UNIVERSITÉ LIBRE DE BRUXELLES ULB

INFO-H-417 - Database Systems Architecture
Project Assignment: External Memory Algorithms

Lev Denisov	000454497
Ferdiansyah Dolot	000455509
Ivan Putera Masli	000456536

January 6, 2018

Contents

List of Figures

List of Tables

Introduction

The goal of this project is to measure the performance of different IO algorithms in order to find the optimal algorithm and parameters for the external merge sort. We test 4 algorithms for reading and writing 32-bit integers and a number of parameters. The machine used for tests is Amazon Web Services m3.large instance with 2 vCPUs, 7.5 GB RAM and 60 GB provisioned disk (SSD). The implementation of the algorithms and the benchmarking code was written in Java 8, we used a micro-benchmarking library JMH [1]. We generate random 32-bit integers with the code given in the listing 1.

```
1 ThreadLocalRandom.current().nextInt(Integer.MIN_VALUE, Integer.MAX_VALUE)
```

Listing 1: Generation of random integers

For writes we generate random data as we write it. For reads and merge sort we generate data before the benchmark and put it in the pre-defined files which are then used throughout the benchmarking. For read and write benchmarking we used streams of size $N = 25000000$ and 10000000 of 32-bit integers which corresponds to 100 MB and 40 MB files per stream. In case number of streams k in the benchmark more than one, the total size written/read is number of streams times file size

$$Total = N * 4\,bytes * k \tag{1}$$

For the merge sort benchmark we used $N = 250000000, 100000000$ 32-bit integers which corresponds to 1 GB and 400 MB files.

1 Streams Experiment

In this section we experiment with 4 streams implementations for reads and writes. The cost formulas used are oversimplified and do not correspond to the real situation but we have not found one agreed cost model for SSDs. The current cost formulas take into account only the number of IOs neglecting the transmission time. While this model works for HDDs, since seek time dominates the costs, in case of SSDs this model might be inadequate and transmission time may be considerable part of read/write latency.

1.1 Stream1 - Unbuffered IO

This is the naive implementation of IO stream which reads and writes from disk every time read/write is called. We expect it to have the lowest performance since it should make N calls to disk to read/write N integers.

$$Cost = k * N \tag{2}$$

1.2 Stream2 - Buffered stream with the default buffer

This is a standard Java implementation of buffered IO stream. It uses the buffer size of $B = 8\,KB$. Buffering means that all the writes and reads go through buffer, which allows to decrease the frequency of real IO calls to the disk. The expected cost formula is:

$$Cost = k * N/B,\ B = 8192 \tag{3}$$

1.3 Stream3 - Buffered stream with the custom buffer

This is a modification of Java's default buffered stream that uses custom buffer size. The expected cost formula is:

$$Cost = k * N/B \tag{4}$$

1.4 Stream4 - Memory-mapped file

Memory Mapping is a method for mapping files to the memory address of the process. The process can access those files directly in their address space.

Memory mapped file can be described as: "File mapping is the process of mapping the disk sectors of a file into the virtual memory space of a process. Once mapped, your application accesses the file as if it were entirely resident in memory. As you read data from the mapped file pointer, the kernel pages in the appropriate data and returns it to your application." [2]. In theory, this method is very effective since it avoids creation of buffers and has less overhead on system calls. The implementation of this stream assumes that only portion of size B of the file is mapped into memory and once the stream goes outside this portion, the next portion is mapped. There are few optimization made in the implementation of this stream. First, the initial file size is always at least 1 MB. This allows to not waste a lot of time on reallocation of file in the very beginning. The best performance can be reached by allocating the final size of the file right away, but to keep the stream usable in case the final size is not known we implemented the strategy of

doubling the file size when its capacity is reached. This allows to keep reallocations rare. The expected cost formula, depending on the buffer size B, is:

$$Cost = k * N/B \tag{5}$$

1.5 Benchmarking Parameters

For all streams these parameters will be used

- k − Number of streams used and number of input files: $k = 1, 5, 10, 15, 20$. We decided not to go higher than 20 because it takes considerable time and does not seem to give more information about the behavior of the system.
- N − Number of operations: $N = 25000000, 10000000$. These numbers are chosen to be high enough to represent real life situations but also with the intention to keep benchmark times relatively low. Even with these values the benchmark takes several hours to finish.
- B − Buffer size for streams 3 and 4: $B = 8096, 16384, 4194304$. We have chosen B to be multiples of 4096, since it is a common size. We use $B = 8096$ to compare the performance of memory-mapped files with the default Java's buffered stream. The value of $B = 8096$ was chosen by mistake, the intention was to test $B = 8192$; on the other hand, this value shows how the buffer size which is not the multiple of 4096 affects the performance. $B = 4194304$ represents relatively large buffer size. It might be interesting to test very large buffer sizes like 1 GB but it would require to use larger files that cannot fit into the buffer of this size. The limiting factors in this case are disk size, RAM size and time, since running tests with multiple streams will require k times more resources. Also, to have the fair comparison of the algorithms we would have to run benchmark with the files of this size for all the streams which in case of unbuffered stream may take very long time.

We will use the *Throughput* as the metric for this benchmark.

$$Throughput = \frac{Total}{time} \tag{6}$$

1.6 Experimental Observation

1.6.1 Output

The figure 1 shows write throughput of different streams on 100 MB files in dependence to number of streams k. It is clear that unbuffered stream (Stream1) has by far the worst performance. Also, we were not able to collect all the observations for memory-mapped Stream4 since on small buffer sizes it kept failing with error "Native memory allocation (mmap) failed to map X bytes for committing reserved memory.". It seems to work stable on bigger buffer size.

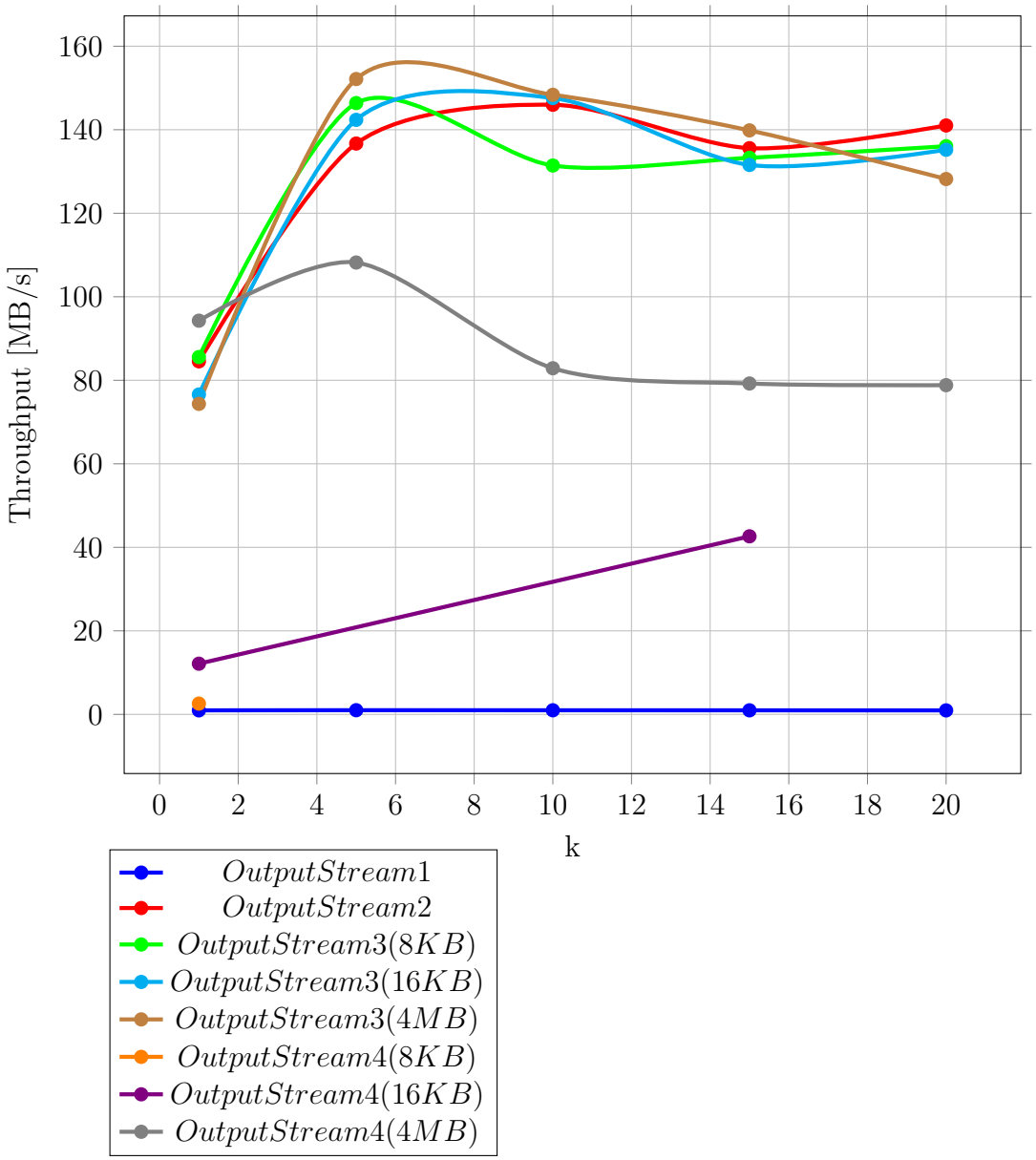

Figure 1: Output streams, 100 MB files

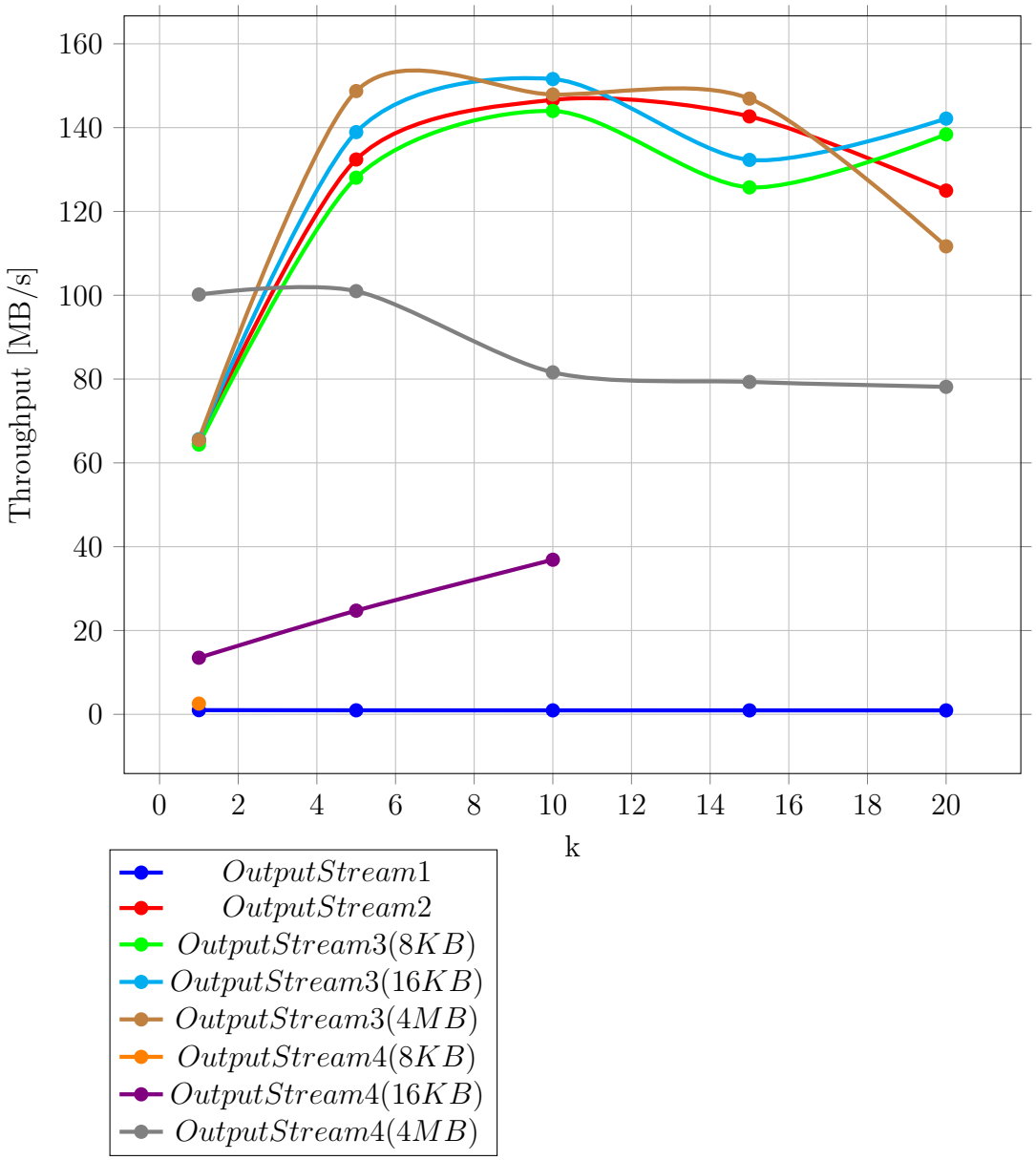

Figure 2: Output streams, 40 MB files

The figure 2 shows the write throughput on 40 MB files. There seems to be no difference between throughput on different file sizes. We can see that buffered stream3 with buffer size of 4 MB has slightly better write performance than the rest of streams.

We can see that the peak write throughput is around 150 MB/s which seems plausible for the modern SSD.

1.6.2 Input

The figures 3 and 4 show read throughput on files 100 MB and 40 MB. As in case of writes, the unbuffered stream has the worst performance. We also did not collect enough data for memory-mapped stream with small buffer sizes due to runtime errors mentioned above. There is no difference in read throughput on different file sizes. From all stable algorithms memory-mapped stream4 with buffer size of 4 MB has by far the best performance.

We can see that the peak read throughput is around 650 MB/s which seems plausible for the modern SSD.

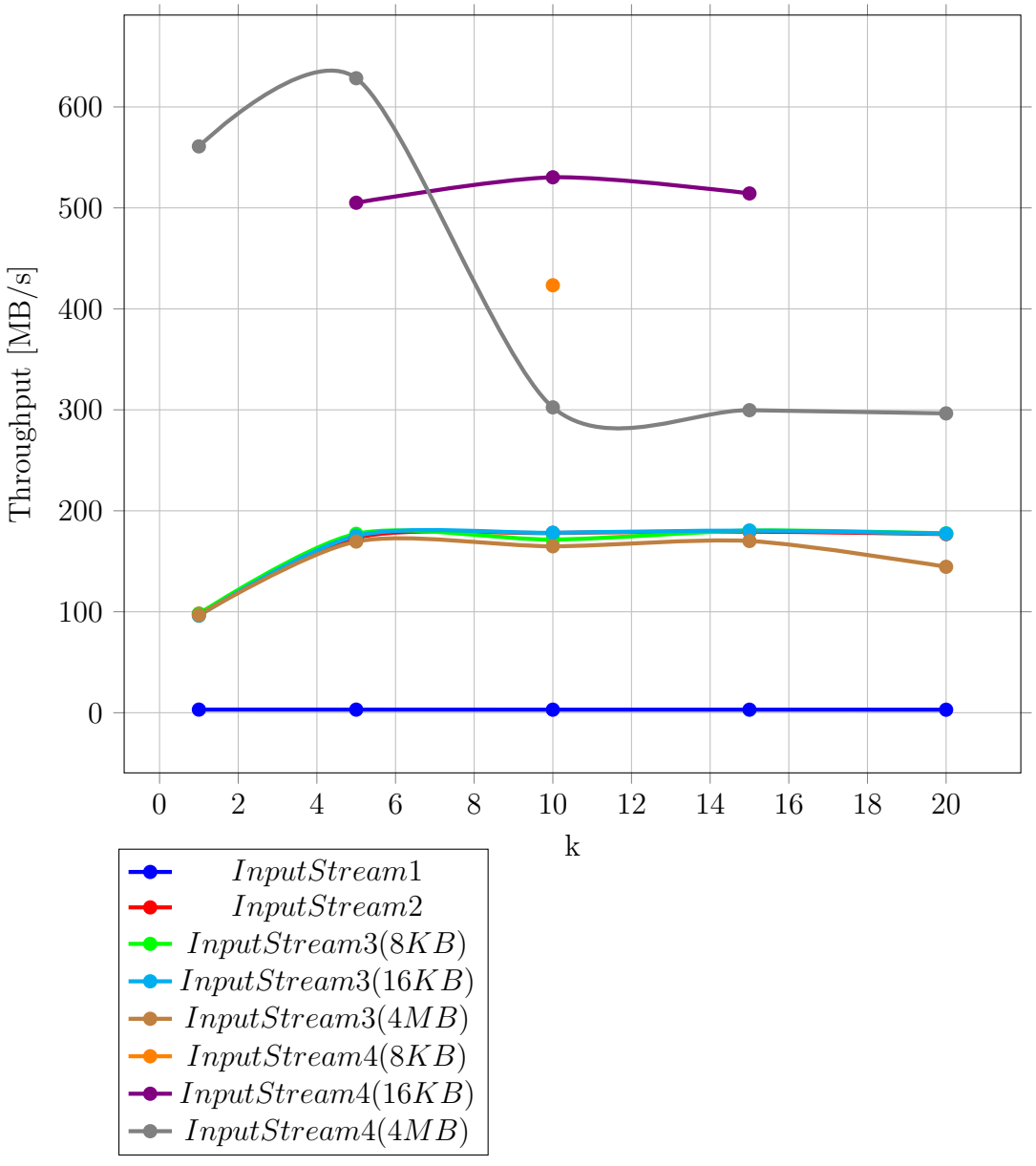

Figure 3: Input streams, 100 MB files

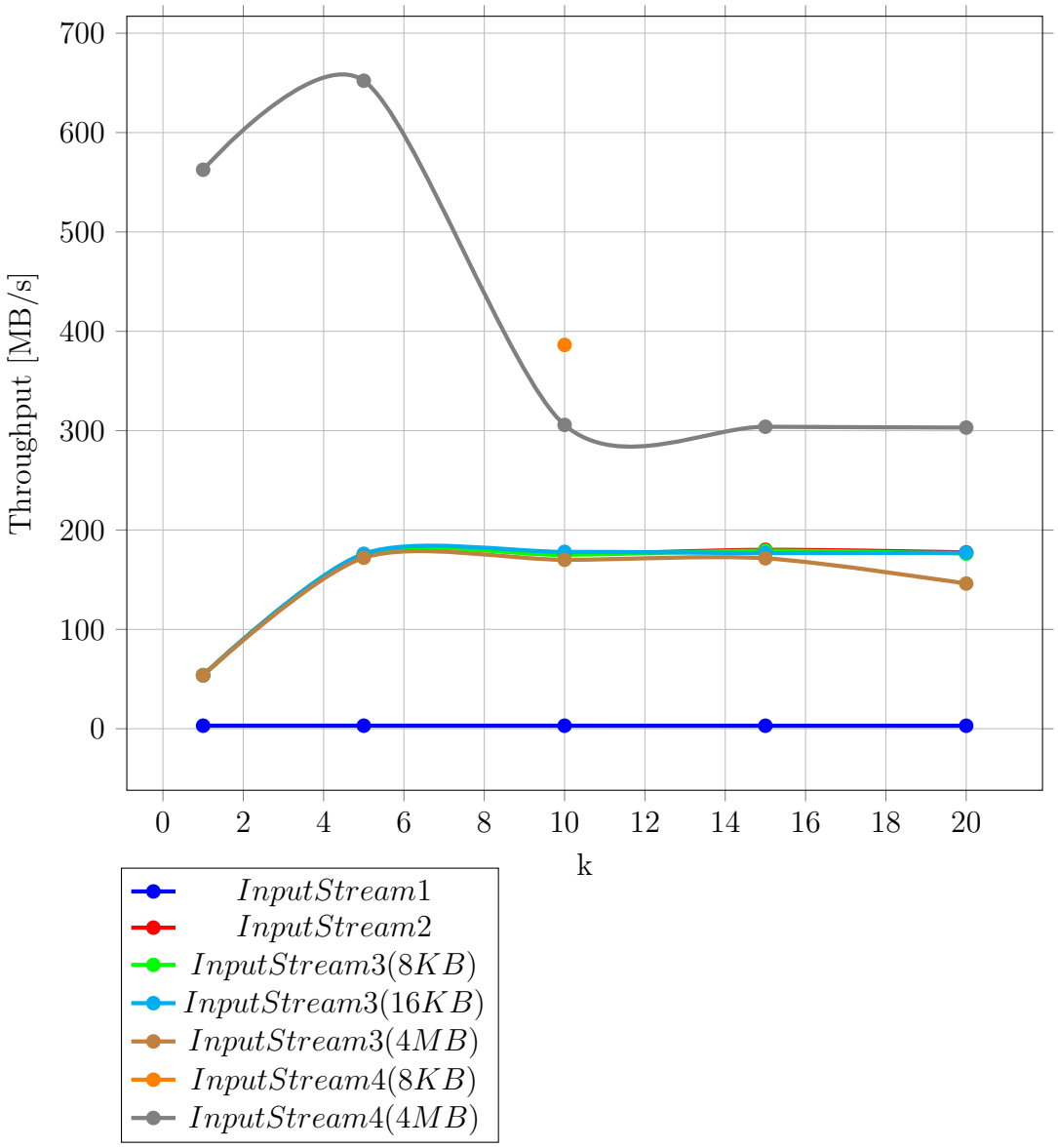

Figure 4: Input streams, 40 MB files

1.7 Evaluation

We can see that unbuffered stream1 has indeed the worst performance. The other streams despite having the same theoretical cost have different performance. First of all, as it was mentioned in the section 1, the cost model does not represent the real situation for SSDs, so the increase in buffer size does not provide big increase in performance.

We can see that buffered streams 2 and 3 have better write performance than memory-mapped stream4. Our hypothesis is that memory mapping has first read part of disk into memory and it makes it slower than buffered output that can do writes directly. Also, the sequential workload seems to be not the best case for memory-mapped files as it would probably perform better then buffered streams on random writes.

One thing to notice is that write throughput grows with number of streams at first but after some point, additional streams hurt the performance. It seems to happen at around 10 simultaneous streams. Since the disk has limited speed, after reaching it the additional streams cannot make throughput bigger but the overhead on these streams

may have negative effect on the performance.

We can see that memory-mapped stream4 with 4 MB buffer has the best performance on read workload. This seems to correspond to lower overhead on direct memory-mapping in comparison to buffer management.

Having more than 5 simultaneous memory-mapped streams seems to negatively affect read performance.

2 Multiway Merge Sort Experiment

In multiway merge sort experiment, we are interested in experimenting with external sort performance with various memory size limit and number of streams to merge in one pass. As we have known from the project description, in order to merge streams, we are required to open multiple streams in parallel. Hence, we will use the best configuration obtained during previous experiment of input and output streams.

2.1 Expected Behavior

Before running experiment, we set several baseline or expected behaviors on choosing current configuration as parameter of multiway merge sort experiment.

- N − Size of input files.
  It is likely that the larger size of input files, the longer time execution will take.

- M − Size of available memory during sort phase.
  In multi way merge sort, size of memory could contribute to processing time. Larger memory could produce less number of files, hence number of streams to merge and number of merge pass will also be smaller. Hence, we expect larger memory to perform better than smaller amount of memory because small to medium (5) number of streams seem to be optimal case from previous streams experiment.

- d − Number of streams to merge in one pass.
  As we have already discussed in the description of M parameter, having the right value of d could help increasing the throughput, and reducing execution time. Splitting the load into several streams could help, but the result will vary on how large is the initial file, how much memory M is available, and what is the optimal size of d. We will expect small to medium number of streams (around 5) to deliver the best performance.

Next section will discuss more detail about benchmarking parameters and how those combinations of parameters could impact the performance by observing convergence values of run time execution.

2.2 Benchmarking Parameters

Multi way merge benchmarking use the best configuration of input and output streams as follows:

	Streams Type	**B**
Input Streams	Stream 3	4MB
Output Streams	Stream 4	4MB

Table 1: Benchmarking Parameter: Input and Output Streams

We also use big and small values of N and M in order to observe behavior and performance of external multi way merge sort (table 2 and 3).

N	**Values**	**File size**
Small	100000000	400 MB
Large	250000000	1 GB

Table 2: Benchmarking Parameter: N

M	**Values**
Small	4MB
Medium	6MB
Large	12 MB

Table 3: Benchmarking Parameter: M

Meanwhile, values of d are derived from previous configuration of input and output streams, since during merge phase, we will try to see behavior of opening multiple streams align with value of d (table 4).

d	**Values**
Small	2
Small	5
Medium	10
Medium	15
Large	20

Table 4: Benchmarking Parameter: d

Multi way merge sort limit memory usage to M and number of streams to merge in one pass to d, hence estimation cost of the algorithm performed in file of N integers will also depend on M and d.

At first, there will be N/M number of unsorted files with size at most M. Then sort is performed on each files, and merge into one sorted file for at most d streams within one pass.

Number of files in i-th pass: $P_i = \frac{N}{M^i}; i = 1, ..., \lceil \log_M N \rceil$.

$$Cost = 2 * N * \left(\sum_{i=1}^{\lceil \log_M N \rceil} \frac{P_i}{min(d, \frac{N}{M^i})} \right)$$

In cost formula above, 2 is required for read and write streams. Within each pass, we need to calculate number of IO calls depending on parameter N, P_i and d. In case of d is larger than number of current pass (N/M^i), we will use number of current pass value.

Please note that the cost formula presented is only an estimation and has already been simplified in order to easily evaluate parameter dependencies. Details such as merge across one pass to another is not considered, since it will not increase the order of magnitude of cost evaluation and could be reasonably ignored.

2.3 Experimental Observation

Figure 5, and figure 6 summarize overall runtime execution of multi way merge sort performance. We will explain and analyze the result of this experiment on next section.

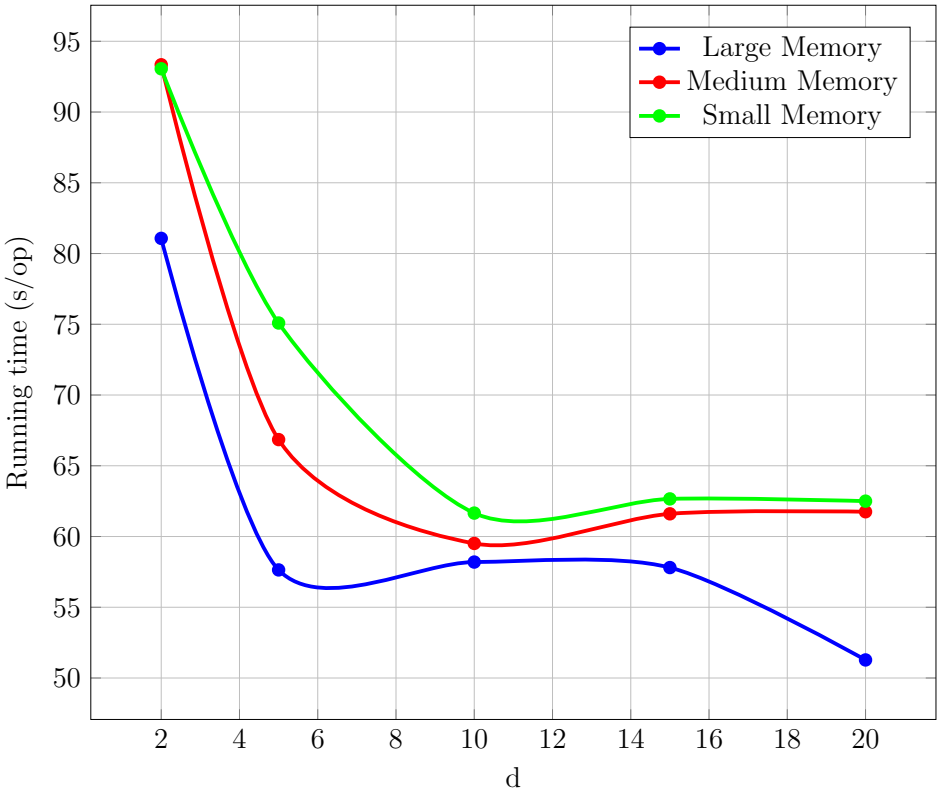

Figure 5: Benchmarking: Multi way merge sort in big file

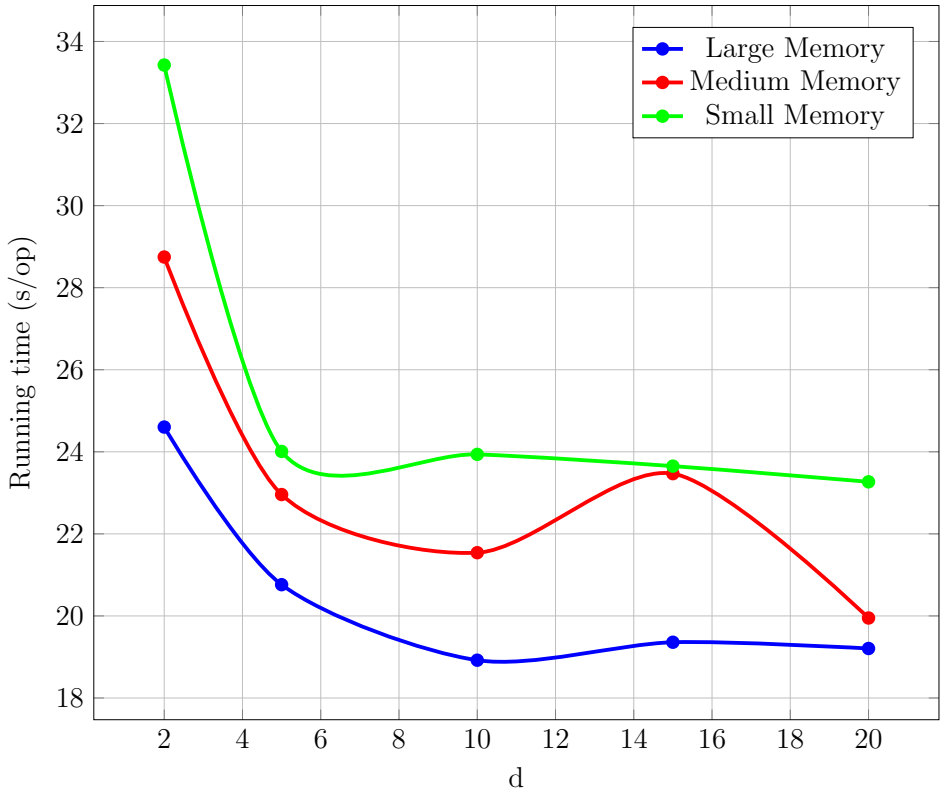

Figure 6: Benchmarking: Multi way merge sort in small file

We also interested to see comparison between the best performed external merge sort and in memory sort. Please refer to figure 7 for reference.

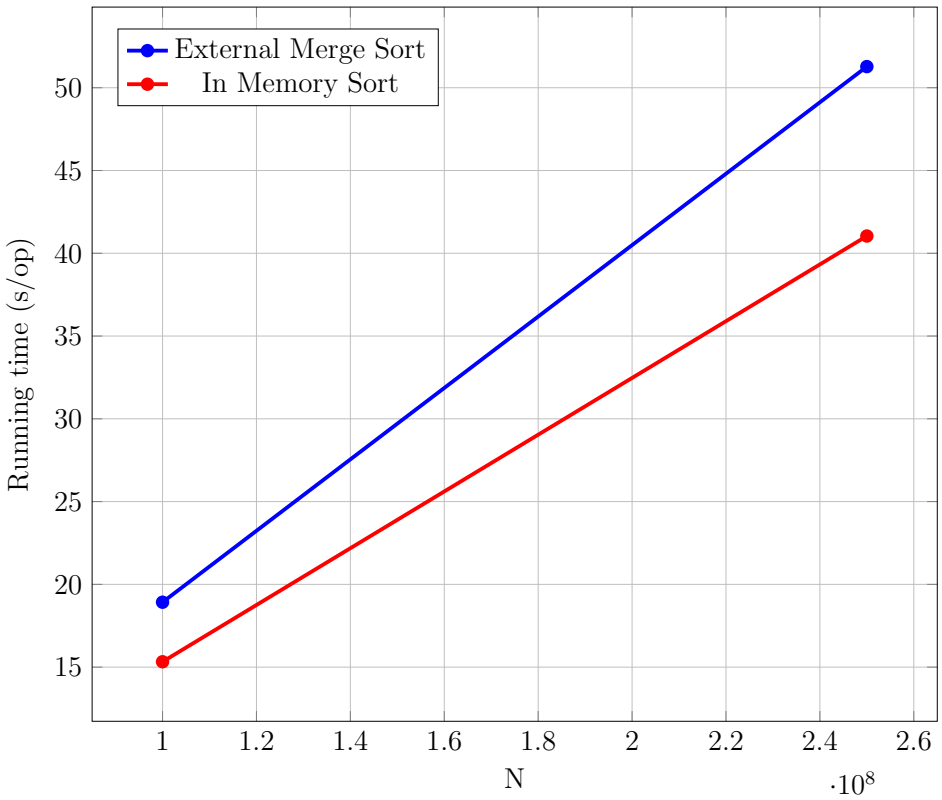

Figure 7: Benchmarking: Comparison between external and in memory sort merge sort

2.4 Evaluation

- External merge sort algorithm tends to perform better on small to medium size of d (please refer to figure 5 and 6).
- The case when $N = 1$GB and $M = 12$MB will produce 19 streams to be merged, so with $d = 20$, it will produce one pass merge and boost up the performance. We consider this as an exception.
- By observing the runtime (s/op) and amount of M, we can conclude that larger M can deliver better performance either on big or small files, by seeing on figure 5, and figure 6.
- In memory sort performs better than best execution of external merge sort in terms of run time execution.

3 Overall Conclusion

In this project we have reached a few conclusions:

- Unbuffered stream IO is very slow and should never be used.
- For sequential writes, buffered stream seems to be the best.
- Memory-mapped files are not ideal for sequential writes. The overall performance seems to be worse than one of the buffered stream. Frequent runtime errors on smaller map sizes make it harder to use it. Perhaps, by improving error handling and setting the right parameters of the machine this problem can be solved. It can probably perform best on random writes with the whole file mapped to memory.
- Memory-mapped files perform better than stream IO in sequential read workload. With enough precautions and error handling it can be used instead of stream IO for reads.
- There seems to be almost no effect of buffer size on the performance of the standard IO streams. This may be the case for SSDs and the situation is different for HDDs.
- The increase in number of simultaneous streams seems to positively affect the performance of reads/writes but only up to a certain point, in our case it is 5 simultaneous streams. After that point, further increase in the number of streams seems to have no effect or even negative effect on the performance.
- In our tests we have seen no effect of the file size on the IO performance. Probably, this effect may appear on much larger files that were not researched in this project.
- For external merge sort algorithm, the amount of memory is crucial and not surprisingly, the more memory the algorithm has, the better its performance. The optimal number of simultaneous streams to merge seems to agree with streams test and is around 5.
- It is hard to predict the effect of different parameters on the performance of disk IO. The research on the internet also suggests that the effect can also be very technology and disk model specific.

- One of the tests that was not included in this report was performed on Amazon Web Services t2.micro instance with EBS storage (Amazon Elastic Block Store). The results were very inconsistent and upon further research we have found that "Each volume receives an initial I/O credit balance of 5.4 million I/O credits, which is enough to sustain the maximum burst performance of 3,000 IOPS for 30 minutes. This initial credit balance is designed to provide a fast initial boot cycle for boot volumes and to provide a good bootstrapping experience for other applications." [3]. Since cloud storages have become very common, it is now important to measure their performance too, and this is not a simple task.

References

[1] Openjdk: Jmh. [Online]. Available: http://openjdk.java.net/projects/code-tools/jmh/

[2] Mapping files into memory. [Online]. Available: https://developer.apple.com/library/content/documentation/FileManagement/Conceptual/FileSystemAdvancedPT/MappingFilesIntoMemory/MappingFilesIntoMemory.html

[3] Amazon ebs volume types. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EBSVolumeTypes.html