Multi-tier Priority Queues and 2-tier Ladder Queue for Managing Pending Events in Sequential and Optimistic Parallel Simulations

Julius Higiro, Meseret Gebre, and Dhananjai M. Rao
Computer Science & Software Engineering (CSE) Department
Miami University, 510 E. High Street, 205 Benton Hall
Oxford, Ohio 45056-6221
raodm@miamiOH.edu

ABSTRACT

The choice of data structure for managing and processing pending events in timestamp priority order plays a critical role in achieving good performance of sequential and parallel Discrete Event Simulation (DES). Accordingly, we propose and evaluate the effectiveness of our novel multi-tiered (2 and 3 tier) data structures and our 2-tier Ladder Queue, for both sequential and optimistic parallel simulations, on distributed memory platforms. Our assessments use (a fine-tuned version of) the Ladder Queue, which has shown to outperform many other data structures for DES. The experimental results based on 2,500 configurations of PHOLD benchmark show that our 3-tier heap and 2-tier ladder queue outperform the Ladder Queue by 10% to 50% in simulations, particularly those with higher concurrency per Logical Process (LP), in both sequential and Time Warp synchronized parallel simulations.

CCS Concepts

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

Keywords

Discrete Event Simulation (DES); Optimistic Parallel Simulation; Time Warp; Binary Heap; Fibonacci Heap; Ladder Queue

1. INTRODUCTION

Sequential and parallel DES are designed as a set of logical processes (LPs) or "agents" that interact with each other by exchanging and processing timestamped events or messages [6]. Events that are yet to be processed are called "pending events". Pending events must be processed by LPs in priority order to maintain causality, with event priorities being determined by their timestamps. Consequently, data structures for managing and prioritizing pending events play a critical role in ensuring efficient sequential and parallel simulations [3, 7, 4, 10]. Effectiveness of data structures for event management is a conspicuous issue in larger simulations, where thousands or millions of events can be pending [1, 9]. Overheads in managing pending events is magnified in fine grained simulations where the time taken to process an event is very short – *i.e.,* LPs use only few 100s to 1000s of instructions per event. Furthermore, the synchronization strategy used in PDES, Time Warp in particular, can further impact the effectiveness of the data structure due to additional operations required for rollback-based recovery.

SIGSIM-PADS'17, May 24–26 2017, Singapore

DOI: http://dx.doi.org/10.1145/3064911.3064921

1.1 Motivation

Many investigations have explored the effectiveness of a wide variety of data structures for managing the pending event set, as discussed in Section 5. Among the various data structures, the Ladder Queue proposed by Tang *et al* [10] has shown to be the most effective data structure for managing pending events [3, 2], particularly in sequential DES. Accordingly, we aimed to replace the heap-based data structures (discussed in Section 4) used in our Time Warp synchronized parallel simulator with the Ladder Queue. Section 4.5 discusses our Ladder Queue implementation and its fine-tuning.

The Ladder Queue outperformed our multi-tier heap-based data structures in certain sequential simulations, consistent with observations by other investigators [3, 10]. However, as detailed in Section 6, the Ladder Queue was substantially slower in two cases – ❶ high concurrency: larger number of concurrent events (*i.e.,* events with same timestamp) per LP, and ❷ Time Warp synchronized parallel simulations conducted on a distributed memory computing cluster. Conversely, our multi-tier data structures performed well in parallel simulations.

To provide a good balance for both sequential and optimistic parallel simulations, we propose a significant change to the design of the Ladder Queue. Our revised data structure, discussed in Section 4.6, is called 2-tier Ladder Queue (`2tLadderQ`). Various configurations of the standard PHOLD benchmark are used to assess the effectiveness of the multi-tier data structures vs. our fine-tuned implementation of the Ladder Queue. Results from our experiments discussed in Section 6 data shows `2tLadderQ` provides comparable perfor-

mance in sequential simulations but outperforms the Ladder Queue in optimistic parallel simulations. Our 3-tier heap (`3tHeap`) outperforms our `2tLadderQ` in high concurrency scenarios.

2. PARALLEL SIMULATOR OVERVIEW

The implementation and assessment of the different data structures has been conducted using our parallel simulation framework called MUSE. It has been developed in C++ and uses the Message Passing Interface (MPI) library for parallel processing. MUSE uses Time Warp and standard state saving approach to accomplish optimistic synchronization of the LPs. A conceptual overview of a parallel simulation is shown in Figure 1. A MUSE simulation is organized as a set of Logical Processes (LPs) that interact with each other by exchanging virtual timestamped events. The simulation kernel implements core functionality associated with LP registration, event processing, state saving, synchronization, and Global Virtual Time (GVT) based garbage collection.

The kernel uses a centralized Least Timestamp First (LTSF) scheduler queue for managing pending events and scheduling event processing for local LPs. With a centralized LTSF scheduler, event exchanges between local LPs do not cause rollbacks. Only events received via MPI can cause rollbacks. The scheduler is designed to permit different data structures to be used for managing pending events. This feature is used to experiment with the different pending event scheduler queues. A scheduler queue is required to implement the following key operations to manage pending events:

❶ **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.

❷ **Peek next event**: This operation returns the next event to be processed. The event is used to update an LP's LVT and schedule it. Note that peek does not dequeue events.

❸ **Dequeue events for next LP**: In contrast to peek, this operation dequeues concurrent events (*i.e.,* events with the same receive time) to be processed by an LP. Concurrent events could have been sent by different LPs on different MPI-processes. A total order within concurrent events is not imposed but can be readily introduced if needed.

❹ **Cancel pending events**: This operation is used as part of rollback recovery process to aggressively 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. This feature also reduces scans required to cancel events in Ladder Queue data structures discussed in Section 4.5 and Section 4.6.

2.1 Experimental Platform

The design of MUSE and the experiments reported in this paper were conducted using a distributed-memory compute cluster consisting of 80 compute nodes interconnected by 1 GBPS Ethernet. Each compute node has two quad-core Intel Xeon ® CPUs (E5520) running at 2.27 GHz with hyper-threading disabled. Each compute node has 32 GB of RAM

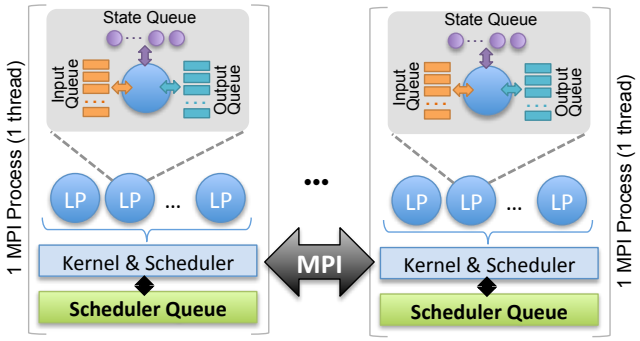


Figure 1: Overview of a parallel MUSE simulation

(4 GB per core) in Non-Uniform Memory Access (NUMA) configuration. The cluster has an independent 1 GBPS Ethernet network to support a shared file system. The nodes run Red Hat Enterprise Linux 6, with Linux (kernel ver 2.6.32) and the cluster runs PBS/Torque. The simulation software was compiled using GCC version 4.9.2 (`-O3` optimization level) with OpenMPI 1.6.4. All debug assertions were turned off for maximum performance.

3. PHOLD BENCHMARK

The experimental analysis have been conducted using a parallelized version of the classic Hold synthetic benchmark called PHOLD (see Section 7.1). It has been used by many investigators because it has shown to effectively emulate the steady-state phase of a typical simulation [3, 10]. Our PHOLD implementation developed using MUSE provides several parameters (specified as command-line arguments) summarized in Table 1. The benchmark consists of a 2-dimensional toroidal grid of Logical Processes (LPs) specified via the `rows` and `cols` parameters. The LPs are evenly partitioned across the MPI-processes used for simulation. The `imbalance` parameter influences the partition, with larger values skewing the partition as shown in Figure 2(a). The `imbalance` parameter has no impact in sequential simulations.

Table 1: Parameters in PHOLD benchmark

Parameter	Description
`rows`	Total number of rows in model.
`cols`	Total number of columns in model. #LPs = `rows` × `cols`
`eventsPerLP`	Initial number of events per LP.
`delay` or λ	Value used with distribution – Lambda (λ) value for exponential distribution *i.e.,* $P(x) = \lambda e^{-\lambda x}$.
`%selfEvents`	Fraction of events LPs send to self
`granularity`	Additional compute load per event.
`imbalance`	Fractional imbalance in partition to have more LPs on a MPI-process.
`simEndTime`	GVT when simulation logically ends.

The PHOLD simulation commences with a fixed number of events for each LP, specified 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. The impact of increasing

the `granularity` parameter (no unit) is summarized in Figure 2(b) – smaller values result in finer grained simulations. For each event, an LP schedules another event to a randomly chosen adjacent LP. The `selfEvents` parameter controls the fraction of events that an LP schedules to itself.

The event timestamps are determined by a given `delay--distrib` and `delay` or λ parameters. Our experiments use an exponential distribution for timestamps, because it has shown to reflect *steady state* event distribution commonly found in a broad range of simulation models (but not all models obviously) [10]. Timestamp of events is computed as $t_{recv} = \text{LVT} + 1 + \lambda e^{-\lambda x}$, where the +1 ensures that events are always scheduled into the future as per MUSE API requirement discussed in Section 2. The impact of changing the λ (*i.e.,* `delay`) is shown in Figure 2(c) – smaller values of λ provide a broader range of timestamp value for future events resulting in fewer concurrent events per LVT. Conversely, larger λ values cause timestamps to be close to the current epoch, increasing both the number of concurrent events per LVT and the possibility of rollbacks. Section 6 explores impact of these parameters on scheduler queue performance using 2,500 different configurations.

4. SCHEDULER QUEUES

The pending events are managed by different scheduler queues that utilize different data structures to implement the key operations discussed in Section 2, namely: enqueue, peek, dequeue, and cancel. In this study we have compared the effectiveness of 6 different non-intrusive queuing data structures (see Section 7.1 for link to source code) namely: ① binary heap (`heap`), ② 2-tier heap (`2tHeap`), ③ 2-tier Fibonacci heap (`fibHeap`), ④ 3-tier heap (`3tHeap`), ⑤ Ladder Queue (`ladderQ`), and ⑥ 2-tier Ladder Queue (`2tLadderQ`). The queues are broadly classified into two categories, namely: single-tier and multi-tier queues. Single-tier queues such as `heap` use only a single data structure for accomplishing the 4 key operations. Conversely, multi-tier queues use organize events into tiers, with each tier implemented using different data structures. Table 2 summarizes the asymptotic time complexities of the 6 data structures discussed in the following subsections.

4.1 Binary Heap (`heap`)

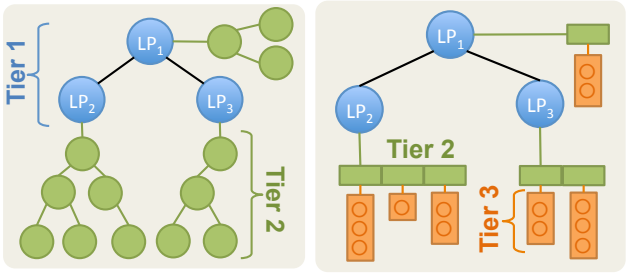
The binary heap based (`heap`) is a commonly used data structure for implementing priority queues. It is a single-tier data structure and is implemented using a conventional array-based approach. A `std::vector` is used as the backing container and C++11 algorithms (`std::push_heap`, `std::pop_heap`) are used to maintain the heap. The heap is prioritized on both timestamp and LP's *ID* (to dequeue batches of events), with lowest timestamp at the root of the heap. Operations on the heap are logarithmic in time complexity – given l LPs each with e events/LP, the time complexity of enqueue and dequeue operations is $O(\log(l \cdot e))$ as shown in Table 2. If event cancellation requires z events to be removed from the heap, the time complexity is $O(z \cdot \log(e \cdot l))$. Consequently, for long or cascading rollbacks the cancellation costs is high.

4.2 Two-tier Heap (`2tHeap`)

The `2tHeap` is designed to reduce the time complexity of cancel operations by subdividing events into two distinct tiers as shown in Figure 3. The first tier has containers for each local LP on an MPI-process. Each of the tier-1 containers contain a heap of events to be processed by a given LP. In `2tHeap` both tiers are maintained as independent binary heaps. Consequently, given l LPs and e pending events per LP, enqueue and dequeue operates require $O(\log e)$ time to insert in tier-2 followed by $O(\log l)$ time to reschedule the LP. Note that the tier-1 heap is updated only if the root event in tier-2 changes after an operation. Consequently, the best case time complexity becomes $\log e$ when compared to $O(\log(e \cdot l))$ for the `heap`. Furthermore, cancellation of events for an anti-message is restricted to just the tier-2 entries of LP_{dest} (see Section 2) with utmost 1 tier-1 operation to update schedule position of LP_{dest}. A `std::vector` is used as the backing storage for both tiers and standard algorithms are used to maintain the min-heap property for both tiers after each operation.

Table 2: Comparison of asymptotic time complexities (*i.e.,* Big O) of different data structures
Legend – l: #LPs, e: #events / LP, c: #concurrent events, z: #canceled events, ${}_{t2}k$: parameter, 1: amortized constant

Name	Enqueue	Dequeue	Cancel
`heap`	$\log(e \cdot l)$	$\log(e \cdot l)$	$z \cdot \log(e \cdot l)$
`2tHeap`	$\log(e \cdot l)$	$\log(e \cdot l)$	$z \cdot \log(e)+$
`fibHeap`	$\log(e) + 1$	$\log(e) + 1$	$z \cdot \log(e) + 1$
`3tHeap`	$\log(\frac{e}{c}) + \log(l)$	$\log(l)$	$e + \log(l)$
`ladderQ`	1	1	$e \cdot l$
`2tLadderQ`	1	1	$e \cdot l \div {}_{t2}k$



(a) 2-tier Heap (b) 3-tier Heap

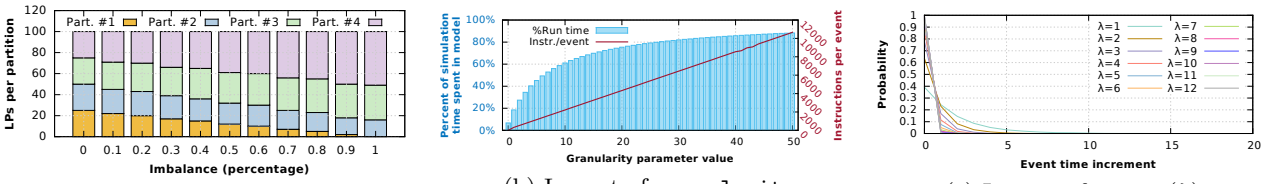
Figure 3: Structure of 2-tier & 3-tier heap

4.3 2-tier Fibonacci Heap (`fibHeap`)

The `fibHeap` is an extension to the previous `2tHeap` data structure and uses a Fibonacci heap for scheduling LPs. The Fibonacci heap is a slightly modified version from the BOOST C++ library. The Fibonacci heap has an amortized constant time for changing key values and finding minimum. Consequently, we use it for the first tier which is responsible for scheduling LPs and use a standard binary heap for the second tier. We do not use Fibonacci heap for the second tier because we found its runtime constants to be higher than a binary heap. Accordingly, the time complexity for enqueue and dequeue operations is $O(\log(e) + 1)$.

4.4 Three-tier Heap (`3tHeap`)

The `3tHeap` builds upon `2tHeap` by further subdividing the second tier into two tiers as shown in Figure 3(b). The binary heap implementation for the first tier that manages LPs for scheduling has been retained from `2tHeap`. However,



(a) Impact of `imbalance` (b) Impact of `granularity` (c) Impact of `delay` (λ)

Figure 2: Impact of varying key parameter values in the PHOLD model

the 2nd tier is implemented as a list of containers sorted based on receive time of events. Each tier-2 container has a 3rd tier list of concurrent events. Assuming each LP has c concurrent events on an average, there are $\frac{e}{c}$ tier-2 entries with each one having c pending events. Inserting events in the `3tHeap` is accomplished via binary search at tier-2 with time complexity $O(\log \frac{e}{c})$ followed by an append to tier-3, a constant time operation. Enqueue to tier-2 is followed by an optional heap fix-up of time complexity $O(\log l)$ as summarized in Table 2. Dequeue operation for a LP removes a tier-2 entry in constant time followed by a $O(\log l)$ heap fix-up for scheduling. Event cancellation has time complexity of $O(e + \log l)$ as it requires inspecting each event in tier-3 followed by heap fix-up. As an implementation optimization, we recycle tier-2 containers to reduce allocation and deallocation overhead.

4.5 Ladder Queue (`ladderQ`)

The `ladderQ` is a priority queue implementation proposed by Tang *et al* [10] with amortized constant time complexity as summarized in Table 2. Several investigators have independently verified that for sequential DES the `ladderQ` outperforms other priority queues, including: simple sorted list, binary heap, Splay tree, Calendar queue, and other multi-list data structures [2, 3, 10]. There are two key ideas underlying the Ladder Queue, namely: ① minimize the number of events to be sorted and ② delay sorting of events as much as possible. The multi-tier data structures also aim to minimize the number of events to be sorted. However, in contrast to the `ladderQ`, the other data structures always fix-up and maintain a minimum heap property.

The ladder queue consists of the following 3 substructures:

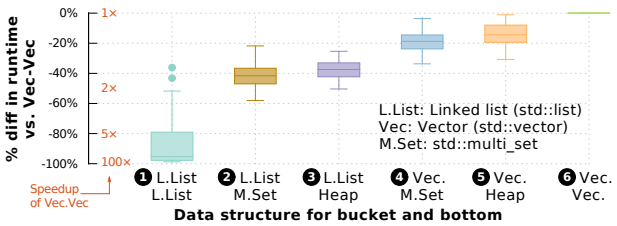
1. *Top*: An unsorted list which contains events scheduled into the distant future or epoch.
2. *Ladder*: Consists of multiple rungs, *i.e.,* list of buckets. Each bucket contains list of events with a finite range of timestamp values. Hence, although events within a bucket are not sorted, the buckets on a rung are organized in a sorted order. The `ladderQ` minimizes the number of events to be finally sorted by recursively breaking large buckets into smaller buckets in lower rungs of its ladder. Lower rungs in the ladder have smaller buckets with smaller time ranges.
3. *Bottom*: This substructure contains a sorted list of events to be processed. Inserts into *Bottom* must preserve sorted order. Hence, the `ladderQ` strives to maintain a short bottom by moving events back into the ladder, as needed [10].

4.5.1 Fine tuning Ladder Queue performance

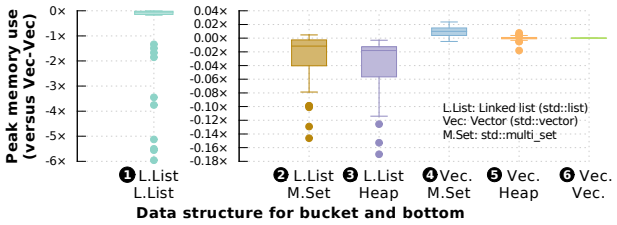
Our implementation closely followed the design in the original paper by Tang *et al* [10]. However, to minimize runtime constants, we have explored different configurations for the buckets and the *Bottom* in the `ladderQ`. Specifically, we have explored the following 6 configurations – ❶ L.List-L.List: using a doubly-linked list (L.List) implemented by `std::list`) for buckets and bottom. Events are inserted into bottom via linear search as proposed by Tang *et al.* ❷ L.List-M.Set: L.List for buckets and a Multi-set ($O(\log n)$ operations) for bottom, ❸ L.List-Heap: a L.List and a binary heap (backed by a `std::vector`) for bottom, ❹ Vec-M.Set: a dynamically growing array (*i.e.,* `std::vector`) for buckets and Multi-set bottom, ❺ Vec-Heap: Vector buckets and binary heap for bottom, and ❻ Vec-Vec: Vector for buckets and bottom. This configuration enables using quick sort (*i.e.,* `std::sort`) for sorting buckets and binary search for inserting events into bottom.

Runtime comparison of the 6 `ladderQ` configurations is summarized in Figure 4. The data was obtained using PHOLD with different parameter settings. The ❻th Vec-Vec configuration was the fastest and performance of other configurations are shown relative to it in Figure 4(a). The L.List-L.List configuration was generally the slowest and performed 85× (or ~98%) slower than the Vec-Vec configuration. The peak memory used for simulations is shown in Figure 4(b), in comparison with the Vec-Vec configuration. As shown by the charts in Figure 4, the increased performance of Vec-Vec comes at about a 6× increase in peak memory footprint when compared to L.List-L.List configuration. This increased footprint arises because the `std::vector` internally doubles its capacity as it grows. With many buckets in the `ladderQ`, each implemented using a `std::vector`, the overall peak memory footprint is higher. Certainly, the increased capacity is used if the number of events in buckets grow. However, the Vec-M.Set and Vec-Heap configurations consume a bit more memory in some configurations, showing that Vec-Vec is not the worst in memory consumption. Consequently, we use the Vec-Vec configuration as it provides the fastest performance among the 6 configurations (see Section 7.1).

The maximum number of rungs in the *Ladder* also influences the overall performance of the `ladderQ` [10]. The chart in Figure 5 illustrates the impact of limiting the maximum number of rungs in the `ladderQ`. When the rungs are too few, the timestamp-based width of buckets is larger and more events with many different timestamps are packed into buckets. This also causes the *Bottom* to be longer with events spanning a broader range of timestamps. Consequently, when inserts happen into *Bottom*, many *Bottom*-to-*Ladder* re-bucketing operations are triggered to ensure bottom is short. These re-bucketing operations with many events significantly degrade performance. However, once sufficient number of rungs (6 rungs in this case) are permitted the events are better subdivide into smaller timestamp-



(a) Comparison sequential runtimes



(b) Peak memory used

Figure 4: Comparison of execution time and peak memory for PHOLD benchmark (different parameter settings) using 6 different `ladderQ` configurations

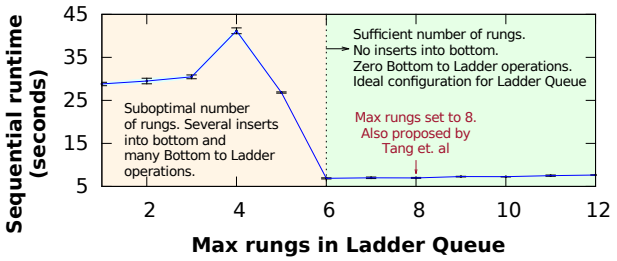


Figure 5: Impact of limiting rungs in Lader

based bucket widths. Small bucket widths in turn minimize inserts into bottom and *Bottom*-to-*Ladder* operations, ensuring good performance.

The chart in Figure 5 shows that a minimum of 6 rungs is required. For some select configurations of larger models we observed (data not shown) that 5 rungs would be sufficient. However, the number of rungs cannot exceed beyond a threshold to avoid infinite spawning of rungs [10]. Moreover, it limits the overheads involved in re-bucketing events from rung-to-rung [10]. Accordingly, based on the observations in Figure 5, we decided to adopt a maximum of 8 rungs, consistent with the threshold proposed by Tang *et al* [10]. Furthermore, we trigger *Bottom*-to-*Ladder* re-bucketing only if the *Bottom* has events at different timestamps to further reduce inefficiencies.

4.5.2 *Shortcoming of Ladder Queue for optimistic PDES*

The amortized constant time complexity of enqueue and dequeue operations enable the `ladderQ` to outperform other data structures in sequential simulations [2, 3, 10]. However, canceling events, requires a linear scan of pending events because *Top* and buckets in rungs are not sorted. In practice, scans of *Top*, *Ladder* rung buckets, and *Bottom* can be avoided based on cancellation times. Nevertheless, in a general case, event cancellation time complexity is proportional to the number of pending events – *i.e.,* $O(e \cdot l)$ as summarized in Table 2. This issue is exacerbated in large simulations where thousands of events are typically present in *Top* and buckets in various rungs.

In this context, it is important to recollect from Section 2 that – as an optimization, MUSE utilizes only one anti-message to from LP_{sender} to LP_{dest} to cancel all n events sent after $t_{rollback}$ (rather than sending n individual anti-messages) which reduces overheads. Furthermore, with our centralized scheduler design, only events received from LPs on other MPI-processes can trigger rollbacks. Consequently, the number of scans of the `ladderQ` that actually occurr is significantly fewer in our case, despite the aggressive cancellation strategy.

4.6 2-tier Ladder Queue (`2tLadderQ`)

A key shortcoming of the Ladder Queue for Time Warp based optimistic PDES arises from the overhead of canceling events used for rollback recovery. Our experiments (see Section 6) show that event cancellation overhead of `ladderQ` is a significant bottleneck in parallel simulation. On the other hand, our multi-tier data structures, where pending events are more organized, performed well.

Consequently, to reduce cost of event cancellation, we propose a 2-tier Ladder Queue (`2tLadderQ`) in which each bucket in *Top* and *Ladder* is further subdivided into ${}_{t2}k$ sub-buckets, where ${}_{t2}k$ is specified by the user. Figure 6 illustrates an overview of the `2tLadderQ` with ${}_{t2}k = 3$ sub-buckets in each bucket. Given a bucket, a hash of the sending LP's ID (or the receiver LP ID, one or the other but not both) is used to locate a sub-bucket into which the event is appended. Currently, we use a straightforward LP_{sender} modulo ${}_{t2}k$ as the hash function. Consequently, enqueue involves just 1 extra modulo instruction over regular `ladderQ` and hence retains its amortized constant time complexity. Similar to buckets, the sub-buckets are implemented using standard `std::vector` with events added or removed only from the end to ensure amortized constant-time operation.

The dequeue operations for a bucket require iterating over each sub-bucket. However, for a small, fixed value of ${}_{t2}k$, the overhead becomes an ammortized constant. The constant overhead is determined by the value of ${}_{t2}k$. Consequently, dequeue also retains the amortized constant characteristic from regular `ladderQ` as summarized in Table 2. Currently, we do not subdivide *Bottom* but leave it as a possible future optimization (link to source code in Section 7.1).

4.7 Performance gain of `2tLadderQ`

The primary performance gain for `2tLadderQ` arises from the reduced time complexity for event cancellation. Since each bucket is sub-divided, only $1 \div {}_{t2}k$ fraction of events need to be checked during cancellation. For example, if ${}_{t2}k$=32, only $\frac{1}{32}$ of the pending events are scanned during cancellation. This significantly reduces the time constants in larger simulations enabling rapid rollback recovery.

The value of ${}_{t2}k$ is a key parameter that influences the overall constants in `2tLadderQ`. For sequential simulation, where event cancellations do not occur, we recommend ${}_{t2}k$=1. With this setting the performance of `2tLadderQ` is very close to that of the regular `ladderQ`. However, in parallel simulation, the value of ${}_{t2}k$ must be greater than 1 to realize benefits of its design. Figure 7 shows the effect of chang-

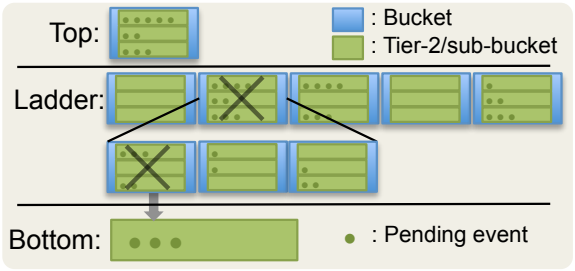


Figure 6: Structure of 2-tier Ladder Queue (`2tLadderQ`) with 3 sub-buckets / bucket (*i.e.*, $_{t2}k$=3)

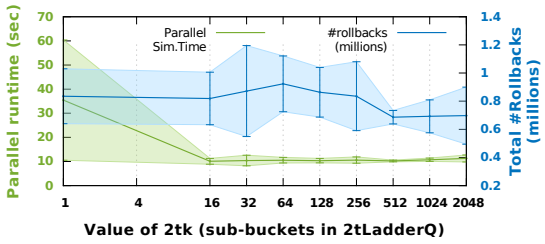


Figure 7: Effect of varying $_{t2}k$

ing the size of $_{t2}k$ in a parallel simulation with 16 MPI processes. The total rollbacks in the simulations were with 10% (except for $_{t2}k$=512, which for this model experienced fewer rollbacks). Nevertheless, for $_{t2}k$=1, the simulation has *much* higher runtime due to event cancellation overheads. The runtime dramatically decreases as $_{t2}k$ is increased. The runtime remains comparable for a broad range of values, namely: $64 \leq {}_{t2}k < 512$. However, for $_{t2}k \geq 512$, we noticed slow increase in runtime due to overhead of larger sub-buckets. Consequently, we have used a value of $_{t2}k$=128 for parallel simulation. We anticipate $_{t2}k$ value to vary depending on the hardware configuration of the compute cluster used for parallel simulation.

5. RELATED WORK

This paper proposes and explores multi-tier data structures for managing the pending event set in sequential and optimistic parallel simulations. Specifically, we compare effectiveness of the data structures against our fine-tuned version of the Ladder Queue [10] because it has shown to be very efficient for sequential Discrete Event Simulation (DES). Recently, Franceschini *et al* [3] compared several priority-queue based event list data structures to evaluate their performance in the context of sequential DEVS simulations. They found that the Ladder Queue outperformed every other priority queue based event lists data structure such as Sorted List, Minimal List, Binary Heap, Splay Tree, and Calendar Queue. We refer readers to the work by Tang *et al* [10] and Franceschini *et al* [3] for comparative discussion on the different data structures. They both use the classic Hold benchmark used in this study.

In contrast to earlier work, rather than using a linked list based implementation, we propose alternative implementation using dynamically growing arrays (*i.e.*, `std::vector`). Furthermore, we trigger *Bottom* to *Ladder* re-bucketing only if the *Bottom* has events at different timestamps to reduce inefficiencies. Our 2-tier Ladder Queue (`2tLadderQ`) is a novel enhancement to the Ladder Queue to enable its efficient use in optimistic parallel simulations.

Dickman *et al* [2] compare event list data structures that consisted of Splay Tree, STL Multiset and Ladder Queue. However, the focus of their paper was in developing a framework for handling event list data structures in shared memory PDES. A central component of their study was the identification of an appropriate data structure and design for the shared event list. Gupta *et al* [4] extended their implementation of Ladder Queue for shared memory Time Warp based simulation environment, so that it supports lock-free access to events in the shared event lists. The modification involved the use of an unsorted lock-free queue in the underlying ladder queue structure. Quaglia [8] proposes a Low-Overhead Constant-Time (LOCT) secheduler that uses tree-like bitmaps which enables quick retrieval of events to be scheduled in a Time Warp simulator. Quaglia's experiments on multithreaded, shared memory architecture shows that the LOCT scheduler can outperform ladder queue, but the ladder queue has better overall efficiency [8]. Marotta *et al* [7] have contributed to the study of event list data structures in threaded PDES through the design of the Non-Blocking Priority Queue (NBPQ) data structure. An event list data structure that is closely related to Calendar Queues with constant time performance.

In contrast to aforementioned efforts, this paper focuses on distributed memory platforms in which each parallel process is single threaded. Consequently, our implementation does not involve thread synchronization issues. However, our 2-tier design has the ability to further reduce lock contention issues in multithreaded environments and could provide further performance boost. To the best of our knowledge, at the time of this paper, the Fibonacci heap (`fibHeap`) and our 3-tier Heap (`3tHeap`) are unique data structures that have potential to be effective in simulations with high concurrency.

6. EXPERIMENTS & DISCUSSIONS

Assessments of the effectiveness of the six scheduler queues from Section 4 have been conduced using different configurations of the PHOLD benchmark discussed in Section 3. The experiments were conducted on the distributed memory compute cluster described in Section 2.1. Our initial experimental analysis proved to be time consuming due to the large number of PHOLD parameters (see Table 1) and combinations of their values. Consequently, we pursued strategies to focus on most influential PHOLD parameters that impacted relative performance of the scheduler queues using Generalized Sensitivity Analysis (GSA) [5]. Section 6.1 discusses GSA experiments used to reduce the PHOLD parameter space and subsequent PHOLD configurations, called `ph3`, `ph4`, and `ph5`, used for further experiments. Section 6.2 and Section 6.3 discuss the results from sequential and parallel simulations conducted using `ph3`, `ph4`, and `ph5`.

6.1 Parameter reduction via GSA

Generalized Sensitivity Analysis (GSA) is based on two-sample Kolmogorov-Smirnov Test (KS-Test) and yields a $d_{m,n}$ statistic that is sensitive to differences in both central tendency and differences in the distribution functions of parameters [5]. The $d_{m,n}$ statistic is the maximum separation between cumulative probability distribution observed in a two-sample KS-Test. The KS-Test is performed with data

from Monte Carlo simulations involving combinations of parameter values from a specified range or probability distribution. The simulation result is then classified into number of "success" (m) or its converse "failure" (n) to compute cumulative probability distribution and $d_{m,n}$ statistic for each parameter. In this study we have defined "failure" to be parameter values for which the `2tLadderQ` runs slower when compared to another scheduler queue. For sequential and parallel simulations we use ${}_{t2}k$=1 and ${}_{t2}k$=128 respectively.

An important aspect of GSA is to ensure that the values for each parameter covers its full range of values. Consequently, we use Sobol random numbers to select a combination of PHOLD parameter values to be used for simulation. Sobol random numbers are quasi-random low-discrepancy sequences that provide uniform coverage of a multidimensional parameter space for PHOLD (see Figure 2). Our parameter ranges also ensure that the peak memory consumption do not cross NUMA threshold, which in our case is 4 GB of RAM. Exceeding the 4 GB NUMA threshold introduces a lot of variance in runtimes requiring many runs to reduce variance to acceptable limits.

The randomly (using Sobol sequences) selected parameter set is used to run the model using two different scheduler queues. Average simulation execution time from 3 different replications is recorded for each scheduler queue along with the parameter-set. The process is repeated for 2,500 different Sobol sequences. The 2,500 data set is then collectively analyzed to compute the $d_{m,n}$ statistics for the different parameters. The results from sequential and parallel GSA are discussed in the following subsections.

6.1.1 GSA results for Sequential simulations

The charts in Figure 8 shows the cumulative m, n, and the $d_{m,n}$ statistics for the 9 different parameters explored using GSA for sequential simulations. The orange impulses show the parameter values and number of samples used for Monte Carlo simulation. Note that the distribution of samples varies depending on the nature of the parameter – *i.e.*,, `eventsPerLP` varies in discrete steps of 1 from 1–20 while `imbalance` varies from 0 to 1.0 in small fractional steps.

The chart in Figure 9 shows the summary of the $d_{m,n}$ statistic or influence of each parameter (see Table 1) on the outcome – *i.e.,* `2tLadderQ` performs better or worse than `3tHeap`. The lightly shaded bands show the 95% Confidence Intervals (CI) computed using standard bootstrap approach using 5000 replications with 1000 samples in each. As expected, the `imbalance` (*i.e.,* skew in partition) has no impact in sequential simulation and has a low impact score of 0.037. Similarly, the GVT computation rate does not impact pending events and consequently its influence is low at 0.051.

Interestingly, other model parameters such as `rows`, `cols`, `self-events`, `simEndTime`, and `granularity` have no influence on relative performance of `2tLadderQ` vs `3tHeap`. The parameter with most influence is `eventsPerLP` with a score of 0.774. This parameter determines total number of concurrent events which influences bucket sizes and number of rungs in `2tLadderQ` as well as the third tier size in `3tHeap`. The parameter λ for exponential distribution has a marginal influence because it influences number of concurrent events as discussed in Section 3 and shown in Figure 2(c).

We have also conducted GSA to determine influential parameters impacting performance of other scheduler queues versus the `2tLadderQ` in sequential simulations (charts via

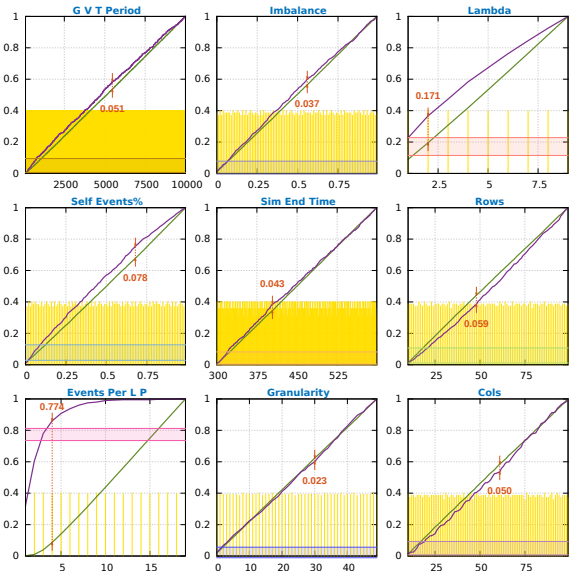


Figure 8: Results from Generalized Sensitivity Analysis (GSA) comparing `3tHeap` and `2tLadderQ` for sequential simulation (see Section 7.1 for more stats)

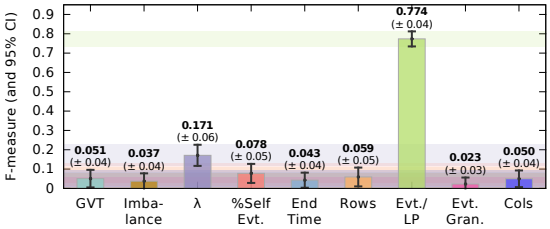


Figure 9: Summary of influential parameters from Figure 8 that cause performance differences between `2tLadderQ` and `3tLadder` in sequential simulations.

link in Section 7.1). Our analysis showed that none of the parameters play an influential role and the `2tLadderQ` performed consistently better or the same when compared to `ladderQ`, `2tHeap`, `fibHeap`, and `heap`. Only `3tHeap` and in few cases `2tHeap` outperformed our `2tLadderQ` in certain configurations. The performance of `ladderQ` and `2tLadderQ` was practically indistinguishable in sequential simulations (with ${}_{t2}k$=1).

Summary: GSA shows that for comparing event queue performance in sequential simulations using our PHOLD benchmark, we just need to focus on 1 or 2 parameters. Other aspects such as: model size, event granularity, fraction of self-events, GVT rate, etc., do not matter for comparison of scheduler queues. The scheduler queues to focus further analysis are: `ladderQ`, `2tLadderQ`, and `3tHeap`.

6.1.2 GSA results for Parallel simulations

GSA for parallel simulations were conducted using the same procedure discussed earlier but using 4 MPI-processes for parallel simulation. These analysis focused only on `ladderQ`, `2tLadderQ`, and `3tHeap` based on the inferences drawn from the earlier analyses. The average simulation execution time from 3 replications is recorded for each scheduler

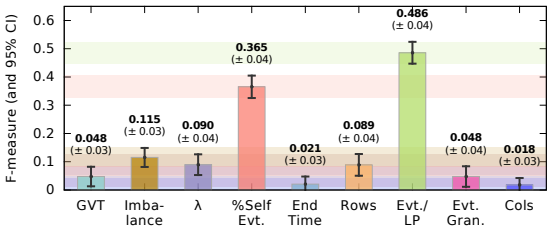


Figure 10: GSA data from parallel simulations (4 MPI-processes) showing influential parameters (`2tLadderQ` vs. `3tHeap`).

queue along with the parameter set. Initially, we observed that the `ladderQ` timings showed a lot of variance in runtime depending on number of rollbacks that occur. Consequently, to reduce variance, we have used a time-window of 10 time-units to curtail optimism and reduce rollbacks. The time-window restricts the simulation kernel from scheduling events that are more than 10 time-units ahead of GVT. We use the same time-window for all scheduler queues for consistent comparison and analysis.

The chart in Figure 10 shows the summary of the $d_{m,n}$ statistic or influence of each parameter (see Table 1) on the outcome – *i.e.,* `2tLadderQ` performs better or worse than `3tHeap`. The lightly shaded bands show the 95% Confidence Intervals (CI) computed using standard bootstrap approach using 5000 replications with 1000 samples in each. The parallel results are consistent with the sequential results and the `eventsPerLP` is the most influential parameter. However, in parallel simulation, the percentage of `selfEvents` (*i.e.,* LPs schedule events to themselves) has a more pronounced influence when compared to λ. The increased impact of `selfEvents` arises due to the use of optimistic synchronization. The self-events are local and can be optimistically processed, with some being rolled back, causing more operations on a larger pending event set. The data also shows that conspicuous `imbalance` in partitioning or load balance has some influence on the outcomes. However, in this study we explore typical parallel simulation scenarios in which load is reasonably well balanced.

6.1.3 PHOLD *configurations for further analysis*

The Generalized Sensitivity Analysis (GSA) enables identification of influential parameters, thereby substantially reducing the parameter space. However, GSA data does not provide an effective data set to analyze trends, such as: scalability, memory usage, rollback behaviors, etc. In order to pursue such analysis we have used 3 different PHOLD configurations called `ph3`, `ph4`, and `ph5`. The fixed characteristics for the 3 configurations with non-influential parameters is summarized in Table 3. We use larger simulation end times for parallel simulation so obtain sufficiently long runtimes using 32 cores. The value of influential parameters, namely: `eventsPerLP`, `%selfEvents`, and λ is varied for comparing different settings, similar to the approach used by other investigators [10, 3].

6.2 Sequential simulation results

Sequential simulations were conducted to assess the effectiveness of the different data structures. We pursued sequential simulations to compare the base case performance of the data structures, consistent with prior investigations [10, 3]. The sequential simulations also serve as a reference for potential use in conservatively synchronized PDES. The sequential experiments were conducted using 3 PHOLD configurations (see Section 6.1.3) on one compute node of our cluster described in Section 2.1. The simulations use only 1 MPI-process and states are not saved. Number of sub-buckets in `2tLadderQ` was set to 1, *i.e.,* ${}_{t2}k$=1. For these experiments, the influential parameters `eventsPerLP`, λ, and `%selfEvents` were varied to explore their impact on relative performance of the data structures. Event `granularity` was set to zero resulting in a fine grained simulation. For each configuration, data from 10 independent replications were collected and analyzed.

Table 3: Configurations used for further analysis

Name	#LPs (Rows×Cols)	Sim. End Time Seq	Parallel
`ph3`	1,000 (100×10)	5000	20000
`ph4`	10,000 (100×100)	500	5000
`ph5`	100,000 (1000×100)	100	1000

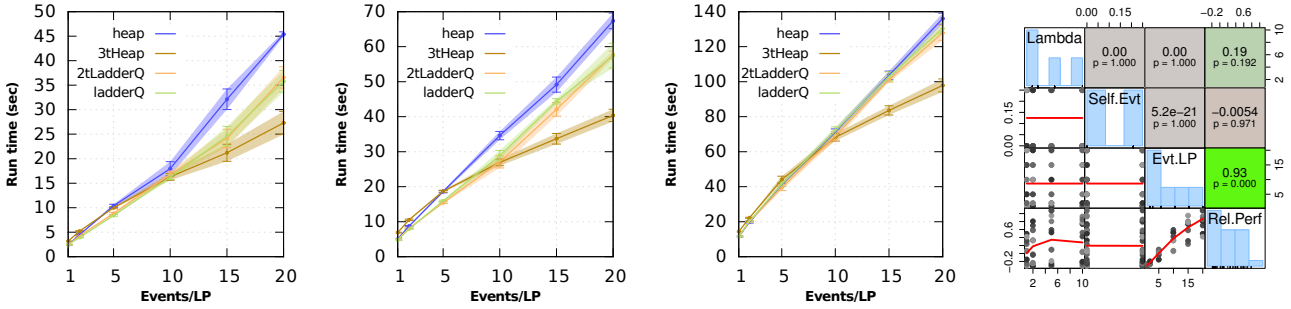
The charts in Figure 11(a)–(c) show change in runtime characteristics as the most influential parameter `eventsPerLP` is varied, for λ=1 (widest range of timestamps) and `%selfEvents` = 0.25. This configuration was generally the best for `ladderQ`. As illustrated by Figure 11(a)–(c), the performance of `ladderQ` and `2tLadderQ` (${}_{t2}k$=1) is comparable as expected. However, the `2tLadderQ` performs slightly (paired t-test p-value $\ll 0.05$, *i.e.,* averages are not equal) better in some cases possibly due to improved caching resulting from smaller tier-2 sub-buckets. These two queues outperform the other queues for lower values of `eventsPerLP`.

However, the `3tHeap` generally outperforms the other queues (except for `2tHeap` in some cases) for higher values of `eventsPerLP`. In all cases, there were no inserts into *Bottom* or *Bottom*-to-*Ladder* operations (discussed in Section 4.5.1) that degrade `ladderQ` performance. The size of the *Bottom* rung was proportional to the number of LPs and `eventsPerLP` – *i.e.,* with larger models, *Bottom* has more events for many LPs with the same timestamp to be scheduled. In the larger configurations, the maximum of 8 rungs were fully used. The maximum rung threshold of 8 was determined to be an effective setting as discussed in Section 4.5.1 and the same value proposed by Tang *et al* [10].

Profiler data (see Section 7.1) showed that the bottleneck in `ladderQ` arises from the overhead of re-bucketing events from rung-to-rung of the Ladder. On the other hand, in `3tHeap` re-bucketing does not occur. Consequently, the overheads of O(log $\frac{e}{c}$) operations in `3tHeap` are amortized as number of concurrent events c increases.

The chart in Figure 11(d) shows the correlation between the 3 influential parameters and the performance difference between `3tHeap` and `ladderQ`. Consistent with the GSA results, the corellogram shows that the most influential parameter is `eventsPerLP` (R=0.93, p=0) followed by λ (R=0.19, p=0.192) with a very weak corellation. The `%selfEvents` has practically no impact on performance. The corellogram also shows that these parameters are independent and have no covariance between each other (R~0, p >0.95).

The charts in Figure 12 shows the peak memory usage



(a) ph3 (b) ph4 (c) ph5 (d) Corellogram

Figure 11: Sequential simulation runtimes and correlation of 3tHeap performance with PHOLD parameters

corresponding to the runtime data in Figure 11. The memory size reported is the "Maximum resident set size" value reported by GNU `/usr/bin/time` command on Linux. The memory usage of `heap` is the lowest in most cases. Since $_{t2}k$=1, the memory usage of `ladderQ` and `2tLadderQ` is comparable as expected. The `3tHeap` initially uses more memory than the other data structures because of many small `std::vector`s and due to `std::vector` doubling its capacity. However, the memory usage is amortized as the `eventsPerLP` increases. Consequently, the improved performance of `3tHeap` over `ladderQ` is realized without significant increase in memory footprint.

6.3 Parallel simulation assessments

The sequential simulation assessments indicated that `ladderQ`, `2tLadderQ`, and `3tHeap` performed the best for a broad range of PHOLD parameter settings. Consequently, we focused on assessing the effectiveness of these 3 queues for Time Warp synchronized parallel simulations. The experiments were conducted on our compute cluster (see Section 2.1) using a varying number of MPI-processes, with one process per CPU-core. In order to ensure sufficiently long runtimes with 32-cores, we increased `simEndTime` for parallel simulations as tabulated in Table 3. The following subsections discuss results from the experiments.

6.3.1 *Throttling optimism with a time-window*

Initially we conducted experiments with fine-grained setting (*i.e.,* `granularity` = 0) from sequential simulations. We noticed that the `ladderQ` had a large variance in runtimes, particularly when it experienced many rollbacks. In several cases, cascading rollbacks significantly slowed the simulations – *i.e.,* `ladderQ` simulations required over 1 hour while `2tLadderQ` would consistently finish in a few minutes. In order to avoid such debilitating rollback scenarios for `ladderQ` and to streamline experimental analysis timeframes (otherwise we would have to run 100s of replications for each configuration to reduce variance) we have throttled optimism using a time-window of 10 time-units. The time-window restricts the simulation kernel from scheduling events that are more than 10 time-units ahead of GVT – *i.e.,* the kernel spins (without optimistically processing pending events but performing other operations) waiting for GVT to advance. The time-window value of 10 is 50% of the maximum timestamp of events generated by exponential distribution with $\lambda = 1$. Consequently, most events in current schedule cycle will fit within this time-window with limited impact on concurrency. We use the same time-window for all scheduler queues for consistent comparison and analysis.

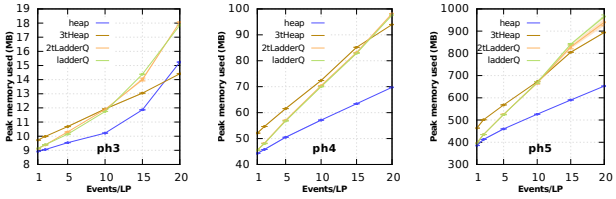
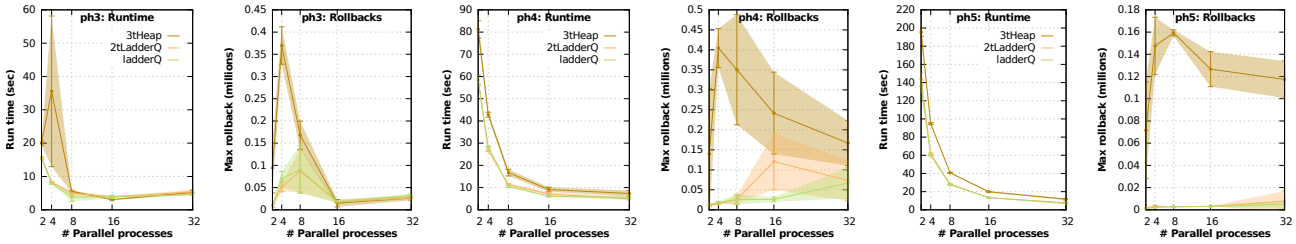


Figure 12: Comparison of peak memory usage

6.3.2 *Efficient case for* `ladderQ`

The charts in Figure 13 show key simulation statistics for low value of `eventsPerLP` = 2 and λ=1 for which `ladderQ` performed well, consistent with the observations in sequential simulations. The statistics show average and 95% CI computed from 10 independent replications for each data point. The peak rollbacks among all of the MPI-processes is shown as it controls overall progress in the parallel simulations. As illustrated by the data in Figure 13, both the `ladderQ` and `2tLadderQ` perform well for all three models. In this configuration, overall the `ladderQ` experienced the fewest rollbacks. Nevertheless, the `2tLadderQ` continues to perform well despite experiencing more rollbacks as shown in Figure 13(b). The good performance of `2tLadderQ` under heavy rollback is consistent with its design objective to enable rapid event cancellation and improve rollback recovery. The maximum of 8 rungs on the ladder was reached in all the simulations, but with only few (1 to 3) buckets per rung. On average, the number of *Bottom* to Ladder operations (that degrade performance) were low per MPI process, about – `ph3`: {9144, 8911}, `ph4`: {1904, 1448}, and `ph5`: {53, 84} for {`ladderQ`, `2tLadderQ`} respectively. We did not observe a strong correlation between number of these operations and rollbacks (see Section 7.1 for additional statistics).

In this configuration, the `3tHeap` runs experienced a lot of rollbacks when compared to the other two queues despite the time-window. For `ph5` data in Figure 13(c), `3tHeap` experienced about 114805 rollbacks on average while `ladderQ` experienced only 2341, almost 50×fewer rollbacks. Consequently, it was slower than the other 2 queues, but its performance is not significantly degraded – ~1.5× slower despite 50× more rollbacks. The peak memory usage for all the 3 queues was comparable in these configurations.



(a) ph3 time & rollbacks (b) ph4 time & rollbacks (c) ph5 time & rollbacks

Figure 13: Statistics from parallel simulation with eventsPerLP=2, $\lambda = 1$, %selfEvents=25%

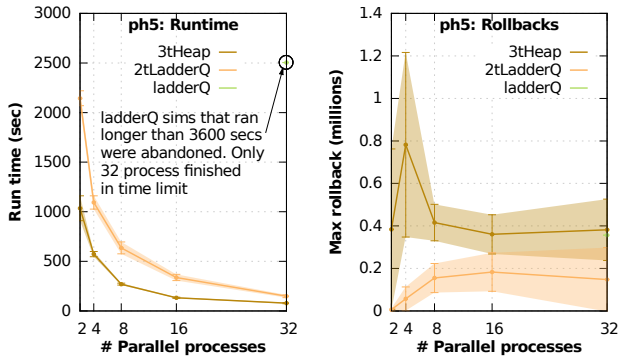


Figure 15: ph5 Statistics (best case for 3tHeap)

6.3.3 *Knee point for* 3tHeap *vs.* ladderQ

The charts in Figure 14 show key simulation statistics for the configuration where 3tHeap and ladderQ performed about the same in sequential (see Figure 11). For ph3, both ladderQ and 2tLadderQ experienced comparable number of rollbacks but the 2tLadderQ performs better due to its design advantages. In the case of ph4 and ph5, both the ladderQ and 3tHeap experienced a comparable number of rollbacks, but much higher than the 2tLadderQ despite having a time-window. Nevertheless, the 3tHeap conspicuously outperforms the ladderQ because it is able to quickly cancel events and complete rollback processing. For ph5, the 3tHeap outperforms the other 2 queues despite the high number of rollbacks. The peak memory usage for all the 3 queues was comparable in these configurations.

6.3.4 *Best case for* 3tHeap

Figure 15 shows simulation time and rollback characteristics in high concurrency configuration with ph5, with eventsPerAgent=20, λ=10, and %Self Evt.=25%. The ladderQ runs exceeded 3600 seconds in most cases even with a time-window, except for 32 processes. Consequently ladderQ experiments with fewer than 32 processes were abandoned. On the other hand 2tLadderQ performed well due to its design. The 3tHeap outperformed the other 2 queues despite experiencing 2× more rollbacks.

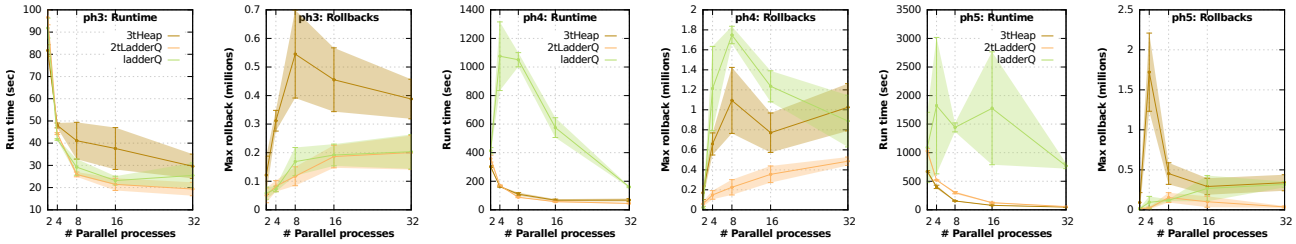
7. CONCLUSIONS

Efficient data structures, *i.e.,* priority queues for managing pending event sets play a critical role in overall performance of both sequential and parallel simulations. In the context of this study, we broadly classified the queues into single-tiered (heap) or multi-tiered (2tHeap, fibHeap, 3tHeap, ladderQ, and 2tLadderQ) data structures based on their design. Multi-tier data structures organize pending events into tiers, with each tier possibly implemented differently. Organizing events into multiple tiers decouples event management and Logical Process (LP) scheduling permitting different algorithms and data structures to suit the different needs.

The comparative analysis used a significantly fine-tuned version of the Ladder Queue (ladderQ) [10]. The objective of fine-tuning was to reduce the runtime constants of the ladderQ without significantly impacting its amortized $O(1)$ time complexity. Reduction in runtime constants is primarily realized by minimizing memory management overheads – *i.e.,* ❶ favor few bulk operations via std::vector than many small linked list nodes in std::list and ❷ recycle memory or substructures rather than reallocating them. Using std::vector (*i.e.,* dynamically growing array) enables use of algorithms with lower time constants, such as: std::sort, over std::multiset or binary heaps. The bulk memory operations do consume additional memory, but our analysis shows that the performance gains significantly outweigh the extra memory used. Ergo other simulation kernels can significantly improve overall performance by replacing linked lists with dynamically growing arrays.

One challenge that arose during design of experiments was exploring the large multidimensional parameter space in the PHOLD synthetic benchmark. Large parameter spaces may also arise with actual simulation models. We propose the use of Generalized Sensitivity Analysis (GSA) to reduce the parameter space. We also propose the use of Sobol random numbers to enable consistent exploration of the parameter space. GSA does require many simulations to be run to fully explore the parameter space. In our case, we ran 2,500 × 3 = 7,500 replications. However, GSA was able to significantly narrow the parameter space, *i.e.,* from 9 down to 2, in a scientific manner. GSA data shows that concurrency per LP indicated by eventsPerLP parameter (*i.e.,* batch of events scheduled per LP), plays the most dominant role in our benchmark. The data was cross-verified using correlograms from longer simulations. Similar GSA analysis can be applied to other models and benchmarks enabling consistent and focused analyses.

The sequential and parallel simulation results showed that 2tLadderQ performs no worse than our fine-tuned ladderQ in sequential simulations (with $_{t2}k$=1). Furthermore, our 2tLadderQ outperforms our ladderQ in parallel simulations because of its design that enables rapid cancellation of events during rollbacks. In fact, the ladderQ required aggressive throttling of optimism without which ladderQ was impractical to use in scenarios with many cascading rollbacks. These



(a) `ph3` time & rollbacks (b) `ph4` time & rollbacks (c) `ph5` time & rollbacks

Figure 14: Statistics from parallel simulation with `eventsPerLP`=10, $\lambda = 10$, `%selfEvents`=25%

experiments were conducted with fine-grained settings (*i.e.,* `granularity`=0) and results may vary with granularity. However, GSA data suggests that the variation with changing granularity would be small. However, increased granularity may allow relaxation of the time-window. The results strongly favor the general use of `2tLadderQ` over the `ladderQ`. Furthermore, the multi-tier organization of `2tLadderQ` can further reduce lock contention and consequent synchronization overheads in multithreaded simulations.

The experiments show that the runtime constants play an important role – for example, the Fibonacci heap with its $O(1)$ time complexity for many operations still did not perform well in our benchmarks. The `3tHeap` has a much lower runtime constants enabling it to outperform the `fibHeap` in almost all cases. In sequential simulations, the advantages of `3tHeap` are realized in simulations that have higher concurrency (*i.e.,* larger batches of events) per LP. Figure 16 summarizes the effective regions observed for the 3 queues. The advantages of `3tHeap` is realized only when each LP has 10 or more concurrent events at each time step. Such scenarios with high `eventsPerLP` arises in epidemic models [9] and detailed simulation models such as packet-level network simulations [10]. However, further experimental analysis with a broader range of models and configurations is needed to formally verify effectiveness of `3tHeap`. Moreover, different implementations, possibly in different programming languages will provide rigorous validation to ensure the results are algorithmic and not an artifact of one specific implementation.

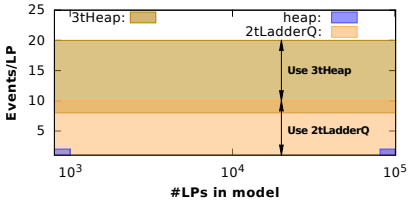


Figure 16: Effective regions of use

The multi-tier data structures enjoy lower runtime constants for event cancellation operations which play an influential role in Time Warp synchronized parallel simulations. Therefore, the multi-tier data structures perform consistently better in optimistic parallel simulations.

In overall summary, our analysis strongly favor broad use of our multi-tier queues, specifically `2tLadderQ` and `3tHeap`, replacing all existing DES data structures. The `2tLadderQ` and `3tHeap` are consistently effective in sequential and parallel simulations, with sequential results also bearing potential application to conservative and multithreaded simulations.

7.1 Supplementary Material

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

Acknowledgments

Support for this work was provided in part by the Miami University Global Health Research Innovation Center.

8. REFERENCES

[1] C. D. Carothers and K. S. Perumalla. On deciding between conservative and optimistic approaches on massively parallel platforms. In *Proceedings of the Winter Simulation Conference*, WSC '10, pages 678–687, Baltimore, Maryland, 2010.

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

[3] R. Franceschini, P.-A. Bisgambiglia, and P. Bisgambiglia. A comparative study of pending event set implementations for pdevs simulation. In *DEVS Integrative M&S Symposium*, pages 77–84, San Diego, CA, USA, 2015. SCS.

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

[5] B. Guven and A. Howard. Identifying the critical parameters of a cyanobacterial growth and movement model by using generalised sensitivity analysis. *Ecological Modelling*, 207(1):11 – 21, 2007.

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

[7] R. Marotta, M. Ianni, A. Pellegrini, and F. Quaglia. A non-blocking priority queue for the pending event set. In *Proceedings of the 9th EAI International Conference on Simulation Tools and Techniques*, SIMUTOOLS'16, pages 46–55, ICST, Brussels, Belgium, Belgium, 2016.

[8] F. Quaglia. A low-overhead constant-time lowest-timestamp-first CPU scheduler for high-performance optimistic simulation platforms. *Simulation Modelling Practice and Theory*, 53:103–122, 2015.

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

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