External-Memory Merge-Sort Algorithm

Danish Amjad
Edoardo Conte
Carlos Muñiz Cuza

December 2018

Contents

1 Introduction and Environment

The goal of this project is to implement a secondary memory sort algorithm, specifically, the multi-way merge sort. In order to accomplish it, we first need to analyze different methods to read and write data on secondary memory. The chapter 2 deeply explores the behaviour of different stream implementations. At its end, a choice is made in order to implement the multi-way merge sort algorithm. Chapter 3 explains the principle of the algorithm, considering its particular implementation. Moreover, it is tested and its performance is measured under different circumstances. The whole project has been written in Java. We used maven as our dependency management tool. We used log4j2 for logging, junit for unit testing and jmh libs for benchmarking.

1.1 Benchmarking Environment

For benchmarking, we have used JMH framework for JAVA. JMH is a benchmark harness for creating and analyzing benchmarks for Java and other JVM languages. JMH provides an easy setup through maven. Moreover, it's also very easy to use. It creates the benchmarks with the help of annotations. It provides a variety of metrics for measuring the performance i.e. throughput, average execution time, single shot time. We have used single shot time which only measures the time for one execution. We have also taken care of JVM warmup issues by introducing warmup iterations.

We tested our results for a wide range of number of integers, N, all in the power of two. For each value of N, We created a temporary file by using random numbers generator provided by JAVA. We tried several values of M and for each value of M we tried different values of d to cover most of the meaningful parameters space.

We used two forks for JVM processes, 2 warmup iterations and 5 measurement iterations in order to get smooth and consistent results across several runs.

2 Observations on streams

In order to implement the multi-way merge sort algorithm, the first step is to implement an efficient class to abstract the writing and reading operations. According to the specifications of the assignment, 4 different implementations are tested. Each reading stream encloses *open*, *close*, $readNext$ and $isEndOfStream$ methods. While, the writing ones enclose *create*, *write* and *close* methods. The following are the 4 classes used for testing:

- **Simple Input/Output Stream:** each element is written and read without the usage of any buffer. Anytime the $readNext$ method is called, only one element is loaded in memory from the file mapped by the stream. Similarly, the *write* method writes one element at a time on disk. However, some observations on the implementation need to be done. In order to read

and write integers, the *DataInputStream* (Output) calss has been used. The object instantiated is wrapped around a *FileInputStream* (Output). Digging in the implementation of *readInt* and *writeInt* methods, we discover that the methods call *read* and *write* methods from the underlying stream 4 times for each call. Each operation reads or write a byte. After 4 operations, an integer is either returned or written. However, in order to perform such operation, each read or write int call requires 4 IOs. In the next sections, we will experiment this implementation constraint.

- **Auto Buffered Input/Output Stream:** the implementation of this stream relies on the usage of $BufferedInputStream$ (Output respectively). This class auto-implements an internal buffer in memory to optimize reading and writing operations for large data. *readInt* and *writeInt* methods from the *DataInputStream* (Output) first operate on the buffer, then, when the buffer is empty or full respectively, new elements are loaded from disk in one batch (or written). A constructor paramenter may be used to specify the size of the buffer.

- **Buffered Input/Output Stream:** this stream is similar to the previous one, but with a manual implementation of the buffer logic. The stream is generated with a parameter which specifies the buffer length. The $ByteBuffer$ class has been used to implement the buffer, while the *read* method of the *InputStream* class has been used to completely load the buffer. Similarly, the *write* method of the *OutputStream* class has been used to flush the entire buffer into disk.

- **Memory Mapped Input/Output Stream:** in order to properly explain this implementation, in the next section we are giving a general introduction to memory mapped streams.

2.1 Memory Mapped Streams

Memory mapped streams take advantage of the OS usage of memory. They simplify the usage of IO operations and increase the performances for large files processing. In order to achieve these results, they map portions of the file inside the memory. More specifically, the kernel memory is used. Due to this decision, the virtual memory manager of the OS is used, which is a highly optimized component. Anytime the user wants to read or to write data on disk, instead, he operates with memory. In a successive moment, the virtual memory manager may decide to persist this information in secondary memory. The optimization for large files processing reside in the default page size, which is 4kB. Anytime, a new portion of the file is requested, an entire page is loaded in memory. At the same time, when a page in memory become full, it is persisted on disk. Another advantage of this logic is that no user memory is used. Consequently, if the application crashes, data is still persisted on disk, since it stays on kernel memory.

However, while this method brings several positive aspects, it also has some drawbacks. Whenever the application handles relatively small files, a lot of kernel memory is wasted. Moreover, the user has no control on memory utilization. All the memory management is left to the operating system, which chooses when loading and removing data chunks. In this scenario, caching for a specific content inside the file cannot be performed. Every time that content is requested, the user does not know if the system has to load it again from disk or it is still in memory. Another drawback is page faults. Every time the application requests for an empty address in memory, the operating system calls a page fault. Usually, it is handled by another thread, which loads data from disk. If the application jumps in different sections of the file, it will cause a lot of page faults, implying performance degradation.

In order to implement this particular stream, we use the $MappedByteBuffer$ class from the *java.nio* library. First, a $FileChannel$ object is created through the *getChannel* method of $RandomAccesFile$ class. Subsequently, the channel is mapped in memory and a $MappedByteBuffer$ is returned. In the Input implementation, the buffer is instantiated with a size equal to the file size. However, the operating system may physically allocate only a part of that buffer in memory, loading one page at a time. Due to this allocation, the application may easily iterate on the buffer, without re-allocating it.

In the Output implementation, the stream has no chance to know the dimension of the final file. In order to leave flexibility to this implementation, an initial buffer size is passed as parameter for the constructor. Whenever the application reaches the end of the mapped buffer, it doubles its size and maps it again. This operation can be considered expensive, so we decide to double the size of the buffer each time. It's important to note that the operating system decides how to effectively manage that portion of memory, while the application can safely address it.

However, in the Java implementation, a limitation is present. The $ByteBuffer$ returned by the *map* method only works with Integers. This implies that the maximum possible size for mapping is 2GB, the max integer. In order to avoid this problem, the application should split a huge file in chunks of 2GB and work on them sequentially. In the next sections, we are considering files with a smaller size, in order to remain in the scope of the project. Regarding multi-way merge sort implementation, only file sizes up to 2GB will be considered.

2.2 Expected behavior

After designing four different implementations for handling IO operations, we are requested to test their performances. Before explaining the testing process, we make some hypotheses on their effective behaviors. We know that for writing and reading a big amount of data, buffering is always better than simple IO operations. We reduce the number of system calls to $\frac{N}{M}$, where N is the number of data and M is the buffer size. However, we can assume that the Buffered Input/Output stream of Java could be slightly more performing than our own implementation. Moreover, we can manually set the buffer size for both

implementations, comparing the two results with the same parameters.

However, we can also compare the memory mapped stream with the buffered one. According to the theoretical foundations, we can infer that the former should be better than the latter. Since the memory is managed by the OS in the kernel space, it should be faster with a large number of IO operations. Also, since the reading and writing operations are sequential, the OS should not handle a significant number of page faults. With these ideas in mind we test the system in the next section.

2.2.1 Expected cost functions

In this sub section we analyze theoretical cost functions for the 4 different stream implementations. The cost function computes the expected number of IO operations. Assuming k as number of parallel streams, N as number of integers considered and B as the buffer size, we build these functions:

- **Simple stream:**

$$k_{eff} * 2 * N * 4 \tag{1}$$

  K_{eff} is considered as $\frac{k}{cores}$, where *cores* varies from 1 to 4, the number of cores of the computer used. $2 * N$ is used to specify both reading and writing operations and 4 is the number of bytes composing an integer.

- **Buffered stream:**

$$k_{eff} * 2 * \frac{N}{B} \tag{2}$$

  The formula applies for both our Java implementation and our own. This time, the number of integers read is divided by the buffer size. Intuitively, we can assume that with a large buffer size the formula does not hold anymore. Internal implementation of the *read* and *write* methods allows for reading and writing a maximum size of bytes at a time. It can be considered as one page, usually 4kB.

- **Memory mapped stream:**

$$k_{eff} * 2 * \frac{N * 4}{4096} \tag{3}$$

  In order to compute this formula, we considered that memory mapped streams read and write one page at a time, of 4kB.

2.3 Experimental observations

In this section we perform several experiments in order to discover different streams performance. We test each stream inside the JMH framework, making each thread writing and reading n integers in an independent file. For each execution we can vary the number of integers used, the number of streams (threads on independent files) and, for the last three implementations, the buffer size. We first consider the comparison between the Simple Input Output Stream with

the Buffered one, exploring their differences. Then, we compare our Buffered Stream implementation with the Java implementation. Executing such comparisons varying mainly the number of elements and the number of streams, we need to define how the buffer size influences the performances of different streams. Finally, we perform a deep comparison between Buffered and Memory Mapped streams. We perform the following measures with 3 warmup iterations, 5 effective iterations and 2 forks (the benchmark is repeated twice).

In order to perform the following experiments, a MacBook Pro has been used. Specifically, 4-core CPU, Intel Core i7, 2.9GHz, with 8MB of cache L3 and 256kB per core, with 16GB of RAM and SSD with 1TB of storage.

2.3.1 Simple IO and Buffered Streams on a single thread

We first consider the comparison between Simple IO and Buffered Streams on a single thread. In order to see the effective difference, we analyze two different configurations for the Buffered stream:

- Buffer size = 4 bytes (1 integer)
- Buffer size = 8 bytes (2 integers)

We would expect that the first configuration is twice slower the second one and comparable to the Simple IO. However, as briefly explained before, the Simple IO uses 4 IO operations in order to read or write an integer. Considering this information, we expect that it performs 8 times slower than the Buffered stream with the buffer size equals to 2 integers.

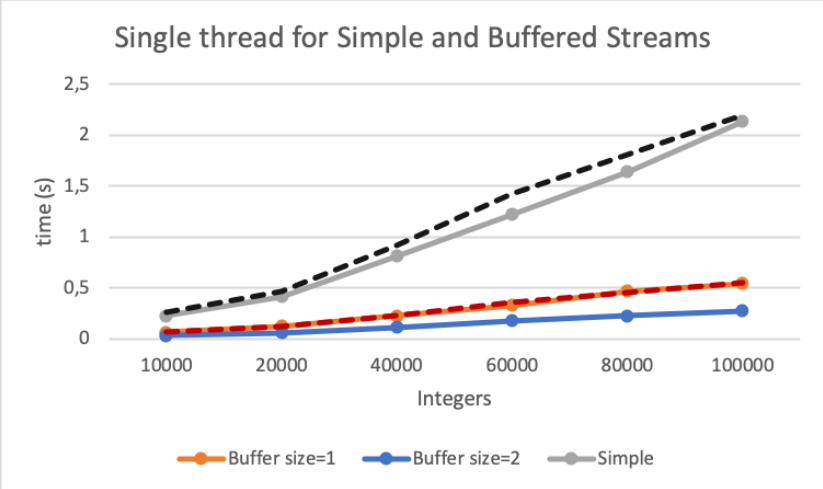

Figure 1: Simple IO and Buffered Streams on single thread

In figure 1 we identify the dotted lines as the expected behaviours. It can be noted that the real behaviour is really close to the expected one. We can see

that effectively, the buffered implementation with only 1 integer in the buffer is performing 1 IO operation for each element, while the Simple one is performing 4 IOs for each element. Concluding, we do not report results for multi-streams architecture in this scenario, since the result is simply scaled with the number of threads.

2.3.2 Buffered streams

In this section, we compare the behaviour of our implementation of Buffered Streams with the one already implemented in Java. As a first comparison, we consider only one thread, with a fixed buffer size and a variable number of integers. As buffer size, we selected 1000 integers, which corresponds to around 4kB, which is a standard size for file pages.

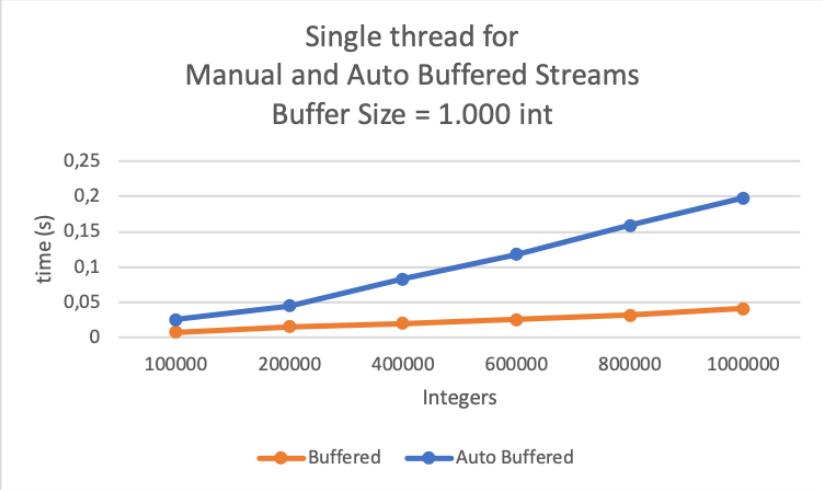

Figure 2: Comparison between the two buffered stream implementations.

Despite the assumptions we have made in the previous chapter, our implementation results to be faster with one single thread. Looking inside the implementation of the Buffered Java class, we see that a byte array has been used. Moreover, every time new data is loaded from disk, previous data in the buffer is kept, until a certain mark is reached. With this functionality, we can cache some data, but it is not useful in our benchmark. Probably, this is one of the reasons why our implementation results to be faster. However, in general, we can assume that the Java implementation adds more overhead, not considered in our benchmark.

As a second comparison, we consider the same implementations in a multi-threaded environment. We consider their performances with the same buffer size, with 1000000 integers and a variable number of threads.

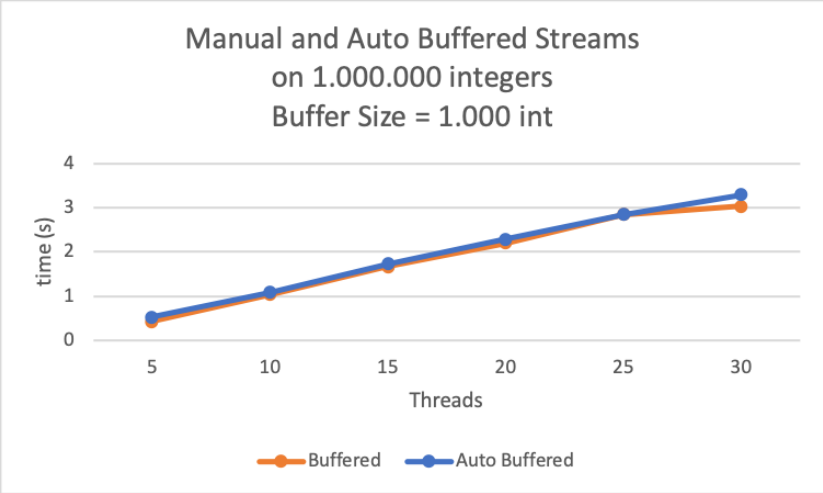

Figure 3: Comparison between buffered streams on multiple threads.

In the figure 3 we can see that the behaviour changes when considering multiple threads. Now, the performances of the two implementations can be comparable. However, we can still note a slight faster performance for our implementation. Due to this fact, in the next sections we are considering our implementation when dealing with buffered streams.

2.3.3 Buffer size for Buffered Stream

In this section, we are comparing the performances of the Buffered stream, with multiple threads and a variable size of the buffer. In order to show how the buffer size affects execution time, we choose to evaluate the performances with 3 different streams number.

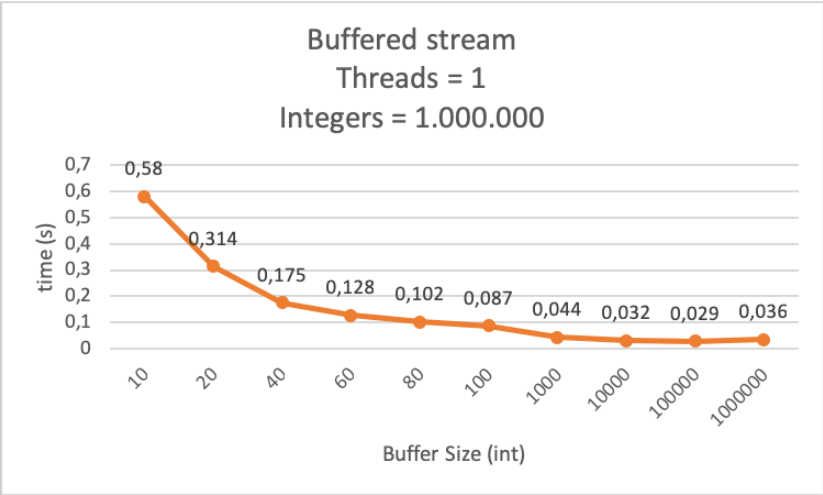

Figure 4: 1 Buffered stream with variable buffer size.

In figure 4 only 1 stream is used. It can be noted that increasing the buffer size, the performance increases almost steadily. This result is quite expected, increasing the buffer size and occupying more memory, the application should reduce the number of IOs.

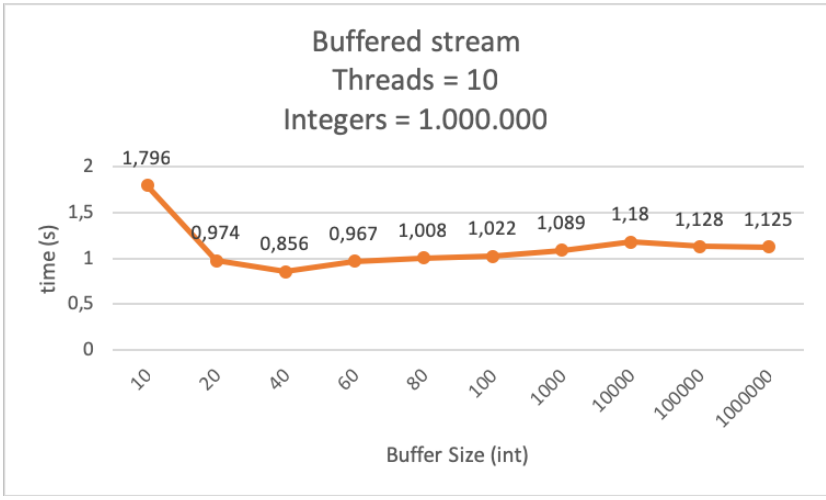

Figure 5: 10 Buffered streams with variable buffer size.

In figure 5 we implement the buffered stream with 10 threads. It can be noted that the performance reaches a peak around 40 integers as buffer size (360 bytes). We can think that having more parallel processes with large memory occupied result to be slower. In figure 6 we use 20 parallel buffered streams. It

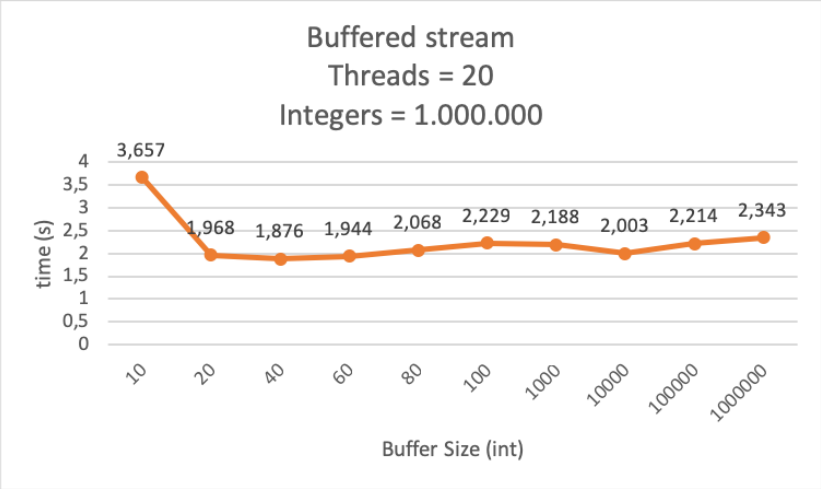

Figure 6: 20 Buffered streams with variable buffer size.

can be noted that performances become more unstable, but we still have the best execution time around 40 integers. We can assume this number is optimal for the usage of buffered streams in a parallel environment. In the next sections we test the performance of buffered streams on different number of threads. In that scenario, we use 40 as buffer size, being the best choice in average.

2.3.4 Initial mapping size for Memory Mapped stream

In this section we focus on memory mapped streams. In the previous section 2.1 we discussed the implementation of this particular stream. We decided to use a parameter for the Output stream, which identifies an initial size for the mapped memory region. Every time the limit is reached, a new portion is mapped with double size. In order to test how this implementation influences the performance, we execute the benchmark for memory mapped streams on 10 threads with variable mapped buffer size. We choose 10 threads in order to remain general and to maximize the effect of the initial mapped region.

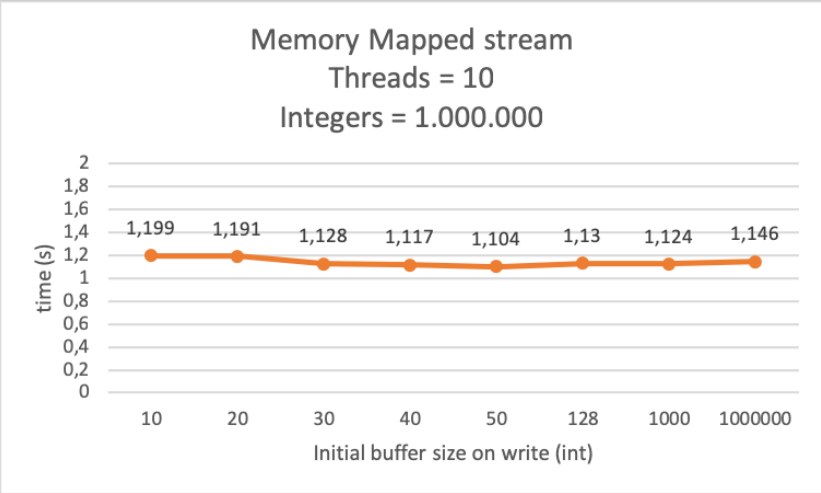

Figure 7: 10 Memory Mapped streams with variable initial mapped buffer.

Figure 7 shows how this variable slightly influences the performance. We plot the chart with the same vertical scale of 5, highlighting how the buffer size affects the two streams differently. This benchmark shows how memory mapped streams are practically independent of this implementation choice. In the next examples, we are considering 50 as initial mapped buffer size, since it performs slightly better.

2.3.5 Buffered and Memory Mapped streams

In this section we compare the performances of buffered and memory mapped streams, under different circumstances. Basically, we make comparisons on single and multi-thread environments, with variable number of integers.

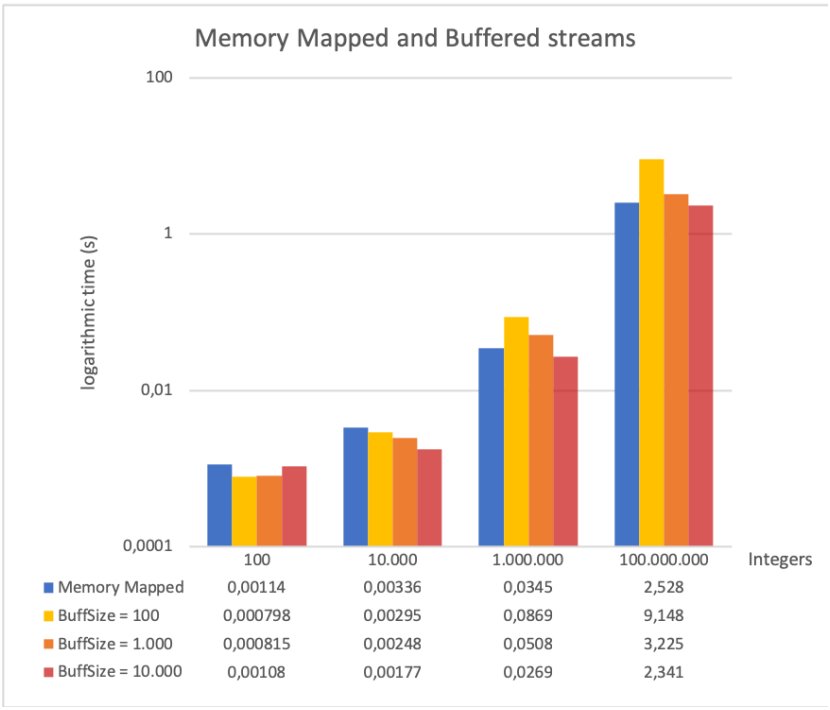

Figure 8: Single Buffered and Memory Mapped streams on variable number of integers.

Figure 8 shows how the performance of the two streams varies according to the number of integers considered and the buffer size for the buffered stream. Regarding the memory mapped stream, an initial size of 1000 integers has been chosen for the mapped buffer. Moreover, the vertical axis has been plotted in logarithmic scale, in order to allow a comparison between different execution times for different orders of magnitude of integers. In addition, the figure shows a table with the actual values, allowing also an absolute comparison.

Figure 8 leaves space for some conclusions. We can see how the memory mapped stream is not always the best solution. More precisely, the buffered stream with a buffer size of 10.000 integers (around 40kB) results to be the best option in most of the cases. However, when the file is small, the buffered stream with a smaller buffer size performs better. It has to be said that the buffered stream is a more flexible solution, but it requires to know the size of the files the application will work with. However, the memory mapped stream performs well for large files, without creating too much overhead (as 40kB of buffer). Indeed, the memory needed for the stream is in the kernel space, handled by the OS, leaving all the space free for the user.

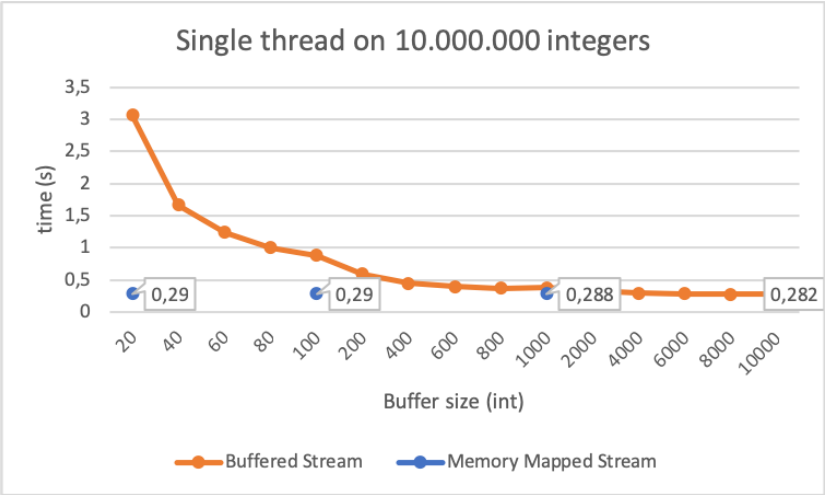

Figure 9: Single Buffered and Memory Mapped streams with variable buffer size.

Figure 9 clearly shows how the performance on 10.000.000 integers (around 40MB) does not depend on buffer size for memory mapped streams, while it strongly affects buffered ones. However, it can be noted, as in figure 8, that for large buffer sizes, the performances between the two streams can be considered as comparable.

Concluding the discussion about stream performances, we propose a final comparison between buffered and memory mapped streams with multiple threads. In our example, we use 100.000 integers and both 40 and 10.000 integers as buffer sizes for buffered streams. We discussed the first number in a previous section; we simply choose it in order to maintain an optimal performance with a large number of threads.

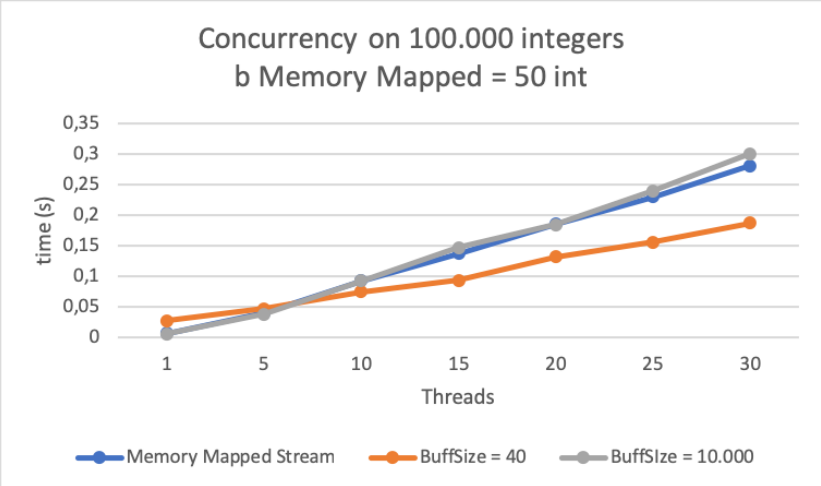

Figure 10: Multi-thread Buffered and Memory Mapped streams.

Figure 10 shows how the performance of memory mapped streams are affected in a multi-threaded environment. By increasing the number of parallel streams, buffered streams, with 40 integers as buffer size, outperform memory mapped ones. Moreover, the choice of buffer size has been conservative for a parallel environment. However, when comparing the two implementations with a single thread, memory mapped implementation occurs to be more performing. Finally, when considering buffered streams with a large buffer size, they maintain similar performance to memory mapped ones for both single and multiple threads. Again, the main drawback for large buffer sizes is the user memory overhead, while memory mapped streams leave it free.

2.4 Expected and actual behaviour

The comparison between actual and expected behaviour of these implementations is quite challenging. As expected, formula do not work on every conditions, but only for bounded values of buffer size and threads, according to each case. Moreover, the average time the computer takes to write a byte into the disk needs to be computed. With all this information in mind, in the current section, some examples of comparison are proposed. An endless number of charts could be created, but only a small and meaningful number of them are proposed.

As first operation, we need to compute the number of seconds the computer takes to read or write one byte. In order to obtain this result, we use the buffered stream with buffer size equals 1. Buffer size and N are specified as number integers. We start from empirical results of this implementation, specifically, execution times for different values of N and k and B fixed to 1. Through the formula:

$$time * 2 * \frac{B}{N} \tag{4}$$

the number of seconds for each byte can be computed. Using the same data of figure 1, we reach this different results:

expected IO sec per byte
0,00000640
0,00000615
0,00000560
0,00000548
0,00000586
0,00000541

Figure 11: Computation of seconds needed to read or write 1 byte.

For the next experiments, we can consider $6\mu s$ as seconds needed for reading or writing 1 byte.

Now, we apply the formula for different situations.

- **Simple streams:** they can be verified in figure 1, where we compare their behaviour with the one of buffered streams, using the theoretical formula. However, we need to compare the actual and expected behaviour of buffered streams.
- **Buffered streams:** In order to compare their behaviours, we perform an analysis where k is fixed to 1 and B is free to vary.

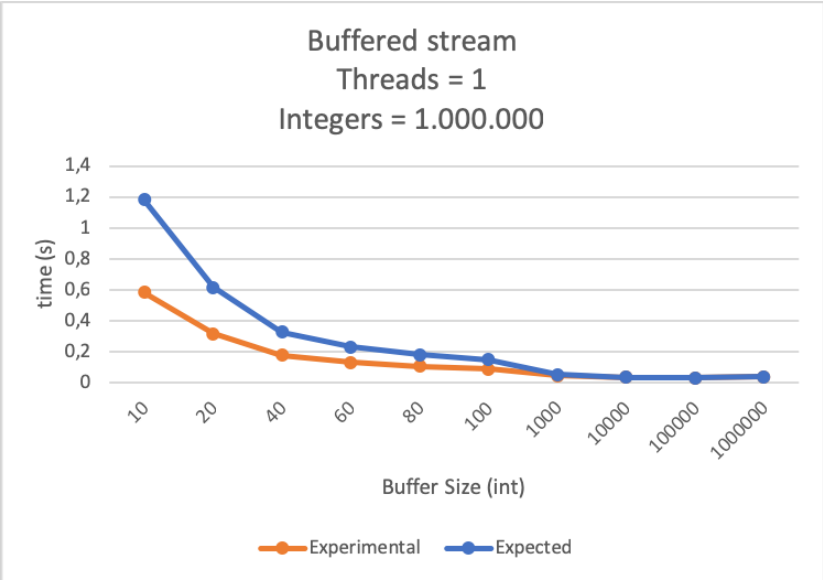

Figure 12: Comparison of buffered stream with varying buffer size.

Figure 12 shows how the behaviour of the two curve is quite similar. However, it is not shown in the image, when the buffer size increases above 1000, the behaviour drastically changes. While the actual one maintains more or less the same value, decreasing just a little, the one predicted by the formula is decreasing significantly. This was actually expected, because even if the buffer is huge, the system cannot perform one single IO operation for the whole buffer.

Moreover, regarding buffered streams, we compare their behaviour with different values of k maintaining B fixed to a couple of values. In the expected formula, k_{eff} is considered to be $\frac{k}{4}$.

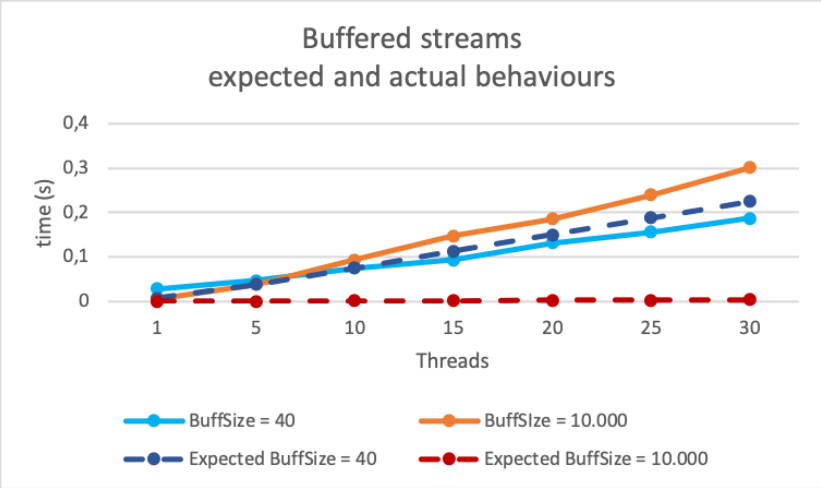

Figure 13: Comparison of buffered streams with varying number of threads.

Figure 13 shows how the actual behaviour follows the expected one only for small values of B. However, as expected, when B increases, the actual behaviour does not follow the expected one. A possible reason for such curve is that the memory available for each thread is limited. Consequently, loading a large portion of the file at once may force the thread to repeat this operation more than once due to context switching. While the control is passed to another thread, this portion of memory may be reused for loading a portion of another file.

- **Memory Mapped streams:** In order to verify their behaviour, we start by modifying the number of threads for some large fixed value of N. We can already tell, from previous charts, that the actual behaviour of memory mapped streams for small files does not follow the expected formula. k_eff is again considered as $\frac{k}{4}$.

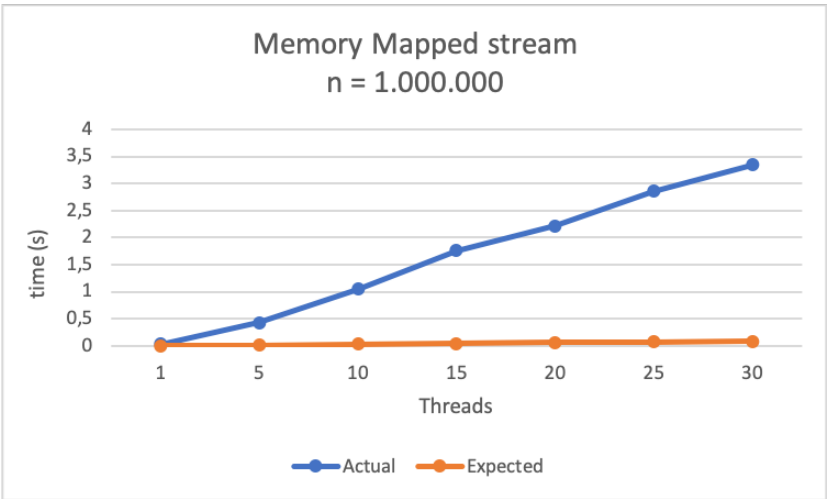

Figure 14: Comparison of memory mapped streams with varying number of threads.

As it can be shown from figure 14, the actual behaviour differs a lot from the expected one. It can be compared to the error of the curve of figure 13, with the largest buffer size. This behaviour could have the same explanation, if too much memory is allocated, it then must be released to make space for other processes.

As last comparison, we compare the general behaviour of Buffered and Memory mapped streams with a varying number of integers (N).

N	Memory Mapped	Expected time	Buffer 100	Expected	Buffer 1.000	Expected	Buffer 10.000	Expected
100	0,00114	1,17188E-06	0,000798	0,000012	0,000815	0,0000012	0,00108	0,00000012
1000	0,00106	1,17188E-05	0,000956	0,00012	0,000938	0,000012	0,00106	0,0000012
10000	0,00336	0,000117188	0,00295	0,0012	0,00248	0,00012	0,00177	0,000012
100000	0,00844	0,001171875	0,0138	0,012	0,00822	0,0012	0,00627	0,00012
1000000	0,0345	0,01171875	0,0869	0,12	0,0508	0,012	0,0269	0,0012
10000000	0,304	0,1171875	0,883	1,2	0,408	0,12	0,277	0,012
100000000	2,528	1,171875	9,148	12	3,225	1,2	2,341	0,12

Figure 15: Comparison of memory mapped and buffered streams with varying number of integers.

Figure 15 shows an expanded table of chart 8. No chart is plotted because data is so different that the error could not be seen. The general behaviour is still present in figure 8. The table shows how the two execution times are so different for small values of N. However, when the value of N increases to 1 million elements (around 4MB) the same order of magnitude is reached. Moreover, it can be shown that for the last three values of N, from 4MB to 400MB, the actual execution time increases by a factor of 10 at each step. This

behaviour maps the expected one in the formula, where the execution time is directly proportional to the number of integers N.

Concluding this chapter, in order to implement a multi-way merge sort algorithm, we need to be as much flexible as possible. In order to obtain the best performance, handling large files, we choose to use memory mapped streams.

3 Observations on multi-way merge sort

The implemented algorithm starts by creating a temporary folder where all the intermediate files are stored. Afterward, a function called *splitInputStream* splits the input file in $\lceil N/M \rceil$ intermediate files. This function takes care of not to read more than M integers from the input file into memory. Then, the file name of each file is kept in memory on a list and use it in a loop until the list is empty. In each iteration d files are pulled out the list and sorted in one file. The reference of the sorted file is pushed back in the list. This process is repeated until just one sorted file is left on the list. All intermediate files are deleted and the temp folder too.

The sorting algorithm is *DWayMergeSort*. The idea is to read smallest values from all the streams until all the streams are emptied. We have created a data structure that contains the stream itself and the latest read value from that stream. We have implemented a *Comparable* interface on this data structure in order to sort the streams based on latest read value from that stream. We have stored these blocks in a priority queue and the queue sorts them automatically. Since, all the streams are sorted based on the latest read value in priority queue, when we read from the head of the queue we always read the smallest value. The value which is pulled out the queue is written in secondary memory using the output stream. This way we ensure that no more than d values are kept on memory at any given time. This process goes on until no new value can be read of any file in the queue. The algorithm ends up writing in secondary memory all the values left on the queue in a sorted manner.

Please note that, we have considered M independent from size of the meta data i.e. stream references, file names, file objects, file buffers etc.

3.1 Expected behavior

We analyzed the theoretical expected behavior for each of the different Input and Output stream mechanisms with the external memory merge sort. Starting from the simple mechanism of read and write one element at a time, we estimated the following cost function:

$$F_1 = 2 * N * log_d \lceil N/M \rceil \tag{5}$$

The 2 factor in front is because we need to read and write and both have the same cost. More importantly, we can notice that because we are reading and writing just one integer at the time the predominant factor is N itself, while the remaining of the formula is just counting how many times we need to read or

write N. So, we can predict that this cost function can improve increasing the number of integer read at a time. Precisely, this is the case for the buffer-based mechanism (implementation 2 and 3) and memory mapping. On one hand, in implementation 2 the buffer size by default is 8192 bytes which is about 2048 integers at a time. On the other hand, for memory mapping some literature suggest that the default number of Kb the kernels loads into memory from the file is 4Kb on each page fault operation. Finally, taking this into account and without losing generalization we use B to denote the number of integers loaded on every I/O operation, the formula is presented next:

$$F_2 = 2 * \frac{N}{B} * log_d \lceil N/M \rceil \tag{6}$$

Now we will focus our attention in the right part of the formula. It can be noticed that with this formula small changes of M will not affect the number of I/O operations keeping the base of the logarithm constant. However, we can predict a great improvement of the execution time increasing M considerably every time. Also, from this formula we can predict that increasing d would improve the execution time as well but with a lesser rate of change. In the next section we run several experiments in order to validate these hypotheses.

3.2 Experimental observations

We ran the benchmark using Memory Map stream implementation using only one thread[1]. This decision was supported by the results achieved by this implementation in the first section of the project. Taking this into account we only focus on playing with different values of values of N, M and d while B is handle internally by the kernel. Also, as we predicted, small changes of M will not affect the execution time of the algorithm so we decide to change the values using power of two. This way we also help the loading of memory in the OS. Finally, we took into account some constraints while running the benchmark i.e.

- $M \leq N$
- $d \leq M$
- $d \leq \lceil \frac{N}{M} \rceil$

There is no point in running for values of M greater than N. It would not make a difference. Same goes for the other two constraints. We ran the experiments on different machines with different specifications. Here are the specifications of the machines:

1. Machine 1
    - Dell Inspiron Laptop

[1] All the experiments were done using two warm up iterations and three real iterations from which the average was the final results plotted

- CPU: Intel(R) Core(TM) i7-4500U CPU @ 1.80GHz, dual core, 4MB cache size
- OS: Ubuntu 18.04 LTS
- Hard disk: SSD with NTFS partition
- RAM: 16GB
- Swap: 16GB
- Java: OpenJDK 1.8.0

2. Machine 2

- HP
- CPU: AMD A12-9720P RADEON R7
- OS: Ubuntu 18.04 LTS
- Hard disk: SSD
- RAM: 8GB
- Swap: 8GB
- Java: OpenJDK 1.8.0

First, we ran the tests on machine 1. The results did not make any sense. There were a lot of anomalies and it took a large amount of time to run tests on this machine. There were a lot of out of memory issues even if there was a lot of memory available. There were errors in committing memory. It turned out that it is a bug in default java agent of OpenJDK. Then, we use IntelliJ IDEA to run the experiments. However, it kept taking a lot of time to run the benchmarks on this machine and several days were lost. Finally, we realized that the emulation of NTFS on Ubuntu was the culprit behind it and we decided to change the PC.

On the second machine, we didn't get these kind of errors as we were using java agent from IntelliJ and the partition type was ext4. All of the following results mentioned were taking for the second machine.

3.2.1 Experimental observations

During this section, we perform several experiments, modifying the value of the available memory and the number of streams to merge at each step. We refer to these parameters as M and d respectively.

In this first experiment we kept N fixed to $8MB$ while M and d varied from $M = [2^6, ..., 2^{12}]$ and $d = [2^4, ..., 2^{10}]$ the results are show in figure 16.

It can be observed that while increasing the size of memory available by two we also have a decrease of almost the double of the execution time. Comparing this result with the expected behaviour using the cost function 6, we get a clearly different slope. However, we can assume that memory map mechanism is faster when less file descriptors are opened. Assuming this, this behaviour makes sense because increasing M we decrease the number of intermediate files.

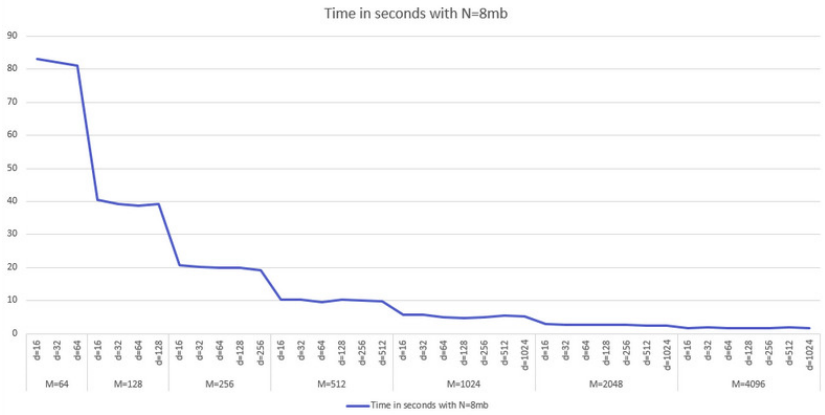

Figure 16: Execution time over 8MB, moving M and d

On the other hand, we can observe a slow decay of the execution time while increasing d. Although this behaviour was expected, it is degraded when d is close to $\lceil N/M \rceil$. Once again we think this behaviour is due to the memory map mechanism.

In order to validate this result we ran the experiments over a bigger file. This time we used $N = 64MB$ and different values of $M = [2^{10}, ..., 2^{15}]$ and $d = [2^8, ..., 2^{10}]$ to reduce waiting time. The goal of this experiment is check if the past results was due to the size of the file our is a general trend. Figure 17 shows the execution time.

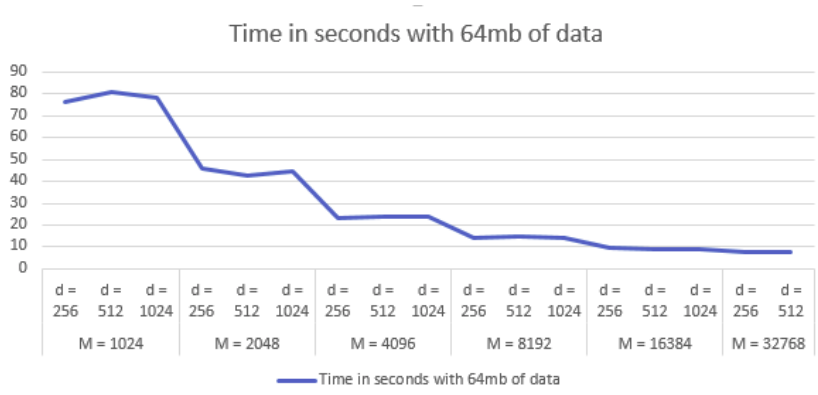

Figure 17: Execution time over 64MB, moving M and d

This trend is actually quite similar to the expected one. Once again, we see an exponential decrease each time we increase M by two and the same behaviour when d is close to $\lceil N/M \rceil$. We also try with other small values of N and the behaviour was the same. Taking this into account, we can conclude

that using memory mapping for small values of N we have an exponential decay of the execution time while increasing the size of M. Moreover, we can find a meticulous decay when increasing d although this trend is not hold for large values of d. In the next section we present some results using bigger files.

3.2.2 Results over big files

For this experiments we ran the algorithm over $N = 500MB$ and $N = 1GB$. We must highlight that although we tried to do the same experiments as before, when running over $1GB$ the waiting time executing all possible combination exceeded the day of waiting. Moreover, for most of the trials, the IDE was crashing. Hence, we only present results for 1GB moving the value of M while keeping d fixed. Figure 18 shows the results for $N = 500MB$.

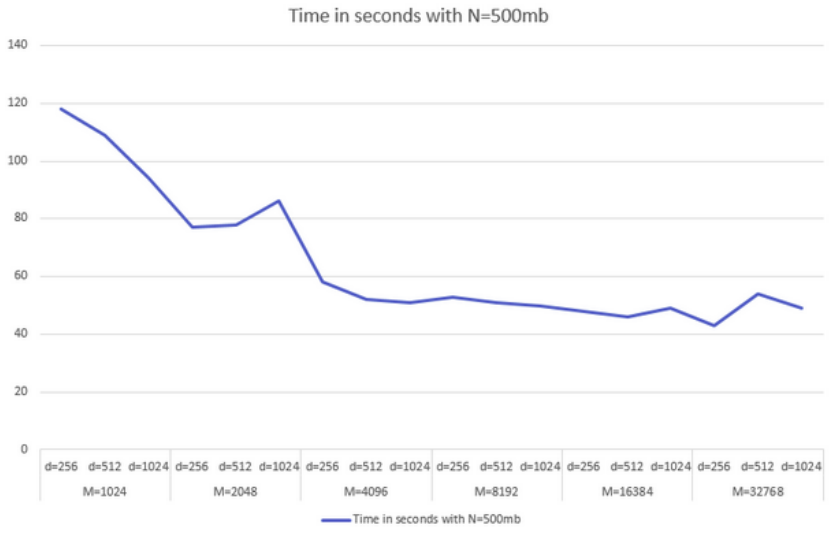

Figure 18: Execution time over 500MB, moving M and d.

It can be noted that this time the behaviour was not the same. In this case, we can see how the curve stays constant while M is getting higher and even we can see a slight degradation at the end. We can also notice how d loses weight and the results do not really change considerably while increasing d.

Previous results leaded us to the idea that when M is too big the performance will decay. Maybe a local minimum could be found and some kind of criteria used to find the value of M. Taking this into account we ran again the algorithm over $N = 1GB$ in order to validate this idea, this time only paying attention to M. Also, we tried with bigger values of M expecting a degradation of the performance. Figure 19 shows the results.

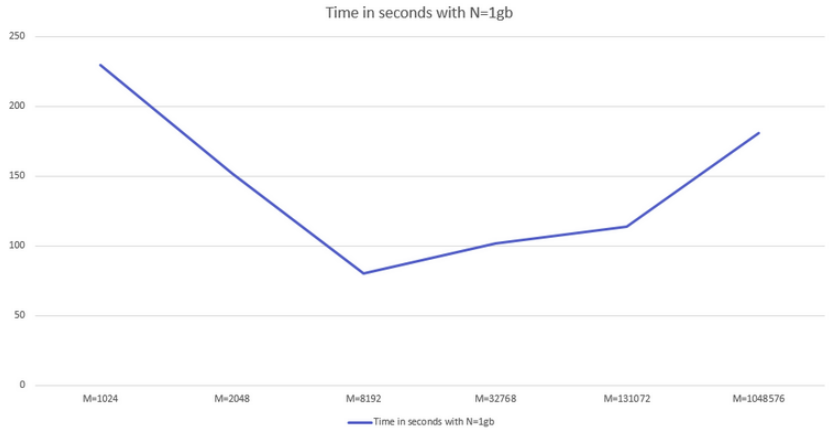

Figure 19: Execution time in seconds for 1GB, varying M

It can be noted that the execution time decreases every time we increase the value of M until we reach 2^{15}. At this point the JVM runs out of memory and it starts doing Garbage Collection (GC). The GC executions are mostly stop-the-world executions. This could explain the sudden increase in the execution time for large value of M.

As a conclusion for these experiments, there were few discrepancies in the expected and actual behaviour. We did not expected the execution time dropping exponentially while increasing M this only could be possible due to the memory mapping mechanism. Taking this into account, as a rule of thumb we recommend to use d lower than $\lceil N/M \rceil$ because in all cases it turned out to be the most optimal one. On the other hand for the value of M the larger the better. However, the amount of available memory should be taken into account otherwise the operating system will start swapping and it will also hinder the performance.

3.2.3 Comparison of In-Memory and External Memory Merge Sort

In order to compare our algorithm with an in-memory sorting algorithm we set up a different configuration. On the first place, we changed the PC once again for one with more computing power and memory. This way we could avoid the problem detected before when N is too big. In this case we used the same PC used for benchmarking stream operations in last section. On the second place, this time M and d where fixed and we modified only $N = [2^18, ..., 2^24]$. For in memory sorting we used heap sort and run both algorithms over the same file. The cost function for I/O operations is approximately the number of times we are reading and writing the integer values.

$$F_in = 2 * N \tag{7}$$

The figure 20 shows the comparison between in-memory and external memory merge sort. For multiway merge sort we used M=$2^(12)$ and d=2^6.

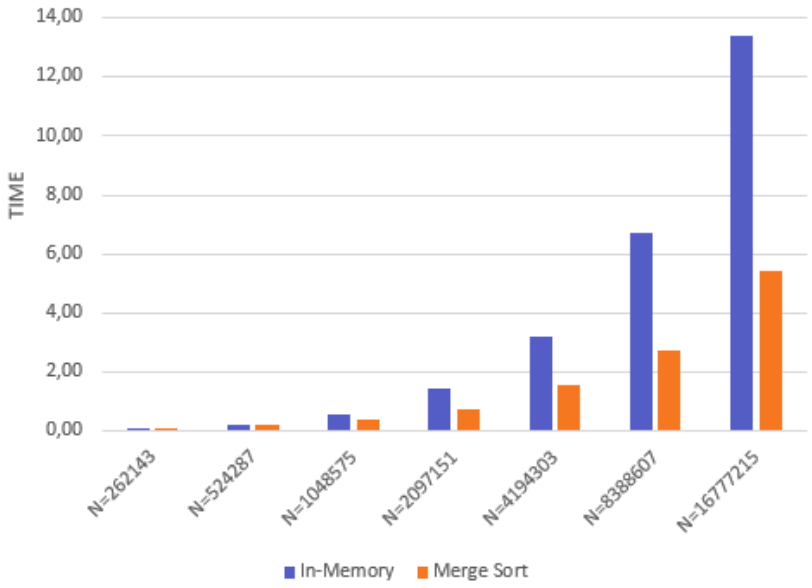

Figure 20: Comparison between in memory algorithm and the algorithm implemented for different file size.

Notice how the implemented algorithm outperform in memory sorting for an increasing margin as N keeps growing. This result was quite surprising for us as the expected behaviour is that in memory sorting outperforms the implemented algorithm. A possible explanation is that we performed internal memory merge sort first and then we performed external memory merge sort within the same jvm. So, the decrease in external might be due to the fact that OS caches the file and we are reusing it in external memory merge sort. Then, we ran again the experiments this time changing the order of execution of each algorithm. However, the results was the same. We assume that because we do not read the file completed but instead small files and sort them on each iterations Memory Mapping mechanism is the cause of this bizarre behaviour. Further experiments need to be done to valid this results.

4 Conclusion

In this project we experimented with different implementations to read and write on secondary memory. We learned how to properly implement a stream object and how to choose among different implementations according to the scenario. Moreover, we used them to implement a more complicated algorithm, which strictly depends on the chosen criteria to handle secondary memory operations. Moreover, we tested the multi-way merge sort algorithm using different parameters and we clearly understood how these values largely affect its performances.

The project allowed us to implement several concepts discussed in class, giving us a practical insight of how effectively databases handle disk operations at low level.