BENCHMARKING PARALLEL PRIORITY QUEUES

Bowen Wu, Fanlin Wang, Kaifeng Zhao, Kaïko Goren, Shiyi Cao

ETH Zürich
Zürich, Switzerland

ABSTRACT

Many applications make use of priority queues. Because of this, there is a natural interest in developing parallel priority queues for high performance. However, priority queues are by their nature sequential because of the uniqueness of the highest-priority element. Many existing works on this fail to benchmark the performance of their methods against strong baselines and lack a unified experimental methodology. In this project, we conduct an in-depth analysis of a variety of parallel priority queues and develop a benchmarking framework to facilitate our experiments. We devised a series of synthetic workloads and introduced a practical application to study the weak and strong scalability respectively. Our results show that most parallel priority queues do not demonstrate good scaling or better performance than single thread approaches. We also show that the performance of a parallel priority queue can be affected by key distributions, workload patterns, and sizes. Finally, we identify the need for re-thinking how parallel priority queues should be used in real applications.

1. INTRODUCTION

High-performance priority queues are fundamental to applications such as task scheduling and discrete event simulation. However, by their very nature priority queues require sequential access (Section 2), making their parallelization a considerable challenge [1]. Previous works [2, 3, 4, 5, 6] have proposed many parallel priority queue designs with different underlying structures (e.g., heaps, red-black trees and skiplists) to overcome those limitations and to improve the scalability. A common failing of those works however is the lack of a strong baseline to fairly benchmark their performance against. This as well as a more unified experimental methodology across implementations would be required to get an accurate view of priority queue parallelization efficacy.

The present work aims to develop a generic unified framework to enable a fair in-depth comparison of a variety of designs. This then allows us to explore the strengths and weaknesses of these different parallel approaches as well as their general viability under given scenarios. We first present some related works, then some general theory in Section 2. In Section 3, we detail a curated collection of popular parallel priority queues (Section 3.1) that we have included or implemented, followed by an overview of our benchmarking platform PPQBENCH (Section 3.2) and its defined benchmarks. Finally in Section 4, we discuss the observed performances and their implications for the scalability and usefulness of the queues.

Related work. There have been many performance studies on priority queues over the decades. An early work [7] proposed two workload patterns for benchmark which are the Hold model (a pop followed by a push) and the Up/Down model (a sequence of push followed by an equal sequence of pop). These workload patterns are widely used in later studies. [8] is an early attempt on benchmarking both single-threaded priority queues like Skew Heap, Median Pointer Linked List, Splay Tree and parallel versions of a subset of these queues. Due to hardware constraints back then, the size of queues in their experiments only reached up to the order of 100,000 and the parallelism is limited to 8 threads. [9] is a more recent benchmark on single-threaded priority queues with an analysis on cache performance. It includes both artificial workloads and applications such as the single source shortest paths, the min-cut problem and a packet-level network simulator. [10] conducted a benchmark on a set of recently proposed relaxed parallel priority queues. They include disparate priority distributions and a plethora of workload patterns. The result was based on operation-per-time throughput.

Both lock-based and lock-free synchronization primitives can be seen in parallel priority queue implementations. In recent years, there is a growing interest in the community on lock-free implementations like [11] and [12]. According to [11], a lock-free implementation is preferred compared to lock-based ones like [13] and [14] because of deadlocks and degraded performance problems caused by locks. The choice for the underlying data structure for a parallel priority queue is critical as suggested in [13]. For example, a binary tree, compared with a skip list, introduces more potential inferences across concurrent `pushs`. Studies in this direction include SkipList [14], chunked linked-list [15], SkipTrie [16], etc.

2. BACKGROUND

In this section, we briefly introduce parallel priority queues, the supported operations, design challenges as well as a concrete application included in our benchmark.

Parallel priority queue. A *priority queue* organizes a collection of items in a way such that the highest-priority element can be efficiently retrieved. Each item is associated with an integer *priority* attribute which is assumed here to be higher for smaller integers. A priority queue supports two main operations: `push` that inserts a new item and `pop` that removes and returns the highest-priority item. A *parallel priority queue* (PPQ) allows concurrent `push` and `pop` operations while still guaranteeing linearizability[1].

By requiring a unique global minimum we introduce an obvious bottleneck for a PPQ: making `pop` calls inherently sequential. This means that most accesses must either be made fully exclusive or require complex (and possibly slow) synchronization. Other than using global locks and optimizing the queue for single-thread access, several techniques have been developed to try and alleviate this performance issue with parallelism. The general idea is to allow for more localized synchronization while minimizing its use as much as possible. Most strategies try to use an underlying data structure that can be segmented and accessed efficiently with locks attached to individual segments [13, 14, 17]. Usually, the data structure is also designed to minimize the maintenance performance cost. Some instead attempt to use lock-free synchronization methods [2, 3, 18]. Another possible approach is to use an alternative class of PPQs that supports retrieving one of the top t minima[2] instead of a single one, thus allowing for concurrent `pop` calls [4, 1]. We call this class *relaxed PPQs* as opposed to the traditional *strict PPQs* with a single top item. While the focus here will be mainly on multi-threaded PPQs with shared memory, some research also addresses fully distributed cases that may arise because of extreme amounts of data impossible to contain in a single machine, or because of a system's design[3]. The selected examples of all those strategies are detailed in Section 3.1.

In the present study, PPQs are specifically defined as C++ classes holding a collection of items and supporting the `push` and `pop` functions. Each item holds a *priority*[4] integer field and a *value* field of any datatype. The `pop` function returns the *value* of the highest-priority item as well as a Boolean flag indicating the success of the operation. An unsuccessful operation may be caused by an empty queue or by other implementation-specific events (see Section 4).

Applications. Priority queues often appear in discrete event simulation and best-first search algorithms like A*. For the application using parallel priority queues, we focus on a trivially parallelized Dijstra's algorithm to study the speedup on a single source shortest paths problem. In this algorithm, multiple threads will refine the distances of different vertices simultaneously. In case of conflicts between two refinements, the modification with a smaller distance wins using a CAS loop. The well-known $decrease\_key$ operation in the sequential Dijkstra's algorithm is simulated by re-insertion. This creates more work but avoids complexity.

[1] Some algorithms only guarantees quiescent consistency for better performance.

[2] t is called the relaxation degree.

[3] e.g. the processors may not have any shared memory or be only connected through a remote network.

[4] In the later part, *priority* and *key* are used interchangeably.

3. SPECIFICATIONS

In this section, we present concrete implementations using the parallelization strategies presented so far and then detail the benchmarking framework developed. The full benchmarks and implementations can be found in [19].

3.1. Collection of Implementations

Tree-based methods. The most direct parallelization strategy is to keep the common heap structure for our PPQ and use either a global lock for the whole heap or finer-grained node locks. [20] proposes such a heap with a global lock for the heap size and individual ones for each node. Each operation first acquires the global lock to modify the heap size then releases it after acquiring the lock to their target node. Accesses to the structure require only the lock to their target node, then acquire those of the node's subtree iteratively layer-by-layer during the reheapify process. This allows some other operations to start before the completion of the reheapify process. However, there still exists inherent contention at the root node determined by the tree structure.

Three implementations of this nature were used in the benchmarks. The STL PQ is taken from the C++ standard library [21] with an added global lock for the whole heap, while the HUNT HEAP and MSPRIORITYQUEUE are based on fine-grained locks [20] respectively implemented naively by hand and taken from the *cds* C++ library [22, 17] with improved memory allocation and locks. We demonstrate the results of MSPRIORITYQUEUE in later experiments representing fine-grained heap since this implementation has better performance. We also explored the lock-free PPQ proposed in [6] based on a red-black binary search tree, but this structure's `pop` operation turned out to be unsound within our benchmarking scenarios and had to be dropped.

Skiplist-based methods. Skiplist-based priority queues (e.g., [14]) were proposed to eliminate the need for rebalancing and memory pre-allocation. Skiplists are search trees based on hierarchical linked lists, with a probabilistic guarantee of being balanced [14] and a possibility to use localized locks.

In this project, we implemented two lock-free versions

based on [3] (LOTAN) and [2] (LINDEN). While LOTAN is quiescently consistent, LINDEN offers a linearizable design and reduces the redundant CAS[5] operations by leveraging batch physical deletion. We optimized the implementations in two aspects. For both implementations, we use the last bit of the pointer as the marked field, as suggested in [2], so that we do not need the *AtomicMarkableReference* class and can hence reduce the overhead of construction and destruction. We also replace all the `std::vector` with simple arrays to reduce overhead. We extend them to work for generic value types and non-unique priorities so that they can be fit into our benchmarking framework.

Multi-dimensional list (MDLIST). [5] proposed a priority queue design based on a tree-like data structure that arranges nodes into higher dimensions to lessen the contention. In addition to an array of child pointers, a node stores an M-dimension coordinate vector used for searching, which is uniquely determined by a hash on the key. A larger M means fewer nodes in each dimension and hence searching is faster; however, it also increases the number of child pointers which results in more contention.

Our implementation is a lock-free version based on [18] and we extended it so that it works for any value type. We additionally used a bit vector appending to the coordinate vector to account for duplicate priority values. We also tried different memory reclamation schemes (freeing the memory once in several pop operations versus freeing memory when the program exits) and experimented on various settings for the hyper-parameters and recorded the best ones for different key ranges.

Balanced concurrent priority queue (BCPQ). This data structure, introduced in [23] is a fully distributed one that does not assume shared memory between the processing elements (PE). It instead uses RDMA[6] through Open MPI[7] to share data.

The queue is built using localized circular buffers each with a local lock and holding a segment of a global sorted list. Insertions are made with a global binary search and deletions with a sequential traversal from the root PE. The queue is balanced before insertions by having PEs pull/push chunks of data from/to their right neighbor in steps to avoid breaking memory consistency.

k-log-structured merge tree. *k-LSM* is a relaxed and lock-free PPQ based on the log-structured merge tree (LSM) [4]. It consists of per-thread local LSMs and a shared LSM. Local LSMs buffer insertions but cannot return accurate results, whereas the shared LSM enforces a better global order but remains a bottleneck for insertion. Each local LSM can contain up to k items beyond which some need to be pushed to the shared LSM. During a pop operation, the minimum is chosen out of both the local queue and the shared queue. This allows the structure to always return one of the $Tk + 1$ smallest elements, where k is a user-defined value and T is the number of threads. The benchmark uses the implementation from the original paper[8].

Intel thread building block (TBB). Intel implements a PPQ using a binary heap [24]. Instead of trying to parallelize concurrent accesses, it puts all incoming operations into a task queue. At every step, a single thread can access the task queue and fetch all the stored operations at that point. That thread is then responsible for those operations, which increases cache locality. Moreover, it achieves a constant time `push` by postponing the expensive heapify operation to the next `pop`. The `pop` operation, besides returning the minimum value, also organizes the out-of-order elements to maintain the heap property.

3.2. PPQBENCH Framework

Benchmark methodology. We have two sets of benchmarks, a synthetic one and a practical one. For the synthetic benchmark, performance is defined as the operation-per-second throughput (*weak scaling* measure). The queue is pre-populated with $100,000$ of elements. Three types of operations are counted: insertions, successful deletions and unsuccessful deletions. Unsuccessful deletions measure how fast a priority queue can check emptiness and return from the pop function call. Checking emptiness under a parallel setting is non-trivial as it either requires sequential execution or needs to check all thread local storage. To cover a wide range of practical use cases, inspired by [10] we devised three distributions of keys (the value fields are set to unused integers) and two workload patterns as illustrated in Table 1. Compared with [10], we studied different ratios of insertion-to-deletion and random keys are pre-generated *before* the benchmark[9]. In addition, we studied the effect of a varying queue size (instead of a fixed one) on performance. By letting the ratio between insertion and deletion to be $1 : 1$, we assume that the initial size of the queue equals the size of the queue throughout the benchmark runtime.

To study a practical application, we also developed a SSSP[10] benchmark, where we apply our queues to a parallel Dijsktra's algorithm. The original Dijkstra's algorithm requires modifying inserted items' priorities. This operation is simulated by re-insertion. We randomly generate a graph of $10,000$ nodes and give each pair of nodes a 0.5 probability to be connected. We reused the implementation of parallel Dijkstra's algorithm from [10]. Here the performance is computed as the time to completion (*strong scaling* measure).

[5] Compare-and-swap
[6] Remote Direct Memory Access
[7] `https://www.open-mpi.org/`
[8] `https://github.com/klsmpq/klsm`
[9] Random generation is slow and irrelevant to the PPQ's performance.
[10] Single source shortest paths

Table 1: Different settings for synthetic benchmark.

Key Distribution	
Uniform	Int keys uniformly generated from $[0, 2^{31} - 1]$.
Ascending	Int keys approximately follows a globally ascending order.
Descending	Int keys approximately follows a globally descending order.
Workload Pattern	
$p\%$-insertion	Each thread has $p\%$ chance executing an insertion for the next operation
Producer-Consumer	Only one thread will insert into the queue, while the other threads only delete the minimum from the queue.

It is worth mentioning that the discrete event simulation (DES) is also a useful application of priority queues. A parallel discrete event simulation algorithm, on the other hand, usually does not require the priority queue itself to be parallel [25, 26]. Instead, in many existing simulators (e.g, SystemC[11] and OMNeT++[12]), the priority queue is the private local data structure of a PE. Because of this, we did not include DES in our study.

Three baselines have been used. The STL PQ serves as a single-thread baseline, the MSPRIORITYQUEUE as a parallel locally locked baseline and TBB as a parallel cache-optimized baseline. To keep the comparison with *k-LSM* reasonable, its relaxation has been set to $k = 4$ and referred as 4-LSM.

4. EXPERIMENTAL RESULTS

In this section, we introduce our experimental settings and analyze the results of the benchmarks.

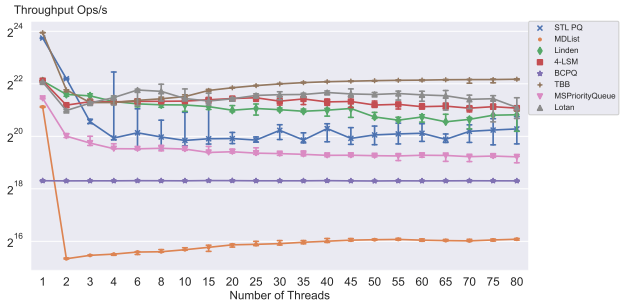
Experimental setup. All the results presented were obtained on the ETHZ Euler cluster [27]. The benchmarks were done on Euler VI nodes running on the CentOS 7 Linux operating system. The nodes are equipped with two 64-core AMD EPYC™ 7742 processors[13] (x86_64 architecture, 2.25 GHz, 64x32 KiB L1i and L1d caches, 64x512 KiB L2 caches, 16x16 MiB L3 caches), four NUMA nodes each and a peak memory bandwidth of 190.7 GiB/s each.

The sources were compiled using GCC 6.3.0 and Open-MPI 3.0.0. The flags used were: -pthread -std=c++14 -O3.

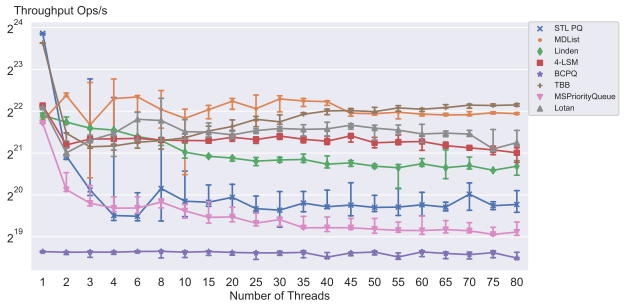
The following experiment results show the median of 10 repeated runs with error bars indicating a 95% confidence interval.

Weak scalability. We examined how different PPQs scale when the number of threads increases. Figure 1 shows a subset of results of the synthetic benchmark with varying thread numbers and 4 MPI processes for BCPQ. We present the results of 50% insertion here. The y-axis is the operation-per-second throughput and the x-axis is the number of threads. First, we observed that in all three plots, throughput drops from one thread to two threads for almost all methods. STL PQ, TBB and MSPRIORITYQUEUE have the largest performance drop (about 4-8x); 4-LSM, LOTAN, LINDEN, MDLIST and BCPQ[14] show a milder drop in performance (less than 2x). The reason for a severe performance drop is that the overuse of shared states and structures (e.g., the task queue in TBB) renders the cache utilization inefficient. Because Euler nodes have only local L1 and L2 cache and shared L3 cache between groups of four cores, single-thread runs will fully utilize L1 and L2, while any level of shared memory will require write-backs to L3 memory[15] or to the main memory for more than four threads. We also observed that a single-threaded execution of any PPQs almost always outperforms the parallel execution and a simple binary heap (STL PQ) without parallelism is unmatched for throughput. When the number of threads increases, heap-based PPQs (STL PQ, MSPRIORITYQUEUE) do not scale and skip/linked-list based priority queues (LINDEN, LOTAN, MDLIST) can at best maintain the same performance as its single thread execution. Generally speaking, lock-free skip/linked-list based priority queues have better performance than heap-based ones. TBB has good and stable performance due to its cache optimization. Interestingly, the relaxed PPQ, 4-LSM does not show significant advantages over strict PPQs. It is constantly out-performed.
Comparing Figure 1a, 1b and 1c, we can see that MDLIST and LOTAN are sensitive to different key distributions, while other implementations remain almost the same performance across all distributions. MDLIST does not perform well when the key is uniformly distributed. One possible explanation is that if two elements have similar keys, their coordinate vectors are more likely to be contiguous in memory, which makes ascending/descending distribution more friendly for cache accesses. Another reason is that the key range is several magnitudes wider in the uniform distribution and there will be more nodes in each dimension, so searching in each dimension takes much longer, in which case one may want to use a MDLIST with a larger M. The performance of LOTAN decreases for the descending key distribution. This is mostly caused by the increasing contention in the descending setting, where the insert and delete operations are more likely to access the same part of the queue.

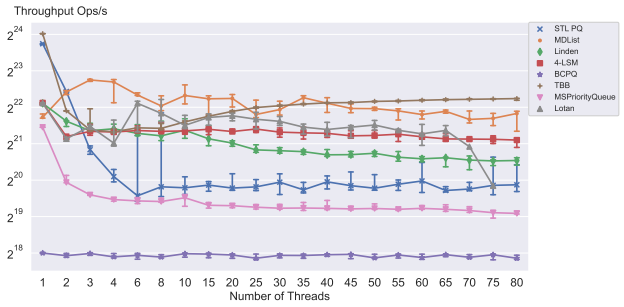
[11] https://www.accellera.org/community/systemc/
[12] https://doc.omnetpp.org/
[13] More details can be found on the product page [28].
[14] Unsurprisingly, BCPQ shows no particular gain or loss from multi-threading.
[15] L3 has at least twice the latency L2 has.



(a) Uniform key distribution and 50% insertion



(b) Ascending key distribution and 50% insertion



(c) Descending key distribution and 50% insertion

Fig. 1: 50% insertion with different key distributions

We also studied PPQs' performance for 70% and 90% insertion. The same observations above also appear for these two settings. What's more, different insertion rates can also influence performance as it is related to the size of the queue whose effect will be studied later.

Strong scalability. We studied the speedup gained from parallelism to solve the same SSSP problem. For the SSSP we only include a subset of our implementations since other implementations are either relaxed or lack necessary features for SSSP. As we can see from Figure 2, our implementations outperform the baselines MSPRIORITYQUEUE and STL PQ. MSPRIORITYQUEUE's performance illustrates the cost of high lock acquisitions rates. For TBB, LINDEN, and LOTAN, they do get a certain speedup in finishing the tasks as the number of threads increases. However, as the number of threads further increases, all queues spend a longer time

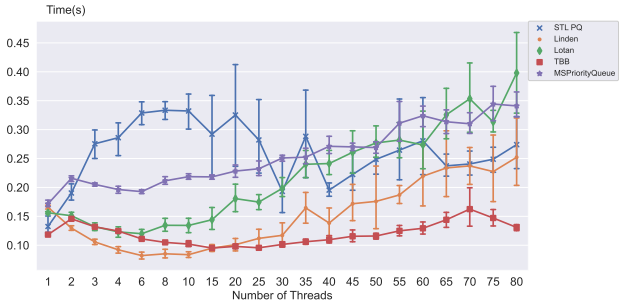


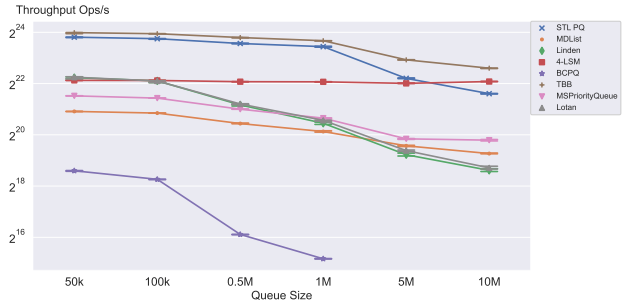
Fig. 2: Single source shortest paths

to finish the jobs, mainly because the overhead of parallelizing the priority queue outweighs the gains we obtain from parallelizing the algorithm. We can also see that LINDEN constantly outperforms LOTAN since LINDEN reduces contentions and redundant CAS operations by carrying out batch physical deletion. Moreover, as LOTAN is not strictly linearizable, when it is applied in SSSP more time will be spent in each update loop to find the shortest path for the current node. The results suggest that PPQs may not be good priority schedulers for certain parallel programs such as SSSP.

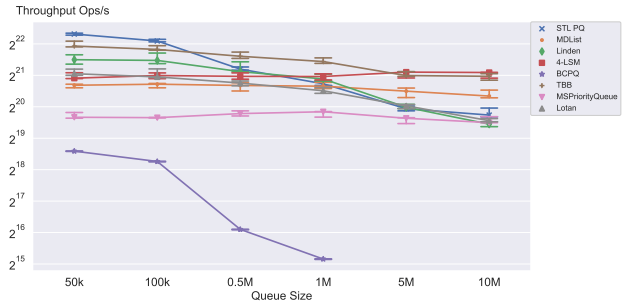
Size scalability. We studied the impact of the queue size on performance. `push` and `pop` on a large queue are normally slower since at least one of those operations must take $\Omega(\log n)$ time [29]. The performance of some structures, however, suffers more from the increasing size. We study three parallelism settings, namely 1, 2 and 40 threads. Size varies from 50 thousands to 10 million items. We keep the ratio of insertions to deletions to $1:1$ to keep the queue size close to its initial state.

The x-axis in Figure 3 is a range of different queue sizes and the y-axis is the operation-per-second throughput. Figure 3a shows the performance of all implementations running with a single thread (BCPQ uses 4 processes again). We observed that a throughput drop happens to all implementations as we increase the initial queue size except for 4-LSM. When we have two threads (Figure 3b), MDLIST and MSPRIORITYQUEUE can also maintain a similar performance across different queue sizes. With 40 threads (Figure 3c), almost all PPQs have the same performance for all different queue sizes except LOTAN and BCPQ. This is possibly because the main bottleneck is the contention caused by the increasing number of threads while the queue size has comparably small effects on the throughput.

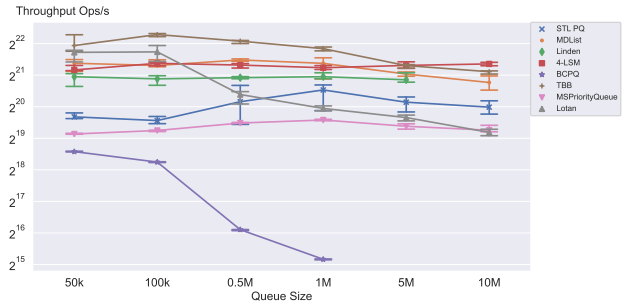
Figure 4 compares BCPQ's scaling for different numbers of PEs. Higher numbers of PEs make the PPQ initially worse but also make its performance drop slower as size increases, showing a clear relation between the queue size and the ideal number of running PEs. Its general performance however stays mediocre enough that one might question if that queue has a reasonable use-case in real-life applications.



(a) 1 thread



(b) 2 threads



(c) 40 threads

Fig. 3: Effect of queue sizes on performance

5. CONCLUSIONS

In this study, we explored the performance of multiple different parallel priority queues under both weak and strong scaling. Compared with previous work, we

1. introduced more performant baseline methods
2. compared a wider variety of strategies
3. studied the effect of the queue size on performance

Through the synthetic and SSSP benchmarks, we show that parallel priority queues do not scale in either weak or strong scaling. This is because the overhead of parallelizing priority queues obliterates the gains from parallelism for the user program. To decide which PPQ to use, key distributions, workload patterns and the scale (size) of the application

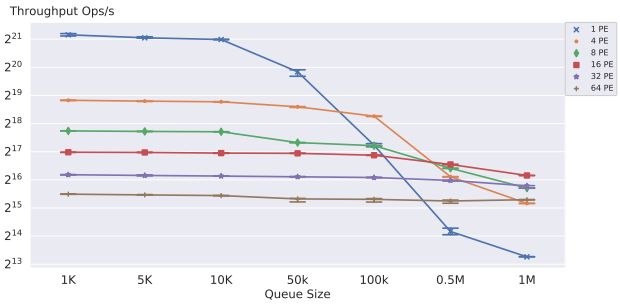


Fig. 4: BCPQ size scaling for different numbers of PEs

should be taken into account, as these factors can have an impact on the performance of some PPQs.

From this study, we realized that the single thread performance is almost always optimal. This implies that an application cannot expect to gain performance from parallelizing priority queues. Instead, a parallel application should either replace priority queues with other more suitable structures or achieve parallelism at a higher level. Moreover, a relaxed priority queue with a stricter guarantee may not be as performant. In fact, in some research, the priority queue is relaxed so drastically that it scales amazingly but its usefulness remains questionable. In addition to a better parallelism scheme, a good cache-aware and NUMA-aware design is also important since priority queues are memory-bound but this has been seemingly mostly ignored in PPQ research so far. Some priority queue designs come with inherent limitations that are non-trivial or impossible to remove. For example, the dimension M of MDLIST is determined at compile time according to the maximum size of the queue, therefore it is not suitable for workloads where the range of keys is unbounded.

We believe that in future developments of parallel priority queues, strong baselines (e.g., Intel TBB) should be included for comparison. Moreover, the parallel applications commonly used for benchmarking do not particularly benefit from PPQs. For both the single source shortest paths problem and parallel discrete event simulation, there are dedicated efficient parallel algorithms that do not use PPQs. Therefore, a lightweight priority scheduler rather than PPQs will be more desirable in modern large scale parallel applications and to best take advantage of PPQs we need to find better real-life use-cases for them[16].

6. REFERENCES

[1] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit, "The spraylist: a scalable relaxed priority queue," in

[16]Especially for distributed priority queues.

ACM Symposium on Principles and Practice of Parallel Programming (PPoPP), 2015, pp. 11–20.

[2] Jonatan Lindén and Bengt Jonsson, "A skiplist-based concurrent priority queue with minimal memory contention," in *International Conference on Principles of Distributed Systems (OPODIS)*, 2013, vol. 8304, pp. 206–220.

[3] Maurice Herlihy and Nir Shavit, *The art of multiprocessor programming*, Morgan Kaufmann, 2008.

[4] Martin Wimmer, Jakob Gruber, Jesper Larsson Träff, and Philippas Tsigas, "The lock-free k-lsm relaxed priority queue," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, New York, NY, USA, 2015, PPoPP 2015, p. 277–278, Association for Computing Machinery.

[5] D. Zhang and D. Dechev, "A lock-free priority queue design based on multi-dimensional linked lists," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 3, pp. 613–626, 2016.

[6] Jong Ho Kim, Helen Cameron, and Peter Graham, "Lock-free red-black trees using cas," *Concurrency and Computation: Practice and experience*, pp. 1–40, 2006.

[7] Jean G. Vaucher and Pierre Duval, "A comparison of simulation event list algorithms," *Commun. ACM*, vol. 18, no. 4, pp. 223–230, Apr. 1975.

[8] Robert Rönngren and Rassul Ayani, "A comparative study of parallel and sequential priority queue algorithms," *ACM Trans. Model. Comput. Simul.*, vol. 7, no. 2, pp. 157–209, Apr. 1997.

[9] Daniel Larkin, Siddhartha Sen, and Robert E. Tarjan, "A back-to-basics empirical study of priority queues," in *Proc. 16th Workshop on Algorithm Engineering and Experiments (ALENEX)*, January 2014, 12 pages.

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

[11] Håkan Sundell and Philippas Tsigas, "Fast and lock-free concurrent priority queues for multi-thread systems," *Journal of Parallel and Distributed Computing*, vol. 65, no. 5, pp. 609 – 627, 2005.

[12] Anastasia Braginsky, N. Cohen, and Erez Petrank, "Cbpq: High performance lock-free priority queue," in *Euro-Par*, 2016.

[13] Douglas W. Jones, "Concurrent operations on priority queues," *Commun. ACM*, vol. 32, no. 1, pp. 132–137, Jan. 1989.

[14] Nir Shavit and Itay Lotan, "Skiplist-based concurrent priority queues," in *International Parallel & Distributed Processing Symposium (IPDPS)*. 2000, pp. 263–268, IEEE Computer Society.

[15] Anastasia Braginsky and Erez Petrank, "Locality-conscious lock-free linked lists," in *Distributed Computing and Networking*, Marcos K. Aguilera, Haifeng Yu, Nitin H. Vaidya, Vikram Srinivasan, and Romit Roy Choudhury, Eds., Berlin, Heidelberg, 2011, pp. 107–118, Springer Berlin Heidelberg.

[16] Rotem Oshman and Nir Shavit, "The skiptrie: Low-depth concurrent search without rebalancing," in *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, New York, NY, USA, 2013, PODC '13, p. 23–32, Association for Computing Machinery.

[17] Max Khizhinsky, "cds::container::mspriorityqueue reference," online: https://scicomp.ethz.ch/wiki/Euler, December 2017.

[18] Alexander Goponenko and Steven Carroll, "A c++ implementation of a lock-free priority queue based on multi-dimensional linked list," 12 2019.

[19] Bowen Wu, Fanlin Wang, Kaifeng Zhao, Kaïko Goren, and Shiyi Cao, "Parallel queues," online: https://gitlab.ethz.ch/dphpc-quality-queuers/parallel-queues, January 2021.

[20] Galen C Hunt, Maged M Michael, Srinivasan Parthasarathy, and Michael L Scott, "An efficient algorithm for concurrent priority queue heaps," *Information Processing Letters*, vol. 60, no. 3, pp. 151–157, 1996.

[21] cppreference.com, "std::priority_queue reference," online: https://en.cppreference.com/w/cpp/container/priority_queue, January 2021.

[22] Max Khizhinsky, "libcds 2.3.3," online: https://github.com/khizmax/libcds, December 2018.

[23] M. D. Grammatikakis and S. Liesche, "Priority queues and sorting methods for parallel simulation," *IEEE Transactions on Software Engineering*, vol. 26, no. 5, pp. 401–422, 2000.

[24] Chuck Pheatt, "Intel® threading building blocks," *J. Comput. Sci. Coll.*, vol. 23, no. 4, pp. 298, Apr. 2008.

[25] K. M. Chandy and J. Misra, "Asynchronous distributed simulation via a sequence of parallel computations," *Commun. ACM*, vol. 24, no. 4, pp. 198–206, Apr. 1981.

[26] Richard M. Fujimoto, "Parallel discrete event simulation," *Commun. ACM*, vol. 33, no. 10, pp. 30–53, Oct. 1990.

[27] "Ethz wiki - euler," online: https://scicomp.ethz.ch/wiki/Euler, December 2020.

[28] "Amd epyc™ 7742," online: https://www.amd.com/en/products/cpu/amd-epyc-7742, 2021.

[29] Mikkel Thorup, "Equivalence between priority queues and sorting," *J. ACM*, vol. 54, no. 6, pp. 28–es, Dec. 2007.