Bachelor Thesis

Bulk-Parallel Priority Queue in External Memory

Thomas Keh

Submission date: July 11, 2014

Supervisors: Dipl. Inform. Timo Bingmann
Prof. Dr. Peter Sanders

Institute of Theoretical Informatics, Algorithmics
Department of Informatics
Karlsruhe Institute of Technology

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Karlsruhe, den 11. Juli 2014

Abstract

We present a priority queue implementation with support for external memory. The focus of our work has been to derive a benefit from parallel shared-memory machines. It's the first parallel optimization of an external-memory priority queue. An additional bulk insertion interface accelerates longer sequences of homogeneous operations, as they are more likely to occur in applications that process large amounts of data. The algorithm will be available as an extension to the Stxxl [6], a popular C++ template library for extra large data sets. Experiments have shown great improvements over the current external-memory priority queue of the Stxxl for homogeneous bulk operations. However, the high overhead for spawning threads, as well as the need for cache synchronization in the global ExtractMin operation, show the inherent limitations of the parallelizability of priority queues.

Zusammenfassung

Wir präsentieren eine Priority Queue mit Unterstützung für externen Speicher. Besonderes Augenmerk wurde darauf gelegt, Vorteile aus parallelen Rechnerarchitekturen mit gemeinsamem Speicher zu ziehen. Es ist die erste parallele Optimierung einer Priority Queue für externen Speicher. Eine zusätzliche Schnittstelle zum blockweisen Einfügen beschleunigt längere Sequenzen von gleichartigen Operationen, wie sie besonders bei Anwendungen auftreten, die große Datenmengen verarbeiten. Der Algorithmus wird als Erweiterung zur Stxxl [6] verfügbar sein, einer bekannten C++-Templatebibliothek für sehr große Datenmengen. Für homogene, blockweise Operationen ergibt sich eine deutliche Verbesserung gegenüber der aktuellen Stxxl Priority Queue für externen Speicher. Die hohen Fixkosten bei der Threaderzeugung, sowie der hohe Aufwand für Cache-Synchronisierung bei der globalen ExtractMin-Operation, zeigen jedoch die inhärenten Grenzen der Parallelisierbarkeit von Priority Queues auf.

Contents

List of Figures

List of Algorithms

List of Tables

1. Introduction

Priority queues (PQ) are important data structures which have numerous applications like job scheduling, graph algorithms (e.g. Dijkstra's shortest path algorithm [7]), discrete event simulation, time forward processing [23], or online sorting.

Since the performance of these algorithms often heavily depend on the one of the priority queue, researchers have payed much attention to improving the performance by making use of parallel machines [9, 16, 15, 21]. There are different approaches to this topic. Some studies dealt with priority queues that can handle concurrent calls from different threads [21], especially without the use of inefficient locking mechanisms. This can be useful for scheduling algorithms that distribute jobs among multiple independent threads. Others distribute not the accesses, but the data among different memory areas or even different machines with NUMA [14, 16].

Furthermore, some modern applications, like very large instances of graph problems or large-scale simulations, might process data that doesn't fit into internal memory entirely. If these applications use regular internal-memory PQs, great performance losses due to paging activity is to be expected. There has been much work on efficient use of external memory in PQs, too [4, 3, 18, 8]. Their memory access pattern is tuned for the use in a two-level memory architecture with a large, but slow, high-latency external storage.

In this work we combine both concepts into an algorithm which makes heavy use of parallelism and has efficient support for external memory. Main ideas for the external memory part come from [18]. For parallel usage, we first discuss problems which occur in the context of uncoordinated, concurrent PQ accesses. Then we introduce additional bulk interfaces which accelerate longer sequences of homogeneous operations, as they are more likely to occur in applications that process large amounts of data. The algorithm will be available as an extension to the Stxxl, a popular C++ template library for extra large data sets [6].

1.1. Overview

Chapter 2 introduces basic definitions and machine models. Chapter 3 first gives an overview over other parallel priority queues and deduces different definitions of a *parallel* priority queue from them. It discusses synchronization issues with uncoordinated concurrent accesses and possible solutions to them, as well as further options to make wider use of parallelism with a more relaxed definition of an ExtractMin operation. Following this, chapter 4 presents some state of the art external-memory priority queues and points out main objectives and concepts for an external-memory PQ. Chapter 5 presents the bulk-parallel priority queue in external memory that has originated from this work. Implementation notes are given in chapter 6, and experimental results in chapter 7. Main conclusions and an outlook for future work can be found in chapter 8. Furthermore, the appendix contains an extra chapter on tournament trees which have a key role in our algorithm.

2. Preliminaries

This chapter gives a definition of a regular priority queue, together with some common variants. Furthermore, models for the memory hierarchy and parallel execution are introduced.

2.1. Priority Queues

A priority queue (PQ) is a container which maintains a set of elements, each one consisting of a priority value (also called key), and additionally some satellite information. It supports at least two operations: The ExtractMin operation returns the element with the smallest key (also called the *minimum element*) and removes it from the queue afterwards. An Insert operation inserts a new item into the set. Some priority queues also have a GetMin operation which returns the minimum without removing it from the queue. However, this can easily be simulated using an one element buffer of extracted elements. Some applications, like Dijkstra's shortest path algorithm [7], require the ability to change the priority value of particular elements. A PQ can meet this requirement with a DecreaseKey operation, which sets the element's key to a new (and usually lower) value.

There are various priority queues for internal memory. They differ in their time complexities, their practical performance in different situations, or in the set of supported operations.

Nearly all of them allow GetMin in constant time and ExtractMin, Insert, and DecreaseKey in logarithmic time. Probably the most popular one is the binary heap. Ready-to-use implementations are available in a wide range of libraries, like the Standard Template Library (STL) in C++. The d-ary heap is similar to the binary heap, but the nodes have d instead of two children. It has been observed, that they perform better on systems with cached memory [12]. Fibonnaci heaps have the advantage of an amortized constant time complexity for Insert and ExtractMin.

For use cases complying with the following restrictions, a radix heap might be the best solution: The keys have to be bounded, non-negative integer values, and the sequence of extracted values must be non-decreasing. Priority queues with the latter property are called *monotone*. A radix heap stores its elements in buckets according to the bit-representation of their keys. Insert is possible in constant time ($\mathcal{O}$(number of buckets)), but ExtractMin is very sensitive to the key distribution (i.e. the bucket sizes).

2.2. External Memory

Modern computer systems contain different memory types in a hierarchical order of increasing size, decreasing speed and decreasing cost per capacity from top to bottom. On ordinary x86-64 systems, there are some registers and a small core-local cache for each processing unit (PU), a shared cache for all cores, and the main memory (also called internal memory (IM)). They are complemented by one or more hard disk drives (HDD) as an external-memory system (EM) with large capacity, but also high latency and lower bandwidth. An illustration for this typical architecture is given in figure 1.

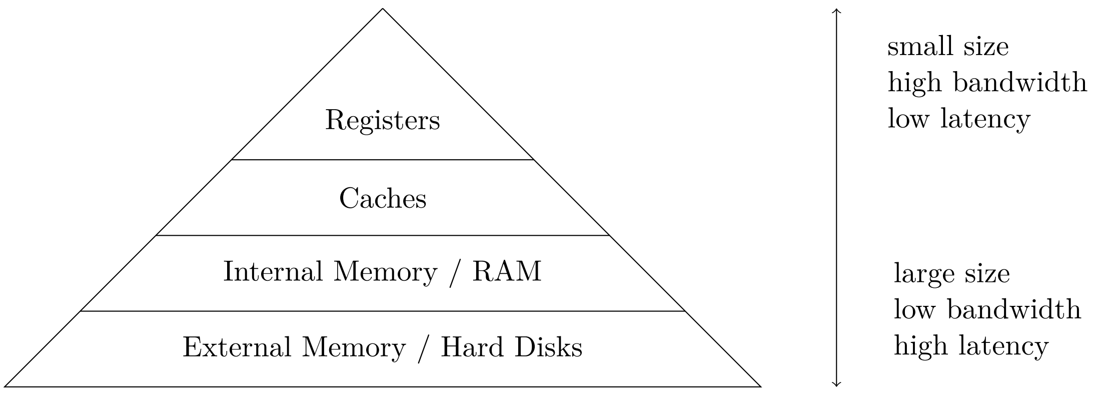


Figure 1: A typical memory hierarchy

Since the internal-memory access time is about 10^5 to 10^6 times faster than the hard disk access time [3], some modern systems replace hard disks by faster solid-state drives (SSD). Unfortunately, HDDs are not completely substitutable since their capacity of currently up to 6 TiB, as well as their low price per capacity, is unrivaled (under 3€ per GiB in some cases (Germany, 2014)). Moreover, regardless of the price, SSDs still don't obviate the need for external-memory algorithms since SSDs are slower than internal memory, too, and they often use the same block-oriented interface like HDDs (SATA, SAS). Table 1 shows some average data rates of different levels in the memory hierarchy.

The CPU-local caches will be of great importance when having concurrent memory accesses. More on this can be found in chapter 2.3. When analyzing external memory algorithms theoretically, a two-level memory model without the caches is mostly sufficient. We use the Parallel Disk Model (PDM) introduced in [10]. It assumes one fast and relatively small internal memory, supplemented by one or more hard disks which only support block-wise transfers. In particular, the model is described by the following parameters:

(i) N is the input size
(ii) M is the size of internal memory
(iii) B is the block transfer size
(iv) D is the number of independent hard disks
(v) P is the number of processor cores.

Using this model we are able to make statements about the I/O complexity of our algorithm by counting the block transfers in relation to the other parameters.

There is a drawback of this model: It does not distinguish between random block accesses and consecutive block accesses, although common hard disks are significantly slower when accessing a random block, as their read/write heads have to move to the corresponding position first. However, we make sure that all hard disk accesses are in a consecutive order.

	Dimension	Access time / Bandwidth
Registers	some Bytes	one CPU-cycle
L1-Cache	128 KiB	600 GiB/s
L2-Cache	1 MiB	200 GiB/s
L3-Cache	6 MiB	100 GiB/s
Main memory	Gigabytes	20 GiB/s
Solid-state drives	Gigabytes	600 MiB/s
Disk storage	Terabytes	100 MiB/s

Table 1: Sizes and data rates for different levels in an exemplary memory hierarchy [20, 22]

2.3. Parallelism

We assume a parallel machine with P independent processing units (also: PU, core, CPU). Of course, our algorithms also work with single-core machines ($P = 1$). The CPUs all have access to a shared memory area, called internal memory or RAM in this work. Furthermore we assume a separate cache for each core. This model is illustrated in figure 2.

If the processing units read or write memory locations, which no other cores access at the same time, the access is similar to one on a single-core machine model. If there is, however, synchronized access to a memory location from at least two cores, great performance losses are to be expected, since the cache coherence protocol must negotiate for write accesses.

This characteristic has great influence on the design of parallel algorithms: Synchronized accesses are very expensive in a parallel program and might decrease the possible parallel speedup S_p of an algorithm. It's often better to avoid synchronization and, instead, distribute *independent* work on separate memory areas among the PUs where possible.

In the case there are still concurrent write accesses to the same memory location, the use of atomic primitives might be indicated in order to avoid race conditions. Atomic primitives are sets of operations that change the state of the system and are executed isolated from concurrent processes. The success is not affected by other threads that access the same memory location. Three common atomic operations which are available as a machine instruction on x86-64 CPUs are listed below.

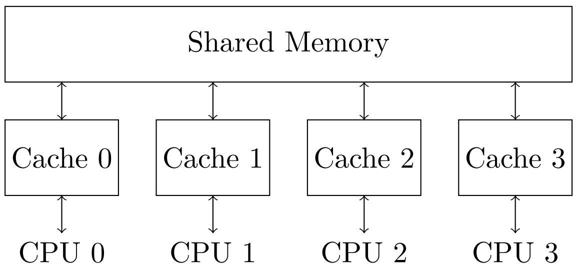


Figure 2: Parallel machine model with cached shared memory

- CompareAndSwap(&a, b, c)

  Compares the value at memory location &a[1] with b. If they are the same, the value at memory location &a will be set to c. CompareAndSwap allows for simple, thread-safe, but lock-free appending of a value to an array:

Algorithm 1: Thread-safe Append

```
Input: Array $A$, head index $h$, value $v$
// Note: A must be large enough.
1 $h_{\text{old}} \leftarrow$ FetchAndAdd(&$h$,1)
2 $A[h_{\text{old}}] = v$
```

- FetchAndAdd(&a,b)

  Executes $a \leftarrow a + b$ atomically and returns the former value of a. Without FetchAndAdd, at least three machine instructions would be used: Fetch a, add b, store result into &a[1]. Similar instructions exist for other arithmetic and binary operators.

- TestAndSet(&a,b)

  Temporarily saves the value from &a[1], stores b in &a and returns the saved value. This atomic instruction can be used for locking:

Algorithm 2: Lock

```
Input: Locking variable $L$ (initialized to $L = 0$)
1 while TestAndSet(&$L$,1) =1 do nothing
```

[1]The &-sign represents “address of ...”

3. Parallelization of a Priority Queue

Parallelizing a priority queue is a many-faceted task. Other than the implementation of a thread-safe and ideally non-blocking PQ, one also has to consider problems like starvation of operations and logical issues with concurrently extracted elements.

Most other parallel PQs focus on the first part, allowing different threads to operate on the same PQ. Some of them are introduced shortly in the first section of this chapter. Subsequently, we discuss possibilities for synchronizing Insert and ExtractMin operations in a concurrent environment. Lastly, we give some thoughts on bulk extraction.

Internal use of parallelism is not dealt with here but in connection with our algorithm in chapter 5.

3.1. Related Work

Much work has already been done on incorporating parallelism into priority queues and very different approaches have emerged from it. Some of them are presented here.

In 1998, Sanders [16] developed a randomized parallel priority queue for distributed memory machines. Each processing unit (PU) contains its own elements instead of having them in shared memory, which makes the algorithm suitable for a wider range of computer systems. The semantics of his PQ is slightly adapted: The ExtractMin operation retrieves the P globally smallest elements and each PU receives one of them (with P being the number of PUs). Insertions are distributed randomly among the PUs. The algorithm has shown good performance on machines with $P > 40$.

Sundell and Tsigas [21] introduced a lock-free concurrent priority queue based on skip lists [17]. They use atomic operations and auxiliary bits on the elements for indicating ongoing modifications in order to allow concurrent access to the data structure without any locks. The performance of common lock-based concurrent PQs decline significantly when a rising number of threads performs operations on it, while the performance of Sundell's approach stays nearly the same. However, their algorithm is not expected to be faster than one, that is used in a sequential manner.

A lock-free priority queue has also been developed by Liu and Spear [13], but it's based on a tree of sorted lists, a so-called *mound*.

Brodal et. al. [9] developed a parallel priority queue with constant time operations, including ExtractMin and DecreaseKey. They also have a MultiInsert_k and a MultiExtractMin_k operation with $O(log(k))$ time complexity. However, the algorithm is mainly of theoretical value, since the number of required processors depends on the input and is not bounded.

Pinotti [15] introduced a parallel priority queue based on a k-bandwidth-heap. This type of heap contains P elements in each node, where the largest one is smaller or equal to all the ones contained in its descendants. This allows concurrently deleting up to P elements. For restoring the data structure afterwards, though, the PQ has to be locked.

3.2. Synchronization between Insert and ExtractMin

Common applications relying on a PQ follow a consistent scheme:

(i) Create an empty PQ.

(ii) Insert some initial elements.

(iii) Extract the element with the highest priority. Do some computation on it, which may cause further insertions. If PQ is not empty, repeat this step.

(iv) Algorithm has finished, when PQ is empty.

If there are multiple threads operating on the same PQ, the expression "element with the highest priority" may be ambiguous, as the following example shows. Figure 3 depicts a sequence of insertions and extractions, executed by two independent processes P_0 and P_1. Square nodes stand for an extracted value. A diamond node d, connected to a square node s by an arrow, means that the extraction of s caused the insertion of d. t_0 to t_3 are time slices, and a thread can insert only one element per time slice.

The situation is as follows: P_0 extracts the starting element with priority 2. As it causes insertions of elements with priority 3, 1, and 0 (highest priority), the next element in row should be the one with priority 0. However, since there is no synchronization between the threads, P_1 extracts a value before the element with priority 0 is inserted by P_0, and this is the one with priority $1 > 0$.

Of course, there are applications where this type of incident is not crucial. For scheduling algorithms, for example, it may be sufficient to receive one of the smallest values. In this

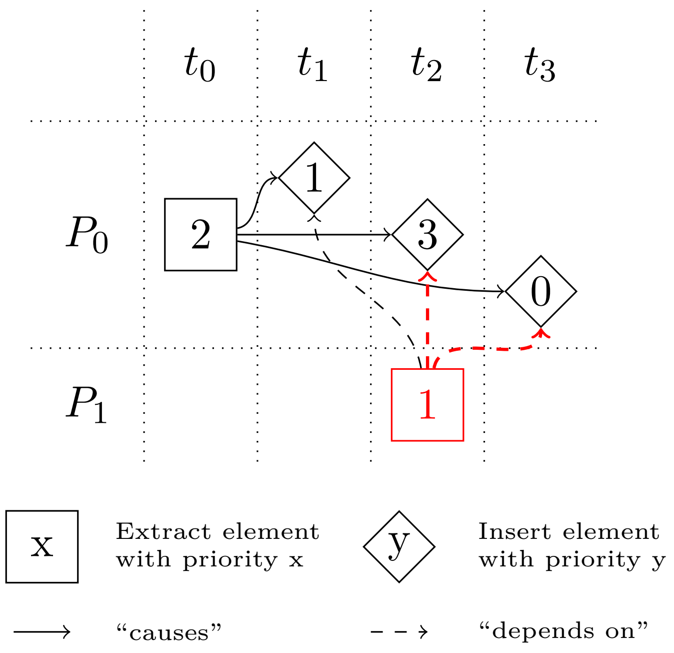

t_0
t_1
t_2
t_3
P_0
P_1
2
1
3
0
1
x
Extract element with priority x
y
Insert element with priority y
"causes"
"depends on"

Figure 3: A possible execution order of six PQ operations, which results in the extraction of a wrong value

case, it's sufficient to provide a thread-safe interface. Yet, we want to derive a benefit from parallel machines for a wider range of applications. For this purpose, the remainder of this section will present two strategies to avoid such ambiguities.

3.3. Bulk Operations

Bulk Insertion

An obvious approach to synchronize accesses to the PQ is to have one main thread which distributes work among the others. In many applications, one extracted element can cause the insertion of multiple other ones. If these elements are inserted together, the insertion work can be distributed among all available threads. Figure 4 illustrates this operation, which will be called *bulk insertion* from now on.

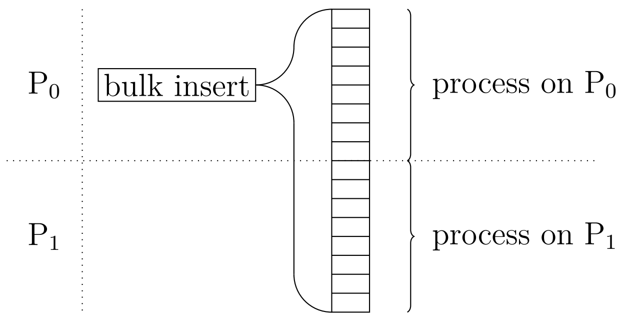


Figure 4: Bulk insertion using two threads P_0 and P_1

Bulk Extraction

The main thread can also initiate a bulk extraction, that is, obtain a bunch of elements and distribute the processing of these elements among the available threads (see figure 5). Note that any insertions occurring during the execution must use a thread-safe interface.

The case of extracting a bulk of elements and processing them in parallel needs a closer look:

In a classical priority queue, ExtractMin means returning the element with the currently smallest key among all keys and removing it from the queue afterwards. This definition consorts well with concurrently extracting all elements which have the same smallest key. To give an example: Dijkstra's algorithm could extract all nodes with the same distance in parallel. The application can take care of this form of parallel execution itself by simply calling ExtractMin multiple times. It's tough to derive a benefit from an extra DeleteAllEqual interface as it's not known how many elements with the same key there are.

Yet the case that there are so many elements with the same priority that parallel execution is worthwhile might be rare depending on the application. Let's talk about some further forms of parallel deletion:

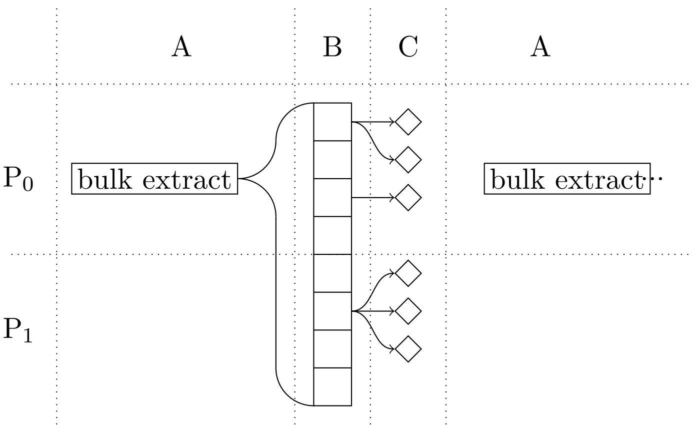


Figure 5: Bulk Extraction. Step A: Main thread extracts all relevant elements. Step B: Processing of the extracted values is distributed among all threads. Step C: Further insertions must use a thread-safe interface. Return to step A after an implicit barrier.

Extracting a fixed number of elements Extracting $k > 1$ elements (possibly having different keys) in parallel means that their order of execution doesn't matter for the correctness of the algorithm. Of course, the number of extracted elements must be limited since one could otherwise use an unordered array of elements instead. As an example of an application where the correct order can be violated on a small scale is priority scheduling, where k enqueued work units can be distributed simultaneously among k free CPUs.

Having k given, the priority queue can go for different strategies when the application wants to extract elements. While it would simply call the regular ExtractMin interface for small k, it could bring forward some work for larger k, or derive a benefit from filling a k-extract buffer in parallel.

Extracting until a specific key An application could relax the definition of ExtractMin like follows: ExtractMin *returns an element with key* $< K$ *and removes it from the queue.* K must be passed for each ExtractMin and could for example be computed using the previously extracted element.

Like for extraction of equally prioritized elements, here it holds too that an extra interface for this type is not expected to yield a great benefit due to not knowing the number n_{extract} of affected elements. Instead, we introduce a PrepareManyExtracts operation which can be used if n_{extract} is expected to be large: It brings forward upcoming work like in the *Extracting a fixed number of elements* case.

3.4. Aggregated Insertion / Lower Limit for Insertion

The bulk insertion interface requires that the application has one main thread which handles all insertions. Aggregated insertion, as it is described below, is an alternative that allows multiple application threads to insert elements independently and still obtain a bulk of elements that can be inserted efficiently in parallel.

Appending a value to a buffer is an operation that can easily be made thread-safe using a FetchAndAdd instruction (see chapter 2.3). The idea is now to buffer any insertion until the next ExtractMin operation occurs, or even longer if it can be assured that the values in the buffer, at least for a while, don't need to be considered for the ExtractMin operation. The applications manually tells the PQ when to process the buffered elements so they are considered for upcoming deletions. We call this FlushAggregatedInserts. The insertion method may depend on the number b of buffered elements. If b is large, the bulk insertion method from chapter 3.3 can be used.

Aggregated insertion can also be used if the application expects that there will be very many insertions in a row, but doesn't know it for sure. For small sequences one then saves the parallel overhead of bulk insertion, while large sequences are still inserted efficiently in parallel.

A further and more interesting use case is the following: There is a large sequence S of intermixed insertions and deletions. However, it's known that none of the values inserted in S is already extracted again while S is processed: $\forall$ deleted values $d \in S, \forall$ inserted values $i \in S : key(d) < key(i)$. This means that there is a lower limit for any insertion in S and inserted values in S don't need to be considered for ExtractMin operations until FlushAggregatedInserts is executed. Figure 6 illustrates this situation. In [2] you can find an algorithm, where this constraint is valid for some key loops.

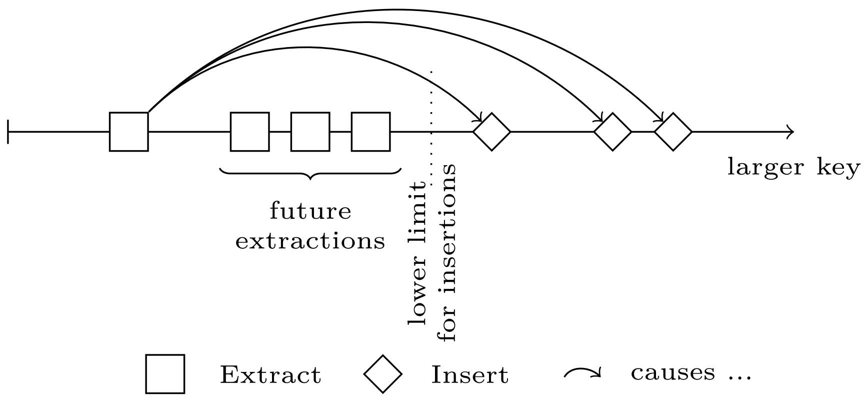


Figure 6: Aggregated insertion. The key of all inserted elements is higher than any of the upcoming extracts before the barrier. An arrow represents that the insertion is caused by the extraction.

4. Priority Queues for External Memory

If the elements in a PQ don't fit into the main memory anymore, classical PQs certainly become very slow, as the paging mechanisms of the operating system might begin juggling around memory pages between internal and external memory (EM). External memory algorithms try to avoid this and reduce the number of I/Os per operation [10]. For this purpose, a priority queue for external memory takes into account the specific properties of disk drive systems or other slow secondary memory types.

This chapter gives an overview over some possibilities for designing a priority queue with support for external memory.

4.1. Related Work

Scientists have already payed much attention to designing external-memory priority queues. Gradually, the I/O-efficiency has been improved more and more (e.g. [8], [4]). Newer studies also took practical objectives into account.

Brengel et al. [3] carried out an experimental study of priority queues in external memory that resulted in two novel external-memory heaps. First, they adapted a radix heap [1] for external memory. This is a monotone PQ, that is, extracted values must occur in a non-decreasing order. The radix buckets reside in EM except for their first disk page. As for all radix based data structures, the input should be evenly distributed among the value universe.

Their second approach is called an external array-heap. It consists of an internal memory heap and a set of sorted arrays in external memory. The arrays have a fixed size and are arranged in slots, assigned to a level. The heap can be viewed as the lowest level. Insert operations go to the lowest level and overflows in one level cause a transfer into the next higher level after sorting and merging if necessary.

Sanders [18] followed a similar approach and improved it among other thing by paying much more attention to cache efficiency. The data structure is called a sequence heap. Here, the external arrays are organized in groups of size k, with k being chosen small enough, that merging all members of a group will be cache-efficient using k-way-merge [11]. Similar to Brengel's approach, an overflow in one group (respective level) causes the creation of a larger array in the subsequent group. All groups are connected by an R-way-merger, where R is the number of groups. In common environments, R is small enough in the context of cache efficiency, too.

4.2. Main Ideas

As we have seen in the related work section, most external memory PQs only store sorted sequences of values in the EM. This stands to reason since accesses occur in an increasing order and values from EM can therefore be fetched consecutively (see chapter 2.2 for the advantages of a consecutive EM access). Only the radix heap solution by Brengel et al. derogates from this principle. In their algorithm, the ExtractMin operation scans a whole bucket for the smallest element. If the bucket is large, this results in extra I/O. Because of that and other limitations of a radix based approach (see chapter 2.1), we will hereafter narrow down to the use of sorted sequences in EM.

There are several tasks and possibilities when designing such a PQ. Some important ones are listed in the following.

(i) Internal priority queue

The Insert operations of established EM priority queues first go into the internal memory (IM). The data is transferred into EM not until either an adequate amount of data has been accumulated, or there is no more space left in IM. Instead of having an unordered buffer of linear complexity for ExtractMin, typically a regular PQ, like one of those introduced in chapter 2.1, is used for IM.

(ii) Multiple sources of the global minimum

Because the PQ consists of at least two relatively independent parts (IM and EM part), there are at least two sources for the globally smallest element. If there are multiple sorted arrays there may be even more sources. In order to avoid performing linear search for determining the global minimum, an EM PQ can maintain the sources for example in a sorted list or in a tournament tree (see appendix A.1), or keep them heap-ordered.

(iii) Extract buffer

An extract buffer can reduce the number of minimum sources to exactly two and furthermore improve the performance by using efficient parallel multiway merging. The extract buffer is usually built by merging all sorted lists in EM and possibly some more data from IM.

(iv) Having first blocks in IM

External memory in our context is a storage system with block-wise data transfers (cf. chapter 2.2). Therefore it makes sense to store the whole block when accessing a value from EM. These blocks can then be used for data-parallel merging, since such a merging algorithm can determine splitting points if the data is randomly accessible.

(v) Overlaying computation and I/O

The latency of hard disk accesses is quite high (about 15 ms on a 7200 rpm drive [22]). This can delay PQ operations and waste valuable CPU-time. Thanks to the direct memory access (DMA) feature of modern computer system, it's possible to fetch data from external memory devices without occupying the processor. This can be used to prefetch data blocks that will probably be accessed soon, more precisely, further blocks of the sorted external sequences in ascending order.

Prefetching is also important on multi disk systems. In a multi disk system with D disks, the external memory bandwidth is theoretically D times higher, than it is for one disk, provided that the bus bandwidth is high enough. If we access a single sorted sequence in EM and want to utilize this higher bandwidth, there needs to be a prefetch buffer with a size of at least D blocks. Of course this improvement requires having spread the blocks of the sequence evenly among the disks.

Note that the amount of data to prefetch should carefully be limited in order to avoid wasting internal memory. If $D = 8$, $B = 2\ MiB$, and the PQ is configured to prefetch a constant number of 8 blocks for each external sequence, in addition to at least one regular block in IM as mentioned above, an external array already occupies 10 MiB of internal memory.

Another possibility to overlay computation and I/O is to buffer EM write operations. Just like for prefetching, the size of the buffer should be oriented towards the number of hard disk drives. Internal memory considerations are less crucial here, since the buffer is only needed during the creation of the sequence.

(vi) Reduce the number of I/Os per element

A priority queue for EM should try to avoid writing the same element multiple times into EM, as EM has a very limited bandwidth and the algorithm loses it's I/O-optimality otherwise. However, when regarding practical considerations, it may be beneficial to do so, e.g. for merging small sequences into one.

5. Bulk-Parallel Priority Queue in External Memory

This chapter presents the bulk-parallel priority queue that has originated from this work. The preceding chapters have revealed that there are different kinds of priority queues. Concerning their properties, we've made the following design decisions:

non-monotone Our PQ supports any order of elements to insert at any time. This excludes the implementation as a radix heap.

arbitrary keys Any kind of ordering is allowed, the keys do not have to be integer or bounded. The application just needs to provide a less-comparator for the elements. Neither an equality-comparator, nor minimum, maximum or any sentinel elements are needed.

external memory In case the contained data doesn't fit into main memory, there is native support for external memory.

non-accessible The elements in the PQ are not accessible, and consequently there is no DecreaseKey operation. The reason behind this is that random access conflicts with efficient external memory support.

non-relaxed The PQ conforms to a strict definition of ExtractMin: "Return the element with the smallest key of *all* elements currently in the queue". An element is considered to be part of the queue, from the time `push()` was called. This is in contrast to relaxed and probabilistic PQs, which return the smallest element only with a given probability.

bulk-parallelism Because our PQ is non-relaxed, we've decided to use a bulk interface for external parallelism. Bulks are sequences of homogeneous operations. This assures a synchronization between inserts and deletes, and therefore guarantees the global minimality of the extracted element. Furthermore we provide thread-safe interfaces for aggregating an insertion bulk, as well as for single insertions.

realistic The purpose of this work has been to develop a priority queue that performs well on real-world systems. We attempt to achieve I/O and computational optimality only for realistic parameters.

The remainder of this chapter describes the architecture and the operations of our priority queue (also called newPQ). Alternative variants and implementations, that have been tried out during development are outlined where reasonable as well.

5.1. Architecture

Our priority queue consists of various data structures. First, the insertion heaps are responsible for taking up newly inserted elements (see chapter 5.5). They are complemented by sorted arrays in the internal memory (chapter 5.2). The external storage is organized in sorted external arrays, where, however, the first block (according to an ascending order) resides solely in internal memory (chapter 5.3). The external memory write buffer, as well as prefetch buffers for all external arrays may reside both in IM and EM. Furthermore, there is an extract buffer for accelerating ExtractMin operations (chapter 5.7) and an aggregation buffer for the AggregatedInsert interface (chapter 5.5). A minima tree manages the smallest elements from all relevant data structures (chapter 5.4). Figure 7 illustrates these parts. A detailed description of them can be found in the remainder of this chapter.

5.2. Internal Arrays

An internal array (hereafter IA) is a sorted sequence of values stored in the main memory. There are three sources where an IA can arise from (see figure 8):

(i) When the insertion heaps are full, respectively they cannot carry the current bulk, they are sorted in parallel and then merged into a merging buffer. The merging buffer's values are then transferred into an internal array. It's important to avoid memory copying here, like described in the implementation notes (chapter 6). As a variant, merging can also be omitted (see chapter 5.9). The operation is called FlushInsertionHeaps.

(ii) The aggregated insert interface (FlushAggregatedInserts) can cause the creation of an internal array if the number of aggregated elements exceeds some constant value, like described in chapter 5.5. In this case, the aggregation buffer is sorted in place, and its elements are transferred into an internal array (again without moving the actual values).

(iii) In an analogous manner, the algorithm handles bulk insertions which exceed this threshold size.

5.3. External Arrays

An external array is a sorted sequence of elements in external memory. The sequence is divided into blocks of size B by the external memory system. A block is always fetched as a whole, we never discard parts of it.

For the purpose of fast and parallel merging, the first block, i.e. the one containing the smallest element, is located in IM instead of in EM. Furthermore, the p following blocks are prefetched in order to better utilize the I/O-bandwidth and to allow parallel data fetching from multiple hard disk drives.

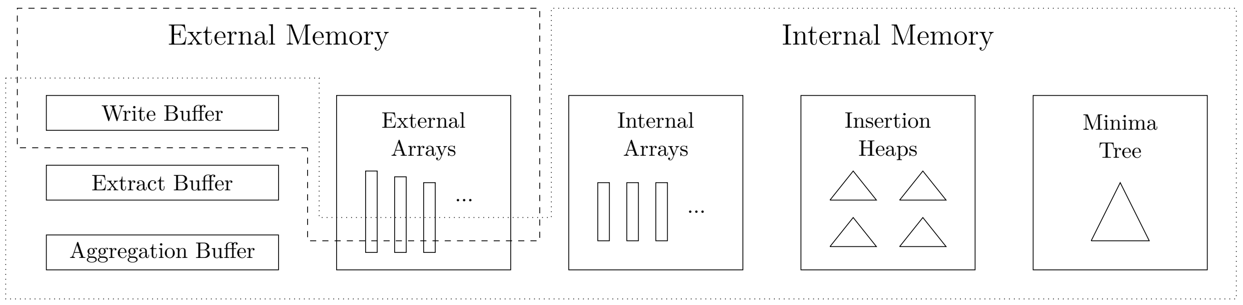


Figure 7: Architecture of the priority queue. Note that one block of each EA is solely in IM and some blocks may reside both in IM and EM due to prefetching.

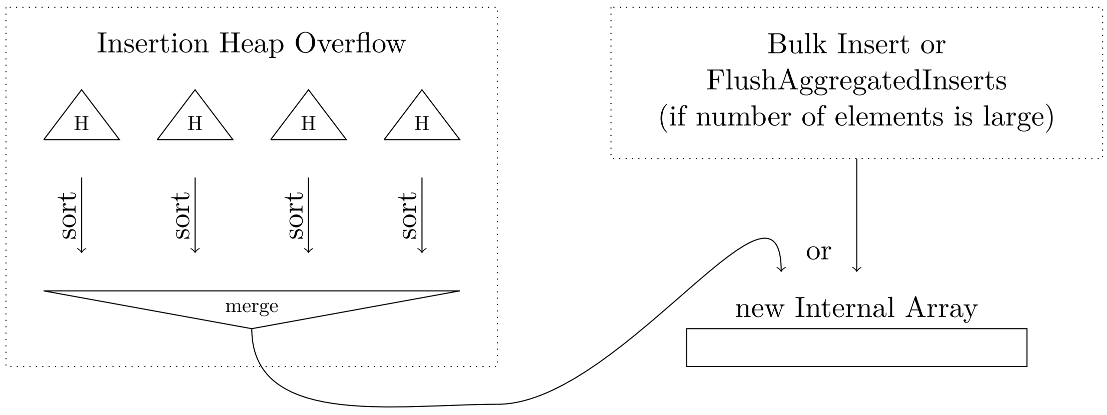

Insertion Heap Overflow
H
H
H
H
sort
sort
sort
sort
merge
Bulk Insert or FlushAggregatedInserts (if number of elements is large)
or
new Internal Array

Figure 8: Internal array creation

There are two ways how an EA can be created:

(i) When there is no space left in IM, the easiest way to make space is getting rid of the internal arrays. They are first merged and then transferred into an external array. The operation is called FlushInternalArrays.

(ii) Each external array stores at least one block of elements in IM, plus the ones potentially in the prefetch buffer. Together, this is a considerable amount of IM consumption. If they take too much space, multiple EAs have to be merged into one, which reduces the memory consumption to $M_{\text{EAs}} = (1 + p) * B$. We call this operation MergeExternalArrays.

The implementation is very similar to the one used for refilling the extract buffer (see chapter 5.7). The difference is first, that by default, internal arrays are not taken into account. Secondly, the merging happens piecewise because random access, needed for parallel multiway merge, is only possible in the first block of each EA.

The MergeExternalArrays operation is quite expensive, therefore it should not be executed too frequently. Luckily, this is the case for realistic use cases. More on this topic can be found in the chapter on memory management (5.8).

5.4. Keeping Track of the Global Minimum

Our algorithm keeps different sorted or heapified sets of data, all of them providing $\mathcal{O}(1)$ access to their local minima. They are called candidates for the global minimum. Namely these candidates come from

(i) one of the insertion heaps
(ii) the extract buffer
(iii) an internal array, that was created while the extract buffer had been full
(iv) an external array, that was created while the extract buffer had been full

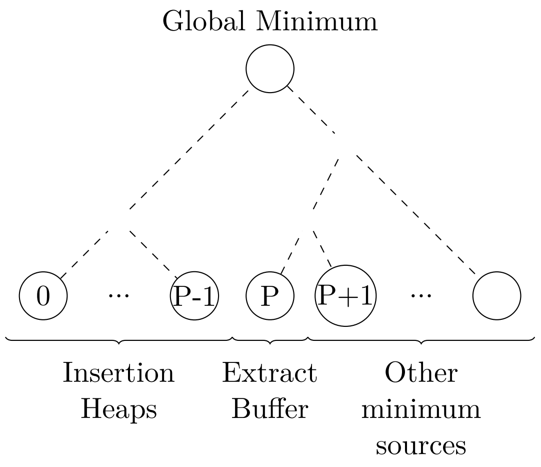


Figure 9: Minima winner tree with index mapping.
P is the number of insertion heaps respectively the number of CPU cores

The ExtractMin procedure needs to find the smallest one of the candidates. Linear search isn't really practicable here, as the number of candidates is theoretically unbounded. Even if there were no internal or external arrays, linear search would require $P + 1$ comparisons on a machine with P cores, which is inefficient if P is large.

During development three approaches were tried out to reduce the work for a ExtractMin from $\mathcal{O}(k)$ for k candidates to $\mathcal{O}(\lceil \log(k) \rceil)$:

Using a heap A heap contains tuples (v, id) of a candidate value v and an identifier id for its source. The tuples are compared according to the value: $(v_0, id_0) < (v_1, id_1) \Leftrightarrow v_0 < v_1$. The ExtractMin operation simply pops the smallest tuple $(v_{\min}, id_{\min})$ from the heap, has immediate access to the minimum value, and can push the proximate candidate according to the source identifier.

Inserting a new candidate for the former winner is quite simple and fast with a binary heap, but if an arbitrary candidate becomes invalid or is replaced, decease-key and delete-key operations are required. This happens for example after inserting a value into a non-empty insertion heap or after FlushInternalArrays. While all this is doable with addressable heaps, the constant factors are higher than with the following approaches.

Winner tree with index mapping The second approach makes use of a winner tree (WT). A detailed introduction to winner trees can be fount in appendix A.1. Like described there, a WT consists of 2^x so called players, which compete against each other for the globally smallest element. The first P players represent the insertion heaps, followed by one player for the minimum of the extract buffer. Arrays, that were created while the extract buffer had not been empty (*new arrays*) and are therefore not represented in the extract buffer, occupy further player positions.

Unfortunately, it's not clear if a player with an index $> P$ belongs to an internal or an external array. One solution is using even numbers for internal and odd numbers for external arrays, but this leads to a superfluously large tree if the number of IAs is very different to the number of EAs. This case occurs quite often, e.g. after FlushInternalArrays.

Instead, index mapping is used. A player's corresponding array is determined with a single mapping TreeIndexToArrayIndex: $\mathbb{N} \to \mathbb{Z}$, so that

$$\text{Array}(\text{ArrayIndex}) = \begin{cases} \text{ExternalArrays}[\text{ArrayIndex}] & \text{if ArrayIndex} >= 0 \\ \text{InternalArrays}[-(\text{ArrayIndex} + 1)] & \text{if ArrayIndex} < 0 \end{cases}$$

We also need two mappings InternalArrayIndexToTreeIndex: $\mathbb{N} \to \mathbb{N}$ and ExternalArrayIndexToTreeIndex: $\mathbb{N} \to \mathbb{N}$ for deactivating a player when it's corresponding array becomes invalid.

Composed winner tree The index mapping implies many indirections, because the tree index cannot always be mapped directly to a source. Especially the comparator used by the index winner tree can have noticeable performance losses. For that reason, we've replaced the index mapping by multiple index winner trees for the different types of candidate sources.

Figure 10 depicts the following structure: First, there is a head winner tree (A) which compares the smallest insertion heap minimum with the extract buffer minimum, the smallest internal array minimum and the smallest external array minimum. The insertion heaps (B), the internal (C) and the external arrays (D) are managed in three separate winner trees.

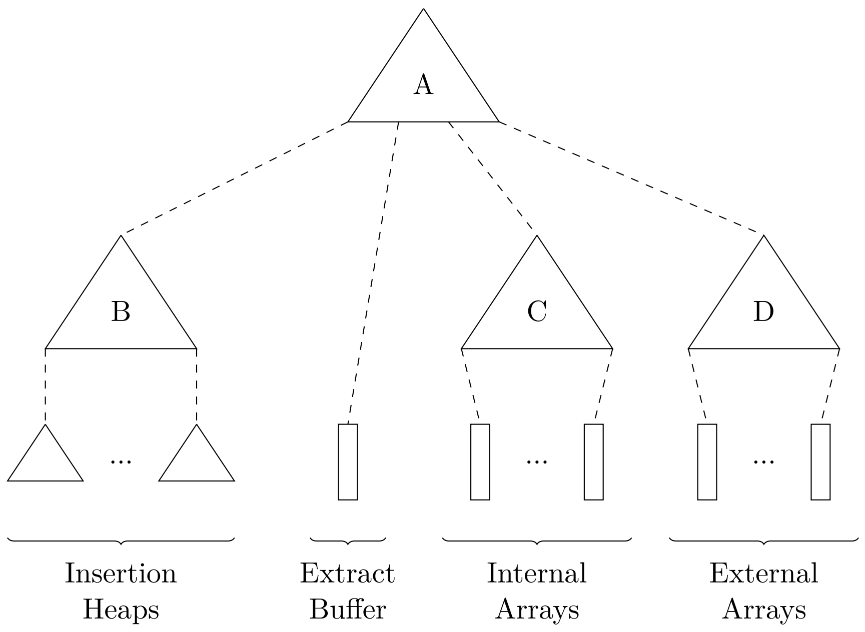


Figure 10: Composed minima winner tree.

5.5. Insertion of Elements

Chapter 3.2 has already discussed different kinds of parallel insertion. This discussion resulted in four different interfaces for insertion of elements into the priority queue. Besides simple, sequential, single insertion there is a bulk insertion and an aggregation interface. Furthermore, a technique formerly used for bulk insertion allows thread-safe, and to a certain degree also parallel, single insertion.

Singe Insertion

Single inserts are used if there will be only a few inserts before the next ExtractMin occurs. They are possible in two flavors: The non-thread-safe and in the majority of cases faster variant choses a random insertion heap, inserts the value and updates the minima tree afterwards. It would also be possible to chose the least filled one in order to distribute the data evenly, but this would cause great overhead. A thread-safe variant is described further below as *bulk insertion with individually locked heaps*.

Bulk Insertion

Bulk insertion can be used if there is a sequence of Insert operations which will definitively not be interrupted by an ExtractMin operation. The insertion consists of three parts: *begin*, *step* and *end*. The *begin* part chooses the proceeding depending on the bulk size b. Note, that either the bulk size or at least the maximum bulk size must be known from the beginning, while the latter may result in inefficient processing.

If $b < K < H$ for a constant K and the size of the heaps H, each element in the bulk will be inserted using sequential SingleInsert, because spawning threads for parallel insertion would cause too much overhead compared to the benefit.

If $b > a \cdot H$, for a factor $a \leq 1$ and $a \cdot H > K$, it's meaningless to push the elements onto the heaps, because they would be sorted closely afterwards. Instead, the elements are sorted in parallel and transferred directly into an internal array.

Otherwise, if $K \leq b \leq a \cdot H$, the elements are inserted in parallel into different heaps. *begin* makes sure, that there is enough space left in the insertion heaps. If not, a FlushInsertionHeaps operation is performed. This avoids any interruption during the parallel execution.

For handling the parallel insertion without race conditions and data losses, three different approaches have been tried out:

Individually locked heaps One way is to use mutual exclusion locks on the heap during a PushHeap operation. When inserting a value, a free insertion heap is chosen. Algorithm 3 describes the strategy for doing so.

Algorithm 3: Thread-safe Insert

```
Input: Element $e$, Insertion Heaps $H$
1 repeat
2     id $\leftarrow$ random number $\in \{0, 1, .., |H| - 1\}$
3     do atomically
4         if $H[id]$ is not locked then
5             Lock $H$[id]
6             break
7 Insert $e$ into $H$[id]
8 Unlock $H$[id]
```

The `do atomically` section can be implemented using a CompareAndSwap instruction if the locking state is stored as a boolean value for each insertion heap.

In the context of bulk insertion, this has turned out to be the slowest implementation. The problem is that the locking state has to be synchronized between the threads, and therefore, they invalidate each others caches.

However, algorithm 3 provides a thread-safe insertion interface with support for concurrent PushHeap execution. If there are few concurrent inserts in relation to the number of insertion heaps, the probability of immediately obtaining an unlocked heap in the first loop iteration is quite high:

$$P[\text{Insertion Heap i is not locked}] = \frac{n_{\text{IHs}}}{n_{\text{Concurrent Inserts}}}$$

Atomic heaps It's possible to build a heap whose push operation is thread-safe and lock-free as follows: There is an array A, a head pointer h, and a heap head pointer p. The push operation increases the head pointer using an atomic FetchAndAdd operation, which is available on all common, modern architectures. Afterwards, the value is written at the old head position (usually the return value of the FetchAndAdd operation), and a heapify request is placed. The heapify requests are processed by a single thread, which is running PushHeap and increasing the p repeatedly until it's equal to h. This doesn't affect ongoing push operations. Pseudocode for this procedure can be found in algorithm 4.

Algorithm 4: Atomic Insertion Heap

```
Input: Array $A$ (begin pointer), Head pointer $h$, Heap head pointer $p$
Push(Element $e$) begin
    $h_{\text{old}} \leftarrow$ AtomicFetchAndAdd(&h,1)
    $*h_{\text{old}} \leftarrow e$
    heapifyRequest $\leftarrow$ true
    Heapify()
Heapify() begin
    if heapifyRequest = false then
        return
    if AtomicCompareAndSwap(&heapifyInProgress, false, true) then
        heapifyRequest $\leftarrow$ false
        while $|p - h| > 0$ do
            $p \leftarrow p + 1$
            PushHeap($A$, $p$)
        heapifyRequest $\leftarrow$ false
        Heapify()
```

One heap per thread The third approach is to bind each thread to exactly one insertion heap. This approach is much more cache-efficient than the other ones. If a thread performs it's heap operations always on the same heap, it is very likely that appreciable parts of it still are in the CPU-local cache. It turned out to be the fastest implementation for bulk insertion.

Aggregated Insertion

Possible use cases for aggregated insertion have already been stated in chapter 3.2. We've implemented it as follows: AggregateInsert(e) atomically pushes the element e to a buffer using the atomic FetchAndAdd instruction, similar to lines 2 and 3 of algorithm 4. ExtractMin operations are not affected by this. FlushAggregatedInserts() eventually inserts the aggregated values as a bulk using the bulk insert interface described further above.

5.6. Extracting Elements

The ExtractMin operation fetches the current minimum source *src* from the minima tree described in chapter 5.4. The value of *src* is saved for return and then removed. Afterwards, a message is sent to the minima tree for either a change in *src*'s player p_{src}, or the deactivation of p_{src} if *src* has run empty.

There is also a BulkExtractMin (k) interface. It returns exactly k elements, given that $|PQ| \geq k$. If k is very large or the insertion heaps are nearly full, the insertion heaps are flushed and the extract buffer is refilled to a size of at least k. These smallest k elements are then returned. The amortized execution time for large k is expected to be smaller, as k minima tree updates can be omitted. If k is small, the BulkExtractMin (k) method simply runs ExtractMin k times.

5.7. The Extract Buffer

The extract buffer (EB) is there to accelerate ExtractMin operations. It's built from the sorted sequences in internal and external memory by merging them into the buffer (the (re-) creation is called RefillExtractBuffer from now on). Immediately after the creation we know that $\forall\ a \in \text{EB},\ b \in \text{EAs} \cup \text{IAs} :\ key(a) \leq key(b)$ (i). This means, that an ExtractMin operation doesn't have to consider the sorted sequences, but only the extract buffer and the insertion heaps. Note that this holds only for arrays that have already existed before running RefillExtractBuffer. For newly created arrays an extra entry in the minima tree is necessary (cf. chapter 5.4) because equation (i) is not valid for them.

The array merger can only access array parts that reside in internal memory. Because there may be values in the EM which are smaller than a value in in the IM, upper bounds have to be found for which it holds that $\forall\, a \in \text{EA} \cup \text{IA} :\ \forall\, e_1 \in a[0...\text{upper bound}_a],\ \forall e_2 \in \text{EM} : e_1 \leq e_2$. This is true for all values that are smaller or equal to the smallest internal maximum value of all external arrays with further data in EM.

Figure 11 shows an example situation where 20, 35, and 40 are the internal maximum values of the EAs. 20 can be ignored since the corresponding array has no further data in EM. 35 is the smaller one of the remaining two and therefore it's the upper bound value. The merger will only consider values smaller or equal to 35.

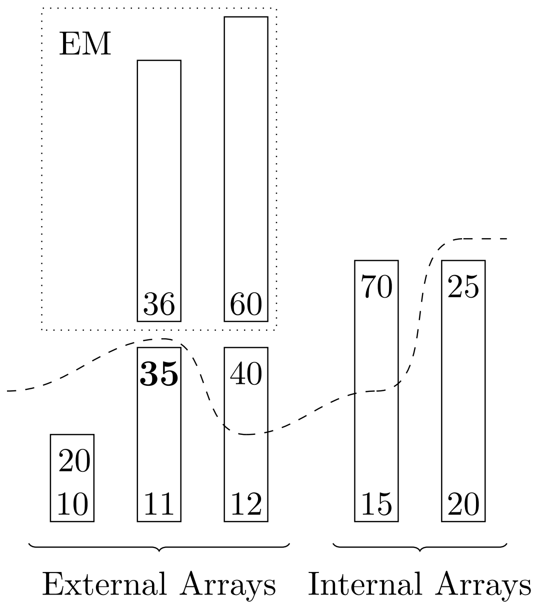


Figure 11: The dashed line depicts the upper bound for merging. 35 is the limiting value.

Furthermore, the size of the extract buffer is limited by the memory management (cf. chapter 5.8). The merger will break when the limit is reached. It's important that it saves the index until which each array has been merged. This index is used for removing the affected values from the arrays after they have been transferred into the EB.

Algorithm 5 shows the refill procedure in detail.

Algorithm 5: Refill Extract Buffer

```
Input: Extract Buffer $E$, Extract Index $i_E$, Array of EAs $E$, Array of IAs $I$, Minima-tree
       $M$, Extract buffer limit $l$
1  $E \leftarrow []$
2  $i_E \leftarrow 0$
3  $M.removeAllArrayPlayers()$
   // As the index of the arrays doesn't matter anymore, we can finally remove empty arrays.
4  foreach Array $a \in E \cap I$ do
5      if $|a| = 0$ then
6          $E \leftarrow E \setminus a$
7          $I \leftarrow I \setminus a$
8  if $|E| + |I| = 0$ then
9      return
10 $minmax \leftarrow undef$                                      // Let min$(undef, x) = x$.
11 foreach External Array $a \in E$ do
12     Wait for the first block. Memory transfer may be in progress.
       // Only consider EAs with further data in EM.
13     if Number of blocks in $a > 1$ then
14         minmax $\leftarrow$ min($minmax, a.max\_of\_block()$)
15 Merge sequences $S \leftarrow []$
16 foreach Array $a \in E \cup I$ do in parallel
17     Determine the maximum fraction $a_{\text{frac}}$ of $a$ (resp. the first block of $a$ if $a \in E$), for which
       it holds that $\forall e \in a_{\text{frac}} : e < minmax$.
18     $S \leftarrow S \cup \{a_{\text{frac}}\}$
19 $size \leftarrow$ min($\sum_{s \in S} |s|, l$).
20 Allocate memory of size $size$ for $E$.
21 Parallel multiway merge $size$ elements of $S$ into $E$.
22 Remove merged elements.
23 Notify extract buffer change to $M$.
```

5.8. Memory Management

Memory management is an important part of the implementation. Internal memory is a limited resource for fast memory and can be used in various ways. A trade-off has been made between the needs of different IM consumers in the PQ implementation. The allocation strategies for them are listed below.

- M_{IHs}: The insertion heaps can have either a fixed size, or a size relative to the available memory. In combination with internal arrays, the fixed variant has turned out to be the better one. Choosing M_{IHs} is a crucial decision. It influences the performance in the following ways:
  - PushHeap and PopHeap operations are slower for large insertion heaps. The performance loss is quite large if a heap doesn't fit in the cache anymore.
  - Parallel sort and parallel multiway merge are the better, the bigger the insertion heaps are, since the constant overhead for the thread creation must be compensated for.
  - Smaller insertion heaps implicate more and smaller internal arrays. If the internal arrays are created when the extract buffer is full, an extra entry in the minima tree is necessary for them. A large minima tree can significantly deteriorate the performance. On the other hand, though, a large number of internal arrays makes merging them in parallel more efficient.
- M_{IAs}: The internal arrays fill all internal space currently available, before they are merged into an external array (cf. chapter 5.2).
- M_{EAs}: External arrays occupy internal memory M_{EA} of at least one external block size B. In addition, up to p blocks may be prefetched from EM. Depending on p, which should be higher on multi-disk systems, M_{EAs} can be a considerable part of the main memory. If there is no space left, MergeExternalArrays is executed (cf. chapter 5.3). Since this operation is quite expensive, it should not be executed frequently. Let's find out how much data can be filled into the PQ before a MergeExternalArrays operation is necessary.

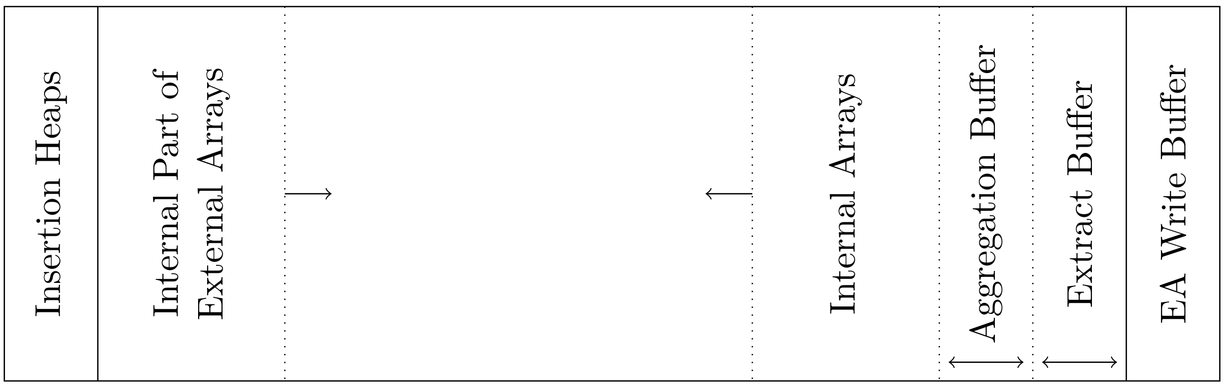


Figure 12: Internal memory allocation

The amount of data in a single EA decreases with a rising number of EAs, since there is less space for IAs, the more EAs exist. When an empty newPQ instance is consecutively filled, the amount of data after the l-th FlushInsertionHeaps operation is given by

$$vol_l \approx \sum_{k=0}^{l} M - M_{\text{miscellaneous}} - k * M_{\text{EA}}$$

Note, that the equation assumes that the IAs fill all space available. Actually, the wasted space cannot be larger or equal to M_{IHs}. Since it usually holds that $M >> M_{\text{IHs}}$, our assumption is appropriate.

The maximum number of EAs before a MergeExternalArrays operation is necessary is given by

$$l_{\max} = \left\lceil \frac{M - M_{\text{miscellaneous}} - M_{\text{IHs}}}{M_{\text{EA}}} \right\rceil$$

If we assume $M = 16$ GiB, $M_{\text{miscellaneous}} = 100$ MiB, $M_{\text{IHs}} = 16$ MiB, $B = 2$ MiB, and $p = 4$ (4-disk system), the maximum volume is given by

$$vol_{\max} \approx vol_{l_{\max}} + M_{\text{IHs}} = 12.652 \text{ TiB}$$

The maximum volume is in this case over 800 times larger than the internal memory. Together with experiences from experiments this allows the assumption, that MergeExternalArrays is executed rarely and will not affect the performance in most cases.

- M_{WB}: Write operations are buffered in order to overlay I/O and computation, as well as to write simultaneously to multiple disks. The size of the buffer should be at least $D * B$. One global buffer is sufficient, since only one external array can be created at a time. Therefore, M_{WB} is constant.
- M_{EB}: The extract buffer described in chapter 5.7 is of variable size, depending on the amount of mergeable data. However, there is an upper bound because the buffer could occupy all internal memory and it's size cannot be decreased from inside (i.e. without the occurance of ExtractMin operations) without causing extra I/O. The algorithm takes care, that there is always some internal memory left for an adequately sized extract buffer.
- M_{AB}: The algorithm reserves some space for the AggregatedInsert functionality. This aggregation buffer is allowed to grow beyond this reservation if there is free space, since the very next operation (FlushAggregatedInserts) will empty the buffer.
- $M_{\text{miscellaneous}}$: There is a constant memory overhead for state variables and the minima tree.

5.9. Variants

There are some variants of our algorithm that have advantages as well as disadvantages depending on the situation. The user can individually decide whether to use these modifications or not.

Internal Arrays disabled Internal arrays have been introduced as a precursor of external arrays for two reasons:

(i) They allow taking full advantage of the fast internal memory, as data is transferred into external memory not until the internal memory has exhausted.

(ii) External arrays are larger if they originate not only from merging the insertion heaps but also from the internal arrays. This means there are fewer external arrays for the same data volume. Too many EAs can be unfavorable if their first blocks have exhausted the internal memory and an expensive MergeExternalArrays operation is necessary.

However, if there are IAs, all elements in external arrays have passed through a merger at least twice. Once for FlushInsertionHeaps and once for FlushInsertionHeaps. This is extra work that can be avoided. Furthermore, ParallelMultiwayMerge is more efficient for a larger number of sorted sequences to merge (see chapter 6). As a consequence, disabling internal arrays can be beneficial if it is assured, that internal memory suffices for holding the first blocks of all external arrays.

Don't merge the insertion heaps The FlushInsertionHeaps operation merges the insertion heaps after sorting them in parallel. This reduces the number of internal arrays by a factor of P, which is important if the extract buffer hasn't been empty during the operation because it would otherwise cause P times more entries in the minima tree (cf. chapter 5.4).

If this case is rare (e.g. if the extract buffer is small and the heaps are big), if it doesn't occur at all (e.g. in the insert-all-delete-all test case), or there are few insertion heaps (P is small), it can stand for reason not to merge the insertion heaps.

6. Implementation Notes

Our PQ has been developed using the C++ programming language in a way that it can easily be integrated into the Stxxl. The following gives a short overview on libraries and algorithms use in the implementation.

Read Write Pool The Stxxl provides a useful class for buffered read and write accesses to the hard disk. When attempting to write to the hard disk the data is first filled into a block of size B and then delivered to the pool, together with a block id (BID) of your choice. Using this BID, the block can be easily fetched back later. Because it's a combined read and write pool there will be no problems if data is fetched before it has been written out completely.

Multiway Merge Our PQ makes heavy use of parallel multiway merging. The implementation is provided by the Gnu parallel library, originally developed by Singler, Sanders and Putze as part of the Multi-Core Standard Template Library MCSTL [19]. The algorithm can be used both in sequential and in parallel mode. Experiments have shown, that parallel merging brings great speedup when merging blocks of size 2 MiB. 2 MiB is the size of the accessible area of an external array for $B = 2$ MiB and is therefore a rough approximation for sequence sizes that occur in our application.

Figure 13 depicts data rates for merging a varying number of blocks. You can see there, that on Intel16, parallel multiway merge brings a speedup of 8.6 for 16 blocks to merge. This case occurs for example in FlushInsertionHeaps when having heaps of size 2 MiB. Note that the Stxxl's priority queue implementation uses this algorithm, too.

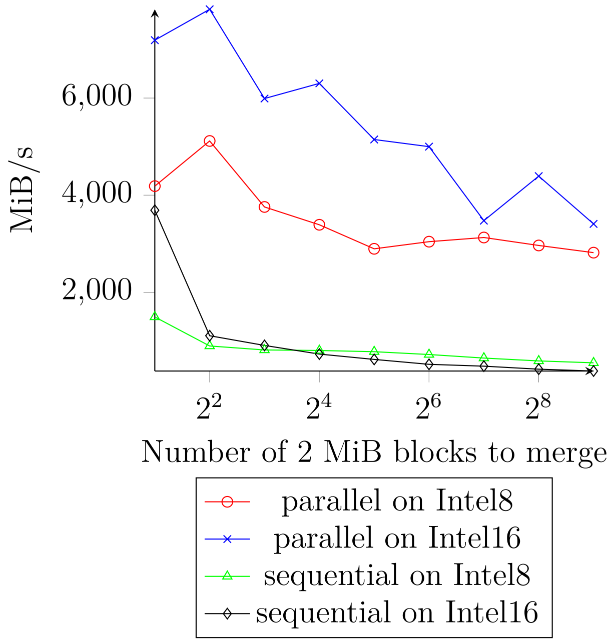


Figure 13: Multiway Merge Performance. block size = 2 MiB.

OpenMP For parallelizing our algorithm, we use OpenMP (Open Multi-Processing) [5]. OpenMP is a multi-platform set of compiler directives and library routines for multithreading. It allows parallel execution of code by forking and joining threads.

Take the following code as an example:

```
#pragma omp parallel for num_threads(8) schedule(static)
for (unsigned int i=0; i<80; ++i) { }
```

Using the `#pragma` directive, the compiler generates code that forks 8 threads and lets each thread execute a sequence of 10 iterations. A barrier is implicitly placed after the for loop. Note the contrast to solutions based exclusively on libraries, which can distribute the iterations not until runtime.

6.1. Implementation Issues

There were some pitfalls in the implementation, which we don't want to withhold:

Random number generation We had a very subtle issue with random number generation. Our benchmark tool fills the priority queue in parallel with random values. Since in C++, `rand()` is not thread-safe, we use `rand_r(&seed)` with one random seed for each thread. The following code was used:

```
std::vector<unsigned int> seeds(P);
// omitted here: initializing the seeds...
#pragma omp parallel for num_threads(P) schedule(static)
for (unsigned int i=0; i<n_elements; ++i) {
        int val = rand_r(&seeds[omp_get_thread_num()]);
        pq.push(val);
}
```

It turned out that `rand_r()` was a serious bottleneck, data rates were much higher when using a constant or a pseudo-random number depending only on `i`. The reason is that probably all seeds share the same cache line, what causes losses due to synchronization. The solution is to use a thread local variable:

```
#pragma omp parallel
{
        unsigned int local_seed = global_seed * omp_get_thread_num();
#pragma omp for schedule(static)
        for (unsigned int i=0; i<n_elements; ++i) {
                int val = rand_r(&local_seed);
                pq.push(val);
        }
}
```

Avoiding dynamic allocation In the beginning, pointers were used for referring to insertion heaps, to the extract buffer, and to sorted arrays because this makes it easy to manage them and keep data in place. However, this has downsides. First, one has to manually delete objects created with C++'s `new` operator. This encourages memory leaks. Secondly, additional indirections can affect the performance negatively. For example the comparators used for the winner trees access the values behind these pointers very frequently.

We have replaced nearly all pointers and use stack-allocated structures now. To avoid memory copying, some C++11 features are used. For example, the command `internal_arrays.emplace_back(values)` creates an `internal_array` object in place and passes the `values` attribute to it's constructor. The constructor uses `std::swap` to swap the contents of the passed vector into the object's own value vector. This prevents deep copying.

In multiple cases, the priority queue needs to remove empty arrays. We use the common erase-remove idiom:

```
internal_arrays.erase( std::remove_if(
        std::begin(internal_arrays), std::end(internal_arrays),
        is_array_empty_operator() ), std::end(internal_arrays) );
```

The `remove_if` function moves non-empty arrays to the begin and empty arrays to the end. When using pointers to arrays this is no problem. However, with stack-allocated structures this can cause great performance losses due to memory copying. The solution is to define a C++11 move operator. The move operator of a class T moves the contents of another object of type T into it's own object.

Avoiding library calls Some performance improvements have been achieved by removing calls to dynamic libraries like *libgomp* by OpenMP. Like described in chapter 5.5, the `bulk_insert_step()` function inserts an element into the insertion heap that is assigned to the calling thread. Since the insertion heaps are indexed with the thread ids, a call to `omp_get_thread_num()` is necessary. Formerly, this was done in `bulk_insert_step()`, resulting in $|bulk|$ expensive calls to a dynamic library. A better method is to request the id only once per thread and to pass it to `bulk_insert_step()`:

```
#pragma omp parallel
{
        unsigned int thread_id = omp_get_thread_num();
#pragma omp for schedule(static)
        for (unsigned int i = 0; i < bulk.size(); ++i) {
                bulk_insert_step(bulk[i], thread_id);
        }
}
```

7. Experiments

In this chapter we evaluate the performance of our priority queue implementation in different situations.

7.1. Testbed

For the experiments, we use an eight-core system with seven hard disk drives and a 16-core system with four HDDs. Due to hyper-threading there are twice as many virtual cores on both machines. A detailed description can be found in table 2. Read and write bandwidths for singe storage drives as well as for parallel accesses have been measured. The external block size B is 2 MiB on all systems.

Intel8	2 x Intel Xeon X5550 2.66 GHz (8 physical cores in total) L1 cache: 4x32 KiB I, 4x32 KiB D L2 cache: 4x256 KiB L3 cache: 8 MiB shared 48 GiB RAM 7 x SATA: 112 MiB/s r/w each, 537 MiB/s parallel r/w
Intel16	2 x Intel Xeon E5-2650 v2 2.60GHz (16 physical cores in total) L1 cache: 8x32 KiB I, 8x32 KiB D L2 cache: 8x256 KiB L3 cache: 20 MiB shared 128 GiB RAM 4 x SATA: 122 MiB/s r/w each, 455 MiB/s parallel r/w

Table 2: Testing Systems

7.2. Competing algorithms

In order to have an objective view on the performance of our algorithm, we will run the same benchmarks on other algorithms. Not all of them are directly comparable, but still provide comparative values for some parts of the priority queue.

The `priority_queue` class of the Stxxl is the main competitor in our experiments. It uses the same external memory back-end, so influences from different parametrization of hard disk accesses can be excluded. Furthermore, the implementation allows us to easily limit the main memory usage, just as we will limit it for our algorithm. Thereby the volume of the test case can be reduced and data is still written out to EM.

Because the interface is compatible to the Stl it makes sense to compete with it's priority queue, too. Since there there is neither EM support nor a simple way to limit the memory usage, we will only compare the internal memory part of our priority queue with it.

An upper bound for the EM performance is given by Stxxl's `sorter` class. It's an EM sorter consisting of two phases. First, the container is in write mode, during which all elements are filled into it. Internally, the elements are buffered in buckets of size $\Theta(M)$. When the bucket overflows, it is sorted and written to EM. After finishing the write phase by calling `sort()`, there are k sorted runs in EM. The sorted sequence is built by merging them in parallel. This scheme is very similar to the one we use in our algorithm, but Sorter conforms to a weaker problem definition, since ExtractMin operations are disallowed in the write phase.

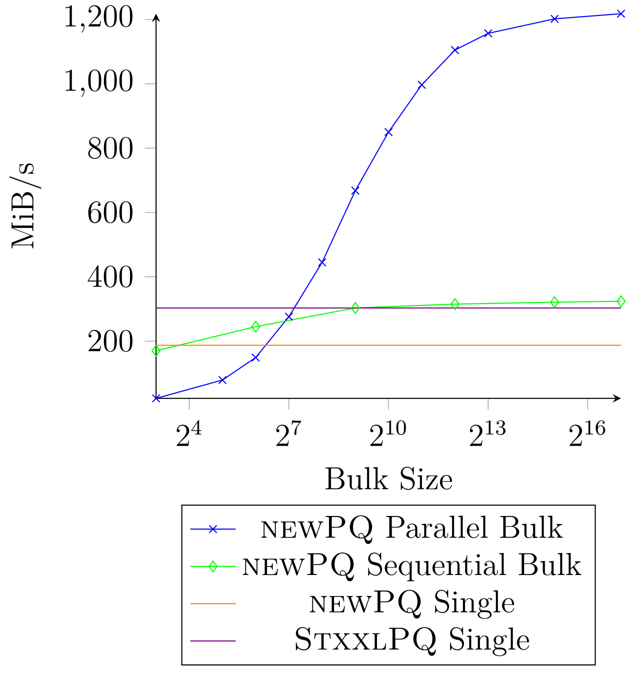
The first part of this chapter focuses on finding adequate values for two important parameters of the priority queue: The number of insertion heaps n_{IHs} and the size of the heaps. Furthermore, we determine the smallest bulk size for which parallel insertion is beneficial. The corresponding benchmarks all run in internal memory, since the choice of these parameters is less crucial in external-memory test cases, where the EM bandwidth is a limiting factor in some cases. Using these parameters, the performance of our priority queue is then intensively compared to the performance of the competing algorithms both in internal and in external memory.

In the following, b will stand for the bulk size, H for the size of a single insertion heap, p for the size of each array's prefetch buffer and w for size of the external array write buffer.

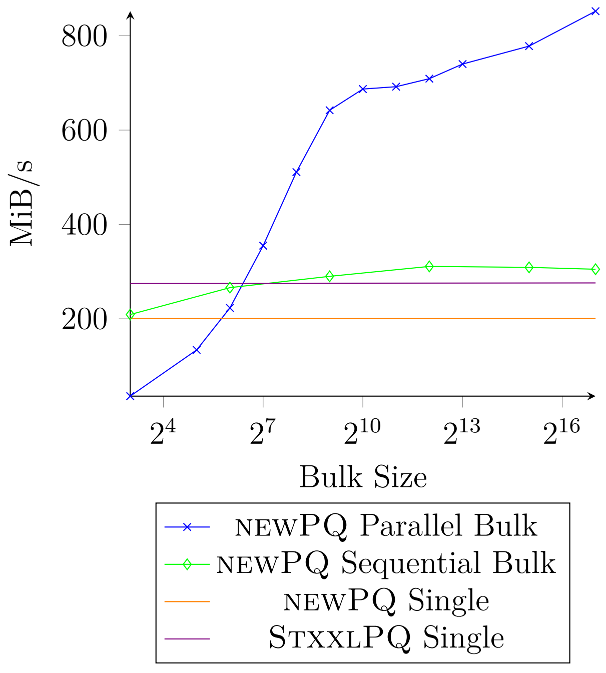
7.3. Bulk Size

For the usage of our bulk insertion interface, it is important to know from which bulk size on it outperforms sequential insertion. Note that there are two variants for sequential insertion: Single insertion (i) and sequential bulk insertion (ii). They differ in the following: (i) inserts the element into a random insertion heap, and updates the minima tree if and only if the root of this heap has changed. (ii) inserts all elements of the bulk into random heaps one after another. We assume that afterwards, the roots of multiple heaps have changed. Therefore, instead of updating the minima tree for each element ($b \cdot \lceil \log_2(n_{\text{IHs}}) \rceil$ comparisons), it is rebuilt as a whole ($n_{\text{IHs}} - 1$ comparisons). In figure 14, we can see that (ii) is faster than (i) for bulk sizes larger than 8. However, this value may depend on the number of insertion heaps.

Parallel bulk insertion outperforms sequential bulk insertion as well as StxxlPQ's bulk insertion for bulks of size > 128. The slight drop of the sequential bulk insertion data rate on Intel8 for bulks larger than 2^{12} seems paradoxical at first glance, but can be explained with suboptimal IM usage: The insertion heaps are emptied if the current bulk doesn't fit into them. The larger the bulks are, the more space in the insertion heaps can be wasted.



(a) Intel16



(b) Intel8

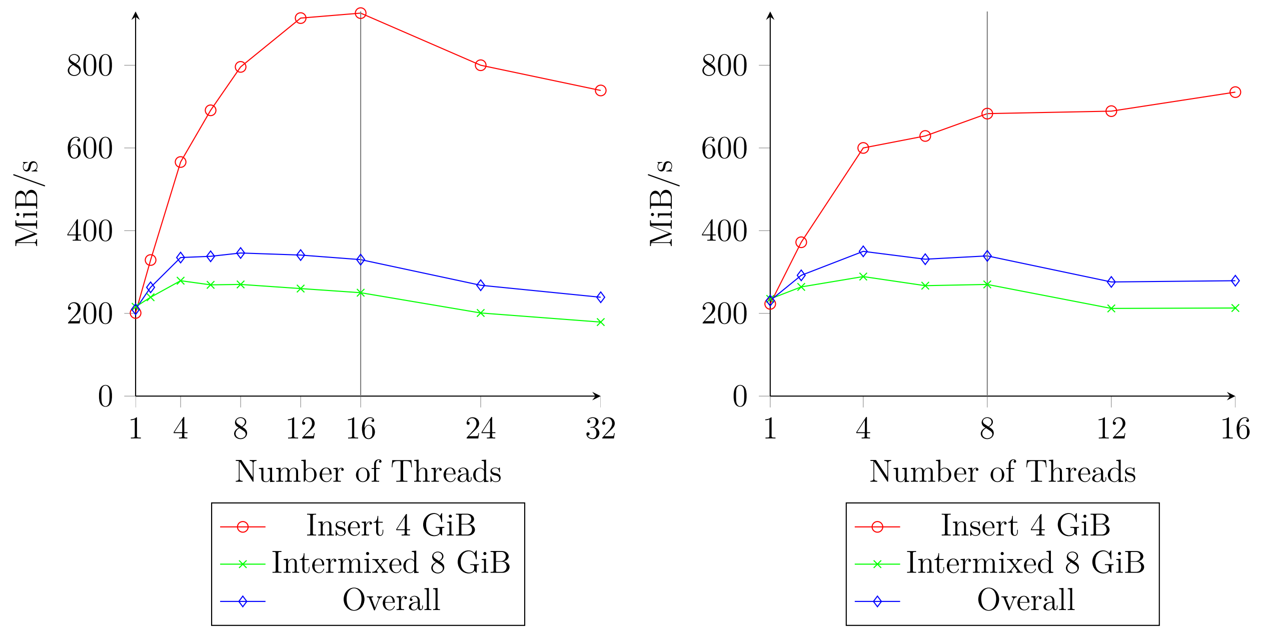
Figure 14: Internal-memory insertion data rate depending on the bulk size
$V = 10$ GiB, $H = 1$ MiB, $M = 48$ GiB, $n_{\text{IHs}} = P_{\text{phys}}$

7.4. Number of Insertion Threads

The number of insertion heaps, which should be equal to the number of threads that insert elements of a bulk, influences the data rate in at least two ways. On the one hand, multiple insertion heaps allow parallel sorting and concurrent PushHeap operations. As long as the number of heaps doesn't exceed the number of cores, more heaps imply faster bulk insertion. On the other hand, each insertion heap needs its own entry in the minima tree, and the larger (i.e. higher) the minima tree is, the more comparisons are needed for *any* change in the insertion heaps.

We can observe this circumstance in figure 15. We ran the test on two systems, both with support for hyper threading. However, (a) shows that the use of hyper threading cores (which share their cache with a physical core) can be disadvantageous.

The intermixed data rate (see chapter 7.7) decreases slightly with a rising number of heaps for $P > 4$ (Intel8) or $P > 8$ (Intel16), because the advantage of parallel insertion cannot fully compensate for the slower deletion. In the remaining tests, we use either $P = P_{\text{phys}}$ or $P = \frac{P_{\text{phys}}}{2}$.



(a) Intel16, 32 virtual, 16 physical cores (b) Intel8, 16 virtual, 8 physical cores

Figure 15: Varying the number of threads / insertion heaps.
$H = 1$ MiB, $M = 48$ GiB, $b = 1024$.
The gray line corresponds to the number of physical cores.

7.5. Heap Size

The size of the insertion heaps is a parameter of great concern. Chapter 5.8 has already stated various influences, the heap size has on the priority queue's performance. An adequate value is now to be found by experiments. Figure 16 shows data rates for different heap sizes.

Besides insert-all-delete-all we also use the test case that is introduced in chapter 7.7. The volume has been chosen so that the insertion heaps overflow three times and are almost completely filled in the end (1 KiB is left). This choice is important, since it influences the number of FlushInsertionHeaps operations (which are quite expensive for large heap sizes) and therefore also the number of sorted arrays. Beginning the delete sequence right after a FlushInsertionHeaps operation would result in disproportionately good ExtractMin performance, since in this case all data is stored in sorted sequences. For homogeneous operations, a heap size of up to 64 MiB has turned out to be a good choice (given that there is enough IM). For intermixed operations, sizes between 10 KiB and 1 MiB are much better. In most cases we use 1 MiB as a trade-off.

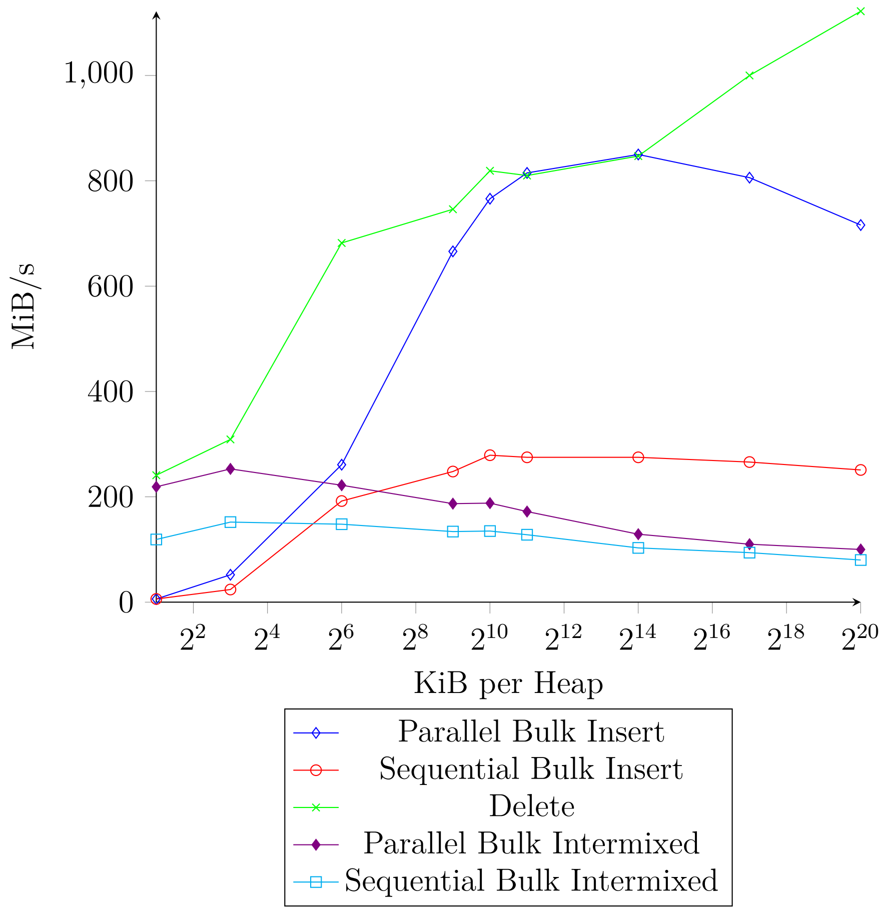


Figure 16: Internal-memory bulk insertion data rate for a varying heap size.
Test case: Bulk Insert V + Delete V, Bulk Insert V + Bulk Intermixed $2 \cdot V$
Intel16. $b = 1024, n_{\text{IHs}} = 16, V = 4 \cdot P \cdot H - 1$ KiB

7.6. Insert-All-Delete-All

A simple and impartial tests case is inserting n elements in a row and deleting n elements afterwards. We ran this test both in internal memory ($M \geq V + c$, c constant overhead) and with use of external memory ($V > M$).

Table 3 lists data rates for insertion and deletion of 10 GiB in each case. The internal memory is sufficiently large so no writes to EM are necessary. Note that Sorter seems to write data to EM, no matter how much IM is available.

	Insert All	Delete All	Overall
newPQ parallel bulk	901	1037	965
newPQ sequential bulk	271	1053	431
newPQ sequential single	166	1015	286
StxxlPQ sequential bulk	286	1202	462
StxxlPQ sequential single	304	1554	508
StxxlPQ sequential bulk (1 thread)	157	1140	276
StxxlPQ sequential single (1 thread)	166	1151	290
Sorter sequential bulk	198	715	310
StlPQ sequential bulk	337	32	59

Table 3: Homogeneous internal-memory insert/delete data rates on Intel16 (MiB/s).
$P = n_{\text{IHs}} = 8, V = 10\text{ GiB}, H = 64\text{ MiB}, b = 1024$

As we can see, in internal memory, newPQ outperforms the other priority queues and even Sorter. Particularly noteworthy is newPQ's bulk insertion data rate of 901 MiB/s, as well as the poor ExtractMin performance of StlPQ for large volumes like this. Furthermore, we recognize the usage of parallel merging in StxxlPQ as it benefits from multiple cores, too. However, the parallel speedup of insertion is only about 1.8 for StxxlPQ in comparison to a factor of 3.3 for newPQ.

Table 4 shows data rates for a data volume, that doesn't fit into IM. $V = 4 \cdot M$ in this case. Naturally, StlPQ cannot compete here. We observe, that the extraction data rates for newPQ and StxxlPQ are lower here. This makes sense because in the IM test case, the data rate is higher than the parallel HDD read bandwidth of the system. The insertion data rates are limited by the EM bandwidth, too. Sorter, designed to sort large amounts of data, is hardly surprising the best algorithm in this test case. Nevertheless, newPQ outperforms the other priority queues in EM-insert-all-delete-all.

	Insert All	Delete All	Overall
newPQ parallel bulk	318	607	417
newPQ sequential bulk	196	617	297
newPQ sequential single	153	629	248
StxxlPQ sequential bulk	271	542	361
StxxlPQ sequential single	293	539	379
StxxlPQ sequential bulk (1 thread)	181	529	269
StxxlPQ sequential single (1 thread)	179	589	275
Sorter sequential bulk	410	580	480

Table 4: Homogeneous insert/delete data rates on Intel8 (MiB/s).
$P = n_{\mathrm{IHs}} = 8, V = 16$ GiB, $M = 4$ GiB,
$H = 1$ MiB, $b = 1024, p = 14 \cdot B, w = 14 \cdot B$

7.7. Intermixed Insert/ExtractMin

The performance for a sequence of intermixed PQ operations is evaluated using the following benchmark: First, n elements are inserted in row. This assures that the PQ operates on an adequately filled state and thus sorted arrays and external-memory storage are taken into account. Afterwards, a loop extracts an element with a probability of $p_{extract} = \frac{b}{b+1}$ and inserts a bulk of size $b = 1024$ with a probability of $p_{insert} = \frac{1}{b+1}$. Of course, this is restricted in a way that nothing is extracted from an empty PQ. The expected value of the number of extractions in a row is equal to the bulk size. The loop finishes when n elements have been inserted and n ones have been extracted. Algorithm 6 shows the procedure in detail.

Algorithm 6: Benchmark Bulk Intermixed

```
Input: Number of elements $n$, Bulk size $b$
$n_i, n_d \leftarrow 0$
for $i \leftarrow 0$ to $n$ do
    $b' \leftarrow$ if $n_i + b > 2 * n$ then $n \mod b$ else $b$
    bulk_insert_begin($b'$)                                  // only for newPQ
    for $j \leftarrow 0$ to $b'$ do
        $r \leftarrow$ random value
        bulk_insert_step($r$)                                // resp. insert()
    bulk_insert_end()                                        // only for newPQ
    $n_i \leftarrow n_i + b'$
for $i \leftarrow 0$ to $3 * n$ do
    $r \leftarrow$ random value $\in \{0, 1, ..., b\}$
    if $n_d < n_i \wedge n_d < n \wedge (r > 0 \vee n_i \geq 2 * n)$ then
        extract_min()
        $n_d \leftarrow n_d + 1$
    else
        $b' \leftarrow$ if $n_i + b > 2 * n$ then $n \mod b$ else $b$
        bulk_insert_begin($b'$)                              // only for newPQ
        for do
            $r \leftarrow$ random value
            bulk_insert_step($r$)                            // resp. insert()
        bulk_insert_end()                                    // only for newPQ
        $i \leftarrow i + b' - 1$
```

There is also a non-bulk variant which is implemented analogous to algorithm 3, but without the inner loops and using $p_{insert} = p_{extract} = \frac{1}{2}$.

Unfortunately, the results for the intermixed test case are not as good as the ones for insert-all-delete-all. Tables 5 and 6 list data rates in internal and external memory.

	Insert 8 GiB	Intermixed 12 GiB	Overall
newPQ parallel bulk	803	284	362
newPQ sequential bulk	307	283	290
newPQ sequential single	205	214	211
StxxlPQ sequential bulk	268	307	293
StxxlPQ sequential single	291	308	302
StxxlPQ sequential bulk (1 thread)	163	240	207
StxxlPQ sequential single (1 thread)	165	236	207
StlPQ sequential bulk	367	64	88
StlPQ sequential single	393	58	81

Table 5: Intermixed data rates in internal memory (MiB/s).
Intel16. $P = n_{\text{IHs}} = 8, b = 1024, M = 48$ GiB$, H = 1$ MiB.

	Fill 8 GiB	Intermixed 16 GiB	Overall
newPQ parallel bulk	252	253	259
newPQ sequential bulk	166	265	221
newPQ sequential single	166	268	223
StxxlPQ sequential bulk	261	282	275
StxxlPQ sequential single	261	284	276
StxxlPQ sequential bulk (1 thread)	157	236	202
StxxlPQ sequential single (1 thread)	159	238	204

Table 6: Intermixed data rates in external memory (MiB/s).
Intel16. $P = n_{\text{IHs}} = 8, b = 1024, M = 4$ GiB$, H = 1$ MiB$, p = 8 \cdot B, w = 8 \cdot B$.

It's particularly striking that the intermixed part is with *sequential* bulk insertion as fast as or even faster than with parallel bulk insertion. Remember that parallel bulk insertion more than doubles the speed of IM-insert-all-delete-all. We spent much time in interpreting this circumstance.

For a more detailed view, we've extracted the exact shares, insertions and deletions have in the intermixed execution time. Table 7 shows that the parallel insertion is nearly three times faster than the sequential one, but deletion is slower with parallel insertion. This is odd since the ExtractMin code is the same in both cases. We've also taken care that the random numbers and their distribution among the insertion heaps are equal for both runs.

	Insert share	ExtractMin share	sum
parallel	10.34 s	77.34 s	87.68 s
sequential	28.68 s	57.35 s	86.03 s

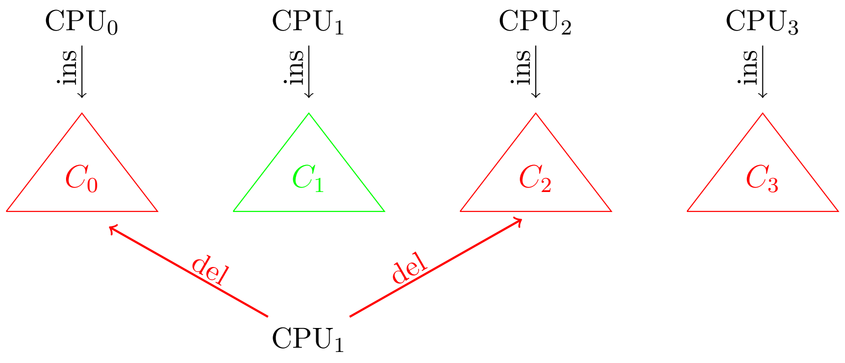
Table 7: Intermixed shares

Cache inefficiency has turned out to be the reason for this. Figure 17 depicts a bulk of four insertions followed by two deletions, both with parallel (a) and with sequential (b) insertion.

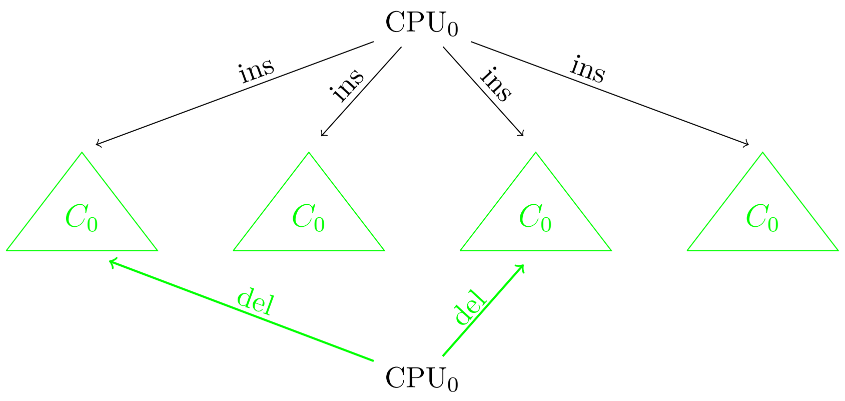
In case (a), each CPU P_i taking part in the bulk insertion updates the insertion heap H_i in it's local cache C_i. ExtractMin is executed by only one CPU P_e and it's not known which one it is. If the minimum value is located in H_i with $i \neq e$, the PopHeap operation is executed on a heap which is not locally cached. The probability for this is quite high ($p_{\text{miss}} = \frac{n_{\text{IHs}}-1}{n_{\text{IHs}}}$).

Since not all heaps fit into one local cache (L2 on Intel16: 256 KiB, heap size: 1 MiB), sequences of consecutive extracts, like they occur in our test case, repeatedly overwrite the local cache. Of course, a solution could be to choose a smaller heap size, but chapter 7.5 has already shown that this has other disadvantages.

In case (b), one CPU C_0 handles all insertions as well as the deletions and thus at least parts of the heaps are cached in C_0. Since the extraction has a large share in the overall execution time of the intermixed test case, this explains the comparatively poor performance of newPQ in the intermixed bulk-parallel test case.



(a) Parallel bulk insertion



(b) Sequential bulk insertion

Figure 17: Two deletions follow four insertions into cached insertion heaps.

8. Conclusions

The objective of this bachelor thesis has been to make much use of parallelism in an external-memory priority queue. We have seen that, in addition to the parallelization of internal routines, there are multiple, in some cases opposing possibilities for providing a parallel interface. Finally, we have chosen a bulk approach where one master thread coordinates sets of concurrent operations.

For the insertion of large bulks, a great speedup compared to the Stxxl implementation has been observed. Unfortunately, the overhead for the thread creation is quite large, so for smaller sets of operations, sequential execution is faster. Furthermore, the winner tree we use for keeping track of the global minimum is more expensive for a larger number of insertion heaps.

The parallel overhead issue also occurred in the context of internal parallelism. We've replaced several parallel sections, since they were slower than a sequential equivalent. In some cases it's hardly predictable whether parallel execution pays off or not. Experimentally determined thresholds have been introduced then. Further improvements are certainly possible here.

We've observed, that the parametrization of the priority queue has a major effect on the performance and may depend on the use case. For example, large heaps are beneficial for homogeneous inserts, whereas for intermixed accesses, smaller heaps are better. The strategies that schedule when to do sorting and how to distribute available main memory are of great importance, too.

A key performance issue occurred in the context of intermixed bulk-parallel Insert and ExtractMin operations. There are cache-inefficiencies when a thread extracts a value from a heap that has been modified by another thread. Solutions to this are conceivable, for example using an intelligent thread pool management. Unfortunately, this conflicts with our implementation (fork and join parallelism) and would therefore go beyond the scope of this thesis.

8.1. Future Work

There are some ideas, how the performance of the priority queue can be further improved.

First, the size of the external array's prefetch buffers is currently fixed. A more flexible dimensioning allows a more generous memory assignment for the buffers and still avoid I/O-intensive external array merges. The amount of external array data being exclusively in internal memory is fixed, too. Dynamically setting this size to more than one block of data for each EA could improve the efficiency of parallel merging due to larger runs.

Analyzing the access patterns of real-world problem instances may also help to improve the performance. Using this knowledge, further work should be done on automatically finding good parameters depending on the machine's properties and suitable for common access patterns.

But the most important step in improving the priority queue is, as already stated above, to introduce a thread pool. Unfortunately, OpenMP doesn't provide such functionality, so additional libraries like *Boost Threads* are needed. A thread pool allows the ExtractMin function to assign the PopHeap instruction to the thread that corresponds to the insertion heap. This is expected to improve the performance of intermixed operations significantly.

A thread pool even allows us to overlap background work of the priority queue and I/O with application processing. Just one possibility among others is the following: The ExtractMin operation immediately returns the minimum value from the minima tree, while updating the minima tree and refilling the extract buffer is assigned to a background thread from the pool. Sorting and merging are tasks that could be done in background, too. One could even conceive intelligent algorithms for assigning additional work (like merging some small external arrays) to a background task while one or more cores are idle.

Similarly, the Stxxl already handles buffered EM accesses in the background and therefore overlaps computation with I/O. But overlapping application processing with any work that doesn't need to be done immediately may be a whole new approach to significantly improve the CPU utilization on parallel machines. Particularly single-threaded applications could profit from that.

A. Appendix

A.1. Tournament Trees

In chapter 5.4, a data structure called *winner tree* is used. Just like *loser trees*, they are a variant of tournament trees. This chapter will introduce both of them in detail and explain why we decided for the use of winner trees, despite the fact that loser trees are expected to perform better on systems with cached memory. In general, a tournament tree allows $\mathcal{O}(1)$ access to the minimum of a set of elements. Replacing the minimum by a new value has a complexity of $\mathcal{O}(\log(n))$.

A.1.1. Winner Trees

There are k players which compete against each other in one-on-one matches for the winning position. k is either a power of two or the game is extended by $k' := 2^{\lceil \log_2 k \rceil} - k$ always-losing players. Then each participant plays against exactly one other one. In further rounds the winners of the previous one play against each other. The construction is finished when there is only one winner left. This results in exactly $k' - 1$ games whose results can be illustrated as inner nodes in a tree of size $\lceil \log_2 k \rceil$ like depicted in figure 18 (left).

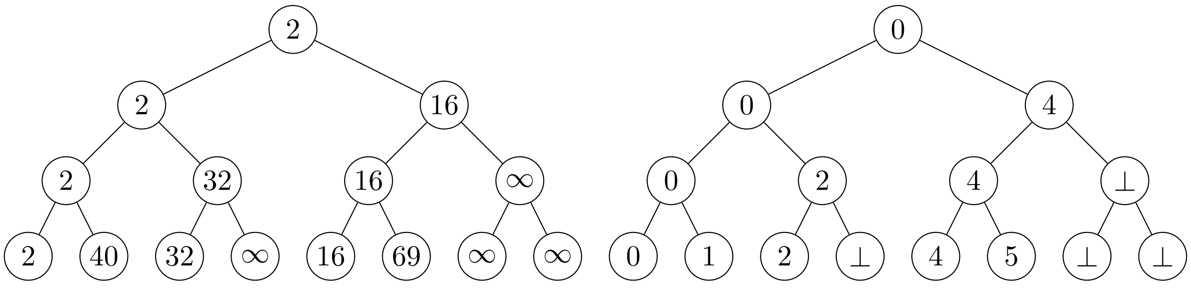


Figure 18: A winner tree (left) and the corresponding index winner tree (right) for the players $\{0 : 2, 1 : 40, 2 : 32, 3 : \text{empty}, 4 : 16, 5 : 69, 6 : \text{empty}, 7 : \text{empty}\}$

The elements we compare using this tree are the elements of the priority queue, which could have any size. Therefore it could be disadvantageous to store them directly inside the nodes, since cache faults are more likely if large elements are moved around. Instead, the leaves are given distinct indices, which are referred to by the inner nodes. We call this variant an *index winner tree* (see figure 18 (right) for an example).

Conveniently, the indices are required anyway because the ReplaceMin operation needs to know which leaf corresponds to the winning value in order to replay all involved games with the new value iteratively up. These are exactly the games on the path from the leaf to the root. Algorithm 7 describes this replay function in detail. Note, that when using indices, a custom comparator is needed which dereferences the indices to their values and determines the winning index.

Algorithm 7: Replay (Winner Tree)

```
Input: Winner Tree $T$, Node labels $L$, Player index $i$
1 node ← leaf $i$
2 while $node \neq root$ do
3     node ← parent of node
4     L[node] ← L[WinnerOfGame(node, sibiling of node)]
```

A.1.2. Loser Trees

There is another type of tournament tree: The loser tree. Each inner node corresponds to the loser of the game played between the winners of the child node games, here. This leaves just one player who is not the loser in any of the $k' - 1$ games and who is therefore the overall winner. Winners are not stored explicitly, except for the overall winner. An example is given in figure 19.

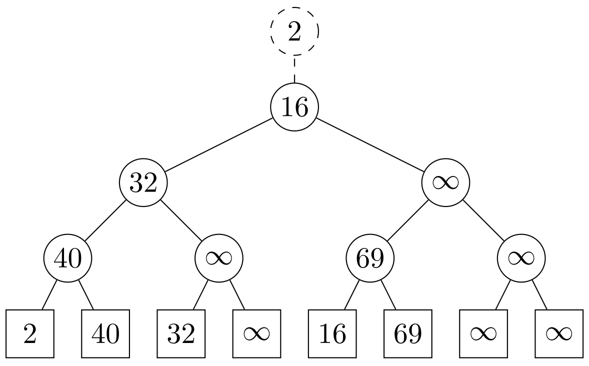


Figure 19: A loser tree for the players $\{0 : 2, 1 : 40, 2 : 32, 3 : \text{empty}, 4 : 16, 5 : 69, 6 : \text{empty}, 7 : \text{empty}\}$. Value 2 wins.

While a loser tree looks less intuitional, it's in many cases the better choice. Let's look at the ReplaceMin operation described in algorithm 8. As you can see, the procedure only references the leaf and its ancestors, which makes two referenced memory locations per iteration instead of three in algorithm 7. This is the main advantage over the winner tree.

Unfortunately, our priority queue does not fulfill the requirements for this kind of loser tree. Quite often, not only the winning player changes it's value (ExtractMin operation), but any other player can do this (e.g. in Insert or FlushInsertionHeaps operations). Algorithm 8 restricts the changing player to the current winner. If we try to eliminate this restriction, we have to reference further memory locations in at least two cases:

(i) We consider the parent p of the updated leaf l. It's possible, that l had been the loser before and therefore $L[p] = L[l]$. In this case we don't know anything about the game result if the sibling of l isn't regarded.

Algorithm 8: Replay after the winning player has changed (Loser Tree)

```
Input: Loser Tree $T$, Node labels $L$, Winning player index $i$
1 contendingValue ← L[leaf $i$]
2 defender ← leaf $i$
  // Note: The winner is not considered to be part of the tree
3 repeat
4     defender ← parent of defender
5     if contendingValue = LoserOfGame(contendingValue,L[defender]) then
6         swap(contendingValue, L[defender])
7 until defender = root
8 winner ← contender
```

(ii) The algorithm has only access to the updated leaf and to all it's ancestors. If the leaf had been the winner, this is sufficient. Otherwise, however, the new overall winner can be a leaf which is not referenced by any of the regarded nodes. An example for this case is given in figure 20. Note that the overall winner is not read in algorithm 8.

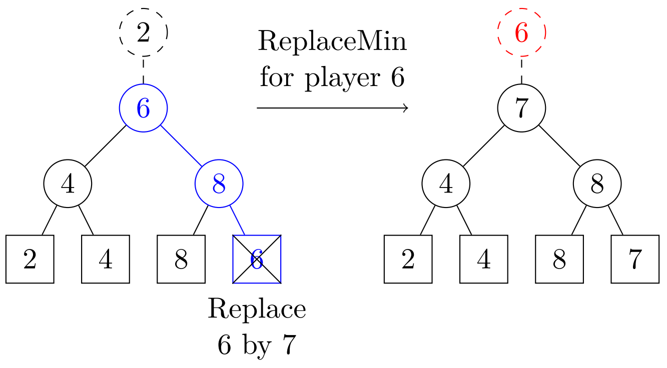


Figure 20: Blue: Regarded nodes. Red: Wrong entry after running ReplaceMin

From this it follows that in our application, a loser tree would have to regard at least two special cases and reference further memory locations. There is no advantage in terms of runtime performance to be expected in this case. Due to that we've decided to use an index based winner tree for keeping track of the global minimum.

References

[1] Ahuja, Ravindra K. and Mehlhorn, Kurt and Orlin, James and Tarjan, Robert E. Faster Algorithms for the Shortest Path Problem. *J. ACM*, 37(2):213–223, April 1990.

[2] Bingmann, Timo and Fischer, Johannes and Osipov, Vitaly. Inducing Suffix and LCP Arrays in External Memory. In *ALENEX*, pages 88–102. SIAM, 2013.

[3] Brengel, Klaus and Crauser, Andreas and Ferragina, Paolo and Meyer, Ulrich. An Experimental Study of Priority Queues in External Memory. In *Algorithm Engineering*, volume 1668 of *Lecture Notes in Computer Science*, pages 345–359. Springer, 1999.

[4] Brodal, Gerth Stølting and Katajainen, Jyrki. Worst-case efficient external-memory priority queues. In *Algorithm Theory — SWAT'98*, volume 1432 of *Lecture Notes in Computer Science*, pages 107–118. Springer, 1998.

[5] Dagum, L. and Menon, R. OpenMP: an industry standard API for shared-memory programming. *Computational Science Engineering, IEEE*, 5(1):46–55, Jan 1998.

[6] Dementiev, Roman and Kettner, Lutz and Sanders, Peter. Stxxl: Standard Template Library for XXL Data Sets. In *Algorithms – ESA 2005*, volume 3669 of *Lecture Notes in Computer Science*, pages 640–651. Springer, 2005.

[7] Dijkstra, Edsger W. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[8] Fischer, Michael J. and Paterson, Michael S. Fishspear: A Priority Queue Algorithm. *J. ACM*, 41(1):3–30, January 1994.

[9] Gerth Stølting Brodal and Jesper Larsson Träff and Christos D. Zaroliagis. A Parallel Priority Queue with Constant Time Operations. *Journal of Parallel and Distributed Computing*, 49(1):4–21, 1998.

[10] J.S. Vitter and E.A.M. Shriver. *Algorithms for parallel memory, I: Two-level memories.* Springer, 1994.

[11] Knuth, Donald E. The Art of Programming, Volume 3: Sorting and Searching, 1998.

[12] LaMarca, Anthony and Ladner, Richard. The influence of caches on the performance of heaps. *Journal of Experimental Algorithmics (JEA)*, 1:4, 1996.

[13] Liu, Yujie and Spear, Michael. A lock-free, array-based priority queue. In *ACM SIGPLAN Notices*, volume 47, pages 323–324. ACM, 2012.

[14] Manchanda, Nakul and Anand, Karan. Non-Uniform Memory Access (NUMA). *New York University*, 2010.

[15] Maria Cristina Pinotti and Geppino Pucci. Parallel priority queues. *Information Processing Letters*, 40(1):33–40, 1991.

[16] Peter Sanders. Randomized Priority Queues for Fast Parallel Access. *Journal of Parallel and Distributed Computing*, 49(1):86–97, 1998.

[17] Pugh, William. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM*, 33(6):668–676, June 1990.

[18] Sanders, Peter. Fast Priority Queues for Cached Memory. *J. Exp. Algorithmics*, 5, December 2000.

[19] Singler, Johannes and Sanders, Peter and Putze, Felix. MCSTL: The multi-core standard template library. In *Euro-Par 2007 Parallel Processing*, pages 682–694. Springer, 2007.

[20] SiSoftware Benchmarks: Intel Mobile Haswell. `http://www.sisoftware.co.uk/?d=qa&f=mem_hsw`. [Online; accessed 30-June-2014].

[21] Sundell, Håkan and Tsigas, Philippas. Fast and lock-free concurrent priority queues for multi-thread systems. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 11–pp, 2003.

[22] Tom's Hardware HDD Benchmark. `http://www.tomshardware.de/charts/enterprise-hdd-charts/-04-Write-Throughput-Average-h2benchw-3.16,3376.html`. [Online; accessed 30-June-2014].

[23] Yi-Jen Chiang and Michael T. Goodrich and Edward F. Grove and Roberto Tamassia and Darren Erik Vengroff and Jeffrey Scott Vitter. External-Memory Graph Algorithms, 1995.