Project Report

Algorithms in Secondary Memory

Database Systems Architecture
INFO-H-417
07.01.2018

Marie Elisabeth Heinrich – 000457502

Jayanthi Kambayatughar – 000457113

Todi Thanasi - 000455013

Table of Contents

List of figures

List of tables

1. Introduction and Environment

This section shall give an overview of the objectives of the project, the structure and the environment used to develop the application and run the performance testing.

1.1. Motivation and Objectives

Nowadays, the IT world is facing more and more data of an immense size, which no longer necessarily fit in memory and may create huge cost if handled inefficiently. Therefore, several ways to read and write files of different sizes shall be examined to gain real-life experience with the theoretic concepts discussed in the lectures.
The objective of this project in particular is to get familiar with different options to read and write data from/to secondary memory as well as implementing an external-memory merge-sort algorithm, complemented by an evaluation of the respective performances.

In section 1.2 we first shall describe all system specifications that were used throughout the project. The basic theoretic concepts of memory mapping and multiway-merge-sort algorithms are explained in section 2. In section 3 we briefly describe our implementation of different read and write streams, differing in input file size (N), buffer size (B), as well as number of streams (k), and discuss the results of our performance tests. Implementation and performance evaluation of the merge-sort algorithm is discussed in section 4 and a final conclusion is drawn in section 5.

1.2. Project requirements and system specifications

Programming Environment	
Programming Language	Java
Version	Version 9
IDE	IntelliJ IDEA 2017.2.5
External Libraries	• Log4J - Core 2.10 (log4j-core-2.10.0.jar) used for logging. • Log4J - Api 2.10 (log4j-api-2.10.0.jar) used for logging. • Apache Commons IO 2.6 (commons-io-2.6.jar) used for I/O related programming. • Apache Commons Lang 3.7 (commons-lang3-3.7.jar) used for simplification of programming. • Google Guava 19.0 (guava-19.0.jar) used for simplification of programming.

System Specifications for Testing

Operating System	Ubuntu 16.04
Version	Kernel version 4.13.0
System Manufacturer	Linux
System Type	Virtual Machine, running on Google Cloud
Disk Type	SSD
Size	100GB
Block Size	4,096 bytes
Processor	2 vCPUs Intel Haswell 2,3 GHz Intel Xeon E5 v3
RAM	7.5GB

Test Data

We built a FileWriter class which is responsible for both - write stream testing and test data file generation. In the files we write random 32bit integers within the [MIN_INT_VAL, MAX_INT_VAL] range. The application that we developed is offering an automated process which takes in consideration all parameter combinations requested by all stream types and generates the test data files for them. In addition, it writes all performed results into a log file. This automated process is faster, minimizes human errors and it is easier to monitor the results. We keep the files generated after the write stream performance testing in order to be used for read stream and merge-sort algorithm testing.

The size of data that we took in consideration is between 1000 integers - 250 Millions integers, which corresponds to file sizes of 4KB-1GB. We consider this amount of data reasonable to fit in real-world applications and at the same time appropriate to find the best stream type to be used or the ideal parameter combination.

Table 1: System Specifications

2. Theoretic background

This section shall introduce the main concepts of memory mapping and Multiway-Merge-Sort Algorithms, which are used in the preceding sections.

2.1. Memory mapping

Memory mapping allows applications to access disks in the same way the application accesses dynamic memory using pointers.

A Memory Mapped File is a segment of virtual memory which has a correlation with the files or a part of a file in secondary memory. Memory mapping is usually chosen if the file is too large and wants to reduce the I/O data movement. Generally, there are two types of memory mapped files:

- **Persisted:** In this type, the data is stored in a source file on disk after all actions that were performed on the file have finished.
- **Non-Persistent:** The data is flushed to disk when all processes are finished. This type is commonly used for creating shared memory in inter process communications.

In order to work with a memory mapped file, one has to create a view on a part or on the entire file. Multiple views are generally necessary when the file is larger than the available logical memory space (2GB for a 32-bit computer). These files are accessed by the operating system’s memory manager which automatically divides it into pages.

The Operating System uses a virtual memory mechanism in order to work with Memory Mapped Files. The pages of the files are loaded into physical memory when requested. In case a process references to the page that is not available, a page fault is thrown and then the OS loads that page into memory.

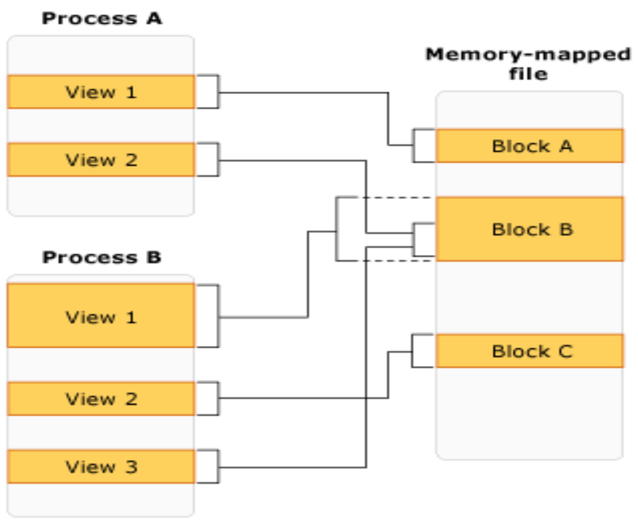


Figure 1: Memory mapped file[2]

A typical example in Windows is the following: every time a .DLL file is loaded into the memory, it is actually loaded as a memory mapped file.[8]

Memory mapped I/O is a method of performing I/O operations between CPU and external devices. These use the same address space to address both memory and IO. The memory of I/O devices are mapped to address values. So, whenever an address is accessed by CPU, it refers to both - memory and the I/O device.

Main Advantages:

- Memory mapped files provide faster access than the normal file access using I/O in case of very large files.
- These can be mapped by more than one processes simultaneously.
- They avoid cost of marshalling and un-marshalling while sharing data among the processes.[9]

Main Disadvantages:

- The application user has no control over how much main memory is assigned at which time as we leave it up to the system to decide when to flush files to disk and when to map which parts. Thus, the exact I/O calculation is difficult to do.
- If the user does not create a proper view, it may lead to unnecessary usage of memory.
- Using memory mapping for small files results to wastage of slack space.
- Only the hardware with memory management unit (MMU) can support memory mapped files.

2.2.Multiway-Merge-Sort Algorithm

Before explaining the algorithm, let us consider some generic terms first:

External memory algorithms:

These algorithms are used to process large data that doesn't fit in main memory at once at the same time. The data is generally stored in secondary devices. Then, chunks of data (instead of the whole file at once) are brought into main memory, are processed and then written back to the secondary memory.

External sorting algorithms:

These algorithms are a special version of external memory algorithms which are used to sort data that doesn't fit into main memory. These generally are of two types: distributed sorting (Quick sort) and merge sorting. The running time of these algorithms is calculated based on the number of passes between the external device or data source and the main memory.

Multiway merge-sort Algorithm:

This is an external sorting algorithm and generally consists of two phases: sort and merge. In the sorting phase, the whole data is divided into lists/chunks that are small enough to fit into the main memory (ideally equal to the available applicable memory space) and are sorted using any in-memory algorithms, like for example quicksort. After sorting they are written back to the disk. This process is repeated until the file is divided in B(R)/M sorted chunks that were written to disk, where B(R) refers to the number of blocks of relation R and M to the available memory as depicted in the figure below.

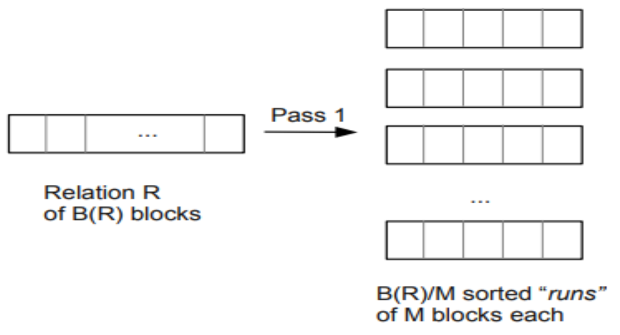


Figure 2: Sorting phase - The data is divided into chunks that fit into main memory

In the merging phase, the data from the sorted lists is read into main memory and the first integers of the respective files are merged using a d-way merge technique, where d is the number of streams that are merged at once. The outcome, a sorted list of data, is written back to disk. With every iteration the length of the sorted file is multiplied by M. This process is repeated until all chunks are merged and only one completely sorted file is left.

To sort a file in multiway merge-sorting with N pages and B buffers, the number of passes to merge all the data will be $\log_{B-1}$ (N/B). We consider only (B-1) buffers because we need one buffer for the output. As we require one pass to read all the data and split it into sorted chunks (pass 0), the total number of passes for multi-way external merge sort is $1+(\log_{B-1}$ (N/B)) (Always consider upper bound). [7]
The total I/O cost are therefore:

$$\text{I/O} = 2 \times \text{Number of pages} \times \text{number of passes}$$
$$= 2 \times N \times [1+\log_{B-1} (N/B)]$$

(We multiply with 2 because, in each pass we both read and write each page)

Advantages:
It is obvious that the less number of disk I/O we need to perform, the faster the algorithm will be. In a 2-way merge sort algorithm, many intermediate files will be created which we need to flush out and read in again. This is significantly increasing the disk I/O operations and thus affecting the time. With k-way merge sort, if the number of files, n is less than k, then there will be 0 intermediate files. And when n>k, we only need to perform merge sort n/k number of times.

Disadvantages:
Though k-way merge sort performs better in terms of I/O operations, it is very complex to implement as it requires the usage of heaps or other similar data structures for efficient time complexity.

3. Stream Observations

In this section we shall discover different read/write stream types in the following order:

- Simple stream type ("Simple")
- Automatically fixed buffered stream ("Buffer"),
- Adjustable buffered stream ("BufferB", with different buffer values B)
- Memory mapped stream ("Mapping")

Goal 1: Distinguish performance as a function of k, N, B for each implementation
For every stream type we shall first give a brief introduction to our implementation of both, the write and the read stream. Then, we analyse and discuss our performance test results for different values of k, N and B for the read and write streams. Our goal here is to distinguish the performance (in terms of execution time) as a function of k, N and B for each stream implementation. In general, the performance testing approach is the same for all write and read stream types. Therefore, we're describing the process only once in detail (refer to 3.3.2.), but then are only showing and discussing graphs in the subsequent sections. Full testing results can be found in the attached tables.

Goal 2: Compare implementations and find the most performant read/write stream
The second goal of the testing is to determine the most performant read and write stream type that will be used for the external multi-way merge-sort algorithm. For the comparison of the streams we are using an iterative approach: After the individual stream discussion to fulfill goal 1, we will compare the respective stream type to the previous one and keep the more performant stream for the next comparison. By this, we're always comparing only two stream types and keep the best. In section 3.6 we'll then conduct a final, summarizing comparison of all implemented types, discuss advantages and disadvantages and give a final recommendation, which one to use for the merge-sort algorithm.

3.1. Testing Application Development

Java source code
We created separate classes for read and write streams to make the code easier to read and conceptually clearer. To have a more generic programming approach we created 2 interfaces "*ReadStream*" and "*WriteStream*" in which the generic methods are declared that will be used in all the stream classes. Based on the project requirements we created the requested methods and added a few extra ones to support our application development. The generic methods declared in the interfaces are explained below. In each of the subsequent sections we will then emphasise the differences for specific streams.

ReadStream interface

- *void OpenFile()* – Method used to open the files for read.
- *int ReadNext* – Method used to return the next integer read from the files.
- *boolean EndOfStream* – Method to check if we reached the end of the file. We try to read the next integer every time we check for the end of the file.
- *boolean IsOpen* – Method used to check if the stream is still open before trying to manipulate it.
- *void Close()* – Method used to close the streams. It is very important to release the used memory.

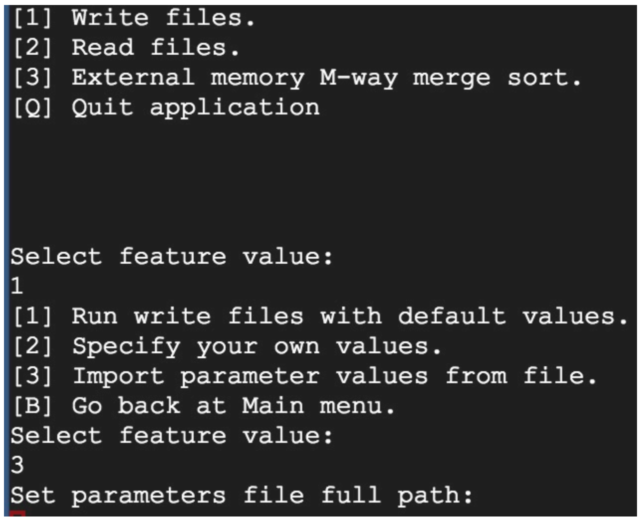
WriteStream interface

- *void CreateFile()* – Method used to create the new files while using write streams.
- *void WriteElement()* - Method used to write an integer to the stream.
- *boolean IsOpen()* – Method used to check if the stream is still open before trying to manipulate it.
- *void Close()* – Method used to close the streams. It is very important to release the used memory.

Application interface

To make the testing as convenient as possible, we developed a command line application interface for the performance testing, which is named DSA.jar in our submission. The following steps describe the process for executing the application in Linux. Other operating systems might have slightly different commands.

To run the testing, navigate to the folder where the DSA.jar file is saved and execute the command *sudo java -jar ./DSA.jar* in the command utility. The application now prompts the four options shown in Figure 3. Enter 1 to open a write stream, 2 for a read stream, 3 for an external multiway merge-sort algorithm or Q to close the application.



```
[1] Write files.
[2] Read files.
[3] External memory M-way merge sort.
[Q] Quit application



Select feature value:
1
[1] Run write files with default values.
[2] Specify your own values.
[3] Import parameter values from file.
[B] Go back at Main menu.
Select feature value:
3
Set parameters file full path:
```

Figure 3: Application me

In a next step the application asks for the input parameters. For this, either the default parameter setting [1], individual parameters [2] or parameters from a file [3] can be selected. Entering B navigates back to the previous menu. According to the choice either the parameters itself together with an integer for the number of execution times (equals to a loop in the source code) or a file path needs to be entered. In principle it is possible to enter several values for N, separated by a comma. However, it should be taken into account that the total size of the created files can quickly become very large, so large N should be handled with care to avoid exceeding the available disk space. A sample structure for the parameter import from a csv file is the following:

```
Stream Type (only 1 per row); Total integers per file (Ex: 1000, 10000, 100000...);Nr of streams; Buffer size range (Ex: 1,6);Nr Loops
Buffer;1000,10000,100000,1000000;1;0,2;6
Buffer;1000,10000,100000,1000000;2;0,2;6
```

This parameter setting will be executed six times as a buffered stream for each N=1000 (and 10000,100000, 1000000 respectively) with k=1 in the first statement followed by k=2 in the second one. All possible value ranges for the stream type, N, k and B can be found in the application menu by selecting *[2] – Specify your own values*. For the stream types *Simple* and *Buffer*, where an individual buffer setting is not applicable, the buffer value will be ignored from the application. The available stream types are the four listed ones in the beginning of this section: *Simple*, *Buffer*, *BufferB* and *Mapping*. The application is case sensitive. In case of typos, *Simple* will always be used as a default.
For *BufferB* and *Mapping* a buffer range has to be set through which the application is looping during the execution. The values correspond to the buffer size in the power of two, e.g. 0,2 means that the statement is executed for a Buffer of size 2^0, 2^1 and 2^2. If own values are inserted, please enter the values exactly as requested by the application and be reminded that the stream types are again case sensitive.

After that, press enter to start the application. If a write stream was chosen, a new folder hierarchy named *FilesFolder* will be created on disk, where all files are written to and ordered by k and N. For each operation selected in the first step (Figure 3) the application creates a csv file containing the execution details and the execution time in milliseconds (ms) and nanoseconds (ns), which can then be opened with excel to visualize the data and calculate the average execution time. At this point, all values of the file should be carefully checked to detect possible typos that have led to other than the desired result.

Before testing a read stream, the desired file should be created by running the respective write stream before, which is creating the file in the right folder/path structure. This was practicable for us, as we always wrote certain file sizes to test the write stream and could use the outcome of this testing (the written files in the right sizes) for the read stream afterwards, thus, minimizing the risk of errors in copying file paths or selecting the wrong file and also make all our results easily replicable. However, if other files than the ones created by the write stream shall be used, *[2] - Specify your own values* needs to be selected in the read stream menu, where an individual path can be specified.

For the Sort-Merge Algorithm, the application expects the following parameter structure if using a csv file, following the same general concepts stated above. Output file specifies the file name of the output file, the sort order should be set to the default value ("true") except another sort order is explicitly requested.

Read Stream Type(only 1 per row); Write Stream Type(only 1 per row); Nr of streams; Nr Integers in Memory (Ex: 1000, 10000, 100000...); Buffer size range (Ex: 1,6);Nr Loops;Input file (full path); Output file; true (Ascending sort order)

```
Mapping;Mapping;10;1000;20,20;5;/home/marie/DSA/FilesFolder/1/10000/10000_0df62a8b-afc9-490d-8983-633716040dc8.dat;Test;true
```

3.2.Data input/output stream

This implementation deals with read/write streams without any buffer.

3.2.1. Goals and expectations

To reach the previously outlined goals, we are varying k and N for this implementation. For k we're testing certain values between 1 and 30 streams as requested in the project description. As we're interested in the trend of the performance, we won't test all values between 1 and 30 and limit our testing to 8 values as shown in Table 2 as we expect this to be sufficient to get familiar with the stream behaviour. For N, we experiment with values ranging from 1.000 – 250.000.000 integers (4kb -1GB in size) to capture a good overview of the stream for small and very large values.
We expect this stream types to perform very poor with increasing file size as we expect the following cost function for I/O's:

$$\text{Total I/O} = k * N$$

With k number of streams, N number of integers

From this function we can see that the I/O's are linearly increasing with increasing file size (N). This is due to the fact that this stream implementation requires one I/O for every integer that is read or written as it is not making use of any buffers to optimize performance. Like for N, we also expect I/O's to increase linearly with the number of streams k as opening new streams requires more integers to be read and written, one after another. This overall leads to the assumption that this stream will have by far the poorest performance of all tested stream implementations.

3.2.2. Implementation

We override the methods of the interface "*ReadStream*" and define what these will do. The *Simple* read stream is using java *DataInputStream* to open files for reading. We also override the methods of the interface "*WriteStream*" and define what these will do. The *Simple* write stream is using java *DataOutputStream* to open files for writing. It needs to be emphasized again for this implementation that the integers are read one by one from the file, meaning that every for read we make an I/O to the disk.

3.2.3. Write stream

In order to test the performance of the stream, the application was executed 6 times for each k and N (in number of integers) shown in Table 2. This can be done by selecting [1] for *write stream* and [3] for *read from file* in our application, where the different k, N and the number of executions (here 6) are specified in a csv file.

Table 2 shows the average time in ms of the 6 executions. For very large files (100M and 250M integers), we only tested the performance for one stream due to the exorbitant execution time. This summarizing table was created for each of our stream implementations in our performance tests. However, as mentioned previously in the succeeding sections we won't show all detailed results anymore but rather focus on the important trends and thus only highlight certain parts of the results.

Throughout the report we are always discussing time to evaluate and compare the performance of the streams. We decided to do so as the captured time gives a good indication of the number of I/O's that were performed in the background. Disk access is several orders of magnitude slower than memory access (even when using a SSD), so high execution times, generally speaking, indicate a higher number of I/O's and the speed of an application is an important measure in case the application shall be brought into production. However, for buffered streams we need to discuss I/O's in addition to the execution time as a decrease in time does not lead to an increase in I/O's in every case for those streams due to buffer, cache and block sizes of the operating system (OS). Nevertheless, for the *Simple* stream implementation we consider a discussion of the time as sufficient.

Average writing time in ms							
k / N	**1.000**	**10.000**	**100.000**	**1.000.000**	**10.000.000**	**100.000.000**	**250.000.000**
1	4	43	401	3939	39805	472981	1113216
2	8	80	790	7991	80613	-	-
3	11	119	1185	11877	122739	-	-
4	16	158	1583	15944	163679	-	-
5	20	198	1997	20473	201869	-	-
10	40	396	3967	40724	406810	-	-
20	85	796	8176	81613	861798	-	-
30	120	1197	12103	122307	1242396	-	-

Table 2: Average writing time in ms for different k and N

In order to only measure the impact of one parameter on the performance, we usually divide the sections into subsections, each discussing the influence of one parameter, before we draw final conclusions or compare the different stream results.

Varying N for k=1 fixed

Figure 4 shows the average writing time in ms for different N and k=1. From a first sight we can already see that with increasing N the writing time is also increasing in a proportional manner (refer to Table 2). But we need to be careful with the trend of the chart, which should not be understood to be exponential, as the width between the tested values for N is not equal in every increase. To show the linear relationship between cost and N, we need to plot the data differently.

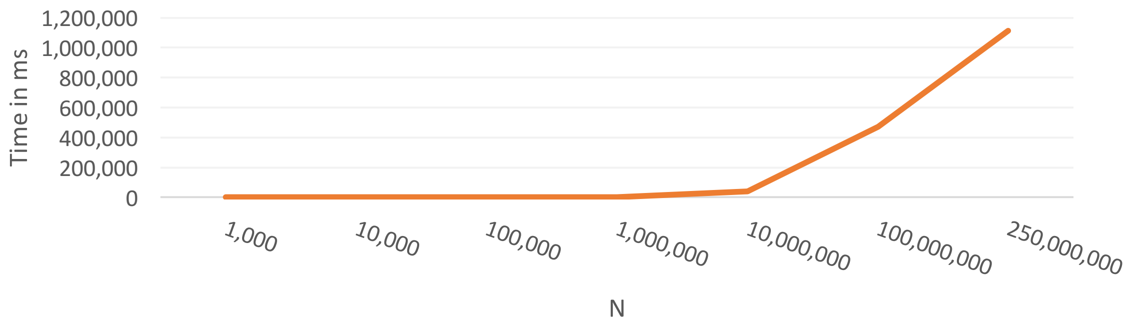


Figure 4: Average writing time in ms for different N

Figure 5 shows the same graph, but in time per integer. In order to highlight the linear scale, we divided the written number of integers of the input file by the execution time to get a comparable measure for performance across the different N and avoid unreadable charts due to large differences in time. It can clearly be seen that the integers per time ratio remains constant with raising N, thus, increasing N will result in a proportional increase of time. The small differences might be due to other processes, but as they're in the scale of nanoseconds, we are considering them as noise.

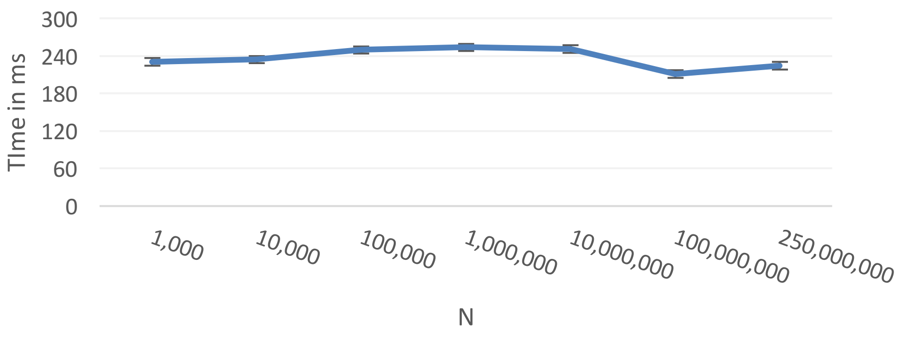


Figure 5: Time per integer for k=1 across all tested N

Varying k for N=1.000.000 fixed

In a second step we keep N fixed and vary k. We chose 1.000.000 for our example, but the discussed results hold for all other tested N as well. Plotting different k for only one specific N shows that this behaviour is also following a linear trend, which is shown in Figure 6. We can derive that with an increasing number of streams, the writing time is also increasing.

The linear trend line in Figure 6 has a R^2 value of 0.999, which shows an almost perfect fit of the regression (values of R^2 are between 0 and 1 with 1 as an optimum). Whenever we use regressions in our report, we usually also provide a function to calculate the execution time in addition to the I/O calculations that can be made based on the formula provided in the beginning of each chapter.

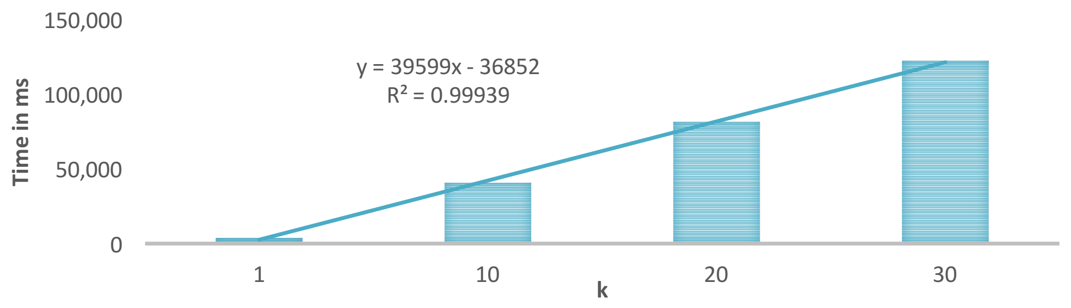


Figure 6: Linear growth for N = 1.000.000 with increasing k

Varying k and N

Another interesting, though expected, behaviour is shown in Figure 7, where three stream sizes (for k = 10, 20, 30) are shown for different N. By this we can see that the trend is the same across all streams, but the more streams are open, the longer is the writing time. This seems to be obvious as for several parallel streams, the integers have to be written for each stream one by one. Considering this together with the linear trend in figure 4 we can also conclude that for k>1 the performance proportionally decreases with higher k, which proves our expectations for this implementation to be correct.

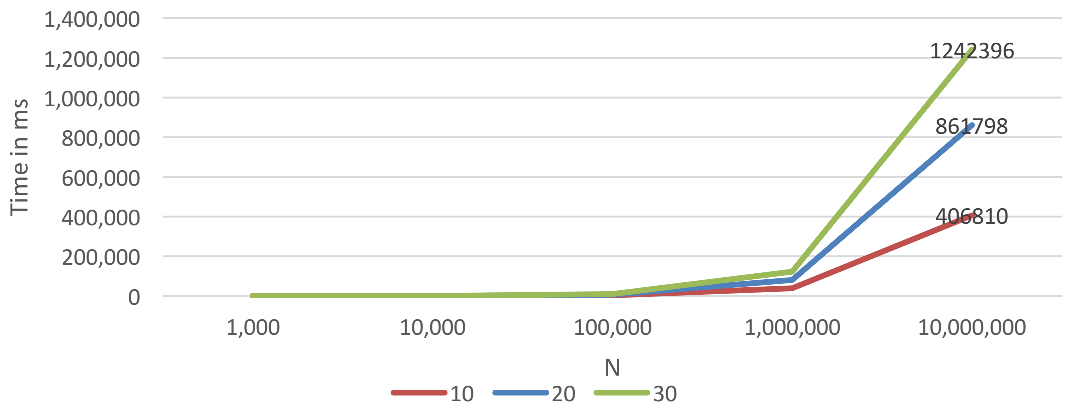


Figure 7: Average writing time in ms for k=10,20,30 and varying N

This behaviour is overall not surprising as for a stream without a buffer the disk needs to be accessed for every performed write, resulting in a high number of I/O's and thus high cost/low performance. Especially for large numbers of N we can see exorbitant execution times, which are not practicable anymore in a production environment.

Thus, we can conclude that our expectations of a proportional relation between execution time and k (N) were met but we don't consider this stream type to be suitable for the use in the merge-sort algorithm. Therefore, we assume that we'd need buffered streams for a better performance, which shall be examined in section 3.3.

3.2.4. Read stream

Read stream were tested in the same way as the write streams. The application was executed 6 times. All detailed results can be found in the attached tables.

Varying N for k=1 fixed

Figure 8 shows the average reading time in ms for different N and k=1. We can again see that with increasing N, the reading time is also increasing in a proportional manner. As mentioned above in write stream, we also need to be careful with the trend of the chart, which should not be understood to be exponential, but rather linear.

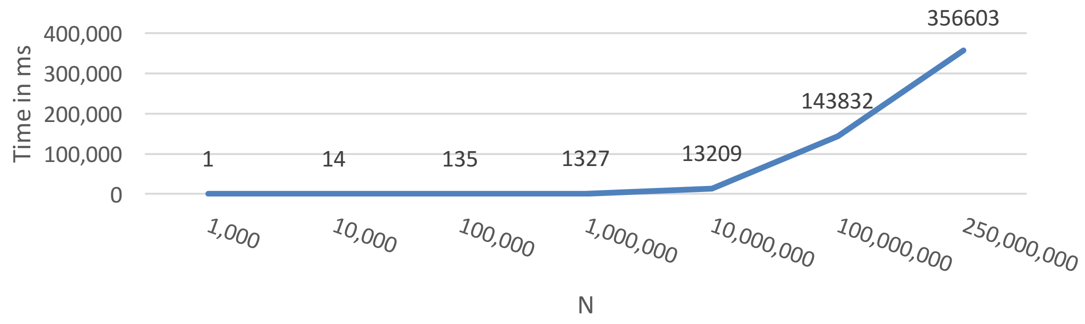


Figure 8: Average reading time in ms for different N

Figure 9 shows the same graph, but in time per integer. It can clearly be seen that the time per integer increases with raising N, thus, increasing N will result in a proportional increase of cost.

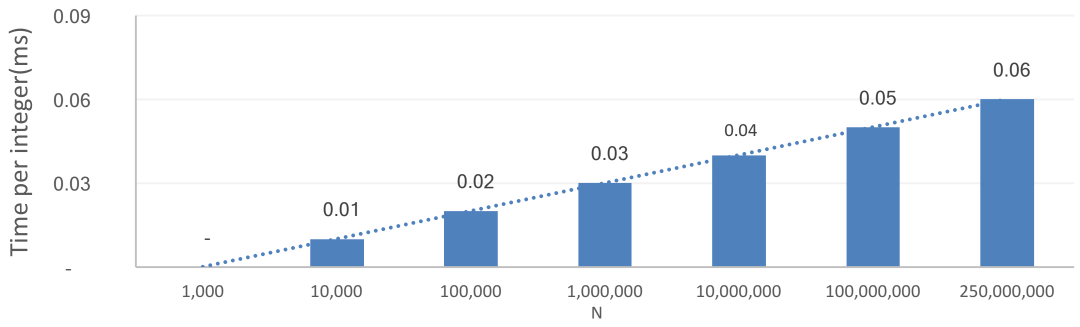


Figure 9: Time per integer for k=1 across all tested N

Varying k for N=1.000.000 fixed

In a second step we keep N fixed and vary k. We chose N=1.000.000 for our example. Similar to the above findings, we can say that as the value of k increases, the reading time as increases proportionally.

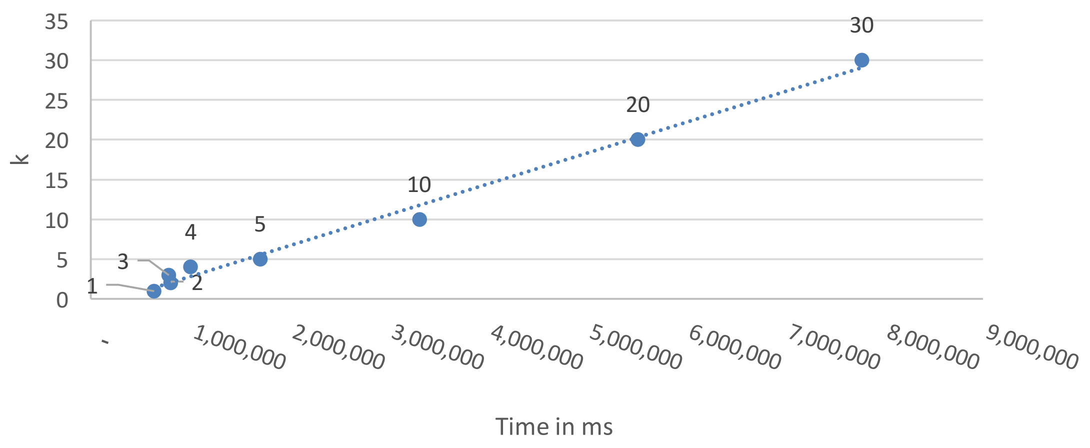
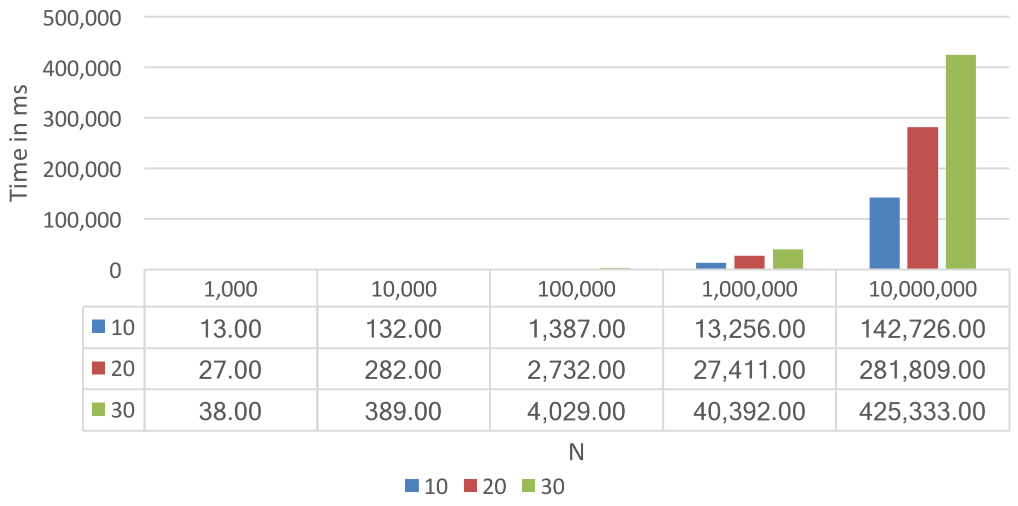


Figure 10: Average reading time for N=1000000

Varying k and N

Now let us consider varying values of N and k=10, 20, 30 and observe the reading time. In Figure 11 we can observe that the reading time is increasing with the increase in both k and N values. The time for k=30 is higher than k=20, which itself is higher than k=10 for all values of N. In the legend we can nicely see the almost perfect linear behaviour due to the same reasons explained in the previous sub-section.



	1,000	10,000	100,000	1,000,000	10,000,000
🟦 10	13.00	132.00	1,387.00	13,256.00	142,726.00
🟥 20	27.00	282.00	2,732.00	27,411.00	281,809.00
🟩 30	38.00	389.00	4,029.00	40,392.00	425,333.00

Figure 11: Average reading time for different k and N

In Figure 12 we can observe again exorbitant values for the reading time for N larger than 1.00.000 integers similar to what we saw for the write stream already.

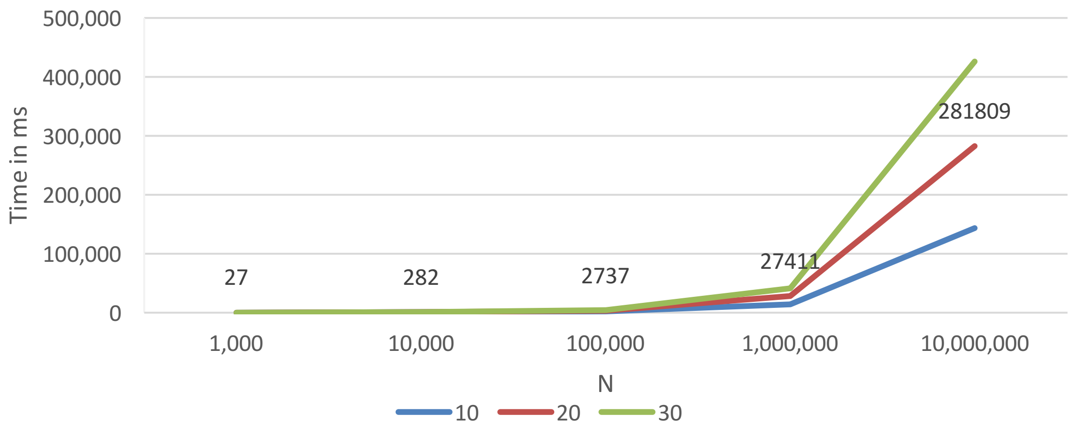


Figure 12: Average reading time for different k and N

Conclusion

We saw that the read stream results follow the same behaviour than the write streams, due to the same implementation logic used. Still, this is surely not a good choice for implementation in the external merge sorting as the increase in the size of stream and the number of streams is affecting the reading time very badly.

3.3. Buffered data input/output stream with fread/fwrite

This implementation deals with read/write streams with a fixed automatically assigned buffer size, which by default is 8.192 bytes in the implementation method we used in our application.

3.3.1. Goals and expectations

For this implementation we are again varying k and N with the same value ranges as depicted in 3.2.

We expect this stream types to perform much better than the previous implementation as we're now using a buffer and thus decreasing the number of I/O's. By using the buffer, the stream does not force an I/O for every integer that needs to be read/written, but instead loads several integers to memory at once, depending on the disk block size and the buffer assigned.
However, as the buffer size is restricted to the default size, it might not be the optimal choice for all file sizes and disk types. Despite the buffer benefit, we still expect relatively high cost for large N as the buffer size is still significantly lower than N and thus still requires many I/O's. As k requires the application to write the integers for each stream in parallel we expect k to increase cost similar to the previous implementation. The default buffer size is 8.192 bytes which is bigger than the physical block size of 4.092 bytes, so we don't get any more benefits in terms of I/O because we will already fill all the physical disk space.

Anyhow, for buffers bigger than the block size we still benefit in terms of total time because of the Operating System disk access strategies in saving the same file in consecutive disk blocks and the Cache that is used by the Operating System. I/O Cost formula:

$$\text{Total I/o} = \text{RoundUP}(k * (N*4/4092))$$

With k number of streams, N number of integers

3.3.2. Implementation

The read Buffer implementation is the same as the simple read stream with only 1 difference: we use *BufferedInputStream* to read integers from the file. The *BufferedInputStream* is using a default buffer size of 8.192 bytes defined by the java jdk. It means that this stream will read as many integers as fit in the default buffer, so we need much less I/O to read the file.
The write Buffer implementation is the same as the simple write stream with again only 1 difference: we use *BufferedOutputStream* to write integers into the file. The *BufferedOutputStream* is using a default buffer size of 8192 bytes defined by the java jdk.

3.3.3. Write stream

The testing and analysis procedure follows the same principle as before.

Varying N for k=1 fixed
At first, the performance for k=1 fixed and increasing N looks similar to the first stream implementation. This is due to the fact that even though we're now using a buffer, an increasing file size N requires more I/O's and thus costs more time.

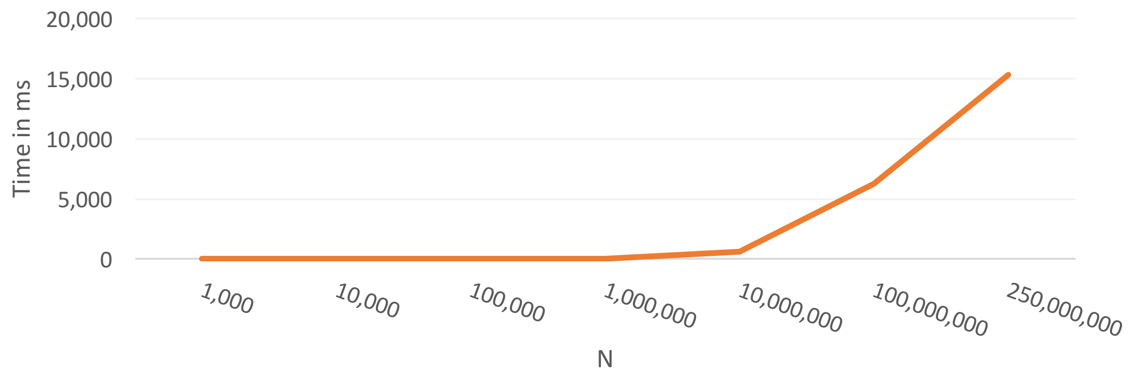


Figure 13: Average writing time in ms for different N

However, we do not see a clearly proportional dependence between N and the time anymore. The figure below shows the performance in number of written integers per time for different N with fixed k.

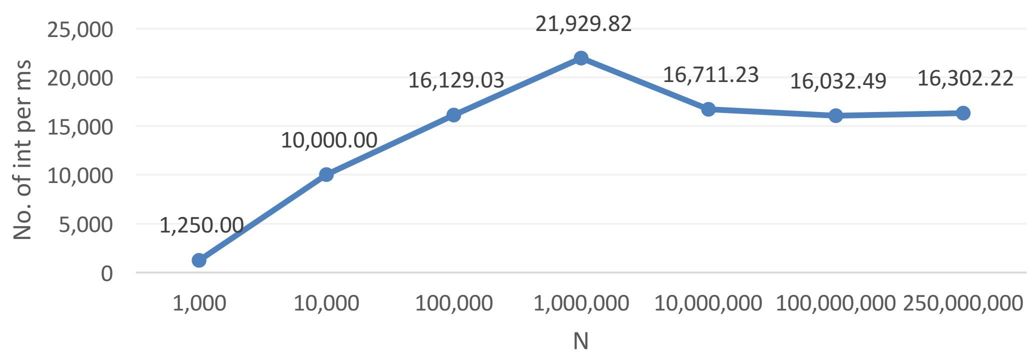


Figure 14: No. of integers written per ms for different N

Instead of a constant time across all N the graph shows significant differences for varying N. The performance is increasing until N = 1.000.000 and then slightly decreasing and plateauing for higher values. Hence, we note that a buffered stream works more efficient for larger N. We also note that there exists a performance maximum, which in this case is reached for N = 1.000.000. When we first saw this behaviour, we were surprised and repeated the testing and checked our implementation. However, we still got the same results. We therefore assume that the following is happening: If we calculate the file size for 1.000.000 integers we get 1.000.000 x 4bytes /1024/ 1024 = 3.8MB. If we query our VM to check the cache size of the disk used, we get a cache of 4MB. Thus it might be the case that the operating system is using cache to speed up the process. We therefore at first see a clear increase in performance for increasing N. However, upon the point where the cache is full, the OS is incrementally pushing parts of the cached file into pages to allow new parts of the file to enter the cache. From this point on, we won't see any further performance increase as the cache is full. When we were running the tests, we were not clearing the cache after each test repetition. If we add the file sizes of the first three tested N to the size of N= 3.8MB from above, we end up at around 4MB at the point where we see a performance maximum in our chart. Afterwards we a plateau for higher N due to the process described above.

Varying k for fixed and varying values of N

The chart below shows longer execution times for increasing values of k. As the relative time difference between the no. of streams is immense, only values for k=1 (red line) and k=2 (green line) are shown, but the trend holds for all higher values as well.

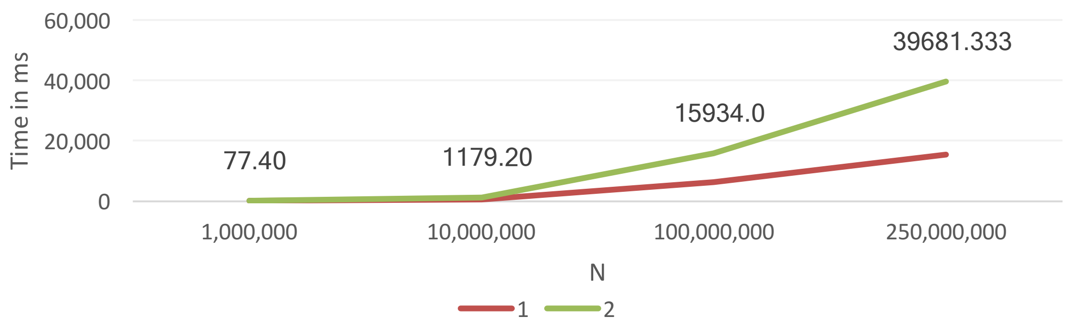


Figure 15: k=1,2 for different N

In contrast to the simple stream implementation, this increase is not linear due to the buffer and cache techniques used by the OS. If we use a regression to determine the cost function, a polynomial function with order 2 actually showed the best fit with very high R^2 values for all tested N between 100.000 and 10.000.000 in our case. However, this regression might vary widely for other system specifications. The equations for two testing examples are displayed in the subsequent figure.

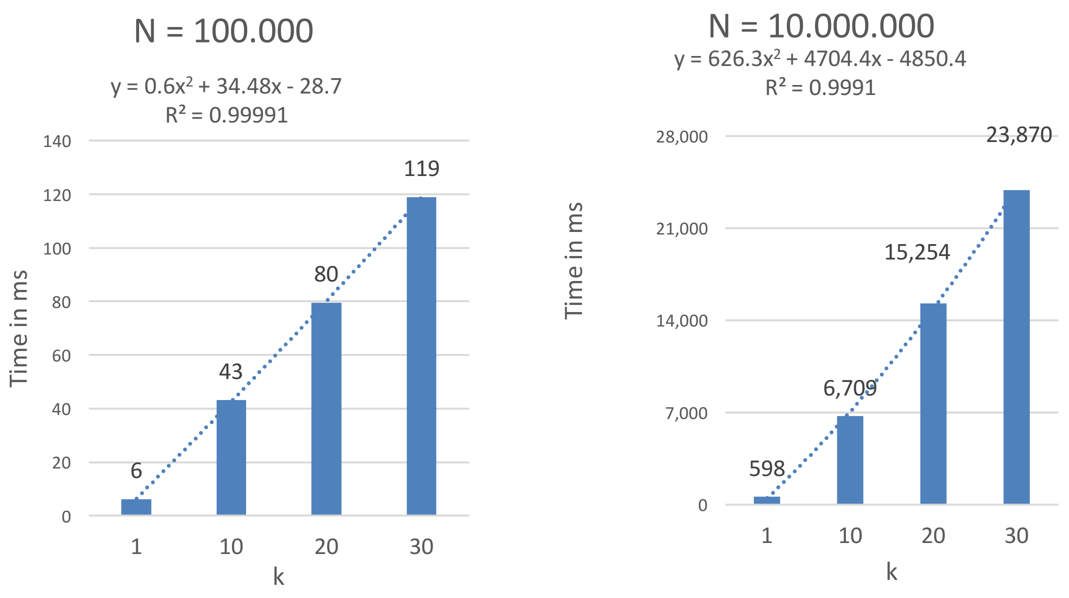


Figure 16: k in function of time for fixed N

Buffered write vs. simple write stream

A direct comparison of the simple and buffered stream implementation is shown in the graph below:

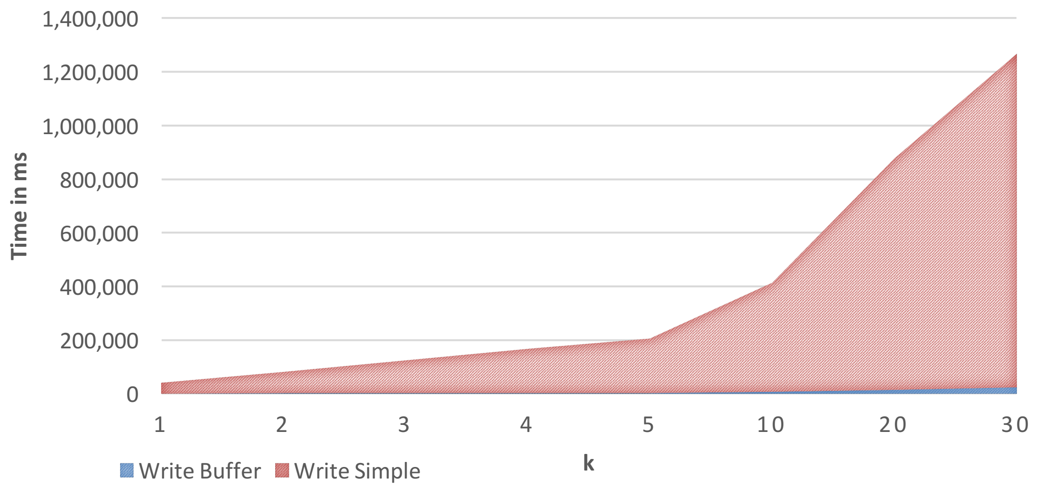


Figure 17: Performance comparison simple and buffered write stream implementation

Very clearly, the buffered stream outperforms the simple one. Even though the chart only shows the performance subject to k, the same tremendous difference holds for varying N, too. As the difference is around two orders of magnitude, we will not consider the simple stream implementation in further discussion anymore.

As a final conclusion for this stream implementation we can say that our expectations were met as it overall outperforms the simple implementation with a performance maximum for N = 1.000.000 and a low number of streams (k). Despite this, we also note that we saw some stream behaviours which can't only be explained by the buffer, but rather suggests that we additionally see the influence of caching in our results.

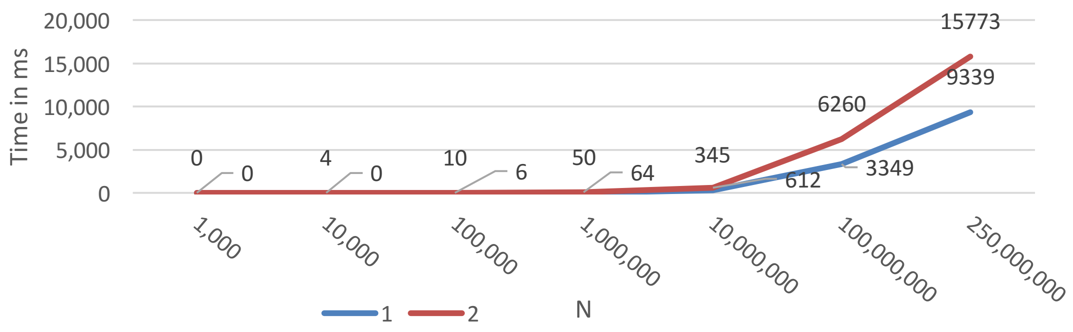
All in all, as the buffer is still fixed to a default value, we'll test in a next step if a more optimal buffer size can be found for other N where the cost were still very high.

3.3.4. Read stream

For the read stream, we followed the same testing approach and found similar behaviour as for the write stream.

Varying N and k

The graph in the chart is very similar to the graph observed in the simple stream with a major difference in the values for the reading time, which are significantly lower than before. The overall trends are similar to the ones explained in chapter 3.3.3 and thus won't be explained in detail again.



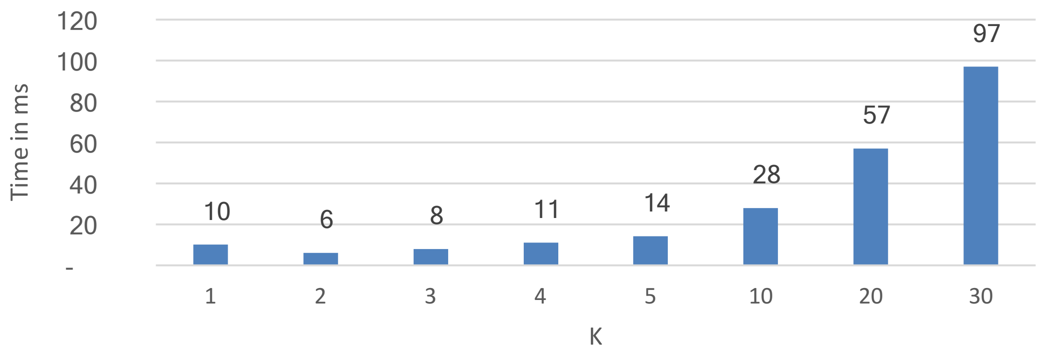


Figure 18: k in function of time for different N (upper) and fixed N=100.000 (lower)

Buffered read vs. simple read stream
For comparing these two streams let us consider a fixed k=10 and varying N.

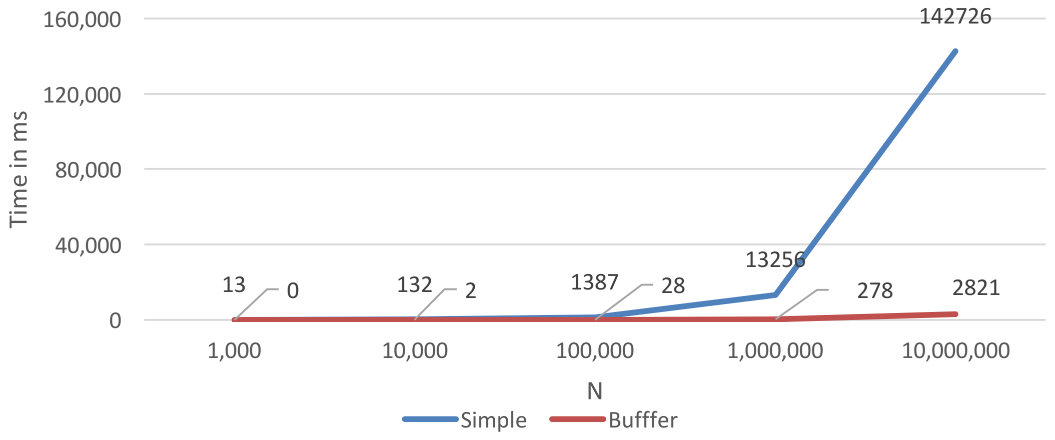


Figure 19: Comparison of Simple and Buffer streams

In Figure 19 we can again clearly see the tremendous difference between implementing *Buffer* and *Simple* streams. Thus, it is clear that we prefer *Buffer* over *Simple* streams for reading the data, especially for large N or k.

3.4. Data input/output stream with fixed buffer in internal memory

This implementation deals with read/write streams with a fixed buffer size which can be set by the user.

3.4.1. Goals and expectations

For this implementation we are varying B and also take into account different N. We will not test different k as this was not requested in the assignment for stream type (3) and (4) implementations. All tests were performed for k=1. For N we are using the same value ranges as depicted in 3.2.

For B we analyze buffer sizes between 2^0 and 2^{30} to be able to compare it with all four stream types in the end. A very small buffer size will show similar behaviour to an unbuffered stream, whereas a very large buffer can become interesting if very big files need to be handled and memory space is not too restricted and huge values for B were requested. We decided to test buffer sizes to the power of two as this is a common implementation practice and simplifies comparisons with streams that have a default buffer included (like *fwrite/fread*). This design decision was mainly driven from an application in practice point of view. I/O's are carried out in block sizes, which are always assigned to values to the power of two. If we now would use buffer sizes in other sizes than to the power of two we would risk avoidable additional I/O's: Using a buffer of 4.100 bytes for example would always lead to two I/O's if the block size is the common value of 4.096 bytes even though it is only slightly higher, which would be a waste of expensive I/O's from a cost/buffer size perspective.

For buffer sizes larger than 2^{30} an “OutOfMemoryError: Requested array size exceeds VM limit” error was thrown as we exceeded available memory, thus we set the highest value to be tested to 2^{30}.

We expect this stream performance to be dependent of the buffer size chosen. We assume that for very small buffer sizes we’ll see a very poor performance which will become better for larger buffer sizes as small buffers will have almost no advantage over the simple stream implementation, because the buffer advantage only comes into play for sufficient large buffer sizes. Furthermore, we expect that from the point where the whole file fits in the buffer, we won’t see any further performance increase for larger buffer sizes as we reached a minimum number of I/O’s.

Therefore, to achieve previously stated goal 1, we want to determine what the optimal buffer size leading to the best performance is for each N. With respect to goal 2, we analyse if this type outperforms the previous one and if so, try to find the buffer size, upon which this stream implementation outperforms the *Buffer* type. The cost function we expect for this stream implementation is the following:

If the buffer size is <= 4092

$$\text{Total I/O} = \text{RoundUP}\ (k * (N*4/B))$$

If the buffer size is > 4092, we don't get any more benefits in terms of I/O because we will already fill all the physical disk space. Anyway, for buffers bigger than the block size we still benefit in terms of total time because of the Operating System disk access strategies in saving the same file in consecutive disk blocks and the Cache that is used by the Operating System.

For other buffer sizes:

$$\text{Total I/o} = \text{RoundUP}\ (k * (N*4/4092))$$

4092 is the disk block size (We don't take in consideration the block header to prevent complex calculations), B is the buffer size, k the number of streams, and N the number of integers.

3.4.2. Implementation

The read *BufferB* implementation is the same as the *Buffer* read implementation with the only difference that here we define the buffer size instead of using the java default buffer size.
At the same time the write *BufferB* implementation is same as *Buffer* write implementation with the only difference that here we define the buffer size instead of using the java default buffer size.

3.4.3. Write stream

Before we start to search for the optimal buffer value, we shall have a look at the overall performance trend in function of the buffer size, shown in the graph below, where we can see the execution times across all tested buffer sizes for k=1.
As expected, for low buffer values we see high execution times that decrease with increasing buffer size following the exponential trend of the buffer size and note that the execution time for a buffer of 1 is around the same order of magnitude as in the simple stream implementation.
This behaviour seems to be logical as the larger the buffer, the more of a file fits in (up to the whole file) and the less I/O's need to be done. Around 256 bytes the performance increase per buffer size increase becomes very low due to the exponential nature of the function and seems to reach a minimum performance plateau around 8.192 bytes, upon which no further performance increase can be achieved by increasing the buffer size. This is important to know when implementing application as with this information the performance can be significantly tuned without wasting unnecessary memory space.

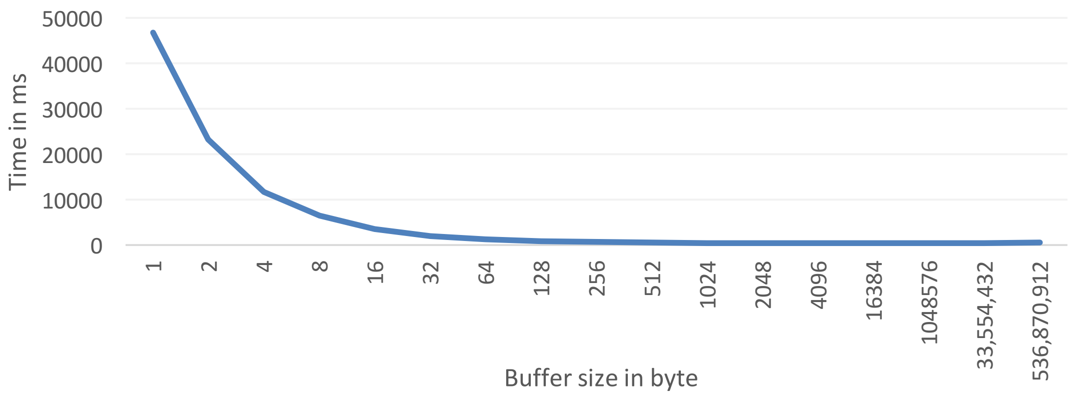


Figure 20:Writing time in function of buffer size for N=10.000.000

However, to determine the optimal buffer value, we need to zoom in into the performance plateau to distinguish if we can find a local minimum for a certain buffer size. When we were analyzing this, we found that we actually had significant fluctuations for buffer sizes $>2^8$. In a first run from 2^{10} on we were only testing values for 2^{12}, 2^{15}, 2^{20}, 2^{25} and 2^{30}. However, to avoid missing the local minimum, we additionally tested all powers of two between 2^{10} and 2^{20} 3 times each for N up to 250.000.000 as the minimum seemed to be within that range. The optimal buffer value varies for different N. Therefore, we decided to representatively plot the buffer sizes for one N and highlight the discovered minimum values for the remaining values of N in the subsequent table. We also excluded all buffer sizes below 16 bytes from the graph as those values are very large compared to the others and made a readable plotting impossible.

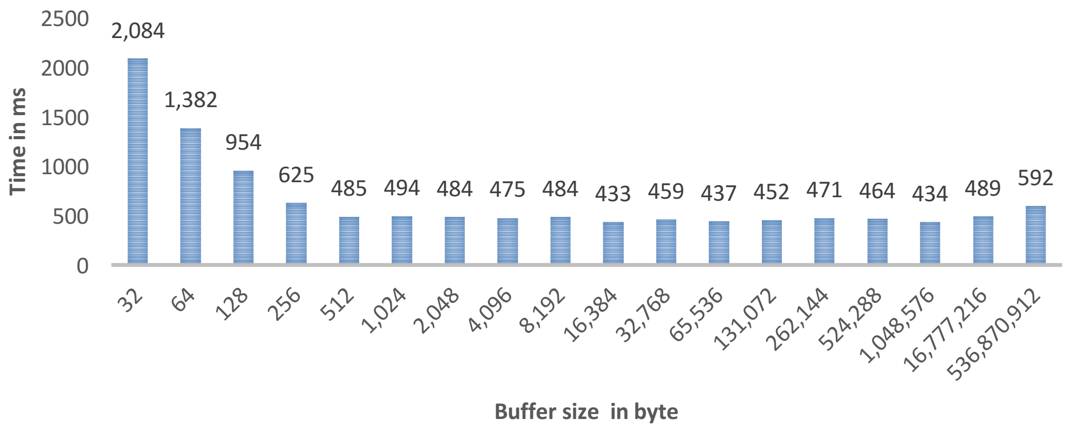


Figure 21: Time for different buffer sizes for N=10.000.000 and k=1

N	**Optimal Buffer Size [bytes]**
100.000	2^{15}
1.000.000	2^{15}
10.000.000	2^{14}
100.000.000	2^{9}
250.000.000	2^{9}

Table 3: Optimal buffer sizes for varying N

From the chart and the table, we can see, that the optimal buffer size varies between 16,384 and 32,768 bytes, so 2-3x the default buffer size (8.192) in the previously seen *Buffer* stream implementation and can explain the again lower execution times as we'll see in the comparison. Surprisingly, for very large values we can find lower optimal buffer values. However, we can see in the graphs that from buffer values higher than 512 bytes the performance increase is really marginal and the execution times are often the same or only differ in nanoseconds for the same N and several buffer sizes upon this point. If the difference is too small we need to be careful with the result interpretation as other processes or caching as explained previously might have had influence on the results as well and we decided to always take the smallest buffer size if the values were the same, so the decreasing trend might be misleading.

Comparison of Buffer and BufferB stream implementation

After determining the optimal buffer sizes for different N, we now want to distinguish if this implementation is outperforming the previous one and if yes, upon which buffer size. Therefore, we plot for both implementations the execution time over different buffer sizes.

The plot shows that *Buffer* is constant as we only use one default buffer size whereas *BufferB* is outperforming *Buffer* from a buffer size of 256 bytes onwards for N=10.000.000. For smaller N, we found slightly lower buffer sizes from which on

BufferB performs better (128bytes), whereas for N>100.000.000 the point was slightly higher (512 bytes).
As these results can be affected by other processes, we take the highest value and conclude that for buffer sizes B>=512 bytes, *BufferB* for sure outperforms *Buffer* in terms of time, which holds for all tested N. For lower buffer sizes, this only holds for low N respectively as shown in the graph below. This is surprising as we would have expected that the lines cross at the buffer size of *Buffer.* This is obviously not the case and we see this for all tested N, so we conclude that the default buffer size seems not to be optimized from a time perspective. So, if an application shall be optimized in terms of time, it might make sense to perform some tests for different buffer sizes upfront to gain some additional performance boost for a comparably low implementation complexity and time investment.

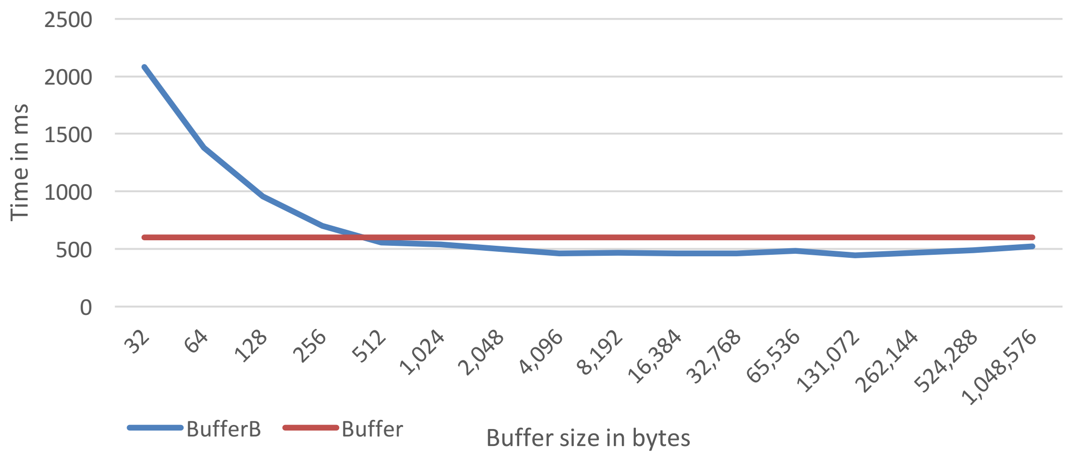


Figure 22: Comparison between BufferB and Buffer write stream for N=10.000.000

Nevertheless, when we were discussing these results we were marvelling, why the default buffer size was set to 8.192 bytes, if further performance increase was possible in all our tested cases. We went back to our I/O formula and also checked our system specifications once again, where we made the following discovery: The block size of disk that we are using is 4.096 bytes. This means that no matter how large the buffer is, the I/O's, so the speak the number of disk accesses, is always bound to the block size of the disk. If we for example use a buffer of 8.192 bytes, we still need to do 2 I/O's for a file size of e.g. 5.000 bytes as the maximal possible number of bytes that can be loaded at once is the size of a block, so in our case 4.096 bytes. This means that upon a buffer size of 4.096 bytes we don't get a performance increase with respect of I/O's anymore.

Nevertheless, why is the default size then not set to 4.096 bytes and why do we still a performance increase in time above this buffer size? With respect to the first part of the question, the Java documentation provides some insights. It is said that the default buffer value fits most of the modern disks. If we search for typical disk block sizes we find that most of them have block sizes between 4.096 and 8.192 bytes, with exceptions for high-end SSDs that can go much higher. Now it is clear, where the 8.192 bytes come from. With respect to the second part of the question we assume that this slight increase is due to caching or other memory handling decisions of the OS and might also be influences by other processes.

Conclusion
As a final conclusion we can say that our expectations were met: For small buffers we saw a very poor performance, but from B>=512 bytes, this implementation outperformed the previous one for all N. We also saw that a performance maximum plateau is reached, upon which further increase of the buffer size does not significantly increase the performance anymore. Here the trade-off between performance optimization and memory occupation comes into play: The more buffer we allow, the more memory we occupy. As memory is expensive, performance tests should always be conducted in order to define the optimum buffer for the respective application and to balance memory occupation and performance tuning. We also saw that with increasing buffer size we can optimize the time up to a specific point but the minimum number of I/O's is determined by the block size of the system, which limits the performance gain from further buffer increases. Thus, we will only consider the *BufferB* implementation for the next comparison.

3.4.4. Read stream

Read streams were tested in the same manner as the write streams.

Varying buffer size and fixed N=10.000.000

According to Figure 23 we see as expected that with increasing buffer the reading time decreases in an exponential manner with the exponential buffer increase. At a particular point the curve starts to be constant (2^{11} bytes) which means no further performance increase can be reached.

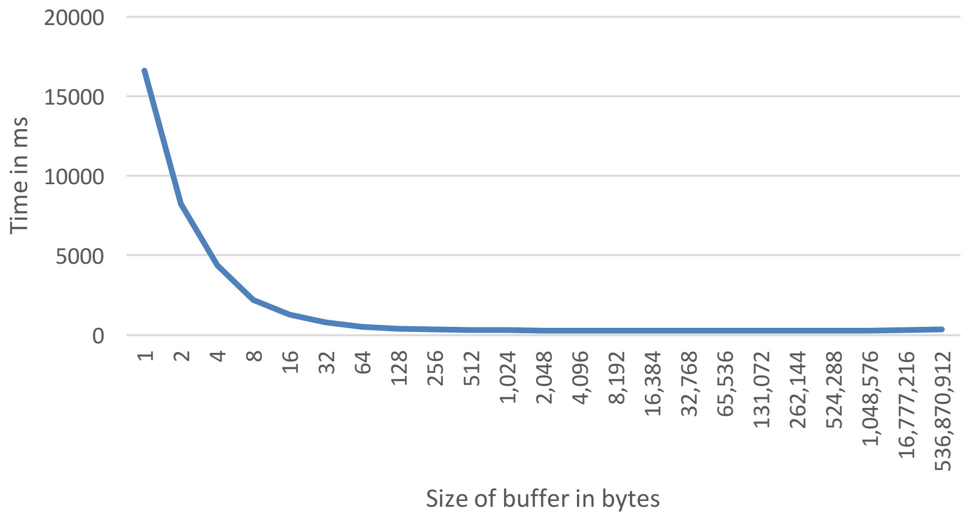


Figure 23: Average time for N=10.000.000

For every N value we found different optimal buffer sizes as shown in the table below. We note that those buffer sizes are overall lower a little lower than the buffer sizes we found for the write stream, which might again be influenced by caching.

Hence, the optimal buffer sizes for the write streams will determine the buffer size chosen for the sort-merge algorithm as we should set this to the highest optimal buffer value found for the tested N.

N	Optimal Buffer Size [bytes]
100.000	2^{12}
1.000.000	2^{14}
10.000.000	2^{17}
100.000.000	2^{17}
250.000.000	2^{27}

Table 4: Optimal buffer sizes for varying N

Comparison of Buffer and BufferB stream implementation

In order to decide which stream is more performant upon which buffer size we exemplarily considered the testing for k=1 and N = 10.000.000 and compared the behaviour of reading times for different buffer sizes. The discussion results hold for all other N as well.
From Figure 24 we can see that the fixed default buffer for *Buffer* as a constant line whereas the cost for *BufferB* decrease with every increase in buffer size and finally start to outperform *Buffer* at 2^8 bytes (clearly visible in the data table of the chart). Hence, we can consider 2^8 as the optimal buffer size in terms of the best memory space/cost ratio.

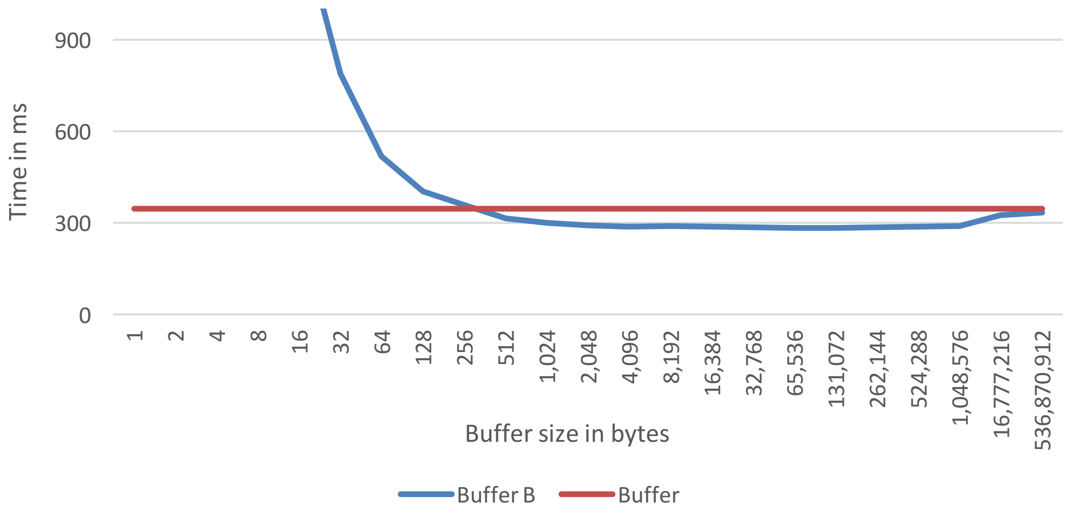


Figure 24: Comparison of buffer and buffer b

We conclude like for the write stream that *BufferB* will be preferred for the merge-sort implementation.

3.5. Data input/output stream through memory mapping

This implementation deals with read/write streams which implement the concept of memory mapping.

3.5.1. Goals and expectations

For this implementation we are again varying B and check the behaviour for different N, using the same ranges as for the previous implementation. All tests were performed for k=1. During the testing, a *java.nio.BufferOverflowException* error was thrown for buffer sizes smaller than 4bytes (equals to 1 integer). We assume this is due to the *mappedBuffer.putInt(*) method used in our application, which is expecting integers and thus does not work for sizes smaller than one integer (4 bytes).

We expect this stream type to show a similar behaviour for different B as *BufferB*, but perform better than the previous one due to the memory mapping concept explained previously. In addition, the mapping technique is leaving in the hand of the Operating System to decide how to load the data from the disks. Saying all that, whatever techniques are used by the OS the minimum I/O that will be needed to read/write all the integers is:

$$\text{Total I/o} = \text{RoundUP}(k * (N*4/4092))$$

3.5.2. Implementation

The implementation of read *Mapping* is more complex than the previous streams because it is using a different technique compared with the previous ones. In this class we override the *OpenFile()* method differently, we open the file now by using the java *FileChannel* class which is responsible to load in memory the file as explained previously. We read from the channel by loading the integers in buffers, do to that we have implemented a new method *OpenNextBuffer()*. This method is mapping in memory the new buffers by using the already opened file channel that will be used to read the integers from the file. Also the method *EndOfStream()* it is as well checking if the file channel has arrived to the end or not. If it is not the end of the file channel we open the next buffer to read. We have added a new method *IsChannelOpen()* which is checking if the channel is still open before manipulating it. In the end, we had to code differently the *Close()* method is taking care to close the channel.

The implementation of write *Mapping* is also more complex than the previous streams because it is using a different technique. In this class we override the *CreateFile()* differently, we create the file by a random access method, we use this file in the *FileChannel*. Also in this method we allocate the memory for the *MappedByteBuffer* which is the buffer that will be used during the writing of the integers into the file. We decided to use map the buffer directly in the operating system virtual memory to benefit from the way the OS is handling its virtual memory, if this will not be used the mapped buffer will be in the Java Memory space and will be managed by the JVM. The *WriteElement()* method is used to write the integers into the mapped buffer and when this buffer is full we map them into the used channel. The OS will decide when to write them in the hard disk.

The Close() method is taking care to first put in the mapped channel the last buffer and it is forcing the operating system to write the buffers in the disk if this is not already done and then close the channel plus the file.

3.5.3. Write stream

Following the same principle as before, we first generally look at the execution time trend for fixed N and varying buffer sizes. As expected, we see a rapid decrease for the first few bytes, so that we decided to cut out the first few sizes in order to get a more detailed picture for the remaining ones.

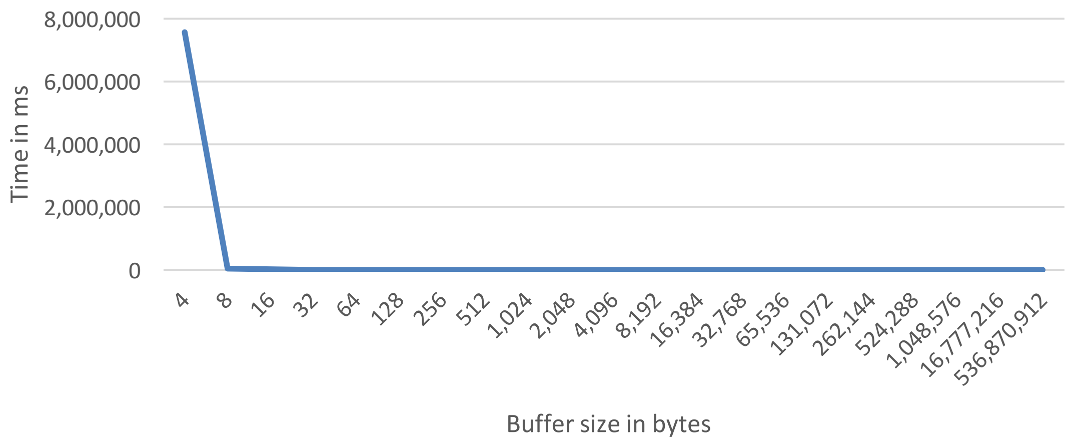


Figure 25: Writing time in function of buffer size for N=10.000.000

After excluding all buffer sizes <128 bytes, we can derive the optimal buffer size for each N, following the same principle as before. The optimal buffer values for different N are shown in the table below and exemplary shown for N=10.000.000 in the graph below.

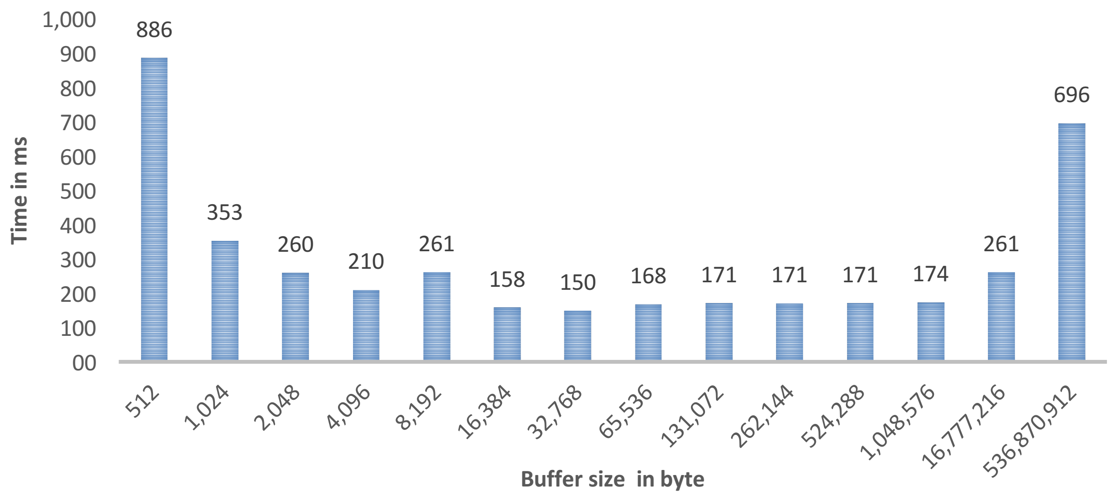


Figure 26: Time for different buffer sizes for N=10.000.000 and k=1

N	Optimal Buffer Size [bytes]
100.000	2^{15}
1.000.000	2^{16}
10.000.000	2^{15}
100.000.000	2^{16}
250.000.000	2^{20}

Table 5: Optimal buffer sizes for varying N

From the table above we can see that for every N an optimal buffer size is existing and around the same value for all file sizes, except the very large one. With increasing file size, the optimal buffer size also shifts to higher values accordingly and we can see in the graph that after reaching the block size of 4.096 bytes the performance increase is only marginal.

Comparison of Mapping, BufferB and Buffer

To determine the stream to be implemented, we compare this stream to the previous one. As the performance is very poor for low buffer sizes, we also take the *Buffer* implementation into account to get a comprehensive picture of the whole buffer size range as this one outperforms *BufferB* and *Mapping* for very low buffer sizes.

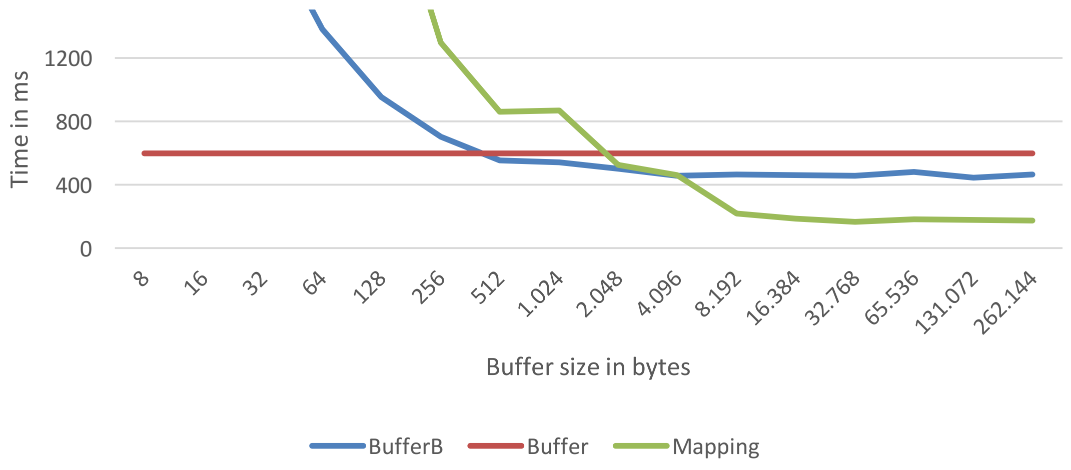


Figure 27: Performance comparison of Buffer, BufferB and Mapping streams

From the chart we can derive that the *Mapping* stream is performing significantly worse than the others for buffers smaller than 2.048 bytes. For every size greater or equal to 4.096 bytes, *Mapping* is the most performant one. However, upon a buffer size of 8.192 bytes the performance maximum is reached and further increase of the buffer size does not lead to an increased performance. As this buffer size equals to the default buffer size that is set for the Buffer stream, we can conclude that *Mapping* is outperforming both, the *BufferB* and *Buffer* stream, if it is assigned to the same buffer size. This also complies with our previous made assumptions that *Mapping* is performing very bad for low buffer values, but very well for large ones and reaches a performance maximum. As

we know already that from an I/O point of view *Mapping* is not performing better than *BufferB*, yet we see a higher performance in terms of time.
We assume this is due to the fact that in the Mapping application we leave it to the OS to handle the disk writes. If the buffer is full, the system is pushing the files to pages but we can't tell, when it will decide to write these pages to disk. We can only conclude that the memory mapping concept is indeed speeds up the application compared to a buffered stream. This leads to the conclusion that we choose *Mapping* for the sort-merge algorithm implementation.

3.5.4. Read stream

For read streams we saw the same behaviour as for the write streams but with different marginal values.

Varying buffer size and N=10.000.000

The graph below shows the reading time for different buffer sizes for N=10.000.000. We can see that initially the time is very high, so in order to find an optimal buffer size we remove those < 2^7 bytes.

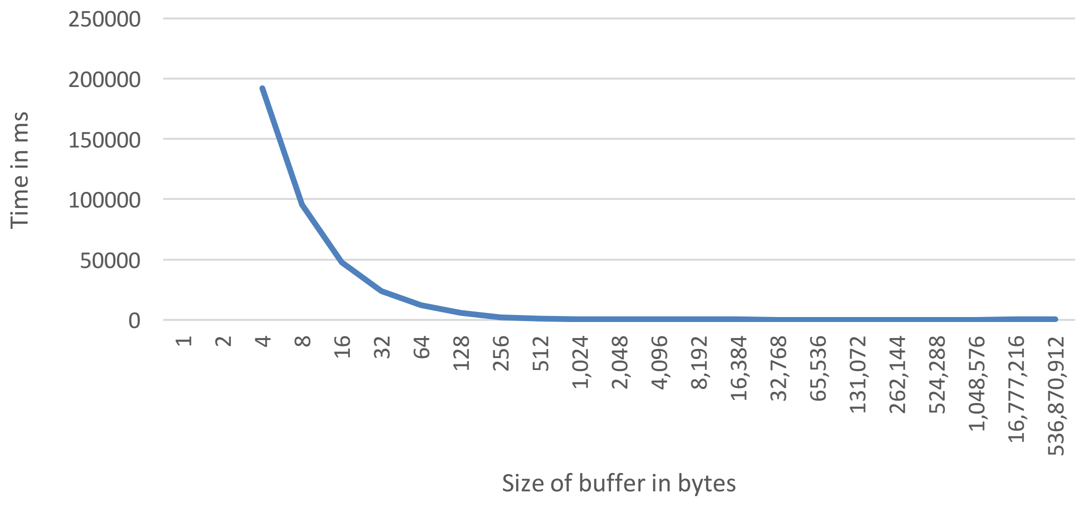


Figure 28: Reading time in function of buffer size for N=10.000.000

Optimal buffer size

We discovered different optimal buffer sizes for different values of N. We note that for increasing N the buffer size increases or as well according to the trend shown in the previous graph.

N	Optimal Buffer Size [bytes]
100.000	2^{12}
1.000.000	2^{18}
10.000.000	2^{18}
100.000.000	2^{20}
250.000.000	2^{20}

Table 6: Optimal buffer sizes for Mapping read stream

Comparison of Mapping, BufferB and Buffer

In order to compare these streams, we exemplarily set k=1 and N=10.000.000 and evaluate the different reading times for different buffer sizes.

From the chart below we see a similar behaviour to the write stream discussed previously. The reading time for the mapping stream decreases with increase in the buffer size being the best performant among the three streams.

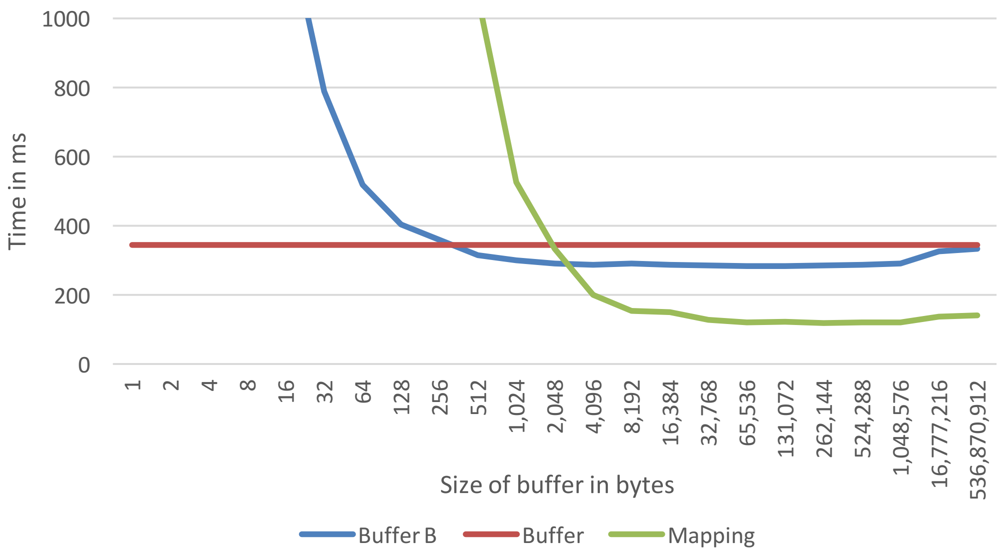


Figure 29: Comparison of mapping, buffer b , buffer

So, from the above observations we can say that, if the size of buffer is less than 2048 bytes, *Mapping* has the worst performance but if size is higher than that, it is the most performant. Clearly, *Buffer* is better than *Mapping* and *BufferB* if the buffer size is very low.

3.6. Summary of stream observations

The following table gives a summary of the advantages and disadvantages of the introduced stream types from a more generic point of view:

Advantages			
Simple	**Buffer**	**BufferB**	**Mapping**
No memory occupied by buffer	Higher performance due to buffering mechanism	Higher performance due to buffering mechanism	Mapping increases performance for large buffer sizes
Less dependent on OS	Easy implementation in Java	Buffer size can be adjusted to application needs	

Disadvantages			
Simple	**Buffer**	**BufferB**	**Mapping**
High cost (time), especially for very large files	Additional memory occupation due to buffer	Additional memory occupation due to buffer	Additional memory occupation due to buffer
	Buffer can’t be adjusted	Can perform very bad if buffer size is set to wrong/ low size	Very slow for low buffer values
		More complex implementation	More complex implementation

Table 7: Advantages and disadvantages of introduces stream types

From a performance point of view, we made the following decision for both, read and write streams:

- **Simple**: Worst performance overall
- **Buffer**: More performant than simple, yet not the best
- **BufferB**: Outperforms *Buffer* for B>=512 bytes
- **Mapping**: Best performance for B >= 8.192 bytes, will be used for the streams in the sort-merge algorithm implementation with a buffer size of 2^{16} to achieve best performance for all N and as memory occupation is not an issue in our case

4. Multiway-Merge-Sort Observations

In this section we are implementing and testing a multiway merge-sort algorithm based on the most performant stream type selected in section 3. Recall that the algorithm is taking a file of size N and the following parameters as an input:

- M — the size of the main memory available during the first sort phase, in number of 32-bit integers;
- d — the number of streams to merge in one pass in the later sort phases;

4.1. Goals and Expectations

The implementation goal of this stream type is to identify good choices for M and d for various inputs. We expect this stream type to perform very poor if M is set to a low value. The more memory is assigned, the better the performance will become. With respect to d we assume that the algorithm is performing better with larger values of d as this should decrease the sorting time. However, this algorithm is still based on the previously implemented read/write streams and thus also faces the same physical limitations. More detailed influences of the parameters will be discussed with the help of our I/O function in the testing result sections.
For our implemented algorithm, we assume the following cost function:

$$I/O = 2\,B(N)\ \log_{\mathrm{d}}(\frac{\mathrm{B(N)}}{\mathrm{M}})$$

In this formula, B(N) refers to the block size of the input file which in our case is always N/4.096. The factor 2 refers to the read and write I/O's that are needed for every memory chunk.

4.2. Implementation

We have implemented the external-memory merge-sort based on the algorithm described in section 2. We start by calling the main method *Sort()* of the defined class and passing the parameters that are explained in the source code or in the application itself. This file is split into smaller files also called chunks in our methods. The number of chunks depends on the specified parameter in *maxNrOfIntegersInMemory*. Before files being saved to disk as chunks, we sort the integers of each file/chunk in Memory by applying the java *Array.Sort* method which is using the Quick Sort algorithm in order to sort the integers.
In addition, we save in a tracking file all the names of the chunks that will be used during the merge phase of the implemented algorithm. In the merge-pass step we read the file names form the tracking file. The number of files to be read is specified by the other parameter *maxNrOfMergeStreams* that is specified by the user. We take care during the merge phase to use priority queues to keep streams in order not to mix the order of the integers to be sorted. The implemented queue supports both, ascending and descending order. The *java.util.PriorityQueue* method is used to implement this part.

In addition, we have used integrator methods provided from the external libraries of A*pache* and *Google Guava* to take in consideration as many chunks as specified from the parameter *maxNrOfMergeStreams* mentioned above during the merge phase.
In the last step, when only 1 chunk is left, meaning that the file is completely sorted, we rename it with the desired name.

4.3.Performance testing approach

Based on our results of section 3, we will use Mapping for both, the input and the output stream, together with a buffer of 2^{16} bytes in order to be able to also use this stream implementation for very large file sizes. We tested the following parameter combinations, including very large values for d (up to 30 like required in the first part of the report) as requested:

Allocated memory and streams			
M/d	**10**	**25**	**30**
1.000	-	-	-
10.000	-	-	-
100.000	-	-	-
1.000.000	-	-	-

Tested Input File Sizes			
N	10.000	1.000.000	10.000.000

Table 8: Parameter Settings for Merge-Sort Algorithm testing

With this, we should get a good overview for a wide range of N and will also see results for the case that the whole file fit in memory. We will not test higher values for d and N as we assume that we can draw meaningful conclusions with our chosen settings already and higher parameter values require inappropriate care and effort in execution as well as large disk sizes due to the tremendous amount of temporary files created. We tested 3 loops for N=100.000.000 with one parameter combination as well, but won't include results in this report as they showed expected behaviour and do not provide us new insights compared to the other tested N, but take exorbitant more time to execute. Recall, that the number of integers shown in the table above equals to the following file sizes:

- 10.000 integers = 0.04 MB
- 1.000.000 integers = 4 MB
- 10.000.000 integers = 40 MB
- (100.000.000 integers = 400MB)

Our objective for this testing is to find the optimal combination of d and M for each N and analyze their influence on the cost of the algorithm. Each testing was executed 5 times and the average was taken, not considering the first warm-up run.
In the application interface we chose *[3] Mway merge-sort* and again *[2] input from file* and recall to specify the following parameters:

Read Stream Type (only 1 per row); Write Stream Type(only 1 per row); Nr of streams; Nr Integers in Memory (Ex: 1000, 10000, 100000...); Buffer size range (Ex: 1,6); Nr Loops; Input file (full path); Output file; true (Ascending sort order)
Example:
Mapping;Mapping;10;1000;20,20;5;/home/username/10000.dat;Test;true

The application executes the statement inserted and produces a csv file containing the specified parameters and the respective execution times.

4.4.Performance testing results

The following sub-sections discuss our testing results. We first start with selecting one N and examine how varying d and M separately influences the cost (time) of our algorithm. Then we continue with an overview of the testing results and define the optimal combination of d and M for each tested N. In a last step we compare our results to an in-memory sort algorithm for those cases, where the whole file was fitting in memory.

4.4.1. Influence of d in function of cost

The following graph shows varying values of d for M=1.000 and N=1.000.000 fixed.

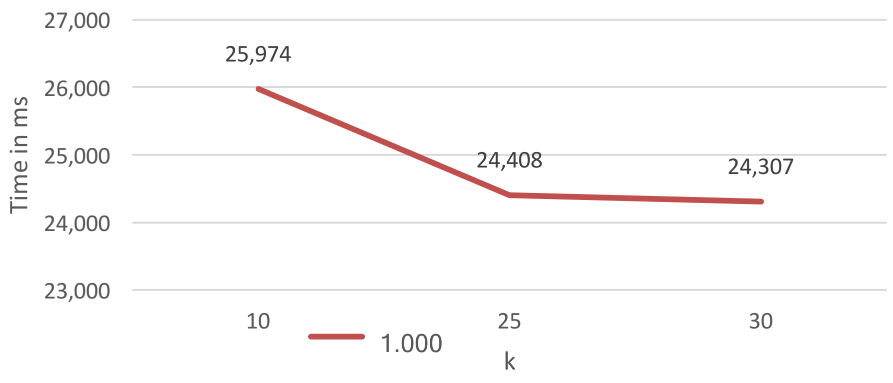


Figure 30: Influence of d with N=1.000.000 and M=1.000 fixed

We see a slightly decreasing trend in time for an increasing number of streams as the number of parallel streams is reducing the time for the sorting process. This can also be seen from the cost formula:

$$I/O = 2\,B(N)\,\log_{\mathrm{d}}\left(\frac{\mathrm{B(N)}}{\mathrm{M}}\right)$$

With increasing d the log term becomes slightly smaller and approaches a minimum (highest performance), but which still is determined by the underlying read/write stream implementations. We can see in our graphic that we are very close to this point at 25-30 streams already as we can’t see significant further improvements.
However, for larger numbers of M we see some different gradients, but as the values for k are still very close and thus don’t suggest completely different behaviour, we decided not to run additional tests.

Nevertheless, in our analysis of the influence of M we will see that the choice of M has a far more significant impact on the performance as d has. The performance gains for increasing d from 10 to 30 in the above shown case are 6%, whereas with increasing the allocated memory from 1.000 to 1.000.000 results in a performance gain of 90% is possible. In some of our results we even saw a stagnating or slightly increasing trend, but as the results differed in a nanosecond scale we treat those occurrences as noise.
If we take a look at the greater picture and compare different sizes of memory for different stream numbers, we see how small the variations along different d are. The lines in the chart below where we plot the time for different M over the tested numbers of streams seem to be almost constant. However, we note that increasing M significantly reduces cost, especially compared to low values (10.000 integers, red line), which shows an exorbitant poor performance.

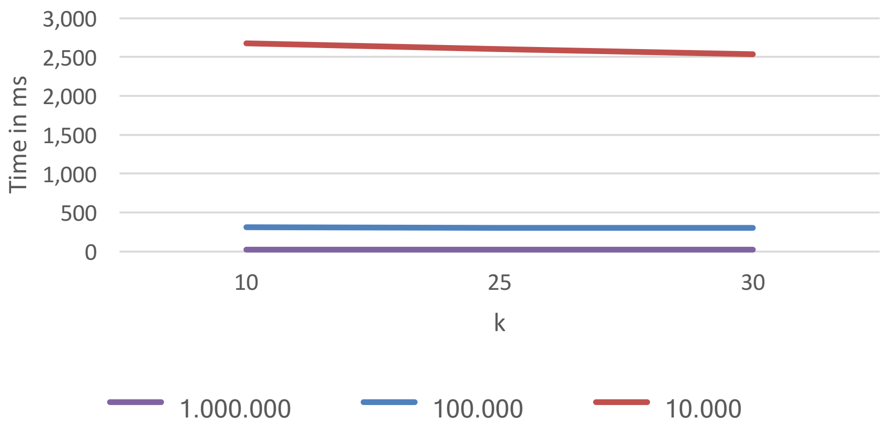


Figure 31: Varying d for different M (int)

4.4.2. Influence of M in function of cost

No, we are plotting the execution times for varying M, exemplarily shown for d=30 and N=100.000.000 fixed. This observation holds for other N and d as well even though the gradient of the curve might differ. As expected we clearly see a significant decrease in cost with rising M, which can also be derived from the cost formula. This follows the logic that the more memory space we assign, the bigger the chunks in which we divide the input file, thus the larger the files that can be sorted or merged at once before another I/O needs to be made again, which leads to significantly longer execution times. With less memory space allocated, more passes during the sorting process are required respectively.

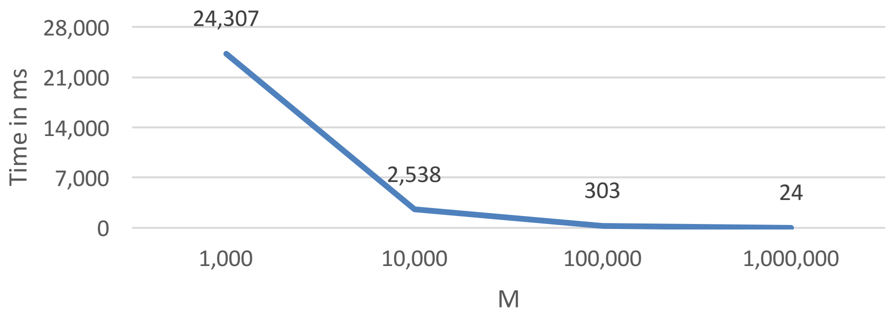


Figure 32: Influence of M for d=30 and N=100.000.000 fixed

4.4.3. Comparison of different N with d, M fixed

In a last step, we shall analyse how the cost for raising N differ on the example of d=25 and M=100.000 fixed. Again, this trend holds for all N tested. The figure below shows the execution times for the different input file sizes. From the graph we can derive that cost scale proportionally with N, similar to what we have seen in the *Simple* write stream discussion, but at significantly lower cost. This can also be derived from the cost formula where N appears as a factor that is multiplying cost. This also makes sense if we think about it: The larger the file, in the more file chunks we need to split it if M and d are fixed and thus the more passes and I/O's are needed.

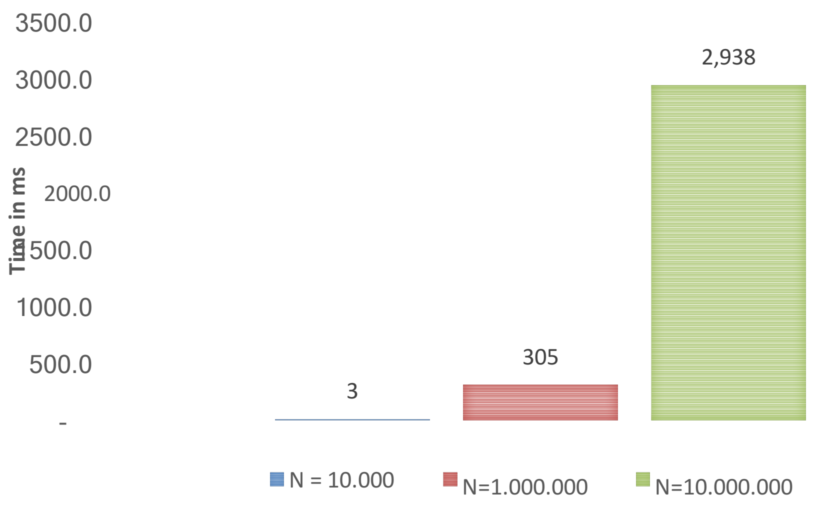


Figure 33: Performance for varying N with d=25 and M=100.000 fixed

4.4.4. Defining optimal d, M combination for each N and conclusions

The following table summarizes the best combination of M and d for the tested N:

	Input File Sizes		
M, d / N	10.000	1.000.000	10.000.000
M	10.000	1.000.000	1.000.000
d	30	30	30

Table 9: Optimal parameter combination for each N

If we analyse this result further, we glean that the best performance for the all tested file sizes was gained for 25 streams and a value of M that equals the value of N or is close to it. However, as mentioned before, even lower numbers of streams might result in the best performance already, but would weren't tested here. Additional tests could be done with all incremental stream sizes to obtain a more precise optimum, which we did not consider to be necessary for the result discussions made in this report. Increasing the Memory size or streams any further does not further optimize performance and would only be a waste of expensive memory space.

The obtained optimal values for M actually raise a further interesting fact: For N = 10.000 and 1.000.000 the assigned memory equals the file size. In this case, the whole file fits in memory and can thus be sorted in-memory in one pass and does not require additional I/O's for the passes coming along with the external merges-sort process. Obviously, in-memory sorts are orders of magnitudes faster than any external sorts as they do not have to deal with any disk speed or access latencies throughout the sorting process. In our application we are calling the *Array.sort()* method, which is implementing a Quicksort.

As a conclusion we can state that our results met all our expectations and that a m-way sort-merge algorithm is certainly not a good choice if memory space is very limited and thus, large files need to be divided into many chunks. For files that completely fit in memory, an in-memory sort is certainly the better option as the sort-merge algorithm does not provide any benefits here. We also keep in mind that if the algorithm shall be improved, we should focus on finding the optimal M for our implementation as the performance gain with higher M is much stronger than the possible performance gains if d is varied. We actually also sport-checked higher values for d like 100 and 250 streams but the performance gains were so marginal compared to the tremendous amount of extra time and disk space (!) needed that we decided to not include and further test those values.

5. Final Conclusion

The objective of this project was to get familiar with different possibilities to read and write data from/to secondary memory. Hereto, we first implemented 4 different read and write stream types, whose performance was analysed for various sizes of k, N and B. The most performant stream types were then chosen to implement an external-memory merge-sort algorithm, followed by an evaluation of the respective performances.

With respect to the four stream types, the *Simple* stream was performing the worst, whereas *Mapping* showed the highest performance (for write streams from buffer sizes of 8.192 bytes onwards) and was chosen for the write as well as the read stream in the sort-merge algorithm implementation. Generally speaking, read streams had lower cost than the write streams.

The final conclusion of the sort-merge algorithm was a very low performance for low sizes of assigned memory but good results for higher memory sizes and number of streams. However, if the files are completely fitting into memory, an in-memory algorithm would be a more performant implementation choice. Next to choosing the optimal read and write streams, the influence of the assigned memory space (M) has a huge impact on the algorithm performance and should be preferred for optimization before d.

All in all, we were able to experience advantages and disadvantages in terms of performance and ease of implementation for many different stream implementations first hand. As stated in the motivation of this report, performance tuning becomes especially important nowadays, when people or corporations have to deal with ever increasing file sizes. In this project we learned that a bit of testing and some adjustments of the underlying algorithms or buffer sizes is already enough to increase the performance of an I/O determined application up to several orders of magnitude. Finally, we were also able to get a better understanding of the performance optimization practices that were introduced in the lectures.

Another important learning from the project was to find and plan the best testing steps. The testing had cost us much time and we had to repeat several tests because we tested it incorrectly or tested wrong parameter combinations that were not yielding the desired solutions. We learned that is of great importance to thoroughly think through the problem that should be solved and plan the testing beforehand in order to get not only the best results but also not to waste time in useless testing as sometime it is enough to understanding and discussing an overall trend rather than having all details.

Sources

1) Course slides
2) https://docs.microsoft.com/en-us/dotnet/standard/io/memory-mapped-files, opened January 6, 2018
3) https://msdn.microsoft.com/en-us/library/ms810613.aspx, opened Jan 6, 2018
4) Oracle documentation: https://docs.oracle.com/javase/7/docs/api/java/nio, opened Jan 6, 2018
5) http://cs.ulb.ac.be/public/_media/teaching/infoh417/slides-lect6.pdf, opened Jan 6, 2018
6) http://www-inst.eecs.berkeley.edu/~cs186/sp03/section/mreview2.pdf, opened Jan 6, 2018
7) http://www.csbio.unc.edu/mcmillan/Media/Comp521F10Lecture17.pdf
8) https://blogs.msdn.microsoft.com/khen1234/2006/01/30/memory-mapped-files-and-how-they-work/, opened January 6, 2018
9) https://www.infoworld.com/article/2898365/microsoft-net/working-with-memory-mapped-files-in-net.html

Appendix

The following tables show the detailed testing results for all graphs presented in our report. The fields with a dash, highlighted in dark grey mean they were not tested as we considered it not to be necessary for solving our problem. Whenever the value 0 appears, it means that results were much smaller than Milliseconds. Even though we also measured the nanoseconds, we decided to use ms throughout the report to make charts more readable and comparable as we would otherwise deal with very large numbers for larger N.

Write streams

WRITE SIMPLE	**AVG WRITING TIME IN MS FOR DIFFERENT NO OF INTEGERS (N)**						
K/N	1.000	10.000	100.000	1.000.000	10.000.000	100.000.000	250.000.000
1	4	43	401	3939	39805	472981	1113216
2	8	80	790	7991	80613	-	-
3	11	119	1185	11877	122739	-	-
4	16	158	1583	15944	163679	-	-
5	20	198	1997	20473	201869	-	-
10	40	396	3967	40724	406810	-	-
20	85	796	8176	81613	861798	-	-
30	120	1197	12103	122307	1242396	-	-

WRITE BUFFER	**AVG WRITING TIME IN MS FOR DIFFERENT NO OF INTEGERS (N)**						
K/N	1.000	10.000	100.000	1.000.000	10.000.000	100.000.000	250.000.000
1	1	1	6	46	598	6.237	15.335
2	0	1	8	77	1.179	15.934	39.681
3	0	1	12	125	1.861	-	-
4	0	1	19	158	2.581	-	-
5	0	2	23	211	3.306	-	-
10	3	4	43	549	6.709	-	-
20	9	12	80	1.230	15.254	-	-
30	15	18	119	1.973	23.870	-	-

BUFFER B WRITE **AVG WRITING TIME IN MS**

BUFFER SIZE 2^X	Buffer size (bytes)	1.000	10.000	100.000	1.000.000	10.000.000	100.000.000	250.000.000
0	1	9,2	48,6	462,8	4488,7	46731,0	-	-
1	2	4,0	23,2	238,8	2265,7	23245,0	238386,7	576043,0
2	4	2,0	11,4	120,4	1157,0	11727,3	-	-
3	8	1,0	5,8	64,2	588,3	6575,7	-	-
4	16	0,8	3,0	31,6	310,3	3555,0	-	-
5	32	0,0	1,0	22,0	171,7	2084,3	-	-
6	64	0,6	1,0	13,8	100,0	1382,0	-	-
7	128	0,0	0,0	12,3	64,7	954,3	-	-
8	256	0,0	0,0	11,3	47,7	625,0	7238,7	19885,3
9	512	0,0	0,0	8,7	37,0	485,3	6391,0	19756,7
10	1,024	0,0	0,0	8,2	33,3	493,7	7751,0	19870,3
11	2,048	0,0	0,0	8,0	29,7	483,7	7990,0	19932,3
12	4,096	0,0	0,0	3,7	28,3	474,7	7817,3	19855,0
13	8,192	0,0	0,0	3,1	28,7	484,0	7992,7	19808,0
14	16,384	0,0	0,0	2,7	35,0	433,3	7899,0	19888,3
15	32,768	0,0	0,0	2,3	28,0	458,7	7917,0	19969,7
16	65,536	0,0	0,0	2,4	29,7	437,0	7935,0	19815,7
17	131,072	0,0	0,0	2,4	28,7	452,3	7951,7	19805,0
18	262,144	0,0	0,0	2,4	28,0	471,0	7941,7	19944,7
19	524,288	0,0	0,0	2,7	38,7	463,7	7900,0	19833,7
20	1,048,576	0,0	0,0	3,0	39,3	433,7	8051,0	19954,3
25	16,777,216	0,0	0,0	6,0	34,0	489,0	-	-
30	536,870,912	0,0	0,0	6,1	37,3	592,0	7524,0	15107,3

MAPPING WRITE **AVG WRITING TIME IN MS**

BUFFER SIZE 2^X	Buffer size (bytes)	1.000	10.000	100.000	1.000.000	10.000.000	100.000.00	250.000.000
0	1	java.nio.BufferOverflowException						
1	2							
2	4	24,2	61,2	597,4	112203,0	7558049,3	-	-
3	8	13,2	26	329,4	48670,2	35035,0	-	-
4	16	4	12,4	173,4	23088,8	18014,7	-	-
5	32	3	7,2	78,0	12955,0	9131,0	151549,7	497284,7
6	64	2,6	3,4	42,2	4264,4	5104,7	-	-
7	128	2	4	31,0	254,2	4902,7	-	-
8	256	2,6	3,6	29,7	182,7	4880,7	17475,3	59522,7
9	512	1,6	2	18,0	46,0	886,3	11531,3	38276,7
10	1,024	1,2	2	8,7	36,3	353,3	6501,0	16442,0
11	2,048	1,2	2	10,3	48,0	260,0	3588,0	8770,0
12	4,096	1,6	2	5,7	35,3	210,0	2390,3	6533,0
13	8,192	1,6	2	4,3	22,3	261,0	1853,7	5098,3
14	16,384	1,6	2	4,0	20,7	158,3	1587,3	4447,0
15	32,768	1,6	2	3,3	20,0	149,7	1533,3	4041,3
16	65,536	1,6	2	4,0	19,0	168,3	1508,0	3916,7
17	131,072	1,6	2	3,7	19,0	171,3	1515,3	3808,0
18	262,144	1,6	2	4,3	19,7	170,7	1463,3	3800,7
19	524,288	1,6	2	6,0	22,7	171,0	1673,0	3782,3
20	1,048,576	1,6	2	6,7	24,0	173,7	1616,7	3839,3
25	16,777,216	6,2	7,2	11,4	36,2	261,0	-	-
30	536,870,912	152,4	163,2	185,4	198,0	695,7	4842,0	14208,3

Read streams

READ SIMPLE	**AVG WRITING TIME IN MS**						
NO OF STREAMS	1.000	10.000	100.000	1.000.000	10.000.000	100.000.000	250.000.000
1	1	14	135	1.327	13.209	143.832	356.603
2	2	26	276	2.610	28.032	-	-
3	4	39	401	3.954	41.217	-	-
4	5	52	526	5.225	55.618	-	-
5	6	65	666	6.548	72.468	-	-
10	13	132	1.387	13.256	142.726	-	-
20	27	282	2.737	27.411	281.809	-	-
30	38	389	4.029	40.392	425.333	-	-

READ BUFFER	**AVG WRITING TIME IN MS**						
NO OF STREAMS	1.000	10.000	100.000	1.000.000	10.000.000	100.000.000	250.000.000
1	0	4	10	50	345	3.349	9.339
2	0	0	6	64	612	6.260	15.773
3	0	0	8	88	900	-	-
4	0	1	11	115	1.169	-	-
5	0	1	14	165	1.436	-	-
10	0	2	28	278	2.821	-	-
20	0	5	57	589	5.811	-	-
30	1	9	97	917	8.710	-	-

BUFFER B READ		**AVG WRITING TIME IN MS**						
BUFFER SIZE 2^X	Buffer size (bytes)	1.000	10.000	100.000	1.000.000	10.000.000	100.000.000	250.000.000
0	1	1,0	23,8	160,4	1566,0	16589,0		
1	2	0,4	10,0	78,4	796,3	8229,7	79844,7	209194,3
2	4	0,0	4,0	41,6	436,3	4382,0	-	-
3	8	0,0	2,0	21,0	221,3	2199,3	-	-
4	16	0,0	1,8	12,6	124,3	1298,0	-	-
5	32	0,0	0,0	8,2	81,0	789,3	-	-
6	64	0,0	0,0	5,0	56,7	518,3	-	-
7	128	0,0	0,2	6,0	41,0	403,0	-	-
8	256	0,0	0,0	7,0	38,7	358,7	3599,3	8898,3
9	512	0,0	0,0	5,0	33,2	314,3	3139,3	7901,7
10	1,024	0,0	0,0	3,0	30,3	300,0	3049,7	7462,3
11	2,048	0,0	0,0	5,5	29,3	291,3	3034,7	7404,3
12	4,096	0,0	0,0	2,7	29,7	286,7	3078,3	7383,3
13	8,192	0,0	0,0	6,0	29,4	290,3	2976,7	7261,3
14	16,384	0,0	0,0	2,7	28,7	287,0	2994,7	7260,7
15	32,768	0,0	0,0	6,0	29,3	284,7	2953,3	7379,7
16	65,536	0,0	0,0	7,0	30,0	283,7	2983,0	7342,0
17	131,072	0,0	0,0	4,0	29,7	283,0	2934,7	7219,3
18	262,144	0,0	0,0	4,3	29,0	284,7	2995,7	7261,0
19	524,288	0,0	0,0	4,0	30,3	287,3	2958,7	7305,0
20	1,048,576	0,0	0,0	4,7	30,3	290,3	3011,3	7464,0
25	16,777,216	0,0	0,0	4,0	30,7	325,3	-	-
30	536,870,912	0,0	0,6	4,0	31,3	333,7	4030,7	10101,7

MAPPING READ		**AVG WRITING TIME IN MS**						
BUFFER SIZE 2^X	Buffer size (bytes)	1.000	10.000	100.000	1.000.000	10.000.000	100.000.000	250.000.000
0	1	java.nio.BufferOverflowException						
1	2							
2	4	11,80	145,20	1.601,40	18.602,60	191.839,00		
3	8	7,80	75,20	1.080,00	9.404,00	95.316,33		
4	16	1,00	52,80	449,00	4.739,40	47.611,67		
5	32	0	36,40	289,20	2.340,20	23.843,33	236.541,67	587.770,67
6	64	1,80	16,40	93,80	1.195,60	11.943,67		
7	128	0	1,40	52,00	559,20	5.932,33		
8	256	0	0	23,0	143,5	2153,0	21724,3	53946,0
9	512	0	0	15,7	122,7	1062,7	10592,0	26337,7
10	1,024	0	0	5,7	102,0	526,0	4957,0	12475,3
11	2,048	0	0	5,7	52,0	332,7	2788,3	6990,7
12	4,096	0	0	1,0	28,0	200,3	1930,7	4713,7
13	8,192	0	0	1,3	17,7	153,3	1498,7	3767,7
14	16,384	0	0	1,0	13,3	151,0	1333,3	3353,7
15	32,768	0	0	1,0	15,0	128,0	1260,3	3167,7
16	65,536	0	0	1,0	13,0	121,3	1223,3	3066,0
17	131,072	0	0	1,0	13,7	122,7	1217,7	3059,3
18	262,144	0	0	1,0	12,0	119,3	1193,7	2991,0
19	524,288	0	0	1,0	12,3	120,7	1192,3	2991,3
20	1,048,576	0,4	0	1,0	12,3	121,3	1189,3	2980,0
25	16,777,216	0,4	0	1,0	14,8	138,0	-	-
30	536,870,912	0	0	1,2	16,2	140,3	1359,0	3385,7

Sort-Merge Algorithm

Average execution time in ms

N = 10.000			
M/k	10	25	30
1.000	251	254	264
10.000	3	3	4
100.000	3	3	3
1.000.000	6	6	6
N=1.000.000			
M/k	10	25	30
1.000	25.974	24.408	24.307
10.000	2.677	2.599	2.538
100.000	309	305	303
1.000.000	24	24	24
N=10.000.000			
M/k	10	25	30
1.000	225.968	226.454	207.942
10.000	22.958	21.453	21.062
100.000	3.059	2.938	2.853
1.000.000	190	170	149
N=100.000.000			
M/k	10	25	30
1.000	2.513.006		
10.000			
100.000	Not tested as it didn’t yield new insights		
1.000.000			