Database Project

Students' names:

Hanqing Zhao

Xudong Fan

Liyuan Wu

2018.01.03

Comments

1 Introduction and Environment

1.1 Project goals

In the assignment, we are asked to implement an external-memory merge-sort algorithm, and examine its performance under different parameters. We explored several different ways to read data from, and write data to secondary memory and made experiments to check the performance of these algorithms. The overall goal is to get real-world experience with the performance of external-memory algorithms.

1.2 Machine Environment

For this project, all the test behaviors were generated in a MacBook Pro (15-inch, 2017) computer with the following specifications:

Computer type: MacBook Pro (15-inch, 2017)

Operating System: macOS 10.13.1 (17B1003)

CPU: 2.9 GHz Intel Core i7

RAM: 8192MB LPDDR3 memory available for JAVA 1.6 environment (16GB in total)

Hard Disk: APPLE SSD SM0512L 512GB

1.3 Programming Language and External Libraries

We used Java 1.6 for these four implementations of input/output streams, and multi-way merge sort. We also wrote some test files in Java 1.6 to measure the running time of these four implementations of the input/output streams and multi-way merge sort.

To implement the project modules, in addition to standard *java.io, java.util* and *java.lang.Math* labraries, an external library *com.jmatio* is included to generate Matlab compatible .mat test data in order to draw testing result illustrations, as a result, some scripts for generating illustrations are written in Matlab 2017a environment.

1.4 Generation and Size of test data

In this project, testing files with 32-bits integers randomly generated within the range 100 (400 bytes) numbers to 1×10^8 (400MB) numbers are used for experiments, scripts for generating different test data could be found at method:

1. testFileGen.java: generates testing files for merge sort algorithm test.
2. main2.java: generates testing files and performs I/O stream tests.

2 Observations on Stream

2.1 Implementation and expected behavior

Each read implementation has the following methods:

1. open(filename (and buffersize for implementation 3 and 4)): Open an existing file for reading.

2. read_next(): Read the next element from the stream.

3. closeFile (): Close the stream.

4. end_of_stream(): A boolean operation that returns true if the end of stream has been reached.

Each output implementation supports the following methods:

1. create (): Create a new file.

2. write (integer): Write a 32-bit integer to the output file.

3. close (): Close the stream.

We should also define, for each of the 4 implementations, a cost formula that estimates the total number of I/Os that needs to be done, in function of N, k, and B. We define the following parameters:

N: Number of integers to be read/written on each stream.

k: Number of streams open at a time.

B: Size of buffer in internal memory.

2.1.1 Implementation 1

We have done this implementation following exactly the instructions in the assignment. For input stream: we mimic read by calling readInt() on a java.io.DataInputStream that is wrapped directly around ajava.io.FileInputStream; For output stream: we wrap a java.io.FileOutputStream by a java.io.DataOutputStream. Each time a 32-bits integer is read from the input stream or write to output stream, it will take one I/O operation, Therefore, the cost of I/Os for this implementation is : $\boldsymbol{C = N * k}$.

Expected performance behavior:

1. According to the cost function $\boldsymbol{C = N * k}$, its performance will decrease with the increase of N and k.

2. The performance is proportional to the size of N when k is fixed and vice versa.

2.1.2 Implementation 2

The second implementation used the java's own buffering mechanism, as the implementation of method one, we implemented the read and write streams, but the content of DataInputStream is BufferedInputStream, not FileInputStream. For input stream: we can mimic fread by calling readInt() on a java.io.DataInputStream that is wrapped around a java.io.BufferedInputStream that itself is wrapped around a java.io.FileInputStream; For output stream: we can wrap a java.io.DataOutputStream that is wrapped around a java.io.BufferedOutputStream that itself is wrapped around a java.io.FileOutputStream.

Suppose that buffer size is B, B integers will be loaded into a buffer in block. Therefore, for each stream, it requires [N/B] I/Os. Finally, The I/O cost is: $\boldsymbol{C} = \boldsymbol{k} * [\boldsymbol{N}/\boldsymbol{B}]$.

Expected performance behavior:

1. Because the time cost is $\boldsymbol{k} * [\boldsymbol{N}/\boldsymbol{B}]$, if buffer_size is larger, the performance will be better.
2. If *B* and *N* are fixed, with the increase of *k*, time cost will increase linearly.
3. Because of the buffer mechanism, this performance will be much better comparing with method 1.

2.1.3 Implementation 3

In this implementation, we introduced a byte array as the buffer. Each time an integer is read or written, it is first read/written to/from the buffer. When the buffer is empty/full, the next buffer_size integers (which is 4*buffer_size bytes because the size of a byte is 8 bits and the size of an int is 32 bits) will be loaded/written from/into a file(Input/Output) Stream. For input stream, the buffer will be re-loaded when the buffer becomes empty. For output stream, the integers loaded in the buffer will be written into an output file when the buffer becomes full. Because we are using the data type ***byte***, we have to make some conversions between ***byte*** and ***int*** before extracting/pushing an integer from/to the buffer. As shown in Table 2.1.

Table 2.1 Code for byte to integer conversion and vice-versa

```
public static int byteArrayToInt(byte b0, byte b1,byte b2, byte b3) {
    return    b3 & 0xFF |
              (b2 & 0xFF) << 8 |
              (b1 & 0xFF) << 16 |
              (b0 & 0xFF) << 24;
}
```

```
public static byte[] intToByteArray(int a) {
    return new byte[] {
              (byte) ((a >> 24) & 0xFF),
              (byte) ((a >> 16) & 0xFF),
              (byte) ((a >> 8) & 0xFF),
              (byte) (a & 0xFF)
};
}
```

We wrapped a data(Input/Output) Stream around a buffer(Input/Output) Stream by which a file(Input/Output) Stream is directly wrapped. Suppose that buffer size is B (integers), the integers are loaded into buffer in block of B integers, hence, for each stream, it requires [N/B] I/Os. Finally, The I/O cost function can be defined as: $\boldsymbol{C} = \boldsymbol{k} * [\boldsymbol{N}/\boldsymbol{B}]$.

Expected performance behavior:

1.It works better than implementation 1 and implementation 2.

2.The larger buffer size is, the better the performance is.

2.1.4 Implementation 4

In this implementation, we use the memory mapping to mapping and unmapping B element portion of the file into internal memory. And whenever we need to read /write outside of the mapped portion, the next B element portion of the file is mapped. Different from the Bytebuffer we implemented before, we use the MappedByteBuffer. MappedByteBuffer is a special ByteBuffer, a subclass of ByteBuffer. It maps files directly to memory (memory here refers to virtual memory, not physical memory). Especially, we can map the entire file, if the file is relatively large, we can map it in sections by defining the MappedByteBuffer size B.

Cost: $\boldsymbol{C} = \boldsymbol{k} * [\boldsymbol{N}/\boldsymbol{B}]$

Expected performance behavior:

Because of the memory mapping mechanism, when data file is very large, its performance will be the best among these 4 methods.

Memory mapping: The standard I/O operation is shown in Figure 2.1. When process need specified data in disk files, Firstly, it read buffer_size data into CPU buffer space, and then, copy data in CPU buffer space to process buffer space, then, this process can use these data.

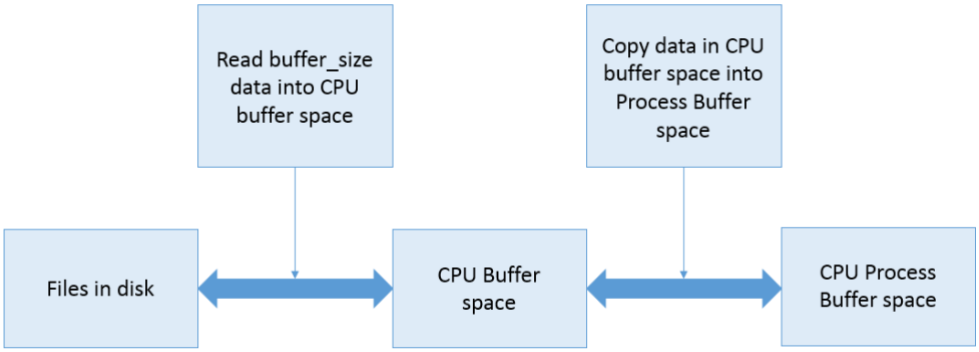


Fig 2.1 Standard I/O operation

But in MappedByteBuffer mechanism, when the process needs certain data, it will copy data in files directly into process buffer space. In this way, the read and write is much faster than standard I/O operation. But it also has some problems, mainly the memory usage and uncertain of files closure, because a mapped byte buffer and the file mapping that it represents remain valid until the buffer itself is garbage-collected.

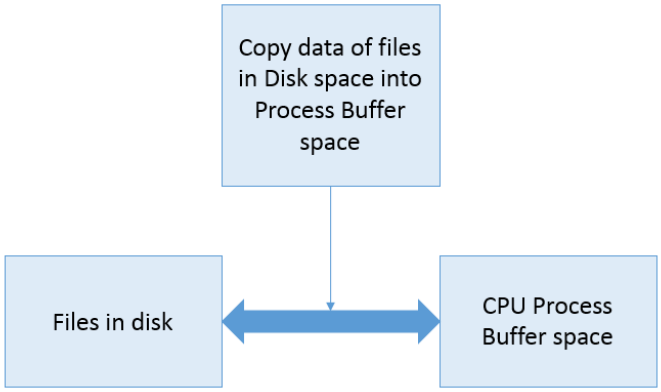


Fig 2.2 mapped buffer operation

2.2 Observations and Experiment

In this part, we have done 2 tests.

The first one is the optimal value of buffer test. We aim to find the best buffer size B for implementation 3 and 4. we ran implementation 3 and implementation 4 with different buffer size B to find the optimal value for each implementation.

The second test is to comparing performance among four implementations. After having the optimal B, we tested four implementations by varying N (fix k) and varying k (fix N) to find the most performance one.

During each experiment, because the experiment environment, like internal memory and background process, is not always the same, so even if all the parameters are the same, the experiment result could be different. And some unnormal experiment results will influence our analysis about the performance of the program. Therefore, to avoid occasionality, we do: for same experiment, we do 5 times, and compute the average time cost. And depending on results data, we made some analysis.

2.2.1 Optimal value of buffer evaluation

To determine the corresponding optimal buffer size B of implementation 3 and implementation 4, we read all 32-bit integers from an external file and write them into another external file. We fixed number of integers in the external file at N=1000000 (for example) and tested with different values of B. We chose B

ranges in [20000 400000 600000 800000 … 20000000].

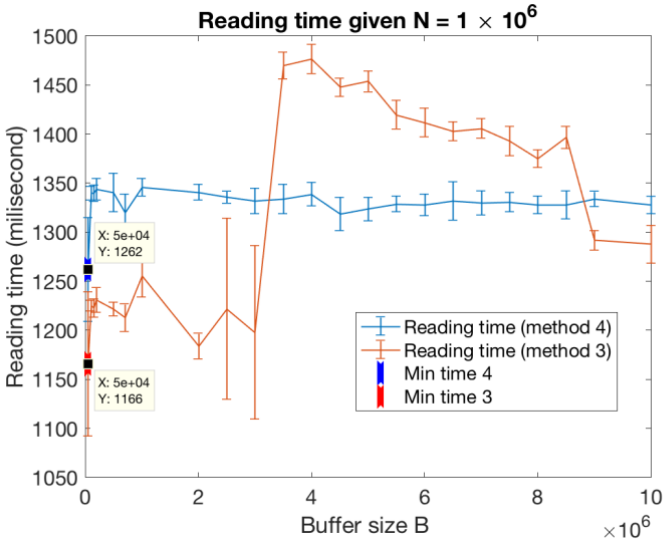


Fig 2.3 Optimal buffer size for reading

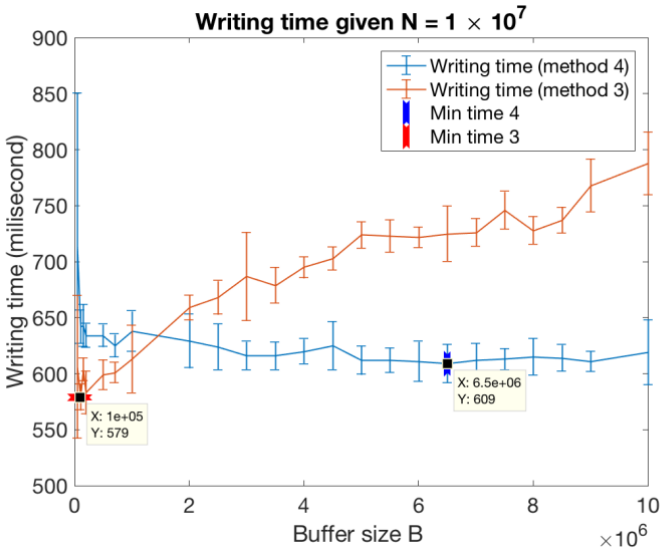


Fig 2.3 Optimal buffer size for writing

As we can see from the above figures, when treating files with ten six-magnitude to ten seventh-magnitude using methods 3 and 4, the total time consumption is dominated by Reading time (50 times larger than Writing time in same file size), as can be seen from Figure 2.3, the best buffer size for both method are around 5×10^4 integers (5% of the file size 1×10^6) , for most circumstances, this value can be regarded as the global optimization value for method 3 & 4 regarding file size between 1×10^6 to 1×10^7.

However, when we only consider writing performance, the best buffer for method 3 appears at a similar position around 1×10^5 integers (1% of the file size 1×10^7). But there is no significant global optimal

buffer size for method 4, as error bars are overlapped.

2.2.2 Stream Performance Test

Firstly, in order to compare the performance of four different stream implementations, we conducted the two following subtests to compare performance between four methods.

In the first subtest, we kept N fixed and ranged the number of stream k. For each value of N in the list [5000, 10000, 500000, 1000000], we tried to open k input and output streams with k=1, 2, 3 ...30.

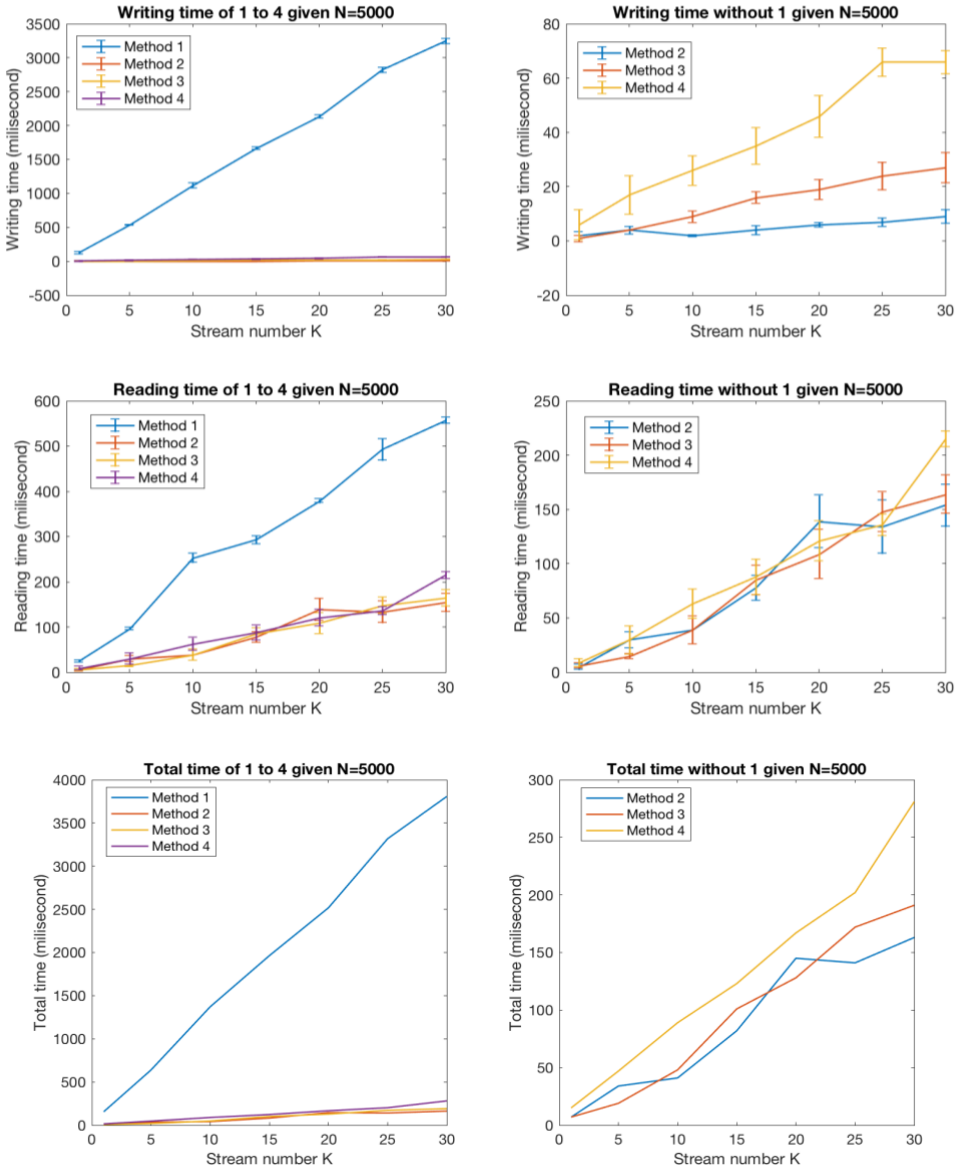


Fig 2.5 Fix N=5000

Fig 2.5 illustrates the performance of four implementations when read/write k streams. It can be seen that implementation 1 is the slowest in both reading and writing tests. And time of running implementation

1 rose gradually when increasing the number of streams. When read/write 30 streams, implementation 1 took around 3.5 seconds while implementation 2, 3 and 4 took less than 0.3 seconds.

Besides, regarding the total time consumption, unlike the following results, the implementation 2 performs better than 3 and 4 when treating files with 1×10^5 integers, this may be explained as, during the treatment of small files, the initialization cost of new buffer (method 3) or new filemapping (method 4) becomes significant.

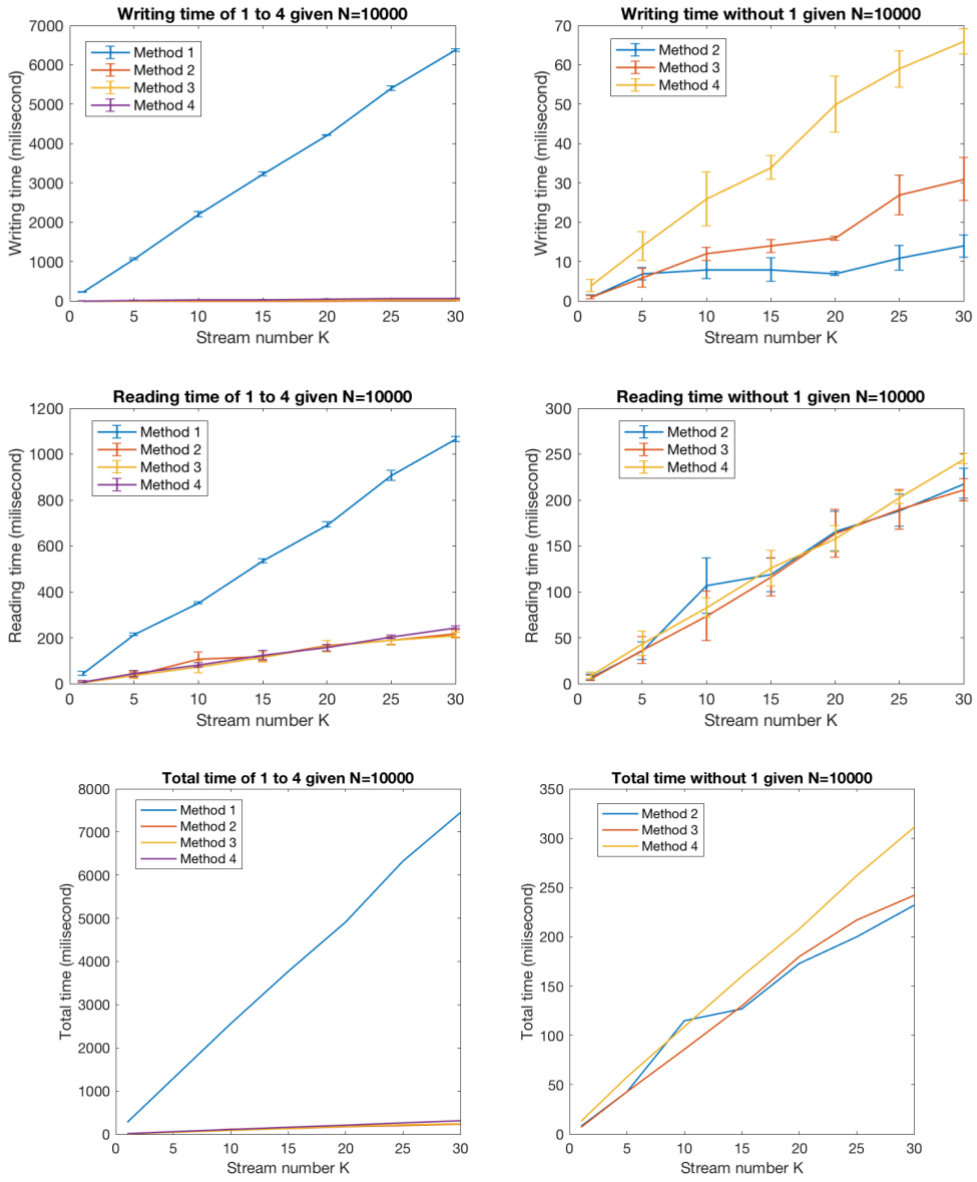


Fig 2.6 Fix N=10000

When we increased N=10000, as shown in Fig 2.6, implementation 1 is still the slowest and took around 7.5 seconds to read/write 30 streams and it took double time comparing with N=5000. We can also see that for implementation2, the writing time is even smaller than implementation 3 and 4.

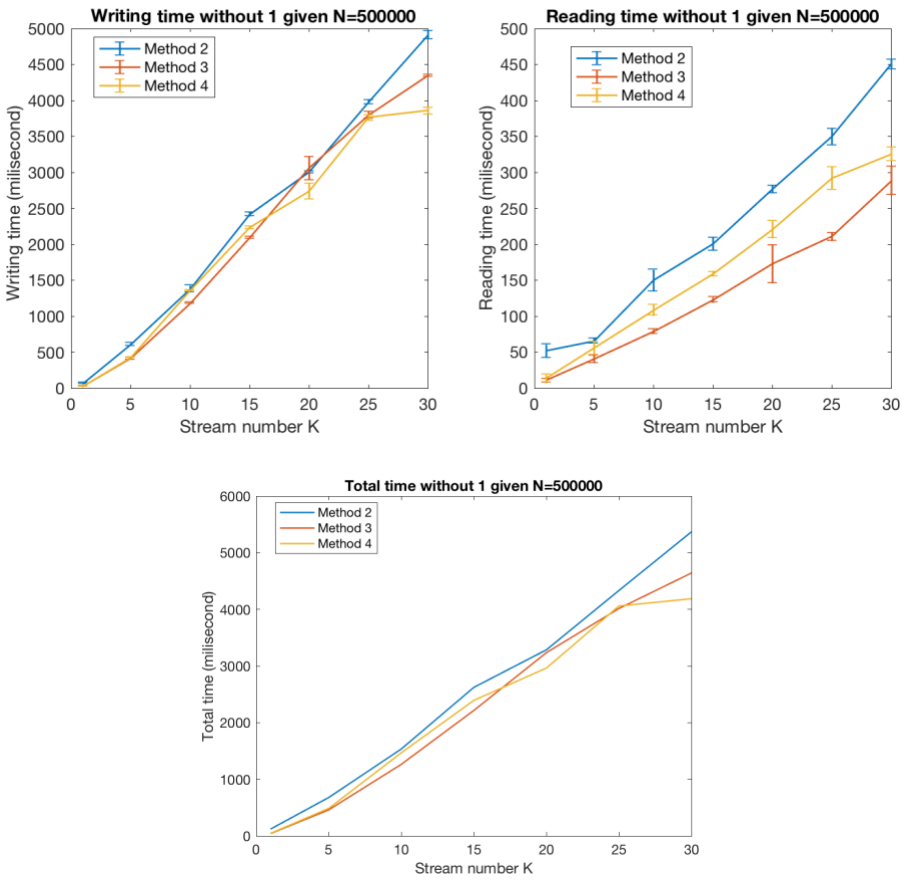
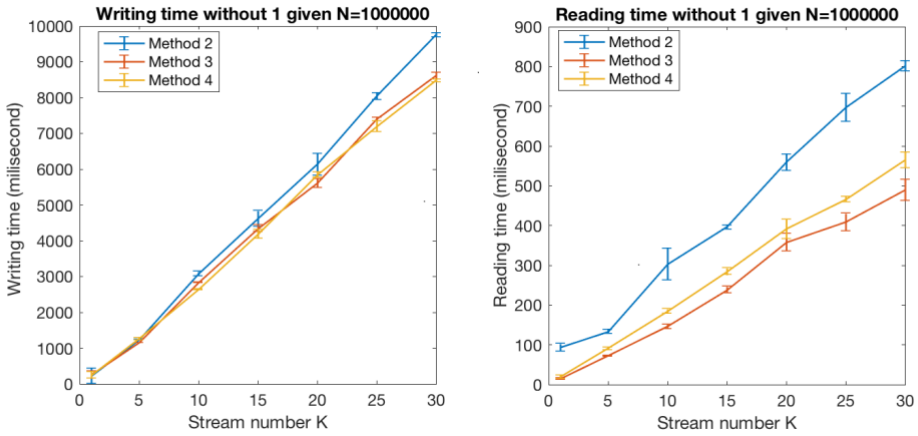


Fig 2.7 Fix N=500000

With N=500000, shown in Fig 2.7, as predicted, all three implementations took more time for read/write as increasing number of streams. In general, implementation 2 was the slowest one, the performance of implementation 3 was between the implementation 2 and implementation 4 at the beginning, but when k gets bigger, implementation 3 may perform the best.

As file size increases, the new buffer and new file mapping initialization cost becomes no longer significant, the performance of implement 3 and 4 gradually surpasses that of implementation 2.



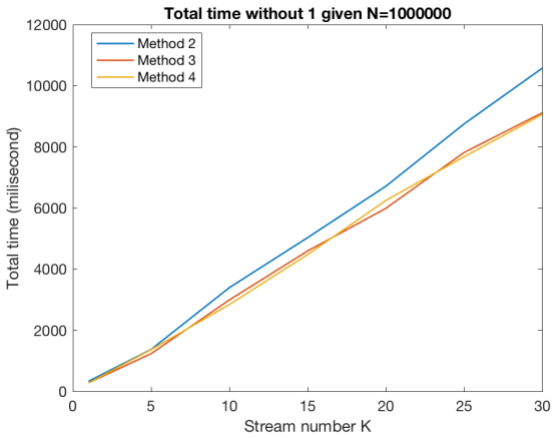


Fig 2.8 Fix N=1000000

With N=1000000, shown in Fig 2.8, as predicted, all three implementations took more time for read/write as increasing number of streams. For reading, implementation 2 was the slowest, implementation 3 was in the middle and implementation 4 performed the best. In general, implementation 2 was the slowest one. For implementation 3 and implementation 4, it's hard to distinguish which one performed better. A further analysis on implementation 3 and 4 are given in Section 2.2.3.

In the second subtest, we kept number of stream k fixed and ranged N. We chose k=5, 10, 20, 30. For each k, we tried with different value of N in the list [5000, 10000, 500000, 1000000].

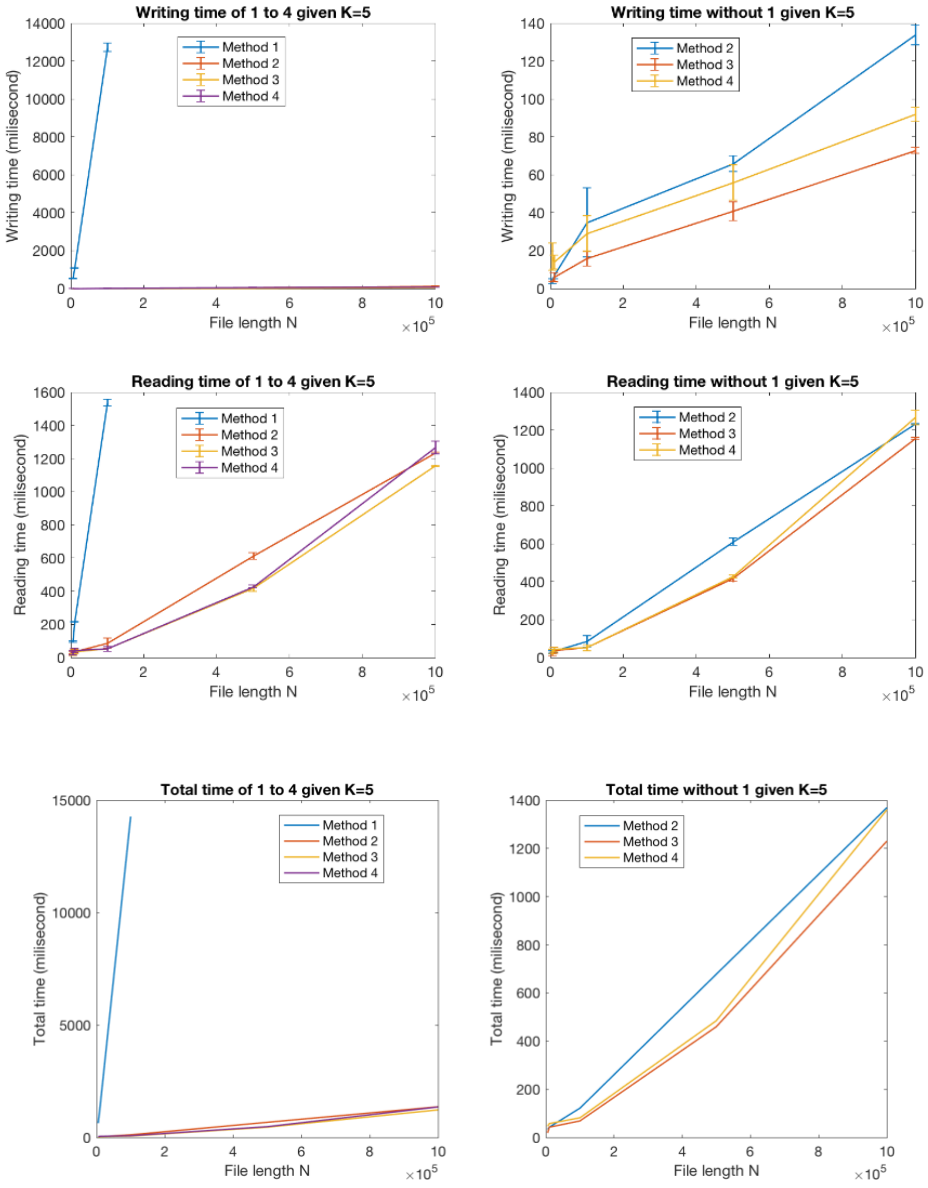


Fig 2.9 Fix k=5

With 5 streams, as shown in Fig 2.9, implementation 4 performed the best and implementation 1 performed the worst.

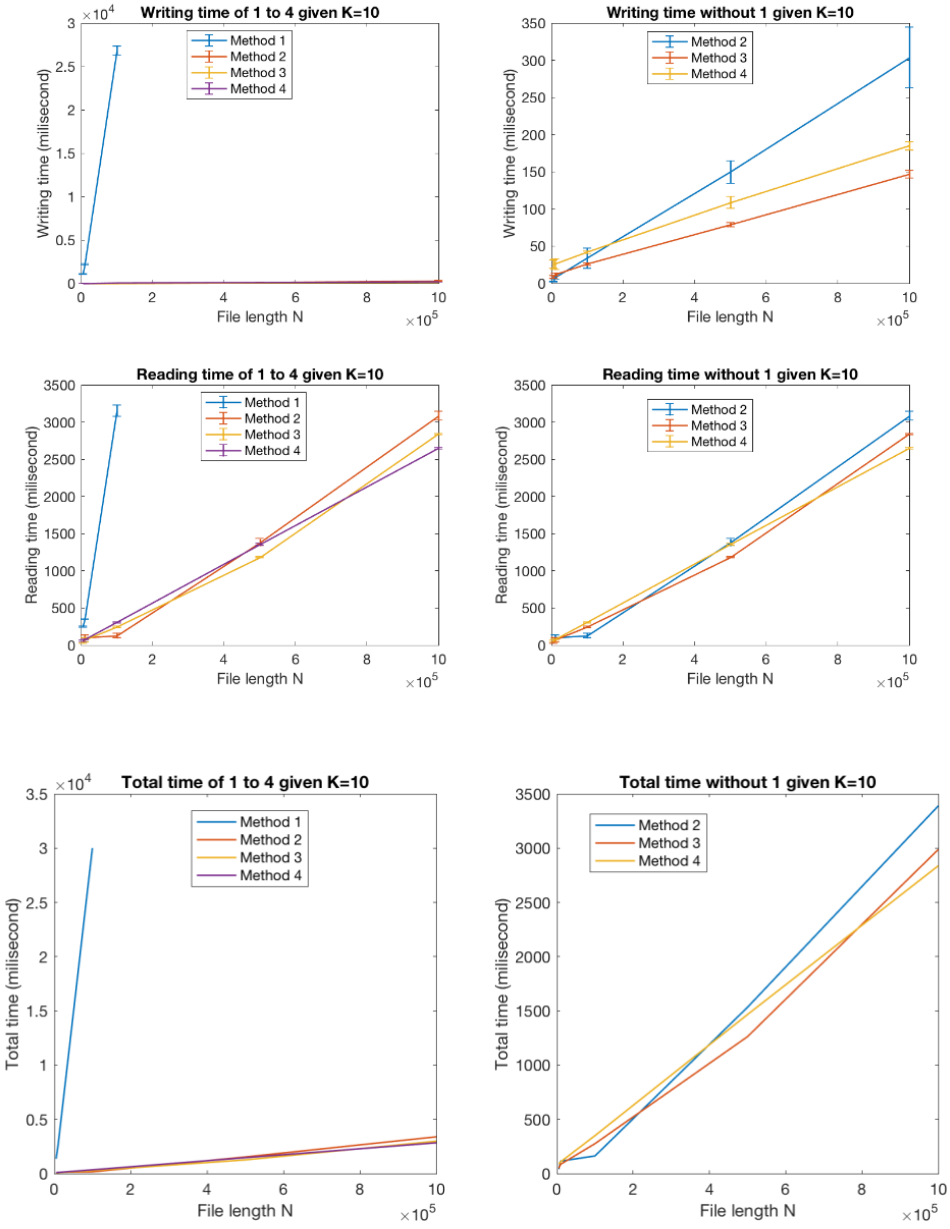


Fig 2.10 Fix k=10

With 10 streams, as shown in Fig 2.10, implementation 4 was still the best but the performance of implementation 3 increased when increasing k. When k grows, the performance of implementation 3 might become better than implementation 4.

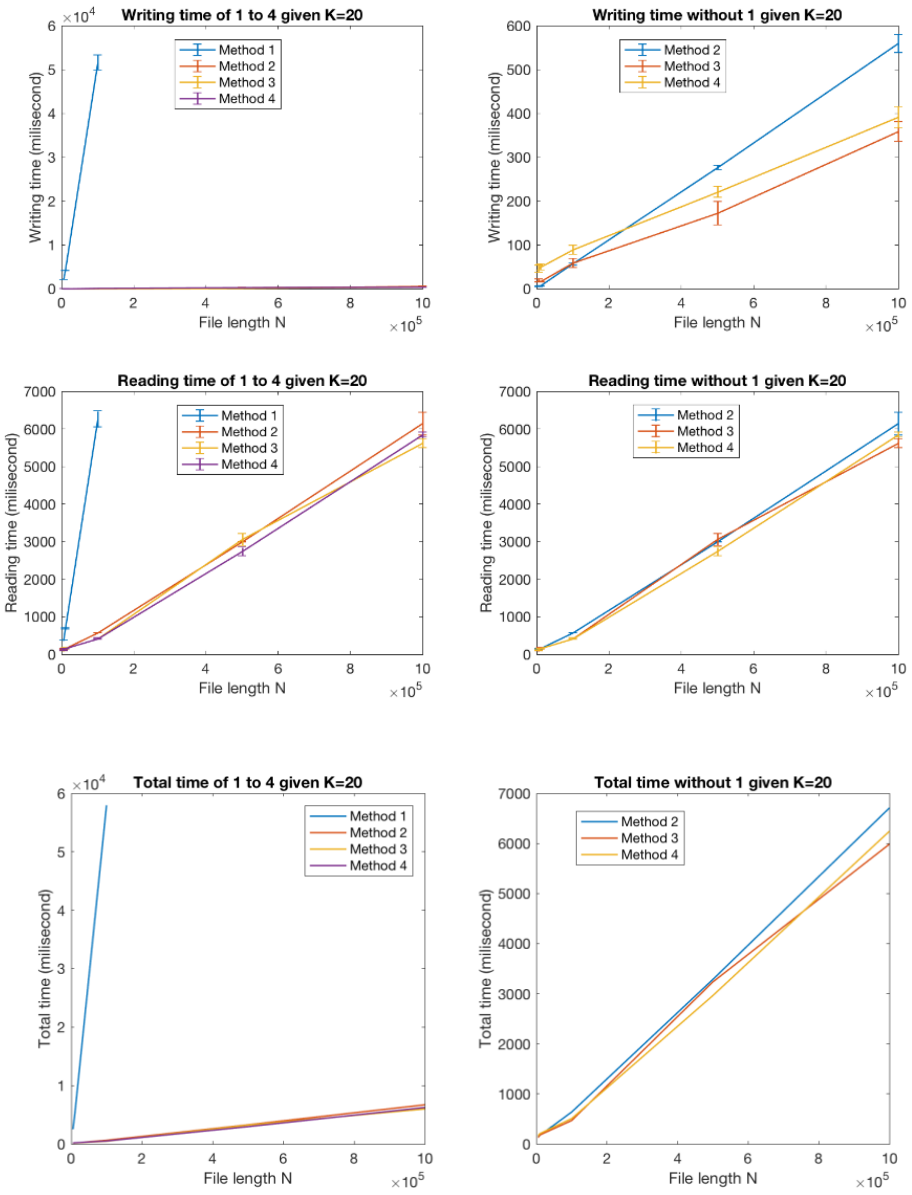


Fig 2.11 Fix k=20

With 20 streams, as shown in Fig 2.11, implementation 2 still performed the worst among implementation 2, 3, 4 and implementation 4 was still the best in a limited scope. When the file size grows, the performance of implementation 3 might become better than implementation 4.

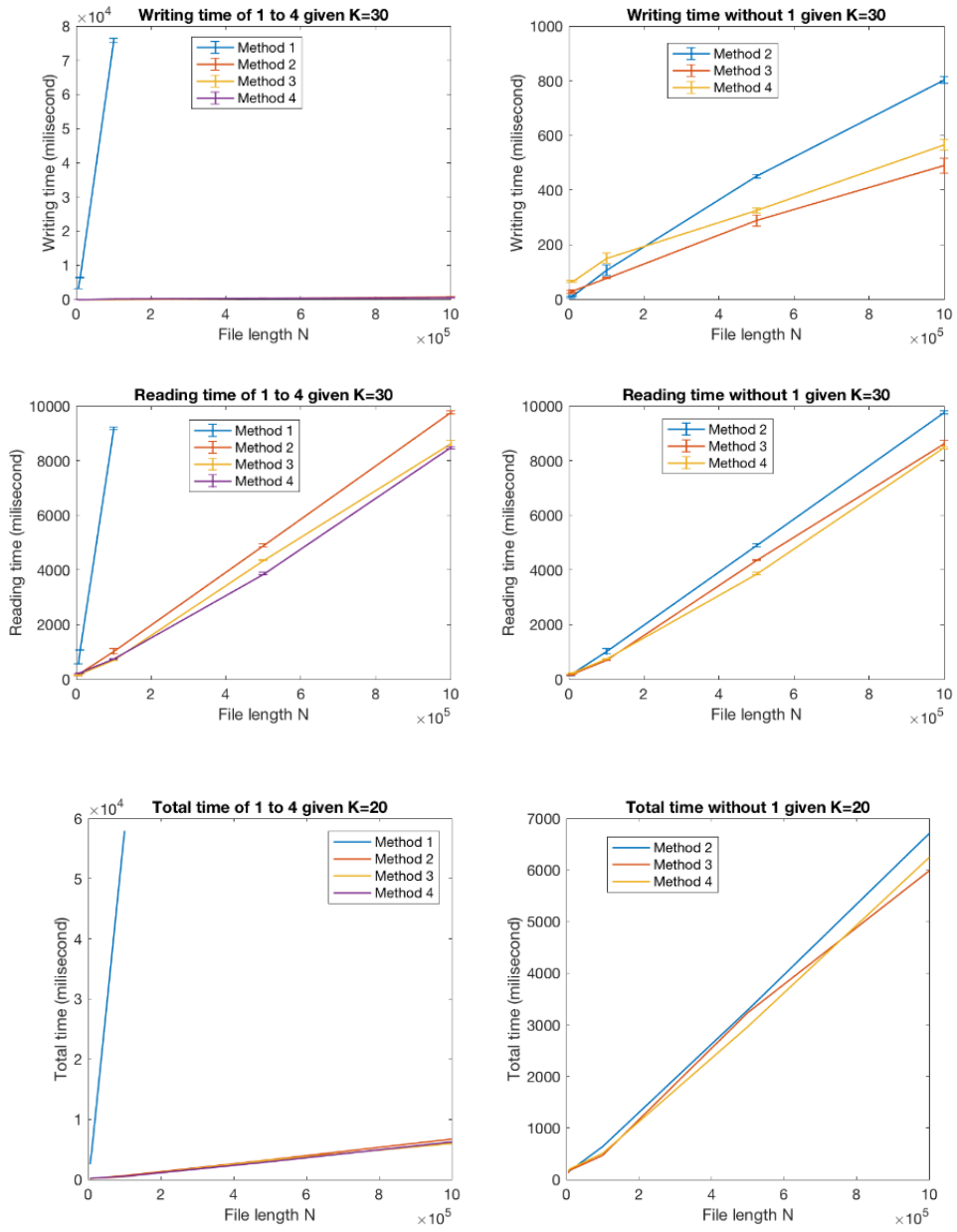


Fig 2.12 Fix k=30

With k=30, as shown in Fig 2.12, for writing, method 3 works better than method 4 and when the file size is small, method 2 works better than method 3; For reading, method 4 works the best. When we are considering the total time for read and write, implementation 3 performed slightly better than implementation 4 at N=1000000, but in general, two of them performed almost similarly.

Secondly, we also draw some 3D images to present the influence of each parameters.

In this experiment, the parameter is **N***(repeat times)* and **k***(number of streams)*, for **k**, the step size is 5, experiment *k* are { 1, 2, 3, 4, 5,…, 30 }, experiment *N* are {100, 500, 1000, 2500, 5000, 7500, 10000, 25000, 50000, 75000, 100000}.

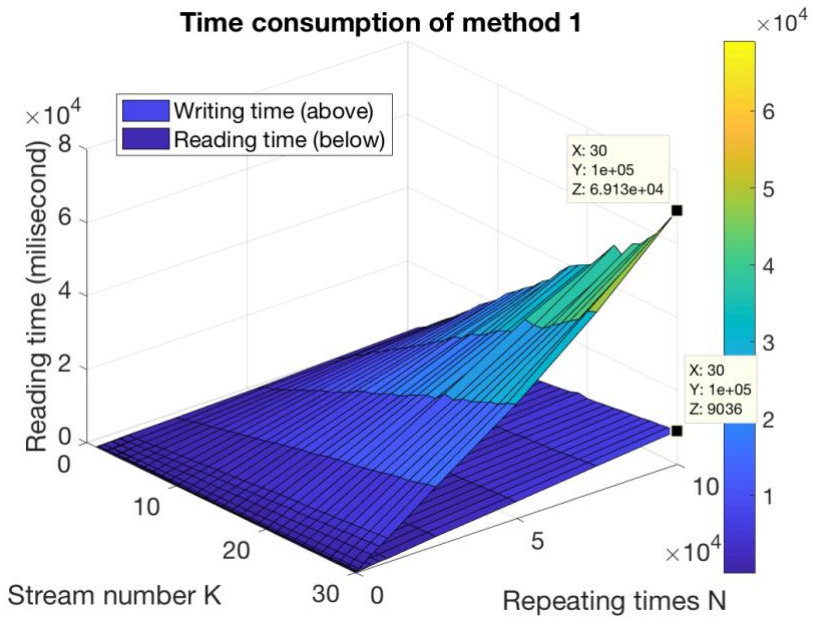


Figure 0.13 Writing and reading time of method 1

Fig 2.13 shows the writing and reading time cost, as we can see, if **N** and **k** are small, the time cost is small, but with the increase of **N** and **k**, the writing time and reading time increase almost linearly, which corresponds to our expectation of this method. And the charts also correspond the cost function.

But there are some vertical Standard Deviation lines which are overlapped in small time section, for example, when *N=2.5*10^4,* and *k=15 and 30*. Hereto, we think that is because the experiment time is not enough (we do 5 times for one test), and the overlapped region is small, beyond the 2σ region, according to Normal distribution knowledge, therefore, considering the probability, we can conclude, when k is bigger, time cost is bigger.

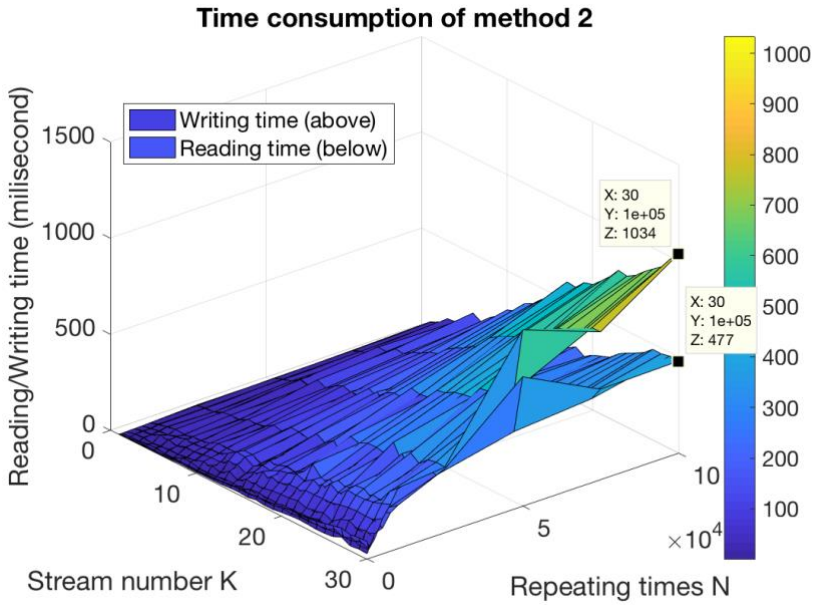


Figure 0.14 Writing and reading time of method 2

The choice of N and k is the same as method 1, **k** are { 1, 2, 3, 4, 5,…, 30 }, experiment **N** are {100, 500, 1000, 2500, 5000, 7500, 10000, 25000, 50000, 75000, 100000}. As shown in Fig 2.14, compared with the results of method 1, time consumption is much more less, the same with our expectation of this method. That is because we used buffer mechanism. And with the increase of N and k, the growth cost is more gently than method 1.

But there is two points which is unnormal, k=30, N=50000, writing and reading time is much higher than its neighbors, we think that is because at that time, there are more background processes, which leads to this result.

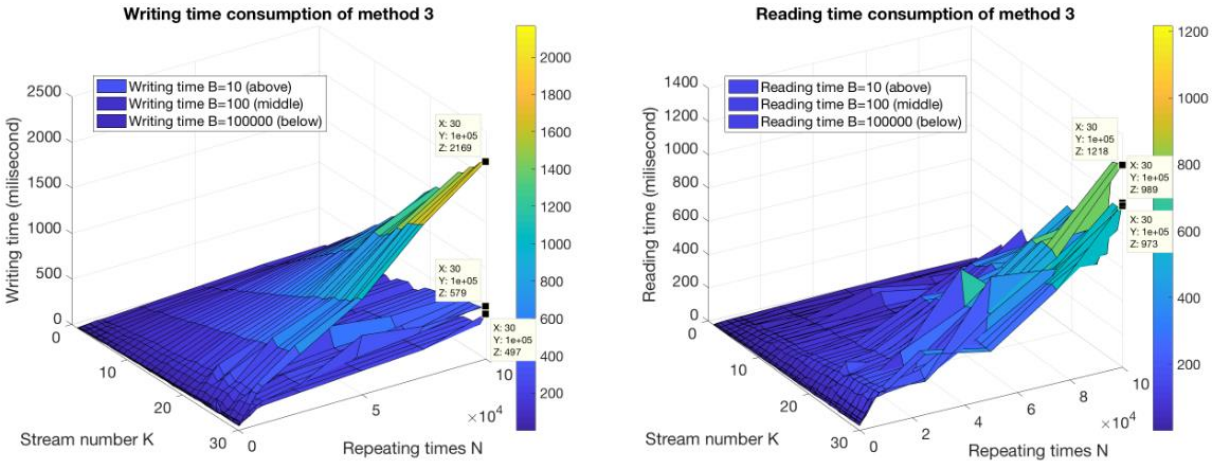


Figure 0.15 Writing and reading time of method 3

The choice of **k** and **N** are the same as before, in the writing part, we can see, when B is small, the growth of time is fast, but when B is relatively large, with the increase of **k** and **N**, the time doesn't have too much growth, which corresponds to the cost function, $\boldsymbol{C = k * [N/B]}$. But in reading result figure, time cost seems to be not influenced by the buffer size B.

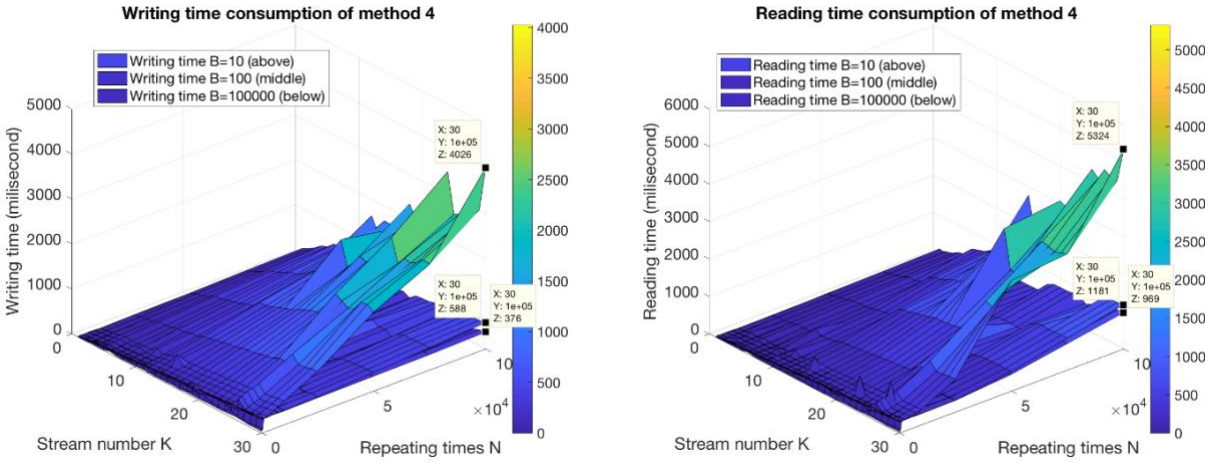


Figure 0.16 Writing and reading time of method 4

In both writing and reading parts, time cost decreases a lot with the increase of B, so the reading performance is better than method 3. When **B** is relatively large, performance of this method is the best.

But when **B** is small, and **k** and **N** are large, its performance is not stable. We think that is maybe because the memory mapping mechanism, the file closure is not stable.

2.2.3 Large file treatment test for method 3&4

As introduced in previous experiments, when file size gradually increases from 5000 to 1×10^6 integers, the performance of method 3 and 4 gradually surpasses that of method 2. This shows that they are more suitable for handling large files, but we still curious about the further performance evolution of method 3 and 4 when files size increases continuously. For this, we included file size to 1×10^8 integers (400 MB), the result of performance evolution of implementation 3 and 4 with large files are shown respectively in Figure 2.17 and Figure 2.18.

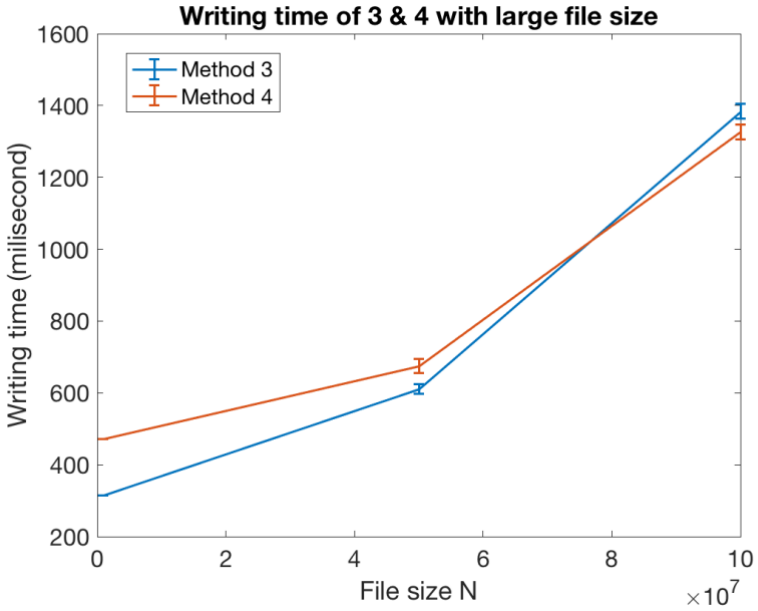


Figure 2.17 Writing time evolution of implementation 3 & 4 with large file size

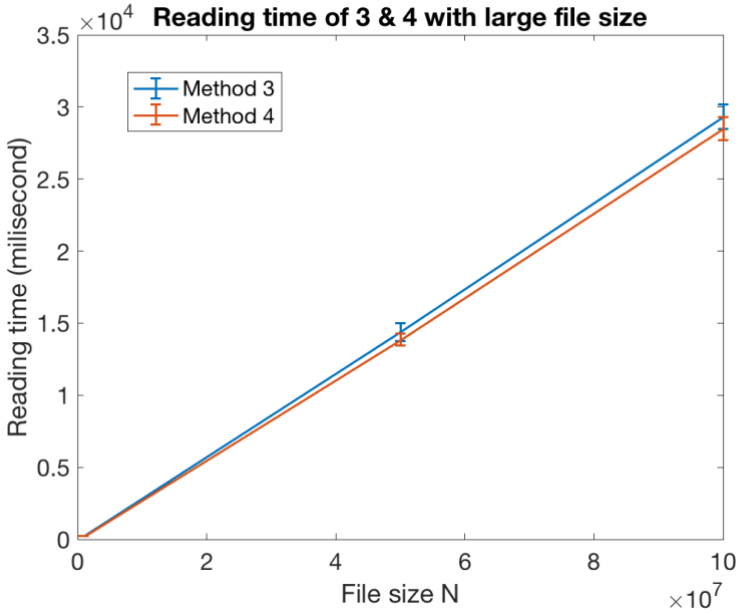


Figure 2.18 Reading time evolution of implementation 3 & 4 with large file size

As shown in above figures, when file size increases from 1×10^7 to 1×10^8 integers, the performance of implementation 4 surpasses that of 3 in both Reading and Writing time consumption, while the aforementioned experiments show that implementation 3 works better for file smaller than 1×10^7 integers.

At this point, we can conclude that implementation 4 is more suitable for treating files larger than

1×10^8 integers and implementation 3 is more suitable for treating files smaller than 1×10^7 integers.

3 Observations on Multi-way Merge Sort

In the multi-way merge sort, we have used the following parameters and the external libraries:

N: The size of the input file, measured in number of 32-bit integers.

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.

3.1 Implementation

In this part if the project, first, we divided the input stream into smaller chunks of maximum **M** 32-bit integers. For each chunk of integers, we sort each stream and save them to Stream_ids.bin. Then, we merge all streams, **D** streams at a time.

For each merge phase operation, the following steps are performed:

Step 1: Get maximum d input streams and put them in a priority queue.

Step 2: Prepared an output stream to write the final merged result into an external file.

Step 3: While the priority queue is not empty, get the top input stream from the queue.

Step 4: Read the next number from the selected input stream, write it to the external file generated in Step 2 and if the input stream is not empty, push it again to the priority queue and go back to Step 3.

Step 5: If there's still some remaining streams, go back to Step 1.

I/O Cost function:

At the beginning, each integer in the input stream is read from the hard disk and then written to the hard disk, which needed $\mathbf{2}*\mathbf{N}$ I/Os and produced $[\boldsymbol{N}/\boldsymbol{M}]$ external files.

After each merge phase, for each integer in different external files, it is also read and written from/to hard disk once, hence, the number of I/Os is $\mathbf{2}*\mathbf{N}*$p, where p is the number of merge phase operation.

If $\mathbf{D}^p = [\boldsymbol{N}/\boldsymbol{M}]$, it is required p merge operations for $[\boldsymbol{N}/\boldsymbol{M}]$ external files. If $\mathbf{D}^{p+1} = [\boldsymbol{N}/\boldsymbol{M}]$, it is required p+1 merge operations for $[\boldsymbol{N}/\boldsymbol{M}]$ external files. Therefore, p= $log_{\mathbf{D}}[\boldsymbol{N}/\boldsymbol{M}]$.

So the final I/O cost = $\mathbf{2}*\boldsymbol{N}+\mathbf{2}*\boldsymbol{N}*log_{\mathbf{D}}[\boldsymbol{N}/\boldsymbol{M}]$.

3.2 Experimental Observations

Due to the testing time limitation, the merge-sort algorithm is tested under filesize between between 1×10^5 to 1×10^6 integers. As introduced in section 2.2.2 and 2.2.3, the best I/O stream implementation for file size between 1×10^5 to 1×10^6 integers is implementation 3, as a result, the stream implementation 3 is used in merge-sort algorithm,

We tested merge-sort algorithm in 4 experiments.

3.2.1 Experiment 1(fix **N** and **M**)

We chose **D** ranges in [0, 100] while fixing **N**=1000000 and **M**=20000.

Expected performance behavior: The performance would get better if **D** gets bigger since it needed less merge operations.

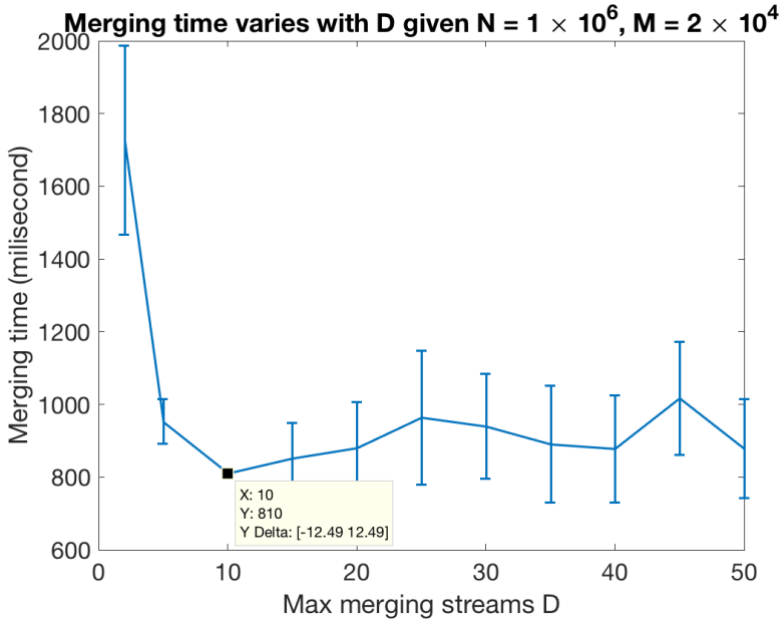


Fig 3.1 Fix N=1000000 and M=20000

Fig 3.1 shows that the merging time will decreased (better performance) at first, and then, when **D** is bigger than 10, the speed decreased. Therefore, the optimal value of d should be d =10=[N/M]/5.

3.2.2 Experiment 2(fix **M** and **D**)

We chose **N** ranges in [0, 10000000] while fixing **M**=100000 and **D**=25.

Expected performance behavior: The running time gets larger when **N** gets bigger because the performance time is relatively proportional to the size of input.

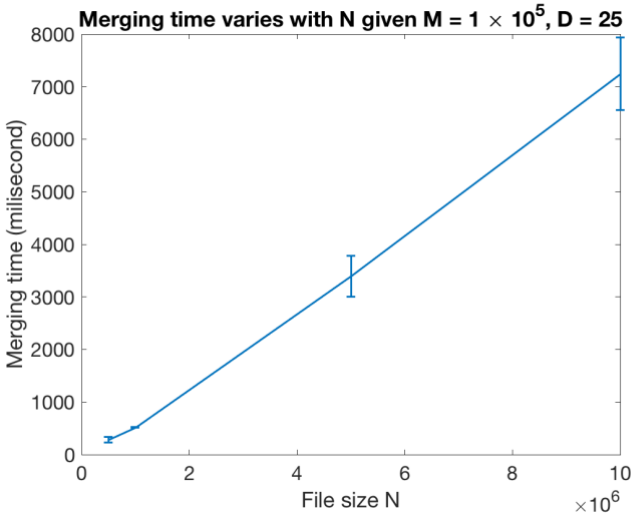


Fig 3.2 Fix M=100000 and D=225

Fig 3.2 shows that the merging time increased (worse performance) when increasing **N**, just as what we expected.

3.2.3 Experiment 3(fix **N** and **D**)

M ranges in the percentage of the size of the file [0, 50%] while fix and **D**=50 and **N**= 1×10^6 respectively. It is noticeable to mention that, in this sub experiment, we only consider M varies under 50% file size, because when M becomes larger than 50% file size, the file can always be merged in one time.

Expected performance behavior: When M becomes larger, the sorting time needed will decrease.

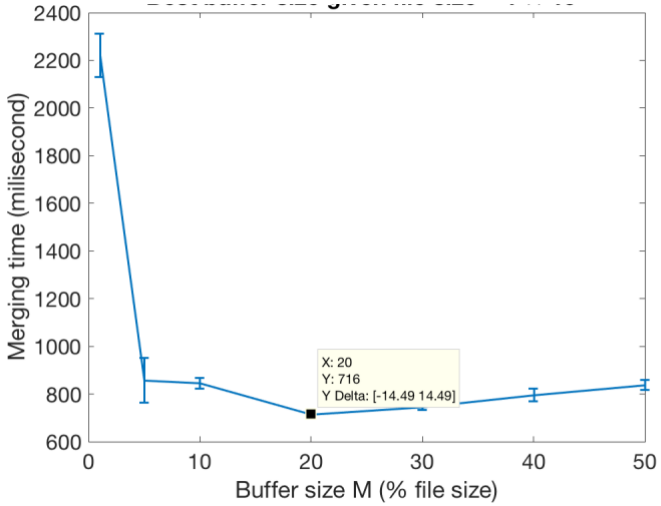


Fig 3.3 Fix N= 1×10^6 and D=50

It contrasts to expected behavior, as shown in Fig 3.3, that when the buffer size **M** gets bigger, the merging time will decrease at first and then increases. One possible reason is that when **M** is large, at the beginning of the merge sort part, the time for dividing the input stream and sorting each stream would get

bigger too. Therefore, it could affect the total merging time. Moreover, for both files size, the best global buffer size M is equal to 20% of file size.

3.2.4 Comparation of our merge-sort implementation and a main memory sort

In this test, First, we fixed N = 1000000 and executed the merge sort implementation with different M to compare the performances of the multiway-merge sort and the internal sort.

M ranges in the percentage of the size of the file **[0, 100%].**

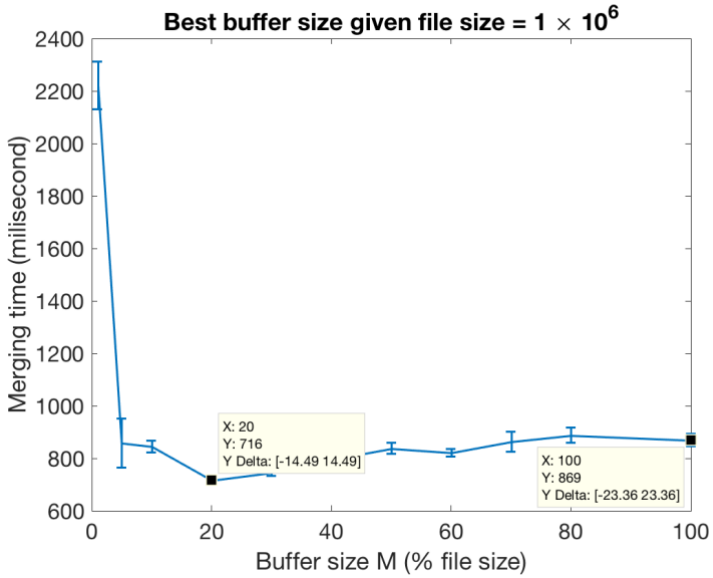


Fig 3.4 Fix N=1000000

As shown in Fig 3.4, when the input file is big enough, the external sort algorithm (multiway-merge sort) is a little bit faster than the internal sort algorithm (when the buffer size is 100%). The optimal value for M was 20%.

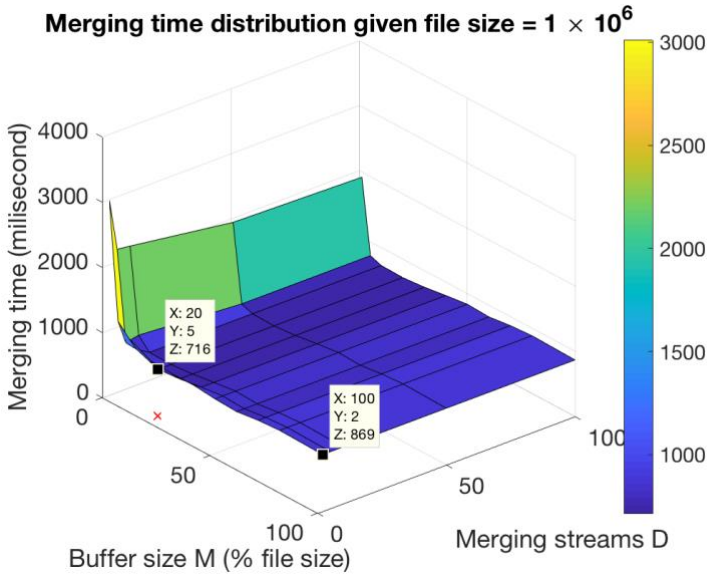


Fig 3.5 Fix N=1000000

As we can see from Fig 3.5, the influence of D was relatively too small compared with the influence of buffer size M to the merging time.

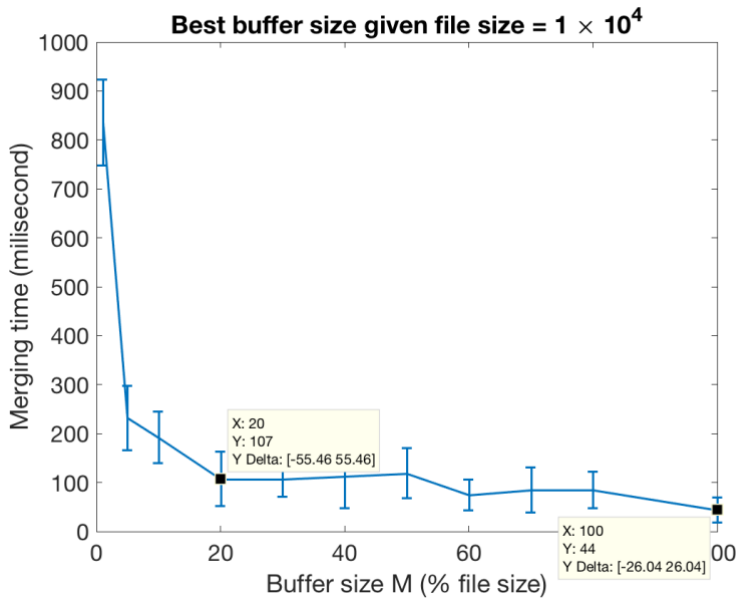


Fig 3.6 Fix N=10000

As shown in Fig 3.6, when the input file is small enough, the external sort algorithm (multiway-merge sort) is slower than the internal sort algorithm (when the buffer size is 100%). The optimal value for M was also 20%.

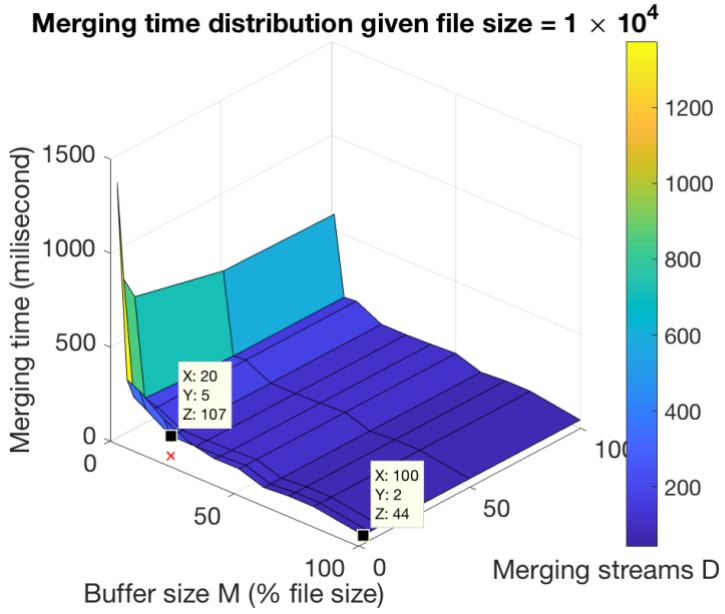


Fig 3.7 Fix N=1000

As shown in Fig 3.7, the influence of D was relatively too small compared with the influence of buffer size M to the merging time.

4 Conclusion

To summarize, we have implemented four different input/output algorithms and a new external sort algorithm (multiway merge sort). We have tested the four implementations varying only one parameter and fixing the other parameters. As our prediction, the test result shows that,

Implementation 2 performed the best for file size smaller than 1×10^4 integers;

Implementation 3 performed the best for file size from 1×10^4 integers to 1×10^7 integers;

Implementation 4 performed the best for file size larger than 1×10^8 integers;

We used this implementation 3 to examine the merge sort algorithm and tested the multi-way merge sort with different value of parameters to sort files between 1×10^5 and 1×10^6 integers, and compared it with PriorityQueue in-memory sorting algorithm (equivalent when M>=file size).

Through the project, we have learned:

1.I/O operations have great effect on execution time.

2.Technique to implement the input/output stream using the Memory-Mapping.

3.A better understanding of the multi-way merge sort algorithm.

4.The file size will also influence the execution time through our experiment.