D. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):405–425, July 1985.

[18] L. Li and C. Tropper. A design-driven partitioning algorithm for distributed verilog simulation. In *Proc. 20th International Workshop on Principles of Advanced and Distributed Simulation (PADS)*, pages 211–218, 2007.

[19] J. Liu, B. chandrasekaran, J. Wu, W. Jiang, S. Kini, W. Yu, D. Buntinas, P. Wyckoff, and D. Panda. Performance comparison of mpi implementations over infiniband, myrinet and quadrics. In *Proc. of ACM/IEEE conference on Supercomputing*, pages 58–71, November 2003.

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

[21] Collin McCurdy and Jeffrey Vetter. Memphis: Finding and fixing numa-related performance problems on multicore platforms. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium*, pages 87 – 96, March 2010.

[22] R. Noronha and N. B. Abu-Ghazaleh. Active nic optimization for time warp. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2002.

[23] J. Owens, W. Dally, D. Jayasimha, S. Keckler, and L. Peh. Research challenges for on-chip interconnection networks. *IEEE Micro*, 27(5):96–108, 2007.

[24] K. Perumalla. Scaling time warp-based discrete event execution to 10^4 processors on a blue gene supercomputer. In *Proc. of the ACM Conference on Computing Frontiers (CF)*, 2007.

[25] P. Peschlow, T. Honecker, and P. Martini. A flexible dynamic partitioning algorithm for optimistic distributed simulation. In *Proc. 20th International Workshop on Principles of Advanced and Distributed Simulation (PADS)*, 2007.

[26] F. Petrini, G. Fossum, J. Fernandez, A. Varbanescu, N. Kistler, and M. Perrone. Multicore surprises: Lessons learned from optimizing Sweep3D on the cell broadband engine. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2007.

[27] M. Quinn. *Parallel Computing: Theory and Practice*. McGraw Hill, 1994.

[28] U. K. V. Rajasekaran, M. Chetlur, G. D. Sharma, R. Radhakrishnan, and P. A. Wilsey. Addressing communication latency issues on clusters for fine grained asynchronous applications — a case study. In *International Workshop on Personal Computer Based Network of Workstations, PC-NOW'99*, April 1999.

[29] P. L. Reiher, F. Wieland, and D. R. Jefferson. Limitation of optimism in the Time Warp operating system. In *Winter Simulation Conference*, pages 765–770. Society for Computer Simulation, December 1989.

[30] V. Sarkar and J. Hennessy. Compile-time partitioning and scheduling of parallel programs. In *Proc. of the SIGPLAN Symposium on Compiler construction*, 1986.

[31] G. D. Sharma, N. B. Abu-Ghazaleh, U. V. Rajasekaran, and P. A. Wilsey. Optimizing message delivery in asynchronous distributed applications. In *11th International Conference on Parallel and Distributed Computing Systems*, September 1998.