Multi-Tier Priority Queues & 2-Tier Ladder Queue for Managing Pending Events

Julius D. Higiro

Miami University

March 7, 2017

Overview

1. Discrete Event Simulation (DES)
2. Research Motivation
3. Research Proposal
4. Related Work
5. Parallel Simulator Overview
6. Simulation Model
7. Scheduler Queues
8. Preliminary Simulation Results
9. Experimental Design
10. Plan of Action & Milestones

Intro to Discrete Event Simulation (DES)

Discrete Event Simulation (DES) is a framework for simulating the behavior of real or imagined systems.

Simulation properties

- an **entity** that is an object of interest in the simulated system.
- a **state** that is represented by a set of variables that describe the system at a particular point in time.
- an **event** that is an instantaneous occurrence that can change the state of the simulated system.
- a virtual simulation **clock** that indicates the time of the last event occurrence that has been simulated.
- an **evolution** of the modeled system that is given by a chronologically ordered sequence of events.
  $\mathbf{s_0, (e_0, t_0), s_1, (e_1, t_1), s_2,(e_2, t_2), \ldots\ldots., s_n,(e_n, t_n)}$

Intro to Discrete Event Simulation (DES)

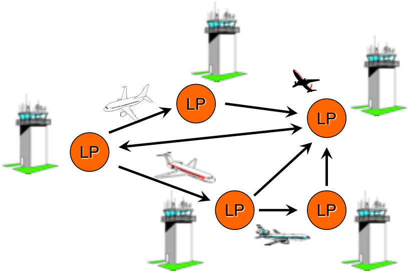
Simulation application

- The model of the real or imagined system.
- The collection of state variables.
- The collection of event handlers.

Simulator

- The environment that manages and executes the simulation.
- Maintains an event list for handling events.
- Manages forward progress of simulation time.

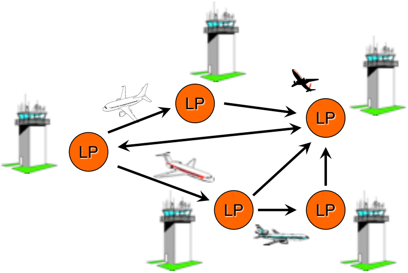
Implementation of DES



- DES is implemented as a set of logical processes (LPs).
- LPs map to physical processes in the simulated system.
- Each LP has the following components:

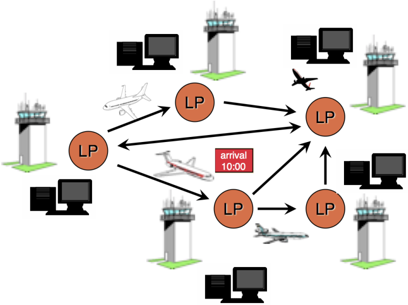
  state queue: maintains saved states of the LP.
  input queue: maintains events received from other LPs.
  output queue: maintains events sent to other LPs.
  local clock: time of last event processed.

Implementation of DES



- LPs interact with each other by exchanging and processing time stamped events.
- Events that are yet to be processed are called **pending events**.
- LPs process these events in time stamp order order and generate new events that are transmitted to other LPs.

Parallelism in DES



- Parallel computing simulates DES models on parallel computers.
- In parallel DES (PDES), LPs exchange time stamped events using message passing.
- Processing events concurrently on different processors is hard.
- **Synchronization problem**: since the simulation runs in parallel, all LPs have their own internal clock, so each LP may be at different points in time relative to each other.

Parallelism in DES

Conservative approach to PDES

- Causality errors are not allowed to occur.
- Strategy required to determine when it is safe to process an event.

Optimistic approach to PDES

- Allows causality errors to occur.
- When a causality error is detected, rollback mechanism is invoked.

Research Motivation

- Data structures for managing and prioritizing pending events play a critical role in ensuring efficient sequential and parallel simulations.
- The importance and effectiveness of data structures for event management is more conspicuous during large simulations.
- Large simulations can have large amounts of LPs and each LP processes and generates many events.
- The synchronization strategy in PDES can also impact the effectiveness of data structure because of the additional processing required during rollback.

Research Proposal

- Our research proposes and explores multi-tier data structures for managing event list in sequential and optimistic parallel simulations.
- We aim to compare the effectiveness of our various data structures against the Ladder Queue pending event data structure.
- **Research Thesis**: Our multi-tier data structures (**2tLadderQ** and **3tHeap**) outperforms all other data structures in sequential and optimistically parallel simulations.

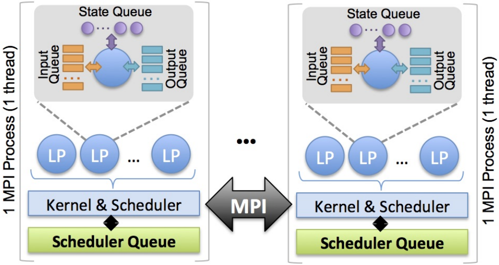
Related Work

- Many investigations have explored the effectiveness of a wide variety of data structures for managing the pending event set.
- Tang et al., proposed a pending event set data structure named Ladder Queue as a priority queue based implementation of an event list that results in amortized O(1) performance. The authors, presented ladder queue as an improvement on existing priority queue based event lists data structures such as calendar queues.
- Franceschini et al., 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.

Related Work

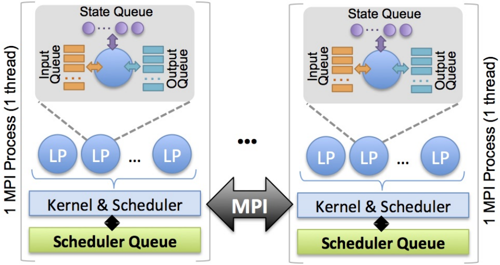
- Dickman et al., compared event list data structures that consisted of Splay Tree, STL Multiset and Ladder Queue. However, the focus of their paper was in developing a framework for handling pending event set data structure in shared memory PDES. A central component of their study was the identification of an appropriate data structure and design for the shared pending event set.
- Gupta et al., 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 et al., have 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.

Parallel Simulator Overview



- Assessment of the data structures will be conducted on MUSE.
- MUSE performs sequential and optimistically parallel simulations.
- The kernel handles LP registration, event processing, state saving, synchronization and garbage collection.

Parallel Simulator Overview



- The kernel uses a centralized Least Time-Stamp First (LTSF) scheduler queue for managing events and scheduling event processing for local LPs.
- Scheduler is designed to permit different data structures to be used for managing pending events.

Parallel Simulator Overview

A scheduler queue is required to implement four key operations to manage pending events.

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

2. **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.

Parallel Simulator Overview

3. **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.

Parallel Simulator Overview

4. **Cancel pending events**: This operation is used as part of rollback recovery process to aggressively remove *all pending events* sent by a given LP ($\mathrm{LP}_{\mathrm{sender}}$) to another LP ($\mathrm{LP}_{\mathrm{dest}}$) at-or-after a given time ($t_{rollback}$). In our implementation, only one anti-message with send time $t_{rollback}$ is dispatched to $\mathrm{LP}_{\mathrm{dest}}$ from $\mathrm{LP}_{\mathrm{sender}}$ to cancel prior events sent by $\mathrm{LP}_{\mathrm{sender}}$ to $\mathrm{LP}_{\mathrm{dest}}$ at-or-after $t_{rollback}$. This is a contrast to conventional aggressive cancellation in which one anti-message is generated per event.

Simulation Model

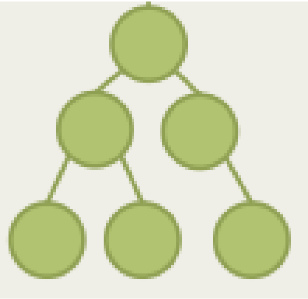
Table: Parameters in PHOLD benchmark

Parameter	Description
rows	Total number of rows in model.
cols	Total number of columns in model. $\#\text{LPs} = $ **rows** x **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.

Scheduler Queues

- MUSE contains 6 scheduling queues for managing pending events.
- The queues are classified into two categories: single-tier and multi-tier queues.
- Single-tier queues use only a single data structure to implement the 4 key operations.
- Multi-tier queues organizes events into tiers and each tier is implemented using different data structures.

Binary Heap (**heap**)



Time Complexity

Enqueue: $\log(e \cdot l)$

Dequeue: $\log(e \cdot l)$

Cancel: $z \cdot \log(e \cdot l)$

Legend:

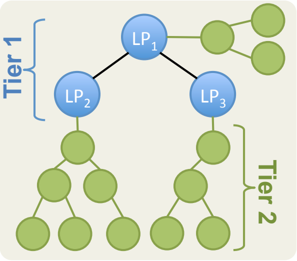
l: #LPs

e: #events / LP

z: #canceled events

- It is a single-tier data structure that it is implemented as an array object.
- A std::vector is used as the backing container and C++11 algorithms (std::push_heap, st::pop_heap) are used to maintain the heap.
- The heap is prioritized based on time stamp with the lowest time stamp at the root of the heap.

2-tier Heap (**2tHeap**)



Time Complexity

Enqueue: $\log(e)+\log(l)$

Dequeue: $\log(e)+\log(l)$

Cancel: $z \cdot \log(e)+\log(l)$

Legend:

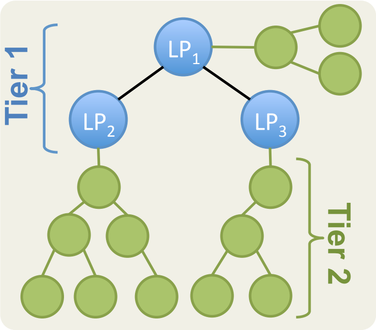
l: #LPs

e: #events / LP

z: #canceled events

- 2tHeap was designed to reduced the time complexity of cancel operations by subdividing events into two distinct tiers.
- The first tier has containers for each local LP on an MPI-process.
- Each of the the tier-1 containers has a heap of events to be processed by a given LP.
- A std::vector is used as the backing container for both tiers and standard algorithms are used to maintain the min-heap property for both tiers after each operation.

2-tier Fibonnaci Heap (**fibHeap**)

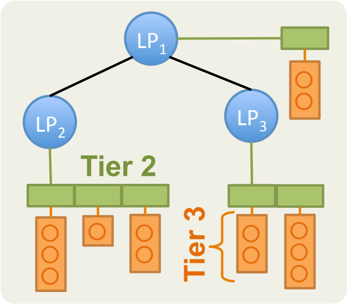


Time Complexity

Enqueue: $\log(e) + 1^*$
Dequeue: $\log(e) + 1^*$
Cancel: $z \cdot \log(e) + 1^*$
Legend:
l: #LPs
e: #events / LP
z: #canceled events
1^*: amortized constant

- The fibHeap is an extension of the 2tHeap data structure. It uses a Fibonnaci heap for scheduling LPs.
- The second tier is a binary heap data structure.

3-tier Heap (**3tHeap**)

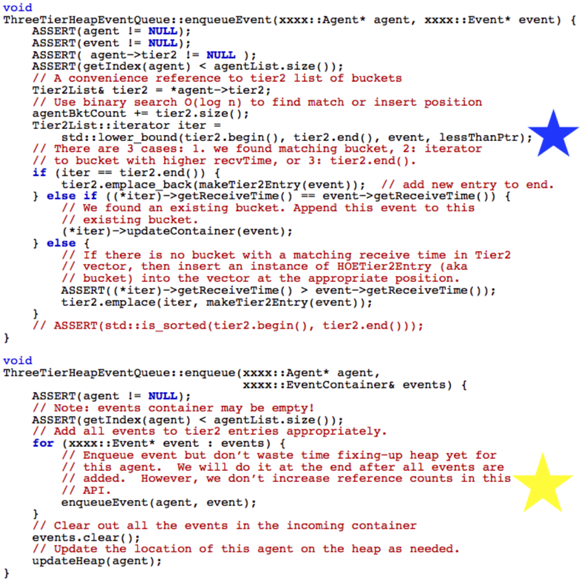


Time Complexity

Enqueue: $\log(\frac{e}{c}) + \log(l)$
Dequeue: $\log(l)$
Cancel: $e + \log(l)$
Legend:
l: #LPs
e: #events / LP
c: #concurrent events
z: #canceled events

- The 3tHeap builds upon 2tHeap by further subdividing the second tier into two tiers.
- The binary heap implementation for the first tier that manages LPs for scheduling has been retained from 2tHeap.
- 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.

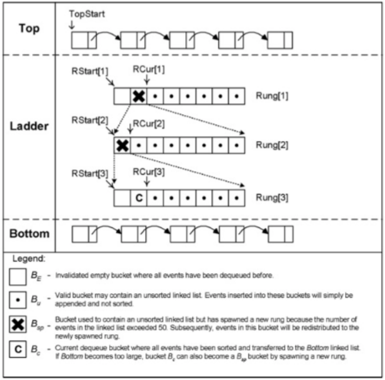
3-tier Heap (**3tHeap**) - Enqueue Operation



```
void
ThreeTierHeapEventQueue::enqueueEvent(xxxx::Agent* agent, xxxx::Event* event) {
    ASSERT(agent != NULL);
    ASSERT(event != NULL);
    ASSERT( agent->tier2 != NULL );
    ASSERT(getIndex(agent) < agentList.size());
    // A convenience reference to tier2 list of buckets
    Tier2List& tier2 = *agent->tier2;
    // Use binary search O(log n) to find match or insert position
    agentBktCount += tier2.size();
    Tier2List::iterator iter =
        std::lower_bound(tier2.begin(), tier2.end(), event, lessThanPtr);
    // There are 3 cases: 1. we found matching bucket, 2: iterator
    // to bucket with higher recvTime, or 3: tier2.end().
    if (iter == tier2.end()) {
        tier2.emplace_back(makeTier2Entry(event));  // add new entry to end.
    } else if ((*iter)->getReceiveTime() == event->getReceiveTime()) {
        // We found an existing bucket. Append this event to this
        // existing bucket.
        (*iter)->updateContainer(event);
    } else {
        // If there is no bucket with a matching receive time in Tier2
        // vector, then insert an instance of HOETier2Entry (aka
        // bucket) into the vector at the appropriate position.
        ASSERT((*iter)->getReceiveTime() > event->getReceiveTime());
        tier2.emplace(iter, makeTier2Entry(event));
    }
    // ASSERT(std::is_sorted(tier2.begin(), tier2.end()));
}

void
ThreeTierHeapEventQueue::enqueue(xxxx::Agent* agent,
                                 xxxx::EventContainer& events) {
    ASSERT(agent != NULL);
    // Note: events container may be empty!
    ASSERT(getIndex(agent) < agentList.size());
    // Add all events to tier2 entries appropriately.
    for (xxxx::Event* event : events) {
        // Enqueue event but don't waste time fixing-up heap yet for
        // this agent.  We will do it at the end after all events are
        // added.  However, we don't increase reference counts in this
        // API.
        enqueueEvent(agent, event);
    }
    // Clear out all the events in the incoming container
    events.clear();
    // Update the location of this agent on the heap as needed.
    updateHeap(agent);
}
```

Ladder Queue (**ladderQ**)



Legend:

- □ B_E - Invalidated empty bucket where all events have been dequeued before.
- ⊡ B_u - Valid bucket may contain an unsorted linked list. Events inserted into these buckets will simply be appended and not sorted.
- ☒ B_{sp} - Bucket used to contain an unsorted linked list but has spawned a new rung because the number of events in the linked list exceeded 50. Subsequently, events in this bucket will be redistributed to the newly spawned rung.
- C B_c - Current dequeue bucket where all events have been sorted and transferred to the *Bottom* linked list. If *Bottom* becomes too large, bucket B_c can also become a B_{sp} bucket by spawning a new rung.

Time Complexity

Enqueue: 1*

Dequeue: 1*

Cancel: $e \cdot l$

Legend:

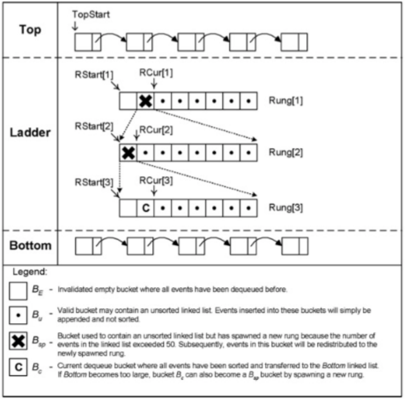
l: #LPs

e: #events / LP

1*: amortized constant

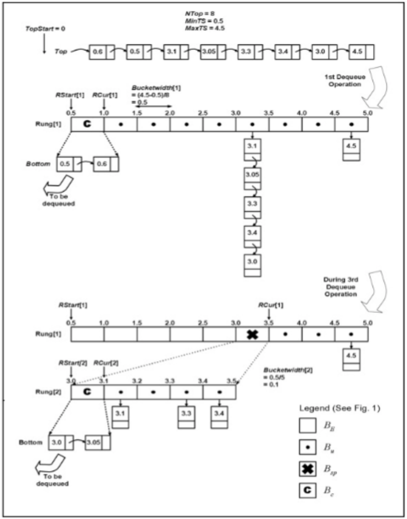
- Ladder Queue is a priority queue implementation proposed by Tang et al. with amortized constant time complexity.
- There are two key ideas underlying the Ladder Queue, namely: minimize the number of events to be sorted and delay sorting of events as much as possible.

Ladder Queue (**ladderQ**)

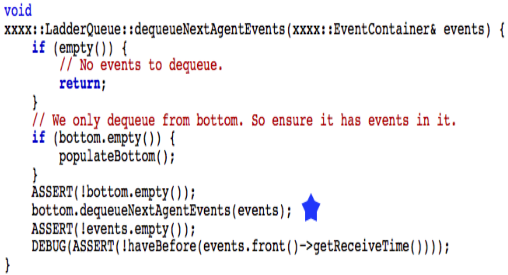


Legend:

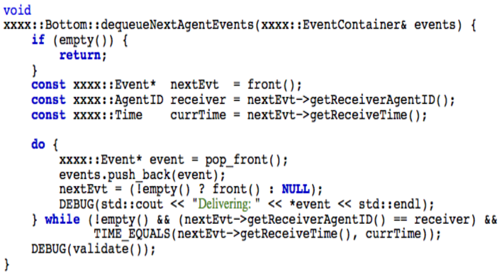
- □ B_E - Invalidated empty bucket where all events have been dequeued before.
- ● B_u - Valid bucket may contain an unsorted linked list. Events inserted into these buckets will simply be appended and not sorted.
- ☒ B_{sp} - Bucket used to contain an unsorted linked list but has spawned a new rung because the number of events in the linked list exceeded 50. Subsequently, events in this bucket will be redistributed to the newly spawned rung.
- C B_c - Current dequeue bucket where all events have been sorted and transferred to the *Bottom* linked list. If *Bottom* becomes too large, bucket B_c can also become a B_{sp} bucket by spawning a new rung.



Ladder Queue (**ladderQ**) Dequeue Operation



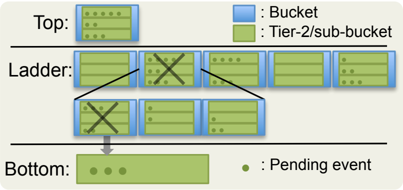
```
void
xxxx::LadderQueue::dequeueNextAgentEvents(xxxx::EventContainer& events) {
    if (empty()) {
        // No events to dequeue.
        return;
    }
    // We only dequeue from bottom. So ensure it has events in it.
    if (bottom.empty()) {
        populateBottom();
    }
    ASSERT(!bottom.empty());
    bottom.dequeueNextAgentEvents(events);
    ASSERT(!events.empty());
    DEBUG(ASSERT(!haveBefore(events.front()->getReceiveTime())));
}
```



```
void
xxxx::Bottom::dequeueNextAgentEvents(xxxx::EventContainer& events) {
    if (empty()) {
        return;
    }
    const xxxx::Event*  nextEvt  = front();
    const xxxx::AgentID receiver = nextEvt->getReceiverAgentID();
    const xxxx::Time    currTime = nextEvt->getReceiveTime();

    do {
        xxxx::Event* event = pop_front();
        events.push_back(event);
        nextEvt = (!empty() ? front() : NULL);
        DEBUG(std::cout << "Delivering: " << *event << std::endl);
    } while (!empty() && (nextEvt->getReceiverAgentID() == receiver) &&
             TIME_EQUALS(nextEvt->getReceiveTime(), currTime));
    DEBUG(validate());
}
```

2-tier Ladder Queue (**2tLadderQ**)



Time Complexity

Enqueue: 1^*

Dequeue: 1^*

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

Legend:

l: #LPs

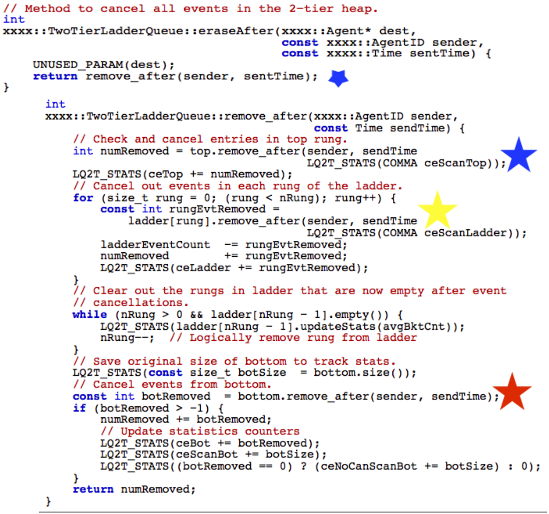
e: #events / LP

1^*: amortized constant

${}_{t2}k$: parameter

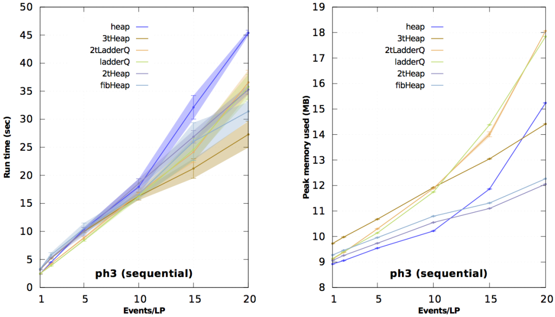
- 2-tier Ladder Queue is the proposed alternative to Ladder Queue because the cost of event cancellation during rollbacks is reduced.
- 2tLadderQ retains the amortized constant time complexity of ladderQ with performance gains during event cancellation.

2-tier Ladder Queue (**2tLadderQ**) Cancel Operation



```
// Method to cancel all events in the 2-tier heap.
int
xxxx::TwoTierLadderQueue::eraseAfter(xxxx::Agent* dest,
                                        const xxxx::AgentID sender,
                                        const xxxx::Time sentTime) {
    UNUSED_PARAM(dest);
    return remove_after(sender, sentTime);
}
      int
      xxxx::TwoTierLadderQueue::remove_after(xxxx::AgentID sender,
                                             const Time sendTime) {
          // Check and cancel entries in top rung.
          int numRemoved = top.remove_after(sender, sendTime
                                            LQ2T_STATS(COMMA ceScanTop));
          LQ2T_STATS(ceTop += numRemoved);
          // Cancel out events in each rung of the ladder.
          for (size_t rung = 0; (rung < nRung); rung++) {
              const int rungEvtRemoved =
                  ladder[rung].remove_after(sender, sendTime
                                            LQ2T_STATS(COMMA ceScanLadder));
              ladderEventCount  -= rungEvtRemoved;
              numRemoved        += rungEvtRemoved;
              LQ2T_STATS(ceLadder += rungEvtRemoved);
          }
          // Clear out the rungs in ladder that are now empty after event
          // cancellations.
          while (nRung > 0 && ladder[nRung - 1].empty()) {
              LQ2T_STATS(ladder[nRung - 1].updateStats(avgBktCnt));
              nRung--;  // Logically remove rung from ladder
          }
          // Save original size of bottom to track stats.
          LQ2T_STATS(const size_t botSize  = bottom.size());
          // Cancel events from bottom.
          const int botRemoved  = bottom.remove_after(sender, sendTime);
          if (botRemoved > -1) {
              numRemoved += botRemoved;
              // Update statistics counters
              LQ2T_STATS(ceBot += botRemoved);
              LQ2T_STATS(ceScanBot += botSize);
              LQ2T_STATS((botRemoved == 0) ? (ceNoCanScanBot += botSize) : 0);
          }
          return numRemoved;
      }
```

Preliminary Sequential Simulation Results

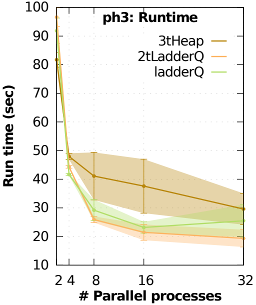


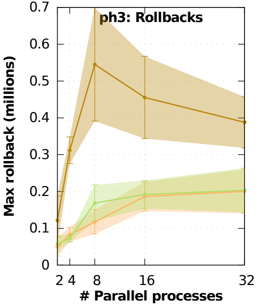
Configuration

#LPs (Rows x Cols): 1,000(100x10)
Sim. End Time (Seq): 5000
Sim. End Time (Parallel): 20000

- Sequential simulation run time and peak memory usage statistics is shown with **%Self-events** $= 25\%$ and $\lambda = 1$ at varied values for **eventsPerLP**.

Preliminary Parallel Simulation Results





Configuration

#LPs (Rows x Cols): $1,000(100 \times 10)$
Sim. End Time (Seq): 5000
Sim. End Time (Parallel): 20000

- Statistics from parallel simulation with **%Self-events** $= 25\%$, $\lambda = 10$ and **eventsPerLP** $= 10$.

Experiment Design

- Assessments of the effectiveness of the six scheduler queues will be performed by running different configurations of the PHOLD benchmark in sequential and optimistically parallel simulations.
- Due to a large number of PHOLD parameters and combinations of their values. We will identify the most influential PHOLD parameters that impact performance of the scheduler queues using **Generalized Sensitivity Analysis**.
- The data to be collected and assessed consists of **simulation run time** and **peak memory usage**.

Plan of Action & Milestones

1. **Sequential & parallel simulation assessment**: Data will be collected to assess the effectiveness of the different data structures in sequential and parallel simulations. Data collection and analysis for sequential simulations will be completed by 14 March 2017. Data collection and analysis for parallel simulations will be completed by 22 March 2017.

2. **Implement 2 additional simulation models**: We will extend the experimental analysis to include 2 additional models in order to evaluate performance of the data structures across different simulation models. The 2 simulation models will be implemented by 30 March 2017.

Plan of Action & Milestones

3. **Assessment using additional simulation models**: The assessment of data structure performance using the models in sequential and optimistic parallel simulations will be completed by 28 April 2017.

4. **Record experimental results and analysis**: The research thesis writing will be completed by 19 May 2017 in preparation for the thesis defense.

References

- Dickman et al., (2013) Event Pool Structures for PDES on Many-core Beowulf Clusters. *In Proceedings of the 1st ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* ACM, New York, NY, USA, 103-114.

- Franceschini et al., (2015) A Comparative Study of Pending Event Set Implementations for PDEVS Simulation. *In Proceedings of the Symposium on Theory of Modeling and Simulation* Society for Computer Simulation International San Diego, CA, USA, 77-84.

- Wilsey et al., (2014) Lock-free Pending Event Set Management in Time Warp. *In Proceedings of the 2nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* ACM, New York, NY, USA, 15-26.

- Jafer et al., (2013) Synchronization methods in parallel and distributed discrete-event simulation Simulation Modeling Practice and Theory 30(2013), 54-73.

- Marotta et al., (2016) A Non-Blocking Priority Queue for the Pending Event Set.In Proceedings of the 9th EAI International Conference on Simulation Tools and Techniques ICST, Brussels, Belgium, 46-55.

- Tang et al., (2005) Ladder Queue: An O(1) Priority Queue Structure for Large-scale Discrete Event Simulation.ACM Trans. Model. Comput. Simul. 15, 3(July 2005), 175-204.

The End