Engineering MultiQueues: Fast Relaxed Concurrent Priority Queues

Marvin Williams
Karlsruhe Institute of Technology, Germany

Peter Sanders
Karlsruhe Institute of Technology, Germany

Roman Dementiev
Intel Deutschland GmbH, München, Germany

Abstract

Priority queues with parallel access are an attractive data structure for applications like prioritized online scheduling, discrete event simulation, or greedy algorithms. However, a classical priority queue constitutes a severe bottleneck in this context, leading to very small throughput. Hence, there has been significant interest in concurrent priority queues with relaxed semantics. We investigate the complementary quality criteria *rank error* (how close are deleted elements to the global minimum) and *delay* (for each element x, how many elements with lower priority are deleted before x). In this paper, we introduce *MultiQueues* as a natural approach to relaxed priority queues based on multiple sequential priority queues. Their naturally high theoretical scalability is further enhanced by using three orthogonal ways of batching operations on the sequential queues. Experiments indicate that MultiQueues present a very good performance–quality tradeoff and considerably outperform competing approaches in at least one of these aspects.

We employ a seemingly paradoxical technique of "wait-free locking" that might be of more general interest to convert sequential data structures to relaxed concurrent data structures.

2012 ACM Subject Classification Computing methodologies → Concurrent computing methodologies

Keywords and phrases concurrent data structure, priority queues, randomized algorithms, wait-free locking

Digital Object Identifier 10.4230/LIPIcs.ESA.2021.81

Related Version This paper continues a previous paper that only appeared as a preprint and SPAA brief announcement listed below. New contributions compared to that are measures for better locality, more analysis of element delays, and more realistic experiments.
Preprint: `https://arxiv.org/abs/1411.1209` [20]
SPAA brief announcement: `https://dl.acm.org/doi/10.1145/2755573.2755616` [19]

Supplementary Material The source code of our MultiQueue implementation as well as the code for the experimental evaluation can be found as follows:
Software: `https://github.com/marvinwilliams/multiqueue`
Software (Experimental Evaluation): `https://github.com/marvinwilliams/multiqueue_experiments`
Software (Archived Source Code and Experimental Data): `https://algo2.iti.kit.edu/williams/esa21_multiqueues/`

Funding This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 882500).

1 Introduction

Priority queues (PQs) are a fundamental data structure for many applications. They manage a set of elements and support operations for efficiently inserting elements and deleting the smallest element (*deleteMin*). Whenever we have to dynamically reorder operations performed by an algorithm, PQs can turn out to be useful. Examples include job scheduling, graph algorithms for shortest paths and minimum spanning trees, discrete event simulation, best first branch-and-bound, and other best first heuristics.

On modern parallel hardware, we often have the situation that p parallel threads want to access the PQ concurrently both to insert and to delete elements. This is problematic for several reasons. First of all, even the semantics of a parallel PQ is unclear. The classical notion of serializability is not only expensive to achieve but also not very useful from an application point of view. For example, in a branch-and-bound application, a serializable parallel PQ could arbitrarily postpone insertion and corresponding deletion of a search tree node on the path leading to the eventual solution. This makes the application arbitrarily slower than the sequential one. The *ideal* semantics of a concurrent PQ (CPQ) would be that any element for which an insertion has started becomes visible instantaneously for deletion from any other thread. This is unattainable for fundamental physical reasons but can serve as a basis for defining the quality of relaxed priority queues (RPQs). In Section 2 we use this approach to define the complementary notions of the *rank error* of deleted elements and the *delay* of elements that are overtaken by larger elements. After discussing related work in Section 3, Section 4 describes the main contribution of this paper: MultiQueues are a simple approach to CPQs based on $c \cdot p$ sequential PQs (SPQs) for some constant $c > 1$. Insertions go to a random SPQ so that each of them contains a representative sample of the globally available elements. Deleted elements are the smaller of the minima of *two* randomly chosen SPQs. By choosing two rather than one queue, fluctuations in the distribution of queue elements are stabilized. Consistency is maintained by locking queues that are changed. See Figure 1 for an example. Since there are more queues than threads, no thread ever has to wait for a lock. Thus, we achieve a lock-free (and even wait-free) algorithm despite using locks which is an interesting feature of MultiQueues.

Although MultiQueues scale better than competitors both in theory and practice, they have the practical problem that they lack cache locality – in each operation, a thread accesses several cache lines from randomly chosen SPQs which are rarely reused but usually cause cache invalidation costs later. We therefore introduce three orthogonal measures for improving locality: (1) Insertion and deletion *buffers* ensure that most operations access only a small number of cache lines. (2) We use sequential queues that allow *bulk access* in a more cache-efficient way than using several single-element operations. (3) Threads optionally *stick* to the same set of queues for several consecutive operations. In the analysis (Section 4.5), we show that MultiQueue operations are almost as fast as their sequential counterparts – scaling linearly with the number of threads. We are also able to analyze rank error and delay

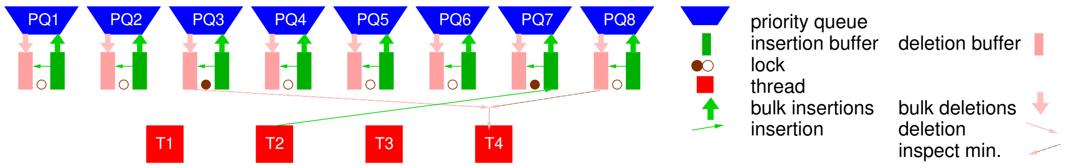

Figure 1 A MultiQueue with 8 sequential queues and 4 threads. Thread T2 currently locks PQ7 to insert an element. T4 inspected PQ3 and PQ8 and now locks PQ8 to remove its smallest element.

under moderately simplifying assumptions – it is linear in expectation and $O(p \log p)$ with high probability. Section 5 summarizes an extensive experimental evaluation of MultiQueues including a comparison with alternative approaches. Section 6 concludes the paper.

2 Preliminaries

A priority queue pq represents a set of elements. We use $n = |pq|$ for the size of the queue. Classical priority queues support the operations *insert* for inserting an element and *deleteMin* for obtaining and removing the smallest element. The most frequently used SPQ is the binary heap [29].

A *relaxed CPQ* does not require the *deleteMin* operation to return the minimum element. The *rank* of an element of a set M is its position in a sorted representation of M. A natural quality criterion for the *deleteMin* operation is the *rank error* of the returned element, i.e., the number of elements in the CPQ at the time of deletion that are smaller than the returned element. Over the entire use of the CPQ, one can look at the mean rank, the largest observed rank, or, more generally, the distribution of observed ranks. A complementary measure that has previously been largely neglected is the *delay* of a queue element x. The delay of x is the number of elements with lower priority than x that are deleted before x. This measure is important because it can be closely tied to the performance of applications. For example, consider a CPQ used in an optimization problem where the currently best queue element leads to the ultimate solution. If the CPQ from now on always remove the second best element, it has excellent rank error but will never find the solution because the best element in infinitely delayed.

It is difficult for a CPQ to consistently and efficiently maintain its exact *size*. A straightforward way to approximate the size can for example be implemented similarly to concurrent hash tables [16]. While knowing the size is not of direct importance for most applications, many of them need some kind of *termination detection*. Sequential algorithms would often terminate after they found the queue to be *empty*. Even the check for emptiness is difficult in concurrent settings since there is no way to know whether some other thread has concurrently called an insertion operation (or is about to do so). Hence, we view termination detection as a problem that has to be solved by the application. We adopt the semantics of most RPQs we are aware of that the *deleteMin* operation is allowed to fail – implying that the queue could not find any remaining elements without being able to prove that the queue is actually empty. A failed *deleteMin* has the current size of the queue as its rank error, i.e., as if ∞ was removed from the queue.

An algorithm is *wait-free* if it is guaranteed to make progress in a bounded number of steps [13]. Since we discuss randomized algorithms, we use this term also if the bound is probabilistic, i.e., if the expected number of steps is bounded.

3 Related Work

There has been considerable work on bulk parallel priority queues (BPQs) in the 1990s [8, 18, 23]. BPQs differ from CPQs in that they assume synchronized batched operation but they are still relevant as a source of ideas for asynchronous implementations. Indeed, Sanders [23] already discusses how these data structures could be made asynchronous in principle: Queue server threads periodically and collectively extract the globally smallest elements from the queue moving them into a buffer that can be accessed asynchronously. Note that within this buffer, priorities can be ignored since all buffered elements have a low rank. Similarly, an insertion buffer keeps recently inserted elements. Moreover, the best theoretical results on BPQs give us an idea of how well RPQs should scale asymptotically. For example, Sanders' BPQ [23] removes the p smallest elements of the BPQ in time $\mathrm{O}(\log n)$. This indicates that the worst-case rank error and delay close to *linear* in the number of threads should be achievable.

Sanders' BPQ [23] is based on the very simple idea to maintain a local SPQ on each thread and to send inserted elements to random threads. This idea is older, stemming from Karp and Zhang [14]. This approach could actually be used as an RPQ. Elements are inserted into the SPQ of a randomly chosen thread. Each thread deletes elements from its local SPQ. It is shown that this approach leads to only a constant factor more work compared to a sequential algorithm for a globally synchronized branch-and-bound application where producing and consuming elements takes constant time. Unfortunately, for a general asynchronous RPQ, the Karp-Zhang-queue [14] has limitations since slow threads could "sit" on small elements, while fast threads would busily process elements with high rank – in the worst case, the rank error could grow arbitrarily large. Our MultiQueue builds on the Karp-Zhang-queue [14], adapting it to a shared memory setting, decoupling the number of queues from the number of threads, and, most importantly, using a more robust protocol for *deleteMin*.

Many previous CPQs are based on the SkipList data structure [17]. At its bottom, the SkipList is a sorted linked list of elements. Search is accelerated by additional layers of linked lists. Each list in level i is a random sample of the list in level $i-1$. Many previous CPQs delete the exact smallest element [26, 27, 15, 6]. This works well if there are not too many concurrent *deleteMin* operations competing for deleting the same elements. However, this inevitably results in heavy contention if very high throughput is required. The SprayList [3] reduces contention for *deleteMin* by removing not the global minimum but an element among the $\mathrm{O}\left(p\log^3 p\right)$ smallest elements. However, for worst case inputs, insertions can still cause heavy contention. This is a fundamental problem of any data structure that attempts to maintain a single globally sorted sequence. Wimmer et al. [31] describe a RPQ for task scheduling based on a hybrid between local and global linked lists and local SPQs. Measurements in Alistarh et al. [3] indicate that this data structure does not scale as well as SprayLists – probably due to a frequently accessed central linked list. Henzinger et al. [12] give a formal specification of RPQs and mention a SkipList based implementation without giving details. Interestingly, for a relaxed FIFO-queue, the same group proposes a MultiQueue-like structure [11].

The *contention avoiding priority queue (CAPQ)* [21, 22] is based on a centralized skip-list S but switches to thread-local insertion and deletion buffers when it detects contention on S. To maintain some global view, operations will still occasionally use S. This combines high accuracy in uncontended situations with high throughput under contention. However, the quality penalty for switching to local buffers is fairly high. Assuming the centralized queue is accessed every $m \in \Theta(p)$ steps (which seems necessary to avoid contention) the paper shows that a rank error in $\mathrm{O}\left(p^2\right)$ is guaranteed for every m-th step. Nothing can be guaranteed for the remaining fraction of $1-1/m$ operations. [1]

For large SPQs, cache-efficient data structures are useful. Unfortunately, the best of these (e.g., [24]) are not useful for CPQs since they are only efficient in an amortized sense and locking an SPQ for an extended period of time could lead to large rank errors and delays.

[1] Such a guarantee could also be achieved with a simplistic version of MultiQueues where each thread inserts and deletes from a fixed queue except that every p steps a thread scans all queues for the globally smallest element.

4 MultiQueues

We first describe the basic MultiQueue in Section 4.1 and then refine it to improve cache locality in Sections 4.2–4.4. In Section 4.5 we provide a simplified theoretical analysis of the run time and quality of the MultiQueue. Section 4.6 discusses implementation details.

4.1 Basic MultiQueue

The basic MultiQueue data structure is an array `Q` of $c \cdot p$ SPQs where c is a tuning parameter and p is the number of parallel threads. Figure 1 gives an example with $c = 2$. Access to each local queue is protected by a lock flag. The *insert* operation locks a random unlocked queue `Q`$[i]$ and inserts the element into `Q`$[i]$. Figure 2 gives pseudocode. Note that this operation is wait-free since we never wait for a locked queue. Since at most p queues can be locked at any time, for $c > 1$ we will have a constant success probability. Hence, the expected time for acquiring a queue is constant. Together with the time for insertion we get expected insertion time $\mathrm{O}(\log n)$.

An analogous implementation of *deleteMin* would lock a random unlocked queue and return its minimal element. However, the quality of this approach leaves a lot to be desired. In particular, it deteriorates not only with p but also with the queue size. One can show that the rank error grows proportional to $\sqrt{n}$ due to random fluctuations in the number of operations addressing the individual queues. Therefore we invest slightly more effort into the *deleteMin* operation by looking at *two* random queues and deleting from the one with the smaller minimum. Figure 2 gives pseudocode for the *insert* and *deleteMin* operations. Our intuition why considering two choices may be useful stems from previous work on randomized load balancing, where it is known that placing a ball on the least loaded of two randomly chosen machines gives a maximum load that is very close to the average load independent of the number of allocated balls [4].

Even when the queue is small, cache efficiency is a major issue for MultiQueues since accessing a queue `Q`$[i]$ from a thread j will move the cache lines accessed by the operation into the cache of thread j. But most likely, some random other thread j' will next need that data causing not only cache misses for j' but also invalidation traffic for j.

```
Procedure insert($e$)
    repeat
        $i := uniformRandom(1..cp)$
        try to lock $Q[i]$
    until lock was successful
    $Q[i].insertToSPQ(e)$
    unlock $Q[i]$
```

```
Procedure deleteMin
    repeat
        $i := uniformRandom(1..cp)$
        $j := uniformRandom(1..cp)$
        if $Q[i].min > Q[j].min$ then swap $i$, $j$
        try to lock $Q[i]$
    until lock was successful
    $e := Q[i].deleteMinFromSPQ$;   unlock $Q[i]$
    return $e$
```

Figure 2 Pseudocode for basic MultiQueue *insert* and *deleteMin*. This code assumes that an empty SPQ returns ∞ as the minimum element. A return value of ∞ from *deleteMin* then indicates that no element could be found (which does not guarantee that all SPQs are empty). A practical implementation might make additional efforts to find elements when encountering empty SPQs.

```
**Procedure** insertToSPQ($e$)
  **if** $D = \emptyset \vee e < \max D$ **then**
    **if** $|D| < b$ **then** $D := D \cup \{e\}$;   **return**
    $(e, D) := (\max D, (D \setminus \{\max D\}) \cup \{e\})$
  **if** $|I| = b$ **then** flush $I$ to $M$
  $I := I \cup \{e\}$
```

```
**Procedure** deleteFromSPQ
  **if** $D = \emptyset$ **then return** $\infty$ // fail
  $e := D.\mathit{deleteMin}$
  **if** $D = \emptyset$ **then**
    refill $D$ from $M \cup I$
  **return** $e$
```

Figure 3 Pseudocode for inserting and deleting elements from an already locked sequential queue represented by the main queue M, an insertion buffer I and a deletion buffer D.

4.2 Buffering

To reduce the average number of cache lines accessed, we represent each SPQ by the main queue M, an *insertion buffer* I and a *deletion buffer* D that is organized as a sorted ring-buffer. Each buffer has a fixed capacity b. We maintain the invariant that (unless the SPQ is empty) D contains the smallest elements of $M \cup D \cup I$. This implies that **min** of the SPQ is always the first element of D.

To maintain this invariant, **inserting** an element e into an SPQ first checks whether D is empty or contains a larger element. If so, e is inserted into D. If D was not full, this finishes the insertion. Otherwise, e becomes the largest element in D. If I is full, it is flushed into the main queue. Finally, e is inserted into I. See Figure 3 for high-level pseudocode.

DeleteMin is straightforward. When D is not empty, its smallest element is removed and returned. If this empties D, it is refilled from $M \cup I$. A simple way to do this is to first flush I into M and then extract the smallest b elements from M into D. Alternatively, we can first refill D from M only and then scan through I to swap elements smaller than $\max D$ with the largest elements from D in a fashion analogous to what is done in *insertSPQ*. This has the advantage that all interactions between the buffers and M is in the form of batches of predictable size. This will be exploited in Section 4.3.

Buffering implies that the operations most often only access the buffers themselves and the lock (accessing and modifying the buffers requires the SPQ to be locked). More specifically, an insertion "usually" reads the largest element from D and then operates on I (assuming that inserted elements rarely go to D). *deleteMin* "usually" only accesses D. When buffers are flushed or refilled, a single thread performs a batch of operations on a single queue and thus can exploit whatever locality the main queue supports. In binary heaps for example, insertions exhibit high locality. Deletions at least exhibit some locality near the root, at the variable specifying the size of the queue, and at the rightmost end of the bottom layer of the tree.

4.3 Batching

To fully exploit that MultiQueues with buffering access the main queues in a bulk fashion we can use SPQs that directly support batch operations. In our prototype, we considered *merging binary heaps*. These are structured like binary heaps but each node contains k sorted elements. The heap invariant remains that nodes contain elements that are no smaller than the elements in the parent node. Insertion and deletion are also analogous to ordinary binary heaps except that compare-and-swap operations are generalized to merge-and-split.

On the one hand, merging binary heaps yield higher cache locality for bulk operations than binary heaps since all elements in each tree node can be stored consecutively and fewer nodes have to be accessed. On the other hand, they come with additional algorithmic complexity and higher worst-case access time. They are also not easily combined with a-ary heaps that are a simpler alternative to achieve somewhat higher locality than in binary heaps.

4.4 Stickiness

As a third measure to improve cache locality we introduce the concept of *stickiness*. The stickiness parameter s controls for how many consecutive operations a thread t reuses a particular local queue for its *insert* and *deleteMin* operations. After the stickiness period of one local queue ends for t or if locking the queue fails, t randomly chooses another queue to stick to. The intuition behind this protocol is that for large enough s and $c \geq 3$, the system converges to a state where threads use disjoint queues most of the time. Stickiness provides a simple mechanism to trade-off increased cache efficiency at the cost of potentially higher rank errors and increased delay.

4.5 Analysis

In this section we analyze the theoretical performance of the MultiQueue under simplifying assumptions.

4.5.1 Running Time

We analyze the asymptotic running time of the operations *insert* and *deleteMin* of the MultiQueue in a realistic asynchronous model of shared memory computing where k threads contending to write the same machine word need time $\mathrm{O}(k)$ to perform those operations (e.g., the aCRQW model [25, Section 2.4.1]).

▶ **Theorem 1.** *With $c > 1$, the expected execution time of the operations insert and deleteMin is $\mathrm{O}(1)$ plus the time for the sequential queue access.*

Proof. Whenever a thread attempts to lock a queue q, there are at most $p-1$ locked queues. Hence the success probability is at least

$$s := 1 - \frac{p-1}{cp} \geq 1 - \frac{1}{c} \in \Omega(1)\,.$$

Hence the expected number of attempts is

$$\sum_{i=1}^{\infty} is(1-s)^{i-1} = \frac{1}{s} \leq \frac{1}{1-\frac{1}{c}} = \frac{c}{c-1} \in \mathrm{O}(1)\,.$$

The stickiness does not affect the expected number of attempts since it only dictates the first queue to attempt to lock but not subsequent attempts in case of failure. The buffers have constant size, so all operations on them are in $\mathrm{O}(1)$. The bulk-inserting of the insertion buffer into the queue and refilling the deletion buffer amortize to the same asymptotic cost as accessing a sequential queue for each element individually. ◀

Note that the above analysis even holds when threads are blocked: In the worst case, each thread holds a lock. The overhead for another thread to avoid these locks is already accounted for in the above proof. Hence all other threads are guaranteed to make progress (in a probabilistic sense). Thus MultiQueues are probabilistically wait-free.

Using different sequential queues, we get the following bounds for the comparison based model and for integer keys:

▶ **Corollary 2.** *MultiQueues with binary heaps need constant average insertion time and expected time $\mathrm{O}(\log n)$ per operation for worst case operations sequences. For integer keys in the range $0..U$, using van Emde Boas search trees [28, 7] as local queues, time $\mathrm{O}(\log\log U)$ per operation is sufficient.*

4.5.2 Quality

Quality analysis is still a partially open problem. However, we will explain how *rank errors* and *delays* can be estimated under simplified but intuitive assumptions in the absence of stickiness.

Let us assume that all m current elements have been allocated uniformly at random to the local queues. This assumption holds if there have been no *deleteMin* operations so far and no locking attempt during *insert* failed. It is an open question whether lock contention and *deleteMin* operations invalidate this assumption in practice. Furthermore, let us assume that we choose the queues to look at for deletion randomly and no queue is currently locked.

Rank Errors

With these assumptions, the probability to delete an element e of rank i is

$$\mathsf{P}(\text{rank} = i) = \left(1 - \frac{2}{cp}\right)^{i-1} \frac{2}{cp}$$

The first factor expresses that the $i - 1$ elements with smaller ranks are not present at the two chosen queues. The second factor is the probability that the element with rank i *is present*. Therefore, the expected rank error in the described situation is

$$\sum_{i=1}^{m} i\mathsf{P}(\text{rank} = i) \leq \sum_{i \geq 1} i \left(1 - \frac{2}{cp}\right)^{i-1} \frac{2}{cp} = \frac{c}{2}p \in \mathrm{O}(p)\,. \tag{1}$$

The cumulative probability that the rank of e is larger than k is

$$\mathsf{P}(\text{rank} > k) = \left(1 - \frac{2}{cp}\right)^{k}, \tag{2}$$

as none of the elements with a rank smaller than k must be present on the two chosen queues. $\mathsf{P}(\text{rank} > k)$ drops to p^{-a} for $k = \frac{ca}{2} p \ln p$, i.e., with probability polynomially large in p, we have a rank error in $\mathrm{O}(p \log p)$.

We can also give qualitative arguments on how the performed operations change the distribution of the elements. Insertions are random and hence move the system towards a random distribution of elements. Deletions tend to remove more elements from queues with small keys than from queues with large keys, thus stabilizing the system. Alistarh et al. [2, 1] confirm our conjecture for a slightly simplified process and also show that it suffices to only "sometimes" look at more than one key. On the other hand, they show that the rank error grows in an unbounded fashion if always only a single queue is considered for deletion.

Delay

The delay of an element e with rank i is equal to the number of deletions of elements with rank higher than i. Let D denote the event that a particular *deleteMin* operation delays or removes element e. Let R denote the event that e is removed. We now compute the conditional probability that e is deleted given that it is delayed or deleted by the *deleteMin* operation. By the definition of conditional probability, we have $\mathsf{P}(\mathsf{R} \mid \mathsf{D}) = \frac{\mathsf{P}(\mathsf{R} \cap \mathsf{D})}{\mathsf{P}(D)}$.

With $\mathsf{P}(\mathsf{R} \cap \mathsf{D}) = \mathsf{P}(\text{rank} = i) = \left(1 - \frac{2}{cp}\right)^{i-1} \frac{2}{cp}$ and $\mathsf{P}(\mathsf{D}) = \mathsf{P}(\text{rank} \geq i) = \left(1 - \frac{2}{cp}\right)^{i-1}$, we get $\mathsf{P}(\mathsf{R} \mid \mathsf{D}) = \frac{2}{cp}$. Since $\mathsf{P}(\mathsf{R} \mid \mathsf{D})$ is constant, the delay follows a geometric distribution with an expected value of $\frac{cp}{2}$ and values in $\mathrm{O}(p \log p)$ with high probability.

4.6 Implementation Details

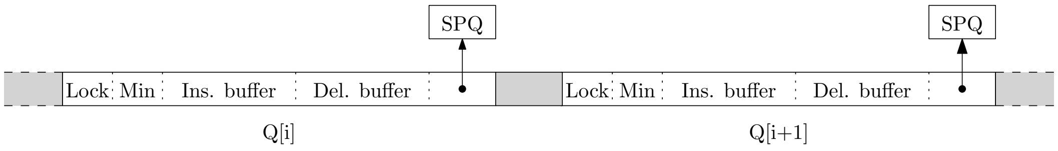

Figure 4 The array containing the lock, the buffers as well the pointer to the SPQ in each entry. The padding blocks (grey) prevent false sharing of neighboring entries.

We first give a detailed description and reasoning of the data structures used in our implementation of the MultiQueue as described above. We then show implementations of the *insert* and *deleteMin* operations.

At its core, a MultiQueue is an array `Q` with one entry for each local queue. Each entry contains a lock, the insertion and deletion buffers, and a pointer to the main queue itself. Moreover, the key of the minimum element in the deletion buffer is redundantly stored next to the lock. Comparing the minima of two local queues thus only involves atomic access to this key and does not access the deletion buffer, which would require locking. Note that this technique allows for slight inaccuracies, as the minimum in the deletion buffer could be different from the explicitly stored minimum key if another thread deletes the minimum from the deletion buffer during comparison. However, consistency is not impacted by this. The entries are padded and aligned to *cache lines* to prevent false sharing of neighboring entries. On systems with nonuniform memory access (NUMA) the padding is extended to *virtual memory pages* that are distributed round-robin over the NUMA-nodes. This balances memory traffic over the nodes. See Figure 4 for an illustration. Additionally, each thread stores local data such as stickiness counters. False sharing between these data structures is also avoided.

The insertion buffer is implemented as a fixed array of size b, preceded by a size counter. The deletion buffer must support efficient removal of the first element, lookup of the last element, and insertions at arbitrary positions. We chose a ring buffer that stores its size and the index of the first element upfront as the underlying data structure. Ring buffers support random lookup and removing the first element in constant time, while inserts at arbitrary positions take at most $b/2$ element moves. We implement both 8-ary heaps and k-merging binary heaps as SPQs, where k is a tuning parameter. 8-ary heaps provide the same theoretical guarantees as binary heaps but better cache locality. To achieve stable worst-case access times, our implementation allocates enough memory so that array resizes by the local queues are unlikely. Having stable worst-case access times is relevant for MultiQueues since an operation that exceptionally takes very long would lock a queue for a long time. If this queue also contains elements of low rank, their delay as well as the rank error of elements deleted in the meantime could become large. On machines with NUMA, the memory for each SPQ is allocated on the same NUMA node.

While our MultiQueue implementation can handle arbitrary element types, the implementation is designed and optimized with elements of small size in mind.

5 Experiments

We first compare our theoretical analysis of rank errors and delays with experimental results. We then perform parameter tuning and examine different implementation variants of the MultiQueue. Afterward, we compare the MultiQueue with other concurrent priority queues in terms of quality, throughput, and scalability. For these experiments, each thread repeatedly either deletes an element from the queue or inserts a new one. This benchmark provides good insights on the maximum throughput and quality of the queues under high contention.

Queue elements are key-value pairs consisting of two 32-bit unsigned integers, where the key determines the element's priority. The queue is filled with n_0 elements prior to the measurement. Unless otherwise noted, $n_0 = 10^6$, the inserted elements are uniformly distributed in $[0, \dots, 2^{32} - 1]$, and the rank errors and delays are measured over the course of 10^7 *deleteMin* operations. To conclude this chapter, we perform a parallel single-source shortest-path (SSSP) benchmark to shed light on the performance under more realistic settings.

Experiments were conducted on two machines: Machine A utilizes an AMD EPYC™ 7702P 64-core processor. Each core runs at 2.0 GHz and supports two hardware threads. The system runs on Ubuntu 20.04 with Linux kernel version 5.4.0. Machine B is a dual socket system with an Intel® Xeon® Platinum 8368 Processor with 2.4 GHz on each socket, yielding 2×38 cores and 152 available hardware threads. The system runs on SUSE Linux Enterprise Server 15 with Linux kernel version 5.3.18. The experiments in Section 5.2 were performed on machine A, for the comparison experiments in Section 5.3 we additionally used machine B. The implementation is written in C++17 and compiled with GCC 10.2.0 with optimization level `-O3`. We use pthreads for thread management and synchronization. Each thread is pinned to a hardware thread.

5.1 Measuring Rank Error and Delay of Relaxed PQs

Measuring the rank errors and delays in real-time imposes the practical problem that we need to know which elements are in the queue at the time of each *deleteMin* operation. We approach this problem as follows. Each thread logs its operations into a preallocated local vector. The log entries get timestamps obtained using a low-overhead high-resolution clock (we currently use the Posix `CLOCK_REALTIME`). For the evaluation, we merge these logs to one global sequence S of operations (sorted by time-stamp). S is then sequentially replayed using an augmented B+ tree (based on a `tlx::btree_map` from the `tlx` library [5]) whose leaves are the current queue elements (sorted by key). By augmenting nodes with their size, this allows determining ranks of deleted elements (and thus rank errors) in logarithmic time (e.g., [25, Section 7.5]). We further augment the interior nodes with delay counters d_v and maintain the invariant that the delay of a leaf e is the sum of the delay counters on the root–e path. When inserting an element, we can establish the invariant by setting d_e to $-\sum_v d_v$ for v on the root–e path. When performing balancing operations on the tree, we can maintain the invariant by pushing the delay counters of the manipulated nodes downward to unchanged subtrees.

5.2 MultiQueue Parameter Tuning

The baseline MultiQueue uses 8-ary heaps as SDQs and does not use stickiness or buffers. As seen in Figure 5, the measured rank errors follow the predicted distribution closely independent of the number of threads and values for c. However, we cannot quite match the prediction in practice.[2] We now inspect the impact of various parameter configurations for

[2] The discrepancy is not due to locking since it persists when threads wait for a long time after each operation. Other sources could be noise in time measurements or inaccuracies due to our random data distribution assumption.

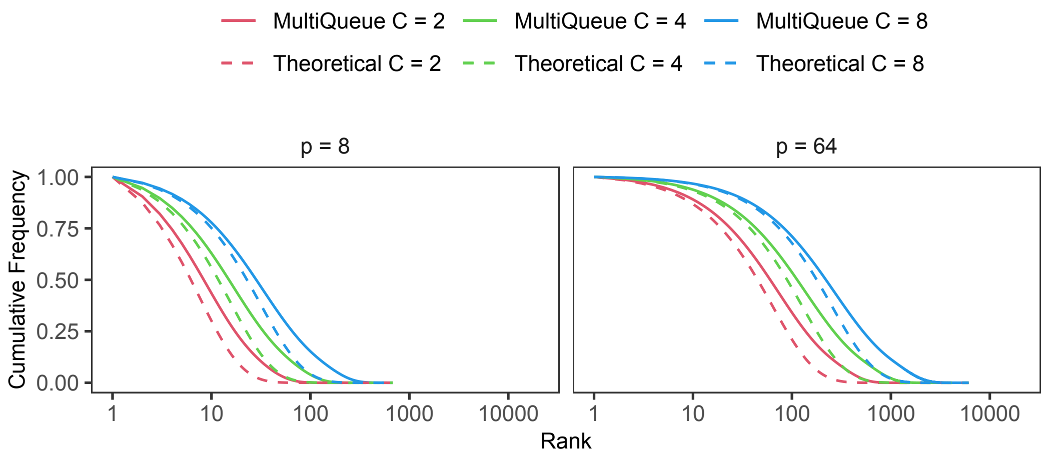

Figure 5 The rank error distribution compared to the distribution from the theoretical analysis for $p = 8$ and $p = 64$, respectively.

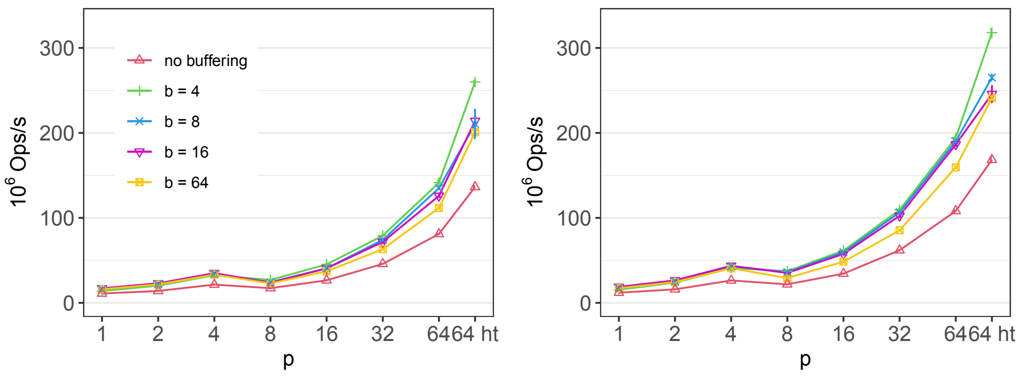

Figure 6 The impact of different buffer sizes to the throughput. The left plot is for $c = 2$, the right one for $c = 4$. The suffix "ht" denotes active hyper-threading.

the MultiQueue on its performance. In particular, we optimize the buffer sizes and explore different values for c and the stickiness. The throughputs are reported for a running time of 3 s and averaged over 5 runs.

Buffer Size. As Figure 6 shows, buffering increases the throughput of the MultiQueue considerably. However, the buffer size has to be chosen carefully, as too large buffers hamper the performance. This is due to the overhead of inserting elements into the deletion buffer. The difference of buffer sizes 4, 8, and 16 is negligible until hyper-threading is active, where a buffer size of 4 gains most. However, the larger the buffers, the less frequent we have to access the SPQ to perform bulk operations. Accessing the SPQ likely generates cache misses, which are especially expensive with NUMA if the SPQ is located on another NUMA node. We therefore use a buffer size of $b = 16$ for further experiments. We have observed no significant differences in rank errors and delays with or without buffering.

Table 1 The throughput, average rank error and delay for $p = 64$. The number in parentheses indicates the expected rank error according to the theoretical analysis.

c	s	Throughput (MOps/s)	Avg. rank error	Avg. delay
2	**1**	125.9	(64) 103.1	103.1
	4	215.5	413.2	412.5
	8	271.6	900.2	896.0
	16	322.5	1853.8	1827.6
	64	411.9	7443.5	7183.2
4	**1**	186.5	(128) 202.6	202.6
	4	327.3	814.2	812.6
	8	405.3	1685.4	1678.0
	16	439.6	3334.9	3303.4
	64	535.3	13349.6	12870.9
8	1	222.5	(256) 405.3	405.1
	4	408.8	1598.6	1592.2
	8	488.0	3284.9	3259.5
	16	497.6	6608.4	6511.9
	64	579.8	26207.6	24902.0
16	1	244.1	(512) 811.7	810.9
	4	461.9	3216.8	3196.1
	8	542.0	6415.2	6338.3
	16	529.3	12814.4	12547.8
	64	603.3	50854.5	47695.3

Stickiness and Number of Queues. Table 1 shows measurements that suggest that rank errors and delays are not just linear in the number of queues cp but also in the stickiness s. While rank errors and delays of MultiQueues are always very similar, this is not generally true. We use the configurations $(c, s) \in \{(4, 1), (4, 4), (8, 8), (16, 8)\}$ for further benchmarks, as they provide interesting trade-offs between throughput and quality.

***k*-Merging Heap.** Our experiments with k-binary merging heaps instead of 8-ary heaps for the SPQs indicated that merging heaps can improve the throughput for high values of s compared to 8-ary heaps. However, the impact is moderate (see Figure 7) in most cases and we stick to 8-ary heaps. Our experiments further showed that using merging heaps has no impact on the quality of the priority queue.

5.3 Comparison with Other Approaches

We compare the MultiQueue to the following state-of-the-art concurrent priority queues.

Linden [15] is a priority queue based on skip lists by Lindén and Jonsson. It only returns suboptimal elements in case of contention on the list head.

Spraylist [3] relaxes skip lists by deleting elements close to the list head to reduce contention.

***k*-LSM [30]** combines thread-local priority queues with a relaxed shared priority queue component. We use the variants with $k = 256$ (**klsm256**) and $k = 1024$ (**klsm1024**).

CAPQ [21] dynamically detects contention to switch from using a shared skip list based priority queue to thread-local buffers.

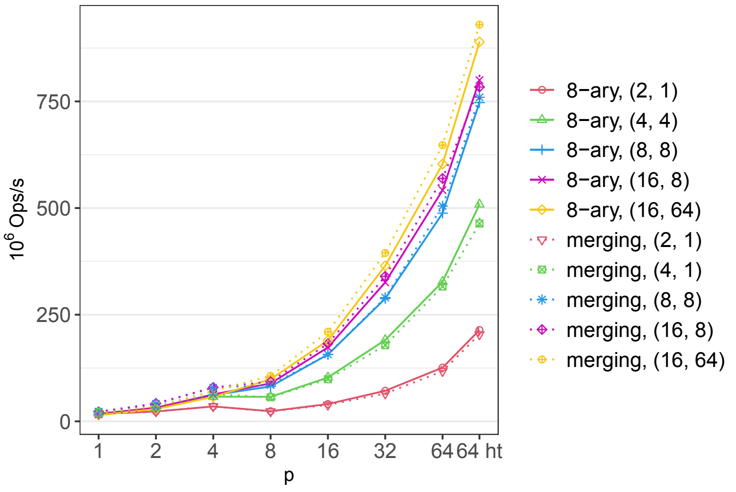

Figure 7 Throughput comparison using 8-ary heaps and k-merging heaps with $k = 16$. The numbers in parentheses are (c, s)-pairs. The suffix "ht" denotes active hyper-threading.

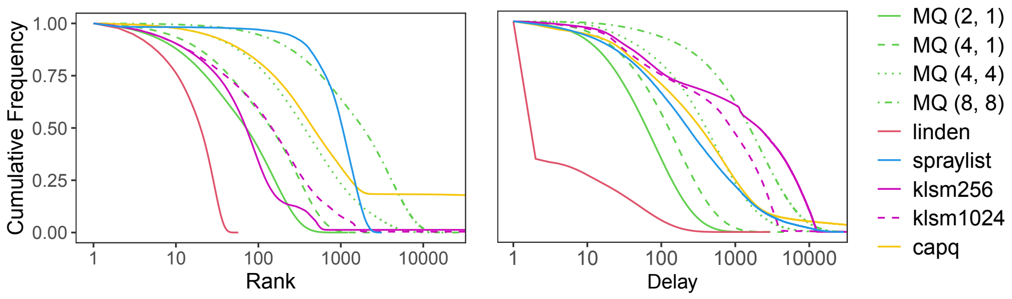

Figure 8 Comparison of rank error and delay distribution for different priority queues for 10^6 *deleteMin* operations, $p = 64$.

We used the implementation found in the Github repository of the k-LSM[3] for all of the above priority queues.

Figure 8 shows the rank error distributions and delays for 64 threads. Unsurprisingly, the Linden queue yields by far the best quality. The k-LSM achieves low rank error but high delay. For the Spraylist it is the other way around as it has high rank error but low delay. Multiqueues have similar delays and rank errors. With small values for c and s they are more accurate than all the competitors except for the Linden queue. The CAPQ exhibits quality comparable to the MultiQueue with the rather loose setting $c = s = 4$.

As seen in Figure 9, the CAPQ is among the fastest priority priority queues for up to 32 threads. Beyond that, it accesses the centralized queue so often that contention deteriorates performance. This could probably be remedied with different settings of the tuning parameters but only with a commensurate effect on even higher rank errors and delays. The very loose MultiQueue with $c = s = 8$ exhibits very good scalability. Even the higher-quality variants can take advantage of all hardware threads and eventually outperform the CAPQ. The k-LSM scales up to around 16 threads while Linden and Spraylist scale

[3] `https://github.com/klsmpq/klsm`

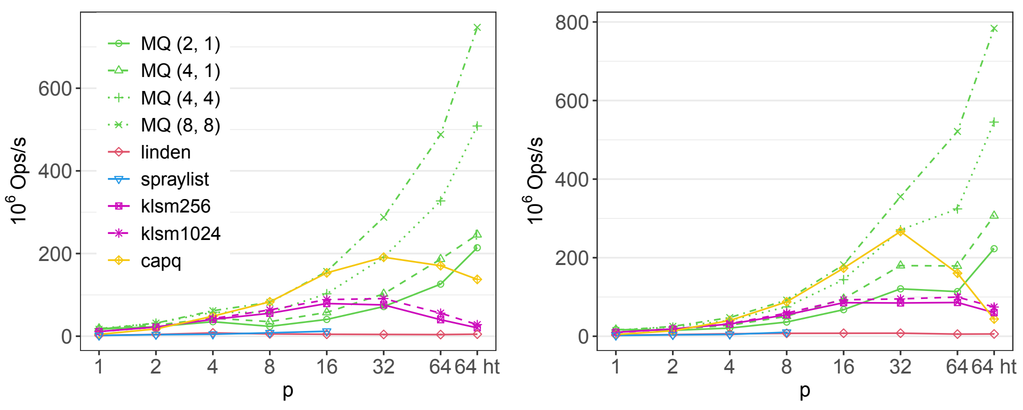

Figure 9 Throughput comparison between the MultiQueue and other implementations. On the left is machine A, and on the right machine B.

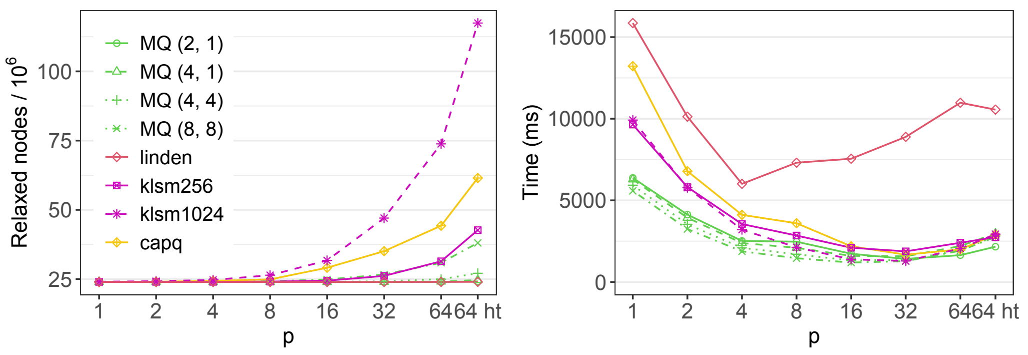

Figure 10 The SSSP benchmark on the *USA* graph running on machine A.

very poorly even on very few threads. The authors of the k-LSM show that the scalability of the k-LSM can be improved by selecting higher values for k such as 4096 at the cost of lower quality [10]. Machines A and B have similar performance with B having a slight advantage when all threads are used but suffering from the switch from one to two sockets. Unfortunately, the Spraylist crashed in our setup on both machines with higher thread count, so we had to exclude it from the SSSP benchmark below.

The parallel SSSP benchmark uses Dijkstra's algorithm to calculate the shortest paths from one node to all other nodes in a weighted graph. Dijkstra's algorithm has to be adapted to be used in a parallel setting with relaxed priority queues. Sagonas and Winblad use a similar benchmark for the CAPQ and describe the algorithm in greater detail [21]. The algorithm terminates as soon as the queue is empty. To detect when the queue is empty, they use the property of the CAPQ and k-LSM that if *deleteMin* fails for all threads some time after the last insertion happened, the queue must be empty. This does not hold for MultiQueues, thus we cannot rely solely on *deleteMin* to guarantee correct termination. Instead, we use a dedicated emptiness detection routine, where a thread checks c designated local queues for emptiness if it thinks that the queue is empty. The MultiQueue is empty if all threads have successfully completed this emptiness check some time after the last insertion has happened. For $c > 2$ the emptiness check is slightly more expensive than a *deleteMin* operation, but it is only executed after multiple failed *deleteMin*s, so the added overhead is minuscule. We report the time to solve the SSSP problem for different thread counts as

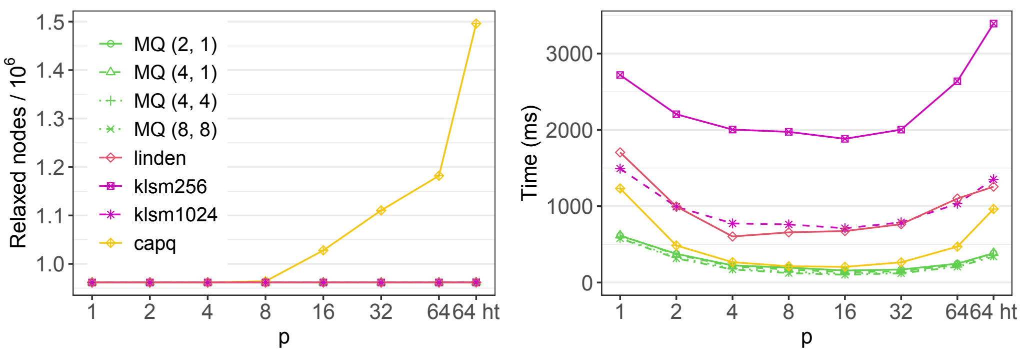

Figure 11 The SSSP benchmark on a random hyperbolic graph with 2^{20} nodes, an average degree of 16 and $\gamma = 2.3$ running on machine A.

well as the number of nodes that were extracted from the queue and then relaxed. We used real road networks and artificial random hyperbolic graphs (rhg) as benchmark instances. The road network *USA*[4] has about 24m nodes and 58m edges. The road network *GER*[5] has about 20m nodes and 42m edges. The weights on these graphs represent the travel time. We used a modified version of the KaGen framework [9] to generate random hyperbolic graphs with 2^{20} and 2^{22} nodes and the geometric distances as edge weights. Figure 10 and 11 give results for the *USA* road network and the rhg with 2^{20} nodes on machine A, respectively.

On the road map graphs, all the relaxed PQs considerably outperform the Linden queue. However, none of them scales particularly well beyond 16 threads with the MultiQueue variants performing best. Despite having considerably better rank errors than the CAPQ, the klsm1024 leads the algorithm to processes many more nodes on the *USA* graph. The k-LSM variants have very high delays, which indicates that the delay is a distinctive metric to measure the quality of relaxed priority queues. Rhgs paint a different picture: While only the CAPQ leads to nodes being processed more than once, both k-LSM variants are noticeably slower than both the CAPQ and MultiQueues. MultiQueues lead to very low overhead considering the extracted nodes in all our benchmarks and are very competitive at the same time. Especially in algorithms where processing the individual elements is expensive, this could be a decisive factor.

6 Conclusions and Future Work

MultiQueues are a simple and efficient approach to relaxed concurrent priority queues. They allow a transparent trade-off between throughput and quality and considerably outperform previous approaches in at least one of these aspects. An important open problem is to complete the theoretical analysis to encompass stickiness. We believe that further practical improvements could be possible by better avoiding conflicting queue accesses of the sticky variant even when few queues are used. It would also be interesting to make MultiQueues contention aware to achieve higher quality and more graceful degradation than the simple binary switch between local and global access used in the CAPQ [21].

A conceptual contribution of our paper is in introducing the quality measure of *delay* as a complement to *rank error*. We give evidence that this is important for the actual performance of applications such as shortest path search and explain how to measure it efficiently. We are

[4] `http://users.diag.uniroma1.it/challenge9/download.shtml`
[5] `https://i11www.iti.kit.edu/resources/roadgraphs.php`

currently having a closer look at using CPQs for branch-and-bound where it seems that the parallel performance of the applications is provably tied to the delays incurred by the CPQ. Beyond priority queues, it would be interesting to see whether our approach to "wait-free" locking can be used for other applications, e.g., for FIFOs.

References

1 Dan Alistarh, Trevor Brown, Justin Kopinsky, Jerry Z. Li, and Giorgi Nadiradze. Distributionally linearizable data structures. In *30th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 133–142, New York, NY, USA, 2018. `doi:10.1145/3210377.3210411`.

2 Dan Alistarh, Justin Kopinsky, Jerry Li, and Giorgi Nadiradze. The power of choice in priority scheduling. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 283–292, 2017.

3 Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. The SprayList: A scalable relaxed priority queue. Technical Report MSR-TR-2014-16, Microsoft Research, September 2014. URL: `http://research.microsoft.com/apps/pubs/default.aspx?id=209108`.

4 P. Berenbrink, A. Czumaj, A. Steger, and B. Vöcking. Balanced allocations: The heavily loaded case. In *32th Annual ACM Symposium on Theory of Computing*, pages 745–754, 2000.

5 Timo Bingmann. TLX: Collection of sophisticated C++ data structures, algorithms, and miscellaneous helpers, 2018. , retrieved Oct. 7, 2020. URL: `https://panthema.net/tlx`.

6 Irina Calciu, Hammurabi Mendes, and Maurice Herlihy. The adaptive priority queue with elimination and combining. *CoRR*, abs/1408.1021, 2014. URL: `http://arxiv.org/abs/1408.1021`, `arXiv:1408.1021`.

7 R. Dementiev, L. Kettner, J. Mehnert, and P. Sanders. Engineering a sorted list data structure for 32 bit keys. In *6th Workshop on Algorithm Engineering & Experiments*, pages 142–151, New Orleans, 2004.

8 N. Deo and S. Prasad. Parallel heap: An optimal parallel priority queue. *The Journal of Supercomputing*, 6(1):87–98, 1992.

9 Daniel Funke, Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Moritz von Looz. Communication-free massively distributed graph generation. In *2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS, Vancouver, BC, Canada*, 2018.

10 Jakob Gruber, Jesper Larsson Träff, and Martin Wimmer. Benchmarking concurrent priority queues: Performance of k-lsm and related data structures, 2016. `arXiv:1603.05047`.

11 Andreas Haas, Michael Lippautz, Thomas A Henzinger, Hannes Payer, Ana Sokolova, Christoph M Kirsch, and Ali Sezgin. Distributed queues in shared memory: multicore performance and scalability through quantitative relaxation. In *Proceedings of the ACM International Conference on Computing Frontiers*, page 17. ACM, 2013.

12 Thomas A Henzinger, Christoph M Kirsch, Hannes Payer, Ali Sezgin, and Ana Sokolova. Quantitative relaxation of concurrent data structures. In *ACM SIGPLAN Notices*, volume 48, pages 317–328. ACM, 2013.

13 M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

14 R. M. Karp and Y. Zhang. Parallel algorithms for backtrack search and branch-and-bound. *Journal of the ACM*, 40(3):765–789, 1993.

15 Jonatan Lindén and Bengt Jonsson. A skiplist-based concurrent priority queue with minimal memory contention. In *Principles of Distributed Systems*, pages 206–220. Springer, 2013.

16 Tobias Maier, Peter Sanders, and Roman Dementiev. Concurrent hash tables: Fast and general?(!). *ACM Transactions on Parallel Computing (TOPC)*, 5, 2019.

17 W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

18 A. Ranade, S. Cheng, E. Deprit, J. Jones, and S. Shih. Parallelism and locality in priority queues. In *Sixth IEEE Symposium on Parallel and Distributed Processing*, pages 97–103, October 1994.

19 Hamza Rihani, Peter Sanders, and Roman Dementiev. Multiqueues: Simpler, faster, and better relaxed concurrent priority queues. *CoRR*, abs/1411.1209, 2014. URL: `http://arxiv.org/abs/1411.1209`, `arXiv:1411.1209`.

20 Hamza Rihani, Peter Sanders, and Roman Dementiev. Multiqueues: Simple relaxed concurrent priority queues. In *27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 80–82, 2015. `doi:10.1145/2755573.2755616`.

21 Konstantinos Sagonas and Kjell Winblad. The contention avoiding concurrent priority queue. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 314–330. Springer, 2016.

22 Konstantinos Sagonas and Kjell Winblad. A contention adapting approach to concurrent ordered sets. *Journal of Parallel and Distributed Computing*, 115:1–19, 2018.

23 P. Sanders. Randomized priority queues for fast parallel access. *Journal Parallel and Distributed Computing, Special Issue on Parallel and Distributed Data Structures*, 49:86–97, 1998.

24 P. Sanders. Fast priority queues for cached memory. *ACM Journal of Experimental Algorithmics*, 5, 2000.

25 Peter Sanders, Kurt Mehlhorn, Martin Dietzfelbinger, and Roman Dementiev. *Sequential and Parallel Algorithms and Data Structures – The Basic Toolbox*. Springer, 2019. `doi:10.1007/978-3-540-77978-0`.

26 Nir Shavit and Itay Lotan. Skiplist-based concurrent priority queues. In *14th Int. Parallel and Distributed Processing Symposium (IPDPS)*, pages 263–268. IEEE, 2000.

27 Håkan Sundell and Philippas Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. In *Int. Parallel and Distributed Processing Symposium (IPDPS)*, pages 11–20. IEEE, 2003.

28 Peter van Emde Boas. Preserving order in a forest in less than logarithmic time. *Information Processing Letters*, 6(3):80–82, 1977.

29 J. W. J. Williams. Algorithm 232 (HEAPSORT). *Communications of the ACM*, 7:347–348, 1964.

30 Martin Wimmer, Jakob Gruber, Jesper Larsson Träff, and Philippas Tsigas. The lock-free k-lsm relaxed priority queue. *ACM SIGPLAN Notices*, 50(8):277–278, 2015.

31 Martin Wimmer, Francesco Versaci, Jesper Larsson Träff, Daniel Cederman, and Philippas Tsigas. Data structures for task-based priority scheduling. In *19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 379–380, New York, NY, USA, 2014. ACM. `doi:10.1145/2555243.2555278`.