Algorithm Engineering
WS 21/22

Maxan Mustermensch
Trav Elingsa Lesperson

Algorithm Engineering Report Template

Abstract

We implement and compare the practical performance of two different approaches of the same sorting algorithms, namely, the classical Mergesort algorithm and (2-way) external memory Mergesort, regarding the available internal memory size. We first explain the classical algorithm, then describe methods to improve them to work on files which are by a factor bigger then the available RAM size, and finally discuss the actual implementation. In the experiments, we test and compare our implementations on four file sizes, respectively 10%,50%,100%,1000% of the RAM size.

1 Introduction

We all know common sorting algorithm like Mergesort that work especially well on data structures like arrays. Problem in the real world when we work on real data, which has an hole different size magnitudes it becomes impossible to handle it the same way. So how can we adapt the algorithm to work on Data which are by a factor bigger than our available memory to work on? and more important how do we still stay efficient while working with such huge input sets?

2 Preliminaries

One of the most common operations executed on data is sorting. At first hand this sounds like an easy task but the problem on hand is efficiency. Looking at data and comparing it costs us a lot of time so to be efficient we try reduce these operations to a minima. The Classical Mergesort has been proven to be by the most efficient sorting algorithm with an average time of $\mathcal{O}(n \log n)$.

2.1 Classical Mergesort algorithm

The idea of the classical Mergesort is to sort data by using a divide and conquer principle. We start to split the data in half, which will result in two separate parts of roughly equal size. We then repeat the splitting procedure with each part. After two splits we have four sub parts of our data each having approximately 1/4 of our original data size. We continue to split the fractions until we only have fractions containing one element each, meaning that if we have k elements in our data, we end up having k fractions with one element each.

Next, we start merging (sorted) fractions back together. We can assume that every fraction containing only one element is already sorted. After merging any sorted fraction with x elements together with any other sorted fraction containing y elements, we end up with a sorted fraction containing x+y elements. We start off by merging the smallest fractions together, resulting in bigger sorted fractions. We repeat this process until we only have one large sorted "fraction" left. This is when our whole data will be sorted.

2.2 External memory Mergesort algorithm

When the data D to be sorted exceeds the available memory size, it gets more complicated. When the data size was less than our memory, the procedure is fairly simple. Like in the Classical Mergesort algorithm, we first load the entire D from the hard drive into our memory and sort it internally. We might use any sorting algorithm of our choice. However, as soon as our D is larger in size than our memory we run into problems. We have to divide D into smaller fractions in order to load these fractions into our RAM so we can sort them. We then have to flush the sorted frac-

tions back to our hard drive and continue this process until D is completely sorted.

The basic idea of our implementation is, that we divide our data into several (sorted) blocks which have roughly one third of our memory size. Each block b contains a maximum M numbers, although the last block might (and most likely will) contain less than M numbers. We then use External memory Mergesort algorithm to read two blocks into our memory and merge them together, creating a third block. Once the third block also reaches a size of roughly one third of our memory, we slowly run out of RAM-space, which is why we flush it back to our hard drive in order to make space. We can then continue to merge until block one and two are empty (which is where we can pull two new blocks from our hard drive into our RAM). This way, we only operate within our RAMs limits, but are able to sort data of any size.

3 Algorithm & Implementation

Our External Memory Mergesort consists of two different sections:

1. We call a function which creates the initial blocks for us. This way, we partition our data into fractions which we sort along the way. We start by creating an array A which has roughly one third of our memory. As we know our block size (which will be given by the user), we know how many numbers we can fit into A. Next, we read our data D line by line from our hard drive into our array until it's full, in which case we flush it back to our hard drive, resulting in our first sorted block B. Afterwards we can empty A again in order to make space in our memory and continue this process until D has been read completely.

After this function we have transformed D, in which we now have multiple sorted blocks. Createblocks() also returns the block quantity, meaning how many blocks we have. This information will come in handy in our second section of code.

2. In the second section of our code, we merge the different sorted fractions together, resulting in sorted fractions with twice the size. We used a concept which we call super blocks. The idea is, that we consider our data (which already consists of multiple blocks B) to consist of multiple super blocks SB (which are sorted). One SB consists of at least one standard block. In our second function we iterate through our data, merging super blocks of size k to super blocks of size 2k, beginning with super block 0 and super block 1. We then continue to merge super block 2 and 3, and so on until there are no super blocks left. In the beginning our number of blocks is equal to our number of super blocks, as each super block consists of only one standard block. However, after the first call of our second function (MergeBlocks()), every super block consists of 2 blocks. After each call of MergeBlocks(), the size of our super blocks double, hence the number of super blocks roughly halves. We repeat this procedure until there is only one super block left, resulting in our whole data being one big sorted super block.

In order to achieve this complex goal, we start by opening two readers (Br1 and Br2). Br1 and Br2 each read one super block (being read block by block). We therefore have to adjust our readers to the appropriate positions. For example, in the beginning, Br2 has to first read past the first super block, so it can read the second one.

Once the readers are properly adjusted, we can start a while loop. Since we know how many super blocks we have in the beginning (because SB = B), and the number of super blocks we have halves each round, we always know how many blocks we have to merge.

We iterate over each pair of super blocks (0/1, 2/3, 4/5,... etc) and create three separate arrays.

A1 gets filled block by block from the first super block of our pair. A2 gets filled from the second super block of our pair. A3 gets filled by continuously merging A1 and A2 and is flushed when its full.

By only using these 3 arrays we ensure that we stay within our RAMs limits.

We fill up A1 and A2 until one of the two super blocks is completely read, which is when we simply read the remaining super block into A3 and simply flush it to our hard drive.

After every pair of super blocks has been merged, the MergeBlocks() function is over. The result is that

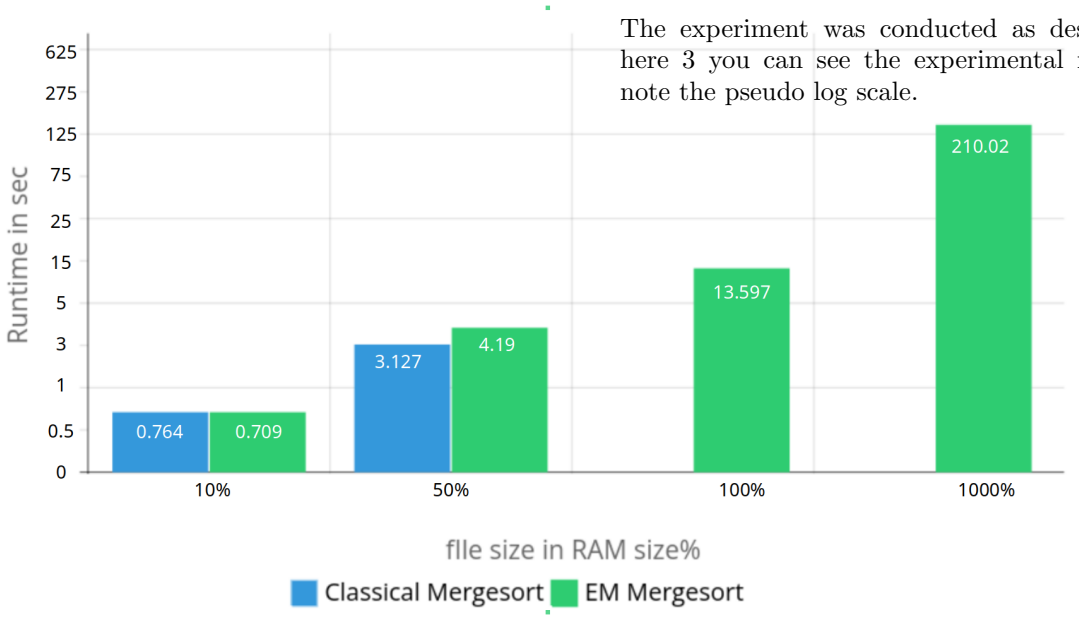
we now have half the number of super blocks, but each super block having twice the size than before.

We now call the MergeBlocks() function again, and repeat this process, until the number of super blocks is equal to 1.

Regarding the AE-principles we strongly relied on the AE-cycle. We started off by creating an abstract idea of how our code could look like. We then started to implement it step by step, fixing it along the way until it worked as expected. We then tested it on various sizes of data, finding out that recursion was sub optimal, as it forced us to close the Filereaders every time we call the function recursively. This way, run time was incredibly long. We then thought of ways to improve the performance of our code and implemented our ideas. We repeated this process over and over again until we were satisfied with the results.

As for libraries we used BufferedReader/-Writer and Filereader/-Writer in order to efficiently read and write our hard drive. As they required IO-Exceptions, we also used the IOException library.

For sorting our arrays in createBlocks() we used the java.Util.Arrays library.



4 Experimental Evaluation

The test was conducted on the three File sizes 10MB, 50MB, 100MB and 1000MB. The classical Mergesort could only be tested on the two smallest file sizes since it load the complete data into the RAM so the file size has to be smaller. the RAM was limited to 100MB as described in 4.1. The Runtime was measured using the time command in fish shell. We looked at the Execution time. As resulting Runtime we took the average of three runs. We user the same Blocksize of 25MB (25% RAM size) for each run. For each test we created a new input file and deleted the old output file to assure as similar condition as possible for each run.

4.1 Data and Hardware

Experiments were conducted on a single core of an Intel i7-2600 CPU with 3.80GHz. The used RAM size was limited to 100MB by using the java execution flag -Xmx100M.

4.2 Results

The experiment was conducted as described in 4. here 3 you can see the experimental results.please note the pseudo log scale.

RAM Size in MB	Block Size in MB	file size in MB ±4Byte	File Size in % of RAM size $\pm 10^{-7}$%	Classical MergeSort runtime in secs	EM MergeSort runtime in secs
100	25	10	10	0,764	0,709
100	25	50	50	3,127	4,19
100	25	100	100	N/A	13,597
100	25	1000	1000	N/A	210,02

Table 1: The running time of classical Mergesort and EM Mergesort on different size input Data. showing some more infos on the data sets and the accuracy.

5 Discussion and Conclusion

In this work we reviewed the implementation and performance of both the Classical Mergesort and External Memory Mergesort algorithm. We showed that we were able to sort data significantly larger than our memory.

While there was no significant difference between the running time of Classical Mergesort and External Memory Mergesort regarding data being 10% and 50% of our RAM, there is a big difference when it comes to data size exceeding our RAM. This is due to the fact that we cannot even perform the Classical Mergesort with data larger than our memory as our whole data need to be loaded in it. To be exact, even data of size 100% RAM cannot be sorted by Classical Mergesort due so small amounts of RAM being used for e.g. pointers and variables.

Though, it comes at no surprise, that Classical Mergesort and External Memory Mergesort have similar run times when using 10% or 50% of our RAM, as in both algorithms we simply load the whole amount of data into our memory for sorting.

However, the External Memory Mergesort is mainly limited by the IO operations it needs in order to read the data from our hard drive into memory and later flush it back to the hard drive. These operations are costly, which is why it is of major importance to reduce the number of IO Operations to a minimum.