Performance comparison of Cross Memory Attach capable MPI vs. Multithreaded Optimistic Parallel Simulations

Dhananjai M. Rao
Miami University
Oxford, Ohio
raodm@miamiOH.edu

ABSTRACT

The growth in many-core CPUs has motivated development of shared-memory, multithreaded solutions to minimize communication and synchronization overheads in Parallel Discrete Event Simulations (PDES). Analogous capabilities, such as Cross Memory Attach (CMA) based approaches have been added to Message Passing Interface (MPI) libraries. CMA permits MPI-processes to directly read/write data from/to a different process's virtual memory space to exchange messages. This paper compares the performance of CMA capable, MPI-based version to our fine-tuned multithreaded version. The paper also discusses implementation and optimization of the multithreaded infrastructure to elucidate the design alternatives being compared and assessed. Our experiments conducted using 2–28 threads and a fine-grained (time per event 0.7 μs) version of PHOLD benchmark shows that message-passing outperforms multithreading (by 10%–20%) in many scenarios but underperforms in others. The complex performance landscape inferred from our experiments suggest that more in-depth analysis of model characteristics is needed to decide between shared-memory multithreading versus message-passing approaches.

CCS CONCEPTS

• **Theory of computation → Data structures design and analysis**; • **Computing methodologies → Discrete-event simulation**; **Distributed simulation**;

KEYWORDS

Discrete Event Simulation (DES); Optimistic Parallel Simulation; Time Warp; Multithreading; Cross Memory Attach; NUMA; Ladder Queue (`ladderQ`); Three Tier Heap (`3tHeap`)

ACM Reference Format:
Dhananjai M. Rao. 2018. Performance comparison of Cross Memory Attach capable MPI vs. Multithreaded Optimistic Parallel Simulations. In *SIGSIM-PADS '18 : SIGSIM-PADS '18: SIGSIM Principles of Advanced Discrete Simulation CD-ROM, May 23–25, 2018, Rome, Italy.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3200921.3200935

SIGSIM-PADS '18 , May 23–25, 2018, Rome, Italy

ACM ISBN 978-1-4503-5092-1/18/05…$15.00
https://doi.org/10.1145/3200921.3200935

1 INTRODUCTION

Modern computational platforms are continuing to trend towards high density architectures with compute nodes having 2 or more, many-core CPUs. In these architectures, the main memory is shared between CPUs. Accordingly, shared-memory approaches for Parallel Discrete Event Simulation (PDES), often accomplished via multithreading, are gaining momentum [1, 3, 5, 11]. The primary advantage of shared-memory design stems from eliminating overheads of message-passing for *intra*-node communication by directly sharing events between threads. Prior investigators have reported good performance improvements using shared-memory multithreading approaches over message-passing designs [2, 10, 11]. Analogous optimizations have also been incorporated into the infrastructure of message-passing libraries. For example, Cross Memory Attach (CMA) capabilities, discussed in Section 1.1, have been added to the Linux kernel and Message Passing Interface (MPI) libraries to further reduce message-passing overheads.

Recently, in our cluster (details in Section 2) the number of CPU-cores per node more than doubled from 8 to 28 cores (with hyperthreading disabled). Consequently, we have significantly revised our MPI-based, optimistically synchronized PDES framework (discussed in Section 3) to operate in multithreaded mode. The overarching objective of multithreaded PDES is to realize better performance when compared to our message-passing design. Our multithreaded design, detailed in Section 4, uses decentralized pending event set design due to its advantages [1, 2] – *i.e.,* each thread has its own pending event queue and scheduler. Furthermore, we have explored several design alternatives to maximize multithreading performance, including: ❶ sharing-events between threads vs. exchanging copies, ❷ NUMA-aware memory allocation for events, and ❸ Lock-based vs. lock-free inter-thread queues. Section 5 compares and contrasts our designs to those proposed by other investigators. The objective is to identify and use the most performant design solution from the aforementioned alternatives.

Literature survey supported our hypothesis that multithreaded PDES would yield performance improvement over MPI-based implementations [1, 2, 11], particularly in fine-grained applications. However, experiments using fine-grained (time-per-event < 0.7 µsec), PHOLD benchmark revealed a more complex performance landscape, with MPI version (with CMA-capability) conspicuously outperforming multithreaded simulations and vice versa. The experiments discussed in Section 6 highlight the complex landscape with no clear winner. The results lead us to conclude that a comprehensive analysis of application characteristics (future work) is needed in order to choose between message-passing and multithreading designs.

1.1 CMA & Open MPI's vader BTL

Cross Memory Attach (CMA) is a mechanism to directly transfer data between the virtual memory space of two processes running on the same compute node – *i.e.,* intra-node Inter-Process Communication (IPC). CMA enables data transfer without passing through kernel space. CMA has been added to the Linux-kernel starting with version 3.2 (Jan 2012 release). In Linux, CMA is accomplished via two system calls, namely `process_vm_readv` and `process_vm_writev`. CMA enables processes to accomplish "zero copy" intra-node data transfers. Note that in Linux parlance "zero copy" implies using a single copy of data (or messages) and avoiding overheads of requiring extra copies.

Starting with Open MPI version 1.8.4 (early 2015), the CMA capabilities of Linux have been used to develop a Byte Transfer Layer (BTL) subsystem called `vader` [9]. The `vader` BTL improves small message latency via "zero copy" transfers, typically via the `process_vm_readv` CMA system call [9]. The BTL has also shown to have substantially better throughput than traditional shared memory BTL, in multi-CPU nodes [9]. An experimental comparison of CMA-based `vader` BTL versus conventional shared memory BTL is discussed in Section 3

2 EXPERIMENTAL PLATFORM

The experiments reported in this paper have been conducted using shared-memory compute nodes with two (dual socket) Intel Xeon® CPUs (E5-2680 v4) with hyperthreading disabled. Each CPU has 14 cores and 35 MiB of shared L3 cache between the cores. Each core has 64 KiB L1 (*i.e.,* 32 KiB instruction + 32 KiB data split cache) and 256 KiB of L2 cache. The 128 GB of DDR4 RAM (64 GB per CPU) in Non-Uniform Memory Access (NUMA) configuration as detailed in Figure 1. The cores on the two CPUs are logically interleaved. Memory access time or distances is 10 units between cores on the same CPU but more than doubles to 21 units for cross-CPU memory access. The compute node runs Red Hat Enterprise Linux (kernel version 3.10.0-514) that supports Cross Memory Allocation (CMA). The simulation software and benchmarks were compiled using Intel C++ Compiler (ICC) version 16.0 at `-O3` optimization level. Open MPI version 2.1.2 with `vader` BTL that utilizes CMA capabilities (also compiled using ICC 16.0) has been used for inter-process communication.

```
$ numactl -H
available: 2 nodes (0-1)
node 0 cpus: 0 2 4 6 8 10 12 14 16 18 20 22 24 26
node 0 size: 130850 MB
node 0 free: 128020 MB
node 1 cpus: 1 3 5 7 9 11 13 15 17 19 21 23 25 27
node 1 size: 131072 MB
node 1 free: 122325 MB
node distances:
node 0 1
  0: 10 21
  1: 21 10
```

Figure 1: NUMA configuration on compute node

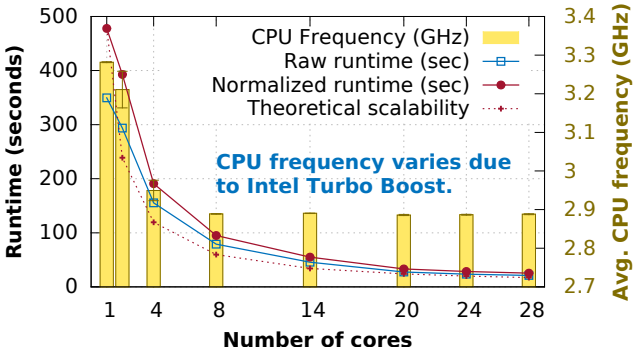


Figure 2: CPU frequency changes due to Intel® Turbo Boost & runtime normalization

2.1 Normalization due to turbo boost

The nodes used for experiments have Intel® Turbo Boost Technology 2.0 enabled. Consequently, the core frequency significantly varies, from base frequency of 2.4 GHz, depending on temperature and utilization of the node as shown in Figure 2. In our experiments the CPU-core frequency varied by about 12%. Furthermore, variations in CPU-core frequencies was also observed between successive runs, when the same number of cores were used – *e.g.,* with 2 cores a ±5% variation was observed. These variations in CPU clock frequency causes runtimes to vary resulting in inconsistent comparisons. Unfortunately, the compute cluster is a shared, state-wide resource which makes modifications to hardware or BIOS settings a cumbersome process.

Consequently, we have normalized all observed runtimes to a common CPU-core frequency of 2.4 GHz using the following equation:

$$t_{norm} = \frac{Cycles_{cpu}}{Utilization_{cpu} \times 2.4 \times 10^9} \quad (1)$$

where t_{norm} is normalized runtime, $Cycles_{cpu}$ is number of CPU cycles used, and $Utilization_{cpu}$ is CPU utilization averaged over the entire run of the program. Given c cores for a run, CPU utilization can be in the range $0 < Utilization_{cpu} \leq c$. The statistics for normalization is obtained by running all of the simulations via Linux `perf` and recording necessary CPU counters.

3 MPI-BASED DESIGN & OPTIMIZATION

The implementation and assessment of multithreading vs. CMA-enabled MPI has been conducted using a Parallel Discrete Event Simulation (PDES) framework called MUSE. It has been developed in C++ using object-oriented approaches and the Message Passing Interface (MPI). MUSE uses Time Warp and standard state saving approach to accomplish optimistic synchronization of the LPs. A conceptual overview of a parallel, MPI-based simulation is shown in Figure 3. The simulation is organized as a set of processes that communicate via MPI. Each process has one thread and manages a set of Logical Processes (LPs) assigned to it. Each process uses a centralized Least Timestamp First (LTSF) priority queue for managing pending events and scheduling event processing for all local LPs. LPs are permitted to generate events only into the future – *i.e.,*

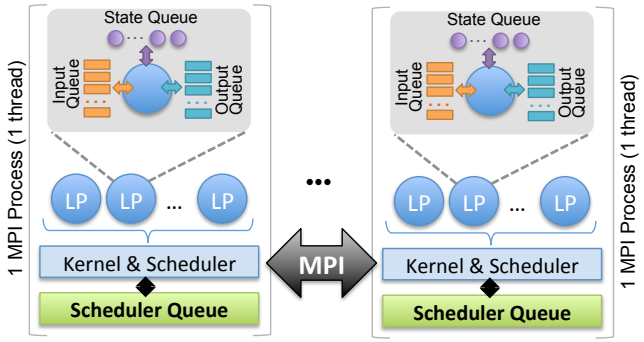


Figure 3: Overview of MPI-based PDES

the timestamp on events must be greater than their Local Virtual Time (LVT). Consequently, with the centralized LTSF scheduler, event exchanges between local LPs cannot cause rollbacks. Only events received via MPI can cause rollbacks.

The Logical Processes (LPs) in a simulation are developed by overriding necessary methods in an `Agent` base class. The input, output, and state queues used for rollback operations in Time Warp are managed by the `Agent` base class in coordination with the simulation-kernel. Similarly, the MUSE provides an `Event` base class that can be extended to implement custom events for use by the model. The simulation-kernel implements core functionality associated with LP registration, event processing, state saving, synchronization, and Global Virtual Time (GVT) based garbage collection.

3.1 PHOLD benchmark

Experimental analysis of design alternatives has been conducted using different configurations of the PHOLD benchmark. The PHOLD benchmark has been used by several investigators [2, 8, 11] for experimental analysis because it has shown to effectively emulate the steady-state phase of a typical simulation. Our PHOLD implementation provides a number of configuration settings to alter its behavior thereby streamlining design of experiments. In our experiments, we have used a PHOLD model with 10,000 LPs organized in a 100 ×100 torroidal grid. The simulation commences with 20 events (40 bytes per event) per LP, resulting in a pending event set of 200,000 events. The timestamps on the events is determined using an exponential distribution (λ=10). The destination LP for each event is computed using two different approaches as detailed in the following subsections. Additional details on our PHOLD benchmark is included in supplementary materials.

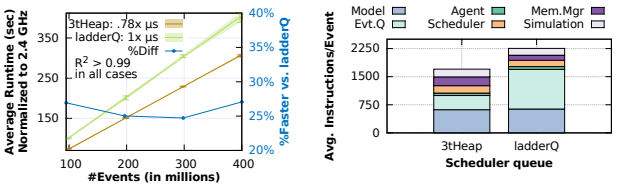
3.1.1 Config #1: Fixed Inter-LP interactions (strong scaling). This configuration reflects a typical parallel simulation experiment in which the properties of the model do not change based on number of processes/threads. In this mode of operation, the range of LPs to which events are scheduled is bounded by a value specified via the `recvr-distrib` command-line argument to PHOLD. Specifically, given an LP with ID k and a `recvr-distrib` value of x, a destination LP d is uniformly chosen from the range $k - \frac{x}{2} < d < k + \frac{x}{2}$, with wrap around due to toroidal space. The choice of destination LPs is determined by value of x and does not change with partitioning. In other words, the behavior of the model is independent of the number parallel processes used, reflecting a strong scaling configuration. However, it must be noted that, as the number of partitions (or parallel threads/processes) are increased, the probability of Inter-Process Communication (IPC) increases, resulting in increased synchronization overheads. In our benchmarks we have used the following values for the `recvr-distrib` (*i.e.*, x) – 10, 100, 1000, and 10000. Note that larger values result in increased probability of IPC.

3.1.2 Config #2: Fixed fraction of remote events (weak scaling). Events exchanged between pairs of processes or threads are called *remote* events. In our design, only remote events can trigger rollbacks, which play an influential role on the performance of optimistic PDES. In other words, communication characteristics strongly influence probability of rollbacks, with increased remote events resulting in increased *probability* of rollbacks [7, 11]. Accordingly, this configuration is designed to fix the number of remote events to assess its impact in a controlled manner. Specifically, each LP chooses a destination such that the fraction of *remote* events remains fixed. The fraction of remote events (in the range 0.0 to 1.0) is specified via a `remote-events` command-line argument to the benchmark. Since remote events between any pair of threads is fixed, the communication and synchronization overheads are also bounded, immaterial of number of process/threads used. This setting is analogous to weak scaling configurations that are often used for performance assessments [11]. It must be noted that in Time Warp synchronized parallel simulations, the net number of inter-process messages may vary due to exchange of anti-messages. In our benchmarks we have used the following fraction of remote events, *i.e.*, value for `remote-events` parameter: 0.1, 0.25, 0.5, 0.75, and 0.9.

3.2 Selection of scheduler queue

The Least Timestamp First (LTSF) priority queue associated with each process (or thread) plays a conspicuous role in realizing efficient and performant parallel simulation. In this study, we have used the Three-tier Heap (`3tHeap`) proposed by Higiro *et al* [4] as the scheduler queue rather than the Ladder Queue (`ladderQ`). The `3tHeap` was chosen because it yielded better performance in several configurations, particularly in simulations with large number of events with small differences in virtual timestamps. Comparison of scheduler queues was performed using single process simulations in which state saving, rollbacks, GVT, etc. are automatically turned-off in the simulation-kernel. Single process simulations have used for comparisons for the following reasons: ① the `ladderQ` has been primary designed for use in sequential simulations. The 1 process simulation is analogous to a sequential simulation, thereby enabling consistent/fair comparisons; ② eliminating synchronization protocol overheads enables effectively isolating impact of scheduler queues; and ③ assess the fine-grained nature of the benchmarks used for further analysis.

The chart in Figure 4(a) illustrates a comparison between our `ladderQ` and `3tHeap` implementations. The experiments were performed on the hardware platform discussed in Section 2 using the configuration of PHOLD benchmark discussed in Section 3.1. The total number of committed events in the simulation (the independent axis in Figure 4(a)) was varied by increasing the simulation end



(a) Time & linear regression (b) Instruction use per event

Figure 4: `ladderQ` vs. `3tHeap`: comparison of runtimes and instructions/event in a single process simulation

times. The chart plots the average runtime from 10 independent replications at each data point along with linear regression fits. The regression fits were very strong with $R^2 > 0.99$ in all cases.

Linear runtimes for `ladderQ` with amortized $O(1)$ runtimes is expected, as per its design discussed by Tang *et al* [8]. The average time for processing an event with `ladderQ` varied between 0.96–1.06 μs (normalized to 2.4 GHz as discussed in Section 2.1), based on 95% CI for the regression fit. The chart in Figure 4(b) shows the distribution of average instructions per event in the model. Each event requires about 2,374 instructions, of which 1,058 instructions (44%) are used by `ladderQ` operations. The PHOLD model uses 635 instructions to process an event, of which 50% is used by random number generation to determine future timestamps and destination LPs for scheduling new events.

Interestingly, the `3tHeap` also exhibits a linear runtime profile for the PHOLD benchmark, as shown in Figure 4(a). We attribute this characteristic to the constant number of events (but with different virtual timestamps) in the simulation. Furthermore, the event processing time decreased to 0.77–0.79 μs (also normalized to 2.4 GHz), with the `3tHeap`, as it requires only 619 instruction/event (instead of the 1,058 instructions for `ladderQ`). The 58% reduction in event count enables the `3tHeap` to consistently outperform the `ladderQ`, but only by about 25%–27% (blue curve in Figure 4(a)) in our benchmarks. The discrepancy in speed is attributed to the following observations – ① some operations (such as, checking if queue is empty) requires a few additional instructions in the case of `3tHeap` which slightly increases instruction counts within the simulation-kernel as illustrated by Figure 4(b); ② the CPU's instructions-per-cycle decreased slightly from 1.05 (*i.e.,* over 1 instruction/cycle on a superscalar core) for `ladderQ` to 0.95 (*i.e.,* a 10% decrease) with the `3tHeap`. However, the CPU-cache hits per instruction was comparable for both queues indicating that caching was not a significant factor in performance difference. Overall, the `3tHeap` provided performance improvement (when compared to Ladder Queue) for the benchmarks reported in this paper. Consequently, we have used it for all empirical analyses in this paper.

3.3 GVT-based Adaptive Time Window (ATW)

The first phase of this investigation focused on identifying effective configuration for MPI-based simulations. We have explored the effectiveness of both standard shared-memory Byte Transfer Layer (BTL) and CMA-based `vader` BTL in Open MPI. In our initial experiments, we observed that aggressive optimism was causing significant number of rollbacks, degrading performance. Therefore,

Algorithm 1: Adaptive Time Window algorithm

```
1  begin rollback(agent, event, GVT)
2      rbDist = e⟶recvTime - gvt;
3      adaptTW = avg(adaptTW, Δt);
4      if adaptTW.samples > 100 then
5          timeWindow = adaptTW.mean;
6      end if
7  end adaptRollback
8  begin scheduleEvent(agent, event, GVT)
9      if timeWindow == 0 then
           // time window not yet set
10         return true
11     end if
12     Δt = event⟶recvTime - GVT
13     if Δt ≤ timeWindow then
14         return true
15     else
16         adaptTW = avg(adaptTW, Δt)
17         timeWindow = adaptTW
18     end if
19 end scheduleEvent
```

we have implemented a GVT-based, Adaptive Time Window (ATW) algorithm summarized in Algorithm 1. It uses average rollback distance with respect to GVT to determine a "safe" time window in which events can be optimistically scheduled. Shorter rollbacks result in decreasing the time window restricting optimism. The event scheduler (*cf.,* `scheduleEvent` in Algorithm 1) uses the adaptive time window and GVT to schedule events. If an event's timestamp is within the time window then it is scheduled for processing. Otherwise, the difference between GVT and the event's timestamp is used to grow the time window. The ATW is a fully distributed algorithm in that it uses only locally available information. Moreover, the averaging approach using by ATW enables the algorithm to be immune to transient fluctuations in steady-state models. On the other hand, the shortcoming of averaging is that it it hinders quickly adaptation to non-transient changes in model characteristics. However, in this paper we have focused on steady-state benchmarks. Consequently, as discussed in Section 3.3.1 this algorithm proved to be effective in managing optimism.

3.3.1 Assessment of ATW for MPI-based PDES. Communication patterns influence rollback characteristics which strongly influences the behavior of the Adaptive Time Window (ATW) algorithm summarized in Algorithm 1. Consequently, have used `Config #2` discussed in Section 3.1.2, with fixed fraction of remote events for assessing impact of the ATW. The charts in Figure 5 compare runtime (average and 95% CI from 10 replications) of the benchmark, with different fractions of remote events, with both CMA-enabled `vader` and shared memory (shown as `Shr. Mem` in charts) Byte Transfer Layers (BTLs) in Open MPI. The inset charts show raw percentage difference in runtime with respect to `CMA/vader+ATW` (—×—) setting, which outperformed all other configurations. As illustrated by the charts,, with shared memory BTL, the ATW (curve —●—)

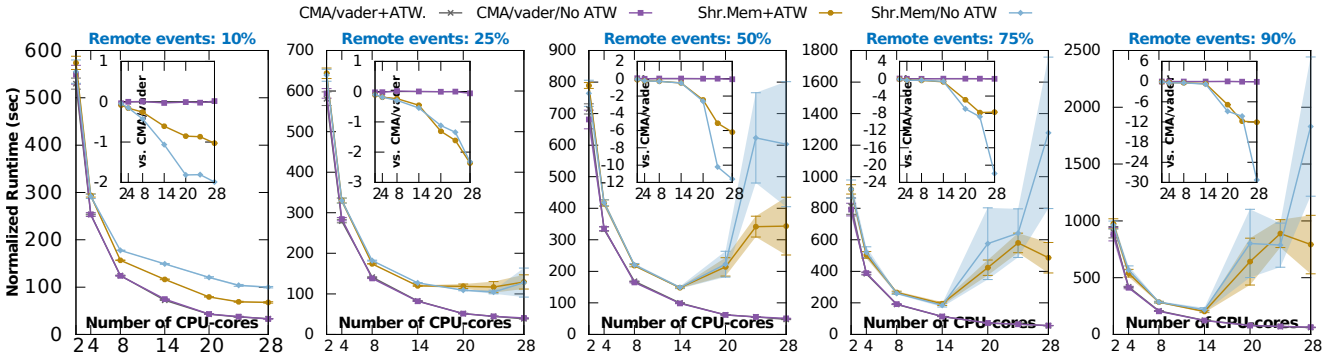


Figure 5: Performance comparison of PHOLD (Config #2 in Section 3.1.2) runtimes with different settings. In inset charts, data points below the zero-axis correspond to results in which `CMA/vader+ATW` performed better.

consistently outperforms the setting without ATW (curve —◆—) and improves performance up to 200×, particularly in configuration with a large fraction of remote events. However, with `vader` BTL, the performance improvement with ATW (curve —■—) was statistically negligible (< ±2%) as illustrated by Figure 5. We attribute the lack of significant performance improvement to the low latency communication enabled by CMA-capable `vader` BTL. However, in some runs of `Config #1` (see Section 3.1.1) the ATW yielded 10% performance improvement. Consequently, we have consistently used `vader` BTL with ATW in our subsequent experiments. We have also used ATW with our multithreaded configurations (*cf.*, Section 6.1), because it provided more conspicuous improvements in several cases.

4 MULTITHREADING DESIGN & TUNING

In conjunction with this paper, we have extended our simulation framework to enable multithreaded PDES. The multithreading capabilities reuse our existing Application Program Interface (API). Consequently, existing models and benchmark can be readily reused. Figure 6 presents an overview of our multithreaded PDES infrastructure. The design of our multithreaded infrastructure mirrors several key elements from the MPI-based design discussed in Section 3. A multithreaded PDES is organized as a collection of interacting threads. Each thread is synonymous to an MPI-process (see Figure 3). Each thread manages lifecycle activities of LPs partitioned to it, including: scheduling events, state saving, rollback recovery, and GVT-based garbage collection. We have retained the API, design, and model-specific characteristics from our MPI-based design. Consistency in API and design enables effective reuse of existing models by isolating it from the underlying framework's operational modes. Moreover, it enables consistent comparison of performance of different framework features and optimizations.

The design of event scheduling and pending event management also similar to our prior MPI-based design. Each thread uses an independent local scheduler queue (*e.g.*, a `3tHeap`) for managing pending events. The internal framework design is identical to the MPI-design thereby enabling reuse of existing priority queue implementations. In this study, we have used the Three-tier Heap data structure proposed by Higiro *et al* as discussed in Section 3.2.

As illustrated by Figure 6, the multithreaded design uses decentralized scheduler queues – *i.e.,* one scheduler queue per thread to manage pending events. Each thread has its own GVT manager (reused from our MPI-based implementation) that uses Mattern's GVT algorithm. Currently, the scheduler queues operations do not involve any locking or lock-free instructions; *i.e.,* they are not designed to be thread-safe. On the other hand, implementation, optimization, and validation of the queues is relatively straightforward when compared to their concurrent counterparts. Most of the design is similar to our MPI-based implementation making them comparable. However, certain key aspects differ and are discussed in detail in the following subsections.

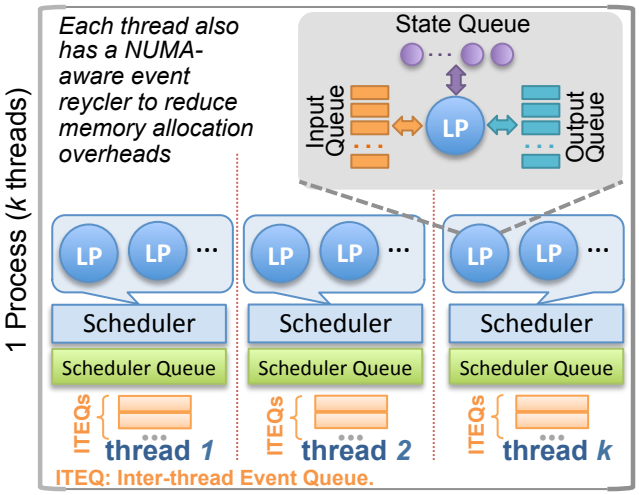


Figure 6: Overview of multithreaded PDES

4.1 Inter-thread Event Queues (ITEQs)

In our framework, all inter-thread communication is accomplished via exchange of *conventional pointers* (we currently do not use `std::shared_ptr`, `std::unique_ptr` etc.) through one or more Inter-thread Event Queues (ITEQs). In our design, the ITEQs of one thread are accessible by any other thread. Given k ITEQs per thread,

when an event e (or kernel message) is to be sent from thread t_a to t_b, an ITEQ $iteq_i (0 \leq i < k)$ in the destination thread t_b is chosen via bit-wise-AND (&) operation – *i.e.,* $iteq_i$ = `recvr`(e) & $\text{ITEQ}_{\text{mask}}$, where $\text{ITEQ}_{\text{mask}} = 2^{\lfloor log_2 k \rfloor} - 1$ and `recvr`(e) is the destination LP's unique ID (an integer value). In other words, $\text{ITEQ}_{\text{mask}}$ is the nearest, lower power of 2 minus 1 so that all of its least-significant-bits set to 1. The mask covers the full set of ITEQs when k is an integral power of 2, but a smaller subset in other cases. We use bit-wise-AND over modulo to ensure maximum performance. Note that, in our design, the selection of ITEQ is based on the destination (or receiving) LP's ID because it generally varies the most between consecutive events exchanged between pairs of threads. The variation helps to distribute events between different ITEQs to further minimize thread contention.

Since ITEQs are the only source of contention between threads, we have assessed the following two different types of ITEQs to ensure a performant implementation:

❶ Lock-based ITEQ: As the name suggested, lock-based ITEQ use standard `std::mutex` (from C++11) to serialize concurrent operations on it. Each ITEQ is just a standard `std::vector` which contains a list of conventional pointers to the queued events. Locking/unlocking of the mutex is performed at individual ITEQ level to ensure finer-grained locking. Events are always appended to the queue and involve locking/unlocking the mutex. However, dequeue operations are performed in bulk – *i.e.,* all queued events are removed as a batch to reduce mutex overheads. The dequeue events are moved into a temporary list that is used for further thread-local processing.

❷ Lock-free ITEQ: Lock free data structures take advantage of special atomic instructions to enable concurrent, thread-safe operations without any direct interaction with the operating system. Lock-free operations are guaranteed to finish in a finite number of steps, with some operations taking longer depending on contention. We have used `boost::lockfree::queue` from the BOOST C++ library for implementing lock-free ITEQ. In order to fully realize its efficiency, we have used a fixed size queue. In BOOST, fixed size queues are implemented using arrays rather than linked lists thereby improving cache performance. We have used a fixed size of 2048 entries. This value is a balance between memory use (as number of ITEQs grow) versus cost of retries if queue is full. In our experiments, this limit was seldom reached. Unlike bulk operation in the lock-based version, in lock-free ITEQs entries are dequeued one at a time as necessitated by its lock-free implementation.

4.1.1 ITEQ processing periodicity. Each thread periodically processes events in its ITEQs at the end of processing a batch of events for each LP, similar to our MPI-based implementation. In our experiments, varying the periodicity did not have a statistically significant impact on simulation performance. Consequently, we have used a periodicity of 1 in our experiments – *i.e.,* poll for incoming events after each LP completes processing events at a given virtual time.

4.1.2 Choice and number of ITEQs. The number of ITEQs per thread (*i.e., k*) is a balance between thread contention and overheads of processing each ITEQ. Fewer ITEQs increase contention but reduce iterations required to process each queue. This subsection

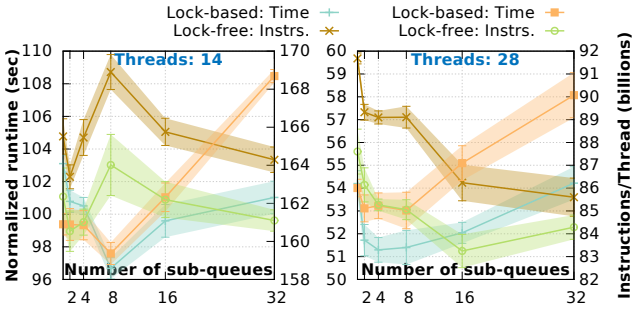


Figure 7: Comparison of lock-based vs. lock-free Inter-thread Event Queues (ITEQs)

discusses experiments conducted to identify effective choice of ITEQs and the value for k, *i.e.,* the number of ITEQs per thread. The experiments have been conducted using `Config #2` (weak scaling mode) of PHOLD, discussed in Section 3.1.2, with 50% of the events being exchanged between threads. Full set of charts is included in supplements.

The charts in Figure 7 provide a comparison of normalized runtime for different simulation configurations with lock-based and lock-free ITEQs (additional charts in supplements). The 14 thread runs used only one CPU (14 cores/CPU) while 28 thread runs used two CPUs. The charts in Figure 7 show that the lock-based queues generally outperform their lock-free counterparts. In all cases, the net number of events (including anti-messages) exchanged between threads was comparable. The net number of events slightly increased by about 6% with increase in threadsdue to 10% increase in rollbacks (charts in supplements). Nevertheless, the poor performance of lock-free queues was counter-intuitive to the putative understanding of lock-free implementation in the scientific community. Consequently, we conducted more detailed analysis using Linux `perf` profiler.

The charts in Figure 7 also provide a comparison of the total number of instructions executed by each thread in the different configurations. Perplexingly, the lock-free implementation generally executes fewer instructions than its lock-based counterpart and is yet slower. The lower number of instructions is expected because lock-free operations are accomplished in user-space using special instructions. In contrast, lock-based operations require interaction with the operating system, thereby generally requiring more instructions. Further analysis of the profiler data showed that the root cause in degraded performance of lock-free queues arises from from two key factors summarized in Figure 8 – ① the number of committed Instructions per CPU-clock cycle (Instr./cycle) is degraded for lock-free queues – *i.e.,* 0.7± 0.16 (lock-based) vs. 0.0.65±0.18 (lock-free) or 7.7% degradation. This degradation is attributed to the atomic instructions which require additional coordination between CPU cores. The degradation in Instr./cycle is most pronounced with 28 threads at 11% (*cf.,* Figure 8). ② A small but consistent degradation was observed in Last Level Cache (LLC) hits reported by Linux `perf` as shown in Figure 8. The slight increase in cache misses is expected with atomic instructions as CPU has to maintain cache coherence across the cores. However, the cache miss rate is small

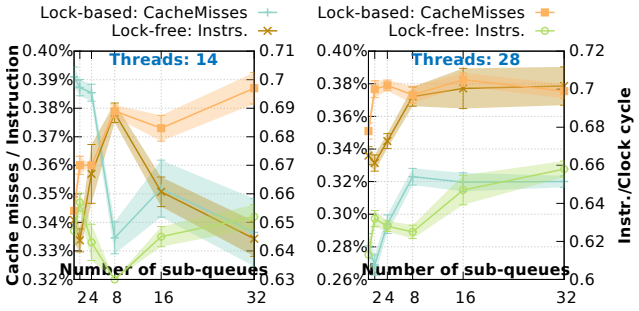


Figure 8: Lock-based vs. lock-free ITEQs: Instrctions/CPU-cycle & cache misses

and does not have a significant impact as illustrated by the 8 and 14 thread configuration in which cache miss rates were comparable and yet the runtimes are slower due to degradation in Instr./cycle.

In summary, these experimental analysis enabled us to identify the following settings for Inter-thread Event Queues (ITEQs) for further empirical analysis: ❶ the lock-based ITEQs are a better design alternative as they provide consistently better performance than the lock-free queues in a broad range of settings; and ❷ Lowest runtimes were generally observed when the number of ITEQs (k) was half the number of threads – *i.e.,* $k = \frac{n}{2}$, where n is number of threads.

4.2 Shared vs. non-shared events

Our multithreaded infrastructure supports two different strategies for exchanging events (as conventional pointers) between threads – ① non-shared mode where copies are exchanged for all inter-thread events, and ② shared mode in which a single copy of an event is shared by two threads. These two modes are primarily a tradeoff between overheads of an extra copy versus the overheads of releasing the event during event cancellation or garbage collection. Freed events are recycled to minimize memory management overheads as discussed in Section 4.3. Furthermore, these two approaches also impact CPU-cache performance.

Non-shared events incur copy overheads but releasing the event for recycling is straightforward and is accomplished using a simple reference counter (a `byte`) in each event. A simple reference counter is necessary and sufficient because because an event is stored locally on a thread. Consequently, only one thread ever modifies the reference counter streamlining recycling of events.

On the other hand, shared events do not have copy overheads but incur some garbage collection overheads. In the case of shared events, two separate reference counters are maintained per event, one for the sending thread and another one for the receiving thread. The dual-counter design eliminates concerns of race conditions and thread-safe operations. However, when one thread releases an event, it cannot be immediately recycled because another thread may still be holding the pointer. Consequently, with shared-events, GVT-based garbage collection requires two phases. First, when the reference counter for the sending process reaches zero, the events are not immediately recycled but stored in a temporary list. This list is processed in subsequent garbage collection cycles and only events where both counters are zero are recycled. The intermediate list can be large (millions of event pointers) depending on number of inter-thread events. Therefore, processing this list adds garbage collection overheads that are absent in the non-shared mode.

4.3 NUMA-aware Memory Management

Non-Uniform Memory Access (NUMA) is the most commonly architecture for high density compute nodes. The compute nodes used for experiments in this paper also have a NUMA configuration as summarized in Figure 1. NUMA-aware Memory Management (NMM) has shown to improve performance in several multithreaded simulation studies [5, 11]. Consequently, we have included NMM for events in our multithreaded simulation framework. The NMM layer can be enabled or disabled at runtime and can be used with both shared or non-shared event modes (*cf.,* Section 4.2). Our NMM is designed as a static object with thread local storage – *i.e.,* each thread gets its own unique, global instance of the NMM. Consequently, thread contention or synchronization is not an issue. Operations of this layer is transparent to the model. The NMM layer provides two key functionalities. It enables NUMA-aware recycling of events to minimize memory management overheads. In addition, it also acts as a lower-level memory management layer as discussed in the following subsections.

4.3.1 NUMA-aware memory recycling. When NMM is enabled, memory for events is allocated on the NUMA-node associated with the destination thread where the brunt of event processing occurs. Each event allocation is routed to the NMM layer which first attempts to recycle previously freed events allocated on the destination NUMA-node. The event recycling infrastructure is relatively straightforward. For each NUMA-node, a hash map called `RecycleMemMap` of free events is maintained. The key into the `RecycleMemMap` is the size (in bytes) requested and each entry contains a list (implemented using `std::stack`) of memory chunks that can be reused. A stack is used here to increase probability of effective cache use. If a given size entry is found in the `RecycleMemMap` it is popped from the stack and returned. Otherwise, a request for memory is dispatched to the lower-level NUMA-memory layer. In order to enable correct recycling upon deallocation, the recycler reserves the first 4 bytes to store the NUMA-node number (an `int`). Then a memory aligned (alignment of events and messages is at 8 byte boundary) pointer is returned. Conversely, deallocation results in the freed chunk being pushed onto the appropriate stack in the `RecycleMemMap`, using the NUMA-node number stored just before the event. Note that all memory management operations occur using conventional pointers and C++'s in-place `new` operator.

4.3.2 Lower level NUMA memory management. The lower-level memory management operation is analogous to custom memory management accomplished by `operator new` in C++ – the memory manager allocates large, fixed-size blocks from which it allocates smaller chunks of varying sizes. The memory manager allocates large, fixed-size blocks of memory via call to `numa_alloc_onnode` library call. The block size is currently set to 64 KiB, but configurable at compile time. The memory manager tracks the blocks on various NUMA nodes to eventually free the memory blocks at the end of simulation (via calls to `numa_free`). Upon receiving a request from

the event recycling layer, it returns the next chunk of memory. If sufficient memory is unavailable, then a new block of NUMA memory is allocated and returned.

4.3.3 Rebalancing of recycled memory chunks. With our NUMA-aware memory recycling strategy we observed significant imbalances in recycler performance based on simulation scenario. In several cases, the recycler hit rate (*i.e.,* ratio of successful reuse of events) would degrade from the desirable 90% or higher hit rate down to about 50%. The imbalance also causes memory growth which in long running simulations would eventually result in memory exhaustion. The source of the imbalance arises primarily from event interaction patterns and such imbalances have also been reported by prior investigations [11].

Accordingly, our NMM checks and redistributes recycled memory chunks to rebalance memory usage across the threads. Redistribution is triggered at end of GVT-based garbage collection and only when the unused recycled memory is 2× times greater than the memory actually allocated by that thread. The extra unused chunks are evenly redistributed to all of the other threads. Redistribution of events is a necessary aspect of NMM without which long running simulations would experience memory exhaustion [11]. As a side effect, it also enables to maintain high recycler hit rates and thereby reducing overall NMM overheads.

5 RELATED WORK

A key aspect of this paper relates to design and assessment of shared-memory multithreaded approaches for optimistic, Parallel Discrete Event Simulation (PDES). The use of shared-memory approaches for PDES have a rich and long running history, for both optimistic and conservative PDES, since early 1990s. Nevertheless, due to space constraints, this section focuses on more recent, closely related optimistic PDES investigations, while referring readers to references therein.

Chen *et al* [1] propose a global schedule mechanism based on distributed event queues to improve performance of shared-memory, multithreaded Time Warp PDES. Our multithreading design also uses distributed event queues, *i.e.,* one per thread. However, we do not use a global schedule mechanism and also account for NUMA in our memory management design. Dickman *et al* [2] explore the effectiveness of single versus multiple scheduler queues for multithreaded optimistic PDES. Their single vs. multiple Least-TimeStamp-First (LTSF) queues is comparable in design to the Inter-Thread Event Queues (ITEQ) used for exchanging events. However, in our design each thread has a single thread-local scheduler queue – *i.e.,* the ITEQs are not the scheduler queues. Even so, consistent with our experimental results, Dickman *et al* also conclude that using multiple queues improve performance. In a follow-up work to Dickman *et al*, Gupta *et al* [3] explore the use of lock-free queues for `bottom` of the Ladder Queue used for managing pending events. They report about 20%–30% performance improvement. Gupta *et al*'s use of lock-free queues for ITEQs is similar to our lock-free ITEQ implementation.

Wang *et al* [11] explore issues of enabling effective, multithreaded PDES and show performance improvements of 3× on Core *i7*, 1.4× on AMD Mangy-Cours (4 CPUs, 12 cores/CPU) , and 2.8× on the Tilera Tile64. Similar to their work, this paper also explores single and multiple CPU configurations, explores NUMA-aware memory management, and effectiveness of multiple inter-thread queues. They compare against MPI-based ROSS simulator that uses a custom shared-memory inter-process message queue. In contrast, we compare against a more recent Cross Memory Attach (CMA) capable MPI implementation, albeit on one hardware platform. In addition, we also explore the effectiveness of lock-free implementation for ITEQs.

Vitali *et al* [10] explore the effectiveness of dynamically reassigning CPU-cores to different simulation-kernel threads to maximize performance. They propose a split design with the top-half focusing on per-LP operation while bottom-half managing inter-LP operations. Our design is monolithic with tight coupling between LP and the simulation-kernel. Pellegrini *et al* [5] propose a Linux-based NUMA-allocator that allows management of per-LP memory consisting of disjoint sets of pages while supporting both static and dynamic bindings. In contrast, our NUMA-aware Memory Management (NMM) layer operates purely in user-space. Recently, Pellegrini *et al* [6] propose fine-grained preemption and dynamic scheduling of high priority tasks to improve performance of multi-threaded PDES. In our design LP operations are not preempted and currently we uses polling to process incoming messages. However, in our experiments we found that 90% of the time we had incoming messages to process, due to the low latency of both CMA-capable MPI and multithreaded ITEQs.

Importantly, it would be remiss not to stress that, similar to this research, every one of the aforementioned investigations also used PHOLD for assessments.

6 EXPERIMENTS & DISCUSSIONS

The experiments in this study focus on comparing the performance of CMA-capable MPI versus comparable multithreading solutions. The objective is to identify better of the two approaches, in terms of performance, so as to inform design choices for optimistic parallel simulations from a more generic context. In our analyses, the runtime characteristics of the CMA-capable MPI implementation has been used as the reference. Our multithreaded framework involves two major design alternatives, namely: use of shared events discussed in Section 4.2 and use of NUMA-aware Memory Management (NMM) discussed in Section 4.3. Accordingly, comparisons with multithreaded simulations has been conducted using the following four configurations: ❶ `Shr.Evt+NUMA`: Shared Events with NUMA-aware Memory Management (NMM), ❷ `Shr.Evt/No NUMA`: Shared Events without NMM, ❸ `No Shr.Evt/NUMA`: Shared events are not used and copies of events are exchanged between threads. However, this mode uses NMM, and ❹ `No Shr.Evt/No NUMA`: In this mode the use of shared events and the NMM are disabled.

All of the experiments in this section have the following common settings to minimize variables and streamline further analyses:

- Each configuration was run on a dedicated node, even if it did not utilize all of the resources. For example, a 2-thread run was conducted on a compute node with all 28 cores reserved. The ten replications for each configuration were run on the same node (one after another and not simultaneously). This setup was used to enable full utilization of caches and to minimize issues with turbo boost side effects.

- Runs with 14 or fewer cores/threads were conducted on a single CPU. Threads were pinned to the CPUs using Linux's `numactl` tool. Similarly process affinity was enforced for MPI-processes using `--cpu-set` feature available in Open MPI.
- All of the configurations were run with Linux `perf` to record CPU usage characteristics (account for Turbo Boost as discussed in Section 2.1) and CPU-cache performance.
- The MPI runs used event recycling and Adaptive Time Window (ATW) as discussed in Section 3.
- All of the multithreading runs used event recycling (immaterial of NUMA-awareness) and Adaptive Time Window (ATW). The number of Inter-Thread Event Queues (ITEQs) was set of half the number of threads, based on the calibration results discussed in Section 4.1.2.

The experiments have been conducted using PHOLD benchmark with 10,000 LPs, each generating 20 (pending event set of 200,000 events) with exponential distribution (λ=10) of time stamps values. As elaborated in Section 3.1, two different communication configurations has been used, namely: ① Config #1: Fixed Inter-LP interactions (strong scaling) and ② Config #2: Fixed fraction of inter-process events (weak scaling). Results from the experiments for these two configurations are discussed in the following subsections.

6.1 Effect of Adaptive Time Window (ATW) with Multithreading

First, we have assessed the effect of using our GVT-based ATW algorithm, discussed in Section 3.3, using the experimental procedure discussed in Section 3.3.1. The charts in Figure 9 provide a comparison of runtimes of the 4 multithreading configurations with and without ATW. For example, the purple curve (—●—) shows percentage difference between simulations with ATW versus without ATW, when using shared events and NUMA-aware Memory Management (NMM). Data points below the zero-axis correspond to results in which ATW performed better. With multithreading, overall the ATW provided performance improvements of up to 10%. In some cases it was a 2% slower which we conjecture was due to throttling of optimism. Nevertheless, since the ATW overall improved performance, we have consistently used ATW in our subsequent experiments, in an identical manner to our MPI-based simulations.

6.2 Config #1: Fixed Inter-LP interactions

The charts in Figure 10 illustrate a comparison of the observed runtime for CMA-capable MPI version versus the 4 different multi-threaded configurations. The charts show mean and 95% CI from 10 replications. The inset charts show raw percentage difference in runtime with respect to the MPI version. In the inset charts, data points above the zero-axis correspond to configurations in which the given multithreaded configuration outperformed the MPI version. As illustrated by the charts in the figure, among the 4 multithreaded configurations, the `No Shr.Evt/No NUMA` configuration (—▲—) is generally the slowest. This is to be expected because of additional event copies as well as NUMA overheads. As expected, the NUMA-aware Memory Manager (NMM) layer reduces the overhead (*cf.*, —◆—), particularly when more than one CPU is used.

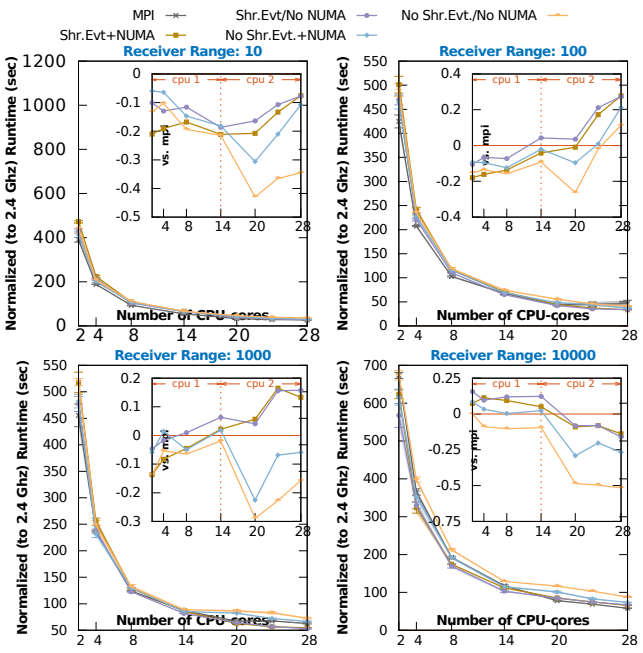


Figure 10: Config #1: Comparison of normalized runtimes. Inset charts show percentage difference versus MPI, with positive values indicating speed-up

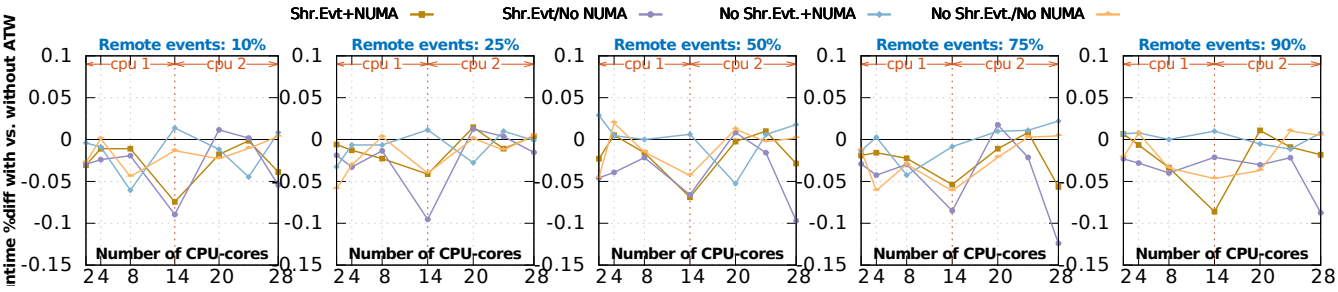
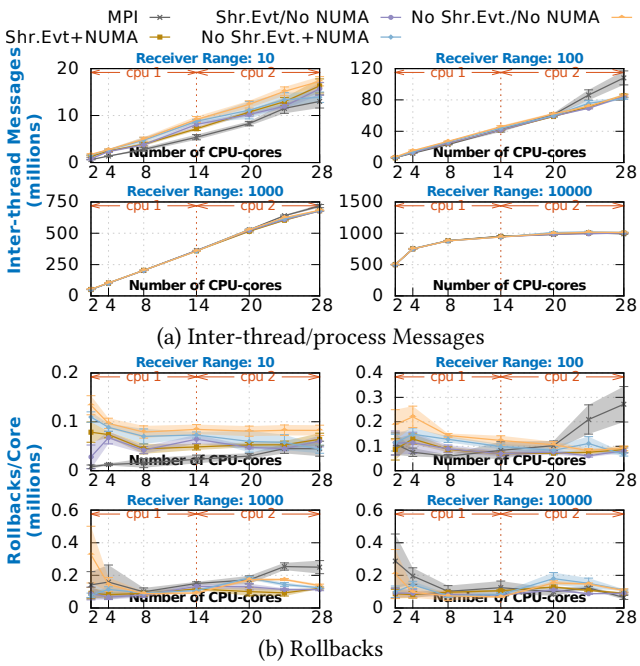


Figure 9: Multithreaded performance comparison of PHOLD runtimes (Config #2 in Section 3.1.2). Data points below the zero-axis correspond to results in which ATW performed better.



(a) Inter-thread/process Messages

(b) Rollbacks

Figure 11: Config #1: Messages & Rollbacks.

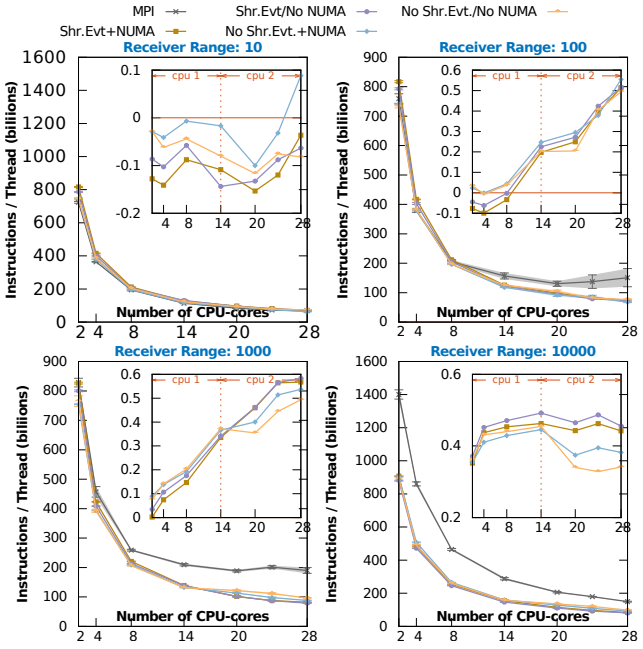


Figure 12: Config #1: Instructions executed. Inset charts shows percentage difference versus MPI, with larger positive values corresponding to fewer instructions.

The shared-events configurations, `Shr.Evt+NUMA` (—●—) and `Shr.Evt/No NUMA` (—■—) outperform the non-shared versions. This is expected because sharing events eliminates overheads of creating an extra copy of the events. However, the `Shr.Evt+NUMA` configuration with NUMA-awareness slightly under-performs the `Shr.Evt/No NUMA` configuration, particularly in single CPU settings by about 5%. The performance degradation is proportional to the slightly increased increased number of rollbacks experienced with `Shr.Evt+NUMA` as shown in Figure 11. However, the slight increase in rollbacks causes the net number of instructions (due to rollback recovery operations) to increase as illustrated by Figure 12. The inset charts in Figure 12 show raw percentage difference in number of instructions for crossvalidation. In these simulations, rebalancing operations used by NMM to redistribute unused memory (discussed in Section 4.3.3) that could impact performance, did not occur.

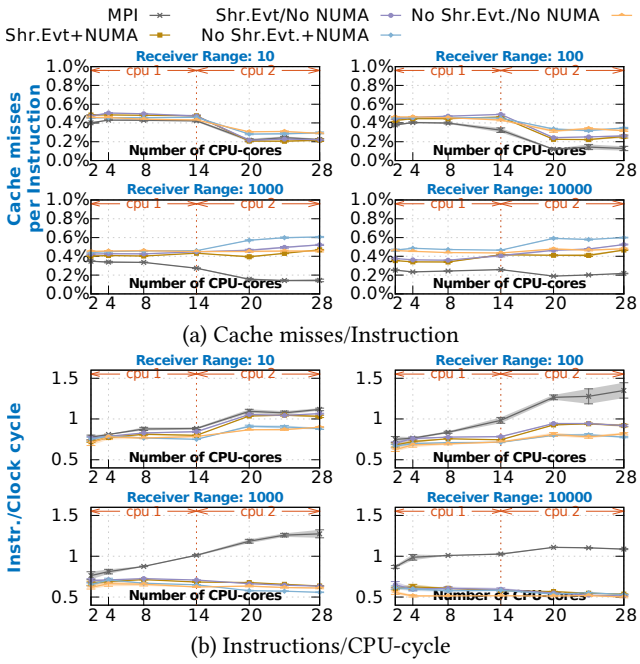
The runtime performance of CMA-capable MPI based runs (—×—) are competitive with the multithreaded version in many instances as illustrated in Figure 12. The "zero copy" capabilities of CMA-based MPI generally outperforms the `No Shr.Evt.` configuration as it eliminates the need to copy messages. In the lowest Inter-Process Communication (IPC) communication scenario with `receiver range` set to 10, the MPI-version experiences fewer rollbacks than the threaded version (*cf.,* Figure 11(b)) and consequently outperforms all of the threaded runs. However, with the `receiver range` setting of 100, the MPI version experienced a conspicuous increase in number of rollbacks as shown in Figure 11(b) which degraded overall performance. Interestingly, in these high rollback scenarios, the superscalar capabilities of the CPUs were effectively utilized as illustrated by Figure 13. The number of instructions retired per CPU-clock cycle more than doubled from about 0.6 to about 1.4 instructions/cycle. We conjecture that this improvement occurs because rollback recovery operates on consecutive events in batches, which improves speculative execution. Full set of charts are included in supplementary materials.

6.3 Config #2: Fixed fraction of remote events

Recollect that in this configuration the fraction of inter-process / inter-thread events called *remote* events is fixed as discussed in Section 3.1.2. The total number of remote events at `remote event` settings of 10%, 25%, 50%, 75%, and 90% settings were on average about 100.6 million, 251.7 million, 507.1 million, 762.8 million, and 917 million respectively (charts in supplements). These averages also include additional anti-messages exchanged during rollback recovery and show a slightly increasing trend, because probability of rollbacks increases with increased communication.

The charts in Figure 14 illustrate a comparison of the observed runtime for CMA-capable MPI version versus the 4 different multithreaded configurations. The inset charts show raw percentage difference in runtime with respect to the CMA-capable MPI version (—×—) . In the inset charts, data points above the zero-axis correspond to configurations in which the given multithreaded configuration outperformed the MPI version. The charts in Figure 15 shows the corresponding number of rollbacks. All plots curves show mean and 95% CI from 10 replications for each data point.

As illustrated by the charts in the figure, the performance landscape is similar to those from `Config #1`. The shared event configurations perform better because extra copies of events are not created. However, between `Shr.Evt+NUMA` (—●—) and `Shr.Evt/No`



(a) Cache misses/Instruction

(b) Instructions/CPU-cycle

Figure 13: Config #1: Cache miss & instructions per cycle

NUMA (—■—), the NUMA-aware Memory Management (NMM) configuration has slightly lower performance. The performance gap arises due to two reasons. First, the NMM has one additional look-up overhead – *i.e.,* the NMM has to look-up the NUMA node ID corresponding to the receiving LP's ID in a hash map. The NMM also has a minor overhead when recycling events as it has to track the NUMA ID. These operations add 90 instructions/event on average, which is a 5% increase. Second source of performance degradation is attributed to the slightly higher number of rollbacks experienced by Shr.Evt+ NUMA (—●—) as illustrated by Figure 15. However, the performance gap between Shr.Evt+NUMA (—●—) and Shr.Evt/No NUMA (—■—) diminishes when cores on the second CPU are used – *i.e.,* core >14 in Figure 14 and Figure 15. In these configurations, the advantages of NMM outweigh its overheads.

Similar to Config #1 results (see Figure 10), the charts in Figure 14 also highlight the complex performance landscape when compared with CMA-capable MPI-based runs. In many cases, the MPI-based runs outperformed the corresponding fastest multithreaded runs by up to 20%. The performance trends are comparable to the trends in number of rollbacks shown in Figure 15. Interestingly, with 75% and 90% remote events, where number of rollbacks are comparable, MPI-based runs performed better with increasing number of cores. Profiler data showed that CPU-cache performances of MPI and the 2 shared-event configurations were comparable (charts in supplements).

However, the number Instructions per CPU-clock cycle (Instr./cycle) was much higher for MPI-based runs than for the multithreading runs, similar to Figure 13(b). We conjecture that the reduced Instr./cycle for multithreading arises from synchronization overheads. Recollect that threads synchronize only to add/remove events from the several shared Inter-Thread Event Queues (ITEQs). Even though having several ITEQs decreases contention, even the short synchronizations negatively impact speculative execution, decreasing Instr./cycle and thereby degrading performance. Nevertheless, overall the complex performance landscape poses a challenge in identifying a clear winning configuration. The results suggest that further analysis on model characteristics and model behaviors would be needed to choose between these alternatives.

7 CONCLUSIONS

The recent, steady trend towards increased CPU-core densities in compute nodes has stimulated investigations in using shared-memory, multithreaded approach over message-passing alternatives for Parallel Discrete Event Simulations (PDES). One of the primary advantages of of shared-memory approaches is that they provide opportunities to reduce communication and synchronization overheads. Motivated by the current research trends, we have significantly redesigned our MPI-based simulation framework to operate using multiple threads. This paper discussed details of our multithreaded framework, its decentralized design, implementation, and assessment.

In our decentralized scheduler design, inter-thread interactions were accomplished using one or more Inter-Thread Event Queues (ITEQs). Two alternative implementations for ITEQs was proposed and evaluated, namely: lock-based versus lock-free. Experimental results showed that despite its novelty, the lock-free implementation underperformed the lock-based implementation. Profiler data showed a key source of 7% performance degradation was due to reduction in instructions per CPU-clock cycle (Instr./cycle) for the lock-free implementation. The observation indicates that maintaining a higher Instr./cycle is more beneficial, suggesting that a spin-lock is a viable candidate for future assessments. The experimental data showed that given k threads, $\frac{k}{2}$ ITEQs yielded a good performance with lock-based ITEQs. This setting provides a good balance between thread contention versus overheads of processing multiple queues.

This paper also discussed and assessed two key design alternatives used in our multithreaded implementation, namely: shared vs. non-shared events and NUMA vs. non-NUMA memory management. Combinations of these alternatives were assessed using a wide range of strong-scaling and weak-scaling configuration of PHOLD benchmark. The simulations were fine grained (time-per-event as < 0.7 µsec), thereby emphasizing communication latencies. Results from our experiments show that overall shared-events setting in which events are shared between threads consistently perform better than the non-shared mode. Shared-events mode performs better because it eliminates the need to make copies of events. The NUMA versus non-NUMA modes had a mixed result based on configuration. When just a single CPU was used, shared-event with non-NUMA memory management performed better. On the other hand, NUMA-aware Memory Management (NMM) performed slightly better with 2 CPUs. However, there is room for fine tuning our NMM – the current implementation of our NMM includes one hash map look-up to determine the NUMA node, given an LP's ID. Reducing overheads of this look-up can further improve effectiveness of our NMM.

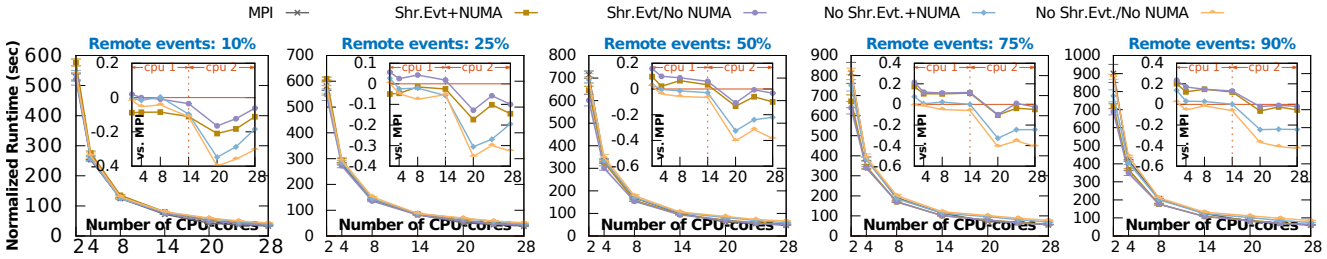


Figure 14: Config #2: Comparison of normalized runtimes. Inset charts show percentage difference versus MPI, with positive values indicating speed-up

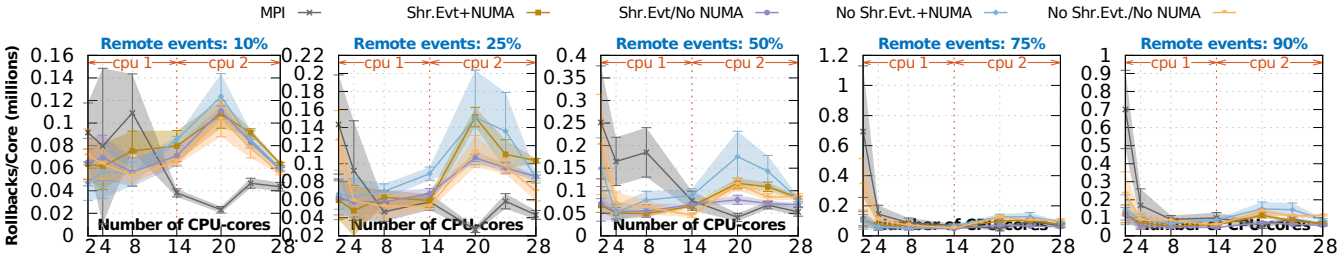


Figure 15: Config #2: Comparison of rollbacks corresponding to runs in Figure 14.

The performance of our multithreaded PDES framework was compared against our existing MPI-based, distributed memory alternative. However, a key aspect taken into account is the Cross Memory Attach (CMA) based capability introduced in the Linux kernel and Open MPI. CMA permits MPI-processes to directly read/write data to a different process's virtual memory space, thereby lowering the latency for exchanging messages. Our experimental analysis revealed a complex performance landscape with no clear winner – *i.e.,* the multithreaded, shared-memory approach performed better only in some cases when compared to the message-passing approach. With our weak-scaling benchmarks, the MPI-based version consistently outperformed the multithreaded version when 2 CPUs were used.

The complex performance landscape suggests that a split design could be beneficial – *i.e.,* multiple threads on a single CPU and message-passing for inter-CPU interactions. Of course, we plan to explore this design in our future work. More importantly, the complex performance landscape suggests that a careful assessment of "influential" model characteristics needs to considered to choose between multithreading versus contemporary CMA-capable message-passing solutions. This requires identification and ranking of model characteristics to determine the most influential ones, which also has considerable potential for future research.

Supplementary Material

Source code for MUSE and supplementary material available online at http://pc2lab.cec.miamiOH.edu/muse/

Acknowledgments

Support for this work was provided in part by the Ohio Supercomputer Center (Grant: PMIU0110-2).

REFERENCES

[1] L. Chen, Y. Lu, Y. Yao, S. Peng, and L. Wu. A well-balanced time warp system on multi-core environments. In *Proceedings of the 2011 IEEE Workshop on Principles of Advanced and Distributed Simulation*, PADS '11, pages 1–9, Washington, DC, USA, 2011. IEEE Computer Society.

[2] T. Dickman, S. Gupta, and P. A. Wilsey. Event pool structures for pdes on many-core beowulf clusters. In *Proceedings of ACM SIGSIM PADS*, pages 103–114, New York, NY, USA, 2013. ACM.

[3] S. Gupta and P. A. Wilsey. Lock-free pending event set management in time warp. In *Proceedings of the ACM SIGSIM PADS*, pages 15–26, New York, NY, USA, 2014. ACM.

[4] J. Higiro, M. Gebre, and D. M. Rao. Multi-tier priority queues and 2-tier ladder queue for managing pending events in sequential and optimistic parallel simulations. In *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '17, pages 3–14, New York, NY, USA, 2017. ACM.

[5] A. Pellegrini and F. Quaglia. Numa time warp. In *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM PADS '15, pages 59–70, New York, NY, USA, 2015. ACM.

[6] A. Pellegrini and F. Quaglia. A fine-grain time-sharing time warp system. *ACM Trans. Model. Comput. Simul.*, 27(2):10:1–10:25, May 2017.

[7] D. M. Rao. Efficient parallel simulation of spatially-explicit agent-based epidemiological models. *Journal of Parallel and Distributed Computing*, 93-94:102–119, 2016.

[8] W. T. Tang, R. S. M. Goh, and I. L.-J. Thng. Ladder queue: An O(1) priority queue structure for large-scale discrete event simulation. *ACM Trans. Model. Comput. Simul.*, 15(3):175–204, July 2005.

[9] J. Vienne. Benefits of cross memory attach for mpi libraries on hpc clusters. In *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment*, XSEDE '14, pages 33:1–33:6, New York, NY, USA, 2014. ACM.

[10] R. Vitali, A. Pellegrini, and F. Quaglia. Towards symmetric multi-threaded optimistic simulation kernels. In *Proceedings of the 2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation*, PADS '12, pages 211–220, Washington, DC, USA, 2012. IEEE Computer Society.

[11] J. Wang, D. Jagtap, N. Abu-Ghazaleh, and D. Ponomarev. Parallel discrete event simulation for multi-core systems: Analysis and optimization. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1574–1584, June 2014.