ABSTRACT

MANAGING PENDING EVENTS IN SEQUENTIAL & OPTIMISTIC PARALLEL DISCRETE EVENT SIMULATIONS

by Julius Didier Higiro

[Insert abstract here]

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. Rao M. Dhananjai

Reader: Dr. Matthew Stephan

Reader: Dr. Jianhui Yue

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. Rao M. Dhananjai

Dr. Matthew Stephan

Dr. Jianhui Yue

Contents

List of Tables

List of Figures

List of Listings

Dedication

[Insert dedication here...]

Acknowledgements

Thank you to my adviser Dr. Rao M. Dhananjai 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. Janhui Yue for their support and input.

Chapter 1

Introduction

Discrete event simulation (DES) is the simulation of a discrete event system that contains states that operate at discrete time steps [1]. 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 [2]. DES has been used in a variety of fields in academia, industry and the public sector as a tool to help inform our knowledge of discrete event systems and to improve decision-making processes [1]. 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 [3]. 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.

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). However, the speedup achieved using multi-core and multi-processor systems requires efficient parallel programs.

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 [4]. Events that are yet to be processed are called "pending events". Pending events must be processed by LPs in priority order to maintain causality, with event priorities being determined by their timestamps. Consequently, data structures for managing and prioritizing pending events play a critical role in ensuring efficient sequential and parallel simulations [5–8]. The effectiveness of data structures for event management is a conspicuous issue in larger simulations, where thousands or millions of events can be pending [9, 10]. 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 PDES called, Time Warp can further impact the effectiveness of the data structure due to additional processing required during rollback-based recovery operations.

Thesis Statement

Many investigations have explored the effectiveness of a wide variety of data structures for managing the pending event set. Among the various data structures, the Ladder Queue proposed by Tang, [11] has shown to be the most effective data structure for managing pending events [8, 12], particularly in sequential DES. Our 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 a data structure that outperforms all other data structures in managing pending events. This task is accomplished by comparing the effectiveness of previous and newly defined data structures against our fine-tuned version of the Ladder Queue [11] because it has shown to be very efficient for sequential DES.

Thesis: multi-tiered data structures, especially the novel, **2tLadderQ** and **3tHeap** pending event structures outperform all other data structures in sequential and optimistically parallel simulations.

Related Work

Recently, Franceschini [8] 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 [11] and Franceschini [8] both use the classic Hold benchmark simulation model used in this research.

In contrast to earlier 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.

Dickman [12] 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 [13] 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 [14] 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.

In contrast to aforementioned efforts, 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.

Chapter 2

Miami University Simulation Environment (MUSE)

The implementation and assessment of the different data structures was conducted using our parallel simulation framework called 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 [15]. 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 parallel simulation is shown in Figure 2.1. The simulation kernel implements core functionality associated with LP registration, event processing, state saving, synchronization and Global Virtual Time (GVT) garbage based collection. Each LP in a simulation maintains an input, output and state queue. The input queue is used to handle pending events that have yet to be processed. 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

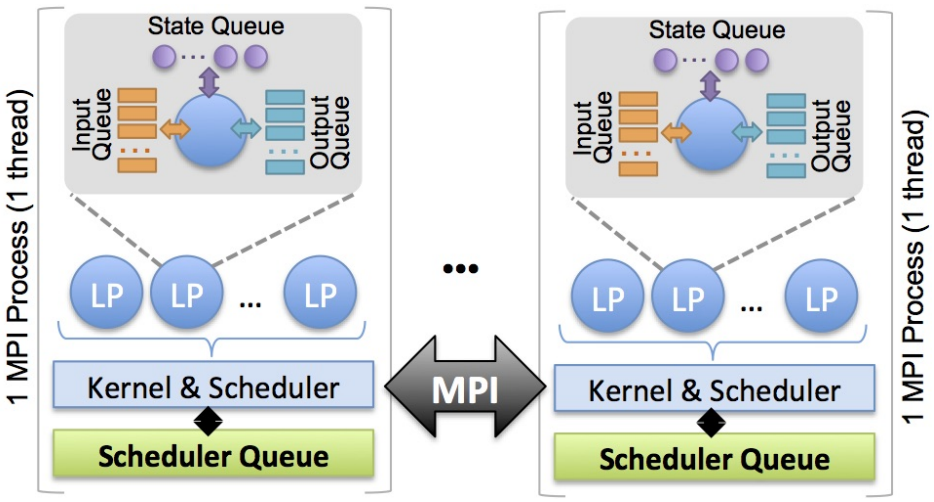


Figure 2.1: Overview of a parallel MUSE simulation

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 [4]. 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) [4].

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 2.1, 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:

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

❷ **Peek next event**: This operation 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.

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.

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 LP_{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.

2.1 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. 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.

❷ **Speedup**: This is a measure of the performance improvement from serial to parallel program execution. It is measured as the ratio of serial and parallel run times. This was used to compare the realized speedups of our programs using different scheduler queues.

❸ **Efficiency**: This is a measure of the processing time dedicated to essential program execution activities such as computations at the expense of parallel programming related overhead tasks.

❹ **Total Parallel Overhead**: The aggregated overhead value associated with a parallel program.

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.

2.2 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

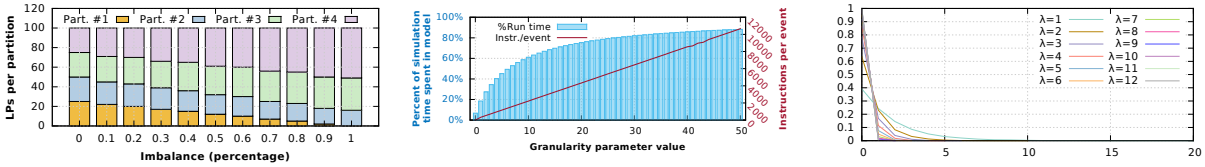
to support a shared file system. The nodes run Red Hat Enterprise Linux 6, with Linux (kernel ver 2.6.32) and the cluster runs PBS/Torque. The simulation software was compiled using GCC version 4.9.2 (**-O3** optimization) with OpenMPI 1.6.4. All debug assertions were turned off for maximum performance.

Chapter 3

Simulation Models

3.1 Parallel HOLD (PHOLD)

The experimental analysis have been conducted using a parallelized version of the classic Hold synthetic benchmark called PHOLD. It has been used by many investigators because it has shown to effectively emulate the steady-state phase of a typical simulation [8, 11]. 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 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.



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

Figure 3.1: Impact of varying key parameter values in the PHOLD model

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

Table 3.1: Parameters in PHOLD benchmark

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

the **granularity** parameter (no unit) is summarized in Figure 2(b) – smaller values result in finer grained simulations. For each event, an LP schedules another event to a randomly chosen adjacent LP. The **selfEvents** parameter controls the fraction of events that an LP schedules to itself. The event timestamps are determined by a given **delay-distrib** and **delay** or λ parameters. Our experiments use an exponential distribution for timestamps, because it has shown to reflect event distribution commonly found in a broad range of simulation models [11]. Time stamp of events is computed as t_{recv} = LVT + 1 + $\lambda e^{-\lambda x}$. The impact of changing the λ (*i.e.,* **delay**) is shown in Figure 3.1(c) –smaller values of λ provide a broader range of time stamp value for future events resulting in fewer concurrent events per LVT. Conversely, larger λ values cause timestamps to be close to the current epoch, increasing both the number of concurrent events per LVT and the possibility of rollbacks. The impact of these parameters on scheduler queue performance were explored using 2,500 different configurations.

3.2 Personal Communication Service Network (PCS)

The experimental analysis was also performed using a network communication simulation model named PCS. The implementation uses parameters summarized in Tables 3.1 and 3.2. The PCS model consists of cells (i.e. cell towers) that transmit and receive phone calls made by mobile cellular phone units that reside at each cell. The cells contain a fixed number of channels that are licensed to individual phone units known as portables. 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.

Table 3.2: Parameters in PCS Model

Parameter	Description
moveIntervalMean	The average move time of a portable from one PCS cell to another PCS cell.
callIntervalMean	The average time before the next call.
callDurationMean	The average completion time of a call.
#channels	The maximum number of channels assigned to each PCS cell.

The PCS simulation commences with a fixed number of events/portables for each LP/cell, specified by the **eventsPerLP** parameter. The portables contain three exponentially distributed timestamps with means specified by **moveIntervalMean**, **callIntervalMean** and **callDuration-Mean** parameters. The minimum of the timestamps is used to determine the time at which the next phone call arrives at a cell, the time a portable completes a phone call, or the time a portable departs its current cell for another cell. The PCS simulation will be used to validate resulting experimental analysis using the synthetic benchmark PHOLD.

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.

Table 4.1: Comparison of algorithmic time complexities of different data structures

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

Name	Enqueue	Dequeue	Cancel
heap	$\log(e \cdot l)$	$\log(e \cdot l)$	$z \cdot \log(e \cdot l)$
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. A **std::vector** is used as the backing container 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. 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.

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.

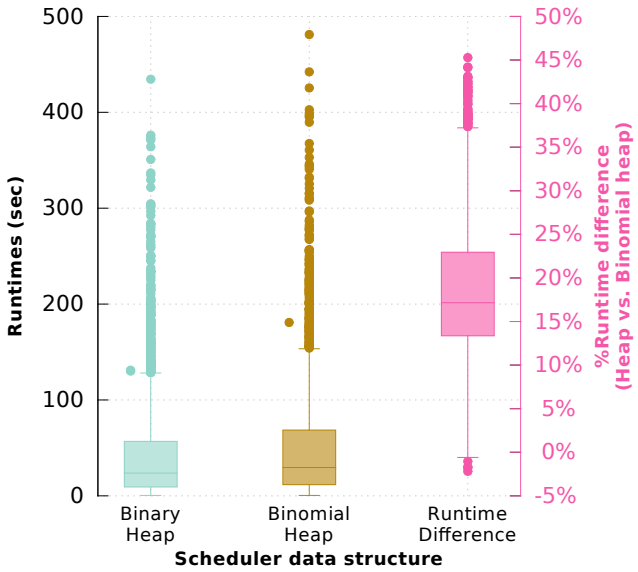
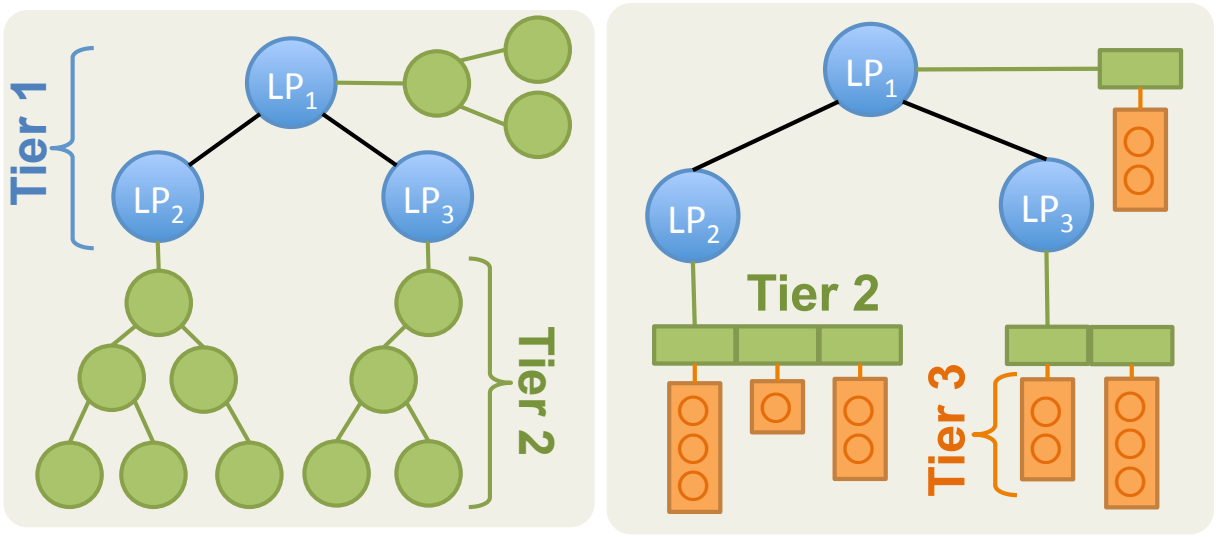


Figure 4.1: Comparison of heap and binomHeap execution time

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 4.1, **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.



(a) 2-tier Heap

(b) 3-tier Heap

Figure 4.2: Structure of 2-tier & 3-tier heap

4.3 Two-tier Heap (2tHeap)

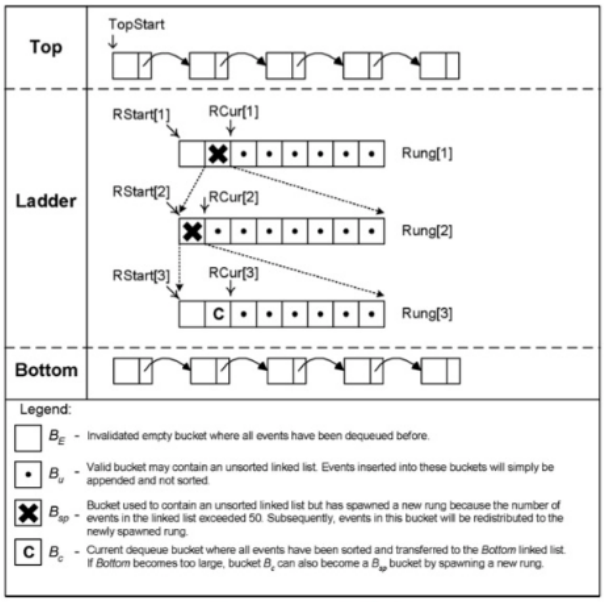
The **2tHeap** is designed to reduce the time complexity of cancel operations by subdividing events into two distinct tiers as shown in Figure 4.2. The first tier has containers for each local LP on an MPI-process. Each of the tier-1 containers contain a heap of events to be processed by a given LP. In **2tHeap** both tiers are maintained as independent binary heaps. Consequently, given l LPs and e pending events per LP, enqueue and dequeue operates require $\log e$ time to insert in tier-2 followed by $\log l$ time to reschedule the LP. Note that the tier-1 heap is updated only if the root event in tier-2 changes after an operation. Consequently, the best case time complexity becomes $\log e$ when compared to $\log e \cdot l$ for the **heap**. Furthermore, cancellation of events for an anti-message is restricted to just the tier-2 entries of LP_{dest} with utmost 1 tier-1 operation to update schedule position of LP_{dest}. A **std::vector** is used as the backing storage for both tiers and standard algorithms are used to maintain the min-heap property for both tiers after each operation.

4.4 2-tier Fibonacci Heap (fibHeap)

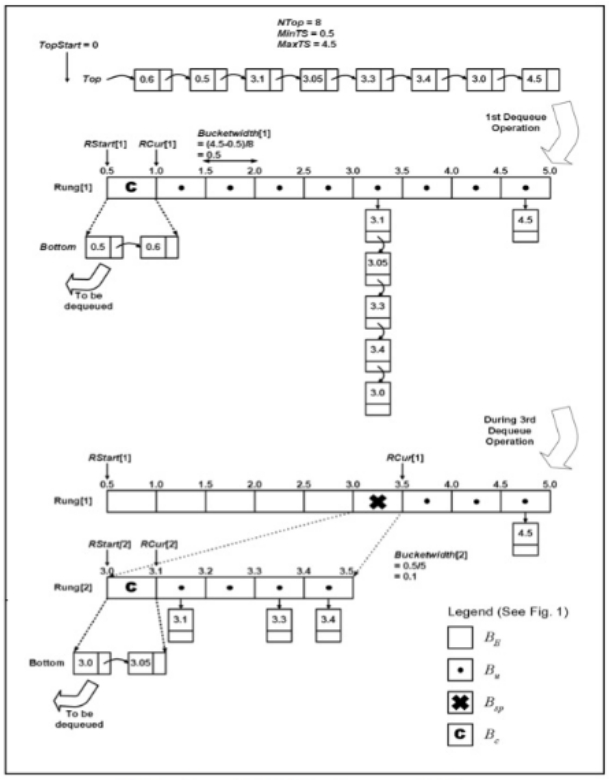
The **fibHeap** is an extension to the previous **2tHeap** data structure and uses a Fibonacci heap for scheduling LPs. The Fibonacci heap is a slightly modified version from the BOOST C++ library. The Fibonacci heap has an amortized constant time for changing key values and finding minimum. Consequently, 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 $\log(e) + 1^*$.

4.5 Three-tier Heap (3tHeap)

The **3tHeap** builds upon **2tHeap** by further subdividing the second tier into two tiers as shown in Figure 4.2(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 deallocation overhead.



(a) Structure



(b) Dequeue Operations

Figure 4.3: Structure of Ladder Queue

Source: Tang et al. [11]

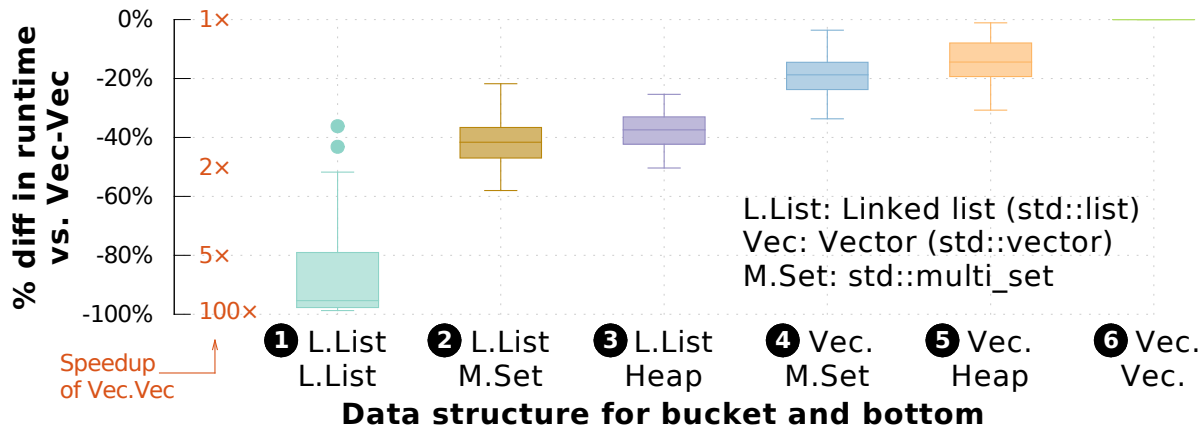
4.6 Ladder Queue (ladderQ)

The **ladderQ** is a priority queue implementation proposed by Tang et al [11] with amortized constant time complexity as summarized in Table 4.1. 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 [8, 11, 12]. 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. The multi-tier data structures also aim to minimize the number of events to be sorted. However, in contrast to the **ladderQ**, the other data structures always fix-up and maintain a minimum heap property. As shown in Figure 4.3(a), the ladder queue consists of the following 3 substructures:

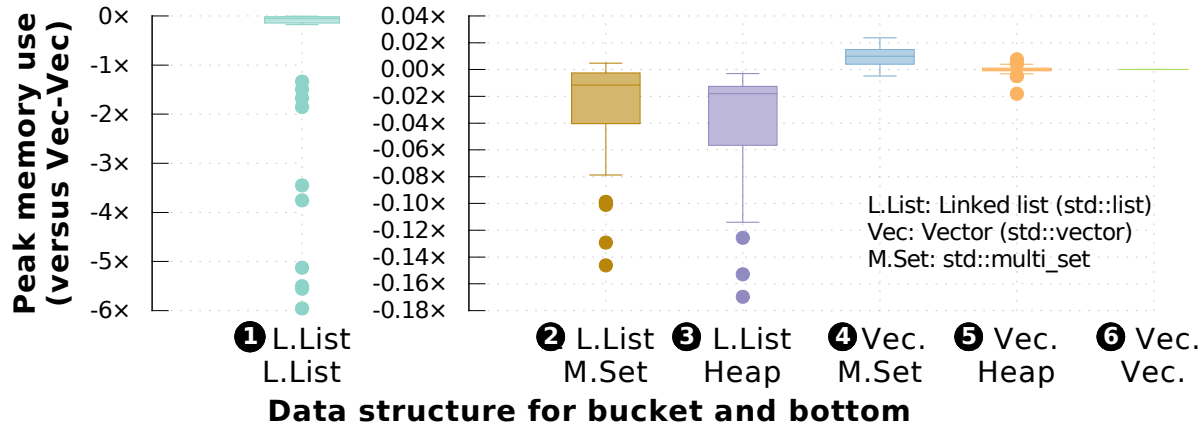
1. *Top*: An unsorted list which contains events scheduled into the distant future or epoch.
2. *Ladder*: Consists of multiple rungs, *i.e.,*list of buckets. Each bucket contains list of events with a finite range of time stamp values. Hence, although events within a bucket are not sorted, the buckets 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.
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 [11].

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 adding 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 4.3(b), the bucket-width computed from the

time stamp in *Top* is (4.5**max** - 0.5**min**)/8 = 0.5. In accordance with their timestamps, events from *Top* are placed into the appropriate buckets in Rung-1. Consequently, events with time stamps 0.5 and 0.6 are placed in the 1st bucket of 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 4.3(b), Rung-2 is generated for time stamped events in the range of 3.0 to 3.5. As shown in the Figure, The bucket containing events are sequentially removed from the bottom most rung in Ladder. 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 [11] with some modifications.



(a) Comparison sequential runtimes



(b) Peak memory used

Figure 4.4: Comparison of execution time and peak memory for PHOLD benchmark (different parameter settings) using 6 different ladderQ configurations

4.6.1 Fine tuning Ladder Queue performance

As previously stated, our implementation closely followed the design in the original paper by Tang et al [11]. However, to minimize runtime constants, we have explored different configurations for the buckets and the *Bottom* in the **ladderQ**. Specifically, we have explored the following 6 configurations – ❶ L.List-L.List: using a doubly-linked list (L.List) implemented by **std::list**) for

buckets and bottom. Events are inserted into bottom via linear search as proposed by Tang et al. ❷ L.List-M.Set: L.List for buckets and a Multi-set ($\log n$ operations) for bottom, ❸ L.List-Heap: a L.List and a binary heap (backed by a **std::vector**) for bottom, ❹ Vec-M.Set: a dynamically growing array (*i.e.,* **std::vector**) for buckets and Multi-set bottom, ❺ Vec-Heap: Vector buckets and binary heap for bottom, and ❻ Vec-Vec: Vector for buckets and bottom. This configuration enables using quick sort(*i.e.,* **std::sort**) for sorting buckets and binary search for inserting events into bottom.

Runtime comparison of the 6 **ladderQ** configurations is summarized in Figure 4.4. 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 4.4(a). The L.List-L.List configuration was generally the slowest and performed 85 ×(or 98%) slower than the Vec-Vec configuration. The peak memory used for simulations is shown in Figure 4.4(b), in comparison with the Vec-Vec configuration. As shown by the charts in Figure 4.4, the increased performance of Vec-Vec comes at about a 6×increase in peak memory footprint when compared to L.List-L.List configuration. This increased footprint arises because the **std::vector** internally doubles its capacity as it grows. With many buckets in the **ladderQ**, each implemented using a **std::vector**, the overall peak memory footprint is higher. Certainly, the increased capacity is used if the number of events in buckets grow. However, the Vec-M.Set and Vec-Heap configurations consume a bit more memory in some configurations, showing that Vec-Vec is not the worst in memory consumption. Consequently, we use the Vec-Vec configuration as it provides the fastest performance among the 6 configurations.

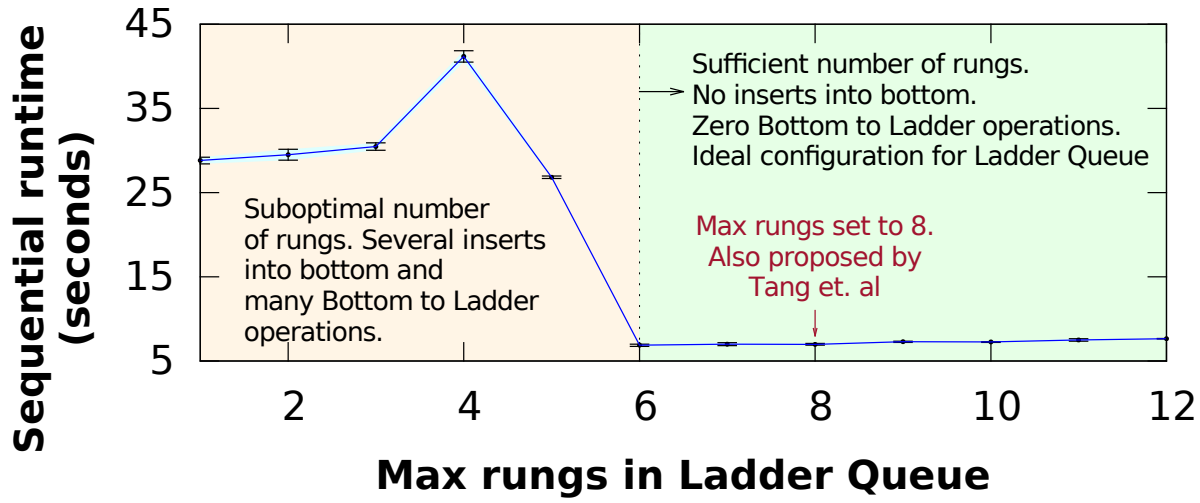


Figure 4.5: Impact of limiting rungs in Ladder

The maximum number of rungs in the *Ladder* also influences the overall performance of the **ladderQ** [11]. The chart in Figure 4.5 illustrates the impact of limiting the maximum number of rungs in the **ladderQ**. When the rungs are too few, the timestamp-based width of buckets is larger and more events with many different timestamps are packed into buckets. This also causes the *Bottom* to be longer with events spanning a broader range of timestamps. Consequently, when inserts happen into *Bottom*, many *Bottom*-to-*Ladder* re-bucketing operations are triggered to ensure bottom is short. These re-bucketing operations with many events significantly degrade performance. However, once sufficient number of rungs (6 rungs in this case) are permitted the events are better subdivide into smaller timestamp-based bucket widths. Small bucket widths in turn minimize inserts into bottom and *Bottom*-to-*Ladder* operations, ensuring good performance.

The chart in Figure 4.5 shows that a minimum of 6 rungs is required. For some select configurations of larger models we observed (data not shown) that 5 rungs would be sufficient. However, the number of rungs cannot exceed beyond a threshold to avoid infinite spawning of rungs [11]. Moreover, it limits the overheads involved in re-bucketing events from rung-to-rung [11]. Accordingly, based on the observations in figure 4.5, we decided to adopt a maximum of 8 rungs, consistent with the threshold proposed by Tang et al [11]. Furthermore, we trigger *Bottom*-to-*Ladder* re-bucketing only if the *Bottom* has events at different timestamps to further

reduce inefficiencies.

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 [8, 11, 12]. 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.

In this context, it is important to recollect from that – as an optimization, **MUSE** utilizes only one anti-message to 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.

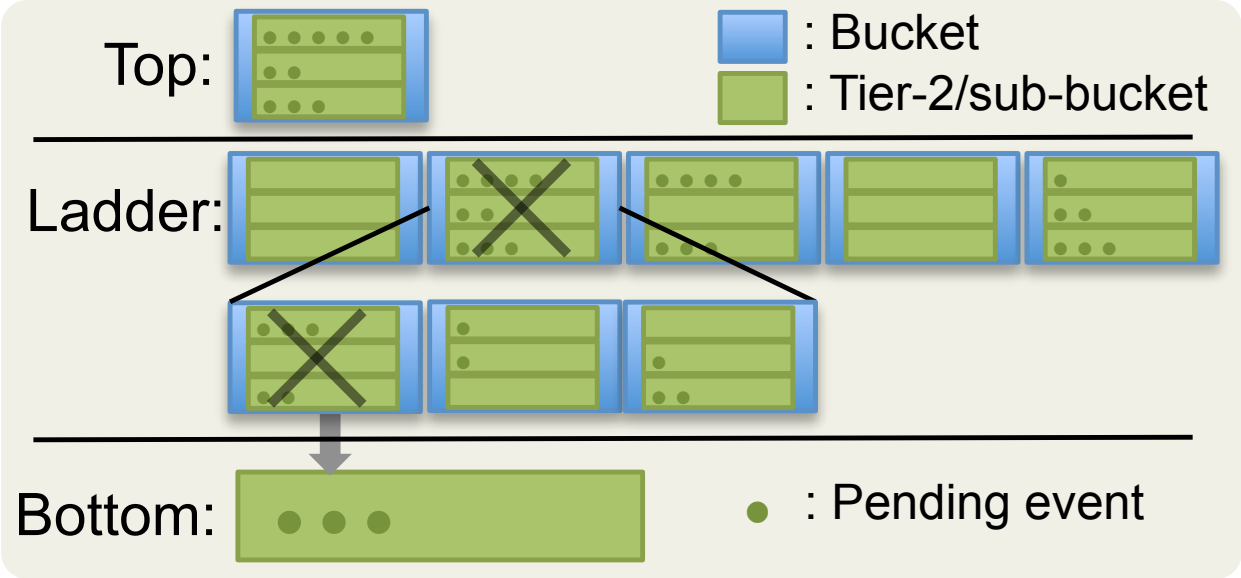


Figure 4.6: Structure of 2-tier Ladder Queue (2tLadderQ) with 3 sub-buckets / bucket (*i.e.*, $_{t2}$ k=3)

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.

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 4.6 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.

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

small, fixed value of $_{tk}$, the overhead becomes an ammortized 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.

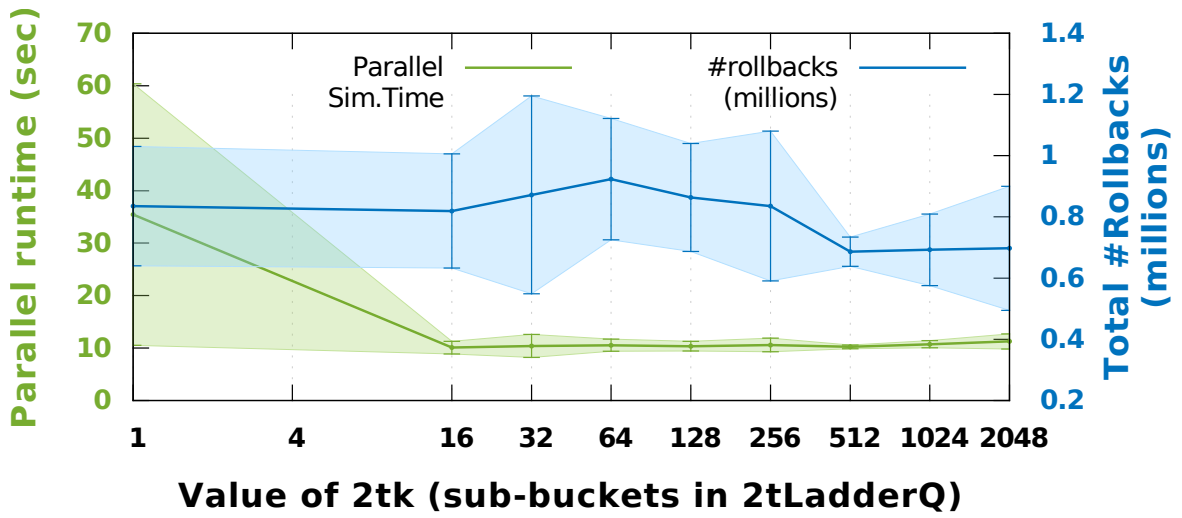


Figure 4.7: Effect of varying tk

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.

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

hardware configuration of the compute cluster used for parallel simulation.

Bibliography

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

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] T. 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.

[13] 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.

[14] 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.

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