Algorithm Engineering
WS 22/23

Jakob Sanowski
Fiona Riesterer

Algorithm Engineering Report Template

Abstract

We implement and compare the sorting algorithms Mergesort and Em-Mergesort for different input sizes. We first explain the basic algorithms and then discuss our implementation. In the experiments, we test and compare Mergesort and Em-Mergesort on files with four different input sizes. The result shows that

1 Introduction

In times of big data, a huge amount of data may has to be sorted. So you would need another sorting technique than classical Mergesort, since not all the data fits into the memory. External memory Mergesort is an algorithm, that was developed from the classical Mergesort. Maybe, it is suitable to use EM-Mergesort also for data sizes that classical Mergesort could handle and that's what we tested during the process.

In section 2, an explanation of the two algorithms and their main differences is given and section 3 gives implementaio details. Our experimental setup and the consequent results are explained in section 4. Eventually, in section 5, we discuss our results.

2 Preliminaries

The preliminaries provide the reader with necessary background information. In this section the basic algorithms and the ideas behind them are explained. The algorithms solve the following problem: Given an unsorted sequence of integers, we want to generate a sorted sequence of those integers.

Both, classical and external memory Mergesort use the divide and conquer approach and are easy to understand. Further explanations are given in the following.

2.1 Classical Mergesort

Classical Mergesort divides a given sequence of integer into sequences half the length of the initial one until the sequence has length one. Then these small sequences are merged recursively together by comparing the contents and creating a sequence the size of the initial two combined. This is done until all subsequences were merged together, which will then be the sorted result. Since the input sequence is splitted in half, the running time of classical Mergesort is in $O(n \cdot \mathsf{log}\ n)$.Algorithm 1 provides the pseudocode of classical Mergesort.

2.2 External Memory Mergesort

External memory Mergesort is working nearl exactly the classical Mergesort. But instead of loading the complete input into memory, only blocks of a given size are loaded and merged. Later on, every sorted partition consists of multiple blocks and these partitions are merged by loading the first block of each partition into memory, merge them and write the sorted sequence back into external memory. As soon as a block is merged completely, the next block of the partition is loaded and so on until there is only one partition left, which is then sorted. So external memory Mergesort results in a running time in $O(N \cdot\ \mathsf{log}\ N$ where N is the input size. Algorithm 2 shows the pseudocode for external memory Mergesort.

3 Algorithm & Implementation

We did not actually improve either of the algorithms, but tried to implement it as efficient as possible to get comparable results. In this schapter we describe how we implemented the two algorithms and what datastructures we used.

Algorithm 1: Classical Mergesort

```
Input: unsorted integer Array A
Output: sorted integer Array A
mergesort(Array A)begin
    if A.length == 1 then
        return A
    else
        sortedA1 ← mergesort(A.firstHalf)
        sortedA2 ← mergesort(A.secondHalf)
        return merge(sortedA1, sortedA2)

merge(Array A1,Array A2)begin
    Array A
     ← newArray[A1.length + A2.length]
    indexA1 ←0
    indexA2 ←0
    indexA ←0
    while indexA1<A1.length
     indexA2<A2.length do
        if A1[indexA1]≤A2[indexA2] then
            A[indexA]← A1[indexA1]
            indexA1++
        else
            A[indexA]← A2[indexA2]
            indexA2++
    while indexA1<A1.length do
        A[indexA]← A1[indexA1]
        indexA1++
    while indexA2<A2.length do
        A[indexA]← A2[indexA2]
        indexA2++
    return A
```

Algorithm 2: External Memory Mergesort

```
Input: unsorted sequence S in file F1, Blocksize bSize
Output: sorted sequence
begin
    // load as much blocks as possible into main
       memory and sort them
    partitionSize← memorySize/bSize
    File file1 ← F1
    File file2 ← F2
    while not all blocks of S were loaded do
        // load as much blocks of S as possible into
           memory
        Array A
        while A.length< partitionSize do
            A.add(read(nextBlock))
        A.sort
        write(A).into(file2)
    // Merge every partition until there is only one
       left
    while partitionSize != file1.size do
        partition1←0
        partition2← partitionSize
        merge(partition1,partition2,bSize,partitionSize,file1)
        partitionSize←2·partitionSize
        // Swap Files here to always write into unused
           file
        swap(file1,file2)

merge(part1,part2,bSize,partitionSize,file) begin
    Array A[bSize]
    indexA1 ←0
    indexA2 ←0
    indexA ←0
    Array A1 ← read(part1, bSize)
    Array A2 ← read(part2, bSize)
    while part1 and part2 have still blocks left to load
     do
        while indexA<bSize do
            if A1[indexA1]≤A2[indexA2] then
                A[indexA]← A1[indexA1]
                indexA++
                if indexA1¡bSize-1 then
                    indexA1++
                else
                    A1 ← read(part1.nextBlock, bSize)
            else
                A[indexA]← A2[indexA2]
                indexA++
                if indexA2¡bSize-1 then
                    indexA2++
                else
                    A2 ← read(part2.nextBlock, bSize)
        write(A).into(file)
```

3.1 Implementation Details

We implemented both algorithms using Java 19. We used binary files to read from and write to and integer arrays to store each loaded block in the main memory. To keep track of where to read, we managed pointers for each first block of two neighboring partitions, that will be merged together. We also needed to know the partitionsize which would double each round until there is only one big partition left.

We generated our own binary files using Rust. We simply added random integer numbers until the given file size is met.

4 Experimental Evaluation

In this section, the experimental setup is described and the results are presented.

4.1 Data and Hardware

Experiments were conducted on a single core of a Intel i7 6700K CPU with 4.2GHz. The system has 16GB of RAM though for the RAM usage was limited to 10MB so that files with up to 10 times the RAM-size could be tested in a reasonable amount of time. To see how the block size affects runtime, we tested each file with a block size of 128kB, 1MB and the (roughtly) largest possible block size of 3384kB.

The test data was created using the program located in the filegen folder of the project. To create a file the program takes a size in kB, a range of numbers and a path. The file is then save at the given path with the specified size, containing random numbers in the specified range. Tested file sizes are 1MB, 5MB, 10MB, 100MB and 1GB.

The test were done using the script in the test folder.

4.2 Results

Figure 1 shows the observed test results. Classical merge sort could only be tested on the 1MB and 5MB file because it needs $n * 2$ of memory.

We can see, that classical merge sort is the fastest on files, that can be completely sorted in memory though only by a small amount. An interesting observation can be made when comparing the files smaller than memory size with the files bigger than memory size. It seems that on files smaller than memory a smaller block size is better. On bigger files we can start to see the benefits of a larger block size though interestingly the algorithm performed better with a 128kB than 1MB block size on the 1GB file.

5 Discussion and Conclusion

In this work, we discussed how merge sort can be expanded to also be able to sort data bigger than half the available memory. This is necessary because in the real world we don't have unlimited memory available but still want to be able to sort data which doesn't fit into memory. As expected, classical merge sort still is faster on data that fit into memory due to less overhead but our expanded merge sort isn't far behind. One thing to note is that none of our algorithms utilise parallel execution. This could be used to further boost performance by parallelising the merging of the sub-lists.

6 References

	1MB	5MB	10MB	100MB	1G
128kB	0,058	0,244	0,573	5,962	67,969
1MB	0,07	0,233	0,567	5,68	68,175
3382kB	0,06	0,26	0,533	5,689	65,629
classic merge sort	0,044	0,21			

Table 1: Test results of em-merge-sort (in seconds)

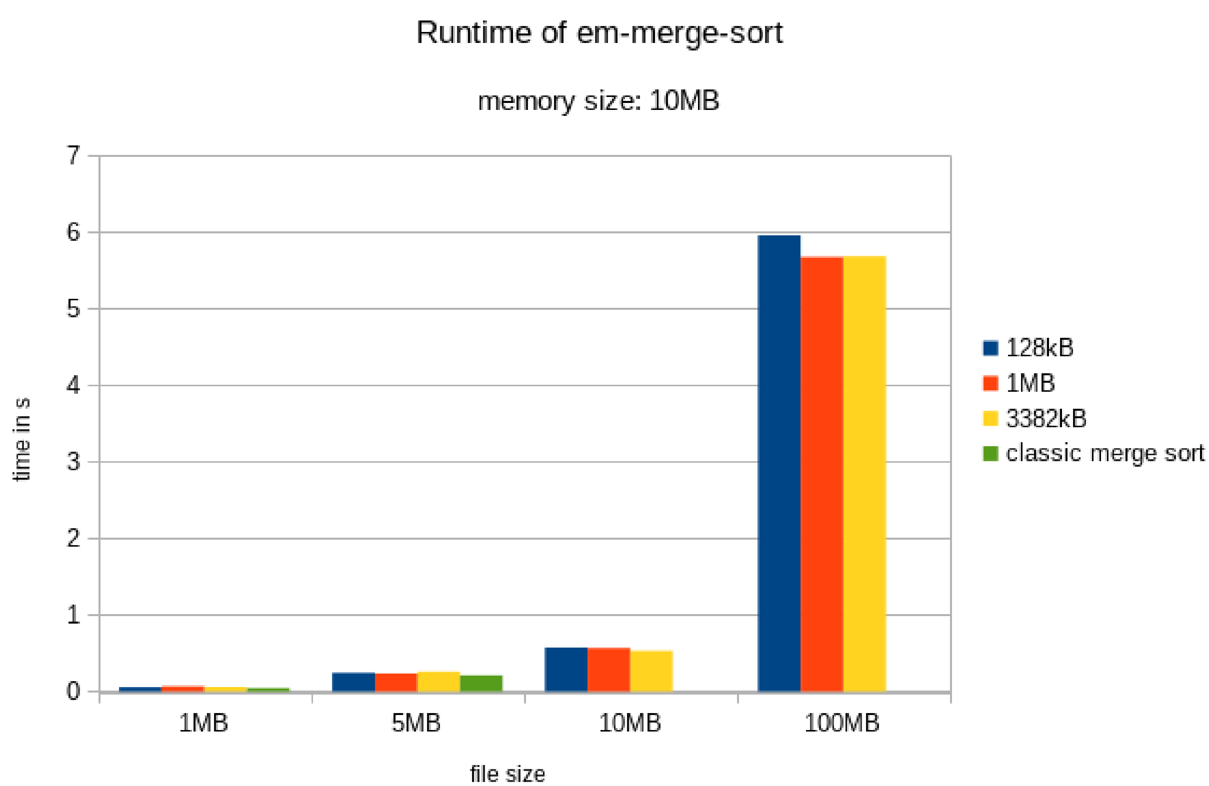

Figure 1: Diagram of the test results. 1GB is not shown because it takes multiple times longer than the rest.