Concurrent Lock-Free Unbounded Priority Queue with Mutable Priorities

Ivan Walulya[1(✉)], Bapi Chatterjee[2], Ajoy K. Datta[3], Rashmi Niyolia[3], and Philippas Tsigas[1]

[1] Department of CS&E, Chalmers University of Technology, Gothenburg, Sweden
{ivanw,tsigas}@chalmers.se
[2] IBM Research Lab, New Delhi, India
bhaskerchatterjee@gmail.com
[3] Department of CS, University of Nevada Las Vegas, Las Vegas, USA
Ajoy.Datta@unlv.edu, rashmi.niyolia@gmail.com

Abstract. The priority queue with DeleteMin and Insert operations is a classical interface for ordering items associated with priorities. Some important algorithms, such as Dijkstra's single-source-shortest-path, Adaptive Huffman Trees, etc. also require changing the priorities of items in the runtime. Existing lock-free priority queues do not directly support the dynamic mutation of the priorities. This paper presents the first concurrent lock-free unbounded binary heap that implements a priority queue with mutable priorities. The operations are provably linearizable. We also designed an optimized version of the algorithm by combining the concurrent operations that substantially improves the performance. For experimental evaluation, we implemented the algorithm in both C/C++ and Java. A number of micro-benchmarks show that our algorithm performs well in comparison to existing implementations.

Keywords: Heap · Lock-free · Linearizability · Concurrent heap
Priority-queue · Elimination

1 Introduction

A priority queue orders a set of items by a numerical cost – often called *priority* – associated with each item. In its most general form, a priority queue abstract data type (ADT) is defined by two operations – Insert and DeleteMin. An Insert $(k, elem)$ inserts an item $elem$ with priority k and a DeleteMin () removes an item with the highest priority from the set of objects. Priority queues are widely used at operating system kernels as well as in user-space. Some well-known applications are discrete event simulations [10], graph search [20], operating systems schedulers [13], SAT solvers [5] and many others. Several of them, such as Dijkstra's single-source-shortest-path (SSSP) algorithm [7], Adaptive

T. Izumi and P. Kuznetsov (Eds.): SSS 2018, LNCS 11201, pp. 365–380, 2018.
https://doi.org/10.1007/978-3-030-03232-6_24

Huffman Trees [25], etc. require updating the priorities after inserting the items. In today's application settings, the underlying datasets grow immensely at runtime necessitating the employed data structure to be adaptable to size variations.

At the same time, the proliferation of multi-core systems have essentially mainstreamed the concurrent data structures. Concurrent data structure designs are evaluated on consistency (correctness) and progress guarantees in addition to scalability with increasing number of processing threads. The most common consistency framework used in concurrent settings is linearizability [16], which relates a concurrent execution on an object to its sequential specification. Linearizability requires that an operation appears to take effect instantaneously at a single linearization point between the operation's invocation and its response.

Consistency may be trivially achieved using mutual exclusion locks that serialize the access to the entire data structure, also called coarse-grained locking. However, it severely limits the concurrent operations. Even if the number of locks increase, i.e. fine-grained locking, they are still vulnerable to pitfalls such as deadlock, priority inversion and convoying. An alternative approach is lock-free implementation. In a lock-free concurrent data structure, at least one non-faulty processing thread is guaranteed to complete its operation in a finite number of steps. Effectively, lock-free data structures foster both scalability and progress guarantee. A stronger progress guarantee is wait-freedom, which ensures that all the non-faulty processes finish their operations in a finite number of steps. However, most often wait-freedom results in poor performance. Another approach to implement consistent concurrent data structure is using software transactional memory (STM) [22]. However, the performance of such implementations largely depends on the design of the STM. Unsurprisingly, using STM to design concurrent data structures has often resulted in unacceptable performance [8].

Thus, an efficient and scalable *unbounded* concurrent lock-free data structure implementing a *mutable priority queue*, i.e. one which offers updating priorities of items dynamically, is highly sought-after in a large number of applications.

Based on the employed data structure, a priority queue implementation can be categorized primarily as: (a) heap[1]-based, and (b) skip-list-based.

The previous attempts on heap-based concurrent priority queues have largely been blocking (lock-based) or impractical non-blocking designs. Hunt et al. [17] presented a fine-grained lock-based heap, which locks each node separately and operations release and re-acquire locks after each step in bubble-up to prevent deadlocks with concurrent bubble-down operations. Tamir et al. [24] extended the work of [17] by including operations, called ChangeKey, to update the priority of items. The focus of their work is on the ChangeKey operations, which they show that improves the overall performance of Dijkstra's SSSP algorithm.

The first attempt to implement a non-blocking concurrent heap was by Herlihy [15]. However, this wait-free algorithm required copying the entire heap making the implementation inherently sequential and of little practical interest. Barnes [3] proposed a wait-free algorithm to address the drawbacks of Herlihy. His definition of the wait-free property is different from the generally accepted

[1] In this work, by a heap we mean a binary heap.

definition. Additionally, no implementation of this algorithm exits. Israeli *et al.* presented a wait-free algorithm for heap-based priority queues [18] which utilizes atomic primitives[2] that are not implemented in existing hardware platforms.

Dragicive et al. [8] designed a lock-free heap that uses STM for concurrency control. Their design offered poor performance due to the overhead of the STM. We point out that all the previously available concurrent heaps are bounded to a fixed size allocated at the initialization. There are available works on skip-list-based concurrent priority queues – Shavit et al. [21], Tsigas *et al.* [23], etc. Alistarh *et al.* [1] proposed an approximate DeleteMin operation in skip-lists. However, the skip-list-based implementations face difficulty to implement the algorithms that require mutable priorities at the runtime: observably, the overall performance of the algorithm degrades [24].

We present CoMPiQ - a **Co**ncurrent lock-free unbounded heap-based **M**utable **Pri**ority **Q**ueue. The Table 1 summarily contrasts our contributions with the relevant existing works.

Table 1. Concurrent priority queues

Paper	Data Structure	Progress Guarantee	Mutable Priority	Unbound	Practical Implementation
Herlihy [15]	Heap	Wait-free	No	No	No
Hunt et al. [17]	Heap	Lock-based	No	No	Scales poorly
Shavit *et al.* [21]	Skip-list	Lock-free	No	Yes	Yes
Tsigas *et al.* [23]	Skip-list	Lock-free	No	Yes	Yes
Dragicive et al. [8]	Heap	Lock-free	No	No	Scales poorly
Tamir *et al.* [24]	Heap	Lock-based	Yes	No	Yes
CoMPiQ	Heap	Lock-free	Yes	Yes	Yes

In the paper, first we present the system model and the sequential specification of the heap data structure (Sect. 2). Then, we describe the lock-free design of the heap in detail (Sect. 3). We present the proof of linearizability and lock-freedom of the concurrent operations (Sect. 4). We implemented the algorithm in both C/C++ and Java. We describe the micro-benchmarks that we used to evaluate the algorithm, wherein we also discuss the performance with respect to the design optimizations. Our experiments demonstrate that the presented algorithm performs well in comparison to the existing counterparts (Sect. 5).

2 Preliminaries

We consider an asynchronous shared memory system with a finite set of n *processing threads* $p_1, ..., p_n$ where n may exceed the number of physical processors. In addition to the atomic `read` and `write` instructions, the system supports *Compare-And-Swap* (`CAS`) atomic read-modify-write instructions. The

[2] SC2 which validates and writes to two disjoint memory locations atomically.

CAS($address, old, new$) instruction checks if the current value at a memory location ($address$) is equivalent to the given value old, and only if true, changes the value of $address$ to the new value (new) and returns TRUE; otherwise the memory location remains unchanged and the instruction returns FALSE.

The ADT *mutable priority queue* is defined by the following operations:

- Insert$(k, elem)$: An Insert$(k, elem)$ inserts an item $elem$ with priority k to the heap. We assume that k belongs to a totally ordered set. Insert is typically a void procedure, however, we return a cross-reference to the insert item instance which can be used in the ChangeKey procedure[3]. In case there is an item $elem'$ available in the heap with the same priority k, the item $elem$ gets inserted and the two items $elem$ and $elem'$ can have arbitrary order by their indexes. Thus, the heap allows items with duplicate priority.
- DeleteMin(): A DeleteMin removes an item with highest priority from the heap and returns that item itself. DeleteMin returns a special item EMPTY making no changes in the heap, if there are no items in the heap.
- ChangeKey(it, k_2): A ChangeKey(it, k_2) changes the heap so that an item $elem$ referenced by the iterator it, if existing in the heap, is placed at the priority k_2. It returns EMPTY if the item referenced by it was deleted from the priority queue.

In our work, a *heap* data structure implements a mutable priority queue. A heap is implemented by way of a *resizable array*. Thus, it contains items that allow for random access using a non-negative index. The array is considered virtually divided in *levels*. In the array, the *root* of the heap occupies the index 1 and is considered to be at the level 0. The *left* and the *right* children of the item at the index i are at the indexes $2i$ and $2i + 1$, respectively. We have considered a *minheap*, which means that the heap maintains the following *heap property*.

Heap Property: An item $elem_1$ with priority k_1 has higher priority than the item $elem_2$ with priority k_2, if $k_1 \leq k_2$. Thus, a parent always has a *smaller* priority compared to its children and the root has the highest priority. Moreover, no item exists at level l unless the level $l - 1$ is completely full.

To demonstrate the correctness of our concurrent heap design we verify the safety and liveness properties. The safety property that we use is *linearizability* [16], whereas, the liveness is proved as *lock-freedom* [14].

Lock-free Implementations utilizing CAS are prone to the ABA problem [19]: a thread P reads a value A from a shared memory location, a concurrent thread $\hat{P}$ changes the value to B and then $\hat{P}$ or another thread changes it back A; when P executes a CAS instruction on the location, it succeeds erroneously as if the location has not been changed since last read by P. Several memory management solutions have been proposed to address the ABA problem [11,19]. For ease of exposition, we assume the availability of a non-blocking memory management and garbage collection.

[3] In our implementation, the Insert operations never returns a *null* or fails to make any change due to the reason of finding the heap *full*. The heap is never full as long as we have sufficient system memory available.

3 Algorithm

Our heap implementation utilizes the lock-free dynamic resizable arrays [6] as the underlying container, which offers both unbounded storage and lock-free progress guarantees. The ADT operations consist of a series of steps, such as modifying the size and then appending an item to the heap, or swapping the item at the root with the item at the bottom, or for that matter swapping any two items in case of a ChangeKey, followed by restoring the heap property. Each step comprises of at least one atomic primitive execution over a shared memory word. The procedures HeapifyUp and HeapifyDown restore the heap property.

In order to achieve lock-free synchronization on concurrent access, we apply the *cooperative technique* described by Barnes [2]. The main idea is to detach operations from the executing threads. A thread that wishes to execute an operation on a slot of the array, creates a description of the work that it needs to perform, and writes the descriptor on the slot: we call it *marking* the slot. The operation can be completed by any thread that encounters the descriptor, which comes handy to ensure lock-freedom if the thread that initiated the operation is delayed or crashes.

Please note that *marking is not locking* a slot. It can be thought as shutting the door of a slot after putting down the description of all that is to be done inside. Thus any concurrent thread instead of busy waiting at the door actually carries the description with itself and tries to finish the work initiated by another thread in case that thread could not finish in time.

In our design, we maintain a *global descriptor* which encapsulates the current size of the heap and allows atomic modification of the size value and the associated heap slots with a sequence of `CAS` instructions. Additionally, we use descriptor objects at the slots during HeapifyUp and HeapifyDown calls. The threads calling HeapifyUp or HeapifyDown synchronize by way of executing `CAS` at these descriptors.

```
1: type Heap {
2:     Slot *vdata[][];
3:     Info *hdescr;
4: }
5: OpType {HPUP, HPDOWN};
                          ▷ Heap initialized
6: Heap* heap ← ⟨vdata, ⟨1, null⟩⟩
```

(a)

```
1: type Info {
2:     bool pending;
3:     size_t size;
4:     size_t pos;
5:     OpType op;
6:     Elem *old, *new;
7:     Info *lup, *rup;
8: }
```

(b)

```
1: type Slot {
2:     Elem *elem;
3:     Info *info;
4: }
5: struct Elem {
6:     value_t key;
7:     T *item
8: }
```

(c)

Fig. 1. Type definitions for the heap structure, descriptors and initialization.

Data types and heap initialization are given in Fig. 1. The *Heap* structure holds pointers to the data storage arrays and a descriptor object, Fig. 1a - line 1 to 4. A descriptor object, Fig. 1b, maintains information about the state including the current size of the heap. Therefore, we initialize the heap with a dummy

descriptor object with size 1, Fig. 1a - line 6. To store auxiliary data with the priorities, our design maintains the heap as an array of pointers to item nodes. Each slot in the heap has a pointer *elem* to an Elem and a pointer $info$ to an Info object which records the *state* of the slot: *stable* or *transient* due to an update, Fig. 1c. An Info descriptor stores enough information, such that a thread encountering a slot in a transient state can help advance the operation.

3.1 Lock-Free ADT Operations

The mutable priority queue operations in the lock-free heap are shown by flow-charts in Figs. 2 and 4. The main procedures called by these operations are shown in Figs. 3, 5 and 6. The pseudo-codes of each of the operations, their subroutines, and detail descriptions thereof are presented in the extended version of the paper [26]. For ease of exposition, the flow-chart based presentation of the algorithm is recursive. However, our implementation is fully non recursive as presented in the pseudo-code in the [26].

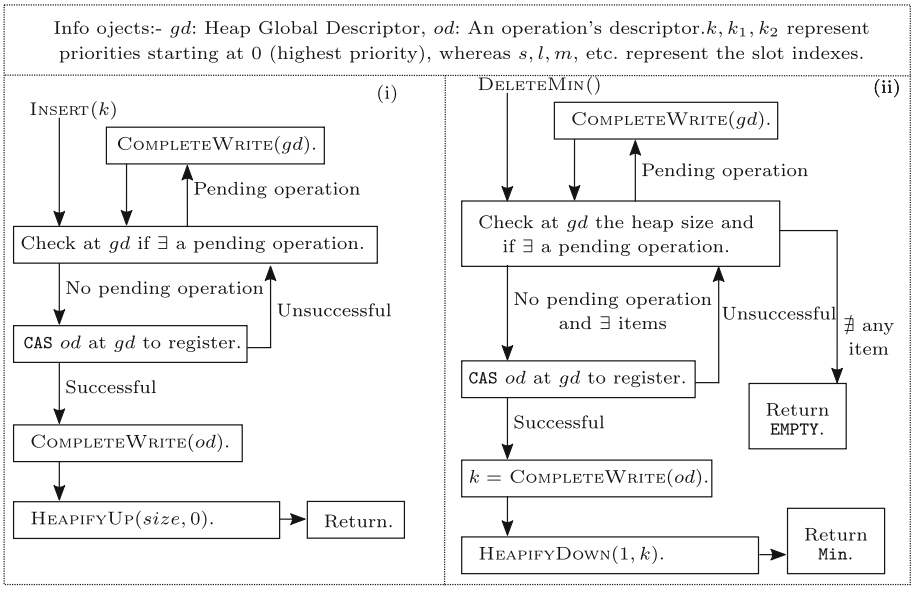

Fig. 2. Insert and DeleteMin operations in CoMPiQ.

The Insert and DeleteMin operations, Fig. 2(i) and (ii), start with an attempt to modify the size of the heap, this is achieved by *registering* the operation by way of executing a `CAS` to write its descriptor at the heap's global descriptor. That initiates the *preliminary phase* of the operation. The registered operation is considered pending until it is ready to call the procedures for

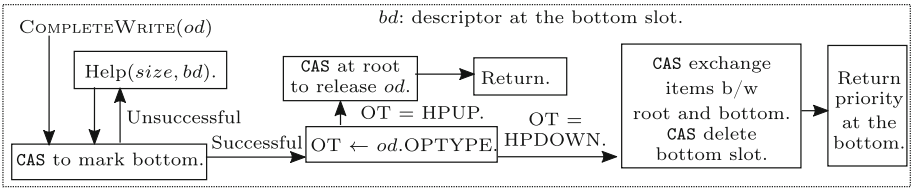

Fig. 3. CompleteWrite procedure.

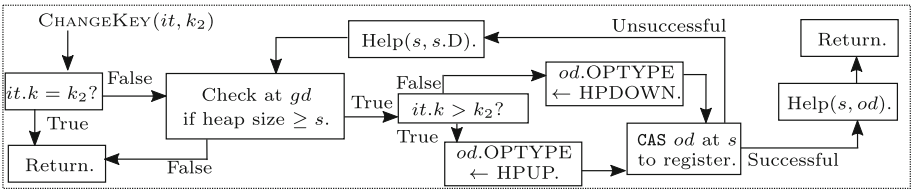

Fig. 4. ChangeKey operation in CoMPiQ.

restoring the heap property. The threads that encounter this operation, can help complete the preliminary phase.

The steps to complete the preliminary phase are taken in the procedure CompleteWrite, see Fig. 3. CompleteWrite first fixes the bottom of the heap and then depending upon the type of restoration required: HPUP or HPDOWN, release the root or bottom. This procedure helps in scaling the method because it releases one end of the heap as soon as the preliminary phase is completed. In case of DeleteMin operation calling CompleteWrite, it returns the priority of the bottom-most item in the heap.

A ChangeKey operation, Fig. 4, starts with checking the size of the heap at the global descriptor to verify if the item with the priority that it desires to change exists in the heap. Thereafter, it attempts to register itself by marking the slot of the item, and calls HeapifyUp or HeapifyDown as needed. If the marking fails, it helps the operation that would have marked the slot and thereafter reattempts marking.

In the Fig. 5, the procedures HeapifyUp and HeapifyDown are shown. They take two inputs: the index of the source slot where it starts and the priority of the destination. HeapifyUp keeps on exchanging the item with its parent up the heap until the destination priority is set at the slot such that heap property is restored. On the other hand, HeapifyDown traverses down the heap to do the same. To exchange the item of the current node with that of either the parent or a child, a `CAS` is used to first put a descriptor over there and thereafter exchange is done atomically. If `CAS` fails then Help is called to first help the obstructing operation and then reattempt. The helping procedure ensures lock-freedom.

The Help call is all about synchronization between concurrent HeapifyUp and HeapifyDown procedures. At a conflict, HeapifyDown is given priority. HeapifyUp allows the HeapifyDown to gain ownership of a child slot. This

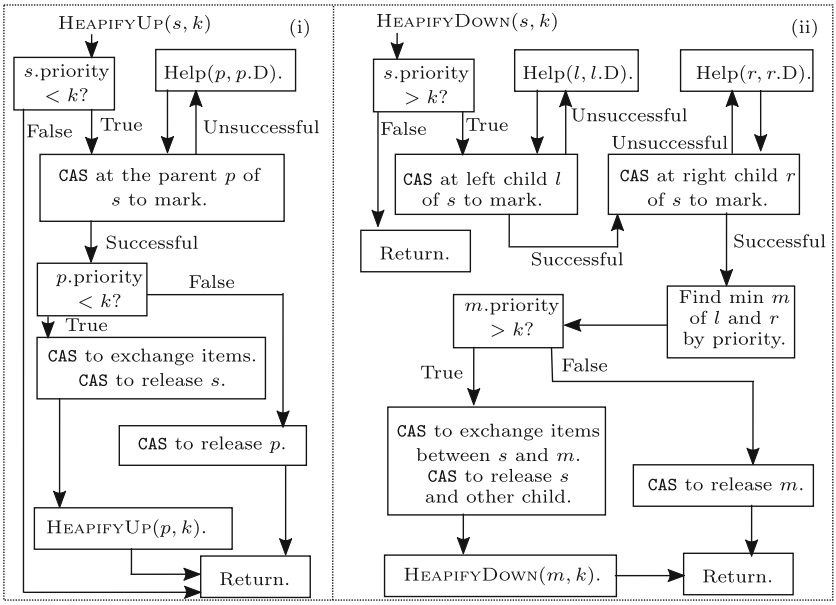

Fig. 5. HeapifyUp and HeapifyDown procedures.

is done by marking the slot with a so called flat descriptor that stores the old information as well. This information is carried by the descriptor at the heap slots, thereby other concurrent operations help accordingly. A HeapifyDown after completing its own task, restores the information of HeapifyUp if that existed at the slot previously.

Please note that, we compare the items at the slots according to their priorities. Moreover, the higher the value of a priority, the lower is the priority as per the min-heap property.

3.2 Design Optimizations

We add two optimizations: (1) "bit-reversal" to ensure that the consecutive Insert operations traverse different subtrees up the heap to restore heap property [17]. (2) Elimination of Insert by handing the items off to the concurrent DeleteMin operations, instead of having the DeleteMin uproot an item out of position from the end of the heap. An eliminated Insert operation can return immediately without even attempting to register itself. Below a brief description of the elimination technique is given.

Elimination Optimization: We observe that the DeleteMin operation lifts an item from the bottom slot in the heap and heapifesDown the heap, while as the Insert operation appends an item to the end of the heap and heapifies Up the heap. Therefore, we can optimize by allowing the Insert to hand-off its item to a concurrent DeleteMin. Thus, the DeleteMin takes an item from

a pending INSERT instead of dislodging one from the end of the heap. Once an INSERT operation successfully hands-off its item, it returns without calling HEAPIFYUP.

We utilize elimination arrays as suggested by Hendlar et. al [12], with each INSERT operation having a dedicated slot in the array. The DELETEMIN operation traverses the array sequentially until it finds a pending INSERT or gets to the end of the array. If the DELETEMIN operation fails to eliminate a pending INSERT, it proceeds with displacing the last item in the heap, otherwise it continues with the item taken from the pending INSERT as described below.

After eliminating a pending INSERT operation (lifting its item from the elimination array), the DELETEMIN compares the lifted item to the item at the root of the heap. If the lifted item has a higher priority, the DELETEMIN returns the lifted item without having to call HEAPIFYDOWN. Otherwise, it proceeds to place the lifted item and returns the item previously at the root.

4 Correctness Proof

To prove linearizability, we define the *linearization point* of each ADT operation. We order the operations, which have *definitely returned*, according to their linearization points, thus obtaining a *sequential history* of execution. Thereby, it is shown that the *concurrent history* of execution of a finite number of ADT operations is *equivalent* to a sequential history. By induction, any concurrent execution is thus shown to be equivalent to a definite sequential history. Additionally, we need to show that each of the ADT operations necessarily brings the heap in a state that satisfies the heap property before its completion.

Proving lock-freedom requires that infinitely often some non-faulty processing thread will complete its operation in a finite number of steps regardless of the failed or delayed threads. To prove lock-freedom, we shall show that no operation *op busy-waits* (by holding locks, for example) when *obstructed* by a concurrent operation op' and goes to help op' to finish its operation. It may well be that *op* is repeatedly obstructed by concurrent operations op_i, $i \in \{1, 2, \ldots\}$ never letting it complete its own operation, however, by virtue of the same protocol it is proved that at least one non-faulty thread completes its operation in finite number of steps. Under the constraints of space, we sketch the two proofs here.

Theorem 1. *The ADT operations implemented by CoMPiQ are linearizable.*

Proof. The linearization points of the ADT operations are the following:

1. INSERT: An INSERT$(k, elem)$ operation begins with checking the global descriptor *gd* of the heap. If it finds that there is a pending concurrent operation, it goes to first help that by calling a COMPLETEWRITE(gd). Thus, an INSERT starts taking steps for itself only after the successful `CAS` that registers it. After that, INSERT calls COMPLETEWRITE to write its descriptor, and on completion, a HEAPIFYUP is called. The HEAPIFYUP finally makes the item *elem* part of the heap with the successful `CAS`. Thus for an INSERT operation

that successfully performs this CAS step, its linearization point is there. In case it gets helped by a concurrent operation the successful CAS that finally makes the item *elem* part of the heap is the linearization point. However, in either case the CAS of linearization point is performed before the completion of INSERT. For detail, see [26]. Clearly, the linearization point of an INSERT operation is between its invoke and return.

2. DELETEMIN: Depending on the return, there can be following cases:
   (a) DELETEMIN returns EMPTY: The linearization point is at the atomic read step where the DELETEMIN reads that the heap-size is 1 i.e. it contains a the dummy descriptor object.
   (b) DELETEMIN returns an item *elem*: In this case, where it registers itself by a successful CAS at *gd*, it is guaranteed that it will itself complete if not obstructed, or will get helped by a concurrent operation. Also, once the descriptor *od* is written, a concurrent INSERT or DELETEMIN operation treats the root of the binary heap as deleted. Thus, the return of the concurrent operation treats the DELETEMIN that successfully put the descriptor as if it had already returned. Therefore, the linearization point of a DELETEMIN in this case is at the step where it registers itself.

   Thus, the linearization point of a DELETEMIN is between invoke and return.

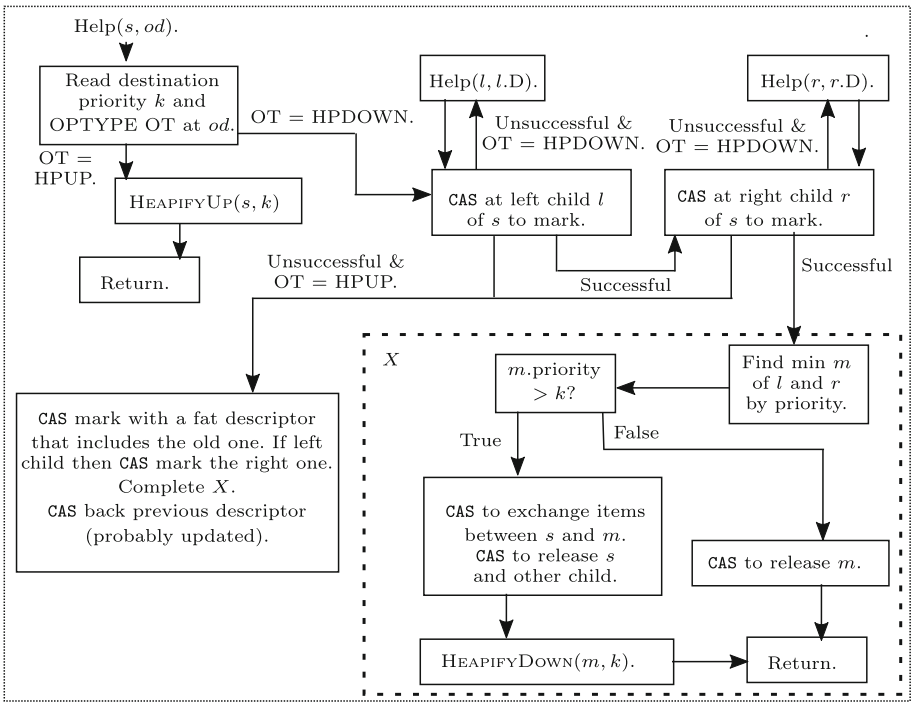

Fig. 6. HELP procedure in CoMPiQ.

3. ChangeKey: Similar to an Insert, a ChangeKey terminates after its item is relocated from one slot to another by way of calling a HeapifyUp or a HeapifyDown. The `CAS` where the item will be visible to every operation with its modified priority is the linearization point of a ChangeKey operation. When a ChangeKey returns without making any changes in the heap, its linearization point is at the atomic `read` step where it reads the size of the heap.

Furthermore, it can be observably determined that no operation returns before the heap property is restored by calling either a HeapifyUp or a HeapifyDown procedure. Any write on a shared memory word in the algorithm happens by way of only a `CAS`. A dummy descriptor at the root ensures that no null pointer is dereferenced. Clearly, the heap invariant is maintained across the linearization points of the ADT operations. □

Theorem 2. *The ADT operations implemented by CoMPiQ are lock-free.*

Proof. We can observe in the algorithm that a concurrent write at any shared word happens only using a `CAS`. Further, if op_1 and op_2 are any two concurrent operations, at no point after the failure of a `CAS`, op_1 or op_2 repeats the same `CAS` step without helping the other operation. This methodology ensures that at least one of the processes do finish its operation in a finite number of steps. □

5 Evaluation

In this section, we present an evaluation of our lock-free heap using micro-benchmarks and a parallelized implementation of Dijkstra's SSSP algorithm described in [24]. For the micro-benchmark, we compare the *heap-based* concurrent priority queue implementations described below:

1. **CoMPiQ:** Our implementation of a lock-free heap as described in Sect. 3 with elimination optimization.
2. **LB-Heap:** A fine grained locking implementation by Hunt et. al. [17]. Releases locks and re-aquires them on each iteration of the heapifyup operation to prevent deadlocks with concurrent heapifydown operation.
3. **Champ:** Modification of `LB-Heap` to remove redundant unlock and lock operations. Deadlocks are prevented using `tryLock()` in the heapifyup and only releasing already acquired locks if a subsequent `tryLock()` fails. We received Java code from the authors, reimplemented it in C/C++ and included the exponential back-off and bit-reversal scheme [17] to reduce contention.
4. **STL-Heap:** The C++ STL `std::priority_queue<T>` made thread-safe with a single global lock (coarse-grained locking). We experimented with multiple lock synchronization primitives, however the mutex was the best performing.

Methodology: We performed our evaluations on a dual-socket server with a 3.4 GHz Intel E5-2687W-v2 having 16 physcores (32 hardware threads by hyper-threading), 16 GB of RAM, running Ubuntu 13.04 Linux. All the algorithms in

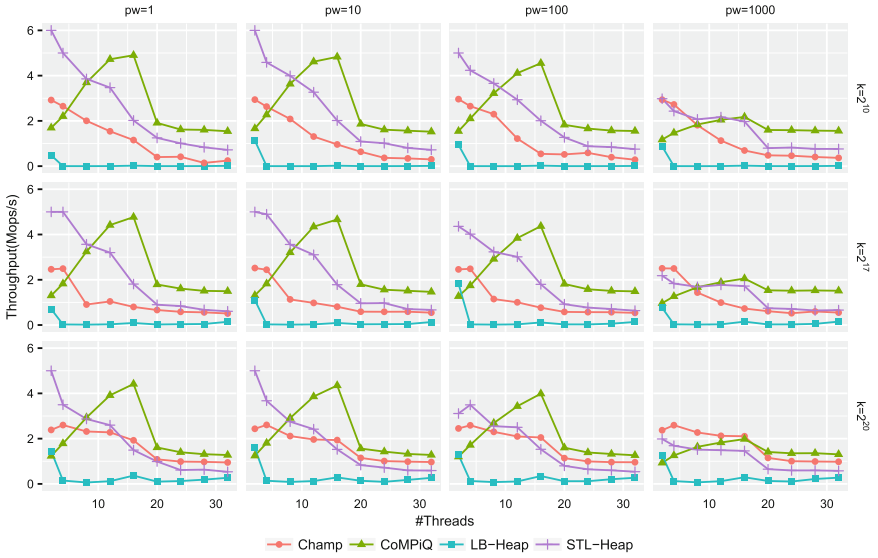

Fig. 7. Throughput `Insert/DeleteMin` operations executed uniformly and randomly independent on the heap implementations as we vary the number of threads and parallel-work (pw) in CPU cycles. `K` represents the initial number of items in the heap.

the micro-benchmark were implemented in C/C++, compiled with gcc version 4.9.2, -O3 and run as part of the ASCYLIB library [4]. Additionally, we pin software threads onto hardware cores so as to leverage CPU affinity within sockets. We utilize SSMEM [4] with epoch-based garbage collection [9].

We measured throughput as Million operations per second (Mops/s), while varying the number of threads, initial heap size and contention (parallel-work: work performed by threads outside accessing the heap). We do not expect the concurrent heap to be repeatedly accessed by threads without work in between so we simulate this work by varying parallel-work(pw), thus giving a more realistic evaluation than just stress testing. The lower the parallel-work, the more contention experienced by threads accessing the heap. We varied the number of items in the heap before starting the measurements with ($k \in \{2^{10}, 2^{17}, 2^{20}\}$). Operations on the heap are randomly chosen with a distribution of 50% Insert and 50% DeleteMin operations. Priorities for inserted items where selected uniformly at random from the range of all 64-bit integers. Each experiment run for 5 seconds, we present the average over 6 runs for each parameter configuration.

Throughput: Figure 7 presents measured throughput in Million operations per second (Mops/s) as we vary the contention in parallel-work (pw) in CPU cycles and the number of threads. We present three sets of graphs for three initial sizes of the heap ($k \in \{2^{10}, 2^{17}, 2^{20}\}$), this is to show the effect of heap size on the execution time of the operations.

The figure shows that with small initial size 2^{10} (row 1, Fig. 7), at low thread contention, the single lock implementation STL-Heap outperforms other implementations. This attributed to the low overheads incurred by STL-Heap using mutual exclusion, and high overheads on both the multi-lock LB-Heap, Champ and lock-free CoMPiQ. Similar observation about the single-lock implementation was made in previous works [17,23].

Champ optimizes on the heapifyup operation of LB-Heap by removing redundant `unlock` and `re-lock` operations in uncontended cases, however, in case of contention, failure to acquire a lock, results in releasing locks held, and an attempt to reacquire them. On modern architectures with private caches, a process that releases a lock has a much higher probability of reacquiring the lock if it attempts to acquire the lock immediately. Thus, an implementation of Champ was showing similar performance figures as LB-Heap. We modified the implementation by adding exponential back-off between releasing a lock, and attempts to re-acquire the same lock. This is the major reason for the performance differences between Champ and LB-Heap.

As we increase the number of threads, contention for the lock increases and performance deteriorates. We observe that the lock-free algorithm with elimination(CoMPiQ), scales up as we increase the thread count. Elimination increases the concurrency exploited by the operations as an Insert completes without contending for the global descriptor or creating contention within the heap with HeapifyUp operations. All implementations degrade in performance as we deploy more than 16 threads due to communication overheads across sockets. We still observe that CoMPiQ offers better throughput on multi-socket executions.

As we increase the initial size of the heap (height of the heap), "bit-reversal" allows for more concurrency, and thus reducing the impact of synchronization overhead on the performance. In this regard, we see that for heap size 2^{20} the performance of the single-lock implementation drops significantly relative to other implementations with increasing thread count. The CoMPiQ performs best with increased opportunities for concurrency and reduced contention on the heap.

Increasing parallel work ($pw \in \{1, 10, 100, 1000\}$) affects the lock-based implementations more than the lock-free implementations because the concurrency overheads no longer dominate performance, but concurrency. Thus, CoMPiQ still outperforms other implementations.

Discussion: Key observations are that – the heap is an inherently sequential data structure and even the most efficient implementation is still outperformed significantly by a single thread executing on a sequential heap for low levels of parallel-work. However, as the parallel work increases, the benefit of increasing concurrency becomes more significant. Additionally, bit-reversal offers more opportunities for disjoint-access allowing better exploitation of concurrency on larger size heaps to offset synchronization overheads. This is less significant in smaller heaps as successive Insert operations conflict on the paths to the root. The root and the size variable create a severe bottleneck in both blocking and non-blocking implementations, as all operations have to modify the size variable, while all DeleteMin operations modify the size and also block the root for

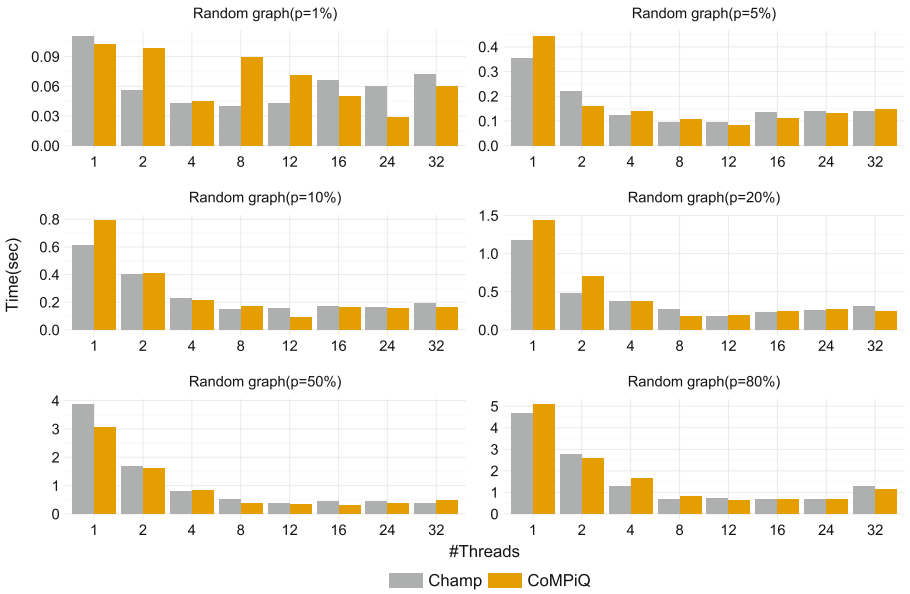

Fig. 8. Runtimes for parallel Dijkstra's SSSP for different random graphs

exclusive access. CoMPiQ uses elimination to reduce on the contention at the bottleneck, thus resulting in better performance.

Parallel SSSP: One important application of priority queues that utilizes the changeKey operation is the Dijkstra's SSSP algorithm. To evaluate the performance of **CoMPiQ**, we implemented CoMPiQ as part of the benchmark suite received from [24] which included a parallel implementation Dijkstra's algorithm and **Champ** which is the only other implementation that supports `changeKey` operation. The parallel Dijkstra's SSSP algorithm availed in the benchmark relies heavily on locks to ensure correctness, with this in mind, we plugged in our implementation without modifying the parallel SSP algorithm for fair comparison. A more optimistic parallel implementation of Dijkstra's SSSP algorithm is left as future work. In the benchmark, running time is measured over several input graphs and number of execution threads. Each input graph is generated with 10,000 vertices, with edges occurring independently randomly with some probability p and a random weight in the range [1–100]. The parallel Dijkstra's SSSP algorithm and the evaluated priority queues are implemented in Java.

Figure 8 shows that the **CoMPiQ** performs comparably with **Champ**. This implies that overheads incurred to ensure lock-freedom do not degrade performance of CoMPiQ when used in parallel applications. Note that node locks are used in this parallelization, thus, as pointed out earlier, we anticipate significant performance improvements with a more optimistic parallelization, that uses atomics to update node weights. We only considered implementations that

support the changeKey operation. Please refer to [24] for an evaluation involving skiplist-based priority queues that do not support `changeKey`.

6 Conclusion

In this paper, we presented a novel algorithm for an array-based unbounded concurrent lock-free heap. The heap implements a priority queue interface with the additional facility of changing the priority of an item in the runtime. Our work contributes to many important applications, which use the priority queue ADT and need to modify the priority of the items dynamically, in a definitive way. Our micro-benchmark based experiments demonstrated that our algorithm performs well in comparison to similar existing algorithms that use locks.

With array-based implementations, it is trivial to represent a d-ary heap, however, implementation of a concurrent multi-way heap creates new challenges. The multi-way heaps lower the traversal cost by reducing the height of the tree, but increase the synchronization overhead as an operation attempts to determine the priorities of all the d-children. The techniques introduced in this article may be useful in implementing non-blocking versions of the heap-ordered d-ary heaps.

References

1. Alistarh, D., Kopinsky, J., Li, J., Shavit, N.: The spraylist: a scalable relaxed priority queue. In: ACM SIGPLAN Notices, vol. 50, pp. 11–20. ACM (2015)
2. Barnes, G.: A method for implementing lock-free shared-data structures. In: 5th SPAA, pp. 261–270 (1993)
3. Barnes, G.: Wait-free Algoritzms for Heaps. Department of Computer Science and Engineering, University of Washington (1994)
4. David, T., Guerraoui, R., Trigonakis, V.: Asynchronized concurrency: The secret to scaling concurrent search data structures. In: 20th ASPLOS, pp. 631–644 (2015)
5. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
6. Dechev, D., Pirkelbauer, P., Stroustrup, B.: Lock-free dynamically resizable arrays. In: Shvartsman, M.M.A.A. (ed.) OPODIS 2006. LNCS, vol. 4305, pp. 142–156. Springer, Heidelberg (2006). https://doi.org/10.1007/11945529_11
7. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numer. Math. **1**(1), 269–271 (1959)
8. Dragicevic, K., Bauer, D.: Optimization techniques for concurrent STM-based implementations: a concurrent binary heap as a case study. In: 23rd IPDPS, pp. 1–8 (2009)
9. Fraser, K.: Practical lock-freedom. Ph.D. thesis, University of Cambridge (2004)
10. Fujimoto, R.M.: Parallel discrete event simulation. Commun. ACM **33**(10), 30–53 (1990)
11. Gidenstam, A., Papatriantafilou, M., Sundell, H., Tsigas, P.: Efficient and reliable lock-free memory reclamation based on reference counting. IEEE Trans. Parallel Distrib. Syst. **20**(8), 1173–1187 (2009)

12. Hendler, D., Shavit, N., Yerushalmi, L.: A scalable lock-free stack algorithm. In: 16th SPAA, SPAA 2004, pp. 206–215 (2004)
13. Henry, G.J.: The unix system: the fair share scheduler. AT&T Bell Lab. Tech. J. **63**(8), 1845–1857 (1984)
14. Herlihy, M.: Wait-free synchronization. ACM TOPLAS **13**(1), 124–149 (1991)
15. Herlihy, M.: A methodology for implementing highly concurrent data objects. ACM Trans. Program. Lang. Syst. **15**(5), 745–770 (1993)
16. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM TOPLAS **12**(3), 463–492 (1990)
17. Hunt, G.C., Michael, M.M., Parthasarathy, S., Scott, M.L.: An efficient algorithm for concurrent priority queue heaps. Inf. Process. Lett. **60**(3), 151–157 (1996)
18. Israeli, A., Rappoport, L.: Efficient wait-free implementation of a concurrent priority queue. In: Schiper, A. (ed.) WDAG 1993. LNCS, vol. 725, pp. 1–17. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-57271-6_23
19. Michael, M.M.: Hazard pointers: safe memory reclamation for lock-free objects. IEEE Trans. Parallel Distrib. Syst. **15**(6), 491–504 (2004)
20. Prim, R.C.: Shortest connection networks and some generalizations. Bell Labs Tech. J. **36**(6), 1389–1401 (1957)
21. Shavit, N., Lotan, I.: Skiplist-based concurrent priority queues. In: 14th IPDPS, pp. 263–268. IEEE (2000)
22. Shavit, N., Touitou, D.: Software transactional memory. Distrib. Comput. **10**(2), 99–116 (1997)
23. Sundell, H., Tsigas, P.: Fast and lock-free concurrent priority queues for multi-thread systems. J. Parallel Distrib. Comput. **65**(5), 609–627 (2005)
24. Tamir, O., Morrison, A., Rinetzky, N.: A heap-based concurrent priority queue with mutable priorities for faster parallel algorithms. In: 19th OPODIS (2016)
25. Vitter, J.S.: Design and analysis of dynamic huffman codes. J. ACM (JACM) **34**(4), 825–845 (1987)
26. Walulya, I., Chatterjee, B., Datta, A.K., Niyoliya, R., Tsigas, P.: Concurrent lock-free unbounded priority queue with mutable priorities. Technical report, 2018:06, ISNN 1652–926X, Department of Computer Science and Engineering, Chalmers University of Technology (2018)