Practical Concurrent Priority Queues
Student Paper for Seminar in Algorithms 2013W
Technical University of Vienna

Jakob Gruber, 0203440

September 23, 2015

Abstract

Priority queues are abstract data structures which store a set of key/-value pairs and allow efficient access to the item with the minimal (maximal) key. Such queues are an important element in various areas of computer science such as algorithmics (i.e. Dijkstra's shortest path algorithm) and operating system (i.e. priority schedulers).

The recent trend towards multiprocessor computing requires new implementations of basic data structures which are able to be used concurrently and scale well to a large number of threads. In particular, lock-free structures promise superior scalability by avoiding the use of blocking synchronization primitives.

Concurrent priority queues have been extensively researched over the past decades. In this paper, we discuss three major ideas within the field: fine-grained locking employs multiple locks to avoid a single bottleneck within the queue; SkipLists are search structures which use randomization and therefore do not require elaborate reorganization schemes; and relaxed data structures trade semantic guarantees for improved scalability.

1 Introduction

In the past decade, advancements in computer performance have been made mostly through an increasing number of processors instead of higher clock speeds. This development necessitates new approaches to data structures and algorithms that take advantage of concurrent execution on multiple threads and processors.

This paper focuses on the priority queue data structure, consisting of two operations traditionally called `Insert` and `DeleteMin`. `Insert` places an item into the queue together with its priority, while `DeleteMin` removes and returns the lowest priority item. Both of these operations are expected to have a complexity of at most $O(\log n)$. Priority queues are used in a large variety of situations such as shortest path algorithms and scheduling.

Concurrent priority queues have been the subject of research since the 1980s [2, 3, 5, 6, 12, 23, 24, 27, 28]. While early efforts have focused mostly on parallelizing Heap structures [13], more recently priority queues based on Pugh's SkipLists [30] seem to show more potential [33, 36, 10, 21]. Current research has

also examined relaxed data structures [40, 1] which trade strictness of provided guarantees against improved scalability.

In the following, we investigate the evolution of concurrent priority queues suitable for general use, i.e. unbounded range queues based on widely available atomic primitives such as Compare-And-Swap (CAS). Section 2 outlines basic concepts and definitions. In Section 3, we cover the Hunt et al. queue as a representative of early heap-based queues using fine-grained locking to avoid the bottleneck of a single global lock. Lock-free SkipList-based structures offering better disjoint-access parallelism are discussed in Section 4, while Section 5 introduces the concept of relaxed data structures and presents two such priority queues. Finally, related work is presented in Section 7, and the paper is concluded in Section 8.

2 Concepts and Definitions

Concurrent data structures are intended to be accessed simultaneously by several processes at once. *Lock-based* structures ensure that only a limited number of processes may enter a critical section at once. *Lock-free* data structures eschew the use of locks, and guarantee that at least a single process makes progress at all times. Since lock-free structures are non-blocking, they are not susceptible to priority inversion, deadlock, and livelock. *Wait-freedom* further guarantees that every process finishes each operation in a bounded number of steps. In practice, wait-freedom often introduces an unacceptable overhead; lock-freedom however has proven to be both efficient and to scale well to large numbers of processes. Recently, Kogan and Petrank have also developed a methodology for implementing efficient wait-free data structures [19].

There are several different criteria which allow reasoning about the correctness of concurrent data structures. *Linearizable* [11] operations appear to take effect at a single instant in time at so-called linearization points. *Quiescently consistent* [34] data structures guarantee that the result of a set of parallel operations is equal to the result of a sequential ordering after a period of quiescence, i.e. an interval without active operations has passed. Linearizability as well as quiescent consistency are composable — any data structure composed of linearizable (quiescently consistent) objects is also linearizable (quiescently consistent). *Sequential consistency* [20] requires the result of a set of operations executed in parallel to be equivalent to the result of some sequential ordering of the same operations.

Lock-free algorithms and data structures are commonly constructed using synchronization primitives such as Compare-And-Swap (CAS), Fetch-And-Add (FAA), Fetch-And-Or (FAO), and Test-And-Set (TAS). The CAS instruction, which atomically compares a memory location to an expected value and sets it to a new value if they are equal, is implemented on most modern architectures and can be considered a basic building block of lock-free programming. More exotic primitives such as Double-Compare-And-Swap (DCAS) and Double-Compare-Single-Swap (DCSS) exist as well, but are not yet widely available and require inefficient software emulations to be used.

An area in memory accessed frequently by a large number of processes is said to be *contended*. Contention is a limiting factor regarding scalability: concurrent reads and writes to the same location must be serialized by the cache coherence

protocol, and only a single concurrent CAS can succeed while all others must retry. *Disjoint-access parallelism* is the concept of spreading such accesses in order to reduce contention as much as possible.

Priority queues are an abstract data structure allowing insertion of items with a given priority and removal of the lowest-priority item in logarithmic time. Search trees and Heaps (which are flattened representations of complete trees such that each node's key is at least as large as those of both children) are concrete data structures which are usually used to implement sequential priority queues. However, both require fairly elaborate reorganization after `DeleteMin` and/or `Insert` operations, which are especially challenging to achieve in a concurrent environment.

SkipLists [30] have become increasingly popular in concurrent data structures because they are both simple to implement, and exhibit excellent disjoint-access parallelism. In contrast to search trees and Heaps, reorganization is not necessary since SkipLists rely on randomization for an expected $O(\log n)$ complexity of `Insert`, `Search` and `Delete` operations. A SkipList may be visualized as a set of linked lists with corresponding levels. The linked list at level 0 is the sorted sequence of all objects in the SkipList, and higher levels provide "shortcuts" into the SkipList such that a list of level $i+1$ contains a subset of the objects in level i. A SkipList node n is said to be of level i if it is in all lists of levels $[0, i]$ and in none of levels $[i+1, \infty]$. Upon insertion, the new node's level is assigned at random according to a geometric distribution; deletion simply removes the node.

```
1  struct slist_t {
2    size_t max_level;
3    node_t head[max_level];
4  };
5
6  struct node_t {
7    key_t key;
8    value_t value;
9    size_t level;
10   node_t *next[level];
11 };
12
13 /* All operations are expected O(log n) time. */
14 void slist_insert(slist_t *l, key_t k, value_t v);
15 bool slist_delete(slist_t *l, key_t k, value_t &v);
16 bool slist_contains(slist_t *l, key_t k);
```

Figure 1: Basic SkipList structure and operations.

3 Fine-grained Locking Heaps

In the remaining paper, we will discuss implementations of several concurrent priority queue implementations. Early designs have mostly been based on search trees [4, 15] and heaps [2, 3, 5, 6, 12, 23, 24, 27, 28]. We chose the priority queue

by Hunt et al. [13] as a representative of early concurrent priority queues since it has been proven to perform well [33] in comparison to other efforts of the time such as [26, 2, 41]. It is based on a Heap structure and attempts to minimize lock contention between threads by a) adding per-node locks, b) spreading subsequent insertions through a bit-reversal technique, and c) letting insertions traverse bottom-up in order to minimize conflicts with top-down deletions.

However, significant limitations to scalability remain. A global lock is required to protect accesses to a variable storing the Heap's size which all operations must obtain for a short time. Disjoint-access through bit-reversal breaks down once a certain amount of traffic is reached, since only subsequent insertions are guaranteed to take disjoint paths towards the root node. Note also that the root node is a severe serial bottleneck, since it is potentially part of every insertion path, and necessarily of every `DeleteMin` operation. Finally, in contrast to later dynamic SkipList-based designs, the capacity of Hunt Heaps is fixed upon creation.

Benchmarking results in the literature have been mixed; a sequential priority queue protected by a single global lock outperforms the Hunt et al. Heap in most cases [13, 36]. Speed-up only occurs once the size of the Heap reaches a certain threshold such that concurrency can be properly exploited instead of being dominated by global locking overhead.

4 Lock-free Priority Queues

Traditional data structures such as the Heap have fallen out of favor; instead, SkipLists [30, 29] have become the focus of modern concurrent priority queue research [33, 36, 10, 21, 1]. SkipLists are both conceptual simple as well as simple to implement; they also exhibit excellent disjoint-access parallelism properties, and do not require rebalancing due to their reliance on randomization.

A state of the art lock-free SkipList implementation based on the CAS instruction by Fraser [8] is freely available[1] under a BSD license. Fraser exploits unused pointer bits to mark nodes as logically deleted, with physical deletion following as a second step.

SkipLists are dynamic data structures in the sense that they grow and shrink at runtime. In consequence, careful handling of memory accesses and (de)allocations are required. As an additional requirement, these memory management schemes must themselves be both scalable and lock-free to avoid limiting the SkipList itself. Fraser in particular employs lock-free epoch-based garbage-collection, which frees a memory segment only once all threads that could have seen a pointer to it have exited the data structure.

4.1 Shavit and Lotan

Shavit and Lotan were the first to propose the use of SkipLists for priority queues [21]. Their initial locking implementation [33] builds on Pugh's concurrent SkipList [29], which uses one lock per node per level.

A crucial observation is that nodes which are only partially connected do not affect correctness of the data structure. As soon as the first level (i.e. `node.level[0]`) has been successfully connected, a node is considered to be in the

[1] `http://www.cl.cam.ac.uk/research/srg/netos/lock-free/`

SkipList. Therefore, both insertions and deletions can be split into steps — insertions proceed bottom-up while deletions proceed top-down. Locks are held only for the current level which helps to reduce contention between threads.

```
1 struct node_t {
2   [...] /**< Standard node members as above. */
3   atomic<bool> deleted; /**< Initially false. */
4   time_t timestamp;
5   lock_t locks[level + 1];
6 };
```

Figure 2: Shavit and Lotan structure.

Likewise, deletions are split into a logical phase (atomically setting the `node.deleted` flag) and a physical phase which performs the actual pointer manipulations and can be seen as a simple call to the underlying SkipList's `sl_delete` function.

A `DeleteMin` call starts the list head, and attempts to atomically set the deletion flag using a `CAS(node.deleted, false, true)` call (or equivalent constructs). If it succeeds, the current node is physically deleted and returned to the caller. Otherwise, `node.next[0]` is set as the new current node and the procedure is repeated. If the end of the list is reached, `DeleteMin` returns false to indicate an empty list.

Note that so far this implementation is not linearizable: consider the case in which a slow thread A is in the middle of a `DeleteMin` call. Within this context, we refer to the node with key i as node i, or simply i. Several `CAS` operations have failed, and A is currently at node j. A fast thread B then first inserts a node i, followed by a node k such that $i < j < k$, i.e. the former and latter nodes are inserted, respectively, before and after thread A's current node. Assuming further that all nodes between j and k have already been deleted, then thread A will return node k. This execution is not linearizable; however, it is quiescently consistent since operations can be reordered to correspond to some sequential execution at quiescent periods.

Shavit and Lotan counteract this by introducing a `timestamp` for each node which is set upon successful insertion. In this variant, each `DeleteMin` operation simply ignores all nodes it sees that have not been fully inserted at the time it was called.

Explicit memory management is required to avoid dereferencing pointers to freed memory areas by other threads after physical deletion. This implementation uses a dedicated garbage collector thread in combination with a timestamping mechanism which frees `node`'s memory only when all threads that might have seen a pointer to `node` have exited the data structure.

Herlihy and Shavit [10] recently described and implemented a lock-free, quiescently consistent version of this idea in Java. While mostly identical, notable differences are that a) the new variant is based on a lock-free skiplist, b) the timestamping mechanism was not employed and thus linearizability was lost, and c) explicit memory management is not required because the Java virtual machine provides a garbage collector.

4.2 Sundell and Tsigas

Sundell and Tsigas proposed the first lock-free concurrent priority queue in 2003 [36]. The data structure is linearizable and is implemented using commonly available atomic primitives CAS, TAS, and FAA. In contrast to other structures covered in this paper, this priority queue is restricted to contain items with distinct priorities. Inserting a new item with a priority already contained in the list simply performs an update of the associated value. A real-time version is also provided which we will not discuss further (interested readers are referred to [36]).

```
1 struct node_t {
2   [...] /**< Standard node members as above. */
3   size_t valid_level;
4   node_t *prev;
5 };
```

Figure 3: Sundell and Tsigas structure.

The structure of each node is basically identical to Figure 1. However, Sundell and Tsigas exploit the fact that the two least significant bits of pointers on 32- and 64-bit systems are unused and reuse these as deletion marks. A set least significant bit on a pointer signifies that the current node is about to be deleted. Reuse of `node.level[i]` pointers prevents situations in which a new node is inserted while its predecessor is being removed, effectively deleting both nodes from the list. Likewise, the reuse of the `node.value` pointer ensures that updates of pointer values (which occur when a node with the inserted priority already exists) handle concurrent node removals correctly.

As in the Shavit and Lotan priority queue, insertions proceed bottom-up while deletions proceed top-down — on the one hand, the choice of opposite directions reduces collisions between concurrent insert and delete operations, while on the other hand removing nodes from top levels first allows many other concurrent operations to simply skip these nodes, further improving performance. `node.valid_level` is updated during inserts to always equal the highest level of the SkipList at which pointers in this node have already been set (as opposed to `node.level`, which equals the final level of the node).

A helping mechanism is employed whenever a node is encountered that has its deletion bit set, which attempts to set the deletion bits on all next pointers and then removes the node from the current level. The `node.prev` pointer is used as a shortcut to the previous node, avoiding a complete retraversal of the list.

This implementation uses the lock-free memory management invented by Valois [37, 38] and corrected by Michael and Scott [25]. It was chosen in particular because this scheme can guarantee validity of `prev` as well as all `next` pointers. Additionally, it does not require a separate garbage collector thread.

A rigorous linearizability proof is provided in the original paper [36] which shows linearization points for all possible outcomes of all operations.

Benchmarks performed by Sundell and Tsigas show their queue performing noticeably better than both locking queues from Sections 4.1 and 3, and slightly

better than a priority queue consisting of a SkipList protected by a single global lock.

4.3 Lindén and Jonsson

One of the most recent priority queue implementations was published by Lindén and Jonsson in 2013 [21]. They present a linearizable, lock-free concurrent priority queue which achieves a speed-up of 30−80% compared to other SkipList-based priority queues by minimizing the number of CAS operations within most `DeleteMin` operations.

A priority queue implementation is called deterministic when the algorithm does not contain randomized elements. It is called strict when `DeleteMin` is guaranteed to return the minimal element currently within the queue (in contrast to relaxed data structures which are discussed further in the next section). All such priority queues share an inherent bottleneck, since all threads calling `DeleteMin` compete for the minimal element, causing both contention through concurrent CAS operations on the same variable as well as serialization effort by the cache coherence protocol for all other processor accessing the same cache line.

```
1 struct node_t {
2   [...] /**< Standard node members as above. */
3   atomic<bool> inserting;
4 };
```

Figure 4: Lindén and Jonsson structure.

In this implementation, most `DeleteMin` operations only perform logical deletion by setting the deletion flag with a single FAO call; nodes are only deleted physically once a certain threshold of logically deleted nodes is reached.

This mechanism requires a new invariant, in that the set of all logically deleted nodes must always form a prefix of the SkipList. Recall that in the Sundell and Tsigas queue, deletion flags for node `n` were packed into `n.next` pointers, preventing insertion of new nodes *after* deleted nodes. This implementation instead places the deletion flag into the lowest level `next` pointer of the previous node, preventing insertions *before* logically deleted nodes.

Once the prefix of logically deleted nodes reaches a specified length (represented by `BoundOffset`), the first thread to observe this fact within `DeleteMin` performs the actual physical deletions by updating `slist.head[0]` to point at the last logically deleted node with a single FAO operation. The remaining `slist.head` pointers are then updated, and all physically deleted nodes are marked for recycling.

Since at any time, the data structure contains a prefix of logically deleted nodes, all `DeleteMin` operations must traverse this sequence before reaching a non-deleted node. In general, reads of nonmodified memory locations are very cheap; however, benchmarks in [21] have shown that after a certain point, the effort spent in long read sequences significantly outweighs the reduced number of CAS calls. It is therefore crucial to choose `BoundOffset` carefully, with the authors recommending a prefix length bound of 128 for 32 threads.

The actual `DeleteMin` and `Insert` implementations are surprisingly simple. Deletions simply traverse the list until the first node for which `(ptr, d)= FAO((node.next[0], d), 1)` returns a previously unset deletion flag (`d = 0`) and then return `ptr`. Insertions occur bottom-up and follow the basic Fraser algorithm [8], taking the separation of deletion flags and nodes into account. The `node.inserting` flag is set until the node has been fully inserted, and prevents moving the list head past a partially inserted node. Fraser's epoch-based reclamation scheme [8] is used for memory management.

The authors provide high level correctness and linearizability proofs as well as a model for the SPIN model checker. Performance has been shown to compare favorably to both Sundell and Tsigas and Shavit and Lotan queues, with throughput improved by up to 80%.

5 Relaxed Priority Queues

The body of work discussed in previous sections creates the impression that the outer limits of strict, deterministic priority queues have been reached. In particular, Lindén and Jonsson conclude that scalability is solely limited by `DeleteMin`, and that less than one modified memory location per thread and operation would have to be read in order to achieve improved performance [21].

Recently, relaxation of provided guarantees have been investigated as another method of reducing contention and improving disjoint-access parallelism. For instance, k-FIFO queues [17] have achieved considerable speed-ups compared to strict FIFO queues by allowing `Dequeue` to return elements up to k positions out of order.

Relaxation has also been applied to concurrent priority queues with some success, and in the following sections we discuss two such approaches.

5.1 Wimmer et al.

Wimmer et al. presented the first relaxed, linearizable, and lock-free priority queue in [40]. It is integrated as a priority scheduler into their *Pheet* task-scheduling system, and an open-source implementation is available[2].

Their paper presents several variations on the common theme of priority queues: a distributed work-stealing queue which can give no guarantees as to global ordering since it consists of separate priority queues at each thread; a relaxed centralized priority queue in which no more than k items are missed by any processor; and a relaxed hybrid data structure which combines both ideas and provides a guarantee that no thread misses more than kP items where P is the number of participating threads. In this section, we examine only the hybrid variant since it provides both the scalability of work-stealing queues and the ordering guarantees of the centralized priority queue.

The hybrid queue consists of a list of globally visible items and one local item list as well as a local sequential priority queue per thread. The thread-local counter `remaining_k` tracks how many more items may be added to the local queue until all local items must be made globally visible to avoid breaking guarantees.

[2] `http://pheet.org`

```
1  struct globals_t {
2    list_of_item_t global_list;
3  };
4
5  /* Thread-local items. */
6  struct locals_t {
7    list_of_item_t local_list;
8    pq_t prio_queue;
9    size_t remaining_k;
10 };
```

Figure 5: Wimmer et al. structure.

Whenever an item is added, it is first added locally to both `local_list` and `prio_queue` and `remaining_k` is decremented. If `remaining_k` reaches zero, then the local item list is appended to the global list, and all not yet seen (by this thread) items of the global queue are added to the local priority queue.

`DeleteMin` simply pops the local priority queue as long is it is non-empty and the popped item has already been deleted. When a non-deleted item is popped, it is atomically marked as deleted and returned to the caller. If instead we are faced with an empty local queue, we attempt to spy, i.e. copy items from another thread's local list.

Both `Insert` and `DeleteMin` periodically synchronize with the global list by adding all items that have not yet been seen locally to `prio_queue`. Memory allocations are handled using the wait-free memory manager by Wimmer [39].

The Wimmer et al. priority queue was evaluated using a label-correcting variant of Dijkstra's shortest path algorithms. Their model creates a task for each node expansion, and therefore comes with a considerable task scheduling overhead. Nonetheless, the parallel implementation scales well up to 10 threads, and further limited performance gains are made until 40 threads. To date, no direct comparisons to other concurrent priority queues have been possible since the data structure is strongly tied to *Pheet*, but a separate implementation of k-priority queues is planned.

5.2 SprayList

The SprayList is another recent approach towards a relaxed priority queue by Alistarh et al. [1]. Instead of the distributed approach described in the previous section, the SprayList is based on a central data structure, and uses a random walk in `DeleteMin` in order to spread accesses over the $O(P \log^3 P)$ first elements with high probability, where P is again the number of participating threads.

Fraser's lock-free SkipList [8] again serves as the basis for the priority queue implementation. The full source code is available online[3]. In the SprayList, `Insert` calls are simply forwarded to the underlying SkipList.

The `DeleteMin` operation however executes a random walk, also called a *spray*, the purpose of which is to spread accesses over a certain section of the SkipList uniformly such that collisions between multiple concurrent `DeleteMin`

[3] `https://github.com/jkopinsky/SprayList`

calls are unlikely. This is achieved by starting at some initial height, walking a randomized number of steps, descending a randomized number of levels, and repeating this procedure until a node n is reached on the lowest level. If n is not deleted, it is logically deleted and returned. Otherwise, a *spray* is either reattempted, or the thread becomes a cleaner, traversing the lowest level of the SkipList and physically removing logically deleted nodes it comes across. A number of dummy nodes are added to the beginning of the list in order to counteract the algorithm's bias against initial items.

The *spray* parameters are chosen such that with high probability, one of the $O(P \log^3 P)$ first elements is returned, and that each of these elements is chosen roughly uniformly at random. The final effect is that accesses to the data structure are spread out, reducing contention and resulting in a noticeably lower number of CAS failures in comparison to strict priority queues described in Section 4.

The authors do not provide any statement as to the linearizability (or other concurrent correctness criteria) of the SprayList, and it is not completely clear how to define it since no sequential semantics are given.

Benchmarks show promising results: the SprayList scales well at least up to 80 threads, and performs close (within a constant factor) to an implementation using a random remove instead of `DeleteMin`, which the authors consider as the performance ideal.

6 Performance Results

In this section, we compare the performance of several different concurrent priority queue implementations:

- *GlobalLock* An instance of the `std::priority_queue<T>` class provided by the C++ standard library, protected by a single global lock.
- *Heap* Hunt et al. provide an implementation of their Heap-based design [13] at `ftp://ftp.cs.rochester.edu/pub/packages/concurrent_heap/`. However, as the original code was written for the MIPS architecture, we chose to use the alternative implementation in *libcds*[4] instead. The heap capacity was 2^{18} — lower values led to a quasi-deadlock situation under high concurrency as described in [7].
- *Noble* An implementation of the Sundell and Tsigas priority queue [36] is available in the Noble library[5]. *Noble* is lock-free, based on SkipLists, and limited to unique priorities. According to the authors, the commercial Noble library includes a more efficient version of this data structure that can also handle duplicate priority values.
- *Linden* Code for the Lindén and Jonsson priority queue [21] is provided by the authors under an open source license[6]. It is lock-free and uses Fraser's lock-free SkipList. The aim of this implementation is to minimize contention in calls to `DeleteMin`. We chose 32 as the `BoundOffset` in

[4] `http://sourceforge.net/p/libcds/code/`
[5] `http://www.non-blocking.com/Eng/download-demo.aspx`
[6] `http://user.it.uu.se/~jonli208/priorityqueue`

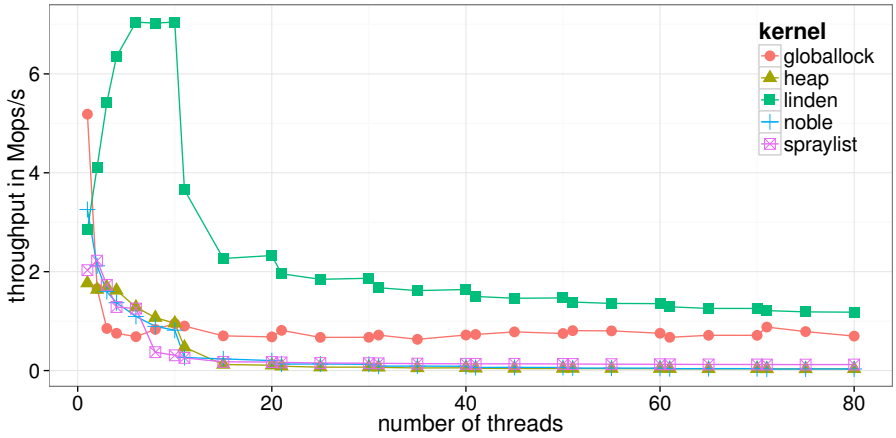
order to optimize performance on a single socket. A `BoundOffset` of 128 performed only marginally better at high thread counts.

- *SprayList* A relaxed concurrent priority queue based on Fraser's SkipList using random walks to spread data accesses incurred by `DeleteMin` calls. Code provided by Alistarh et al. is available on Github[7].

Unfortunately, we were not able to obtain an implementation of the Shavit and Lotan priority queue for benchmarking. The Wimmer et al. data structure was omitted in the following benchmarks since it is tightly coupled with the task-scheduling framework *Pheet*, and cannot directly be compared to other implementations in its current form.

In each test run, the examined priority queue was initially filled with 2^{15} elements. We then ran a tight loop of 50% insertions and 50% deletions for a total of 10 seconds, where all `Insert` operations within this context implicitly choose a key uniformly at random from the range of all 32-bit integers. This methodology seems to be the de facto standard for concurrent priority queue benchmarks [1, 21, 33, 36]. Each run was repeated for a total of 10 times and we report on the average throughput.

All benchmarks were compiled with `-O3` using GCC 4.8.2, and executed with threads pinned to cores. Evaluations took place on an 80-core Intel Xeon E7-8850 machine with each processor clocked at 2 GHz. The benchmarking code was adapted from Lindén and Jonsson's benchmarking suite and is available at `https://github.com/schuay/seminar_in_algorithms`.



The *Linden* priority queue emerged as the clear winner of these benchmarks, with impressive scalability while all threads remain on the same socket. In fact, it is the only tested implementation to scale to any significant degree; *Heap* achieves minor gains for up to 3 threads, while all others immediately lose throughput under concurrency. Performance of the *Linden* queue drops significantly once it is executed on more than 10 threads, thus incurring communication overhead across sockets. This effect is repeated to a lesser degree every time an additional socket becomes active. However, the *Linden* queue has clearly superior throughput until the tested maximum of 80 threads.

[7] `https://github.com/jkopinsky/SprayList`

GlobalLock also performs surprisingly well at high concurrency levels. It remains competitive when compared to *Linden* at about $\frac{2}{3}$ of its throughput. For single-threaded execution, it achieves the highest throughput of all tested implementations. It is interesting to note that the *GlobalLock* shows complementary behavior to *Linden* when new sockets become active; we believe that the additional latency incurred by cross-socket communications might lead to reduced contention at the single lock.

All remaining implementations behave similarly, displaying very low throughput under concurrency. This is a surprise in the case of *Noble*; the original benchmarks performed by Sundell and Tsigas led us to expect improvements upon both *GlobalLock* and *Heap* [36]. However, we are somewhat reassured in our benchmarks by the fact that Lindén and Jonsson's results are similar.

The *SprayList* result is even more unexpected, and we believe that it may be caused by our use of `malloc` instead of the included custom lock-free allocator, which we were unable to execute without errors.

As a final observation we note that even the most efficient strict concurrent priority queue never exceeds the throughput of the simple sequential heap executed on a single thread by more than a factor of two. There is hope however since relaxed data structures continue to scale beyond 10 threads (see the benchmarks performed in [1] for an example).

7 Related Work

There have been many publications on more specialized variants of concurrent priority queues which were not covered in this paper.

Liu and Spear present a lock-free, array-based priority queue in [22] using the non-standard DCAS and DCSS instructions. In addition to the `Insert` and `DeleteMin` operations, it also provides an `ExtractMany` function which returns the n highest priority items in the queue, and a function that returns items which "probably" have high priority. Other implementations based on non-standard instructions can be found in [14, 9].

Another possibility is the restriction of priorities to a particular set, i.e. bounded range priorities. Shavit and Zemach present two such data structures in [35] using combining funnels and bins of items. Experimental results have shown strong scalability until at least 256 threads.

Concurrent priority queues have also been studied in other contexts such as distributed memory systems. Karp and Zhang describe a distributed priority queue in which each processor owns a local priority queue, inserts are sent to a random priority queue, and `DeleteMin` operations simply access the local queue [16]. Sanders extends this idea to remove the globally best elements instead [31].

Finally, the principle of relaxation has also been applied to other data structures. Kirsch, Lippautz, and Payer invented an efficient k-FIFO queue [17], The trend towards relaxed data structures in general is examined further in [32, 18].

8 Conclusion

Priority queues are one of the most important abstract data structures in computer science, and much effort has been put into parallelizing them efficiently.

In this paper, we have outlined the evolution of concurrent priority queues from initial heap-based designs, through a period of increasingly efficient SkipList queues, to current research into relaxed data structures.

The switch from heaps to SkipLists as the backing data structure highlights how a simple change in direction can help revitalize an entire field of research. SkipList-based priority queues are the current state of the art in strict shared-memory concurrent priority queues: they provide strong guarantees and scale well to up to the tens of threads in practice. Important limiting factors are contention at the front of the list and the large number of CAS failures. The Lindén and Jonsson queue is designed to minimize the latter; but the former is inherent to all strict priority queues, which could mean that the peak performance in such structures has been reached.

Recently invented relaxed priority queues do not exhibit the inherent bottleneck at the front of the list, as they do not necessarily return the minimal element within the queue and are able to spread `DeleteMin` accesses over a larger area of the structure. In consequence, relaxed queues scale noticeably better and to larger thread counts than strict designs. Further research is necessary in order to fully explore the possibilities provided by relaxed data structures.

References

[1] Dan Alistarh et al. *The SprayList: A Scalable Relaxed Priority Queue*. Tech. rep. 2014.

[2] Rassul Ayani. "LR-algorithm: concurrent operations on priority queues". In: *Proceedings of the 2nd International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE. 1990, pp. 22–25.

[3] Jit Biswas and James C Browne. *Simultaneous update of priority structures*. Tech. rep. Texas Univ., Austin (USA). Dept. of Computer Sciences, 1987.

[4] Joan Boyar, Rolf Fagerberg, and Kim S Larsen. *Chromatic priority queues*. Citeseer, 1994.

[5] Sajal K Das, Falguni Sarkar, and M Cristina Pinotti. "Distributed priority queues on hypercube architectures". In: *Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 1996, pp. 620–627.

[6] Narsingh Deo and Sushil Prasad. "Parallel heap: An optimal parallel priority queue". In: *The Journal of Supercomputing* 6.1 (1992), pp. 87–98.

[7] Kristijan Dragicevic and Daniel Bauer. "A survey of concurrent priority queue algorithms". In: *Proceedings of the 23rd International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE. 2008, pp. 1–6.

[8] Keir Fraser. "Practical lock-freedom". PhD thesis. PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579, 2004.

[9] Michael Barry Greenwald. "Non-blocking synchronization and system design". PhD thesis. Stanford University, 1999.

[10] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2012.

[11] Maurice P Herlihy and Jeannette M Wing. "Linearizability: A correctness condition for concurrent objects". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12.3 (1990), pp. 463–492.

[12] Qin Huang and WW Weihl. "An evaluation of concurrent priority queue algorithms". In: *Proceedings of the 3rd International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE. 1991, pp. 518–525.

[13] Galen C Hunt et al. "An efficient algorithm for concurrent priority queue heaps". In: *Information Processing Letters* 60.3 (1996), pp. 151–157.

[14] Amos Israeli and Lihu Rappoport. "Efficient wait-free implementation of a concurrent priority queue". In: *Distributed Algorithms*. Springer, 1993, pp. 1–17.

[15] Theodore Johnson. *A Highly Concurrent Priority Queue Based on the B-link Tree*. Tech. rep. MR 1311488, 1991.

[16] Richard M Karp and Yanjun Zhang. "Randomized parallel algorithms for backtrack search and branch-and-bound computation". In: *Journal of the ACM (JACM)* 40.3 (1993), pp. 765–789.

[17] Christoph M Kirsch, Michael Lippautz, and Hannes Payer. *Fast and scalable k-fifo queues*. Tech. rep. 2012-04, Department of Computer Sciences, University of Salzburg, 2012.

[18] Christoph M Kirsch and Hannes Payer. "Incorrect systems: it's not the problem, it's the solution". In: *Proceedings of the 49th Annual Design Automation Conference (DAC)*. ACM. 2012, pp. 913–917.

[19] Alex Kogan and Erez Petrank. "A methodology for creating fast wait-free data structures". In: *ACM SIGPLAN Notices*. Vol. 47. 8. ACM. 2012, pp. 141–150.

[20] Leslie Lamport. "How to make a multiprocessor computer that correctly executes multiprocess programs". In: *IEEE Transactions on Computers (TC)* 100.9 (1979), pp. 690–691.

[21] Jonatan Lindén and Bengt Jonsson. "A Skiplist-Based Concurrent Priority Queue with Minimal Memory Contention". In: *Principles of Distributed Systems*. Springer, 2013, pp. 206–220.

[22] Yujie Liu and Michael Spear. "A lock-free, array-based priority queue". In: *ACM SIGPLAN Notices* 47.8 (2012), pp. 323–324.

[23] Carlo Luchetti and M Cristina Pinotti. "Some comments on building heaps in parallel". In: *Information processing letters* 47.3 (1993), pp. 145–148.

[24] Bernard Mans. "Portable distributed priority queues with MPI". In: *Concurrency: Practice and Experience* 10.3 (1998), pp. 175–198.

[25] Maged M Michael and Michael L Scott. *Correction of a Memory Management Method for Lock-Free Data Structures*. Tech. rep. DTIC Document, 1995.

[26] RV Nageshwara and Vipin Kumar. "Concurrent access of priority queues". In: *IEEE Transactions on Computers (TC)* 37.12 (1988), pp. 1657–1665.

[27] Stephan Olariu and Zhaofang Wen. "Optimal parallel initialization algorithms for a class of priority queues". In: *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 2.4 (1991), pp. 423–429.

[28] Sushil K Prasad and Sagar I Sawant. "Parallel heap: A practical priority queue for fine-to-medium-grained applications on small multiprocessors". In: *Proceedings of the 7th International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE. 1995, pp. 328–335.

[29] William Pugh. *Concurrent maintenance of skip lists*. Tech. rep. 1998.

[30] William Pugh. "Skip lists: a probabilistic alternative to balanced trees". In: *Communications of the ACM* 33.6 (1990), pp. 668–676.

[31] Peter Sanders. "Randomized priority queues for fast parallel access". In: *Journal of Parallel and Distributed Computing* 49.1 (1998), pp. 86–97.

[32] Nir Shavit. "Data structures in the multicore age". In: *Communications of the ACM* 54.3 (2011), pp. 76–84.

[33] Nir Shavit and Itay Lotan. "Skiplist-based concurrent priority queues". In: *Proceedings of the 14th International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE. 2000, pp. 263–268.

[34] Nir Shavit and Asaph Zemach. "Diffracting trees". In: *ACM Transactions on Computer Systems (TOCS)* 14.4 (1996), pp. 385–428.

[35] Nir Shavit and Asaph Zemach. "Scalable concurrent priority queue algorithms". In: *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing (PODC)*. ACM. 1999, pp. 113–122.

[36] Håkan Sundell and Philippas Tsigas. "Fast and lock-free concurrent priority queues for multi-thread systems". In: *Proceedings of the 17th International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE. 2003, 11–pp.

[37] John D Valois. "Lock-free linked lists using compare-and-swap". In: *Proceedings of the 14th annual ACM symposium on Principles of Distributed Computing (PODC)*. ACM. 1995, pp. 214–222.

[38] John David Valois. "Lock-free data structures". PhD thesis. 1996.

[39] Martin Wimmer. "Wait-free hyperobjects for task-parallel programming systems". In: *Proceedings of the 27th International Symposion on Parallel & Distributed Processing (IPDPS)*. IEEE. 2013, pp. 803–812.

[40] Martin Wimmer et al. "Data Structures for Task-based Priority Scheduling". In: *arXiv preprint arXiv:1312.2501* (2013).

[41] Yong Yan and Xiaodong Zhang. "Lock bypassing: An efficient algorithm for concurrently accessing priority heaps". In: *Journal of Experimental Algorithmics (JEA)* 3 (1998), p. 3.