Algorithm Engineering
WS 24/25

Diana Gaus
Lukas Erne

Improved Mergesort through limitation of external memory access

Abstract

Sorting is a fundamental concept in algorithm design, and Mergesort, being one of the most efficient and widely used methods, is frequently employed for this purpose. Although Mergesort already demonstrates good runtimes, its performance can theoretically be further improved by leveraging hardware characteristics to reduce access to external memory, instead relying more on the significantly faster internal memory. This variation is called External Memory Mergesort. After the implementation description we compare both methods, as well as the std::sort function, in the evaluation. Even though the std::sort method outperform our EM-Mergesorts, the results suggest that as input sizes grow, the External Memory Mergesort might be faster than classical Mergesort, especially when larger internal memory and block sizes are used. Therefor we demonstrate that optimization of memory usage does not necessary improve running time for all inputs, but can lead to more efficiency for large input data.

1 Introduction

Sorting datasets is one of the most fundamental practices in algorithmics and has been addressed using many different methods that follow various approaches and are suitable for different situations. Among these methods, Quicksort, Bucketsort, and many others, Mergesort is one of the most well-known and efficient algorithms in its problem domain. Like all algorithms Mergesort is limited by the capabilities of the underlying hardware. One of these limitations is the size of the main memory. In this report, we will investigate a variation of Mergesort designed to address this limitation. The classical Mergesort will be compared with the External Memory Mergesort to demonstrate that the modification of the algorithm allows us to sort files that are many times larger then our computers main memory.
Section 2 will describe both algorithms and their differences, while Section 3 will focus on the implementation of External Memory Mergesort. In Section 4, we describe the experimental setup along with the associated empirical results. Finally, Section 5 will discuss the results and draw conclusions based on the findings.

2 Preliminaries

We consider the following problem: Given a file f, in which the data is stored unsorted. Each line contains exactly one data point x. The output is a file with all the data points sorted by

$$(\forall x \in f) \quad [x_i \leq x_{i+1}]$$

Same as before, each line contains exactly one data point x. In the following we summarize the procedure of classical Mergesort and our improved External Memory Mergesort.

2.1 Classical Mergesort

The classic Mergesort can only sort files that have at most the same size as the computers main memory plus the memory overhead of the algorithm. The algorithm loads file f into main memory at the start of the procedure. The data is then divided into several blocks. The first block contains the first half of all the data, while the second block contains the other half. These two blocks are then each divided into two smaller blocks. This process is repeated until each block contains no more than two elements. Then the merging phase begins. Each block is processed one by one, and the two elements within each block are sorted in ascending order. Afterward, two adjacent blocks are merged. The four elements are compared and sorted again in ascending order. This merging and sorting of adjacent blocks continues until the entire list is restored. The output is written into a sorted file f of all elements. The asymptotic running time of the algorithm is $\mathcal{O}(n \cdot \log(n))$.

2.2 External Memory Mergesort

The External Memory Mergesort consists of two parts. In the first part, the input data is partially sorted. Similar to the description of Mergesort above, the input is sorted by dividing it into smaller sequences and sorting them. The algorithm reads as many blocks of data from the file as can be sorted in main memory. The sorted sequence is then written back into the file. In the second part, the partially sorted sequences are merged in pairs of two, resulting in larger sorted partial sequences. This sorted blocks are written in the External Data again. The larger sequences are merged again, by performing step one again, until all input elements are sorted. The output is a sorted file f of all elements.

Algorithm 1: External Memory Mergesort

```
Input: File $f$, Input size $N$, Internal memory
       size $M$, Block size $B$, Current
       sequence size $X = B$
Output: Sorted file $f$
1  $i \leftarrow 0$;
2  while $i+1 < \frac{X}{N}$ do
3      for $B$ in $X_i \vee B$ in $X_{i+1}$ do
4          Load block $X_1[B]$ and $X_2[B]$ into
            internal memory;
5          Sort block $X_1[B]$ and $X_2[B[i+1]]$ in
            internal memory;
6          if $s$ is ouput file then
7              Store sorted block in file $s$
8          end
9          else
10             Store sorted block in file $f$
11         end
12     end
13     $i \leftarrow i + 1$;
14 end
15 if more than one sequence then
16     $X \leftarrow 2X$;
17     $s \leftarrow f$;
18     $f \leftarrow s$;
19     go to 2;
20 end
21 Output sorted file $s$
```

2.2.1 External Memory Model

As we will see in section 4 in practice, for this algorithm the usual RAM model of evaluation is insufficient. The external memory Mergesort assumes limited main memory space, and has to deal with file operations that impact the performance signigicantly. Therefore the external memory model is used to evaluate the running time. In this model, the IO operations between main and external memory are used as a performance measure. As discussed in the lecture, the asymptotic running time of external memory mergesort in this model is $\mathcal{O}(N \times \log(N))$, where N is the input size.

3 Algorithm Engineering and Implementation

3.1 Algorithm Engineering

The external memory Mergesort initially takes, in addition to the file f, the input size N, the internal memory size M, and the block size B, where B must be at most $\frac{M}{3}$. The elements are then divided into n = $\lceil \frac{N}{B} \rceil$ blocks, with each block containing B elements. In the first sorting phase, each block is its own sequence. The first and second blocks, and thus the first and second regions, are now loaded into the internal memory. The first element of the first block is compared with the first element of the second block. The smaller element is placed in the output buffer of the internal memory. For example, if the first element of the first block is identified as the smaller one, the second element from the first block and the first element from the second block are then compared. This process is repeated until the output buffer reaches a size of B. At this point, the elements in the output buffer are transferred to the external memory in a second file s. Once the first and second blocks are successfully sorted, they form a new sequence of size $2B$. All remaining blocks are sorted in the same manner, forming new sequence of size $2B$. Then the newly formed first and second sequence from file s are compared. A key feature of this algorithm is that once a block from a sequence is fully sorted, the next block from that sequence is immediately provided for comparison. Once the first two sequences are sorted, they form a new sequence and gets wirtten in the external memory file f again. The remaining sequences are processed in the same manner. These newly formed sequences are compared again. These steps are repeated until all elements have been merged into one sequence. Finally, a sorted file f containing all the elements is obtained.

4 Experimental Evaluation

In this section, we evaluate our algorithm by comparing the sorting time with the std::sort function and the classical Mergesort.

4.1 Data and Hardware

Experiments were conducted on a single core of an AMD Ryzen 5 5500U with 2.10 GHz clock speed and an artificially limited RAM size of 64 MB. The implementation was tested on five different input files with roughly 10%, 50%, 100%, 1000% of our RAM size. Our experiments include a comparison of the std::sort function provided by the C++ standart library and our implementation of the internal memory Mergesort on the same input files. We also compared the classical Mergesort to our External Memory Mergesort on the two smallest instances of our input data. For our External Memory Mergesort we used a blocksize B of 16 MB.

4.2 Experimental Results

Figure 1 illustrates the performance analysis conducted on the external memory Merge sort as described in section 4.1. As we can see from the plot, the sorting algorithm provided in the C++ standart library is faster than our Mergesort implementation by a constant factor for the cases where the data can be sorted in main memory. When the data is larger than our main memory, the factor by which std::sort is faster becomes smaller. This confirms that the necessary IO operations for sorting a large file make up a significant amount of running time for the algorithm. Figure 2 compares the computation time of Mergesort and external memory Mergesort on files that fit in main memory. Here, the external memory Mergesort is slower, because we read in the data from the file in blocks of 16 MB, while for the classical Mergesort the data is loaded in one read call. This suggests that in reality, a larger block size than 16 MB could be more performant on our hardware.

5 Discussion and Conclusions

In this conclusion we discuss the overall performance of the External Memory Mergesort on real-world instances. As noted in the experimental evaluation, the External Memory Mergesort cannot compete with our implementation of the classical Mergesort for smaller input data, because of the larger amount of

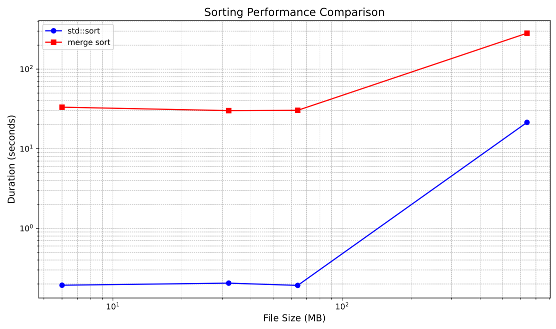

Figure 1: We can see, that ther std:sort outperforms the EM-Mergesort. With increasing file size, the distance increases notably as well.

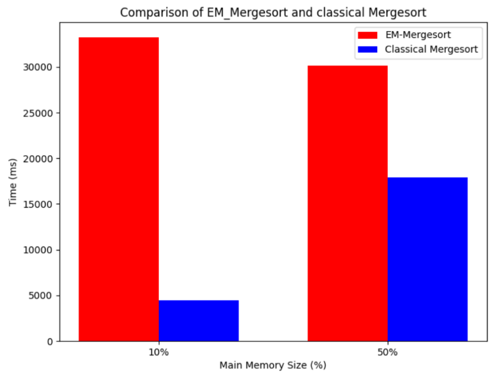

Figure 2: While the classical Mergesort performs better in both file sizes, the distance between the two gets smaller with increasing usage of main memory.

IO operations. However, when comparing classical Mergesort with External Memory Mergesort, it is important to note that the External Memory Mergesort can deal with input data for which the classical merge sort fails. Additionally, the runtime of the EM-Mergesort improves automatically with larger internal memory, resulting in larger blocks.