ABSTRACT

MANAGING PENDING EVENTS IN SEQUENTIAL & OPTIMISTIC PARALLEL DISCRETE EVENT SIMULATIONS

by Julius Didier Higiro

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 multi-tiered (2 and 3 tier) data structures, including our proposed 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 the PHOLD benchmark and the PCS simulation model 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.

# MANAGING PENDING EVENTS IN SEQUENTIAL & OPTIMISTIC PARALLEL DISCRETE EVENT SIMULATIONS

Thesis

Submitted to the

Faculty of Miami University

in partial fulfillment of

the requirements for the degree of

Master of Science in Computer Science

by

Julius Didier Higiro

Miami University

Oxford, Ohio

2017

Advisor: Dr. Dhananjai M. Rao

Reader: Dr. Matthew Stephan

Reader: Dr. Karen Davis

This thesis titled

MANAGING PENDING EVENTS IN SEQUENTIAL & OPTIMISTIC PARALLEL DISCRETE EVENT SIMULATIONS

by

Julius Didier Higiro

has been approved for publication by

College of Engineering and Computing

and

Department of Computer Science and Software Engineering

---

Dr. Dhananjai M. Rao

---

Dr. Matthew Stephan

---

Dr. Karen Davis

# Contents

# List of Tables

# List of Figures

# Acknowledgements

Thank you to my adviser Dr. Dhananjai M. Rao for his encouragement and support during the completion of this thesis and throughout my time at Miami University. Without his guidance, I certainly would not be where I am now. Additionally, I would like to acknowledge my committee members Dr. Matthew Stephan, and Dr. Karen Davis for their support and input.

# Chapter 1

# Introduction

Discrete event simulation (DES) is a computational methodology for modeling and analysis of a wide spectrum of systems. In DES, the system being modeled is logically subdivided into small, independent, but interacting entities with their own independent states. The model and implementation of an entity is called a Logical Process (LP), which manage the state assocated with them. Accordingly, a DES is essentially designed as a set of logical processes (LPs) that interact with each other. LPs interact by exchanging and processing discrete-timestamped events or messages [1]. Processing an event essentially introduces a change in an LP's state and causes the LP to generate additional events to itself or other LPs in the model.

A key aspect of DES is that state changes occur at discrete times [2]. At each point in time in a simulation, a virtual time-stamp is assigned to an event and the event precipitates a transition from one state to another state. This change in system state is used to represent the dynamic nature and behavior of a real-world system [3].

DES has been used in a variety of fields in academia, industry, and the public sector as a tool to help inform knowledge and to improve decision-making processes [2]. DES provides an effective means for analyzing real or artificial systems without the constraint of limited resources such as time, financial costs, or safety. For example, the simulation of a battlefield environment can deliver insightful information to military planners on enemy troop movements, tactics, and capabilities during strategic planning efforts [4]. A discrete event simulation of the battlefield allows military

leaders to examine the impacts of decisions without the real-world risks associated with committing forces to dangerous environments.

## 1.1 Parallel Simulation

Parallelism in computing frameworks that support DES increase performance throughput that is needed to construct and execute large scale and complex simulation models. With the growth and prevalence of semiconductor technology, cheaper and powerful multi-processors can be instrumented to achieve greater computing power for parallel discrete event simulations (PDES) [5, 6]. In parallel simulations, LPs are subdivided or partitioned to operate on different compute units. However, event processing on the different compute units must be synchronized to ensure causally consistent event processing. Consequently, the speedup achieved using multi-core and multi-processor systems requires efficient strategies to minimize synchronization costs.

Currently, two broad types of synchronization methods are used in PDES, namely: conservative and optimistic approaches [1]. Conservative methods tightly coordinate event processing so that causal violations do not occur. Optimistic methods, such as Time Warp [1], loosely synchronize LPs – they permit temporary causal violations to occur but detect and recover from causal violations. Recently, optimistic synchronization methods have outperformed conservative methods for certain classes of systems [1].

## 1.2 Managing Pending Events

Sequential and parallel DES are designed as a set of logical processes (LPs) that interact with each other by exchanging and processing timestamped events or messages [1]. 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 [7–10]. The effectiveness of data structures for event management is a conspicuous issue in larger simulations, where thousands or millions

of events can be pending [11, 12]. 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. Furthermore, the synchronization strategy used in Time Warp (an optimistic synchronization strategy) can further impact the effectiveness of the data structure due to additional processing required during rollback-based recovery operations.

## 1.3 Thesis Statement

This research proposes and explorers multi-tier data structures for the improved management of the pending event set in sequential and optimistic parallel simulations. The objective of the research is to develop and assess effectiveness of novel data structures for managing pending events. Specifically, this thesis proposes multi-tiered data structures called 2-tier Ladder Queue (2tLadderQ) and 3-tier Heap (3tHeap) for managing pending events. We conduct experimental assessment of the proposed data structures by comparing their effectiveness using benchmark simulations and a fine-tuned version of the Ladder Queue [13]. We use the Ladder Queue, with amortized O(1) time complexity for comparison because it has shown to to be very efficient for sequential DES.

**Thesis:** The multi-tiered, **2tLadderQ** and **3tHeap** pending event structures outperform other priority queue based implementation of the pending event set, specifically, **Ladder Queue**, **Binary Heap**, **Binomial Heap**, **Fibonacci Heap** and **2-Tier Heap**. The contributions of the thesis are the development of two novel data structures **2tLadderQ** and **3tHeap** and the identification of key influential model characteristics for determining the choice of scheduler queue.

# Chapter 2

# Background and Related Work

Many investigations have explored the effectiveness of a wide variety of data structures for managing the pending event set in sequential and parallel discrete event simulation (PDES). The prior investigations in PDES area fall under two broad categorizes, namely: shared memory versus distributed memory data structures. Shared memory data structures focus on managing pending events in PDES that use multiple threads for parallelism. Such simulations are typically performed on large shared memory machines with many-core CPUs or dedicated GPGPUs or coprocessors. These data structures focus on enabling concurrent access to add or remove pending events from multiple threads while avoiding race conditions. Race conditions are voided using conventional lock-based approaches such as semaphores or mutexes. Recently, lock-free data structures based on special check-and-set (CAS) instructions have also been proposed to enable efficient, thread-safe, and concurrent access.

Distributed memory data structures focus on enabling efficient single-threaded operation. However, these data structures need to enable managing events received over communication channels from other remote processes involved in PDES. This thesis focuses on sequential simulations as well as distributed memory PDES based on Time Warp synchronization. Accordingly, this chapter focuses on closely related work in sequential and distributed memory PDES.

Dickman [14] 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

pending event set data structure in shared memory PDES. A central component of their study was the identification of an appropriate data structure and design for the shared pending event set. Gupta [15] 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 pending event set. The modification involved the use of an unsorted lock-free queue in the underlying Ladder Queue structure. Marotta [16] contributed to the study of pending event set data structures in threaded PDES through the design of the Non-Blocking Priority Queue (NBPQ) data structure. A pending event set data structure that is closely related to Calendar Queues with constant time performance [17].

Recently, Franceschini [10] compared several priority-queue based pending event data structures to evaluate their performance in the context of sequential DEVS simulations. They found that Ladder Queue outperformed every other priority queue based pending event data structure such as Sorted List, Minimal List, Binary Heap, Splay Tree, and Calendar Queue. Tang [13] and Franceschini [10] both use the classic Hold benchmark simulation model used in this research [17].

## 2.1 Ladder Queue (ladderQ)

The Ladder Queue (ladderQ) is a priority queue implementation proposed by Tang et al [13] with amortized constant time complexity. 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 [10, 13, 14]. There are two key ideas underlying the Ladder Queue, namely: minimize the number of events to be sorted and delay sorting of events as much as possible. However, in contrast to the **ladderQ**, the other data structures always fix-up and maintain a minimum heap property [17].

As shown in Figure 2.1, the ladder queue consists of the following 3 substructures:

1. *Top*: An unsorted list which contains events scheduled into the distant future or epoch [17].
2. *Ladder*: Consists of multiple rungs, *i.e.,* list of buckets. Each bucket contains list of events with a finite range of time stamp 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 and the maximum number of rungs in Ladder is 8 [17].

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. The default threshold value at which events from Bottom are moved into Ladder is 50 [13, 17].

At the beginning of a simulation, enqueue operations only involve the insertion of events into *Top*. As the simulation progresses, the insertion of events can occur at any level of the data structure. The insertion of events in *Top* and Ladder is an O(1) operation that involves appending events to a list that remains unsorted. The onset of dequeue operations involves moving unsorted events from *Top* into a newly formed rung in Ladder. The time range or bucket-width of a rung is established by taking the difference between the highest and lowest time stamp and dividing the difference by the total number of events. As shown in Figure 2.1, the bucket-width computed from the time stamp in *Top* is (6.0**max** - 1.0**min**)/ 10 = 0.5. In accordance with their timestamps, events from *Top* are placed into the appropriate buckets in $\text{Rung}_1$. In cases, where the number of events in a bucket exceeds the established threshold, a new rung is generated to store those events. For example, in Figure 2.1, $\text{Rung}_2$ is generated for time stamped events in the range of 1.0 to 1.5. Next, the bucket containing events are sequentially removed from the bottom most rung ($\text{Rung}_2$) in Ladder into the lower substructure. The events are inserted in sorted LTSF order into *Bottom*, where events are dequeued for further processing. The clearing of events in Ladder and Bottom kickoffs the movement of additional events from *Top* into the two lower substructures. The implementation of Ladder Queue in MUSE adheres to the functionality described in [13] with some modifications.

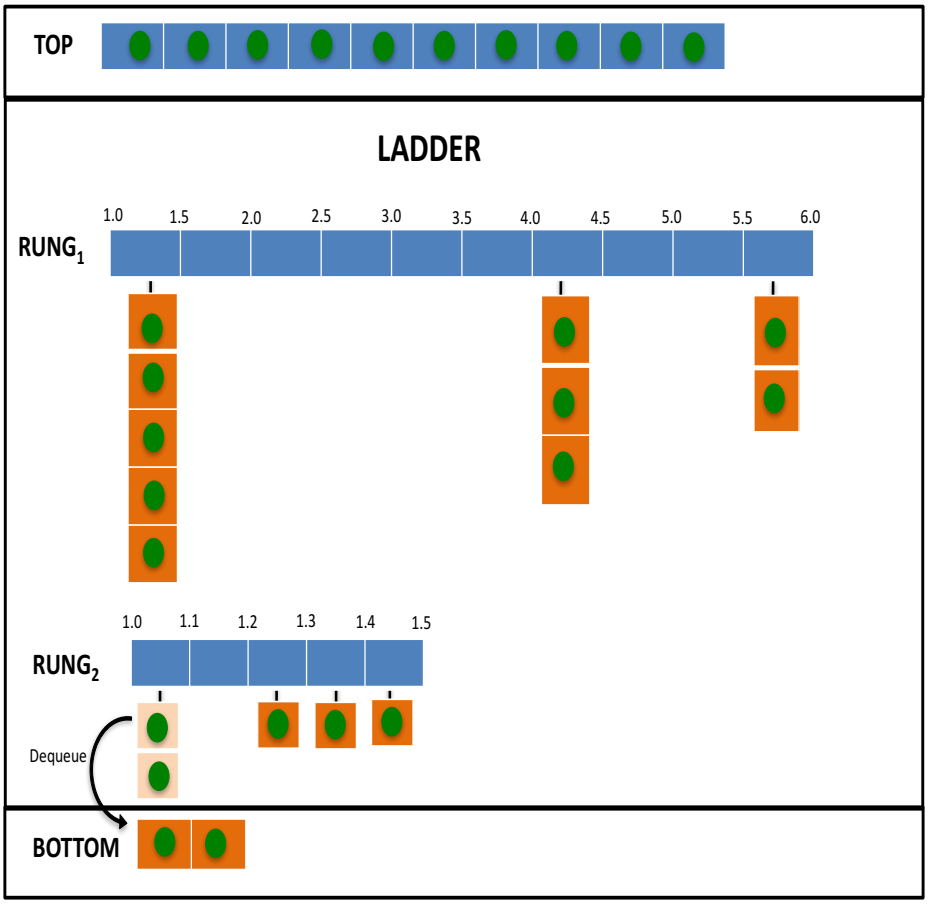



**Figure 2.1: Structure of Ladder Queue**
**Source: Tang et al. [13]**

## 2.2 Distinguishing aspects of this research

Our research 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, the Fibonnacci heap (**fibHeap**) and our 3-tier Heap (**3tHeap**) are unique data structures that have potential to be effective in simulations with high concurrency.

Since it has been established that the Ladder Queue (ladderQ) outperforms other data structures, we aim to use it for empirical assessment of our proposed data structures. However, in contrast to existing work, rather than using a linked list based implementation, we propose an alternative implementation using dynamically growing arrays, that is, std::vector from the C++ library. 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.

## 2.3 Miami University Simulation Environment (MUSE)

The implementation and assessment of the different data structures was conducted using our parallel simulation framework called Miami University Simulation Environment (MUSE). The application was developed as part of a master's thesis written by Meseret Gebre in the Department of Computer Science at Miami University in 2009 [18]. MUSE was developed in C++ and uses the Message Passing Interface (MPI) library for parallel processing. It also uses Time Warp and standard state saving approach to accomplish optimistic synchronization of the LPs to maintain causality in event processing.

A conceptual overview of a MUSE-based parallel simulation is shown in Figure 2.2. The simulation kernel implements core functionality associated with LP registration, event processing, state saving, synchronization and Global Virtual Time (GVT) garbage based collection [17]. Each LP in a simulation maintains an input, output and state queue. The input queue is used to retain

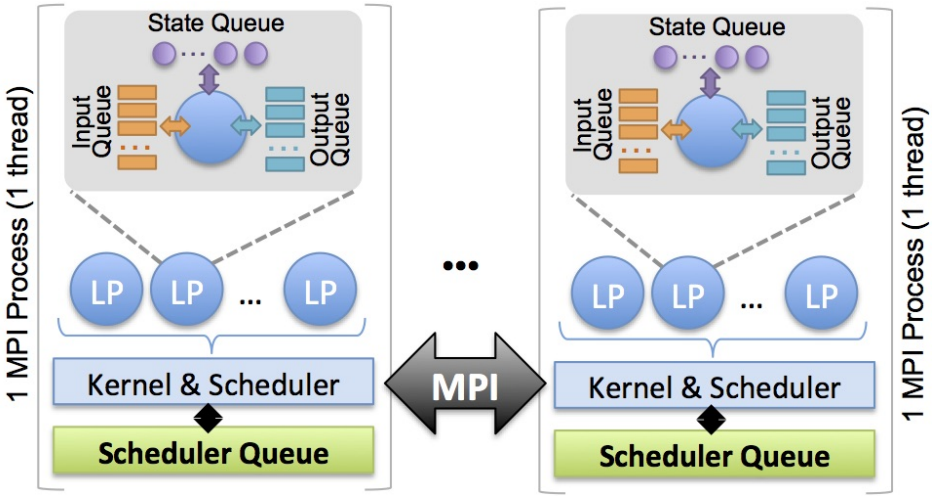



**Figure 2.2: Overview of a parallel MUSE simulation [18]**

events that have already been processed but have not yet been garbage collected. The output queue stores anti-messages, which are events that are sent to other LPs to cancel out previously sent events. The state queue stores the state of the LP at each discrete point in virtual simulation time. A Time Warp LP also maintains a local virtual time (LVT) that is updated to the time-stamp of the event most recently processed by the LP.

In a Time Warp based simulation such as MUSE, the simulation is organized as a set of LPs that interact with each other by exchanging virtual times-tamped events. LPs process events in non-decreasing receive-time order and generate new events that are transmitted to LPs on local or remote processors. Synchronization of event processing is achieved through the adherence to the local causality constraint, which requires that LPs only process events in Least Time-stamp First (LTSF) order [1]. This is in contrast to the conservative synchronization protocol that blocks event processing until it is guaranteed that an LP cannot receive a future event with a receive-time lesser than it's LVT (altogether avoiding the manifestation of causality errors) [1].

A singular advantage of the Time Warp approach in parallel simulations is the ability to withstand violations of the causality constraint. Time Warp LPs proceed optimistically with event processing and during occasions that an LP encounters an event (*named a* ***straggler***) with a receive time lesser than the LVT, a rollback operation is performed. A rollback requires that an LP undo all event processing that occurred at the LVT equal to the straggler time stamp and forward. The

LP performs a rollback to a state with an LVT preceding the straggler time stamp and it sends an anti-message to all other agents with the purpose of cancelling the previously sent events.

As shown in Figure 2.2, the kernel also maintains a centralized LTSF scheduler queue for managing pending events and scheduling event processing for local LPs. LPs are permitted to generate events only into the future –*i.e.,* the time stamp on events must be greater than their Local Virtual Time (LVT). Consequently, with a centralized LTSF scheduler, event exchanges between local LPs cannot cause rollbacks. Only events received via MPI can cause rollbacks in our simulation. 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 discussed in the subsequent chapter. A scheduler queue is required to implement the following key operations to manage pending events [17]:

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 [17].

❷ **Peek next event**: This operation is expected to return the next event to be processed. This information is used to determine next LP and to update its LVT prior to event processing. Note that peek does not dequeue events [17].

❸ **Dequeue events for next LP**: In contrast to peek, this operation is expected to dequeue the events to be dispatched for processing by an LP. This operation is performed by the kernel immediately after a peek operation. The operation must dequeue the next set of concurrent events, *i.e.,* events with the same receive time sent to an LP. However, the concurrent events could have been sent by different LPs on different MPI-processes. Dispatching concurrent events in a single batch streamlines modeling broad range of scenarios. An total order within concurrent events is not imposed but can be readily introduced if needed [17].

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 ($\text{LP}_{\text{sender}}$) to another LP ($\text{LP}_{\text{dest}}$) at-or-after a given time ($t_{rollback}$). In our implementation, only one anti-message with send time $t_{rollback}$ is dispatched to $\text{LP}_{\text{dest}}$ from $\text{LP}_{\text{sender}}$ to cancel prior events sent by $\text{LP}_{\text{sender}}$ to

$LP_{dest}$ at-or-after $t_{rollback}$. This is a contrast to conventional aggressive cancellation in which one anti-message is generated per event. 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. Note that this feature is reliant on the First-In-First-Out (FIFO) communication guarantee provided by MPI [17].

## 2.4 Experimental Platform

The design of MUSE and the experiments reported were conducted using a distributed-memory compute cluster consisting of 80 compute nodes interconnected by 1 GBPS Ethernet. Each compute node has 8 cores from two quad-core Intel Xeon ®CPUs (E5520) running at 2.27 GHz with hyper-threading 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 with OpenMPI 1.6.4. All debug assertions were turned off for maximum performance [17].

# Chapter 3

# Simulation Benchmarks

We conduct assessment of data structures for managing pending events using both mathematical and experimental approaches. Mathematical approaches use time complexity analyses of the data structures to provide asymptotic comparisons of the data structures. However, the theoretical analysis often requires simplifying assumptions to ensure the analysis is mathematically tractable. Such simplifying assumptions do not account for time constants and peculiar runtime characteristics of a Parallel Discrete Event Simulation (PDES). These factors play a crucial role in practical applicability and effectiveness of data structures. Consequently, in the PDES domain, experimental approaches are strongly favored over analytical approaches.

Experimental analyses of parallel simulations is typically conducted using target models. However, for general purpose solutions such as the one proposed in this research, synthetic benchmarks are used to provide model agnostic analysis. Several synthetic benchmarks have been proposed by the PDES community in the past. Among the various benchmarks, the PHOLD and PCS benchmarks have gained general acceptance and are widely used for experimental analysis [19]. These benchmarks have been used for empirical analysis and are discussed in further detail in the following sections.

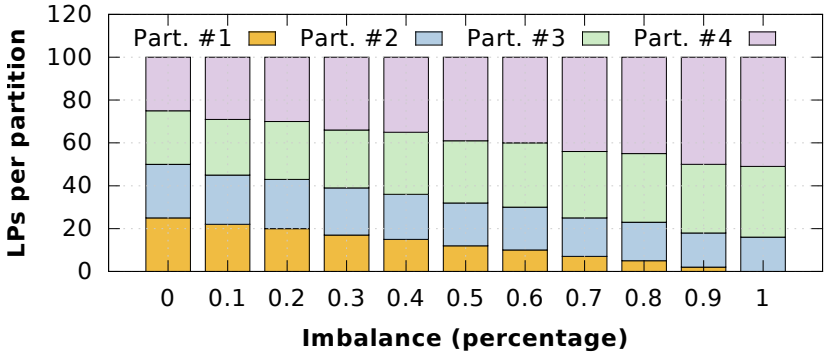
## 3.1 Parallel HOLD (PHOLD)

Experimental analysis has been conducted using a parallelized version of the classic Hold synthetic benchmark called PHOLD. It has been used by many investigators because it is shown to effectively emulate the steady-state phase of a typical simulation [10, 13]. Our PHOLD implementation developed using MUSE provides several parameters (specified as command-line arguments) summarized in Table 3.1. The benchmark consists of a 2-dimensional toroidal grid of 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 Figure3.1(a). The **imbalance** parameter has no impact in sequential simulations [17].

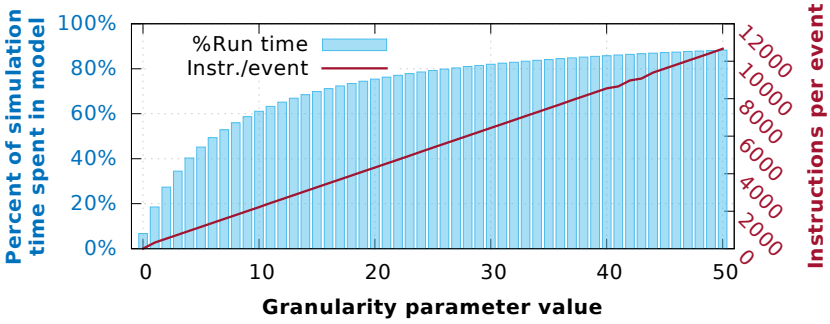
**Table 3.1: Parameters in PHOLD benchmark [17]**

| 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 $\lambda$ | Value used with distribution – Lambda ($\lambda$) 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. |

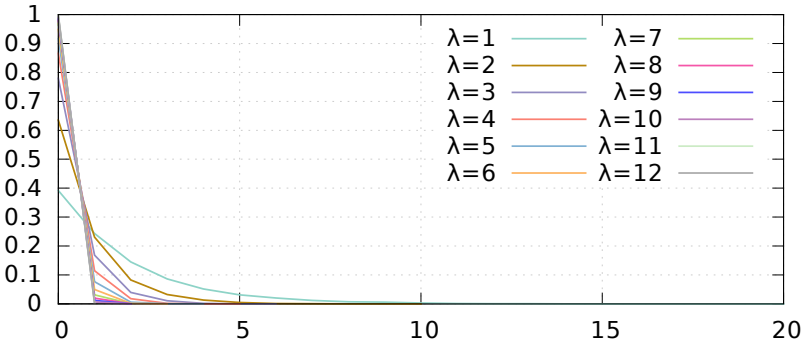
The PHOLD simulation commences with a fixed number of events for each LP, specified by the **eventsPerLP** parameter. For each event received by an LP a fixed number of trigonometric operations determined by **granularity** are performed to place CPU load. The impact of increasing the **granularity** 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




(a) Impact of **imbalance**




(b) Impact of **granularity**




(c) Impact of **delay** ($\lambda$)

**Figure 3.1: Impact of varying key parameter values in the PHOLD model [17]**

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 $\lambda$ parameters. Our experiments use an exponential distribution for timestamps because it reflects event distribution commonly found in a broad range of simulation models [13]. Time stamp of events is computed as $t_{recv}$ = LVT + 1 + $\lambda e^{-\lambda x}$. The impact of changing the $\lambda$ (*i.e.,* **delay**) is shown in Figure 3.1(c) –smaller values of $\lambda$ provide a broader range of time stamp value for future events resulting in fewer concurrent events per LVT. Conversely, larger $\lambda$ values cause timestamps to be close to the current epoch, increasing both the number of concurrent events per LVT and the possibility of rollbacks. The impact of these parameters on scheduler queue performance were explored using 2,500 different configurations [17].

## 3.2 Personal Communication Service Network (PCS)

We performed experimental analysis using a network communication simulation model named PCS. The model was developed to simulate large scale wireless communication networks [20]. The implementation uses parameters summarized in Table 3.2. The PCS model consists of **Cells** (i.e. cellular towers) that transmit and receive phone calls made by mobile cellular phone units that reside at each Cell. A Cell is the central entity (LP) object type for the simulation. The Cells contain a fixed number of channels that are licensed to individual mobile phone units known as **Portables**. A **channel** is a wireless channel via which a **portable** can send/receive information from a Cell. A Portable at a given cell communicates to local or remote Portables, if a channel is available. Otherwise, the call is considered a blocked phone call. The Portables are mobile and travel to various cells throughout the network. The MUSE implementation of PCS models each cell as an LP and a portable as an event.

The PCS simulation commences with a fixed number of portables/events and channels for each Cell/LP, specified by the **portables** and **channels** parameters. The portables contain three exponentially distributed timestamps fields with means specified by **moveIntervalMean**, **callIntervalMean** and **callDurationMean** parameters. The minimum of the timestamps is used to determine the behavior of the portables (i.e., completion of a phone call, arrival of the next portable call at a cell,

and the departure of a portable from its current cell to a neighboring cell). We will use the PCS simulation to validate resulting experimental analysis using the synthetic benchmark PHOLD. The evaluation of the data structures using **PCS** provides a way to evaluate the persistence of trends and outcomes observed in the **PHOLD** simulation across a different model.

**Table 3.2: Parameters in PCS Model**

| Parameter | Description |
|---|---|
| **rows** | Total number of rows in model. |
| **cols** | Total number of columns in model. #Cells/LPs = **rows** × **cols**. |
| **portables** | The Portable represents a mobile phone unit that resides within the Cell for a period of time and then moves to one of the four neighboring Cells |
| **moveIntervalMean** | This value represents the mean used for an exponential random distribution used to generate the time when a portable will move to an adjacent cell. |
| **callIntervalMean** | The mean time between two successive calls to a portable associated with this Cell. This value is the mean of an exponential random distribution from where the next call timestamp value is determined. |
| **callDurationMean** | The mean call completion time. This value represents the mean used for a Poisson distribution used to generate the length/duration of a call to a portable. |
| **#channels** | The maximum number of channels assigned to each PCS cell. |
| **imbalance** | Fractional imbalance in partition to have more LPs on a MPI-process. |
| **simEndTime** | GVT when simulation logically ends. |

## 3.3 Performance Metrics

As previously stated, we compared the effectiveness of our various PES structures to include the novel structures (**2tLadderQ** and **3tHeap**) against the fine-tuned version of the Ladder Queue using two benchmark simulations. The assessment of the data structures involved the following metrics:

❶ **Raw execution run time**: This is the elapsed time between the start and the end of program execution. In a parallel program, the elapse time requires the end time of the last process to finish program execution. Serial and parallel wall clock timings were collected and used in the performance analysis of the various data structures.

❺ **Peak Memory Usage**: The total memory used by a serial or parallel program.

❻ **Cache Hits or Misses**: Number of times that accessed data is found or not found to reside in cache memory.

# Chapter 4

# Scheduler Queues

The pending events are managed by distinct scheduler queues that utilize different data structures to implement the 4 key operations (i.e. enqueue, peek dequeue, and cancel). We have compared the effectiveness of 7 different non-intrusive queue data structures namely: ① binary heap (**heap**), ② binomial heap (**binomHeap**), ③ 2-tier heap (**2tHeap**), ④ 2-tier Fibonacci heap (**fibHeap**), ⑤ 3-tier heap (**3tHeap**), ⑥ Ladder Queue (**ladderQ**), and ⑦ 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** and **binomHeap** use only a single data structure for accomplishing the 4 key operations. Conversely, multi-tier queues organize events into tiers, with each tier implemented using different data structures. Table 4.1 summarizes the algorithmic time complexities of the 7 data structures discussed in the following subsections [17].

**Table 4.1: Comparison of algorithmic time complexities of different data structures [17]**

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

| Name | Enqueue | Dequeue | Cancel |
|---|---|---|---|
| **heap** | $\log(e \cdot l)$ | $\log(e \cdot l)$ | $z \cdot \log(e \cdot l)$ |
| **binomHeap** | $\log(e \cdot l)$ | $\log(e \cdot l)$ | $z \cdot \log(e \cdot l)$ |
| **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** | $1^*$ | $1^*$ | $e \cdot l \div {}_{t2}k$ |

## 4.1 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. Figure 4.1 shows a collaboration UML diagram for the `HeapEventQueue` implementation in MUSE. A **std::vector** is used as the backing container (`eventList` in Figure 4.1) and algorithms (**std::push_heap**, **std::pop_heap**) are used to maintain the heap. The heap is prioritized on both time stamp and LP's *ID* (to dequeue batches of events), with the lowest time stamp at the root of the heap [18]. 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 $\log(e \cdot l)$ as shown in Table 4.1. 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 [17].

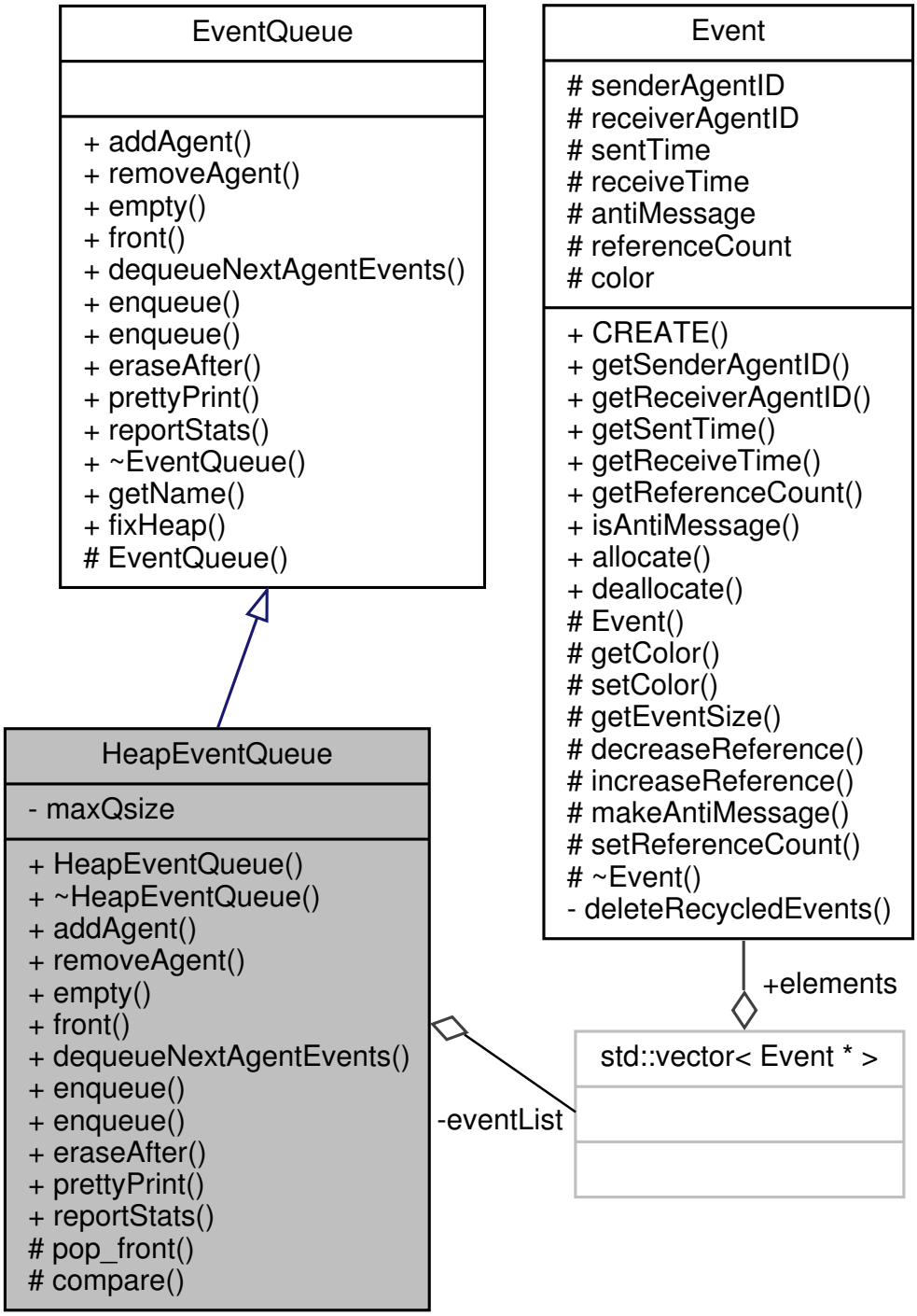



**Figure 4.1: Collaboration UML diagram for Binary Heap based event queue implementation in MUSE**

## 4.2 Binomial Heap (binomHeap)

The **binomHeap** is another single-tier data structure that uses Binomial heap from the BOOST C++ library. In similarity to **heap**, **binomHeap** is prioritized on both time stamp and LP's *ID*, with the lowest time stamped event at the root. Additionally, the operations on **binomHeap** are logarithmic with a time complexity for enqueue and dequeue operations of $\log(e \cdot l)$. A special property of Binomial heap is the ability to merge two heaps into a single heap with a time complexity of $\log(n)$. Given that **binomHeap** is a single-tier data structure and all of the LPs on an MPI-process share a PES queue, the merge operation is not relevant.

### 4.2.1 Run time comparison of heap vs. binomHeap

Due to similarities between Binary and Binomial Heap, we performed a comparison of **heap** and **binomHeap** to identify which one of the two data structures was the more effective priority queue based implementation of the PES queue. The comparison involved the production of serial run times for 2500 different configurations of PHOLD benchmark for both data structures. As shown in Figure 4.2, **heap** had a lower serial run time than **binomHeap**. The average run time for **heap** was 43.87 seconds and the average run time for **binomHeap** was 52.17 seconds. An unpaired two sample t-test was performed on the two sample run times to determine if averages were generally different. The data showed t-stat > t-critical (1.96) and the p-value: 2.91E-315 « 0.05. Thus showing the averages were statistically different. A paired two sample t-test was also conducted and the paired t-test resulted in a p-value 2.01E-315 « 0.05. Therefore, the null hypothesis (H0: difference between samples is zero) was rejected and we concluded that **heap** is generally faster than **binomHeap**. Based on the results, Binomial Heap was not used in any further implementation and assessment of the PES queue.

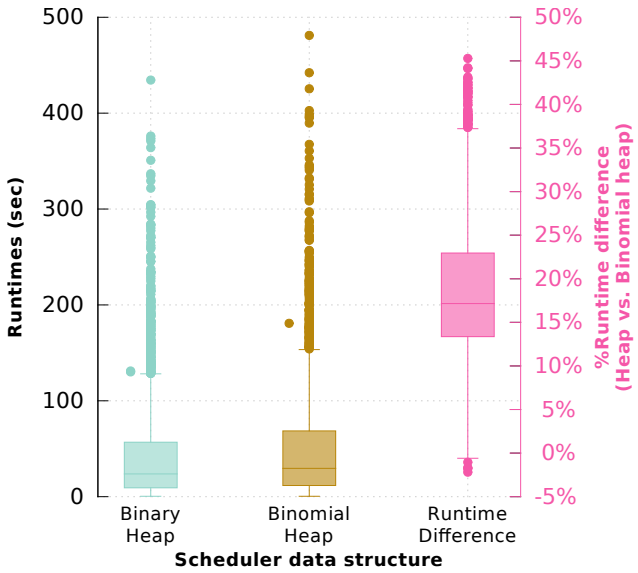
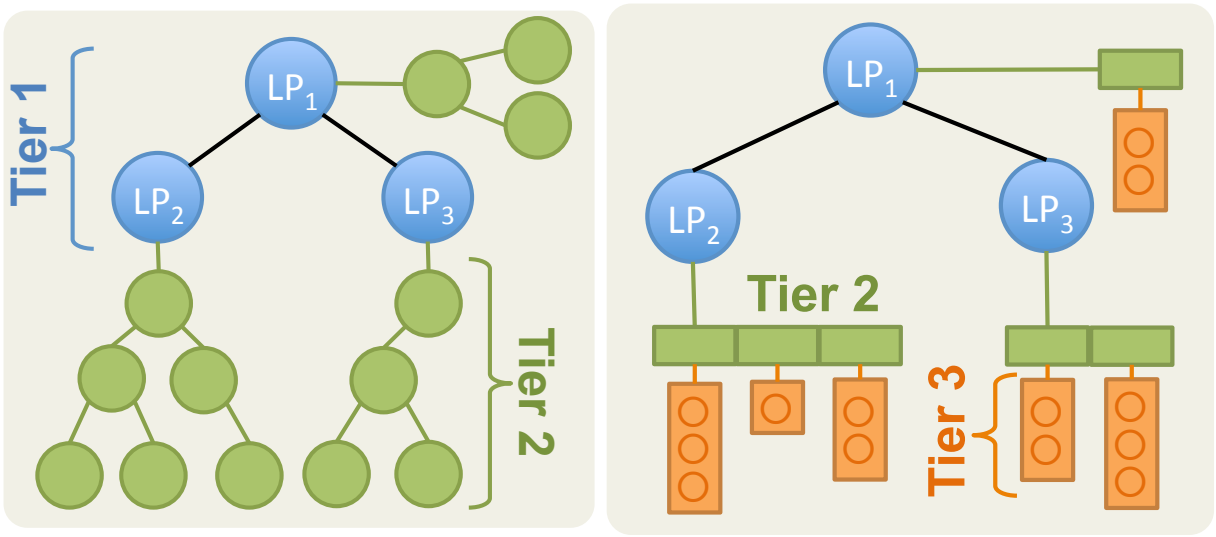



**Figure 4.2: Comparison of heap and binomHeap execution time**

## 4.3 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 4.3. 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. A standard `std::vector` is used as the backing container to maintain pointers to the pending events [17]. Standard heap operations implemented by C++ `algorithms` library are used to maintain the heap property. In MUSE, the **2tHeap** has been implemented in a class called `TwoTierHeapEventQueue`. Figure 4.4 shows a collaboration diagram illustrating the relationship between this class and other key classes in MUSE [18].




(a) 2-tier Heap

(b) 3-tier Heap

**Figure 4.3: Structure of 2-tier & 3-tier heap**

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 $_{\text{dest}}$ with utmost 1 tier-1 operation to update schedule position of LP $_{\text{dest}}$. 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 [17].

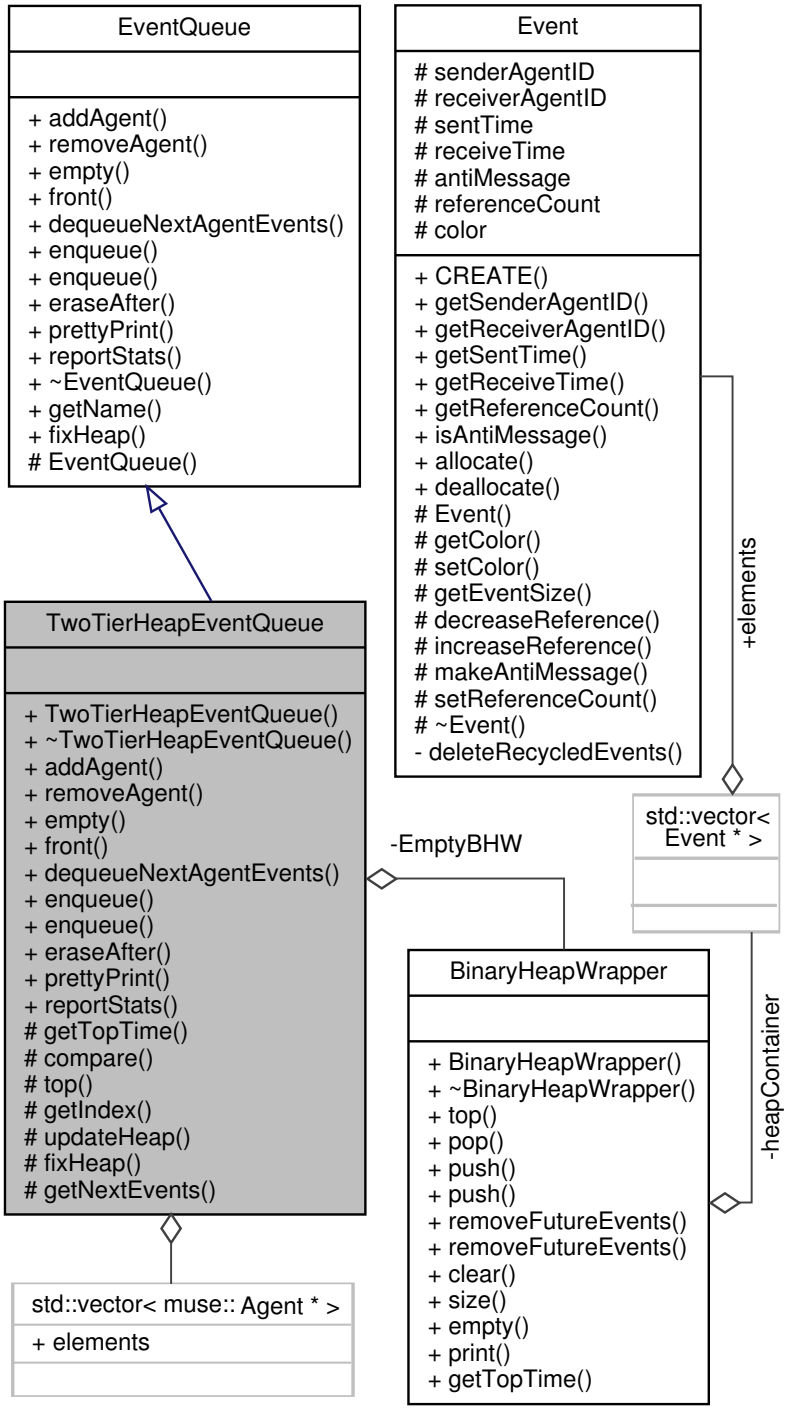



**Figure 4.4: Collaboration UML diagram for Two-tier Heap based event queue implementation in MUSE**

## 4.4 2-tier Fibonacci Heap (fibHeap)

The **fibHeap** is an extension to the previous **2tHeap** data structure and uses a Fibonacci heap for scheduling LPs. The Fibonacci heap is a slightly modified version from the BOOST C++ library. The Fibonacci heap has an amortized constant time for changing key values and finding minimum. Consequently, we use it for the first tier which is responsible for scheduling LPs and use a standard binary heap for the second tier. We do not use Fibonacci heap for the second tier because we found its runtime constants to be higher than a binary heap. Accordingly, the time complexity for enqueue and dequeue operations is $\log(e) + 1^*$ [17].

## 4.5 Three-tier Heap (3tHeap)

The **3tHeap** builds upon **2tHeap** by further subdividing the second tier into two tiers as shown in Figure 4.3(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 4.1. 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 de-allocation overhead [17].

## 4.6 Ladder Queue (ladderQ)

The **ladderQ** is a widely used priority queue implementation that has been used as the reference for experimental assessments. The `ladderQ` data structure is discussed in detail in Sec-

tion 2.1. Similar to the other event queue implementation, the **ladderQ** has been integrated into MUSE by suitably implementing the necessary Application Program Interface (API) methods in the `muse::EventQueue` base class. The various sub-data structures such as *Top*, *Rung*, and *Bottom* have been implemented using sub-classes as shown in Figure 4.5.

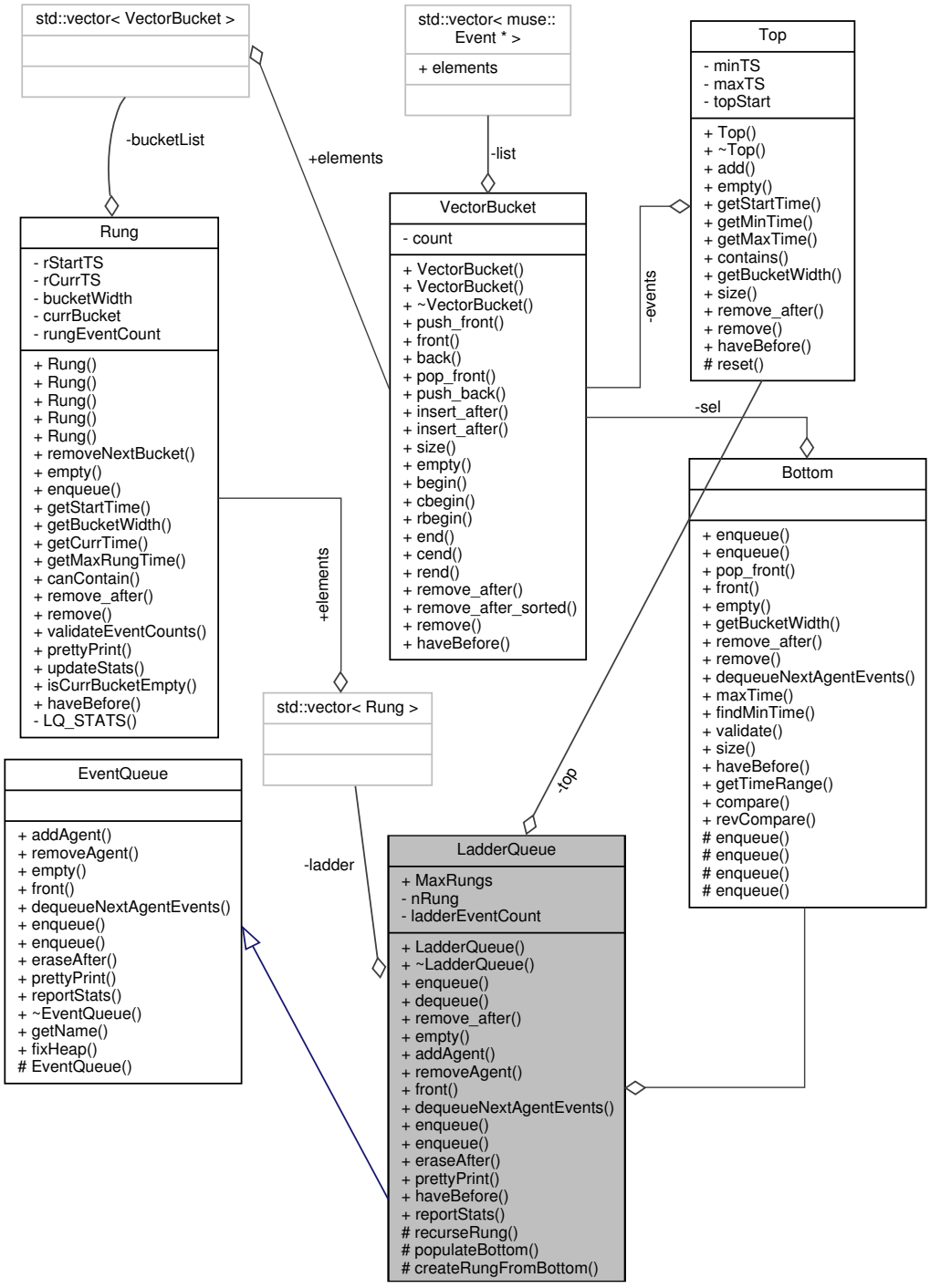


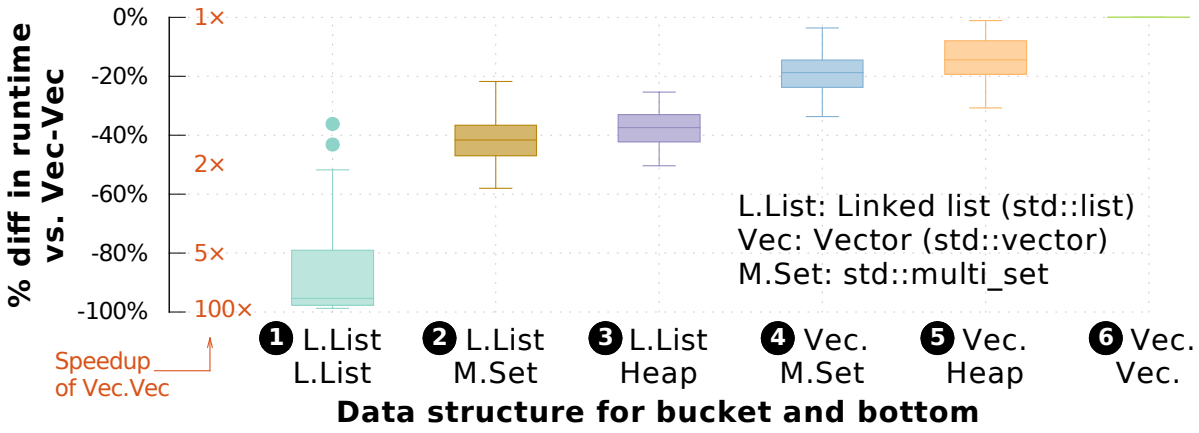

**Figure 4.5: Collaboration UML diagram for Ladder Queue based event queue implementation in MUSE**

### 4.6.1 Fine tuning Ladder Queue performance

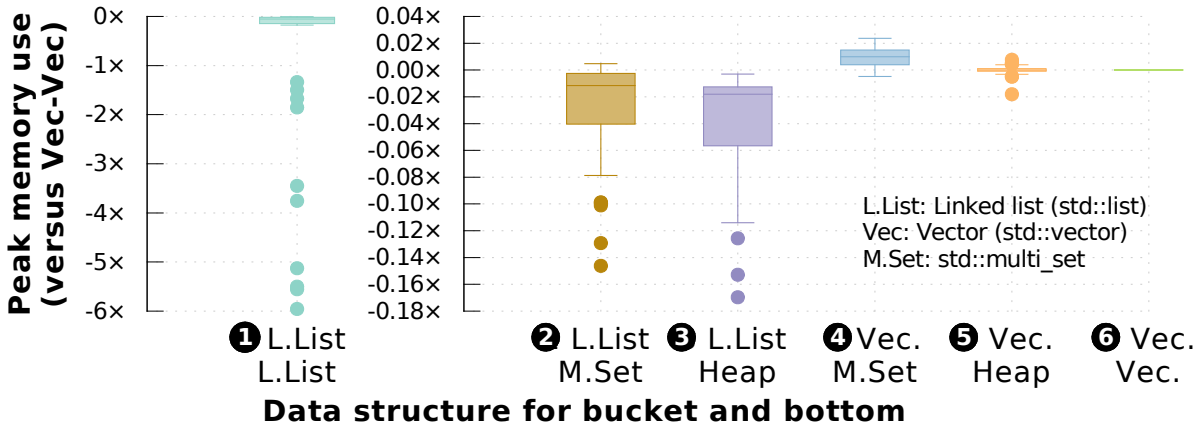
Our implementation closely followed the design in the original paper by Tang et al [13]. However, to minimize runtime constants, we have explored different configurations for the buckets and the *Bottom* in the **ladderQ**. Specifically, we have explored the following 6 different configurations – ❶ L.List-L.List: 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 [17].

Runtime comparison of the 6 **ladderQ** configurations is summarized in Figure 4.6. The data was obtained using **PHOLD** with different parameter settings. The $❻^{th}$ Vec-Vec configuration was the fastest and performance of other configurations are shown relative to it in Figure 4.6(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 4.6(b), in comparison with the Vec-Vec configuration. As shown by the charts in Figure 4.6, the increased performance of Vec-Vec comes at about a 6×increase in peak memory footprint when compared to L.List-L.List configuration. This increased footprint arises because the **std::vector** internally doubles its capacity as it grows. With many buckets in the **ladderQ**, each implemented using a **std::vector**, the overall peak memory footprint is higher. Certainly, the increased capacity is used if the number of events in buckets grow. However, the Vec-M.Set and Vec-Heap configurations consume a bit more memory in some configurations, showing that Vec-Vec is not the worst in memory consumption. Consequently, we use the Vec-Vec configuration as it provides the fastest performance among the 6 configurations [17].

The maximum number of rungs in the *Ladder* also influences the overall performance of the **ladderQ** [13]. The chart in Figure 4.7 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




(a) Comparison sequential runtimes




(b) Peak memory used

**Figure 4.6: Comparison of execution time and peak memory for PHOLD benchmark (different parameter settings) using 6 different ladderQ configurations [17]**

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 [17].

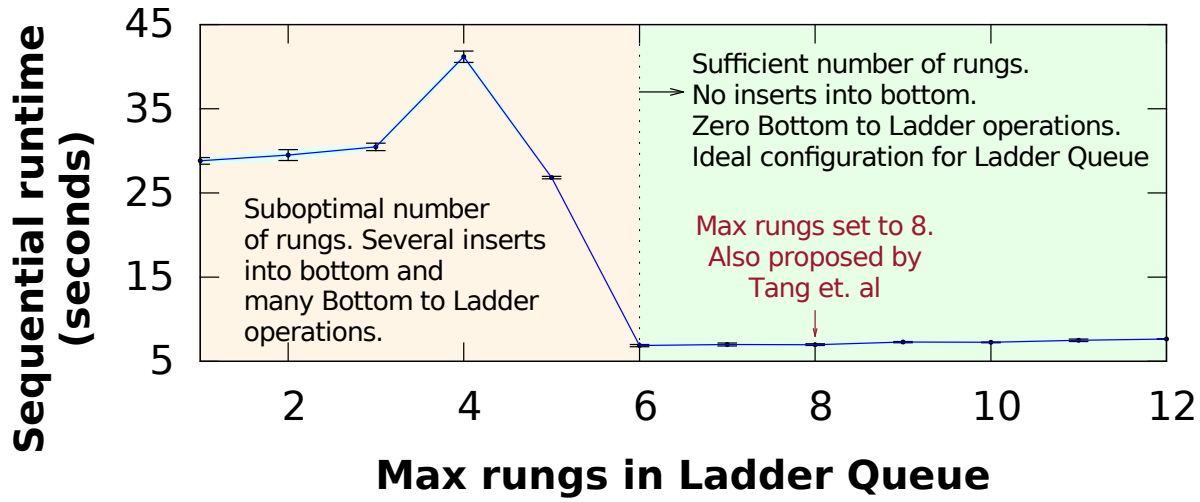



**Figure 4.7: Impact of limiting rungs in Ladder [17]**

The chart in Figure 4.7 shows that a minimum of 6 rungs is required. For some select configurations of larger models we observed (data not shown) that 5 rungs would be sufficient. However, the number of rungs cannot exceed beyond a threshold to avoid infinite spawning of rungs [13]. Moreover, it limits the overheads involved in re-bucketing events from rung-to-rung [13]. Accordingly, based on the observations in figure 4.7, we decided to adopt a maximum of 8 rungs, consistent with the threshold proposed by Tang et al [13]. Furthermore, we trigger *Bottom*-to-*Ladder* re-bucketing only if the *Bottom* has events at different timestamps to further reduce inefficiencies [17].

### 4.6.2 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 [10, 13, 14]. 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 4.1. This issue is exacerbated in large simulations where thousands of events are typically present in *Top* and buckets in various rungs [17].

In this context, it is important to recollect from that – as an optimization, **MUSE** utilizes only one anti-message from LP $_{\text{sender}}$ to LP $_{\text{dest}}$ to cancel all $n$ events sent after $t_{rollback}$ (rather than sending $n$ individual anti-messages) which reduces overheads. Furthermore, with our centralized scheduler design, only events received from LPs on other MPI-processes can trigger rollbacks. Consequently, the number of scans of the **ladderQ** that actually occur is significantly fewer in our case, despite the aggressive cancellation strategy [17].

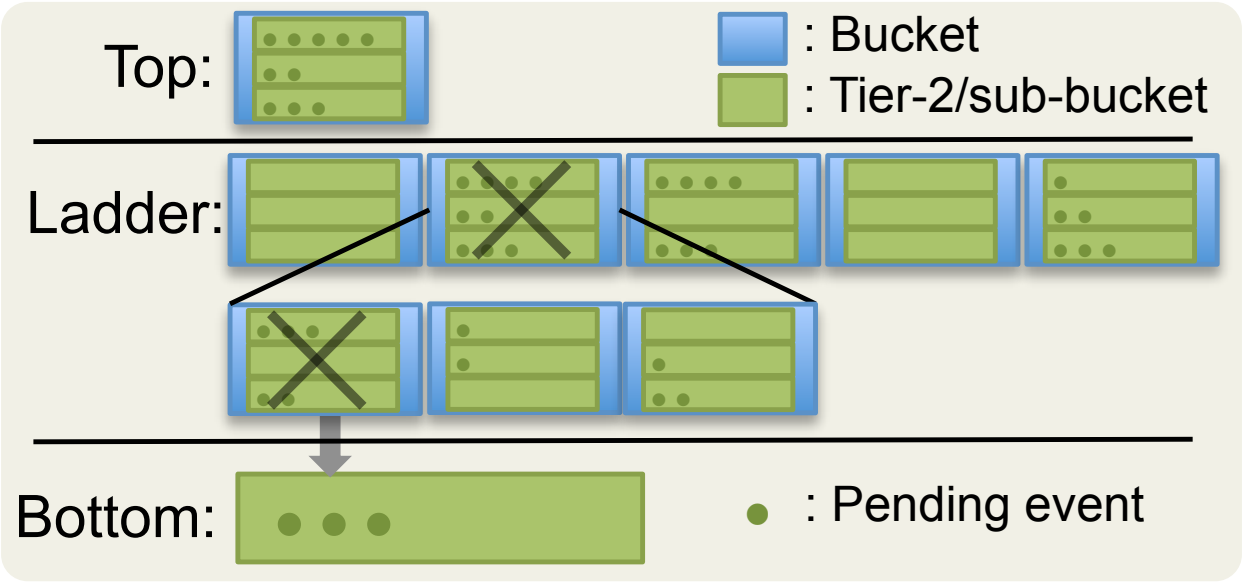



**Figure 4.8: Structure of 2-tier Ladder Queue (2tLadderQ) with 3 sub-buckets / bucket (*i.e.*, $_{t2}$ $k$=3) [17]**

## 4.7 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 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 [17].

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 $_{tk}$ sub-buckets, where $_{tk}$ is specified by the user. Figure 4.8 illustrates an overview of the **2tLadderQ** with $_{tk}$ = 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 $_{tk}$ 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 [17].

The dequeue operations for a bucket require iterating over each sub-bucket. However, for a small,

fixed value of ${}_{tk}$, the overhead becomes an amortized constant. The constant overhead is determined by the value of ${}_{tk}$. Consequently, dequeue also retains the amortized constant characteristic from regular **ladderQ** as summarized in Table 4.1. Currently, we do not subdivide *Bottom* but leave it as a possible future optimization [17].

### 4.7.1 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 $1 \div {}_{tk}$ 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 [17].

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 4.9 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 [17].

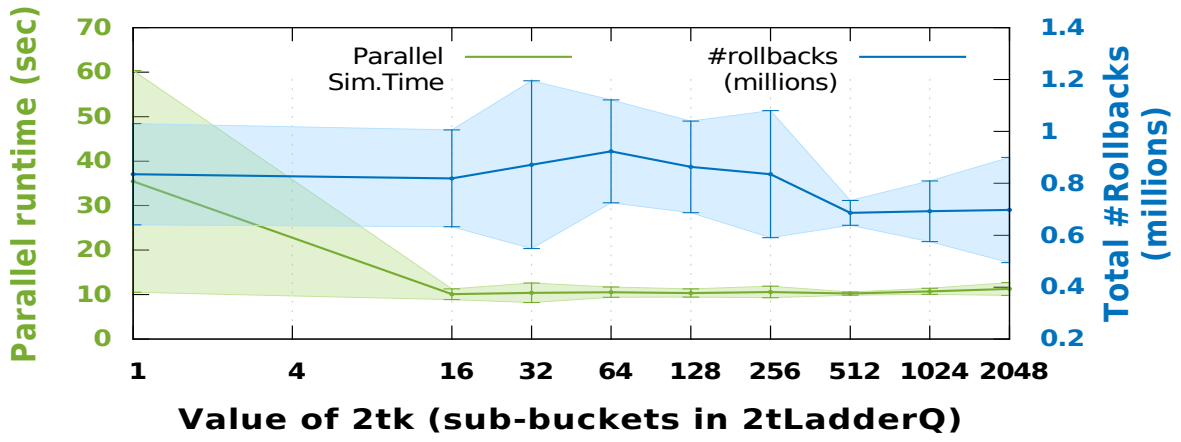



**Figure 4.9: Effect of varying tk [17]**

# Chapter 5

# Experiments

We conduct assessments of the effectiveness of the six scheduler queues using different configurations of the **PHOLD** benchmark and the **PCS** simulation. The experiments were conducted on our previously described distributed memory computing cluster. The **PHOLD** benchmark was compiled using (**-O3** optimization) and the **PCS** benchmark was compiled without the compiler optimization turned on. Our initial experimental analysis proved to be time consuming due to the large number of parameters (see Tables 3.1 and 3.2) and combinations of their values. Consequently, we pursued strategies to focus on the most influential parameters that impacted relative performance of the scheduler queues using Generalized Sensitivity Analysis (GSA) [17, 21].

## 5.1 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 [21]. 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 [17].

An important aspect of GSA is to ensure that the values for each parameter covers its full range of values. Consequently, we use Sobol random numbers to select a combination of **PHOLD** parameter values to be used for simulation. Sobol random numbers are quasi-random low-discrepancy sequences that provide uniform coverage of a multidimensional parameter space for **PHOLD**[22]. 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 [17].

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 [17].

### 5.1.1 GSA results for sequential simulations using PHOLD

The charts in Figure 5.1 show the cumulative $m$, $n$, and the $d_{m,n}$ statistics for the 9 different **PHOLD** 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 [17].

The chart in Figure 5.2 shows the summary of the $d_{m,n}$ statistic or influence of each parameter (see Table 3.1) on the outcome – *i.e.,* **2tLadderQ** performs better or worse than **3tHeap**. The lightly shaded bands show the 95% Confidence Intervals (CI) computed using standard bootstrap approach using 5000 replications with 1000 samples in each. As expected, the **imbalance** (*i.e.,* skew in partition) has no impact in sequential simulation and has a low impact score of 0.037. Similarly, the GVT computation rate does not impact pending events and consequently its influence

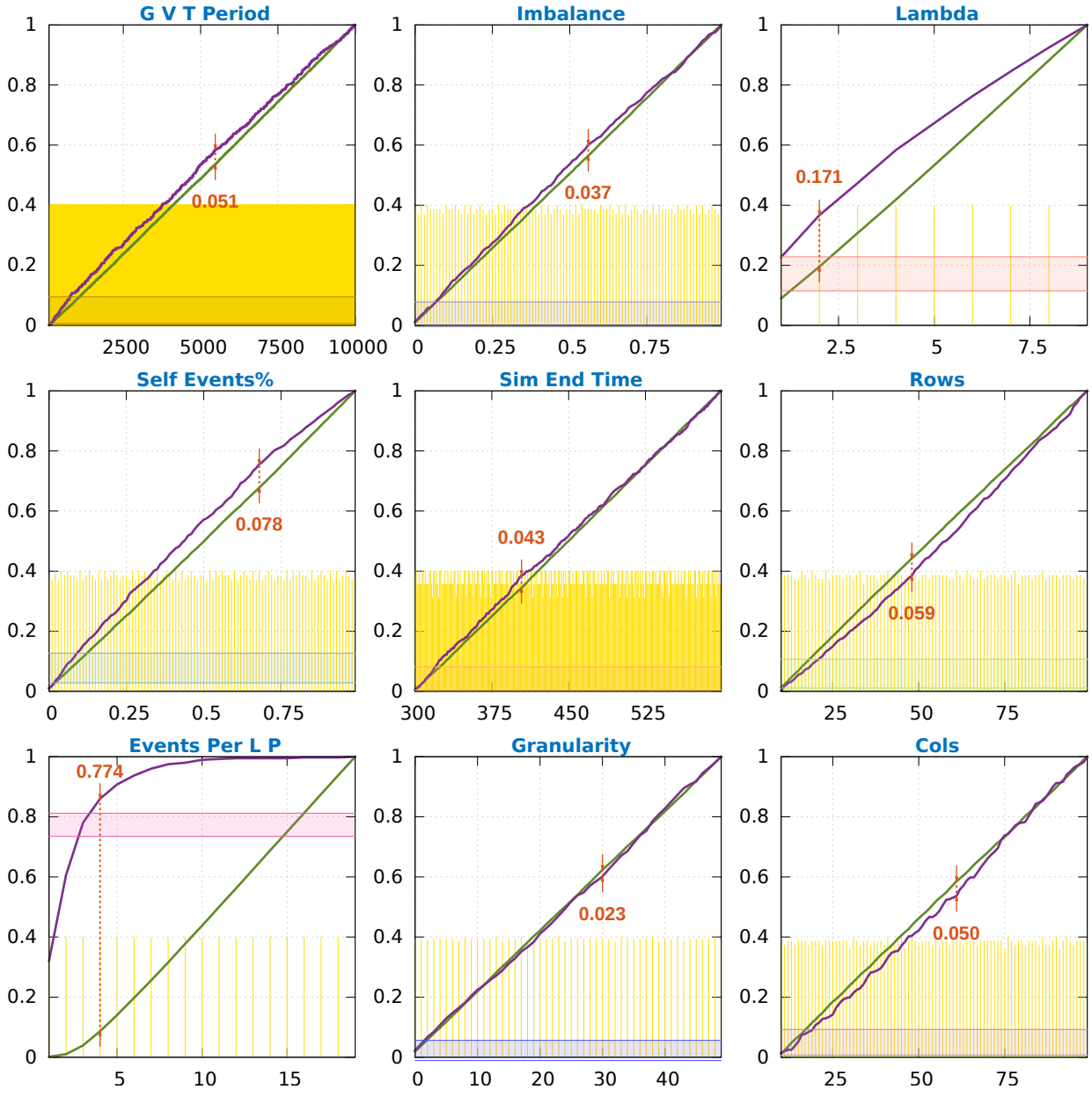



**Figure 5.1: Results from Generalized Sensitivity Analysis (GSA) comparing 2tLadderQ and 3tHeap for sequential simulation using the PHOLD benchmark [17].**

is low at 0.051 [17].

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 the 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 $\lambda$ for exponential distribution has a marginal influence because it influences number of concurrent events as discussed in Section 3.1 and shown in Figure 3.1(c) [17].

We have also conducted GSA to determine influential parameters impacting performance of other scheduler queues versus the **2tLadderQ** in sequential simulations. Our analysis shows that none of the parameters play an influential role. 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 ${}_{2t}k$=1) [17].

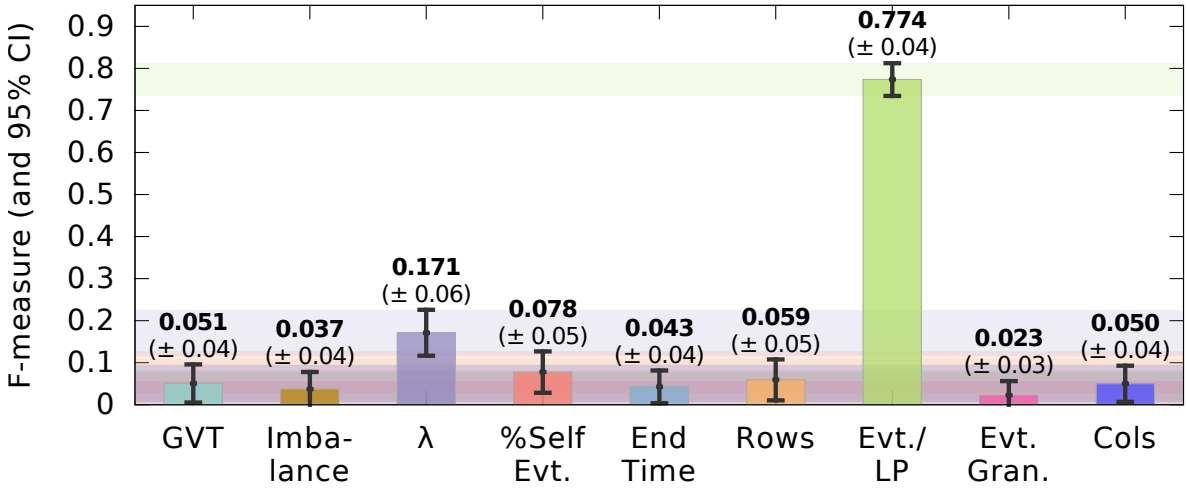



**Figure 5.2: Summary of influential parameters from Figure 5.1 that cause performance differences between 2tLadderQ and 3tHeap in sequential simulations using PHOLD [17].**

### 5.1.2 GSA results for sequential simulations using PCS

The charts in Figure 5.3 shows the two-sample KS-Test statistics for the cumulative probability distribution functions of the 10 different **PCS** parameters. The GSA data shows that in sequential simulations, most of the parameters do not influence performance difference between **2tLadderQ** and **3tHeap** with $d_{m,n} \ll 0.1$. As expected, **portables** was the most influential parameter in the simulation. This parameter is equivalent to the **eventsPerLP** parameter in the **PHOLD** benchmark. In similarity to the GSA results using **PHOLD**, higher values of **portables** parameter cause the **3tHeap** to perform better than the **2tLadderQ**. The summary of influential parameters in Figure 5.4 shows the **portables** parameter with a high impact score of 0.796. In addition, **rows** and **cols** parameters displayed marginal influence with low impact scores of 0.328 and 0.291, respectively.

GSA to determine influential **PCS** parameters impacting the performance of other scheduler queues against **2tLadderQ** showed that no parameter was more influential than **portables**. Furthermore, the results showed that **2tLadderQ** mostly outperformed the other scheduler queues with the exception of **2tHeap** and **3tHeap** in certain configurations.

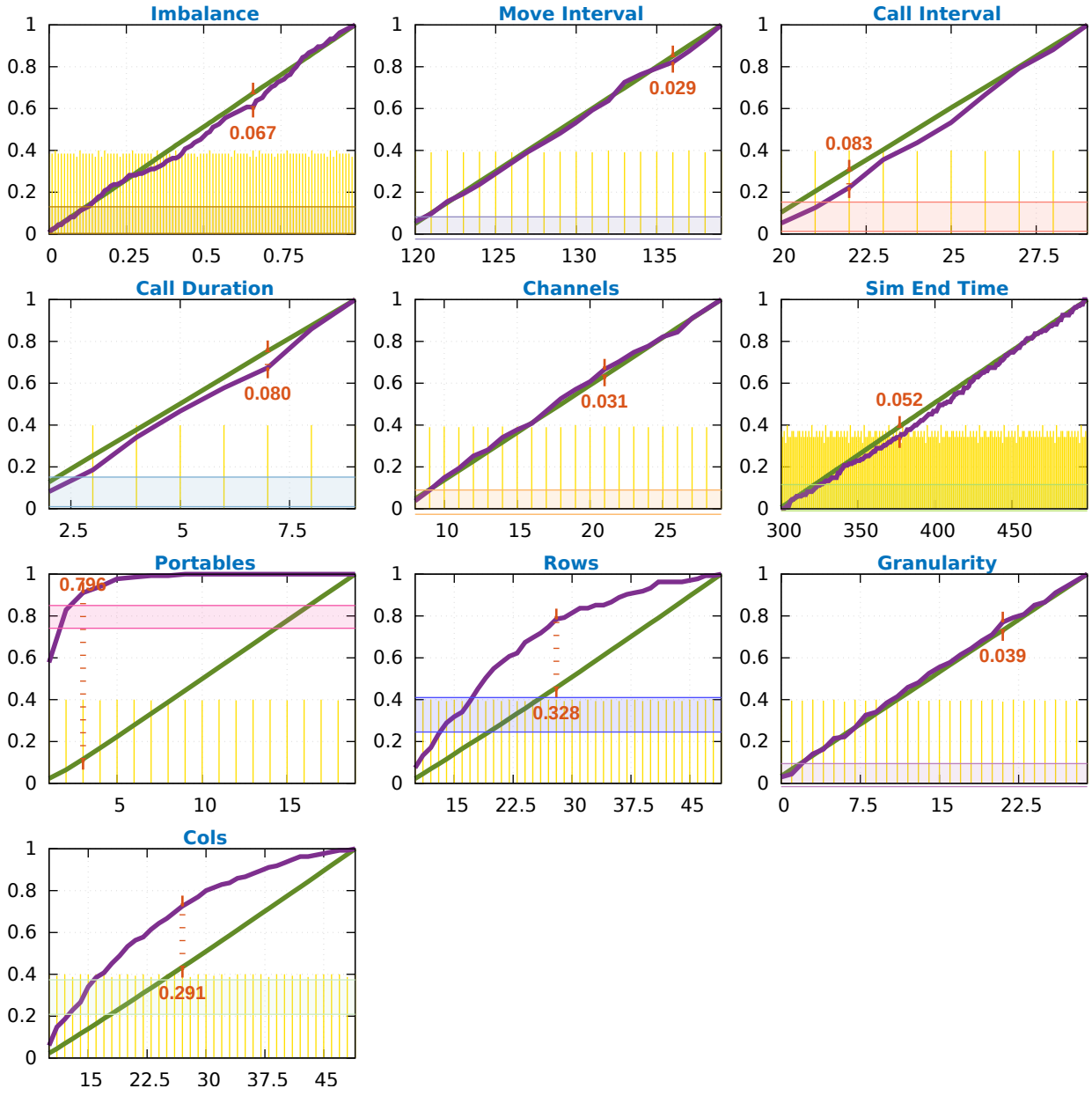



**Figure 5.3: Results from Generalized Sensitivity Analysis (GSA) comparing 2tLadderQ and 3tHeap for sequential simulation using PCS.**

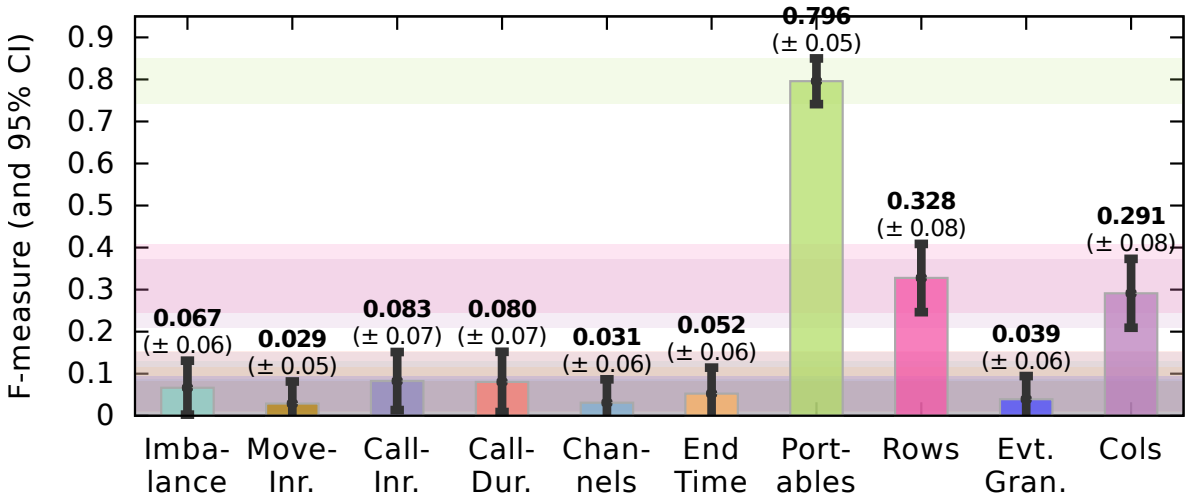



**Figure 5.4: Summary of influential parameters from Figure 5.3 that cause performance differences between 2tLadderQ and 3tHeap in sequential simulations using PCS.**

### 5.1.3 Summary of GSA results for sequential simulations

GSA shows that for comparing event queue performance in sequential simulations using our **PHOLD** benchmark and **PCS** simulation, we just need to focus on 1 or 2 parameters. Other aspects such as: event granularity, fraction of self-events, GVT rate, etc., do not matter for comparison of scheduler queues. We focus our continuing analysis on the following scheduler queues: **ladderQ**, **2tLadderQ**, and **3tHeap**.

### 5.1.4 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. For the parallel **PHOLD** simulation, 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 [17]. This consideration was not applied to the **PCS** simulation to remain consistent with the simulation properties as presented in [20].

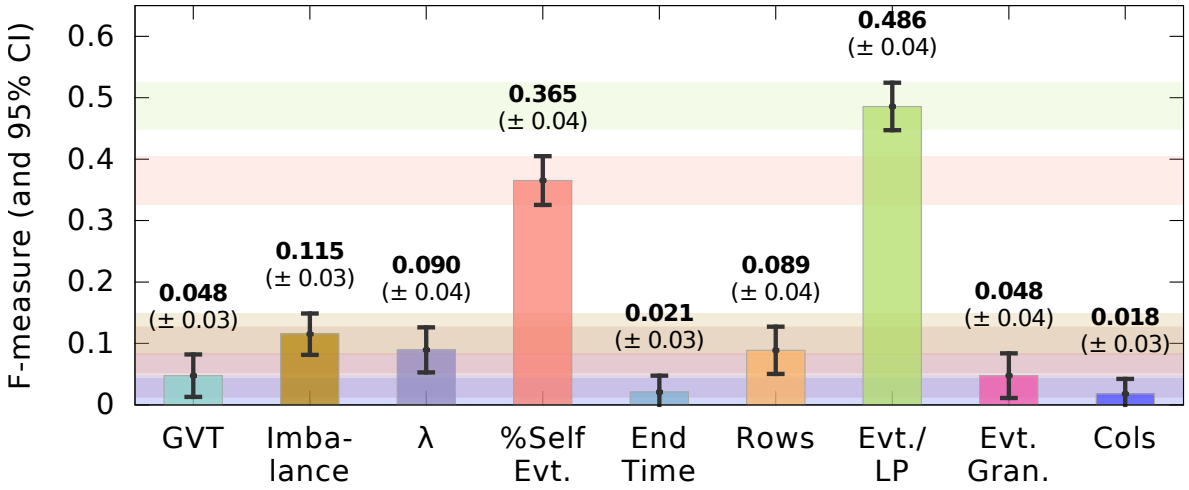



**Figure 5.5: GSA data from parallel simulations (4 MPI-processes) showing influential PHOLD parameters (2tLadderQ vs. 3tHeap) [17].**

The chart in Figure 5.5 shows the summary of the $d_{m,n}$ statistic or influence of each parameter (see Table 3.1) 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 $\lambda$. 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 [17].

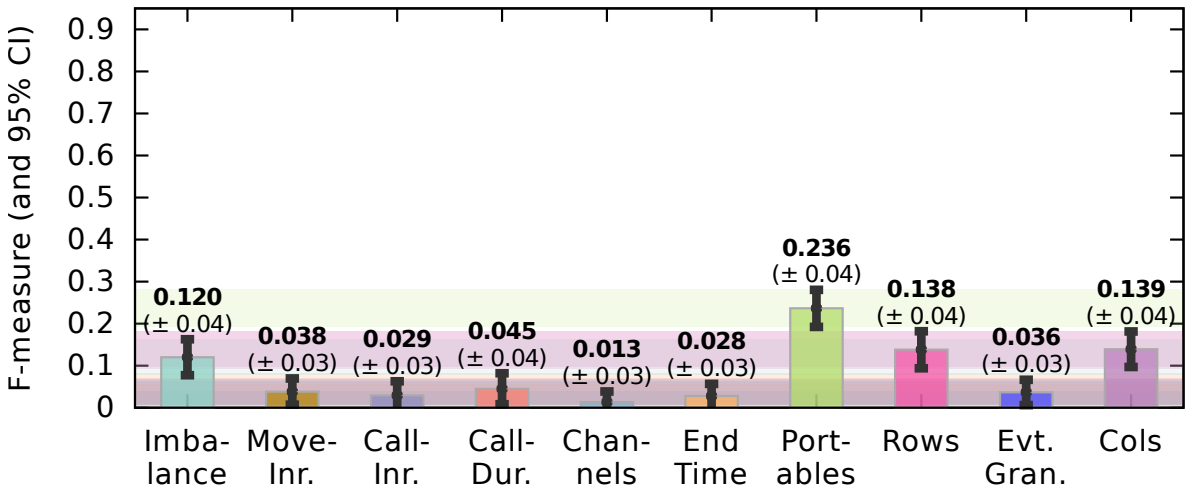



**Figure 5.6: GSA data from parallel simulations (4 MPI-processes) showing influential PCS parameters (2tLadderQ vs. 3tHeap).**

The chart in Figure 5.6 summarizes the $d_{m,n}$ statistic for parameters of the parallel **PCS** simulation (see Table 3.2). **Portables** is the most influential parameter with a subdued score of 0.236. Similarly, the model size has a marginal influence in the parallel simulation with scores of 0.138 and 0.139 for the model rows and columns. The **imbalance** parameter has a slightly more pronounced effect in the parallel simulation but as previously mentioned, a well balanced scenario is presumed.

## 5.2 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 6 different configurations for the simulations. The **PHOLD** configurations are called **ph3**, **ph4**, and **ph5** and the **PCS** configurations are called **pcs6**, **pcs7**, and **pcs8**. The fixed characteristics for the 6 configurations is summarized in Table 5.1. We use larger simulation end times for parallel simulation to obtain sufficiently long runtimes using 32 cores. The value of influential parameters, namely: **eventsPerLP**, **%selfEvents**, and $\lambda$ for the

**PHOLD** configuration and **portables** for **PCS** is varied for comparing different settings, similar to the approach used by other investigators [10, 13, 17].

**Table 5.1: Configurations of PHOLD and PCS 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 |
| **pcs6** | 100 (10×10) | 5000 | 50000 |
| **pcs7** | 1,000 (100×10) | 1000 | 4500 |
| **pcs8** | 10,000 (100×100) | 100 | 200 |

## 5.3 Sequential Simulations

We conduct sequential simulations to assess the effectiveness of the different data structures. We pursued sequential simulations to compare the base case performance of the data structures, consistent with prior investigations [10, 13]. The sequential simulations also serve as a reference for potential use in conservatively synchronized PDES. The sequential experiments were conducted using the 6 configurations listed in table 5.1 on one compute node of our cluster described in Section 2.4. 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 the **PHOLD** experiments, the influential parameters **eventsPerLP**, $\lambda$, and **%selfEvents** were varied to explore their impact on relative performance of the data structures [17]. For the **PCS** experiments, the **portables** was the only influential parameter that was varied. Event **granularity** in both simulation models was set to zero resulting in a fine grained simulation. For each configuration, data from 10 independent replications were collected and analyzed [17].

### 5.3.1 PHOLD sequential simulation results

The charts in Figure 5.7(a)–(c) show the change in runtime characteristics as the most influential parameter **eventsPerLP** is varied, for $\lambda$=1 (widest range of timestamps) and **%selfEvents** = 0.25. This configuration was generally the best for **ladderQ**. As illustrated by Figure 5.7(a)–(c), the performance of **ladderQ** and **2tLadderQ** (${}_{2t}k$=1) is comparable, as expected. However, the **2tLadderQ** performs slightly (paired $t$-test p-value $< 0.05$, *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** [17].

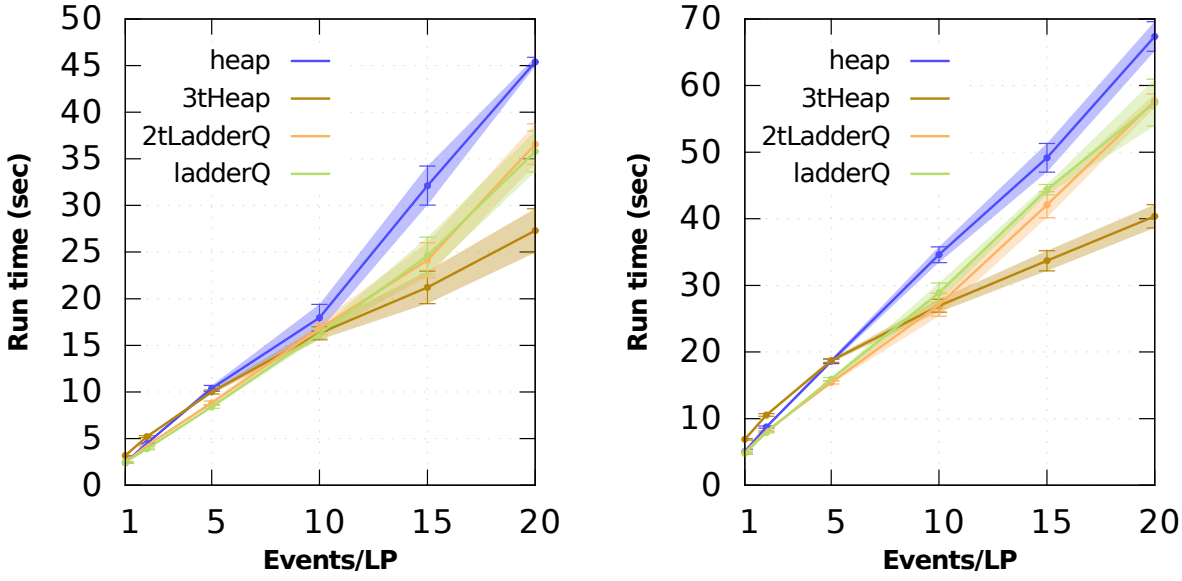
However, the **3tHeap** generally outperforms the other queues (except for **2tHeap** in some cases) for higher values of **eventsPerLP**. In all cases, there were no inserts into *Bottom* or *Bottom*-to-*Ladder* operations (discussed in Section 4.6.1) that degrade **ladderQ** performance. The size of 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 time stamp 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 4.6.1 and the same value proposed by Tang et al [13, 17].

Profiler data showed that the bottleneck in **ladderQ** arises from the overhead of re-bucketing events from rung-to-rung of the Ladder. On the other hand, in **3tHeap** re-bucketing does not occur. Consequently, the overheads of O(log $\frac{e}{c}$) operations in **3tHeap** are amortized as number of concurrent events $c$ increases [17].

The chart in Figure 5.7(d) shows the correlation between the 3 influential parameters and the performance difference between **3tHeap** and **ladderQ**. Consistent with the GSA results, the correlogram shows that the most influential parameter is **eventsPerLP** (R=0.93, p=0) followed by $\lambda$ (R=0.19, p=0.192) with a very weak correlation. The **%selfEvents** has practically no impact on performance. The correlogram also shows that these parameters are independent and have no covariance between each other (R ~ 0, p >0.95) [17].

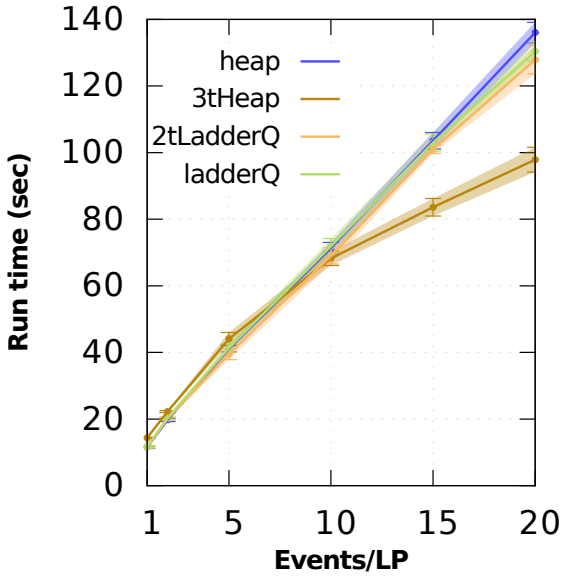
The charts in Figure 5.8 shows the peak memory usage corresponding to the runtime data in Figure 5.7. The memory size reported is the "Maximum resident set size" value reported by GNU **/usr/bin/time** command on Linux. The memory usage of **heap** is the lowest in most cases. Since $_{2t}k$=1, the memory 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 [17].



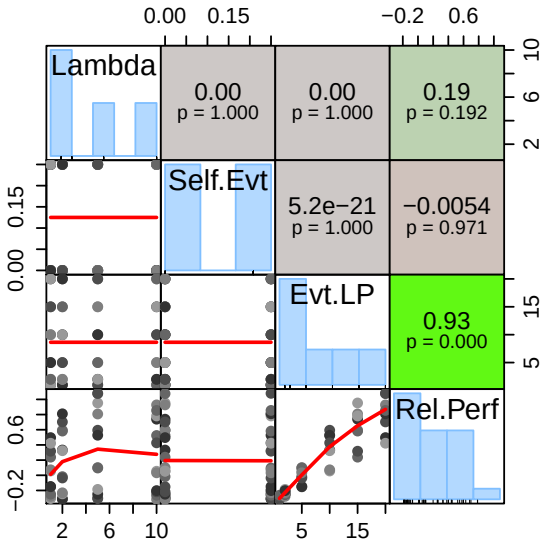

(a) **ph3**




(b) **ph4**




(c) **ph5**




(d) Corellogram

**Figure 5.7: Sequential simulation runtimes and correlation of 3tHeap performance with PHOLD parameters [17]**

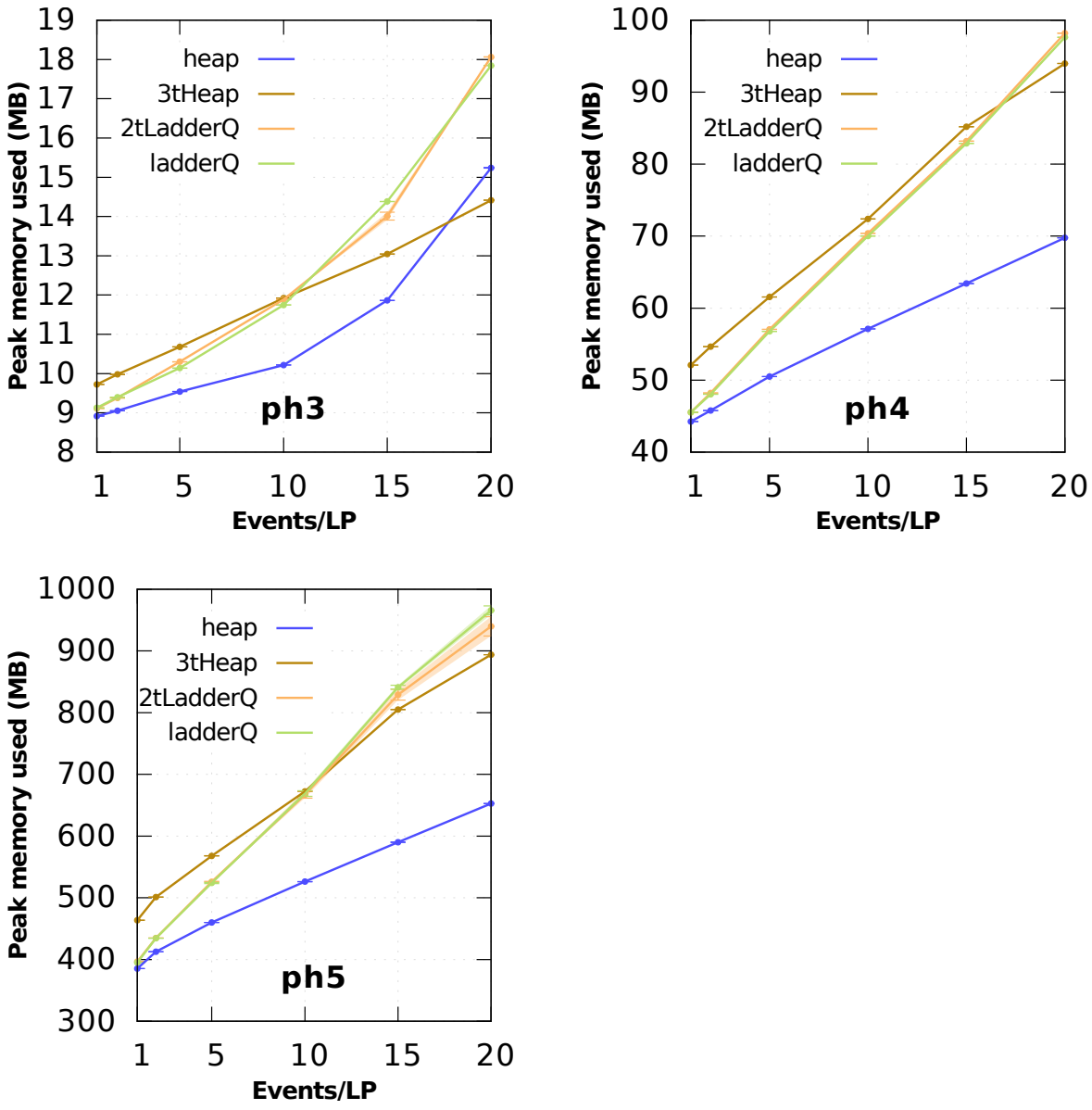



**Figure 5.8: Comparison of peak memory usage [17]**

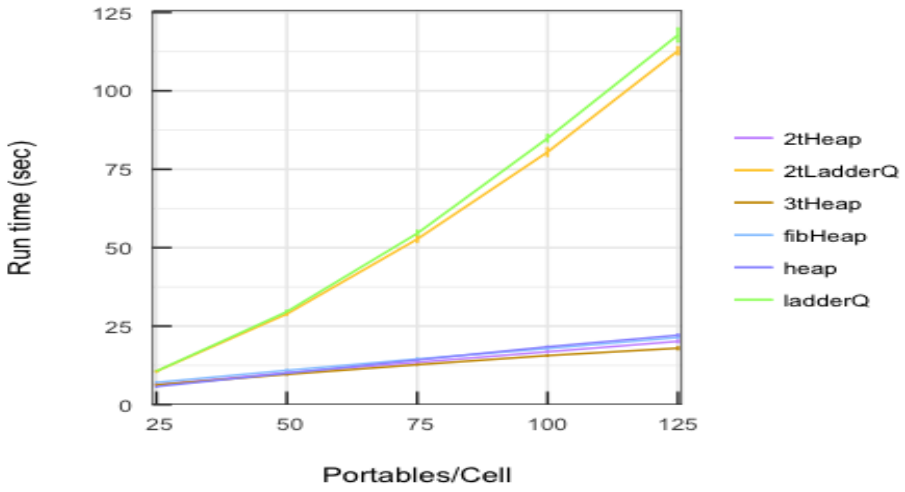
### 5.3.2 PCS sequential simulation results

The charts in Figure5.9(a)-(c) show runtime characteristics as the number of **portables** is varied using our 3 PCS configurations. As illustrated in the charts, the performance of **ladderQ** and **2tLadderQ** ($_{t2}k = 1$) is comparable. However, these two data structures were largely outperformed by the other scheduler queues. In fact, mean runtimes of **ladderQ** and **2tLadderQ** displayed an order of magnitude that was exceedingly high in comparison with the remaining queues. For the **pcs8** configuration with 125 **portables per cell**, the mean runtime values of the **ladderQ** and **2tLadderQ** were approximately 500× higher than the runtime means of the other queues. Similarly to sequential simulation using the **PHOLD** benchmark, profile data from the **PCS** configurations showed that overhead of re-bucketing portables from rung-to-rung in both **ladderQ** and **2tLadderQ** degraded runtime performance. However, the impact of the overhead was even more pronounced in the **PCS** simulations.

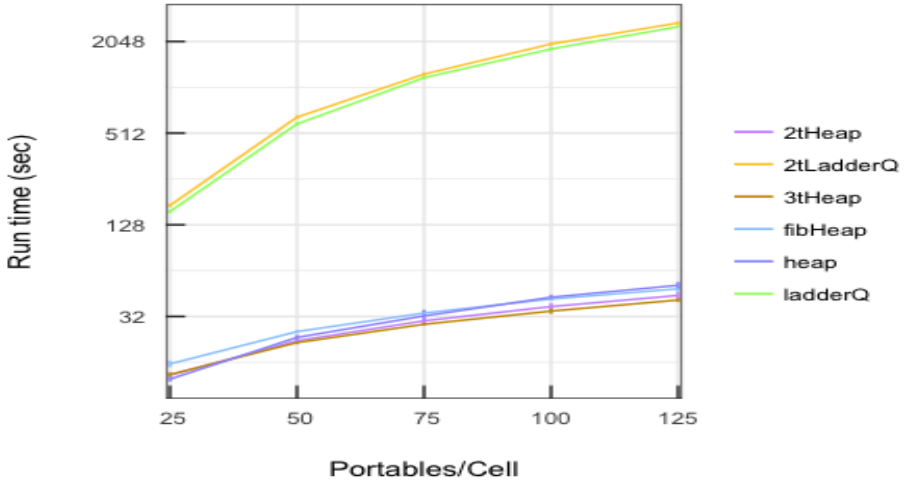
The charts in Figure5.10 (a)-(c) provide an illustration of the runtime performance of the scheduler queues without **ladderQ** and **2tLadderQ**. For the most part, **3tHeap** outperformed all of the other scheduler queues for higher number of **portables per cell**. As previously noted the re-bucketing operation is not a property of **3tHeap** and its design supports more efficient handling of an increasing number of concurrent events.

The charts in Figure 5.11 show peak memory usage associated with runtime data displayed in Figures 5.9 and 5.10. The peak memory usage is the lowest and tends to decrease with increasing number of **portables per cell** for **3tHeap** in configurations **pcs6** and **pcs7**. On the other hand **fibHeap** has the lowest memory usage in the **pcs8** configuration.

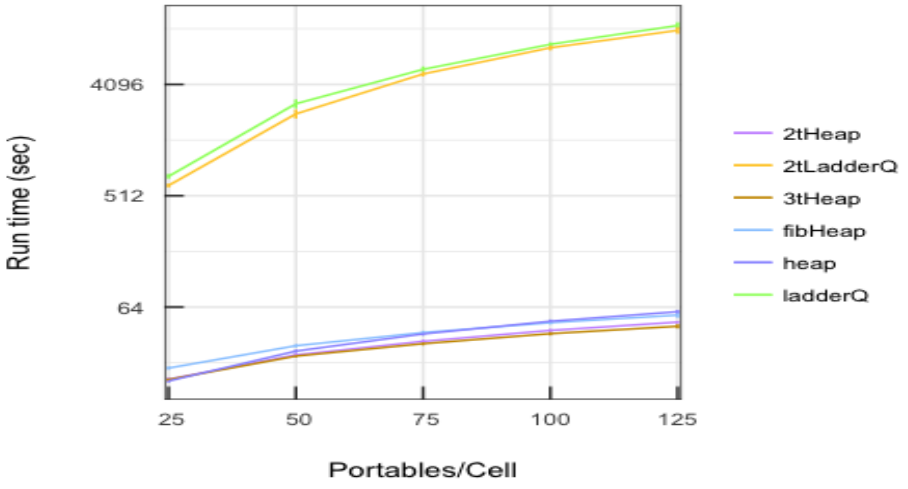
The charts in Figure 5.12(a)-(c) show the profile results for the 3 configurations. As illustrated in the charts **ladderQ** and **2tLadderQ** displayed more effective use of the CPU data cache in all configurations, which is critical to performance. However, the percentage of instructions was also the highest for the two scheduler queues. This is consistent with the larger runtimes observed for **ladderQ** and **2tLadderQ**. On the other hand, **3tHeap** displayed a comparably higher percentage of cache and branch misses with respect to most of the scheduler queues, however, it also showed the lowest percentage of instructions in each configuration scenario.




(a) **pcs6**

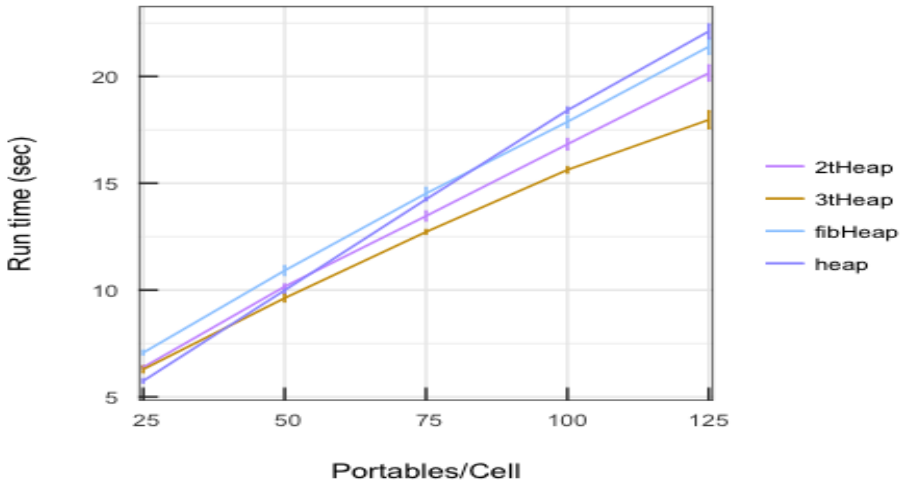



(b) **pcs7**

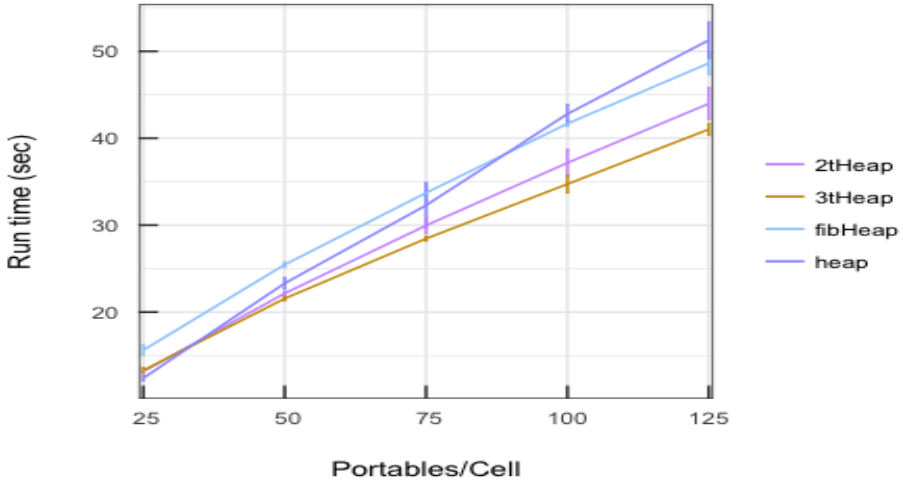



(c) **pcs8**

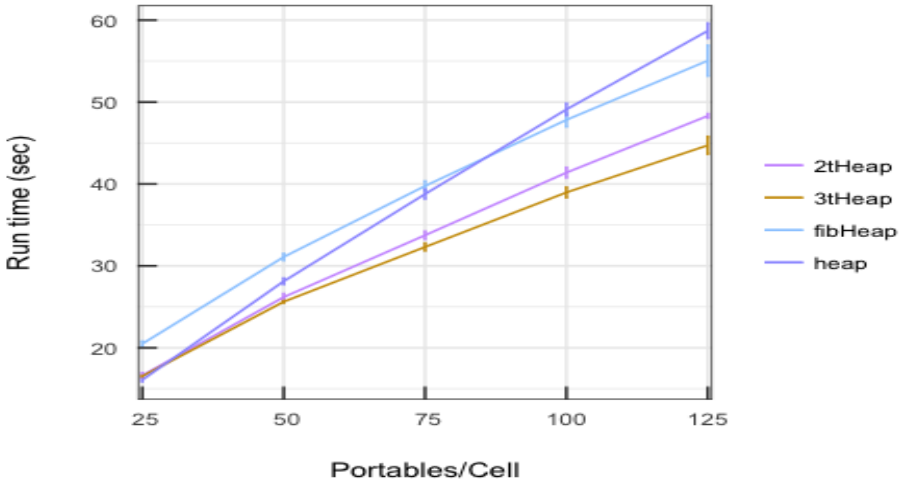
**Figure 5.9: Sequential simulation runtimes with PCS parameters**




(a) **pcs6**




(b) **pcs7**




(c) **pcs8**

**Figure 5.10: Sequential simulation runtimes with PCS parameters**

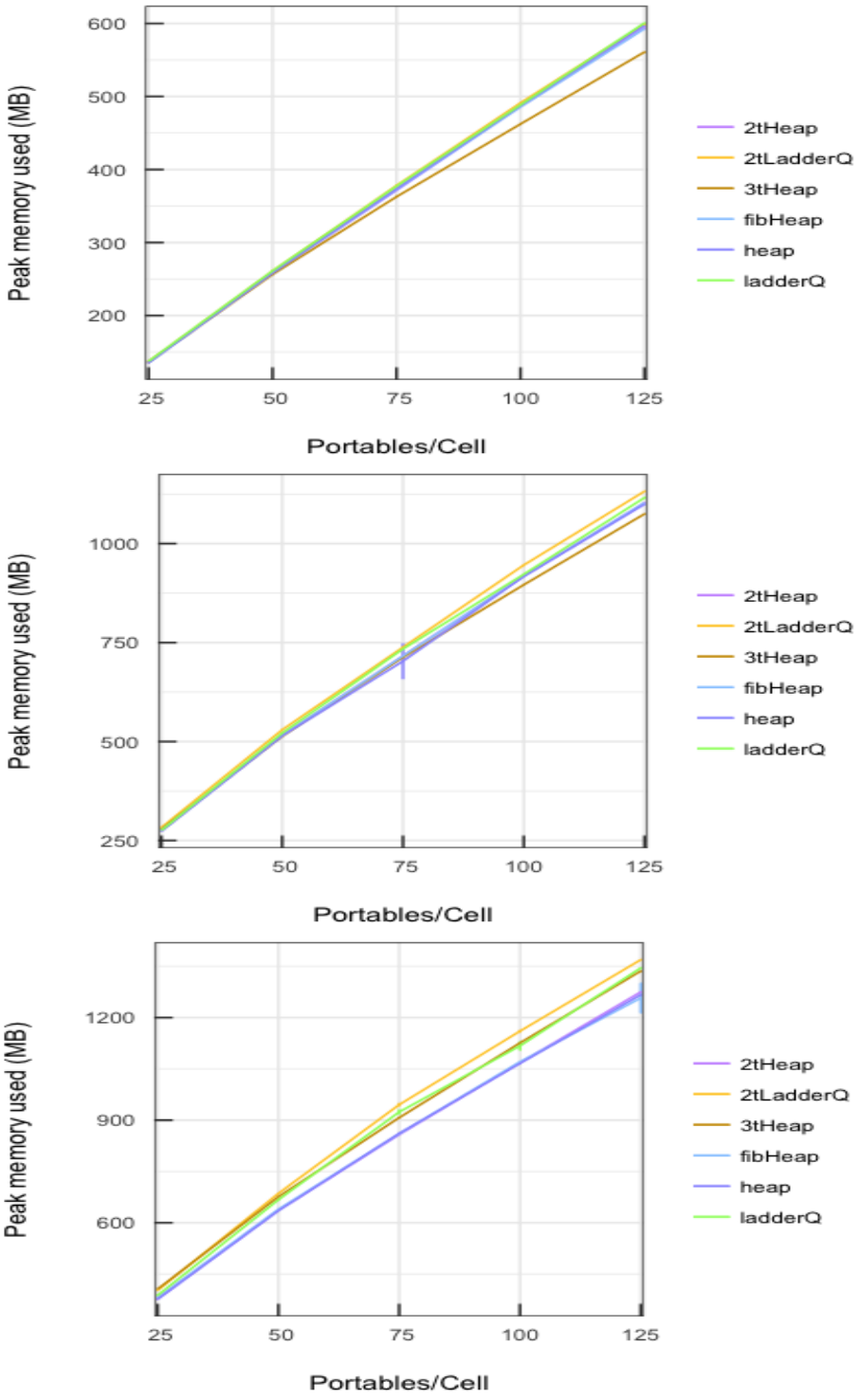
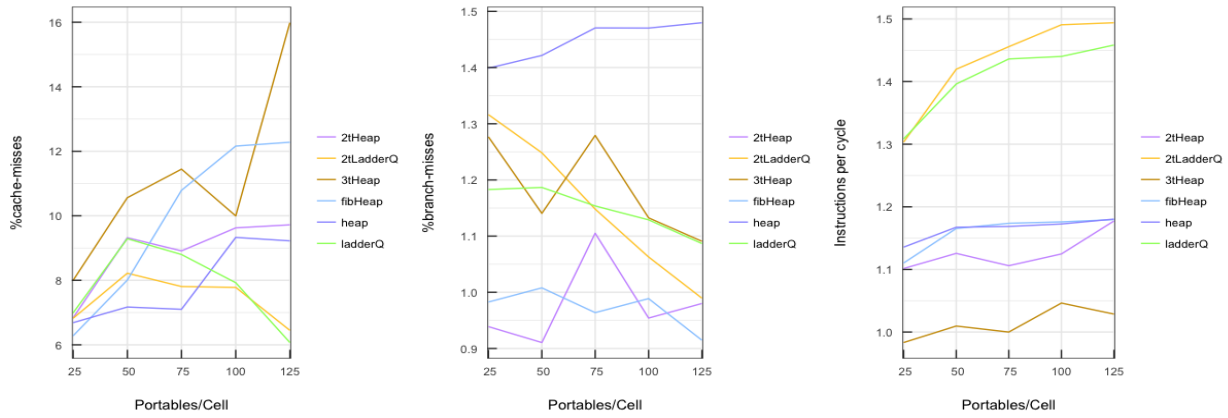
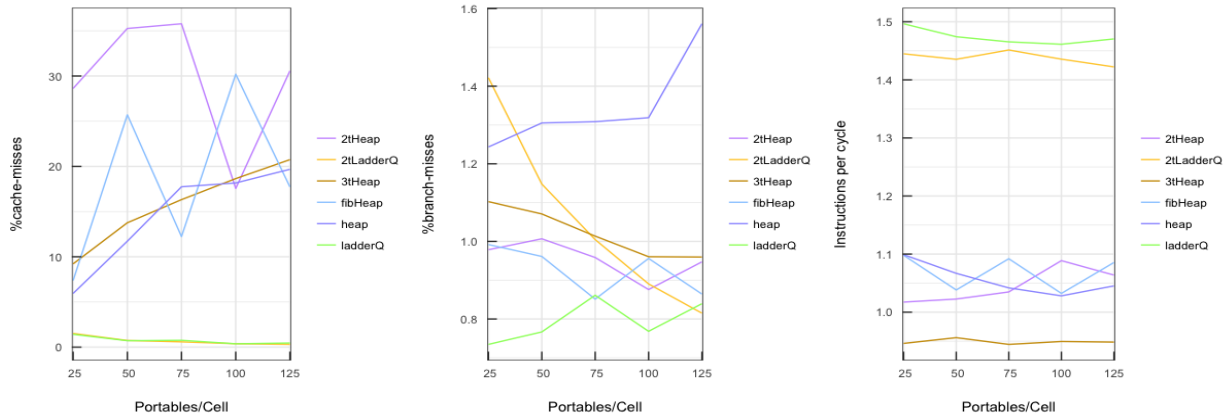


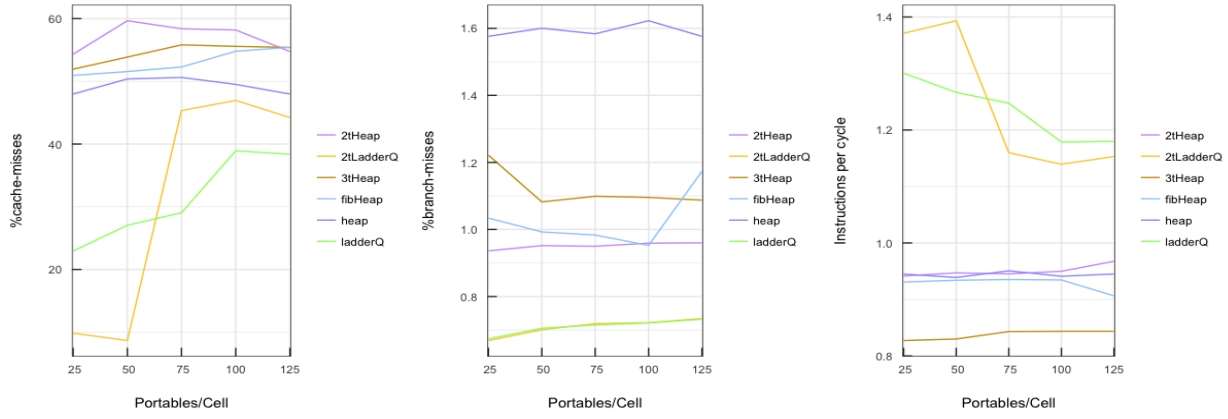

**Figure 5.11: Comparison of peak memory usage**




(a) **pcs6**




(b) **pcs7**




(c) **pcs8**

**Figure 5.12: Profile results from PCS sequential simulation**

## 5.4 Parallel simulation assessments

The sequential simulation assessments indicated that **ladderQ**, **2tLadderQ**, and **3tHeap** performed the best for a broad range of **PHOLD** parameter settings. In particular, **3tHeap** performed the best for the **PCS** experiments. 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 2.4) 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 5.1. The following subsections discuss results from the experiments [17].

### 5.4.1 Throttling optimism with a time-window

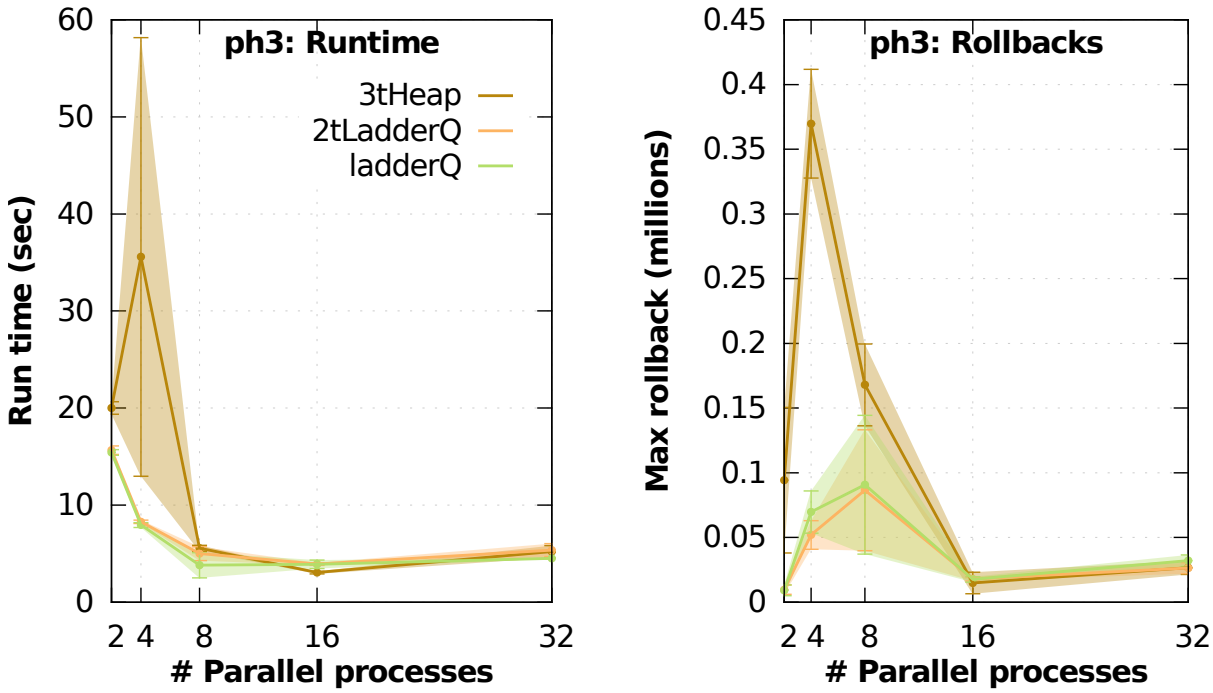
Initially we conducted experiments with a fine-grained setting (*i.e.,* **granularity** = 0) from sequential simulations. We noticed that the **ladderQ** had a large variance in runtimes, particularly when it experienced many rollbacks. In several cases, cascading rollbacks significantly slowed the simulations – *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 the 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 [17].

### 5.4.2 Efficient case for ladderQ

The charts in Figure 5.13, Figure 5.14, and Figure 5.15 show key simulation statistics for low value of **eventsPerLP** = 2 and $\lambda$=1 for which **ladderQ** performed well, consistent with the observations in sequential simulations. The statistics show average and 95% CI computed from 10 independent

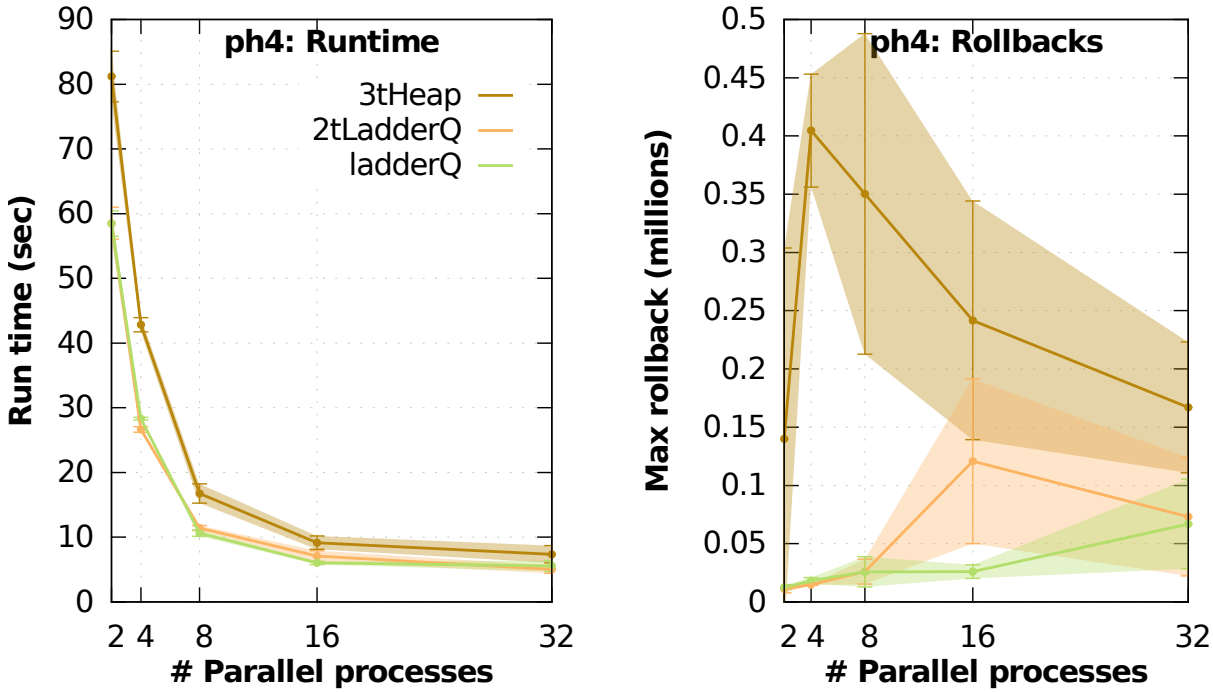
replications for each data point. The peak rollbacks among all of the MPI-processes is shown as it controls overall progress in the parallel simulations. As illustrated by the charts, both the **ladderQ** and **2tLadderQ** perform well for all three models. In this configuration, overall the **ladderQ** experienced the fewest rollbacks. Nevertheless, the **2tLadderQ** continues to perform well despite experiencing more rollbacks as shown in Figure 5.14. 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 [17].

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 5.15, **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 [17].



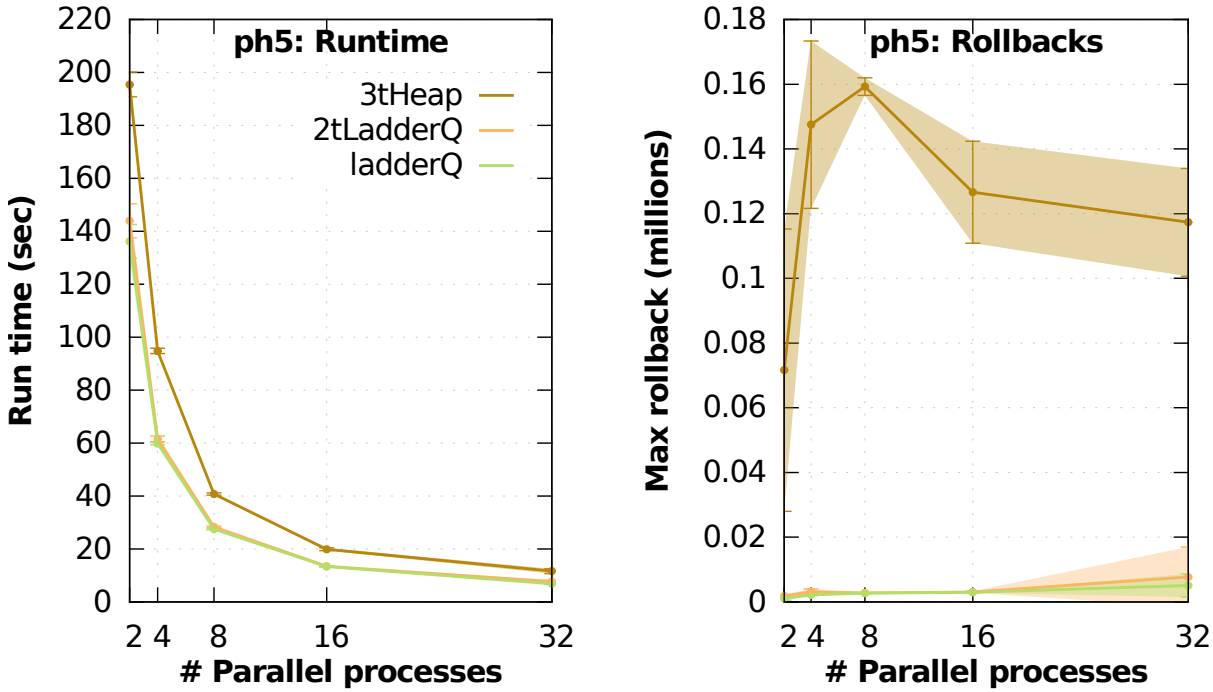

(a) **ph3** time & rollbacks

**Figure 5.13: Statistics from PH3 configuration of PHOLD parallel simulation with eventsPerLP=2, $\lambda = 1$, %selfEvents=25% [17]**




(b) **ph4** time & rollbacks

**Figure 5.14: Statistics from PH4 configuration of PHOLD parallel simulation with eventsPerLP=2, $\lambda = 1$, %selfEvents=25% [17]**

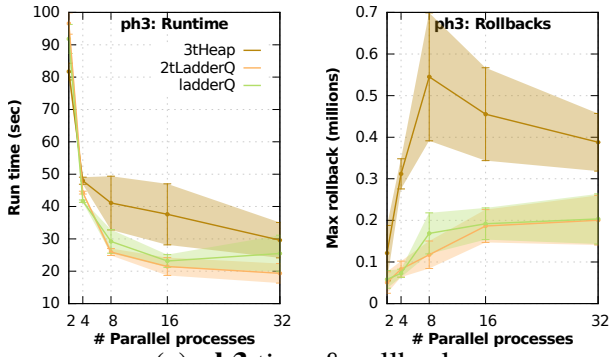



(c) **ph5** time & rollbacks

**Figure 5.15: Statistics from PH5 configuration of PHOLD parallel simulation with eventsPerLP=2, $\lambda = 1$, %selfEvents=25% [17]**

### 5.4.3 Knee point for 3tHeap vs. ladderQ

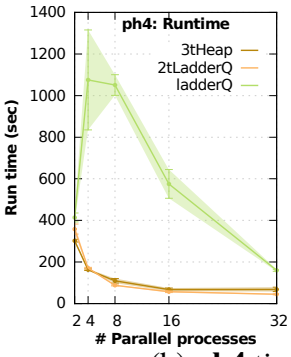
The charts in Figure 5.16 show key simulation statistics for the configuration where **3tHeap** and **ladderQ** performed about the same in sequential (see Figure 5.7). For **ph3**, both **ladderQ** and **2tLadderQ** experienced comparable number of rollbacks but the **2tLadderQ** performs better due to its design advantages. In the case of **ph4** and **ph5**, both the **ladderQ** and **3tHeap** experienced a comparable number of rollbacks. However, the observed number of rollbacks for **ladderQ** and **3tHeap** was much higher than the number of rollbacks for **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 [17].

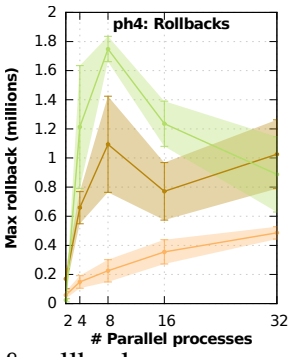




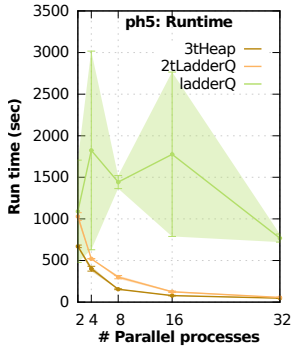


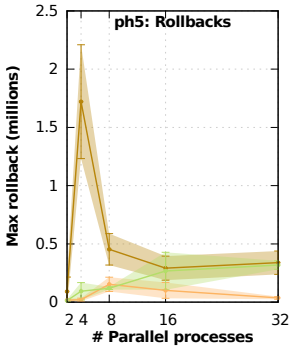
(a) **ph3** time & rollbacks




(b) **ph4** time & rollbacks




(c) **ph5** time & rollbacks

**Figure 5.16: Statistics from PHOLD parallel simulation with eventsPerLP=10, $\lambda = 10$, %selfEvents=25% [17]**

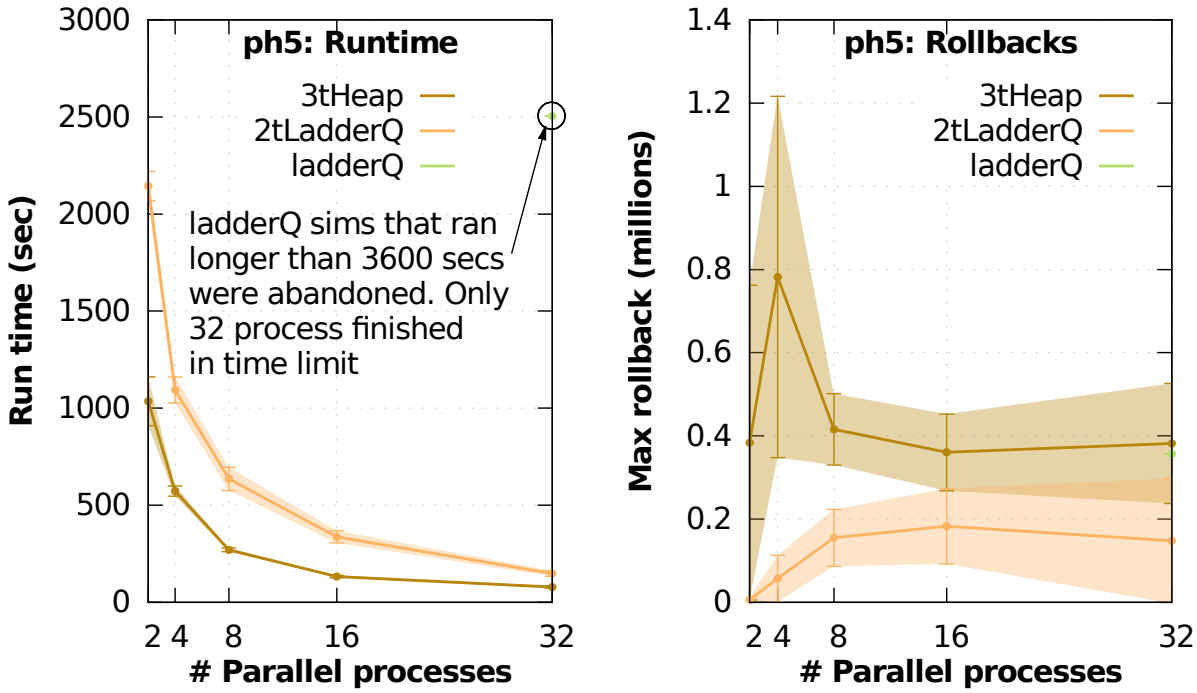
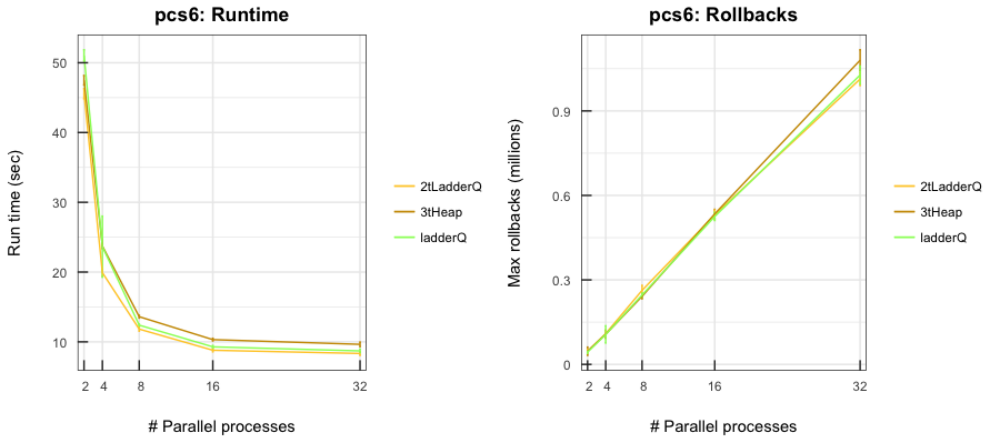
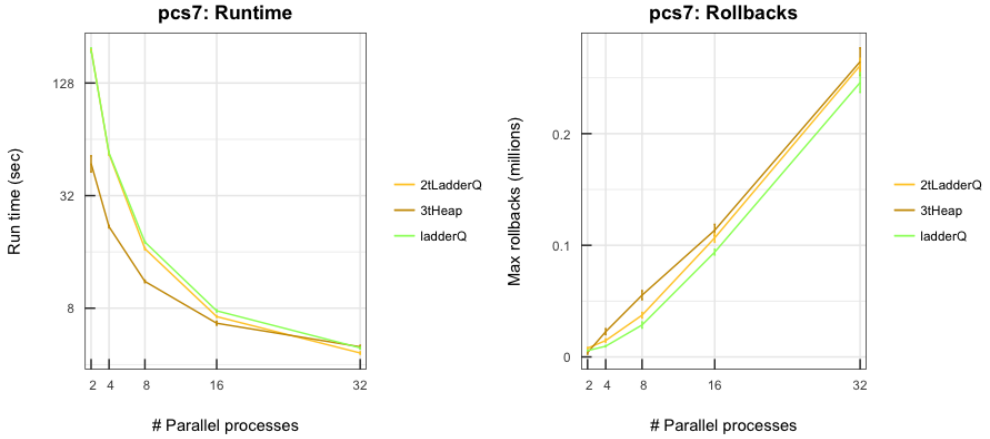


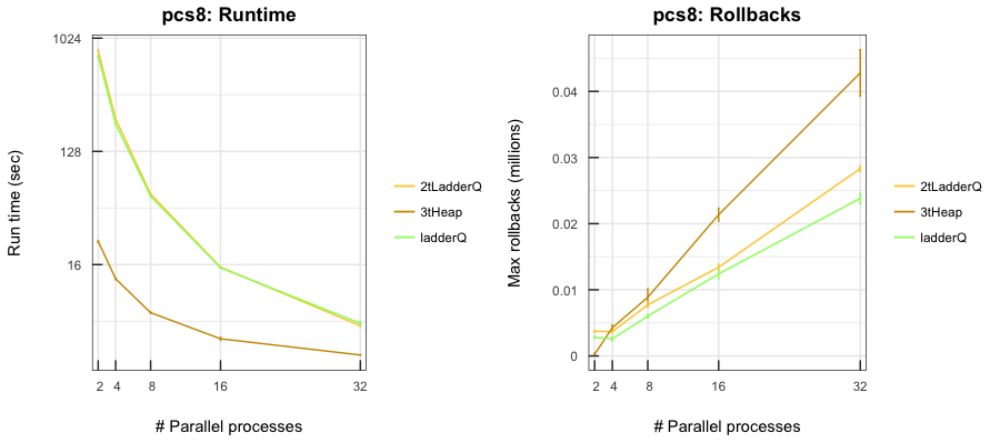

**Figure 5.17: ph5 Statistics (best case for 3tHeap) [17]**




(a) **pcs6** time & rollbacks

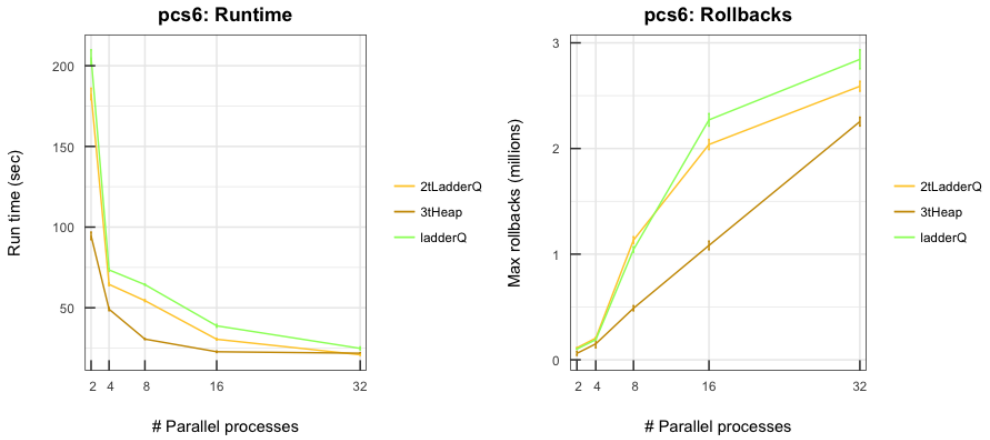



(b) **pcs7** time & rollbacks

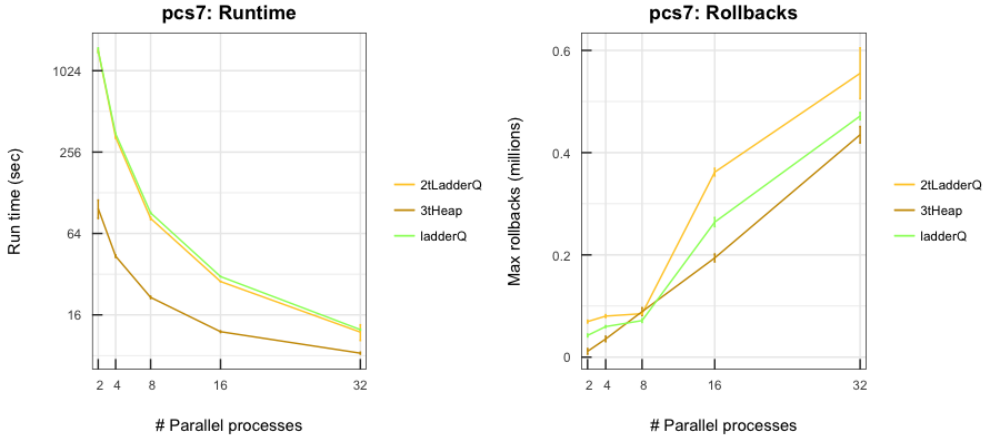



(c) **pcs8** time & rollbacks

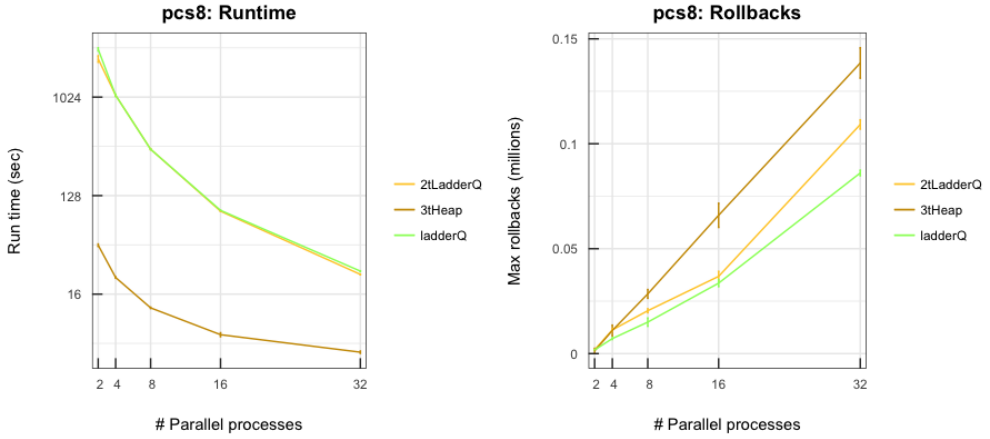
**Figure 5.18: Statistics from PCS parallel simulation with portables=25**




(a) **pcs6** time & rollbacks




(b) **pcs7** time & rollbacks




(c) **pcs8** time & rollbacks

**Figure 5.19: Statistics from PCS parallel simulation with portables=75**

### 5.4.4 Best case for 3tHeap

Figure 5.17 shows simulation time and rollback characteristics in high concurrency configuration with **ph5**, with **eventsPerAgent**=20, $\lambda$=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 [17].

Figures 5.18 and 5.19 show parallel simulation runtime and rollback characteristics with **pcs6**, **pcs7** and **pcs8** for # of **portables** set equal to 25 and 75, respectively. Generally, **3tHeap** outperformed the other scheduler queues for all configurations. The performance benefits of **3tHeap** was more noticeable in the **pcs8** configuration while undergoing a greater number of rollbacks, as illustrated in Figures 5.18(c) and 5.19(c). In the **pcs8** configuration with 75 **portables per cell** and 32 processors, **3tHeap** experienced an average of 203624 rollbacks, while **2tLadderQ** experienced 137800 and **ladderQ** experienced 90540 rollbacks.

# Chapter 6

# Conclusions

Efficient data structures, *i.e.,* priority queues for managing pending event sets play a critical role in the overall performance of both sequential and parallel simulations. In the context of this study, we broadly classified the queues into single-tiered (**heap** and **ladderQ**) or multi-tiered (**2tHeap**, **fibHeap**, **3tHeap**, and **2tLadderQ**) data structures based on their design. Multi-tier data structures organize pending events into tiers, with each tier possibly implemented using different structures. Organizing events into multiple tiers decouples event management and Logical Process (LP) scheduling permitting different algorithms and data structures to suit the different needs [17].

The comparative analysis used a fine-tuned version of the Ladder Queue (**ladderQ**) proposed by Tang et al. [13]. Our objective in fine-tuning was to reduce the runtime constants of the **ladderQ** without significantly impacting its amortized O(1) time complexity. We realize the reduction in runtime constants by minimizing memory management overheads – *i.e.,* ❶ favor few bulk operations via **std::vector** than many small ones via **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. Accordingly, other simulation kernels can significantly improve overall performance by replacing linked-lists with dynamically growing arrays [17].

One challenge that arose during design of experiments was exploring the large multidimensional parameter space in the **PHOLD** synthetic benchmark and **PCS** simulation model. Since large parameter spaces may arise with actual simulation models. We propose the use of Generalized Sensitivity Analysis (GSA) to reduce the parameter space. We 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. However, it was able to significantly narrow the parameter space, *i.e.,* from 9 down to 1. 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 [17].

The sequential and parallel simulation results showed that **2tLadderQ** performs no worse than **ladderQ** in sequential simulations (with ${}_{t2}k$=1). Furthermore, the **2tLadderQ** performs better 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 it was impractical to use in scenarios with many cascading rollbacks. These experiments were conducted with fine-grained settings (*i.e.,* **granularity**=0) and may vary with granularity. However, GSA data suggests that the variation with changing granularity would be small, but may allowing 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 [17].

The experiments show that multi-tier queues also incur additional overhead as part of the two step process. Moreover, 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. Consequently, in sequential simulations, their advantages were realized in simulations that have higher concurrency (*i.e.,* larger batches of events) per LP. 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 [12] and detailed simulation models such as packet-level

network simulations [13]. The simulations results using **PCS** showed that **3tHeap** is the desired scheduler queue for optimal runtime performance in sequential and optimistic parallel simulations. However, further experimental analysis with such models is necessary to validate effectiveness of the data structures [17].

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 favor the use of **2tLadderQ** and **3tHeap**, as they are consistently effective in sequential and parallel simulations, with sequential results also bearing potential application to conservative and multithreaded simulations [17].

### Future Work

The results in this thesis provide us with an understanding of the effectiveness of multi-tier data structures for managing pending events in sequential & optimistic parallel simulations. However, the thesis can be extended and our findings further strengthened through additional testing and analysis. An area worthy of further exploration is to determine whether or not our research findings are attributed to the characteristics and design of our multi-tier data structures or influenced by the framework that supported the testing.

As such, we propose the implementation of our multi-tier data structure (i.e. **ladderQ**, **2tLadderQ** and **3tHeap** in a different programming language such as Java and comparing performance. Furthermore, the data structures should be assessed using different parallel simulation frameworks and experimental platforms. Franceschini et.al., evaluated different implementation of the pending event using their Ruby based discrete event simulator [10]. While, Luca Toscano et al., designed and implemented a Time Warp based parallel simulation framework developed in Erlang [23]. The approaches and frameworks presented in these studies could be applied to the evaluation of our multi-tier data structures. Lastly, we constrained our simulation models to the benchmark **PHOLD** and the **PCS** model. It would be useful to assess the performance of our data structures using a wider range of simulation models to examine the durability of our results

# Bibliography


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

[2] G. Fishman, *Discrete-event simulation: modeling, programming, and analysis*. Springer Science & Business Media, 2013.

[3] R. M. Fujimoto, "Parallel discrete event simulation," *Communications of the ACM*, vol. 33, no. 10, pp. 30–53, 1990.

[4] R. R. Hill, J. O. Miller, and G. A. McIntyre, "Simulation analysis: applications of discrete event simulation modeling to military problems," in *Proceedings of the 33nd conference on Winter simulation*, pp. 780–788, IEEE Computer Society, 2001.

[5] P. Pacheco, *An introduction to parallel programming*. Elsevier, 2011.

[6] R. Bryant, O. David Richard, and O. David Richard, *Computer systems: a programmer's perspective*, vol. 2. Prentice Hall Upper Saddle River, 2003.

[7] D. W. Jones, "An empirical comparison of priority-queue and event-set implementations," *Communications of the ACM*, vol. 29, no. 4, pp. 300–311, 1986.

[8] R. Rönngren and R. Ayani, "A comparative study of parallel and sequential priority queue algorithms," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 7, no. 2, pp. 157–209, 1997.

[9] R. Brown, "Calendar queues: a fast 0 (1) priority queue implementation for the simulation event set problem," *Communications of the ACM*, vol. 31, no. 10, pp. 1220–1227, 1988.

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

[11] C. D. Carothers and K. S. Perumalla, "On deciding between conservative and optimistic approaches on massively parallel platforms," in *Proceedings of the Winter Simulation Conference*, WSC '10, pp. 678–687, Winter Simulation Conference, 2010.

[12] J.-S. Yeom, A. Bhatele, K. Bisset, E. Bohm, A. Gupta, L. V. Kale, M. Marathe, D. S. Nikolopoulos, M. Schulz, and L. Wesolowski, "Overcoming the scalability challenges of epidemic simulations on blue waters," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pp. 755–764, IEEE, 2014.

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

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

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

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

(Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2016.

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

[18] M. R. Gebre, "Muse: A parallel agent-based simulation environment," 2009.

[19] P. A. Wilsey, "Some properties of events executed in discrete-event simulation models," in *Proceedings of the 2016 annual ACM Conference on SIGSIM Principles of Advanced Discrete Simulation*, pp. 165–176, ACM, 2016.

[20] C. D. Carothers, R. M. Fujimoto, Y.-B. Lin, and P. England, "Distributed simulation of large-scale pcs networks," in *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 1994., MASCOTS'94., Proceedings of the Second International Workshop on*, pp. 2–6, IEEE, 1994.

[21] B. Guven and A. Howard, "Identifying the critical parameters of a cyanobacterial growth and movement model by using generalised sensitivity analysis," *Ecological Modelling*, vol. 207, no. 1, pp. 11 – 21, 2007.

[22] G. Levy, "An introduction to quasi-random numbers," *Numerical Algorithms Group Ltd., http://www. nag. co. uk/IndustryArticles/introduction_to_quasi_random_numbers. pdf (last accessed in April 10, 2012)*, p. 143, 2002.

[23] L. Toscano, G. D'Angelo, and M. Marzolla, "Parallel discrete event simulation with erlang," in *Proceedings of the 1st ACM SIGPLAN workshop on Functional high-performance computing*, pp. 83–92, ACM, 2012.