A Non-Blocking Priority Queue for the Pending Event Set

Romolo Marotta, Mauro Ianni, Alessandro Pellegrini and Francesco Quaglia
romolo.marotta@gmail.com, mianni@dis.uniroma1.it, pellegrini@dis.uniroma1.it, quaglia@dis.uniroma1.it
DIAG – Sapienza, University of Rome

ABSTRACT

The large diffusion of shared-memory multi-core machines has impacted the way Parallel Discrete Event Simulation (PDES) engines are built. While they were originally conceived as data-partitioned platforms, where each thread is in charge of managing a subset of simulation objects, nowadays the trend is to shift towards share-everything settings. In this scenario, any thread can (in principle) take care of CPU-dispatching pending events bound to whichever simulation object, which helps to fully share the load across the available CPU-cores. Hence, a fundamental aspect to be tackled is to provide an efficient globally-shared pending events' set from which multiple worker threads can concurrently extract events to be processed, and into which they can concurrently insert new produced events to be processed in the future. To cope with this aspect, we present the design and implementation of a concurrent non-blocking pending events' set data structure, which can be seen as a variant of a classical calendar queue. Early experimental data collected with a synthetic stress test are reported, showing excellent scalability of our proposal on a machine equipped with 32 CPU-cores.

CCS Concepts

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

1. INTRODUCTION

Parallel Discrete Event Simulation (PDES) is a well known method to speedup the execution of large/complex discrete event models [9]. The core concept it relies on is the partitioning of the model into different simulation objects, interacting via cross-scheduling of events. Multiple threads take care of concurrently executing events bound to the objects, and the trajectories of the state updates of different objects is driven by synchronization protocols (e.g. [6, 15]) aimed at guaranteeing causal consistency among the processed events.

Most of the historical proposals along the path of devising/building efficient PDES platforms are based on the notion of "assignment" of objects to threads. In particular, each thread running within the PDES system is in charge of handling the execution of a subset of the simulation objects, by also managing the corresponding pools of their pending events in isolation (with respect to the execution of other threads). In this scenario, load balancing has been supported by periodically migrating objects (and their associated event pools) across threads according to different policies [18, 5, 10, 3]. Still, the objects and their pools are accessed in isolation by threads after any rebalance operation takes place.

However, the advent and large diffusion of shared-memory multi-core machines led to a shift in the design of PDES environments. In particular, the rising approach is to slide towards a share-everything paradigm, where the binding between threads and objects has (as an extreme) a lifetime equal to the processing of individual events. Examples of PDES platforms adhering to this paradigm have been envisaged in, e.g., [21, 7, 13, 8]. With this organization, event pools are no longer accessed by threads in isolation, rather in a shared mode. This is done in order to assign the highest priority pending events (i.e., those with lower timestamps) destined to whichever simulation object to any thread that has already finished processing its last assigned one. Multiple threads becoming free can therefore poll some shared event pool to extract the lowest-timestamp event concurrently. Additionally, they can concurrently access the pool to post newly-scheduled events resulting from the processing activities at the involved objects.

Although the problem of optimizing the performance of event-pool data structures has been long studied in the literature, most of the solutions are tailored for isolated (non-concurrent) accesses. In fact, the issue of optimizing event pools' management in face of concurrent accesses in PDES systems has only recently attracted attention by the research community (see, e.g., [11]) even though concurrent data structures for managing (ordered) sets have been proposed [12, 22] long before.

In this paper we present a non-blocking (lock-free) concurrent pending events' pool data structure, and its management logic, which also shows an amortized $O(1)$ time complexity. The latter feature is not provided by existing proposals like non-blocking (skip)lists [12, 22], which pay a linear (or at least logarithmic) cost for insertion operations. Our proposal is suited for conventional hardware since its only requirement from the underlying ISA (Instruction Set

Architecture) is the support for atomic Compare-and-Swap (CAS) operations, which is commonly found in off-the-shelf processors. As an example, such a support is offered by the Compare-and-Exchange (CMPXCHG) machine instruction in x86 processors.

We release our non-blocking pending events' set as free software[1], and we report in this article a preliminary performance study showing excellent scalability of our proposal when tested on top of a 32-core HP ProLiant machine. The tests have been carried out under synthetic workloads which, as we will discuss, are anyhow representative of stress scenarios one may expect when employing a shared events' pool within PDES systems.

2. RELATED WORK

Optimized event set data structures have been the object of several studies in the literature. One main thread of research has been devoted to time-complexity optimization to manage sequential insertions and extractions, namely when a single thread manages the pool of events in isolation. The seminal article in [4] presents the Calendar Queue, a timestamp-ordered data structure based on multi-lists, each one associated with a well-defined time bucket. It offers amortized constant time insertion of events with generic timestamps and constant time extraction of the event with the minimum timestamp. The Ladder Queue [23] is a variant of the Calendar Queue which is more suited for contexts with skewed distribution of the timestamps of the events to be managed. Compared to the Calendar Queue, the Ladder Queue entails a dynamic splitting of an individual bucket in sub-intervals (hence sub-lists of records) if the number of elements associated with it exceeds a given threshold. This is not supported by the Calendar Queue, which instead requires to resize the overall data structure, rather than a single bucket. The LOCT Queue [19] is an additional variant which allows to reduce the actual overhead for constant time insertion/extraction operations thanks to the introduction of a compact hierarchical bitmap indicating the status of any bucket (empty or not). As hinted, none of these proposals has been devised for concurrent accesses. Therefore, their usage in scenarios with workload sharing (hence event-pool sharing) across multiple threads would require global locking to serialize the accesses. As also discussed in [11], this would be detrimental to scalability.

A few studies, such as [2], have attempted to provide queuing data structures enabling parallel accesses. These have been based on the partitioning of the data structure and on fine-grain locking approaches, only blocking a portion of the data structure upon performing an operation. However, the intrinsic scalability limitations of locking still make these proposals unsuited for large levels of parallelism. In fact, as shown in [20], they can reveal effective for parallel machines with relatively small counts of CPU-cores.

Non-blocking management of sets has also been studied, and there exist various proposals such as non-blocking linked lists [12] or the skip-list [22]. However, these solutions do not allow executing all the operations (either an insertion or an extraction) in constant time. In particular, non-blocking linked lists pay a linear cost for ordered insertions while the skip-list pays logarithmic cost for this same operation. Given that algorithms in the non-blocking class may suffer from abort/retries caused by concurrent conflicting accesses to some portion of the data structure, aborting operations that may entail non-constant time can be significantly adverse to performance. We note that the likelihood of conflicts can be expected to increase for larger thread counts and for frequent accesses to the shared pool. The latter aspect highlights the risk for reduced effectiveness in PDES applications with finer grain events, where threads may spend a significant portion of their lifetime in the management of housekeeping tasks (like the management of the shared pool).

To the best of our knowledge, the work in [11] is the first to propose a non-blocking events' set data structure with constant time operations. This proposal is a variant of the Ladder Queue, where the elements are always bound to the correct bucket, but the bucket list is not ordered. Constant time is achieved since the extraction from an unordered bucket returns the first available element, which not necessarily corresponds to the one with the minimum timestamp. In other words, this variant is based on a partial ordering of elements, which allows to get rid of the costly guarantee of returning the lowest-timestamp event upon extractions. In fact, this proposal has been devised for PDES systems relying on speculative processing, where unordered extractions leading to causal inconsistencies within the simulation model trajectory are recovered (in terms of their effects on the simulation model trajectory) via proper rollback/recovery mechanisms. However, still for speculative PDES, some recent results [7, 21] have shown the relevance of delivering event records from the shared pool in correct order, as a means to build synchronization schemes able to exploit alternative forms of recoverability standing aside of the traditional Time Warp protocol [15].

Last, the recent proposal in [13] explores the idea of managing a shared pool concurrently by relying on Hardware Transactional Memory (HTM) support, currently available on few processors by major vendors. Insertions and extractions are run as HTM-based transactions, hence concurrently in non-blocking mode. However, the level of scalability of this approach is limited by the level of parallelism in the underlying HTM-equipped machine, which still appears modest. Also, HTM-based transactions can abort for several reasons, not only because of conflicting concurrent accesses to a same portion of the data structure. As an example, they can abort because of conflicting accesses to the same cache line by multiple cores, which might be adverse to PDES with (very) large data set.

Compared to literature results, our proposal is the unique that jointly offers: i) non-blocking concurrent accesses, ii) amortized constant time operations, iii) total order while managing timestamped event records, and iv) independence on specific hardware support.

3. THE NON-BLOCKING QUEUE

3.1 Basics

Given that we target amortized constant time for any operation, our non-blocking priority queue (NBPQ) is based on a multi-list data structure, inspired to the basic ideas underlying the Calendar Queue and its variants. Moreover, this strategy allows confining concurrent potentially-conflicting accesses to a same individual list, thus fully avoiding conflicts when concurrently accessing different lists. The time

[1] `https://github.com/HPDCS/NBPQ`

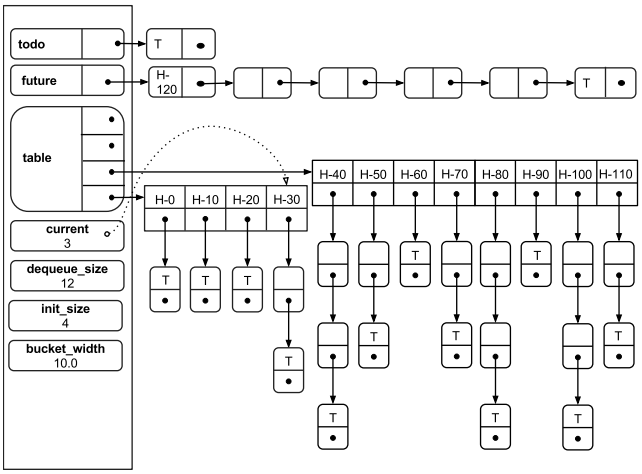


Figure 1: The NBPQ data structure.

axis is subdivided into equal slots and each list, that we also refer to as bucket, is associated with a particular time slot $[a, b)$. Each bucket keeps events with timestamp $T \in [a, b)$. The length of time slots is called *bucket width* and generally should change over time in order to control the average number of items per bucket. Similar considerations can be made for the number of buckets, which should be variable in order to cover a well-suited interval of time, depending on the locality of the events and its variation over time.

In our solution we explicitly focus on the second point since changing the bucket width would mean changing the strategy for referring buckets. In a fully non-blocking approach this should be done while still enabling lock-free accesses according to the old strategy up to the point in time where the new one is finalized. This type of approach will be the objective of future work we plan to carry out. In any case, the reshuffle operation changing the bucket size in our current version of the NBPQ can be executed as a critical section, which would lead to very reduced intrusiveness along time if executed infrequently, with respect to normal insertion/extraction operations on the queue (which are the ones whose concurrency needs to be optimized). Also, the selection of the new bucket size can be inherited, in terms of policy, from already existing solutions, a few of which are PDES-oriented such as [19]. Clearly, avoiding the resize of the bucket at all leads to the scenario where the amortizing factor for the cost of the queue operations is not directly controllable, and purely depends on the distribution of event records across the different buckets. Performance implications by this scenario will be analyzed in Section 4.

A representation of our NBPQ data structure is provided in Figure 1. The buckets are arranged in a dynamic array `table` in order to allow accesses based on indices. The queue uses the following variables: `current` stores the index of the bucket which contains the minimum timestamp; `init_size` is the initial number of buckets in the queue; `bucket_width` is the width of a bucket, i.e., the length of the time interval covered by each bucket.

Since each bucket covers a fixed interval of time, at start-up the whole set of lists covers a time interval equal to $[0, \texttt{bucket\_width} \cdot \texttt{init\_size})$. In order to be able to keep events with timestamp in the interval in $[\texttt{bucket\_width} \cdot \texttt{init\_size}, +\infty)$, we use two overflow stacks, pointed by `future` and `todo`. A variable `enqueue_size` is used to represent the left limit of the covered timestamps in $[\texttt{bucket\_width} \cdot \texttt{init\_size}, +\infty)$. Also, a variable `dequeue_size` keeps the right limit of the time interval covered by all buckets. `todo` is initially set to `null`. At start-up, we have `dequeue_size` = `enqueue_size`. The organization of our queue and its initialization are shown in Algorithm 1.

Algorithm 1 NBPQ object.

```
NBPQueue {
      pointer<Stack> future;
      pointer<Stack> todo;
      pointer<HarrisSet> table[32];
      integer current;
      integer dequeue_size;
      integer init_size;
      double bucket_width;
}
1: procedure NBPQueue(integer size, double width)
2:     new ←NBPQueue()
3:     new.todo ←Stack(0)
4:     new.future ←Stack(size)
5:     new.init_size ← size
6:     new.dequeue_size ← size
7:     new.bucket_width ← width
8:     return new

Stack {
      pointer<Node> top;
      pointer<Node> replica;
      integer enqueue_size;
}
1: procedure Stack(integer index)
2:     new ←new Stack()
3:     new.top ← null
4:     new.enqueue_size ← index
5:     return new
```

Along its lifetime, our NBPQ works according to the following rules: (a) dequeue operations return an event with timestamp in the interval $[0, \texttt{dequeue\_size} \cdot \texttt{bucket\_size})$; (b) the stack pointed by `future` contains events with timestamps in the interval $[\texttt{enqueue\_size} \cdot \texttt{bucket\_size}, +\infty)$; (c) enqueue tasks can only decrease `current`[2], while dequeue tasks can only increase `current`.

Upon enqueueing an event e with timestamp T_e the index is computed as $i_e = \lfloor \frac{T_e}{\texttt{bucket\_width}} \rfloor$. If $i_e < \texttt{enqueue\_size}$, the event is inserted into the corresponding i_e-th bucket (in sorted manner if the bucket is not empty), otherwise it is pushed to `future`. Finally if $i_e \leq \texttt{current}$, to cope with concurrent manipulations of `current`, the index of the current minimum is updated.

On the other hand, the DEQUEUE procedure checks that the bucket at index `current` is not empty. In this case the first node of the list is removed and then returned, otherwise `current` is incremented by one and the check is repeated until a non-empty bucket is found or the value of `current` is equal to `dequeue_size`. If the last condition is true and `future` is empty, we can return `null` (the queue keeps no events). Conversely, if the overflow structure has some items, it contains the new minimum. Thus we have to move items from `future` into the array. The first step consists in doubling the number of buckets in the

[2]Differently from a classical Calendar Queue, insertion of events with timestamps smaller than the minimum one in the queue are allowed, due to the concurrent nature of the data structure.

Algorithm 2 Augmented Treiber's Stack algorithm.

```
1: procedure tryPush(Node n)
2:     tmp ← top
3:     n.next← tmp
4:     return ¬ISMARKED(tmp) ∧ CAS(&top, tmp, n)

5: procedure denyPush( )
6:     repeat
7:         tmp ← top
8:     until ISMARKED(tmp) ∨ CAS(&top, UNMARK(tmp), MARK(tmp))
```

queue[3]. At this point a new enqueue of an event e', such that $i_{e'} \leq 2 \cdot$ `enqueue_size`, could be served by a new allocated bucket. To allow this without blocking, we store the current `future` in a temporary storage `todo` and we exchange the overflow structure with a new one, that covers timestamps in $[2 \cdot$ `bucket_width` $\cdot$ `enqueue_size`$, +\infty)$. Clearly the new `future` has an `enqueue_size` that is twice as the previous one. Now new enqueues can be served coherently, while dequeues have to move every event from `todo` into `table`. When `todo` is empty, `dequeue_size` is doubled and dequeues can restart from `current` = `dequeue_size`. At this point we have restored the initial condition `dequeue_size` = `enqueue_size`.

As a final note, although for space constraints we cannot provide a formal proof, our solution is correct with respect to the *linearizability* property [14].

3.2 The NBPQ Algorithm

Our NBPQ algorithm relies on a few baseline lock-free data structures, namely Harris' Sorted Linked List [12] and Treiber's Stack [24], to which we have added features suited for our purposes, as we explain below.

Harris' Sorted Linked List. In order to support events with equal timestamps, we have slightly modified the SEARCH routine of the non-blocking linked list by Harris in a way similar in spirit to what is done in [17], so as to retrieve a right node such that it has a key greater than or equal to the search key. Anyhow, we have left the capability to invoke the original search by adding a parameter that allows to switch between the two versions. In particular, given a search key k, SEARCH(k, $<$) calls the original version which returns a right node with key greater than or equal to k. On the contrary, a right node with timestamp strictly greater than k is obtained by using SEARCH(k, $\leq$).

Treiber's Stack. A major change to the original data structure algorithm has been put in place by adding the new DENYPUSH procedure, which prevents any enqueue from succeeding. This is achieved by marking the `top` field as it is done the for logical deletion in the Harris' list. After that DENYPUSH completes, any push attempt fails to add an event and only pops are allowed. Consequently, we have defined a TRYPUSH routine that tries to insert a node into the stack with a `CAS` only if the `top` field is not marked and it returns true if and only if the swap succeeds. This allows us to ensure that pop and TRYPUSH cannot alternate each other after a completed DENYPUSH. Finally, our stack

[3]As we shall discuss, the NPBQ exposes an API for the notification of a newly-computed commit horizon T for the simulation, so that the queue can simply shift the minimum time it keeps to T, and recover memory for obsolete buckets. This avoids that doubling operations eventually lead to unbounded growth of the NBPQ data structure.

Algorithm 3 Non-blocking ENQUEUE.

```
1: procedure enqueue(event e)
2:     index ← ⌊ newNode.t / bucket_width ⌋
3:     newNode ← new node(e)
4:     insert(newNode, index)
5:     flushCurrent(index)

6: procedure insert(node newNode, integer index)
7:     repeat
8:         tmp ← future
9:         tmpSize ← tmp.enqueue_size
10:        if index ≥ tmpSize ∧ tmp.tryPush(newNode) then
11:            return
12:    until index < tmpSize
13:    bucket ← table[h1(index)][h2(index)]
14:    repeat
15:        ⟨leftNode, rightNode⟩ ← bucket.search(newNode.t, ≤)
16:        newNode.next ← rightNode
17:    until CAS(&leftNode.next, rightNode, newNode)

18: procedure h1(integer index)
19:    tmp ← (ibsr(index) − ibsr(init_size) + 1)
20:    return tmp & (-(index ≥ init_size))

21: procedure h2(integer index)
22:    return index & (∼ ( (index ≥ init_size) << ibsr(index)))

23: procedure flushCurrent(integer n)
24:    repeat
25:        old ← current
26:        ind ← old >> 32
27:        if n > ind then
28:            return
29:    until CAS(&current, old, (n << 32) | ABAMark())
```

maintains the left limit of the time interval that it covers in the `enqueue_size`variable. The complete stack algorithm, along with our modifications, is shown in Algorithm 2.

3.2.1 Queue Operations

The ENQUEUE operation (Algorithm 3) is split into two phases. The first one consists in connecting the new node to the structure. The INSERT routine first retrieves the `future` stack and then checks `enqueue_size` in order to discover where the node has to be connected. If it is lower than or equal to the linear index i of the event to be inserted, computed as $i = \lfloor \frac{timestamp}{\texttt{bucket\_width}} \rfloor$, then INSERT attempts to connect the new item to the stack with a TRYPUSH and returns whether it succeeds, otherwise it restarts from the beginning. The try-loop ends when either the TRYPUSH succeeds or the condition $i <$ `enqueue_size` is true. In this case we compute the location of the i-th bucket and we insert the node using the augmented search of the Harris' Sorted Linked List. This guarantees that events with same timestamps are managed in FIFO order[4]. At this point the insertion phase is completed and the event is connected either to `future` or to `table`. The second phase guarantees that `current` points to, or precedes, the bucket containing the minimum. This is done by the FLUSHCURRENT procedure that tries to exchange the `current` variable with a `CAS` until either it succeeds or i is strictly greater than the current value of `current`.

[4]The expansion operation which shall be discussed later might hamper the FIFO ordering. This ordering could be anyhow preserved by resorting to a per-node counter which is atomically incremented upon a push operation on the Treiber's stack.

Algorithm 4 Non-blocking DEQUEUE.

```
1: procedure DEQUEUE( )
2:     while true do
3:         oldCur ← current
4:         index ← oldCur >> 32
5:         min ← table[h1(index)][h2(index)]
6:         ⟨_, right⟩ ← SEARCH(min.t, min, <)
7:         newCur ← current
8:         if newCur ≠ oldCur ∧ (newCur >> 32) ≤ index then
9:             continue
10:        candidate ← right
11:        rNext ← right.next
12:        if cand ≠ tail then
13:            isNotMarked ← ¬ISMARKED(rightNext)
14:            casRes ← CAS(&cand.next, rNext, MARK(rNext))
15:            if isNotMarked ∧ casResult then
16:                return cand.event
17:            else
18:                continue
19:        index ← index + 1
20:        tmpSize ← dequeue_size
21:        if index = tmpSize then
22:            tmpFut ← future
23:            eSize ← tmpFut.enqueue_size
24:            if tmpSize = eSize ∧ tmpFut.next= null then
25:                return null
26:            if ¬EXPANDARRAY(tmpSize) then
27:                continue
28:        CAS(&current, oldCur, (index << 32) | ABAMARK())
```

Algorithm 5 The EXPANDARRAY routine.

```
1: procedure EXPANDARRAY(oldSize)
2:     tmpFut ← future
3:     if tmpFut.enqueue_size = oldSize then
4:         newBlock ← new array[oldSize]
5:         if ¬ CAS(&table[h1(oldSize)], null, newBlock) then
6:             release newBlock
7:         tmpTodo ← todo
8:         if tmpTodo.enqueue_size < oldSize then
9:             CAS(todo, tmpTodo, tmpFut)
10:        CAS(future, tmpFut, new future(2 · oldSize))
11:    tmpTodo ← todo
12:    tmpTodo.DENYPUSH()
13:    while UNMARK(todo.next) ≠ null do
14:        top ← UNMARK(todo.next)
15:        copy ← COPY(top)
16:        while true do
17:            if successful insertion of copy in table as invalid then
18:                break
19:            else if a replica n of top is found then
20:                RELEASE(copy)
21:                copy ← n
22:                break
23:        CAS(&top.replica, null, copy)
24:        if top.replica≠ copy then
25:            retry-loop to mark copy
26:        retry-loop to validate top.replica
27:        CAS(&todo.next, top, MARK(top.next))
28:    CAS(&dequeue_size, oldSize, 2 · oldSize)
29:    return oldSize < dequeue_size
```

The DEQUEUE operation (Algorithm 4) searches for the minimum in the bucket corresponding to `current`. This is done by resorting on the standard search offered by Harris' List, invoked as SEARCH(`bucket_width` · `current`, <). This ensures that the left node is always the head of the list, while the right node has the minimum key in the bucket. Once the minimum is identified, DEQUEUE restarts if `current` is concurrently updated by another thread, otherwise it tries to logically remove the right node by marking it with a `CAS`. If it succeeds, DEQUEUE completes and returns after searching again for the minimum in the bucket in order to remove marked nodes. However, the bucket might be empty and the right node might be `tail`. Now four cases are possible:

1. `current` < `dequeue_size` − 1;
2. `current` = `dequeue_size` − 1, while `future` is empty and its `enqueue_size` is equal to `dequeue_size`;
3. `current` = `dequeue_size` − 1, while `future` is non-empty and its `enqueue_size` is equal to `dequeue_size`;
4. `current` = `dequeue_size` − 1 and `future` is such that `enqueue_size` ≠ `dequeue_size`.

Case 1 implies that the next time interval is covered by `table`, thus the next bucket is the candidate to search for the minimum. This is achieved by incrementing `current` with a `CAS` and restarting the operation. Case 2 means that we have passed all the buckets in `table` which cover the interval [0, `dequeue_size` · `bucket_width`) and no items are stored in the overflow data structure associated with the time interval [`dequeue_size` · `bucket_width`, +∞), thus we can safely return `null` to indicate that the queue is empty. Case 3 occurs when `current` points to the last bucket in `table` and, at the same time, `future` is non-empty. It means that `table` can be expanded and every item in `future` can be inserted into appropriate buckets. We thus invoke the EXPANDARRAY routine, that returns true if `dequeue_size` is doubled at the ending of its invocation. If it returns false, then we restart from the beginning, otherwise we are sure that `table` covers the next time interval and we can increment `current` with a `CAS`. Independently of the result of this `CAS`, we restart from the beginning. Case 4 allows to detect whether an expansion is already occurring, thus EXPANDARRAY is executed in order to help in the expansion operation. Finally DEQUEUE completes only if it succeeds to mark a node or it meets the empty condition (case 2).

The EXPANDARRAY operation (Algorithm 5) has two goals: extending `table` and moving the elements from the overflow structure to `table`. The first step takes place only if `enqueue_size` associated with the current `future` is equal to `dequeue_size` of the invoking queue. Before moving nodes, we verify that the `enqueue_size` associated with `future` is equal to the `dequeue_size` read when the operation is started. This avoids that two expansions can occur simultaneously. The expansion of the array is performed by allocating the new required block and connecting it to the first level array with a single-shot `CAS`, because the old value is `null`. Before the stack could be emptied, we have to communicate to all threads that an expansion phase is taking place. This is done by making `todo` point to the current future with a single-shot `CAS`. The exchange is performed only if `enqueue_size`, stored in the stack currently pointed by `todo`, is strictly lower than `dequeue_size` seen when DEQUEUE is invoked. This avoids that `todo` is reset to its initial condition, while other threads are moving events. Now `table` is capable of storing items in an increased interval of time, but this information is not published yet. Thus we exchange `future` with a new stack such that its `enqueue_size` is equal to 2 · `dequeue_size`. Since the size of the array is always increasing, it is guaranteed that exactly one `CAS` can successfully update `future`, because new expansions cannot occur until `dequeue_size` is updated. When the new stack is installed, ENQUEUE sees a wider `table` than DEQUEUE. After that this condition is met, the stack referred by `todo` can be emptied with a sequence of consecutive pops, but we have to invoke DENYPUSH first. Preventing TRYPUSH from succeeding allows us to reach a stable condition in which the

stack is empty. In fact, we have ensured that a very slow thread cannot push an event after the stack is detected as empty. Finally, when this condition is verified, the current `dequeue_size` is doubled with a single-shot `CAS`, restoring the condition `enqueue_size` = `dequeue_size`.

During a pop operation, a thread-local copy c of the top node of the stack is inserted into `table`, flagged as to be validated. Then, the original copy's `replica` field is atomically updated with the address of c using a `CAS`. After that, the unique copy corresponding to `replica` is validated, and the original node can be removed from the stack. In this way we avoid that a slow thread can copy into `table` an event which had already been dequeued.

3.2.2 Garbage Collection

The problem of releasing memory buffers is non-trivial in the context of non-blocking data structures. In particular, since we rely on Harris's non-blocking list, the extraction of an event from the NBPQ initially entails marking it as no longer valid. But the actual unlink from the list and the free operation of the buffer need to be carried out subsequently. However, at any time interval, it is not trivial to discover whether a thread operating on event buffers is still keeping a valid reference to the corresponding memory areas (e.g. for list traversing).

To cope with this issue, we have adopted the following strategy to realize a garbage collector which can be used to safely reclaim event buffers. Whenever an event is extracted from the queue, it is connected to a thread-private list, so that a reference to its memory buffer is never lost. Contextually, we expose an ad-hoc PRUNE(BOUND) API which accepts a timestamp value as its argument. It is therefore the responsibility of the overlying software level to properly invoke the PRUNE() function, whenever it determines that a set of events up to a certain timestamp value are safe to be deleted (since the threads within the simulation engine are working on successive buckets). PRUNE() simply scans the list of disconnected events `free()`'ing buffers associated with a timestamp $T < \left\lfloor \frac{bound}{\texttt{bucket\_width}} \right\rfloor \cdot \texttt{bucket\_width}$. This can be safely done without any synchronization, due to the thread-separate nature of the recollection lists where the event buffers are placed after extraction. Although this way of reclaiming memory requires an explicit interaction with the software exploiting the NBPQ, this is somehow coherent with the nature of simulation systems relying on pending event sets, e.g., because of periodical revaluation of the commitment horizon of the whole simulation.

4. EXPERIMENTAL RESULTS

We tested our proposal using a multi-thread program in which each thread repeatedly enqueues or dequeues an event with a given probability P_E and P_D respectively, such that $P_E + P_D = 1$. Each thread maintains a local variable `local_time` representing the timestamp of the last dequeued event. We tested with three different probability distributions of the timestamp increment (starting from `local_time`) of newly-scheduled events—uniform, triangular and exponential. Additionally, for each distribution we considered different mean inter-arrival time values $\mathbb{E}[T] \in \{1, 10, 50\}$. Table 1 provides the definition of these distributions. Relying on different distributions allowed us to assess our proposal when considering different patterns of the binding between event timestamps and NBPQ buckets. Also, continuous access by the threads to the event pool leads our tests to represent stress cases where the scalability (and performance) of some overlying multi-thread PDES engine would be essentially limited by the scalability while managing the fully-shared event pool. In other words we make our tests independent of any strategy (e.g. conservative vs optimistic) used to actually process the events and to keep causality across the simulation objects. Further, we make our tests not biased in terms of reduction of contention in the access to the shared pool depending on the granularity of the event-processing routine—the routine we run upon the extraction of an event from the pool only entails the generation of other events to be immediately injected in the pool.

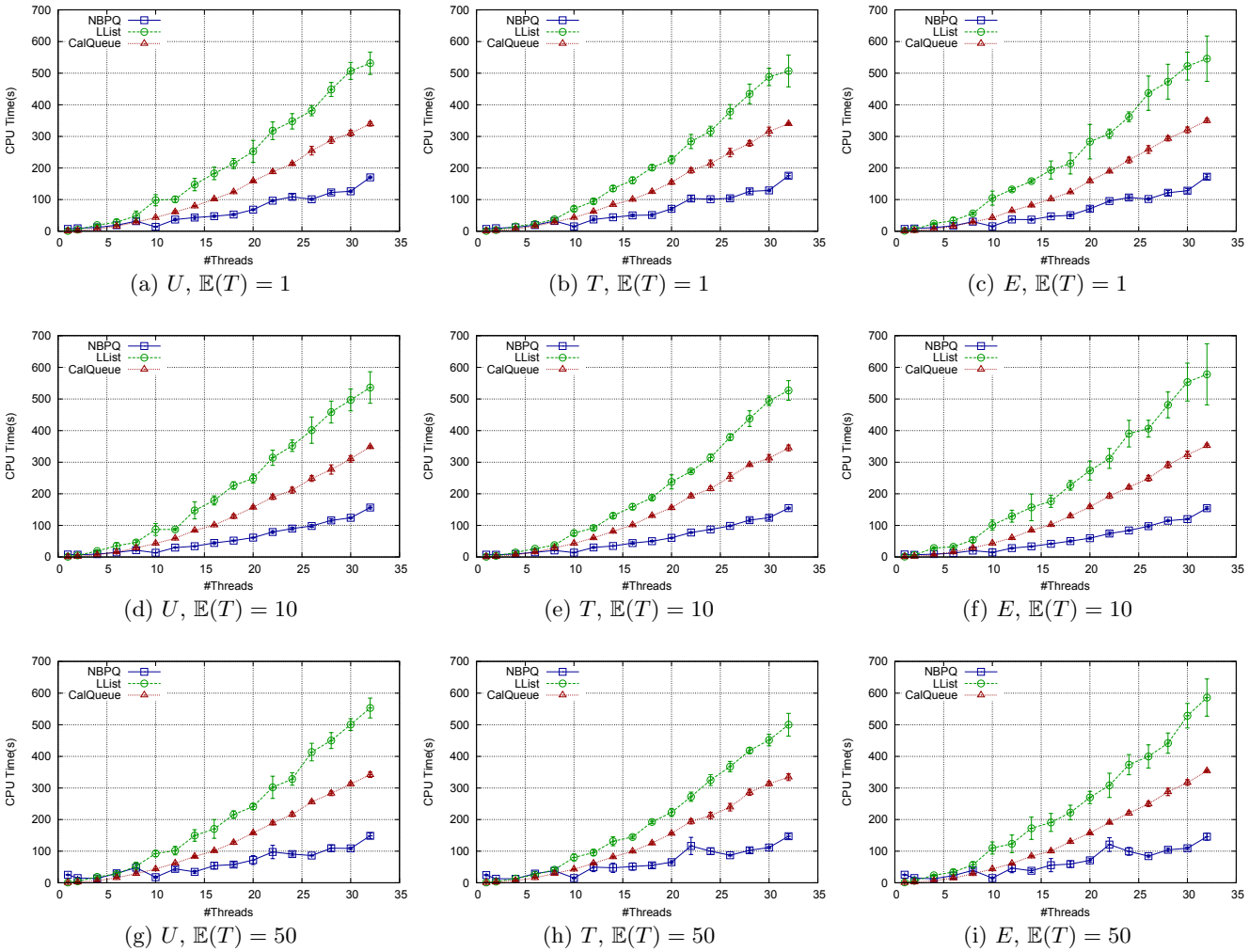
PROBABILITY DISTRIBUTION	FORMULA
Uniform (U)	$2\mathbb{E}[T]\cdot\texttt{rand}$
Triangular (T)	$\frac{3\mathbb{E}[T]}{2} \cdot \sqrt{\texttt{rand}}$
Exponential (E)	$-\mathbb{E}[T] \cdot \ln(\texttt{rand})$

Table 1: Probability distributions used in evaluation tests and their formula given the mean value $\mathbb{E}[T]$.

All the tests have been run on a 32-core HP ProLiant machine running Linux (kernel 2.6) equipped with 64 GB of RAM. The number of threads running the test-bed program has been varied from 1 to 32. Each test ends when the total number of performed operations (which is independent of the number of employed threads) reaches a given threshold $\#OPS = 1280000$. This guarantees that for every thread count we worked with the highest concurrency degree (no thread is ever switched off before the ending condition is reached). Our NBPQ is initialized with a `bucket_width` equal to 1 and `init_size` is set to 32768. The PRUNE routine (which garbage collects obsolete buckets) is invoked periodically every 5000 completed operations. We also run the same experiments with a single ordered linked-list (LList) and a traditional Calendar Queue (CalQueue), both protected via a classical global spinlock to guarantee atomicity of concurrent manipulations. In these configurations, the invocation of the memory allocator for acquiring new event buffers is placed outside of the lock-based critical section, thus the critical section only entails actual data structure manipulations. CalQueue is set up with a maximum number of buckets equal to 32768.

In each run we sampled the $CPUtime$ parameter, that is the sum of time spent by each thread both in user mode and kernel mode. Since the employed machine was executing only our test-bed multi-thread program (and the operating system), $CPUtime$ captures the fact that in a blocking algorithm (say the tested global lock based approaches to manage the event pool) a thread actively waits in order to eventually acquire the spinlock. The throughput has been then evaluated as $\frac{\#OPS}{CPUtime}$.

In a first test settings (test 1) we configured an equal probability for ENQUEUE and DEQUEUE operations, say $P_D = P_E = 0.5$. Although the expected average size of the queue is zero, it is not empty all the time, allowing us to capture the base cost of our algorithm and the resilience to performance degradation potentially caused by inadequate bucket width (or the gain in case of adequate sizing). In Figure 2 we show the measured $CPUtime$ values for all the



(a) U, $\mathbb{E}(T) = 1$ (b) T, $\mathbb{E}(T) = 1$ (c) E, $\mathbb{E}(T) = 1$

(d) U, $\mathbb{E}(T) = 10$ (e) T, $\mathbb{E}(T) = 10$ (f) E, $\mathbb{E}(T) = 10$

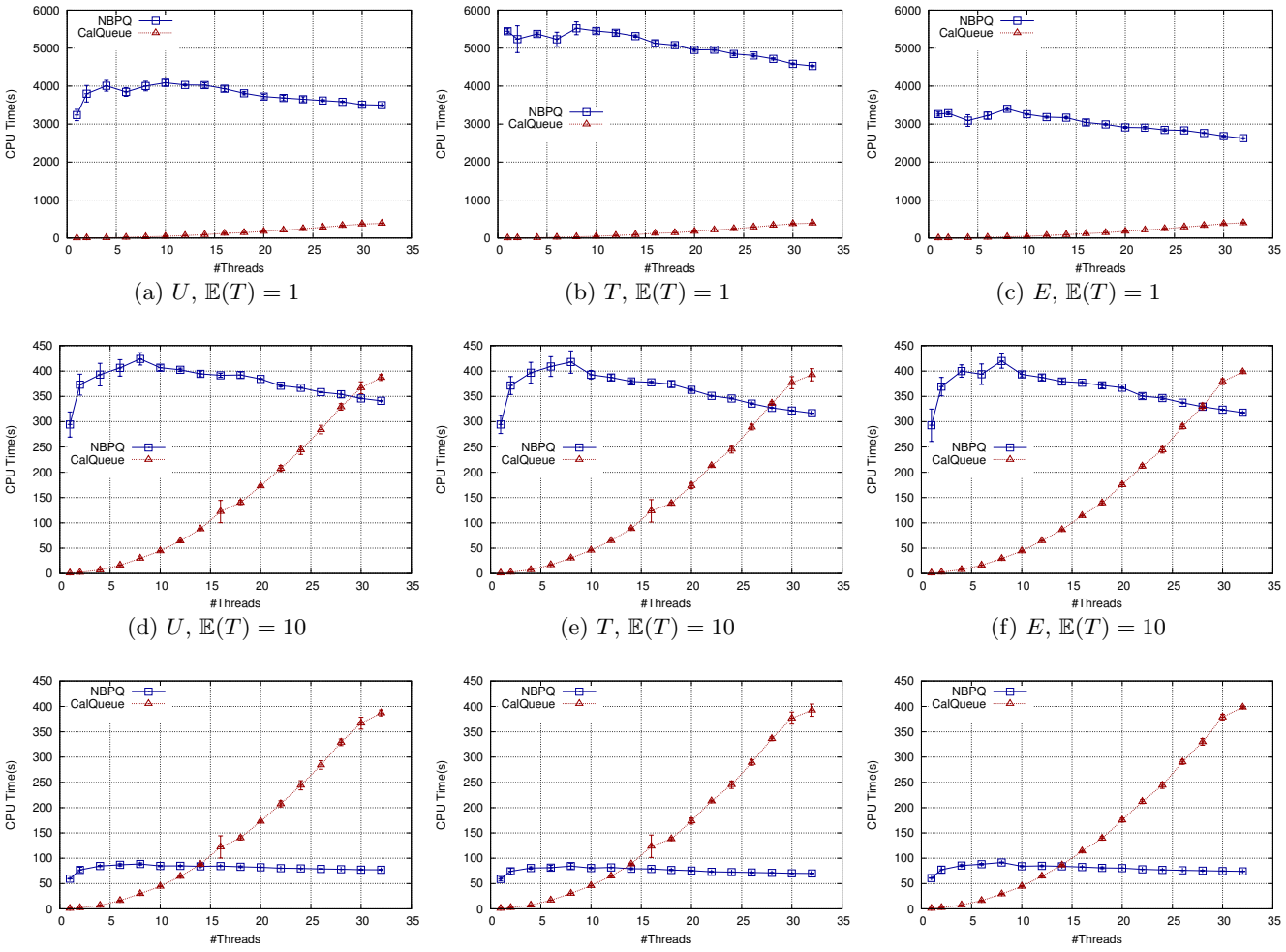
(g) U, $\mathbb{E}(T) = 50$ (h) T, $\mathbb{E}(T) = 50$ (i) E, $\mathbb{E}(T) = 50$

Figure 2: CPU times for test 1.

tested configurations of timestamp increment distributions and their mean values. LList and CalQueue both appear not to be affected by the selected settings, which is due to the fact that the queue contains no more than few elements at any time. Thus, the computation steps performed by LList for managing the event records are a few, while CalQueue is slightly more efficient thanks to its indexed access that spreads events in multiple buckets, reducing the number of items to be traversed per insertion. This leads to advantages over LList for large thread counts, since the global lock protected critical section lasts less time. However, the main advantages in performance (thanks to better scalability) are achieved exactly with the employment of non-blocking concurrent accesses as provided by NBPQ. Overall, our algorithm performs better than the alternatives because ENQUEUE operations are both fast and non-blocking, while the number of empty buckets is non-critical for DEQUEUE operations.

We note that detecting the queue emptiness is an operation somehow expensive in our solution, since it requires the scan of the whole `table`. This explains why our algorithm shows similar performance in all the tested configurations. In order to verify this hypothesis, we have repeated the tests with 32 threads and `init_size` equal to 2. This ensures that the `current` index is close to `table`'s size most of the time, thus detecting an empty queue requires fewer buckets to be traversed. This allowed achieving a speed-up ranging from 1.3x to 1.6x. The improvement is not larger since, with this settings, we moved contention from `current` to the Treiber's Stack pointed by `future` or `todo`. A solution to detect the queue emptiness more efficiently could be to use a counter updated with a `fetch&add` instruction, a strategy we plan to explore as future work.

In a second test settings (test 2) we pre-populated the event set and compared the steady state behavior of the global lock-protected CalQueue with the one of NBPQ. As discussed in [20], for the considered timestamp increment distributions, the steady state access time is reached after a number of operations that is five times the queue size. Thus we imposed $P_D = 0.3$ and $P_E = 0.7$ for the 30% initial part of execution of the test, achieving an expected queue size of about 153600, and $P_D = P_E = 0.5$ for the remaining 760000 operations. In Figure 3 we show how the performance of CalQueue is still not relevantly affected by the used timestamp increment distributions (images from left to right) and their respective mean values (from top to



(a) U, $\mathbb{E}(T) = 1$ (b) T, $\mathbb{E}(T) = 1$ (c) E, $\mathbb{E}(T) = 1$

(d) U, $\mathbb{E}(T) = 10$ (e) T, $\mathbb{E}(T) = 10$ (f) E, $\mathbb{E}(T) = 10$

(g) U, $\mathbb{E}(T) = 50$ (h) T, $\mathbb{E}(T) = 50$ (i) E, $\mathbb{E}(T) = 50$

Figure 3: CPU times for test 2.

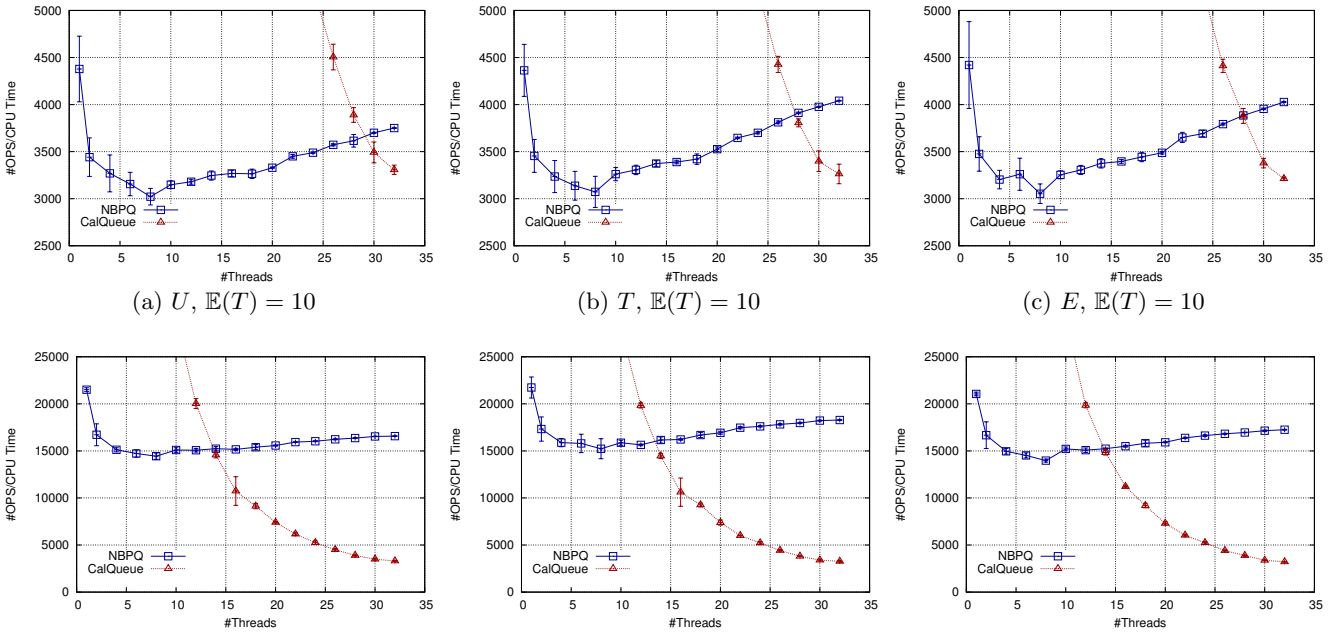
$\mathbb{E}[T]$	$\frac{BucketWidth_{NBPQ}}{BucketWidth_{CQ}}$
1	≈ 15000
10	≈ 1200
50	≈ 200

Table 2: Relative bucket size values.

bottom), due to its capability to resize the array and the bucket width, a capability not yet offered (in non-blocking fashion) by our proposal. In fact this leads to an access time that is constant with respect to the queue size.

NBPQ shows degraded performance when running with with $\mathbb{E}[T] = 1$ (Figure 3(a), 3(b) and 3(c)), since we did not rely on bucket resize in this experiment, and a single bucket becomes the container of half of the pending events, leading to spending 99.9% of the execution time in the ENQUEUE operation. However, a ten times smaller bucket width (Figures 3(d), 3(e) and 3(f)) leads to an execution time that is comparable with the one offered by CalQueue, while a fifty times finer grain bucket makes our non-blocking algorithm definitely outperform the rival in all the test cases (Figures 3(g), 3(h) and 3(i)). These results prove that, even if non-blocking scaling of the bucket size is not yet supported by our approach, running the scaling within a critical section would still lead to a big potential for higher scalability of our solution, with respect to the global lock-protected CalQueue. Also, the flat behavior of the response time of NBPQ (even with suboptimal bucket size) is a clear indication of its potential for higher scalability (say at thread counts much larger that 32) compared to CalQueue.

In Table 2 we show the average ratio between the bucket width of our NBPQ proposal ($BucketWidth_{NBPQ}$) and the bucket width of CalQueue at steady state ($BucketWidth_{CQ}$), just depending on its dynamical resize. A value of this ratio close to 1 means that the statically-selected bucket size used in NBPQ is close to the well suited value dynamically recomputed by the CalQueue algorithm. When this ratio is about 1000:1, the performance of the two queueing data structures under high contention (larger thread counts) is comparable. This is an interesting result, still indicating that, in order to achieve scalability with a non-blocking queue based on calendars, it is not mandatory to guess an optimal width of the bucket. In fact, we only need that buckets are short enough



(a) U, $\mathbb{E}(T) = 10$ (b) T, $\mathbb{E}(T) = 10$ (c) E, $\mathbb{E}(T) = 10$

(d) U, $\mathbb{E}(T) = 50$ (e) T, $\mathbb{E}(T) = 50$ (f) E, $\mathbb{E}(T) = 50$

Figure 4: Thread throughput for test 2.

to guarantee fast ENQUEUE operations. However, given that we use non-blocking lists, these can be up to 1000 times longer than the counterpart blocking alternatives.

Still reasoning about the ability of NBPQ to efficiently handle longer lists in the buckets, let us assume to have p threads in the system. When one thread is accessing the global lock-protected CalQueue for a time interval k, the other $p-1$ threads could be waiting for the access the same amount of time k. Thus the average access time of each thread is $O(p \cdot k)$. Ideally, in order to provide a non-blocking priority queue as fast as the blocking CalQueue we can spend up to $O(p \cdot k)$ time per operation, thus we could choose a bucket p times wider than the optimal size. However, given that our experiments have been run with up to 32 threads, the ideal ratio among the buckets size in the two scenarios (blocking vs non-blocking) is significantly lower than the one we measured. The reason for this discrepancy stands in the usage of spinlocks, which have been shown to make threads in a critical section run slower by a factor that depends on actual contention [1]. Hence, when we have $p-1$ threads that are spinning, the thread holding the lock runs $O(p)$ times slower. It means that a thread completes an operation after $O(p^2 k)$ time. Thus to reach a comparable execution time with p threads, our non-blocking algorithm can run up to $O(p^2)$ times slower, which is achieved just when processing many more elements in the ENQUEUE operation.

In other words increasing the per-thread work for ENQUEUE operations is not a dramatic choice provided that we can handle more ENQUEUE operations at a time and these are in buckets that do not contain the minimum. DEQUEUE operations are not significantly affected by the bucket width, which only determines the length of the scan to find the next non-empty bucket. Indeed, a DEQUEUE from a non-empty bucket consists in trying a `CAS` on the first unmarked node, but in case of an unsuccessful exchange we can restart from the beginning of the bucket. On the contrary, the search for a non-empty bucket takes advantage from an increased width, because it allows DEQUEUE operations to scan a lower number of buckets. Moreover, even if NBPQ executes as fast as CalQueue, it still takes advantage from the absence of adverse effects caused by spinlocking. In fact, as shown in Figures 4(a), 4(b) and 4(c), the per-thread throughput is increased, still suggesting how NBPQ could exploit more processors if available. On the contrary, a bucket 200 times larger than the optimum makes the throughput of each thread stable (see Figures 4(d), 4(e) and 4(f)), showing that our algorithm is working at its maximum capability.

5. CONCLUSIONS AND FUTURE WORK

In this paper we have presented NBPQ, a non-blocking implementation of a priority queue for the pending event set, explicitly targeting PDES systems organized according to the multi-threaded paradigm and where the workload distribution among the threads is (fully) driven by the share everything model. Our proposal allows multiple concurrent threads to perform enqueue/dequeue operations on the pending event set without the need for using locking primitives. From our experimental assessment, our proposal looks promising under various execution (i.e., event generation/extraction) patterns and for different scaling levels.

Future work is planned along multiple directions. On the one hand, as mentioned, we plan to devise a more efficient way to detect the queue emptiness, by relying on the atomic `fetch&add` primitive. On the other hand, we plan to extensively test our implementation on real environments, such as the PDES platforms in [7, 16, 21] oriented to workload share across worker threads, and with real world applications. Finally, we plan to design a non-blocking technique to allow the resize of the bucket width at runtime, thus en-

abling its dynamical adaptation to variations in the events' timestamps generation pattern. This would lead to a priority queue that is scalable (an already achieved target) and which is able to provide optimized performance in face of different workloads.

Acknowledgements

Mauro Ianni and Alessandro Pellegrini are also working with Value Up S.r.l., an InResLab partner. This work is partially supported by the WFDA: "Wait-Free Synchronization Algorithms in High Performance Data Processing on Multi-Core Environments" project funded with the support of MISE research funds.

6. REFERENCES

[1] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990.

[2] R. Ayani. LR-Algorithm: concurrent operations on priority queues. In *Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing*, SPDP, pages 22–25, Dallas, TX, USA, 1990. IEEE Computer Society.

[3] A. Boukerche and S. K. Das. Dynamic load balancing strategies for conservative parallel simulations. In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, PADS, pages 20–28, 1997.

[4] R. Brown. Calendar queues: a fast O(1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10):1220–1227, 1988.

[5] C. D. Carothers and R. M. Fujimoto. Efficient execution of Time Warp programs on heterogeneous, NOW platforms. *IEEE Transactions on Parallel and Distributed Systems*, 11(3):299–317, 2000.

[6] K. M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, SE–S5(5):440–452, 1979.

[7] D. Cingolani, A. Pellegrini, and F. Quaglia. RAMSES: Reversibility-based agent modeling and simulation environment with speculation support. In S. Hunold, A. Costan, D. Ginenéz, A. Iosup, L. Ricci, M. E. Gómez Requena, V. Scarano, A. L. Varbanescu, S. L. Scott, S. Lankes, J. Weidendorfer, and M. Alexander, editors, *Proceedings of Euro-Par 2015: Parallel Processing Workshops*, PADABS, pages 466–478. LNCS, Springer-Verlag, 2015.

[8] T. Dickman, S. Gupta, and P. A. Wilsey. Event pool structures for PDES on many-core Beowulf clusters. In *Proceedings of the 2013 ACM/SIGSIM Conference on Principles of Advanced Discrete Simulation*, pages 103–114. ACM Press, 2013.

[9] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, 1990.

[10] D. W. Glazer and C. Tropper. On process migration and load balancing in Time Warp. *IEEE Transactions on Parallel and Distributed Systems*, 4(3):318–327, 1993.

[11] S. Gupta and P. A. Wilsey. Lock-free pending event set management in time warp. In *Proceedings of the 2014 ACM SIGSIM conference on Principles of advanced discrete simulation*, PADS, pages 15–26. ACM Press, 2014.

[12] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In J. Welch, editor, *Proceedings of the 15th International Conference on Distributed Computing*, volume 2180 of *DISC*, pages 300–314. Springer Berlin/Heidelberg, 2001.

[13] J. Hay and P. A. Wilsey. Experiments with hardware-based transactional memory in parallel simulation. In *Proceedings of the 2015 ACM/SIGSIM Conference on Principles of Advanced Discrete Simulation*, PADS, pages 75–86, New York, New York, USA, 2015. ACM Press.

[14] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[15] D. R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and System*, 7(3):404–425, 1985.

[16] D. E. Martin, T. J. McBrayer, and P. A. Wilsey. WARPED: A Time Warp simulation kernel for analysis and application development. In *Proceedings of the 29th Hawaii International Conference on System Sciences. Volume 1: Software Technology and Architecture*, HICSS, pages 383–386. IEEE Computer Society, 1996.

[17] A. Pellegrini, S. Peluso, F. Quaglia, and R. Vitali. Transparent speculative parallelization of discrete event simulation applications using global variables. *International Journal of Parallel Programming*, apr 2016.

[18] S. Peluso, D. Didona, and F. Quaglia. Supports for transparent object-migration in PDES systems. *Journal of Simulation*, 6(4):279–293, nov 2012.

[19] F. Quaglia. A low-overhead constant-time lowest-timestamp-first CPU scheduler for high-performance optimistic simulation platforms. *Simulation Modelling Practice and Theory*, 53:103–122, 2015.

[20] R. Rönngren and R. Ayani. A comparative study of parallel and sequential priority queue algorithms. *Transactions on Modeling and Computer Simulation*, 7(2):157–209, 1997.

[21] E. Santini, M. Ianni, A. Pellegrini, and F. Quaglia. Hardware-transactional-memory based speculative parallel discrete event simulation of very fine grain models. In *2015 IEEE 22nd International Conference on High Performance Computing*, HiPC, pages 145–154. IEEE, dec 2015.

[22] H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. In *Proceedings of the 2003 International Parallel and Distributed Processing Symposium*, volume 65 of *IPDPS*, page 11. IEEE Comput. Soc, 2005.

[23] W. T. Tang, R. S. M. Goh, and I. L. Thng. Ladder queue: An O(1) priority queue structure for large-scale discrete event simulation. *Transactions on Modeling and Computer Simulation*, 15(3):175–204, 2005.

[24] R. K. Treiber. Systems programming: coping with parallelism. Technical report, IBM US Research Centers, Yorktown, San Jose, Almaden, US, 1986.