The Lock-Free k-LSM Relaxed Priority Queue

Martin Wimmer Jakob Gruber
Jesper Larsson Träff

Faculty of Informatics, Parallel Computing
Vienna University of Technology
1040 Vienna/Wien, Austria
{wimmer,gruber,traff}@par.tuwien.ac.at

Philippas Tsigas

Computer Science and Engineering
Chalmers University of Technology
412 96 Göteborg, Sweden
tsigas@chalmers.se

Abstract

We present a new, concurrent, lock-free priority queue that relaxes the *delete-min* operation to allow deletion of any of the $\rho+1$ smallest keys instead of only a minimal one, where ρ is a parameter that can be configured at runtime. It is built from a logarithmic number of sorted arrays, similar to log-structured merge-trees (LSM). For keys added and removed by the same thread the behavior is identical to a non-relaxed priority queue. We compare to state-of-the-art lock-free priority queues with both relaxed and non-relaxed semantics, showing high performance and good scalability of our approach.

Categories and Subject Descriptors E.1 [*Data*]: Data Structures—Distributed data structures

Keywords Task-parallel programming, priority-queue, concurrent data structure relaxation, shared memory

1. Preliminaries and Ordering Semantics

A priority queue is a data structure for storing *keys* in an ordered fashion which must support insertion of keys as well as finding and deleting a minimal (or maximal) key, and may have additional operations like decreasing or deleting an arbitrary key. In a concurrent setting, the delete-min operation is an obvious scalability bottleneck, which can be addressed in various ways. We focus on semantics that provide trade-offs between scalability and linearizability guarantees on *update operations* (`insert` and `delete_min`) as well as *read operations* (`find_min`).

Afek et al. [1] introduced an alternative consistency model called *quasi linearizability* which allows operations to occur out of a correct linearizable order bounded by their *distance* in such an order. This model has later been expanded upon by Henzinger et al. [4] (*quantitative relaxation*) and Wimmer et al. [7, 8] (ρ-*relaxation*). Our data-structure ensures ρ-relaxation semantics, where `find_min` and `delete_min` are allowed to return any of the $\rho+1$ smallest keys instead of the minimal one, as well as *local ordering semantics*, where a thread will never return a key larger than the smallest key available inserted by the same thread.

PPoPP'15, February 7–11, 2015, San Francisco, CA, USA
ACM 978-1-4503-3205-7/15/02
http://dx.doi.org/10.1145/2688500.2688547

2. The Relaxed k-LSM Priority Queue

Our new priority queue is based on *log-structured merge-trees* [6]. A log-structured merge-tree (LSM) is a priority queue that operates on a logarithmic number of sorted arrays of sizes padded to the next power of two. Each array size is guaranteed to occur at most once in an LSM. All operations can be performed in $O(\log n)$ amortized time for queues with n items.

Our k-LSM priority queue consists of one thread-local LSM priority queue per thread, bounded in size by the parameter k, and one unbounded LSM shared by all threads. Once a thread-local LSM reaches the size k it is merged into the shared LSM. A `find_min` will return the smaller of the thread-local minimal key and one of the k smallest keys in the shared LSM, which is selected at random. This approach guarantees that `find_min` returns one of the $\rho = Tk$ smallest keys, where T is the number of threads.

3. Experimental Evaluation and Comparison

We have compared our lock-free k-priority queue to a state-of-the-art lock-free priority queue by Lindén and Jonsson [5], and to the recent relaxed lock-free priority queues by Alistarh et al. [2] and Wimmer et al. [8]. We show that our data structure has high absolute performance, due to its cache-efficient layout and small amount of synchronization operations, as well as good scalability.

The reported experiments were performed on a system consisting of eight Intel Xeon E7-8850 processors with 10 cores each and 1TB of main memory. All data-structures and benchmarks were implemented in C++ and compiled with `gcc 4.9.1`.

To compare with the priority queues of Lindén and Jonsson and Alistarh et al. we used the latter's *throughput benchmark* where all threads randomly insert and delete keys from a queue with a given prefill. The benchmark creates extremely high contention on the priority queue and thus provides good insights into its scalability. We report mean throughput per second for 30 10-second runs.

The throughput results are shown in Figure 1. For the k-LSM queue, the parameter k has a significant impact on both scalability and absolute performance. The larger k, the less we need to rely on the shared LSM. Performance and scalability furthermore improves with smaller prefill values, since this also reduces the reliance on the shared LSM priority queue. For large k as well as for the distributed LSM where all data is stored in thread-local LSMs, almost linear, in some cases even superlinear scalability can be observed. The absolute throughput per thread is close to the throughput for a binary heap protected by a single lock on one thread. The Lindén and Jonsson skip-list provides similar performance to but better scalability than our k-LSM with $k = 0$, but remains sensitive to the inherent scalability bottlenecks of non-relaxed priority queues. A direct comparison between the SprayList by Alistarh et al. and k-LSM is more difficult. The SprayList will return one

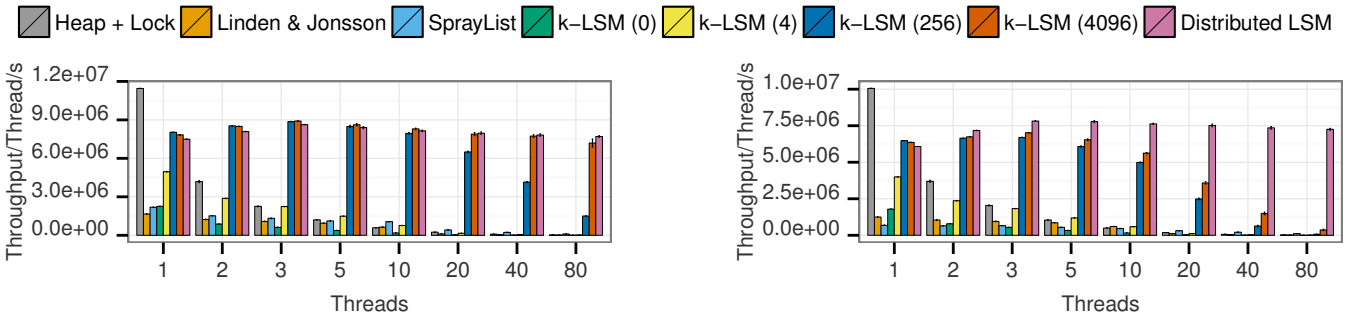

Figure 1. Throughput per thread per second for priority queues prefilled with 10^6 (left) and 10^7 (right) elements.

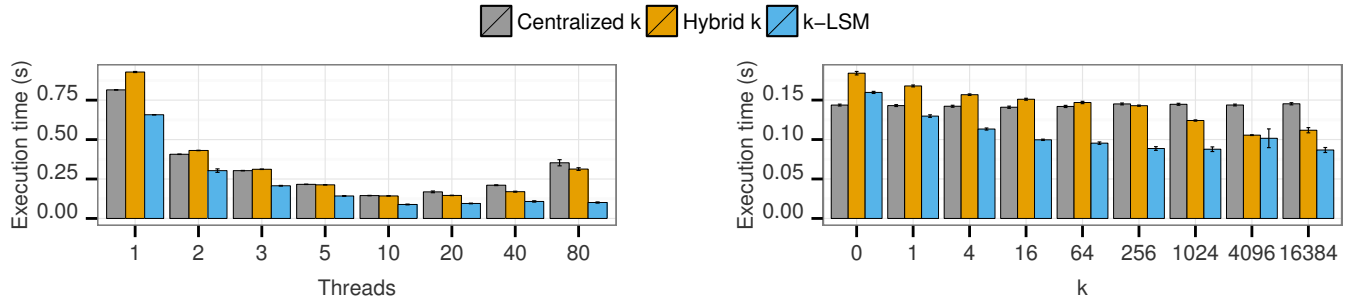

Figure 2. Execution times for SSSP benchmark for varying numbers of threads ($k = 256$) and values for k (10 threads).

of the $O(T \log_2^3 T)$ smallest keys whp. where T is the number of threads, as opposed to k-LSM, which will return one of the Tk smallest keys on `delete_min`. The relaxation of SprayList grows superlinearly with the number of threads, whereas for k-LSM it grows linearly. Assuming small constant factors for the SprayList's relaxation, we see k-LSM (4) as the closest match for small numbers of threads, and k-LSM (256) for larger numbers. The performance of both data structures is fairly similar, with k-LSM having the advantage of providing configurable relaxation as well as local ordering semantics.

The data structures by Wimmer et al. [8] are part of a task scheduling system, and cannot be used stand-alone. It is therefore not possible to do a fair comparison with the throughput benchmark. Instead, we used the *SSSP benchmark* from Wimmer et al., which is a straight-forward parallelization of Dijkstra's single-source shortest path algorithm that uses a lazy deletion scheme and reinsertion instead of an explicit `decrease_key` operation. We adapted our priority queue to support the lazy deletion scheme. Since this has a significant impact on performance, but is neither available with the Lindén and Jonsson priority queue nor the SprayList, and since there was no easy way to directly delete keys from these priority queues with the available implementations, we did not include these data structures in the SSSP benchmark. The SSSP benchmark was run on Erdős-Rényi [3] random graphs with 10000 nodes and edge probability 50%. All experiments were repeated 30 times, and mean values with confidence intervals are shown in Figure 2. We compare our k-LSM priority queue against the centralized and hybrid k-priority queues by Wimmer et al. [8]. While the algorithm of the benchmark seems to have limited scalability, our priority queue provides stable performance while the other priority queues experience a slowdown with more threads.

4. Conclusion

We sketched a novel relaxed concurrent priority queue with lock-free progress guarantees. The sequential performance is close to a binary heap, and the data structure shows good scalability for sufficiently high values of the relaxation parameter k. Comparison with recent lock-free relaxed priority queues shows that our priority queue delivers competitive and often superior performance. In contrast to the SprayList, our priority queue gives fixed and easily calculated relaxation guarantees and local ordering semantics. In comparison with the priority queues by Wimmer et al. [8] we achieve both better scalability and absolute performance for the SSSP application.

References

[1] Y. Afek, G. Korland, and E. Yanovsky. Quasi-Linearizability: Relaxed consistency for improved concurrency. In *Principles of Distributed Systems (OPODIS)*, volume 6490 of *Lecture Notes in Computer Science*, pages 395–410, 2010.

[2] D. Alistarh, J. Kopinsky, J. Li, and N. Shavit. The spraylist: A scalable relaxed priority queue. In *20st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2014.

[3] P. Erdős and A. Rényi. On random graphs. *Publicationes Mathematicae Debrecen*, 6:290–297, 1959.

[4] T. A. Henzinger, C. M. Kirsch, H. Payer, A. Sezgin, and A. Sokolova. Quantitative relaxation of concurrent data structures. In *40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 317–328, 2013.

[5] J. Lindén and B. Jonsson. A skiplist-based concurrent priority queue with minimal memory contention. In *Principles of Distributed Systems (OPODIS)*, volume 8304 of *Lecture Notes in Computer Science*, pages 206–220, 2013.

[6] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[7] M. Wimmer. *Variations on Task Scheduling for Shared Memory Systems*. PhD thesis, Vienna University of Technology, 2014.

[8] M. Wimmer, F. Versaci, J. L. Träff, D. Cederman, and P. Tsigas. Data structures for task-based priority scheduling. In *19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 379–380, 2014.