Database Systems Architecture

Algorithms in Secondary Memory

Project Assignment

2015-2016

Université Libre de Bruxelles École Polytechnique

Tzu-Jou Hsiao	000424157
Bahadır Han Akyüz	000423438
Gabby Nikolava	000426940

Table of Contents

1. Introduction and Environment

1.1. Outline

This report analyzes the performance of write/read file and external merge-sort in Java through graphs. The graphs compare the performance between four methods, and represent with changing one variable and fixing the others.

1.2. Motivation

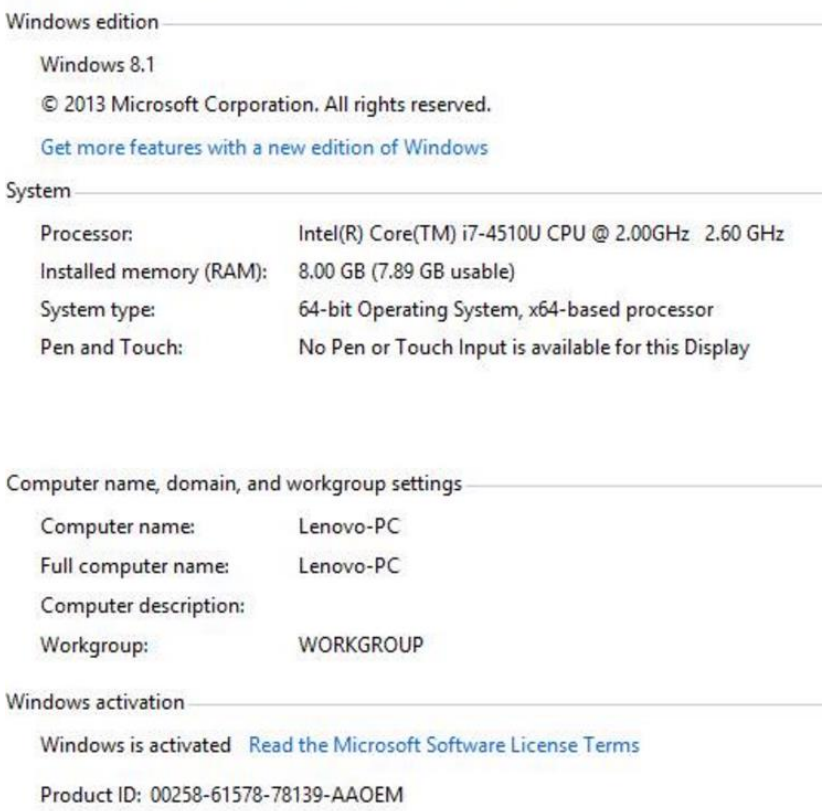
Reading and writing data from/to external memory is one of the most costly operations in computer transactions. Data contained in blocks moves at different speeds between memory hierarchies of the computer instead of same access cost. In order to have a better understanding on how different parameters influence the performance of an external memory related transaction, we will implement an external merge-sort (1) algorithm in Java, and then analyze the performance of this algorithm by changing the value of the parameters. Three parameters used are number of memory available (M), number of streams to merge in one pass (d) and the size of input file (N). Before implementing an external merge-sort, we consider Java has different ways to read/write file. First, we analyze four different methods of writing/reading integer file and then use the method with the best performance to implement the external merge-sort algorithm.

Those four methods are:

- FileInputStream/FileOutputStream with integer reads/writes
- BufferedInputStream/BufferedOutputStream with integer reads/writes
- BufferedInputStream/BufferedOutputStream with integer reads/writes and variable buffer size
- Memory Mapping with RandomAccessFile

1.3. Environment

The experiments to analyze the methods and algorithm are conducted on the following environment:



To support the implementation, the following Java libraries are needed:

I/O streams	**External Merge-Sort**
java.io.File; java.io.IOException; java.io.FileNotFoundException; java.io.RandomAccessFile; java.nio.MappedByteBuffer; java.nio.channels.FileChannel; java.io.FileOutputStream; java.io.BufferedOutputStream; java.io.DataOutputStream; java.io.OutputStream; java.io.DataInputStream; java.io.FileInputStream; java.io.InputStream; java.io.BufferedInputStream;	java.io.IOException; java.util.Comparator; java.util.ArrayList; java.util.Collections; java.util.PriorityQueue; java.util.Queue; java.util.Random; java.util.LinkedList;

Since the experiment to read and write the file is conducted separately (one method is used to read file and another method is used to write file), the test data was generated by executing the method to write file first and then the read method will use the same file generated to be read. The integer file in test data are generated randomly using Random() class with minimum integer value 0 and maximum integer value 1000.

```
public static ArrayList<String> generateRandomType1(String baseFileName,
                int numFiles, int maxLength) {
        ArrayList<String> fileNames = new ArrayList<String>();
        for (int i = 1; i <= numFiles; i++) {
                String outputFile = baseFileName + Integer.toString(i) + ".txt";
                ProjectOutputStream out = new ProjectOutputStream();
                try {
                        out.create(outputFile);
                        fileNames.add(outputFile);
                        // for each int in int buffer
                        for (int x = 0; x < maxLength; x++) {
                                // Generate random ( unsorted ) int 0 -1000
                                int next = intGenerator.nextInt(1000);
                                // write int to data output stream
                                out.write(next);
                        }
                        out.close();
```

The sizes of test data used in the experiment are 100, 10.000, 100.000, 1.000.000, 10.000.000.

2. Observation on Streams

2.1. Expected Behavior

FileInputStream/FileOutputStream with integer reads/writes

This method is implemented using FileInputStream and DataInputStream for reading file and FileOutputStream and DataOutputStream for writing file. The DataInputStream and DataOutputStream class allow the application to read and write one integer from/to file each time. Using this method, each time we read or write the disk is accessed to get the integer value. Therefore, we assume that this method will give the worst performance compare to other method.

```
public void open (String fileName) throws FileNotFoundException{
        try{
                if(null == is){
                        is = new FileInputStream(new File(fileName) );
                        ds = new DataInputStream(is);
                }
        } catch (Exception e) {
                System.out.println(e.toString());
                throw e;
        }
```

BufferedInputStream/BufferedOutputStream with integer reads/writes

This method is implemented using BufferedInputStream and BufferedOutputStream to read/write file. The implementation is basically similar to the first method in the sense that we still use FileInputStream, FileOutputStream, DataInputStream and DataOutputStream class. The difference is that when we use BufferedInputStream and BufferedOutputStream, a buffer will be created in the memory. When reading or writing the integer will be buffered in the memory until it reaches certain buffer capacity (default buffer size: 8 kB).

```
public void open (String fileName) throws FileNotFoundException{
        try{
                if(null == is){
                        is = new FileInputStream(new File(fileName) );
                        bis = new BufferedInputStream( is );
                        ds = new DataInputStream(bis);
                }
        } catch (Exception e) {
                System.out.println(e.toString());
                throw e;
        }
```

Since this method uses buffer, it means disk is not accessed as often as the first method. Hence we expect that the second method will have better performance compared to the first method.

BufferedInputStream/BufferedOutputStream with integer reads/writes and variable buffer size

The third method is the same with the second method with additional parameter buffer size. With this method, we can decide the size of the buffer and during the experiment we will analyze how different buffer size influences the performance.

```
public void open (String fileName) throws FileNotFoundException{
        try{
                if(null == is){
                        is = new FileInputStream(new File(fileName) );
                        bis = new BufferedInputStream( is, buffer );

                        ds = new DataInputStream(bis);
                }
        } catch (Exception e) {
                System.out.println(e.toString());
                throw e;
        }
```

Considering that we can define the buffer size, compared to the second method, the third method is expected to give a better performance if we put buffer size > 8 kB because it's the default value in Java.

Memory Mapping with RandomAccessFile

Memory Mapping is a method of read/write data by mapping a portion of file into the memory which skipped some levels in memory hierarchy. Each byte in the mapped memory is correlated to the corresponding byte in the disk. This allows the application to read or writes to disk while reading or writing in memory, which will increase I/O performance significantly. The disk is accessed every time a new mapping is done. Since we can only map a certain portion of file at one time, we have to consider the file size in order not to over map the remaining file portion. Considering this reason, we expect that this method will give the best performance amongst other methods.

In our experiment, memory mapping is implemented using RandomAccessfile and FileChannel class as shown in the following code to read file.

```
public void open (String fileName) throws FileNotFoundException{
    try{
        if(null == rafi){
            rafi = new RandomAccessFile( new File(fileName),"r");
            fci = rafi.getChannel();
            mbbi =fci.map(FileChannel.MapMode.READ_ONLY, 0, Math.min(buffer,fci.size()));
        }
    } catch (Exception e) {
        System.out.println(e.toString());

    }


}

public int read_next() throws IOException{
    return mbbi.getInt();
}

public boolean isEnd() throws IOException{
    return mbbi.hasRemaining();
}
```

The following are the codes for writing file.

```
public void create (String fileName) throws FileNotFoundException{
    try{
        rafo = new RandomAccessFile( new File(fileName),"rw");
        fco = rafo.getChannel();
        mbbo =fco.map(FileChannel.MapMode.READ_WRITE, 0, buffer);
    } catch (Exception e) {
        System.out.println(e.toString());
    }


}

public void write(int x) throws IOException{
    mbbo.putInt(x);
}

public void close() throws IOException{
    mbbo.force();
    rafo.close();
    fco.close();

}
```

The following table summarizes the expected performance for all the different types of I/O

Method	**Data Stream**	**Buffered**	**Buffered with variable buffer**	**Mapped with variable buffer**
Performance	4th	3rd	2nd	1st

2.2. Experimental Observations

To analyze which method delivers the best performance, we conduct the experiment in three parts. In the first part we vary the number of stream (k), in the second part we vary the buffer size (B), and in the third part we vary the number of integer read/written (N).

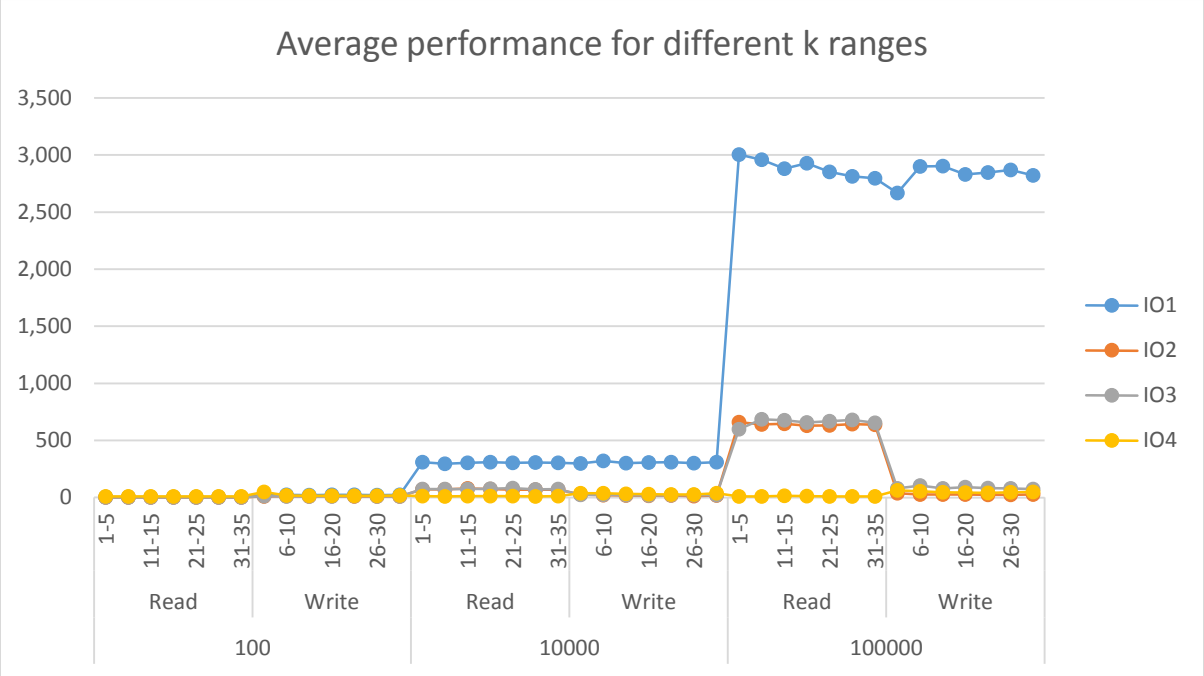
2.2.1. Various Streams Numbers (k)

We try to show how different numbers of k changes performance of reading/writing for IO methods. Firstly, we would like to give an overall idea to show which the best option for IO operation is. In order to do that, we make tests for changing "k" values from 1 to 35. We also wanted to see effects of buffer size and integer number in a file. We measured all of those parameters, firstly we would like to show which is the roughly best for average conditions in our assumption.

- K values: 1 to 35.
- File Size (integer number in a file): 100,1000,10000.
- Buffer Size: 8, 512, 1024, 2048, 4096, 8192, 16384.

In order to make analysis more understandable, we used ranges from different k values such as 1-5, 6-10… 31-35.
While we get logs, we compute values for 4 times and obtain the averages to receive more appropriate results and get rid of temporal performance issues.

Average elapsed time (milliseconds) for each file to read/write and different k ranges are seen respectively on y axis and x axis.

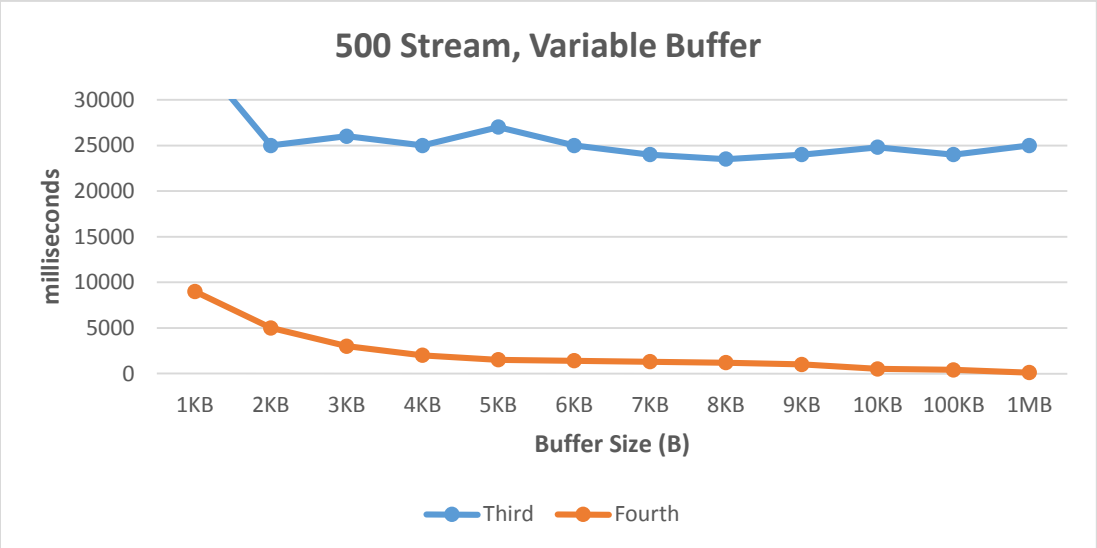


We can say that IO4 is the best option for reading and IO3 is the best options for overwriting. On the other hand, it is easily seen that first IO method doesn't perform well in all cases. However, we have to mention that on above graph either k (file number), N (file size) or B (buffer size) are variables, and graph just shows averages to give an overall idea.

2.2.2. Various Buffer Sizes (B)

Read File: Variable B, Fixed k and N

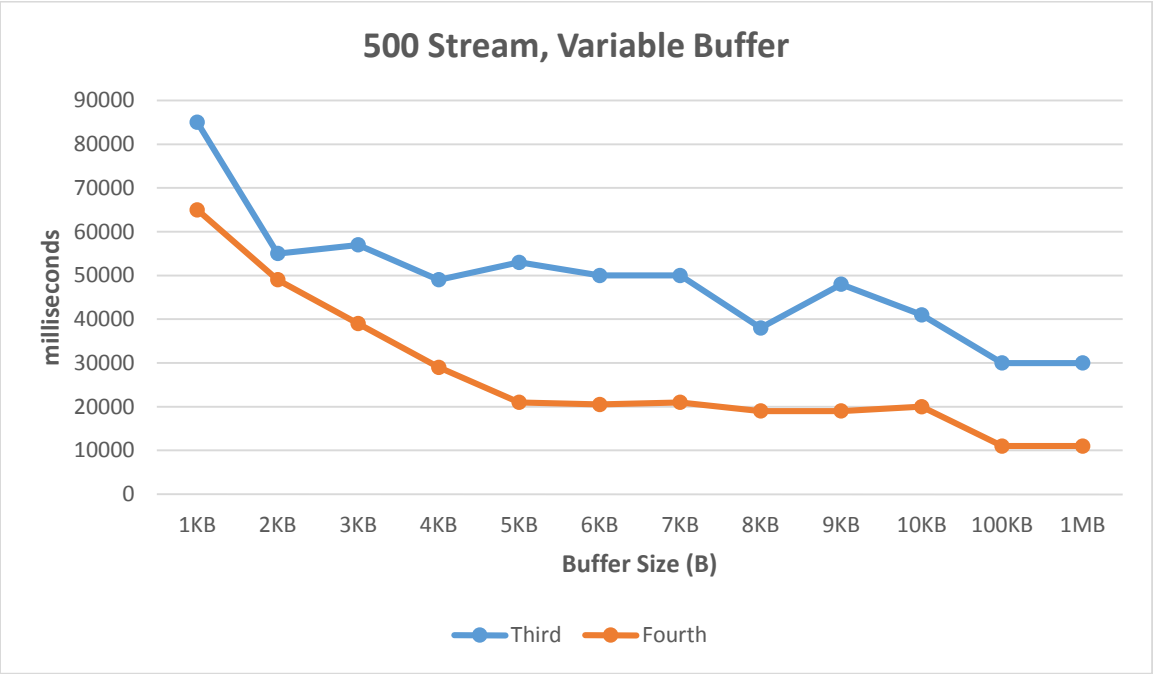
In this second part we use k = 500 and N = 2.562.144 (equal to 1 MB file size). Since only the third, fourth method use buffer, we exclude first and second method.



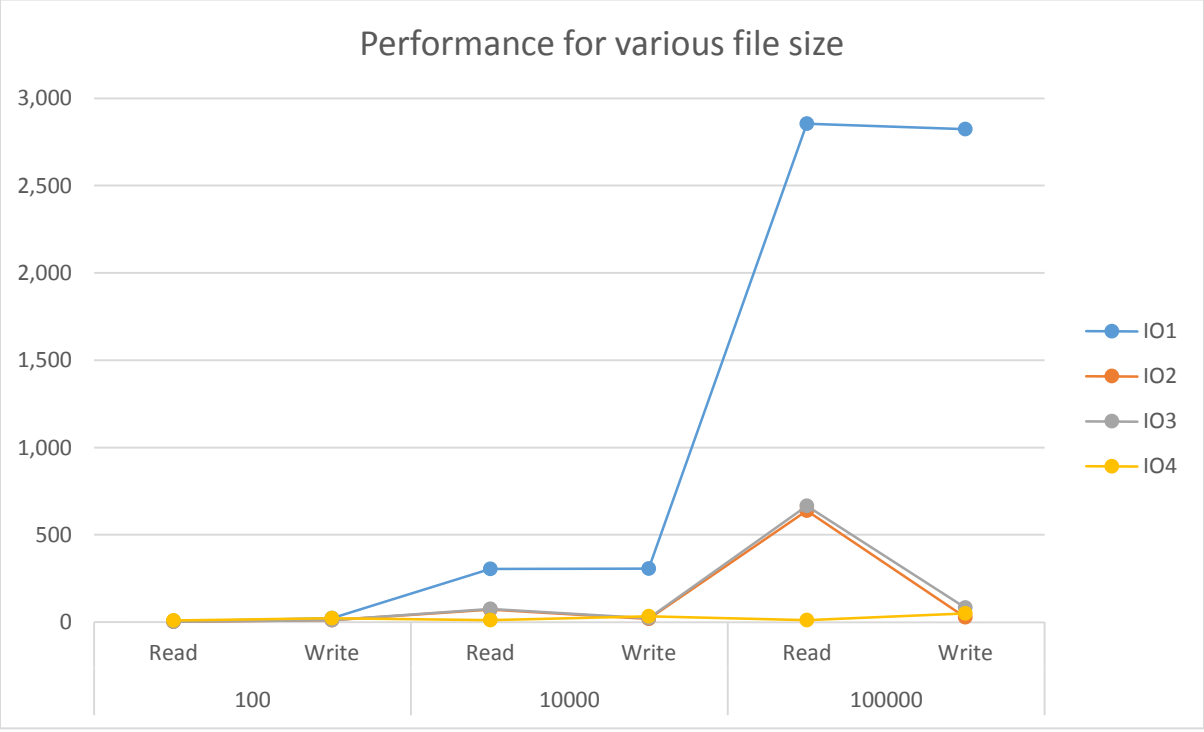
Write File: Variable B, Fixed k and N

In this second part we also use the same parameter value: k = 500 and N = 2.562.144 (equal

to 1 MB file size). Since only the third and fourth method use buffer, we exclude first and second method.



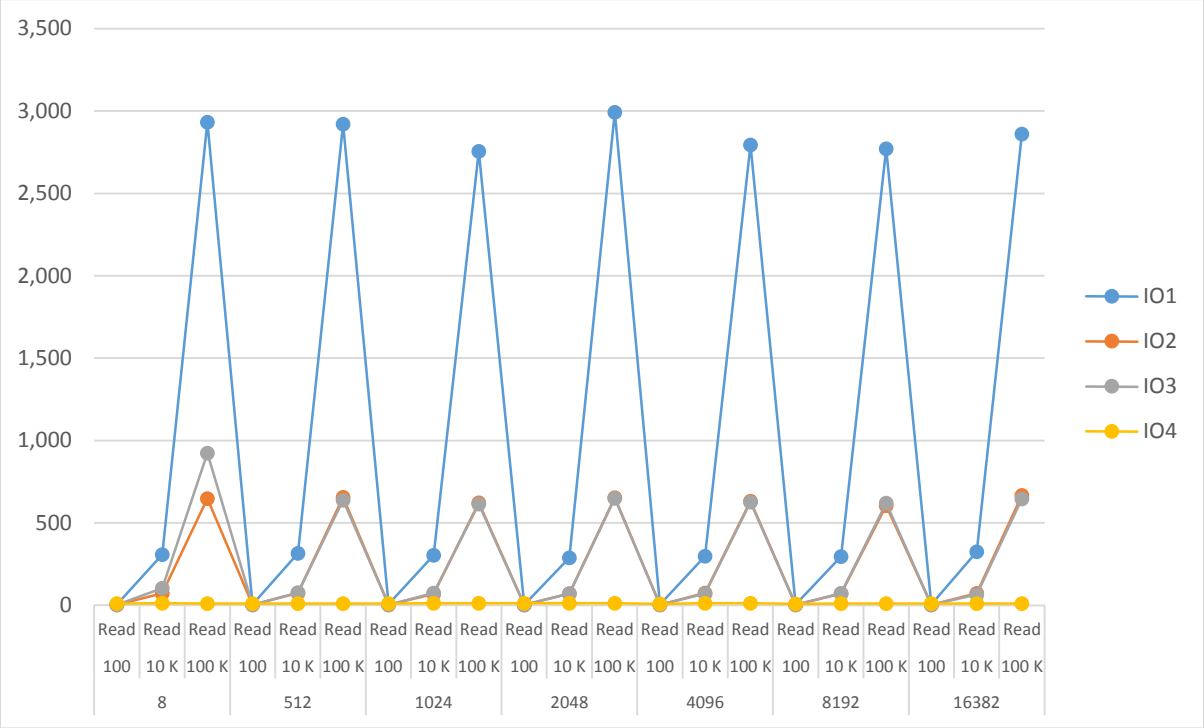
2.2.3. Various File Sizes (N)



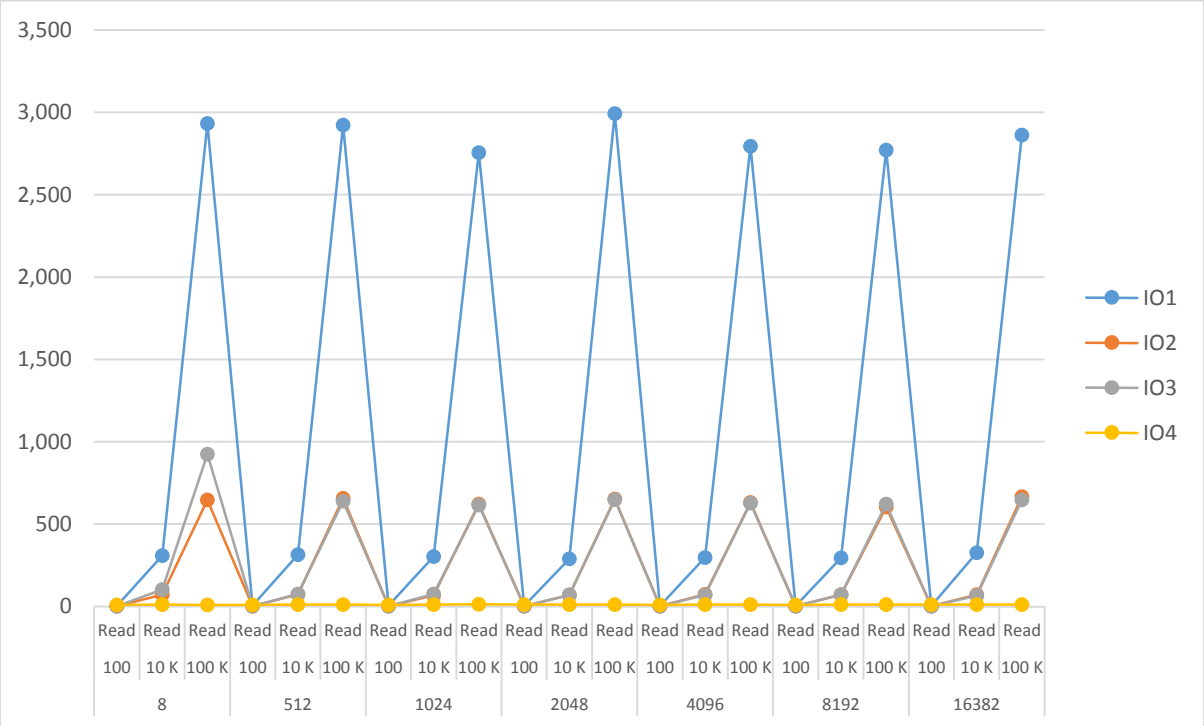
If we want to show how IOs work on different sizes of files, we can check above graph. For relatively small amount of data (100 row on left) each of methods gives almost same values that means we can use any of them. Once we have bigger files, we can see disadvantage of first method. For writing data even for the big files method 2 and 3 are preferable with IO4 but for reading IO4

should be used. We should mention that buffer sizes and k values are also varying in that graph and values shows the average.

To see the effect of the buffer sizes we can check below graph (reading and writing different graphs are created):



Now, we can see that, buffer size has not big impact on those processes. Just for really small buffer size (8 on left size) IO2 is closer IO4 than IO3. That was one of our expectations.



On the other hand, for writing data into a file, Buffer Size matters. When we have a modern computer (assume that buffersize >8) methods 2, 3, 4 are similar for performance issue and this situation is independent of files size.

2.3. Discussions

The following table summarizes the ranking for each method based on the variable parameter used during the experiment compared to the ranking expected:

Method	First	Second	Third	Fourth
Expected	4th	3rd	2nd	1st
Read File: Variable k, Fixed B,N	4th	2nd	3rd	1st
Write File: Variable k, Fixed B,N	4th	1st	3rd	2nd
Read File: Variable B, Fixed k,N	-	-	2nd	1st
Write File: Variable B, Fixed k,N	-	-	2nd	1st
Read File: Variable N, Fixed k,B	4th	2nd	3rd	1st
Write File: Variable N, Fixed k,B	4th	1st	3rd	2nd

As expected, in all experiment the first method always delivers the worst performance. On the other hand, the fourth method which is expected to give the best performance in all experiment apparently doesn't always come first. We noticed that with small buffer size (1kB) memory mapping comes second for reading file and comes third for writing file. In the second part of the experiment, when variable buffer size is increased to 2 kB, fourth method starts to give the best performance. Even after we increase the buffer size up to 1 MB, fourth method always comes first. This result is the same for both writing and reading file.

Besides, we noticed that the second method which we expect to give the second worst performance in all experiment comes first for writing file in experiment with variable k and N. Since we exclude the second method in the second part of the experiment, we don't know how it is actually ranked compared to the third, fourth method. But considering that for the second part of the experiment we use k=500 and N=2.562.144, we can predict that the second method will not give a performance better than 5000 milliseconds (performance of fourth method for reading file). This is also the same for writing file: the second method will not give better performance than 20.000 milliseconds.

Based on the experiment, we can conclude that even though memory mapping doesn't always give the best performance, it comes first for reading and writing file with big number of integer file and a lot

of stream opened at the same time (500 streams) if we use buffer size more than 2 kB. Using bigger buffer size does increase the performance but it is not significant. Once the buffer size reached 4 kB, the performance starts to stabilize. With these reasons, we decide to use memory mapping for the external merge-sort algorithm implementation.

3. Observations on Multi-way Merge Sort

3.1. Expected Behavior

The External Merge-Sort algorithm is implemented in Java in the following steps:

- Split file:

Input file with size N is split into several smaller file with size M (size of the memory available). After splitting we will have (N/M) smaller file.

- Merge and Sort the sub file:
  a. Load the first d (number of stream to merge in one pass) streams.
  b. Merge the integers. Since the integers are already sorted in each stream, we need to compare the integer in each stream before merging them. The first integers in each stream are compared. The smallest integer is then written to the buffer. Each time the smallest integer is identified, the pointer of the stream where the integer is in is moved to the next integer. Compare the integers again.
  c. When the buffer is full, write the merged integers to a new file.
  d. Repeat step b and c until there is no integer left in each stream. Load the next d streams and repeat the process.
  e. Each time we load a file, we remove the filename from the queue. Each time we write the merged integers, the name of this new file is added to the queue as reference. Repeat a, b, c, and d until there is only one file left in queue.

From this implementation, we expect that the bigger the size of the file (N), the more the I/Os. Since bigger file size require more reading and writing process. On the other hand, the more memory we have, then the I/O operations will be smaller considering that we read and write less from/to the disk. The more number of stream to merge in one pass, then the smaller the I/O will be because there will be less reading and writing operation from/to disk.

The total cost of the operation is (1): **$2B(R)\lceil \log B(R) \rceil$**which can be written as **$2N\lceil \lceil \log_M MN \rceil$**. This formula means that the cost of sorting operation equal to two times of number of integer (one for reading from disk and one for writing it back to disk) times the number of passes required. It suggests that the cost of sorting operation is linear to the value of N and M.

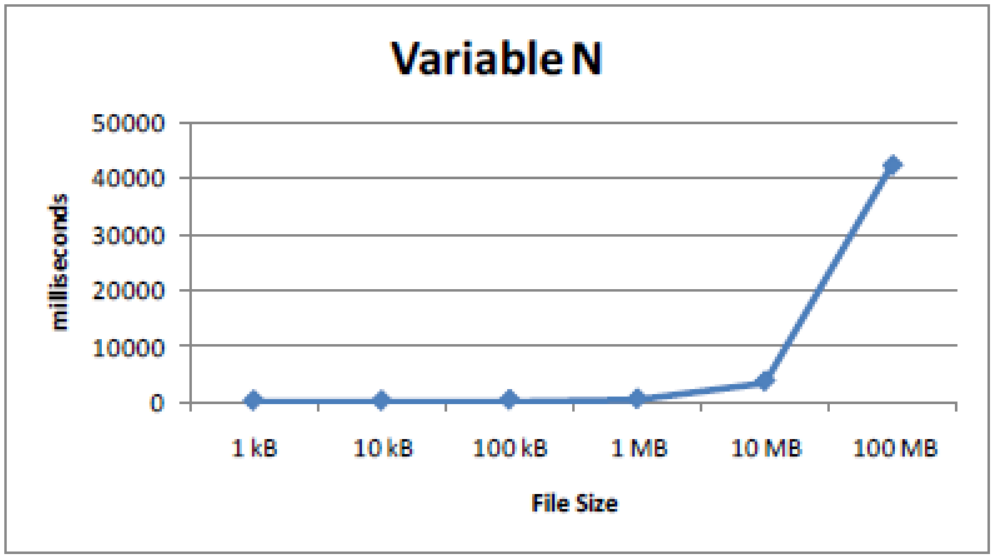
3.2. Experimental Observations

3.2.1. Graph Representation

We conduct the experiment in three parts. In the first part we vary the file size (N), in the second part we vary number of memory available (M), and in the third part we vary the number of streams to merge in one pass (d).

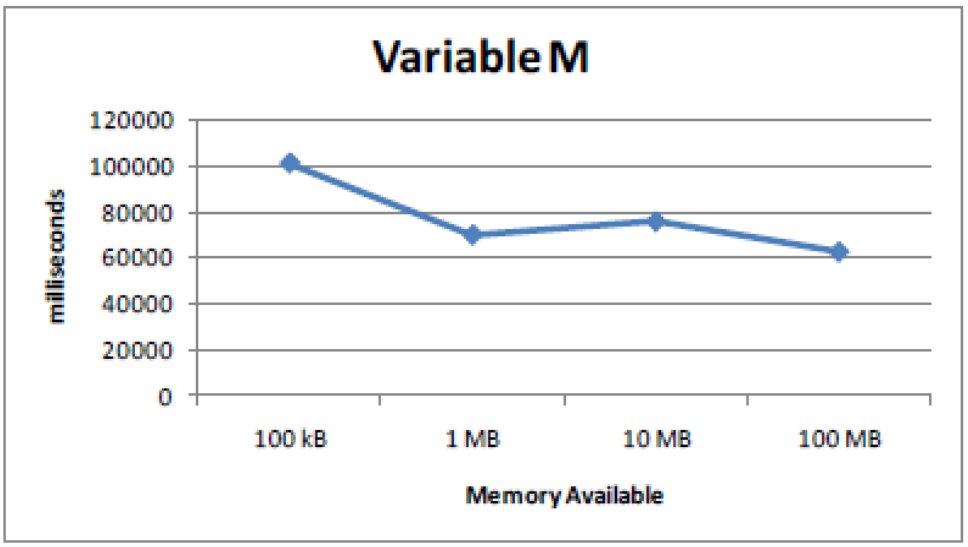
Variable N, Fixed M and d

In the first part we use M = 1 MB and d=10. We initially planned to experiment with file size 200 MB, 500 MB and 1 G as well. But when we run the application for 200 MB, it gives an out of memory error message. As shown in the following graph, we observe that the bigger the file size, the more time it needs to sort the file.



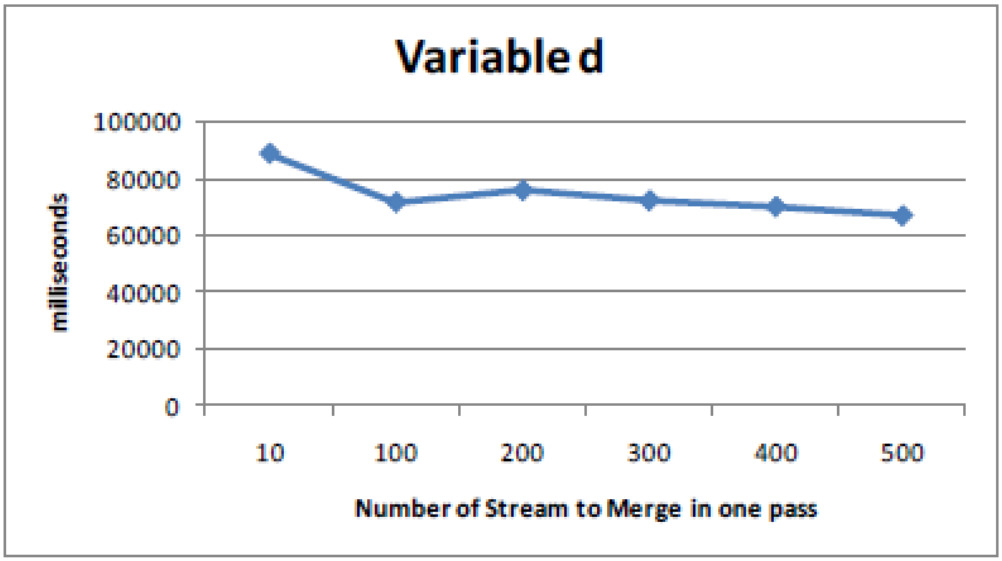
Variable M, Fixed N and d

We use N = 100 MB and d = 100. We planned to experiment with M = 1 kB and M = 10 kB. But it gives an out of memory error message. In this experiment we see that when we add the memory from 100 kB to 1 MB, the process finish faster. But when we add the memory to 10 MB, it doesn't give a significant difference.



Variable d, Fixed M and N

For this experiment, we use N = 100 MB and M = 1 MB. From the following graph when adding the amount of stream to merge from 10 to 100, the merge sort process run faster. But if we continue adding d, the line start to stabilize indicating that d affect the performance but not very significantly.



3.3. Discussions

In the experiment, the time needed to complete a merge-sort process represents disk I/O. As mentioned in (1), the number of passes: $\lceil \log_M B(R) \rceil$ can be written as $\lceil \log_M N \rceil$ where N is the file size or the number of integer available in the file.

This statement is proven by the experiment in 3.2 where we vary N as expected.

4. Overall Conclusion

By implementing the External Merge-Sort algorithm, we have a real world experience on how different parameter (M, N and d) affect the cost of sorting operation. When facing the similar scenarios, we know which Java class is suitable. For example, many Java examples use memory mapping deal with Client/Server environment, especially for quicker transition speed of big data.

The experiment for choosing the best read/write file method has also help to understand that indeed disk access contribute the most to operation cost. The more we access the disk means the more cost will be needed which is depicted by the more time we need to complete the operation.

5. References

1. Memory Mapped File in Java, from

http://www.codeproject.com/Tips/683614/Things-to-Know-about-Memory-Mapped-File-in-Java

http://howtodoinjava.com/2015/01/16/java-nio-2-0-memory-mapped-files-mappedbytebuffer-tutorial/

2. K-way Merge,from

http://www.sinbadsoft.com/blog/sorting-big-files-using-k-way-merge-sort/

http://stackoverflow.com/questions/5055909/algorithm-for-n-way-merge

3. Sorting and Priority Queues, from

http://algs4.cs.princeton.edu/20sorting/

https://github.com/vbohush/SortingAlgorithmAnimations/blob/master/src/net/bohush/sorting/MergeSortPanel.java

4. Merge-Sort, from

http://interactivepython.org/runestone/static/pythonds/SortSearch/TheMergeSort.html

http://www.cs.umd.edu/~meesh/351/mount/lectures/lect6-divide-conquer-mergesort.pdf

5. Included code in java:

http://www.vogella.com/tutorials/JavaAlgorithmsMergesort/article.html#mergesort

6. Memory Mapped I/O:

http://www.developer.com/java/other/article.php/10936_1548681_4/Introduction-to-Memory-Mapped-IO-in-Java.htm