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

Removed for double blind review
Institution
Street address
City, State Post code
double@blind.review

ABSTRACT

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

CCS CONCEPTS

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

KEYWORDS

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

1 INTRODUCTION

Sequential and parallel DES are designed as a set of logical processes (LPs) or "agents" that interact with each other by exchanging and processing timestamped events or messages [6]. Events that are yet to be processed are called "pending events". Pending events must be processed by LPs in priority order to maintain causality, with event priorities being determined by their timestamps. Consequently, data structures for managing and prioritizing pending events play a critical role in ensuring efficient sequential and parallel simulations [3, 4, 7, 8]. Effectiveness of data structures for event management is a conspicuous issue in larger simulations, where thousands or millions of events can be pending [1, 9]. Large pending event sets can arise when a model has many LPs or when each LP generates / processes many events. Overheads in managing pending events is magnified in fine grained simulations where the time taken to process an event is very short – *i.e.,* LPs use only few 100s to 1000s of instructions per event. Furthermore, the synchronization strategy used in PDES, Time Warp in particular [6], can further impact the effectiveness of the data structure due to additional processing 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* [9] has shown to be the most effective data structure for managing pending events [2, 3], particularly in sequential DES. Accordingly, we aimed to replace the heap-based data structures (discussed in Section 4) used in our Time Warp synchronized parallel simulator with the Ladder Queue. Section 4.5 discusses our Ladder Queue implementation and its fine-tuning.

The Ladder Queue outperformed our multi-tier heap-based data structures in certain sequential simulations, consistent with observations by other investigators [3, 9]. However, as detailed in Section 6, the Ladder Queue was substantially slower in two cases – ❶ high concurrency: larger number of concurrent events (*i.e.,* events with same timestamp) per LP, and ❷ Time Warp synchronized parallel simulations conducted on a distributed memory computing cluster. Conversely, our multi-tier data structures performed well in parallel simulations.

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

SIGSIM-PADS'17, Singapore

DOI: N.A

2 PARALLEL SIMULATOR OVERVIEW

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

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

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

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

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

❹ **Cancel pending events**: This operation is used as part of rollback recovery process to aggressively remove *all pending events* sent by a given LP (LP_{sender}) to another LP (LP_{dest}) at-or-after a given time ($t_{rollback}$). In our implementation, only one anti-message with send time $t_{rollback}$ is dispatched to LP_{dest} from LP_{sender} to cancel prior events sent by LP_{sender} to LP_{dest} at-or-after $t_{rollback}$. This feature short circuits the need to send a large number of anti-messages thereby enabling faster rollback recovery. This feature also reduces scans required to cancel events in Ladder Queue data structures discussed in Section 4.5 and Section 4.6.

2.1 Experimental Platform

The design of XXXX and the experiments reported in this paper were conducted using a distributed-memory compute cluster consisting of 80 compute nodes interconnected by 1 GBPS Ethernet. Each compute node has two quad-core Intel Xeon ® CPUs (E5520) running at 2.27 GHz with hyperthreading disabled. Each compute node has 32 GB of RAM (4 GB per core) in Non-Uniform Memory Access (NUMA) configuration. The cluster has an independent 1 GBPS Ethernet network to support a shared file system. The nodes run Red Hat Enterprise Linux 6, with Linux (kernel ver 2.6.32) and the cluster runs PBS/Torque. The simulation software was compiled using GCC version 4.9.2 (`-O3` optimization level) with OpenMPI 1.6.4. All debug assertions were turned off for maximum performance.

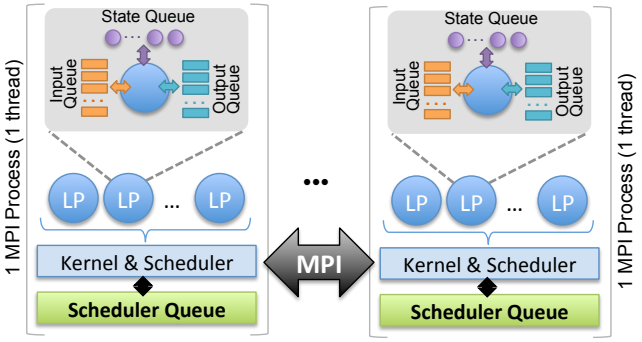


Figure 1: Overview of a parallel XXXX simulation

3 PHOLD BENCHMARK

The experimental analysis have been conducted using a parallelized version of the classic Hold synthetic benchmark called PHOLD (source code in supplements). It has been used by many investigators because it has shown to effectively emulate the steady-state phase of a typical simulation [3, 8]. Our PHOLD implementation developed using XXXX provides several parameters (specified as command-line arguments) summarized in Table 1. The benchmark consists of a 2-dimensional grid of Logical Processes (LPs) specified via the `rows` and `cols` parameters. The LPs are evenly partitioned across the MPI-processes used for simulation. The `imbalance` parameter influences the partition, with larger values skewing the partition as shown in Figure 2(a). The `imbalance` parameter has no impact in sequential simulations.

Table 1: Parameters in PHOLD benchmark

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

The PHOLD simulation commences with a fixed number of events for each LP, specified by the `eventsPerLP` parameter. For each event received by an LP a fixed number of trigonometric operations determined by `granularity` are performed to place CPU load. The impact of increasing the `granularity` parameter (no unit) is summarized in Figure 2(b) – smaller values result in finer grained simulations. For each event, an LP schedules another event to a randomly chosen adjacent LP. The `selfEvents` parameter controls the fraction of events that an LP schedules to itself.

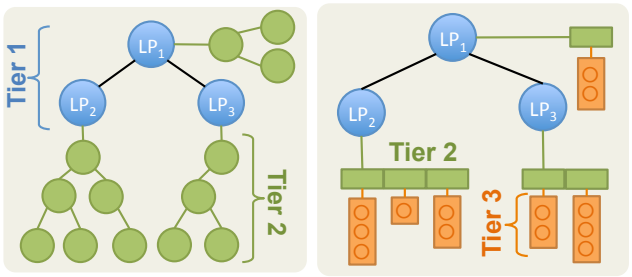
The event timestamps are determined by a given `delay--distrib` and `delay` or λ parameters. Our experiments use an exponential distribution for timestamps, because it has shown to reflect event distribution commonly found in a broad range of simulation models [8]. Timestamp of events is computed as t_{recv} = LVT + 1 + $\lambda e^{-\lambda x}$. The impact of changing the λ (*i.e.,* `delay`) is shown in Figure 2(c) – smaller values of λ provide a broader range of timestamp value for future events resulting in fewer concurrent events per LVT. Conversely, larger λ values cause timestamps to be close to the current epoch, increasing both the number of concurrent events per LVT and the possibility of rollbacks. Section 6 explores impact of these parameters on scheduler queue performance using 2,500 different configurations.

4 SCHEDULER QUEUES

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

4.1 Binary Heap (`heap`)

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



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

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

Table 2: Comparison of algorithmic time complexities of different data structures

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

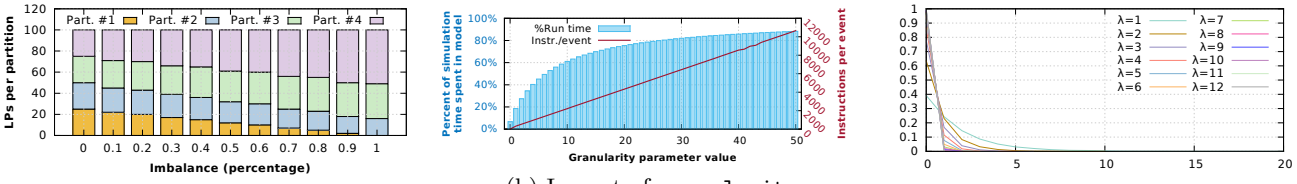
Name	Enqueue	Dequeue	Cancel
`heap`	$\log(e \cdot l)$	$\log(e \cdot l)$	$z \cdot \log(e \cdot l)$
`2tHeap`	$\log(e)+$ $\log(l)$	$\log(e)+$ $\log(l)$	$z \cdot \log(e)+$ $\log(l)$
`fibHeap`	$\log(e) + 1^*$	$\log(e) + 1^*$	$z \cdot \log(e) + 1^*$
`3tHeap`	$\log(\frac{e}{c}) + \log(l)$	$\log(l)$	$e + \log(l)$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

4.2 Two-tier Heap (`2tHeap`)

The `2tHeap` is designed to reduce the time complexity of cancel operations by subdividing events into two distinct tiers as shown in Figure 3. The first tier has containers for each local LP on an MPI-process. Each of the tier-1 containers contain a heap of events to be processed by a given LP. In `2tHeap` both tiers are maintained as independent binary heaps. Consequently, given l LPs and e pending events per LP, enqueue and dequeue operates require $\log e$ time to insert in tier-2 followed by $\log l$ 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 $\log e \cdot l$ 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.

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.



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

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

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

4.4 Three-tier Heap (`3tHeap`)

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

4.5 Ladder Queue (`ladderQ`)

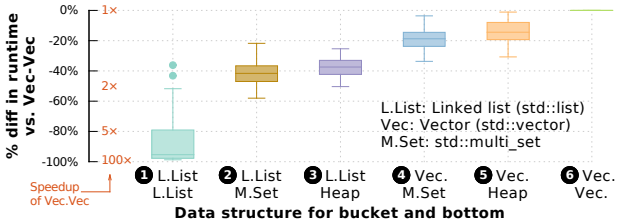
The `ladderQ` is a priority queue implementation proposed by Tang *et al* [8] with amortized constant time complexity as summarized in Table 2. Several investigators have independently verified that for sequential DES the `ladderQ` outperforms other priority queues, including: simple sorted list, binary heap, Splay tree, Calendar queue, and other multi-list data structures [2, 3, 8]. There are two key ideas underlying the Ladder Queue, namely: ① minimize the number of events to be sorted and ② delay sorting of events as much as possible. The multi-tier data structures also aim to minimize the number of events to be sorted. However, in contrast to the `ladderQ`, the other data structures always fix-up and maintain a minimum heap property.

The ladder queue consists of the following 3 substructures:

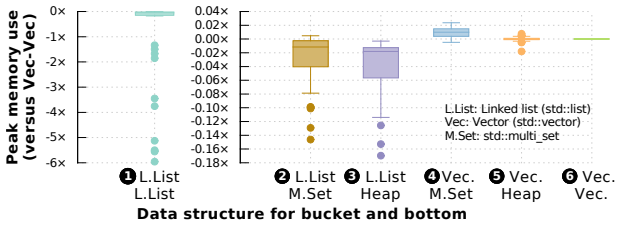
(1) *Top*: An unsorted list which contains events scheduled into the distant future or epoch.
(2) *Ladder*: Consists of multiple rungs, *i.e.,* list of buckets. Each bucket contains list of events with a finite range of timestamp values. Hence, although events within a bucket are not sorted, the buckets on a rung are organized in a sorted order. The `ladderQ` minimizes the number of events to be finally sorted by recursively breaking large buckets into smaller buckets in lower rungs of its ladder. Lower rungs in the ladder have smaller buckets with smaller time ranges.
(3) *Bottom*: This substructure contains a sorted list of events to be processed. Inserts into *Bottom* must preserve sorted order. Hence, the `ladderQ` strives to maintain a short bottom by moving events back into the ladder, as needed [8].

4.5.1 Fine tuning Ladder Queue performance. Our implementation closely followed the design in the original paper by Tang *et al* [8]. 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 ($\log n$ operations) for bottom, ❸ L.List-Heap: a L.List and a binary heap (backed by a `std::vector`) for bottom, ❹ Vec-M.Set: a dynamically growing array (*i.e.,* `std::vector`) for buckets and Multi-set bottom, ❺ Vec-Heap: Vector buckets and binary heap for bottom, and ❻ Vec-Vec: Vector for buckets and bottom. This configuration enables using quick sort (*i.e.,* `std::sort`) for sorting buckets and binary search for inserting events into bottom.

Runtime comparison of the 6 `ladderQ` configurations is summarized in Figure 4. The data was obtained using PHOLD with different parameter settings. The ❻th Vec-Vec configuration was the fastest and performance of other configurations are shown relative to it in Figure 4(a). The L.List-L.List configuration was generally the slowest and performed 85× (or ~98%) slower than the Vec-Vec configuration. The peak memory used for simulations is shown in Figure 4(b), in comparison with the Vec-Vec configuration. As shown by the charts in Figure 4, the increased performance of Vec-Vec



(a) Comparison sequential runtimes



(b) Peak memory used

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

comes at about a 6× increase in peak memory footprint when compared to L.List-L.List configuration. This increased footprint arises because the `std::vector` internally doubles its capacity as it grows. With many buckets in the `ladderQ`, each implemented using a `std::vector`, the overall peak memory footprint is higher. Certainly, the increased capacity is used if the number of events in buckets grow. However, the Vec-M.Set and Vec-Heap configurations consume a bit more memory in some configurations, showing that Vec-Vec is not the worst in memory consumption. Consequently, we use the Vec-Vec configuration as it provides the fastest performance among the 6 configurations (source code in supplements).

The maximum number of rungs in the *Ladder* also influences the overall performance of the `ladderQ` [8]. The chart in Figure 5 illustrates the impact of limiting the maximum number of rungs in the `ladderQ`. When the rungs are too few, the timestamp-based width of buckets is larger and more events with many different timestamps are packed into buckets. This also causes the *Bottom* to be longer with events spanning a broader range of timestamps. Consequently, when inserts happen into *Bottom*, many *Bottom*-to-*Ladder* re-bucketing operations are triggered to ensure bottom is short. These re-bucketing operations with many events significantly degrade performance. However, once sufficient number of rungs (6 rungs in this case) are permitted the events are better subdivide into smaller timestamp-based bucket widths. Small bucket widths in turn minimize inserts into bottom and *Bottom*-to-*Ladder* operations, ensuring good performance.

The chart in Figure 5 shows that a minimum of 6 rungs is required. For some select configurations of larger models we observed (data not shown) that 5 rungs would be sufficient. However, the number of rungs cannot exceed beyond

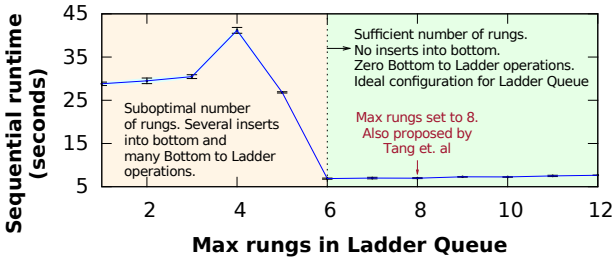


Figure 5: Impact of limiting rungs in Lader

a threshold to avoid infinite spawning of rungs [8]. Moreover, it limits the overheads involved in re-bucketing events from rung-to-rung [8]. 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* [8]. Furthermore, we trigger *Bottom*-to-*Ladder* re-bucketing only if the *Bottom* has events at different timestamps to further reduce inefficiencies.

4.5.2 Shortcoming of Ladder Queue for optimistic PDES. The amortized constant time complexity of enqueue and dequeue operations enable the `ladderQ` to outperform other data structures in sequential simulations [2, 3, 8]. 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.,* $e \cdot l$ 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, XXXX utilizes only one anti-message to from LP_{sender} to LP_{dest} to cancel all n events sent after $t_{rollback}$ (rather than sending n individual anti-messages) which reduces overheads. Furthermore, with our centralized scheduler design, only events received from LPs on other MPI-processes can trigger rollbacks. Consequently, the number of scans of the `ladderQ` that actually occurr is significantly fewer in our case, despite the aggressive cancellation strategy.

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

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

Consequently, to reduce cost of event cancellation, we propose a 2-tier Ladder Queue (`2tLadderQ`) in which each bucket in *Top* and *Ladder* is further subdivided into $_{t2}k$

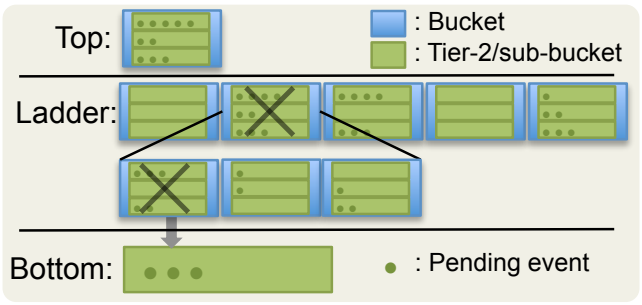


Figure 6: Structure of 2-tier Ladder Queue (`2tLadderQ`) with 3 sub-buckets / bucket (*i.e.,* $_{t2}k$=3)

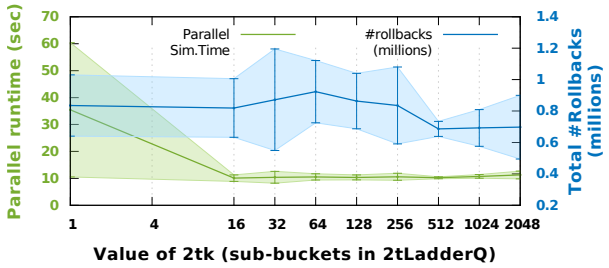


Figure 7: Effect of varying $_{t2}k$

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

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

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

5 RELATED WORK

This paper proposes and explores multi-tier data structures for managing the pending event set in sequential and optimistic parallel simulations. Specifically, we compare effectiveness of the data structures against our fine-tuned version of the Ladder Queue [8] because it has shown to be very efficient for sequential Discrete Event Simulation (DES). Recently, Franceschini *et al* [3] compared several priority-queue based event list data structures to evaluate their performance in the context of sequential DEVS simulations. They found that the Ladder Queue outperformed every other priority queue based event lists data structure such as Sorted List, Minimal List, Binary Heap, Splay Tree, and Calendar Queue. We refer readers to the work by Tang *et al* [8] and Franceschini *et al* [3] for comparative discussion on the different data structures. They both use the classic Hold benchmark used in this study.

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

Dickman *et al* [2] compare event list data structures that consisted of Splay Tree, STL Multiset and Ladder Queue. However, the focus of their paper was in developing a framework for handling event list data structures in shared memory PDES. A central component of their study was the identification of an appropriate data structure and design for the shared event list. Gupta *et al* [4] extended their implementation of Ladder Queue for shared memory Time Warp based simulation environment, so that it supports lock-free access to events in the shared event lists. The modification involved

the use of an unsorted lock-free queue in the underlying ladder queue structure. Marotta *et al* [7] have contributed to the study of event list data structures in threaded PDES through the design of the Non-Blocking Priority Queue (NBPQ) data structure. An event list data structure that is closely related to Calendar Queues with constant time performance.

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

6 EXPERIMENTS & DISCUSSIONS

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

6.1 Parameter reduction via GSA

Generalized Sensitivity Analysis (GSA) is based on two-sample Kolmogorov-Smirnov Test (KS-Test) and yields a $d_{m,n}$ statistic that is sensitive to differences in both central tendency and differences in the distribution functions of parameters [5]. The $d_{m,n}$ statistic is the maximum separation between cumulative probability distribution observed in a two-sample KS-Test. The KS-Test is performed with data from Monte Carlo simulations involving combinations of parameter values from a specified range or probability distribution. The simulation result is then classified into number of "success" (m) or its converse "failure" (n) to compute cumulative probability distribution and $d_{m,n}$ statistic for each parameter. In this study we have defined "failure" to be parameter values for which the `2tLadderQ` runs slower when compared to another scheduler queue. For sequential and parallel simulations we use ${}_{t2}k$=1 and ${}_{t2}k$=128 respectively.

An important aspect of GSA is to ensure that the values for each parameter covers its full range of values. Consequently, we use Sobol random numbers to select a combination of PHOLD parameter values to be used for simulation.

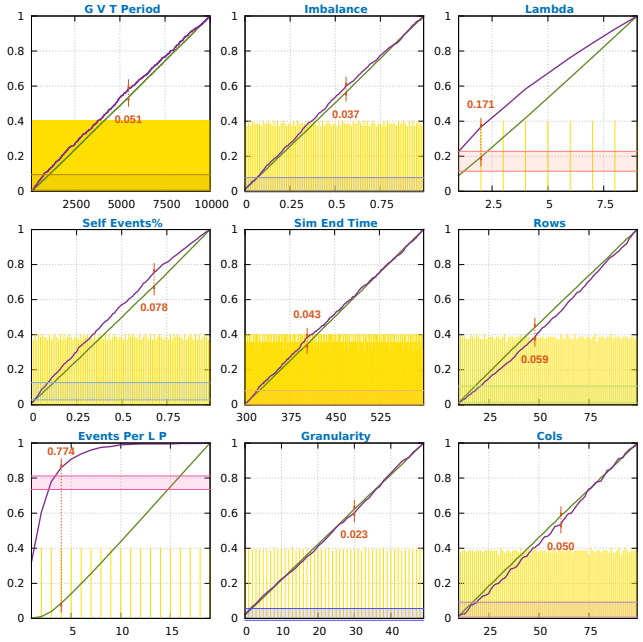


Figure 8: Results from Generalized Sensitivity Analysis (GSA) comparing `3tHeap` and `2tLadderQ` for sequential simulation (more stats in supplements).

Sobol random numbers are quasi-random low-discrepancy sequences that provide uniform coverage of a multidimensional parameter space for PHOLD (see Figure 2). Our parameter ranges also ensure that the peak memory consumption do not cross NUMA threshold, which in our case is 4 GB of RAM. Exceeding the 4 GB NUMA threshold introduces a lot of variance in runtimes requiring many runs to reduce variance to acceptable limits.

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

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

The chart in Figure 9 shows the summary of the $d_{m,n}$ statistic or influence of each parameter (see Table 1) on the outcome – *i.e.,* `2tLadderQ` performs better or worse than

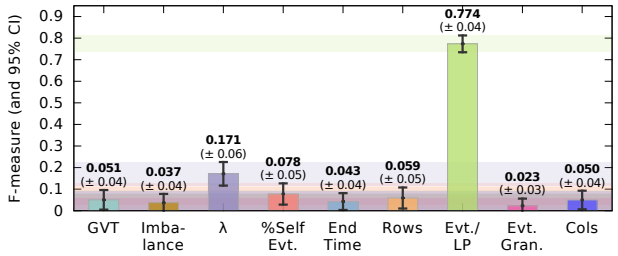


Figure 9: Summary of influential parameters from Figure 8 that cause performance differences between `2tLadderQ` and `3tLadder` in sequential simulations.

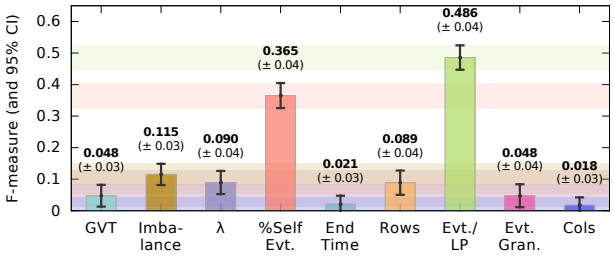


Figure 10: GSA data from parallel simulations (4 MPI-processes) showing influential parameters (`2tLadderQ` vs. `3tHeap`).

`3tHeap`. The lightly shaded bands show the 95% Confidence Intervals (CI) computed using standard bootstrap approach using 5000 replications with 1000 samples in each. As expected, the `imbalance` (*i.e.,* skew in partition) has no impact in sequential simulation and has a low impact score of 0.037. Similarly, the GVT computation rate does not impact pending events and consequently its influence is low at 0.051.

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

We have also conducted GSA to determine influential parameters impacting performance of other scheduler queues versus the `2tLadderQ` in sequential simulations (charts in supplements). Our analysis showed that none of the parameters play an influential role and the `2tLadderQ` performed consistently better or the same when compared to `ladderQ`, `2tHeap`, `fibHeap`, and `heap`. Only `3tHeap` and in few cases `2tHeap` outperformed our `2tLadderQ` in certain configurations. The performance of `ladderQ` and `2tLadderQ` was practically indistinguishable in sequential simulations (with ${}_{t2}k$=1).

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

6.1.2 GSA results for Parallel simulations. GSA for parallel simulations were conducted using the same procedure discussed earlier but using 4 MPI-processes for parallel simulation. These analysis focused only on `ladderQ`, `2tLadderQ`, and `3tHeap` based on the inferences drawn from the earlier analyses. The average simulation execution time from 3 replications is recorded for each scheduler queue along with the parameter set. Initially, we observed that the `ladderQ` timings showed a lot of variance in runtime depending on number of rollbacks that occur. Consequently, to reduce variance, we have used a time-window of 10 time-units to curtail optimism and reduce rollbacks. The time-window restricts the simulation kernel from scheduling events that are more than 10 time-units ahead of GVT. We use the same time-window for all scheduler queues for consistent comparison and analysis.

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

6.1.3 PHOLD configurations for further analysis. The Generalized Sensitivity Analysis (GSA) enables identification of influential parameters, thereby substantially reducing the parameter space. However, GSA data does not provide an effective data set to analyze trends, such as: scalability, memory usage, rollback behaviors, etc. In order to pursue such analysis we have used 3 different PHOLD configurations called `ph3`, `ph4`, and `ph5`. The fixed characteristics for the 3 configurations with non-influential parameters is summarized in Table 3. We use larger simulation end times for parallel simulation so obtain sufficiently long runtimes using 32 cores. The value of influential parameters, namely: `eventsPerLP`, `%selfEvents`, and λ is varied for comparing different settings, similar to the approach used by other investigators [3, 8].

Table 3: Configurations used for further analysis

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

6.2 Sequential simulation results

Sequential simulations were conducted to assess the effectiveness of the different data structures. We pursued sequential simulations to compare the base case performance of the data structures, consistent with prior investigations [3, 8]. The sequential simulations also serve as a reference for potential use in conservatively synchronized PDES. The sequential experiments were conducted using 3 PHOLD configurations (see Section 6.1.3) on one compute node of our cluster described in Section 2.1. The simulations use only 1 MPI-process and states are not saved . Number of sub-buckets in `2tLadderQ` was set to 1, *i.e.,* ${}_{t2}k$=1. For these experiments, the influential parameters `eventsPerLP`, λ, and `%selfEvents` were varied to explore their impact on relative performance of the data structures. Event `granularity` was set to zero resulting in a fine grained simulation. For each configuration, data from 10 independent replications were collected and analyzed.

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

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

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

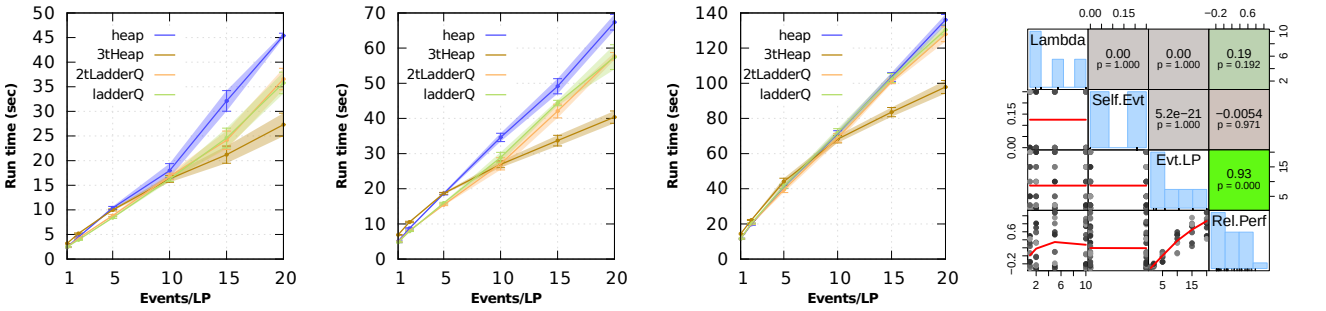
The chart in Figure 11(d) shows the correlation between the 3 influential parameters and the performance difference between `3tHeap` and `ladderQ`. Consistent with the GSA results, the corellogram shows that the most influential parameter is `eventsPerLP` (R=0.93, p=0) followed by λ (R=0.19, p=0.192)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

The charts in Figure 12 shows the peak memory usage corresponding to the runtime data in Figure 11. The memory size reported is the "Maximum resident set size" value reported by GNU `/usr/bin/time` command on Linux. The memory usage of `heap` is the lowest in most cases. Since ${}_{t2}k$=1, the memory usage of `ladderQ` and `2tLadderQ` is comparable as expected. The `3tHeap` initially uses more memory than the other data structures because of many small `std::vector`s and due to `std::vector` doubling its capacity. However, the memory usage is amortized as the `eventsPerLP` increases. Consequently, the improved performance of `3tHeap` over `ladderQ` is realized without significant increase in memory footprint.

6.3 Parallel simulation assessments

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

6.3.1 Throttling optimism with a time-window. Initially we conducted experiments with fine-grained setting (*i.e.,* `granularity` = 0) from sequential simulations. We noticed that the `ladderQ` had a large variance in runtimes, particularly when it experienced many rollbacks. In several cases, cascading rollbacks significantly slowed the simulations – *i.e.,* `ladderQ` simulations required over 1 hour while `2tLadderQ` would consistently finish in a few minutes. In order to avoid such debilitating rollback scenarios and to streamline experimental analysis timeframes we have throttled optimism using a time-window of 10 time-units. The time-window restricts the simulation kernel from scheduling events that are more than 10 time-units ahead of GVT. The time-window value of 10 is 50% of the maximum timestamp of events generated by exponential distribution with $\lambda = 1$. Consequently, most events in current schedule cycle will fit within this time-window with limited impact on concurrency. We use the same time-window for all scheduler queues for consistent comparison and analysis.



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

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

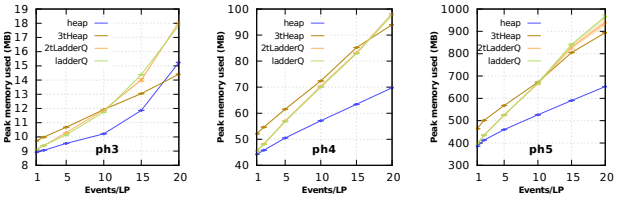


Figure 12: Comparison of peak memory usage

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

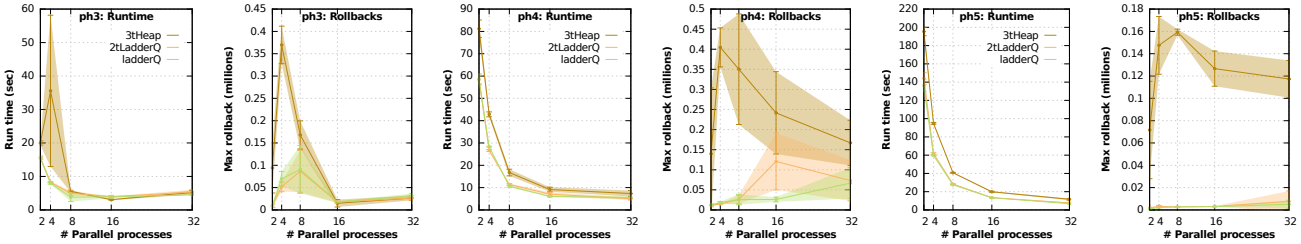
In this configuration, the `3tHeap` runs experienced a lot of rollbacks when compared to the other two queues despite the time-window. For `ph5` data in Figure 13(c), `3tHeap` experienced about 114805 rollbacks on average while `ladderQ` experienced only 2341, almost 50×fewer rollbacks. Consequently,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

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

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

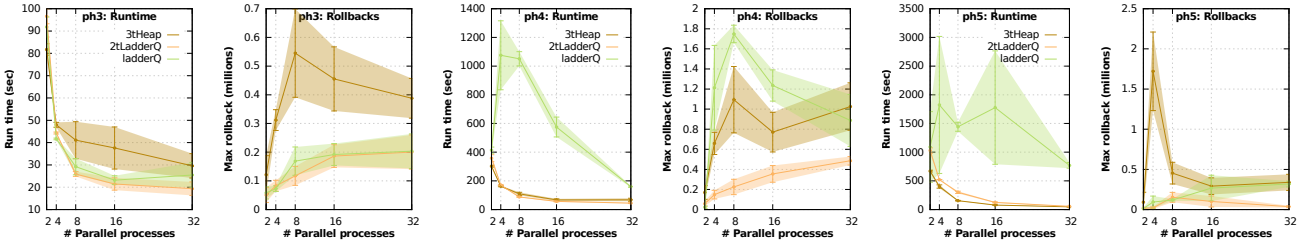
7 CONCLUSIONS

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



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

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



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

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

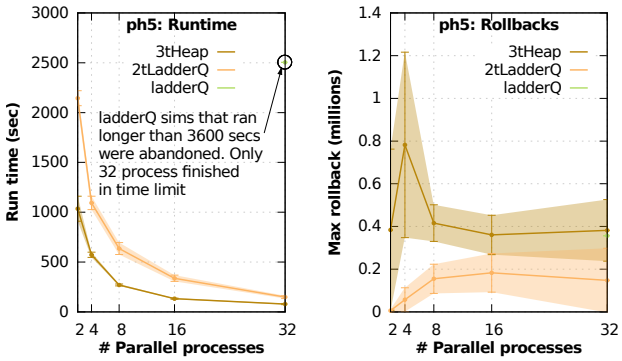


Figure 15: ph5 Statistics (best case for 3tHeap)

The comparative analysis used a significantly fine-tuned version of the Ladder Queue (`ladderQ`) [8]. The objective of fine-tuning was to reduce the runtime constants of the `ladderQ` without significantly impacting its amortized O(1*) time complexity. Reduction in runtime constants is primarily realized by minimizing memory management overheads – *i.e.,* ❶ favor few bulk operations via `std::vector` than many small linked list nodes in `std::list` and ❷ recycle memory or substructures rather than reallocating them. Using `std::vector` (*i.e.,* dynamically growing array) enables use of algorithms with lower time constants, such as: `std::sort`, over `std::multiset` or binary heaps. The bulk memory operations do consume additional memory, but our analysis shows that the performance gains significantly outweigh the extra memory used. Ergo other simulation kernels can significantly improve overall performance by replacing linked lists with dynamically growing arrays.

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

The sequential and parallel simulation results showed that `2tLadderQ` performs no worse than our fine-tuned `ladderQ` in sequential simulations (with $_{t2}k$=1). Furthermore, our `2tLadderQ` outperforms our `ladderQ` in parallel simulations because of its design that enables rapid cancellation of events during rollbacks. In fact, the `ladderQ` required aggressive throttling of optimism without which `ladderQ` was impractical to use in scenarios with many cascading rollbacks. These experiments were conducted with fine-grained settings (*i.e.,* `granularity`=0) and results may vary with granularity. However, GSA data suggests that the variation with changing

granularity would be small. However, increased granularity may allow relaxation of the time-window. The results strongly favor the general use of `2tLadderQ` over the `ladderQ`. Furthermore, the multi-tier organization of `2tLadderQ` can further reduce lock contention and consequent synchronization overheads in multithreaded simulations.

The experiments show that the runtime constants play an important role – for example, the Fibonacci heap with its $O(1^*)$ time complexity for many operations still did not perform well in our benchmarks. The `3tHeap` has a much lower runtime constants enabling it to outperform the `fibHeap` in almost all cases. In sequential simulations, the advantages of `3tHeap` are realized in simulations that have higher concurrency (*i.e.,* larger batches of events) per LP. Figure 16 summarizes the effective regions observed for the 3 queues. The advantages of `3tHeap` is realized only when each LP has 10 or more concurrent events at each time step. Such scenarios with high `eventsPerLP` arises in epidemic models [9] and detailed simulation models such as packet-level network simulations [8]. However, further experimental analysis with such models is needed to formally verify effectiveness of `3tHeap`.

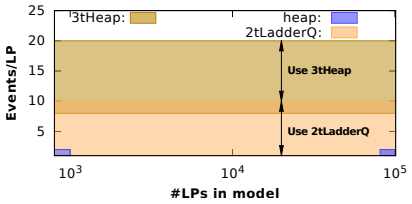


Figure 16: Effective regions of use

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

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

REFERENCES

[1] 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)*. Winter Simulation Conference, 678–687.

[2] Tom Dickman, Sounak Gupta, and Philip A. Wilsey. 2013. Event Pool Structures for PDES on Many-core Beowulf Clusters. In *Proceedings of the 1st ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM PADS '13)*. ACM, New York, NY, USA, 103–114.

[3] Romain Franceschini, Paul-Antoine Bisgambiglia, and Paul Bisgambiglia. 2015. A Comparative Study of Pending Event Set Implementations for PDEVS Simulation. In *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium (DEVS '15)*. Society for Computer Simulation International, San Diego, CA, USA, 77–84.

[4] Sounak Gupta and Philip A. Wilsey. 2014. Lock-free Pending Event Set Management in Time Warp. In *Proceedings of the 2Nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM PADS '14)*. ACM, New York, NY, USA, 15–26.

[5] 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.

[6] 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.

[7] 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)*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium, 46–55.

[8] 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.

[9] Jae-Seung Yeom, Abhinav Bhatele, Keith Bisset, Eric Bohm, Abhishek Gupta, Laxmikant V. Kale, Madhav Marathe, Dimitrios S. Nikolopoulos, Martin Schulz, and Lukasz Wesolowski. 2014. Overcoming the Scalability Challenges of Epidemic Simulations on Blue Waters. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS '14)*. IEEE Computer Society, Washington, DC, USA, 755–764.

Supplementary material includes source code and additional statistics to aid in double blind review process. In the final version of the paper the supplementary material will be made available online along with source code repository. The community can use the source code to reproduce the results reported in the paper.

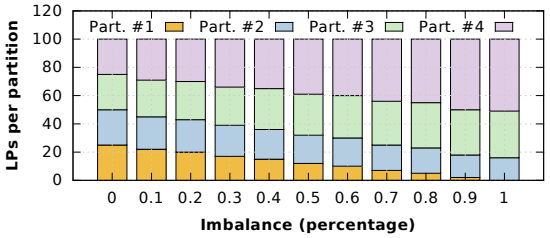
Supplementary Materials

This document is provided as supplementary material to aid in double-blind review process. In the final version of the paper the supplementary material will be made available online along with full copy of the source code. The community can use the source code to verify/falsify the results reported in the paper.

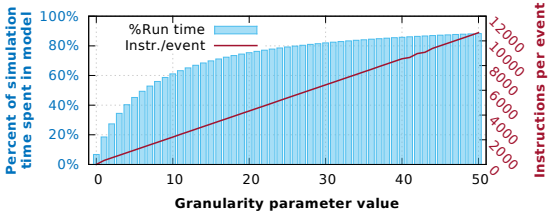
Table of Contents

Reducing the parameter space for analysis

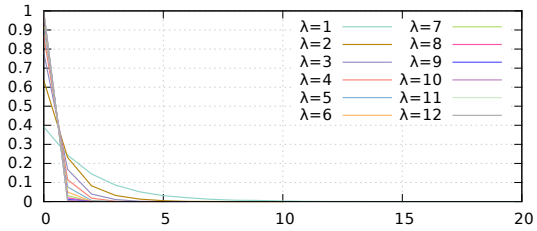
In this study we have used the standard PHOLD benchmark (also used by other investigators) to assess the comparative performance of the 6 different scheduler queues used in this study. However, that required exploring 9 different parameters with a broad range of parameter settings. The large parameter space makes it impractical for experimental analysis of 6 different queues. Consequently, we have used Generalized Sensitivity Analysis (GSA) to reduce the parameter space for analysis. It is important to ensure that GSA covers the necessary range of parameters pertinent to this study. So first we have characterized the behavior of key parameters in PHOLD. The impact of 3 key parameters in PHOLD is shown in the figures below (same figures from PADS paper):



(a) impact of `imbalance`



(b) impact of `granularity`



(c) impact of λ (`delay`)

For other parameters we have explored a broad range of values, namely: ❶ `rows`: 1 to 100, ❷ `cols`: 1 to 100, ❸ `eventsPerLP`: 1 to 20, and ❹ `GVT` rate/period (i.e., number of scheduler cycles after which a GVT computation is triggered if one is not already underway): 1000 to 10,000. These are the range of values in the x-axes of the GSA sub-charts. For `rows` and `cols` we have limited them to 100×100 so that at `eventsPerLP` = 20, the peak memory usage of the simulation will not exceed the NUMA memory limit (4 GB/core) of the compute nodes. The NUMA hardware configuration of the compute nodes is shown in the adjacent figure. Exceeding the NUMA threshold caused the sequential simulation times to vary a lot and 3 simulation replications were no longer sufficient to get a good estimate.

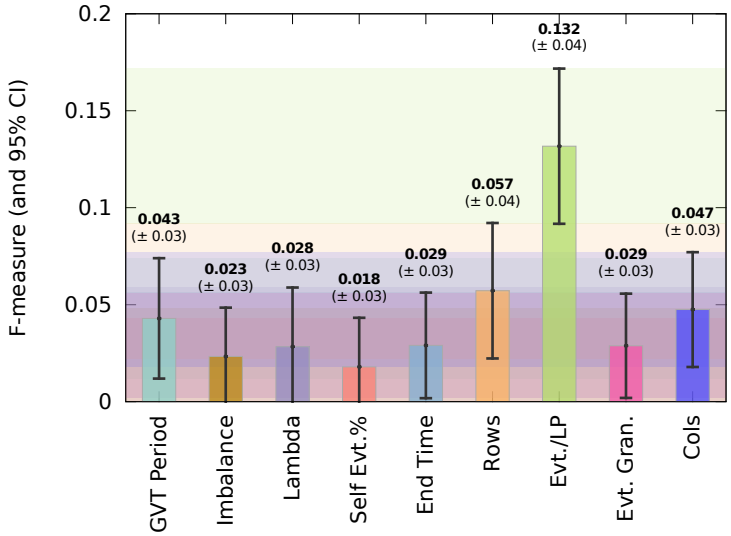
```
$ numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 2 4 6
node 0 size: 16384 MB
node 0 free: 15383 MB
node 1 cpus: 1 3 5 7
node 1 size: 16371 MB
node 1 free: 15304 MB
node distances:
node   0   1
  0:  10  20
  1:  20  10
```

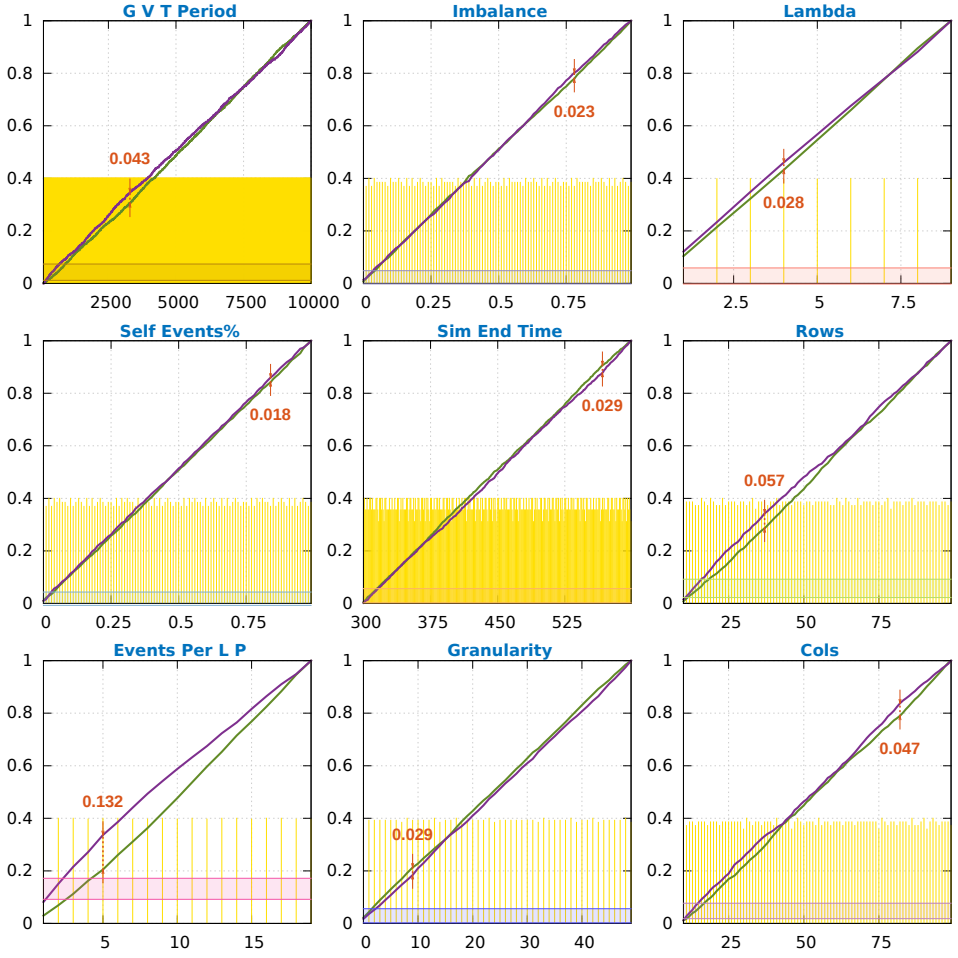
Generalized Sensitivity Analysis (GSA) for Sequential Simulations

This section includes results from the various GSA that was conducted to assess the impact of various PHOLD parameters on relative performance of the six scheduler queues. The 2-tier Ladder Queue (2tLadderQ) is used as the reference for comparison in all cases. The data in each figure is from 2500 (parameter combination) × 3 (reps/combo) = 7500 simulations.

Ladder Queue (`ladderQ`) vs. 2-tier Ladder Queue (`2tLadderQ`)

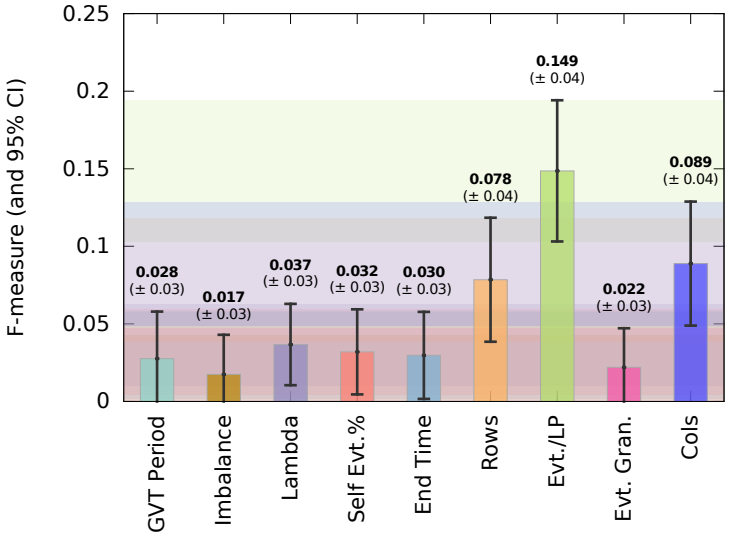
The GSA data shows that in sequential simulations, the performance of `2tLadderQ` (with $_{2t}$k=1) is very close to that of `ladderQ` immaterial of the parameter settings. The `eventsPerLP` parameter slightly increases the performance of `2tLadderQ`.

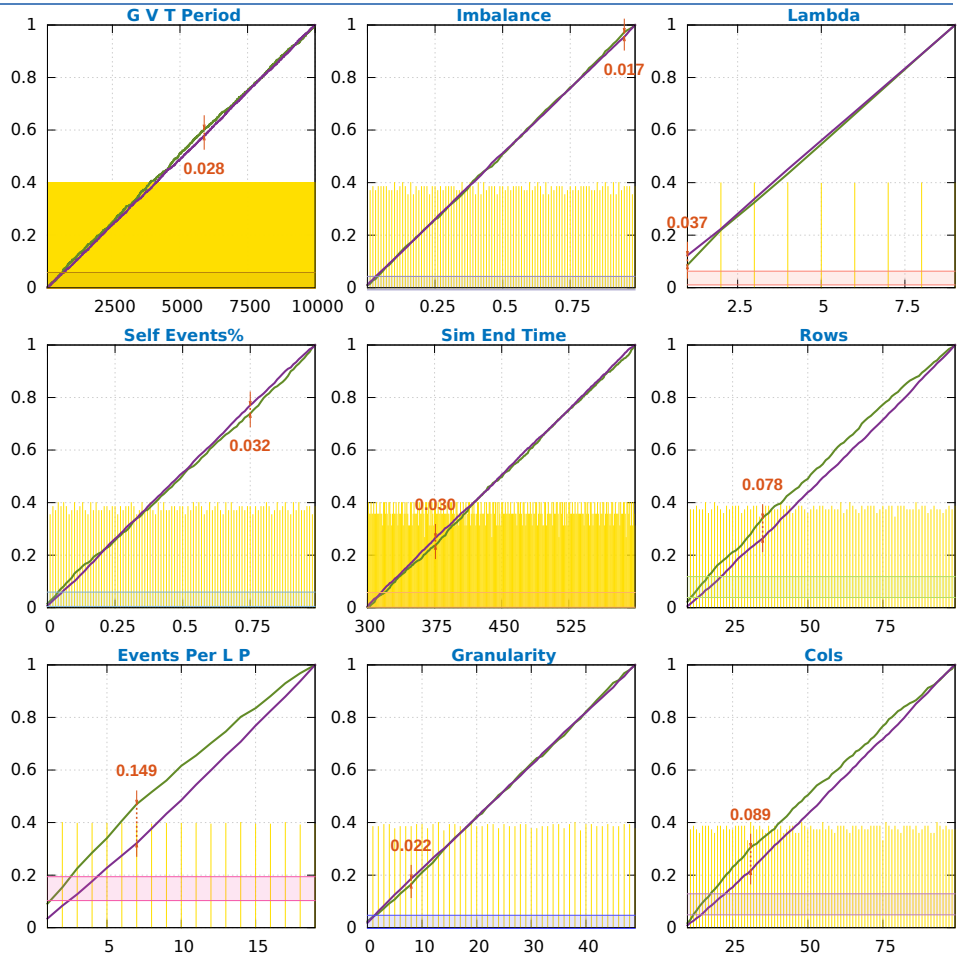




2-tier Ladder Queue (`2tLadderQ`) vs. 1-tier binary Heap (`heap`)

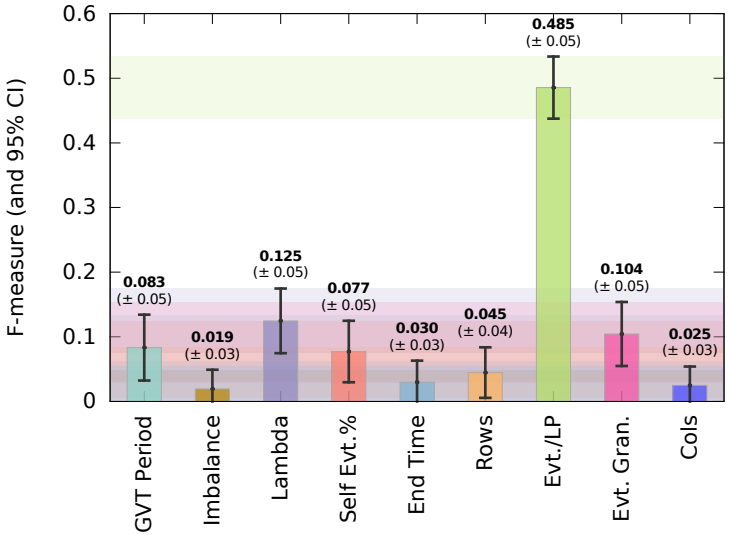
The GSA data shows that in sequential simulations, most of the parameters do not influence performance difference between `2tLadderQ` (with $_{2t}k=1$) and `heap` with $d_{m,n} << 0.1$. Experiments show that `2tLadderQ` generally performs better than heap. Higher values of `Evt./Agent` (`eventsPerLP`) parameter cause the `2tLadderQ` to perform much better than the `heap`.

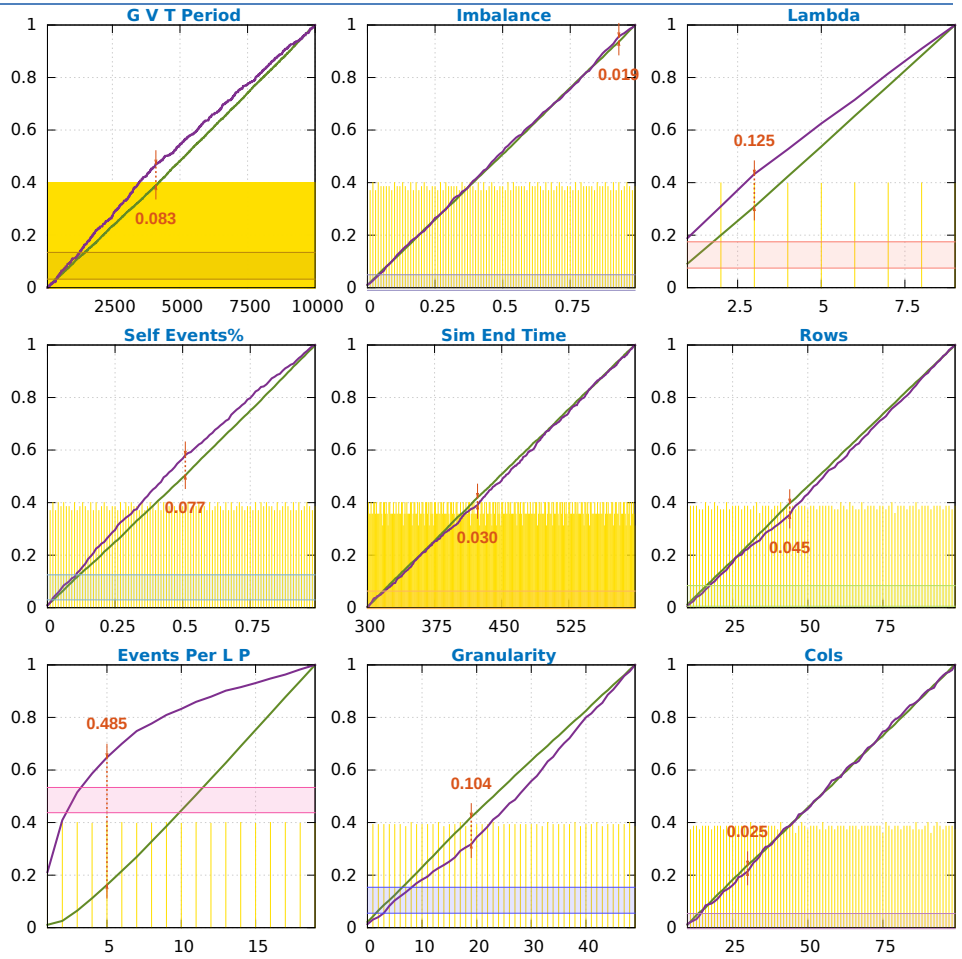




2-tier Ladder Queue (`2tLadderQ`) vs. 2-tier Heap (`2tHeap`)

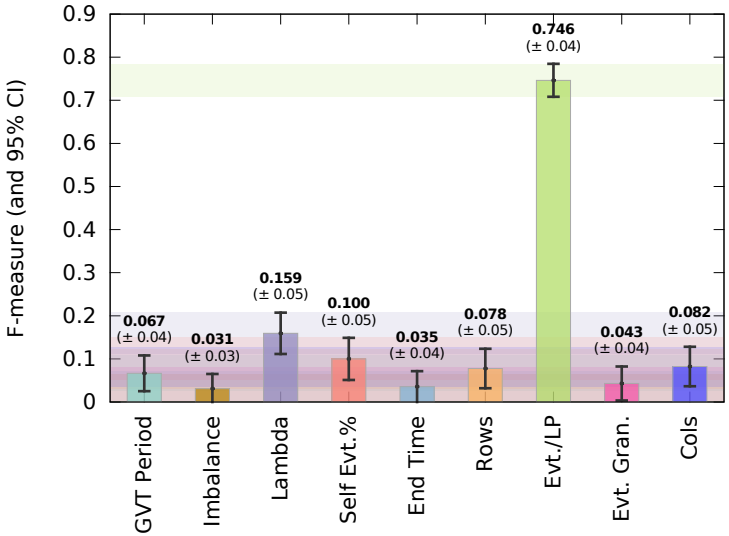
The GSA data shows that in sequential simulations, most of the parameters do not influence performance difference between `2tLadderQ` (with $_{2t}$k=1) and `2tHeap` with $d_{m,n}$ << 0.1. Experiments show that `2tLadderQ` generally performs better than heap. However, higher values of `Evt./LP` (`eventsPerLP`) parameter cause the `2tHeap` to start performing better than the `2tLadderQ`.

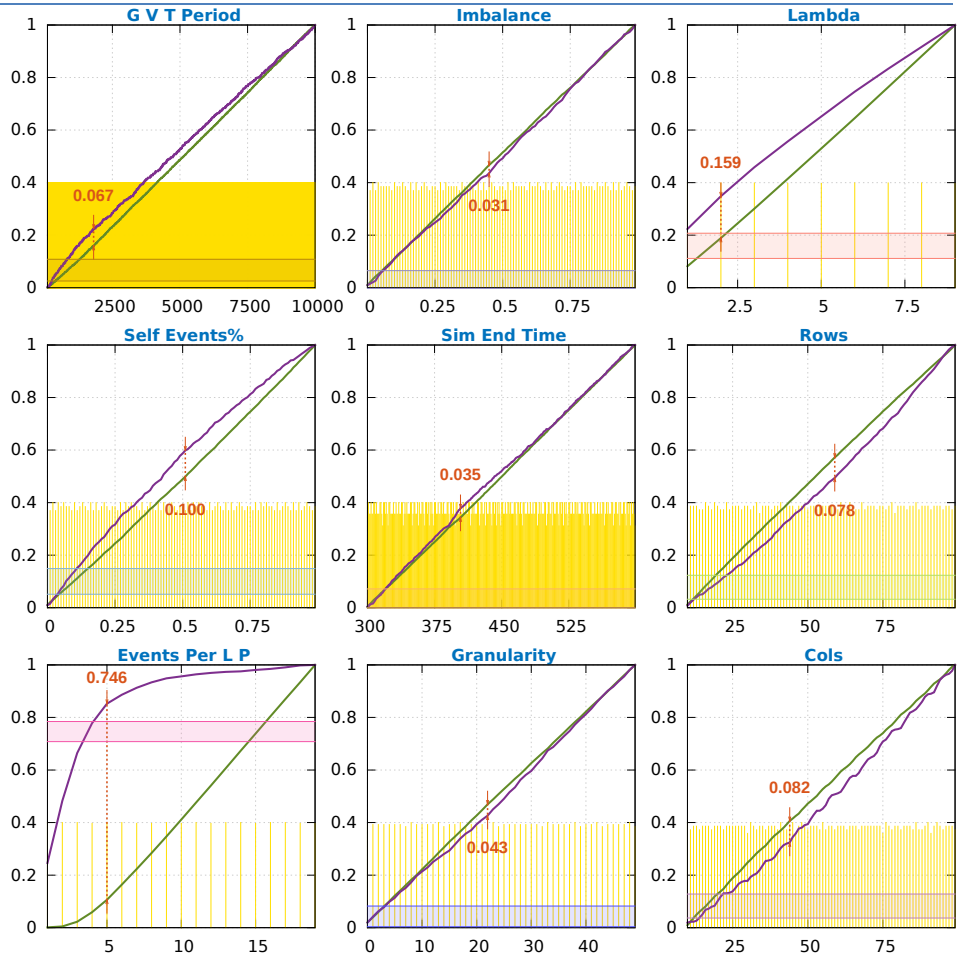




2-tier Ladder Queue (`2tLadderQ`) vs. 2-tier Fibonacci Heap (`fibHeap`)

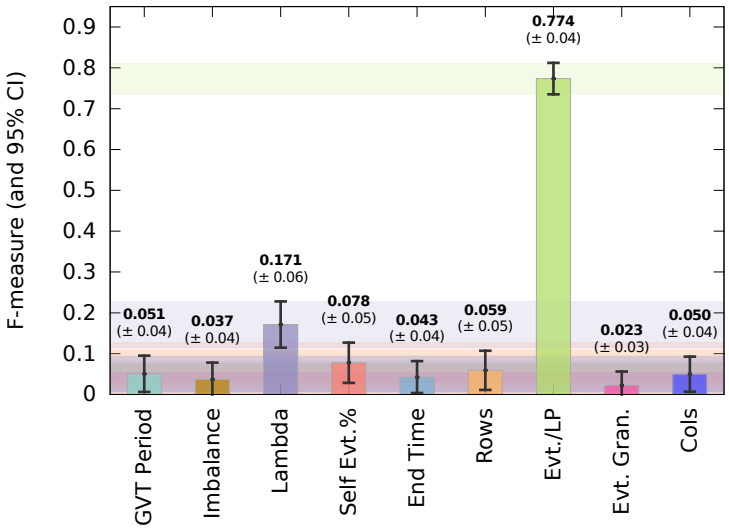
The GSA data shows that in sequential simulations, most of the parameters do not influence performance difference between `2tLadderQ` (with $_{2t}$k=1) and `fibHeap` with $d_{m,n}$ << 0.1. Experiments show that `2tLadderQ` generally performs better than `fibHeap`. However, higher values of `Evt./LP` (`eventsPerLP`) parameter cause the `fibHeap` to start performing better and catches up with the performance of the `2tLadderQ`.

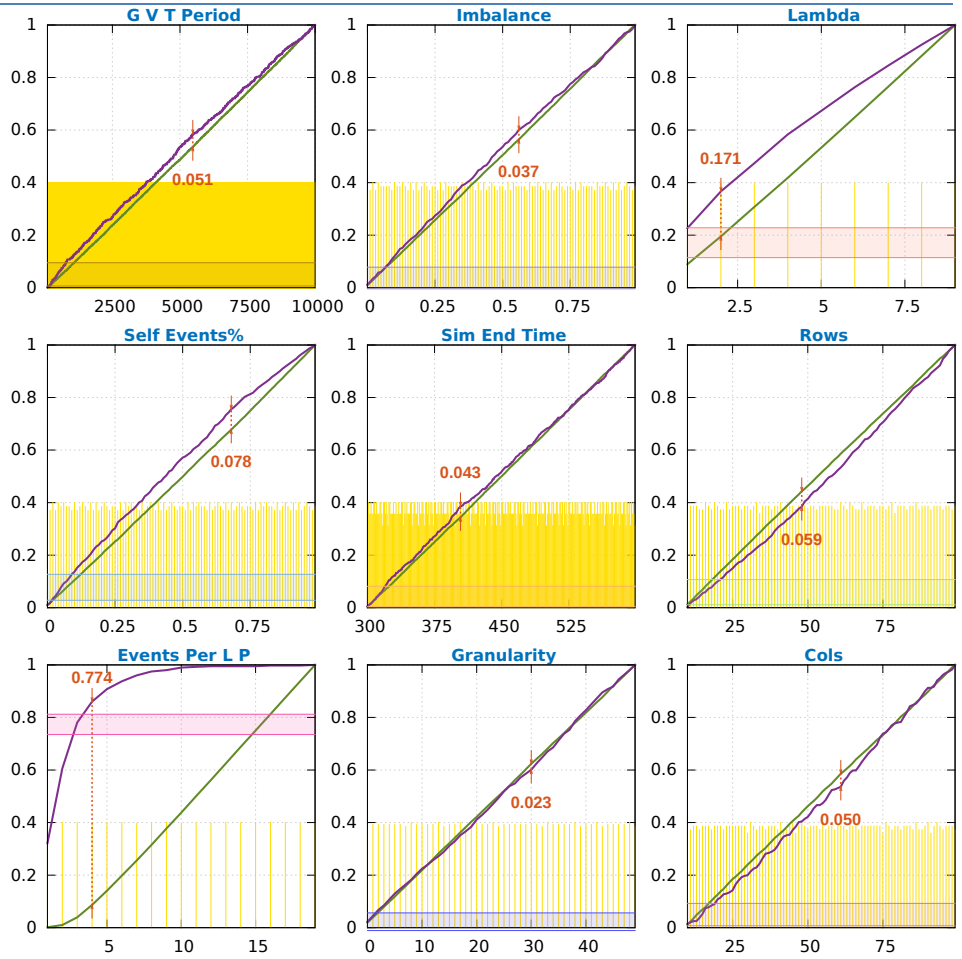




2-tier Ladder Queue (`2tLadderQ`) vs. 3-tier Heap (`3tHeap`)

This is the same data shown in the PADS paper. The GSA data shows that in sequential simulations, most of the parameters do not influence performance difference between `2tLadderQ` (with $_{2t}k=1$) and `3tHeap` with $d_{m,n} << 0.1$. Experiments show that 2tLadderQ generally performs better than heap. However, higher values of `Evt./LP` (`eventsPerLP`) parameter cause the `3tHeap` to start performing better than the `2tLadderQ`.





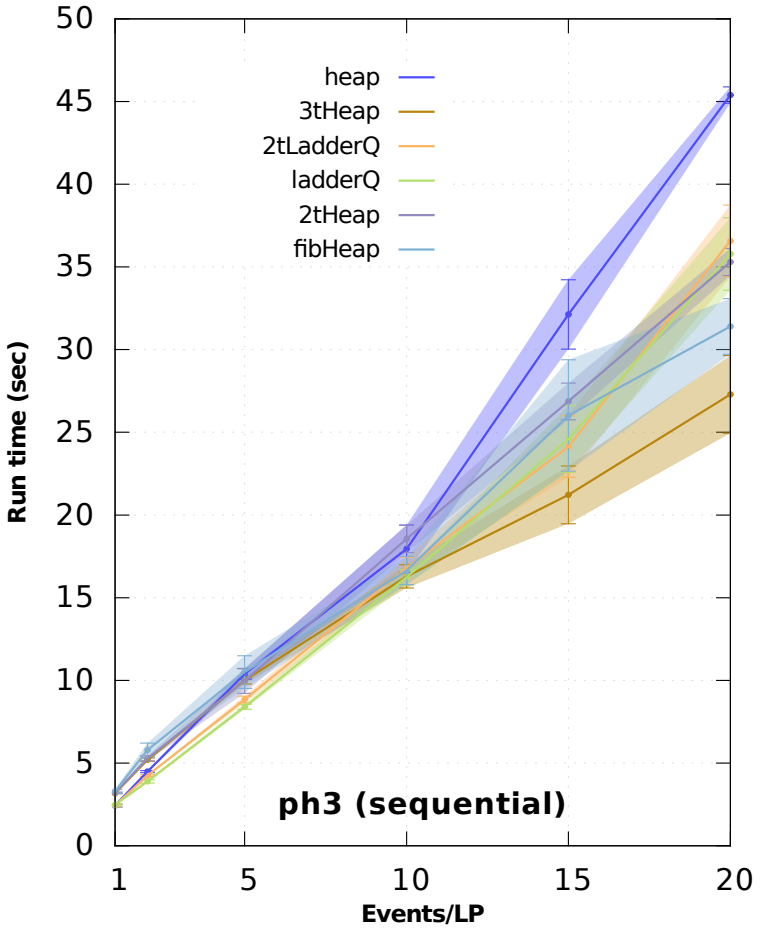
Additional sequential simulation results

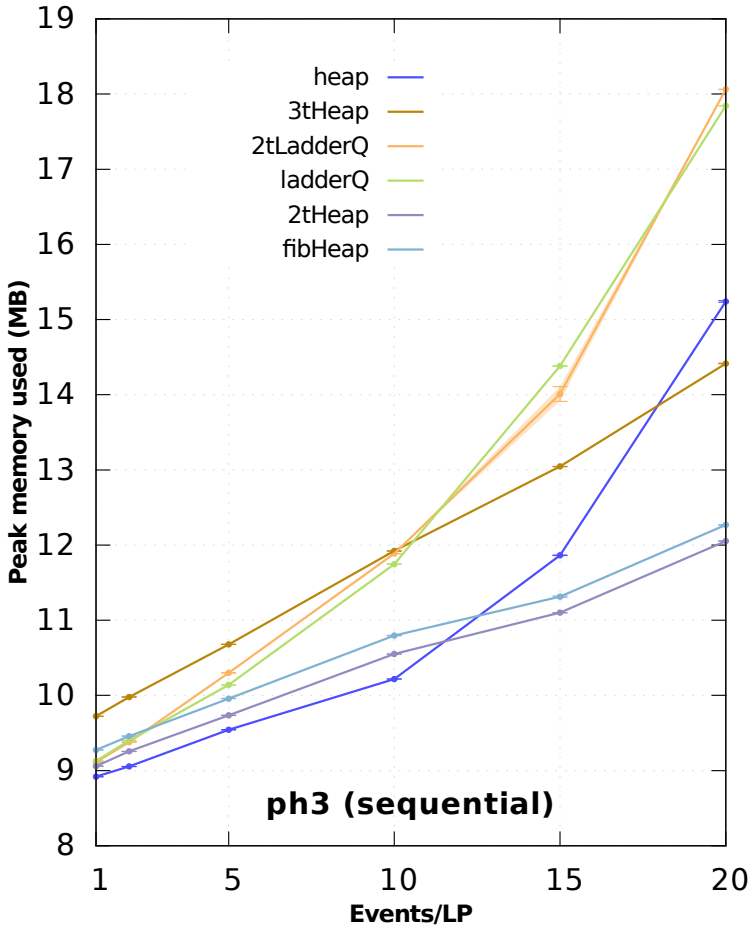
The sequential simulation experiments reported in the paper systematically explore the parameter space of the most influential parameters identified via GSA, namely:

1. `eventsPerLP` = {1, 2, 5, 10, 15, 20}
2. `%Self-Events` = {0.0, 0.25}
3. λ (`delay`) = {1, 2, 5, 10}

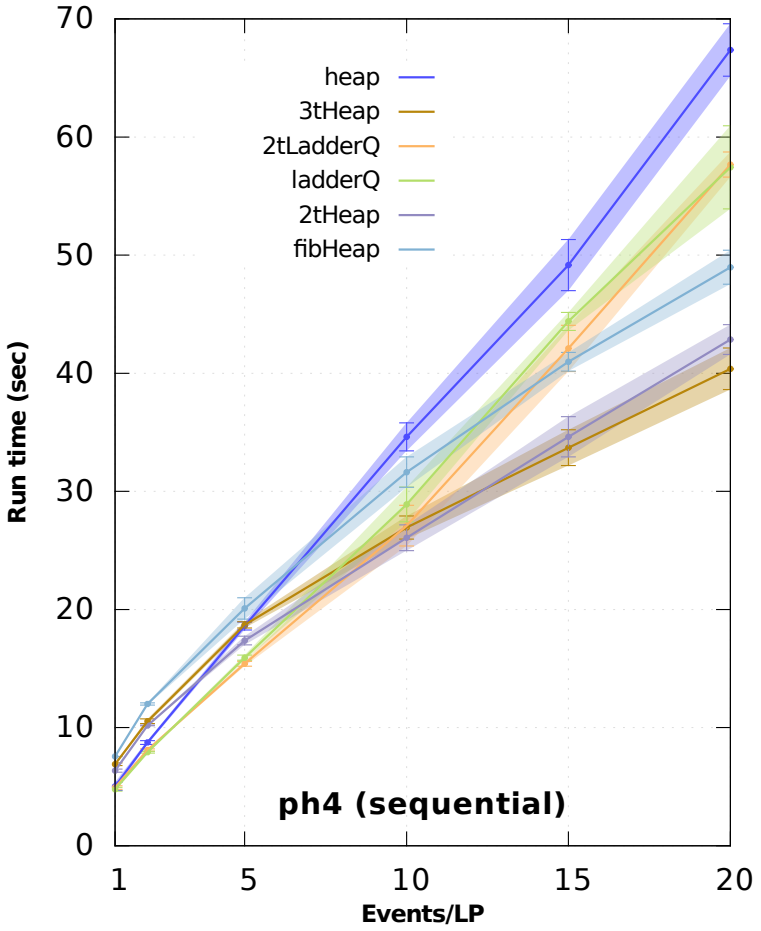
The combination of parameters resulted in 48 different combination of parameters. Ten simulation replications for each configuration (total of 480 simulations) have been used to report the simulation results in the paper. The paper included data for the 4 queues, namely: `heap`, `ladderQ`, `2tLadderQ`, and `3tHeap`. These queues performed best in at least one configuration explored. The charts below show the full data set for simulation execution times and the peak memory used for all 6 queues using the same configurations show in the paper. The data is average values from 10 independent simulation replications along with 95% Confidence Intervals (CI)

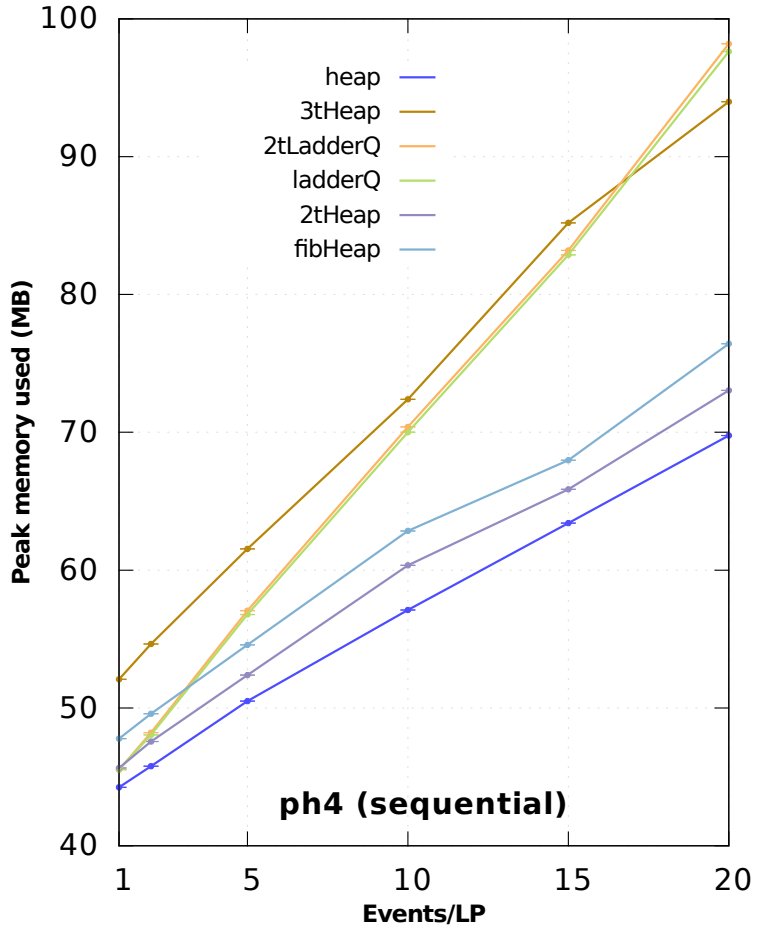
Sequential simulation runtime and peak memory use for the `ph3` PHOLD configuration with 1,000 (10^3) agents. The data is for λ=1, %Self-Events = 25%. The full set of data for all the other configurations will be in raw form as a table online.



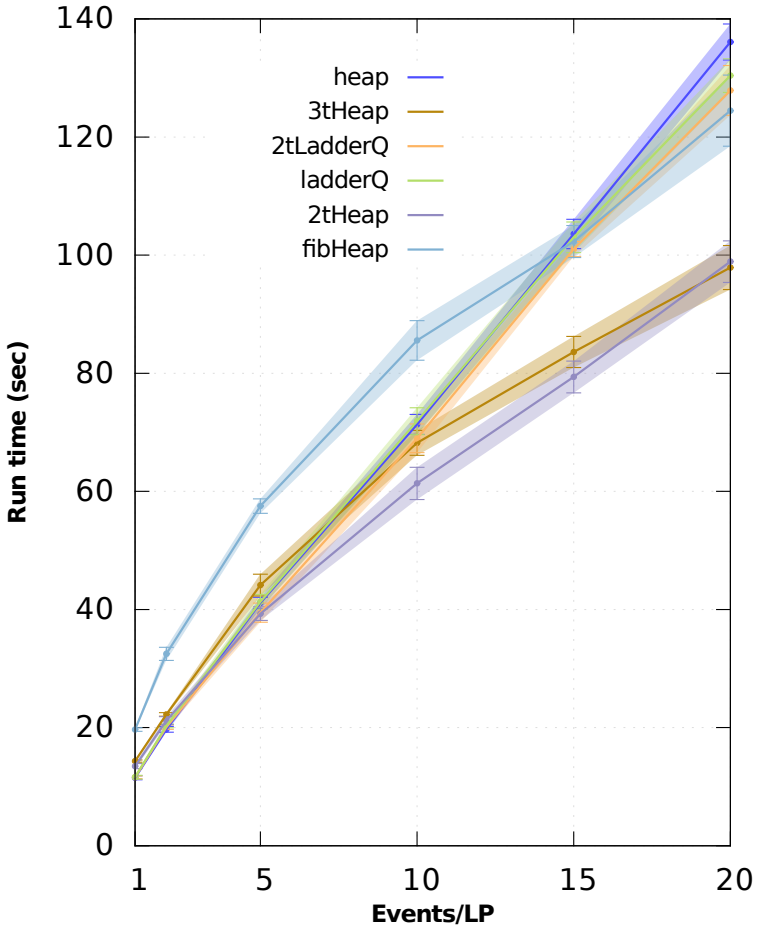


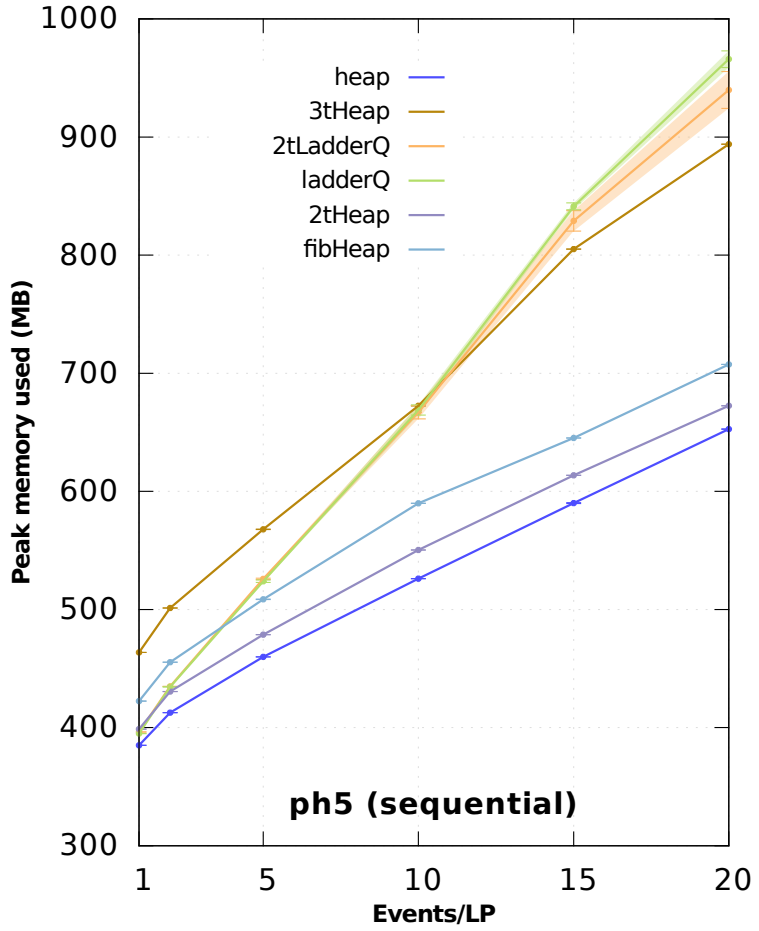
Sequential simulation runtime and peak memory use for the `ph4` PHOLD configuration with 10,000 (10^4) agents. The data is for λ=1, %Self-Events = 25%. The full set of data for all the other configurations will be in raw form as a table online.





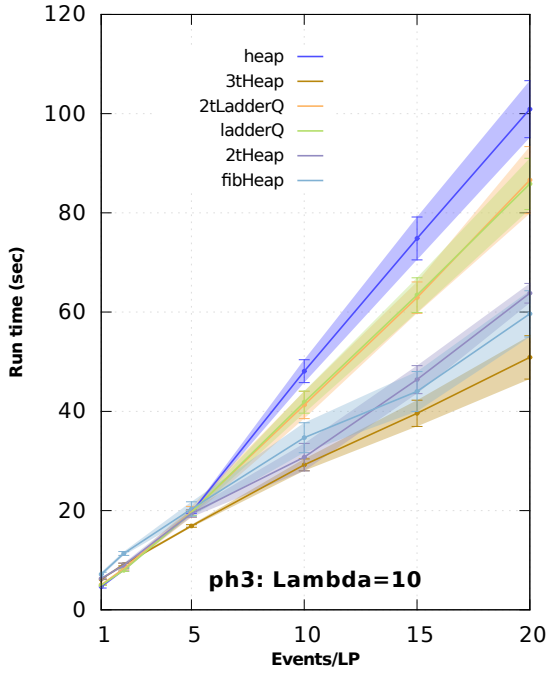
Sequential simulation runtime and peak memory use for the `ph5` PHOLD configuration with 100,000 (10^5) agents. The data is for λ=1, %Self-Events = 25%. The full set of data for all the other configurations will be in raw form as a table online.

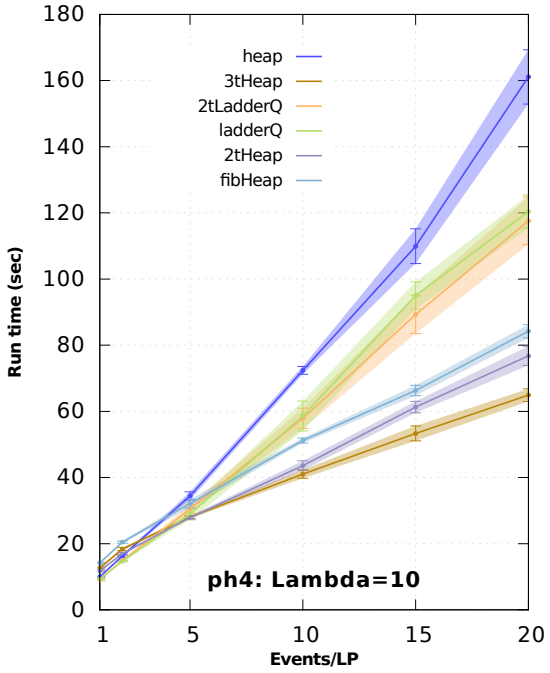


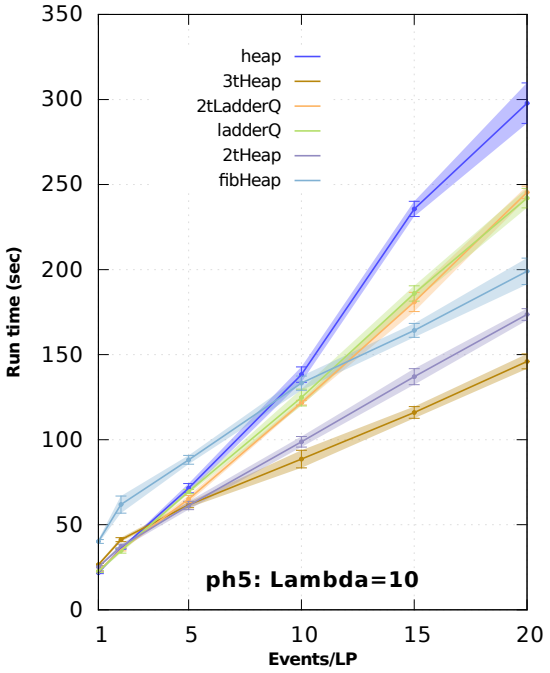


Sequential simulation: λ =10 (best-case for `3tHeap` in our data)

The simulation runtime for the 3 PHOLD configurations with λ =10 is shown below. Recollect that with λ =10, majority of the events are scheduled close to the current epoch and consequently the number of concurrent event with the same timestamp value increases per LP. This is generally the best-case timing for `3tHeap` in our experiments. In this situation the `3tHeap` consistently outperforms all the other queues.

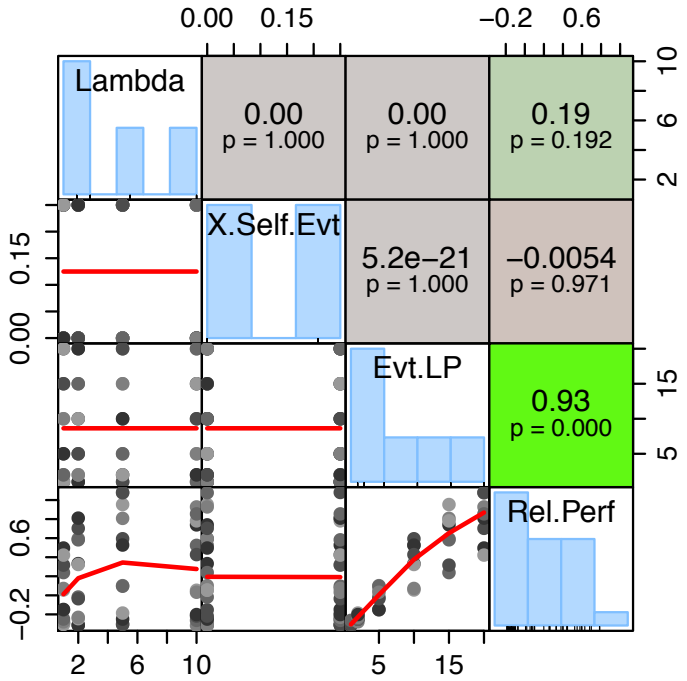






Cross-verification of GSA results between `ladderQ` and `3tHeap`

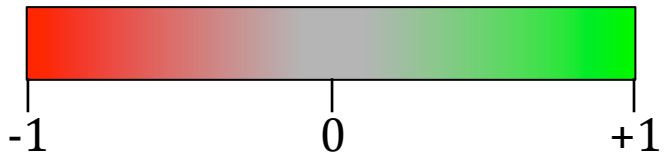
The simulation results from 10 replications of 48 different configurations provide an additional opportunity to cross-verify the GSA results by exploring correlations between the 3 parameters and the performance difference observed between `2tLadderQ` and `3tHeap`. For this we have converted the raw runtimes to percentage difference in runtime between `2tLadderQ` and `3tHeap`. We chose the `ph4` configuration, which was the mid-sized configuration. The corellogram was plotted using R and the `PerformanceAnalytics` package.



The diagonal shows the parameters being analyzed and the bottom-right corner corresponds to the relative performance between the queues.

The lower-triangle of the corellogram show the scatter plots between the pair of parameters on the diagonals. For example, the bottom-right most scatterplot corresponds to the data `Evt.LP` vs `Rel.Perf`. The red lines in the scatter plot show the LOESS smoothed fitted curve that eases observing potential trends in the data. Flat/horizontal lines correspond to low correlation while diagonal lines show high correlation.

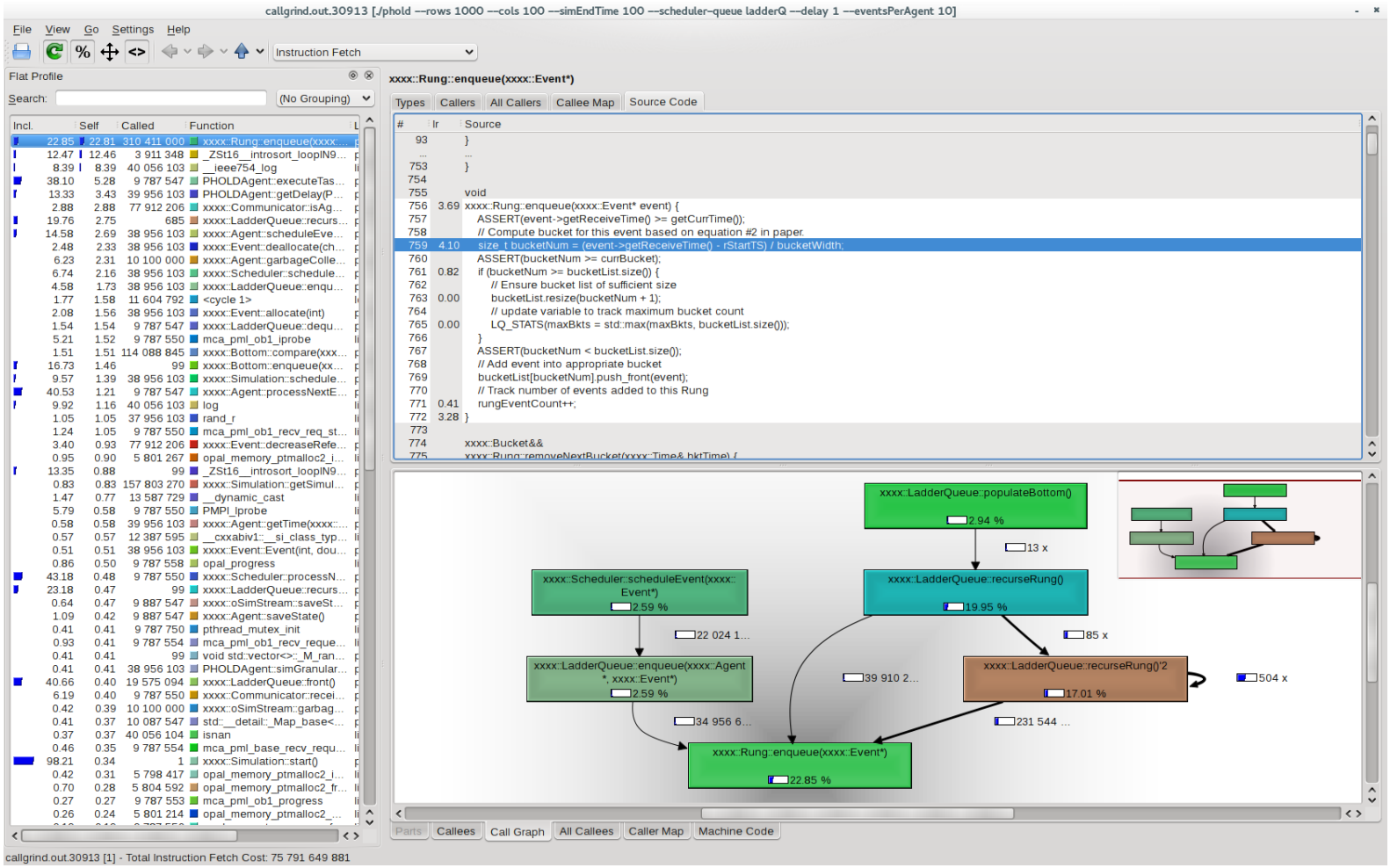
The upper-triangle of the corellogram show the Pearson correlation coefficient and p-value between each pair of parameters on the diagonal. The boxes are also color coded as shown in the scale below:



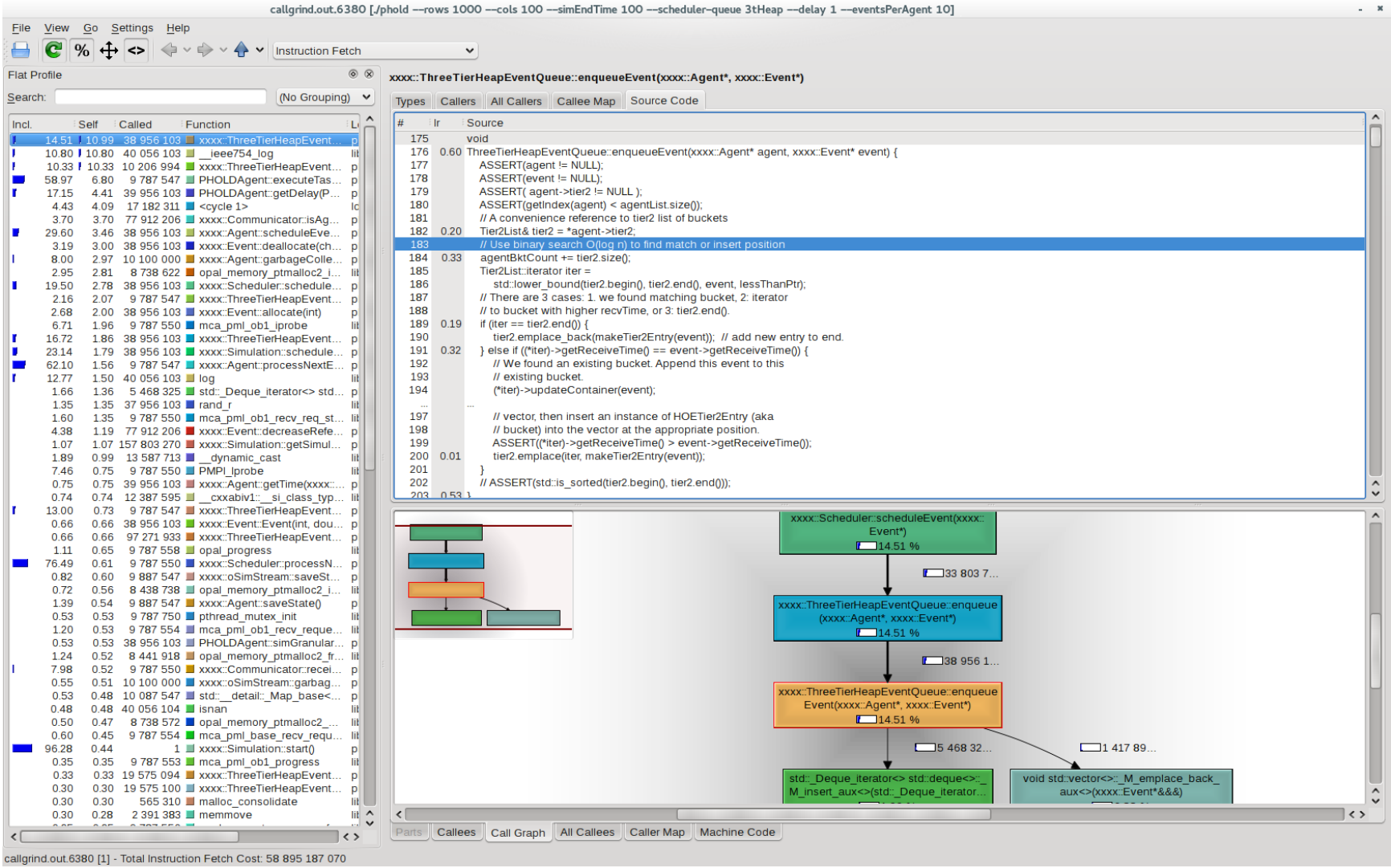
The corellogram verifies the results from GSA analysis by confirming that only Evt/LP has the strongest correlation with relative performance between `ladderQ` and `3tHeap`. Lambda (λ) also has a small influence on the relative performance, consistent with GSA results.

Profiler data for Ladder Queue and 3-tier Heap

The runtime profile was collected using `valgrind`'s `callgrind` tool on a sequential simulation using the `ph4` (medium size) model with λ=1, `eventsPerAgent`=10, and %Self Events=0.0. The profiler data shows that the overhead of re-bucketing events from rung-to-rung of the ladder takes time, even though each operation is very light weight coming in at less than 22 CPU cycles per call.



Profile data for Ladder Queue showing increasing overhead of rung insertions and/or rung-to-rung re-bucketing



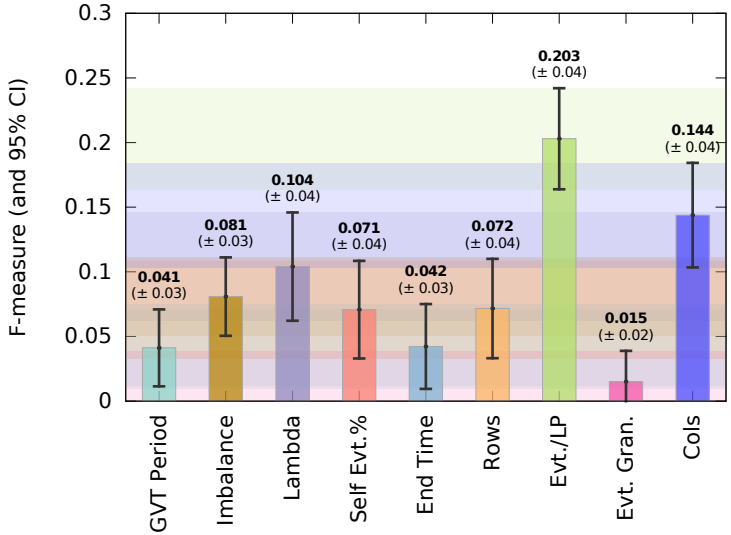
Profile data for 3tHeap for the same configuration of Ladder Queue. Most time is spent in inserting in 2nd/3rd tier

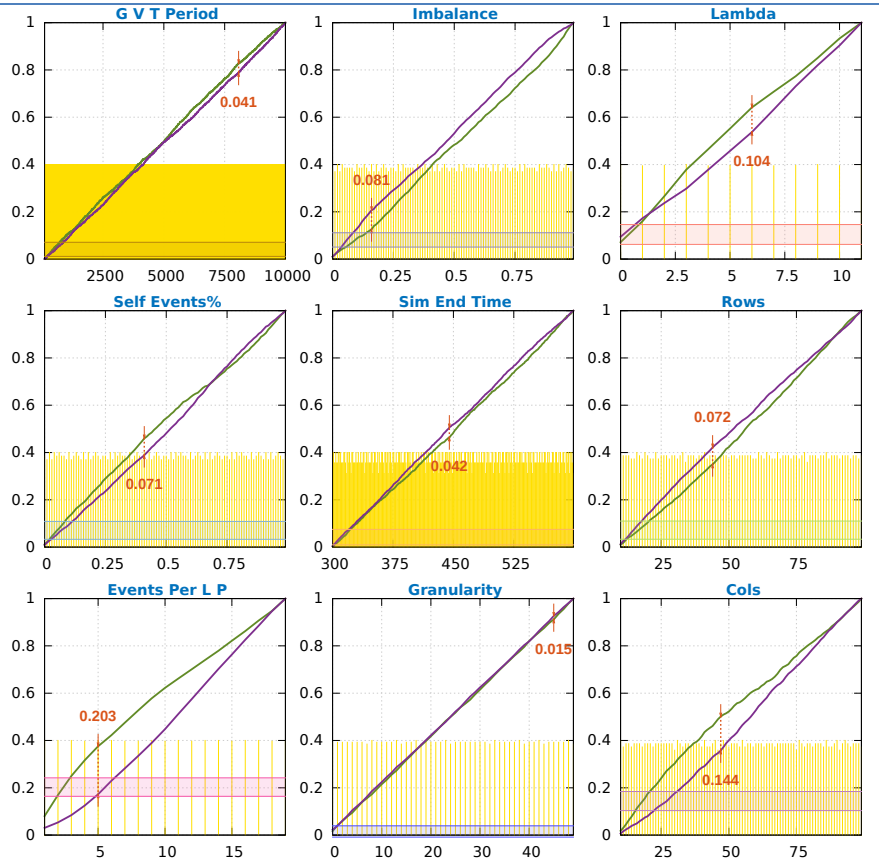
Generalized Sensitivity Analysis (GSA) for Parallel Simulations

This section includes results from the GSA that was conducted to assess the impact of various PHOLD parameters on relative performance of `ladderQ`, `2tLadderQ`, and `3tHeap` in parallel simulations. The parallel simulations were conducted using 4 MPI-processes and with a time-window of 10 simulation units. The time-window is consistently used in all parallel simulations to ensure that cascading rollbacks thereby providing runtimes with lower variance do not bog down the `ladderQ`. The runtimes for each GSA configuration are average of 3 runs. The data is from 2500 × 3 = 7500 simulations. The 2-tier Ladder Queue (2tLadderQ) is used as the reference for comparison in all cases

Ladder Queue (`ladderQ`) vs. 2-tier Ladder Queue (`2tLadderQ`)

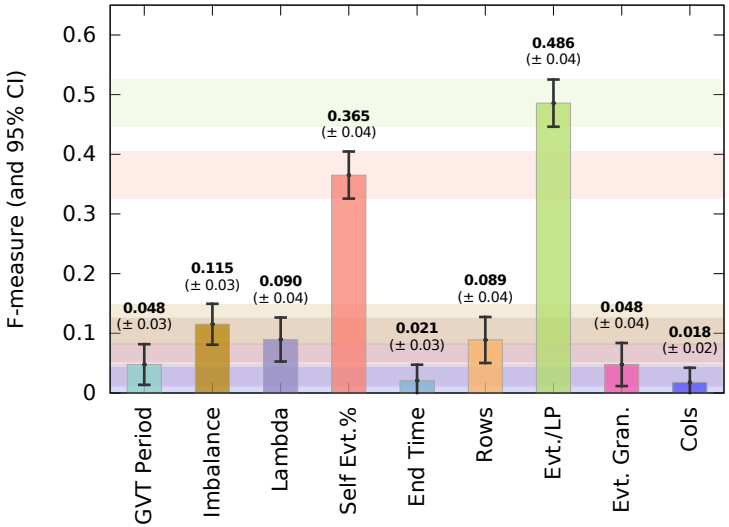
The GSA data shows that in parallel simulations, most of the parameters do not influence performance difference between `2tLadderQ` (with $_{2t}k$=128) and `ladderQ` with $d_{m,n} << 0.1$. Experiments show that `2tLadderQ` generally performs as well or better than `ladderQ`. However, higher values of `Evt./LP` (`eventsPerLP`) parameter cause the `2tLadderQ` to perform better.

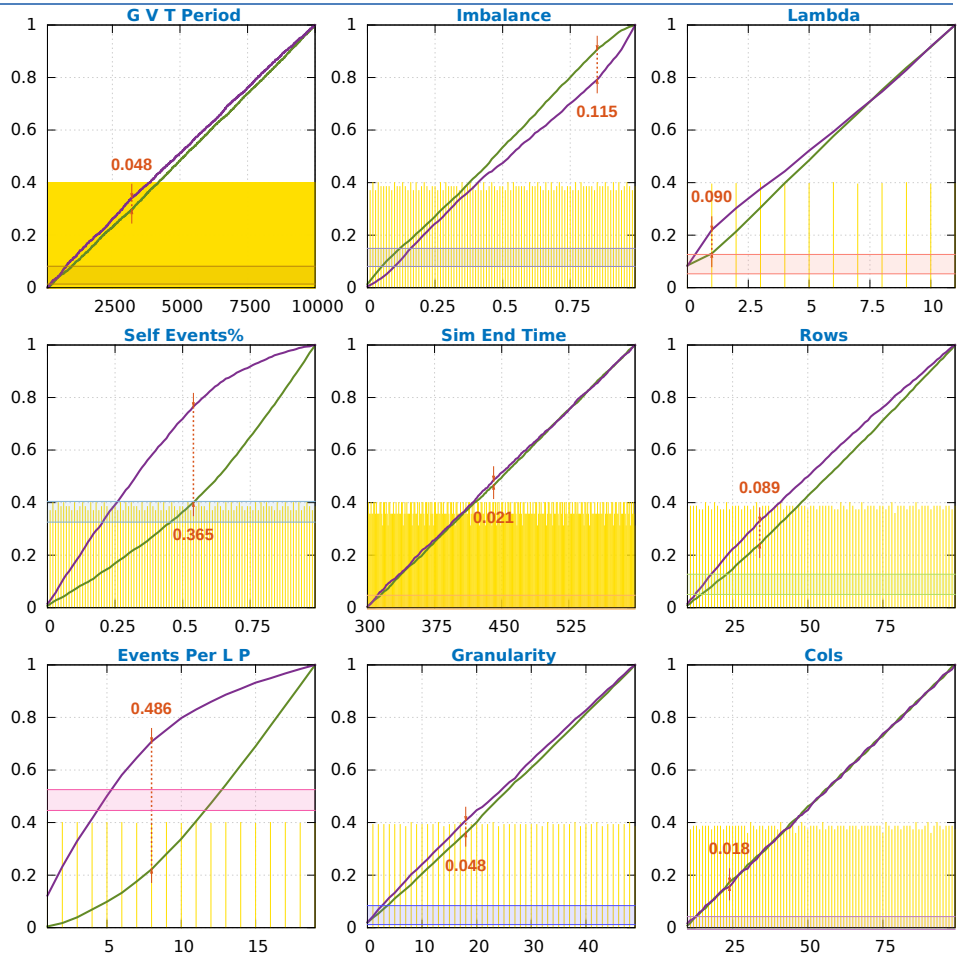




2-tier Ladder Queue (`2tLadderQ`) vs. 3-tier Heap (`3tHeap`)

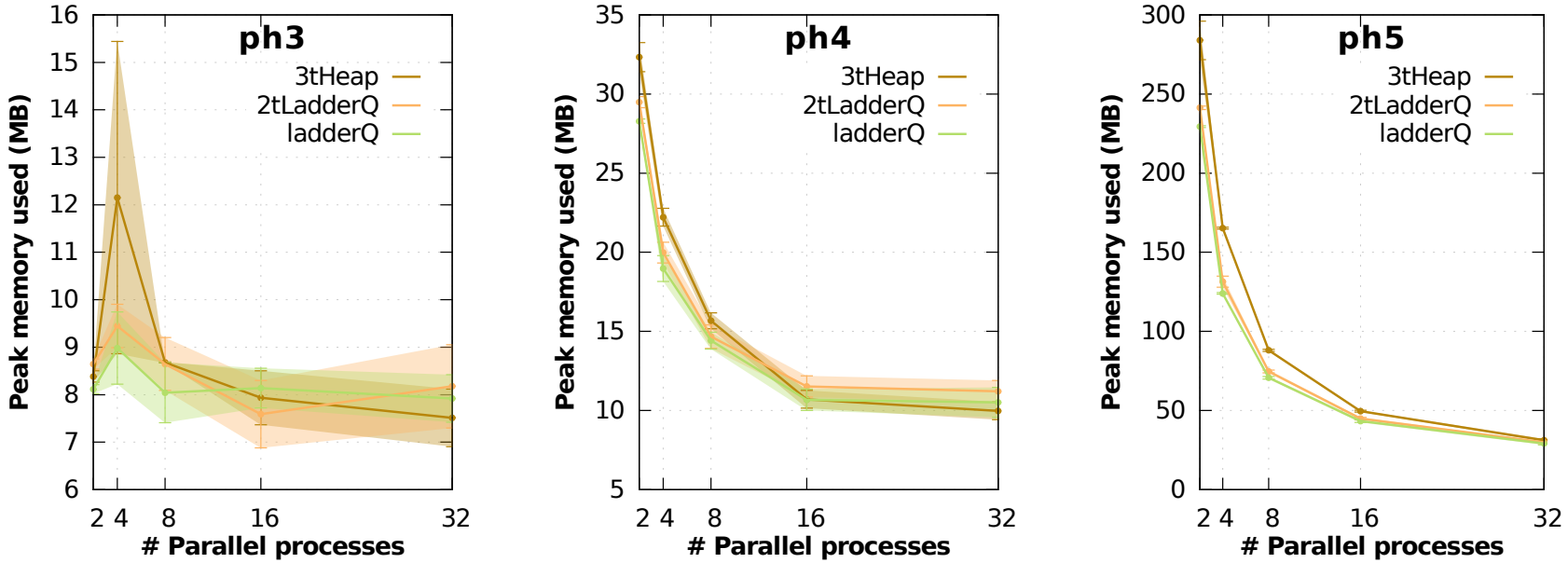
This is the same data shown in the PADS paper. The GSA data shows that in parallel simulations, most of the parameters do not influence performance difference between `2tLadderQ` (with $_{2t}k$=1) and `3tHeap` with $d_{m,n}$ << 0.1. Experiments show that `3tHeap` generally performs better for higher values of values of `Evt./Agent` (`eventsPerAgent`). The `%Self Events` parameter also has some influence but the overall influence of this parameter was not conspicuous in experimental data.



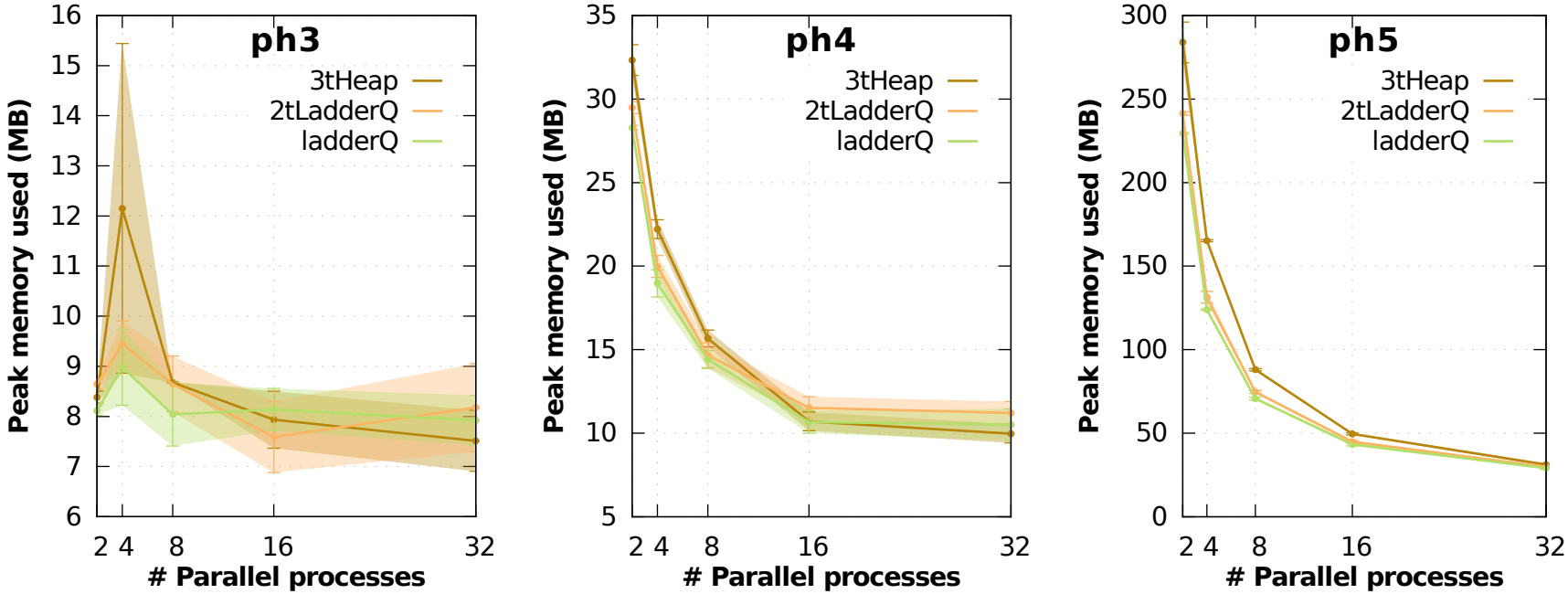


Additional parallel simulation results

The sequential simulation assessments clearly show that `ladderQ`, `2tLadderQ`, and `3tHeap` performed the best for broad range of PHOLD parameter settings. Consequently, we focused on assessing the effectiveness of these 3 queues for Time Warp synchronized parallel simulations. The experiments were conducted on our compute cluster 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. The main paper showed simulation time and rollback data for two key configurations for `ph3`, `ph4`, and `ph5`. The memory usage for the first two configuration in the paper are shown below:



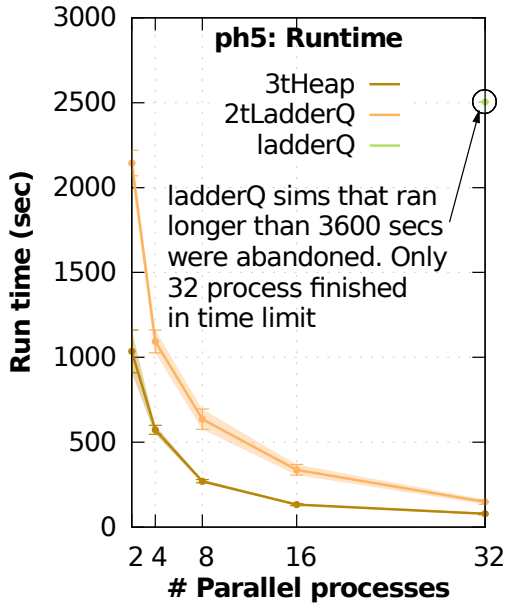
The peak memory (value reported for "`Maximum resident set size`" by `/usr/bin/time`) usage for the 3 different PHOLD configuration with `eventsPerLP`=2, λ= 1, and `%Self-Events`=0.25. This data corresponds to the best-case configuration for the `ladderQ`. This is the best case for `ladderQ` because λ= 1 produces the widest range of timestamps thereby significantly reducing the effective number of `eventsPerLP` at any given time. Consequently, buckets are smallest in this configuration.

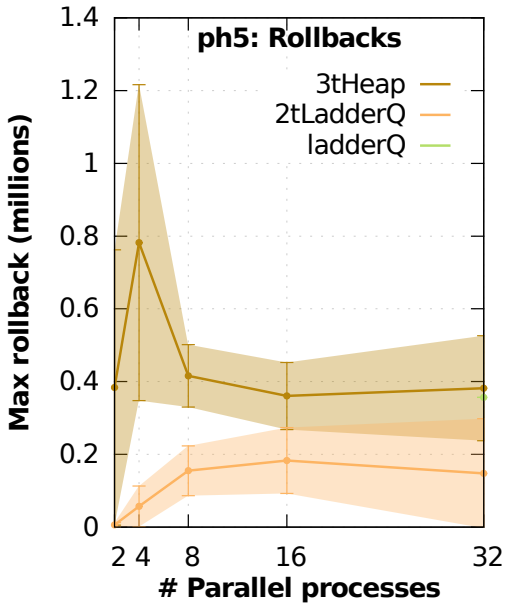


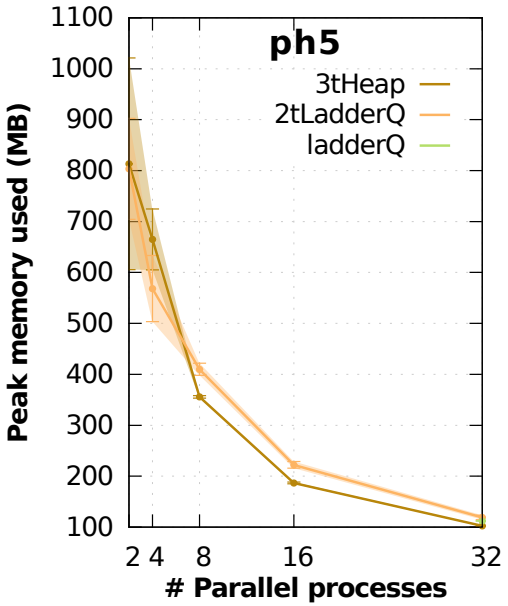
The peak memory (value reported for "`Maximum resident set size`" by `/usr/bin/time`) usage for the 3 different PHOLD configuration with `eventsPerLP`=10, λ= 10, and `%Self-Events`=0.25. This data corresponds to the knee-case where the performance of the ladder queues and `3tHeap` were similar in sequential simulations. This is the scenario in which the 3tHeap just starts to outperform the `ladderQ`. Note that this is not the best case for `3tHeap` which generally performs best for `eventsPerLP` > 10.

PH5: Best case for `3tHeap` (Evts./LP=20, λ=10, %SelfEvents=0.25)

The best case for `3tHeap` arises with high concurrency where a larger batch of events is scheduled per LP in each simulation cycle. In this configuration the `ladderQ` simulations did not complete in 3600 seconds, which was the time limit. This time limit is about 3× -- 30× more than the runtime of `3tHeap` in this configuration. The `ladderQ` simulation for 32 processes successfully completed and that is the only data point plotted in the charts below. This is the worst-case scenario for the ladder queue structures. However, the `2tLadderQ` performs well. The `3tHeap` considerably outperforms the other queues despite having a higher number of rollbacks.

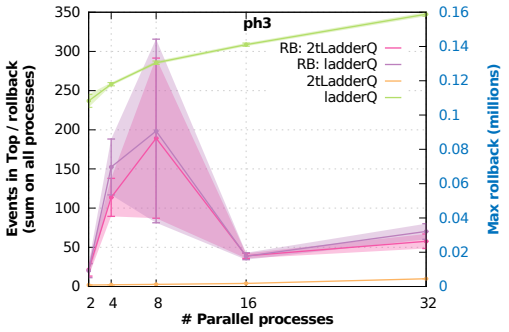


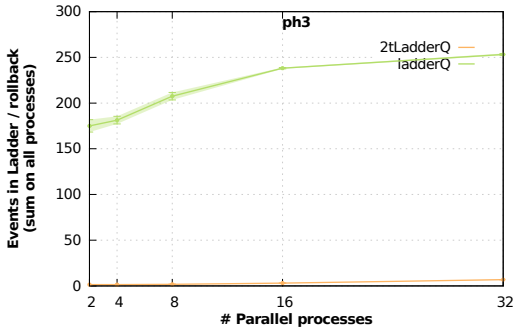


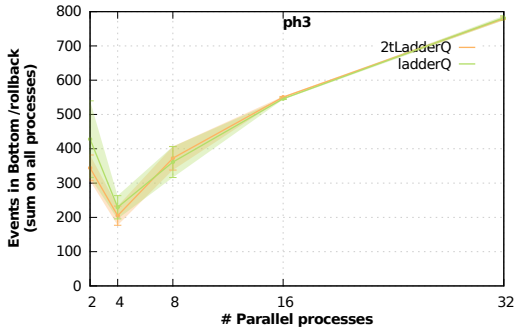


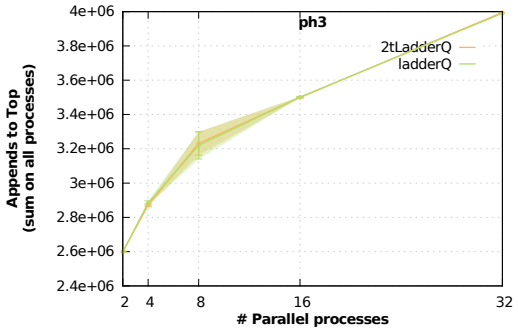
PH3: 2-tier ladder queue operations (Evts./LP=2, λ=1, %SelfEvents=0.25)

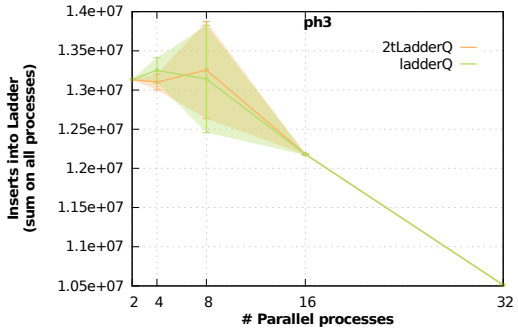
The charts show comparison of key operations between `ladderQ` and `2tLadderQ`. Most of the data between the 2 queues are similar except for the number of events scanned per rollback in Top, Ladder rungs, and bottom. The difference arises from breaking the bucket into 128 sub-buckets. This essentially reduces the number scans by 1/128th as evidenced in the first 3 charts. In the case of `ph3` the total number of Bottom→Ladder operations were high but the number of events being moved were very few and that did not significantly degrade their performance. Furthermore as shown by the rollbacks in the first figure, these operations do not show strong correlation with rollbacks but with partitioning differences.

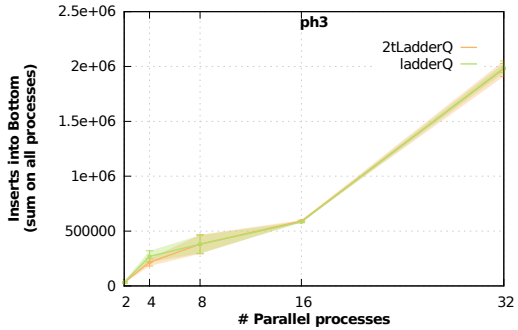


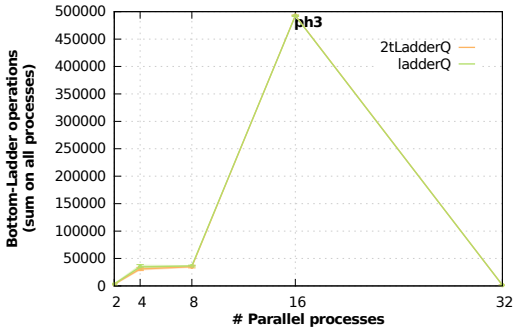






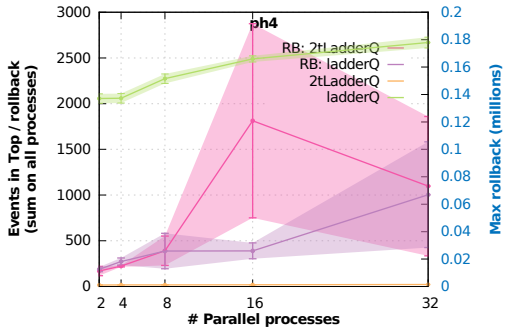


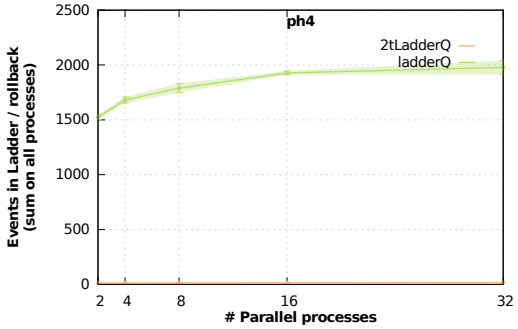


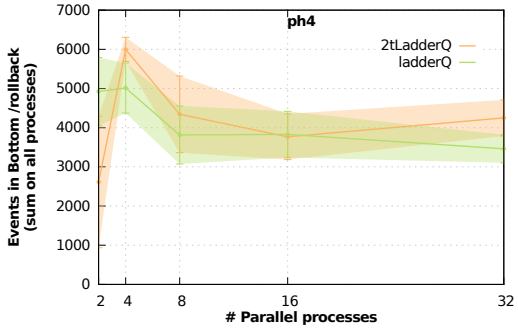


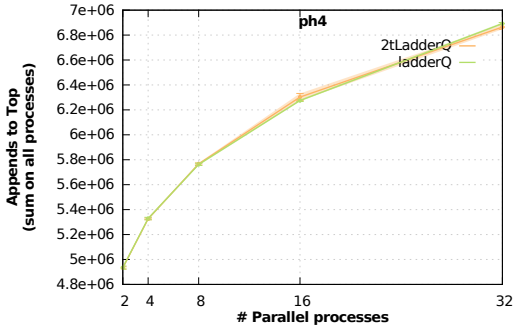
PH4: 2-tier ladder queue operations (Evts./LP=2, λ=1, %SelfEvents=0.25)

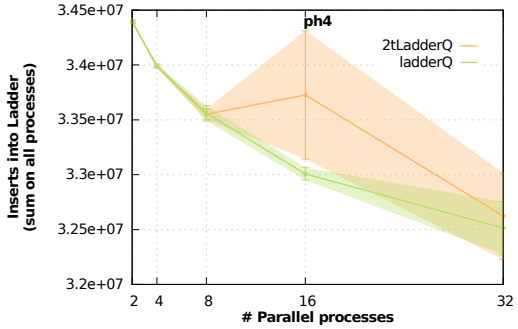
The charts show comparison of key operations between `ladderQ` and `2tLadderQ`. Most of the data between the 2 queues are similar except for the number of events scanned per rollback in Top, Ladder rungs, and bottom. The difference arises from breaking the bucket into 128 sub-buckets. This essentially reduces the number scans by 1/128th as evidenced in the first 3 charts where the curve for `2tLadderQ` is close to the x-axis. The Bottom→Ladder operations are much lower in this configuration but increase as the events get spread out between the processes. However, the number of events decrease and do not hurt performance.

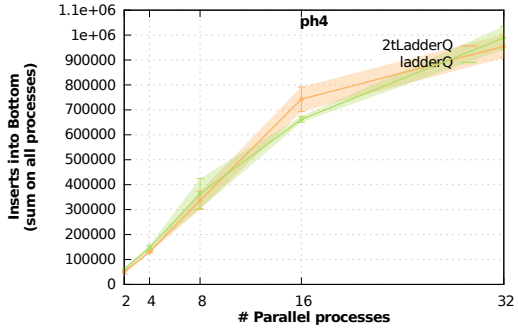


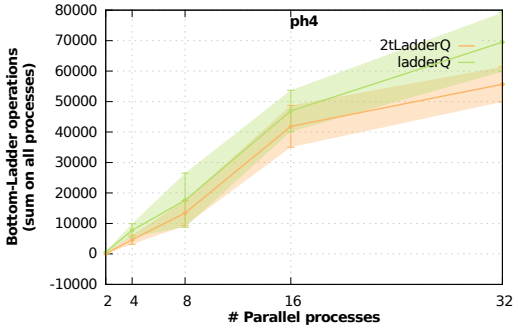






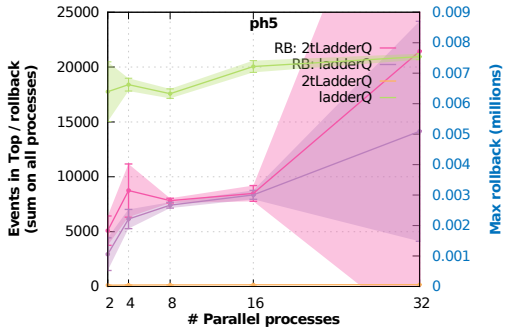


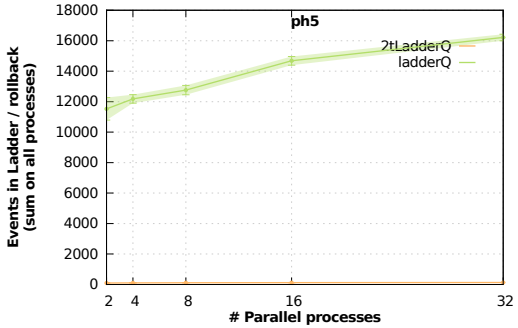


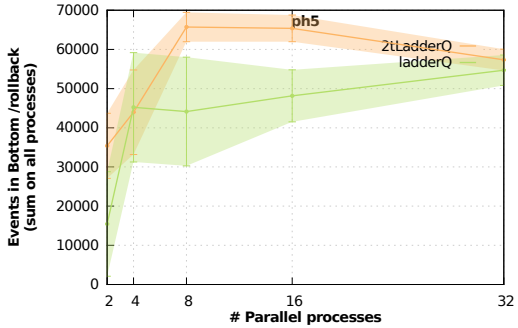


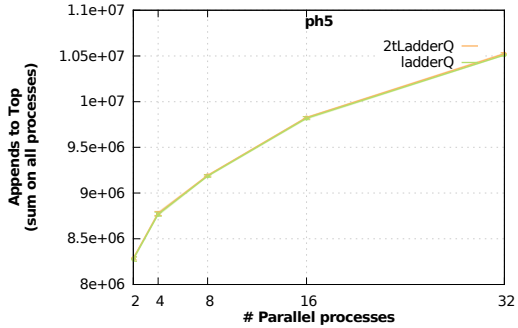
PH5: 2-tier ladder queue operations (Evts./LP=2, λ=1, %SelfEvents=0.25)

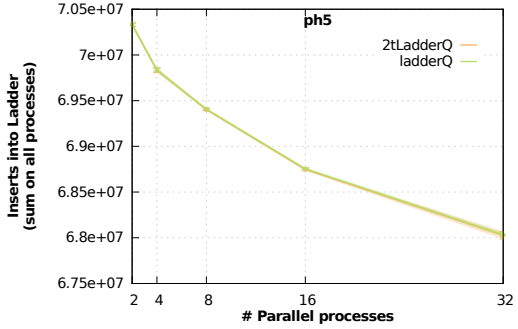
The charts show comparison of key operations between `ladderQ` and `2tLadderQ`. Most of the data between the 2 queues are similar except for the number of events scanned per rollback in Top, Ladder rungs, and bottom. The difference arises from breaking the bucket into 128 sub-buckets. The number Bottom→Ladder operations were considerably lower in this case but the bottom lengths are typically much longer due to the larger number of LPs. This is the best configuration for both ladder queues.

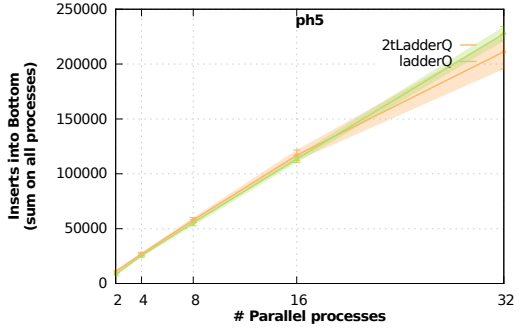


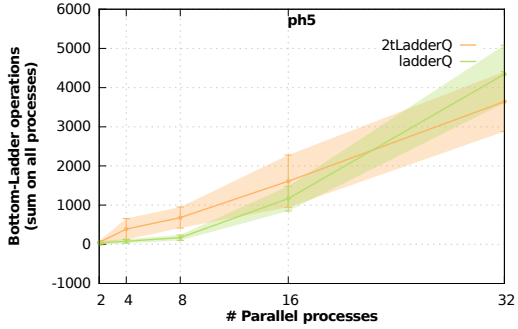






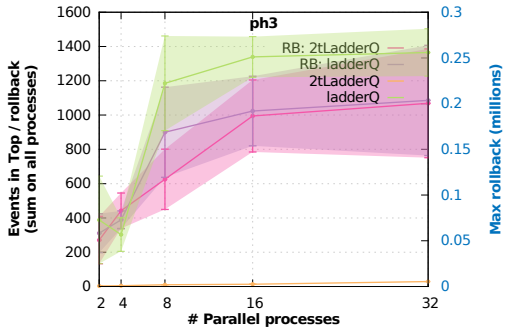


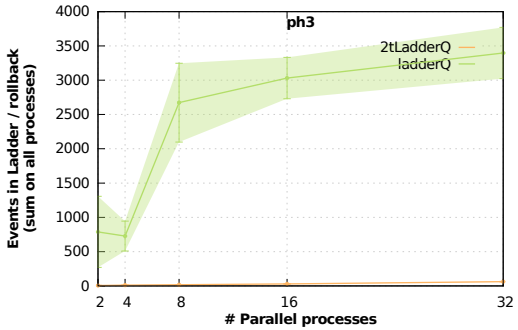


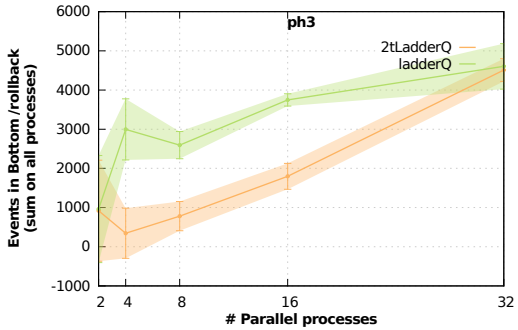


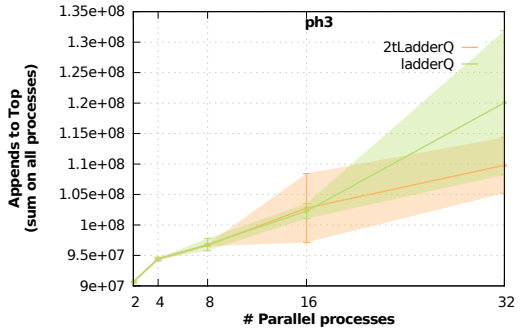
PH3: 2-tier ladder queue operations (Evts./LP=10, λ=10, %SelfEvents=0.25)

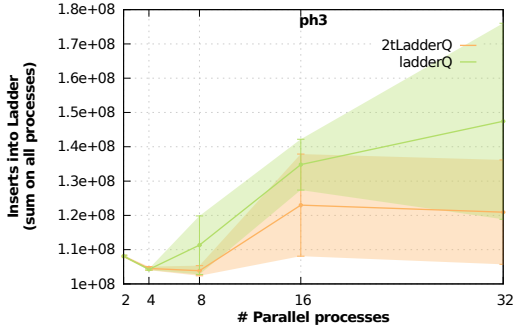
The charts show comparison of key operations between `ladderQ` and `2tLadderQ`. Most of the data between the 2 queues are similar except for the number of events scanned per rollback in Top, Ladder rungs, and bottom. The difference arises from breaking the bucket into 128 sub-buckets. This essentially reduces the number scans by 1/128th as evidenced in the first 3 charts. In the case of `ph3` the total number of Bottom→Ladder operations were high but the number of events being moved were very few and that did not significantly degrade their performance. Furthermore as shown by the rollbacks in the first figure, these operations do not show strong correlation with rollbacks but with partitioning differences.

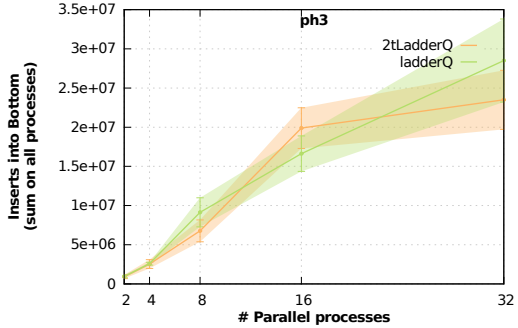


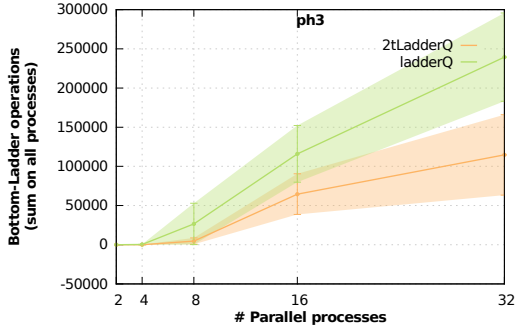






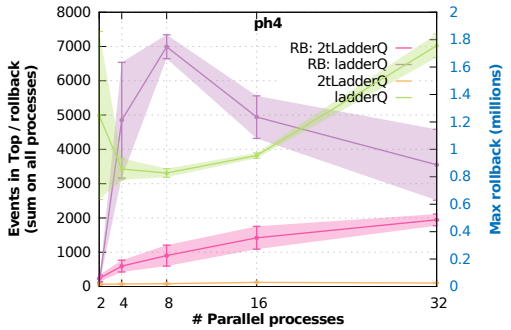


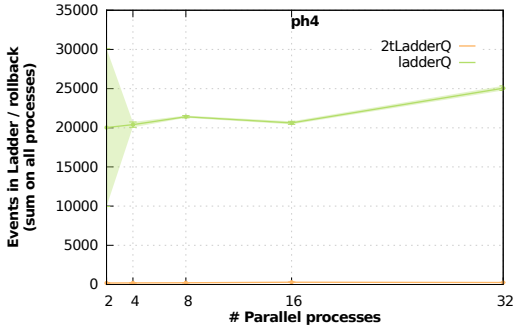


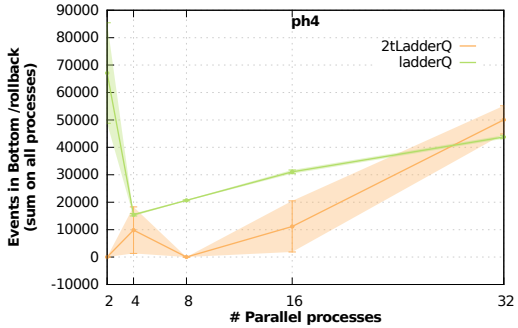


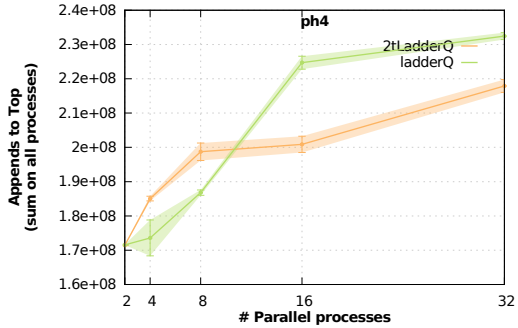
PH4: 2-tier ladder queue operations (Evts./LP=10, λ=10, %SelfEvents=0.25)

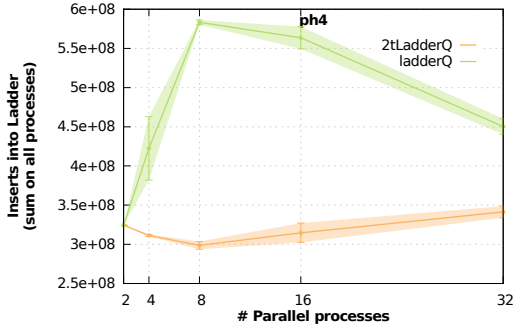
The charts show comparison of key operations between `ladderQ` and `2tLadderQ`. Unlike the earlier configurations that were the efficient configuration for `ladderQ`. However, the ratio of number of events scanned per rollback in Top, Ladder rungs, and bottom remain consistently 1/128th. However, the number of Bottom→Ladder operations for `ladderQ` were much higher, possibly due to higher rollbacks for `ladderQ`.

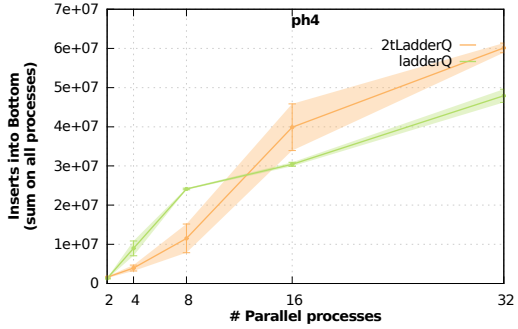


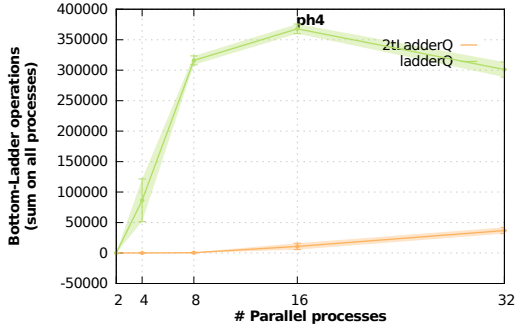






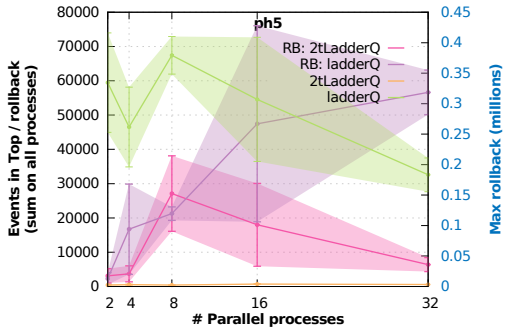


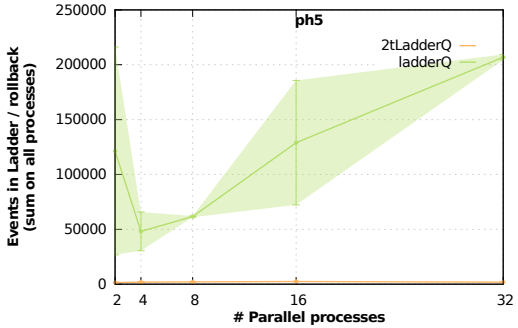


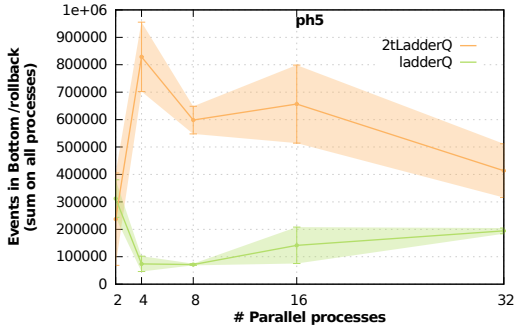


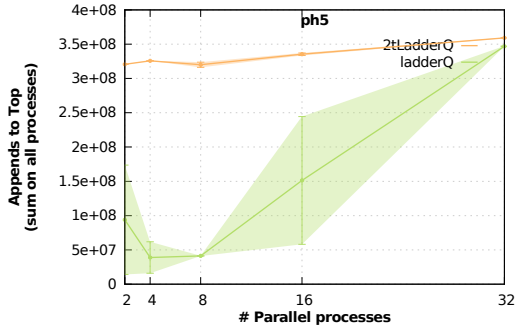
PH5: 2-tier ladder queue operations (Evts./LP=10, λ=10, %SelfEvents=0.25)

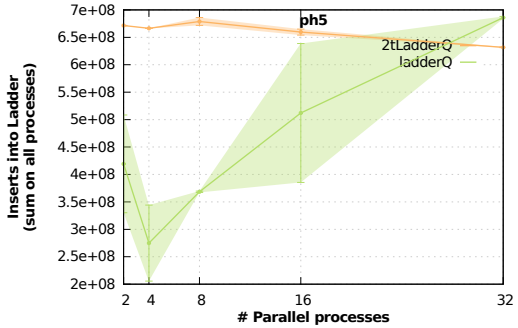
The charts show comparison of key operations between `ladderQ` and `2tLadderQ`. Unlike the earlier configurations that were the efficient configuration for `ladderQ`. However, the ratio of number of events scanned per rollback in Top, Ladder rungs, and bottom remain consistently 1/128th. However, the number of Bottom→Ladder operations for `ladderQ` were much higher, possibly due to higher rollbacks for `ladderQ`. The increased rollbacks also occur because of the increased overhead of canceling events. This causes processes to optimistically simulate ahead only to get rolledback and requiring Bottom→Ladder operations as events are scheduled for reprocessing.

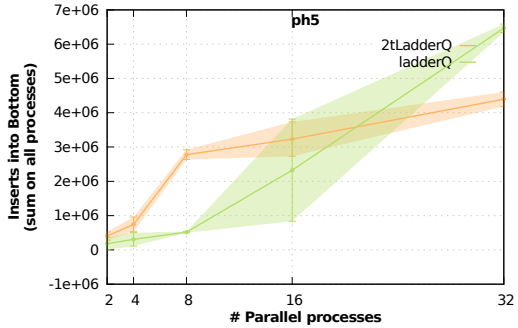


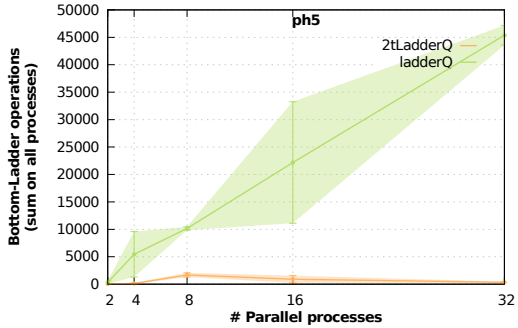






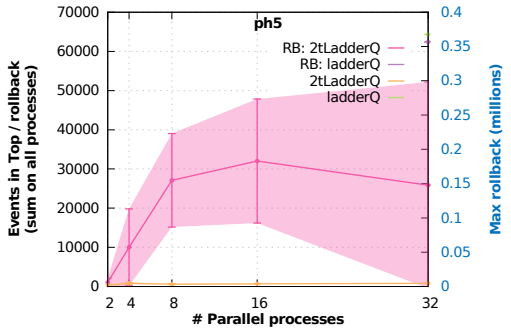


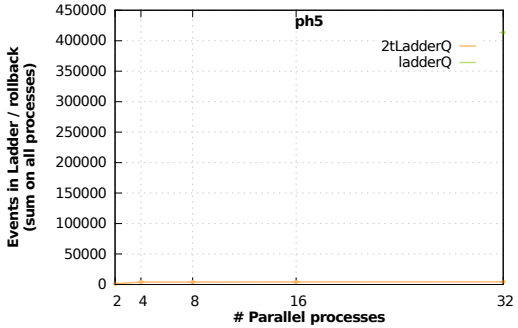


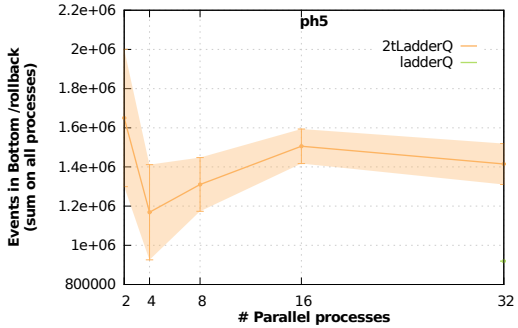


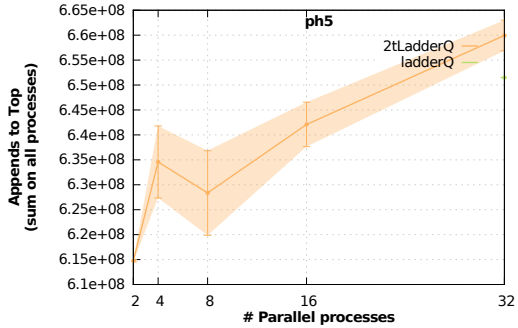
PH5: Ladder queue statistics (Best case for 3tHeap, Evts./LP=20, λ=10, %SelfEvents=0.25)

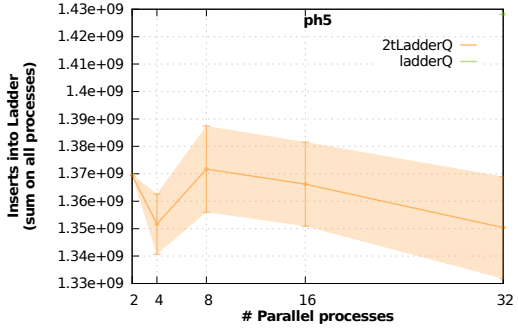
The charts show the key operations occurring in the `2tLadderQ`. The statistics suggest that the 2tLadderQ is operating in optimal modes with very few Bottom→Ladder operations. However, the length of bottom is long as indicated by the number of events checked per rollback. However, the number of events scanned in top and in the rungs is rather few about 1/128th of the events in these structures. The data for `ladderQ` is not seen because the simulations did not complete in the 3600 seconds allotted for the jobs.

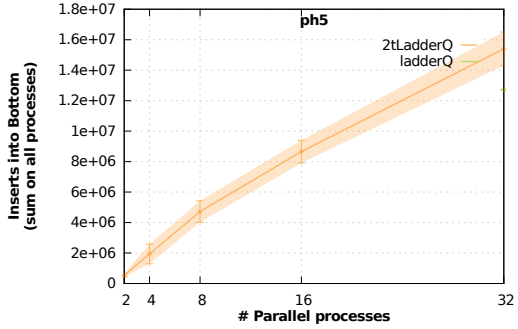


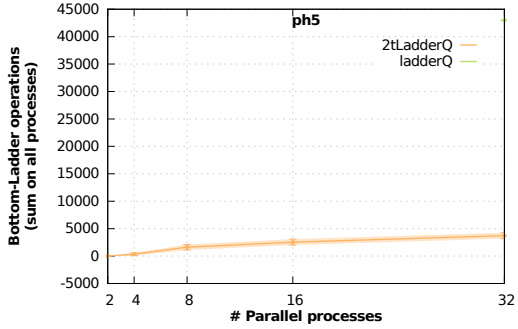












Source code Listing for queues

The source code listing for the six different queues are included in the subsequent pages. The source code is included to provide as much information as possible for review as part of the double-blind review process. The final version of the paper will include links to the full source repository. The scientific community will be able to use the online source to verify/falsify the results presented in this paper and supplements.

Source file(s)	Description	Next pages
EventQueue.h	Interface for all event queues	1--3
HeapEventQueue.h, .cpp	Source for `heap`	4--7
TwoTierHeapEventQueue.h, .cpp	Source for `2tHeap`	8--12
FibonacciHeapEventQueue.h, .cpp	Source code for `fibHeap`	13--18
ThreTierHeapEventQueue.h, .cpp	Source code for `3tHeap`	19--25
LadderQueue.h, .cpp	Source code for `ladderQ`	26--47
TwoTierLadderQueue.h, .cpp	Source code for `2tLadderQ`	48--62
PHOLDSimulation.h, .cpp	Top level driver for PHOLD benchmark	63--66
PHOLDAgent.h, .cpp	Agent/LP source code for PHOLD benchmark	67--70
PholdState.h, .cpp	State for `PHOLDAgent` (LP)	71--72

```
#ifndef EVENT_QUEUE_H
#define EVENT_QUEUE_H

#include "DataTypes.h"
#include "Agent.h"

BEGIN_NAMESPACE(xxxx)

/** Base class for the different queues used by the different
    Schedulers.

    <p>The primary objective of an EventQueue is to act as a fast
    priority queue that provides access to events to be scheduled in a
    Time Warp simulation.  The priority (or sorting order) of events
    is based both on the receive-time of the event and the agent to
    which the event is scheduled.  The objective is to obtain all
    events scheduled to a given agent at a given receive-time in one
    batch/set.</p>

    <p>The EventQueue class serves as an abstract base class that
    defines the interface available to the Scheduler class hierarchy.
    The EventQueue class cannot be directly instantiated.  Instead one
    of the derived classes must be instantiated and used.</p>
*/
class EventQueue {
public:
    /** Add/register an agent with the event queue.

        This method must be invoked to add/register each agent that is
        present on the local Simulator/process (.i.e, do not register
        all agents that are present on other compute nodes/processes).
        This method must be called prior to commencement of scheduling
        events via this event queue.

        \param[in,out] agent A pointer to the agent to be registered.
        The pointer is internally used by derived classes to manage
        corss references etc.  Consequently, the object pointed must
        not be deleted until the event queue operations are completed.

        \return This method returns an internal cross-reference to be
        stored in an agent.  This cross-reference is used by derived
        classes to optimize event management operations.
    */
    virtual void* addAgent(xxxx::Agent* agent) = 0;

    /** Remove/unregister an agent from the event queue.

        This method is invoked just before an agent is removed from
        the simulation.  Typically this method is invoked to inform
        the event queue that the data structures associated with the
        agent can be removed.

        \param[in,out] agent A pointer to the agent to be removed.
        The pointer is internally used by derived classes to manage
        cross references etc.  Consequently, the object pointed must
        not be deleted by this method.
    */
    virtual void removeAgent(xxxx::Agent* agent) = 0;

    /** Update the specified Agent's information.

        This method is invoked after events associated with an agent
        have been processed.  This method provides the event queue
        with an opportunity to appropriately relocate the agent in
        this internal data structures.

        \param[in] crfPointer Pointer to the event queue's interal
        cross-reference that was returned by addAgent() when the agent
        was added to this event queue.

        \param[in] uTime The time of the agents's events that where
        just processed.
    */
    // virtual void update(void* crfPointer, const Time uTime) = 0;

```



```
    /** Determine if the event queue is empty.

        This method must be implemented by the derived classes to
        report if any events are pending to be processed in the event
        queue.

        \return This method returns true if the event queue is
        logically empty.
    */
    virtual bool empty() = 0;

    /** Obtain pointer to the highest priority (lowest receive time)
        event.

        This method can be used to obtain a pointer to the highest
        priority event in this event queue, without de-queuing the
        event.

        \note The event returned by this method is not dequeued.

        \return A pointer to the next event to be processed.  If the
        queue is empty then this method returns NULL.
    */
    virtual xxxx::Event* front() = 0;

    /** Method to obtain the next batch of events to be processed by
        one agent.

        The agents are scheduled to process all events at a given
        simulation time.  The next concurrent events (i.e., events
        with the same receive time) with the lowest time stamp are to
        be placed in the supplied event container.  The event
        container is then passed to the corresponding agent for
        further processing.

        \param[out] events The event container in which the next set
        of concurrent events are to be placed.  Note that the order of
        concurrent events in the event container is unspecified.
    */
    virtual void dequeueNextAgentEvents(xxxx::EventContainer& events) = 0;

    /** Enqueue a new event.

        This method must be used to enqueue/add an event to this event
        queue.  Once added the reference count on the event is
        increased.

        \param[in] agent The agent to which the event is to be
        scheduled.  This agent corresponds to the agent ID returned by
        event->getReceiverAgentID() method.

        \param[in] event The event to be enqueued.  This parameter can
        never be NULL.
    */
    virtual void enqueue(xxxx::Agent* agent, xxxx::Event* event) = 0;

    /** Enqueue a batch of events.

        This method can be used to enqueue/add a batch of events to
        this event queue.  Note that the event reference count is not
        changed by this method.  This method provides a convenient
        approach to enqueue a batch of events, particularly after a
        rollback.

        \param[in] agent The agent to which the event is to be
        scheduled.  This agent corresponds to the agent ID returned by
        event->getReceiverAgentID() method.

        \param[in] event The list of events to be enqueued.  The
        reference counts of the events in the container remains
        unmodified.  The list of events become part of the event
        queue.
    */
    virtual void enqueue(xxxx::Agent* agent, xxxx::EventContainer& events) = 0;

```

```
151     /** Dequeue all events sent by an agent after a given time.
152
153         This method can be used to remove/erase events sent by a given
154         agent after a given simulation time.  This API is needed to
155         cancel events during a rollback.
156
157         \param[in] dest The agent whose currently secheduled events
158         are to be checked and cleaned-up.  This agent must be a valid
159         agent that has been registered/added to this event queue.  The
160         pointer cannot be NULL.
161
162         \param[in] sender The ID of the agent whose events have to be
163         removed.  This agent must be a valid agent that has been
164         registered/added to this event queue.  The pointer cannot be
165         NULL.
166
167         \param[in] sentTime The time from which the events are to be
168         removed.  All events (including those sent at this time) sent
169         by the sender agent are removed from this event queue.
170
171         \return This method returns the number of events actually
172         removed.
173     */
174     virtual int eraseAfter(xxxx::Agent* dest, const xxxx::AgentID sender,
175                            const xxxx::Time sentTime) = 0;
176
177     /** Print full contents of scheduler queue to given output stream.
178
179         This is a convenience method that is used primarily for
180         troubleshooting purposes.  This method is expected to print
181         the complete set of events in the event queue to the given
182         output stream.  The format of the output is dependent of the
183         final derived class that is overridin this event.
184
185         \param[out] os The output stream to which the contents of the
186         queue are to be written.
187     */
188     virtual void prettyPrint(std::ostream& os) const = 0;
189
190     /** Method to report aggregate statistics.
191
192         This method is invoked at the end of simulation after all
193         agents on this rank have been finalized.  This method can
194         report any aggregate statistics from the event queue. The
195         statistics must be written to the supplied output stream.
196
197         \param[out] os The output stream to which the statistics are
198         to be written.
199     */
200     virtual void reportStats(std::ostream& os) = 0;
201
202     /** The virtual destructor.
203
204         The destructor does not have any specific task as the base
205         class does not manage or perform any special operations.
206     */
207     virtual ~EventQueue() {}
208
209     /** Obtain the human-readable identifier/name.
210
211         This string is an identifier that can be used to identify the
212         final derived class.  It is handy when reporting errors or
213         issues.
214
215         \return This method returns the identifier set by the derived
216         class during instantiation.
217      */
218     const std::string& getName() const { return name; }
219
220     /** Fixup the heap after a value at a given index position has
221         been updated.
222
223         This method can be used to fix up the heap after the value at
224         the given index position has been updated.  This method moves
225         the value at the given index position up or down the heap to
```



```
226         restore the heap properly.
227
228         \note The time complexity of this method is O(log n).
229
230         \param[in] vec The vector whose entry is to be updated.  This
231         vector is the backing vector in which all values, except the
232         one at index position, satisfy the heap property.
233
234         \param[in] index The index position in the backing vector
235         whose value is to be appropriately placed to restore the heap
236         property.
237
238         \param[in] comp A binary predicate comparator to be used to
239         compare elements in the heap.
240
241         \return This method returns the index position in the vector
242         where the element was finally placed in the heap.
243     */
244     template<typename Val, typename Compare>
245     static size_t fixHeap(std::vector<Val>& vec, size_t index, Compare comp) {
246         if (index >= vec.size()) {
247             return index;  // No further operation can be done!
248         }
249         const Val value = vec[index];  // A convenience copy!
250         size_t parent   = (index > 0 ? (index - 1) / 2 : 0);
251
252         if (comp(vec[parent], value)) {
253             // The value need to move towards the root.
254             while ((index > 0) && comp(vec[parent], value)) {
255                 std::swap(vec[index], vec[parent]);
256                 index  = parent;
257                 parent = (index - 1) / 2;
258             }
259         } else {
260             // The value needs to move towards the leaves
261             const size_t len = vec.size();
262             size_t child1 = 2 * (index + 1) - 1;
263             size_t child2 = std::min(vec.size() - 1, child1 + 1);
264             while ((child1 < len) && (comp(value, vec[child1]) ||
265                                       comp(value, vec[child2]))) {
266                 if (comp(vec[child1], vec[child2])) {
267                     child1 = child2;
268                 }
269                 std::swap(vec[index], vec[child1]);
270                 index  = child1;
271                 child1 = 2 * (index + 1) - 1;
272                 child2 = std::min(vec.size() - 1, child1 + 1);
273             }
274         }
275         return index;
276     }
277
278
279 protected:
280     /** The constructor for this EventQueue.
281
282         The constructor has been made private to ensure that this
283         class is never directly instantiated.  Instead, one of the
284         derived EventQueue classes must be instsantiated and used.
285
286         \param[in] name A human-readable identifier/name to be set
287         with this qeueue.
288     */
289     EventQueue(const std::string& name) : name(name) {}
290
291 private:
292     /** A human-readable identifier/name associated with this queue.
293
294         This instance variable is set in the constructor and is never
295         changed during the life time of this class.  This string is an
296         identifier that can be used to identify the final derived
297         class.
298     */
299     std::string name;
300 };
```

```
301
302 END_NAMESPACE(xxxx)
303
304 #endif
```

```
#ifndef HEAP_EVENT_QUEUE_H
#define HEAP_EVENT_QUEUE_H

#include <vector>
#include "EventQueue.h"

BEGIN_NAMESPACE(xxxx)

/** A standard heap-based event queue for managing events.

    <p>This class provides a very simple (good candidate for base-case
    comparisons) heap-based event queue for managing events for
    simulation.  This class uses standard C++ algorithms (such as: \c
    std::make_heap, \c std::push_heap, \c std::pop_heap) to manage a
    heap of events.  The events are stored in a backing
    std::vector.</p>

    <p>The behavior of the data structure exposed by this class is
    very close to the \c std::priority_queue.  The differences between
    the two are subtle in the sense that this class provides a few
    extra features, namely:

    <ol>

    <li>Explicit control over addition and removal of events from the
    heap.</li>

    <li>Methods for bulk removal of events necessary for efficient
    event managment during parallel simulation (to cancel events after
    a rollback).

    </ol>

    </p>
*/
class HeapEventQueue : public EventQueue {
public:
    /** The constructor for the HeapEventQueue.

        The default (and only) constructor for this class.  The
        constructor does not have any specific task to perform other
        than set a suitable identifier in the base class.
    */
    HeapEventQueue();

    /** The destructor.

        The destructor does not have any special tasks to perform.
        All the dynamic memory management is handled by the standard
        containers (namely std::vector) that is used internally by
        this class.
    */
    ~HeapEventQueue();

    /** Add/register an agent with the event queue.

        <p>This method implements the corresponding API method in the
        class.  Refer to the API documentation in the base class for
        intended functionality.</p>

        <p>This class does not utilize the agent registration
        information and consequently this method does not perform any
        special action.</p>

        \param[in,out] agent A pointer to the agent to be registered.
        This value is not used.

        \return This method returns NULL as its internal
        cross-reference to be stored in an agent.
    */
    virtual void* addAgent(xxxx::Agent* agent);

    /** Remove/unregister an agent with the event queue.

        <p>This method implements the corresponding API method in the
```



```
        class.  Refer to the API documentation in the base class for
        intended functionality.</p>

        <p>This method removes all events scheduled for the specified
        agent in its internal data structures.</p>

        \param[in,out] agent A pointer to the agent whose events are
        to be removed from the heap managed by this class.
    */
    void removeAgent(xxxx::Agent* agent) override;

    /** Determine if the event queue is empty.

        This method implements the base class API to report if any
        events are pending to be processed in the event queue.

        \return This method returns true if the event queue is
        logically empty.
    */
    virtual bool empty() { return eventList.empty();  }

    /** Obtain pointer to the highest priority (lowest receive time)
        event.

        This method can be used to obtain a pointer to the highest
        priority event in this event queue, without de-queuing the
        event.

        \note The event returned by this method is not dequeued.

        \return A pointer to the next event to be processed.  If the
        queue is empty then this method returns NULL.
    */
    virtual xxxx::Event* front();

    /** Method to obtain the next batch of events to be processed by
        one agent.

        In XXXX agents are scheduled to process all events at a given
        simulation time.  The next concurrent events (i.e., events
        with the same receive time) with the lowest time stamp are to
        be placed in the supplied event container.  The event
        container is then passed to the corresponding agent for
        further processing.

        \param[out] events The event container in which the next set
        of concurrent events are to be placed.  Note that the order of
        concurrent events in the event container is unspecified.
    */
    virtual void dequeueNextAgentEvents(xxxx::EventContainer& events);

    /** Enqueue a new event.

        This method must be used to enqueue/add an event to this event
        queue.  Once added the reference count on the event is
        increased.

        \param[in] agent The agent to which the event is to be
        scheduled.  This agent corresponds to the agent ID returned by
        event->getReceiverAgentID() method.  Currently, this method
        does not really use this value.

        \param[in] event The event to be enqueued.  This parameter can
        never be NULL.
    */
    virtual void enqueue(xxxx::Agent* agent, xxxx::Event* event);

    /** Enqueue a batch of events.

        This method can be used to enqueue/add a batch of events to
        this event queue.  Once added the reference count on each one
        of the events is increased.  This method provides a convenient
        approach to enqueue a batch of events, particularly after a
        rollback.
```

```
151         \param[in] agent The agent to which the event is to be
152         scheduled.  This agent corresponds to the agent ID returned by
153         event->getReceiverAgentID() method.  Currently, this value is
154         not used.
155
156         \param[in] event The list of events to be enqueued.  The
157         reference counts of the events in the container remains
158         unmodified.  The list of events become part of the event
159         queue.
160     */
161     virtual void enqueue(xxxx::Agent* agent, xxxx::EventContainer& events);
162
163     /** Dequeue all events sent by an agent after a given time.
164
165         This method implements the base class API method.  This method
166         can be used to remove/erase events sent by a given agent after
167         a given simulation time.  This API is needed to cancel events
168         during a rollback.
169
170         \param[in] dest The agent whose currently secheduled events
171         are to be checked and cleaned-up.  This agent must be a valid
172         agent that has been registered/added to this event queue.  The
173         pointer cannot be NULL.  This parameter is not used.
174
175         \param[in] sender The ID of the agent whose events have to be
176         removed.  This agent must be a valid agent that has been
177         registered/added to this event queue.  The pointer cannot be
178         NULL.
179
180         \param[in] sentTime The time from which the events are to be
181         removed.  All events (including those sent at this time) sent
182         by the sender agent are removed from this event queue.
183
184         \return This method returns the number of events actually
185         removed.
186     */
187     virtual int eraseAfter(xxxx::Agent* dest, const xxxx::AgentID sender,
188                            const xxxx::Time sentTime);
189
190     /** Print full contents of scheduler queue to given output stream.
191
192         This is a convenience method that is used primarily for
193         troubleshooting purposes.  This method prints all the events
194         in this queue, with each event on its own line.
195
196         \param[out] os The output stream to which the contents of the
197         queue are to be written.
198     */
199     virtual void prettyPrint(std::ostream& os) const;
200
201     /** Method to report aggregate statistics.
202
203         This method is invoked at the end of simulation after all
204         agents on this rank have been finalized.  This method can
205         report any aggregate statistics from the event
206         queue. Currently, this method does not have any additional
207         statistics to report.
208
209         \param[out] os The output stream to which the statistics are
210         to be written.
211     */
212     virtual void reportStats(std::ostream& os);
213
214 protected:
215     /** Convenience method to remove the front event.
216
217         This is an internal convenience method that is used to remove
218         the front (i.e., event with lowest timestamp) event from this
219         queue.  This method uses std::pop_heap to remove the event,
220         thereby ensuring that the heap property is maintained.
221
222         \return If the event queue is empty, then this method return
223         NULL.  Otherwise it removes the lowest timestamp event from
224         the queue and returns it.
225     */
```



```
226     xxxx::Event* pop_front();
227
228     /** Comparator method to sort events in the heap.
229
230         This is the comparator method that is passed to various
231         standard C++ algorithms to organize events as a heap.  This
232         comparator method gives first preference to receive time of
233         events.  Tie between two events with the same recieve time is
234         broken based on the receiver agent ID.
235
236         \param[in] lhs The left-hand-side event to be used for
237         comparison.  This parameter cannot be NULL.
238
239         \param[in] rhs The right-hand-side event to be used for
240         comparison. This parameter cannot be NULL.
241
242         \return This method returns if lhs < rhs, i.e., the lhs event
243         should be scheduled before the rhs event.
244     */
245     static inline bool compare(const xxxx::Event* const lhs,
246                                const xxxx::Event* const rhs) {
247         return ((lhs->getReceiveTime() > rhs->getReceiveTime()) ||
248                 ((lhs->getReceiveTime() == rhs->getReceiveTime() &&
249                   (lhs->getReceiverAgentID() > rhs->getReceiverAgentID()))));
250     }
251
252 private:
253     /** The backing storage for events managed by this class.
254
255         This vector contains the list of events being managed by the
256         class.  The events in the vector are stored and maintained as
257         a heap.  The heap is created and managed using standard C++
258         algorithms, namely: \c std::make_heap, \c std::push_heap, and
259         \c std::pop_heap.
260     */
261     std::vector<Event*> eventList;
262
263     /** This instance variable tracks to maximum number of events in
264         the eventList.
265
266         This instance variable is used to track the maximum number of
267         events in the event list.  This value is reported (typically
268         at the end) when the reportStats() method is invoked.
269     */
270     size_t maxQsize;
271 };
272
273 END_NAMESPACE(xxxx)
274
275 #endif
```

```cpp
#ifndef HEAP_EVENT_QUEUE_CPP
#define HEAP_EVENT_QUEUE_CPP

#include <algorithm>
#include "HeapEventQueue.h"

BEGIN_NAMESPACE(xxxx)

HeapEventQueue::HeapEventQueue() : EventQueue("HeapEventQueue"), maxQsize(0) {
    // Nothing else to be done.
}

xxxx::HeapEventQueue::~HeapEventQueue() {
    // Nothing else to be done.
}

void*
HeapEventQueue::addAgent(xxxx::Agent* agent) {
    UNUSED_PARAM(agent);
    return NULL;
}

void
HeapEventQueue::removeAgent(xxxx::Agent* agent) {
    ASSERT( agent != NULL );
    const AgentID id = agent->getAgentID();
    long currIdx     = eventList.size() - 1;
    // NOTE: Here the heap is sorted based on receive time for
    // scheduling.  However, we are canceling based on sentTime.
    // Consequently, doing any clever optimizations to minimize
    // iterations will backfire!
    while (!eventList.empty() && (currIdx >= 0)) {
        Event* const evt = eventList[currIdx];
        ASSERT(evt != NULL);
        // the antiMessage's and if the event is from same sender
        if (evt->getReceiverAgentID() == id) {
            // This event is for the agent being removed. Delete it.
            evt->decreaseReference();
            // Now it is time to patchup the hole and fix up the heap.
            // To patch-up we move event from the bottom up to this
            // slot and then fix-up the heap.
            eventList[currIdx] = eventList.back();
            eventList.pop_back();
            EventQueue::fixHeap(eventList, currIdx, compare);
            // Update the current index so that it is within bounds.
            currIdx = std::min<long>(currIdx, eventList.size() - 1);
        } else {
            // Check the previous element in the vector to see if that
            // is a candidate for cancellation.
            currIdx--;
        }
    }
}

xxxx::Event*
HeapEventQueue::front() {
    return !empty() ? eventList.front() : NULL;
}

void
HeapEventQueue::dequeueNextAgentEvents(xxxx::EventContainer& events) {
    if (eventList.empty()) {
        return;  // Nothing to be removed
    }
    // Initialize iterators
    const xxxx::Event*  nextEvt    = front();
    const xxxx::AgentID receiver   = nextEvt->getReceiverAgentID();
    const xxxx::Time    currTime   = nextEvt->getReceiveTime();
    // Remove all concurrent events for the next agent from the queue
    do {
        xxxx::Event* event = pop_front();
        events.push_back(event);
        nextEvt = (!empty() ? front() : NULL);
        DEBUG(std::cout << "Delivering: " << *event << std::endl);
    } while (!empty() && (nextEvt->getReceiverAgentID() == receiver) &&
```



```cpp
             TIME_EQUALS(nextEvt->getReceiveTime(), currTime));
}

void
HeapEventQueue::enqueue(xxxx::Agent* agent, xxxx::Event* event) {
    UNUSED_PARAM(agent);
    ASSERT(agent != NULL);
    ASSERT(event != NULL);
    event->increaseReference();
    eventList.push_back(event);
    maxQsize = std::max(maxQsize, eventList.size());
    std::push_heap(eventList.begin(), eventList.end(), compare);
}

void
HeapEventQueue::enqueue(xxxx::Agent* agent, xxxx::EventContainer& events) {
    UNUSED_PARAM(agent);
    ASSERT(agent != NULL);
    // Due to bulk adding, ensure that the heap container has enough
    // capacity.
    eventList.reserve(eventList.size() + events.size() + 1);
    // Add all the events to the conainer.
    for(EventContainer::iterator curr = events.begin(); (curr != events.end());
        curr++) {
        xxxx::Event* event = *curr;
        eventList.push_back(event);
        std::push_heap(eventList.begin(), eventList.end(), compare);
    }
    maxQsize = std::max(maxQsize, eventList.size());
    // Clear out events in the container as per API expectations
    events.clear();
}

// #define REBUILD_HEAP 1
#ifdef REBUILD_HEAP
int
HeapEventQueue::eraseAfter(xxxx::Agent* dest, const xxxx::AgentID sender,
                           const xxxx::Time sentTime) {
    UNUSED_PARAM(dest);
    ASSERT(dest != NULL);
    // Copy events to be retained into a temporary vector and finally
    // swap it with sel.
    size_t removedCount = 0;       // Count of events removed.
    std::vector<Event*> retained;  // Temporary vector with non-canceled events
    retained.reserve(eventList.size());
    for (auto curr = eventList.begin(); (curr != eventList.end()); curr++) {
        xxxx::Event* const event = *curr;
        if ((event->getSenderAgentID() == sender) &&
            (event->getSentTime() >= sentTime)) {
            event->decreaseReference();   // Canceled event!
            removedCount++;
        } else {
            retained.push_back(event);    // Reschedule the event.
        }
    }
    // Update the sel with list of retained events
    eventList.swap(retained);
    std::make_heap(eventList.begin(), eventList.end(), compare);
    return removedCount;
}
#else
int
HeapEventQueue::eraseAfter(xxxx::Agent* dest, const xxxx::AgentID sender,
                           const xxxx::Time sentTime) {
    UNUSED_PARAM(dest);
    ASSERT(dest != NULL);
    int  numRemoved = 0;
    long currIdx    = eventList.size() - 1;
    // NOTE: Here the heap is sorted based on receive time for
    // scheduling.  However, we are canceling based on sentTime.
    // Consequently, doing any clever optimizations to minimize
    // iterations will backfire!
    while (!eventList.empty() && (currIdx >= 0)) {
        // // // ASSERT(currIdx < eventList.size());
        Event* const evt = eventList[currIdx];
```

```
151         ASSERT(evt != NULL);
152         // An event is deleted only if the *sent* time is greater than
153         // the antiMessage's and if the event is from same sender
154         if ((evt->getSenderAgentID() == sender) &&
155             (evt->getSentTime() >= sentTime)) {
156             // This event needs to be cancelled.
157             evt->decreaseReference();
158             numRemoved++;
159             // Now it is time to patchup the hole and fix up the heap.
160             // To patch-up we move event from the bottom up to this
161             // slot and then fix-up the heap.
162             eventList[currIdx] = eventList.back();
163             eventList.pop_back();
164             EventQueue::fixHeap(eventList, currIdx, compare);
165             // Update the current index so that it is within bounds.
166             currIdx = std::min<long>(currIdx, eventList.size() - 1);
167         } else {
168             // Check the previous element in the vector to see if that
169             // is a candidate for cancellation.
170             currIdx--;
171         }
172     }
173     // Return number of events canceled to track statistics.
174     return numRemoved;
175 }
176 #endif
177
178 void
179 HeapEventQueue::prettyPrint(std::ostream& os) const {
180     os << "HeapEventQueue [size=" << eventList.size() << "]:\n";
181     for (auto curr = eventList.begin(); (curr != eventList.end()); curr++) {
182         os << " " << (*curr) << std::endl;
183     }
184     os << std::endl;
185
186 }
187
188 void
189 HeapEventQueue::reportStats(std::ostream& os) {
190     os << "HeapEventQueue:\n"
191        << "\tMax queue size: " << maxQsize << std::endl;
192 }
193
194 xxxx::Event*
195 HeapEventQueue::pop_front() {
196     ASSERT(!empty());
197     std::pop_heap(eventList.begin(), eventList.end(), compare);
198     xxxx::Event* retVal = eventList.back();
199     eventList.pop_back();
200     return retVal;
201 }
202
203 END_NAMESPACE(xxxx)
204
205 #endif
```

```
#ifndef TWO_TIER_HEAP_EVENT_QUEUE_H
#define TWO_TIER_HEAP_EVENT_QUEUE_H

#include <vector>
#include "EventQueue.h"
#include "BinaryHeapWrapper.h"

BEGIN_NAMESPACE(xxxx)

/** A two-tier-heap aka "2tHeap" or "heap-of-heap" event queue for
    managing events.

    <p>This class provides a heap-of-heap based event queue for
    managing events for simulation.  The two-tiers are organized as
    follows:</p>

    <p><u>First tier:</u> This class uses standard C++ algorithms
    (such as: \c std::make_heap, \c std::push_heap, \c std::pop_heap)
    to manage a heap of events for each agent.  The events are stored
    in a backing std::vector in each agent.  It is the same per-agent
    infrastructure as used by Fibonacci heap (implemented in AgentPQ
    class).</p>

    <p><u>Second tier:</u> This class specifically handles the
    necessary behavior of the second tier of operations -- that is
    scheduling of agents by maintaining heap of agents.</p>

    \note On the long run it would be better to avoid reliance on
    std::push_heap or std::pop_heap methods due to implicit dependence
    in the fixHeap() method.
*/
class TwoTierHeapEventQueue : public EventQueue {
public:
    /** The constructor for the TwoTierHeapEventQueue.

        The default (and only) constructor for this class.  The
        constructor does not have any specific task to perform other
        than set a suitable identifier in the base class.
    */
    TwoTierHeapEventQueue();

    /** The destructor.

        The destructor does not have any special tasks to perform.
        All the dynamic memory management is handled by the standard
        containers (namely std::vector) that is used internally by
        this class.
    */
    ~TwoTierHeapEventQueue();

    /** Add/register an agent with the event queue.

        <p>This method implements the corresponding API method in the
        EventQueue class.  Refer to the API documentation in the base
        class for intended functionality.</p>

        <p>This class uses the supplied agent pointer to setup the
        list of agents managed and scheduled by this class.</p>

        \param[in,out] agent A pointer to the agent to be registered.
        This parameter is used to setup the TwoTierHeapAdapter object
        associated with the agent.

        \return This method returns the iterator to the position of
        the agent in its internal vector as a cross-reference to be
        stored in an agent.
    */
    virtual void* addAgent(xxxx::Agent* agent);

    /** Remove/unregister an agent with this event queue.

        <p>This method implements the corresponding API method in the
        EventQueue class.  Refer to the API documentation in the base
        class for intended functionality.</p>

```



```
        <p>This class uses the supplied agent pointer to delete the
        TwoTierHeapAdapter object associated with the agent by the
        addAgent method.</p>

        \param[in,out] agent A pointer to the agent to be registered.
        This parameter is used to delete the TwoTierHeapAdapter object
        associated with the agent.
    */
    virtual void removeAgent(xxxx::Agent* agent);

    /** Determine if the event queue is empty.

        This method implements the base class API to report if any
        events are pending to be processed in the event queue.

        \return This method returns true if the event queue of the
        top-agent is logically empty.
    */
    virtual bool empty() {
        return (agentList.empty() || top()->schedRef.eventPQ->empty());
    }

    /** Obtain pointer to the highest priority (lowest receive time)
        event.

        This method can be used to obtain a pointer to the highest
        priority event in this event queue, without de-queuing the
        event.

        \note The event returned by this method is not dequeued.

        \return A pointer to the next event to be processed.  If the
        queue is empty then this method returns NULL.
    */
    virtual xxxx::Event* front();

    /** Method to obtain the next batch of events to be processed by
        one agent.

        <p>In XXXX agents are scheduled to process all events at a
        given simulation time.  The next concurrent events (i.e.,
        events with the same receive time) with the lowest time stamp
        are to be placed in the supplied event container.  The event
        container is then passed to the corresponding agent for
        further processing.</p>

        <p>This method essentially delegates the dequeue process to
        the agent with the next lowest timestamp.  Once the agent has
        been dequeued, this method fixes the heap by placing the
        top-agent in its appropriate location in the heap.</p>

        \param[out] events The event container in which the next set
        of concurrent events are to be placed.  Note that the order of
        concurrent events in the event container is unspecified.
    */
    virtual void dequeueNextAgentEvents(xxxx::EventContainer& events);

    /** Enqueue a new event.

        This method must be used to enqueue/add an event to this event
        queue.  Once added the reference count on the event is
        increased.  This method adds the event to the specified agent.
        Next this method fixes the heap to ensure that the agent with
        the least-time-stamp is at the top of the heap.

        \param[in] agent The agent to which the event is to be
        scheduled.  This agent corresponds to the agent ID returned by
        event->getReceiverAgentID() method.

        \param[in] event The event to be enqueued.  This parameter can
        never be NULL.
    */
    virtual void enqueue(xxxx::Agent* agent, xxxx::Event* event);

    /** Enqueue a batch of events.
```

```
151
152         This method can be used to enqueue/add a batch of events to
153         this event queue.  Once added the reference count on each one
154         of the events is increased.  This method provides a convenient
155         approach to enqueue a batch of events, particularly after a
156         rollback.  Next this method fixes the heap to ensure that the
157         agent with the least-time-stamp is at the top of the heap.
158
159         \param[in] agent The agent to which the event is to be
160         scheduled.  This agent corresponds to the agent ID returned by
161         event->getReceiverAgentID() method.  Currently, this value is
162         not used.
163
164         \param[in] event The list of events to be enqueued.  The
165         reference counts of the events in the container remains
166         unmodified.  The list of events become part of the event
167         queue.
168     */
169     virtual void enqueue(xxxx::Agent* agent, xxxx::EventContainer& events);
170
171     /** Dequeue all events sent by an agent after a given time.
172
173         This method implements the base class API method.  This method
174         can be used to remove/erase events sent by a given agent after
175         a given simulation time.  This API is needed to cancel events
176         during a rollback.  Next this method fixes the heap to ensure
177         that the agent with the least-time-stamp is at the top of the
178         heap.
179
180         \param[in] dest The agent whose currently secheduled events
181         are to be checked and cleaned-up.  This agent must be a valid
182         agent that has been registered/added to this event queue.  The
183         pointer cannot be NULL.  This parameter is not used.
184
185         \param[in] sender The ID of the agent whose events have to be
186         removed.  This agent must be a valid agent that has been
187         registered/added to this event queue.  The pointer cannot be
188         NULL.
189
190         \param[in] sentTime The time from which the events are to be
191         removed.  All events (including those sent at this time) sent
192         by the sender agent are removed from this event queue.
193
194         \return This method returns the number of events actually
195         removed.
196     */
197     virtual int eraseAfter(xxxx::Agent* dest, const xxxx::AgentID sender,
198                            const xxxx::Time sentTime);
199
200     /** Print full contents of scheduler queue to given output stream.
201
202         This is a convenience method that is used primarily for
203         troubleshooting purposes.  This method prints all the events
204         in this queue, with each event on its own line.
205
206         \param[out] os The output stream to which the contents of the
207         queue are to be written.
208     */
209     virtual void prettyPrint(std::ostream& os) const;
210
211     /** Method to report aggregate statistics.
212
213         This method is invoked at the end of simulation after all
214         agents on this rank have been finalized.  This method can
215         report any aggregate statistics from the event
216         queue. Currently, this method does not have any additional
217         statistics to report.
218
219         \param[out] os The output stream to which the statistics are
220         to be written.
221     */
222     virtual void reportStats(std::ostream& os);
223
224 protected:
225     /** Convenience method to get the top-event time for a given
```



```
226         agent.
227
228         This method is a convenience wrapper to call
229         BinaryHeapWrapper::getTopTime() method.
230
231         \return The receive time of top event's recv time or
232         TIME_INFINITY if heap is empty.
233     */
234     xxxx::Time getTopTime(const xxxx::Agent* const agent) const {
235         return agent->schedRef.eventPQ->getTopTime();
236     }
237
238     /** Comparator method to sort events in the heap.
239
240         This is the comparator method that is passed to various
241         standard C++ algorithms to organize events as a heap.  This
242         comparator method gives first preference to receive time of
243         events.  Tie between two events with the same recieve time is
244         broken based on the receiver agent ID.
245
246         \param[in] lhs The left-hand-side event to be used for
247         comparison.  This parameter cannot be NULL.
248
249         \param[in] rhs The right-hand-side event to be used for
250         comparison. This parameter cannot be NULL.
251
252         \return This method returns if lhs < rhs, i.e., the lhs event
253         should be scheduled before the rhs event.
254     */
255     inline bool compare(const xxxx::Agent* const lhs,
256                         const xxxx::Agent* const rhs) {
257         return getTopTime(lhs) >= getTopTime(rhs);
258     }
259
260     /** Convenience method to obtain the top-most or front agent.
261
262         This method can be used to obtain a pointer to the top/front
263         agent -- that is, the agent with the lowest timestamp event to
264         be scheduled next.
265
266         \return A pointer to the top-most agent in this heap.
267     */
268     xxxx::Agent* top() {
269         return agentList.front();
270     }
271
272     /** Obtain the current index of the agent from it's
273         cross-reference.
274
275         This method is a refactored utility method that has been
276         introduced to streamline the code.  This method essentially
277         obtains the index position of the given agent in the agentList
278         vector from the agent's fibHeapPtr corss-reference.  This
279         cross-reference is consistently updated by the various methods
280         in this class to enable rapid access to the location of the
281         agent.
282
283         \param[in] agent The agent whose index value in the agentList
284         is to be determined.
285
286         \return The index position of the agent in the agentList
287         vector (if all checks pass).
288     */
289     size_t getIndex(xxxx::Agent *agent) const;
290
291     /** Update position of agent in the scheduler's heap.
292
293         This is an internal helper method that is used to update the
294         position of an agent in the scheduler's heap.  This method
295         essentially performs sanity checks, uses the fixHeap() method
296         to update position of the agent, and updates cross references
297         for future use.
298
299         \param[in] agent The agent whose position in the heap is to be
300         updated.  This pointer cannot be NULL.
```

```
301
302         \return This method returns the updated index position of the
303         agent in agentList (the vector that serves as storage for the
304         heap).
305     */
306     size_t updateHeap(xxxx::Agent* agent);
307
308     /** Fix-up the location of the agent in the heap.
309
310         This method can be used to update the location of an agent in
311         the heap.
312
313         \note The implementation for this method has been heavily
314         borrowed from libstdc++'s code base to ensure that heap
315         updates are consistent with std::make_heap API.
316         Unfortunately, this does imply that there is a chance this
317         method may be incompatible with future versions.
318
319         \param[in] currPos The current position of the agent in the
320         heap whose position is to be updated.  This value is the index
321         position of the agent in the agentList vector.
322
323         \return This method returns the new position of the agent in
324         the agentList vector.
325     */
326     size_t fixHeap(size_t currPos);
327
328     /** The getNextEvents method.
329
330         This method is a helper that will grab the next set of events
331         to be processed for a given agent.  This method is invoked in
332         dequeueNextAgentEvents() method in this class.
333
334         \param[out] container The reference of the container into
335         which events should be added.
336     */
337     void getNextEvents(Agent* agent, EventContainer& container);
338
339 private:
340     /** The backing storage for events managed by this class.
341
342         This vector contains the list of agents being managed by the
343         class.  The agents in the vector are stored and maintained as
344         a heap.  The heap is created and managed using standard C++
345         algorithms, namely: \c std::make_heap, \c std::push_heap, and
346         \c std::pop_heap.
347     */
348     std::vector<xxxx::Agent*> agentList;
349 };
350
351 END_NAMESPACE(xxxx)
352
353 #endif
```

```cpp
#ifndef TWO_TIER_HEAP_EVENT_QUEUE_CPP
#define TWO_TIER_HEAP_EVENT_QUEUE_CPP

#include <algorithm>
#include "TwoTierHeapEventQueue.h"

BEGIN_NAMESPACE(xxxx)

TwoTierHeapEventQueue::TwoTierHeapEventQueue() :
EventQueue("TwoTierHeapEventQueue") {
    // Nothing else to be done.
}

TwoTierHeapEventQueue::~TwoTierHeapEventQueue() {
    // Nothing else to be done.
}

void*
TwoTierHeapEventQueue::addAgent(xxxx::Agent* agent) {
    agentList.push_back(agent);
    // Create the binary heap adapter that manages events for the agent.
    agent->schedRef.eventPQ = new BinaryHeapWrapper();
    // Return index of agent used to quickly update the heap
    return reinterpret_cast<void*> (agentList.size() - 1);
}

void
TwoTierHeapEventQueue::removeAgent(xxxx::Agent* agent) {
    ASSERT(agent != NULL);
    ASSERT(agent->schedRef.eventPQ != NULL);
    // Remove all events for this agent from the 2nd tier heap.
    agent->schedRef.eventPQ->clear();
    // Update the heap to place agent with LTSF
    updateHeap(agent);
}

xxxx::Event*
TwoTierHeapEventQueue::front() {
    xxxx::Event* retVal = NULL;
    if (!empty()) {
        retVal = top()->schedRef.eventPQ->top();
    }
    return retVal;
}

void
TwoTierHeapEventQueue::getNextEvents(Agent* agent, EventContainer& container) {
    ASSERT(container.empty());
    ASSERT(agent->schedRef.eventPQ->top() != NULL );
    BinaryHeapWrapper* const eventPQ = agent->schedRef.eventPQ;
    const Time currTime = eventPQ->getTopTime();
    do {
        Event* event = eventPQ->top();
        // We should never process an anti-message.
        if (event->isAntiMessage()) {
            std::cerr << "Anti-message Processing: " << *event << std::endl;
            std::cerr << "Trying to process an anti-message event, "
                      << "please notify XXXX developers of this issue"
                      << std::endl;
            abort();
        }

        // Ensure that the top event is greater than LVT
        if (event->getReceiveTime() <= agent->getTime(Agent::LVT)) {
            std::cerr << "Agent is being scheduled to process an event ("
                      << *event << ") that is at or below it LVT (LVT="
                      << agent->getTime(Agent::LVT) << ", GVT="
                      << agent->getTime(Agent::GVT)
                      << "). This is a serious error. Aborting.\n";
            std::cerr << *agent << std::endl;
            abort();
        }

        ASSERT(event->getReferenceCount() < 3);
```



```cpp
        // We add the top event we popped to the event container
        event->increaseReference();
        container.push_back(event);

        DEBUG(std::cout << "Delivering: " << *event << std::endl);

        // Finally it is safe to remove this event from the eventPQ as
        // it has been added to the inputQueue
        eventPQ->pop();
    } while (!empty() && (TIME_EQUALS(eventPQ->getTopTime(), currTime)));
}

void
TwoTierHeapEventQueue::dequeueNextAgentEvents(xxxx::EventContainer& events) {
    if (!empty()) {
        // Get agent and validate.
        xxxx::Agent * const agent = top();
        ASSERT(agent != NULL);
        ASSERT(getIndex(agent) == 0);
        // Have the events give up its next set of events
        getNextEvents(agent, events);
        ASSERT(!events.empty());
        // Fix the position of this agent in the scheduler's heap
        updateHeap(agent);
    }
}

void
TwoTierHeapEventQueue::enqueue(xxxx::Agent* agent, xxxx::Event* event) {
    ASSERT(agent != NULL);
    ASSERT(event != NULL);
    ASSERT(getIndex(agent) < agentList.size());
    // Add event to the agent's heap first.
    agent->schedRef.eventPQ->push(event);
    // Now update the position of the agent in this tier for scheduling.
    updateHeap(agent);
}

void
TwoTierHeapEventQueue::enqueue(xxxx::Agent* agent,
                               xxxx::EventContainer& events) {
    ASSERT(agent != NULL);
    ASSERT(getIndex(agent) < agentList.size());
    // Add events to the agent's 1nd tier heap (if any)
    if (!events.empty()) {
        agent->schedRef.eventPQ->push(events);
    }
    // Update the 2nd tier heap for scheduling.
    updateHeap(agent);
}

int
TwoTierHeapEventQueue::eraseAfter(xxxx::Agent* dest, const xxxx::AgentID sender,
        const xxxx::Time sentTime) {
    ASSERT(dest != NULL);
    ASSERT(getIndex(dest) < agentList.size());
    // Get agent's heap to cancel out events.
    int numRemoved = dest->schedRef.eventPQ->removeFutureEvents(sender, sentTime
);
    // Update the 2nd tier heap for scheduling.
    updateHeap(dest);
    return numRemoved;
}

void
TwoTierHeapEventQueue::reportStats(std::ostream& os) {
    UNUSED_PARAM(os);
    // No statistics are currently reported.
}

void
TwoTierHeapEventQueue::prettyPrint(std::ostream& os) const {
    os << "TwoTierHeapEventQueue::prettyPreint() : not implemented.\n";
}
```

```
150 size_t
151 TwoTierHeapEventQueue::getIndex(xxxx::Agent *agent) const {
152     ASSERT(agent != NULL);
153     size_t index = reinterpret_cast<size_t> (agent->fibHeapPtr);
154     ASSERT(index < agentList.size());
155     ASSERT(agentList[index] == agent);
156     return index;
157 }
158
159 size_t
160 TwoTierHeapEventQueue::updateHeap(xxxx::Agent* agent) {
161     ASSERT(agent != NULL);
162     size_t index = getIndex(agent);
163     if (agent->oldTopTime != getTopTime(agent)) {
164         index = fixHeap(index);
165         // Update the position of the agent in the scheduler's heap
166         // Validate
167         ASSERT(agentList[index] == agent);
168         ASSERT(getIndex(agent) == index);
169         // Update time value as well for future access
170         agent->oldTopTime = getTopTime(agent);
171         // Validation check.
172         ASSERT(getTopTime(agentList[0]) <= getTopTime(agentList[1]));
173     }
174     // Return the new index position of the agent
175     return index;
176 }
177
178 size_t
179 TwoTierHeapEventQueue::fixHeap(size_t currPos) {
180     ASSERT(currPos < agentList.size());
181     xxxx::Agent* value = agentList[currPos];
182     const size_t len = (agentList.size() - 1) / 2;
183     size_t secondChild = currPos;
184     // This code was borrowed from libstdc++ implementation to ensure
185     // that the fix-ups are consistent with std::make_heap API.
186     while (secondChild < len) {
187         secondChild = 2 * (secondChild + 1);
188         if (compare(agentList[secondChild], agentList[secondChild - 1])) {
189             secondChild--;
190         }
191         agentList[currPos] = std::move(agentList[secondChild]);
192         agentList[currPos]->fibHeapPtr = reinterpret_cast<void*> (currPos);
193         currPos = secondChild;
194     }
195     if (((agentList.size() & 1) == 0) &&
196             (secondChild == (agentList.size() - 2) / 2)) {
197         secondChild = 2 * (secondChild + 1);
198         agentList[currPos] = std::move(agentList[secondChild - 1]);
199         agentList[currPos]->fibHeapPtr = reinterpret_cast<void*> (currPos);
200         currPos = secondChild - 1;
201     }
202     // Use libstdc++'s internal method to fix-up the vector from the
203     // given location.
204     // std::__push_heap(agentList.begin(), currPos, 0, value,
205     //                  __gnu_cxx::__ops::__iter_comp_val(compare));
206
207     size_t parent = (currPos - 1) / 2;
208     while ((currPos > 0) && (compare(agentList[parent], value))) {
209         agentList[currPos] = std::move(agentList[parent]);
210         agentList[currPos]->fibHeapPtr = reinterpret_cast<void*> (currPos);
211         currPos = parent;
212         parent = (currPos - 1) / 2;
213     }
214     agentList[currPos] = value;
215     agentList[currPos]->fibHeapPtr = reinterpret_cast<void*> (currPos);
216     // Return the final index position for the agent
217     return currPos;
218 }
219
220
221 END_NAMESPACE(xxxx)
222
223 #endif
```

```
/*
 * File:   AgentPQ.h
 *
 * This implementation is based from  Dietmar Kuehl's f_heap implementation.
 // ------------------------------------------------------------------------
 // This implementation is based on the description found in "Network Flow",
 // R.K.Ahuja, T.L.Magnanti, J.B.Orlin, Prentice Hall. The algorithmic stuff
 // should be basically identical to this description, however, the actual
 // representation is not.
 // ------------------------------------------------------------------------
 * Created on April 19, 2009, 3:08 AM
 */

#ifndef AGENTPQ_H
#define AGENTPQ_H

#include <functional>
#include <vector>
#include <iostream>
#include "EventQueue.h"
#include "BinaryHeapWrapper.h"

BEGIN_NAMESPACE(xxxx);

class AgentPQ : public EventQueue {

protected:
    //This is the node define. what the fibonacci heap will maintain.
    class node {
    public:

        node(Agent * data) : m_parent(0), m_lost_child(0), m_data(data) {
            m_children.reserve(8);
        }

        ~node() {
        }

        void destroy();
        node* join(node* tree); // add tree as a child
        void cut(node* child); // remove the child

        int lost_child() const {
            return m_lost_child;
        }

        void clear() {
            m_parent = 0;
            m_lost_child = 0;
        }

        int rank() const {
            return m_children.size();
        }

        bool is_root() const {
            return m_parent == 0;
        }

        node* parent() const {
            return m_parent;
        }

        void remove_all() {
            m_children.erase(m_children.begin(), m_children.end());
        }

        Agent* data() const {
            return m_data;
        }

        void data(Agent* data) {
            m_data = data;
        }

```



```
        std::vector<node*>::const_iterator begin() const {
            return m_children.begin();
        }

        std::vector<node*>::const_iterator end() const {
            return m_children.end();
        }

        void ppHelper(std::ostream& os, const std::string &indent) const {
            os << indent << ((is_root()) ? "*" : ">") << *data() << std::endl;
            for (size_t i = 0; (i < m_children.size()); i++) {
                if (m_children[i] != 0) {
                    m_children[i]->ppHelper(os, indent + "-");
                }
            }//end for
        }//end ppHelper

    private:
        int m_index; // index of the object in the parent's vector
        node* m_parent; // pointer to the parent node
        std::vector<node*> m_children;
        int m_lost_child;
        Agent* m_data;

        node(node const&);
        void operator=(node const&);
    };

protected:
    std::vector<node*> m_roots;
    int m_size;

public:
    typedef node* pointer;

    AgentPQ();
    ~AgentPQ();

    /** Add/register an agent with the event queue.

        This method implements the EventQueue API method that is
        invoked to add/register each agent that is present on the
        local Simulator/process.  This method creates a new Fibonacci
        node entry for the agent and adds it as the current root of
        the Fibonacci heap. The minimum value is reset to NULL, to
        ensure the minimum root is recomputed when needed.

        \param[in,out] agent A pointer to the agent to be registered.

        \return This method returns a pointer to the node that was
        created.  This node is never deleted and is strictly
        associated with the agent.
     */
    virtual void* addAgent(Agent* data);

    /** Remove/unregister an agent with the event queue.

        <p>This method implements the corresponding API method in the
        class.  Refer to the API documentation in the base class for
        intended functionality.</p>

        <p>This method removes all events scheduled for the specified
        agent in its internal data structures.</p>

        \param[in,out] agent A pointer to the agent whose events are
        to be removed from the heap managed by this class.
    */
    void removeAgent(xxxx::Agent* agent) override;

    /** Determine if the event queue is empty.

        This method implements the base class API to report if any
        events are pending to be processed in the event queue.

        \return This method returns true if the heap is logically
```

```
151         empty.
152      */
153     virtual bool empty() {
154         return (m_size == 0) || (top()->schedRef.eventPQ->empty());
155     }
156
157     /** Obtain pointer to the highest priority (lowest receive time)
158         event.
159
160         This method implements base class API method to return a
161         pointer to the highest priority event in this event queue,
162         without de-queuing the event.
163
164         \note The event returned by this method is not dequeued.
165
166         \return A pointer to the next event to be processed.  If the
167         heap is empty then this method returns NULL.
168      */
169     virtual xxxx::Event* front();
170
171     /** Method to obtain the next batch of events to be processed by
172         one agent.
173
174         In XXXX agents are scheduled to process all events at a given
175         simulation time.  The next concurrent events (i.e., events
176         with the same receive time) with the lowest time stamp are to
177         be placed in the supplied event container.
178
179         \param[out] events The event container in which the next set
180         of concurrent events are to be placed.  Note that the order of
181         concurrent events in the event container is unspecified.
182      */
183     virtual void dequeueNextAgentEvents(xxxx::EventContainer& events);
184
185     /** Enqueue a new event.
186
187         This method must be used to enqueue/add an event to this event
188         queue.  Once added the reference count on the event is
189         increased.
190
191         \param[in] agent The agent to which the event is to be
192         scheduled.  This agent corresponds to the agent ID returned by
193         event->getReceiverAgentID() method.
194
195         \param[in] event The event to be enqueued.  This parameter can
196         never be NULL.
197      */
198     virtual void enqueue(xxxx::Agent* agent, xxxx::Event* event);
199
200     /** Enqueue a batch of events.
201
202         This method can be used to enqueue/add a batch of events to
203         this event queue.  Once added the reference count on each one
204         of the events is increased.  This method provides a convenient
205         approach to enqueue a batch of events, particularly after a
206         rollback.
207
208         \param[in] agent The agent to which the event is to be
209         scheduled.  This agent corresponds to the agent ID returned by
210         event->getReceiverAgentID() method.
211
212         \param[in] event The event to be enqueued.  This parameter can
213         never be NULL.
214      */
215     virtual void enqueue(xxxx::Agent* agent, xxxx::EventContainer& events);
216
217     /** Dequeue all events sent by an agent after a given time.
218
219         This method can be used to remove/erase events sent by a given
220         agent after a given simulation time.  This API is needed to
221         cancel events during a rollback.
222
223         \param[in] dest The agent whose currently secheduled events
224         are to be checked and cleaned-up.  This agent must be a valid
225         agent that has been registered/added to this event queue.  The
```



```
226         pointer cannot be NULL.
227
228         \param[in] sender The ID of the agent whose events have to be
229         removed.  This agent must be a valid agent that has been
230         registered/added to this event queue.  The pointer cannot be
231         NULL.
232
233         \param[in] sentTime The time from which the events are to be
234         removed.  All events (including those sent at this time) sent
235         by the sender agent are removed from this event queue.
236
237         \return This method returns the number of events actually
238         removed.
239      */
240     virtual int eraseAfter(xxxx::Agent* dest, const xxxx::AgentID sender,
241             const xxxx::Time sentTime);
242
243     Agent* top();
244     void update(pointer n, double old_top_time);
245     //void     remove(pointer);
246
247     bool empty() const {
248         return m_size == 0;
249     }
250
251     int size() const {
252         return m_size;
253     }
254
255     virtual void prettyPrint(std::ostream& os) const {
256         for (size_t i = 0; (i < m_roots.size()); i++) {
257             if (m_roots[i] != 0) {
258                 m_roots[i]->ppHelper(os, "-");
259             }
260         }
261     }
262
263     /** Method to report aggregate statistics.
264
265         This method is invoked at the end of simulation after all
266         agents on this rank have been finalized.  This method is meant
267         to report any aggregate statistics from this queue.
268         Currently, this method does not write any specific statistics.
269
270         \param[out] os The output stream to which the statistics are
271         to be written.
272      */
273     virtual void reportStats(std::ostream& os);
274
275 protected:
276     /** The getNextEvents method.
277
278         This method is a helper that will grab the next set of events
279         to be processed for a given agent.  This method is invoked in
280         dequeueNextAgentEvents() method in this class.
281
282         \param[out] container The reference of the container into
283         which events should be added.
284     */
285     void getNextEvents(Agent* agent, EventContainer& container);
286
287 private:
288     void decrease(pointer, Agent*);
289     void increase(pointer, Agent*);
290     void add_root(node* n);
291     void cut(node* n);
292     void find_min() const;
293     mutable node* m_min;
294
295     inline xxxx::Time getTopTime(const Agent* const agent) const {
296         return agent->schedRef.eventPQ->getTopTime();
297     }
298
299     inline bool compare(const Agent *lhs, const Agent * rhs) const {
300         return (getTopTime(lhs) >= getTopTime(rhs));
```

```
301     }
302
303     AgentPQ(AgentPQ const&); // deliberately not implemented
304     void operator=(AgentPQ const&); // deliberately not implemented
305 };
306
307
308 END_NAMESPACE(xxxx); //end namespace
309
310 #endif /* AGENTPQ_H */
```

```cpp
#ifndef AGENTPQ_CPP
#define AGENTPQ_CPP

#include "AgentPQ.h"
#include "BinaryHeapWrapper.h"

using namespace xxxx;

//ctor

AgentPQ::AgentPQ() : EventQueue("FibonacciHeap"), m_size(0) {
}

//dtor

AgentPQ::~AgentPQ() {
    if (m_size == 0)
        return;

    int idx = m_roots.size();
    while (idx-- != 0) {
        if (m_roots[idx] != 0) {
            m_roots[idx]->destroy();
            delete m_roots[idx];
        }//end if
    }//end while
}

//node methods below

void
AgentPQ::node::destroy() {
    int idx = m_children.size();
    while (idx-- != 0) {
        if (m_children[idx] != 0) {
            m_children[idx]->destroy();
            delete m_children[idx];
        }//end if
    }//end while
}

AgentPQ::node*
AgentPQ::node::join(node* tree) {
    tree->m_index = m_children.size();
    tree->m_parent = this;
    m_children.push_back(tree);
    m_lost_child = 0;
    return this;
}

void
AgentPQ::node::cut(node* child) {
    unsigned int index = child->m_index;
    if (m_children.size() > index + 1) {
        m_children[index] = m_children[m_children.size() - 1];
        m_children[index]->m_index = index;
    }//end if
    m_children.pop_back();
    ++m_lost_child;
}

//fibonacci heap helper methods below

void
AgentPQ::add_root(node* n) {
    unsigned rank = n->rank();
    if (m_roots.size() <= rank) {
        while (m_roots.size() < rank) {
            m_roots.push_back(0);
        }//end while
        m_roots.push_back(n);
        n->clear();

    } else if (m_roots[rank] == 0) {
        m_roots[rank] = n;
```



```cpp
        n->clear();

    } else {
        node* r = m_roots[rank];
        m_roots[rank] = 0;
        if (compare(n->data(), r->data())) {
            n->clear();
            add_root(r->join(n));
        } else {
            r->clear();
            add_root(n->join(r));
        }
    }//end if-else-ladder
}//end add_root

void
AgentPQ::find_min() const {
    if (m_size == 0) {
        m_min = 0;
    } else {
        std::vector<node*>::const_iterator end = m_roots.end();
        std::vector<node*>::const_iterator it = m_roots.begin();
        while (*it == 0) {
            ++it;
        }//end while
        m_min = *it;

        for (++it; it != end; ++it) {
            if (*it != 0 && (getTopTime(m_min->data()) >=
                             getTopTime((*it)->data()))) {
                m_min = *it;
            }//end if
        }//end for
    }//end if-else
}

void
AgentPQ::cut(node* n) {
    node* p = n->parent();
    p->cut(n);
    if (p->is_root()) {
        m_roots[p->rank() + 1] = 0;
        add_root(p);
    } else if (p->lost_child() == 2) {
        cut(p);
        add_root(p);
    }
}

//fibonacci heap modifying methods

void
AgentPQ::update(pointer n, Time old_top_time) {
    if (getTopTime(n->data()) > old_top_time) {
        // Means we need to push the data towards leafs
        decrease(n, n->data());
    } else if (getTopTime(n->data()) < old_top_time) {
        // Means we need to move up the heap towards root
        increase(n, n->data());
    }
}

void
AgentPQ::increase(pointer n, Agent* data) {
    n->data(data);
    if (!n->is_root() && compare(n->parent()->data(), n->data())) {
        cut(n);
        add_root(n);
    }
    m_min = 0;
}

void
AgentPQ::decrease(pointer n, Agent* data) {
    n->data(data);
```

```
151     std::vector<node*>::const_iterator it = n->begin();
152     std::vector<node*>::const_iterator end = n->end();
153     for (; it != end; ++it)
154         if (compare(n->data(), (*it)->data())) {
155             if (n->is_root()) {
156                 m_roots[n->rank()] = 0;
157             } else {
158                 cut(n);
159             }
160             for (it = n->begin(); it != end; ++it) {
161                 add_root(*it);
162             }
163             n->remove_all();
164 
165             add_root(n);
166             break;
167         }
168     m_min = 0;
169 }
170 
171 //void
172 //FibonacciHeap::remove(pointer n)           {
173 //  if (n->is_root())
174 //    m_roots[n->rank()] = 0;
175 //  else
176 //    cut(n);
177 //
178 //  std::vector<node*>::const_iterator it = n->begin();
179 //  std::vector<node*>::const_iterator end = n->end();
180 //  for (it = n->begin(); it != end; ++it){
181 //    add_root(*it);
182 //  }
183 //
184 //  m_min = 0;
185 //  --m_size;
186 //  delete n;
187 //}
188 
189 //fibonacci heap basic interface
190 
191 Agent*
192 AgentPQ::top() {
193     if (m_min == 0) {
194         find_min();
195     }
196     return m_min->data();
197 }
198 
199 //void
200 //FibonacciHeap::pop() {
201 //  if (m_min == 0){
202 //    find_min();
203 //  }
204 //
205 //  node* del = m_min;
206 //  m_roots[m_min->rank()] = 0;
207 //
208 //  std::vector<node*>::const_iterator end = del->end();
209 //
210 //  for (std::vector<node*>::const_iterator it = del->begin(); it != end; ++it){
211 //    add_root(*it);
212 //  }
213 //  --m_size;
214 //
215 //  delete del;
216 //  m_min = 0;
217 //}
218 
219 void*
220 AgentPQ::addAgent(Agent* data) {
221     // Create the binary heap that is used to maintain events for this agent.
222     data->schedRef.eventPQ = new BinaryHeapWrapper();
223     // Now create Fibonacci heap node entry for this agent.
224     node* n = new node(data);
225     ++m_size;
```



```
226     add_root(n);
227     m_min = 0;
228     return n;
229 }
230 
231 void
232 AgentPQ::removeAgent(Agent* agent) {
233     const Time oldTopTime = agent->oldTopTime;
234     agent->schedRef.eventPQ->clear();
235     pointer ptr = reinterpret_cast<pointer>(agent->fibHeapPtr);
236     ASSERT( ptr != NULL );
237     update(ptr, oldTopTime);
238     agent->oldTopTime = getTopTime(agent);
239 }
240 
241 xxxx::Event*
242 AgentPQ::front() {
243     xxxx::Event* retVal = NULL;
244     if (!empty()) {
245         retVal = top()->schedRef.eventPQ->top();
246     }
247     return retVal;
248 }
249 
250 void
251 AgentPQ::getNextEvents(Agent* agent, EventContainer& container) {
252     // First do some sanity checks to ensure things look fine.
253     ASSERT(agent != NULL );
254     ASSERT(agent->schedRef.eventPQ != NULL);
255     ASSERT(agent->schedRef.eventPQ->top() != NULL);
256     ASSERT(container.empty());
257     BinaryHeapWrapper* const eventPQ = agent->schedRef.eventPQ;
258     const xxxx::Time agentLVT = agent->getTime(Agent::LVT);
259     // Extract next batch of events with same receive time to be processed.
260     const Time currTime = eventPQ->top()->getReceiveTime();
261     do {
262         Event* const event = eventPQ->top();
263         ASSERT(event != NULL);
264         // We should never process an anti-message.
265         ASSERT(!event->isAntiMessage());
266         // Ensure that the top event is greater than LVT
267         ASSERT(event->getReceiveTime() > agentLVT);
268         // Ensure reference counts looks correct
269         ASSERT(event->getReferenceCount() < 3);
270         // We add the top event we popped to the event container
271         event->increaseReference();
272         container.push_back(event);
273 
274         DEBUG(std::cout << "Delivering: " << *event << std::endl);
275         // Finally it is safe to remove this event from the eventPQ as
276         // it has been added to the inputQueue
277         eventPQ->pop();
278     } while ((!eventPQ->empty()) &&
279             (TIME_EQUALS(eventPQ->getTopTime(), currTime)));
280 }
281 
282 void
283 AgentPQ::dequeueNextAgentEvents(xxxx::EventContainer& events) {
284     if (!empty()) {
285         xxxx::Agent *agent = top();
286         // agent->getNextEvents(events);
287         getNextEvents(agent, events);
288         ASSERT(!events.empty());
289         pointer ptr = reinterpret_cast<pointer>(agent->fibHeapPtr);
290         update(ptr, agent->oldTopTime);
291         agent->oldTopTime = getTopTime(agent);
292     }
293 }
294 
295 void
296 AgentPQ::enqueue(xxxx::Agent* agent, xxxx::Event* event) {
297     agent->schedRef.eventPQ->push(event);
298     pointer ptr = reinterpret_cast<pointer>(agent->fibHeapPtr);
299     update(ptr, agent->oldTopTime);
300     agent->oldTopTime = getTopTime(agent);
```

```
301 }
302
303 void
304 AgentPQ::enqueue(xxxx::Agent* agent, xxxx::EventContainer& events) {
305     const Time oldTopTime = agent->oldTopTime;
306     agent->schedRef.eventPQ->push(events);
307     pointer ptr = reinterpret_cast<pointer>(agent->fibHeapPtr);
308     update(ptr, oldTopTime);
309     agent->oldTopTime = getTopTime(agent);
310 }
311
312 int
313 AgentPQ::eraseAfter(xxxx::Agent* dest, const xxxx::AgentID sender,
314         const xxxx::Time sentTime) {
315     const Time oldTopTime = dest->oldTopTime;
316     int numRemoved = dest->schedRef.eventPQ->removeFutureEvents(sender, sentTime
    );
317     pointer ptr = reinterpret_cast<pointer>(dest->fibHeapPtr);
318     update(ptr, oldTopTime);
319     dest->oldTopTime = getTopTime(dest);
320     return numRemoved;
321 }
322
323 void
324 AgentPQ::reportStats(std::ostream& os) {
325     UNUSED_PARAM(os);
326     // No statistics are currently reported.
327 }
328
329 #endif
```

```
#ifndef THREE_TIER_HEAP_EVENT_QUEUE_H
#define THREE_TIER_HEAP_EVENT_QUEUE_H

#include <vector>
#include <stack>
#include <algorithm>
#include "Avg.h"
#include "EventQueue.h"
#include "TwoTierHeapOfVectorsEventQueue.h"

BEGIN_NAMESPACE(xxxx)

/** Class to encapsulate information for Tier2 entries/buckets.

        This is a simple class that encapsulates a list of events (in
        eventList) all with exactly the same receive time to a given
        agent.  These objects are cached/reused by the scheduler queue to
        reduce memory allocation operations, particularly for the
        eventList because memory management turns out to be he most
        expensive operation.
*/
class HOETier2Entry {
private:
    /** The receive time of events in this tier2 entry. Note that all
        the events in this entry are must/are concurrent -- that is,
        destined for the same agent at the same time.
     */
    Time recvTime;

    /** The list of entities in this HOE entry class */
    std::vector<xxxx::Event*> eventList;

public:
    /** Constructor to create a tier2 entry with 1 initial event in
        it.

        \param[in] event The event to be added to this tier2 entry.
        The receive time of the event is used as the receive time
        value.
    */
    HOETier2Entry(xxxx::Event* event) : recvTime(event->getReceiveTime()),
                                        eventList(1, event) {}

    /** Reset the information in this tier2 entry.

        This method is synonymous to the constructor except, it is
        used to reset/recycle an existing tier2 entry.

        \param[in] event The event to be added to this tier2 entry.
        The receive time of the event is used as the receive time
        value.
    */
    void reset(xxxx::Event* event) {
        recvTime = event->getReceiveTime();
        eventList.clear();
        eventList.emplace_back(event);
    }

    /** Appends events to the EventContainer list.
     *
     *  The method is used to append concurrent events to their respective
     *  position in the tier2 container.
     */
    void updateContainer(xxxx::Event* event){
        eventList.emplace_back(event);
    }

    /** Obtain pointer to the first event in this list.

        \return Pointer to the first event in this list. This return
        value cannot/should-not be NULL.
    */
    inline xxxx::Event* getEvent() const {
        return eventList.front();
    }
```



```

    /** \brief compares the receive times of events

        The method is used to determine whether or not an event
        already exists in the tier2 container.

        \returns True if lhs receiveTime is equal to rhs receiveTime
    */
    inline bool operator==(const HOETier2Entry &rhs) {
        return (this->recvTime == rhs.recvTime);
    }

    inline bool operator<(const HOETier2Entry &rhs) {
        return (this->recvTime < rhs.recvTime);
    }

    inline Time getReceiveTime() const {
        return recvTime;
    }

    inline const std::vector<xxxx::Event*>& getEventList() const {
        return eventList;
    }

    inline std::vector<xxxx::Event*>& getEventList() {
        return eventList;
    }
};

/** A three-tier-heap aka "3tHeap" or "heap-of-heap" event queue for
    managing events.

    <p>This class provides a heap-of-heap based event queue for
    managing events for simulation.  The two-tiers are organized as
    follows:</p>

    <p><u>First tier:</u> This class uses standard C++ algorithms
    (such as: \c std::make_heap, \c std::push_heap, \c std::pop_heap)
    to manage a heap of events for each agent.  The events are stored
    in a backing std::vector in each agent.  It is the same per-agent
    infrastructure as used by Fibonacci heap (implemented in AgentPQ
    class).</p>

    <p><u>Second tier:</u> This class specifically handles the
    necessary behavior of the second tier of operations -- that is
    scheduling of agents by maintaining heap of agents.</p>

    \note On the long run it would be better to avoid reliance on
    std::push_heap or std::pop_heap methods due to implicit dependence
    in the fixHeap() method.
*/
class ThreeTierHeapEventQueue : public EventQueue {
public:
    /** The constructor for the TwoTierHeapEventQueue.

        The default (and only) constructor for this class.  The
        constructor does not have any specific task to perform other
        than set a suitable identifier in the base class.
    */
    ThreeTierHeapEventQueue();

    /** The destructor.

        The destructor does not have any special tasks to perform.
        All the dynamic memory management is handled by the standard
        containers (namely std::vector) that is used internally by
        this class.
    */
    ~ThreeTierHeapEventQueue();

    /** Add/register an agent with the event queue.

        <p>This method implements the corresponding API method in the
        class.  Refer to the API documentation in the base class for
        intended functionality.</p>
```

```
151
152         <p>This class uses the supplied agent pointer to setup the
153         list of agents managed and scheduled by this class.</p>
154
155         \param[in,out] agent A pointer to the agent to be registered.
156         This value is not used.
157
158         \return This method returns the iterator to the position of
159         the agent in its internal vector as a cross-reference to be
160         stored in an agent.
161     */
162     virtual void* addAgent(xxxx::Agent* agent);
163
164     /** Remove/unregister an agent with the event queue.
165
166         <p>This method implements the corresponding API method in the
167         class.  Refer to the API documentation in the base class for
168         intended functionality.</p>
169
170         <p>This method removes all events scheduled for the specified
171         agent in its internal data structures.</p>
172
173         \param[in,out] agent A pointer to the agent whose events are
174         to be removed from the vector managed by this class.
175     */
176     void removeAgent(xxxx::Agent* agent) override;
177
178     /** Determine if the event queue is empty.
179
180         This method implements the base class API to report if any
181         events are pending to be processed in the event queue.
182
183         \return This method returns true if the event queue of the
184         top-agent is logically empty.
185     */
186     virtual bool empty() {
187         return (agentList.empty() || top()->tier2->empty());
188     }
189
190     /** Obtain pointer to the highest priority (lowest receive time)
191         event.
192
193         This method can be used to obtain a pointer to the highest
194         priority event in this event queue, without de-queuing the
195         event.
196
197         \note The event returned by this method is not dequeued.
198
199         \return A pointer to the next event to be processed.  If the
200         queue is empty then this method returns NULL.
201     */
202     virtual xxxx::Event* front();
203
204     /** Method to obtain the next batch of events to be processed by
205         one agent.
206
207         <p>In XXXX agents are scheduled to process all events at a
208         given simulation time.  The next concurrent events (i.e.,
209         events with the same receive time) with the lowest time stamp
210         are to be placed in the supplied event container.  The event
211         container is then passed to the corresponding agent for
212         further processing.</p>
213
214         <p>This method essentially delegates the dequeue process to
215         the agent with the next lowest timestamp.  Once the agent has
216         been dequeued, this method fixes the heap by placing the
217         top-agent in its appropriate location in the heap.</p>
218
219         \param[out] events The event container in which the next set
220         of concurrent events are to be placed.  Note that the order of
221         concurrent events in the event container is unspecified.
222     */
223     virtual void dequeueNextAgentEvents(xxxx::EventContainer& events);
224
225     /** Enqueue a new event.
```



```
226
227         This method must be used to enqueue/add an event to this event
228         queue.  Once added the reference count on the event is
229         increased.  This method adds the event to the specified agent.
230         Next this method fixes the heap to ensure that the agent with
231         the least-time-stamp is at the top of the heap.  This method
232         essentially uses an internal helper method to accomplish its
233         tasks.
234
235         \param[in] agent The agent to which the event is to be
236         scheduled.  This agent corresponds to the agent ID returned by
237         event->getReceiverAgentID() method.
238
239         \param[in] event The event to be enqueued.  This parameter can
240         never be NULL.
241     */
242     virtual void enqueue(xxxx::Agent* agent, xxxx::Event* event);
243
244     /** Enqueue a batch of events.
245
246         This method can be used to enqueue/add a batch of events to
247         this event queue.  Once added the reference count on each one
248         of the events is increased.  This method provides a convenient
249         approach to enqueue a batch of events, particularly after a
250         rollback.  Next this method fixes the heap to ensure that the
251         agent with the least-time-stamp is at the top of the heap.
252         This method uses an internal helper method to accomplish its
253         tasks.
254
255         \param[in] agent The agent to which the event is to be
256         scheduled.  This agent corresponds to the agent ID returned by
257         event->getReceiverAgentID() method.  Currently, this value is
258         not used.
259
260         \param[in] event The list of events to be enqueued.  This
261         container can and will be be empty in certain situations.  The
262         reference counts of the events in the container remains
263         unmodified.  The list of events become part of the event
264         queue.
265     */
266     virtual void enqueue(xxxx::Agent* agent, xxxx::EventContainer& events);
267
268     /** Dequeue all events sent by an agent after a given time.
269
270         This method implements the base class API method.  This method
271         can be used to remove/erase events sent by a given agent after
272         a given simulation time.  This API is needed to cancel events
273         during a rollback.  Next this method fixes the heap to ensure
274         that the agent with the least-time-stamp is at the top of the
275         heap.
276
277         \param[in] dest The agent whose currently scheduled events
278         are to be checked and cleaned-up.  This agent must be a valid
279         agent that has been registered/added to this event queue.  The
280         pointer cannot be NULL.  This parameter is not used.
281
282         \param[in] sender The ID of the agent whose events have to be
283         removed.  This agent must be a valid agent that has been
284         registered/added to this event queue.  The pointer cannot be
285         NULL.
286
287         \param[in] sentTime The time from which the events are to be
288         removed.  All events (including those sent at this time) sent
289         by the sender agent are removed from this event queue.
290
291         \return This method returns the number of events actually
292         removed.
293     */
294     virtual int eraseAfter(xxxx::Agent* dest, const xxxx::AgentID sender,
295                            const xxxx::Time sentTime);
296
297     /** Print full contents of scheduler queue to given output stream.
298
299         This is a convenience method that is used primarily for
300         troubleshooting purposes.  This method prints all the events
```

```
301         in this queue, with each event on its own line.
302
303         \param[out] os The output stream to which the contents of the
304         queue are to be written.
305     */
306     virtual void prettyPrint(std::ostream& os) const;
307
308     /** Method to report aggregate statistics.
309
310         This method is invoked at the end of simulation after all
311         agents on this rank have been finalized.  This method can
312         report any aggregate statistics from the event
313         queue. Currently, this method does not have any additional
314         statistics to report.
315
316         \param[out] os The output stream to which the statistics are
317         to be written.
318     */
319     virtual void reportStats(std::ostream& os);
320
321 protected:
322     /** Enqueue a new event.
323
324         This method must be used to enqueue/add an event to this event
325         queue.  Once added the reference count on the event is
326         increased.  This method adds the event to the specified agent.
327         Next this method fixes the heap to ensure that the agent with
328         the least-time-stamp is at the top of the heap.
329
330         \param[in] agent The agent to which the event is to be
331         scheduled.  This agent corresponds to the agent ID returned by
332         event->getReceiverAgentID() method.
333
334         \param[in] event The event to be enqueued.  This parameter can
335         never be NULL.
336     */
337     virtual void enqueueEvent(xxxx::Agent* agent, xxxx::Event* event);
338
339     /** Convenience method to remove events.
340
341         This is an internal convenience method that is used to remove
342         the front (i.e., events with lowest timestamp) event list from this
343         queue.
344     */
345     void pop_front(xxxx::Agent* agent);
346
347     /** Convenience method to obtain the top-most or front agent.
348
349         This method can be used to obtain a pointer to the top/front
350         agent -- that is, the agent with the lowest timestamp event to
351         be scheduled next.
352
353         \return A pointer to the top-most agent in this heap.
354     */
355     inline xxxx::Agent* top() {
356         return agentList.front();
357     }
358
359     /** Convenience method to get the top-event time for a given
360         agent.
361
362         This method returns the top event time in the vector of events queue.
363         If agent's vector of event queue is empty, then it returns infinity.
364
365         \return The receive time of top event's recv time or
366         TIME_INFINITY if vector is empty.
367     */
368     inline xxxx::Time getTopTime(const xxxx::Agent* const agent) const {
369         return agent->tier2->empty() ? TIME_INFINITY :
370             agent->tier2->front()->getReceiveTime();
371     }
372
373     /** Comparator method to sort events in the heap.
374
375         This is the comparator method that is passed to various
```



```
376         standard C++ algorithms to organize events as a heap.  This
377         comparator method gives first preference to receive time of
378         events.  Tie between two events with the same recieve time is
379         broken based on the receiver agent ID.
380
381         \param[in] lhs The left-hand-side event to be used for
382         comparison.  This parameter cannot be NULL.
383
384         \param[in] rhs The right-hand-side event to be used for
385         comparison. This parameter cannot be NULL.
386
387         \return This method returns if lhs < rhs, i.e., the lhs event
388         should be scheduled before the rhs event.
389     */
390     inline bool compare(const Agent *lhs, const Agent * rhs) const {
391         return getTopTime(lhs) >= getTopTime(rhs);
392     }
393
394     /** Comparator method to sort events in the heap.
395
396         This is the comparator method that is passed to various
397         standard C++ algorithms to organize events as a heap.  This
398         comparator method gives first preference to receive time of
399         events.  Tie between two events with the same recieve time is
400         broken based on the receiver agent ID.
401
402         \param[in] lhs The left-hand-side event to be used for
403         comparison.  This parameter cannot be NULL.
404
405         \param[in] rhs The right-hand-side event to be used for
406         comparison. This parameter cannot be NULL.
407
408         \return This method returns if lhs < rhs, i.e., the lhs event
409         should be scheduled before the rhs event.
410     */
411     inline static bool lessThan(const HOETier2Entry& lhs,
412                                 const xxxx::Event* const event) {
413         return (lhs.getReceiveTime() < event->getReceiveTime());
414     }
415
416     /** Comparator method to sort events in the heap.
417
418         This is the comparator method that is passed to various
419         standard C++ algorithms to organize events as a heap.  This
420         comparator method gives first preference to receive time of
421         events.  Tie between two events with the same recieve time is
422         broken based on the receiver agent ID.
423
424         \param[in] lhs The left-hand-side event to be used for
425         comparison.  This parameter cannot be NULL.
426
427         \param[in] rhs The right-hand-side event to be used for
428         comparison. This parameter cannot be NULL.
429
430         \return This method returns if lhs < rhs, i.e., the lhs event
431         should be scheduled before the rhs event.
432     */
433     inline static bool lessThanPtr(const HOETier2Entry* const lhs,
434                                    const xxxx::Event* const event) {
435         return (lhs->getReceiveTime() < event->getReceiveTime());
436     }
437
438     /** The getNextEvents method.
439
440         This method is a helper that will grab the next set of events
441         to be processed for a given agent.  This method is invoked in
442         dequeueNextAgentEvents() method in this class.
443
444         \param[out] container The reference of the container into
445         which events should be added.
446     */
447     void getNextEvents(Agent* agent, EventContainer& container);
448
449         /** Obtain the current index of the agent from it's
450         cross-reference.
```

```
451
452         This method is a refactored utility method that has been
453         introduced to streamline the code.  This method essentially
454         obtains the index position of the given agent in the agentList
455         vector from the agent's fibHeapPtr corss-reference.  This
456         cross-reference is consistently updated by the various methods
457         in this class to enable rapid access to the location of the
458         agent.
459
460         \param[in] agent The agent whose index value in the agentList
461         is to be determined.
462
463         \return The index position of the agent in the agentList
464         vector (if all checks pass).
465     */
466     size_t getIndex(xxxx::Agent *agent) const;
467
468     /** Update position of agent in the scheduler's heap.
469
470         This is an internal helper method that is used to update the
471         position of an agent in the scheduler's heap.  This method
472         essentially performs sanity checks, uses the fixHeap() method
473         to update position of the agent, and updates cross references
474         for future use.
475
476         \param[in] agent The agent whose position in the heap is to be
477         updated.  This pointer cannot be NULL.
478
479         \return This method returns the updated index position of the
480         agent in agentList (the vector that serves as storage for the
481         heap).
482     */
483     size_t updateHeap(xxxx::Agent* agent);
484
485     /** Fix-up the location of the agent in the heap.
486
487         This method can be used to update the location of an agent in
488         the heap.
489
490         \note The implementation for this method has been heavily
491         borrowed from libstdc++'s code base to ensure that heap
492         updates are consistent with std::make_heap API.
493         Unfortunately, this does imply that there is a chance this
494         method may be incompatible with future versions.
495
496         \param[in] currPos The current position of the agent in the
497         heap whose position is to be updated.  This value is the index
498         position of the agent in the agentList vector.
499
500         \return This method returns the new position of the agent in
501         the agentList vector.
502     */
503     size_t fixHeap(size_t currPos);
504
505     /** Convenience method to determine if an event is a future event.
506
507         This method is a helper method used in the eraseAfter() method
508         to determine if a given event is a future event from a given
509         sender agent.
510
511         \param[in] sender The sender agent to be used in comparison.
512
513         \param[in] sendTime The reference time for comparison
514
515         \param[in] evt The event to be checked if it is future event.
516
517         \return This method returns true if the event is sent from a
518         given sender agent and its send time is greater-or-equal to
519         the given sentTime.
520      */
521     inline bool
522     isFutureEvent(const xxxx::AgentID sender, const xxxx::Time sentTime,
523                   const xxxx::Event* evt) const {
524         return ((evt->getSenderAgentID() == sender) &&
525                 (evt->getSentTime() >= sentTime));
```



```
526     }
527
528     /** Helper method to reuse tier2 entries or create a new one.
529
530         This method is a convenience method to recycle tier2 entry
531         object is available.  If the recycle bin is empty, then this
532         method creates a new object.
533
534         \param[in] event The event to be used to initialize and to be
535         contained in the newly created tier2 entry.
536
537         \return A tier2 entry initialized and containing the given
538         event.
539      */
540     HOETier2Entry* makeTier2Entry(xxxx::Event* event) {
541         if (!tier2Recycler.empty()) {
542             HOETier2Entry* entry = tier2Recycler.back();
543             tier2Recycler.pop_back();
544             entry->reset(event);
545             return entry;
546         }
547         return new HOETier2Entry(event);
548     }
549
550 private:
551     /** The backing storage for events managed by this class.
552
553         This vector contains the list of agents being managed by the
554         class.  The agents in the vector are stored and maintained as
555         a heap.  The heap is created and managed using standard C++
556         algorithms, namely: \c std::make_heap, \c std::push_heap, and
557         \c std::pop_heap.
558     */
559     std::vector<xxxx::Agent*> agentList;
560
561     /** Stats object to track the average tier-2 bucket size.  This
562         value is the one that primarily determines if tier-2
563         operations are going to be optimal or not.  Higher this value,
564         the better for this event queue.
565     */
566     Avg avgSchedBktSize;
567
568     /** The number of times the fixHeap method performed heap-fixing
569         operations.  This variable is fine-grained in that it
570         accumulates the average number of compares that occur to fix
571         up the heap of agents.  The fixHeap method has a q1 = O(log
572         n1) compares.  So if this method is called m times, the
573         statistics reports (q1 + q2 + ... + qm) / m.
574     */
575     Avg fixHeapSwapCount;
576
577     /** The average queue size for each agent.  This value determines
578         the time takes to find a bucket into which an event is to be
579         inserted.
580     */
581     Avg agentBktCount;
582
583     /** A stack to recycle Tier2 entries to minimize memory allocation
584         calls for these small blocks used in this queue.
585     */
586     std::deque<HOETier2Entry*> tier2Recycler;
587 };
588
589 END_NAMESPACE(xxxx)
590
591 #endif
```

```cpp
#ifndef THREE_TIER_HEAP_EVENT_QUEUE_CPP
#define THREE_TIER_HEAP_EVENT_QUEUE_CPP

#include "ThreeTierHeapEventQueue.h"
#include <algorithm>

BEGIN_NAMESPACE(xxxx)

// A convenience shortcut used just in this source file
using Tier2List = std::deque<HOETier2Entry*>;
// using Tier2List = std::vector<Tier2Entry>;

ThreeTierHeapEventQueue::ThreeTierHeapEventQueue() :
    EventQueue("HeapOfVectorsEventQueue") {
    // Nothing else to be done.
}

ThreeTierHeapEventQueue::~ThreeTierHeapEventQueue() {
    // Clear up memory allocated for HOETier2Entry
    for (HOETier2Entry* entry : tier2Recycler) {
        delete entry;
    }
}

void*
ThreeTierHeapEventQueue::addAgent(xxxx::Agent* agent) {
    agentList.push_back(agent);
    // Create the vector that is used to manage events for the agent.
    agent->tier2 = new Tier2List();
    return reinterpret_cast<void*>(agentList.size() - 1);
}

void
ThreeTierHeapEventQueue::removeAgent(xxxx::Agent* agent) {
    ASSERT( agent != NULL );
    ASSERT(!empty());
    // Decrease reference count for all events in the agent event queue
    // before agent removal.
    ASSERT( agent->tier2 != NULL );
    // Logically remove events in this agent's tier2 queues/buckets
    Tier2List& tier2eventPQ = *agent->tier2;
    for (xxxx::HOETier2Entry* bucket : tier2eventPQ) {
        for (Event* evt : bucket->getEventList()) {
            evt->decreaseReference();  // logically remove event
        }
        // Free the memory reserved for this bucket
        delete bucket;
    }
    // Clear out tier2 queue (so this agent's time becomes PINFINITY)
    agent->tier2->clear();
    // Update the heap to place agent with LTSF
    updateHeap(agent);
}

xxxx::Event*
ThreeTierHeapEventQueue::front() {
    return (!top()->tier2->empty()) ? top()->tier2->front()->getEvent() : NULL;
}

void
ThreeTierHeapEventQueue::pop_front(xxxx::Agent* agent) {
    // Decrease reference count for all events in the front of the
    // agent event queue before the list of events is removed from the
    // event queue.
    std::vector<xxxx::Event*>& eventList =
        agent->tier2->front()->getEventList();
    for (Event* evt: eventList) {
        evt->decreaseReference();
    }
    // agent->tier2->erase(agent->tier2->begin());
    tier2Recycler.emplace_back(agent->tier2->front());
    agent->tier2->pop_front();
}

void
```



```cpp
ThreeTierHeapEventQueue::getNextEvents(Agent* agent,
                                       EventContainer& container) {
    ASSERT(container.empty());
    ASSERT(agent->tier2 != NULL);
    Tier2List& tier2 = *agent->tier2;
    ASSERT(tier2.front()->getEvent() != NULL);
    // All events in tier2 front should have same receive times
    const xxxx::Time eventTime = tier2.front()->getReceiveTime();
    // Copy all the events out of the tier2 front into the return contianer
    // container = std::move(agent->tier2->front().getEventList());
    std::vector<xxxx::Event*>& evtList = tier2.front()->getEventList();
    container.assign(evtList.begin(), evtList.end());
    DEBUG({
            // Do validation checks on the events in tier2
            for (const Event* event : container) {
                //  All events must have the same receive time
                ASSERT( event->getReceiveTime() == eventTime );

                // We should never process an anti-message.
                if (event->isAntiMessage()) {
                    std::cerr << "Anti-message Processing: " << *event
                              << std::endl;
                    std::cerr << "Trying to process an anti-message event, "
                              << "please notify XXXX developers of this issue"
                              << std::endl;
                    abort();
                }
                // Ensure that the top event is greater than LVT
                if (event->getReceiveTime() <= agent->getTime(Agent::LVT)) {
                    std::cerr << "Agent is being scheduled to process "
                              << "an event ("
                              << *event << ") that is at or below it LVT (LVT="
                              << agent->getTime(Agent::LVT) << ", GVT="
                              << agent->getTime(Agent::GVT)
                              << "). This is a serious error. Aborting.\n";
                    std::cerr << *agent << std::endl;
                    abort();
                }
                // Ensure reference counts are consistent.
                ASSERT(event->getReferenceCount() < 3);
                DEBUG(std::cout << "Delivering: " << *event << std::endl);
            }
        });
    // Recycle the entry at the beginning of the queue.
    tier2Recycler.emplace_back(tier2.front());
    tier2.pop_front();
    // std::rotate(tier2.begin(), tier2.begin() + 1, tier2.end());
    // tier2.pop_back();
    // Track bucket/block size statistics
    avgSchedBktSize += container.size();
}

void
ThreeTierHeapEventQueue::dequeueNextAgentEvents(xxxx::EventContainer& events) {
    if (!empty()) {
        // Get agent and validate.
        xxxx::Agent* const agent = top();
        ASSERT(agent != NULL);
        ASSERT(getIndex(agent) == 0);
        // Have the events give up its next set of events
        getNextEvents(agent, events);
        ASSERT(!events.empty());
        // Fix the position of this agent in the scheduler's heap.
        updateHeap(agent);
    }
}

void
ThreeTierHeapEventQueue::enqueue(xxxx::Agent* agent, xxxx::Event* event) {
    // Use helper method (just below this one) to add event and fix-up
    // the queue.  First Increase event reference count for every
    // event added to the event queue.
    ASSERT( event->getReferenceCount() < 2 );
    event->increaseReference();
    enqueueEvent(agent, event);
```

```
151     updateHeap(agent);
152 }
153
154 void
155 ThreeTierHeapEventQueue::enqueueEvent(xxxx::Agent* agent, xxxx::Event* event) {
156     ASSERT(agent != NULL);
157     ASSERT(event != NULL);
158     ASSERT( agent->tier2 != NULL );
159     ASSERT(getIndex(agent) < agentList.size());
160     // A convenience reference to tier2 list of buckets
161     Tier2List& tier2 = *agent->tier2;
162     // Use binary search O(log n) to find match or insert position
163     agentBktCount += tier2.size();
164     Tier2List::iterator iter =
165         std::lower_bound(tier2.begin(), tier2.end(), event, lessThanPtr);
166     // There are 3 cases: 1. we found matching bucket, 2: iterator
167     // to bucket with higher recvTime, or 3: tier2.end().
168     if (iter == tier2.end()) {
169         tier2.emplace_back(makeTier2Entry(event));  // add new entry to end.
170     } else if ((*iter)->getReceiveTime() == event->getReceiveTime()) {
171         // We found an existing bucket. Append this event to this
172         // existing bucket.
173         (*iter)->updateContainer(event);
174     } else {
175         // If there is no bucket with a matching receive time in Tier2
176         // vector, then insert an instance of HOETier2Entry (aka
177         // bucket) into the vector at the appropriate position.
178         ASSERT((*iter)->getReceiveTime() > event->getReceiveTime());
179         tier2.emplace(iter, makeTier2Entry(event));
180     }
181     // ASSERT(std::is_sorted(tier2.begin(), tier2.end()));
182 }
183
184 void
185 ThreeTierHeapEventQueue::enqueue(xxxx::Agent* agent,
186                                  xxxx::EventContainer& events) {
187     ASSERT(agent != NULL);
188     // Note: events container may be empty!
189     ASSERT(getIndex(agent) < agentList.size());
190     // Add all events to tier2 entries appropriately.
191     for (xxxx::Event* event : events) {
192         // Enqueue event but don't waste time fixing-up heap yet for
193         // this agent.  We will do it at the end after all events are
194         // added.  However, we don't increase reference counts in this
195         // API.
196         enqueueEvent(agent, event);
197     }
198     // Clear out all the events in the incoming container
199     events.clear();
200     // Update the location of this agent on the heap as needed.
201     updateHeap(agent);
202 }
203
204 int
205 ThreeTierHeapEventQueue::eraseAfter(xxxx::Agent* dest,
206                                     const xxxx::AgentID sender,
207                                     const xxxx::Time sentTime) {
208     int  numRemoved = 0;
209     ASSERT( dest->tier2 != NULL );
210     Tier2List& tier2eventPQ = *dest->tier2;
211     long currIdx = tier2eventPQ.size() - 1;
212     while (!tier2eventPQ.empty() && (currIdx >= 0)) {
213         if (tier2eventPQ[currIdx]->getReceiveTime() > sentTime) {
214             std::vector<xxxx::Event*>& eventList =
215                 tier2eventPQ[currIdx]->getEventList();
216             size_t index = 0;
217             while (!eventList.empty() && (index < eventList.size())) {
218                 Event* const evt = eventList[index];
219                 ASSERT(evt != NULL);
220                 if (isFutureEvent(sender, sentTime, evt)) {
221                     evt->decreaseReference();
222                     numRemoved++;
223                     eventList[index] = eventList.back();
224                     eventList.pop_back();
225                 } else {
```



```
226                     index++;  // onto next event in this bucket
227                 }
228             }
229             // If all events are canceled then this bucket needs to be
230             // removed from the tier2 entry.
231             if (eventList.empty()) {
232                 tier2Recycler.emplace_back(tier2eventPQ[currIdx]);
233                 tier2eventPQ.erase(tier2eventPQ.begin() + currIdx);
234             }
235         }
236         currIdx--;
237     }
238     // Update the 1st tier heap for scheduling.
239     updateHeap(dest);
240     // Return number of events canceled to track statistics.
241     return numRemoved;
242 }
243
244 void
245 ThreeTierHeapEventQueue::reportStats(std::ostream& os) {
246     UNUSED_PARAM(os);
247     const long comps = std::log2(agentList.size()) *
248         avgSchedBktSize.getCount() + fixHeapSwapCount.getSum();
249     os << "Average #buckets per agent  : " << agentBktCount    << std::endl;
250     os << "Average scheduled bucket size: " << avgSchedBktSize  << std::endl;
251     os << "Average fixHeap compares    : " << fixHeapSwapCount << std::endl;
252     os << "Compare estimate            : " << comps            << std::endl;
253 }
254
255 void
256 ThreeTierHeapEventQueue::prettyPrint(std::ostream& os) const {
257     os << "HeapOfVectorsEventQueue::prettyPrint() : not implemented.\n";
258 }
259
260 size_t
261 ThreeTierHeapEventQueue::getIndex(xxxx::Agent *agent) const {
262     ASSERT(agent != NULL);
263     size_t index = reinterpret_cast<size_t>(agent->fibHeapPtr);
264     ASSERT(index < agentList.size());
265     ASSERT(agentList[index] == agent);
266     return index;
267 }
268
269 size_t
270 ThreeTierHeapEventQueue::updateHeap(xxxx::Agent* agent) {
271     ASSERT(agent != NULL);
272     size_t index = getIndex(agent);
273     if (agent->oldTopTime != getTopTime(agent)) {
274         index = fixHeap(index);
275         // Update the position of the agent in the scheduler's heap
276         // Validate
277         ASSERT(agentList[index] == agent);
278         ASSERT(getIndex(agent) == index);
279         // Update time value as well for future access
280         agent->oldTopTime = getTopTime(agent);
281         // Validation check.
282         ASSERT(getTopTime(agentList[0]) <= getTopTime(agentList[1]));
283     }
284     // Return the new index position of the agent
285     return index;
286 }
287
288 size_t
289 ThreeTierHeapEventQueue::fixHeap(size_t currPos) {
290     ASSERT(currPos < agentList.size());
291     xxxx::Agent* value    = agentList[currPos];
292     const size_t len      = (agentList.size() - 1) / 2;
293     size_t secondChild    = currPos;
294     int          opCount  = 0;
295     // This code was borrowed from libstdc++ implementation to ensure
296     // that the fix-ups are consistent with std::make_heap API.
297     while (secondChild < len)  {
298         secondChild = 2 * (secondChild + 1);
299         if (compare(agentList[secondChild], agentList[secondChild - 1])) {
300             secondChild--;
```

```
301         }
302         agentList[currPos] = std::move(agentList[secondChild]);
303         agentList[currPos]->fibHeapPtr = reinterpret_cast<void*>(currPos);
304         currPos = secondChild;
305         opCount++;  // track statistics on number of operations performed
306     }
307     if (((agentList.size() & 1) == 0) &&
308         (secondChild == (agentList.size() - 2) / 2)) {
309         secondChild        = 2 * (secondChild + 1);
310         agentList[currPos] = std::move(agentList[secondChild - 1]);
311         agentList[currPos]->fibHeapPtr = reinterpret_cast<void*>(currPos);
312         currPos            = secondChild - 1;
313         opCount++;  // track statistics on number of operations performed
314     }
315     // Use libstdc++'s internal method to fix-up the vector from the
316     // given location.
317     // std::__push_heap(agentList.begin(), currPos, 0, value,
318     //                  __gnu_cxx::__ops::__iter_comp_val(compare));
319
320     size_t parent = (currPos - 1) / 2;
321     while ((currPos > 0) && (compare(agentList[parent],value))) {
322         agentList[currPos] = std::move(agentList[parent]);
323         agentList[currPos]->fibHeapPtr = reinterpret_cast<void*>(currPos);
324         currPos = parent;
325         parent  = (currPos - 1) / 2;
326         opCount++;  // track statistics on number of operations performed
327     }
328     agentList[currPos] = value;
329     agentList[currPos]->fibHeapPtr = reinterpret_cast<void*>(currPos);
330     // Update aggregate statistics
331     fixHeapSwapCount += opCount;
332     // Return the final index position for the agent
333     return currPos;
334 }
335
336 END_NAMESPACE(xxxx)
337
338 #endif
```

```
#ifndef LADDER_QUEUE_H
#define LADDER_QUEUE_H

#include <list>
#include <queue>
#include <vector>
#include <set>
#include "Avg.h"
#include "Event.h"
#include "EventQueue.h"

/** \file LadderQueue.h

    \brief Implementation for a LadderQueue.

    The LadderQueue.h and LadderQueue.cpp collectively provide
    implementation for a LadderQueue data structure.  The data
    structure is detailedin the following paper:

    W. Tang, R. Goh, and I. Thng, "Ladder queue: An O(1) priority
    queue structure for large-scale discrete event simulation", ACM
    TOMACS, Vol 15, Issue 3, Pages 175--204, July 2005. URL:
    http://doi.acm.org/10.1145/1103323.1103324
*/

// The threshold value (number of events in bottom) after which events
// from bottom are placed into a new rung in the ladder.
#define THRESH 50

/** \def LQ_STATS(x)

    \brief Define a convenient macro for conditionally compiling
    additional statistics collection regarding ladder queue.

    Define a custom macro LQ_STATS (note the all caps) macro to be used
    to conditionally compile in debugging code to generate detailed
    logs.  This helps to minimize code modification to insert and
    remove debugging messages.
*/
#define COMMA ,
#define LQ_STATS(x) x
// #define LQ_STATS(x)

BEGIN_NAMESPACE(xxxx)

// Defintion for a vector events
using EventVector = std::vector<xxxx::Event*>;

// The definition for a singly-linked list of Events
using EventList = std::list<xxxx::Event*>;

/** A generic bucket that is used for both Top and Rungs of ladderQ.

    <p>This bucket does not store events in it directly. Instead it
    provides a singly-linked list for storing Events.
    </p>
*/
class ListBucket {
public:
    /** Constructor to create a bucket with a singly-linked list of Events.
    */
    ListBucket() : count(0) {}

    /** A move constructor to facilitate moving objects (if needed).

        \param[in,out] src The source object whose data is to be moved
        into this.  The source object does not contain any useful
        information after the move is complete.
    */
    ListBucket(ListBucket&& src) : list(std::move(src.list)),
                                   count(std::move(src.count)) {
        // Reset count in source to aid debugging.
        src.count = 0;
    }

```



```
    /** The destructor for this class.

        The destructor decreases the reference count on all the events
        in its list to free-up any pending events.
    */
    ~ListBucket();

    // Definition of an iterator for the event list.
    using iterator = EventList::iterator;
    // Definition of a const iterator for the event list.
    using const_iterator = EventList::const_iterator;

    /** Convenience method to add events to the event list.

        This is an internal convenience method that is used to add
        events to the front of the event list.
    */
    void push_front(xxxx::Event* event) {
        list.push_front(event);
        count++;
    }

    /** Obtain pointer to the highest priority (lowest receive time)
        event.

        This method can be used to obtain a pointer to the highest
        priority event in this event list.

        \return A pointer to the next event to be processed.  If the
        list is empty then this method returns NULL.
    */
    xxxx::Event* front() const {
        return (!list.empty() ? list.front() : NULL);
    }

    /** Obtain pointer to the last event in the list.

        This method can be used to obtain a pointer to the last event
        in this event list.

        \return A pointer to the last event to be processed.  If the
        list is empty then this method returns NULL.
    */
    xxxx::Event* back() const {
        return (!list.empty() ? list.back() : NULL);
    }

    /** Convenience method to remove an event from the event list.

        This is an internal convenience method that is used to remove
        the front (i.e., events with lowest timestamp) event from this
        list.

        \return A pointer to the highest priority event in this event list.
    */
    xxxx::Event* pop_front() {
        xxxx::Event* retVal = list.front();
        list.pop_front();
        count--;
        return retVal;
    }

    /** Convenience method to insert an event into the event list.

        This is an internal convenience method that is used to insert
        an event at a specified location in the event list.
    */
    void insert_after(ListBucket::iterator pos, xxxx::Event* event) {
        list.insert(++pos, event);
        count++;
    }

    /** Obtain the count of events.

        \return The sum of events in the singly-linked list.
```

```
151         */
152         size_t size() const { return count; }
153
154         /** Determine if the event list is empty.
155
156             \return This method returns true if the event list of the
157             bucket is logically empty.
158         */
159         bool empty() const { return list.empty(); }
160
161         /** Obtain the iterator to the beginning of the list.
162
163             \return The iterator to the first element in the list.
164         */
165         ListBucket::iterator begin() { return list.begin(); }
166
167         /** Obtain the iterator to the beginning of the list.
168
169             \return The const iterator to the first element in the list.
170         */
171         ListBucket::const_iterator cbegin() { return list.cbegin(); }
172
173         /** Obtain the iterator to the end of the list.
174
175             \return The iterator to the last element in the list.
176         */
177         ListBucket::iterator end() { return list.end(); }
178
179         /** Obtain the iterator to the end of the list.
180
181             \return The const iterator to the last element in the list.
182         */
183         ListBucket::const_iterator cend() { return list.cend(); }
184
185         /** Convenience method to remove all events sent by the sender
186             at-or-after the given send Time.
187
188             This method will compare the timestamps of all events in the
189             list with that of the specified event. Any Event (from the
190             same sender) that was sent at a time greater than or equal to
191             that of the specified event will be deleted.
192
193             \param[in] sender The sender agent whose events are to be
194             removed.
195
196             \param[in] sendTime The time at-or-after which events from the
197             sender are to be removed from the given list.
198
199             \return This method returns the number of events that were
200             removed.
201         */
202         int remove_after(xxxx::AgentID sender, const Time sendTime);
203
204         /** Convenience method to remove all events sent by the sender
205             at-or-after the given send Time.
206
207             This method will compare the timestamps of all events in the
208             list with that of the specified event. Any Event (from the
209             same sender) that was sent at a time greater than or equal to
210             that of the specified event will be deleted.
211
212             \param[in] sender The sender agent whose events are to be
213             removed.
214
215             \param[in] sendTime The time at-or-after which events from the
216             sender are to be removed from the list.
217
218             \return This method returns the number of events that were
219             removed.
220         */
221         int remove_after_sorted(xxxx::AgentID sender, const Time sendTime) {
222             // No difference between sorted and unsorted version for ListBucket
223             return remove_after(sender, sendTime);
224         }
225
```



```
226         /** Remove all events for a given receiver agent ID.
227
228             This is a convenience method that removes all events for a
229             given receiver agent.  This method is used to remove events
230             scheduled for an agent, when an agent is removed from the
231             scheduler.
232         */
233         int remove(xxxx::AgentID receiver);
234
235         // The method below is purely for troubleshooting one scenario
236         // where an event would get stuck in the ladder and not get
237         // scheduled correctly.
238         bool haveBefore(const Time recvTime) const;
239
240     private:
241
242         /** The singly-linked list of events.
243         */
244         EventList list;
245
246         /** The total number of events in the singly-linked list of events. This
247             information is primary used to quickly respond to the size()
248             method calls.
249         */
250         size_t count;
251     };
252
253     /** A generic bucket that is used for both Top and Rungs of ladderQ.
254
255         <p>This bucket does not store events in it directly. Instead it
256         provides a vector for storing Events.
257         </p>
258     */
259     class VectorBucket {
260     public:
261         /** Constructor to create a bucket with a vector container for holding
262             Events.
263         */
264         VectorBucket() : count(0) {}
265
266         /** A move constructor to facilitate moving objects (if needed).
267
268             \param[in,out] src The source object whose data is to be moved
269             into this.  The source object does not contain any useful
270             information after the move is complete.
271         */
272         VectorBucket(VectorBucket&& src) : list(std::move(src.list)),
273                                            count(std::move(src.count)) {
274             // Reset count in source to aid debugging.
275             src.count = 0;
276         }
277
278         /** The destructor for this class.
279
280             The destructor decreases the reference count on all the events
281             in the vector of events to free-up any pending events.
282         */
283         ~VectorBucket();
284
285         // Definition of an iterator for the vector containing events.
286         using iterator = EventVector::iterator;
287         // Definition of a const iterator for the vector containing events.
288         using const_iterator = EventVector::const_iterator;
289         // Definition of a reverse iterator for the vector containing events.
290         using reverse_iterator = EventVector::reverse_iterator;
291
292         /** Convenience method to add events to the vector of events.
293
294             This is an internal convenience method that is used to add
295             events to the vector of events.
296         */
297         void push_front(xxxx::Event* event) {
298             list.push_back(event);
299             count++;
300         }
```

```
301
302     /** Obtain pointer to the highest priority (lowest receive time)
303         event.
304
305         This method can be used to obtain a pointer to the highest
306         priority event in the vector of events.
307
308         \return A pointer to the next event to be processed.  If the
309         vector is empty then this method returns NULL.
310     */
311     xxxx::Event* front() const {
312         return (!list.empty() ? list.back() : NULL);
313     }
314
315     /** Obtain pointer to the last event in the vector .
316
317         This method can be used to obtain a pointer to the last event
318         in this vector of events.
319
320         \return A pointer to the last event to be processed.  If the
321         vector is empty then this method returns NULL.
322     */
323     xxxx::Event* back() const {
324         return (!list.empty() ? list.front() : NULL);
325     }
326
327     /** Convenience method to remove an event from the vector of events.
328
329         This is an internal convenience method that is used to remove
330         the front (i.e., events with lowest timestamp) event from the
331         vector of events.
332
333         \return A pointer to the highest priority event in the vector of events.
334     */
335     xxxx::Event* pop_front() {
336         xxxx::Event* retVal = list.back();
337         list.pop_back();
338         count--;
339         return retVal;
340     }
341
342     /** Convenience method to add events to the vector of events.
343
344         This is an internal convenience method that is used to add
345         events from a bucket to the vector of events.
346     */
347     void push_back(VectorBucket&& bucket);
348
349     /** Convenience method to insert an event into the vector of events.
350
351         This is an internal convenience method that is used to insert
352         an event at a specified location in the vector of events.
353     */
354     void insert_after(VectorBucket::iterator pos, xxxx::Event* event) {
355         list.insert(pos + 1, event);
356         count++;
357     }
358
359     /** Convenience method to insert an event into the vector of events.
360
361         This is an internal convenience method that is used to insert
362         an event at a specified location in the vector of events.
363     */
364     void insert_after(VectorBucket::reverse_iterator pos, xxxx::Event* event) {
365         const size_t idx = list.size() - (pos - rbegin());
366         list.insert(list.begin() + idx, event);
367         count++;
368     }
369
370     /** Obtain the count of events.
371
372         \return The sum of events in the vector of events.
373     */
374     size_t size() const { return count; }
375
```



```
376     /** Determine if the vector of events is empty.
377
378         \return This method returns true if the vector of events of the
379         bucket is logically empty.
380     */
381     bool empty() const { return list.empty(); }
382
383     /** Obtain the iterator to the beginning of the vector of events.
384
385         \return The iterator to the first element in the vector of events.
386     */
387     VectorBucket::iterator begin() { return list.begin(); }
388
389     /** Obtain the iterator to the beginning of the vector of events.
390
391         \return The iterator to the first element in the vector of events.
392     */
393     VectorBucket::const_iterator cbegin() const { return list.cbegin(); }
394
395     /** Obtain the iterator to the beginning of the vector of events.
396
397         \return The reverse iterator to the beginning element in the
398          vector of events.
399     */
400     VectorBucket::reverse_iterator rbegin() { return list.rbegin(); }
401
402     /** Obtain the iterator to the end of the vector of events.
403
404         \return The iterator to the last element in the vector of events.
405     */
406     VectorBucket::iterator end() { return list.end(); }
407
408     /** Obtain the iterator to the end of the vector of events.
409
410         \return The const iterator to the last element in the vector of events.
411     */
412     VectorBucket::const_iterator cend() const { return list.cend(); }
413
414     /** Obtain the iterator to the end of the vector of events.
415
416         \return The reverse iterator to the last element in the vector of events
    .
417     */
418     VectorBucket::reverse_iterator rend() { return list.rend(); }
419
420     /** Convenience method to remove all events sent by the sender
421         at-or-after the given send Time.
422
423         This method will compare the timestamps of all events in the
424         list with that of the specified event. Any Event (from the
425         same sender) that was sent at a time greater than or equal to
426         that of the specified event will be deleted.
427
428         \param[in] sender The sender agent whose events are to be
429         removed.
430
431         \param[in] sendTime The time at-or-after which events from the
432         sender are to be removed from the vector.
433
434         \return This method returns the number of events that were
435         removed.
436     */
437     int remove_after(xxxx::AgentID sender, const Time sendTime);
438
439     /** Convenience method to remove all events sent by the sender
440         at-or-after the given send Time.
441
442         This method will compare the timestamps of all events in the
443         list with that of the specified event. Any Event (from the
444         same sender) that was sent at a time greater than or equal to
445         that of the specified event will be deleted.
446
447         \note This method assumes a sorted vector of events and shortcircuit
448         scans the vector if last event's time is less-or-equal to sendTime.
449
```

```
450         \param[in] sender The sender agent whose events are to be
451         removed.
452
453         \param[in] sendTime The time at-or-after which events from the
454         sender are to be removed from the given list.
455
456         \return This method returns the number of events that were
457         removed.
458     */
459     int remove_after_sorted(xxxx::AgentID sender, const Time sendTime);
460
461     /** Remove all events in this vector bucket for a given receiver
462         agent ID.
463
464         This is a convenience method that removes all events for a
465         given receiver agent in this bucket.  This method is used to
466         remove events scheduled for an agent, when an agent is removed
467         from the scheduler.
468     */
469     int remove(xxxx::AgentID receiver);
470
471     // The method below is purely for troubleshooting one scenario
472     // where an event would get stuck in the ladder and not get
473     // scheduled correctly.
474     bool haveBefore(const Time recvTime) const;
475
476 private:
477     /** The vector of events.
478      */
479     EventVector list;
480
481     /** The total number of events in the vector of events. This
482         information is primary used to quickly respond to the size()
483         method calls.
484     */
485     size_t count;
486 };
487
488 // using Bucket = ListBucket;
489 using Bucket = VectorBucket;
490
491 /** The class that forms the Top rung of a ladder queue.
492
493     The top-rung of the ladder queue stores events in a VectorBucket that is
494     backed by a vector.
495
496     \note Do not call push_back directly. Instead use the add method
497     in this class to add events.
498 */
499 class Top {
500     friend class Rung;
501     friend class LadderQueue;
502 public:
503     /** Construct and initialize top to empty state.
504
505         The constructor uses a convenience method in this class to
506         reset the timestamps to zero.
507     */
508     Top() { reset(); }
509
510     /** The destructor
511
512         Currently the destructor has nothing to do.
513      */
514     ~Top();
515
516     /** Convenience method to add events to the top-rung.
517
518         This is an internal convenience method that is used to add
519         events to the top-rung by calling push_front method of VectorBucket
520         and adding an event to the bucket's vector of events.
521     */
522     void add(xxxx::Event* event);
523
524     /** Determine if the top-rung is empty.
```



```
525
526         \return This method returns true if the bucket of events is
527         logically empty.
528     */
529     bool empty() const { return events.empty(); }
530
531     /** Return the current start-time for top.
532
533         \note This value changes when events are added/removed. So
534         don't think about caching this value.
535
536         \return The current start time. This value is used for
537         scheduling events and creating rungs.
538     */
539     Time getStartTime() const { return topStart; }
540
541     /** Returns the minimum timestamp of events in this rung.
542
543         \note This value changes when events are added/removed. So
544         don't think about caching this value.
545
546         \return The minimum timestamp of events in this rung.
547     */
548     Time getMinTime() const { return minTS; }
549
550     /** Returns the maximum timestamp of events in this rung.
551
552         \note This value changes when events are added/removed. So
553         don't think about caching this value.
554
555         \return The maximum event timestamp in this rung.
556     */
557     Time getMaxTime() const { return maxTS; }
558
559     /** Convenience method to determine if given time is within the
560         <i>current</i> minmum and maximum time.
561
562         \param[in] ts The timestamp value to be checked.
563
564         \return This method returns true if minTS <= ts <= maxTS.
565         Otherwise it returns false.
566     */
567     bool contains(const Time ts) const {
568         return (ts >= minTS) && (ts <= maxTS);
569     }
570
571     /** Convenience method compute the bucket size for the top-level
572         rung of the Ladder queue.
573
574         \return The suggested bucket width (in terms of time) for the
575         top-level rung of the Ladder queue.
576     */
577     double getBucketWidth() const {
578         DEBUG(std::cout << "minTS=" << minTS << ", maxTS=" << maxTS
579                         << ", size=" << size() << std::endl);
580         return std::max((maxTS - minTS + size() - 1.0) / size(), 0.01);
581     }
582
583     /** Obtain the count of events.
584
585         \return The sum of events in the top-rung of the Ladder queue.
586     */
587     int size() const { return events.size(); }
588
589     /** Convenience method to remove all events sent by the sender
590         at-or-after the given send Time.
591
592         This method will compare the timestamps of all events in the
593         list with that of the specified event. Any Event (from the
594         same sender) that was sent at a time greater than or equal to
595         that of the specified event will be deleted.
596
597         \param[in] sender The sender agent whose events are to be
598         removed.
599
```

```
600         \param[in] sendTime The time at-or-after which events from the
601         sender are to be removed from the vector.
602
603         \return This method returns the number of events that were
604         removed.
605     */
606     int remove_after(xxxx::AgentID sender, const Time sendTime);
607
608     /** Remove all events for a given receiver agent in the bucket
609         encapsulated by this object.
610
611         This is a convenience method that removes all events for a
612         given receiver agent in this object.  This method is used to
613         remove events scheduled for an agent, when an agent is removed
614         from the scheduler.
615     */
616     int remove(xxxx::AgentID receiver);
617
618     // The method below is purely for troubleshooting one scenario
619     // where an event would get stuck in the ladder and not get
620     // scheduled correctly.
621     bool haveBefore(const Time recvTime) const {
622         return events.haveBefore(recvTime);
623     }
624
625 protected:
626     /** Helper method to reset top either during construction or
627         whenever it is emptied to move events into the ladder.
628
629         \param[in] topStart An optional start time for the top rung.
630     */
631     void reset(const Time topStart = 0);
632
633 private:
634     /** Instsance variable to track the current minimum timestamp of
635         events in top.  This value changes each time a new event is
636         added to the top via the add emthod.
637     */
638     xxxx::Time minTS;
639
640     /** Instance variable to track the current maximum timestamp of
641         events in top.  This value changes each time a new event is
642         added to the top via the add method.
643     */
644     xxxx::Time maxTS;
645
646     /** Instance variable to track the last time top was reset.  This
647         is used for debugging/troubleshooting purposes.
648     */
649     xxxx::Time topStart;
650
651     /**The VectorBucket backed by a vector that stores events.
652      */
653     Bucket events;
654 };
655
656 /** The bottom most rung of the Ladder queue.  The bottom rung
657     is the same as that of the standard ladder queue.  However, in this
658     implementation of ladder queue, the size of the bottom has been relaxed.
659     So bottom can be pretty long.  This implies ladder queue will not be O(1).
660     It will be O(n log n). However, it should perform just fine as the ladder
661     queue.
662
663     \note In XXXX we have an API requirement/guarantee that all the
664     concurrent events we have will be scheduled simultaneously.  This
665     eases agent development in many applications. Consequently, it is
666     imperative that bottom be allowed to be long to contain all
667     concurrent events.
668
669 */
670 class Bottom {
671     friend class LadderQueue;   // NOTE: uses sel directly
672 public:
673
674     /** Add events from a Bucket (VectorBucket or ListBucket) into the bottom.
```



```
675
676         This method is used to bulk move events from a rung of the
677         ladder (or top) into the bottom.  The events are added and
678         sorted in preparation for scheduling.
679
680         \param bucket The bucket from where events are to be
681         moved into the bottom rung.
682     */
683     void enqueue(Bucket&& bucket) {
684         // Delegate to overloaded method to handle different scenarios
685         // depending on whether linked-list or vector is used for Buckets
686         enqueue(std::move(bucket), sel);
687     }
688
689     /** Add a single event into the bottom.
690
691         This method is used to add a single event from a rung of the
692         ladder (or top) into the bottom. The event is added into a sorted list
693         or vector of events in preparation for scheduling.
694
695         \param Event The event to be added into the bottom rung.
696     */
697     void enqueue(xxxx::Event* event) {
698         // Delegate to overloaded method to handle different scenarios
699         // depending on whether linked-list or vector is used for Buckets
700         enqueue(event, sel);
701     }
702
703     /** Convenience method to remove an event from the vector or list of events.
704
705         This is an internal convenience method that is used to remove
706         the front (i.e., events with lowest timestamp) event
707         depending on whether linked-list or vector is used for Buckets
708
709         \return A pointer to the highest priority event in the vector of events.
710     */
711     xxxx::Event* pop_front() { return sel.pop_front(); }
712
713     /** Obtain pointer to the highest priority (lowest receive time)
714         event.
715
716         This method can be used to obtain a pointer to the highest
717         priority event in the vector or list of events.
718
719         \return A pointer to the next event to be processed.  If the
720         vector is empty then this method returns NULL.
721     */
722     xxxx::Event* front() const { return sel.front(); }
723
724     /** Determines if Bottom is empty.
725
726         \return This method returns true if the bucket of events is
727         logically empty.
728     */
729     bool empty() const {
730         return sel.empty();
731     }
732
733     /** Determine bucket width to move bottom into ladder.
734
735         This method is invoked only when the ladder is empty and the
736         bottom is long and needs to be moved into the ladder.  This
737         method must compute and return the preferred bucket width.
738
739         \note If the bottom is empty this method returns bucket width
740         of 0.
741      */
742     double getBucketWidth() const;
743
744     /** Convenience method to remove all events sent by the sender
745         at-or-after the given send Time.
746
747         This method will compare the timestamps of all events in the
748         list with that of the specified event. Any Event (from the
749         same sender) that was sent at a time greater than or equal to
```

```
750         that of the specified event will be deleted.
751
752         \note When a vector is used for bucket, this method assumes a sorted
753         vector of events and shortcircuit scans the vector if last event's
754         time is less-or-equal to sendTime.
755
756         \param[in] sender The sender agent whose events are to be
757         removed.
758
759         \param[in] sendTime The time at-or-after which events from the
760         sender are to be removed from the vector.
761
762         \return This method returns the number of events that were
763         removed.
764     */
765     int remove_after(xxxx::AgentID sender, const Time sendTime) {
766         return sel.remove_after_sorted(sender, sendTime);
767     }
768
769     /** Remove all events for a given receiver agent in the bucket
770         encapsulated by this object.
771
772         This is a convenience method that removes all events for a
773         given receiver agent in this object.  This method is used to
774         remove events scheduled for an agent, when an agent is removed
775         from the scheduler.
776     */
777     int remove(xxxx::AgentID receiver);
778
779     /** Convenience method used to dequeue the next set of events for
780         scheduling.
781
782         This method is used to provide necessary implementation to
783         interface with the XXXX scheduler.  This method dequeues the
784         next batch of the concurrent events for processing by a given
785         agent.
786
787         \param[out] events The container to which all the events to be
788         processed is to be added.
789     */
790     void dequeueNextAgentEvents(xxxx::EventContainer& events);
791
792     /** Convenience method for debugging/troubleshooting.
793
794         \return The highest timestamp from the events in the bottom.
795         If no events are present this method returns TIME_INFINITY.
796     */
797     xxxx::Time maxTime() const;
798
799     /** Convenience method for debugging/troubleshooting.
800
801         \return The minimum timestamp from the events in the bottom.
802         If no events are present this method returns TIME_INFINITY.
803     */
804     xxxx::Time findMinTime() const;
805
806     /** Convenience method to check if the entries in the bottom are
807         sorted correctly. This method is purely used for
808         troubleshooting/debugging.
809     */
810     void validate();
811
812     /** Obtain the count of events.
813
814         \return The sum of events in the bottom of the Ladder queue.
815     */
816     inline size_t size() const { return sel.size(); }
817
818     /** Event comparison function used by various structures in ladder
819         queue.
820
821         \return This method returns true if lhs is greater than rhs.
822         That is, lhs should be scheduled after rhs.
823      */
824     static inline bool compare(const xxxx::Event* const lhs,
```



```
825                                const xxxx::Event* const rhs) {
826         return ((lhs->getReceiveTime() > rhs->getReceiveTime()) ||
827                 ((lhs->getReceiveTime() == rhs->getReceiveTime() &&
828                   (lhs->getReceiverAgentID() > rhs->getReceiverAgentID()))));
829     }
830
831     /** Event comparison function used by various structures in ladder
832         queue. This comparison reverses the sort order -- it places
833         lowest timestamp towards end of the vector.  This make it more
834         efficient for popping element off the end of the vector using
835         the pop_front() method in this class.
836
837         \return This method returns true if lhs is greater than rhs.
838         That is, lhs should be scheduled after rhs.
839      */
840     static inline bool revCompare(const xxxx::Event* const lhs,
841                                   const xxxx::Event* const rhs) {
842         return compare(rhs, lhs);
843     }
844
845     // The method below is purely for troubleshooting one scenario
846     // where an event would get stuck in the ladder and not get
847     // scheduled correctly.
848     bool haveBefore(const Time recvTime) const {
849         return sel.haveBefore(recvTime);
850     }
851
852     /** Method to determine the range of receive time values currently
853         in bottom.  This value is used to decide if it is worth moving
854         events from bottom into the ladder.
855
856         \return The difference in maximum and minimum receive
857         timestamp of events in the bottom.  This value is zero if all
858         events have the same receive time.  If the bottom is empty,
859         then this method also returns zero.
860     */
861     xxxx::Time getTimeRange() const {
862         if (sel.empty()) {
863             return 0;
864         }
865         return (sel.back()->getReceiveTime() - sel.front()->getReceiveTime());
866     }
867
868 protected:
869     // Two different strategies based on the type of bucket used -->
870     // linked-list vs. vector
871     void enqueue(ListBucket&& bucket, ListBucket& botList);
872     void enqueue(VectorBucket&& bucket, VectorBucket& botList);
873
874     // Two different strategies based on the type of bucket used -->
875     // linked-list vs. vector
876     void enqueue(xxxx::Event* event, ListBucket& botList);
877     void enqueue(xxxx::Event* event, VectorBucket& botList);
878
879 private:
880     // Sorted Event List (SEL) for the bottom
881     Bucket sel;
882 };
883
884 /** An alternative implementation for Bottom that uses a binary heap.
885     The objective of having multiple implementations for Bottom is to identify
886     the best data structure for the type of operations that are performed on
887     Bottom in both sequential and TimeWarp simulations.
888 */
889 class HeapBottom {
890     friend class LadderQueue;   // NOTE: uses sel directly
891 public:
892
893     /** Constructor to create a heap based Bottom for Ladder Queue.
894
895         The constructor initializes the max event time to zero.
896     */
897     HeapBottom() : maxEvtTime(0) {}
898
899     /** Add events from a Bucket into the heap bottom.
```

```
900
901         This method is used to bulk move events from a rung of the
902         ladder (or top) into the bottom.  The events are added and
903         place in a vector backed heap in preparation for scheduling.
904
905         \param bucket The bucket from where events are to be
906         moved into the bottom rung.
907     */
908     void enqueue(Bucket&& bucket);
909
910     /** Add a single event into the heap bottom.
911
912         This method is used to add a single event into the bottom.
913         The event is added into a vector backed heap in
914         preparation for scheduling.
915
916         \param Event The event to be added into the bottom rung.
917     */
918     void enqueue(xxxx::Event* event);
919
920     /** Convenience method to remove an event from the vector of events.
921
922         This is an internal convenience method that is used to remove
923         the front (i.e., event with lowest timestamp) event.
924
925         \return A pointer to the highest priority event in the vector of events.
926     */
927     xxxx::Event* pop_front();
928
929     /** Obtain pointer to the highest priority (lowest receive time)
930         event.
931
932         This method can be used to obtain a pointer to the highest
933         priority event in the vector of events.
934
935         \return A pointer to the next event to be processed.
936     */
937     xxxx::Event* front() const { return sel.front(); }
938
939     /** Determines if HeapBottom is empty.
940
941         \return This method returns true if the vector of events is
942         logically empty.
943     */
944     bool empty() const { return sel.empty();  }
945
946     /** Convenience method to remove all events sent by the sender
947         at-or-after the given send Time.
948
949         This method will compare the timestamps of all events in the
950         list with that of the specified event. Any Event (from the
951         same sender) that was sent at a time greater than or equal to
952         that of the specified event will be deleted.
953
954         \param[in] sender The sender agent whose events are to be
955         removed.
956
957         \param[in] sendTime The time at-or-after which events from the
958         sender are to be removed from the vector.
959
960         \return This method returns the number of events that were
961         removed.
962     */
963     int remove_after(xxxx::AgentID sender, const Time sendTime);
964
965     /** Remove all events for a given receiver agent in the bucket
966         encapsulated by this object.
967
968         This is a convenience method that removes all events for a
969         given receiver agent in this object.  This method is used to
970         remove events scheduled for an agent, when an agent is removed
971         from the scheduler.
972     */
973     int remove(xxxx::AgentID receiver);
974
```



```
975     /** Convenience method used to dequeue the next set of events for
976         scheduling.
977
978         This method is used to provide necessary implementation to
979         interface with the XXXX scheduler.  This method dequeues the
980         next batch of the concurrent events for processing by a given
981         agent.
982
983         \param[out] events The container to which all the events to be
984         processed is to be added.
985     */
986     void dequeueNextAgentEvents(xxxx::EventContainer& events);
987
988     /** Convenience method for debugging/troubleshooting.
989
990         \return The highest timestamp from the events in the heap-bottom.
991         If no events are present this method returns TIME_INFINITY.
992     */
993     xxxx::Time maxTime() const;
994
995     /** Convenience method for debugging/troubleshooting.
996
997         \return The minimum timestamp from the events in the heap-bottom.
998         If no events are present this method returns TIME_INFINITY.
999     */
1000    xxxx::Time findMinTime() const;
1001
1002    /** Convenience method to check if the entries in the bottom are
1003        sorted correctly. This method is purely used for
1004        troubleshooting/debugging.
1005    */
1006    void validate();
1007
1008    /** Obtain the count of events.
1009
1010        \return The sum of events in the heap-bottom of the Ladder queue.
1011    */
1012    inline size_t size() const { return sel.size(); }
1013
1014    // The method below is purely for troubleshooting one scenario
1015    // where an event would get stuck in the ladder and not get
1016    // scheduled correctly.
1017    bool haveBefore(const Time recvTime) const;
1018
1019    /** Method to determine the range of receive time values currently
1020        in bottom.  This value is used to decide if it is worth moving
1021        events from bottom into the ladder.
1022
1023        \note This method is called often, particularly in parallel
1024        simulation.  Consequently, it needs to be quick. To ensure it
1025        is quick, we track the maximum event time in the min-heap
1026        using the maxEvtTime instance variable.
1027
1028        \return The difference in maximum and minimum receive
1029        timestamp of events in the bottom.  This value is zero if all
1030        events have the same receive time.  If the bottom is empty,
1031        then this method also returns zero.
1032    */
1033    xxxx::Time getTimeRange() const {
1034        if (sel.empty()) {
1035            return 0;
1036        }
1037        return (maxEvtTime - sel.front()->getReceiveTime());
1038    }
1039
1040    /** Determine bucket width to move bottom into ladder.
1041
1042        This method is invoked only when the ladder is empty and the
1043        bottom is long and needs to be moved into the ladder.  This
1044        method must compute and return the preferred bucket width.
1045
1046        \note If the bottom is empty this method returns bucket width
1047        of 0.
1048     */
1049    double getBucketWidth() const;
```

```
1050
1051 protected:
1052     // Currently there are no protected members in this class
1053     void print(std::ostream& os = std::cout) const;
1054 private:
1055     // Vector of events
1056     EventVector sel;
1057     // The maximum event time in the heap bottom.
1058     xxxx::Time maxEvtTime;
1059 };
1060
1061
1062 /** Comparator for multi-set to ensure least-time-stamp-first (LTSF)
1063     ordering in the set of events.  This definition is necessary
1064     because the Bottom::compare used with heap arranges events in
1065     reverse order when used with std::multiset.
1066 */
1067 struct MultiSetComparator {
1068     inline bool operator()(const xxxx::Event* const lhs,
1069                            const xxxx::Event* const rhs) {
1070         return Bottom::compare(rhs, lhs);
1071     }
1072 };
1073
1074 // The definition for a multi-set of Events
1075 using EventMultiSet = std::multiset<xxxx::Event*, MultiSetComparator>;
1076
1077 /** An alternative implementation for Bottom that uses a std::multiset
1078     instead of a binary heap.  The objective of having multiple
1079     implementations for Bottom is to identify the best data structure
1080     for the type of operations that are performed on Bottom in both
1081     sequential and TimeWarp simulations.
1082 */
1083 class MultiSetBottom {
1084     friend class LadderQueue;   // NOTE: uses sel directly
1085 public:
1086     // MultiSetBottom() : sel(MultiSetComparator()) {}
1087
1088     /** Add events from a Bucket into the multi-set bottom.
1089
1090         This method is used to bulk move events from a rung of the
1091         ladder (or top) into the bottom.  The events are added and
1092         place in a multi-set bottom in preparation for scheduling.
1093
1094         \param bucket The bucket from where events are to be
1095         moved into the bottom rung.
1096     */
1097     void enqueue(Bucket&& bucket);
1098
1099     /** Add a single event into the Multi-set bottom.
1100
1101         This method is used to add a single event into the bottom.
1102         The event is added into a multi-set backed Bottom in
1103         preparation for scheduling.
1104
1105         \param Event The event to be added into MultiSetBottom.
1106     */
1107     void enqueue(xxxx::Event* event);
1108
1109     /** Convenience method to remove an event from the multi-set container
1110         of events.
1111
1112         This is an internal convenience method that is used to remove
1113         the front (i.e., event with lowest timestamp) event.
1114
1115         \return A pointer to the highest priority event in the multi-set
1116         container of events.
1117     */
1118     xxxx::Event* pop_front();
1119
1120     /** Obtain pointer to the highest priority (lowest receive time)
1121         event.
1122
1123         This method can be used to obtain a pointer to the highest
1124         priority event in the multi-set container that stores the events.
```



```
1125
1126         \return A pointer to the next event to be processed.
1127     */
1128     xxxx::Event* front() const { return *sel.cbegin(); }
1129
1130     /** Determines if MultiSetBottom is empty.
1131
1132         \return This method returns true if the multi-set container that holds
1133         the events is logically empty.
1134     */
1135     bool empty() const { return sel.empty();  }
1136
1137     /** Convenience method to remove all events sent by the sender
1138         at-or-after the given send Time.
1139
1140         This method will compare the timestamps of all events in the
1141         list with that of the specified event. Any Event (from the
1142         same sender) that was sent at a time greater than or equal to
1143         that of the specified event will be deleted.
1144
1145         \param[in] sender The sender agent whose events are to be
1146         removed.
1147
1148         \param[in] sendTime The time at-or-after which events from the
1149         sender are to be removed from the multi-set container.
1150
1151         \return This method returns the number of events that were
1152         removed.
1153     */
1154     int remove_after(xxxx::AgentID sender, const Time sendTime);
1155
1156     /** Remove all events for a given receiver agent in the bucket
1157         encapsulated by this object.
1158
1159         This is a convenience method that removes all events for a
1160         given receiver agent in this object.  This method is used to
1161         remove events scheduled for an agent, when an agent is removed
1162         from the scheduler.
1163     */
1164     int remove(xxxx::AgentID receiver);
1165
1166     /** Convenience method used to dequeue the next set of events for
1167         scheduling.
1168
1169         This method is used to provide necessary implementation to
1170         interface with the XXXX scheduler.  This method dequeues the
1171         next batch of the concurrent events for processing by a given
1172         agent.
1173
1174         \param[out] events The container to which all the events to be
1175         processed is to be added.
1176     */
1177     void dequeueNextAgentEvents(xxxx::EventContainer& events);
1178
1179     /** Convenience method for debugging/troubleshooting.
1180
1181         \return The highest timestamp from the events in the multi-set bottom.
1182         If no events are present this method returns TIME_INFINITY.
1183     */
1184     xxxx::Time maxTime() const;
1185
1186     /** Convenience method for debugging/troubleshooting.
1187
1188         \return The minimum timestamp from the events in the multi-set bottom.
1189         If no events are present this method returns TIME_INFINITY.
1190     */
1191     xxxx::Time findMinTime() const;
1192
1193     /** Convenience method to check if the entries in the bottom are
1194         sorted correctly. This method is purely used for
1195         troubleshooting/debugging.
1196     */
1197     void validate();
1198
1199     /** Obtain the count of events.
```

```
1200
1201            \return The sum of events in the multi-set bottom of the Ladder queue.
1202        */
1203        inline size_t size() const { return sel.size(); }
1204
1205        // The method below is purely for troubleshooting one scenario
1206        // where an event would get stuck in the ladder and not get
1207        // scheduled correctly.
1208        bool haveBefore(const Time recvTime) const;
1209
1210        /** Method to determine the range of receive time values currently
1211            in bottom.  This value is used to decide if it is worth moving
1212            events from bottom into the ladder.
1213
1214            \return The difference in maximum and minimum receive
1215            timestamp of events in the bottom. This value is zero if all
1216            events have the same receive time. If the bottom is empty,
1217            then this method also returns zero.
1218        */
1219        xxxx::Time getTimeRange() const {
1220            if (sel.empty()) {
1221                return 0;
1222            }
1223            return ((*sel.rbegin())->getReceiveTime() -
1224                    (*sel.begin())->getReceiveTime());
1225        }
1226
1227        /** Determine bucket width to move bottom into ladder.
1228
1229            This method is invoked only when the ladder is empty and the
1230            bottom is long and needs to be moved into the ladder.  This
1231            method must compute and return the preferred bucket width.
1232
1233            \note If the bottom is empty this method returns bucket width
1234            of 0.
1235         */
1236        double getBucketWidth() const;
1237
1238    protected:
1239        // Currently there are no protected members in this class
1240        void print(std::ostream& os = std::cout) const;
1241    private:
1242        // The multi-set container that stores events.
1243        EventMultiSet sel;
1244    };
1245
1246    /** Class that represents one rung in the ladder queue.
1247
1248        The rung uses the same strategy for receive time-based
1249        bucket creation as the regular ladder queue.
1250    */
1251    class Rung {
1252    public:
1253        /** The constructor to create an empty rung.
1254
1255            The constructor merely initializes all the instance variables
1256            to default initial values to create an empty rung.
1257        */
1258        Rung() : rStartTS(TIME_INFINITY), rCurrTS(TIME_INFINITY),
1259                 bucketWidth(0), currBucket(0), rungEventCount(0) {
1260            LQ_STATS(maxBkts = 0);
1261        }
1262
1263        /** Convenience constructor to create a rung using events from the
1264            top rung.
1265
1266            This is a delegating constructor that delegates the actual
1267            tasks to the overloaded constructor.
1268
1269            \param[in] top The top bucket from where the events are to be
1270            created.
1271        */
1272        explicit Rung(Top& top);
1273
1274        /** Convenience constructor to create a rung with events from a
```



```
1275            given bucket.
1276
1277            \param[in,out] bkt The bucket from where events are to be
1278            moved into this newly created rung.  After this operation data
1279            in the bucket is cleared.
1280
1281            \param[in] rStart The start time for this rung.
1282
1283            \param[in] bucketWidth The delta in receive time for each
1284            bucket in this rung.  The bucketWidth must be > 0.
1285        */
1286        Rung(Bucket&& bkt, const Time rStart, const double bucketWidth);
1287
1288        /** Convenience constructor to create a rung with events from a
1289            given EventVector.
1290
1291            \param[in,out] list The vector from where events are to be
1292            moved into this newly created rung. After this operation data
1293            in the vector is cleared.
1294
1295            \param[in] rStart The start time for this rung.
1296
1297            \param[in] bucketWidth The delta in receive time for each
1298            vector in this rung. The bucketWidth must be > 0.
1299        */
1300        Rung(EventVector&& list, const Time rStart, const double bucketWidth);
1301
1302        /** Convenience constructor to create a rung with events from a
1303            given multi-set container.
1304
1305            \param[in,out] set The multi-set container from where events are to be
1306            moved into this newly created rung. After this operation data
1307            in the container is cleared.
1308
1309            \param[in] rStart The start time for this rung.
1310
1311            \param[in] bucketWidth The delta in receive time for each
1312            container in this rung. The bucketWidth must be > 0.
1313        */
1314        Rung(EventMultiSet&& set, const Time rStart, const double bucketWidth);
1315
1316        /** Remove the next bucket in this rung for moving to another rung
1317            in the ladder.
1318
1319            This method must be used to remove the next bucket from this
1320            rung. The bucket is logically removed (or moved) out of this
1321            rung.
1322
1323            \param[out] bktTime The simulation receive time associated
1324            with the bucket being moved out.
1325        */
1326        Bucket&& removeNextBucket(xxxx::Time& bktTime);
1327
1328        /** Determine if this rung is empty.
1329
1330            This is a convenience method that is used to determine if this
1331            rung contains any events to be processed.
1332
1333            \return This method returns true if the rung does not have any
1334            events -- i.e., when the rung is empty.
1335        */
1336        bool empty() const { return (rungEventCount == 0); }
1337
1338        /** Add an event to suitable bucket in this rung.
1339
1340            This method computes a bucket index (based on equation #2 in
1341            LQ paper) using the formula:
1342
1343            \code
1344            size_t bucketNum = (event->getReceiveTime() - rStartTS) / bucketWidth;
1345            \endcode
1346
1347            \param[in] event The event to be added to a suitable bucket in
1348            this rung.
1349        */
```

```
1350      void enqueue(xxxx::Event* event);
1351
1352      /** Obtain the start time for this rung.
1353
1354          This method returns the rung starting time that was set when
1355          this rung was created.
1356
1357          \return The starting time of this rung that determines the
1358          lowest timestamp event that can be added to this rung.
1359      */
1360      xxxx::Time getStartTime() const { return rStartTS; }
1361
1362      /** Obtain the bucket width (i.e., difference in receive times for
1363          adjacent buckets) for this rung.
1364
1365          This method returns the bucket width that was set when this
1366          rung was created.
1367
1368          \return The bucket width for this rung.
1369      */
1370      double getBucketWidth() const { return bucketWidth; }
1371
1372      /** The current bucket value in this ladder queue.
1373
1374          The current minimum time of events that can be added to this
1375          rung of the ladder queue.
1376
1377          \return The minimum timestamp of events that can be added to
1378          the rung of this ladder queue.
1379      */
1380      xxxx::Time getCurrTime() const { return rCurrTS; }
1381
1382      /** The maximum receive time value of event that can be added to
1383          this rung.
1384
1385          \return The maximum receive time of an event that can be added
1386          to a bucket in this rung.
1387      */
1388      xxxx::Time getMaxRungTime() const {
1389          return rStartTS + (bucketList.size() * bucketWidth);
1390      }
1391
1392      /** Convenience method to determine if a given event can be added
1393          to this rung.
1394
1395          \param[in] event The event whose receive time is to be used to
1396          check to see if it can be added to this ladder.
1397
1398          \return Returns true if the event can be added to this rung.
1399          Otherwise it returns false.
1400      */
1401      bool canContain(xxxx::Event* event) const;
1402
1403      /** Remove all events from the given sender sent at-or-after the
1404          specified send time from all buckets in this rung.
1405
1406          This method linearly scans the buckets, checks, and removes
1407          all events that were sent by the sender at-or-after the
1408          specified send time.
1409
1410          \param[in] sender The sender agent whose events are to be
1411          removed.
1412
1413          \param[in] sendTime The time at-or-after which events from the
1414          sender are to be removed from the given list.
1415
1416          \param[out] ceScanRung The stats object to be updated with
1417          number of events scanned in the buckets in this rung.
1418
1419          \return This method returns the total number of events that
1420          were removed from this rung.
1421      */
1422      int remove_after(xxxx::AgentID sender, const Time sendTime
1423                       LQ_STATS(COMMA Avg& ceScanRung));
1424
```



```
1425      /** Remove all events for a given receiver agent in this rung.
1426
1427          This is a convenience method that removes all events for a
1428          given receiver agent in this rung.  This method is used to
1429          remove events scheduled for an agent, when an agent is removed
1430          from the scheduler.
1431      */
1432      int remove(xxxx::AgentID receiver
1433                 LQ_STATS(COMMA Avg& ceScanRung));
1434
1435      /** Check to ensure that the number of events in various buckets
1436          matches the count instance variable.
1437
1438          This method is used only for troubleshooting/debugging
1439          purposes. If counts don't match then assert fails in this
1440          method causing the simulation to abort.
1441      */
1442      void validateEventCounts() const;
1443
1444      /** Print a user-friendly version of the events in this queue.
1445
1446          Currently this method is not implemented.
1447      */
1448      void prettyPrint(std::ostream& os) const;
1449
1450      /** Update the statistics object with data from this rung.
1451
1452          \param[out] avgBktCnt Update the average number of buckets in
1453          this rung.
1454      */
1455      void updateStats(Avg& avgBktCnt) const;
1456
1457      /** Convenience method to determine if the current bucket in this
1458          rung is empty.
1459
1460          \return This method returns true if the current bucket in this
1461          rung is empty.
1462      */
1463      bool isCurrBucketEmpty() const {
1464          return (currBucket >= bucketList.size() ||
1465                  bucketList[currBucket].empty());
1466      }
1467
1468      /** This method is purely for troubleshooting one scenario where
1469          an event would get stuck in the ladder and not get scheduled
1470          correctly.
1471
1472          \param[in] recvTime The time to be used for checking to see if
1473          sub-buckets have an event before this time.
1474
1475          \return Returns true if an event before this receiveTime (for
1476          any agent) is pending in a sub-bucket.
1477      */
1478      bool haveBefore(const Time recvTime) const;
1479
1480  protected:
1481
1482  private:
1483      /** The lowest timestamp event that can be added to this rung.
1484          This value is set when a rung is created and is never changed
1485          during the lifetime of this rung.
1486      */
1487      xxxx::Time rStartTS;
1488
1489      /** The timestamp of the lowest event that can be currently added
1490          to this rung. This value logically starts with rStartTS and
1491          grows to the time stamp of last bucket in this rung as buckets
1492          are dequeued from this rung.
1493      */
1494      xxxx::Time rCurrTS;
1495
1496      /** The width of the bucket in simulation receive time
1497          differences. This value can be fractional.
1498      */
1499      double bucketWidth;
```

```
1500
1501        /** The index of the current bucket on this rung to which events
1502            can be added.  This is also the next bucket that will be
1503            dequeued from the rung.
1504        */
1505        size_t currBucket;
1506
1507        /** The vector containing the set of Buckets.
1508        */
1509        std::vector<Bucket> bucketList;
1510
1511        /** Total number of events still present in this rung. This is
1512            used to report size and check for empty quickly.
1513        */
1514        int rungEventCount;
1515
1516        /** Statistics object to track the maximum number of buckets used
1517            in this rung
1518        */
1519        LQ_STATS(size_t maxBkts);
1520    };
1521
1522    /** The top-level ladder queue
1523
1524        <p>This class represents the top-level ladder queue class
1525        that interfaces with the XXXX scheduler. This class implements
1526        the top-level logic associated with ladder queue to enqueue,
1527        dequeue, and cancel events from the ladder queue.</p>
1528    */
1529    class LadderQueue : public EventQueue {
1530    public:
1531        LadderQueue() : EventQueue("LadderQueue"), nRung(0), ladderEventCount(0) {
1532            ladder.reserve(MaxRungs);
1533            LQ_STATS(ceTop    = ceLadder   = ceBot  = 0);
1534            LQ_STATS(insTop   = insLadder  = insBot = 0);
1535            LQ_STATS(maxRungs = maxBotSize = 0);
1536        }
1537
1538        /** The destructor.
1539
1540            Currently the destructor does not have anything special to do
1541            as the different encapsulated objects handle all the necessary
1542            clean-up.
1543        */
1544        ~LadderQueue();
1545
1546        /** Enqueue an event into the laadder queue.
1547
1548            Depending on the scenario the event is appropriately added to
1549            one of: top, ladder rung, or the bottom.
1550
1551            \param[in] e The event to be enqueued for scheduling in the
1552            ladder queue.
1553        */
1554        void enqueue(xxxx::Event* e);
1555
1556        /** Dequeue an event from the ladder queue.
1557
1558            This method removes the highest priority event from Bottom.
1559            If the Bottom is empty, it is re-populated with events.
1560
1561        */
1562        xxxx::Event* dequeue();
1563
1564        /** Cancel all events from a given sender that were sent
1565            at-or-after the specified send time.
1566
1567            This method essentially calls the corresponding method(s) in
1568            top, rung, and bottom to cancel pending events.
1569
1570            \param[in] sender The sender agent whose events are to be
1571            removed.
1572
1573            \param[in] sendTime The time at-or-after which events from the
1574            sender are to be removed from the given list.
```



```
1575
1576            \return This method returns the number of events that were
1577            removed.
1578        */
1579        int remove_after(xxxx::AgentID sender, const Time sendTime);
1580
1581        /** Determine if the ladder queue is empty.
1582
1583            Implements the interface method used by XXXX::Scheduler.
1584
1585            \return Returns true if top, ladder, and bottom are all empty
1586            -- i.e., there are no pending events.
1587        */
1588        virtual bool empty() {
1589            return top.empty() && (ladderEventCount == 0) &&  bottom.empty();
1590        }
1591
1592        /** Implementation for method used by XXXX::Scheduler.
1593
1594            This method is called by XXXX kernel to inform the scheduler
1595            queue about an agent being added during initialization.  The
1596            ladder queue does not utilize this information and
1597            consequently this method does not have any special operation
1598            to perform.
1599
1600            \param[in] agent The agent being added. This pointer is not
1601            really used.
1602
1603            \return This method simply returns nullptr as the ladder queue
1604            does not use any cross references in xxxx::Agent for its
1605            operations.
1606        */
1607        virtual void* addAgent(xxxx::Agent* agent);
1608
1609        /** Remove an agent just before simulation completes.
1610
1611            This method is invoked by the XXXX kernel to inform that an
1612            agent is being removed.  This method removes all pending
1613            events for the specified agent from the ladder queue.
1614
1615            \param[in] agent The agent whose sender ID is used to remove
1616            all pending events in the top, rungs, and bottom.
1617        */
1618        virtual void removeAgent(xxxx::Agent* agent);
1619
1620        /** Implement interface method to peek at the next event to
1621            schedule.
1622
1623            \note In order to enable peeking of the front event, the
1624            bottom may need to get populated.
1625
1626            \return A pointer to the next event to schedule (if any).  The
1627            event is not dequeued.
1628         */
1629        virtual xxxx::Event* front();
1630
1631        /** This method is used to provide necessary implementation to
1632            interface with the XXXX scheduler. This method dequeues the
1633            next batch of the concurrent events for processing by a given
1634            agent.
1635
1636            \param[out] events The container to which all the events to be
1637            processed is to be added.
1638        */
1639        virtual void dequeueNextAgentEvents(xxxx::EventContainer& events);
1640
1641        /** Add an event to be scheduled to this ladder queue.
1642
1643            This method implements the core API used by agents to schedule
1644            events for each other.
1645
1646            \param[in] agent The receiver agent for which the event is
1647            scheduled. This pointer is not used.
1648
1649            \param[in] event The event to be scheduled. This simply calls
```

```
1650        the overloaded enqueue method. The reference count on the
1651        event is increased by this method to account for this event
1652        being present in the ladder queue.
1653    */
1654    virtual void enqueue(xxxx::Agent* agent, xxxx::Event* event);
1655
1656    /** Enqueue a batch of events
1657
1658        This API to schedule a block of events. This API is typically
1659        used after a rollback.
1660
1661        \param[in] agent The receiver agent for which the event is
1662        scheduled. This pointer is not used.
1663
1664        \param[in] events The list of events to be scheduled. This
1665        simply calls the overloaded enqueue method to enqueue one
1666        event at a time.
1667    */
1668    virtual void enqueue(xxxx::Agent* agent, xxxx::EventContainer& events);
1669
1670    /** Implement XXXX kernel API to cancel all events sent by a given
1671        agent after a given time.
1672
1673        \param[in] dest The destination agent whose events are to be
1674        cancelled.
1675
1676        \param[in] sender The sender agent ID whose events are to be
1677        cancelled.
1678
1679        \param[in] sentTime The send time at-or-after which all events
1680        from the sender are to be cancelled.
1681    */
1682    virtual int eraseAfter(xxxx::Agent* dest, const xxxx::AgentID sender,
1683                           const xxxx::Time sentTime);
1684
1685    /** Print a human understandable version of the events in this
1686        queue.
1687     */
1688    virtual void prettyPrint(std::ostream& os) const;
1689
1690    /** Convenience method to check to see if ladder queue has events
1691        before the specified receive time.
1692
1693        This method is used for troubleshooting/debugging only.
1694
1695        \param[in] recvTime The receive time for checking.
1696
1697        \return Returns true if an event before this receiveTime (for
1698        any agent) is pending in the bottom rung.
1699    */
1700    bool haveBefore(const Time recvTime,
1701                    const bool checkBottom = false) const;
1702
1703    /** Method to report aggregate statistics.
1704
1705        This method is invoked at the end of simulation after all
1706        agents on this rank have been finalized.  This method is meant
1707        to report any aggregate statistics from this queue.  This
1708        method writes statistics only if LQ_STATS macro is enabled.
1709
1710        \param[out] os The output stream to which the statistics are
1711        to be written.
1712    */
1713    virtual void reportStats(std::ostream& os);
1714
1715    /** The maximum number of rungs that are normally created in the
1716        ladder queue.  The default value for this set to 8 based on
1717        the value suggested by Tang et. al. in the original Ladder
1718        Queue paper.  However, this value can make some difference in
1719        the overall performance and possibly fine tuned to suit the
1720        application needs based on the concurrency and number of
1721        events in the model.
1722    */
1723    static size_t MaxRungs;
1724
```



```
1725 protected:
1726     /** Check and create rungs in the ladder and return the next
1727         bucket of events from the ladder.
1728
1729         This method implements the corresponding recurseRung method
1730         from the LQ paper. Refer to the paper for the details.
1731     */
1732     Bucket&& recurseRung();
1733
1734     /** This is a convenience method that is used to move events from
1735         the ladder into bottom.
1736
1737         This method moves events from the current bucket in the last
1738         rung of the ladder into the bottom. If this method is called
1739         when bottom is not empty, it does not perform any operation
1740         and returns immediately. If the ladder does not have any
1741         events, but the top has events, then this method first moves
1742         events from top-rung into the ladder and then removes events
1743         from the last rung into the bottom-rung.
1744     */
1745     void populateBottom();
1746
1747     /** Method to create a new ladder rung from the current bottom.
1748
1749         This method should be called only when the following 2
1750         conditions are met:
1751
1752         1. Length of bottom is > LQ2T_THRESH
1753
1754         2. The bottom has events that are at different time stamps --
1755            that is bottom.getTimeRange() > 0.
1756
1757         \return This method returns the index of the rung created so
1758         that the caller can readily work with that rung.
1759     */
1760     int createRungFromBottom();
1761
1762 private:
1763     Top top;
1764
1765     /** The ladder in the queue. The ladder consists of a set of
1766         rungs. The currently used rung in the ladder is indicated by
1767         the nRung instance variable. If the ladder is empty, then
1768         nRung is (or should be) 0
1769     */
1770     std::vector<Rung> ladder;
1771
1772     /** The currently used last rung in the ladder queue. If the
1773         ladder is empty, then nRung is (or should be) 0. Otherwise
1774         this value is (or should be) in the range 0 < nRung <=
1775         ladder.size(). Rungs below nRung are not used and they do not
1776         contain any events to be scheduled.
1777     */
1778     size_t nRung;
1779
1780     /** Instance variable to track the current number of pending
1781         events in all the rungs of the ladder. This is a convenience
1782         instance variable to quickly detect pending events in the
1783         ladder without having to iterate through each rung.
1784     */
1785     int ladderEventCount;
1786
1787     Bottom bottom;
1788     // HeapBottom bottom;
1789     // MultiSetBottom bottom;
1790
1791     LQ_STATS(Avg ceTop);
1792     LQ_STATS(Avg ceBot);
1793     LQ_STATS(Avg ceLadder);
1794
1795     LQ_STATS(Avg ceScanTop);
1796     LQ_STATS(Avg ceScanLadder);
1797
1798     /** The ceScanBot statistic tracks size of bottom rung scanned
1799         when at one (or more) events were canceled from bottom.
```

```
1800        */
1801        LQ_STATS(Avg ceScanBot);
1802
1803        /** The ceNoCanScanBot statistic tracks size of bottom rung
1804            scanned but did not cancel any events.
1805        */
1806        LQ_STATS(Avg ceNoCanScanBot);
1807
1808        LQ_STATS(int insTop);
1809        LQ_STATS(int insLadder);
1810        LQ_STATS(int insBot);
1811        LQ_STATS(size_t maxRungs);
1812        LQ_STATS(Avg avgBktCnt);
1813        LQ_STATS(Avg botLen);
1814        LQ_STATS(Avg avgBktWidth);
1815
1816        /** Gauge to track the number of events and times bottom was
1817            redistributed to the last rung of the ladder.
1818
1819            Redistributing bottom to the ladder ensures that the bottom
1820            does not get too long. But it is an expensive operation
1821            because all the sorting that was done is lost. So it is a
1822            balance and we track and report this number for reference.
1823        */
1824        LQ_STATS(Avg botToRung);
1825
1826        /** Gauge to track the maximum length of bottom. The length of
1827            bottom plays an important role in the overall performance of
1828            the ladder queue.
1829        */
1830        LQ_STATS(size_t maxBotSize);
1831    };
1832
1833    END_NAMESPACE(xxxx)
1834
1835    #endif
```

```cpp
#ifndef LADDER_QUEUE_CPP
#define LADDER_QUEUE_CPP

#include <algorithm>
#include <functional>
#include "LadderQueue.h"

// ----------------------[ ListBucket methods ]-----------------------

xxxx::ListBucket::~ListBucket() {
    for (auto& event : list) {
        event->decreaseReference();
    }
}

int
xxxx::ListBucket::remove_after(xxxx::AgentID sender, const Time sendTime) {
    size_t removedCount = 0;
    ListBucket::iterator curr = begin();
    while (curr != list.end()) {
        xxxx::Event* const event = *curr;
        if ((event->getSenderAgentID() == sender) &&
            (event->getSentTime() >= sendTime)) {
            event->decreaseReference();
            curr = list.erase(curr);
            removedCount++;
        } else {
            curr++;
        }
    }
    count -= removedCount;  // Track remaining events
    return removedCount;
}

bool
xxxx::ListBucket::haveBefore(const Time recvTime) const {
    EventList::const_iterator next = list.begin();
    while (next != list.end()) {
        xxxx::Event* const event = *next;
        if (event->getReceiveTime() <= recvTime) {
            return true;
        }
        next++;
    }
    return false;
}

int
xxxx::ListBucket::remove(xxxx::AgentID receiver) {
    size_t removedCount = 0;
    ListBucket::iterator curr = list.begin();
    while (curr != list.end()) {
        xxxx::Event* const event = *curr;
        if (event->getReceiverAgentID() == receiver) {
            event->decreaseReference();
            curr = list.erase(curr);
            removedCount++;
        } else {
            curr++;  // onto next event.
        }
    }
    count -= removedCount;  // Track remaining events
    return removedCount;
}

// ----------------------[ VectorBucket methods ]-----------------------

xxxx::VectorBucket::~VectorBucket() {
    for (auto& event : list) {
        event->decreaseReference();
    }
}

void
xxxx::VectorBucket::push_back(VectorBucket&& bucket) {
```



```cpp
    list.reserve(list.size() + bucket.size());
    list.insert(list.end(), bucket.list.begin(), bucket.list.end());
    count += bucket.size();
    bucket.list.clear();
    bucket.count = 0;
}

int
xxxx::VectorBucket::remove_after(xxxx::AgentID sender, const Time sendTime) {
    size_t removedCount = 0;
    size_t curr = 0;
    while (curr < list.size()) {
        xxxx::Event* const event = list[curr];
        if ((event->getSenderAgentID() == sender) &&
            (event->getSentTime() >= sendTime)) {
            // Free-up event.
            event->decreaseReference();
            removedCount++;
            // To minimize removal time replace entry with last one
            // and pop the last entry off.
            list[curr] = list.back();
            list.pop_back();
        } else {
            curr++;  // on to the next event in the list
        }
    }
    count -= removedCount;  // Track remaining events
    return removedCount;
}

int
xxxx::VectorBucket::remove_after_sorted(xxxx::AgentID sender,
                                        const Time sendTime) {
    // Since bucket is sorted we can shortcircuit scan if last event's
    // time is less-or-equal to sendTime.
    if (list.empty() || (sendTime >= list.front()->getReceiveTime())) {
        return -1;  // this bucket does not have events to be cancelled.
    }
    size_t removedCount = 0;
    EventVector::iterator curr = list.begin();
    while (curr != list.end()) {
        xxxx::Event* const event = *curr;
        if ((event->getSenderAgentID() == sender) &&
            (event->getSentTime() >= sendTime)) {
            // Free-up event.
            event->decreaseReference();
            removedCount++;
            // To preserved sorted order erase from list correctly.
            curr = list.erase(curr);
        } else {
            curr++;  // on to the next event in the list
        }
    }
    count -= removedCount;  // Track remaining events
    return removedCount;
}

int
xxxx::VectorBucket::remove(xxxx::AgentID receiver) {
    size_t removedCount = 0;
    size_t curr = 0;
    while (curr < list.size()) {
        xxxx::Event* const event = list[curr];
        if (event->getReceiverAgentID() == receiver) {
            list.erase(list.begin() + curr);
            event->decreaseReference();
            removedCount++;
        } else {
            curr++;
        }
    }
    count -= removedCount;  // Track remaining events
    return removedCount;
}

```

```
151 bool
152 xxxx::VectorBucket::haveBefore(const Time recvTime) const {
153     EventVector::const_iterator next = list.begin();
154     while (next != list.end()) {
155         xxxx::Event* const event = *next;
156         if (event->getReceiveTime() <= recvTime) {
157             return true;
158         }
159         next++;
160     }
161     return false;
162 }
163
164 // ----------------------------[ Top methods ]-----------------------------
165
166 xxxx::Top::~Top() {
167 }
168
169 void
170 xxxx::Top::reset(const Time startTime) {
171     minTS    = TIME_INFINITY;
172     maxTS    = 0;
173     topStart = startTime;
174 }
175
176 void
177 xxxx::Top::add(xxxx::Event* event) {
178     events.push_front(event);
179     minTS = std::min(minTS, event->getReceiveTime());
180     maxTS = std::max(maxTS, event->getReceiveTime());
181 }
182
183 int
184 xxxx::Top::remove_after(xxxx::AgentID sender, const Time sendTime) {
185     return events.remove_after(sender, sendTime);
186 }
187
188 int
189 xxxx::Top::remove(xxxx::AgentID receiver) {
190     return events.remove(receiver);
191 }
192
193 // ----------------------------[ Bottom methods ]--------------------------
194
195 void
196 xxxx::Bottom::enqueue(ListBucket&& bucket, ListBucket&) {
197     // Note that pop_front must be O(1) here -- which it is since
198     // bucket is a linked list.
199     while (!bucket.empty()) {
200         enqueue(bucket.pop_front());
201     }
202 }
203
204 void
205 xxxx::Bottom::enqueue(VectorBucket&& bucket, VectorBucket&botList) {
206     // For vector-to-vector copy use different strategy for buckets
207     botList.push_back(std::move(bucket));
208     // Now sort the whole bottom O(n*log(n)) operation
209     std::sort(botList.begin(), botList.end(), Bottom::compare);
210 }
211
212 void
213 xxxx::Bottom::enqueue(xxxx::Event* event, ListBucket& botList) {
214     if (botList.empty() || !compare(event, botList.front())) {
215         // Empty list or event is smaller than head.  Add event to the
216         // front of the list.
217         botList.push_front(event);
218     } else {
219         // Insert event in sorted list of events. For this we need to
220         // search for the correct insertion location.
221         ASSERT(compare(event, botList.front()));
222         ListBucket::iterator next = botList.begin();
223         ListBucket::iterator prev = next++;
224         while ((next != botList.end()) && compare(event, *next)) {
225             prev = next++;
```



```
226         }
227         botList.insert_after(prev, event);
228     }
229     DEBUG(validate());
230 }
231
232 void
233 xxxx::Bottom::enqueue(xxxx::Event* event, VectorBucket& botList) {
234     VectorBucket::reverse_iterator iter =
235         std::upper_bound(botList.rbegin(), botList.rend(), event, revCompare);
236     botList.insert_after(iter, event);
237     DEBUG(validate());
238 }
239
240 double
241 xxxx::Bottom::getBucketWidth() const {
242     if (empty()) {
243         return 0;
244     }
245     ASSERT(sel.front() != NULL);
246     ASSERT(sel.back()  != NULL);
247     const double maxTS = sel.front()->getReceiveTime();
248     const double minTS = sel.back()->getReceiveTime();
249     return (maxTS - minTS + size() - 1.0) / sel.size();
250 }
251
252 int
253 xxxx::Bottom::remove(xxxx::AgentID receiver) {
254     return sel.remove(receiver);
255 }
256
257 void
258 xxxx::Bottom::dequeueNextAgentEvents(xxxx::EventContainer& events) {
259     if (empty()) {
260         return;
261     }
262     const xxxx::Event*  nextEvt  = front();
263     const xxxx::AgentID receiver = nextEvt->getReceiverAgentID();
264     const xxxx::Time    currTime = nextEvt->getReceiveTime();
265
266     do {
267         xxxx::Event* event = pop_front();
268         events.push_back(event);
269         nextEvt = (!empty() ? front() : NULL);
270         DEBUG(std::cout << "Delivering: " << *event << std::endl);
271     } while (!empty() && (nextEvt->getReceiverAgentID() == receiver) &&
272              TIME_EQUALS(nextEvt->getReceiveTime(), currTime));
273     DEBUG(validate());
274 }
275
276 xxxx::Time
277 xxxx::Bottom::findMinTime() const {
278     if (empty()) {
279         return TIME_INFINITY;
280     }
281     return sel.front()->getReceiveTime();
282 }
283
284 xxxx::Time
285 xxxx::Bottom::maxTime() const {
286     if (empty()) {
287         return TIME_INFINITY;
288     }
289     /*
290     Bucket::const_iterator next = sel.cbegin();
291     Bucket::const_iterator prev = next++;
292     while (next != sel.cend()) {
293         prev = next++;
294     }
295     return (*prev)->getReceiveTime();
296     */
297     return sel.back()->getReceiveTime();
298 }
299
300 void
```

```
301 xxxx::Bottom::validate() {
302     if (sel.empty()) {
303         return;
304     }
305     Bucket::iterator next = sel.begin();
306     Bucket::iterator prev = next++;
307     while ((next != sel.end()) &&
308            ((*next)->getReceiveTime() >= (*prev)->getReceiveTime())) {
309         prev = next++;
310     }
311     if (next != sel.end()) {
312         std::cout << "Error in LadderQueue.Bottom: Event " << **next
313                   << " was found after " << **prev << std::endl;
314     }
315     ASSERT( next == sel.end() );
316 }
317
318 // ----------------------[ HeapBottom methods ]---------------------------
319
320 void
321 xxxx::HeapBottom::enqueue(Bucket&& bucket) {
322     // Note that pop_front must be O(1) here -- which it is since
323     // bucket is a linked list.  Due to bulk adding, ensure that the
324     // heap container has enough capacity.
325     sel.reserve(sel.size() + bucket.size() + 1);
326     // Add all the events to the conainer.
327     while (!bucket.empty()) {
328         xxxx::Event* const event = bucket.pop_front();
329         maxEvtTime = std::max(maxEvtTime, event->getReceiveTime());
330         sel.push_back(event);
331     }
332     // Restore heap properties for the queue.
333     std::make_heap(sel.begin(), sel.end(), Bottom::compare);
334 }
335
336 void
337 xxxx::HeapBottom::enqueue(xxxx::Event* event) {
338     ASSERT(sel.empty() || (front()->getReceiveTime() <= findMinTime()));
339     maxEvtTime = std::max(maxEvtTime, event->getReceiveTime());
340     sel.push_back(event);
341     std::push_heap(sel.begin(), sel.end(), Bottom::compare);
342 }
343
344 xxxx::Event*
345 xxxx::HeapBottom::pop_front() {
346     std::pop_heap(sel.begin(), sel.end(), Bottom::compare);
347     xxxx::Event* retVal = sel.back();
348     sel.pop_back();
349     maxEvtTime = (sel.empty() ? 0 : std::max(maxEvtTime,
350                                              retVal->getReceiveTime()));
351     return retVal;
352 }
353
354 int
355 xxxx::HeapBottom::remove_after(xxxx::AgentID sender, const Time sendTime) {
356     // Prior to Dec 23 2016, this method would copy events to be
357     // retained into a temporary vector and make heap again.  This
358     // approach was too slow for larger list.  So instead, this method
359     // removes events one at a time from the heap, similar to the one
360     // done in BinaryHeapWrapper::removeFutureEvents.
361     size_t removedCount = 0;
362     long currIdx        = sel.size() - 1;
363     while (!sel.empty() && (currIdx >= 0)) {
364         ASSERT(currIdx < (long) sel.size());
365         xxxx::Event* const event = sel[currIdx];
366         ASSERT(event != NULL);
367         // An event is deleted only if the *sent* time is greater than
368         // the sendTime and if the event is from same sender
369         if ((event->getSenderAgentID() == sender) &&
370             (event->getSentTime() >= sendTime)) {
371             // This event needs to be removed.
372             maxEvtTime = std::max(maxEvtTime, event->getReceiveTime());
373             event->decreaseReference();
374             removedCount++;
375             // Now it is time to patchup the hole and fix up the heap.
```



```
376             // To patch-up we move event from the bottom up to this
377             // slot and then fix-up the heap.
378             sel[currIdx] = sel.back();
379             sel.pop_back();
380             // Fix-up the heap using helper method.
381             EventQueue::fixHeap(sel, currIdx, Bottom::compare);
382             // Update the current index so that it is within bounds.
383             currIdx = std::min<long>(currIdx, sel.size() - 1);
384         } else {
385             // Check the previous element in the vector to see if that
386             // is a candidate for cancellation.
387             currIdx--;
388         }
389     }
390     // If the queue is empty reset the maxEvtTime
391     maxEvtTime = (!sel.empty() ? maxEvtTime : 0);
392     // Return number of events canceled to track statistics.
393     return removedCount;
394 }
395
396 int
397 xxxx::HeapBottom::remove(xxxx::AgentID receiver) {
398     // Copy events to be retained into a temporary vector and finally
399     // swap it with sel.
400     size_t removedCount = 0;
401     EventVector retained;
402     retained.reserve(sel.size());
403     for (auto curr = sel.begin(); (curr != sel.end()); curr++) {
404         xxxx::Event* const event = *curr;
405         if (event->getReceiverAgentID() == receiver) {
406             maxEvtTime = std::max(maxEvtTime, event->getReceiveTime());
407             event->decreaseReference();
408             removedCount++;
409         } else {
410             retained.push_back(event);
411         }
412     }
413     // Update the sel with list of retained events
414     sel.swap(retained);
415     if (!sel.empty()) {
416         std::make_heap(sel.begin(), sel.end(), Bottom::compare);
417     } else {
418         maxEvtTime = 0;  // empty heap!
419     }
420     return removedCount;
421 }
422
423 void
424 xxxx::HeapBottom::dequeueNextAgentEvents(xxxx::EventContainer& events) {
425     if (sel.empty()) {
426         return;
427     }
428
429     const xxxx::Event*  nextEvt    = front();
430     const xxxx::AgentID receiver   = nextEvt->getReceiverAgentID();
431     const xxxx::Time    currTime   = nextEvt->getReceiveTime();
432
433     do {
434         xxxx::Event* event = pop_front();
435         events.push_back(event);
436         nextEvt = (!empty() ? front() : NULL);
437         DEBUG(std::cout << "Delivering: " << *event << std::endl);
438     } while (!empty() && (nextEvt->getReceiverAgentID() == receiver) &&
439              TIME_EQUALS(nextEvt->getReceiveTime(), currTime));
440     // Reset maximum event time if heap is empty
441     if (sel.empty()) {
442         maxEvtTime = 0;
443     }
444     DEBUG(validate());
445 }
446
447 xxxx::Time
448 xxxx::HeapBottom::maxTime() const {
449     if (empty()) {
450         return TIME_INFINITY;
```

```
451     }
452     xxxx::Time maxTime = front()->getReceiveTime();
453     for (auto curr = sel.begin(); (curr != sel.end()); curr++) {
454         maxTime = std::max(maxTime, (*curr)->getReceiveTime());
455     }
456     ASSERT( maxTime == maxEvtTime );
457     return maxTime;
458 }
459
460 xxxx::Time
461 xxxx::HeapBottom::findMinTime() const {
462     if (empty()) {
463         return 0;
464     }
465     xxxx::Time minTime = front()->getReceiveTime();
466     // for (auto curr = sel.begin(); (curr != sel.end()); curr++) {
467     //     minTime = std::min(minTime, (*curr)->getReceiveTime());
468     // }
469     return minTime;
470 }
471
472 double
473 xxxx::HeapBottom::getBucketWidth() const {
474     if (empty()) {
475         return 0;
476     }
477     const double minTS = front()->getReceiveTime();
478     return (maxEvtTime - minTS + size() - 1.0) / sel.size();
479 }
480
481 bool
482 xxxx::HeapBottom::haveBefore(const Time recvTime) const {
483     if (empty()) {
484         return false;
485     }
486     for (auto curr = sel.begin(); (curr != sel.end()); curr++) {
487         if ((*curr)->getReceiveTime() <= recvTime) {
488             return true;
489         }
490     }
491     return false;
492 }
493
494 void
495 xxxx::HeapBottom::validate() {
496     // Heap's are implemented using standard C++ algorithms and should
497     // be compatible with them.
498     ASSERT(std::is_heap(sel.begin(), sel.end(), Bottom::compare));
499 }
500
501 void
502 xxxx::HeapBottom::print(std::ostream& os) const {
503     os << "Bottom:";
504     for (auto curr = sel.begin(); (curr != sel.end()); curr++) {
505         os << " " << (*curr)->getReceiveTime();
506     }
507     os << std::endl;
508 }
509
510 // ----------------------[ MultiSetBottom methods ]--------------------------
511
512 void
513 xxxx::MultiSetBottom::enqueue(Bucket&& bucket) {
514     // Note that pop_front must be O(1) here -- which it is since
515     // bucket is a linked list.
516     // Add all the events to the conainer.
517     while (!bucket.empty()) {
518         sel.insert(bucket.pop_front());
519     }
520 }
521
522 void
523 xxxx::MultiSetBottom::enqueue(xxxx::Event* event) {
524     ASSERT(sel.empty() || (front()->getReceiveTime() <= findMinTime()));
525     sel.insert(event);
```



```
526 }
527
528 xxxx::Event*
529 xxxx::MultiSetBottom::pop_front() {
530     xxxx::Event* retVal = front();
531     sel.erase(sel.begin());
532     return retVal;
533 }
534
535 int
536 xxxx::MultiSetBottom::remove_after(xxxx::AgentID sender, const Time sendTime) {
537     // Since MutliSet events are sorted based on receive time there is
538     // only simple sanity checks we can do here...
539     if (sendTime > maxTime()) {
540         // Since max event time is greater than sentTime the bottom
541         // cannot have an event to be cancelled.
542         return 0;
543     }
544     size_t removedCount = 0;
545     EventMultiSet::iterator currIdx = sel.begin();
546     while (!sel.empty() && (currIdx != sel.end())) {
547         xxxx::Event* const event = *currIdx;
548         ASSERT(event != NULL);
549         // An event is deleted only if the *sent* time is greater than
550         // the sendTime and if the event is from same sender
551         if ((event->getSenderAgentID() == sender) &&
552             (event->getSentTime() >= sendTime)) {
553             // This event needs to be removed.
554             event->decreaseReference();
555             removedCount++;
556             currIdx = sel.erase(currIdx);
557         } else {
558             // Check the next event to see it is a candidate for
559             // cancellation.
560             currIdx++;
561         }
562     }
563     // Return number of events canceled to track statistics.
564     return removedCount;
565 }
566
567 int
568 xxxx::MultiSetBottom::remove(xxxx::AgentID receiver) {
569     size_t removedCount = 0;
570     EventMultiSet::iterator currIdx = sel.begin();
571     while (!sel.empty() && (currIdx != sel.end())) {
572         Event* const event = *currIdx;
573         if (event->getReceiverAgentID() == receiver) {
574             event->decreaseReference();
575             removedCount++;
576             currIdx = sel.erase(currIdx);
577         } else {
578             currIdx++;
579         }
580     }
581     return removedCount;
582 }
583
584 void
585 xxxx::MultiSetBottom::dequeueNextAgentEvents(xxxx::EventContainer& events) {
586     if (sel.empty()) {
587         return;
588     }
589
590     const xxxx::Event*  nextEvt    = front();
591     const xxxx::AgentID receiver   = nextEvt->getReceiverAgentID();
592     const xxxx::Time    currTime   = nextEvt->getReceiveTime();
593
594     do {
595         xxxx::Event* event = pop_front();
596         events.push_back(event);
597         nextEvt = (!empty() ? front() : NULL);
598         DEBUG(std::cout << "Delivering: " << *event << std::endl);
599     } while (!empty() && (nextEvt->getReceiverAgentID() == receiver) &&
600              TIME_EQUALS(nextEvt->getReceiveTime(), currTime));
```

```
601     DEBUG(validate());
602 }
603
604 xxxx::Time
605 xxxx::MultiSetBottom::maxTime() const {
606     if (empty()) {
607         return TIME_INFINITY;
608     }
609     xxxx::Time maxTime = (*sel.crbegin())->getReceiveTime();
610     ASSERT(maxTime >= front()->getReceiveTime());
611     return maxTime;
612 }
613
614 xxxx::Time
615 xxxx::MultiSetBottom::findMinTime() const {
616     if (empty()) {
617         return 0;
618     }
619     xxxx::Time minTime = front()->getReceiveTime();
620     ASSERT(minTime <= maxTime());
621     return minTime;
622 }
623
624 double
625 xxxx::MultiSetBottom::getBucketWidth() const {
626     if (empty()) {
627         return 0;
628     }
629     const double maxTS = (*sel.rbegin())->getReceiveTime();
630     const double minTS = (*sel.begin())->getReceiveTime();
631     return (maxTS - minTS + size() - 1.0) / sel.size();
632 }
633
634 bool
635 xxxx::MultiSetBottom::haveBefore(const Time recvTime) const {
636     if (empty()) {
637         return false;
638     }
639     for (auto curr = sel.begin(); (curr != sel.end()); curr++) {
640         if ((*curr)->getReceiveTime() <= recvTime) {
641             return true;
642         }
643     }
644     return false;
645 }
646
647 void
648 xxxx::MultiSetBottom::validate() {
649     // MultiSet are implemented using standard C++ algorithms and
650     // consequently no special validation is deemed necessary.
651 }
652
653 void
654 xxxx::MultiSetBottom::print(std::ostream& os) const {
655     os << "MultiSetBottom:";
656     for (auto curr = sel.begin(); (curr != sel.end()); curr++) {
657         os << " " << (*curr)->getReceiveTime();
658     }
659     os << std::endl;
660 }
661
662 // ----------------------------[  Rung methods ]-----------------------------
663
664 xxxx::Rung::Rung(Top& top) : Rung(std::move(top.events), top.minTS,
665                                   top.getBucketWidth()) {
666     // Reset of top counters and update the values of topStart for
667     // next Epoch is now done in populateBottom.
668 }
669
670 xxxx::Rung::Rung(Bucket&& bkt, const Time minTS, const double bktWidth) :
671     rStartTS(minTS), rCurrTS(minTS), bucketWidth(bktWidth), currBucket(0),
672     rungEventCount(0) {
673     // Initialize variable to track maximum bucket count
674     LQ_STATS(maxBkts = 0);
675     DEBUG(std::cout << "bucketWidth = " << bucketWidth << std::endl);
```



```
676     ASSERT(bucketWidth > 0);
677     ASSERT(rungEventCount == 0);
678     // Move events from given bucket into buckets in this Rung.
679     DEBUG(std::cout << "Adding " << bkt.size() << " events to rung\n");
680     while (!bkt.empty()) {
681         // Remove event from the top linked list.
682         xxxx::Event* event = bkt.pop_front();
683         // Add to the appropriate bucket in this rung
684         enqueue(event);
685     }
686     DEBUG(validateEventCounts());
687 }
688
689 xxxx::Rung::Rung(EventVector&& list, const Time minTS, const double bktWidth) :
690     rStartTS(minTS), rCurrTS(minTS), bucketWidth(bktWidth), currBucket(0),
691     rungEventCount(0) {
692     // Initialize variable to track maximum bucket count
693     LQ_STATS(maxBkts = 0);
694     DEBUG(std::cout << "bucketWidth = " << bucketWidth << std::endl);
695     ASSERT(bucketWidth > 0);
696     ASSERT(rungEventCount == 0);
697     // Move events from given bucket into buckets in this Rung.
698     DEBUG(std::cout << "Adding " << list.size() << " events to rung\n");
699     for (auto curr = list.begin(); (curr != list.end()); curr++) {
700         // Add to the appropriate bucket in this rung
701         enqueue(*curr);
702     }
703     // Clear out entries in the supplied list
704     list.clear();
705     DEBUG(validateEventCounts());
706 }
707
708 xxxx::Rung::Rung(EventMultiSet&& set, const Time minTS, const double bktWidth) :
709     rStartTS(minTS), rCurrTS(minTS), bucketWidth(bktWidth), currBucket(0),
710     rungEventCount(0) {
711     // Initialize variable to track maximum bucket count
712     LQ_STATS(maxBkts = 0);
713     DEBUG(std::cout << "bucketWidth = " << bucketWidth << std::endl);
714     ASSERT(bucketWidth > 0);
715     ASSERT(rungEventCount == 0);
716     // Move events from given set into buckets in this Rung.
717     DEBUG(std::cout << "Adding " << list.size() << " events to rung\n");
718     for (auto curr = set.begin(); (curr != set.end()); curr++) {
719         // Add to the appropriate bucket in this rung
720         enqueue(*curr);
721     }
722     // Clear out entries in the supplied list
723     set.clear();
724     DEBUG(validateEventCounts());
725 }
726
727 bool
728 xxxx::Rung::canContain(xxxx::Event* event) const {
729     const xxxx::Time recvTime = event->getReceiveTime();
730     const int bucketNum = (recvTime - rStartTS) / bucketWidth;
731     return ((bucketNum >= (int) currBucket) && (recvTime >= rStartTS));
732 }
733
734 void
735 xxxx::Rung::enqueue(xxxx::Event* event) {
736     ASSERT(event->getReceiveTime() >= getCurrTime());
737     // Compute bucket for this event based on equation #2 in paper.
738     size_t bucketNum = (event->getReceiveTime() - rStartTS) / bucketWidth;
739     ASSERT(bucketNum >= currBucket);
740     if (bucketNum >= bucketList.size()) {
741         // Ensure bucket list of sufficient size
742         bucketList.resize(bucketNum + 1);
743         // update variable to track maximum bucket count
744         LQ_STATS(maxBkts = std::max(maxBkts, bucketList.size()));
745     }
746     ASSERT(bucketNum < bucketList.size());
747     // Add event into appropriate bucket
748     bucketList[bucketNum].push_front(event);
749     // Track number of events added to this Rung
750     rungEventCount++;
```

```
751  }
752
753  xxxx::Bucket&&
754  xxxx::Rung::removeNextBucket(xxxx::Time& bktTime) {
755      ASSERT(!empty());
756      ASSERT(currBucket < bucketList.size());
757      // Find next non-empty bucket in this rung (there has to be one as
758      // the previous asserts passed necessary checks)
759      while ((currBucket < bucketList.size()) && bucketList[currBucket].empty()) {
760          currBucket++;
761      }
762      DEBUG(validateEventCounts());
763      ASSERT(currBucket < bucketList.size());
764      ASSERT(!bucketList[currBucket].empty());
765      // Track events that will be removed when this method returns
766      rungEventCount -= bucketList[currBucket].size();
767      ASSERT(rungEventCount >= 0);
768      // Save information about the bucket to be removed & returned.
769      const int retBkt = currBucket;
770      bktTime = rStartTS + (retBkt * bucketWidth);
771      // Advance current bucket to next time.
772      currBucket++;
773      rCurrTS = rStartTS + (currBucket * bucketWidth);
774      // Sanity check on counters...
775      if (currBucket >= bucketList.size()) {
776          ASSERT(rungEventCount == 0);
777      }
778      return std::move(bucketList[retBkt]);
779  }
780
781  int
782  xxxx::Rung::remove_after(xxxx::AgentID sender, const Time sendTime
783                           LQ_STATS(COMMA Avg& ceScanRung)) {
784      if (empty() || (sendTime > getMaxRungTime())) {
785          return 0;
786      }
787      int numRemoved   = 0;
788      for (size_t bucketNum = currBucket; (bucketNum < bucketList.size());
789           bucketNum++) {
790          if (!bucketList[bucketNum].empty() &&
791              (rStartTS + (bucketNum + 1) * bucketWidth) >= sendTime) {
792              LQ_STATS(ceScanRung += bucketList[bucketNum].size());
793              numRemoved += bucketList[bucketNum].remove_after(sender, sendTime);
794          }
795      }
796      rungEventCount -= numRemoved;
797      DEBUG(validateEventCounts());
798      return numRemoved;
799  }
800
801  int
802  xxxx::Rung::remove(xxxx::AgentID receiver
803                     LQ_STATS(COMMA Avg& ceScanRung)) {
804      if (empty()) {
805          return 0;
806      }
807      int numRemoved = 0;
808      for (size_t bucketNum = currBucket; (bucketNum < bucketList.size());
809           bucketNum++) {
810          if (!bucketList[bucketNum].empty()) {
811              LQ_STATS(ceScanRung += bucketList[bucketNum].size());
812              numRemoved += bucketList[bucketNum].remove(receiver);
813          }
814      }
815      rungEventCount -= numRemoved;
816      DEBUG(validateEventCounts());
817      return numRemoved;
818  }
819
820  void
821  xxxx::Rung::validateEventCounts() const {
822      int numEvents = 0;
823      for (const auto& bucket : bucketList)  {
824          numEvents += bucket.size();
825      }
```



```
826      if (numEvents != rungEventCount) {
827          DEBUG(std::cout << "Rung event count mismatch! Expecting: "
828                          << rungEventCount << " events, but found: "
829                          << numEvents << "." << std::endl);
830          ASSERT(numEvents == rungEventCount);
831      }
832  }
833
834  void
835  xxxx::Rung::prettyPrint(std::ostream& os) const {
836      // Compute minimum, maximum, empty, and average bucket sizes.
837      size_t minBkt = -1U, maxBkt = 0, emptyBkt = 0, sizeSum = 0;
838      for (const Bucket& bkt : bucketList) {
839          if (!bkt.empty()) {
840              minBkt   = std::min(minBkt, bkt.size());
841              maxBkt   = std::max(maxBkt, bkt.size());
842              sizeSum += bkt.size();
843          } else {
844              emptyBkt++;
845          }
846      }
847      double avgBktSz = sizeSum / (double) (bucketList.size() - emptyBkt);
848      os << "start time="    << rStartTS    << ", curr time=" << rCurrTS
849         << ", bkt. width=" << bucketWidth << ", bkt count=" << bucketList.size()
850         << ", curr buckt=" << currBucket  << ", events="    << rungEventCount
851         << ", min bkt="    << minBkt      << ", maxBkt="    << maxBkt
852         << ", empty bkt="  << emptyBkt    << ", avg size="  << avgBktSz
853         << std::endl;
854  }
855
856  void
857  xxxx::Rung::updateStats(Avg& avgBktCnt) const {
858      LQ_STATS(avgBktCnt += maxBkts);
859  }
860
861  bool
862  xxxx::Rung::haveBefore(const Time recvTime) const {
863      for (size_t i = 0; (i < bucketList.size()); i++) {
864          if (bucketList[i].haveBefore(recvTime)) {
865              return true;
866          }
867      }
868      return false;
869  }
870
871  // ------------------------[  LadderQueue methods ]----------------------
872
873  // The maximum number of rungs typically allowed in the ladder.  This
874  // value is set to 8 by default based on Tang et. al.  It can be set
875  // via command-line parameter --lq-max-rungs 8.
876  size_t xxxx::LadderQueue::MaxRungs = 8;
877
878  xxxx::LadderQueue::~LadderQueue() {
879      // Nothing else to be dne here.
880  }
881
882  void
883  xxxx::LadderQueue::reportStats(std::ostream& os) {
884      UNUSED_PARAM(os);
885      LQ_STATS({
886              // Collect final bucket counts from the ladder
887              for (size_t i = 0; (i < nRung); i++) {
888                  ladder[i].updateStats(avgBktCnt);
889              }
890              // Compute net number of compares for ladderQ
891              // const long comps = log2(botLen.getMean()) * botLen.getSum();
892              // std::make_heap has 3N time complexity.
893              const long comps = 3 * botLen.getSum() +
894                  log2(botLen.getMean()) * botLen.getSum() / 3;
895              os << "Events cancelled from top  : "    << ceTop
896                 << "\nEvents scanned in top     : " << ceScanTop
897                 << "\nEvents cancelled from ladder: " << ceLadder
898                 << "\nEvents scanned from ladder : " << ceScanLadder
899                 << "\nEvents cancelled from bottom: " << ceBot
900                 << "\nEvents scanned from bottom : " << ceScanBot
```

```
                << "\nNo cancel scans of bottom  : " << ceNoCanScanBot
                << "\nInserts into top           : " << insTop
                << "\nInserts into rungs         : " << insLadder
                << "\nInserts into bottom        : " << insBot
                << "\nMax rung count             : " << maxRungs
                << "\nAverage #buckets per rung  : " << avgBktCnt
                << "\nAverage bottom size        : " << botLen
                << "\nMax bottom size            : " << maxBotSize
                << "\nAverage bucket width       : " << avgBktWidth
                << "\nBottom to rung operations  : " << botToRung
                << "\nCompare estimate           : " << comps
                << std::endl;
    });
}

void
xxxx::LadderQueue::enqueue(xxxx::Event* event) {
    if (top.getStartTime() < event->getReceiveTime()) {
        DEBUG(std::cout << "Added to top: " << *event << std::endl);
        top.add(event);
        LQ_STATS(insTop++);
        return;
    }
    // Try to see if the event fits in the ladder
    size_t rung = 0;
    while ((rung < nRung) && !ladder[rung].canContain(event)) {
        ASSERT((rung == 0) || ladder[rung].empty() ||
               (ladder[rung - 1].getCurrTime() >= ladder[rung].getCurrTime()));
        rung++;
    }
    if (rung < nRung) {
        DEBUG(ASSERT(bottom.empty() ||
                     (event->getReceiveTime() > bottom.maxTime())));
        ladder[rung].enqueue(event);
        ladderEventCount++;  // Track events added to the ladder
        DEBUG(std::cout << "Added to rung " << rung  << "(max bottom: "
                        << bottom.maxTime() << "): " << *event << "\n");
        LQ_STATS(insLadder++);
        return;
    }
    // Event does not fit in the ladder. Must go into bottom
    if ((bottom.size() > THRESH) && (bottom.getTimeRange() > 0)) {
        // Move events from bottom into ladder rung
        rung = createRungFromBottom();
        ASSERT(rung == nRung - 1);
        ASSERT(rung < ladder.size());
        // Due to rollback-reprocessing the event may be even
        // earlier than the last rung we just created!
        if (ladder[rung].canContain(event)) {
            ladder[rung].enqueue(event);
            ladderEventCount++;
            DEBUG(std::cout << "Added to rung " << rung  << "(max bottom: "
                            << bottom.maxTime() << "): " << *event << "\n");
            LQ_STATS(insLadder++);
            return;
        }
    }
    // At this point bottom must be able to contain the event, so
    // enqueue it.
    bottom.enqueue(event);
    LQ_STATS(maxBotSize = std::max(maxBotSize, bottom.size()));
    DEBUG(ASSERT(!haveBefore(bottom.front()->getReceiveTime())));
    DEBUG(std::cout << "Added to bottom: " << *event << std::endl);
    LQ_STATS(insBot++);
    // Check used only when detailed debugging is enabled.
    DEBUG({ if (bottom.size() > 50000) {
                std::cerr << "Warning: Bottom is very large: size: "
                          << bottom.size() << ".  Min event timestamp: "
                          << bottom.findMinTime()
                          << ", Max event time: " << bottom.maxTime()
                          << ", ladder size: " << ladder.size()
                          << " with events: " << ladderEventCount
                          << std::endl;
                prettyPrint(std::cerr);
                ASSERT(ladder.empty());
```



```
            }
        });
}

int
xxxx::LadderQueue::createRungFromBottom() {
    ASSERT(!bottom.empty());
    ASSERT(bottom.getTimeRange() > 0);
    ASSERT(!ladder.empty() || (nRung > 0));
    DEBUG(std::cout << "Moving events from bottom to a new rung. Bottom has "
                    << bottom.size() << " events." << std::endl);
    // Compute the start time and bucket width for the rung.  Note
    // that with rollbacks, ladder can be empty and that situation
    // needs to be handled.
    const double bucketWidth = (ladder.empty() ? bottom.getBucketWidth() :
                                ladder[nRung - 1].getBucketWidth());
    // The paper computes rStart as RCur[NRung-1].  However, due to
    // rollback-reprocessing the bottom may have events that are below
    // RCur[NRung-1].  Consequently, we use the minimum of the two
    // values as as rstart
    const Time ladBkTime = ((nRung > 0) ? ladder[nRung - 1].getCurrTime() :
                            TIME_INFINITY);
    const Time rStart = std::min(ladBkTime, bottom.front()->getReceiveTime());
    // Create a new rung and add it to the ladder.
    DEBUG(std::cout << "Moving bottom to rung. Events: " << bottom.size()
                    << ", rStart = " << rStart << ", bucketWidth = "
                    << (bucketWidth / bottom.size()) << std::endl);
    ladderEventCount += bottom.size();  // Update ladder event count
    LQ_STATS(botToRung += bottom.size());
    const double bktWidth = (bucketWidth + bottom.size() - 1.0) / bottom.size();
    DEBUG(std::cout << "bktWidth = " << bktWidth << std::endl);
    // Add rung and move move bottom into the last rung of the ladder.
    nRung++;
    if (nRung > ladder.size()) {
        ladder.push_back(Rung(std::move(bottom.sel), rStart, bktWidth));
        ASSERT(nRung == ladder.size());
    } else {
        ladder[nRung - 1] = Rung(std::move(bottom.sel), rStart, bktWidth);
    }
    DEBUG(std::cout << "1. Bucket width: " << bktWidth << std::endl);
    LQ_STATS(avgBktWidth += bktWidth);
    LQ_STATS(maxRungs = std::max(maxRungs, nRung));  // Track max rungs
    ASSERT(bottom.empty());
    return nRung - 1;
}

xxxx::Bucket&
xxxx::LadderQueue::recurseRung() {
    ASSERT(!empty());
    ASSERT(nRung > 0);
    ASSERT(!ladder.empty());
    // Now the last rung in ladder is the rung that has the next
    // bucket of events.
    xxxx::Time bktTime = 0;  // set by removeNextBucket call below
    Rung& lastRung     = ladder[nRung - 1];
    Bucket&& bkt       = lastRung.removeNextBucket(bktTime);
    ASSERT(!bkt.empty());
    ASSERT(!ladder.empty());
    // Check and create new rung in the ladder if the bucket is large.
    if ((bkt.size() > THRESH) && (nRung < MaxRungs)) {
        // Note: Here bucket width can dip low. But that is needed to
        // ensure consistent ladder setup.
        const double bucketWidth = (lastRung.getBucketWidth() + bkt.size() -
                                    1.0) / bkt.size();
        // Create a new rung in the ladder
        nRung++;
        if (nRung > ladder.size()) {
            ladder.push_back(Rung(std::move(bkt), bktTime, bucketWidth));
        } else {
            ladder[nRung - 1] = Rung(std::move(bkt), bktTime, bucketWidth);
        }
        DEBUG(std::cout << "2. Bucket width: " << bucketWidth << std::endl);
        LQ_STATS(avgBktWidth += bucketWidth);
        LQ_STATS(maxRungs = std::max(maxRungs, nRung));
        return recurseRung();  // Recurse now looking at newly added rung
```

```
    }
    // Track events being removed from the ladder
    ladderEventCount -= bkt.size();
    ASSERT(ladderEventCount >= 0);
    // Return bucket being removed.
    return std::move(bkt);
}

xxxx::Event*
xxxx::LadderQueue::dequeue() {
    if (empty()) {
        return NULL;
    }
    if (!bottom.empty()) {
        xxxx::Event* retVal = bottom.pop_front();
        if (empty()) {
            // The whole queue is now empty. Reset the structure to
            // accomodate new events.
            top.reset();
        }
        return retVal;
    }
    // Since bottom is empty move events into bottom.
    populateBottom();
    ASSERT(!bottom.empty());
    return bottom.pop_front();
}

void
xxxx::LadderQueue::populateBottom() {
    if (!bottom.empty()) {
        return;
    }
    if (ladderEventCount == 0) {   // NRung == 0
        if (top.empty()) {
            // There are no events in the ladder queue in this case
            ASSERT(empty());
            return;
        }
        // Move all events from top into buckets in first rung of the ladder!
        nRung++;
        ASSERT(nRung == 1);
        ladderEventCount += top.size();  // Track events in ladder
        // Move events to ladder
        if (nRung > ladder.size()) {
            ladder.push_back(Rung(top));
            ASSERT(nRung == ladder.size());
        } else {
            ladder[nRung - 1] = Rung(top);
        }
        // Reset top counters and update the values of topStart for
        // next Epoch
        top.reset(top.getMaxTime());
        LQ_STATS(maxRungs = std::max(maxRungs, nRung));
        LQ_STATS(avgBktWidth += ladder.back().getBucketWidth());
        DEBUG(std::cout << "3. Bucket width: "
                        << ladder.back().getBucketWidth() << std::endl);
        DEBUG(prettyPrint(std::cout));
        ASSERT(top.empty());
    }
    // Bottom is empty. So we need to move events from the current
    // bucket in the ladder to bottom.
    ASSERT(!ladder.empty());
    ASSERT(bottom.empty());
    bottom.enqueue(recurseRung());  // Transfer bucket_k into bottom
    LQ_STATS(maxBotSize = std::max(maxBotSize, bottom.size()));
    ASSERT(!bottom.empty());
    LQ_STATS(maxBotSize = std::max(maxBotSize, bottom.size()));
    DEBUG(ASSERT(!haveBefore(bottom.front()->getReceiveTime())));
    LQ_STATS(botLen += bottom.size());
    // Clear out the rungs if we have used-up the last bucket in the ladder.
    while ((nRung > 0) && ladder[nRung - 1].empty()) {
        LQ_STATS(ladder[nRung - 1].updateStats(avgBktCnt));
        nRung--;  // Logically remove rung from ladder
    }
```



```
}

int
xxxx::LadderQueue::remove_after(xxxx::AgentID sender, const Time sendTime) {
    LQ_STATS(ceScanTop += top.size());
    int numRemoved = top.remove_after(sender, sendTime);
    LQ_STATS(ceTop += numRemoved);
    // Cancel out events in each active rung of the ladder.
    for (size_t rung = 0; (rung < nRung); rung++) {
        const int rungEvtRemoved =
            ladder[rung].remove_after(sender, sendTime
                                      LQ_STATS(COMMA ceScanLadder));
        ladderEventCount  -= rungEvtRemoved;
        numRemoved        += rungEvtRemoved;
        LQ_STATS(ceLadder += rungEvtRemoved);
    }
    // Clear out the rungs in ladder that are now empty after event
    // cancellations.
    while (nRung > 0 && ladder[nRung - 1].empty()) {
        LQ_STATS(ladder[nRung - 1].updateStats(avgBktCnt));
        nRung--;  // Logically remove rung from ladder
    }
    // Remove events from bottom.  If the return value is -1, the scan
    // of bottom was quickly short circuited and no events needed to
    // be canceled.
    // Save original size of bottom to track stats.
    LQ_STATS(const size_t botSize  = bottom.size());
    const int botRemoved = bottom.remove_after(sender, sendTime);
    if (botRemoved > -1) {
        numRemoved         += botRemoved;
        LQ_STATS(ceBot     += botRemoved);
        LQ_STATS(ceScanBot += botSize);
        LQ_STATS((botRemoved == 0) ? (ceNoCanScanBot += botSize) : 0);
    }
    return numRemoved;
}

bool
xxxx::LadderQueue::haveBefore(const Time recvTime,
                              const bool checkBottom) const {
    if (top.haveBefore(recvTime)) {
        std::cout << "Top has event that is <= " << recvTime << std::endl;
        prettyPrint(std::cout);
        return true;
    }
    for (size_t rung = 0; (rung < nRung); rung++) {
        if (ladder[rung].haveBefore(recvTime)) {
            std::cout << "Rung #" << rung << " has event that is <= "
                      << recvTime << std::endl;
            prettyPrint(std::cout);
            return true;
        }
    }

    if (checkBottom && bottom.haveBefore(recvTime)) {
        std::cout << "Bottom has event that is <= " << recvTime << std::endl;
        prettyPrint(std::cout);
        return true;
    }
    return false;
}

// ------------------------[  EventQueue implementation ]----------------------

void*
xxxx::LadderQueue::addAgent(xxxx::Agent* agent) {
    UNUSED_PARAM(agent);
    return NULL;
}

void
xxxx::LadderQueue::removeAgent(xxxx::Agent* agent) {
    ASSERT( agent != NULL );
    const AgentID receiver = agent->getAgentID();
    // Remove events for agent from top
```

```
1201      LQ_STATS(ceScanTop += top.size());
1202      int numRemoved    = top.remove(receiver);
1203      LQ_STATS(ceTop += numRemoved);
1204      // Next remove events for agent from all the rungs in the ladder
1205      for (Rung& rung : ladder) {
1206          int rungEvtRemoved = rung.remove(agent->getAgentID()
1207                                           LQ_STATS(COMMA ceBot));
1208          ladderEventCount  -= rungEvtRemoved;
1209          numRemoved        += rungEvtRemoved;
1210          LQ_STATS(ceLadder += rungEvtRemoved);
1211      }
1212      // Finally remove events from bottom for the agent.
1213      // Save original size of bottom to track stats.
1214      LQ_STATS(const size_t botSize  = bottom.size());
1215      const int botRemoved = bottom.remove(receiver);
1216      LQ_STATS(ceScanBot  += botSize);
1217      LQ_STATS((botRemoved == 0) ? (ceNoCanScanBot += botSize) : 0);
1218      numRemoved          += botRemoved;
1219      LQ_STATS(ceBot      += botRemoved);
1220  }
1221
1222
1223  xxxx::Event*
1224  xxxx::LadderQueue::front() {
1225      if (empty()) {
1226          // Nothing to return.
1227          return NULL;
1228      }
1229      if (bottom.empty()) {
1230          populateBottom();
1231          DEBUG(prettyPrint(std::cout));
1232      }
1233      ASSERT(!bottom.empty());
1234      return bottom.front();
1235  }
1236
1237  void
1238  xxxx::LadderQueue::dequeueNextAgentEvents(xxxx::EventContainer& events) {
1239      if (empty()) {
1240          // No events to dequeue.
1241          return;
1242      }
1243      // We only dequeue from bottom. So ensure it has events in it.
1244      if (bottom.empty()) {
1245          populateBottom();
1246      }
1247      ASSERT(!bottom.empty());
1248      bottom.dequeueNextAgentEvents(events);
1249      ASSERT(!events.empty());
1250      DEBUG(ASSERT(!haveBefore(events.front()->getReceiveTime())));
1251  }
1252
1253  void
1254  xxxx::LadderQueue::enqueue(xxxx::Agent* agent, xxxx::Event* event) {
1255      UNUSED_PARAM(agent);
1256      event->increaseReference();
1257      enqueue(event);
1258  }
1259
1260  void
1261  xxxx::LadderQueue::enqueue(xxxx::Agent* agent, xxxx::EventContainer& events) {
1262      UNUSED_PARAM(agent);
1263      for (auto& curr : events) {
1264          enqueue(curr);
1265      }
1266      events.clear();
1267  }
1268
1269  int
1270  xxxx::LadderQueue::eraseAfter(xxxx::Agent* dest, const xxxx::AgentID sender,
1271                                const xxxx::Time sentTime) {
1272      UNUSED_PARAM(dest);
1273      return remove_after(sender, sentTime);
1274  }
1275
```



```
1276  void
1277  xxxx::LadderQueue::prettyPrint(std::ostream& os) const {
1278      // Print information on top.
1279      os << "Top: Events=" << top.size()
1280         << ", startTime=" << top.getStartTime()
1281         << ", minTime="   << top.getMinTime()
1282         << ", maxTime="   << top.getMaxTime()  << std::endl;
1283      // Print info on each rung of the ladder
1284      std::cout << "Ladder (rungs=" << nRung << ", size="
1285                << ladder.size() << "):\n";
1286      for (size_t i = 0; (i < nRung); i++) {
1287          os << "[" << i << "]: ";
1288          ladder[i].prettyPrint(os);
1289      }
1290      // Print info on bottom
1291      os << "Bottom: Events=" << bottom.size()
1292         << ", min="           << (!bottom.empty() ? bottom.findMinTime() : -1.0)
1293         << ", max="           << (!bottom.empty() ? bottom.maxTime()     : -1.0)
1294         << std::endl;
1295  }
1296
1297  #endif
```

```
#ifndef TWO_TIER_LADDER_QUEUE_H
#define TWO_TIER_LADDER_QUEUE_H

#include <forward_list>
#include <queue>
#include <vector>
#include <typeinfo>
#include <set>
#include "Avg.h"
#include "Event.h"
#include "EventQueue.h"

/** \file LadderQueue.h

    \brief Enhancement of LadderQueue to improve performance of
    Optimistic Parallel Simulations by minimizing rollbacks.

    The LadderQueue data structure is detailed in the following paper:

    W. Tang, R. Goh, and I. Thng, "Ladder queue: An O(1) priority
    queue structure for large-scale discrete event simulation", ACM
    TOMACS, Vol 15, Issue 3, Pages 175--204, July 2005. URL:
    http://doi.acm.org/10.1145/1103323.1103324

    <p>One major disadvantage of the LadderQueue is that canceling
    events due to a rollback is expensive -- the whole queue has to be
    scanned.</p>

    <p>In order to reduce the overhead of scanning events for
    canceling, this TwoTierLadderQueue further subdivides each bucket
    in Top and ladder Rung to store events in a 2nd Tier based on
    their sender's ID.  It uses a simple hash function on the sender's
    AgentID to identify the 2nd tier bucket and enqueue's the event
    into that bucket.  Currently, the hash function is simply
    implemented as a modulo t2k, with t2k being an implementation
    dependent value.  Since second tier buckets (implemented as
    std::vector) are preallocated, Small t2k values increase 2nd tier
    bucket sizes increasing time for cancellation.  On the other hand
    if buckets are not used, then the space/time invested to create
    them can become an overhead.</p>

    <p>Note that the TwoTierLadderQueue would have very similar
    characteristics to LadderQueue in sequential or 1 process
    simulation as there are no rollbacks</p>
*/

// Bucket size after which new rung is created in ladder
#define LQ2T_THRESH 50

/** \def LQ2T_STATS(x)

    \brief Define a convenient macro for conditionally compiling
    additional statistics collection regarding ladder queue.

    Define a custom macro LQ2T_STATS (note the all caps) macro to be
    used to conditionally compile in debugging code to generate
    detailed logs.  This helps to minimize code modification to insert
    and remove debugging messages.
*/
#define COMMA ,
#define LQ2T_STATS(x) x
// #define LQ2T_STATS(x)

BEGIN_NAMESPACE(xxxx)

/** A convenience alias for list of events maintained by a sub-bucket. */
using BktEventList = std::vector<xxxx::Event*>;

/** Alias to the data structure for holding a vector of sub-buckets in
    a TwoTierBucket.
*/
using SubBucketList = std::vector<BktEventList>;

constexpr bool SenderID   = true;
constexpr bool ReceiverID = false;
```



```

/** A generic two tier bucket that is used for both Top and Rungs of
    the 2-tier ladderQ.

    <p>This bucket does not store events in it directly.  Instead it
    splits into t2k sub-buckets based on a simple hash function.
    Currently, the hash function is simply implemented as a modulo
    t2k, with t2k being an implementation dependent value.  Splitting
    the events into sub-buckets makes event cancellations easier.
    </p>
*/
class TwoTierBucket {
public:
    /** The shared parameter indicating the number of sub-buckets to
        be used in each 2-tier bucket.  This value defaults to 32.  It
        is overriden by by command-line argument when ladder queue is
        used.
    */
    static int t2k;

    /** Constructor to create a bucket with fixed number (i.e., t2k)
        of tier-2 lists.
    */
    TwoTierBucket() : subBuckets(t2k), count(0) {}

    /** A move constructor to facilitate moving objects (if needed).

        \param[in,out] src The source object whose data is to be moved
        into this.  The source object does not contain any useful
        information after the move is complete.
    */
    TwoTierBucket(TwoTierBucket&& src) : subBuckets(std::move(src.subBuckets)),
                                         count(std::move(src.count)) {
        // Reset count in source to aid debugging.
        src.count = 0;
    }

    /** The destructor for this class.

        The destructor decreases the reference count on all the events
        in its list to free-up any pending events.
    */
    ~TwoTierBucket();

    /** The hash function used to distribute events into sub-buckets.

        \param[in] sender The sender's ID to be hashed.

        \return The hash value based on sender ID.  The return value,
        say hash, must be in the range 0 <= hash < t2k.
    */
    inline int hash(const xxxx::AgentID sender) const {
        // Use a simple hashing function for now.
        return (sender % t2k);
    }

    /** Add an event to this TwoTier bucket based on sender's ID

        This method is a template specialization to use the sender's
        ID for hashing to find sub-bucket. The event is added to the
        sub-bucket identified using the hash function in this class.

        \param[in] event The event to be added to this bucket.  This
        method does not alter the refernece counts on events (as the
        top-level TwoTierLadderQueue performs the reference count
        management).
    */
    template <bool Sendr, typename std::enable_if<Sendr>::type* = nullptr>
    void push_back(xxxx::Event* event) {
        const size_t subBktIdx = hash(event->getSenderAgentID());
        subBuckets[subBktIdx].push_back(event);
        count++;
    }

    /** Add an event to this TwoTier bucket based on receiver's ID
```

```
151
152         This method is a template specialization to use the receiver's
153         ID for hashing to find sub-bucket. The event is added to the
154         sub-bucket identified using the hash function in this class.
155
156         \param[in] event The event to be added to this bucket.  This
157         method does not alter the refernece counts on events (as the
158         top-level TwoTierLadderQueue performs the reference count
159         management).
160     */
161     template <bool Recvr, typename std::enable_if<!Recvr>::type* = nullptr>
162     void push_back(xxxx::Event* event) {
163         const size_t subBktIdx = hash(event->getReceiverAgentID());
164         subBuckets[subBktIdx].push_back(event);
165         count++;
166     }
167
168     /** Move all the events from the given two tier bucket into this
169         bucket.
170
171         This method moves all the events from the sub-buckets in
172         srcBkt to corresponding sub-buckets in this.
173
174         \param[in,out] srcBkt The bucket from where the events are to
175         be moved into this bucket.
176     */
177     void push_back(TwoTierBucket&& srcBkt);
178
179     /** Helper method to move all events from a 2-tier bucket into a
180         single list of events.
181
182         This method is a convenience method that is used by the bottom
183         tier to combine all the events from various sub-buckets into a
184         single list of events.
185
186         \param[out] dest The destination event list to which all the
187         events are to added.
188
189         \param[in,out] srcBkt The bucket from where the events are to
190         be moved.  After this call the srcBkt will not have any events
191         in it.
192     */
193     static void push_back(BktEventList& dest, TwoTierBucket&& srcBkt);
194
195     /** Obtain the count of events which includes the events in sub-buckets.
196
197         \return The sum of events in all of the sub-buckets.
198     */
199     size_t size() const {
200         ASSERT( getEventCount() == count );
201         return count;
202     }
203
204     /** Convenience method to determine if the bucket is empty.
205
206         \return This method returns true if this bucket does not
207         contain any events in it.
208      */
209     bool empty() const {
210         ASSERT( getEventCount() == count );
211         return (count == 0);
212     }
213
214     /** Convenience method to remove all events sent by the sender
215         at-or-after the given send Time.
216
217         This method removes computes the sub-bucket that contains all
218         the events for this sender and removes events from that
219         sub-bucket.
220
221         \param[in] sender The sender agent whose events are to be
222         removed.
223
224         \param[in] sendTime The time at-or-after which events from the
225         sender are to be removed from the given list.
```



```
226
227         \param[in,out] scans Statistics object (if stats is enabled)
228         to track number of events scanned.
229
230         \return This method returns the number of events that were
231         removed.
232     */
233     int remove_after(xxxx::AgentID sender, const Time sendTime
234                      LQ2T_STATS(COMMA Avg& scans));
235
236     /** Remove all events in this bucket for a given receiver agent
237         ID.
238
239         This is a convenience method that removes all events for a
240         given receiver agent in this bucket.  This method is used to
241         remove events scheduled for an agent, when an agent is removed
242         from the scheduler.  This method has to search through all the
243         sub-buckets because the condition is based on receiver (and
244         not sender).
245
246         \param[in] receiver The receier ID whose events are to be
247         removed from the sub-buckets.
248
249         \return This method returns the number of events removed.
250     */
251     int remove(xxxx::AgentID receiver);
252
253     /** This method is purely for troubleshooting one scenario
254         where an event would get stuck in the ladder and not get
255         scheduled correctly.
256
257         \param[in] recvTime The time to be used for checking to see if
258         sub-buckets have an event before this time.
259
260         \return Returns true if an event before this receiveTime (for
261         any agent) is pending in a sub-bucket.
262     */
263     bool haveBefore(const Time recvTime) const;
264
265     /** Convenience method to remove all events from a given sender
266         that were sent at-or-after the given sendTime.
267
268         This method linearly scans the given event list, checks, and
269         removes all events that were sent by the sender at-or-after
270         the specified send time.
271
272         \note This method assumes unsorted list of events and does not
273         preserve order of events if an event is cancelled -- this is
274         because Events to be removed are moved to the back and popped
275         to reduce deletion time.
276
277         \param[in,out] list The list of events from where all events
278         for the sender are to be removed.  This method linearly scans
279         through this list.  If events are removed, the order of events
280         in the list is not preserved.
281
282         \param[in] sender The sender agent whose events are to be
283         removed.
284
285         \param[in] sendTime The time at-or-after which events from the
286         sender are to be removed from the given list.
287
288         \return This method returns the number of events that were
289         removed.
290     */
291     static int remove_after(BktEventList& list, xxxx::AgentID sender,
292                             const xxxx::Time sendTime);
293
294     /** Remove all events in a given event list for a given receiver
295         agent ID.
296
297         This is a convenience/helper method that removes all events
298         for a given receiver agent in a given sub-bucket/list.  This
299         method is used to remove events scheduled for an agent, when
300         an agent is removed from the scheduler.  This method has to
```

```
301         search through all the events in the list to remove them.
302
303         \param[in] subBkt The sub-bucket or list from where events are
304         to be removed.
305
306         \param[in] receiver The receier ID whose events are to be
307         removed from the sub-buckets.
308
309         \return This method returns the number of events removed.
310     */
311     static int remove(BktEventList& list, xxxx::AgentID receiver);
312
313     /** Obtain a reference to the list of sub-buckets in this bucket.
314
315         \return Mutable reference to the list of sub-buckets in this
316         bucket.
317     */
318     SubBucketList& getSubBuckets() { return subBuckets; }
319
320     /** Convenience method to reset count of events in this bucket to
321         zero.
322
323         This method is used in TwoTierRung operations to reset the
324         events in this bucket to zero, after events have been moved
325         out of this bucket.
326     */
327     void resetCount() {
328         count = 0;
329     }
330
331 protected:
332     /** Return sum of events in each sub-bucket.
333
334         This method is used purely for validation/debugging.  This
335         method iterates over each sub-bucket in the list and returns
336         the actual count of events.  This value must be consistent
337         with the value in the count instance variable.
338
339         \return The actual sum of events in various sub-buckets.
340     */
341     size_t getEventCount() const;
342
343     /** Clear out all the events in this bucket.
344
345         This method clears out all the events in various sub-buckets
346         in this 2-tier bucket.  It also sets count to zero.
347
348         \note This method decreases references on any pending events.
349     */
350     void clear();
351
352 private:
353     /** The list of tier-2 sub-buckets that contain events distributed
354         based on the hash of the receiver's ID.
355     */
356     SubBucketList subBuckets;
357
358     /** The total number of events in all of the sub-buckets.  This
359         information is primary used to quickly respond to the size()
360         method calls
361     */
362     size_t count;
363 };
364
365 /** The class that forms the Top rung of a 2-tier ladder queue.
366
367     The top-rung of the 2-tier ladder queue behaves similar to the
368     ladder queue with respect to managing time stamps.  However, the
369     organization is different -- events are not stored in a linear
370     list.  Instead they are stored in sub-buckets based on a hash of
371     the sender agent's ID.
372
373     \note Do not call push_back directly.  Instead use the add method
374     in this class to add events.
375 */
```



```
376 class TwoTierTop : public TwoTierBucket {
377     friend class TwoTierRung;
378     friend class TwoTierLadderQueue;
379 public:
380     /** Construct and initialize top to empty state.
381
382         The constructor uses a convenience method in this class to
383         reset the timestamps to zero.
384     */
385     TwoTierTop() {
386         reset();
387     }
388
389     /** The destructor
390
391         Currently the destructor has nothing much to do as the base
392         class does all of the necessary clean-ups.
393      */
394     ~TwoTierTop() {}
395
396     /** Method to add events to top and update current minimum and
397         maximum time stamp values.
398
399         \param[in] event The event to be added to the top.  This
400         pointer cannot be NULL.
401     */
402     void add(xxxx::Event* event);
403
404     /** Return the current start-time for top.
405
406         \note This value changes when events are added/removed. So
407         don't think about caching this value.
408
409         \return The current start time. This value is used for
410         scheduling events and creating rungs.
411     */
412     Time getStartTime() const { return topStart; }
413
414     /** Returns the minimum timestamp of events in this rung.
415
416         \note This value changes when events are added/removed. So
417         don't think about caching this value.
418
419         \return The minimum timestamp of events in this rung.
420     */
421     Time getMinTime() const { return minTS; }
422
423     /** Returns the maximum timestamp of events in this rung.
424
425         \note This value changes when events are added/removed. So
426         don't think about caching this value.
427
428         \return The maximum event timestamp in this rung.
429     */
430     Time getMaxTime() const { return maxTS; }
431
432     /** Convenience method to determine if given time is within the
433         <i>current</i> minmum and maximum time.
434
435         \param[in] ts The timestamp value to be checked.
436
437         \return This method returns true if getMinTime() <= ts <=
438         getMaxTime().  Otherwise it returns false.
439      */
440     bool contains(const Time ts) const {
441         return (ts >= minTS) && (ts <= maxTS);
442     }
443
444     /** Convenience method compute the bucket size for the top-level
445         rung of the TwoTierLadder queue.
446
447         \return The suggested bucket width (in terms of time) for the
448         top-level rung of the TwoTierLadder.
449     */
450     double getBucketWidth() const {
```

```
451         DEBUG(std::cout << "minTS=" << minTS << ", maxTS=" << maxTS
452               << ", size=" << size() << std::endl);
453         return std::max((maxTS - minTS + size() - 1.0) / size(), 0.01);
454     }
455
456 protected:
457     /** Helper method to reset top either during construction or
458         whenever it is emptied to move events into the ladder.
459
460         \param[in] topStart An optional start time for the top rung.
461     */
462     void reset(const Time topStart = 0);
463
464 private:
465     /** Instsance variable to track the current minimum timestamp of
466         events in top.  This value changes each time a new event is
467         added to the top via the add emthod.
468     */
469     xxxx::Time minTS;
470
471     /** Instsance variable to track the current maximum timestamp of
472         events in top.  This value changes each time a new event is
473         added to the top via the add emthod.
474     */
475     xxxx::Time maxTS;
476
477     /** Instsance variable to track the last time top was reset.  This
478         is used for debugging/troubleshooting purposes.
479     */
480     xxxx::Time topStart;
481 };
482
483 /** The bottom most rung of the TwoTierLadder queue.  The bottom rung
484     is the same as that of the standard ladder queue.  However, in
485     2-tier ladder queue, the size of the bottom has been relaxed.  So
486     bottom can be pretty long.  This implies that 2-tier ladder queue
487     will not be O(1).  It will be O(n log n).  However, it should
488     perform just fine as the ladder queue.
489
490     \note In XXXX we have an API requirement/guarantee that all the
491     concurrent events we have will be scheduled simultaneously.  This
492     eases agent development in many applications.  Consequently, it is
493     imperative that bottom be allowed to be long to contain all
494     concurrent events.
495
496     \note Do not use front() / back() to access the first event in
497     bottom. Instead use the first_event() method.
498 */
499 class OneTierBottom : public BktEventList {
500 public:
501     /** The default and only constructor.  It does not have any
502         special work to do as the base class handles most of the
503         tasks.
504     */
505     OneTierBottom() {}
506
507     /** Add events from a TwoTierBucket into the bottom.
508
509         This method is used to bulk move events from a rung of the
510         ladder (or top) into the bottom.  The events are added and
511         sorted in preparation for scheduling.
512
513         \param bucket The 2-tier bucket from where events are to be
514         moved into the bottom rung.
515     */
516     void enqueue(TwoTierBucket&& bucket);
517
518     /** Add a single event to the bottom rung.
519
520         This method uses binary-search (O(log n)) to insert an event
521         into the bottom.
522
523         \param[in] event The event to be added to the bottom rung.
524         This pointer cannot be NULL.  No operations are done on the
525         reference-counters in this method.
```



```
526     */
527     void enqueue(xxxx::Event* event);
528
529     /** Convenience method to dequeue events after a given time.
530
531         \param[in] sender The sender agent whose events are to be
532         removed.
533
534         \param[in] sendTime The time at-or-after which events from the
535         sender are to be removed from the given list.
536     */
537     int remove_after(xxxx::AgentID sender, const Time sendTime);
538
539     /** Remove all events for a given receiver agent in the bucket
540         encapsulated by this object.
541
542         This is a convenience method that removes all events for a
543         given receiver agent in this object.  This method is used to
544         remove events scheduled for an agent, when an agent is removed
545         from the scheduler.
546     */
547     int remove(xxxx::AgentID receiver) {
548         // Use static convenience method do to this task.
549         return TwoTierBucket::remove(*this, receiver);
550     }
551
552     /** Convenience method used to dequee the next set of events for
553         scheduling.
554
555         This method is used to provide necessary implemntation to
556         interface with the XXXX scheduler.  This method dequeues the
557         next batch of the concurrent events for processing by a given
558         agent.
559
560         \param[out] events The container to which all the events to be
561         processed is to be added.
562     */
563     void dequeueNextAgentEvents(xxxx::EventContainer& events);
564
565     /** Convenience method for debugging/troubleshooting.
566
567         \return The highest timestamp from the events in the bottom.
568         If no events are present this method returns TIME_INFINITY.
569     */
570     xxxx::Time maxTime() const {
571         // purely for debugging
572         return (!empty() ? front()->getReceiveTime() : TIME_INFINITY);
573     }
574
575     /** Convenience method for debugging/troubleshooting.
576
577         \return The minimum timestamp from the events in the bottom.
578         If no events are present this method returns TIME_INFINITY.
579     */
580     xxxx::Time findMinTime() const {
581         // purely for debugging
582         return (!empty() ? back()->getReceiveTime() : TIME_INFINITY);
583     }
584
585     /** Method to determine the range of receive time values currently
586         in bottom.  This value is typically used to decide if it is
587         worth moving events from bottom into the ladder.
588
589         \return The difference in maximum and minimum receive
590         timestamp of events in the bottom.  This value is zero if all
591         events have the same receive time.  If the bottom is empty,
592         then this method also returns zero.
593     */
594     xxxx::Time getTimeRange() const {
595         if (empty()) {
596             return 0;
597         }
598         return (front()->getReceiveTime() - back()->getReceiveTime());
599     }
600
```

```
601     /** Determine bucket width to move bottom into ladder.
602
603         This method is invoked only when the ladder is empty and the
604         bottom is long and needs to be moved into the ladder.  This
605         method must compute and return the preferred bucket width.
606
607         \note If the bottom is empty this method returns bucket width
608         of 0.
609      */
610     double getBucketWidth() const;
611
612     /** Convenience method to check if the entries in the bottom are
613         sorted correctly.  This method is purely used for
614         troubleshooting/debugging.
615     */
616     void validate() const;
617
618     /** Event comparison function used by various structures in ladder
619         queue.
620
621         \param[in] lhs The left-hand-side event for comparison.  The
622         pointer cannot be NULL.
623
624         \param[in] lhs The right-hand-side event for comparison.  The
625         pointer cannot be NULL.
626
627         \return This method returns true if lhs is less than rhs.
628         That is, lhs should be scheduled before rhs.
629     */
630     static inline bool compare(const xxxx::Event* const lhs,
631                                const xxxx::Event* const rhs) {
632         return ((lhs->getReceiveTime() > rhs->getReceiveTime()) ||
633                 ((lhs->getReceiveTime() == rhs->getReceiveTime() &&
634                   (lhs->getReceiverAgentID() > rhs->getReceiverAgentID()))));
635     }
636
637     /** Convenience method to check to see if bottom has events before
638         the specified receive time.
639
640         This method is used for troubleshooting/debugging only.
641
642         \param[in] recvTime The receive time for checking.
643
644         \return Returns true if an event before this receiveTime (for
645         any agent) is pending in the bottom rung.
646     */
647     bool haveBefore(const Time recvTime) const {
648         return (findMinTime() <= recvTime);
649     }
650
651     /** Convenience method to consistently access the first event in
652         the bottom, consistent with the way bottom is sorted.
653
654         \note Calling this method when the bottom is empty has
655         undefined behavior.
656
657         \return The next event with the lowest time stamp.
658     */
659     xxxx::Event* first_event() {
660         return back();
661     }
662
663 protected:
664     // Currently this class does not have any protected members.
665
666 private:
667     // Currently this class does not have any private members
668 };
669
670 /** Class that represents one rung in the 2-tier ladder queue.
671
672     The 2-tier rung uses the same strategy for receive time-based
673     bucket creation as the regular ladder queue.  However, the
674     organization of each bucket is different -- events are not stored
675     in a linear list.  Instead they are stored in sub-buckets based on
```



```
676     a hash of the sender agent's ID.
677 */
678 class TwoTierRung {
679 public:
680     /** The constructor to create an empty rung.
681
682         The constructor merely initializes all the instance variables
683         to default initial values to create an empty rung.
684     */
685     TwoTierRung() : rStartTS(TIME_INFINITY), rCurrTS(TIME_INFINITY),
686                     bucketWidth(0), currBucket(0), rungEventCount(0) {
687         LQ2T_STATS(maxBkts = 0);
688     }
689
690     /** A move constructor required to quickly move rungs in a ladder
691         to shrink/grow it.
692
693         \param[in] src The source rung from where events are to be
694         copied.
695     */
696     TwoTierRung(TwoTierRung&& src) :
697         rStartTS(src.rStartTS), rCurrTS(src.rCurrTS),
698         bucketWidth(src.bucketWidth), currBucket(src.currBucket),
699         bucketList(std::move(src.bucketList)),
700         rungEventCount(src.rungEventCount) {
701         LQ2T_STATS(maxBkts = src.maxBkts);
702     }
703
704     /** Convenience constructor to create a rung using events from the
705         top rung.
706
707         This is a delegating constructor that delegates the actual
708         tasks to the overloaded constructor.
709
710         \param[in] top The top bucket from where the events are to be
711         created.
712     */
713     explicit TwoTierRung(TwoTierTop&& top) :
714         TwoTierRung(std::move(top), top.getMinTime(),
715                     top.getBucketWidth()) {
716             // Reset of top counters etc. is done by caller in
717             // TwoTierLadderQueue::populateBottom()
718     }
719
720     /** Convenience constructor to create a rung with events from a
721         given bucket.
722
723         \param[in,out] bkt The bucket from where events are to be
724         moved into this newly created rung.  After this operation data
725         in the bucket is cleared.
726
727         \param[in] rStart The start time for this rung.
728
729         \param[in] bucketWidth The delta in receive time for each
730         bucket in this rung.  The bucketWidth must be > 0.
731     */
732     TwoTierRung(TwoTierBucket&& bkt, const Time rStart,
733                 const double bucketWidth) : rungEventCount(0) {
734         move(std::move(bkt), rStart, bucketWidth);
735     }
736
737     /** Convenience method initialize a rung by moving events from a
738         given bucket.
739
740         It is assumed that this rung is empty prior to this operation.
741
742         \param[in,out] bkt The bucket from where events are to be
743         moved into this rung.  After this operation data in the bucket
744         is cleared.
745
746         \param[in] rStart The start time for this rung.
747
748         \param[in] bucketWidth The delta in receive time for each
749         bucket in this rung.  The bucketWidth must be > 0.
750     */
```

```
751     void move(TwoTierBucket&& bucket, const Time rStart,
752               const double bucketWidth);
753
754     /** Convenience constructor to create a rung with events from the
755         bottom rung.
756
757         This operation is used to redistribute bottom to the ladder
758         ensures that the bottom does not get too long.
759
760         \param[in,out] bottom The bottom rung from where events are to
761         be moved into this newly created rung.  After this operation
762         bottom will be empty.
763
764         \param[in] rStart The start time for this rung.
765
766         \param[in] bucketWidth The delta in receive time for each
767         bucket in this rung.  The bucketWidth must be > 0.
768     */
769     TwoTierRung(OneTierBottom&& bottom, const Time rStart,
770                 const double bucketWidth) : rungEventCount(0) {
771         move(std::move(bottom), rStart, bucketWidth);
772     }
773
774     /** Convenience method to create a rung with events from the
775         bottom rung.
776
777         This operation is used to redistribute bottom to the ladder
778         ensures that the bottom does not get too long.  This method
779         assumes that the this rung is empty to begin with.
780
781         \param[in,out] bottom The bottom rung from where events are to
782         be moved into this newly created rung.  After this operation
783         bottom will be empty.
784
785         \param[in] rStart The start time for this rung.
786
787         \param[in] bucketWidth The delta in receive time for each
788         bucket in this rung.  The bucketWidth must be > 0.
789     */
790     void move(OneTierBottom&& bottom, const Time rStart,
791               const double bucketWidth);
792
793     /** Remove the next bucket in this rung for moving to another rung
794         in the ladder.
795
796         This method must be used to remove the next bucket from this
797         rung.  The bucket is logically removed (or moved) out of this
798         rung.
799
800         \param[out] bktTime The simulation receive time associated
801         with the bucket being moved out.
802     */
803     TwoTierBucket&& removeNextBucket(xxxx::Time& bktTime);
804
805     /** Determine if this rung is empty.
806
807         This is a convenience method that is used to determine if this
808         rung contains any events to be processed.
809
810         \return This method returns true if the rung does not have any
811         events -- i.e., when the rung is empty.
812     */
813     bool empty() const { return (rungEventCount == 0); }
814
815     /** Add an event to suitable bucket in this rung.
816
817         This method computes a bucket index (based on equation #2 in
818         LQ paper) using the formula:
819
820         \code
821         size_t bucketNum = (event->getReceiveTime() - rStartTS) / bucketWidth;
822         \endcode
823
824         \param[in] event The event to be added to a suitable bucket in
825         this rung.
```



```
826     */
827     void enqueue(xxxx::Event* event);
828
829     /** Obtain the start time for this rung.
830
831         This method returns the rung starting time that was set when
832         this rung was created.
833
834         \return The starting time of this rung that determines the
835         lowest timestamp event that can be added to this rung.
836     */
837     xxxx::Time getStartTime() const { return rStartTS; }
838
839     /** Obtain the bucket width (i.e., difference in receive times for
840         adjacent buckets) for this rung.
841
842         This method returns the bucket width that was set when this
843         rung was created.
844
845         \return The bucket with for this rung.
846     */
847     double getBucketWidth() const { return bucketWidth; }
848
849     /** The current bucket value in this ladder queue.
850
851         The current minimum time of events that can be added to this
852         rung of the ladder eueue.
853
854         \return The minimum timestamp of events that can be added to
855         the rung of this ladder queue.
856     */
857     xxxx::Time getCurrTime() const {
858         return rCurrTS;
859     }
860
861     /** The maximum receive time value of event that can be added to
862         this rung.
863
864         \return The maximum receive time of an event that can be added
865         to a bucket in this rung.
866     */
867     xxxx::Time getMaxRungTime() const {
868         return rStartTS + (bucketList.size() * bucketWidth);
869     }
870
871     /** Convenience method to determine if a given event can be added
872         to this rung.
873
874         \param[in] event The event whose receive time is to be used to
875         check to see if it can be added to this ladder.
876
877         \return Returns true if the event can be added to this rung.
878         Otherwise it returns false.
879     */
880     bool canContain(xxxx::Event* event) const;
881
882     /** Remove all events from the given sender sent at-or-after the
883         specified send time from all buckets in this rung.
884
885         This method linearly scans the buckets, checks, and removes
886         all events that were sent by the sender at-or-after the
887         specified send time.
888
889         \param[in] sender The sender agent whose events are to be
890         removed.
891
892         \param[in] sendTime The time at-or-after which events from the
893         sender are to be removed from the given list.
894
895         \param[out] ceScanRung The stats object to be updated with
896         number of events scanned in the buckets in this rung.
897
898         \return This method returns the total number of events that
899         were removed from this rung.
900     */
```

```
901     int remove_after(xxxx::AgentID sender, const Time sendTime
902                      LQ2T_STATS(COMMA Avg& ceScanRung));
903
904     /** Remove all events for a given receiver agent in this rung.
905
906         This is a convenience method that removes all events for a
907         given receiver agent in this rung.  This method is used to
908         remove events scheduled for an agent, when an agent is removed
909         from the scheduler.
910
911         \param[in] receiver The receiving agent ID whose events are to
912         be removed from all the buckets in this rugn.
913
914         \param[out] ceScanRung The stats object to be updated with
915         number of events scanned in the buckets in this rung.
916     */
917     int remove(xxxx::AgentID receiver
918                LQ2T_STATS(COMMA Avg& ceScanRung));
919
920     /** Check to ensure that the number of events in various buckets
921         matches the count instance variable.
922
923         This method is used only for troubleshooting/debugging
924         purposes.  If counts don't match then assert fails in this
925         method causing the simulation to abort.
926     */
927     void validateEventCounts() const;
928
929     /** Print a user-friendly version of the events in this queue.
930
931         Currently this method is not implemented.
932     */
933     void prettyPrint(std::ostream& os) const;
934
935     /** Update the statistics object with data from this rung.
936
937         \param[out] avgBktCnt Update the average number of buckets in
938         this rung.
939     */
940     void updateStats(Avg& avgBktCnt) const;
941
942     /** Convenience method to determine if the current bucket in this
943         rung is empty.
944
945         \return This method returns true if the current bucket in this
946         rung is empty.
947     */
948     bool isCurrBucketEmpty() const {
949         return (currBucket >= bucketList.size() ||
950                 bucketList[currBucket].empty());
951     }
952
953     /** This method is purely for troubleshooting one scenario where
954         an event would get stuck in the ladder and not get scheduled
955         correctly.
956
957         \param[in] recvTime The time to be used for checking to see if
958         sub-buckets have an event before this time.
959
960         \return Returns true if an event before this receiveTime (for
961         any agent) is pending in a sub-bucket.
962     */
963     bool haveBefore(const Time recvTime) const;
964
965 protected:
966     // Currently this class does not have any protected members.
967
968 private:
969     /** The lowest timestamp event that can be added to this rung.
970         This value is set when a rung is created and is never changed
971         during the lifetime of this rung.
972     */
973     xxxx::Time rStartTS;
974
975     /** The timestamp of the lowest event that can be currently added
```



```
976         to this rung.  This value logically starts with rStartTS and
977         grows to the time stamp of last bucket in this rung as buckets
978         are dequeued from this rung.
979     */
980     xxxx::Time rCurrTS;
981
982     /** The width of the bucket in simulation receive time
983         differences.  This value can be fractional.
984     */
985     double bucketWidth;
986
987     /** The index of the current bucket on this rung to which events
988         can be added.  This is also the next bucket that will be
989         dequeued from the rung.
990     */
991     size_t currBucket;
992
993     /** The deque containing the set of vectors in this bucket list.
994      */
995     std::deque<TwoTierBucket> bucketList;
996
997     /** Total number of events still present in this rung.  This is
998         used to report size and check for empty quickly.
999     */
1000    int rungEventCount;
1001
1002    /** Statistics object to track the maximum number of buckets used
1003        in this rung */
1004    LQ2T_STATS(size_t maxBkts);
1005 };
1006
1007 /** The top-level 2-tier ladder queue
1008
1009     <p>This class represents the top-level 2-tier ladder queue class
1010     that interfaces with the XXXX scheduler.  This class implements
1011     the top-level logic associated with ladder queue to enqueue,
1012     dequeue, and cancel events from the ladder queue.</p>
1013
1014     <p>The logic for most of the operations is consistent with those
1015     proposed by the Tang et. al, except for the following:
1016
1017     <ol>
1018
1019     <li>The size of the bottom is not restricted.  So events are never
1020     moved from bottom back into the ladder.</li>
1021
1022     <li> The number of buckets in a rung is restricted to 100</li>
1023
1024     </ol>
1025
1026     </p>
1027 */
1028 class TwoTierLadderQueue : public EventQueue {
1029 public:
1030     /** The constructor that creates an empty ladder queue.
1031
1032         The constructor also initializes various statistics variables
1033         used by the this queue to report detailed statistics about its
1034         operations at the end of simulation.
1035      */
1036     TwoTierLadderQueue() : EventQueue("LadderQueue"), nRung(0),
1037                            ladderEventCount(0) {
1038         ladder.reserve(MaxRungs);
1039         LQ2T_STATS(ceTop    = ceLadder   = ceBot  = 0);
1040         LQ2T_STATS(insTop   = insLadder  = insBot = 0);
1041         LQ2T_STATS(maxRungs = maxBotSize = 0);
1042     }
1043
1044     /** The destructor.
1045
1046         Currently the destructor does not have anything special to do
1047         as the different encapsulated objects handle all the necessary
1048         clean-up.
1049     */
1050     ~TwoTierLadderQueue() {}
```

```
1051
1052        /** Enqueue an event into the laadder queue.
1053
1054            Depending on the scenario the event is appropriately added to
1055            one of: top, ladder rung, or the bottom.
1056
1057            \param[in] e The event to be enqueued for scheduling in the
1058            ladder queue.
1059        */
1060        void enqueue(xxxx::Event* e);
1061
1062        /** Cancel all events from a given sender that were sent
1063            at-or-after the specified send time.
1064
1065            This method essentially calls the corresponding method(s) in
1066            top, rung, and bottom to cancel pending events.
1067
1068            \param[in] sender The sender agent whose events are to be
1069            removed.
1070
1071            \param[in] sendTime The time at-or-after which events from the
1072            sender are to be removed from the given list.
1073
1074            \return This method returns the number of events that were
1075            removed.
1076        */
1077        int remove_after(xxxx::AgentID sender, const Time sendTime);
1078
1079        /** Determine if the ladder queue is empty.
1080
1081            Implements the interface method used by XXXX::Scheduler.
1082
1083            \return Returns true if top, ladder, and bottom are all empty
1084            -- i.e., there are no pending events.
1085        */
1086        virtual bool empty() {
1087            return top.empty() && (ladderEventCount == 0) &&  bottom.empty();
1088        }
1089
1090        /** Implementation for method used by XXXX::Scheduler.
1091
1092            This method is called by XXXX kernel to inform the scheduler
1093            queue about an agent being added during initialization.  The
1094            ladder queue does not utilize this information and
1095            consequently this method does not have any special operation
1096            to perform.
1097
1098            \param[in] agent The agent being added.  This pointer is not
1099            really used.
1100
1101            \return This method simply returns nullptr as the ladder queue
1102            does not use any cross references in xxxx::Agent for its
1103            operations.
1104        */
1105        virtual void* addAgent(xxxx::Agent* agent);
1106
1107        /** Remove an agent just before simulation completes.
1108
1109            This method is invoked by the XXXX kernel to inform that an
1110            agent is being removed.  This method removes all pending
1111            events for the specified agent from the ladder queue.
1112
1113            \param[in] agent The agent whose sender ID is used to remove
1114            all pending events in the top, rungs, and bottom.
1115        */
1116        virtual void removeAgent(xxxx::Agent* agent);
1117
1118        /** Implement interface method to peek at the next event to
1119            schedule.
1120
1121            \note In order to enable peeking of the front event, the
1122            bottom may need to get populated.
1123
1124            \return A pointer to the next event to schedule (if any).  The
1125            event is not dequeued.
```



```
1126         */
1127        virtual xxxx::Event* front();
1128
1129        /** This method is used to provide necessary implemntation to
1130            interface with the XXXX scheduler.  This method dequeues the
1131            next batch of the concurrent events for processing by a given
1132            agent.
1133
1134            \param[out] events The container to which all the events to be
1135            processed is to be added.
1136        */
1137        virtual void dequeueNextAgentEvents(xxxx::EventContainer& events);
1138
1139        /** Add an event to be scheduled to this ladder queue.
1140
1141            This method implements the core API used by agents to schedule
1142            events for each other.
1143
1144            \param[in] agent The receiver agent for which the event is
1145            scheduled.  This pointer is not used.
1146
1147            \param[in] event The event to be scheduled. This simply calls
1148            the overloaded enqueue method.  The reference count on the
1149            event is increased by this method to account for this event
1150            being present in the ladder queue.
1151        */
1152        virtual void enqueue(xxxx::Agent* agent, xxxx::Event* event);
1153
1154        /** Enqueue a batch of events
1155
1156            This API to schedule a block of events.  This API is typically
1157            used after a rollback.
1158
1159            \param[in] agent The receiver agent for which the event is
1160            scheduled.  This pointer is not used.
1161
1162            \param[in] events The list of events to be scheduled. This
1163            simply calls the overloaded enqueue method to enqueue one
1164            event at a time.
1165        */
1166        virtual void enqueue(xxxx::Agent* agent, xxxx::EventContainer& events);
1167
1168        /** Implement XXXX kernel API to cancel all events sent by a given
1169            agent after a given time.
1170
1171            \param[in] dest The destination agent whose events are to be
1172            cancelled.
1173
1174            \param[in] sender The sender agent ID whose events are to be
1175            cancelled.
1176
1177            \param[in] sentTime The send time at-or-after which all events
1178            from the sender are to be cancelled.
1179        */
1180        virtual int eraseAfter(xxxx::Agent* dest, const xxxx::AgentID sender,
1181                               const xxxx::Time sentTime);
1182
1183        /** Print a human understandable version of the events in this
1184            queue.
1185
1186            currently this method is not implemented.
1187         */
1188        virtual void prettyPrint(std::ostream& os) const;
1189
1190
1191        /** Convenience method to check to see if ladder queue has events
1192            before the specified receive time.
1193
1194            This method is used for troubleshooting/debugging only.
1195
1196            \param[in] recvTime The receive time for checking.
1197
1198            \return Returns true if an event before this receiveTime (for
1199            any agent) is pending in the bottom rung.
1200        */
```

```
1201      bool haveBefore(const Time recvTime,
1202                      const bool checkBottom = false) const;
1203
1204      /** Method to report aggregate statistics.
1205
1206          This method is invoked at the end of simulation after all
1207          agents on this rank have been finalized.  This method is meant
1208          to report any aggregate statistics from this queue.  This
1209          method writes statistics only if LQ2T_STATS macro is enabled.
1210
1211          \param[out] os The output stream to which the statistics are
1212          to be written.
1213      */
1214      virtual void reportStats(std::ostream& os);
1215
1216      /** The maximum number of rungs that are normally created in the
1217          ladder queue.  The default value for this set to 8 based on
1218          the value suggested by Tang et. al. in the original Ladder
1219          Queue paper.  However, this value can make some difference in
1220          the overall performance and possibly fine tuned to suit the
1221          application needs based on the concurrency and number of
1222          events in the model.
1223      */
1224      static size_t MaxRungs;
1225
1226  protected:
1227      /** Check and create rungs in the ladder and return the next
1228          bucket of events from the ladder.
1229
1230          This method implements the corresponding recurseRung method
1231          from the LQ paper. Refer to the paper for the details.
1232      */
1233      TwoTierBucket&& recurseRung();
1234
1235      /** This is a convenience method that is used to move events from
1236          the ladder into bottom.
1237
1238          This method moves events from teh current bucket in the last
1239          rung of the ladder into the bottom.  If this method is called
1240          when bottom is not empty, it does not perform any operation
1241          and returns immediately.  If the ladder does not have any
1242          events, but the top has events, then this method first moves
1243          events from top-rung into the ladder and then removes events
1244          from the last rung into the bottom-rung.
1245      */
1246      void populateBottom();
1247
1248      /** Method to create a new ladder rung from the current bottom.
1249
1250          This method should be called only when the following 2
1251          conditions are met:
1252
1253          1. Length of bottom is > LQ2T_THRESH
1254
1255          2. The bottom has events that are at different time stamps --
1256             that is bottom.getTimeRange() > 0.
1257
1258          \return This method returns the index of the rung created so
1259          that the caller can readily work with that rung.
1260      */
1261      int createRungFromBottom();
1262
1263  private:
1264      TwoTierTop top;
1265
1266      /** The ladder in the queue.  The lader consists of a set of
1267          rungs.  The currently used rung in the ladder is indicated by
1268          the nRung instance variable.  If the ladder is empty, then
1269          nRung is (or should be) 0
1270      */
1271      std::vector<TwoTierRung> ladder;
1272
1273      /** The currently used last rung in the ladder queue.  If the
1274          ladder is empty, then nRung is (or should be) 0.  Otherwise
1275          this value is (or should be) in the range 0 < nRung <=
```



```
1276          ladder.size().  Rungs below nRung are not used and they do not
1277          contain any events to be scheduled.
1278      */
1279      size_t nRung;
1280
1281      /** Instance variable to track the current number of pending
1282          events in all the rungs of the ladder.  This is a convenience
1283          instance variable to quickly detect pending events in the
1284          ladder without having to iterate through each rung.
1285      */
1286      int ladderEventCount;
1287
1288      OneTierBottom bottom;
1289
1290      LQ2T_STATS(Avg ceTop);
1291      LQ2T_STATS(Avg ceBot);
1292      LQ2T_STATS(Avg ceLadder);
1293
1294      LQ2T_STATS(Avg ceScanTop);
1295      LQ2T_STATS(Avg ceScanLadder);
1296
1297      /** The ceScanBot statistic tracks size of bottom rung scanned
1298          when at one (or more) events were canceled from bottom.
1299      */
1300      LQ2T_STATS(Avg ceScanBot);
1301
1302      /** The ceNoCanScanBot statistic tracks size of bottom rung
1303          scanned but did not cancel any events.
1304      */
1305      LQ2T_STATS(Avg ceNoCanScanBot);
1306
1307      LQ2T_STATS(int insTop);
1308      LQ2T_STATS(int insLadder);
1309      LQ2T_STATS(int insBot);
1310      LQ2T_STATS(size_t maxRungs);
1311      LQ2T_STATS(Avg avgBktCnt);
1312      LQ2T_STATS(Avg botLen);
1313      LQ2T_STATS(Avg avgBktWidth);
1314
1315      /** Gague to track the number of events and times bottom was
1316          redistributed to the last rung of the ladder.
1317
1318          Redistributing bottom to the ladder ensures that the bottom
1319          does not get too long.  But it is an expensive operation
1320          because all the sorting that was done is lost.  So it is a
1321          balance and we track and report this number for reference.
1322      */
1323      LQ2T_STATS(Avg botToRung);
1324
1325      /** Gauge to track the maximum length of bottom.  The length of
1326          bottom plays an important role in the overall performance of
1327          the ladder queue.
1328      */
1329      LQ2T_STATS(size_t maxBotSize);
1330  };
1331
1332
1333  END_NAMESPACE(xxxx)
1334
1335  #endif
```

```cpp
#ifndef TWO_TIER_LADDER_QUEUE_CPP
#define TWO_TIER_LADDER_QUEUE_CPP

#include <algorithm>
#include <functional>
#include "TwoTierLadderQueue.h"

// The maximum number of buckets 1 rung can have.
#define MAX_BUCKETS 100

/** The number of sub-buckets to be used in each 2-tier bucket */
int xxxx::TwoTierBucket::t2k = 32;

// ----------------------[ TwoTierBucket methods ]----------------------

xxxx::TwoTierBucket::~TwoTierBucket() {
    clear();
}

void
xxxx::TwoTierBucket::clear() {
    for (BktEventList& subBkt : subBuckets) {
        for (xxxx::Event* event : subBkt) {
            event->decreaseReference();
        }
    }
    count = 0;
}

void
xxxx::TwoTierBucket::push_back(TwoTierBucket&& srcBkt) {
    ASSERT(srcBkt.subBuckets.size() == (size_t) t2k);
    ASSERT(srcBkt.subBuckets.size() == subBuckets.size());
    // Move evens from srcBkt into corresponding sub-buckets
    for (int idx = 0; (idx < t2k); idx++) {
        // Obtain reference to subbucket to be moved.
        BktEventList& src  = srcBkt.subBuckets[idx];
        BktEventList& dest = subBuckets[idx];
        // Move events from src to dest.
        dest.insert(dest.end(), src.begin(), src.end());
        // Update counters (also used to troubleshooting).
        count += src.size();
        // Clear out the source as events have been logically moved
        // out of it.
        src.clear();
    }
}

void
xxxx::TwoTierBucket::push_back(BktEventList& dest, TwoTierBucket&& srcBkt) {
    // Move all entries from each sub-bucket in srcBkt to the end of dest.
    for (BktEventList& subBkt : srcBkt.subBuckets) {
        dest.insert(dest.end(), subBkt.begin(), subBkt.end());
        subBkt.clear();
    }
    // Reset count as part of move semantics
    srcBkt.count = 0;
}

int
xxxx::TwoTierBucket::remove_after(xxxx::AgentID sender, const Time sendTime
                                  LQ2T_STATS(COMMA Avg& scans)) {
    const size_t subBktIdx = hash(sender);
    LQ2T_STATS(scans += subBuckets[subBktIdx].size());
    int removedCount = remove_after(subBuckets[subBktIdx], sender, sendTime);
    count -= removedCount;  // Track remaining events
    return removedCount;
}

// Helper method to remove events from a sub-bucket.
int
xxxx::TwoTierBucket::remove_after(BktEventList& list, xxxx::AgentID sender,
                                  const Time sendTime) {
    size_t removedCount = 0;
    size_t curr = 0;
```



```cpp
    while (curr < list.size()) {
        xxxx::Event* const event = list[curr];
        if ((event->getSenderAgentID() == sender) &&
            (event->getSentTime() >= sendTime)) {
            // Free-up event.
            event->decreaseReference();
            removedCount++;
            // To minimize removal time replace entry with last one
            // and pop the last entry off.
            list[curr] = list.back();
            list.pop_back();
        } else {
            curr++;  // on to the next event in the list
        }
    }
    return removedCount;
}

// This method is not performance critical as it is only called once
// at the end of simulation.
int
xxxx::TwoTierBucket::remove(xxxx::AgentID receiver) {
    size_t removedCount = 0;
    // Remove events from each sub-bucket.
    for (BktEventList& list : subBuckets) {
        // Use helper method to remove events.
        removedCount += remove(list, receiver);
    }
    count -= removedCount;  // Track remaining events
    return removedCount;
}

// static helper method also used by OneTierBottom.  This is called
// few times at the end of simulation.  So it is not performance
// critical.
int
xxxx::TwoTierBucket::remove(BktEventList& list, xxxx::AgentID receiver) {
    int removedCount = 0;  // statistics tracking
    // Linear scan through events in a given sub-bucket
    BktEventList::iterator curr = list.begin();
    while (curr != list.end()) {
        if ((*curr)->getReceiverAgentID() == receiver) {
            (*curr)->decreaseReference();
            curr = list.erase(curr);
            removedCount++;
        } else {
            curr++;
        }
    }
    return removedCount;  // let caller know the events removed
}

// This method is not performance critical.  It is used only for
// troubleshooting/debugging
bool
xxxx::TwoTierBucket::haveBefore(const Time recvTime) const {
    for (const BktEventList& list : subBuckets) {
        for (const xxxx::Event* const event : list) {
            if (event->getReceiveTime() <= recvTime) {
                return true;
            }
        }
    }
    return false;
}

// Actually counts events in each bucket for validation purposes.
// This method is not performance critical.  It is used only for
// troubleshooting/debugging
size_t
xxxx::TwoTierBucket::getEventCount() const {
    int sum = 0;
    for (const BktEventList& subBkt : subBuckets) {
        sum += subBkt.size();
    }
```

```
151     return sum;  // total number of events
152 }
153
154 // -------------------------[ TwoTierTop methods ]-------------------------
155
156 // Helper method called from constructor and when events are moved
157 // from top into ladder.
158 void
159 xxxx::TwoTierTop::reset(const Time startTime) {
160     minTS    = TIME_INFINITY;
161     maxTS    = 0;
162     topStart = startTime;
163     clear();
164 }
165
166 void
167 xxxx::TwoTierTop::add(xxxx::Event* event) {
168     push_back<SenderID>(event);   // Call base-class method.
169     // Update running timestamps.
170     minTS = std::min(minTS, event->getReceiveTime());
171     maxTS = std::max(maxTS, event->getReceiveTime());
172 }
173
174 // -------------------------[ OneTierBottom methods ]-------------------------
175
176 void
177 xxxx::OneTierBottom::enqueue(xxxx::TwoTierBucket&& bucket) {
178     // Move events from bucket into the bottom.
179     TwoTierBucket::push_back(*this, std::move(bucket));
180     // Now sort the whole bottom O(n*log(n)) operation
181     std::sort(begin(), end(), OneTierBottom::compare);
182     DEBUG(validate());
183 }
184
185 void
186 xxxx::OneTierBottom::enqueue(xxxx::Event* event) {
187     BktEventList::iterator iter =
188         std::lower_bound(begin(), end(), event, compare);
189     insert(iter, event);  // base class method.
190     DEBUG(validate());
191 }
192
193 void
194 xxxx::OneTierBottom::dequeueNextAgentEvents(xxxx::EventContainer& events) {
195     if (empty()) {
196         return;  // no events to provide
197     }
198     // Reference information used for checking in the loop below.
199     const xxxx::Event*  nextEvt  = back();
200     const xxxx::AgentID receiver = nextEvt->getReceiverAgentID();
201     const xxxx::Time    currTime = nextEvt->getReceiveTime();
202     // Move all events from bottom to the events-container for scheduling.
203     do {
204         // Back event is the lowest timestamp (or highest priority)
205         // based on sorting order in OneTierBottom::compare()
206         xxxx::Event* event = back();
207         events.push_back(event);
208         pop_back();   // remove from bottom.
209         // erase(begin());
210         // Check and work with the next event.
211         nextEvt = (!empty() ? back() : NULL);
212         DEBUG(std::cout << "Delivering: " << *event << std::endl);
213     } while (!empty() && (nextEvt->getReceiverAgentID() == receiver) &&
214              TIME_EQUALS(nextEvt->getReceiveTime(), currTime));
215     DEBUG(validate());
216 }
217
218 double
219 xxxx::OneTierBottom::getBucketWidth() const {
220     if (empty()) {
221         return 0;
222     }
223     ASSERT(front() != NULL);
224     ASSERT(back()  != NULL);
225     // Assumes that bottom is sorted with the lowest timestamp at the
```



```
226     // end for fast pop_back
227     const double maxTS = front()->getReceiveTime();
228     const double minTS = back()->getReceiveTime();
229     return (maxTS - minTS + size() - 1.0) / size();
230 }
231
232 int
233 xxxx::OneTierBottom::remove_after(xxxx::AgentID sender, const Time sendTime) {
234     // Since bucket is sorted we can shortcircuit scan if last event's
235     // time is less-or-equal to sendTime.
236     if (empty() || (sendTime >= front()->getReceiveTime())) {
237         return -1;  // Since bucket does not have events to be cancelled.
238     }
239     size_t removedCount = 0;
240     iterator curr = begin();
241     while (curr != end()) {
242         xxxx::Event* const event = *curr;
243         if ((event->getSenderAgentID() == sender) &&
244             (event->getSentTime() >= sendTime)) {
245             // Free-up event.
246             event->decreaseReference();
247             removedCount++;
248             // In sorted mode we have to preserve the order. So
249             // cannot swap & pop in this situation
250             curr = erase(curr);
251         } else {
252             curr++;  // onto next event
253         }
254     }
255     return removedCount;
256 }
257
258 // This method is used only for debugging. So it is not performance
259 // critical.
260 void
261 xxxx::OneTierBottom::validate() const {
262     if (empty()) {
263         return;  // yes. bottom is valid.
264     }
265     // Ensure events are sorted in timestamp order.
266     BktEventList::const_iterator next = cbegin();
267     BktEventList::const_iterator prev = next++;
268     while ((next != cend()) &&
269            ((*next)->getReceiveTime() >= (*prev)->getReceiveTime())) {
270         prev = next++;
271     }
272     if (next != cend()) {
273         std::cout << "Error in LadderQueue.Bottom: Event " << **next
274                   << " was found after " << **prev << std::endl;
275     }
276     ASSERT( next == cend() );
277 }
278
279 // -------------------------[ TwoTierRung methods ]-------------------------
280
281 void
282 xxxx::TwoTierRung::move(TwoTierBucket&& bkt, const Time minTS,
283                         const double bktWidth) {
284     // Setup starting & current timestamp for this rung.
285     rStartTS = rCurrTS = minTS;
286     // Ensure bucket width is not ridiculously small
287     bucketWidth = bktWidth;
288     currBucket  = 0;  // current bucket in this rung.
289     // Initialize variable to track maximum bucket count
290     LQ2T_STATS(maxBkts = 0);
291     DEBUG(std::cout << "bucketWidth = " << bucketWidth << std::endl);
292     ASSERT(bucketWidth > 0);
293     ASSERT(rungEventCount == 0);
294     // Move events from given bucket into buckets in this Rung.
295     DEBUG(std::cout << "Adding " << bkt.size() << " events to rung\n");
296     for (BktEventList& list : bkt.getSubBuckets()) {
297         // Add all events from sub-buckets to various buckets in this rung.
298         while (!list.empty()) {
299             // Remove event from the top linked list.
300             xxxx::Event* event = list.back();
```

```
            list.pop_back();
            // Add to the appropriate bucket in this rung using a
            // helper method in this class.
            enqueue(event);
        }
    }
    // Reset bucket counters as we have moved all the events out
    bkt.resetCount();
    DEBUG(validateEventCounts());
}

void
xxxx::TwoTierRung::move(OneTierBottom&& bottom, const Time rStart,
                        const double bktWidth) {
    rStartTS = rCurrTS = rStart;
    // Ensure bucket width is not ridiculously small
    bucketWidth = bktWidth;
    currBucket  = 0;  // current bucket in this rung.
    ASSERT(rungEventCount == 0);
    // Initialize variable to track maximum bucket count
    LQ2T_STATS(maxBkts = 0);
    DEBUG(std::cout << "bucketWidth = " << bucketWidth << std::endl);
    ASSERT(bucketWidth > 0);
    ASSERT(rungEventCount == 0);
    // Move events from bottom into buckets in this Rung.
    DEBUG(std::cout << "Adding " << bottom.size() << " events to rung\n");
    for (xxxx::Event* event : bottom) {
        // Add to the appropriate bucket in this rung using a
        // helper method in this class.
        enqueue(event);
    }
    // Finally clear out the events in bottom.
    bottom.clear();
    DEBUG(validateEventCounts());
}


bool
xxxx::TwoTierRung::canContain(xxxx::Event* event) const {
    const xxxx::Time recvTime = event->getReceiveTime();
    const int bucketNum = (recvTime - rStartTS) / bucketWidth;
    return ((bucketNum >= (int) currBucket) && (recvTime >= rStartTS));
}

void
xxxx::TwoTierRung::enqueue(xxxx::Event* event) {
    ASSERT(event != NULL);
    ASSERT(event->getReceiveTime() >= getCurrTime());
    // Compute bucket for this event based on equation #2 in LQ paper.
    size_t bucketNum = (event->getReceiveTime() - rStartTS) / bucketWidth;
    ASSERT(bucketNum >= currBucket);
    if (bucketNum >= bucketList.size()) {
        // Ensure bucket list of sufficient size
        bucketList.resize(bucketNum + 1);
        // update variable to track maximum bucket count
        LQ2T_STATS(maxBkts = std::max(maxBkts, bucketList.size()));
    }
    ASSERT(bucketNum < bucketList.size());
    // Add event into appropriate bucket
    bucketList[bucketNum].push_back<SenderID>(event);
    // Track number of events added to this Rung
    rungEventCount++;
}

xxxx::TwoTierBucket&&
xxxx::TwoTierRung::removeNextBucket(xxxx::Time& bktTime) {
    ASSERT(!empty());
    ASSERT(currBucket < bucketList.size());
    // Find next non-empty bucket in this rung (there has to be one as
    // the previous asserts passed necessary checks)
    while ((currBucket < bucketList.size()) && bucketList[currBucket].empty()) {
        currBucket++;
    }
    DEBUG(validateEventCounts());
    ASSERT(currBucket < bucketList.size());
```



```
    ASSERT(!bucketList[currBucket].empty());
    // Track events that will be removed when this method returns
    rungEventCount -= bucketList[currBucket].size();
    ASSERT(rungEventCount >= 0);
    // Save information about the bucket to be removed & returned.
    const int retBkt = currBucket;
    bktTime = rStartTS + (retBkt * bucketWidth);
    // Advance current bucket to next time.
    currBucket++;
    rCurrTS = rStartTS + (currBucket * bucketWidth);
    // Sanity check on counters...
    if (currBucket >= bucketList.size()) {
        ASSERT(rungEventCount == 0);
    }
    return std::move(bucketList[retBkt]);
}

int
xxxx::TwoTierRung::remove_after(xxxx::AgentID sender, const Time sendTime
                                LQ2T_STATS(COMMA Avg& ceScanRung)) {
    if (empty() || (sendTime > getMaxRungTime())) {
        return 0;  // no events removed.
    }
    // Check each bucket in this rung and cancel out events.
    int numRemoved = 0;
    for (size_t bktNum = currBucket; (bktNum < bucketList.size()); bktNum++) {
        if (!bucketList[bktNum].empty() &&
            (rStartTS + (bktNum + 1) * bucketWidth) >= sendTime) {
            // Have the bucket remove necessary event(s) and update stats
            numRemoved +=
                bucketList[bktNum].remove_after(sender, sendTime
                                                LQ2T_STATS(COMMA ceScanRung));
        }
    }
    // Update events left in this rung.
    rungEventCount -= numRemoved;
    DEBUG(validateEventCounts());
    return numRemoved;
}

int
xxxx::TwoTierRung::remove(xxxx::AgentID receiver
                          LQ2T_STATS(COMMA Avg& ceScanRung)) {
    if (empty()) {
        return 0;  // no events to be removed.
    }
    // Have each bucket in the rung remove events
    int numRemoved = 0;
    for (size_t bktNum = currBucket; (bktNum < bucketList.size()); bktNum++) {
        if (!bucketList[bktNum].empty()) {
            // This stat needs to be tracked by the bucket and not here.
            LQ2T_STATS(ceScanRung += bucketList[bktNum].size());
            // Remove appropriate set of events.
            numRemoved += bucketList[bktNum].remove(receiver);
        }
    }
    rungEventCount -= numRemoved;
    DEBUG(validateEventCounts());
    return numRemoved;
}

void
xxxx::TwoTierRung::validateEventCounts() const {
    int numEvents = 0;
    for (const auto& bucket : bucketList)  {
        numEvents += bucket.size();
    }
    if (numEvents != rungEventCount) {
        DEBUG(std::cout << "Rung event count mismatch! Expecting: "
                        << rungEventCount << " events, but found: "
                        << numEvents << "." << std::endl);
        ASSERT(numEvents == rungEventCount);
    }
}
```

```
451 // Method called just before a rung is removed from the ladder queue.
452 void
453 xxxx::TwoTierRung::updateStats(Avg& avgBktCnt) const {
454     LQ2T_STATS(avgBktCnt += maxBkts);
455 }
456
457 // This method is used only for troubleshooting/debugging purposes.
458 bool
459 xxxx::TwoTierRung::haveBefore(const Time recvTime) const {
460     for (size_t i = 0; (i < bucketList.size()); i++) {
461         if (bucketList[i].haveBefore(recvTime)) {
462             return true;
463         }
464     }
465     return false;
466 }
467
468 void
469 xxxx::TwoTierRung::prettyPrint(std::ostream& os) const {
470     // Compute minimum, maximum, empty, and average bucket sizes.
471     size_t minBkt = -1U, maxBkt = 0, emptyBkt = 0, sizeSum = 0;
472     for (const TwoTierBucket& bkt : bucketList) {
473         if (!bkt.empty()) {
474             minBkt   = std::min(minBkt, bkt.size());
475             maxBkt   = std::max(maxBkt, bkt.size());
476             sizeSum += bkt.size();
477         } else {
478             emptyBkt++;
479         }
480     }
481     double avgBktSz = sizeSum / (double) (bucketList.size() - emptyBkt);
482     os << "start time="    << rStartTS     << ", curr time=" << rCurrTS
483        << ", bkt. width=" << bucketWidth << ", bkt count=" << bucketList.size()
484        << ", curr buckt=" << currBucket  << ", events="     << rungEventCount
485        << ", min bkt="    << minBkt      << ", maxBkt="     << maxBkt
486        << ", empty bkt="  << emptyBkt    << ", avg size="   << avgBktSz
487        << std::endl;
488 }
489
490 // --------------------[ TwoTierLadderQueue methods ]--------------------
491
492 // The maximum number of rungs typically allowed in the ladder.  This
493 // value is set to 8 by default based on Tang et. al.  It can be set
494 // via command-line parameter --lq-max-rungs 8.
495 size_t xxxx::TwoTierLadderQueue::MaxRungs = 8;
496
497 void
498 xxxx::TwoTierLadderQueue::reportStats(std::ostream& os) {
499     UNUSED_PARAM(os);
500     LQ2T_STATS({
501             // Collect final bucket counts from the ladder
502             for (size_t i = 0; (i < nRung); i++) {
503                 ladder[i].updateStats(avgBktCnt);
504             }
505             // Compute net number of compares for ladderQ
506             // const long comps = log2(botLen.getMean()) * botLen.getSum();
507             // std::make_heap has 3N time complexity.
508             const long comps = 3 * botLen.getSum() +
509                 log2(botLen.getMean()) * botLen.getSum() / 3;
510             os << "Events cancelled from top  : "    << ceTop
511                << "\nEvents scanned in top      : " << ceScanTop
512                << "\nEvents cancelled from ladder: " << ceLadder
513                << "\nEvents scanned from ladder : " << ceScanLadder
514                << "\nEvents cancelled from bottom: " << ceBot
515                << "\nEvents scanned from bottom : " << ceScanBot
516                << "\nNo cancel scans of bottom  : " << ceNoCanScanBot
517                << "\nInserts into top           : " << insTop
518                << "\nInserts into rungs         : " << insLadder
519                << "\nInserts into bottom        : " << insBot
520                << "\nMax rung count             : " << maxRungs
521                << "\nAverage #buckets per rung  : " << avgBktCnt
522                << "\nAverage bottom size        : " << botLen
523                << "\nMax bottom size            : " << maxBotSize
524                << "\nAverage bucket width       : " << avgBktWidth
525                << "\nBottom to rung operations  : " << botToRung
```



```
526                << "\nCompare estimate           : " << comps
527                << std::endl;
528         });
529 }
530
531 void
532 xxxx::TwoTierLadderQueue::enqueue(xxxx::Event* event) {
533     if (top.getStartTime() < event->getReceiveTime()) {
534         DEBUG(std::cout << "Added to top: " << *event << std::endl);
535         top.add(event);
536         LQ2T_STATS(insTop++);
537         return;
538     }
539     // Try to see if the event fits in the ladder. nRung is max rung index
540     size_t rung = 0;
541     while ((rung < nRung) && !ladder[rung].canContain(event)) {
542         ASSERT((rung == 0) || ladder[rung].empty() ||
543                (ladder[rung - 1].getCurrTime() >= ladder[rung].getCurrTime()));
544         rung++;
545     }
546     if (rung < nRung) {
547         DEBUG(ASSERT(bottom.empty() ||
548                      (event->getReceiveTime() > bottom.maxTime())));
549         ladder[rung].enqueue(event);
550         ladderEventCount++;  // Track events added to the ladder
551         DEBUG(std::cout << "Added to rung " << rung  << "(max bottom: "
552                         << bottom.maxTime() << "): " << *event << "\n");
553         LQ2T_STATS(insLadder++);
554         return;
555     }
556     // Event does not fit in the ladder. It must go into bottom.
557     // However, to ensure good performance we must keep bottom short.
558     if ((bottom.size() > LQ2T_THRESH) && (bottom.getTimeRange() > 0)) {
559         // Move events from bottom into ladder rung
560         rung = createRungFromBottom();
561         ASSERT(rung == nRung - 1);
562         ASSERT(rung < ladder.size());
563         // Due to rollback-reprocessing the event may be even
564         // earlier than the last rung we just created!
565         if (ladder[rung].canContain(event)) {
566             ladder[rung].enqueue(event);
567             ladderEventCount++;
568             DEBUG(std::cout << "Added to rung " << rung  << "(max bottom: "
569                             << bottom.maxTime() << "): " << *event << "\n");
570             LQ2T_STATS(insLadder++);
571             return;
572         }
573     }
574     // At this point, event must go into bottom, so enqueue it.
575     bottom.enqueue(event);
576     LQ2T_STATS(maxBotSize = std::max(maxBotSize, bottom.size()));
577     DEBUG(ASSERT(!haveBefore(bottom.first_event()->getReceiveTime())));
578     DEBUG(std::cout << "Added to bottom: " << *event << std::endl);
579     LQ2T_STATS(insBot++);
580 }
581
582 // Implementation close to the version from the paper.
583 xxxx::TwoTierBucket&&
584 xxxx::TwoTierLadderQueue::recurseRung() {
585     ASSERT(!empty());
586     ASSERT(nRung > 0);
587     ASSERT(!ladder.empty());
588     // Now the last rung in ladder is the rung that has the next
589     // bucket of events.
590     xxxx::Time bktTime    = 0;  // set by removeNextBucket call below
591     TwoTierRung& lastRung = ladder[nRung - 1];
592     TwoTierBucket&& bkt   = lastRung.removeNextBucket(bktTime);
593     ASSERT(!bkt.empty());
594     ASSERT(!ladder.empty());
595     // Check and create new rung in the ladder if the bucket is large.
596     if ((bkt.size() > LQ2T_THRESH) && (nRung < MaxRungs)) {
597         // Note: Here bucket width can dip a bit low. But that is
598         // needed to ensure consistent ladder setup.
599         const double bucketWidth = (lastRung.getBucketWidth() + bkt.size() -
600                                     1.0) / bkt.size();
```

```
601             // Create a new rung in the ladder
602             nRung++;
603             if (nRung > ladder.size()) {
604                 ladder.push_back(TwoTierRung(std::move(bkt), bktTime, bucketWidth));
605                 ASSERT(nRung == ladder.size());
606             } else {
607                 ladder[nRung - 1].move(std::move(bkt), bktTime, bucketWidth);
608             }
609             DEBUG(std::cout << "2. Bucket width: " << bucketWidth << std::endl);
610             LQ2T_STATS(avgBktWidth += bucketWidth);
611             LQ2T_STATS(maxRungs = std::max(maxRungs, nRung));
612             return recurseRung();  // Recurse now looking at newly added rung
613         }
614         // Track events being removed from the ladder
615         ladderEventCount -= bkt.size();
616         ASSERT(ladderEventCount >= 0);
617         // Return bucket being removed.
618         return std::move(bkt);
619     }
620
621     // Move events from ladder (or top) into bottom.
622     void
623     xxxx::TwoTierLadderQueue::populateBottom() {
624         if (!bottom.empty()) {
625             return;
626         }
627         if (ladderEventCount == 0) {   // nRung == -1
628             if (top.empty()) {
629                 // There are no events in the ladder queue in this case
630                 ASSERT(empty());
631                 return;
632             }
633             // Move all events from top into buckets in first rung of the ladder!
634             nRung++;
635             ASSERT(nRung == 1);
636             ladderEventCount += top.size();     // Track events in ladder
637             // Move events to ladder
638             if (nRung > ladder.size()) {
639                 ladder.push_back(TwoTierRung(std::move(top)));
640                 ASSERT(nRung == ladder.size());
641             } else {
642                 ladder[nRung - 1].move(std::move(top), top.getMinTime(),
643                                        top.getBucketWidth());
644             }
645             // Reset top counters and update the values of topStart for
646             // next Epoch
647             top.reset(top.getMaxTime());
648             LQ2T_STATS(maxRungs = std::max(maxRungs, nRung));
649             LQ2T_STATS(avgBktWidth += ladder.back().getBucketWidth());
650             DEBUG(std::cout << "3. Bucket width: "
651                             << ladder.back().getBucketWidth() << std::endl);
652             DEBUG(prettyPrint(std::cout));
653             ASSERT(top.empty());
654         }
655         // Bottom is empty. So we need to move events from the current
656         // bucket in the ladder to bottom.
657         ASSERT(!ladder.empty());
658         ASSERT(bottom.empty());
659         bottom.enqueue(recurseRung());  // Transfer bucket_k into bottom
660         ASSERT(!bottom.empty());
661         LQ2T_STATS(maxBotSize = std::max(maxBotSize, bottom.size()));
662         DEBUG(ASSERT(!haveBefore(bottom.first_event()->getReceiveTime())));
663         LQ2T_STATS(botLen += bottom.size());
664         // Clear out the rungs if we have used-up the last bucket in the ladder.
665         while (nRung > 0 && ladder[nRung - 1].empty()) {
666             LQ2T_STATS(ladder[nRung - 1].updateStats(avgBktCnt));
667             nRung--;  // Logically remove rung from ladder
668         }
669     }
670
671     int
672     xxxx::TwoTierLadderQueue::createRungFromBottom() {
673         ASSERT(!bottom.empty());
674         ASSERT(bottom.getTimeRange() > 0);
675         DEBUG(std::cout << "Moving events from bottom to a new rung. Bottom has "
```



```
676                         << bottom.size() << " events." << std::endl);
677         // Compute the start time and bucket width for the rung.  Note
678         // that with rollbacks, ladder can be empty and that situation
679         // needs to be handled.
680         const double bucketWidth = (ladder.empty() ? bottom.getBucketWidth() :
681                                     ladder[nRung - 1].getBucketWidth());
682         // The paper computes rStart as RCur[NRung-1].  However, due to
683         // rollback-reprocessing the bottom may have events that are below
684         // RCur[NRung-1].  Consequently, we use the minimum of the two
685         // values as as rstart
686         const Time ladBkTime = ((nRung > 0) ? ladder[nRung - 1].getCurrTime() :
687                                 TIME_INFINITY);
688         const Time rStart = std::min(ladBkTime,
689                                      bottom.first_event()->getReceiveTime());
690         ASSERT(rStart < ladBkTime);
691         ASSERT(bottom.maxTime() < ladBkTime);
692         ASSERT(bottom.maxTime() <= top.getStartTime());
693         // Create a new rung and add it to the ladder.
694         DEBUG(std::cout << "Moving bottom to rung. Events: " << bottom.size()
695                         << ", rStart = " << rStart << ", bucketWidth = "
696                         << (bucketWidth / bottom.size()) << std::endl);
697         ladderEventCount += bottom.size();  // Update ladder event count
698         LQ2T_STATS(botToRung += bottom.size());
699         const double bktWidth = (bucketWidth + bottom.size() - 1.0) / bottom.size();
700         DEBUG(std::cout << "bktWidth = " << bktWidth << std::endl);
701         // Add rung and move move bottom into the last rung of the ladder.
702         nRung++;
703         if (nRung > ladder.size()) {
704             ladder.push_back(TwoTierRung(std::move(bottom), rStart, bktWidth));
705             ASSERT(nRung == ladder.size());
706         } else {
707             ladder[nRung - 1].move(std::move(bottom), rStart, bktWidth);
708         }
709         DEBUG(std::cout << "1. Bucket width: " << bktWidth << std::endl);
710         LQ2T_STATS(avgBktWidth += bktWidth);
711         LQ2T_STATS(maxRungs = std::max(maxRungs, nRung)); // Track max rungs
712         ASSERT(bottom.empty());
713         return nRung - 1;
714     }
715
716     int
717     xxxx::TwoTierLadderQueue::remove_after(xxxx::AgentID sender,
718                                            const Time sendTime) {
719         // Check and cancel entries in top rung.
720         int numRemoved = top.remove_after(sender, sendTime
721                                           LQ2T_STATS(COMMA ceScanTop));
722         LQ2T_STATS(ceTop += numRemoved);
723         // Cancel out events in each rung of the ladder.
724         for (size_t rung = 0; (rung < nRung); rung++) {
725             const int rungEvtRemoved =
726                 ladder[rung].remove_after(sender, sendTime
727                                           LQ2T_STATS(COMMA ceScanLadder));
728             ladderEventCount  -= rungEvtRemoved;
729             numRemoved        += rungEvtRemoved;
730             LQ2T_STATS(ceLadder += rungEvtRemoved);
731         }
732         // Clear out the rungs in ladder that are now empty after event
733         // cancellations.
734         while (nRung > 0 && ladder[nRung - 1].empty()) {
735             LQ2T_STATS(ladder[nRung - 1].updateStats(avgBktCnt));
736             nRung--;  // Logically remove rung from ladder
737         }
738         // Save original size of bottom to track stats.
739         LQ2T_STATS(const size_t botSize  = bottom.size());
740         // Cancel events from bottom.
741         const int botRemoved  = bottom.remove_after(sender, sendTime);
742         if (botRemoved > -1) {
743             numRemoved += botRemoved;
744             // Update statistics counters
745             LQ2T_STATS(ceBot += botRemoved);
746             LQ2T_STATS(ceScanBot += botSize);
747             LQ2T_STATS((botRemoved == 0) ? (ceNoCanScanBot += botSize) : 0);
748         }
749         return numRemoved;
750     }
```

```
751
752 // This method is purely for debugging. So performance is not
753 // important
754 bool
755 xxxx::TwoTierLadderQueue::haveBefore(const Time recvTime,
756                                      const bool checkBottom) const {
757     // Check top
758     if (top.haveBefore(recvTime)) {
759         std::cout << "Top has event that is <= " << recvTime << std::endl;
760         prettyPrint(std::cout);
761         return true;
762     }
763     // Check each rung of the ladder
764     for (size_t rung = 0; (rung < nRung); rung++) {
765         if (ladder[rung].haveBefore(recvTime)) {
766             std::cout << "Rung #" << rung << " has event that is <= "
767                       << recvTime << std::endl;
768             prettyPrint(std::cout);
769             return true;
770         }
771     }
772     // Check bottom rung.
773     if (checkBottom && bottom.haveBefore(recvTime)) {
774         std::cout << "Bottom has event that is <= " << recvTime << std::endl;
775         prettyPrint(std::cout);
776         return true;
777     }
778     // When control drops here it mean the whole 2-tier ladder queue
779     // does not have an event with timestamp lower than recvTime.
780     return false;
781 }
782
783 // ------------------------[  EventQueue implementation ]----------------------
784
785 void*
786 xxxx::TwoTierLadderQueue::addAgent(xxxx::Agent* agent) {
787     UNUSED_PARAM(agent);
788     return NULL;  // 2-tier queue has no cross-references to store in agent
789 }
790
791 void
792 xxxx::TwoTierLadderQueue::removeAgent(xxxx::Agent* agent) {
793     ASSERT( agent != NULL );
794     const AgentID receiver = agent->getAgentID();
795     // Remove events for agent from top
796     LQ2T_STATS(ceScanTop += top.size());
797     int numRemoved    = top.remove(receiver);
798     LQ2T_STATS(ceTop += numRemoved);
799
800     // Next remove events for agent from all the rungs in the ladder
801     for (TwoTierRung& rung : ladder) {
802         int rungEvtRemoved = rung.remove(agent->getAgentID()
803                                          LQ2T_STATS(COMMA ceBot));
804         ladderEventCount  -= rungEvtRemoved;
805         numRemoved        += rungEvtRemoved;
806         LQ2T_STATS(ceLadder += rungEvtRemoved);
807     }
808     // Finally remove events from bottom for the agent.
809     LQ2T_STATS(const size_t botSize = bottom.size());
810     const int botRemoved  = bottom.remove(receiver);
811     LQ2T_STATS(ceScanBot += botSize);
812     LQ2T_STATS((botRemoved == 0) ? (ceNoCanScanBot += botSize) : 0);
813     numRemoved           += botRemoved;
814     LQ2T_STATS(ceBot     += botRemoved);
815 }
816
817
818 xxxx::Event*
819 xxxx::TwoTierLadderQueue::front() {
820     if (empty()) {
821         // Nothing to return.
822         return NULL;
823     }
824     if (bottom.empty()) {
825         populateBottom();
```



```
826         DEBUG(prettyPrint(std::cout));
827     }
828     ASSERT(!bottom.empty());
829     return bottom.first_event();
830 }
831
832 void
833 xxxx::TwoTierLadderQueue::dequeueNextAgentEvents(xxxx::EventContainer& events) {
834     if (empty()) {
835         // No events to dequeue.
836         return;
837     }
838     // We only dequeue from bottom. So ensure it has events in it.
839     if (bottom.empty()) {
840         // Move events from top or a ladder rung into bottom.
841         populateBottom();
842     }
843     ASSERT(!bottom.empty());
844     bottom.dequeueNextAgentEvents(events);
845     ASSERT(!events.empty());
846     DEBUG(ASSERT(!haveBefore(events.front()->getReceiveTime())));
847 }
848
849 // The main interface method used by XXXX to schedule event.
850 void
851 xxxx::TwoTierLadderQueue::enqueue(xxxx::Agent* agent, xxxx::Event* event) {
852     UNUSED_PARAM(agent);
853     event->increaseReference();
854     enqueue(event);
855 }
856
857 // Method for block addition (typically used during rollback recovery)
858 void
859 xxxx::TwoTierLadderQueue::enqueue(xxxx::Agent* agent,
860                                   xxxx::EventContainer& events) {
861     UNUSED_PARAM(agent);
862     for (auto& curr : events) {
863         enqueue(curr);
864     }
865     events.clear();
866 }
867
868 // Method to cancel all events in the 2-tier heap.
869 int
870 xxxx::TwoTierLadderQueue::eraseAfter(xxxx::Agent* dest,
871                                      const xxxx::AgentID sender,
872                                      const xxxx::Time sentTime) {
873     UNUSED_PARAM(dest);
874     return remove_after(sender, sentTime);
875 }
876
877 void
878 xxxx::TwoTierLadderQueue::prettyPrint(std::ostream& os) const {
879     // Print information on top.
880     os << "Top: Events=" << top.size()
881        << ", startTime=" << top.getStartTime()
882        << ", minTime="   << top.getMinTime()
883        << ", maxTime="   << top.getMaxTime()  << std::endl;
884     // Print info on each rung of the ladder
885     std::cout << "Ladder (rungs=" << nRung << ", size="
886               << ladder.size() << "):\n";
887     for (size_t i = 0; (i < nRung); i++) {
888         os << "[" << i << "]: ";
889         ladder[i].prettyPrint(os);
890     }
891     // Print info on bottom
892     os << "Bottom: Events=" << bottom.size()
893        << ", min="          << (!bottom.empty() ? bottom.findMinTime() : -1.0)
894        << ", max="          << (!bottom.empty() ? bottom.maxTime()     : -1.0)
895        << std::endl;
896 }
897
898 #endif
```

```
#ifndef PHOLD_SIMULATION_H
#define PHOLD_SIMULATION_H


/*
   This is a Synthetic Simulation done for benchmarking. P-HOLD simulation is
   meant to mimic a typical load for a given simulation and can be scaled.

   rows = Number of agents per row
   cols = Number of agents per column
   eventsPerAgent = Number of initial events each agent starts with
   delay = The Delay time for the receive time, this will be max range for a ran
dom from [0,1]
   computeNodes = The max number of nodes to run P-HOLD.
   simEndTime = The end time for the simulation.

   @note Please see PHOLDAgent.cpp for more detail :-)


*/
class PHOLDSimulation {
public:
    /** Destructor.

        Currently the destructor for this class does not have much
        operation to perform and is merely peresent to adhere to
        conventions and serve as a place holder for future changes.
    */
    ~PHOLDSimulation();

    /** The main interface method for this class that should be used
        to start and run the PHOLD Simulation.  The command-line
        arguments to the program should be passed-in as the arguments
        to this method.  These values are typically the default values
        passed to a standard C/C++ main() function.

        \param[in] argc The number of command-line arguments.

        \param[in] argv The actual command-line arguments.
    */
    static void run(int argc, char** argv);

protected:
    /** The default constructor.

        The default constructor is intentionally protected as this
        class is not meant to be directly instantiated.  Instead the
        PHOLDSimulation::run() static method should be used to run
        the simulation.  The constructor initializes the simulation
        configuration variables to default initial values.
    */
    PHOLDSimulation();

    /** Helper method to process command-line arguments (if any).

        This method uses the ArgParser utility class to parse any
        command-line arguments supplied to the simulation and updates
        the configuration (several private instance variables)
        variables in this class.  The configuration variables are used
        by various method in the class to create various agents and
        initialize the simulation to match the supplied configuration.

        \param[in] argc The number of command-line arguments.

        \param[in] argv The actual command-line arguments.

        \return This method returns true if the command-line arguments
        were successfully processed.
    */
    bool processArgs(int argc, char** argv);

    /** This method is used to create the phold agents in the
        simulation and register with the simulation kernel.  The
        number of phold agents created by this method is max_agents /
        max_nodes (because the phold agents are evenly distributed
```



```
        across the various compute nodes used for simulation).  The
        method also handles the edge case where the last compute node
        gets any additional agents (if max_agents is not evenly
        divisible by max_nodes).

        \note This method must be called only after the simulation
        kernel has already been initialized.
    */
    void createAgents();

    /**
       Convenience method to setup time duration for simulation and
       initiate the actual simulation.  This is a convenience method
       that is primarily used to streamline the top-level run() method
       in this class.
    */
    void simulate();

    /** Helper method to compute cumulative sum upto a given value.

        Convenience method to compute cumulative sum of series 1 + 2 +
        ... + val

        \param[in] val The value for which cumulative sum of series is
        to be returned.

        \param[in] scale The scale for multiplying the cumulative sum.

        \return This method returns scale * val * (val + 1) / 2.
    */
    int cumlSum(const int val, const double scale = 1) const;

private:
    /** Percentage of imbalance required in initial partitioning.

        This value is a percentage in the range 0 to 0.99 (99%) that
        is specifed as a command-line argument \c --imbalance.  This
        value is used to skew the even initial partitioning by the
        specified percentage. For example, given 1000 agents, 0.25
        imbalance, on 4 processes, the partition of agents will be
        {188, 229, 271, 312} (i.e., 250 agents were distributed to the
        4 processes weighted by the rank of the processes).
    */
    double imbalance;

    /** Fraction of events that agents should schedule to themselves.

        This value indicates the probability that agents should
        schedule events to themselves.  This value is in the range 0.0
        to 0.99 and is used with the boolean expression ((rand() %
        1000) / 1000.0) < selfEvents.  If the boolean expression
        returns true, then agents schedule events to themselves.  Note
        that at a given virtual time, the maximum number of
        self-events is limited by events * selfEvents.
    */
    double selfEvents;

    /** A user-specified granularity -- that is, the number of dummy
        loops to run by each agent for each event to add some
        CPU-load/granularity for each event.

        This instance variable is set to indicate the number of loops
        or iterations that must be executed by the agent to simulate
        some granularity, i.e., CPU used per event.  This value is set
        to zero by default and can be modified via the command-line
        argument \c --granularity.
    */
    size_t granularity;

    /** The number of columns in the grid of agents in PHold. This
        value is specified as a command-line argument \c --cols.
    */
    int cols;

    /** The number of rows in the grid of agents in PHold. This
```

```
150         value is specified as a command-line argument \c --rows.
151     */
152     int rows;
153
154     /** The initial number of events generated by each agent in
155         PHold. This value is specified as a command-line argument \c
156         --eventsPerAgent.
157     */
158     int events;
159
160     /** Fixed lookahead virtual time delay for generating events.
161
162         The virtual time events for each agent are generated which
163         this fixed look ahead value.
164     */
165     int lookAhead;
166
167     /** The maximum uniform-random/mean delay to be added to the
168         lookahead when scheduling events.  This value is not zero,
169         then it used to compute a random delay factor as: lookAhead +
170         (rand() % delay).
171     */
172     int delay;
173
174     /** The type of delay distribution to be for generating delays.
175         Valid delay distributions are: uniform, poisson, exponential,
176         reverse_poisson, reverse_exponential.  The reverse forms of
177         the distribution are mirror images of the regular
178         distributions causing a heavy tailed distribution.
179     */
180     std::string delayDistrib;
181
182
183     /** The virtual time when the simulation is deemed to be complete.
184         This value is set via the command-line argument --endTime
185     */
186     int end_time;
187
188     /** The MPI rank of this simulation process.  */
189     int rank;
190
191     /** Flag to indicate if the delay histogram should be printed. */
192     bool delayHist;
193 };
194
195 #endif
```

```cpp
#ifndef PHOLD_SIMULATION_CPP
#define PHOLD_SIMULATION_CPP

#include "Simulation.h"
#include "PHOLDSimulation.h"
#include "PHOLDAgent.h"
#include "PholdState.h"
#include "ArgParser.h"

PHOLDSimulation::PHOLDSimulation() {
    rows         = 3;
    cols         = 3;
    events       = 3;
    delay        = 0;
    lookAhead    = 1;
    imbalance    = 0.0;
    selfEvents   = 0.0;
    end_time     = 100;
    granularity  = 0;
    delayDistrib = "exponential";
    delayHist    = false;
}

PHOLDSimulation::~PHOLDSimulation() {}

bool
PHOLDSimulation::processArgs(int argc, char** argv) {
    // Make the arg_record
    ArgParser::ArgRecord arg_list[] = {
        {"--cols", "The Number of columns in the space.", &cols,
         ArgParser::INTEGER },
        {"--rows", "The Number of rows in the space.",
         &rows, ArgParser::INTEGER },
        {"--lookahead", "Minimum delay time for events", &lookAhead,
         ArgParser::INTEGER},
        {"--delay", "The maximum random time added to look ahead [0, 100]",
         &delay, ArgParser::INTEGER },
        {"--delay-distrib", "The type of delay distribution to be used",
         &delayDistrib, ArgParser::STRING},
        {"--eventsPerAgent", "Initial number of events per agent",
         &events, ArgParser::INTEGER },
        {"--eventsPerLP", "Same as eventsPerAgent",
         &events, ArgParser::INTEGER },
        {"--selfEvents", "Fraction of events agents send to themselves [0, 1]",
         &selfEvents, ArgParser::DOUBLE},
        {"--simEndTime", "The end time for the simulation.", &end_time,
         ArgParser::INTEGER },
        {"--imbalance", "Desired imbalance in partitioning [0, 0.99]",
         &imbalance, ArgParser::DOUBLE},
        {"--granularity", "Granularity (no units) per events", &granularity,
         ArgParser::LONG},
        {"--print-hist", "Print delay histogram", &delayHist,
         ArgParser::BOOLEAN},
        {"", "", NULL, ArgParser::INVALID}
    };

    // Let the kernel initialize using any additional command-line
    // arguments.
    XXXX::Simulation* const kernel = XXXX::Simulation::getSimulator();
    try {
        kernel->initialize(argc, argv);
    } catch (std::exception& exp) {
        std::cerr << "Exiting simulation due to initialization error: "
                  << exp.what() << std::endl;
        return false;
    }

    // Use the XXXX argument parser to parse command-line arguments
    // and update instance variables
    ArgParser ap(arg_list);
    ap.parseArguments(argc, argv, true);

    rank = kernel->getSimulatorID();
    // Check to ensure we have at least as many agents as compute-nodes
    if (rows * cols < (int) kernel->getNumberOfProcesses()) {
```



```cpp
        std::cerr << "The number of agents (i.e., rows * cols) must be "
                  << "greater than number of MPI processes.\n";
        return false;
    }
    // Ensure the delay distribution specified is indeed valid.
    PHOLDAgent::DelayType delayType = PHOLDAgent::toDelayType(delayDistrib);
    if (delayType == PHOLDAgent::INVALID_DELAY) {
        std::cerr << "Invalid delay distribution specified. Valid values are: "
                  << "uniform, poisson, exponential, reverse_poission, "
                  << "reverse_exponential.\n";
        return false;
    }
    // Everything went well.
    return true;
}

int
PHOLDSimulation::cumlSum(const int val, const double scale) const {
    return (scale * val * (val - 1) / 2);
}

void
PHOLDSimulation::createAgents() {
    XXXX::Simulation* kernel = XXXX::Simulation::getSimulator();
    const int max_agents     = rows * cols;
    const int max_nodes      = kernel->getNumberOfProcesses();
    const int skewAgents     = (max_nodes > 1) ? (max_agents * imbalance) : 0;
    const double factor      = (skewAgents > 0) ?
        (skewAgents / (double) cumlSum(max_nodes)) : 0;
    const int agentsPerNode  = (max_agents - skewAgents) / max_nodes;
    ASSERT( agentsPerNode > 0 );
    const int agentStartID   = (agentsPerNode * rank) + cumlSum(rank, factor);
    const int agentEndID     = (rank == max_nodes - 1) ? max_agents :
        ((agentsPerNode * (rank + 1)) + cumlSum(rank + 1, factor));
    // Converte delay type string to suitable enumeration.
    const PHOLDAgent::DelayType delayType =
        PHOLDAgent::toDelayType(delayDistrib);
    // Create the agents.
    for (int i = agentStartID; (i < agentEndID); i++) {
        PholdState* state = new PholdState();
        PHOLDAgent* agent = new PHOLDAgent(i, state, rows, cols, events, delay,
                                           lookAhead, selfEvents, granularity,
                                           delayType);
        kernel->registerAgent(agent);
        // Have the first agent print the delay histogram
        if (delayHist && (i == agentStartID)) {
            agent->printDelayDistrib(std::cout);
        }
    }
    std::cout << "Rank " << rank << ": Registered agents from "
              << agentStartID    << " to "
              << agentEndID      << " agents.\n";
}

void
PHOLDSimulation::simulate() {
    // Convenient local reference to simulation kernel
    XXXX::Simulation* const kernel = XXXX::Simulation::getSimulator();
    // Setup start and end time of the simulation
    kernel->setStartTime(0);
    kernel->setStopTime(end_time);
    // Finally start the simulation here!!
    kernel->start();
    // Now we finalize the kernel to make sure it cleans up.
    kernel->finalize();
}

void
PHOLDSimulation::run(int argc, char** argv) {
    // validate input
    ASSERT(argc > 0);
    ASSERT(argv != NULL);

    // Create simulation, populate variables
    PHOLDSimulation sim;
```

```
151     if (sim.processArgs(argc, argv)) {
152         // Create Agents
153         sim.createAgents();
154         // Run simulation
155         sim.simulate();
156     }
157 }
158 
159 /*
160     The main method coordinates all the activtes of the
161     simulation. Note that the main method runs on all the parallel
162     processes used for simulation.
163 
164     \param[in] argc The number of command-line arguments.
165 
166     \param[in] argv The command-line arguments to be parsed
167 */
168 int
169 main(int argc, char** argv) {
170     PHOLDSimulation::run(argc, argv);
171     return 0;
172 }
173 
174 #endif
```

```
#ifndef PHOLD_AGENT_H
#define PHOLD_AGENT_H


/*
 * File:   PHOLDAgent.h
 *
 * Created on January 15, 2009, 11:04 PM
 */

#include <random>
#include "Agent.h"
#include "PholdState.h"

class PHOLDAgent : public XXXX::Agent {
public:
    /** Predefined enumerations for the different types of random
        number distributions for generating delays in receive time of
        events generated by this model.
    */
    enum DelayType { UNIFORM, POISSON, EXPONENTIAL, REVERSE_POISSON,
                     REVERSE_EXPONENTIAL, INVALID_DELAY };

    /** Convenience helper method to convert a given string to
        corresponding delay type enumeration value.

        \param[in] delay The delay string to be convereted to the
        corresponding delay type.  The string is converted to all
        lower-case for convenience/comparisons.

        \return Enumeration corresponding to the delay type.  If delay
        type is not know, this method returns
        PHOLDAgent::INVALID_DELAY as the type.
    */
    static DelayType toDelayType(const std::string& delay);

    PHOLDAgent(XXXX::AgentID , PholdState *,int x, int y, int n, int d,
               int lookAhead = 1, double selfEvents = 0.0,
               size_t granularity = 0,
               PHOLDAgent::DelayType delayType = UNIFORM);

    void initialize() throw (std::exception);

    void executeTask(const XXXX::EventContainer* events);

    void finalize();

    /** A helper method to print the distribution of event delays.

        This is a utility method that can be used to print the
        distribution profile of the event delays generated by the
        delay and distribution patterns set for this agent.  This
        method is just an adaptation of the example(s) for different
        random number generations from http://en.cppreference.com

        \param[out] os The output stream to where the distribution is
        to be printed.
    */
    void printDelayDistrib(std::ostream& os = std::cout);

protected:
    /** Simulate some granularity (i.e., CPU usage) for the event.

        This method merely runs a loop generating random numbers to
        simulate some processing done for each event.  Currently, the
        granularity is a fixed value, set when an agent is created.

    */
    void simGranularity();

    /** Obtain a random delay value based on the type of distribution
        set for this sagent.

        \param[in] delType The type of distribution to be used.

```



```
        \return A random delay based on the the delay value and type
        of distribution set for this agent.
    */
    int getDelay(DelayType delType);

    /** Compute the maximum possibly delay value.  This method is
        typically called only once in the constructor to determine the
        maximum delay value that can be generated to setup the
        maxDelay static instance variable in this class.

        \return The maximum delay value that is to be generated by the
        specified random number distribution.
    */
    int getMaxDelayValue() const;

private:
    int X,Y,N,Delay;

    /** The type of random number distribution to be used for
        generating delays for event receive times.  The delays are
        useful for modeling different types of behaviors in simulation
        models.  The default delay type is UNIFORM.  This value is set
        in the constructor and is never changed during the lifetime of
        this agent.
    */
    DelayType delayType;

    /** Fixed lookahead virtual time delay for generating events.

        The virtual time events generated by this agent have this
        fixed look ahead virtual time value added to them.
    */
    int lookAhead;

    /** Fraction of events that the agent should schedule to itself.

        This value indicates the probability that this agent should
        schedule events to itself.  This value is in the range 0.0 to
        0.99 and is used with the boolean expression ((rand() % 1000)
        / 1000.0) < selfEvents.  If the boolean expression returns
        true, then this agent schedule events to itself.  Note that at
        a given virtual time, the maximum number of self-events is
        limited by events * N.
    */
    double selfEvents;

    /** The random seed value that is used to determine number of
        self/other events.

        This seed is used to determine the probability with which
        agents should generate events to themselves versus other
        agents in the simulation.
    */
    unsigned int seed;

    /** The random number generator used to generate random delays for
        this agent.  The actual implementation depends on the
        implementation.
    */
    std::default_random_engine rng;

    /** A user-specified granularity -- that is, the number of loops
        to run for each event to add some load/granularity for each
        event.

        This instance variable is set to indicate the number of loops
        or iterations that must be executed by the agent to simulate
        some granularity, i.e., CPU used per event.
    */
    size_t granularity;

    /** The maximum delay value that can be generated by a given
        distribution.  This value is computed once (by the first agent
        created) and is used to generate reverse_poisson and
        reverse_exponential distributions.
```

```
151         */
152         static int maxDelay;
153     };
154
155     #endif
```

```cpp
#ifndef PHOLD_AGENT_CPP
#define PHOLD_AGENT_CPP

#include "PHOLDAgent.h"
#include "Event.h"
#include "MTRandom.h"
#include "Simulation.h"
#include <cstdlib>
#include <cctype>
#include <random>
#include <algorithm>
#include <map>

using namespace std;
using namespace XXXX;

// The static maxDelay value
int PHOLDAgent::maxDelay = -1;

PHOLDAgent::PHOLDAgent(AgentID id, PholdState* state, int x, int y,
                       int n, int d, int lookAhead, double selfEvents,
                       size_t granularity, DelayType type) :
    Agent(id, state), X(x),Y(y), N(n), Delay(d), delayType(type),
    lookAhead(lookAhead),  selfEvents(selfEvents), rng(id),
    granularity(granularity) {
     // Setup the random seed used for generating self/other events
    seed = id;
    // Setup the maximum random delay value for reverse distributions
    if ((Delay > 0) && (maxDelay == -1)) {
        maxDelay = getMaxDelayValue();
        std::cout << "maxDelay = " << maxDelay << std::endl;
    }
}

int
PHOLDAgent::getDelay(const DelayType delType) {
    switch (delType) {
    case UNIFORM: {
        std::uniform_int_distribution<int> uni(0, Delay);
        return uni(rng);
    }
    case POISSON: {
        std::poisson_distribution<int> poi(Delay);
        return poi(rng);
    }
    case EXPONENTIAL: {
        std::exponential_distribution<double> exp(Delay);
        return 2 * exp(rng);
    }
    case REVERSE_POISSON: {
        std::poisson_distribution<int> poi(Delay);
        return maxDelay - (poi(rng) % maxDelay);
    }
    case REVERSE_EXPONENTIAL: {
        std::exponential_distribution<double> exp(Delay);
        return maxDelay - ((int) (2 * exp(rng)) % maxDelay);
    }
    case INVALID_DELAY:
    default:
        return 0;
    }
}

int
PHOLDAgent::getMaxDelayValue() const {
    int maxDel = Delay;  // default maximum delay that can be generated.
    std::default_random_engine rnd;
    if ((delayType == POISSON) || (delayType == REVERSE_POISSON)) {
        maxDel = 0;
        std::poisson_distribution<int> poi(Delay);
        for (int i = 0; (i < 100000); i++) {
            maxDel = std::max<int>(maxDel, poi(rnd));
        }
    }
    if ((delayType == EXPONENTIAL) || (delayType == REVERSE_EXPONENTIAL)) {
```



```cpp
        maxDel = 0;
        std::exponential_distribution<double> poi(Delay);
        for (int i = 0; (i < 100000); i++) {
            // The 2 factor is from http://en.cppreference.com/
            // example.  Since exponential distribution is real/double
            // the 2 factor provides a better spread.
            maxDel = std::max<int>(maxDel, 2 * poi(rnd));
        }
    }
    return maxDel;
}

void
PHOLDAgent::initialize() throw (std::exception){
    // We generate N events with random receive times to self
    for (int i = 0; i < N; i++){
        const int RndDelay = (Delay > 0) ? getDelay(delayType) : 0;
        Time  receive(getTime() + 1 + RndDelay);
        ASSERT( receive > getTime() );
        if ( receive < Simulation::getSimulator()->getStopTime() ){
            Event * e = Event::create(getAgentID(),receive);
            scheduleEvent(e);
        }
    }
} // End initialize

#pragma GCC push_options
#pragma GCC optimize ("-O0")
void
PHOLDAgent::simGranularity() {
    for (size_t i = 0; (i < granularity); i++) {
        double sum = 0;
        for (size_t delay = 0; (delay < 25L); delay++) {
            sum += sin(0.5);
        }
    }
}
#pragma GCC pop_options

void
PHOLDAgent::executeTask(const EventContainer* events) {
    PholdState *my_state = dynamic_cast<PholdState*>(getState());
    // For every event we get we send out one event
    for (size_t i = 0; (i < events->size()); i++) {
        // Simulate some event granularity
        simGranularity();
        // First make a random receive time for the future
        const int RndDelay = (Delay > 0) ? getDelay(delayType) : 0;
        const Time  receiveTime(getTime() + lookAhead + RndDelay);

        if ( receiveTime < Simulation::getSimulator()->getStopTime() ){
            // Now we need to choose the agent to send this event to.
            int receiverAgentID = getAgentID();
            if (selfEvents < 0) {
                receiverAgentID = (receiverAgentID + i) % (X * Y);
            }
            if ((selfEvents >= 0) && (selfEvents < 1) &&
                (((rand_r(&seed) % 1000) / 1000.0) > selfEvents)) {
                // Schedule event to different agent.  We do this with
                // equal probability to choose 1 of 4 neighbours.
                const int Change[4] = {-1, -Y, Y, 1};
                // Compute index into the Change array
                int index  = my_state->getIndex();
                // Determine the receiver neighbor
                int new_index = (index + 1) % 4;
                // Update the destination agent.
                receiverAgentID = getAgentID() + Change[index];
                my_state->setIndex(new_index);
                // Handle wrap around cases.
                if(receiverAgentID < 0) {
                    receiverAgentID += X * Y;
                }
                if(receiverAgentID >= (X * Y)) {
                    receiverAgentID -= X * Y;
                }
```

```
151             }
152             ASSERT((receiverAgentID >= 0) && (receiverAgentID < X * Y));
153             // Make event
154             Event * e = Event::create(receiverAgentID, receiveTime);
155             // Schedule the event
156             scheduleEvent(e);
157         }
158     }
159 } // End executeTask
160 
161 void
162 PHOLDAgent::finalize() {}
163 
164 // Convert delay string to enumeration
165 PHOLDAgent::DelayType
166 PHOLDAgent::toDelayType(const std::string& delay) {
167     // Convert delay to all lower-case letters
168     std::string delay_str(delay);
169     std::transform(delay.begin(), delay.end(), delay_str.begin(), ::tolower);
170     // Convert string to delays based on their value.
171     if ((delay_str == "uniform") || (delay_str == "1")) {
172         return UNIFORM;
173     } else if ((delay_str == "poisson") || (delay_str == "2")) {
174         return POISSON;
175     } else if ((delay_str == "exponential") || (delay_str == "3")) {
176         return EXPONENTIAL;
177     } else if ((delay_str == "reverse_poisson") || (delay_str == "4")) {
178         return REVERSE_POISSON;
179     } else if ((delay_str == "reverse_exponential") || (delay_str == "5")) {
180         return REVERSE_EXPONENTIAL;
181     }
182     return INVALID_DELAY;
183 }
184 
185 void
186 PHOLDAgent::printDelayDistrib(std::ostream& os) {
187     // Generate occurrences of random delay delay values.
188     std::map<int, int> hist;
189     for (int n = 0; (n < 10000); ++n) {
190         ++hist[getDelay(delayType)];
191     }
192     // Determine highest frequency of occurrences to scale histogram.
193     int maxFreq = 0;
194     for (const auto& p : hist) {
195         maxFreq = std::max(maxFreq, p.second);
196     }
197     // Now print vertical histogram (maxFreq == 75 stars).
198     for (const auto& p : hist) {
199         os << p.first << "("  << p.second << ")"
200            << std::string(p.second * 75 / maxFreq, '*') << std::endl;
201     }
202 }
203 
204 #endif
```

```
1  #ifndef _PHOLDSTATE_H
2  #define _PHOLDSTATE_H
3
4  #include "State.h"
5
6  class PholdState : public XXXX::State {
7
8  public:
9      State * getClone();
10     PholdState();
11
12     inline int getIndex() const {return index;}
13     inline void setIndex(int new_index) {index=new_index;}
14
15 private:
16     int index;
17 };
18
19 #endif
```

```cpp
#ifndef _PHOLDSTATE_CPP
#define _PHOLDSTATE_CPP

#include "PholdState.h"

PholdState::PholdState() : index(0) {}

XXXX::State*
PholdState::getClone(){
    //keep in mind this shallow copy works because there are no complex
    //pointers that i need to take care of. Just primative datatypes.
    PholdState* clone = new PholdState();
    clone->setIndex(this->getIndex());
    clone->timestamp = this->timestamp;
    return clone;
    //Caller should handle deleting the memory when done with State pointer
}

#endif
```