in External Memory with STXXL*

Timo Bingmann, Thomas Keh, and Peter Sanders

Karlsruhe Institute of Technology, Karlsruhe, Germany
{bingmann,sanders}@kit.edu

Abstract. We propose the design and an implementation of a bulk-parallel external memory priority queue to take advantage of both shared-memory parallelism and high external memory transfer speeds to parallel disks. To achieve higher performance by decoupling item insertions and extractions, we offer two parallelization interfaces: one using "bulk" sequences, the other by defining "limit" items. In the design, we discuss how to parallelize insertions using multiple heaps, and how to calculate a dynamic prediction sequence to prefetch blocks and apply parallel multiway merge for extraction. Our experimental results show that in the selected benchmarks the priority queue reaches 64% of the full parallel I/O bandwidth of SSDs and 49% of rotational disks, or the speed of sorting in external memory when bounded by computation.

1 Introduction

Priority queues (PQs) are fundamental data structures which have numerous applications like job scheduling, graph algorithms, time forward processing [8], discrete event simulation, and many greedy algorithms or heuristics. They manage a dynamic set of items, and support operations for inserting new items (*push*), and reading and deleting (*top*/*pop*) the item smallest w.r.t. some order.

Since the performance of such applications usually heavily depends on the PQ, it is unavoidable to consider parallelized variants of PQs as parallelism is today the only way to get further performance out of Moore's law. However, even the basic semantics of a parallel priority queue (PPQ) are unclear, since PQ operations inherently sequentialize and synchronize algorithms. Researchers have previously focused on parallelizing *main memory* PQs which provide lock-free concurrent access, and/or relaxed operations delivering *some* small item.

In this work we propose a PPQ for applications where data *does not* fit into internal memory and thus requires efficient *external memory techniques*. Parallelizing external memory algorithms is one of the main algorithmic challenges termed as "Big Data". We propose a "bulk" and a "limit" parallelization interface for PQs, since the requirements of external memory applications are different from those working on smaller PQ instances. One application of these interfaces is *bulk-parallel time forward processing*, where one uses the graph's structure to

* This paper is a short version of the technical report [6].

identify layers of nodes that can be processed independently. For example, the inducing process of an external memory suffix sorting algorithm [5] follows this pattern. This paper continues work started in Thomas Keh's bachelor thesis [13].

We implemented our PPQ design in C++ with OpenMP and STXXL [9], and compare it using four benchmarks against the fastest EM priority queue implementations available. In our experiments we achieve 49% of the full I/O throughput of parallel rotational disks and 64% of four parallel solid-state-disks (SSDs) with about 2.0/1.6 GiB/s read/write performance. We reach these percentages in all experiments except when internal work is clearly the limitation, where our PPQ performs equally well as a highly tuned sorter. For smaller bulk sequences, the PPQ's performance gradually degrades, however, already for bulks larger than 20 K or 80 K 64-bit integers (depending on the platform) our PPQ outperforms the best existing parallelized external memory PQ.

After preliminaries and related work, we discuss our parallelization interfaces in Section 2. Central is our PPQ design in Section 3 where we deal with parallel insertion and extraction. Details of our implementation, the rationale of our experiments, and their results are discussed in Section 4.

1.1 Preliminaries

A PQ is a data structure holding a set of items, which can be ordered w.r.t some relation. All PQs support two operations: *insert* or *push* to add an item, and *deleteMin* or *top* and *pop* to retrieve and remove the smallest item from the set. In this paper we use the *push*, *top*, *pop* notation, since our implementation's interface aims to be compatible to the C++ Standard Template Library (STL). Addressable PQs additionally provide a *decreaseKey* operation, but we omit this function since it is difficult to provide efficiently in external memory.

We use the external memory (EM) model [21], which assumes an internal memory (called RAM) containing up to M items, and D disks containing space for N items, used for input, output and temporary data. Transfer of B items between disks and internal memory costs one I/O operation, whereas internal computation is free. While the EM model is good to describe asymptotically optimal I/O efficient algorithms, omitting computation time makes the model less and less practical as I/O throughput increases. For example, data transfer to a single modern SSD reaches more than 450 MiB/s (MiB = 2^{20} bytes), while sorting 1 GiB of random 64-bit integers sequentially reaches only about 85 MiB/s on a current machine. Thus exploiting parallelism in modern machines is unavoidable to achieve good performance with I/O efficient algorithms. For this experimental paper, we assume a shared memory system with p processors or threads, which have a simple set of explicit synchronization primitives. In future, one could consider a detailed theoretical analysis using the parallel external memory model [3].

1.2 Related Work

There has been significant work on bulk-parallel PQs in an internal memory setting [15,11,17]. We owe to the earlier of these results [15,11] the idea to replace

Listing 1. Bulk Pop/Push Loop

```
vector<item> work;
while (!ppq.empty()) {
  ppq.bulk_pop(work, max_size);
  ppq.bulk_push_begin(approx_bulk_size);
#pragma omp parallel for
  for (i = 0; i < work.size(); ++i) {
    // process work[i], maybe bulk_push()
  }
  ppq.bulk_push_end();
}
```

Listing 2. Bulk-Limit Loop

```
for (...) {
  ppq.limit_begin(L, bulk_size);
  while (ppq.limit_top() < L) {
    top = ppq.limit_top();
    ppq.limit_pop();
    // maybe use limit_push()
  }
  ppq.limit_end();
}
```

elements in a heap by sorted sequence and the basic operations on heap nodes by sorting and merging. Previous work on external sequential PQs [2,7,18] reduced the number of times an element moves between heap nodes from $\mathcal{O}(\log_2 N)$ to $\mathcal{O}(\log_{M/B} N/M)$ by increasing the degree of the involved tree structures.

There has also been a lot of work on concurrent PQs that allow asynchronous insertion and deletion by independent threads. Since this is not scalable with strict PQ semantics, there has recently been interest in concurrent PQs with relaxed semantics. Bulk-parallel PQs can be viewed as synchronous relaxed PQs with simple and clear semantics. We refer to recent work for details [1,16].

We parallelize the external *sequence heap* [18]. At the bottom level, a sequence heap consists of R groups of $k = \mathcal{O}(M/B)$ sorted external arrays. This PQ design was implemented for external memory in STXXL [9], and later also in TPIE [14], so it is probably the most widely used today. Beckmann, Dementiev and Singler [4] have partially parallelized sequence heaps without touching the sequential semantics. However, this gives only little opportunity for parallelization – mostly for merging in groups with large external arrays.

The most sophisticated parallelization tool we use in our PPQ is the parallel k-way merge algorithm first proposed by Varman et al. [20], and engineered by Singler et al. in the MCSTL [19] and later the GNU Parallel Mode library. Since this algorithm's details and implementation are important for our PPQ design, we briefly describe it: given p processors and k sorted arrays with in total n items and of maximum length m, each array is split into p range-disjoint parts where the sum of each processor's parts are of equal size. The partition is calculated by running p intertwined multisequence selection algorithms, which take $\mathcal{O}(k \cdot \log k \cdot \log m)$. After partitioning, the work of merging the p disjoint areas can be done independently by the processors, e.g., using a k-way tournament tree in time $\mathcal{O}(\frac{n}{p} \log k)$. For our EM setting it is important that the output is generated as p equal-sized parts in parallel, with each part being written in sequence. We also note that the multisequence selection is implemented sequentially.

2 Bulk-Parallel Interface and Limit Items

Before we discuss our PPQ design, we focus on the proposed application interface. As suggested by the related work on PQs, substantial performance gains from

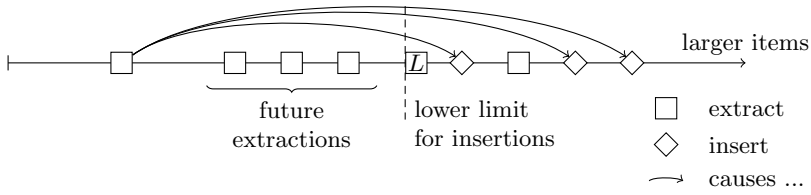

Fig. 1. Decoupling insertion and extraction operations with a limit item L.

parallelization are only achievable when loosening some semantics of the PQ. Put plainly, an alternating sequence of dependent *push*/*pop* s is inherently sequential. Since we focus on large amounts of data, the more natural relaxation of a PQ is to require insertion and extraction of multiple items, or "bulks" of items. This looser semantic *decouples* insert and delete operations both among themselves (i.e., items within a bulk) as well as the operation phases from another. This enables us to apply parallel algorithms on larger amounts of items, and our experiments in Section 4 show how speedup depends on the bulk sizes.

Thus the primary interface of our EM PPQ is bulk insertion and extraction (see Listing 1). A bulk insertion phase is started with *bulk_push_begin*(k), where k is an estimate of the bulk size. Thereafter, the application may insert a bulk of items using *bulk_push*, possibly concurrently from multiple threads, and terminate the sequence with *bulk_push_end*. There are two bulk extraction primitives: *bulk_pop*(v, k) which extracts up to k items into v, and *bulk_pop_limit*(v, L, k), which extracts at most k items strictly smaller than a limit item L. The limit extraction also indicates whether more items smaller than L are available.

Beyond the primary bulk interface, we also propose a second interface (see Listing 2), which is geared towards the canonical processing loop found in most sequential applications using a PQ: extract an item, inspect it, and reinsert zero or more items into the PQ. To decouple insertions and extractions in this loop, we let the application define a "limit item" L, and require that all insertions thereafter are larger or equal to L (see Figure 1). By defining this limit, all extractions of items less than L become decoupled from insertions. The drawback of this second interface is that the application does not process items in parallel, however, this can easily be accomplished by using *bulk_pop_limit*.

3 Design of a Bulk-Parallel Priority Queue

Our PPQ design (see Figure 2) is based on Sanders' sequence heap [18], but we have to reevaluate the implicit assumptions, duplicate data structures for independent parallel operations and apply parallel algorithms where possible. After briefly following the lifetime of an item in the PPQ, we first discuss separately how insertions and extractions can be processed in parallel, and then focus on the difficulty of balancing both.

An item is first inserted into an *insertion heap*, which is kept in heap order. As simple binary heaps are not particularly cache-efficient, they are given a fixed

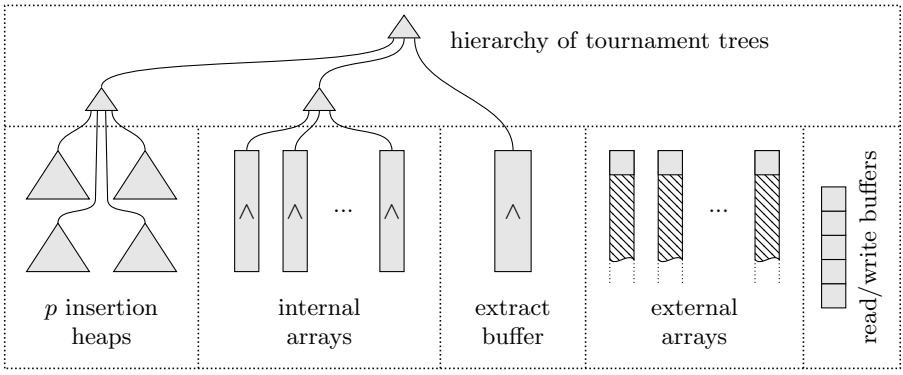

Fig. 2. Components of our PPQ. All lightly shaded parts are in internal memory.

maximum size. When full, an insertion heap is sorted and transformed into an *internal array*. To limit the number of internal arrays, they may be merged with others to form longer internal arrays. When memory is exhausted, all internal arrays and the extract buffer are merged into one sorted *external array* which is written to disk. Again, shorter external arrays may also be merged together. Extracts from the set of external arrays are amortized using the extract buffer.

Insertion, Multilevel Merging, and External Writing. To accelerate parallel push operations, the first obvious step is to have p insertion heaps, one for each processor. This decouples insertions on different processors and parallelizes the work of maintaining the heaps. Once a heap is full, the processor can independently sort the heap using a general sorter. Remarkably, these initial steps are among the most time consuming in a sequence heap, and can be parallelized well. In our PPQ design, we then use a critical section primitive to synchronize adding the new internal array to the common list. This was never a bottleneck, since such operations happen only when an insertion heap is full.

In bulk push sequences, we can accelerate individual push operations much further. While pushing, no items from the insertion heap can be extracted, thus we can postpone reestablishing heap order to *bulk_push_end*; a *bulk_push* just appends to the insertion heap's array. If the heap overflows, then the array is sorted anyway. In our experiments, this turned out to be the best option, probably because the loop sifting items up the heap becomes very tight and cache efficient. For larger bulk operations (as indicated by the user's estimation) we even let the insertion heap's array grow beyond the usual limit to fill up the available RAM, since sorting is more cache efficient than keeping a heap.

Instead of separating internal arrays into groups, as in a sequence heap, we label them using a *level number* starting at zero. If the number of internal arrays on one level grows larger than a tuning parameter (about 64) and there is enough RAM available, then all internal arrays of one level are merged together and added to the next higher level. The decisive difference of parallel multiway merge over sequentially merging sorted arrays is that *no state* is kept to amortize operations. Hence, in our PPQ design the indicated tournament trees over the

insertion heaps and arrays are useless for parallel operations. When applying parallel multiway merge, we want to have the total number of items as large as possible, however, at the same time the number of sequences should be kept as small as possible.

When the PPQ's alloted memory is exhausted, one large parallel multiway merge is performed directly into EM. This is possible without an extra copy buffer, by using just $\Theta(p)$ write buffers and overlapping I/O and computation, since parallel multiway merge outputs p sorted sub-sequences. We use $\geq 2p$ write buffer blocks to keep the merge boundaries in memory; thus avoiding any rereading of blocks from disk during the merge.

An item may travel multiple times to disk and back, since the extract buffer is included while merging into EM. However, as in the sequence heap structure, this only occurs when internal memory is exhausted and all items are written to disk; thus we can amortize the extra I/Os for the extract buffer with the $\Theta(M/B)$ I/Os needed to flush main memory.

Extraction, Prediction, and Minimum of Minima. To support fast non-bulk *pop* operations, we keep a hierarchy of tournament trees to save results of pairwise comparisons of items. The trees are built over the insertion heaps, internal arrays, and extract buffer. External arrays need not be included, since extraction from them is buffered using the extract buffer. The tournament trees need to be updated each time an insertion heap's minimum element changes, or a heap is flushed into an internal array. In bulk push operations these actions are obviously postponed until the bulk's end.

When merging external arrays with parallel multiway merge we are posed (again) with the discrepancy between parallelism, which requires large item counts for efficiency, and relatively small disk blocks (by default 2–8 MiB). To alleviate the problem, we increase the number of read buffers and calculate an *optimal block prediction sequence*, as also done for sorting [12], which contains the order in which the EM blocks are needed during merging and fetch as many as fit into RAM. In sorting, the prediction sequence is fixed and can be determined by sorting the smallest items of each block as a representative (also called "trigger" element). In the parallel disk model, the independent disks need to be considered as well. In our PPQ setting, the prediction sequence becomes a *dynamic problem*, since external arrays may be added. We define four states for an external block: in external memory, hinted for prefetching, loaded in RAM, and finished (see Figure 3). To limit the main memory usage of the PPQ, the number of prefetched and blocks loaded in RAM must be restricted.

Since the next k external blocks needed for merging are determined by the k smallest block minima, we keep track of these items in a tournament tree over the block minima sequences of the external arrays (see items h_i in Figure 3). This allows fast calculation of the next block when another can be prefetched. However, when a new external array is added, the dynamic prediction sequence changes, and we may have to cancel prefetch hints. This is done by resetting the tournament tree back to the first block minima merely hinted for prefetching, but not loaded in RAM, and replaying it till the new k smallest block minima are

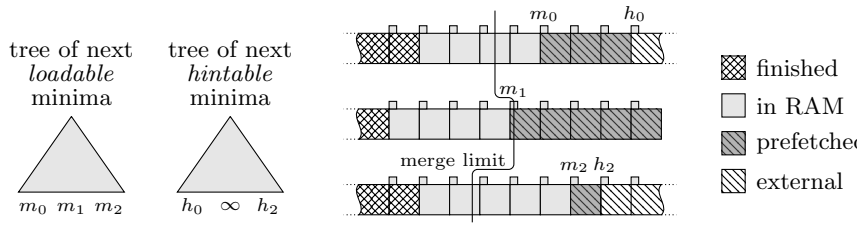

Fig. 3. Establishing the dynamic prefetching sequence and upper merge limit.

determined. This costs less than $k + k \log S$ comparisons, where S is the number of sequences. We then compare the new predictions with the old by checking how many blocks are to be prefetched in each array, and cancel or add hints.

For parallel merging, we need to solve another problem: the merge ranges within the blocks in RAM must be limited to items smaller than the smallest item still in EM, since otherwise the PQ invariant may be violated. To determine the smallest item in EM we reuse the block minima sequences, and build a second tournament tree over them containing the smallest items of the next "loadable" block, not guaranteed to be in RAM (items m_i in Figure 3). When performing a parallel multiway merge into the extract buffer, all hinted external blocks are first checked (in order) whether the prefetch is complete, and the tournament tree containing the smallest external items is updated. The tip then contains $\overline{m} = \min_i m_i$, the overall smallest external item, which serves as *merge limit*. We then use binary search within the loaded blocks of each array and find the largest items smaller than $\overline{m}$. We thus limit the multisequence selection and merge range on each array by $\overline{m}$. Additionally, by using smaller selection ranks during parallel multiway merging one can adapt the total number of elements merged. These rank limits enable us to efficiently limit the extract buffer's size and the output size of *bulk_pop*(v, k) and *bulk_pop_limit*(v, L, k) operations. To limit extraction up to L, we simply use $\min(L, \overline{m})$ as merge limit.

As with internal arrays, the number of external arrays should be kept small for multiway merge to be efficient. One may suspect that merging from EM is I/O bound, however, if the merge output buffers are smaller than the read buffers, then this is obviously not the case. Thus, the parallelization bottleneck of refilling the extract buffer or of *bulk_pop* operations largely depends on the number of arrays. We also adapt the number of read buffers (both for prefetching and holding blocks in RAM) dynamically to the number of external arrays. Each newly added external array requires at least one additional read buffer.

As with internal arrays, instead of keeping external arrays in separate groups, we label them with a level number, and merge levels when the contained number grows too large. This enables more dynamic memory pooling than in the rigid sequence heap data structure, while maintaining the optimal I/O complexity.

Trade-Offs between Insertion and Extraction. As already discussed, to enable non-bulk *pop* operations we keep a hierarchy of tournament trees. Using this hierarchy instead of one large tournament tree skews the depth of nodes

in the tree, making replays after *pop*s from the extract buffer and the insertion heaps cheaper than from internal arrays.

When a new external array is created, then the read prediction sequence may change and previous prefetch requests need to be canceled and new ones issued. In long bulk push sequences (as the ascending sequence in our experiments), this can amount to many superfluous prefetch reads of blocks. Thus we disable prefetching during *bulk_push* operations and issue all hints at the end. This suggests that bulk push sequences should be as long as possible, and that they are interleaved with *bulk_pop* operations.

4 Implementation in STXXL and Experimental Results

We implemented our PPQ design in C++ with OpenMP and the STXXL library [9], since it provides a well-designed interface to asynchronous I/O, and allowing easy overlapping of I/O and computation. It also contains two other PQ implementations that we compare our implementation to. Our implementation will be available as part of the next STXXL release 1.4.2, which will be publicly available under the liberal Boost software license. At the time of submission it is available in the public development repository.

Other PQ Implementations. In these experiments we compare our PPQ implementation (**PPQ**) with the sequential sequence heap [18] (**SPQS**) contained in the STXXL, a partially parallelized version [4] of it (**SPQP**), which uses parallel multiway merging only when merging external arrays, and with the STXXL's highly tuned stream sorting implementation [10] (**Sorter**) as a baseline.

Experimental Platforms. We run the experiments on two platforms. Platform **A-Rot** is an Intel Xeon X5550 from 2009 with 2 sockets, 4 cores and 4 Hyperthreading cores per socket at 2.66 GHz clock speed and 48 GiB RAM, and eight rotational Western Digital Blue disks with 1 TB capacity and about 127 MiB/s transfer speed each, which are attached via an Adaptec ASR-5805 RAID controller. Platform **B-SSD** is an Intel Xenon E5-2650 v2 from 2014 with 2 sockets, 8 cores and 8 Hyperthreading cores per socket at 2.6 GHz clock speed with 128 GiB RAM. There are four Samsung SSD 840 EVO disks with 1 TB each attached via an Adaptec ASA-7805H Host adapter, yielding together 2 GiB/s read and 1.6 GiB/s write transfer speed to/from EM. The platforms run Ubuntu Linux 12.04 and 14.04, respectively, and all our programs were compiled with gcc 4.6.4 and 4.8.2 in *Release* performance mode using STXXL's CMake build system.

Experiments and Parameters. To compare the three PQs we report results of four sets of experiments. In all experiments the PQ's items are plain 64-bit integer keys (8 bytes), which places the spotlight on internal comparison work as payload only increases I/O volume. (See our report [6] for additional results with 24 byte items.) The PQs are allotted 16 GiB of RAM on both platforms, since in a real EM application multiple data structures exist simultaneously and thus have to share RAM.

In the first two experiments, called a) *push-rand-pop* and b) *push-asc-pop*, the PQ is filled a) with n uniformly random generated integer items, or b) with n ascending integers, and then the n items are extracted again. In these canonical benchmarks, the PQ is used to just sort the items, but it enables us to compare the PQs against the highly optimized sorting implementation, which also employs parallelism where possible. In the ascending sequence, the first items inserted are removed first, forcing the PQs to cycle items. Considering the amount of internal work, the *push-asc-pop* benchmark is an easy case, since all buffers are sorted and merging is skewed. Thus the focus of this benchmark is on I/O overlapping. On the other hand, in the *push-rand-pop* benchmark the internal work to sort and merge the random numbers is very high, which makes it a test of internal processing speed. We ran the experiments for $n = 2^{27}, \ldots, 2^{35}$, which is an item volume of 1 GiB, ..., 256 GiB.

The third and forth experiments, *asc-rbulk-rewrite* and *bulk-rewrite*, fully rewrite the PQ in bulks: the PQ is filled with n ascending items, then the n items are extracted in bulks of random or fixed size v, and after each bulk extraction v items are pushed again. During the rewrite, in total n items are extracted and n items inserted with higher ids. We measure only the bulk pop/push cycles as these experiments are designed to emulate traversing a graph for time forward processing. We use bulk rewriting in two different experiment scenarios: in the first, we select the bulk size uniformly at random from 0 to 640 000, and let n increase as in the first two experiments. For the second, $n = 4 \cdot 2^{30}$ items (32 GiB) is fixed and the bulk size v is varied from 5 000 to 5 120 000.

All experiments were run only once due to long execution times and little variation in the results over large ranges of input size. During the runs we pinned the OpenMP threads to cores, which is important since it keeps the insertion heaps local. Due to the large I/O bandwidth of the SSDs, we increased the number of write buffers of the PPQ to 2 GiB on B-SDD to better overlap I/O and computation. Likewise, we allotted 128 MiB read buffers per external array. On A-Rot we set only 256 MiB write buffers and 32 MiB read buffers per array. For the STXXL PQ, of the 16 GiB of RAM one fourth is allocated for read and one fourth for write buffers. We used in all experiments the new "linuxaio" I/O interface of STXXL 1.4.1, which uses system calls to Linux's asynchronous I/O interface with native command queuing (NCQ) and bypasses system disk cache.

Results and Interpretation. The results measured in our experiments are shown in Figure 4 as throughput in items per second. We measured "throughput" at the PQ interface, and this is not necessarily the I/O throughput to/from disk, since the PQs may keep items in RAM. In all four experiments, items are read or written *twice*, so throughput is two times item size divided by time. If one assumes that a container writes and reads all items once to/from disk (as the sorter does), then on A-Rot at most 39 million items/s and on B-SSD at most 106 million item/s could be processed, considering the maximum I/O bandwidth as measured using `stxxl_tool`.

In all our experiments, except the bulk size benchmark, our PPQ is faster than the parallelized and sequential STXXL PQ. Assuming the PQs use 12 GiB

of the 16 GiB RAM for storing items, then the containers only need EM for about $n \geq 2^{30.5}$ (indicated by dashed horizontal line in plots). In Table 1 we show the average execution time speedups of our PPQ for the available competitors, averaged over all inputs where the input cannot fit into RAM. Remarkably, on both platforms the PPQ is faster than the sorter for both inputs except random on A-Rot, which indicates that I/O overlaps computation work very well, often even better than the sorter. Comparing to SPQS, we achieved speedups of 3.6 – 4.7 on A-Rot (which has 8 real cores), and speedups of 3.4–6.7 on B-SSD (16 real cores). Compared to the previously parallelized SPQP, we only gain 1.4 –1.9 on A-Rot and 2.2 – 4.3 speedup on B-SSD. While this relative comparison may not seem much, by comparing the PPQ's throughput to the sorter and the absolute I/O bandwidth of the disks, one can see that the PPQ reaches 64% of the available I/O bandwidth in *push-asc-pop* on B-SSD, and 49% on A-Rot. For *asc-rbulk-rewrite* the PQ-throughput is naturally higher than the possible I/O bandwidth, since the PQs keep items in RAM. In *push-rand-pop*, the PPQ is limited by compute time of sorting random integers, just as the STXXL sorter is. For *asc-rbulk-rewrite*, which is a main focus of the PPQ, we achieve a speedup of 1.9 on A-Rot and 2.7 on B-SSD for bulk sizes of on average 320 000 items. Considering the increasing bulk sizes in *bulk-rewrite*, we see that larger bulks yield better performance up to a certain sweet spot, but the break even of the PPQ over the SPQP is quite low: 20 K items for A-Rot and 80 K items for B-SSD.

5 Conclusions and Future Work

We presented a PPQ design and implementation for EM, and successfully demonstrated that for specific benchmarks the high I/O bandwidth of parallel disks and even SSDs can be utilized. By relaxing semantics, our bulk-parallel interface enables parallelized processing of larger amounts of items in the PPQ. In the future, we want to apply our PPQ's bulk-parallel processing to the eSAIS external suffix and LCP sorting algorithm [5], where in the largest recursion level each alphabet character (and repetition count) is a bulk.

During our work on the PPQ two important issues remained untouched: how does one balance work in an EM algorithm library when the user application, the EM containers, and I/O overlapping require parallel work? We left this to the operating system scheduler and block the user application during parallel merging,

	Platform A-Rot			Platform B-SSD		
Experiment	SPQP	SPQS	Sorter	SPQP	SPQS	Sorter
push-rand-pop	1.39	3.58	0.87	2.25	4.83	0.83
push-asc-pop	1.81	3.40	1.37	4.29	6.71	1.20
asc-rbulk-rewrite	1.89	4.70		2.91	3.43	

Table 1. Speedup of PPQ over parallelized STXXL PQ (SPQP), sequential STXXL PQ (SPQS), and STXXL Sorter for 64-bit integers, averaged for experiments $n \geq 2^{30.5}$.

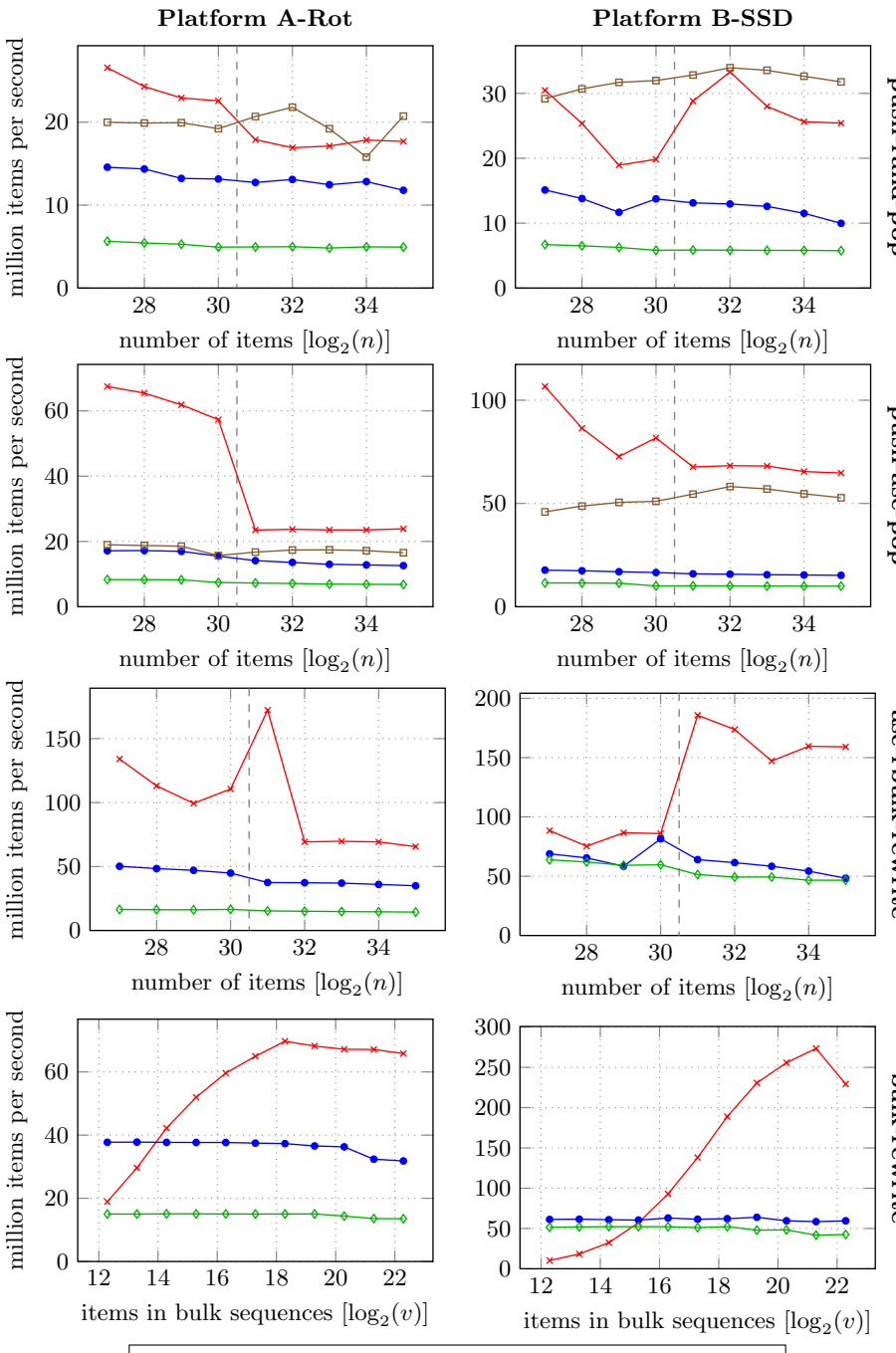

Fig. 4. Experimental results of our four benchmarks with 64-bit integer items.

which is not desirable. As indicated by theory and experiments, *bulk_pop_limit* requires large bulks to work efficiently, however, the PPQ cannot know the resulting bulk sizes without performing a costly multisequence selection. One could require the user application to provide an estimate of the resulting size, or develop an online oracle. Finally, experiments with other internal memory PPQs and d-ary heaps may improve performance by using larger insertion heaps.

References

1. Alistarh, D., Kopinsky, J., Li, J., Shavit, N.: The SprayList: A scalable relaxed priority queue. Tech. Rep. MSR-TR-2014-16, Microsoft Research (September 2014)
2. Arge, L.: The buffer tree: A technique for designing batched external data structures. Algorithmica 37(1), 1–24 (2003)
3. Arge, L., Goodrich, M.T., Nelson, M., Sitchinava, N.: Fundamental parallel algorithms for private-cache chip multiprocessors. In: SPAA. pp. 197–206. ACM (2008)
4. Beckmann, A., Dementiev, R., Singler, J.: Building a parallel pipelined external memory algorithm library. In: IPDPS'09. pp. 1–10. IEEE (2009)
5. Bingmann, T., Fischer, J., Osipov, V.: Inducing suffix and LCP arrays in external memory. In: ALENEX'13. pp. 88–102. SIAM (2013)
6. Bingmann, T., Keh, T., Sanders, P.: A bulk-parallel priority queue in external memory with STXXL (Apr 2015), see ArXiv e-print arXiv:1504.00545
7. Brodal, G.S., Katajainen, J.: Worst-case efficient external-memory priority queues. In: SWAT'98. LNCS, vol. 1432, pp. 107–118. Springer (1998)
8. Chiang, Y.J., Goodrich, M.T., Grove, E.F., Tamassia, R., Vengroff, D.E., Vitter, J.S.: External-memory graph algorithms. In: SODA'95. pp. 139–149. SIAM (1995)
9. Dementiev, R., Kettner, L., Sanders, P.: STXXL: Standard template library for XXL data sets. Software & Practice and Experience 38(6), 589—-637 (2008)
10. Dementiev, R., Sanders, P.: Asynchronous parallel disk sorting. In: SPAA'03. pp. 138–148. ACM (2003)
11. Deo, N., Prasad, S.: Parallel heap: An optimal parallel priority queue. The Journal of Supercomputing 6(1), 87–98 (1992)
12. Hutchinson, D.A., Sanders, P., Vitter, J.S.: Duality between prefetching and queued writing with parallel disks. SIAM Journal on Computing 34(6) (2005)
13. Keh, T.: Bulk-parallel priority queue in external memory (2014), Bachelor Thesis, Karlsruhe Institute of Technology, Germany
14. Petersen, L.H.: External Priority Queues in Practice. Master's thesis, Aarhus Universitet, Datalogisk Institut, Denmark (2007)
15. Pinotti, M.C., Pucci, G.: Parallel priority queues. IPL 40(1), 33–40 (1991)
16. Rihani, H., Sanders, P., Dementiev, R.: Multiqueues: Simpler, faster, and better relaxed concurrent priority queues. arXiv preprint arXiv:1411.1209 (2014)
17. Sanders, P.: Randomized priority queues for fast parallel access. Journal of Parallel and Distributed Computing 49(1), 86–97 (1998)
18. Sanders, P.: Fast priority queues for cached memory. JEA 5, 7 (2000)
19. Singler, J., Sanders, P., Putze, F.: MCSTL: The multi-core standard template library. In: Euro-Par 2007 Parallel Processing, pp. 682–694. Springer (Jan 2007)
20. Varman, P.J., Scheufler, S.D., Iyer, B.R., Ricard, G.R.: Merging multiple lists on hierarchical-memory multiprocessors. Journal of Parallel and Distributed Computing 12(2), 171–177 (1991)
21. Vitter, J.S., Shriver, E.A.: Algorithms for parallel memory, i: Two-level memories. Algorithmica 12(2-3), 110–147 (1994)