Managing Pending Events in Sequential & Parallel Simulations using 3-Tier Heap & 2-Tier-Ladder Queue

DHANANJAI M. RAO, CSE Department, Miami University, USA
JULIUS D. HIGIRO, CSE Department, Miami University, USA

Performance of sequential and parallel Discrete Event Simulation (DES) is strongly influenced by the data structure used for managing and processing pending events. Accordingly, we propose and evaluate the effectiveness of our 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 experiments compare the performance of our data structures against a performance-tuned version of the Ladder Queue, which has shown to outperform many other data structures for DES. The core simulation-based empirical assessments are in C++ and are based on 2,500 configurations of well-established PHOLD and PCS benchmarks. In addition, we use an Avian Influenza Epidemic Model (AIM) for experimental analyses. We have conducted experiments on two computing clusters with different hardware to ensure our results are reproducible. Moreover, to fully establish the robustness of our analysis and data structures, we have also implemented pertinent queues in Java and verified consistent, reproducible performance characteristics. Collectively, our analyses show that our 3-tier heap and 2-tier ladder queue outperform the Ladder Queue by 60× in some 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**;

Additional Key Words and Phrases: Discrete Event Simulation (DES); Optimistic Parallel Simulation; Time Warp; Binary Heap; Fibonacci Heap; Ladder Queue

ACM Reference Format:
Dhananjai M. Rao and Julius D. Higiro. 2018. Managing Pending Events in Sequential & Parallel Simulations using 3-Tier Heap & 2-Tier-Ladder Queue. *ACM Trans. Model. Comput. Simul.* 0, 0, Article 1 (January 2018), 25 pages. https://doi.org/0000001.0000001

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 [10]. 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 [5, 6, 12, 18]. The effectiveness of data structures for event management is a conspicuous issue in larger simulations, where thousands or millions of events can be pending [2, 16]. Overheads in managing pending events is magnified

Authors' addresses: Dhananjai M. Rao, CSE Department, Miami University, 510 E. High Street, Oxford, OH, 45056, USA, raodm@miamiOH.edu; Julius D. Higiro, CSE Department, Miami University, 510 E. High Street, Oxford, OH, 45056, USA, higirodj@miamiOH.edu.

XXXX-XXXX/2018/1-ART1 $15.00
https://doi.org/0000001.0000001

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.

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* [18] has shown to be the most effective data structure for managing pending events [4, 5], 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 [5, 18]. However, as detailed in Section 7, the Ladder Queue was substantially slower in two cases – ❶ high concurrency: a 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 PHOLD and PCS benchmarks are used to assess the effectiveness of the multi-tier data structures vs. our fine-tuned implementation of the Ladder Queue. In addition, we have also conducted experimental analysis using an Avian Influenza Epidemic Model (AIM) detailed in Section 3.3. Results from our experiments discussed in Section 7 data shows `2tLadderQ` provides comparable performance 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 OVERVIEW OF PARALLEL SIMULATOR

The implementation and assessment of the different data structures have 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.

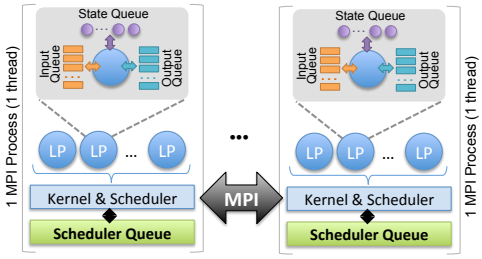


Fig. 1. Overview of a parallel MUSE simulation

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 Platforms

The design of MUSE and the experiments reported in this paper were conducted using the following two different distributed-memory compute clusters:

① **RedHawk cluster**: Each compute node has two quad-core 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) interconnected by 1 GBPS Ethernet. The simulation software was compiled using GCC version 4.9.2 (`-O3` optimization level) with OpenMPI 1.6.4.

② **Oakley cluster**: The second cluster was used to ensure that the experimental results are consistent and reproducible. The compute nodes on this cluster have two hex-core Intel Xeon ® x5650 CPUs running at 2.67 GHz (with hyperthreading disabled) and 48 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) interconnected by 40 GBPS Infiniband. The simulation software was compiled using ICC version 16.0.3 (`-O3` optimization level) with OpenMPI 1.10.5.

3 BENCHMARKS AND APPLICATION USED FOR EXPERIMENTAL ASSESSMENTS

The experimental analyses have been conducted primarily using parallelized version of the classic Hold synthetic benchmark called PHOLD. We have also used the Personal Communication Service Network (PCS) model proposed by Carothers *et al* [1]. These benchmark applications have been used by many investigators for assessment of data structures [5, 18, 19]. Furthermore, we have also experimented with a parallelized epidemiological model of the global spread of avian influenza via migratory waterfowl [15–17]. A brief overview of these benchmarks is presented in the following subsections.

3.1 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 [5, 18]. 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 total number of LPs in the simulation is `rows` × `cols`. LPs

Table 1. Parameters in PHOLD benchmark

Parameter	Description	Parameter	Description
`rows`	Number of rows in model.	`cols`	Number of columns in model.
`events-PerLP`	Initial number of events per LP.	`simEnd-Time`	Simulation end time.
`delay-distrib`	Event timestamp distribution*	`recvr-distrib`	Receiver ID distribution*
`%self-Events`	Fraction of events LPs send to themselves.		
`delay` or λ	Parameter for distribution specified by `delay-distrib`.	`recvr-range`	Parameter for distribution specified by `recvr-distrib`.
`granularity`	Additional compute load per event.	`imbalance`	Imbalance in partition, *i.e.,* more LPs on some MPI-processes.

*Distribution can be one of: "uniform", "poisson", or "exponential"

are evenly partitioned across the MPI-processes used for simulation. However, the `imbalance` parameter influences the partition, with larger values skewing the partition such that more LPs are assigned to some partitions (see [9] for details). The `imbalance` parameter has no impact on sequential simulations.

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. For each event, an LP schedules another event to a randomly chosen adjacent LP determined by `recvr-distrib` and `recvr-range` parameters. 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.

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 our queues. Section 7 explores the impact of these parameters on scheduler queue performance using 2,500 different configurations.

3.2 Personal Communication Service Network (PCS) model

The PCS model [1] has also been used to assess effectiveness of the proposed priority queues. The PCS model consists of "cells" (*i.e.,* cellular towers) that transmit and receive phone calls made by "portables" (*i.e.,* cell phones). Portables are mobile and travel to various cells. Cells are modeled using LPs organized as a matrix specified by `rows` ×`cols` command-line arguments to the model. Each cell contains a fixed number of wireless channels that are time-division multiplexed to portables on demand for cellular communication. Life cycle of each portable is modeled using an event with the following three fields: ① `moveIntervalMean`: mean for exponential distribution to model movement of portables between cells, ② `callIntervalMean`: mean for exponential distribution to model call intervals, and ③ `callDurationMean`: mean of a Poisson distribution used to determine duration of calls [1]. The minimum of these 3 fields determines the behavior of a portable – *i.e.,* completion of a phone call, arrival of the next portable call at a cell, and the departure of a portable from its current cell to a neighboring cell. The simulation ends when a given `simEndTime` is reached.

3.3 Avian Influenza Epidemic Model (AIM)

The Avian Influenza Epidemic Model (AIM) characterizes the spread of avian influenza viral strains via 22 high risk migratory waterfowl species. Migratory waterfowl populations and migration flyways are automatically generated from Geographic Information System (GIS) data were obtained from the Global Register for Migratory Species (GROMS) database. Collocated birds of the same species are modeled as one aggregate agent called a "flock". Each flock is modeled by 1 LP. The full model consists of 3,088 flocks, modeled as interacting LPs and is simulated for 2 years to study the dispersion of viral strains due to migratory birds. The model involves a very dynamic interaction patterns and a broad range of timestamps. This model also uses rollback-reduction optimizations accomplished using proxy agents (see [16] for details) to minimize rollbacks and to provide good scalability.

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 namely: ❶ binary heap (`heap`), ❷ 2-tier heap (`2tHeap`), ❸ 2-tier Fibonacci heap (`fibHeap`), ❹ 3-tier heap (`3tHeap`), ❺ Ladder Queue (`ladderQ`), and ❻ 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 to 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.

Table 2. Comparison of asymptotic time complexities (*i.e.,* Big O) of different data structures

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	$e \cdot l \div {}_{t2}k$

Legend – l: #LPs, e: #events / LP, c: #concurrent events, z: #canceled events, ${}_{t2}k$: parameter, 1: amortized constant

4.1 Binary Heap (`heap`)

The binary heap (`heap`) is a commonly used single-tier data structure for implementing priority queues. It outperformed a Binomial heap in our analyses. It has been 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 the 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 $O(\log(e \cdot l))$ 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 are 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 2. The first tier has containers for each local LP on an MPI-process. Each of the tier-1 containers contains a binary heap of events (the second tier) 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.

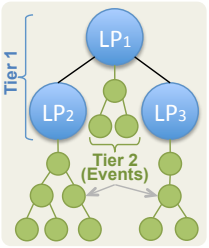


Fig. 2. 2-tier Heap (`2tHeap`)

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 modified version of the BOOST C++ library. The Fibonacci heap has an amortized constant time for changing key values and finding a 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 the 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. The binary heap implementation for the first tier that manages LPs for scheduling has been retained from `2tHeap`. However, 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. Algorithm 1 shows the key steps to enqueue an event. 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, an amortized constant time operation. Enqueue to tier-2 is followed by an optional heap fix-up to maintain LP with least timestap at top of heap. The heap fix-up has 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. Our experiments suggest that recycling tier-2 containers is an important implementation detail necessary for reducing the runtime constants associated with `3tHeap`.

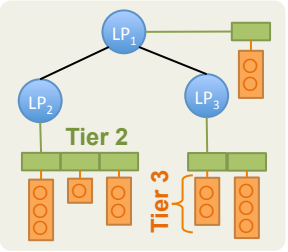


Fig. 3. 3-tier Heap (`3tHeap`)

4.5 Ladder Queue (`ladderQ`)

The `ladderQ` is a priority queue implementation proposed by Tang *et al* [18] with amortized constant time complexity as summarized in Table 2. Several investigators have independently verified that for sequential Discrete Event Simulation (DES) the `ladderQ` outperforms other priority queues, including: simple sorted list, binary heap, Splay tree, Calendar queue, and other multi-list data structures [4, 5, 18]. 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:

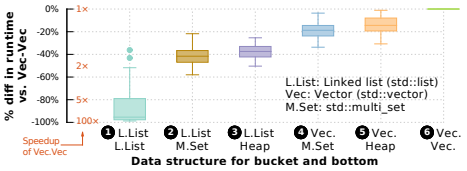
(1) *Top*: An unsorted list which contains events scheduled into the distant future or epoch.
(2) *Ladder*: Consists of multiple rungs, *i.e.,* list of buckets. Each bucket contains events with a finite range of timestamp values. Hence, although events within a bucket are not sorted, the buckets on a rung are organized in a sorted order. The `ladderQ` minimizes the number of events to be finally sorted by recursively breaking buckets into a smaller range of timestamp values.
(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 [18].

ALGORITHM 1: `3tHeap` enqueue

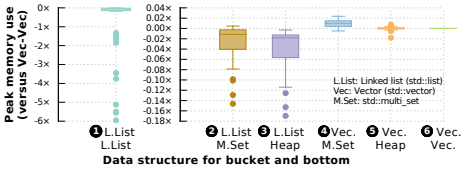
Input: *event* to be added to the 3tHeap
Output: Event inserted in 2nd-tier. 1st-tier heap updated to maintain LTSF at top of heap

```
recvr = agentList[event->receiver];
currTime = minTime(recvr->tier2);
/* Do binary search & find tier-2 bucket */
iter = lower_bound(recvr->tier2, event->timestamp);
if iter == recvr->tier2.end() then
    t2Bkt = getRecycledBucket(); t2Bkt.append(event);
    recvr->tier2.append(t2Bkt);
else
    if *iter->timestamp == event->timestamp then
        (*iter)->append(event);
    else
        t2Bkt = getRecycledBucket(); t2Bkt.append(event);
        recvr->tier2.insert(iter, t2Bkt)
    end
end
if currTime != minTime(recvr->tier2) then
    fixHeap(recvr)
end
```

4.5.1 Fine tuning Ladder Queue performance. Our implementation closely followed the design in the original paper by Tang *et al* [18]. 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.



(a) Comparison sequential runtimes



(b) Peak memory used

Fig. 4. Comparison of execution time and peak memory for PHOLD benchmark (different parameter settings) for 6 different `ladderQ` implementations.

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 grows. 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.

The maximum number of rungs in the *Ladder* also influences the overall performance of the `ladderQ` [18]. The chart in Figure 5 illustrates the impact of limiting the maximum number of rungs in the `ladderQ`. This experiment was conducted using using PHOLD simulation involving 10,000 agents (model ph4 in Table 3). 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 subdivided into smaller timestamp-based bucket widths. Small bucket widths in turn minimize inserts into bottom and *Bottom*-to-*Ladder* operations, ensuring good performance.

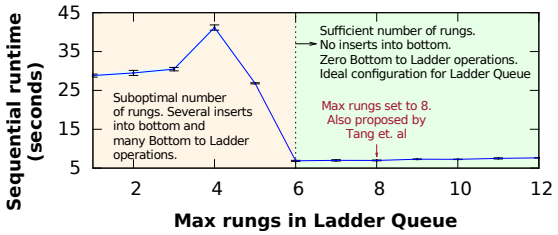


Fig. 5. Impact of limiting rungs in Lader

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 [18]. Moreover, it limits the overheads involved in re-bucketing events from rung-to-rung [18]. 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* [18]. Furthermore, we trigger *Bottom*-to-*Ladder* re-bucketing only if the *Bottom* has events with 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 [4, 5, 18]. 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 occur 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 showed 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 the 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.

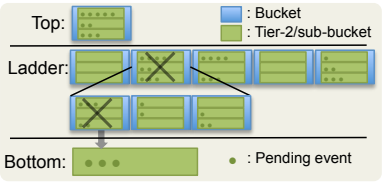


Fig. 6. 2-tier Ladder Queue (`2tLadderQ`) with 3 sub-buckets / bucket (*i.e.*, ${}_{t2}k$=3)

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 amortized 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.

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 changing the size of ${}_{t2}k$ in a parallel PHOLD simulation involving 10,000 agents (model ph4 in Table 3) using 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 a slow increase in runtime due to overhead of larger sub-buckets. This trend was consistent for the three models used in this study. Consequently, we have used a value of ${}_{t2}k$ = 128 for parallel simulation, which serves as a conservative upper-bound value.

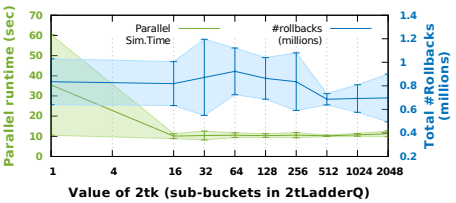


Fig. 7. Effect of varying ${}_{t2}k$

4.7.1 Re-bucketing rungs. In model with large variations in timestamps such as the AIM (see Section 3.3), we observed that some rungs of the `ladderQ` can grow to have a large number (>10,000) of sparsely populated buckets. This causes a significant increase in memory usage. In order to circumvent this issue, we have introduced re-bucketing of rungs in `2tLadderQ`. Re-bucketing computes a new, larger range of timestamp values for each bucket such that the number of buckets in a given rung does not exceed a fixed threshold. In our implementation we have used the bucket threshold of 50 that was proposed by Tang *et al* [18]. Next, events are redistributed to the new buckets. In our experiments, we observed that re-bucketing is a rare occurrence and was only observed to occur a few times with the AIM. Nevertheless, as discussed in Section 7.4, re-bucketing decreases peak memory usage by 5× when simulating AIM.

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 the effectiveness of the data structures against our fine-tuned version of the Ladder Queue [18] because it has shown to be very efficient for sequential Discrete Event Simulation (DES). Recently, Franceschini *et al* [5] 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* [18] and Franceschini *et al* [5] 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 with 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.

Dahl *et al* [3] propose an append-queue data structure for pending event set management. Append-queue maintains a list of sorted events for each LP in a sender-queue, with each LP having a separate sender-queue. For scheduling, a separate schedule-list is used to maintain a sorted list of head's of each sender-queue. This structure is comparable to our 2-tier heap implementation, except we maintain binary heaps in the two tiers versus sorted lists as proposed by Dhal *et al* [3].

Dickman *et al* [4] 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* [6] 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. Hay and Wilsey [8] explore the effectiveness of using hardware-based transaction memory (TSX) to manage pending events in their optimistic parallel simulator. They explore two variants of TSX, namely Hardware Lock Elision (HLE) and Restricted Transactional Memory (RTM). Their experiments conducted using a multi-set data structure shows that HLE outperforms conventional locking mechanisms by up to 27%.

Quaglia [14] proposes a Low-Overhead Constant-Time (LOCT) scheduler 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 show that the LOCT scheduler can outperform ladder queue. However, the ladder queue realizes a better overall efficiency than LOCT

because it throttles optimism due to its higher queue management costs [14]. Marotta *et al* [11, 12] 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. They further extend their data structure by introducing conflict-resiliency to alleviate overhead of concurrent extractions from the bucket with lowest time stamp events [13].

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. 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 PARAMETER REDUCTION VIA GENERALIZED SENSITIVITY ANALYSIS (GSA)

Our initial experimental analysis focused on performance comparison of different scheduler queues using PHOLD and PCS benchmarks. However, our initial experiments proved to be cumbersome and time-consuming due to the large number of parameters (for example, see Table 1) and combinations of their values. Consequently, we pursued strategies to focus on most influential subset of parameters that impacted relative performance of the scheduler queues using Generalized Sensitivity Analysis (GSA) [7]. 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 [7]. The $d_{m,n}$ statistic is the maximum separation between cumulative probability distribution observed in a two-sample KS-Test.

In our experiments, the KS-Test has been performed for each pair of queues being compared using data from Monte Carlo simulations involving 2,500 different combinations of parameter values generated from a specified range. The combination of values is generated using Sobol random numbers to provide uniform coverage of the multidimensional parameter space. Each combination of parameters is used to conduct simulations. The simulation results are 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 `ladderQ` runs slower when compared to another scheduler queue. For sequential and parallel simulations we use ${}_{t2}k$=1 and ${}_{t2}k$=128 respectively. Our parameter ranges also ensure that the peak memory consumption does not cross NUMA threshold (which in our case is 4 GB of RAM) as it introduces a lot of variance in runtimes requiring too many runs to reduce variance to acceptable limits. The results from sequential and parallel GSA are discussed in the following subsections.

6.1 PHOLD GSA results

The results from sequential GSA analysis for `ladderQ` vs. `3tHeap` conducted using the PHOLD benchmark with different parameter settings and 9 different combinations of statistical distributions is shown in Figure 8. The 9 combinations correspond to the distributions used for event timestamps and receiver LP ids – *i.e.,* `delay-distrib` and `recvr-distrib` in Table 1. The error bars show the 95% Confidence Intervals (CI) computed using standard bootstrap approach using 5000 replications with 1000 samples in each.

The parallel simulations were conducted using 4 MPI-processes for parallel simulation. These analysis focused only on `ladderQ`, `2tLadderQ`, and `3tHeap` as they generally outperformed all other queues in our earlier sequential GSA analyses. Initially, we observed that the `ladderQ` timings showed a lot of variance in runtime depending on the number of rollbacks that occur in parallel simulation. Consequently, to reduce variance, we have used a time-window of 10 time-units to

reduce rollbacks as elaborated further in Section 7.2.1. We use the same time-window for all scheduler queues for consistent comparison and analysis.

As illustrated by the charts in Figure 8, the most influential parameters that highlight difference in performance between `ladderQ` and `3tHeap` are: ① `eventsPerLP` (`E/LP`) which influences the number of concurrent events (*i.e.,* events with same timestamp) to be processes by an LP in each cycle; and ② λ, the event timestamp distribution parameter that determines how close (or far) the timestamp values are to increase (or decrease) probability of concurrent events. 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. Remaining parameters, namely: GVT period (`GVT`), % self events (`%SE`), simulation end time (`Time`), and event granularity (`Gran.`) do not have a strong influence on explaining the performance difference between `ladderQ` and `3tHeap`. Similar GSA results were also observed for sequential simulation.

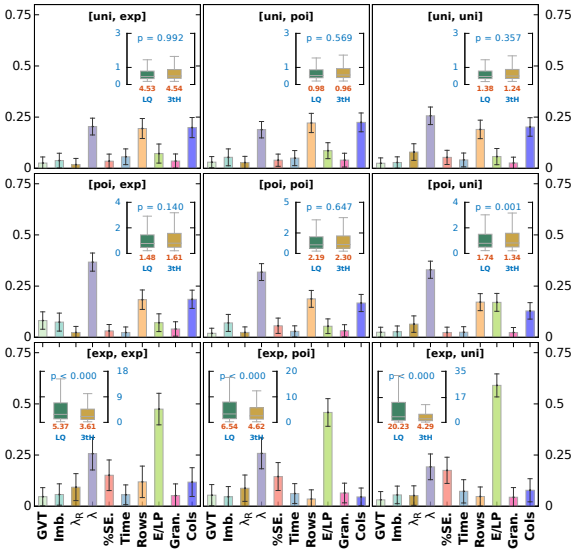


Fig. 8. Results from GSA of PHOLD parallel simulations for different event timestamp & receiver distributions Box plots compare runtimes along with mean and p-values from paired T-tests.

We have also conducted GSA with `ladderQ` vs. `2tLadderQ`. Our data show that these two queues have comparable performance in sequential simulations, with `ladderQ` slightly outperforming the `2tLadderQ` due to the extra hashing operation implemented using modulo operation. We have also conducted GSA to determine influential parameters impacting the performance of other scheduler queues versus the `ladderQ` in sequential simulations. Our analysis showed that none of the parameters play an influential role and the `ladderQ` generally performed consistently better or the same.

6.2 PHOLD GSA summary & configurations for further analysis

GSA shows that for comparing event queue performance in sequential and parallel simulations using PHOLD benchmark, the experiments need to focus on 2 parameters, namely: `eventsPerLP`, and λ. Other aspects such as: model size, event granularity, fraction of self-events, GVT rate, etc., are not influential for performance comparisons of scheduler queues. The exponential distributions are a better setting for analysis as they provide statistically significant results with a reduced parameter space. In addition, exponential distribution patterns have been widely reported and used in the literature for performance analyses [5, 18]. Moreover, the scheduler queues to focus further analysis are: `ladderQ`, `2tLadderQ`, and `3tHeap`.

6.2.1 PHOLD configurations for further 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 are 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 [5, 18].

Table 3. PHOLD configurations 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

6.3 PCS benchmark GSA results

We have conducted GSA analysis using the PCS benchmark using the same analysis procedure discussed earlier in this section. We have explored the multidimensional parameter space of PCS benchmark using 2,500 different combinations. The GSA analysis showed that for both sequential and parallel PCS simulations, the most influential parameter to explain performance differences of the queues is the number of portables initially assigned to each cell. The `portables` parameter determines the net number of events in the simulation because lifecycle of a portable is modeled using an event, as discussed in Section 3.2. The number of events in turn influence the performance of the priority queues used to manage pending events. Other parameters do not play a significant role in influencing the relative performance differences. Accordingly, our PCS-based experimental analyses have been conducted by varying the number of portables in a 1,000 cells (100×10) network while retaining default values [1] for all other parameters.

7 EXPERIMENTS & DISCUSSIONS

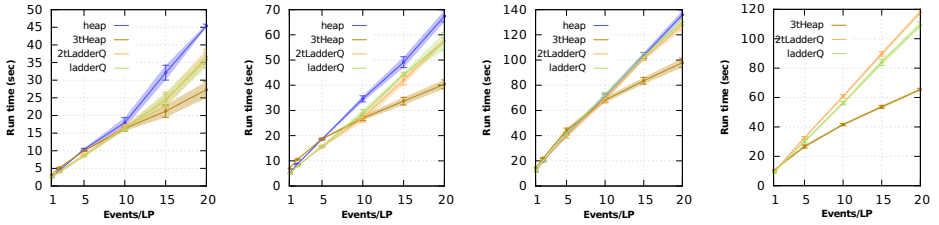
Assessments of the effectiveness of the six scheduler queues from Section 4 have been conducted using different configurations of the PHOLD, the PCS benchmark, and Avian Influenza Epidemic Model (AIM) discussed in Section 3. The experiments were conducted on RedHawk and Oakley compute clusters described in Section 2.1.

7.1 PHOLD sequential simulation results

We pursued sequential simulations to compare the base case performance of the data structures, reflecting analyses reported by other investigators [5, 18]. The sequential simulations also serve as a reference for potential use in conservatively synchronized PDES. The sequential experiments were conducted using the 3 PHOLD configurations from Table 3. The simulations were run in sequential mode (*i.e.,* no state saving, rollbacks, GVT, etc.) on one compute node. The software was compiled at `-O3` optimization level with all debug assertions compiled out via compiler flags. The number of sub-buckets in `2tLadderQ` was set to 1, *i.e.,* ${}_{t2}k$=1. For these experiments, the influential parameters `eventsPerLP` and λ identified via GSA analysis (see Section 6.2) were varied to explore their impact on the relative performance of the data structures. For each configuration, data from 10 independent replications were collected and analyzed.

The charts in Figure 9(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 in Figure 9(a)–(c), the performance of `ladderQ` and `2tLadderQ` (${}_{t2}k$=1) is comparable as expected. These two queues outperform the other queues for lower values of `eventsPerLP`.

However, the `3tHeap` outperforms the other queues for higher values of `eventsPerLP`. The trends were consistent on both RedHawk and Oakley clusters. In all cases, there were no inserts into *Bottom* or *Bottom*-to-*Ladder* operations (discussed in Section 4.5.1) that could degrade `ladderQ`

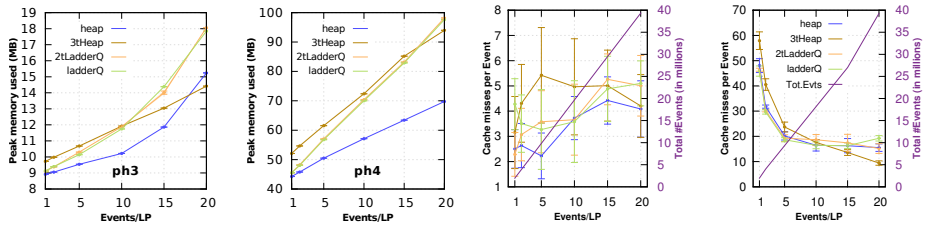


(a) RedHawk: ph3 (b) RedHawk: ph4 (c) RedHawk: ph5 (c) Oakley: ph5

Fig. 9. Sequential simulation runtimes from PHOLD with λ=1 (exponential distribution) and %self events=0.25

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 was 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* [18]. Profiler data 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 the number of concurrent events c increases.

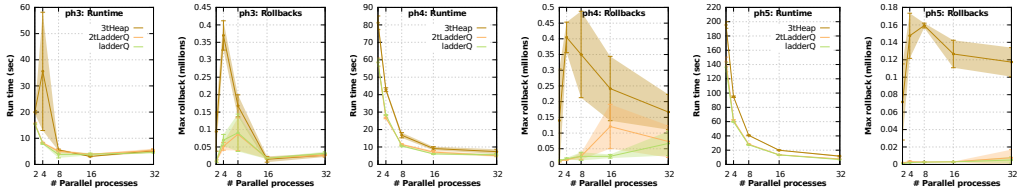
7.1.1 Peak memory usage. The charts in Figure 10 shows the peak memory usage and observed cache misses corresponding to the runtime data in Figure 9. The memory size reported is the "Maximum resident set size" value reported by GNU `/usr/bin/time` command on Linux. The memory usage of `heap` is the lowest 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::vectors` 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 a significant increase in memory footprint.



(a) ph3 Peak memory (b) ph4 Peak memory (c) ph3 Cache misses (d) ph4 Cache misses

Fig. 10. Comparison of peak memory usage and cache misses corresponding to Figure 9(a)–(b)

7.1.2 Caching characteristics. The L3 cache misses were recorded using Linux `perf` utility on RedHawk cluster on which the CPU has 8 MB L3-cache. The charts in Figure 10 show the average number of cache misses per event – *i.e.,* "total number cache misses" ÷ "total number of events". As illustrated by the charts, the cache misses for the different data structures is comparable with `3tHeap` having a slightly better cache performance only for the larger `ph4` case. Since overall the caching characteristics are comparable, we can also eliminate caching as an influential parameter. Consequently, the performance differences observed in Figure 9 is attributed to the inherent design of these data structures.



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

Fig. 11. Statistics from parallel simulation on RedHawk with `eventsPerLP=2`, $\lambda = 1$, `%selfEvents=25%`

7.2 PHOLD Parallel simulation assessments

The parallel simulation experiments were conducted on the two compute clusters (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.

7.2.1 Throttling optimism with a time-window. During our initial experimentation with PHOLD, 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 large variances due to cascading rollback scenarios for `ladderQ` and to streamline experimental analysis time frames (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 optimistically processing 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 an exponential distribution with $\lambda = 1$. Hence, most events in current schedule cycle will fit within this time-window. We use the same time-window for all scheduler queues for consistent comparison and analysis.

7.2.2 Efficient case for `ladderQ`*.* The charts in Figure 11 show key simulation statistics for a low value of `eventsPerLP` = 2 and λ=1 for which `ladderQ` performed well, consistent with the observations in sequential simulations. The charts show averages and 95% CI computed from 10 independent replications for each data point. As illustrated by the data in Figure 11, both the `ladderQ` and `2tLadderQ` perform well for all three models. The charts in Figure 12 shows the observations on Oakley cluster. The performance trends of all of the runs on RedHawk and Oakley were similar, with Oakley having a faster runtime when compared to RedHawk. Other than the raw timing differences, the trends and behaviors were consistent establishing that the performance characteristics of the queues are consistently reproducible.

In these configuration, overall the `ladderQ` experienced the fewest rollbacks. The `2tLadderQ` continues to perform well despite experiencing more rollbacks as shown in Figure 11(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 a 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.

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 11(c), 3tHeap experienced about 114805 rollbacks on average while ladderQ experienced only 2341, almost 50×more 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 were comparable in these configurations.

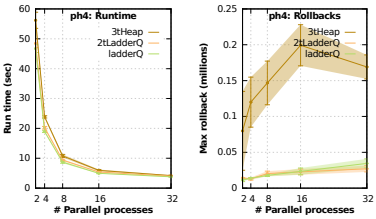
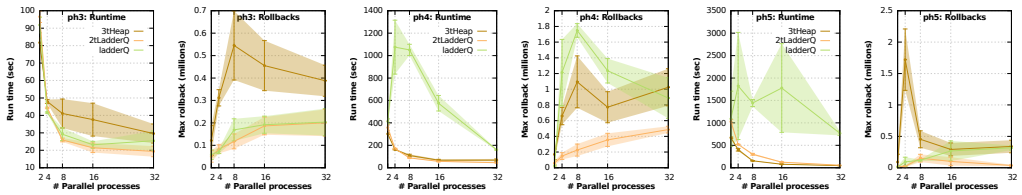


Fig. 12. ph4 (same configuration as Figure 11(b)) on Oakley

7.2.3 Knee point for 3tHeap *vs.* ladderQ. The charts in Figure 13 show key simulation statistics for the configuration where 3tHeap and ladderQ performed about the same in sequential (see Figure 9). For ph3, both ladderQ and 2tLadderQ experienced a 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.

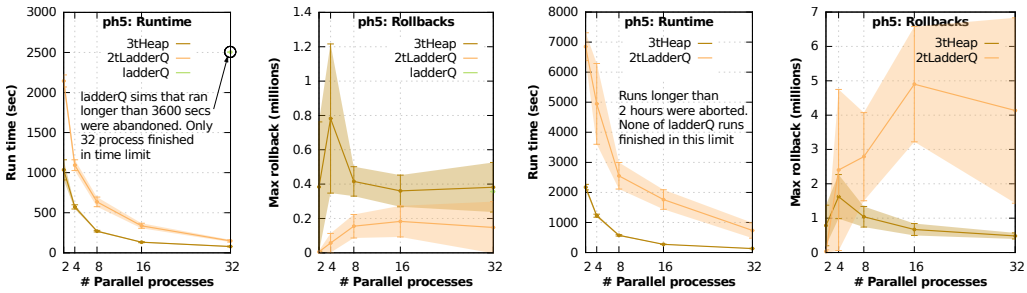


(a) ph3 time & rollbacks (b) ph4 time & rollbacks (c) ph5 time & rollbacks

Fig. 13. Statistics from parallel simulation on RedHawk with eventsPerLP=10, $\lambda = 10$, %selfEvents=25%

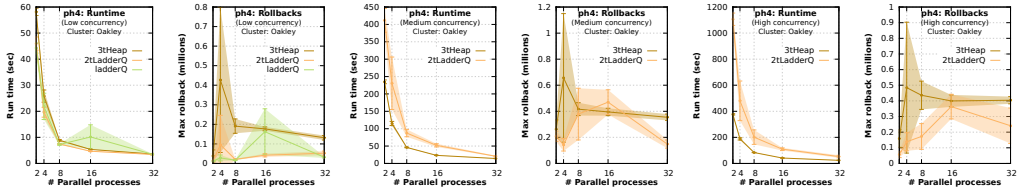
7.2.4 Best case for 3tHeap. Figure 14 shows simulation time and rollback characteristics in high concurrency configuration with ph5, with eventsPerAgent=20, λ=10, and %Self Evt.=25%. Consistent with all the PHOLD parallel runs, a time-window of 10 time-units has been used for these runs as well. Nevertheless, on RedHawk, 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 and data is not shown. On the other hand 2tLadderQ performed well due to its design. The 3tHeap outperformed the other 2 queues despite experiencing 2× more rollbacks. On Oakley that has a 40 GBPS Infiniband network, 2tLadderQ experienced a lot more rollbacks than 3tHeap. The increased rollbacks are attributed to slower performance of 2tLadderQ which takes much longer than 3tHeap to recover. Due to the increased number of rollbacks on Oakley, none of the ladderQ runs finished in 2 hours even with a time-window and consequently were abandoned. The data from Oakley cluster underscores the advantages of the 3tHeap when simulating models with high concurrency on high speed interconnects.

7.2.5 Experiments without throttling optimism. The PHOLD experiments discussed in the previous subsections utilized throttling to enable runtime comparisons with ladderQ across different configurations as discussed in Section 7.2.1. In contrast, this section discusses results from experiments conducted without the use of throttling. The charts in Figure 15 show runtime statistics from parallel simulations that did not throttle optimism. The experiments were conducted on



(a) ph5 time & rollbacks on RedHawk (b) ph5 time & rollbacks on Oakley

Fig. 14. Statistics from parallel simulation with ph5 with `eventsPerLP`=20, $\lambda = 10$, `%selfEvents`=25%



(a) `eventsPerLp`=2, $\lambda = 1$ (b) `eventsPerLp`=10, $\lambda = 10$ (c) `eventsPerLp`=20, $\lambda = 10$

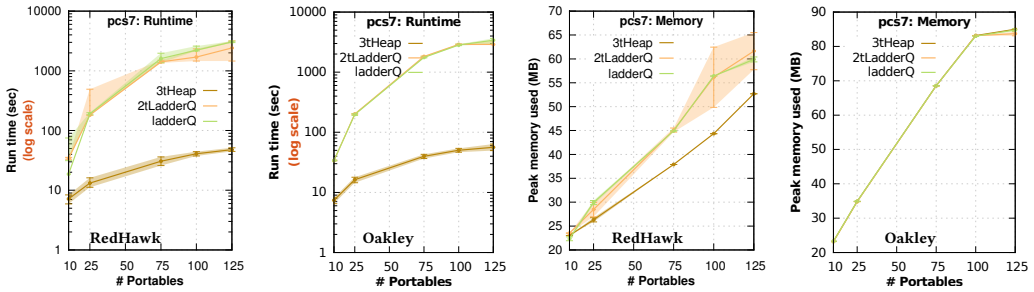
Fig. 15. Parallel simulations without throttling conducted on Oakley using different settings for ph4

Oakley cluster using different settings of ph4 model. The charts in Figure 15(a) correspond to the low concurrency configuration, but throttled data shown in Figure 12. The runtimes for the three queues are consistent as excepted. The slight increase in variance for `ladderQ` is also expected because variance in rollbacks is amplified due to cancellation overheads. In continuation with earlier discussions (see Section 7.2.1), the charts in Figure 15(b) and Figure 15(c) do not include plots for `ladderQ` because the runs exceeded 3600 seconds and were abandoned. The experimental results show that the `3tHeap` outperforms the `2tLadderQ` in configurations with higher concurrency, as expected, despite experiencing slightly higher number of rollbacks.

7.3 Personal Communication Service (PCS) sequential & parallel simulation results

The Generalized Sensitivity Analysis (GSA) for PCS model discussed in Section 6.3 established that the number of portables is the most influential parameter to explain performance differences between scheduler queues. Accordingly, our PCS-based experimental analyses have been conducted by varying the number of portables in a 1,000 cells (100×10) network while retaining default values [1] for all other parameters. For each configuration, data from 10 independent replications were collected and analyzed on both RedHawk and Oakley clusters. Note that for PCS simulation time-window based throttling was not necessary.

The charts in Figure 16 illustrate the runtime characteristics observed for sequential simulations on both RedHawk and Oakley. As expected, the performance of the `ladderQ` and `2tLadderQ` is comparable in sequential simulations. In the case of PCS simulations, the `3tHeap` significantly outperformed both the `2tLadderQ` and the `ladderQ`. For example, with 125 portables (a modest setting for this model), on RedHawk the average runtime of `3tHeap` was 49.68 seconds while `ladderQ` took 3088.64 seconds to finish resulting in a speedup of 62×! Corresponding runtimes on Oakley was 59.9 seconds for `3tHeap` and 3529 seconds for `ladderQ` resulting in a speedup of 58.9×. The performance difference is primarily due to a large number of "portables" (*i.e.,* events)

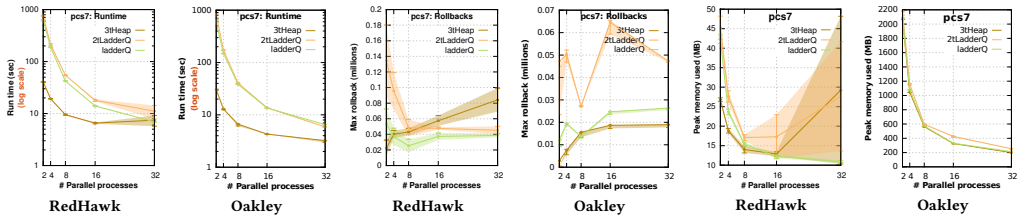


(a) Runtimes on RedHawk & Oakley (b) Memory use on RedHawk & Oakley

Fig. 16. Statistics from sequential simulations of PCS network with 1,000 cells

in the simulation, resulting in many concurrent events per "cell" (*i.e.,* an LP in this model). As illustrated by the charts in Figure 16(a), the performance characteristics were similar on Oakley as well, establishing consistently reproducible results. In other words, the performance differences are not based on hardware or compiler but purely attributed to the design of the data structures. Moreover, as illustrated by the charts in Figure 16(b), the memory consumption of the queues are comparable, with the `3tHeap` having a slightly lower peak memory footprint. These experiments exemplify the effectiveness of `3tHeap` over the `ladderQ` in simulations with a large number of concurrent events.

The charts in Figure 17 show key runtime statistics from parallel simulations conducted on both RedHawk and Oakley using 1,000 (100×10) cells (*i.e.,* LPs), with 100 portables per cell (*i.e.,* 100 events per LP). This configuration was chosen to provide some load for `3tHeap` with 32-processes (and yet runtime is < 4 seconds). The charts in Figure 17(a) illustrate similar runtime trends on both clusters with `3tHeap` outperforming the other queues, consistent with sequential simulation observations. However, the rollback patterns on RedHawk and Oakley were different because of the differences in interconnect technologies. The number of rollbacks on Oakley were lower when compared to RedHawk which is attributed to faster interconnects. Overall the PCS model had much fewer rollbacks than PHOLD, and consequently, both the `ladderQ` and `2tLadderQ` had about the same performance. The memory usage for the different queues was comparable. Importantly, the results add further credence that the performance differences between the queues are reproducible and platform-independent as they are attributed purely to the design of the data structures.



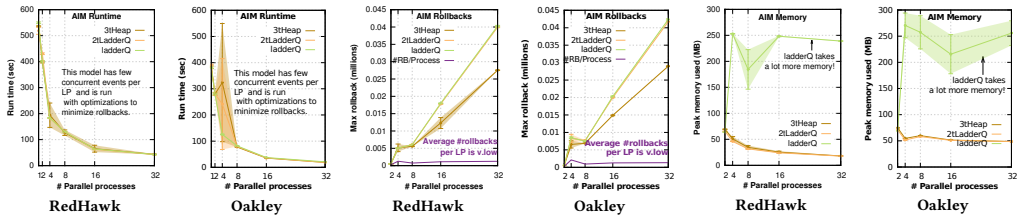
(a) PCS run times (b) PCS rollbacks (c) PCS peak memory use

Fig. 17. Statistics from parallel simulations of PCS network with 1,000 cells and 100 portables per cell

7.4 Avian Influenza Epidemic Model (AIM) simulation results

The AIM introduced in Section 3.3 with 3,088 LPs modeling flocks of migratory waterfowl was executed with a varying number of processors on RedHawk and Oakley clusters. Parallel simulations used a time-window of 2419200 seconds (*i.e.,* 28 days), that was identified as an effective setting [16]. The charts in Figure 18 show the key runtime characteristics from 10 independent simulation replications for the three key scheduler queues. The runtime with 1 process corresponds to sequential simulation where all the Time Warp related operations (such as: GVT, state saving, garbage collection, etc.) are turned off.

As illustrated by the runtime charts in Figure 18(a), all the three queues had comparable performance in both sequential and parallel simulation. Performance gains of using `3tHeap` in this application is limited because of few (about 2) events per simulation cycle. This setting is comparable to the runtime characteristics of low-concurrency PHOLD configuration shown in Figure 11. The performance of `2tLadderQ` is also not conspicuous because the simulation uses rollback-reducing optimization [16] which significantly reduces the average number of rollbacks (< 900 per process). Furthermore, the average pending event sizes in this model is small and overhead of event cancellation is not pronounced. Consequently, the performance of `2tLadderQ` and the `ladderQ` is comparable.



(a) Runtime (b) Rollbacks (c) Memory

Fig. 18. Simulation statistics for AIM on RedHawk and Oakley clusters

However, a key difference between the runtime characteristics of the queues is evident in their memory usage. The `ladderQ` consumes about 5× more memory than the other two queues. The source of the increased memory footprint arises due to the broad range of timestamp values used in the model. Migratory phases of the model use large time steps of 86400 (*i.e.,* 24 hours) while infection transmission phases use very small time steps of 0.125. The wide range of timestamp values causes the number of buckets in some of the rungs of the `ladderQ` become large. Particularly during rollback phase, the number of buckets sometimes exceeds 150,000 buckets. Combined with the minimum footprint of a `std::vector` object used to manage buckets, the peak memory footprint of `ladderQ` becomes much higher. On the other hand with `2tLadderQ` these rungs are rebucketed to eliminate uncontrolled growth of buckets as discussed in Section 4.7.1, thereby preserving both space and time efficiency.

8 PERFORMANCE ANALYSIS USING JAVA IMPLEMENTATION

In order to establish robustness and reproducibility of the performance characteristics, the proposed 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 Java implementation of the queues is semantically identical to their C++ counterparts. We have invested best efforts to maintain syntactic similarity with C++ to the maximum

extent possible. The difference between the C++ and Java versions arise from the idiomatic implementation styles of the two languages. Moreover, we have ported the C++ implementations for standard random number generators, namely: `std::exponential_distribution`, `std::poisson_distribution`, and `std::uniform_distribution`, to Java. We have tested to ensure that even the pseudo-random number generation in both C++ and Java is identical – *i.e.,* they produce exactly the same sequence given the same seed. Supplementary materials include source code listings for both programming languages along with a more detailed comparison of the source codes.

8.1 Characterizing rollbacks

The Java benchmark used to assess the performance of the queues is a highly customized sequential simulation of the PHOLD benchmark discussed in Section 3.1. The benchmark embodies most of the key characteristics of a sequential discrete event simulation, including event scheduling and event processing, but rollbacks do not occur in sequential simulations. However, performance differences between `2tLadderQ` and `ladderQ` arise only with rollbacks. Consequently, the Java benchmark has been designed to model the occurrence of rollbacks by triggering calls to cancel future pending events. The rollback behaviors in Java have been modeled based on statistical analysis of rollback profiles recorded from actual parallel simulations of the PHOLD benchmark on 6 processes using exponential distributions. Specifically, we have analyzed number of "inter-rollback schedules" – *i.e.,* how many schedules cycles of event processing are completed before a rollback occurs. Note that each schedule involves an LP processing 1 or more concurrent events (*i.e.,* events with the same timestamp). The chart in Figure 19 shows a histogram of average inter-rollback schedules from 5 independent simulations using 6 processes (more details in supplements). Rollbacks occurred at the communication boundaries between the 6 processes, with the 7^{th} region arising due to toroidal (*i.e.,* wraparound) grid used in PHOLD. The frequency of occurrences has been color coded. As illustrated by the chart in Figure 19, a majority of the inter-rollback schedules were of zero length, with longer inter-rollback schedules decreasing exponentially. The observation is typical for rollbacks because: ① they typically happen in the near future, and ② one rollback triggers a series of rollbacks on adjacent LPs to which they have optimistically scheduled events.

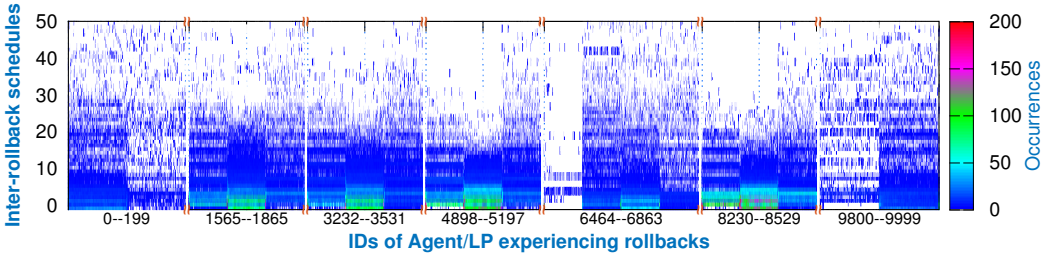


Fig. 19. Histogram of average inter-rollback schedules from 5 independent parallel simulations of PHOLD benchmark using 6 parallel processes. LPs with zero rollbacks are not shown. Full data and analysis is included in supplementary materials.

The inter-rollback schedule histograms have been fit to statistical distributions to ease characterization of rollbacks in our sequential Java benchmark. Fitting statistical distributions to the observed data has been conducted using R version 3.2.1 and the `fitdistrplus` package. We analyzed several different statistical distributions, including: Exponential, Poisson, Negative binomial, Geometric, Normal, Uniform, and Weibull. Among the standard distributions, the Geometric, Exponential, and Poisson distributions were the only ones that provided sufficiently low Standard Error (SE)

among the various fits. We have used these three distributions for further statistical analyses. Figure 20 illustrates an example of fitting the rollback profile. The geometric (SE = 0.0014) and exponential (SE=0.0015) distributions are a good fit, with geometric distribution having a slightly lower Standard Error (SE). The Q-Q plots also show good agreement between the empirical and theoretical quantiles verifying the statistical fit.

8.1.1 Autocorrelation Analysis. A critical first step to statistical analysis is to establish independence of samples – *i.e.,* validate part of the assumption that inter-rollback schedules are Independent and Identically Distributed (IID). Accordingly, prior to statistical fitting, we have used standard Autocorrelation Function (ACF) to verify that the inter-rollback schedule occurrences are independent and do not have any inherent correlation between them. Autocorrelation is computed by sliding the sequence of inter-rollback schedules, 1 at a time (*i.e.,* lag of 1), and comparing them against the original sequence to compute the Pearson correlation coefficient. Independence of the rollback occurrences is established by comparing the correlation coefficients against an 95% significance ($\alpha = 0.05$) ACF threshold computed as $\pm 2/\sqrt{N}$, where N is the number of observations. This standard method is based on the statistical inference that — if a time series is completely random, and the sample size is large, the lagged-correlation coefficient is approximately normally distributed with mean 0 and variance $\frac{1}{N}$.

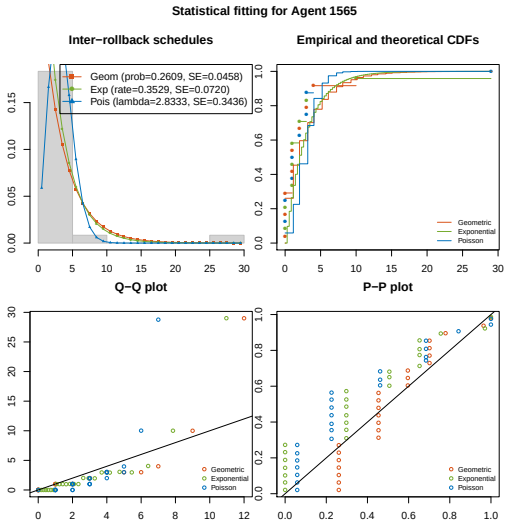


Fig. 20. Example statistical fit to inter-rollback schedule profile

The gray dots in Figure 21 shows a summary of the ACF values for the various agents that experienced rollbacks. As illustrated by the chart, for most of the agents, their rollback characteristics do not have much autocorrelation with ACF values above 80%. However, some of the rollbacks do show stronger correlations due to two reasons – ❶ there were very few rollbacks for some of the agents causing them to be outliers in the data, and ❷ event exchange patterns of agents have some correlation and consequently rollbacks used for event cancellation also reflect this characteristic. Overall the chart in Figure 21 shows that the rollback characteristics show independence (*i.e.,* they are randomly distributed) and are amenable to be fitted to classic statistical distributions.

8.1.2 Results from statistical fitting. The results from fitting statistical distributions to all of the agents experiencing rollbacks is shown in Figure 21. As illustrated by the Standard Error (SE) curves, the Geometric distribution had the lowest errors for all of the 2,001 agents with rollbacks. The chart in Figure 21 also shows the probabilities for the Geometric distributions for each of the agents (in light orange impulses). Although the model with 10,000 agents was evenly partitioned and each agent had similar behaviors, the rollback distributions are not consistent across the parallel processes. However, within a process, the geometric probabilities show some consistency, with fitted probabilities lying within specific ranges. The chart in Figure 22(a) shows a histogram of the Geometric probability distribution for agents 1565–1865. As illustrated by the chart, the Geometric probabilities are uniformly distributed with most of the probabilities lying in the range 0.05–0.20.

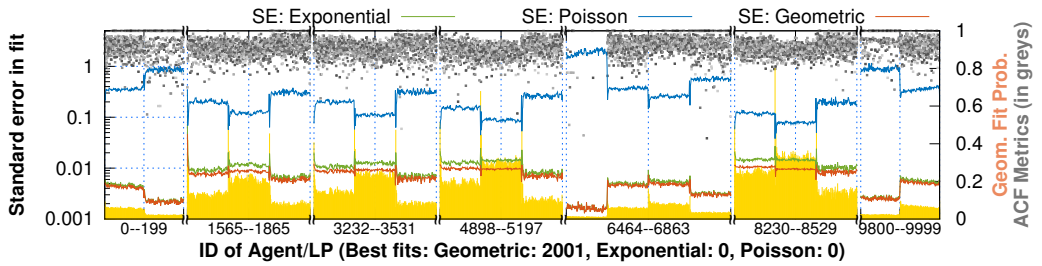
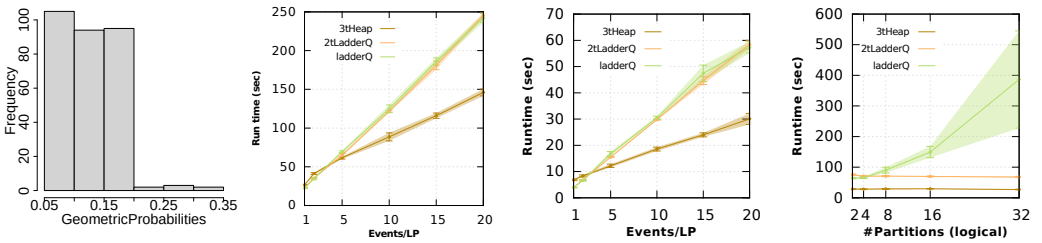


Fig. 21. Standard Error (SE) observed from statistical fitting of Poisson (▬), Exponential (▬), and Geometric (▬) distributions (plotted against Y1-axis) for the agents experiencing rollbacks in the simulation. Note that agents that do not experience rollbacks are not shown. The gray points show the corresponding Autocorrelation Function (ACF) values (plotted against Y2-axis). The light orange impulses (▮) show the Geometric probabilities that were fit to the data (plotted against Y2-axis). Full data and analysis is included in supplementary materials.

The Standard Error (SE) in each of the fits is in the range 0.0013–0.0458 with an average SE_μ = 0.006. This range lies in the middle of the rollback frequencies – *i.e.,* it not too high (as in 8230–8529 LP range) nor is it too low (as in 0–199 or 9800–9999 LP range) and is used for modeling rollbacks.



(a) Fitted probabilities (c) MUSE (C++) (b) Java (d) #Partitions vs. runtime

Fig. 22. Simulated rollback probabilities and key runtime characteristics from Java benchmarking

8.2 Results from Java benchmarking

The chart in Figure 22(b) shows the runtime of the Java benchmark with 10,000 LPs (100×100) using exponential distribution with $\lambda = 10$ for timestamps. This configuration is the same as the high-concurrency configuration discussed in Section 7.2.4. The charts show averages and 95% confidence intervals computed from 10 independent simulation replications for each data point. As illustrated by the charts in Figure 22(a) and Figure 22(b), the Java benchmark results are consistent with the trends observed in the actual MUSE simulator. MUSE takes longer to run because the Java benchmark is a simplified version. Recollect that the queue implementations in both languages are semantically identical. The consistent performance trends clearly establish that the advantages of `3tHeap` are reproducible on multiple platforms and programming languages.

The curves in Figure 22(d) illustrate the impact of increasing the number of virtual partitions in the Java benchmark. Increasing the number of partitions essentially increases the number of simulated rollbacks using geometric probabilities uniformly selected from the range 0.0013–0.0458. This chart highlights the advantages of `2tLadderQ` which is able to quickly cancel out pending events. The `ladderQ` consumes considerably higher time to complete the same operations. The experimental

results from the Java implementation are consistent with the sequential and parallel simulation observations, establishing that the results are reproducible across platforms and programming languages. In other words, the advantages of `3tHeap` and `2tLadderQ` are algorithmic and arise from its design rather than from hardware or compiler optimizations.

8.2.1 Analysis of runtime constants using linear regression. The curves in Figure 23 show a linear regression fit of the observed sequential Java runtimes (see Figure 22(c)) with respect to the total number of events. The linear regression fits were very strong with $R^2 > 0.99$ in all three cases. Linear runtimes for `ladderQ` and `2tLadderQ` with amortized $O(1)$ runtimes is expected, because $runtime = |events| * C$ (where C is a runtime constant). The linear regression 95% CI shows a per-event time constant (C) for `ladderQ` and `2tLadderQ` to be 0.0525µs—0.0639µs and 0.0573µs—0.0589µs respectively. Interestingly, this range is exactly 10×faster (on Xeon E5520 @ 2.27 GHz) than the values reported by Tang *et al* (on a Pentium 4 @ 2.4 GHz, in Figure 5 in [18] for large queues). The strong ($R^2 > 0.99$) linear regression is an unambiguous indicator of $O(1)$ time complexity validating our implementations. Furthermore, the `3tHeap` also exhibits similar amortized $O(1)$ trends, but with a much lower per-event access time of 0.0234µs—0.0255µs. Despite an initial overhead, the overall design of the `3tHeap` enables it to also deliver an amortized $O(1)$ time complexity but with ∼2× faster runtime constant. The lower runtime constant enables the `3tHeap` to deliver conspicuous performance improvements over the `ladderQ`.

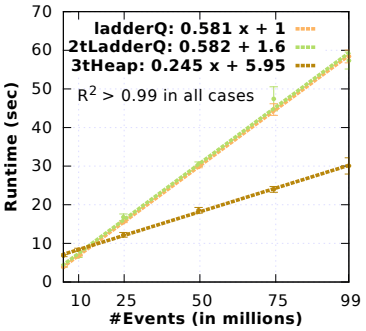


Fig. 23. Linear regression fitting

9 CONCLUSIONS

Efficient data structures (*i.e.,* priority queues) for managing pending events 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 the use of different algorithms and data structures to suit the different needs at each tier.

Our comparative performance analyses used a significantly fine-tuned version of the Ladder Queue (`ladderQ`) [18]. The objective of fine-tuning was to reduce the runtime constants of the `ladderQ` without 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. 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 the design of experiments was exploring the large multidimensional parameter space in PHOLD and PCS benchmarks. 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 suggest 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 6 queues × 2,500 parameter combinations × 6 replications

= 90,000 total 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.,* a batch of events scheduled per LP), plays the most dominant role to explain performance differences between the data structures. 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. The results strongly favor the general use of `2tLadderQ` over the `ladderQ`.

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 much lower runtime constants enabling it to outperform other data structures in many cases. The advantages of `3tHeap` are realized in simulations that have higher concurrency (*i.e.,* larger batches of events) per LP. Such a scenario was common in PCS model that yielded a conspicuous 62× speedup. Figure 24 summarizes the effective regions observed for the 3 queues. The advantages of `3tHeap` is clearly realized when each LP has 7 or more concurrent events at each time step. With fewer concurrent events, its performance is comparable or lower than `2tLadderQ`. The performance of `2tLadderQ` and `ladderQ` are comparable as long as the hashing function used to determine sub-buckets for `2tLadderQ` is very lightweight. In this study we used a modulo operator (*i.e., sub-bucket* = `receiver_id` % $_{t2}k$) as the hash. However, we recommend a faster hash using bit-wise operators (*i.e., sub-bucket* = `receiver_id` & $_{t2}k$) by restricting $_{t2}k$ to be an integral power of 2 (*i.e.,* $_{t2}k = 2^n$).

The experiments conducted on two different compute cluster using three different model and a broad range of parameter settings establishes that the performance trends are consistently reproducible. Moreover, implementations in both C++ and Java also show similar performance characteristics. The multi-platform and multi-programming-language analyses unambiguously establish that the advantages of `3tHeap` and `2tLadderQ` are algorithmic and stem from its design rather than from hardware, programming-language, or compiler optimizations.

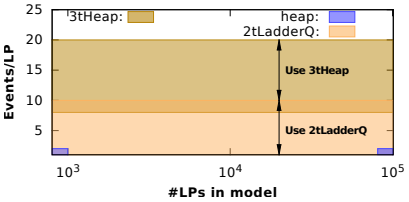


Fig. 24. Effective regions of use

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. Moreover, when compared to `ladderQ` and `2tLadderQ`, the `3tHeap` has a much more straightforward design without any hyperparameters such as rung counts, thresholds etc. This significantly simplifies implementation and analysis of `3tHeap`. 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.

ACKNOWLEDGMENTS

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

REFERENCES

[1] C. D. Carothers, R. M. Fujimoto, Y. B. Lin, and P. England. 1994. Distributed simulation of large-scale PCS networks. In *Proceedings of International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. Durham, NC, USA, 2–6. https://doi.org/10.1109/MASCOT.1994.284456

[2] Christopher D. Carothers and Kalyan S. Perumalla. 2010. On Deciding Between Conservative and Optimistic Approaches on Massively Parallel Platforms. In *Proceedings of the Winter Simulation Conference (WSC '10)*. IEEE, Baltimore, Maryland, 678–687.

[3] Jörgen Dahl, Malolan Chetlur, and Philip A. Wilsey. 2001. Event List Management in Distributed Simulation. In *Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing (Euro-Par '01)*. Springer-Verlag, Berlin, Heidelberg, 466–475. http://dl.acm.org/citation.cfm?id=646666.699734

[4] Tom Dickman, Sounak Gupta, and Philip A. Wilsey. 2013. Event Pool Structures for PDES on Many-core Beowulf Clusters. In *Proceedings of ACM SIGSIM PADS*. ACM, New York, NY, USA, 103–114.

[5] Romain Franceschini, Paul-Antoine Bisgambiglia, and Paul Bisgambiglia. 2015. A Comparative Study of Pending Event Set Implementations for PDEVS Simulation. In *DEVS Integrative M&S Symposium*. SCS, San Diego, CA, USA, 77–84.

[6] Sounak Gupta and Philip A. Wilsey. 2014. Lock-free Pending Event Set Management in Time Warp. In *Proceedings of the ACM SIGSIM PADS*. ACM, New York, NY, USA, 15–26.

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

[8] Joshua Hay and Philip A. Wilsey. 2015. 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)*. ACM, New York, NY, USA, 75–86. https://doi.org/10.1145/2769458.2769462

[9] Julius Higiro, Meseret Gebre, and Dhananjai M. Rao. 2017. 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)*. ACM, New York, NY, USA, 3–14. https://doi.org/10.1145/3064911.3064921

[10] Shafagh Jafer, Qi Liu, and Gabriel Wainer. 2013. Synchronization methods in parallel and distributed discrete-event simulation. *Simulation Modelling Practice and Theory* 30 (2013), 54–73.

[11] Romolo Marotta, Mauro Ianni, Alessandro Pellegrini, and Francesco Quaglia. 2016. A Lock-Free O(1) Event Pool and Its Application to Share-Everything PDES Platforms. In *Proceedings of the 20th International Symposium on Distributed Simulation and Real-Time Applications (DS-RT '16)*. IEEE Press, Piscataway, NJ, USA, 53–60. https://doi.org/10.1109/DS-RT.2016.33

[12] Romolo Marotta, Mauro Ianni, Alessandro Pellegrini, and Francesco Quaglia. 2016. 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)*. ACM, ICST, Brussels, Belgium, Belgium, 46–55.

[13] Romolo Marotta, Mauro Ianni, Alessandro Pellegrini, and Francesco Quaglia. 2017. 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)*. ACM, New York, NY, USA, 15–26. https://doi.org/10.1145/3064911.3064926

[14] Francesco Quaglia. 2015. A low-overhead constant-time Lowest-Timestamp-First CPU scheduler for high-performance optimistic simulation platforms. *Simulation Modelling Practice and Theory* 53 (2015), 103–122. https://doi.org/10.1016/j.simpat.2015.01.009

[15] Dhananjai M. Rao. 2014. Accelerating Parallel Agent-based Epidemiological Simulations. In *Proceedings of the 2Nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM PADS '14)*. ACM, New York, NY, USA, 127–138. https://doi.org/10.1145/2601381.2601387

[16] Dhananjai M. Rao. 2016. Efficient parallel simulation of spatially-explicit agent-based epidemiological models. *J. Parallel and Distrib. Comput.* 93-94 (2016), 102–119. https://doi.org/10.1016/j.jpdc.2016.04.004

[17] Dhananjai M. Rao and Alexander Chernyakhovsky. 2008. Parallel Simulation of the Global Epidemiology of Avian Influenza. In *Proceedings of the 40th Conference on Winter Simulation (WSC '08)*. Winter Simulation Conference, 1583–1591.

[18] Wai Teng Tang, Rick Siow Mong Goh, and Ian Li-Jin Thng. 2005. Ladder Queue: An O(1) Priority Queue Structure for Large-scale Discrete Event Simulation. *ACM Trans. Model. Comput. Simul.* 15, 3 (July 2005), 175–204.

[19] Philip A. Wilsey. 2016. Some Properties of Events Executed in Discrete-Event Simulation Models. In *Proceedings of the 2016 Annual ACM Conference on SIGSIM Principles of Advanced Discrete Simulation (SIGSIM-PADS '16)*. ACM, New York, NY, USA, 165–176.

Received November 26 2017