Algorithm Engineering
WS 22/23

Arsen Ukrainets
Kevin Kusche

Comparing EM-MergeSort and Classical MergeSort

Abstract

We implement EM-MergeSort, which is a sorting algorithm that allows sorting of data which is larger than the main memory. We compare its practical performance depending on the RAM size and compared to the classical MergeSort. First we explain the classical MergeSort algorithm, then look at the modified version, and also discuss its implementation. In the experiments, we test and compare our implementations on random files of different sizes. The results suggest that the number of I/O operations has a bigger impact on the running time the bigger the input data is.

1 Introduction

Sorting an input sequence is a very common task and consequently there are many different sorting algorithms each with their own advantages and disadvantages. However for these algorithms to work, we typically assume that the data fits into main memory in its entirety, which might not always be the case, especially seeing how the amounts of data which are processed become increasingly large.
That's why, in this report, we look at a sorting algorithm specifically modified to deal with input data which doesn't fit into main memory. We implement this algorithm, EM-MergeSort, and compare it to the classical MergeSort as well as to another instance of EM-MergeSort with bigger RAM size. Through these comparisons we observe that for smaller data, the number of I/O operations doesn't matter as much as for bigger data.
The remainder of this work is structured as follows. Section 2 explains the basic underlying algorithm. Section 3 deals with the modified version and its implementation. In Section 4, we describe the experimental setup and provide empirical results. Finally, in Section 5, we discuss and interpret these results.

2 Preliminaries

The problem is as follows: We're given a file containing N integer values and some main memory, which can hold at most M integer values, with $M < N$. The goal is to sort the content of the file, so that after running the algorithm, we have a file containing the same values but in the desired order.
Though it's not possible to only use a standard sorting algorithm under the given conditions, we rely on one to sort parts of the input, which is why we briefly cover MergeSort in the next subsection.

2.1 MergeSort

MergeSort consists of two phases, the splitting phase and the merging phase.
In the splitting phase the input is recursively split into two subsets of (almost) equal size until each subset contains only 1 element.
In the merging phase two subsets are recursively merged into one until the result of the last merge yields the whole set. Because of the precondition, that before every merge both subsets are sorted, a merge can be performed in linear time, resulting in a total running time in $\mathcal{O}(N \log N)$.

3 Algorithm & Implementation

As already stated, sorting an input which doesn't fit in main memory can't be done with any of the classical sorting algorithms. Therefore, to achieve this goal, we use a modified version of MergeSort, called EM-MergeSort.
In this section we first describe how EM-MergeSort works and how the modifications compared to the

classical MergeSort impact its running time and then provide some implementation details.

3.1 EM-MergeSort

Since the input data doesn't fit into main memory, EM-MergeSort works by only loading in parts of the input at a time.
In the first round, the whole internal memory is filled with M values, which are then sorted with a classical sorting algorithm (MergeSort in our case). Then they are written back into external memory, after which the next M values are transferred into main memory etc. This is repeated until we reach the end of the input. We therefore end up with $\lceil \frac{N}{M} \rceil$ data runs of size M (except for the last one, which might be smaller than M).
After the initial step it's now no longer possible to load in all the data we want to merge at the same time. So instead when we want to merge two data runs, we only load in the first block of each of them and we also reserve one block of space as an output buffer. Now we simply merge the two blocks normally and write the result into the output buffer until one of the input blocks is fully processed or the output buffer is full.
In the first case we simply load in the next block of the respective data run, if there is one. In the latter case we write the output back into external memory and clear the output buffer. We repeat this until every data run is processed. In each round, the number of data runs is halved until eventually there's only one fully sorted data run left.
Let's first take a look at the number of I/O operations in the EM model. During the initial phase every block in the input needs to be read and written once, which results in $2N/B$ I/O operations, for a block size of B. After that, there are $\log_2(N/M)$ rounds. In each round every block in the input needs to be read and written, just like in the initial step, yielding $2N/B$ I/O operations per round. This sums up to an overall number of $2N/B(1 + \log_2(N/M))$ I/O operations.
Now to examine the running time in the RAM model. During the initial phase, we perform classical MergeSort on M elements at a time, which we do a total of N/M times. The initial phase therefore takes time in $\mathcal{O}(N/M(M \log M)) = \mathcal{O}(N \log M)$. After that, as already mentioned, it takes $\log_2(N/M)$ rounds to sort the whole data and in each of these rounds, we merge all of the runs in linear time in the number of input elements.
In total we end up with a running time of $\mathcal{O}(N \log M + N \log(N/M)) = \mathcal{O}(N \log N)$, which is the same as the running time of classical MergeSort.

3.2 Implementation

We implemented the algorithm in Java. For the type of values we used Integers and we stored them in binary files.
The classical MergeSort takes an int array and sorts it. After completion the array which was passed into the method is sorted non-descendingly.
To read from and write to the binary files we used DataInput- and DataOutputStream respectively. In particular we wrote a new class inheriting from DataInputStream, extending its functionality so it could detect the end of a data run and read in a specific number of blocks instead of a number of Bytes. Since the values are stored as Bytes, we first need to convert them to Integers, before we can try to sort them. For this we used ByteBuffer and the method asIntBuffer().
The rest of the method works just like the algorithm described in subsection 3.1 for the most part, following are some details: The RAM size is set in the constructor, as it doesn't change. The actual method takes the input file name, the output file name and the block size as arguments. After the initial phase, at the start of each round, the input and output file are swapped. Then two input streams, one for each data run to merge, are opened on the input file and one output stream is opened on the output file. At the end of a round, the streams are closed. This way only two files are needed.

4 Experimental Evaluation

In this section we evaluate our algorithm by running it on randomly generated files of different sizes

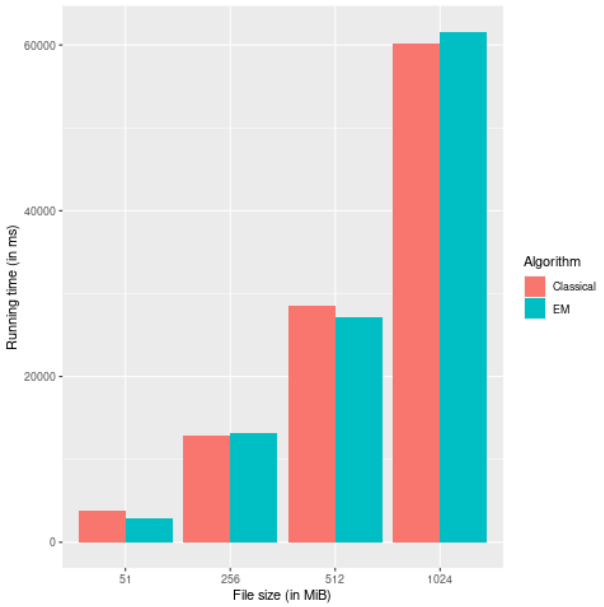

Figure 1: Comparison of the classical MergeSort and the EM-MergeSort (RAM size 512MiB).

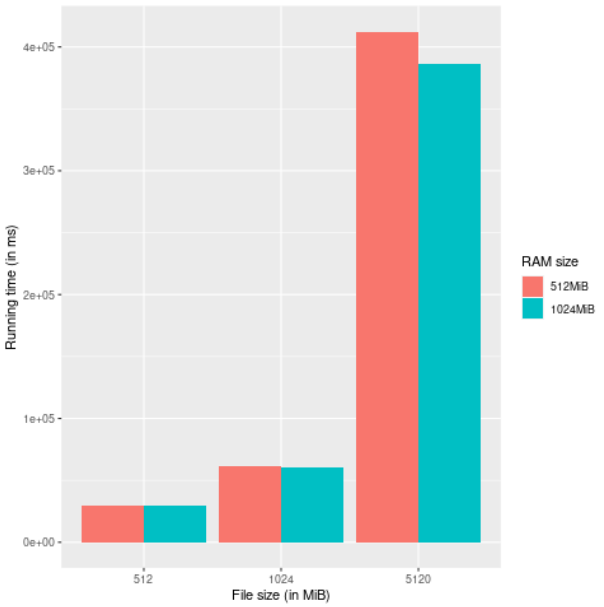

Figure 2: Comparison of the EM-MergeSort with two different RAM sizes.

(51MiB, 256MiB, 512MiB and 5120MiB) and comparing it to the classical MergeSort.

4.1 Data and Hardware

The tests were conducted on an AMD Ryzen 5 4500U with 2.38GHz and an artificially small RAM size of 512MiB and 1GiB, with a block size of 128MiB.

4.2 Results

The running times for the two algorithms on files of the respective size are shown in Figure 1. For files which fit into main memory, the running time is almost identical, which comes as no surprise, since in that case EM-MergeSort just runs classical MergeSort. However even for a size of 1024MiB the difference is fairly small with around 1.5 seconds, which could just be random, seeing how the EM-MergeSort was about 1.4 seconds faster for the 512 MiB instance. To further examine this matter, we also compared the EM-MergeSort with two different RAM sizes as shown in Figure 2. For the smaller instances no significant difference is apparent, but for the larger one we can observe a sizable discrepancy of about 26 seconds.

5 Discussion and Conclusion

In this work we looked at how to efficiently sort data which is too big to fit into main memory. To this end we implemented EM-MergeSort and compared its running time to the classical MergeSort. For the file sizes which allowed the application of the classical MergeSort, the difference was not very significant, which is surprising, considering that for the biggest instance EM-MergeSort takes two rounds to complete (including the initial round) while the classical MergeSort only needs one round, which should incur two times the amount of I/O operations. Therefore it seems like the number of I/O operations isn't the most important contributor to the running time in this case.

However when comparing the different RAM sizes we did observe a notable difference for the file of size 5120MiB. Here the algorithm with a RAM of size 512MiB at its disposal takes five rounds, the other one only four. Since the difference seems to grow with the size of the input data, we assume that the EM model becomes more relevant for larger data, but for data which is only slightly bigger than the main memory, the RAM model seems more applicable.