ABSTRACT

THREAD SAFE MULTI-TIER PRIORITY QUEUE FOR MANAGING PENDING EVENTS IN MULTI-THREADED DISCRETE EVENT SIMULATIONS

by Matthew Michael DePero

Parallel Discrete Event Simulation (PDES) conducted using emerging shared memory many-core CPUs presents capacity for even greater performance by 1) eliminating the need for message passing and associated serialization/deserialization overheads, and 2) reducing memory requirements by allowing a single copy of an event to be shared between multiple threads. However, the overall performance of a PDES is highly contingent on the speed and capacity of its pending event set data structure. Accordingly, we present a simple, thread-safe priority queue called 3tSkip for managing pending events. Our design takes advantage of contemporary synchronization primitives, including atomics and lock-free data structures to ensure good performance. The priority queue has been incorporated into a redesigned version of a parallel simulator called MUSE, to enable PDES on shared memory platforms. The effectiveness of the proposed solution has been assessed using standard PDES benchmarks. Our analysis identifies many critical design obstacles to multi-threaded design and presents novel solutions to those design obstacles. Our solution achieves significant speedup in high granularity scenarios, when compared to existing MUSE simulator, though more work is required before multithreaded design becomes effective in a broad range of scenarios.

THREAD SAFE MULTI-TIER PRIORITY QUEUE FOR MANAGING PENDING EVENTS IN MULTI-THREADED DISCRETE EVENT SIMULATIONS

Thesis

Submitted to the

Faculty of Miami University

in partial fulfillment of

the requirements for the degree of

Master of Science in Computer Science

by

Matthew Michael DePero

Miami University

Oxford, Ohio

2018

Advisor: Dr. Dhananjai Rao

Reader: Dr. Mike Zmuda

Reader: Dr. Karen Davis

This thesis titled

THREAD SAFE MULTI-TIER PRIORITY QUEUE FOR MANAGING PENDING EVENTS IN MULTI-THREADED DISCRETE EVENT SIMULATIONS

by

Matthew Michael DePero

has been approved for publication by

College of Engineering and Computing

and

Department of Computer Science and Software Engineering

Dr. Dhananjai Rao

Dr. Mike Zmuda

Dr. Karen Davis

Contents

List of Figures

Acknowledgments

I would like to extend a huge thank you to my adviser Dr. Dhananjai Rao for all of his support and guidance throughout this process. Dr. Rao has always gone above and behind the call of duty to drive success in all students at Miami University, and I will take his lessons and wisdom with me long into my career. I also would like to thank my committee members Dr. Mike Zmuda and Dr. Karen Davis for their invaluable time and feedback, without which this thesis wouldn't be possible.

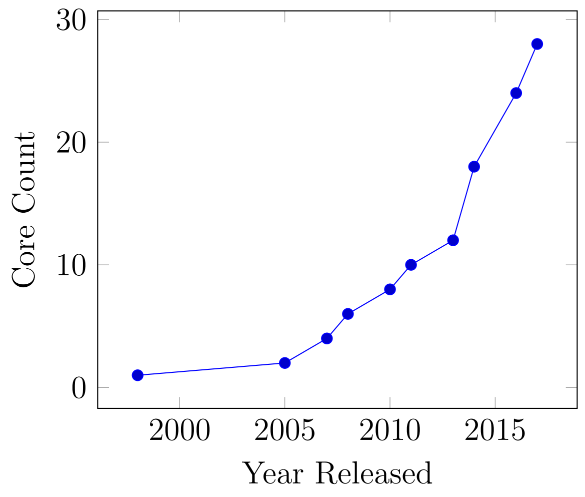
Chapter 1

Introduction

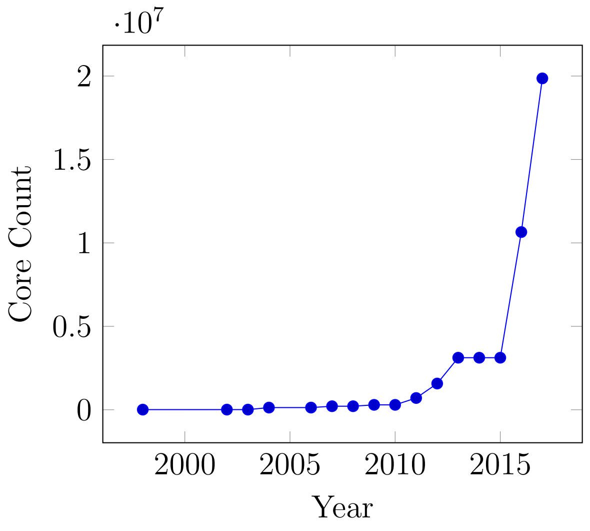
Computer-based simulations are a fundamental scientific tool that are used for analysis and design in a diverse set of domains, including: manufacturing, medicine, VLSI design, data communication networks, and epidemiology just to name a few. Several different methodologies have been proposed for conducting simulations. Amongst the various methods, Discrete Event Simulation (DES) has emerged as a dominant methodology with broad applicability. More importantly, this type of simulation has the potential to effectively utilize parallel computing platforms to enable simulation of large models in a reasonable timeframe, especially in scenarios where time sensitivity is imperative such as predicting the spread of a disease.

DES consists of independent entities, or Logical Processes (LPs), that are simulated and interact with each other by creating timestamped events that are sent to other LPs for processing. [1] Oftentimes, events must be processed in timestamp order to preserve causality in a DES. For example, if LP_1 creates an event e_1 that modifies LP_2 at $t = 2$, LP_2 must receive and process e_1 before it can do any processing for $t > 2$. Accordingly, events are managed using a priority queue, with entry priorities determined by timestamps of events. Typically, event management is handled via a centralized scheduler data structure that maintains the set of pending events to be simulated in chronological order based on simulated time. Due to the interdependent nature of LPs, however, managing the communication and scheduling of events is often a bottleneck on performance. [2]

Management of pending events using priority queues also plays an important part in Parallel Discrete Event Simulation (PDES). Furthermore, PDES also involves additional event management operations to synchronize parallel processes in order to maintain this causality property. Parallel simulations have grown in prevalence due to steady advancements in computing technologies. As shown by Figure 1.1(a) the number of cores per CPU in Intel's high performance family of processors has grown steadily since their inception. Today, even



(a) Core Count of Intel Xeon Family CPU



(b) Largest Core Count of Supercomputer per Year

Figure 1.1: Multi-Core and Many-Core CPU Trends

consumer available multi-core CPUs such as Intel Xeon E7-8894V4 have as many as 24 cores running 48 threads, and servers can pack as many as 4 of these CPUs, providing 192 threads on a single machine. Additionally, Intel recently released its research grade Xeon Phi family of chips in 2017 (formerly Knights Landing), boasting as many as 72 cores on a single chip. This increasing core density has contributed to an exponential growth in the overall core counts in distributed memory supercomputers as well, as noted in Figure 1.1(b).

The advancement and dichotomy in hardware technologies has spurred two distinct approaches for Parallel Discrete Event Simulation (PDES). The two kinds of parallelization are:

1. **Distributed memory parallelism**: that which occurs on separate machines, each with their own independent memory. Interactions between parallel processes running on different machines is accomplished via high speed interconnects such as: 40 GBPS Infiniband or 100 GPBS Omnipath networks via message passing.

2. **Shared memory parallelism**: that which occurs on the same machine with shared memory, where all parallel processes have access to the same main memory. Consequently, data structures can be directly shared and accessed between threads, eliminating the need for communication over interconnects.

Often, a combination of both is used to maximize performance. Distributed memory architecture requires each machine to have its own data structure for event management, requiring synchronization between independent simulation states. On the other hand, multi-core CPUs allow for concurrent threads to use the same shared data structure, removing the need to synchronize state between parallel executions. However, in the worst case, threads will access and mutate this structure at a near constant rate with very little processing in between, a case that can be common to DES. The shared scheduling data structure's ability to handle heavy concurrency, therefore, is critical to the amount of speedup that can be obtained from multi-core parallelization on one machine. Diminishing speedup can become apparent once a given number threads are used for simulation due to this contention overhead, and a trade off exists between concurrency utilization and overhead to maintain the parallel threads. [3]

We examine the scheduling data structure used in DES in the context of the Miami University Simulation Environment (MUSE), detailed in Section 2.3. Specifically, we modify the 3-Tier Heap priority queue developed by Higiro detailed in Section 3.1 to allow safe concurrent operation by multiple threads. [4] A design pattern is currently implemented in MUSE to allow existing sequential structures to achieve this thread-safety on a multi-core platform through a Message Passing Interface (MPI) implementation detailed in Section 4.2, however we examine a new version in which the structure itself is inherently thread-safe with a focus on reducing overheads such as blocking which heavily bottlenecks the benefits of parallelism. We also discuss the design considerations that were taken into account while attempting to convert MUSE into a thread-safe, shared memory parallel simulation kernel.

Chapter 2

Background

2.1 Discrete Event Simulation (DES)

Discrete event simulation is a computational methodology which allows for a wide range of systems to be modeled and analyzed. In DES, the system is divided into independent entities with their own independent states. One of these entities is called a Logical Process (LP) which manages its own state. Accordingly, a DES is essentially designed as a set of logical processes (LPs) that interact with each other. LPs interact by exchanging discrete-timestamped events or messages. [1] Processing an event essentially introduces a change in an LP's state and may cause the LP to generate additional events to itself or other LPs in the model. DES has been used in a variety of fields as a tool to help inform knowledge and to improve decision-making processes. DES provides an effective means for analyzing real or artificial systems without the constraint of limited resources such as time, financial costs, or safety.

Parallel Discrete Event Simulation (PDES) Parallelism increases throughput in discrete event simulations, allowing larger and more complex problems to be solved. Many large scale discrete event simulations require significant processing time to finish. Running a parallel discrete event simulation (PDES) increases the speed and allows them to be more useful in time sensitive scenarios. One common method for constructing a PDES is to organize each parallel unit as set of LPs. The LPs are divided or partitioned to operate on different compute machines. Each LP contains its own state and local virtual time (LVT) representing the virtual time it last processed events. This allows logical processes to communicate with one another with timestamped messages without needing to keep all LPs at the same virtual time across the parallel system. However, event processing on the different compute units must then be synchronized to ensure causally consistent event processing. Consequently, the speedup achieved using multiple compute machines requires efficient strategies to minimize synchronization costs. There are two main ways that these types of parallel discrete event simulations handle synchronization: conservative and optimistic. [1]

Conservative vs. Optimistic Synchronization Conservative synchronization uses the timestamp on events to ensure that causality errors do not to occur during parallel simulation. A strategy is put in place to determine if an event is safe to process; only when all previous events that could possibly affect the event in question have been processed is the current event allowed to be processed. Consider a logical process with event e_1 and time-stamp t_1. If the process can determine that it isn't possible to receive an event with a smaller timestamp $t < t_1$ then it processes e_1, otherwise it waits. Links between logical processes are statically specified, and they hold an associated clock. These clocks on the links allow the process to determine if it is possible to receive a new event past the given timestamp or not. But this type of synchronization can lead to deadlock, meaning that multiple logical processes are waiting on each other to reach a safe state before processing. There are a few methods of handling or avoiding deadlock. Some involve special messages, others simply detect the deadlock and recover from it, and there are many more possible ways to avoid deadlock. Regardless, events are never processed out of order in a conservatively synchronized system.

One of the significant bottlenecks in conservatively synchronized parallel simulation is that the inherent parallelism in a model may not be effectively used. In fact, the need to process a single event can often prevent processing of other pending events, preventing effective parallelism. An example of such a scenario occurs in the parallel simulation of the avian influenza, discussed by Rao and illustrated in Figure 2.1. [5] The green curve tracks the potential parallelism in the model. However, the blue curves track the number of events processed by a conservatively synchronized kernel. The difference between the two curves illustrates the potential parallelism that goes unutilized in a conservatively synchronized simulation.

In optimistic synchronization, events are processed regardless of potential causality error. The system must then detect when a causality error has occurred and correct it. This correction requires the LP to rollback to a previous state before the causality error occurred in virtual time. This design does not require static links, which allows much more simple dynamically allocated logical processes. Optimistic synchronization also takes advantage of parallelism where causality errors have the possibility of occurring but do not.

Conservative synchronization generally works well for problems that are easy to foresee future events and account for them. It requires detailed work to ensure the synchronization

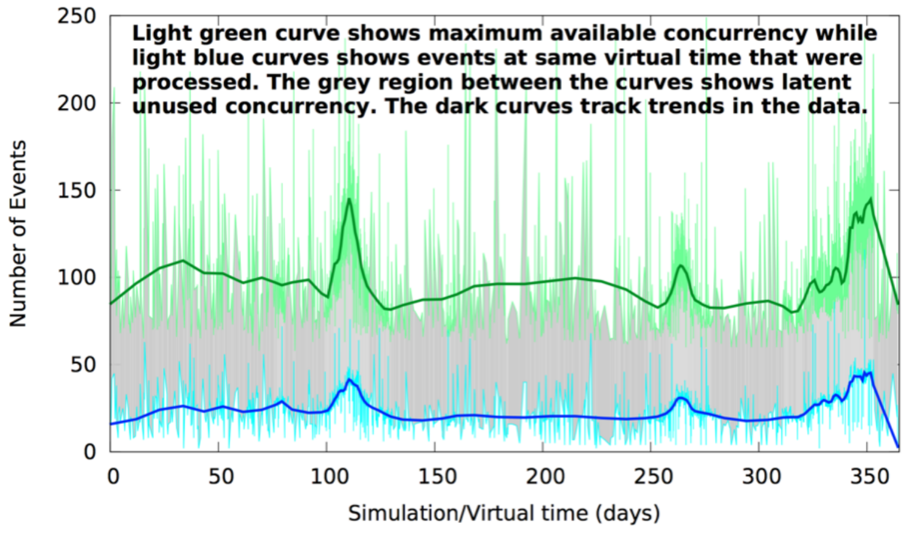


Figure 2.1: Potential vs Utilized Concurrency in Parallel Avian Influenza Simulation [5]

mechanism won't cause deadlock or will always recover when it occurs. On the other hand, optimistic synchronization does not need to foresee future events and exploits parallelism much more effectively. But, rollbacks come with their own overhead and demand increased memory to maintain previous states. Optimistic is typically more complex to implement than conservative synchronization. The most common protocol for rolling back optimistically synchronized simulations is known as Time Warp. [1]

Time Warp In Time Warp, a rollback is triggered when an event is received with a timestamp less than the logical process's local virtual time, meaning the LP has already processed events with timestamps greater than the incoming event, breaking causality. This incoming event is known as a straggler. The LP must be implemented with three queues to use Time Warp: an input, output, and state queue, as well as maintain their own local virtual time. The input queue holds events previously received by the LP, the output queue holds events sent to other LPs, and the state queue holds all previous and current states of the LP. Finally, the logical processes must exchange time stamped events such that the unit of virtual time is consistent across the simulation. [1]

There are three steps required to implement Time Warp, outlined in Figure 2.2. First, when a straggler event is received, the state queue is iterated until it finds the state with a timestamp immediately before the incoming event. Second, the LP is rolled back, changing its current state to the previous state found in the queue. The input queue is then rearranged

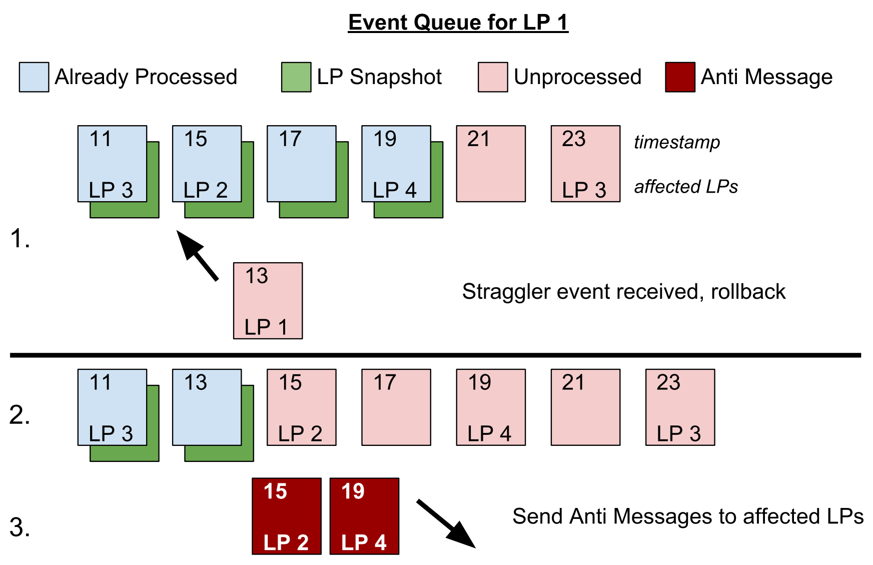


Figure 2.2: Three Steps of Time Warp for Optimistic DES

to add the straggler to the proper location. Finally, the output queue is used to send anti-messages to the other logical processes informing them that previously valid messages were rolled back, which will cause a rollback in all subsequent logical processes. Then execution resumes following the input queue. 2.2.

2.2 Shared Memory Multithreading

Multithreading involves running multiple independent executions of some part of a process in the same memory context. Typically, this involves taking advantage of multi-core CPUs by having these separate executions (threads) run concurrently on separate physical CPU cores. This design provides numerous advantages over sequential programs including increased computing capacity, resilience to latency issues such as I/O operations, ability to access and mutate a shared memory space, and more.

Due to the independent nature of threads, considerations must be made to enable proper synchronization between threads. This cross-thread communication can lead to issues when threads attempt to read or modify the same memory at the same time. For example, if thread t_1 wishes to increment variable $x = 10$ by 1, it must first read the value of x, calculate the new value, and then write the new value to memory. If thread t_2 attempts to execute this increment operation at the same time as t_1, both threads may read $x = 10$, calculate $x = 11$ independently, and write 11 to memory, even though the true result of both operations should lead to $x = 12$. This is called a data race, and it results from threads attempting to modify a shared state between threads in a non-thread-safe way.

Oracle Corporation describes three recognizable levels of thread-safety for a given procedure: [6]

1. **Unsafe**: The procedure is not safe to run concurrently, and unexpected behavior or side effects are possible if multiple threads perform the operation at the same time. This operation can be made Thread Safe - Serializable by introducing a lock/unlock before and after the operation by concurrent threads.

2. **Thread Safe - Serializable**: The procedure is safe to be called by multiple threads running concurrently, however it does not utilize available concurrency due to requiring only one thread to run at a time, usually with a lock or other mechanism that serializes the execution.

3. **Thread Safe - MT-Safe**: The procedure is both safe and utilizes concurrency, thus providing speed up in execution time compared to performing the same operations in a serialized way. The amount of possible concurrency is dependent on how much interdependence exists between the operations being performed on each thread.

Thread Safe - Serializable operations provide thread-safety, but do so by blocking (or waiting) while a conflicting thread finishes its execution, thus forcing only one thread to execute at a time. This blocking can bottleneck performance since any speedup associated with parallel execution of a critical section is lost when a thread must stop and wait for another thread to finish before it can begin. This is referred to as a lock-based approach to thread-safety. *Thread Safe - MT-Safe* operations are preferred as they allow safe execution in a non-blocking way, but require greater considerations to ensure thread-safety.

Figure 2.3 shows a simple example of lock-based vs lock-free thread-safe operation. In Algorithm 1, a thread must acquire a lock, blocking until the lock is acquired, before it continues execution. This implementation is simple, but no parallelization is realized with this approach. Algorithm 2 shows a lock-free example, using a *tryInsert*() method that may fail if there is a conflict with another thread. Importantly, at least one thread will succeed if a conflict arises during the *tryInsert*() if called by two threads at the same time. This approach is more complicated, as *tryInsert* must utilize hardware instructions such as Compare and Swap (CAS) to attempt to modified shared memory, but lock-free operations reduce thread contention and lead to more efficient parallelization.

Two types of these non-blocking concurrent operations exist: lock-free and wait-free. Lock-free operations guarantee system wide progress, meaning any concurrent operation

Algorithm 1 Lock-Based Insert

```
1: function INSERT(v)
2:     sharedGuard.lock()    ▷ blocks until free
3:     queue.insert(v)
4:     sharedGuard.unlock()
5: end function
```

Algorithm 2 Lock-Free Insert

```
1: function INSERT(v)
2:     do
3:         success ← queue.tryInsert(v)
4:     while NOT success
5: end function
```

Figure 2.3: Lock-based vs. Lock-free Insert Pseudocode Example

conducted by two or more threads will always succeed and move forward by at least one thread without blocking, even if some threads fail and must re-try (See Algorithm 2). This differs from lock-based execution, where two threads must compete for some type of shared lock, meaning successful operation of one thread is contingent on the successful completion of another thread holding the lock, leading to blocking. Wait-free is even more stringent, guaranteeing per-thread progress for any operation in a finite number of steps. In other words, a thread is guaranteed to never block or fail an unpredictable number of times on any operation, regardless of the state of other threads. This is the ideal state for any non-blocking algorithm, however it is the most difficult to achieve due to constraints that exist at the hardware level, and can even result in slowdown compared to lock-free due to the overhead of these constraints. [7]

The amount of speedup that can be realized in $Thread\ Safe$ - MT-$Safe$ operations is based on ① the amount of possible concurrency theoretically available to an operation, and ② the amount of that concurrency utilized by implementation. This is formalized in computer architecture by Amdahl's Law, which presents the theoretical limits to speedup in the context of parallelized resources. [8] The equation is given by

$$S_{latency}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

Where $S_{latency}$ is the limit of theoretical speedup possible for the entire task, s is the speedup of the part of the system that is able to take advantage of the parallelization, and p is the portion of original execution time occupied by the part of the system described by s. The ramifications of this concept are demonstrated in Figure 2.4 which shows the theoretical reduction in execution time for two hypothetical processes A and B, each with an original sequential execution time of 100. A has 40% of its overall execution time benefit from parallelization, while B has 60% benefit. With just this difference, a 4x speedup applied

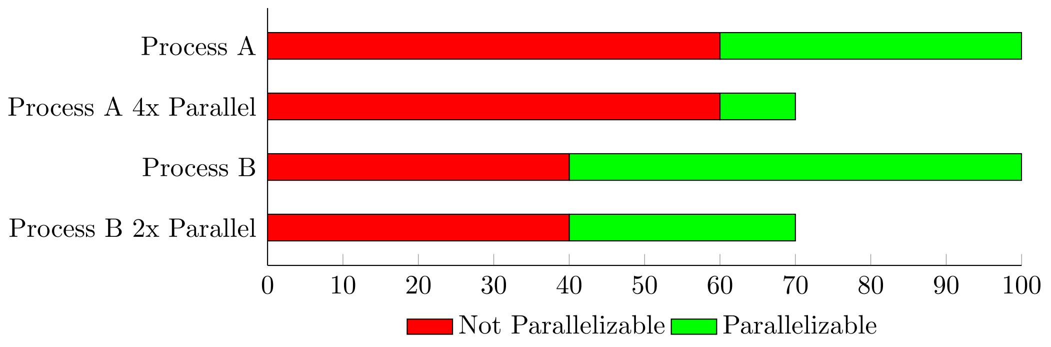


Figure 2.4: Effect of Concurrency Potential on Theoretical Speedup with Amdahl's Law

to A's parallelizable section results in the same overall theoretical execution time as an only 2x speedup applied to B.

This property for performance demonstrates the importance of implementing non-blocking, thread-safe structures and algorithms, which also is what makes multi-threading as a means for performance gain so challenging. Additionally, multi-threading has various overheads, making it even more critical to be as efficient as possible when designing thread-safe systems to ensure maximum utilization is achieved by the multi-threaded design, rather than focusing on speed or number of threads. This is especially difficult on priority queues, where the primary contention point is the same across all threads (the top of the queue), making it difficult to achieve a high degree of concurrency.

2.3 Architectural Overview of MUSE

The system used to implement and test our thread-safe design is a parallel simulation framework called the Miami University Simulation Environment (MUSE). The application was developed as part of a master's thesis by Meseret Gebre in the Department of Computer Science at Miami University in 2009 [9]. MUSE is written in C++ and currently utilizes the Message Passing Interface (MPI) library for parallel processing via multiple independent processes as discussed in Section 2.1. It also uses Time Warp with a standard state saving approach to accomplish optimistic synchronization.

The MUSE simulation kernel implements core functionality associated with LP registration, event processing, state saving, synchronization and Global Virtual Time (GVT) based garbage collection. Each LP in a simulation maintains an input, output and state queue. The input queue is used to retain events that have already been processed but have not yet been garbage collected. The output queue stores potential anti-messages, which are sent to

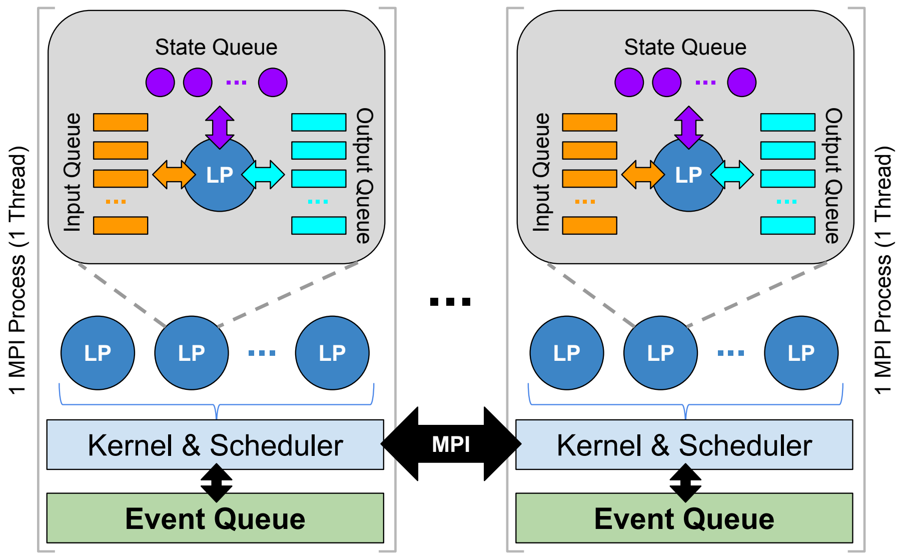


Figure 2.5: Current Architectural Overview of Parallel Simulation in MUSE

other LP in the case of a rollback to cancel out previously sent events. The state queue stores the state of the LP at each discrete point in virtual simulation time. A Time Warp LP also maintains a local virtual time (LVT) that is updated to the time-stamp of the event most recently processed by the LP. [3] An overview of this architecture is shown in Figure 2.5.

High Level Design of MUSE The MUSE core has seven API classes available to the user. These publicly visible classes are used in different ways to get a simulation running with MUSE. MUSE core also has classes not available to the API user. These classes are used by the simulation kernel to help with getting the simulation to schedule agents correctly, synchronize multiple kernels in the distributed simulation and also to communicate across parallel kernels by sending messages/events across the wire. The relationships can be seen in Figure 2.5. [5]

When dealing with logical process based simulations, we clearly need a way to describe our LPs in the simulation. MUSE defines this concept by the Agent class. The Agent class is dependent on the State class. Two classes use this Agent: Simulation and Scheduler. With the singleton instance of the simulation kernel, the agent can be registered and the Simulation kernel will take responsibility of including the agent in the simulation. Once the agent is registered, the kernel will register the agent with the Scheduler which handles organizing events for processing in timestamp order. [5] Note that the Simulation class is also used for setting the begin/end time of the simulation.

When an agent wants to communicate to another agent, it must create an event. The Event class uses data types described in DataTypes header for construction parameters. Scheduling of events is done through the Agent class. The Agent class intelligently decides internally to either pass the work onto the simulation kernel or if the event is to itself, it bypasses the scheduler and automatically adds it to the Event Queue to be processed. [5]

Within the agent, the user can use one of the subclasses of SimStream to perform IO operations. We have developed the oSimStream which handles outputting data to any stream safely. There is a default oSimStream class in the Agent class. This can be used just like using std::cout. [5]

The GVTManager class implements Mattern's GVT algorithm. [1] This works via the root kernel (usually with SimulatorID zero) by circulating a GVTMessage between other parallel kernels. When a message reaches a kernel, the kernel polls the scheduler for the event/agent that will execute next. This timestamp by definition will be the local global virtual time or LGVT. LGVT is the least timestamp of all agents' LVT (local virtual time) being handled by that kernel. It updates the GVTMessage accordingly and passes it to the next kernel in a ring fashion, eventually resulting in the least LGVT across all kernels, which represents the GVT of the overall simulation. [5]

Chapter 3

Related Work

3.1 Pending Event Sets

At the core of all DES is the Pending Event Set (PES) that manages the list of events waiting to be simulated and organized by timestamp. As a result, there exists a breadth of literature offering a range of solutions to the PES, each with their pros and cons in a variety of contexts. One of the original data structures used for this purpose was the Calendar Queue (*CQ*) presented by Brown in 1988. [10] Calendar Queue is structured as an array of buckets each associated with a set span of time (like a desk calendar with indexed months and days of set time). The width, or time span, associated with each bucket must be carefully chosen, but when optimal, *CQ* allows for amortized constant time enqueue of events and constant time retrieval of the minimum timestamp event. A popular variant of *CQ* is the Ladder Queue (*LadderQ*) proposed by Tang, et al. in 2005 which was empirically shown to outperform existing popular structures, including CQ. [11] This is done through the reducing of bucket conflicts when the distribution of event timestamps is non-uniform by allowing buckets to be split dynamically when a large number of events is added to any one bucket. Should a bucket begin filling up, a new rung of the ladder is spawned to allow events in the overflowing bucket to be better organized. How the distribution of time spans and rungs looks in ladder queue can be seen in Figure 3.1.

One limitation of *LadderQ* is evident in optimistic PDES which requires cancellation of events when a causality error is detected (see Section 2.1: Conservative vs. Optimistic Synchronization). The 2-Tier Ladder Queue (*2tLadder*) proposed by Higiro et al. in 2017 provides performance gains on *LadderQ* in the optimistic parallel use case where rollbacks are present. [4] This is done by creating a second tier in each bucket to further organize pending events. While only a single additional constant time operation is necessary to distribute

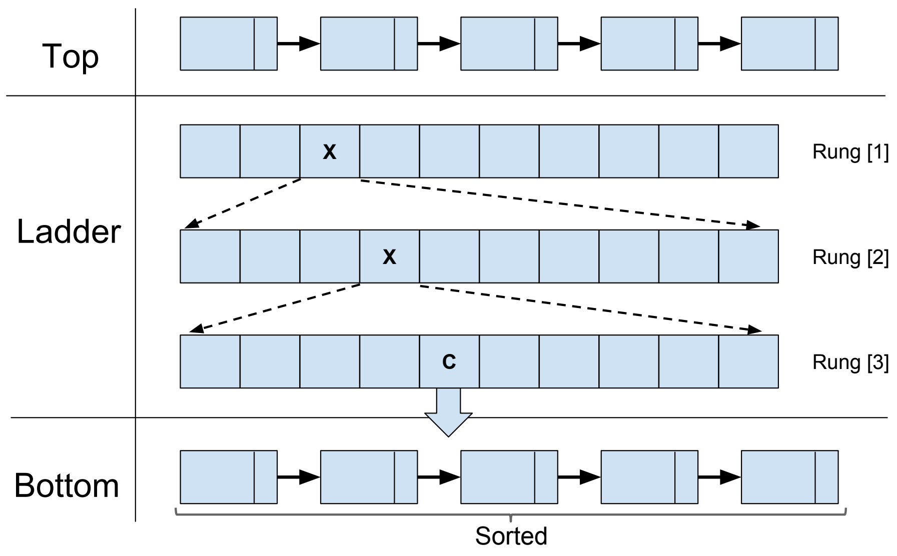


Figure 3.1: Ladder Queue as Described by Tang, et al. [11]

events into this second tier, a rollback has the ability to process cancelled events by only looking at a single sub-bucket, rather than needing to process the entirety of a bucket in the traditional $LadderQ$.

3.1.1 3-Tier Heap

Just as 2-Tier Ladder Queue provides performance gains on Ladder Queue in optimistic simulations, the 3-Tier Heap Queue ($3tHeap$) by Higiro also achieves state of the art performance by distributing events into multiple tiers of heap based queues to increase the speed of enqueue, dequeue, and rollback operations. [4] The structure of $3tHeap$ is shown in Figure 3.2 with the top tier heap sorting agents based on their next available event, the second tier heap sorting events for that agent into individual timestamps, and the third tier vector holding the individual events at each timestamp.

This structure provides numerous benefits. First, it achieves performance gains in most all cases compared to contemporary pending event sets, especially in highly concurrent parallel simulations where rollbacks are frequent. The structure also optimizes very efficiently by using c++ standard library vectors and heaps, both of which are fine tuned to benefit greatly from optimization by the compiler. Additionally, the simplicity of its design makes it relatively easy to implement and extend into refined use cases. As a result, we choose this structure as the foundation for our thread safe pending event set structure as described in

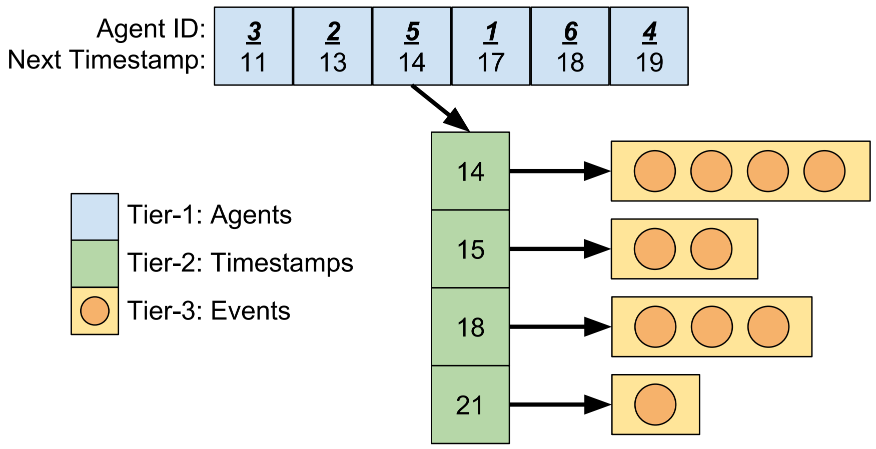


Figure 3.2: Three-Tier Queue Design as Described by Higiro [4]

Section 5.1.2.

3.2 Robustness Analysis using Java Implementation

The key performance differences between these data structures arises from the runtime constants associated with the queues. The runtime constants are influenced by a variety of factors including: hardware, programming language, and compiler optimizations. Consequently, to establish robustness and reproducibility of the performance characteristics, the 3-tier Heap ($3tHeap$) and 2-tier Ladder Queue ($2tLadder$) have been implemented in Java. The fine-tuned version of Ladder Queue ($ladderQ$) has also been implemented in Java for experimental comparison. The difference between the C++ and Java versions arise from the idiomatic implementation styles of the two languages. A key difference from a performance perspective is the use of 32-bit int in Java for ArrayLists versus 64-bit size_t in C++ for std::vector – the 64-bit data types require additional CPU cycles.

A simplified version of the standard PHOLD benchmark used by Tang et al. and Higirio et al. has also been ported to Java. The benchmark is a simplified version of a MUSE sequential simulation and performs the core tasks of scheduling events, dequeuing events, and simulating the processing of events. Note that rollbacks do not occur in sequential simulation. However, the performance differences between $2tLadder$ and $ladderQ$ arise only with rollbacks, with minimal difference in the sequential case. Consequently, we have used runtime profiles of parallel simulations to synthetically characterize the occurrence of rollbacks (based on geometric distribution) and trigger rollback of pending events in these data structures.

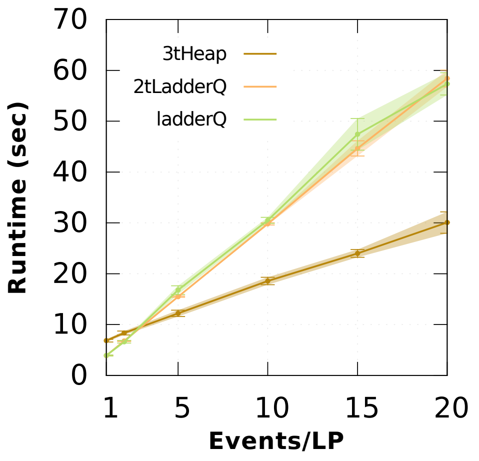
Figure 3.3 illustrate the runtime of the Java benchmark with 10,000 LPs (100 x 100) using exponential distribution with $\lambda = .1$ for timestamps in PHOLD. This configuration is the same as the high-concurrency configuration discussed by Higiro et al. The charts show averages and 95% confidence intervals computed from 10 independent simulation replications for each data point. The experiments were conducted on Red Hawk, that has Intel Xeon ®CPUs (E5520) running at 2.27 GHz (with hyperthreading disabled) and 32 GB of RAM (4 GB per core) in Non-Uniform Memory Access (NUMA) configuration. The nodes run Red Hat Enterprise Linux 6, with Linux (kernel ver 2.6.32) and OpenJDK 1.7.0_99 JVM.

Runtime characteristics of sequential Java benchmark along with linear regression to estimate runtime constants are shown in Figure 3.3. Note that curves for *ladderQ* and *2tLadder* are very close, which is as expected because there are no rollbacks in sequential simulation.

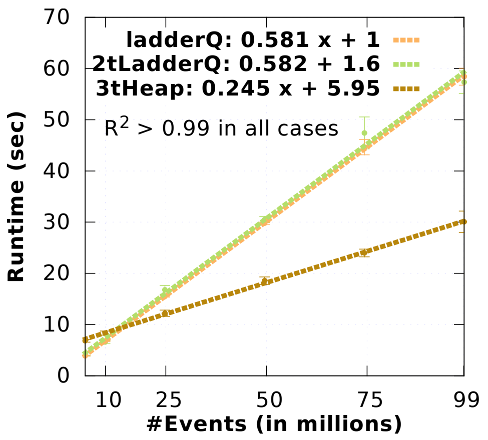
The performance trends of the Java implementation is similar to the MUSE simulation trends shown by Higiro et al. (see Figure 3.3(b) and [4]). Note that we refer to performance trends and not the raw timings because the Java implementation is a simplified version of the C++ simulator. In this scenario where a large number of concurrent events are to be processed, the *3tHeap* significantly outperforms both *2tLadder* and *ladderQ*. Consistency in trends in both Java and C++ establish that the performance differences arise due to the inherent design of the data structures and not due to platform or compiler peculiarities in the sequential case.

The charts also illustrate the result of a linear regression fit used to estimate the runtime constants of the 3 scheduler queues. The linear regression fits were very good with the coefficient of determination $R^2 > 0.99$. The linear runtime trends for *ladderQ* and *2tLadder* is expected because they have amortized $O(1)$ time complexity, and consequently $runtime = |events| * C$ (where C is a runtime constant). The linear regression with 95% confidence interval yields a per-event time constant (C) for *ladderQ* and *2tLadder* to be $0.0525\mu s - 0.0639\mu s$ and $0.0573\mu s - 0.0589\mu s$ respectively. The *3tHeap* also exhibits a similar amortized $O(1)$ trends, but with a much lower per-event time of $0.0234\mu s - 0.0255\mu s$.

Figure 3.4 illustrates the runtime performance difference between the three data structures in the presence of rollbacks. Rollbacks introduce the extra operation of occasionally canceling pending events. The chart in Figure 3.4(a) shows the number of synthetic rollbacks introduced in the benchmark. The number of rollbacks were very similar in all the runs and consequently the curves overlap. The data in Figure 3.4(b) shows the observed average runtime and 95% CI for the three queues estimated from 10 independent replications for each



(a) Java Benchmark Runtimes



(b) Linear Regression Fits

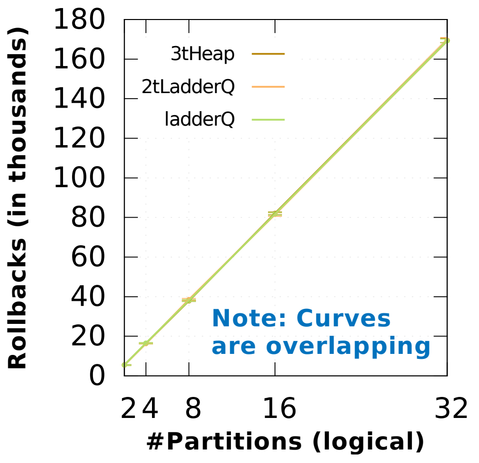
Figure 3.3: Java Robustness Benchmark (No Rollbacks)

data point.

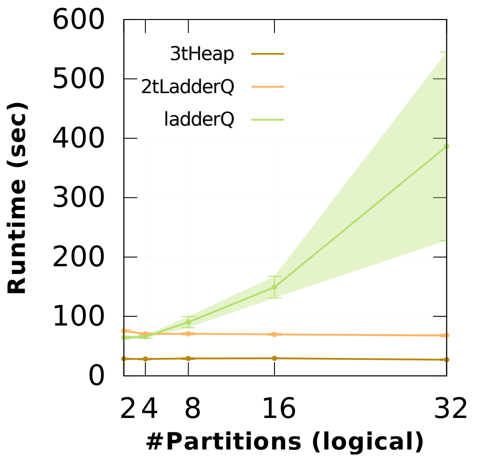
As illustrated by the data, *2tLadder* significantly outperforms the traditional *ladderQ* in the presence of rollbacks as per its intended design, with similar high performance achieved by *3tHeap*. These observations are consistent with the actual MUSE parallel simulation runtime trends as well. The consistency of sequential and parallel simulation trends in both Java and C++ establish that the advantages of the queues are reproducible across platforms and programming languages. In other words, the advantages of *3tHeap* and *2tLadder* in handling rollbacks are algorithmic and arise from its design rather than from hardware or compiler optimizations.

3.3 Lock-Free Pending Event Sets

The above mentioned Pending Event Sets are all designed for the sequential use case, with parallelization available only on the process level with message passing vs shared memory multi-threading. With their original implementation, therefore, a blocking, lock-based design pattern is needed to enable multi-threading. An example of such a design pattern is currently used by MUSE and outlined in Section 4.2. Lock-free structures avoid this locking design by intelligently accessing and manipulating state, often with very low level instructions that avoid the need for a thread to stop wait for a lock to be released. Examples include lock-free linked lists proposed by Harris in 2001 or lock-free skip lists proposed by Sundell et al. in 2005 (providing linear and logarithmic ordered insertion times respectively). [12, 13] These



(a) Number of Synthetic Rollbacks



(b) Java Benchmark Runtimes

Figure 3.4: Java Robustness Benchmark (with Synthetic Rollbacks)

structures are made possible by the use of low level atomic instructions, such as compare-and-swap (CAS), which allow them to be concurrently accessed without the need for locks and without the risk of race conditions.

Utilizing these types of primitive lock-free structures, there currently exist lock-free pending event set structures for the context of DES. For example, a version of thread safe Calendar Queue was proposed by Marotta et al. in 2017 that fixed many issues provided by various previous attempts at such a structure. [14] For example, previous attempts allowed for concurrent enqueue and dequeue, but required threads to block when a resize operation is necessary. Marotta's version shows reliable success and scalability on machines as large as 32 cores while accessing the same shared structure. A variation of Ladder Queue was proposed by Gupta et al. [15] While ladder queue has been shown to demonstrate performance gains over Calendar Queue, the version presented by Gupta is tailored exclusively to optimistic parallel processing due to the events in the bottom rung no longer being sorted, and thus the dequeue operation does not necessarily retrieve the overall least timestamp. Additionally, the version created by Gupta does not include the second tier proposed by Higiro, making it less efficient at handling rollbacks in optimistic simulation than it could be. Hardware Transactional Memory (HTM) support has also been explored as a means to provide a concurrent pending event set. [16] Manipulation of the structure can be done in constant time as HTM-based transactions. These types of structures however do not scale well on modern machines and can be prone to runtime issues unrelated to conflicted concurrent access.

Jagtap et al. [17] discuss their experiences with developing a multithreaded version of the ROSS Time Warp simulator. [18] Follow-up expanded research is presented in [19]. In their multi-threaded Time Warp simulator called ROSS-MT, event exchanges between LPs are accomplished using a single input queue per thread. Communication occurs by inserting a copy of the message in the input queue of the destination thread. The receiver thread dequeues events from the input queue and inserts them into the thread's event priority queue. This two stage insertion is used to avoid lock contention on the main scheduler event queue, however our research aims to accomplish event exchanges directly between threads via pointers to a shared queue. Jagtap also uses a NUMA-aware memory management scheme to reduce access latencies by having the event recycler choose events that were originally allocated by the receiving thread. We hope to implement a similar scheme in MUSE. In contrast, our research aims to run on a cluster of shared memory machine and not just one compute node. Moreover, we aim to use lock-free structures where possible. Pellegrini et al. also discuss NUMA issues and propose the use of Linux's NUMA API for optimizing memory access. [20]

Chapter 4

Problem Analysis

4.1 Problem Statement

Due to the importance of the pending event set structure, a large amount of research has already been done to find structures that allows very fast access, insertion, and rollback time complexities. The primary problem this research attempts to solve is finding a way to modify those already fast data structures for shared memory multi-threading in the context of a multi-core or many-core CPU to minimize conflicts that prevent immediate access/mutation and avoids locking. In other words, we want a structure that is safe to concurrently access at any time and is tolerant to high contention. As discussed above, the structures that have shown the most promise for PDES in both the sequential and optimistic use case are the 2-Tier Ladder Queue and 3-Tier Heap Queue proposed by Higiro. [4] We choose to investigate the 3-Tier Heap Queue ($3tHeap$) as the candidate for our thread safe structure due to its relative simplicity, ease of implementation, and other design benefits discussed in Section 5.1.2.

The four key data structure operations As discussed by Higiro et al, the following 4 operations on the data structures will be of primary focus, namely:

1. **Enqueue one or more future events:** This operation adds the given set of events to the pending event set. Multiple events are added to reprocess events after a rollback.

2. **Peek next priority event:** This operation returns the next event to be processed in priority order without removing it. The event is used to update an LP's LVT and schedule it. Note that in multi-threading this event could be removed by another thread immediately after being peeked.

3. **Dequeue events for next LP:** In contrast to peek, this operation dequeues all events at the next timestamp for a single LP to be processed. Concurrent events could have been sent by different LPs on different MPI-processes. Events at the same timestamp do not need to be processed in any particular order.

4. **Cancel pending events:** This operation is used as part of rollback recovery process to rapidly remove all pending events sent by a given LP (LP_{sender}) to another LP (LP_{dest}) at-or-after a given time ($t_{rollback}$). In our implementation, only one anti-message with send time $t_{rollback}$ is dispatched to LP_{dest} from LP_{sender} to cancel prior events sent by LP_{sender} to LP_{dest} at-or-after $t_{rollback}$. This feature short-circuits the need to send a large number of anti-messages thereby enabling faster rollback recovery.

We aim to optimize the worst case scenario of parallel DES. That is, where the amount of time processing each event is very small and thus requests to our scheduler data structure are frequent and near constant. For example, a naive approach to the problem is simply to lock the entire data structure anytime a thread needs to read or write to the scheduler. Because each thread must near constantly access and update the scheduler to continue processing new events, this naive approach would provide zero speedup in our worst case scenario since the blocking nature of the thread safety essentially renders event processing to be sequential. In contrast, real world processing of events calls for some level of process time per event. This process time between event handling operations (later referred to as granularity) relaxes some contention off the shared queue by requiring threads to do work before they request the next set of events. With high enough granularity, even a naive locking approach could achieve near perfect linear speedup as the proportion of time spent processing events outweighs the time our scheduler queue must spend organizing them, however we would like to ensure good performance independent of this event granularity.

4.2 Preliminary Investigation

In conjunction with the research conducted in this paper, a multi-threaded solution to DES simulation was developed and incorporated into MUSE. Outlined in Figure 4.1, our preliminary implementation uses multiple threads on a shared memory system but is agnostic to the pending event set data structure being used. That is, it operates by creating a separate scheduler queue for each thread, which is accessed sequentially and thus does not need any inherent thread safety. The LPs in the simulation are partitioned (or mapped) to specific threads. Each thread manages the lifecycle activities of the LPs it is allocated, even though

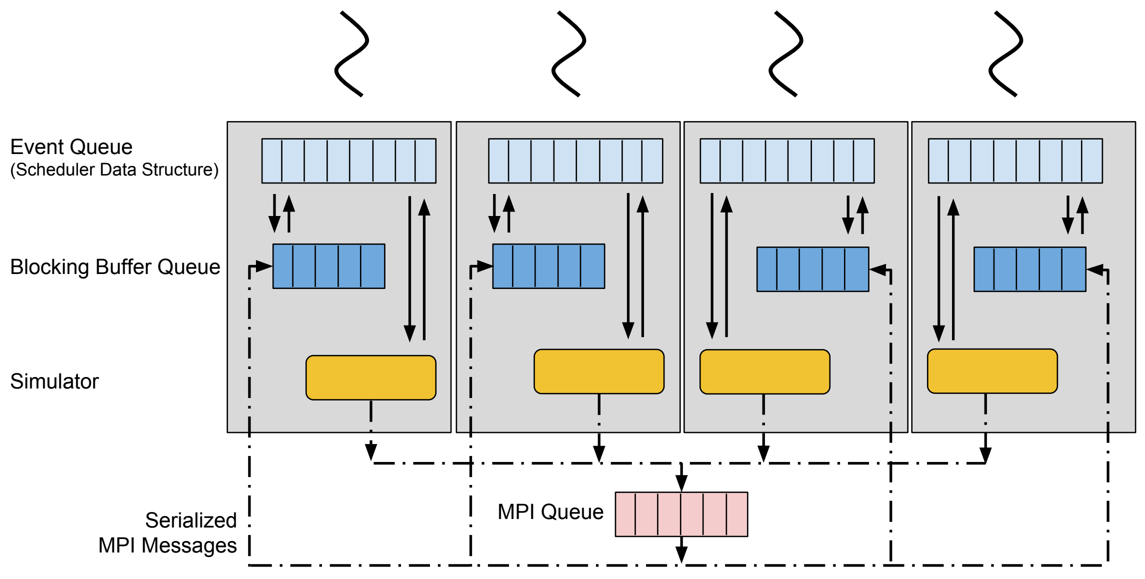


Figure 4.1: Preliminary Implementation for Multi-Threaded MUSE

all events are accessible via the shared memory space. This design accounts for the following three types of event exchanges between the LPs in a simulation:

1. **Intra-thread events:** These events are exchanged between LPs managed by a single thread. Since these events are restricted to a single thread (no risk of concurrent access), they are directly inserted into the pending event queues. This setup is identical to the single threaded implementation of MUSE.

2. **Inter-thread events:** These events are exchanged between LPs managed by different threads but on the same process. Inter-thread communication is accomplished by using an input buffer for each thread that receives incoming events and holds them until they can be safely inserted into to the non-thread safe Event Queue for that thread.

3. **Inter-process events:** Inter-process events are sent via serialized MPI messages (MPI maintains its own internal buffer queue) similar to the current MUSE design. However, on the receiving end multiple threads read messages from MPI leading to two possible scenarios, namely: ① the event is read by the thread that is managing the LP to which the event is destined and ② event is read by a thread that is not managing the LP. In the former case, the incoming event is processed by the thread directly. In the latter case, the event is inserted into the blocking buffer queue of the actual thread that is managing the LP to which the event is destined, similar to "inter-thread" exchanges.

Messages are regularly removed from the blocking buffer queue and then inserted into the Event Queue. By use of these buffers, this solution allows each thread to simulate without ever being significantly bottlenecked by another thread. Additionally, this solution is data structure agnostic and can thus be used with any scheduler queue, including non-thread safe data structures. However, this design experiences several issues, including:

1. Memory overhead of needing to make copies of events, which take up unnecessary space in buffers

2. Extra CPU overheads due to needing to copy, serialize, and transfer messages between threads, despite being on a shared memory system.

3. The blocking in-bound event queues are lock based and heavy inter-thread communication will give raise to contention on the blocking buffer queue, still resulting in locking overhead.

4. LPs are set to individual threads, meaning we do not gain the benefit of threads always pulling the least timestamp event across all LPs, increasing optimistic synchronization overhead compared to a shared scheduler queue design.

Ongoing Investigations for Improved Concurrency In continuation with this preliminary investigation, our research focuses on fundamentally enhancing the design using a single centralized scheduler queue shared by all the threads. The single scheduler queue is in contrast to the multiple scheduler queues (1 per thread) used by the preliminary implementation. The objective of a single centralized scheduler queue is to minimize the four key shortcomings of our preliminary design discussed above. However, using a single shared scheduler queue for multiple threads will significantly increase contention from multiple threads for performing the 4 key operations, namely: enqueue, peek, dequeue, and cancel events.

In order to minimize the cost of contention, we aim to use modern thread safe techniques that minimizes blocking by utilizing atomic instructions and synchronization. The goal is to make a thread safe version of the existing 3-tier heap ($3tHeap$) queue that allow threads to concurrently access the same shared data structure in memory while being bottlenecked by operations of another thread. The new structure must be able to provide thread safe operation for insertion of events to be scheduled, priority-order retrieval for least-timestamp-first scheduling, and out of order deletion due to rollbacks in a thread safe way. Due to the complexity of how information is stored in the existing structures, creating a non-blocking version comes down to identifying the minimal slice of the structure that must be protected

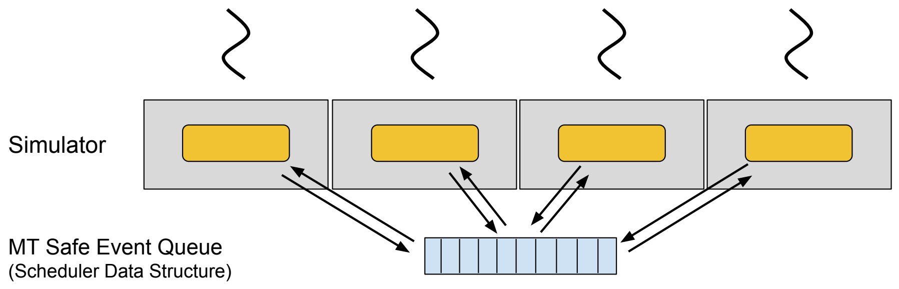


Figure 4.2: Proposed Implementation for Multi-Threaded MUSE

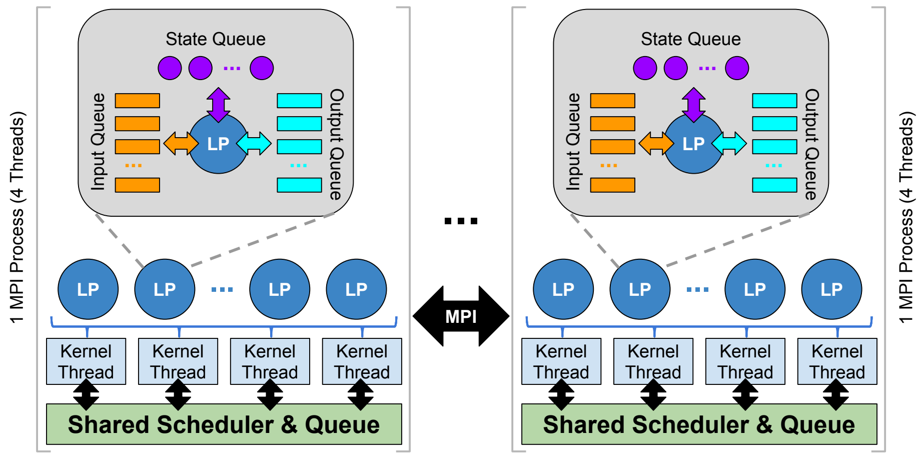


Figure 4.3: Proposed System Architecture Overview for Multi-Threaded MUSE

(critical sections) in order to safely conduct all three operations so as to avoid unnecessarily contention any one part of the queue at a time. Downfalls of this implementation are that its design is contingent on very specific data structures, unlike our preliminary solution which can be applied to any PES data structure.

4.3 Design Considerations

A multi-threading extension to MUSE will require restructuring its architecture to support concurrent operation. With shared memory multi-threading enabled, each thread will run as a scheduler permitting different agents to process events scheduled for them. Note that the subset of *LP*s that each scheduler operates on continuously changes depending on the pending events the unified scheduler queue that is shared by all the threads. The desired architecture, outlined in Figure 4.3, also brings in a new range of scheduling and event processing issues that need to be addressed.

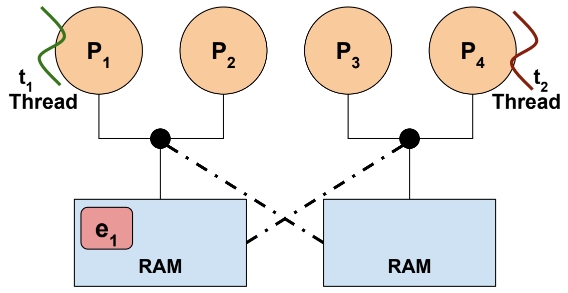
Rollback Management In configurations where each thread has its own sequential scheduler queue, rollback cancellation is performed immediately upon the triggering message being received, even if there are other pending events in the queue. Two types of messages trigger a rollback: ① a straggler event (i.e., event with lower timestamp than an *LP*'s LVT) or ② an anti-message. Upon being received, all pending events from the sender after the send time of the incoming event are immediately canceled.

However, with a shared scheduler queue, the rolling-back *LP* (i.e., the LP whose events are to be canceled) could be concurrently processing the events that need to be canceled when an event is received. Event processing of *LP*s is non-preemptable, and consequently the only alternative would be to enqueue the known straggler or anti-message and then processes the event once the *LP* in question has completed its event processing. To avoid blocking, the kernel must then temporarily enqueue the triggering event so that it may later be processed once the *LP* is no longer actively processing.

LGVT Estimation Another important challenge that arises when operating with multiple threads and a single centralized scheduler queue is estimation of Local Global Virtual Time (LGVT) on the multi-threaded process that is used to eventually compute the Global Virtual Time (GVT) in the overall simulation. With one scheduler-per-thread, LGVT estimation is relatively straightforward because obtaining a consistent snapshot of all threads is trivial. However, obtaining a consistent snapshot from multiple concurrent threads is a challenge. This is because each thread is operating independently and could be sending/receiving messages from other threads or MPI. In multi-threaded scenarios, two different approaches are possible:

1. **Synchronizing threads:** This is more or less a stop everything approach in which all threads are forced to synchronize LVT's. Once synchronized, a consistent snapshot of all threads on the process can be determined by one thread to compute LGVT for GVT estimation. This approach is relatively straightforward and is used by other investigators as well. [15, 21] Another advantage of this approach is that it is agnostic to which thread processes an LP and which thread handles the GVT estimation.

2. **Independent GVT managers per thread:** An alternative approach would be to to explore independent GVT managers on a per-thread basis. Essentially, GVT managers would perform just as they do in optimistic parallelization across processes, except they would be assigned at the thread level rather than the process level. This is much more complicated and expensive, and was not necessary for our implementation.



(a) Overview of NUMA Memory Architecture

available:	2 nodes (0-1)	
node 0	cpus:	0 2 4 6
	size:	12288 MB
	free:	10697 MB
node 1	cpus:	1 3 5 7
	size:	12275 MB
	free:	8990 MB
	node distances:	
node	0	1
0:	10	20
1:	20	10

(b) NUMA Configuration on Miami Red Hawk

Figure 4.4: NUMA Memory Layout Architecture and Configuration

NUMA-Aware Event Recycling and Memory Management Most supercomputing clusters, including the Miami University Red Hawk cluster, have compute nodes that have two or more CPUs in Non-Uniform Memory Access (NUMA) configuration as shown in the Figure 4.4(a). Each CPU has its own local Random Access Memory (RAM) modules that are fast to access. However, accessing other RAM modules is possible but incurs much higher latency. Consequently, with NUMA, access to different regions of the virtual memory space of a process have significantly different access times. For example, in Figure 4.4(a), assume that event e_1 is stored in the first RAM module as shown. In this scenario, access to event from thread t_1 is much faster than access from thread t_2. If thread t_2 is the intended thread that must process event e_1, then the overall event processing will be slowed down.

Figure 4.4(b) shows the NUMA configuration on a compute node on Red Hawk that has two Intel Xeon E5620 @ 2.4 GHz CPUs in NUMA configuration, labeled node-0 and node-1. As illustrated by the output of *numactl*, the time to access local modules is fast at about 10 time units (not to be confused with latency as distance is an pseudometric) while access to remote RAM blocks is 2 slower at 20 time units.

In order to avoid significant slowdown due to NUMA configurations, shared memory parallel processing must implement NUMA aware event access and recycling – that is, events will be reused based on the destination thread to which events are to be dispatched. This will require restructuring the event management infrastructure of MUSE to accept an optional destination LP ID. The destination LP ID will then be converted to a thread ID. If the thread is local then the NUMA-aware event recycler will select the appropriate event to be recycled to minimize NUMA issues.

Chapter 5

Implementation and Solution

5.1 Pending Event Set Data Structure

A Pending Event Set that can store and manage events in a thread safe way requires two primary features:

1. A Concurrent Priority Queue (CPQ) data structure with ① enqueue, ② peekMin, ③ deleteMin, and ④ deleteAt operations.
2. Wrapper logic that utilizes CPQs to efficiently manage lists of events across multiple agents.

Shared state would be contained to inside CPQs, with the wrapper logic performing operations on that state to allow multiple threads to schedule and process events in a $PDES$ simulation.

5.1.1 Concurrent Priority Queue

Gruber discusses that skip-list backings for priority queues are the current state of the art for strict shared memory concurrent priority queues. [22] He furthers that Lindén-Jonsson's skip-list based queue has particularly good performance under very high contention conditions, which is the property we are most interested in for our event queue. [23] Specifically, skip-lists are relatively simple to implement, and by making balancing decisions for the search tree that are agnostic to the state of the queue, skip-lists are particularly useful in concurrent structures where contention for shared state is the primary bottleneck on performance. As a result, we chose to use the Lindén-Jonsson queue as the starting point for our CPQ backing

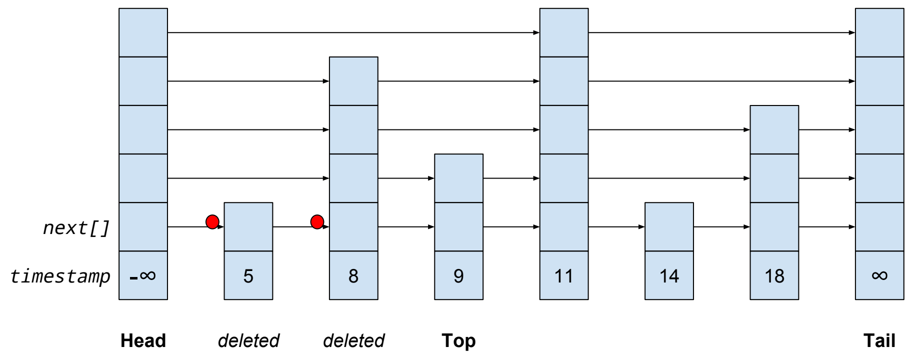


Figure 5.1: Structural Overview of Lindén & Jonsson Concurrent Skip List Priority Queue

data structure. It's important to note that the backing priority queue and the wrapper logic are independent, so future research on a better CPQ could be easily incorporated into our overall pending event set structure.

The concurrent queue is designed to handle simultaneous insert and delete operations without data loss, duplication, or inaccuracy. The structure of the queue can be seen in Figure 5.1. There are two important regions of the queue: ① the $next[\ ]$ skip links above level 0, which are used to reduce search complexity when traversing the queue, and ② skip links on level 0, which represents the true order of the queue and is traversed to find the min node (top). The height of these levels for a single node is chosen at random with a geometric distribution, meaning that for level (starting with level 0 which is always present) the likelihood of a node reaching the level above it is cut in half. As a result, traversing links from the top down creates an effective binary search for long lists.

As a result of this design, we can concurrently insert and delete while maintaining causality by manipulating pointers on level 0 (the true state of the queue), then restructuring the upper region purely to decrease search time. This is done by using the $next[0]$ pointer (level 0) to hold the deleted state by flipping its last bit, allowing for the detection of a conflicting inserts and deletes on the same node with the compare-and-swap (CAS) operation. [23] If a thread attempts to insert a new node directly before a node that is also trying to be deleted (which should thus be the new top), or if a thread attempts to delete the same node as another thread, the CAS instruction will detect the conflict because the $next[0]$ pointer will have changed, causing the operation to fail, depending on which thread managed to manipulate the $next[0]$ pointer first.

Modifications to Lindén & Jonsson The primary modification that needed to be made to the original queue was the ability to delete an arbitrary key from the middle of the queue (*deleteAt* operation). This is because in the final PES queue, an inserted event for an agent could likely change the priority of that agent, thus requiring that agent to be reorganized. This operation is also necessary for rollback functionality. This change was done by creating a second operation in the delete process in addition to flipping the bit on the *next*[0] pointer: a compare-and-swap on the value reference for the node. Delete operations still flip the bit on the pointer to avoid conflicting operations, however the thread must also then successfully set the value reference to *null* in order to successfully complete the delete operation. This allows restructure threads to delete an element from the middle of the queue by setting the value to *null*, and allowing the node to naturally move to the top of the queue and then be ignored.

This is a novel modification to the Lindén & Jonsson queue, however it comes at a cost. For one, nodes deleted via the *deleteAt* operation must waste space in the queue as they wait to be moved to the top where they are actually deleted. In a situation where many *deleteAt* operations are being conducted (which is true in our case), this results in a queue that could be multiple times larger than necessary. So long as we can achieve non-locking operation, this increased size can be acceptable due to the constant time deleteMin and the log time insert, however it is results in both significantly increased space as well as increased runtime due to needing to delete nodes twice. Additionally, restructuring a node in the queue (in our current implementation) requires a deleteAt followed by a subsequent enqueue as shown in Figure 5.4 Algorithm 8. In theory, it should be possible to simply restructure the node in place without the need to fully remove and re-insert. This operation has been implemented by Higiro in his $3tHeap$, however would present extensive challenges to be conducted concurrently. Both of these problems are an area of future work on the CPQ, as the *deleteAt* and restructure operations are necessary to the operation of a PES structure.

It's important to note the reason for this type of convoluted *deleteAt* operation. Ideally, we would simply logically delete the node from the middle of the queue by setting the *next*[0] pointer of the previous node to the successor node (thus bypassing the node we want to delete). This is not possible concurrently, however, as it is possible another node could be trying to insert a new node immediately before the node we are trying to delete, creating a data race on the *next*[0] pointer for the previous node where the state of the queue is different depending on which thread manipulates the pointer first. This scenario is demonstrated in Figure 5.2 with the pointer in question highlighted in red, and is discussed by Lindén & Jonsson as the motivation behind having the delete flag and the next pointer be the same point in memory. [23]

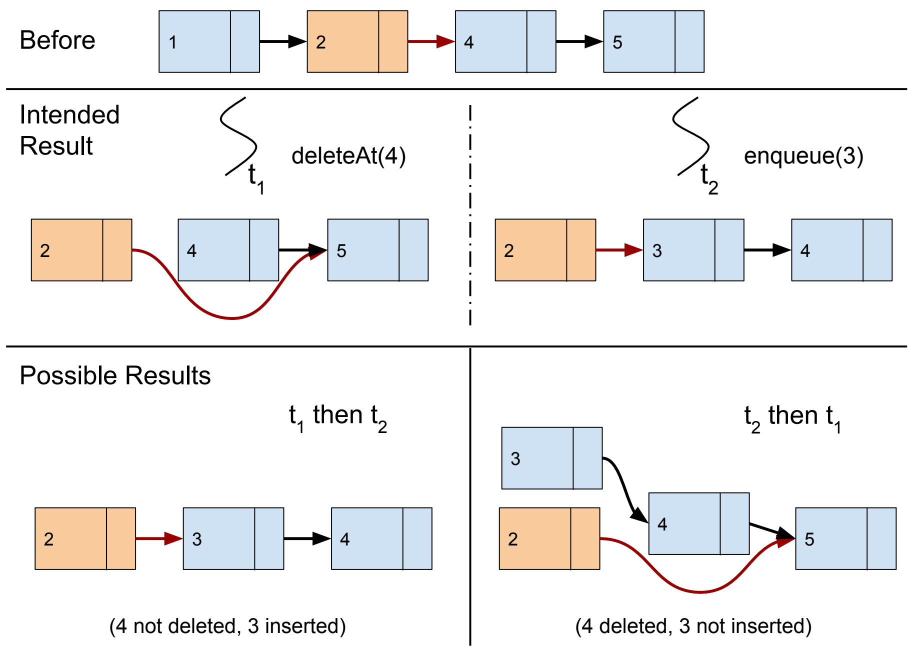


Figure 5.2: Data Race Scenario Presented by Naive Concurrent Insert and DeleteAt of Linked-list

5.1.2 3-Tier Structure Design

We base our PES structure on the Three Tier Heap ($3tHeap$) by Higiro described in Section 3.1.1 to create our Three Tier Skip-List with Multi-Threading Support ($3tSkipMT$). [4] The overall layout of our structure mimics that of $3tHeap$ except with a concurrent priority queue (CPQ) as the backing structure rather than the C++ standard library's heap based priority queue. Our interpretation uses a CPQ for Tier-1 and Tier-2 (each of which is sorted) and a standard vector queue with locks for Tier-3 (which is not sorted, and does not have high contention).

This three tier design also complements the design constraints outlined in Section 5.2, specifically the Agent State Critical Section where no two threads can be processing (dequeue events for) the same agent at the same time. By having agents sorted in the top level queue, any thread can quickly and safely pop the top agent off the Tier-1 queue to gain exclusive processing rights to that one agent and prevent any other thread from attempting to process events for that agent concurrently (since the agent is removed from the queue in a thread safe way). Concurrent inserts of events can still occur on that agent while it is processing by directly accessing the agent's Tier-2 and Tier-3 queues (agents hold a reference to their tier-2 queues), but no two threads can dequeue events for the same agent at the same time by using this multi-tiered structure.

This multi-tiered design also limits contention overhead by reducing the likelihood that any two threads will be concurrently manipulating a single CPQ at the same time. The bottleneck to this is need for the top level queue to be accessed by every thread before it can access the lower tier queues, however this is still significantly less than if all events were stored in a single queue with constant contention by all threads. Additionally, inserts for one agent have zero contention with inserts for another. Lastly, we achieve all the same benefits of the sequential $3tHeap$, such as constant time dequeue and $log|n|$ inserts where n is the number of timestamps for events on a single agent rather than the number of events overall across agents.

5.2 Simulation Kernel Integration and Wrapper Logic

Converting a sequential system into a parallel one is no easy task. Before building a data structure that supported concurrent operation in the context of Parallel Discrete Event Simulation (PDES), we found it valuable to instead work from the top down in redesigning the system. We decided to first modify the MUSE simulation kernel (see Section 2.3) to support shared memory parallelization, and use this process to determine the properties a Pending Event Set would need (as has been described above), and then use that knowledge to design the data structure and later include it in MUSE.

Multiple design issues became apparent during the implementation of a thread safe non-blocking kernel compared to the original sequential version...

1. **Kernel Pipeline Critical Sections** The first aspect of converting MUSE to a thread safe, shared memory platform involved identifying and protecting critical sections in the kernel. The original MUSE makes many underlying assumptions about variables and state that are no longer valid in concurrent programming.

   For example, in Algorithm 3, the kernel first checks if the event queue is empty, and then begins processing it. In concurrent programming, it's possible a thread t_1 is removing events at the same time as t_2, meaning the moment t_1 calls $eventPQ.empty()$ and gets false, t_2 may have just removed the last event and made the queue empty, even though t_1 thinks the queue should have items in it. Also in Algorithm 3 Lines 5-6, if $eventPQ$ was shared between threads, the agent retrieved on Line 5 could also be concurrently retrieved by another thread, meaning Line 6 would be a data race to pull events associated with that same agent and process them.

Algorithm 3 Sequential MUSE - Kernel

```
1: function StartSimulation
2:     while $GVT \leq endTime$ do
3:         $LGVT \leftarrow eventPQ$.nextTime()
4:         if NOT $eventPQ$.empty() then
5:             $agent \leftarrow eventPQ$.front.agent
6:             $events \leftarrow eventPQ$.dequeue()
7:             $agent$.process(events)
8:         end if
9:         updateGVT()
10:     end while
11: end function
```

Algorithm 4 Concurrent MUSE - Kernel

```
1: function SingleSimThread
2:     while $GVT \leq endTime$ do
3:         $agent \leftarrow agentPQ$.pop()
4:         $events \leftarrow agent$.events.dequeue()
5:         $agent$.handleRollbacks($events$)
6:         $threadLGVT = events$.time
7:         $agent$.process($events$)
8:         $agentPQ$.restructureTop($agent$)
9:         updateGVT()
10:     end while
11: end function
```

Algorithm 5 Sequential MUSE - Schedule

```
1: function ScheduleEvent($event$)
2:     $agent$.handleRollbacks($event$)
3:     $eventPQ$.enqueue($event$)
4: end function
```

Algorithm 6 Concurrent MUSE - Schedule

```
1: function ScheduleEvent($event$)
2:     $agent = event$.recipient
3:     $agent$.enqueue($event$)
4:     $agentPQ$.restructureTop($agent$, $event$)
5: end function
```

Figure 5.3: MUSE Sequential vs Concurrent - High Level Kernel Pseudocode

Overall, the kernel pipeline must make any calls to shared state in a single operation, as is done in Algorithm 4. Any state that is shared between threads is accessed in a single function call, resulting in no possibility that state changes between calls. For example, Algorithm 4 Line 6 sets the local threadLGVT from events that have already been dequeued while Algorithm 3 Line 3 gets LGVT from what would be a shared state before the events associated with that LGVT have been removed from the queue.

2. **Agent State Critical Section** Processing events on an agent is a very important critical section for our concurrent version. The next state for a given agent is very critically dependent on the previous state, and no level of concurrency can exist between state transitions. Put more simply, an agent must have fully processed all previous events before it can begin processing future events, meaning two agents can not process events at the same time without conflicts.

   Our solution to this issue creates a situation where no two agents will ever be processing events concurrently. This is done by creating top-tier thread safe queue that contains agents prioritized by their next timestamp (See Algorithm 4, *agentPQ*). By removing the agent from this queue in a thread safe way, a thread ensures that it alone has control of manipulating that agent's state and that no other agent is processing events. This also applies to rollbacks, as rollbacks change agent state, thus requiring our version to process rollbacks as part of the dequeue process rather than scheduling process as had been done previously (See Algorithm 6 Line 5).

   Other threads can still access the agent directly, however these threads would only be able to schedule new events onto that agent, not remove events for processing. This design allows for agents to still concurrently add new events to an agent without blocking, while also ensuring a critical section is achieved for agent processing and state changing.

3. **Synchronization** As discussed in Section 2.1, Synchronization is necessary when events are processed in parallel and can cause causality issues. This is usually a problem with distributed parallel simulations due to different processes on different compute nodes getting out of sync and sending events out of time. The issue presents itself in multi-threaded simulations, where one thread may be processing events at some timestamp while another thread inserts events for that agent at the same timestamp.

   The result of this situation is not serious, as we simply need to rollback the agent before re-processing the events at that timestamp, however it presents an unavoidable overhead to making simulations multi-threaded. One benefit is that while the situation

Algorithm 7 Restructure After Dequeue

```
1: function RestructureTop(agent)
2:     agent.restructureLock.lock()
3:     nextMin ← agent.nextMin
4:     agentPQ.insert(agent, nextMin)
5:     agent.restructureLock.release()
6: end function
```

Algorithm 8 Restructure After Enqueue

```
1: function RestructureTop(agent, evt)
2:     agent.restructureLock.lock()
3:     if evt.time < agent.nextMin then
4:         agentPQ.delete(agent)
5:         agentPQ.insert(agent, evt.time)
6:     end if
7:     agent.restructureLock.release()
8: end function
```

Figure 5.4: MUSE Concurrent - Top Tier Restructure Pseudocode

is possible, sharing the event queue between threads prevents any one thread from getting drastically out of sync as is possible in distributed optimistic parallel simulation. Each thread always pulls the lowest timestamp event across all concurrent threads, requiring the errant event to be both at the next possible timestamp in the queue and for an agent that is processing at that next timestamp concurrently, which is unlikely. Regardless, thread-based rollbacks will be a minor overhead that will especially be apparent in smaller simulation with a fewer number of agents.

4. **LGVT Management** In sequential and optimistic simulations, local global virtual time (LGVT) calculation is straight forward as we simply keep track of the timestamp of the last event we processed (Algorithm 3 Line 3). With multiple threads sharing the same set of events, however, we cannot simply set a single variable as a data race would occur. For example, if a thread pulls event e_1 at timestamp $t = 1$ and another thread pulls the next event e_2 at timestamp $t = 2$, the variable for LGVT could be set to either timestamp, depending on which thread sets the value first after they pull their respective events. This means its very likely that LGVT could be set to $t = 2$ before e_1 has even begun being processed, which is invalid. LGVT should always be the

   The simplest solution to this was to keep a thread local instance of LGVT per thread. This *threadLGVT* holds the minimum timestamp processed by that thread, and the minimum value across all values would be the true LGVT for the process. There still exists a data race between accessing a *threadLGVT* and that thread moving on to the next event, but this will only ever cause LGVT to be underestimated, never overestimated as demonstrated in the example above.

5. **Top Tier Restructure** The top level queue *agentPQ* in Algorithm 4 must be restructured ① when events are removed and ② when an event with timestamp less that the previous min event are scheduled. This is a function that cannot be done concurrently, and can result in conflicts if not handled properly. This is because the $nextMin$ value of the agent can change while a thread is restructuring if concurrent enqueues or dequeues are present on the agent. Figure 5.4 shows the difference in logic necessary to restructure in concurrent mode, and critically requires a lock on the agent to be done properly.

   The most important insight about the restructure logic is that a concurrent dequeue and enqueue can still occur, just not the restructure portion of the operation. For example, a thread t_1 will always restructure following a dequeue operation, but a thread t_2 can still perform the enqueue operation while t_1 is processing, t_2 just has to block while t_1 restructures before it can be allowed to start its own restructure operation. Restructure is a required critical section, but by limiting this necessary blocking to only the critical portion of the operation, we avoid excessive overhead.

   Following a Dequeue (Algorithm 7), it is only possible for other threads to be inserting new events, not removing them (because dequeue on an agent is a critical section in and of itself). As a result, we can safely ask the agent for $nextMin$ (so long as we have the lock) because this min can only get smaller (if a thread inserts an earlier event) not larger (only possible by removing an event). As a result, even if another thread inserts and changes this $nextMin$ while we're restructuring, we safely know that that insert thread will restructure itself once it can get the lock, and another thread attempting to dequeue will not pop an agent that isn't actually the minimum timestamp (which would be possible if nextMin was able to increase in value concurrently).

   Following an Enqueue (Algorithm 8), other threads may be enqueuing or dequeuing. This is fine, because we have a constant value for our restructure time (*evt*.time) which is the timestamp of the event that was just scheduled. This means we aren't dependent on the shared queue state for our key value in the priority queue, and the only thread safety we need to worry about is the *restructureLock* to avoid race conditions on the previous key. If another thread enqueues or dequeues to change the value of *agent*.nextMin after we check it on Line 3, that thread will handle the necessary restructure after this thread give up the lock.

6. **Garbage Collection Issues** The sequential version of $MUSE$ uses reference counters to keep track of when we are done with an event and tag it for recycling by the garbage

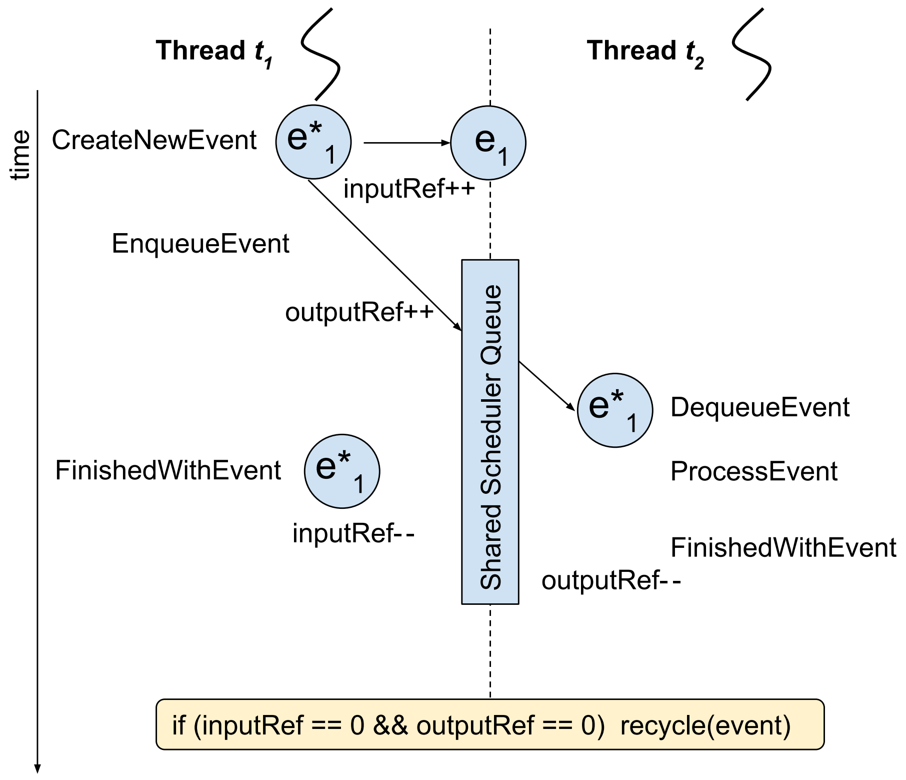


Figure 5.5: 2 Reference Counter Garbage Collection Overview

collector. Because of concurrency, this method could result in data races between threads, and provides no way to determine when the other thread was finished with an event and when it left the recipient agent's state queue. As a result, we instead implemented two reference counters on each event: ① the reference count on the thread that created and sent the event, and ② reference count on the thread that receives the event, processes it, and inserts it into the agent state queue until it is completely no longer necessary by the simulation.

By using this two counter system, we can ensure that both threads are independently finished with an event before scheduling it for recycling, without needing to add time complexity or excessive overhead to our simulation. See Figure 5.5 for an overview of the two counter garbage collection system.

Chapter 6

Results and Discussion

6.1 Experiments

PHOLD Benchmark The PHOLD benchmark has been used by many investigators because it has shown to effectively emulate the steady-state phase of a typical simulation. Our PHOLD implementation developed using MUSE provides several parameters (specified as command-line arguments) summarized in Figure 6.2. The benchmark consists of a 2-dimensional toroidal grid of Logical Processes (LPs) specified via the rows and cols parameters as shown in Figure 6.2. The total number of LPs in the simulation is rows cols. LPs are evenly partitioned across the MPI-processes used for simulation.

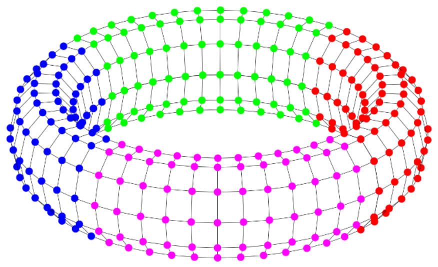


Figure 6.1: Arrangement of LPs in PHOLD Benchmark

Figure 6.1 shows the arrangement of LPs (represented by small circles) partitioned to 4 parallel processes with colors indicating the partitions. However, the imbalance parameter influences the partition, with larger values skewing the partition such that more LPs are assigned to some partitions. The imbalance parameter has no impact on sequential simulations.

The PHOLD simulation commences with a fixed number of events for each LP, spec-

Parameter	Description
rows	Number of rows in model.
cols	Number of columns in model.
eventsPerLP	Initial number of events per LP.
simEndTime	Simulation end time.
%*selfEvents*	Fraction of events LPs send to themselves.
partitions	Number of partitions to divide *LP*s into for rollback simulation
delayDistrib	Event timestamp distribution*
recvrDistrib	Receiver ID distribution*
delay or λ	Parameter for distribution specified by *delayDistrib*
recvrRange	Parameter for distribution specified by *recvrDistrib*
granularity	Additional compute load per event
imbalance	Imbalance in partition, i.e. more LPs on some partitions.

*distribution can be one of "uniform", "poisson", or "exponential"

Figure 6.2: PHOLD configuration options

ified by the *eventsPerLP* parameter. For each event received by an LP a fixed number of trigonometric operations determined by *granularity* are performed to place CPU load. For each event, an LP schedules another event to a randomly chosen adjacent LP determined by *recvrDistrib* and *recvrRange* parameters. The *selfEvents* parameter controls the fraction of events that an LP schedules to itself. The event timestamps are determined by a given *delayDistrib* and *delay* or λ parameters. The combination of parameters can be used to model different interaction patterns and simulation-time behaviors of various models. Specifically, combinations of these parameters influence the number of concurrent events (i.e., events with the same timestamp) that are scheduled to be processed by a given LP. The number of concurrent events strongly influences rollback probabilities as well as the effective performance of various scheduler queues.

We test our multi-threaded *PES* data structure using the above parameters and compare results where applicable to Three Tier Heap (3*tHeap*) by Higiro. [4] This is because 3*tHeap* is structurally very similar to Three Tier Skip-List with Multi-Threading Support (3*tSkipMT*) besides their underlying container structure and usage of multiple threads. Additionally, 3*tHeap* was shown by Higiro to be a state of the art structure for sequential and optimistic *DES* simulations, making it the target to beat to demonstrate useful speedup.

Environment The experiments were run on the Miami University Redhawk Supercomputing cluster. Each node of the cluster contains two Intel®E5620 CPUs @ 2.4GHz providing a total of 8 cores with 32GB of RAM in NUMA configuration. Trials were individually run on a single dedicated compute node with up to 8 threads (1 per core), with the average of 3 runs being taken per data point. Any outliers were manually reviewed and re-run if necessary.

6.1.1 Experimental Data

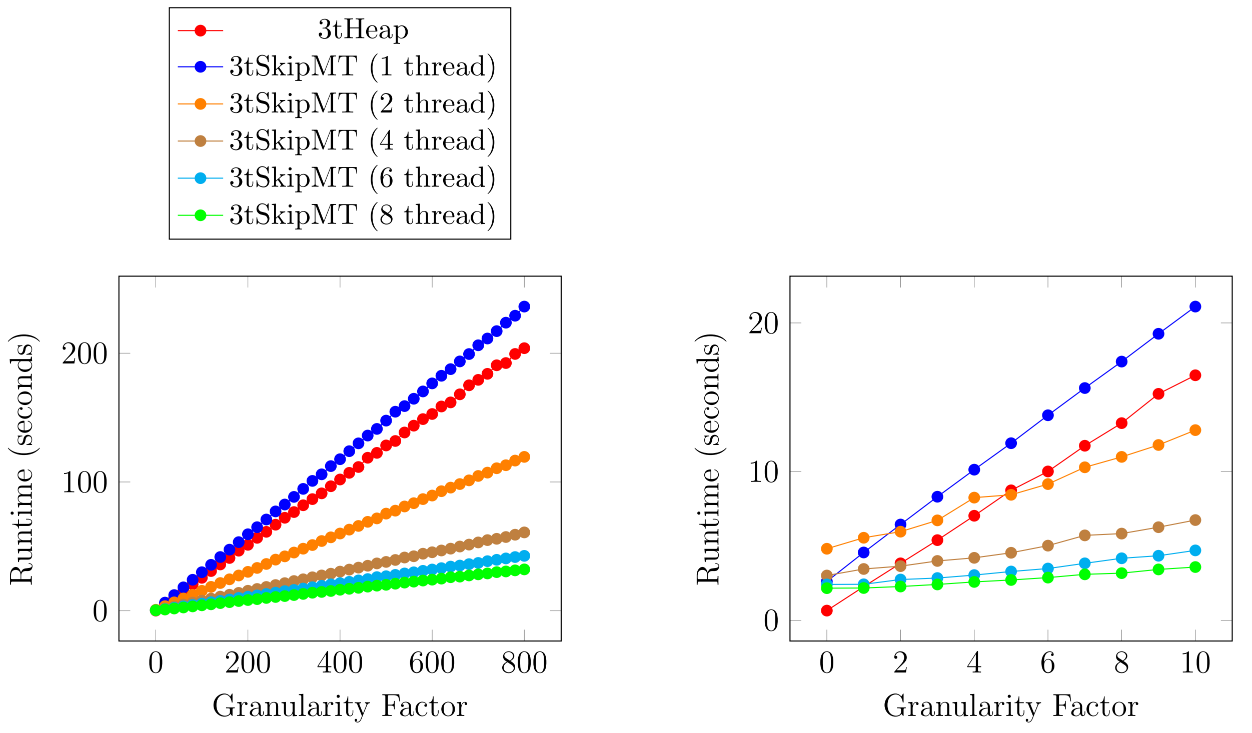
We collected preliminary data across the various parameters for PHOLD described in Section 6.1, finding significant comparative results with the *granularity* and *row/col* factors (total number of LPs in the simulation). Additionally, we measure scalability with both an optimized vs unoptimized compile of the benchmark ($-O3$ vs $-O0$ compiler flags) to compare the impact of optimization on the two structures. These results can be seen in Figure 6.3.

Granularity data was run with 400 LPs with varying granularity factor with an optimized compile. Scalability data was run with granularity factor of 5 and varying row/col values with both optimized trials and non-optimized trials. A granularity factor of 5 was chosen for scalability data as it provides a realistic amount of compute time between events as determined by our granularity impact analysis visible in Figure 6.4.

Four factors overall were tested to generate our result: ① The number of Logical Processes (LPs) in the simulation, ② the granularity factor of the PHOLD experiment, ③ optimization vs no optimization at compile time, and ④ the number of threads/CPU cores utilized.

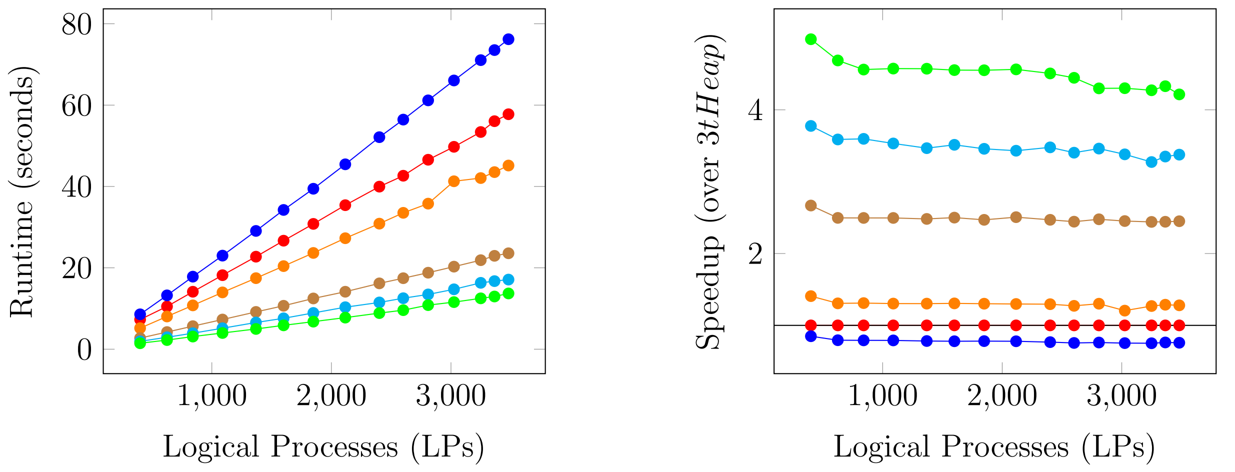
6.2 Results and Discussion

Scalability and Simulation Size As shown in Figure 6.3(c), the baseline $3tHeap$ outperformed our $3tSkipMT$ when run sequentially. This baseline also scaled better than sequential $3tSkipMT$ as more LPs were added to the simulation, however both structures scaled linearly to size. The runtime comparisons between these two sequential trials demonstrates the expected overhead produced with our multi-threaded implementation and less efficient backing data structure necessary for lock-free operation. Importantly, we see both of these trends scale in a linear fashion, implying that we do not lose asymptotic complexity when providing multi-threading support. We also notice diminishing marginal returns on additional threads, achieving much less speedup from 4 vs. 8 threads compared to 2 vs 4 threads. This represents the overhead experienced from thread conflict on shared memory. While more threads provides more compute power, more threads also adds additional memory contention, resulting in less relative speedup once 6-8 threads are being used.



(a) Granularity Factor - Zoomed Out

(b) Granularity Factor - Zoomed In



(c) Sim Size vs Runtime

(d) Speedup over $3tHeap$ at various sim sizes

Figure 6.3: Relative Performance of $3tHeap$ vs. $3tSkipMT$ with multiple threads

Granularity Granularity in this context represents the amount of compute time used to calculate the next state for an LP when processing an event. In our experiment, 0 granularity represents immediately generating a new event the moment we receive the previous one. This is unrealistic, as in real DES simulations there are often complicated equations that must be solved at each time step, occasionally requiring multiple rounds of computation. This gap between event processing presents a critical factor in utilizing multi-threading benefits in a shared memory system. Granularity heavily reduces contention between threads by increasing the time each thread can run independently before needing to query the shared portion of the system, reducing contention and allowing each thread to better take advantage of the additional computing resources.

This is demonstrated in Figure 6.3(a) and Figure 6.3(b) where both sequential simulations are heavily impacted by granularity while parallel simulations are less and less affected by increased granularity as more threads are added to the system. Furthermore, Figure 6.3(b) shows how with granularity < 2, two threads results in higher runtime than sequential $3tSkipMT$. This is because with such small granularity, the contention overhead between the threads is so great that it becomes faster to simply process the simulation sequentially.

This finding makes it important to measure the actual impact of granularity in our experiments. To do this, we ran thousands of sequential trials with varying sizes and varying granularity factors. From this data, an average runtime per event was generated for each granularity factor and plotted in Figure 6.4. Due to the slightly different implementation between our multi-threaded simulation and our baseline sequential version, linear regression was done separately for each type of simulation kernel. In both cases, we find a linear relationship of roughly 1.05-1.25 μs runtime increase per event per granularity factor.

Effect of Optimization To test the effectiveness of optimization on our structure, we ran sequential trials of $3tHeap$ and $3tSkipMT$ with and without the $-O3$ flag to the g++ compiler to give us optimized and non-optimized trials. We ran our optimization analysis sequentially in order to test the effectiveness of optimization on the data structure, not its impact on multi-threaded performance. This is because while our data structure becomes faster when optimized, optimization also leads to reduced per event processing time (granularity) which we already found to heavily decrease multi-threaded performance. Because of these confounding variables, we opted to test only the underlying pending event set structure for optimization benefits. These trials were run with PHOLD granularity set to 0 to increase stress on the data structure, as opposed to previous trials that were run with more realistic granularity factors.

Figure 6.5 shows the runtimes and relative speedup with and without optimization.

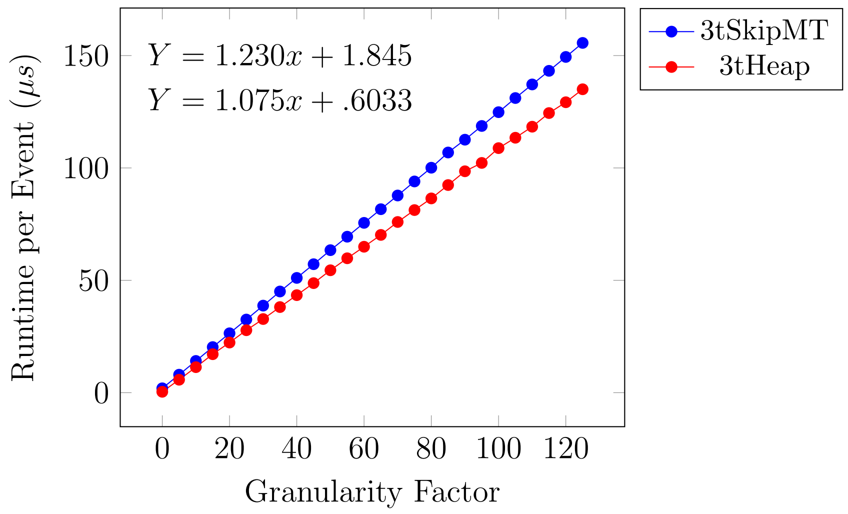


Figure 6.4: Impact of Granularity Factor on Per Event Runtime for Sequential Simulations

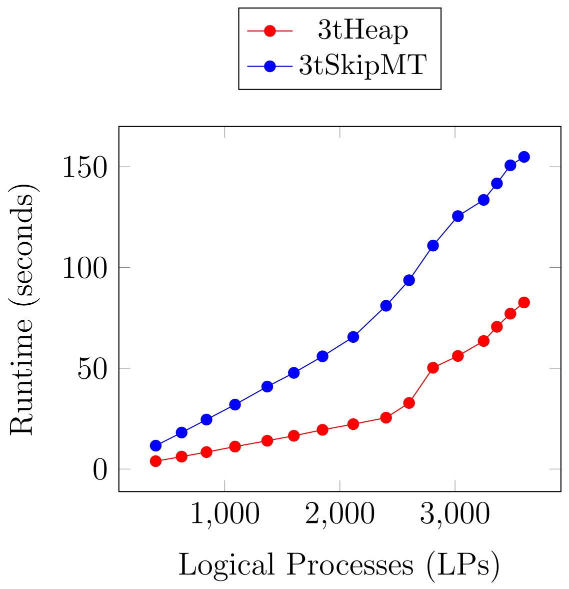
$3tSkipMT$ had slower runtimes across all simulation sizes due to it not being as efficient of a data structure, and had fairly consistent speedup when optimized with more speedup at smaller simulation sizes. $3tHeap$ was faster, and had more volatile speedup with a very large amount of speedup for small simulations and rapidly switching to lower speedup at larger simulation sizes. This is because $3tHeap$ utilizes vectors for its backing queues, meaning caching is much more effective than the linked list type structure utilized by $3tSkipMT$. Once enough LPs are in the system, this caching benefit is reduced and speedup quickly tapers off.

Overall, we found that our structure achieves significant speedup from optimization and scales at roughly the same rate as $3tHeap$ with optimizations enabled. $3tHeap$ achieves much more significant optimization benefits in small simulations due to its ability to benefit from caching, however in large simulations this difference in speedup is not as apparent.

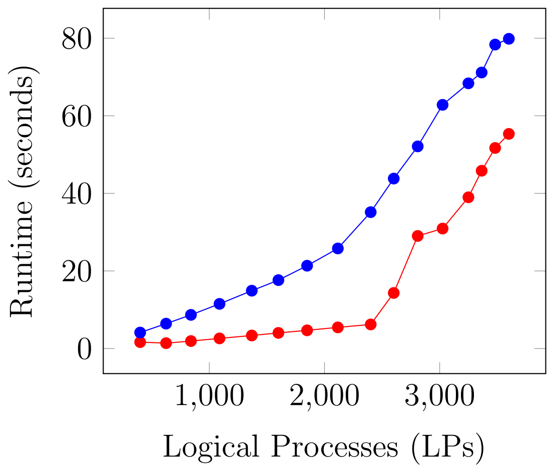
6.3 Comparative Analysis

The Ultimate Share-Everything PDES System (USE) was described by Ianni et al. in 2018 as a solution to very similar challenges that we set out to solve in our present analysis. [24] They utilize a conflict resilient Calendar Queue that is lock-free and aims to maximize performance with high thread contention, similar to our $3tSkipMT$. [14] While the two systems share many similarities, comparison is still challenging due to differing designs with differing kernel implementations.

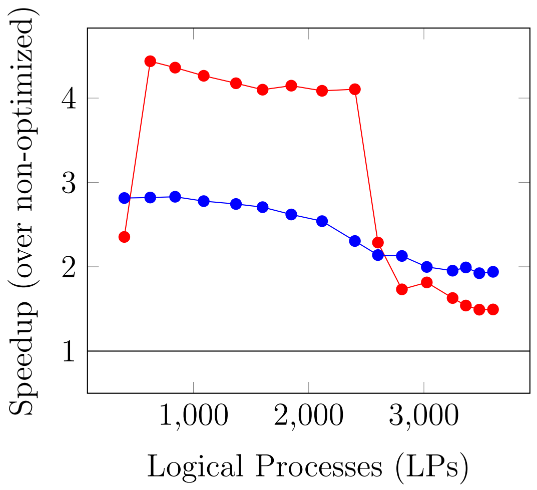
Integration of the USE system into our existing benchmarking PHOLD test was outside



(a) Not Optimized: Runtimes vs sim size



(b) Optimized: Runtimes vs sim size



(c) Speedup gained from optimization

Figure 6.5: Impact of Compiler Optimization (at $-O3$ level) on Data Structure Performance

the scope of our analysis. However, the USE kernel includes its own version of the PHOLD benchmark with many of the same factors used by our own benchmark. For our comparison, we overrode the granularity portion of the USE PHOLD benchmark to mimic our own granularity function, allowing us to add the same granularity factor to the individual event processing portion of each system. We also adjusted the event distribution framework of USE PHOLD to mimic to toroidal shape used in MUSE PHOLD as described in Figure 6.1. While these changes do not create a perfect comparison environment, they at least allow us to compare the impact of the same granularity factor change across the two implementations.

Our comparative experiment runs each version of a shared memory PDES system in such a way that a similar number of both LPs and total committed events are simulated. The two systems are then simulated with a various number of threads with an increasing granularity factor to see the impact of granularity on speedup with various threads. This is meant to test the resilience to contention between the two systems, as this is the key result that both systems attempt to maximize.

Our data, displayed in Figure 6.6 and detailed in Figure 6.7, shows expected results for $3tSkipMT$, where speedup starts very low, even experiencing slowdown with 0 granularity due to such high thread contention, but quickly levels off to approach roughly linear speedup per thread increase. USE had unexpected behavior, where zero granularity resulted in an expected distribution of speedup, but increases in granularity resulted in less increased performance per thread, eventually resulting in 8 threads having the same performance as 2. Overall, this analysis is merely a preliminary attempt to compare two very different systems, but does at least confirm the effectiveness of $3tSkipMT$ to achieve very significant speedup in reasonably high granularity scenarios.

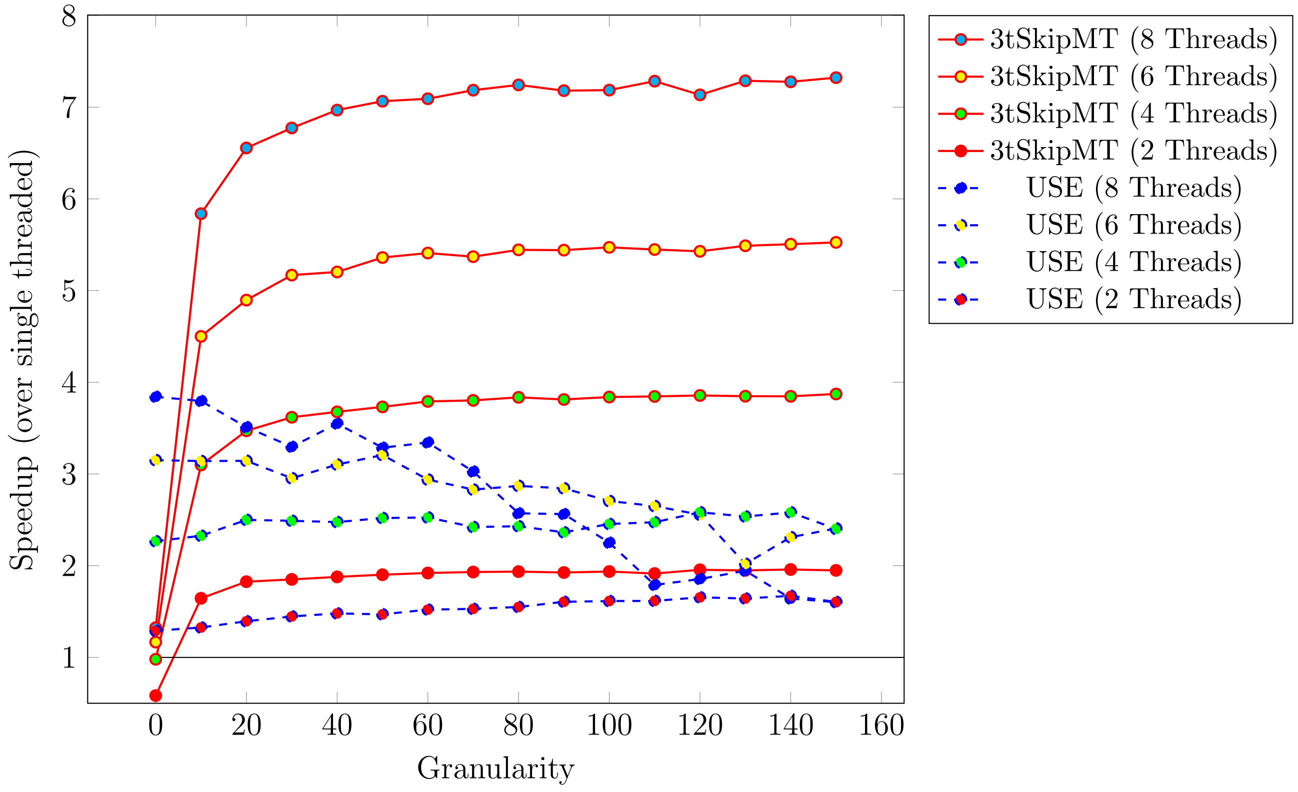


Figure 6.6: Comparative Analysis of *3tSkipMT* against the Ultimate Share Everything (USE) Simulator

	3tSkipMT					*USE*				
granularity	1 Thread	2 Thread	4 Thread	6 Thread	8 Thread	1 Thread	2 Thread	4 Thread	6 Thread	8 Thread
0	1.443	2.473	1.473	1.237	5.577	21.427	16.597	9.450	6.800	1.090
10	10.607	6.450	3.423	2.357	6.363	24.157	18.213	10.383	7.690	1.817
20	19.993	10.950	5.760	4.083	7.613	26.730	19.170	10.693	8.503	3.050
30	28.877	15.607	7.980	5.587	8.823	29.087	20.103	11.683	9.847	4.263
40	37.930	20.197	10.313	7.290	8.903	31.597	21.350	12.763	10.183	5.443
50	47.333	24.887	12.683	8.830	10.383	34.130	23.227	13.543	10.643	6.700
60	56.617	29.480	14.933	10.467	10.983	36.720	24.140	14.537	12.493	7.983
70	65.673	34.010	17.270	12.230	12.980	39.290	25.693	16.223	13.883	9.140
80	74.840	38.663	19.510	13.747	16.207	41.670	26.883	17.147	14.520	10.333
90	83.647	43.433	21.940	15.373	17.223	44.140	27.477	18.673	15.523	11.650
100	92.797	47.923	24.170	16.960	20.627	46.427	28.757	18.910	17.160	12.913
110	102.017	53.287	26.527	18.727	27.473	49.170	30.423	19.877	18.553	14.007
120	111.533	57.040	28.923	20.547	27.883	51.723	31.253	20.030	20.233	15.633
130	120.493	61.830	31.310	21.953	27.827	54.120	32.950	21.337	26.793	16.533
140	130.030	66.387	33.800	23.617	34.597	56.813	33.990	22.013	24.587	17.870
150	138.653	71.140	35.803	25.093	36.623	58.667	36.557	24.433	24.380	18.937

Figure 6.7: Runtimes for Comparative Analysis of *3tSkipMT* and *USE* in Seconds

Chapter 7

Conclusions

As high performance computing technology continues to grow, particularly in the context of shared memory multi-core parallelism, software platforms and research tools must adapt in order to fully take advantage these increased resources. Shared memory parallelization presents a particularly challenging problem as most sequential implementations are incompatible with concurrent programing due to assumptions that cannot be made when state is shared between multiple threads. Additionally, speedup from multi-threading can be severely bottlenecked by contention between threads, especially when access to shared state is near constant. This means applications that aim to take advantage of multi-core parallelism must utilize intelligent design to reduce contention between threads as much as possible without lose of data or accuracy.

Our research aimed to tackle this challenge in the context of Discrete Event Simulation (DES). Specifically, we implement the Three-Tier Skip List with Multi-Threading Support ($3tSkipMT$) pending event set data structure with respective kernel implementation for the Miami University Simulation Environment (MUSE). We explore many design obstacles associated with both a thread safe PES and its utilization in an existing optimistic parallel DES simulation kernel. This implementation was tested using the PHOLD benchmark to determine relative runtimes of synthetic DES simulations with $3tHeap$ by Higiro and our $3tSkipMT$ with various numbers of threads.

First, we implemented a lock-free concurrent priority queue by Lindén & Jonsson to serve as the backing structure to our pending event set. This queue maintains the state of the data structure and is non-blocking in nature. We made a novel modification to this queue by allowing the *deleteAt* operation for mid queue deletion of a particular key in the queue, critical to restructuring and rollback operations. Next, our implementation identified multiple key critical sections to the kernel pipeline inherent to DES. These includes agent state transitions, LGVT management, the top tier queue restructure, and garbage collection. These critical sections were accounted for and implemented with as little locking as possible to avoid overheads.

Our results demonstrate the efficacy of shared memory multi-threaded systems, though with some limitations. We achieve significant speedup compared to sequential equivalents in a realistic synthetic benchmark, and we find that multi-threading can be implemented without impacting the asymptotic complexity for large simulations with many LPs. We see an overhead associated with multi-threading that stems from both ① requiring a less efficient backing data structure that cannot be well optimized and ② contention that forms from many threads competing for shared memory. As more threads are introduced, this contention overhead begins out outweigh the increase in system resources.

We also identified granularity as a critical factor in reducing this contention overhead and realizing utility from multi-threading. Granularity decreases contention of threads by increasing the amount of processing that can be fully parallelized by multiple cores before needing to access shared state again. This means multi-threaded simulations scale significantly better than sequential simulations as a larger and larger granularity is introduced, however very low granularity can lead to minimal gains from multi-threading, even resulting in slowdown at very small values. In other words, speedup from multi-threading is most realized when a large amount of computing must be done between events. This also means that simple simulations with very little computation between events will be less likely to realize significant benefits from our design, even possibly experiencing slowdown.

Overall, our introductory investigation finds that shared memory parallelization has the potential to provide a range of benefits and increase the performance of DES simulations, however more research much be conducted before our design can be applied to a wide range of scenarios.

7.1 Future Work

Two primary bottlenecks limit the effectiveness of our described implementation:

1. The speed of the underlying concurrent priority queue data structure
2. Amount of contention that exists between threads

The biggest area for future research would be in the underlying concurrent priority queue. Our study uses the Lindén & Jonsson Skip List based queue. While effective, the linked-list structure of this queue prevents it from optimizing well and is inherently not as fast as a heap based implementation such as the one presented by higiro in $3tHeap$. Additionally, in order to make the queue work for our purposes, a novel modification had to be made that reduces the performance of our queue even further. A more efficient and fine-tuned backing queue has the potential to make our multi-threaded design significantly faster.

Additionally, more work can be done to further refine the algorithm on top of the underlying priority queue to further reduce thread contention on the queue. This would be done by reducing the number of times a thread must access the same queue as another thread, or by reducing the amount of time spent modifying the queue. This is especially true for the restructure operation, where currently restructures are conducted by simply removing and re-inserting a node from the queue, while a more fine-tuned operation could be possible.

One possible area for further analysis could be to begin measuring thread contention by the number of failed lock-free operations. That is, to keep track of the number of times a lock-free operation such as a $tryInsert$ function fails due to contention on threads. Attempting to minimize this value at low granularity values would be the objective of future research in this area.

Bibliography

[1] S. Jafer, Q. Liu, and G. Wainer, "Synchronization methods in parallel and distributed discrete-event simulation," *Simulation Modelling Practice and Theory*, vol. 30, pp. 54 – 73, 2013.

[2] R. Franceschini, P.-A. Bisgambiglia, and P. Bisgambiglia, "A comparative study of pending event set implementations for pdevs simulation," in *Proceedings of the Symposium on Theory of Modeling &#38; Simulation: DEVS Integrative M&#38;S Symposium*, DEVS '15, (San Diego, CA, USA), pp. 77–84, Society for Computer Simulation International, 2015.

[3] K. Muthalagu, "Threaded warped : An optimistic parallel discrete event simulator for cluster of multi-core machines," Master's thesis, 2012.

[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, (New York, NY, USA), pp. 3–14, ACM, 2017.

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

[6] O. Corporation, "Thread safety (multithreaded programming guide)," 2010.

[7] M. P. Herlihy, "Impossibility and universality results for wait-free synchronization," in *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, (New York, NY, USA), pp. 276–290, ACM, 1988.

[8] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), (New York, NY, USA), pp. 483–485, ACM, 1967.

[9] M. R. Gebre, "Muse: A parallel agent-based simulation environment," Master's thesis, 2009.

[10] R. Brown, "Calendar queues: A fast 0(1) priority queue implementation for the simulation event set problem," *Commun. ACM*, vol. 31, pp. 1220–1227, Oct. 1988.

[11] 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.*, vol. 15, pp. 175–204, July 2005.

[12] T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, (London, UK, UK), pp. 300–314, Springer-Verlag, 2001.

[13] H. Sundell and P. Tsigas, "Fast and lock-free concurrent priority queues for multi-thread systems," *Journal of Parallel and Distributed Computing*, vol. 65, no. 5, pp. 609 – 627, 2005.

[14] R. Marotta, M. Ianni, A. Pellegrini, and F. Quaglia, "A conflict-resilient lock-free calendar queue for scalable share-everything pdes platforms," in *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '17, (New York, NY, USA), pp. 15–26, ACM, 2017.

[15] S. Gupta and P. A. Wilsey, "Lock-free pending event set management in time warp," in *Proceedings of the 2Nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM PADS '14, (New York, NY, USA), pp. 15–26, ACM, 2014.

[16] J. Hay and P. A. Wilsey, "Experiments with hardware-based transactional memory in parallel simulation," in *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM PADS '15, (New York, NY, USA), pp. 75–86, ACM, 2015.

[17] D. Jagtap, N. Abu-Ghazaleh, and D. Ponomarev, "Optimization of parallel discrete event simulator for multi-core systems," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pp. 520–531, May 2012.

[18] C. D. Carothers, D. Bauer, and S. Pearce, "Ross: a high-performance, low memory, modular time warp system," in *Proceedings Fourteenth Workshop on Parallel and Distributed Simulation*, pp. 53–60, 2000.

[19] J. Wang, D. Jagtap, N. Abu-Ghazaleh, and D. Ponomarev, "Parallel discrete event simulation for multi-core systems: Analysis and optimization," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, pp. 1574–1584, June 2014.

[20] 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, (New York, NY, USA), pp. 59–70, ACM, 2015.

[21] T. Dickman, S. Gupta, and P. A. Wilsey, "Event pool structures for pdes on many-core beowulf clusters," in *Proceedings of the 1st ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM PADS '13, (New York, NY, USA), pp. 103–114, ACM, 2013.

[22] J. Gruber, "Practical concurrent priority queues," *CoRR*, vol. abs/1509.07053, 2015.

[23] J. Lindén and B. Jonsson, "A skiplist-based concurrent priority queue with minimal memory contention," in *Principles of Distributed Systems* (R. Baldoni, N. Nisse, and M. van Steen, eds.), (Cham), pp. 206–220, Springer International Publishing, 2013.

[24] M. Ianni, R. Marotta, D. Cingolani, A. Pellegrini, and F. Quaglia, "The ultimate share-everything pdes system," in *Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '18, (New York, NY, USA), pp. 73–84, ACM, 2018.