ALGORITHMS IN THE SECONDARY MEMORY

INFO-H 417 Database System Architecture

Group 14

Martina Megasari	(000439159)
Pandu Wicaksono	(000438477)
Shibo Cheng	(000438640)

Table of Contents

1 Introduction and Environment

The project goal is to implement an external-memory merge-sort algorithm and examine its performance under different parameters. The project mainly focuses on implementing and testing two parts. The first part is the input and output streams. We implement and test four different input/output streams which use read/write functions, fread/fwrite functions, fread/fwrite functions with buffer size and memory mapping. The best performed input/output streams are selected after testing. In the second part, the external multi-way merge-sort algorithm uses the streams selected from the first part to read and write data to disk. The algorithm is implemented as the one described in Section 15.4.1 of "Database Systems the Complete Book". The details of streams and algorithm are discussed in later parts.

This table below summarizes our machine testing environment:

Machine Type	ASUS UX302LG
Hard Disk Type	Hybrid Storage: - Operating System: 14GB SanDisk SSD U110 16GB (SSD) - Data Disk: 278GB Western Digital WDC WD7500LPCX-80HWST0 (SATA)
Operating System	Windows 10 Home Single Language 64-bit
CPU	Intel® Core™ i7-4500U @ 1.80GHz Haswell ULT 22nm Technology
RAM	8.00GB Dual-Channel DDR3 @ 798MHz (11-11-11-28)
System Type	64-bit Operating System, x64-based processor
Total Memory Available	72GB
Programming Language	Java
Java Version	jdk1.8.0_112
Libraries Used	- Java IO - Java Util - Java Lang Management - Java New Input Output (NIO)
External Library Used	-

Java 5 introduced the java.lang.management package and methods to report CPU, user and system time per thread to benchmark application. [1]

- "User time" is the time spent running application's own code in user space.
- "System time" is the time spent running OS code on behalf of your application in kernel space, for example, it can be I/O time caused by application's system calls.
- "CPU time" is the total time spent using a CPU running the application which is user time plus system time.

Before Java 5, the conventional way to benchmark a task is java.lang.System.currentTimeMillis. It calculates the time difference between start and end of the task, which is the real-world elapsed time experienced by a user waiting for a task to complete. This timing method is strongly affected by other applications, background processes and so on. [1] Therefore, we choose the timing methods from java.lang.management to exclude those effects. The testing in the project focuses on application's I/O cost and total running time, hence "System time" and "CPU time" are chosen as the benchmark. In our program, we calculate the user time and system time for the four stream implementations and merge-sort to compare and analyze each implementation's I/O time cost and total time cost. For the report, we count the CPU Time (CPU Time = user time + system time), use system time and CPU time, and convert the time into milliseconds.

We generate our test data using java.util.Random as implemented in class FileGenerator.java. The range of the integer is [MIN_INT_VAL, MAX_INT_VAL]. The file size of the data for Stream Test is approximately 128MB (32,000,000 integers) and we prepared 30 different files for testing maximum 30 number of streams (maximum k = 30). During Stream Test, we only read N number of integer from the file as specified in the parameter for testing. For External Multi-way Merge Sort Test, the file size of the data varies because for this test the input is only one file. The table below summarizes our files during our testing:

File Name	File Size in MB	Number of Integer in File
Input15625.data	0.0625	15,625
Input31250.data	0.125	31,250
Input62500.data	0.25	62,500
Input125000.data	0.5	125,000
Input250000.data	1	250,000
Input500000.data	2	500,000
Input1000000.data	4	1,000,000
Input2000000.data	8	2,000,000
Input4000000.data	16	4,000,000
Input8000000.data	32	8,000,000
Input16000000.data	64	16,000,000
Input32000000.data	128	32,000,000

2 Observations on Stream

The four kinds of input streams can open an existing file, read the next element from the stream and check if the end of stream has been reached. The output streams have functions of creating a new file, writing and element to the stream and closing the stream. We will discuss the details of the four types streams and test their performance by calculating the execution time of reading one element and writing one element. The parameters for each stream's cost formula that estimates the total number of I/Os are N - the number of integers need to be read or write, k - the number of streams to read/write, B - the number of integers in the buffer.

2.1 Expected Behavior

The project implements four types of streams. Each type of streams includes an input stream and an output stream. The four input streams can open an existing file, read the next element from the stream and check if the end of stream has been reached. The output streams have functions of creating a new file, writing an element to the stream and closing the stream.

Their performances are tested by calculating the execution time of an operation that reads one element and writes one element repeatedly under different number of integer, number of streams and buffer size. Therefore, we define the parameters for each stream's cost formula that estimates the total number of I/Os as follows:

- N - the number of integers need to be read or write,
- k - the number of streams to read/write,
- B - the number of integers in the buffer (for Stream 3) or number of mapped element (for Stream 4).

In general, the time needed to perform the task is expected to be increasing when we increase N or k; and decreasing when we increase B.

2.1.1 Stream 1 – read / write Functions

For read/write system calls, every time the program needs to read and write it will go to the disk to obtain the data. For each element it read one time and write one time. Also, the program needs to deal with N number of integer. Hence the cost function for this stream implementation is:

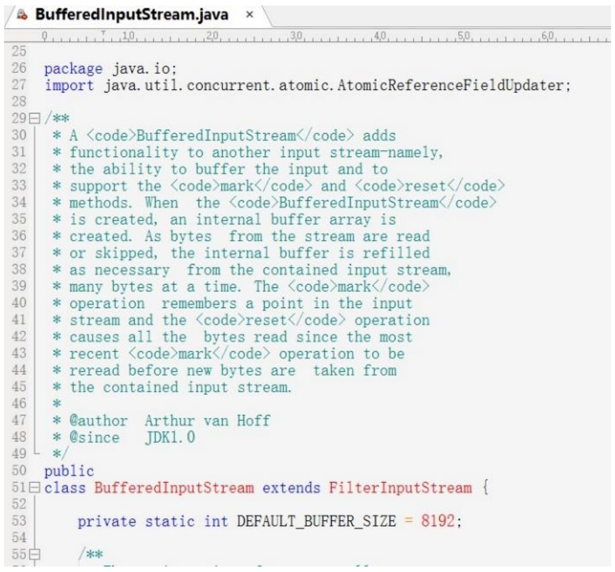
$$C = 2 * N * k$$

We expect that the time needed to perform the task is increasing when we increase N or k.

2.1.2 Stream 2 – fread / fwrite Functions

In our Java project, fread/fwrite functions are mimicked by readInt() and writeInt() on java.io.BufferedInputStream and BufferedOutputStream. When the stream is created, an internal buffer array is created. [2]

For BufferedInputStream, the read() method reads default amount of bytes into buffer at one time until the bytes are needed. For BufferedOutputStream, the write() method writes default amount of bytes into the buffer and flushes to file on the disk when the buffer is full. According to the source code of constructor in JDK, the default buffer size is 8192 bytes, equals to 2048 integers. (C:\Program Files\Java\jdk1.8.0_112\src.zip\java\io\ BufferedInputStream.java)



```

package java.io;
import java.util.concurrent.atomic.AtomicReferenceFieldUpdater;

/**
 * A <code>BufferedInputStream</code> adds
 * functionality to another input stream-namely,
 * the ability to buffer the input and to
 * support the <code>mark</code> and <code>reset</code>
 * methods.  When  the <code>BufferedInputStream</code>
 * is created, an internal buffer array is
 * created. As bytes  from the stream are read
 * or skipped, the internal buffer is refilled
 * as necessary  from the contained input stream,
 * many bytes at a time. The <code>mark</code>
 * operation  remembers a point in the input
 * stream and the <code>reset</code> operation
 * causes all the  bytes read since the most
 * recent <code>mark</code> operation to be
 * reread before new bytes are  taken from
 * the contained input stream.
 *
 * @author  Arthur van Hoff
 * @since   JDK1.0
 */
public
class BufferedInputStream extends FilterInputStream {

    private static int DEFAULT_BUFFER_SIZE = 8192;

    /**
```

Hence the cost function for this stream is:

$$C = 2 * \frac{N}{2048} * k$$

We expect that the time needed to perform the task is increasing when we increase N or k.

2.1.3 Stream 3 – fread / fwrite Functions with Buffer Size

The third implementation is similar to the second except the buffer size B is not fixed to default 8192 bytes / 2048 integers but changed according to parameter in the test. Hence the cost function is:

$$C = 2 * \frac{N}{B} * k$$

Based on our research on BufferedInputStream, Java set default buffer size to 8192 bytes because they found 8192 bytes is a "point of diminishing return" at which increasing the buffer size does not improve either overall read time or CPU usage [3]. Hence, we expect that the time needed to perform the task is increasing when we increase N or k; and decreasing when we increase B until the point of diminishing return (B = 2048 integer). Thus, we expect that the performance after B is larger than the point of diminishing return is not improving again.

2.1.4 Stream 4 – Memory Mapping

Memory mapping is a method which maps whole file or portion of file into virtual memory, this part of memory is called memory-mapped file which has a direct byte-for-byte correlation with the whole or portion of the file. [4]

The application deals with the memory mapped file like in the memory although the file is not actually copied into the Java heap space in memory like BufferedInputStream does. When a file is read by buffered I/O, BufferedInputStream needs to Invoke the system call into the kernel for the read operation, allocate memory for buffer in both user space and kernel space. The file will be copied twice, once from disk to kernel space and once from kernel space to user space. Though copying speed in memory is much faster than I/O with disks, memory mapping can be faster by copying only once with no overhead of the system call. Operating system copies the disk sector associated with the file into the physical memory pages when page fault happens, reads the file data as if reading data from a memory byte buffer. [5]

The buffer size B here is the number of element to be mapped into internal memory namely the number of elements to be copied into physical memory pages by operating system. Hence the cost function is:

$$C = 2 * \frac{N}{B} * k$$

Unlike Stream 3, we haven't found any point of diminishing return for Stream 4. Hence, we can increase the number of B to improve the performance as long as we have sufficient memory in our machine. Our expectation for Stream 4 is time needed to perform the task is increasing when we increase N or k; and decreasing when we increase B.

2.2 Experimental Observations

To further test the capabilities of each implementation, we test each implementation using variable N (number of integers), B (number of buffer element), and k (number of stream). To understand how each variable affects the performance, we test the implementation by varying the value of the chosen variable and lock the value of other variables. For testing the number of N, we lock B by 1 or the optimum B value from the B testing result and we lock k by 1. For testing the number of B, we lock the N by Small Input (N = 500,000) and Large Input (N = 32,000,000) and k by 1. For testing the number of k, we lock the N by Small Input and Large Input then we lock B by the optimal value of B from the B testing result. Testing is done 10 times then we calculate the average in order to level out fluctuations due to other processes.

We decide to choose the variable of N by determining the file size first then dividing the file size by 4 which is the size of one integer. We choose the power of 2 to decide the number of file size. In our opinion the file size by the power of 2 is sufficient to give us a clear understanding of how N is affecting the performance and to find which stream is the most performant. The limit of our testing based on the number of N to 32,000,000 because by using this number we can already found which stream is the most performant, hence increasing the value of N even further will not bring any new information from this testing. For testing variable B and k we use two values of N: Small Input (N = 500,000) and Large Input (N = 32,000,000). The table below represents our chosen N.

File Size in MB	File Size in KB	File Size in Byte	Integer
0.0625	63	62,500	15,625
0.125	125	125,000	31,250
0.25	250	250,000	62,500
0.5	500	500,000	125,000
1	1,000	1,000,000	250,000
2	2,000	2,000,000	500,000
4	4,000	4,000,000	1,000,000
8	8,000	8,000,000	2,000,000
16	16,000	16,000,000	4,000,000
32	32,000	32,000,000	8,000,000
64	64,000	64,000,000	16,000,000
128	128,000	128,000,000	32,000,000

For variable B, we choose the power of 2 starting from 64 (2^6) and the limit is determined by the number of Small Input and Large Input. For Small Input (N = 500,000) the limit is 524,288 and for Large Input (N = 32,000,000) the limit is 16,777,216. This is because in our opinion the power of 2 is sufficient to give us a clear understanding of how the number of B is affecting the performance and to find which stream is the most performant. The table below summarizes our chosen B:

B	$\log_2 B$
64	6
128	7
256	8
512	9
1,024	10
2,048	11
4,096	12
8,192	13
16,384	14

32,768	15
65,536	16
131,072	17
262,144	18
524,288	19
1,048,576	20
2,097,152	21
4,194,304	22
8,388,608	23
16,777,216	24

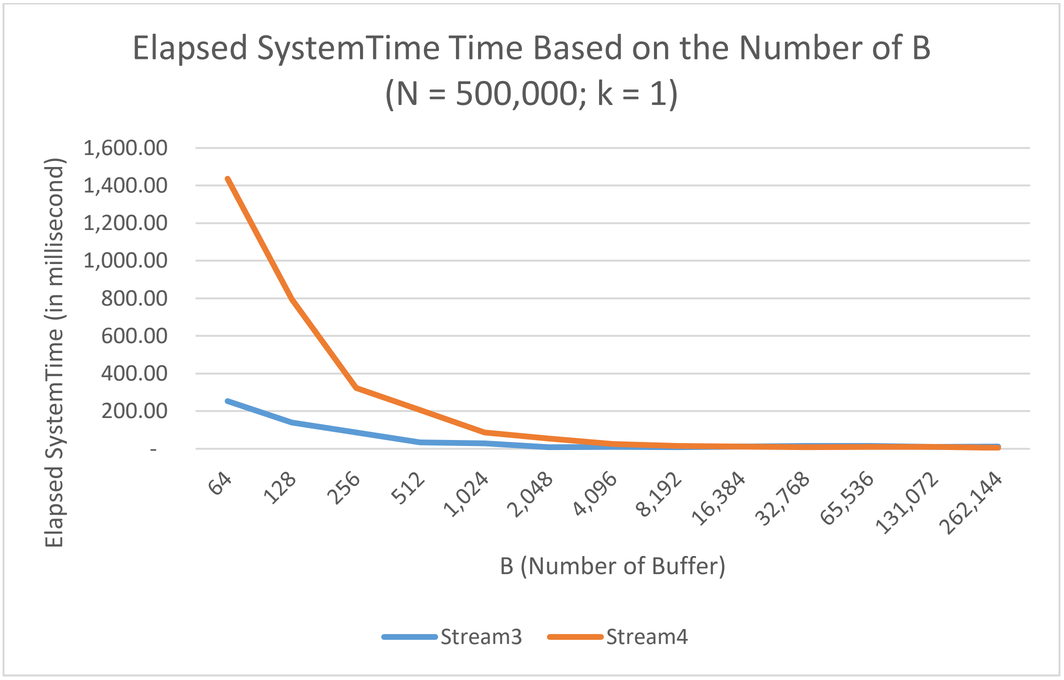
For variable k, we choose the number of k = 1, 10, 20, and 30. In our opinion this choice is sufficient to give us a clear understanding of how the number of k is affecting the performance and give a good level of performance benchmarking for each stream.

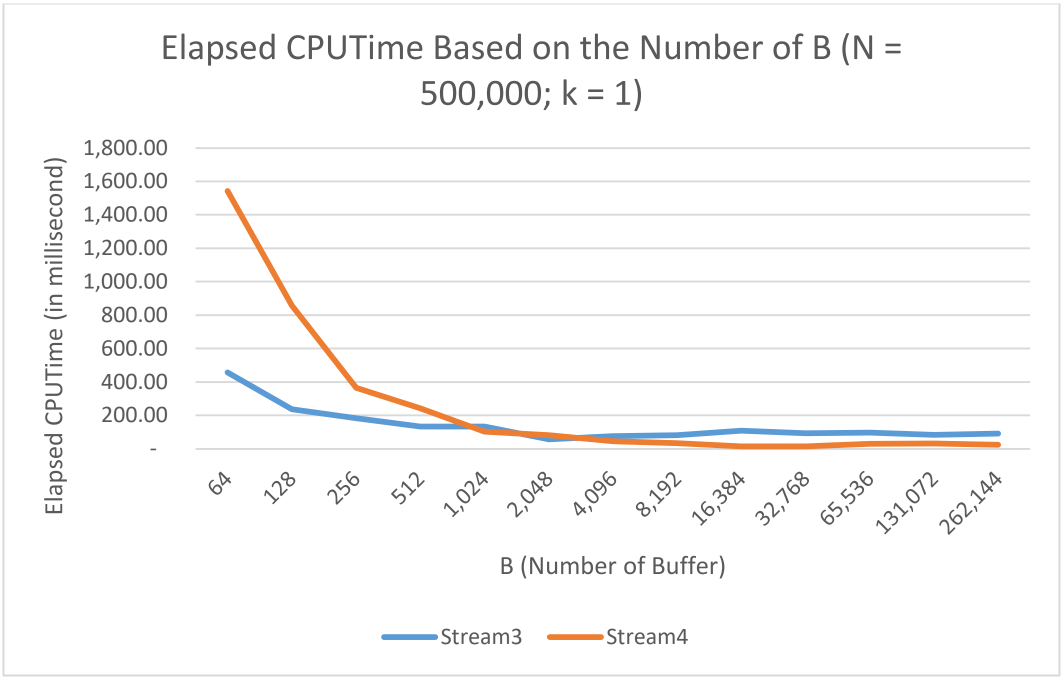
2.2.1 Based on the Number of B (Number of Buffer Element)

In our test, we perform the test for Small Input (N = 500,000) and Large Input (N = 32,000,000). For small input, we limit the number of B to 262,144 and for large input we limit it to 16,777,216. This is because higher number of B element will exceed the number of N which implies that all of the elements can be put directly into the buffer (for Stream 3) and can be directly map once (for Stream 4) hence in our opinion it is not giving us more insight about the effect of B to the performance.

2.2.1.1 Small Input

The information below is our testing result of variable B for Stream 3 and 4 using SystemTime and CPUTime.





B	SystemTime		CPUTime	
	Stream3	Stream4	Stream3	Stream4
64	253.13	1,435.94	457.81	1,542.19
128	139.06	793.75	235.94	857.81
256	85.94	323.44	182.81	365.63
512	34.38	206.25	134.38	242.19
1,024	28.13	85.94	134.38	103.13
2,048	7.81	53.13	56.25	81.25
4,096	9.38	25.00	76.56	45.31
8,192	7.81	14.06	81.25	34.38
16,384	10.94	10.94	107.81	15.63
32,768	14.06	7.81	93.75	14.06
65,536	15.63	9.38	96.88	29.69
131,072	9.38	9.38	82.81	31.25
262,144	10.94	4.69	92.19	23.44

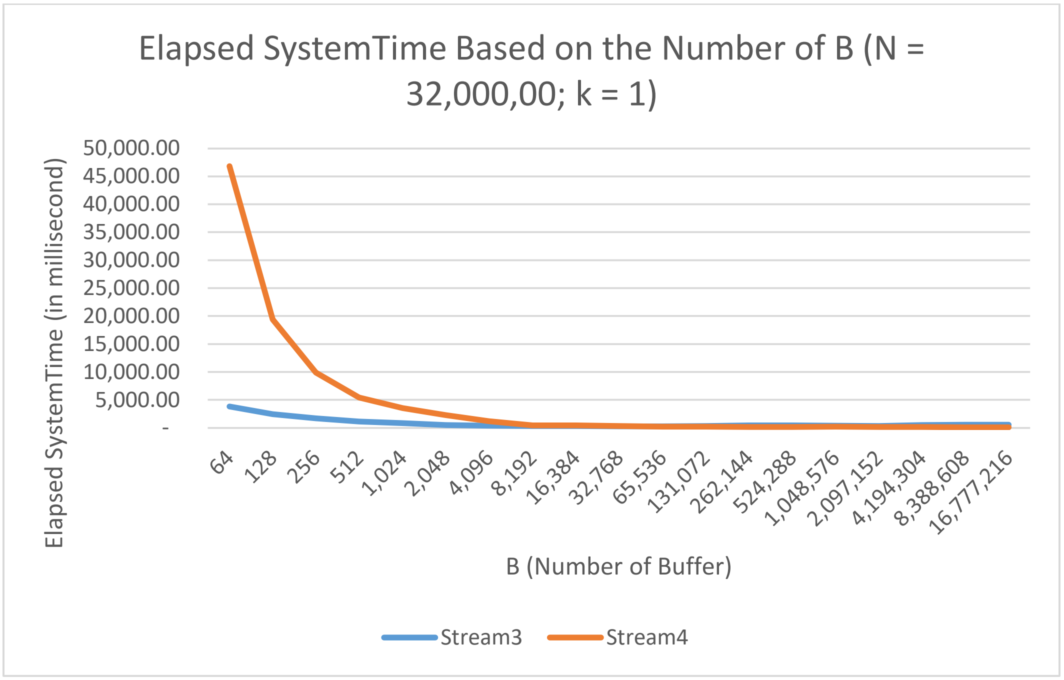
Based on our experiment, we derived the following points:

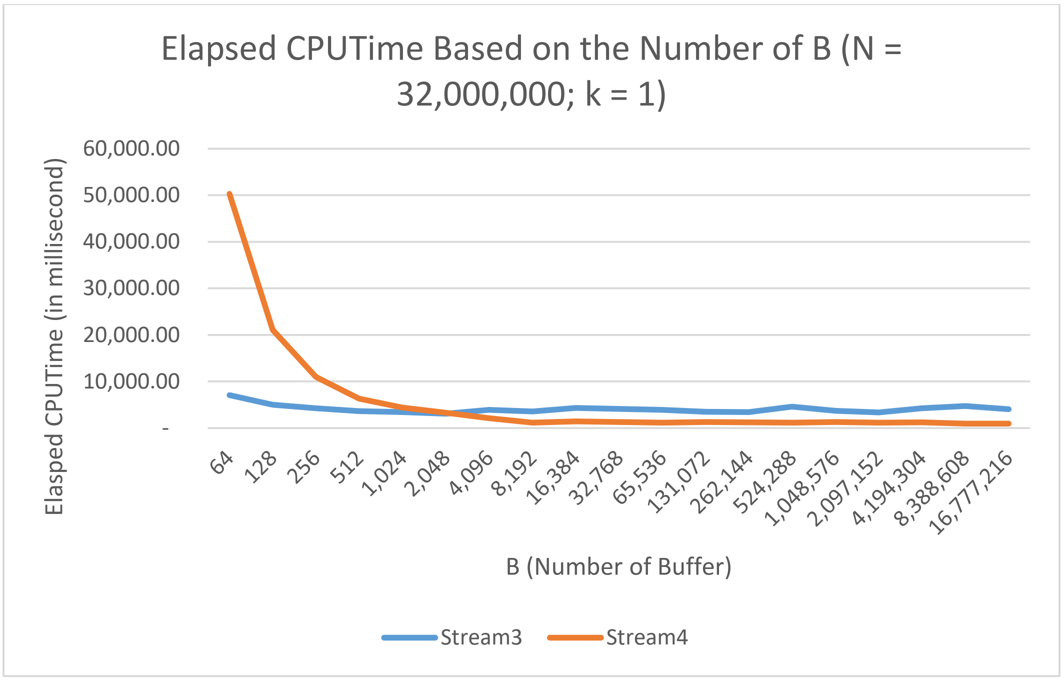
- For Stream 3, after B = 2,048 the performance is not improving that much.
- For Stream 4 after B = 32,768 the performance becomes worse.
- SystemTime and CPUTime give approximately the same trend to the result. However, the difference between SystemTime and CPUTime is large. This implies that in this test, the performance of each stream is highly affected by running application code in user space rather than the system calls (I/O).

- We concluded that the optimum value of B using Small Input is B = 2,048 for Stream 3 and B = 32,768 for Stream 4.

2.2.1.2 Large Input

The information below is our testing result of variable B for Stream 3 and 4 using SystemTime and CPUTime.





B	SystemTime		CPUTime	
	Stream3	Stream4	Stream3	Stream4
64	3,848.44	46,848.44	7,048.44	50,254.69
128	2,435.94	19,409.38	4,965.63	21,060.94
256	1,731.25	9,875.00	4,265.63	10,957.81
512	1,153.13	5,410.94	3,592.19	6,325.00
1,024	853.13	3,514.06	3,410.94	4,379.69
2,048	487.50	2,259.38	3,043.75	3,254.69
4,096	400.00	1,190.63	3,870.31	2,089.06
8,192	312.50	468.75	3,543.75	1,126.56
16,384	335.94	453.13	4,290.63	1,437.50
32,768	284.38	335.94	4,087.50	1,273.44
65,536	245.31	206.25	3,867.19	1,184.38
131,072	306.25	189.06	3,473.44	1,321.88
262,144	448.44	142.19	3,448.44	1,209.38
524,288	443.75	165.63	4,579.69	1,128.13
1,048,576	404.69	185.94	3,681.25	1,279.69
2,097,152	354.69	134.38	3,323.44	1,121.88
4,194,304	500.00	154.69	4,251.56	1,200.00
8,388,608	559.38	90.63	4,695.31	971.88
16,777,216	539.06	104.69	4,046.88	914.06

Based on our experiment, we derived the following points:

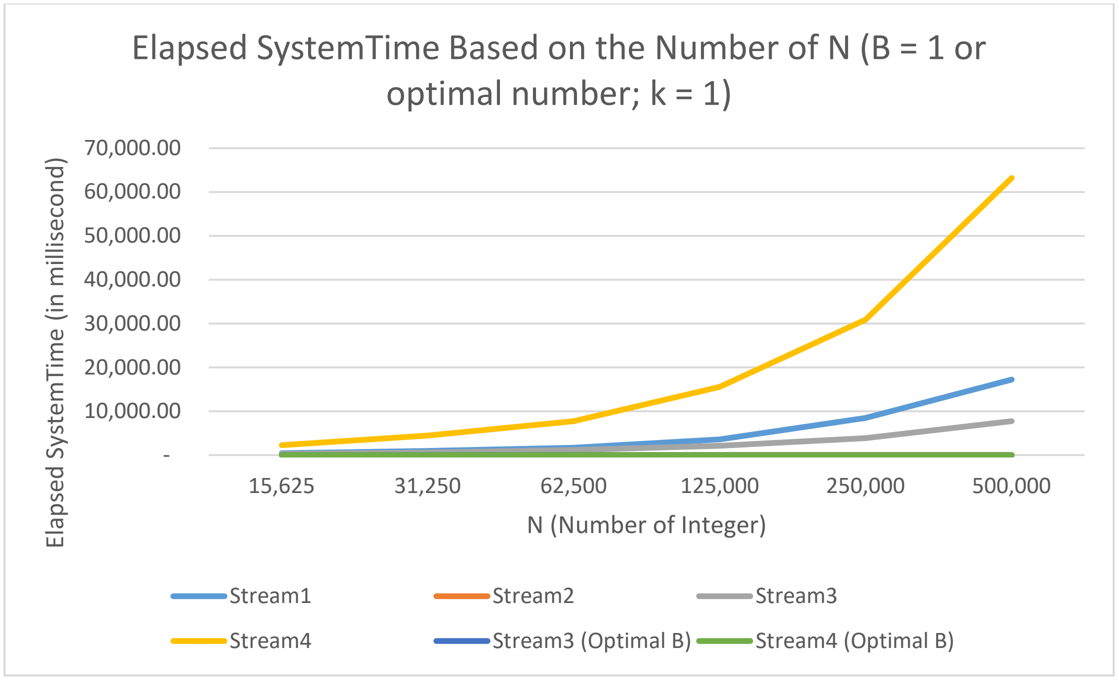
- For Stream 3, SystemTime improvement is limited until B = 8,192, however CPUTime improvement is limited until B = 2,048.
- For Stream 4, we can increase the number of B to improve the performance. However, we need to have a huge gap of B in order to gain huge performance improvement. Also, sometimes improving the number of B by a small factor is decreasing the performance rather than improving the performance. Hence, for Stream 4, the choice of B element is very important.
- SystemTime and CPUTime difference trend is the same as Small Input. Hence, the performance of each stream is highly affected by running application code in user space rather than the system calls (I/O).

We concluded that the optimum value of B using small input is B = 2,048 (because increasing the number furthermore is not giving much performance improvement) for Stream 3 and B = 8,388,608 for Stream 4 (because using this B value, we can gain huge performance improvement from the previous value of B).

2.2.2 Based on the Number of N (Number of Integer)

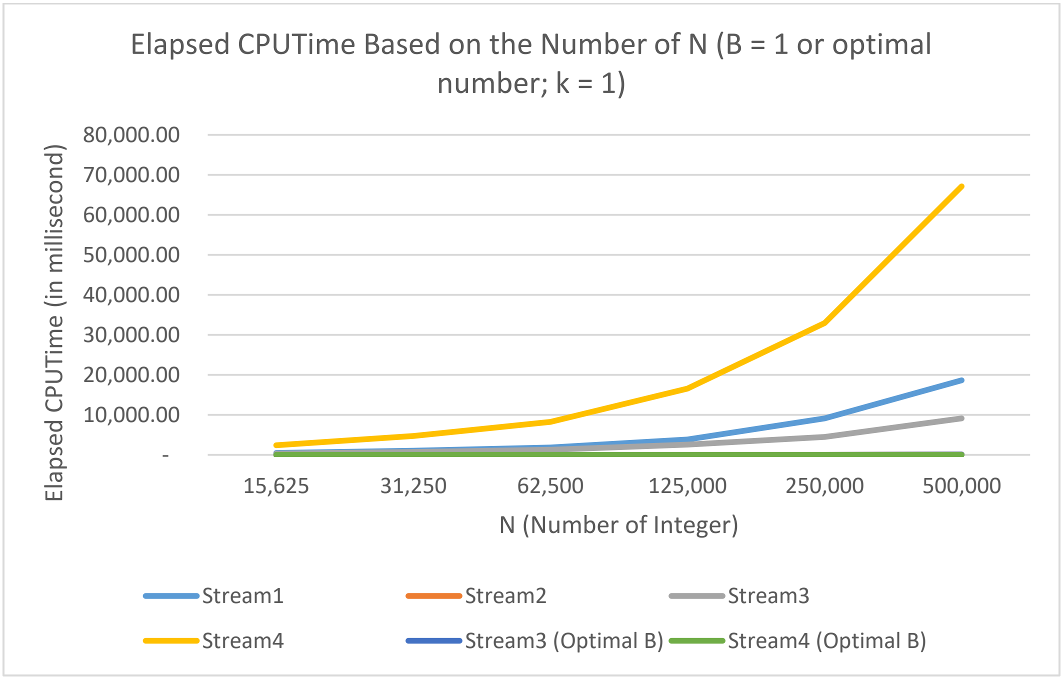
2.2.2.1 Small Input

We have tested the four implementation of stream based on the number of N for Small Input. For Stream 3 and 4, we tested the stream using B = 1 and B = optimal value of B (taken from variable B testing result). For Stream 3, the optimal B value is 2048 and for Stream 4 the optimal B value is 32,768. The information below is our test result based on the number of N using SystemTime:



N	**Stream1**	**Stream2**	**Stream3**	**Stream4**	**Stream3 (Optimal B)**	**Stream4 (Optimal B)**
15,625	457.81	3.13	303.13	2,293.75	3.13	6.25
31,250	984.38	4.69	635.94	4,481.25	6.25	3.13
62,500	1,678.13	4.69	1,150.00	7,757.81	3.13	1.56
125,000	3,557.81	4.69	2,162.50	15,528.13	6.25	3.13
250,000	8,501.56	9.38	3,870.31	30,896.88	9.38	7.81
500,000	17,209.38	20.31	7,759.38	63,165.63	17.19	10.94

Below information is our result testing based on the number of N using CPUTime:



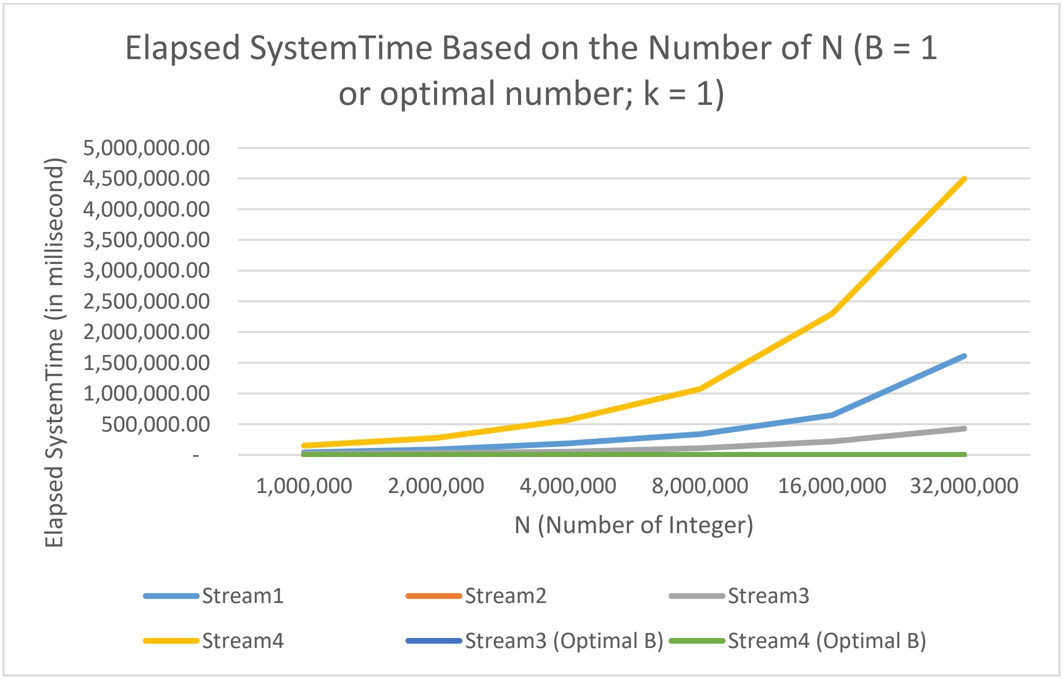
N	Stream1	Stream2	Stream3	Stream4	Stream3 (Optimal B)	Stream4 (Optimal B)
15,625	490.63	20.31	367.19	2,448.44	23.44	9.38
31,250	1,043.75	7.81	743.75	4,764.06	7.81	4.69
62,500	1,828.13	15.63	1,318.75	8,279.69	14.06	10.94
125,000	3,832.81	32.81	2,548.44	16,584.38	25.00	7.81
250,000	9,143.75	54.69	4,507.81	32,976.56	21.88	20.31
500,000	18,651.56	96.88	9,131.25	67,100.00	75.00	29.69

Based on our experiment, we derived the following points:

- If we only use one buffer element (B = 1) for Stream 3 and Stream 4, their performance is very bad. Whereas if we use optimum number of B (B = 2048 for Stream 3 and B = 32,768 for Stream 4) their performance is very good and significantly increased.
- SystemTime and CPUTime give approximately the same trend to the result. The difference between SystemTime and CPUTime is very small. This implies that in this test, the performance of each stream is highly affected by the system calls (I/O) rather than running application code in user space.
- Sometimes, increasing the number of N doesn't increase the elapsed time (highlighted by the red number). From our opinion, this is due to the difference in the number of N is too small for the SystemTime to properly quantify. During the Large Input test, this condition is not supposed to happen again due the gap between the number of N is big enough for SystemTime to properly quantify.

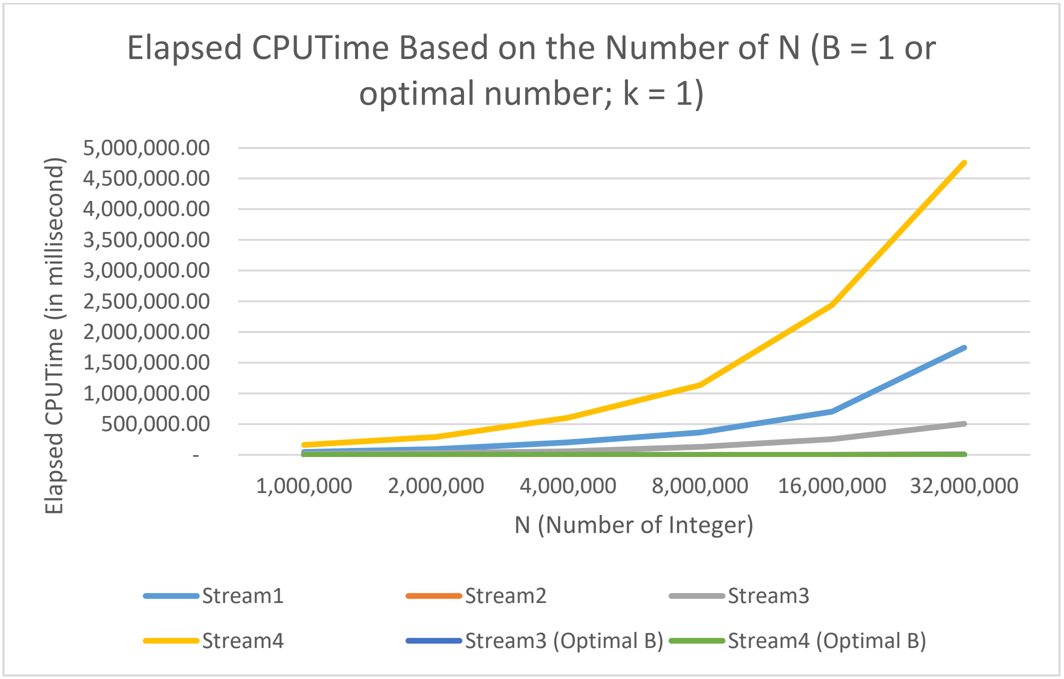
2.2.2.2 Large Input

We have tested the four implementation of stream based on the number of N for Large Input. For Stream 3 and 4, we tested the stream using B = 1 and B = optimal value of B (derived from variable B testing). For Stream 3, the optimal B value is 2048 and for Stream 4 the optimal B value is 8,388,608. Below information is our result testing based on the number of N using SystemTime:



N	Stream1	Stream2	Stream3	Stream4	Stream3 (Optimal B)	Stream4 (Optimal B)
1,000,000	43,937.50	109.38	16,406.25	153,140.63	125.00	15.63
2,000,000	87,609.38	31.25	30,078.13	277,765.63	62.50	15.63
4,000,000	186,453.13	78.13	51,421.88	568,515.63	93.75	31.25
8,000,000	337,609.38	156.25	112,875.00	1,073,062.50	375.00	31.25
16,000,000	648,734.38	375.00	219,031.25	2,301,062.50	375.00	93.75
32,000,000	1,611,343.75	781.25	428,343.75	4,495,625.00	1,078.13	156.25

Below information is our result testing based on the number of N using CPUTime:



N	Stream1	Stream2	Stream3	Stream4	Stream3 (Optimal B)	Stream4 (Optimal B)
1,000,000	48,125.00	328.13	19,265.63	161,375.00	296.88	78.13
2,000,000	94,796.88	453.13	34,625.00	295,156.25	421.88	78.13
4,000,000	202,156.25	828.13	60,875.00	603,890.63	937.50	187.50
8,000,000	367,500.00	1,250.00	132,046.88	1,137,453.13	1,796.88	312.50
16,000,000	704,921.88	2,750.00	257,984.38	2,438,109.38	1,906.25	531.25
32,000,000	1,747,687.50	5,625.00	505,671.88	4,758,625.00	6,062.50	1,265.63

Based on our experiment, we derived the following points:

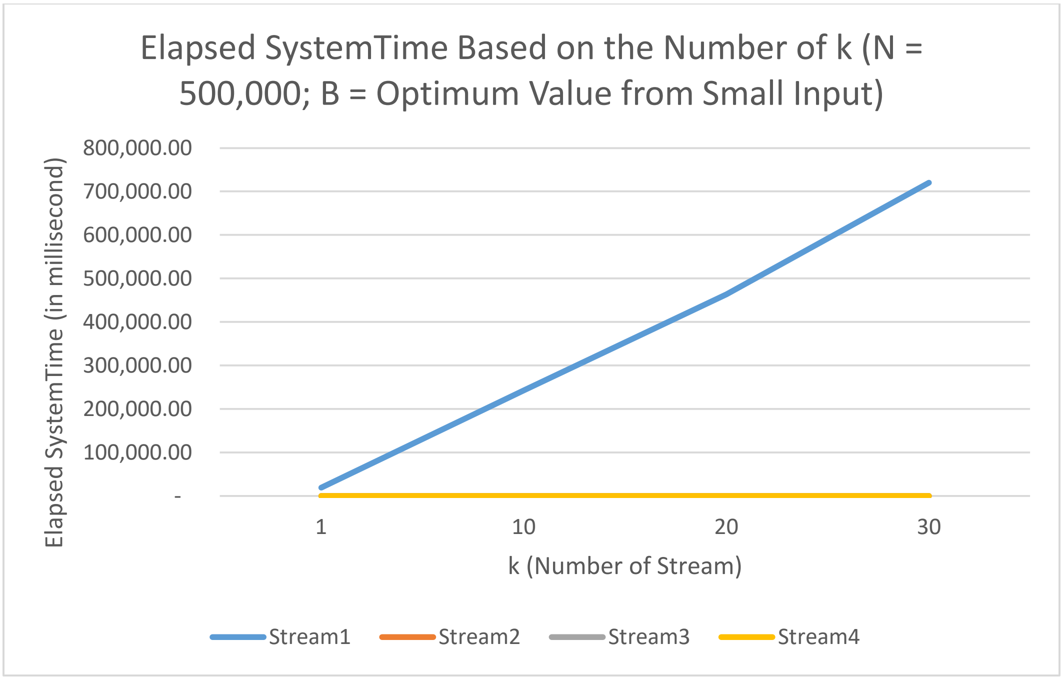
- The result that we got from the Large Input is the same as the result from the Small Input. However, in this test, the elapsed time needed is linearly increasing without any decrease. This is due to the gap of N is big enough to quantify, hence the effect of increasing N is clearly stated in this test. Thus, the elapsed time needed is linearly increasing implies that the performance becomes worse if we increase the number of N).

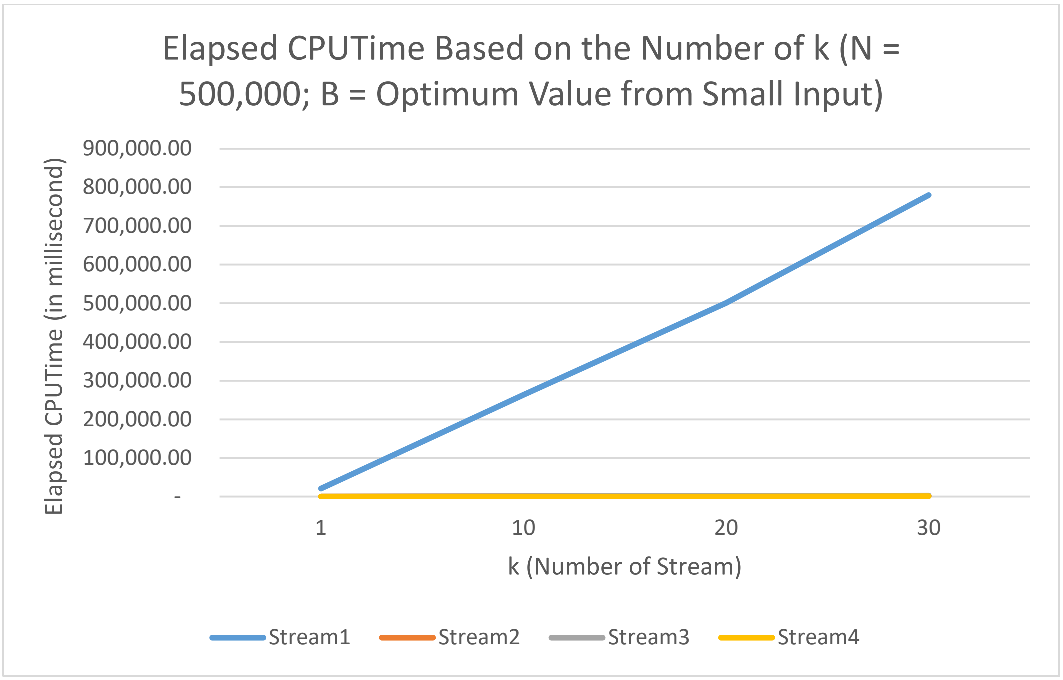
2.2.3 Based on the Number of K (Number of Stream)

For Small Input, we test all of the four stream, for Large Input, Stream 1 is not included. This is due to Stream 1 already requires too much time to complete the task compared to other stream during Small Input, hence it will not be the most performant. Also, the graph for the Large Input is clearer for analysis between Stream 2, Stream 3, and Stream 4 if we exclude the Stream 1.

2.2.3.1 Small Input

The information below is our test result of variable k using SystemTime and CPUTime for Small Input. Both Stream 3 and Stream 4 are using optimum value of B which is B = 2,048 for Stream 3 and B = 32,768 for Stream 4.





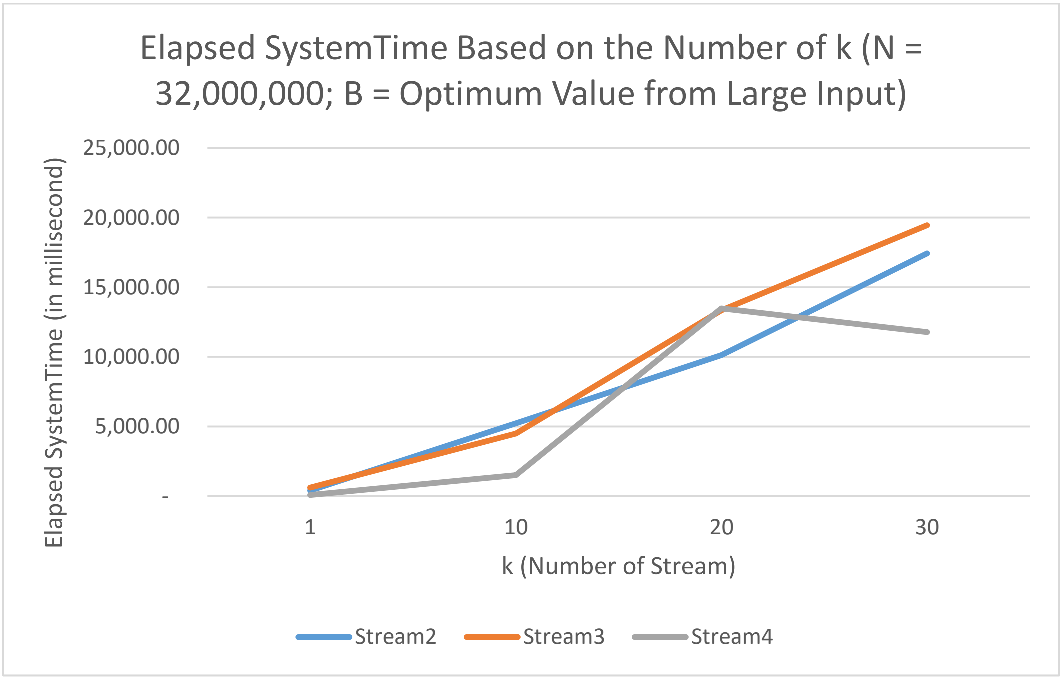
K	SystemTime				CPUTime			
	Stream1	Stream2	Stream3	Stream4	Stream1	Stream2	Stream3	Stream4
1	19,576.56	17.19	14.06	6.25	21,187.50	140.63	125.00	31.25
10	243,289.06	189.06	209.38	87.50	263,268.75	1,165.63	1,196.88	282.81
20	463,542.19	285.94	295.31	164.06	501,101.56	1,898.44	2,009.38	575.00
30	720,057.81	343.75	421.88	242.19	779,515.63	2,276.56	2,579.69	792.19

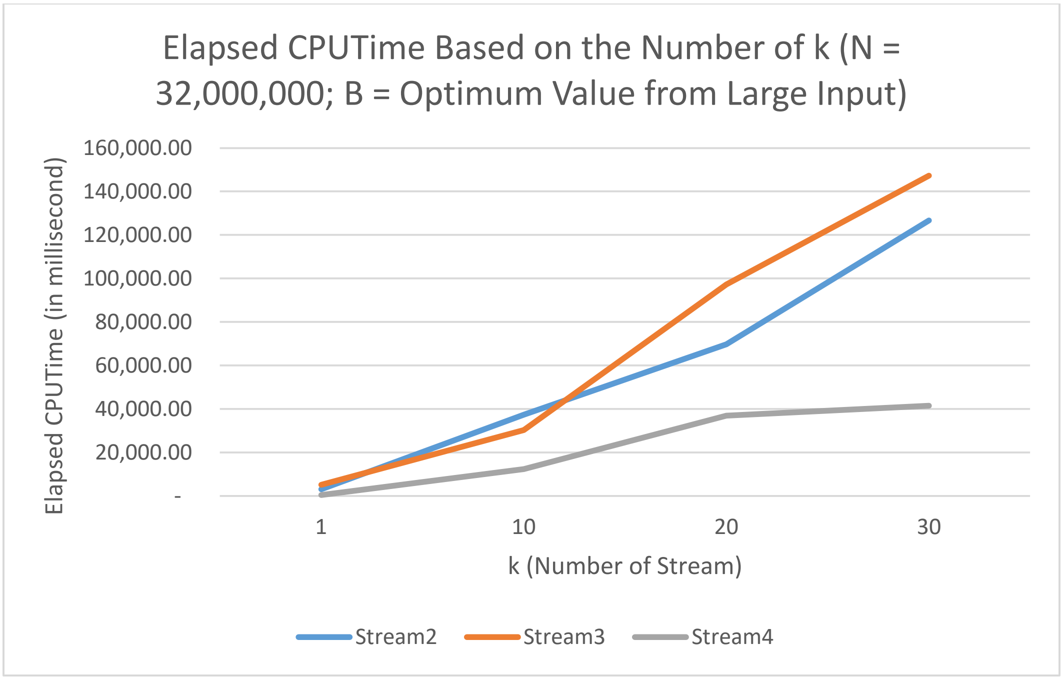
Based on our experiment, we derived the following points:

- If we increase the number of k, each implementation needs more time to finish the test.
- Stream 4 performance is the best amongst the other Stream, measured both in SystemTime and CPUTime.
- For Stream 1, the difference between SystemTime and CPUTime is small. This implies that its performance is highly affected by the system calls (I/O) rather than running application code in user space.
- For Stream 2 and Stream 3, the difference between SystemTime and CPUTime is huge. This implies that its performance is highly affected by running application code in user space rather than the system calls (I/O).

2.2.3.2 Large Input

Below information is our testing result of variable k using SystemTime and CPUTime for Large Input. Both Stream 3 and Stream 4 is using optimum value of B which is B = 2,048 for Stream 3 and B = 8,388,608 for Stream 4.





K	SystemTime			CPUTime		
	Stream2	Stream3	Stream4	Stream2	Stream3	Stream4
1	395.31	595.31	78.13	3,110.94	5,228.13	550.00
10	5,221.88	4,485.94	1,510.94	37,440.63	30,312.50	12,359.38
20	10,125.00	13,346.88	13,465.63	69,835.94	97,293.75	36,935.94
30	17,435.94	19,442.19	11,773.44	126,714.06	147,207.81	41,598.44

Based on our experiment, we derived the following points:

- If we increase the number of k, each implementation needs more time to finish the test.
- Stream 4 performance is the best amongst the other Stream, measured both in SystemTime and CPUTime.
- There is an anomaly for Stream 4 k = 30 SystemTime which is smaller than K = 20. However, the CPUTime is normal (k = 30 is bigger than k = 20). This is probably because the machine is faster when executing system calls (I/O) for k = 20 during the test as such condition could happens.

2.3 Analysis Between Expected Behavior and Experimental Observations

Comparing the expected behavior and experimental observation, we found out that most of the expected behavior occurred in the experiment result.

From Stream 1 and Stream 2 experiment result, increasing N and k will also increase the time needed to complete the task hence decreasing the overall performance. The experiment result is in-line with our expected behavior.

For Stream 3, the experiment result of increasing N and k is in-line with our expected behavior in which the overall performance will decrease as time needed to complete the task is increasing. For variable B, our experiment result shows that there is a point of diminishing return as we have expected. As the performance is not improving that much after B = 2048 integer, we conclude that the optimum number of B is 2048 integer / 8192 bytes (which is also in-line with Java default buffer size).

For Stream 4, the expected behavior of increasing N and k are in-line with the experiment result, hence the overall performance will decrease as the time needed to complete the task is increasing. For variable B, we found out that the experiment result is also in-line with the expected behavior. However, in our testing parameter we concluded that the optimum value for B using Small Input (N = 500,000) is 32,768 integers and the optimum value for B using Large Input (N = 32,000,000) is 8,388,608 integers.

Comparing all of the stream implementation, we concluded that Stream 4 is the best stream. Hence, Stream 4 will be used as input/output stream for the external multi-way merge sort test.

3 Observations on External Multi-Way Merge Sort

3.1 Expected Behavior

The program takes one input file containing N number of integers, M and d as the input parameters. M is the size of main memory available during the first sort phase, in number of 32-bit integers, d is the number of streams to merge in one pass in the later sort phases. The multi-way merge-sort algorithm is implemented in two phases.

1. During the first sort phase, input file is split into $\lceil N/M \rceil$ streams with size at most M.

   The program reads the integers of the input file one by one and store it into a priority queue until the size of the priority queue reaches M or the end of input stream is reached. The priority queue can sort automatically using heap sort. Then we use poll() method of the priority queue to output the sorted integers into output streams. All the integers are read once and written once in this phase, the cost is:

$$C = 2 * N$$

2. In the second sort phase, merge every d streams into one sorted stream and put the resulting stream at the end of the queue. This is done repeatedly until only one stream remains.

   Again, we use a priority queue of size d and create a structure called “Data” containing one integer and the stream reference number of the integer to help with the merge. First, the first integer of each sorted streams and the reference are constructed into a Data and stored into the priority queue. The queue is sorted based on the integer. Then the queue polled the current smallest integer into the output stream, retrieve the integer’s reference number, go to the referenced stream to add the next integer into the queue. When all the d sorted streams reach the end and the queue is empty after polling, the merging of d sorted stream is completed.

   Above is the procedure of one pass, all the integers are read once and written once, the cost is 2*N. The later phase repeats the pass until only one stream left. Since the number of streams to merge in one time is d and the initial total number of streams is $\lceil N/M \rceil$, the number of passes is $\log_d \lceil N/M \rceil$. Hence the cost function for the second the phase is:

$$C = 2 * N * \log_d \lceil N/M \rceil$$

   Combining the two sort phase, the total cost function of external multi-way merge-sort algorithm is:

$$C = 2 * N * (\log_d \left\lceil \frac{\mathrm{N}}{\mathrm{M}} \right\rceil + 1)$$

Based on the equation, an increasing performance can be expected if we increase M, increase d, and decrease N. Also, since we are using the optimum value of B from the Stream Test, B is constant in this test, hence is not included in the cost function.

3.2 Experimental Observations

To further test the capabilities of the external multi-way merge sort, we test each implementation by:

- varying variable N (number of integers)
- varying variable M (number of memory available)
- varying variable d (number of streams to merge)
- comparing our external multi-way merge sort implementation and heapsort (main memory sort) using N ≤ M
- finding best combination of M and d given N

From the stream experiment we found that Stream 4 is the most performant, thus we will use Stream 4 for this testing. The number of buffer of Stream 4 that we chose is the optimum B value for Large Input (B = 8,388,608). This is because in the condition where the number of integer is smaller than 8,388,608, Stream 4 will choose the minimum of the number of integer and 8,388,608 (memory mapping implementation in Java cannot map reference more than the number of integer / file size). If we use the optimum value for Small Input B = 32,728 then for N more than 32,728 the program needs to map the files more than one time which might affect the overall performance. Hence, B value of Large Input from the Stream Test is more appropriate in this test.

To understand how each variable affects the performance, we test the implementation by varying the value of the chosen variable and lock the value of other variables. For testing the value of N, we lock M by 250,000 and we lock d by 2. For testing the value of M, we lock N by Small Input (N = 2,000,000) and Large Input (N = 32,000,000) and d by 2. For testing the value of d, we lock N by Small Input (N = 2,000,000) and Large Input (N = 32,000,000) then we lock M by 250,000. Testing is done 10 times then we calculate the average in order to level out fluctuations due to other processes.

The reason why we lock M = 250,000 for testing N is because this number covers all possibilities when testing this variable which is N < M, N = M, N > M. And for variable d this number ensures that the number of streams from N / M is bigger than two hence ensuring the sort is using more than one pass (because N = 2,000,000 for Small Input and N = 32,000,000 for Large Input).

The reason why we lock d = 2 is that we want to ensure that the number of d is not affecting the other variables. An example of d is affecting the other is when:

- N = 500,000;
- M = {50,000; 100,000; 250,000};
- d = 8.

In this example, when M = 50,000, the number of files that we need to deal with is 10 and we need two passes to complete it (because N / M = 10 and d = 8, hence ceil $\log_8 10$ = 2). When M = 100,000, there will be only 5 files after the first phase, thus only one pass is needed (because N / M = 5 < 8). And when M = 250,000, there will be only 2 files and only one pass is needed (because N / M = 2 < 8). Hence in this example if we choose d = 8, it might also affect the whole performance of the algorithm whereas we only want to know the effect of M.

In this project, we also compare external multi-way merge sort with main memory sort in the case that the input file is small enough to fit in memory. We have tried the number of integer that can fit into the memory and we found out that the maximum number of integer that can fit into memory is 32,000,000 using our N choosing method. When we tried to increase the integer into 64,000,000 we encountered a Java Heap Space error. For this comparison, we use M as the same as N limit from Small Input and Large Input. Hence for Small Input M = 2,000,000 and Large Input M = 32,000,000. We lock d = 2 with the same reason as explained above and we vary N as long as N ≤ M.

To find the best combination of M and d given N input, we perform the test with Small Input (N = 2,000,000) and Large Input (N = 32,000,000). For Small Input, we vary the M from 15,625 until 2,000,000 and choose d = 2 (reusing the result from M testing) and d = 128 (obtained from N / smallest M = 2,000,000 / 15,625 = 128). By using d = 128, we ensure that the algorithm will merge the sub-list using only one pass because the number of streams to merge is determined by the minimum of 128 and (N / chosen M). For Large Input, we vary M from 15,625 until 32,000,000 and choose d = 2 (reusing the result from M testing) and d = 2048 (obtained from N / smallest M = 32,000,000 / 15,625 = 2,048). The reason why we chose d = 2,048 is the same as Small Input. After obtaining the experiment result, we can calculate how many d steams of file that are being used by the algorithm from N / M, hence we can find the best combination of M and d.

The method to choose the number of N is the same as the Stream Test. We decide to choose the value of N by determining the file size first then dividing the file size by 4 which is the size of one integer. We choose the power of 2 to decide the number of file size. In our opinion the file size by the power of 2 is sufficient to give us a clear understanding of how the number of N is affecting the performance. For testing variable M and d we use two values of N: Small Input (N = 2,000,000) and Large Input (N = 32,000,000). In our opinion N = 32,000,000 maximum is sufficient to analyze the external multi-way merge sort implementation. This table below represents our chosen N.

File Size in MB	File Size in KB	File Size in Byte	Integer
0.0625	63	62,500	15,625
0.125	125	125,000	31,250
0.25	250	250,000	62,500
0.5	500	500,000	125,000
1	1,000	1,000,000	250,000
2	2,000	2,000,000	500,000
4	4,000	4,000,000	1,000,000
8	8,000	8,000,000	2,000,000
16	16,000	16,000,000	4,000,000
32	32,000	32,000,000	8,000,000
64	64,000	64,000,000	16,000,000
128	128,000	128,000,000	32,000,000

For variable M*,* our chosen value is similar to the number of N. This is because M will divide the N into stream of files. For testing variable N and d, we choose M = 250,000 as explained above. Below table summarizes our chosen M:

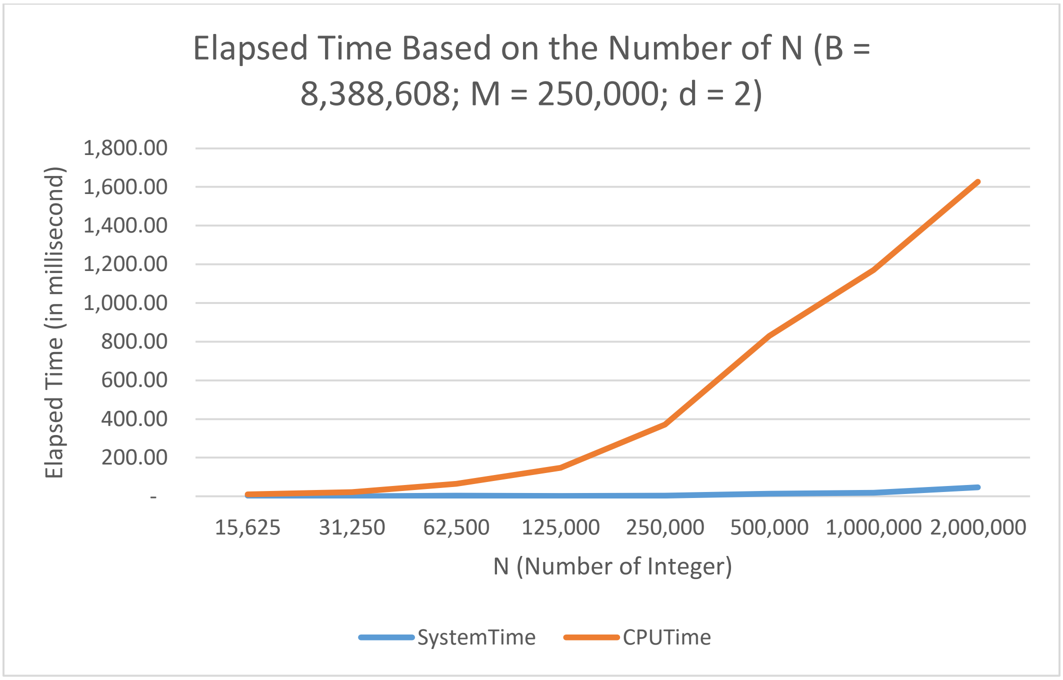
M
15,625
31,250
62,500
125,000
250,000
500,000
1,000,000
2,000,000
4,000,000
8,000,000
16,000,000
32,000,000

For variable d, we choose the value of d = 2, 4, and 8 for Small Input and d = 2, 4, …, 128 for Large Input. This is because for Small Input, at most d = 8 will be used (because at maximum N / M = 2,000,000 / 250,000 = 8). The same reason applies to Large Input as at most d = 128 will be used (because in maximum N / M = 32,000,000 / 250,000 = 128). In our opinion this choice is sufficient to give us a clear understanding of how the number of d is affecting the performance.

3.2.1 Based on the Number of N (Number of Integer)

For Small Input, N = 2,000,000 maximum and B = 8,388,608. For Large Input, N = 32,000,000 maximum and B = 8,388,608.

3.2.1.1 Small Input

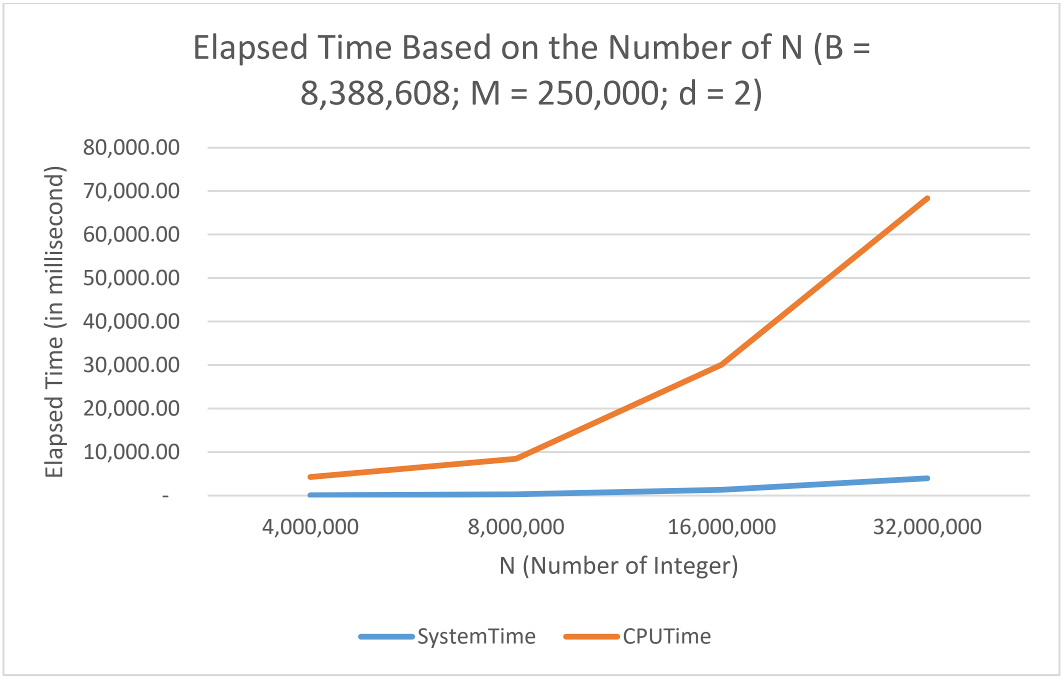


N	SystemTime	CPUTime
15,625	3.13	10.94
31,250	-	21.88
62,500	4.69	65.63
125,000	1.56	148.44
250,000	4.69	371.88
500,000	14.06	829.69
1,000,000	18.75	1,170.31
2,000,000	46.88	1,626.56

Based on our experiment, we derived the following points:

- If we increase the value of N the algorithm needs more time to sort the integers. Also the difference between SystemTime and CPUTime is huge. This implies that its performance is highly affected by running application code in user space rather than the system calls (I/O).
- Using Small Input, sometimes increasing the number of N is not increasing the SystemTime (highlighted by red number). This is because the input is too small to be properly quantified by the SystemTime. We will further analyze that this result won't appear using Large Input.

3.2.1.2 Large Input



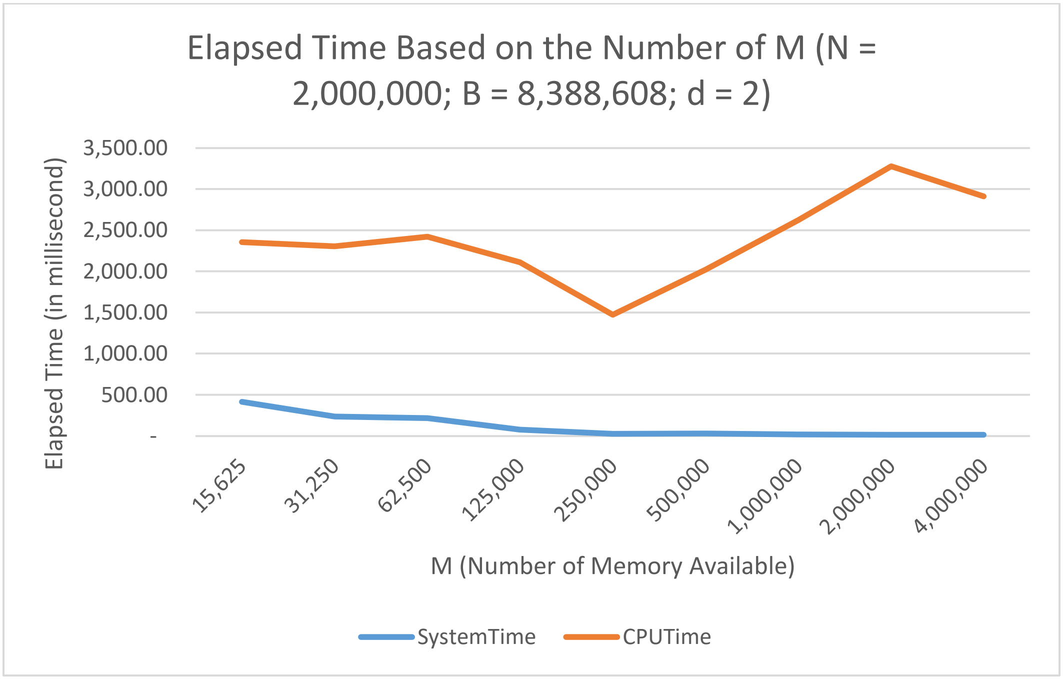
N	SystemTime	CPUTime
4,000,000	117.19	4,282.81
8,000,000	306.25	8,489.06
16,000,000	1,375.00	30,109.38
32,000,000	3,982.81	68,310.94

The performance of Large Input is the same as Small Input. Hence, the performance of external multi-way merge sort will decrease if we increase the number of N. Also, in this test the SystemTime is always increasing, this means that the gap of the number of integer is huge enough to be properly quantified by SystemTime.

3.2.2 Based on the Number of M (Number of Memory Available)

3.2.2.1 Small Input

For testing M using Small Input, we choose N = 2,000,000; B = 8,388,608 and d = 2.



M	SystemTime	CPUTime
15,625	414.06	2,356.25
31,250	234.38	2,303.13
62,500	215.63	2,421.88
125,000	75.00	2,110.94
250,000	26.56	1,471.88
500,000	31.25	2,018.75
1,000,000	18.75	2,623.44
2,000,000	15.63	3,278.13
4,000,000	12.50	2,909.38

Based on our experiment, we derived the following points:

- The difference between CPUTime and SystemTime is huge which means that the overall performance is highly affected by the UserTime.

- The SystemTime is decreasing as we increase the number of M. As SystemTime is affected by system calls (I/O), we can conclude that this value is indeed expected to be decreasing due to decreasing number of pass.

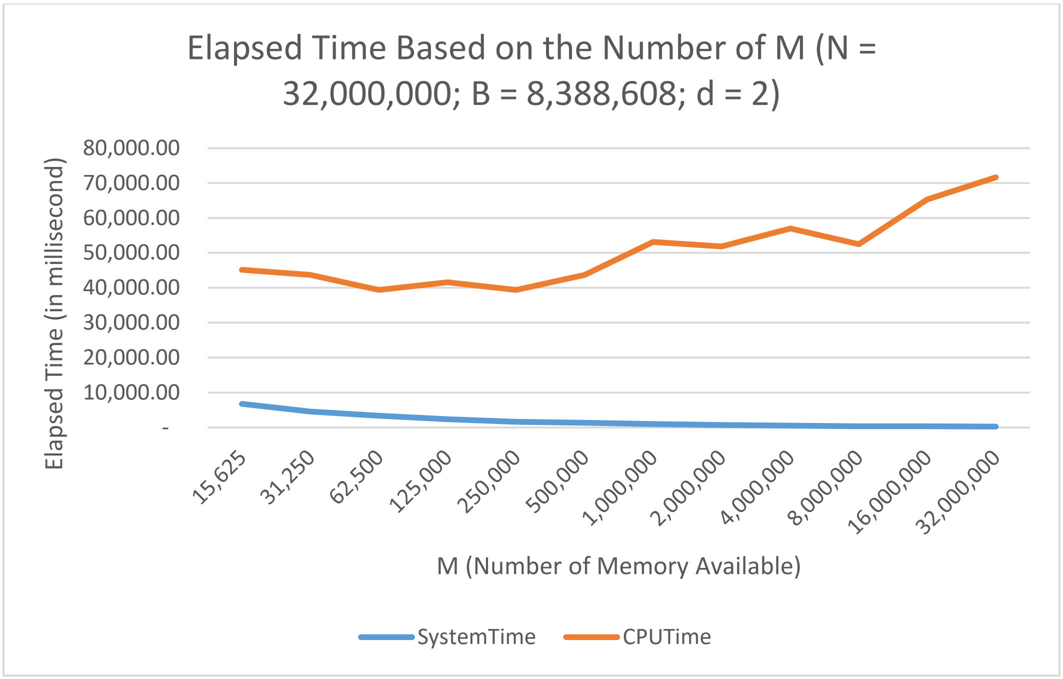
- However, CPUTime is decreasing until M = 250,000 (optimum point) then increasing again and becoming worse which means that UserTime is decreasing until optimum M point and increasing after that point. UserTime is time spent for application code in user space. Based on our analysis, after optimum value of M this value is increasing due to time spent to sort the sub-list in the first pass is higher if we increase the value of M (refer to yellowed cells in the table below). In the next pass, the time spent to sort and merge is not high. This is because we merge 2 streams every time (d = 2), first we input one element from the first stream and we input one element from the second stream to the priority queue. After that, priority queue will find which element is the least then output that element (perform comparison of element once). Should the output element come from the first stream, the next element from the first stream is added into priority queue and vice versa. Hence, UserTime spent in this operation is minimal as it only needs to perform one comparison per inputting one element. This also means that the time spent to sort and merge the second and remaining pass is very small compared to time spent to sort the first pass. Combining both the cost to sort the first pass and to sort and merge the remaining pass, we concluded that we need to find a balance when choosing appropriate value of M, taking into account the time spent to sort the first pass and the time to deal with the remaining pass. That is why if we increase M the performance will improve until certain point (optimum M), then it becomes worse. This table below illustrates the cost of sorting the sub-list in the first pass:

M	#Pass	#Sublist	Sorting First Pass
			#Sublist * M * LOG (M)
15625	8	128	27,863,138
31250	7	64	29,863,138
62500	6	32	31,863,138
125000	5	16	33,863,138
250000	4	8	35,863,138
500000	3	4	37,863,138
1000000	2	2	39,863,138
2000000	1	1	41,863,138

- From this result, we also noted that it is possible that predicted cost function might differ from the actual implementation of external multi-way merge sort due to other factor.

3.2.2.2 Large Input

For testing M using Large Input, we choose N = 32,000,000; B = 8,388,768 and d = 2.



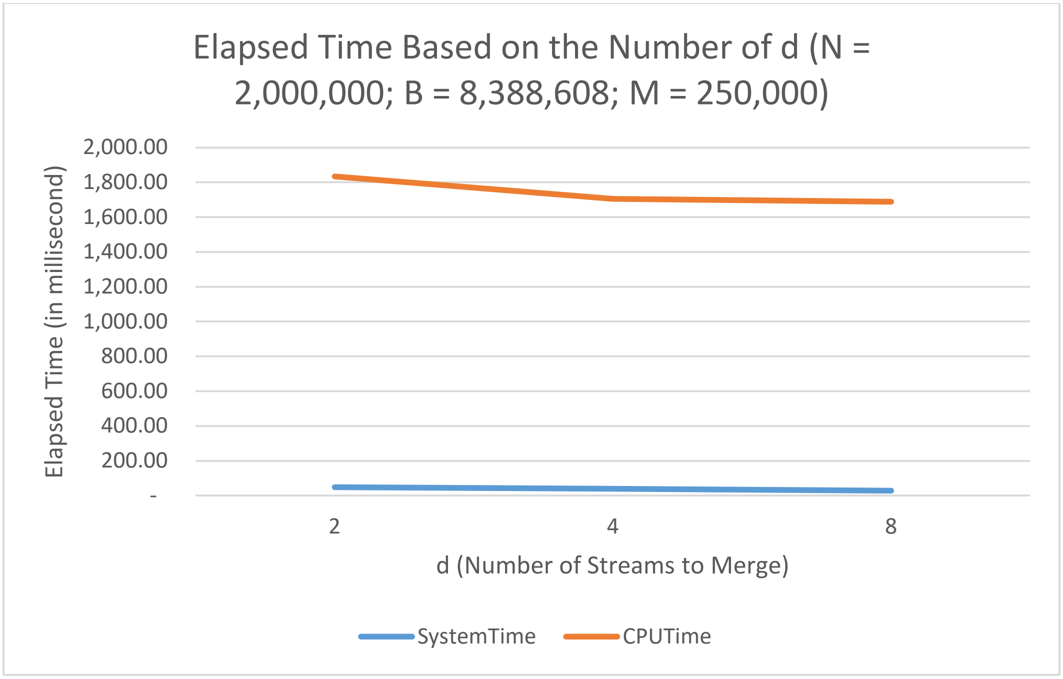
M	SystemTime	CPUTime
15,625	6,746.88	45,137.50
31,250	4,512.50	43,673.44
62,500	3,390.63	39,340.63
125,000	2,398.44	41,596.88
250,000	1,603.13	39,334.38
500,000	1,325.00	43,698.44
1,000,000	1,020.31	53,095.31
2,000,000	667.19	51,807.81
4,000,000	512.50	56,946.88
8,000,000	335.94	52,493.75
16,000,000	359.38	65,256.25
32,000,000	268.75	71,581.25

The result of our testing using Large Input is the same as our testing using Small Input with B = 8,388,608. Hence, performance of external multi-way merge sort is improving until optimum M point (M = 250,000) then decreasing when we increase the number of M even more.

3.2.3 Based on the Number of d (Number of Streams to Merge)

For Small Input, N = 2,000,000 and d = 8 maximum. For Large Input, N = 32,000,000, d = 128 maximum. B = 8,388,608 and M = 250,000 for both inputs.

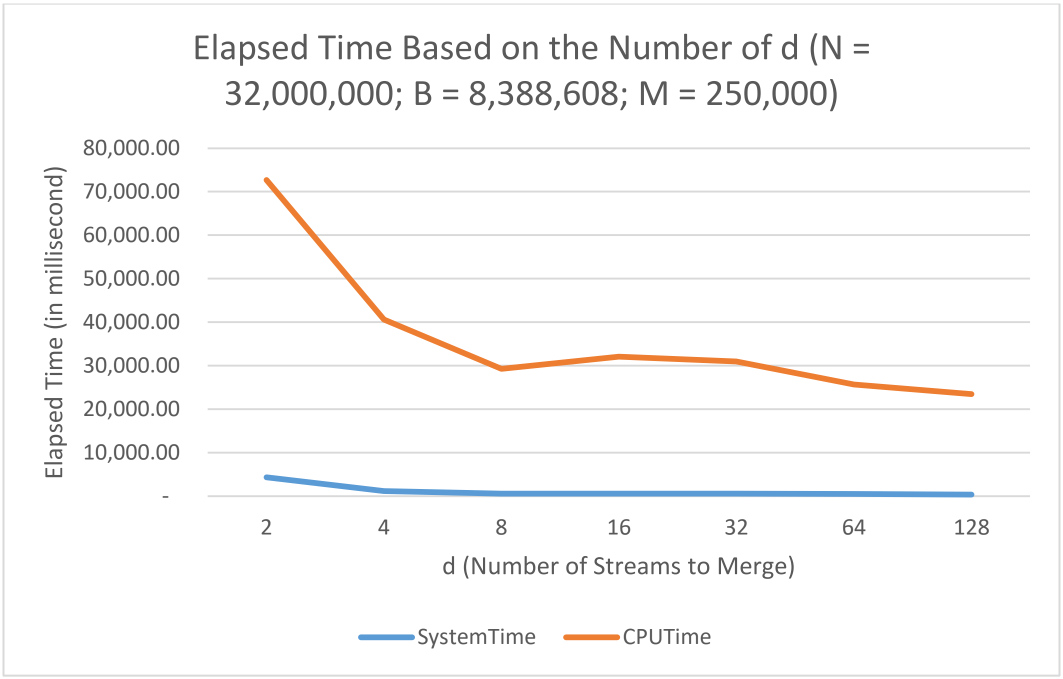
3.2.3.1 Small Input



d	SystemTime	CPUTime
2	48.44	1,834.38
4	39.06	1,704.69
8	28.13	1,687.50

Based on our experiment, we can conclude that if we increase the number of d the algorithm needs less time to sort the integers. This is due to decreasing number of the pass the algorithm needs to perform. Also the difference between SystemTime and CPUTime is huge. This implies that its performance is highly affected by running application code in user space rather than the system calls (I/O).

3.2.3.2 Large Input



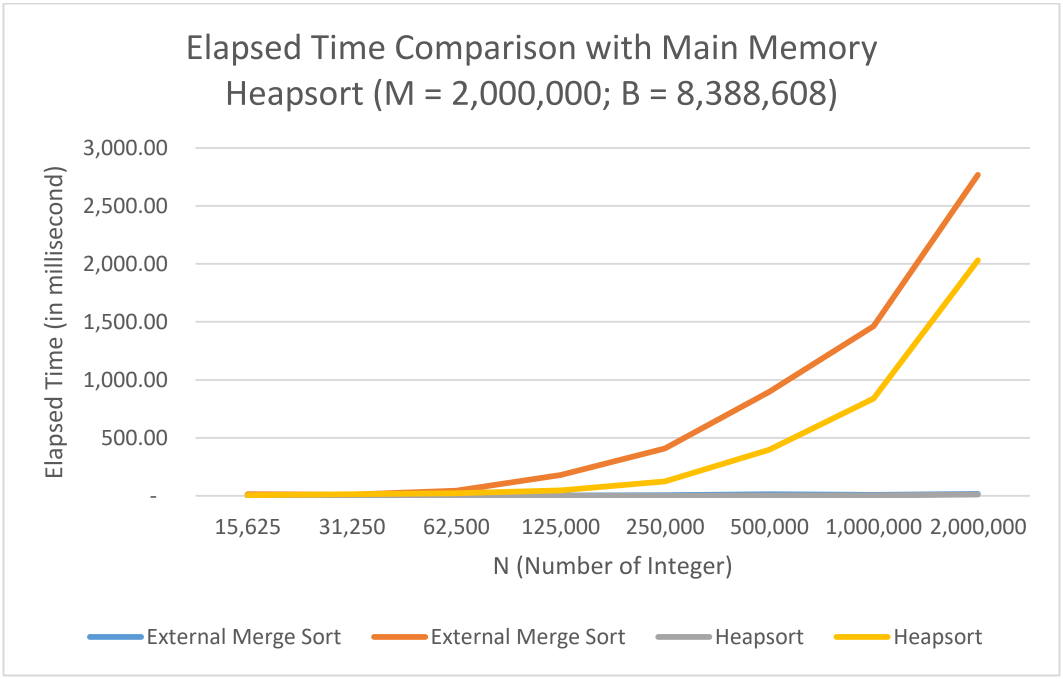
d	SystemTime	CPUTime
2	4,373.44	72,695.31
4	1,214.06	40,635.94
8	639.06	29,318.75
16	581.25	32,100.00
32	598.44	30,959.38
64	515.63	25,714.06
128	381.25	23,467.19

The performance of Large Input is the same as Small Input. Hence, if we increase the value of d the algorithm needs less time to sort the integers due to decreasing number of pass. Also the difference between SystemTime and CPUTime is huge. This implies that its performance is highly affected by running application code in user space rather than the system calls (I/O).

3.2.4 Comparison with Main Memory Heapsort

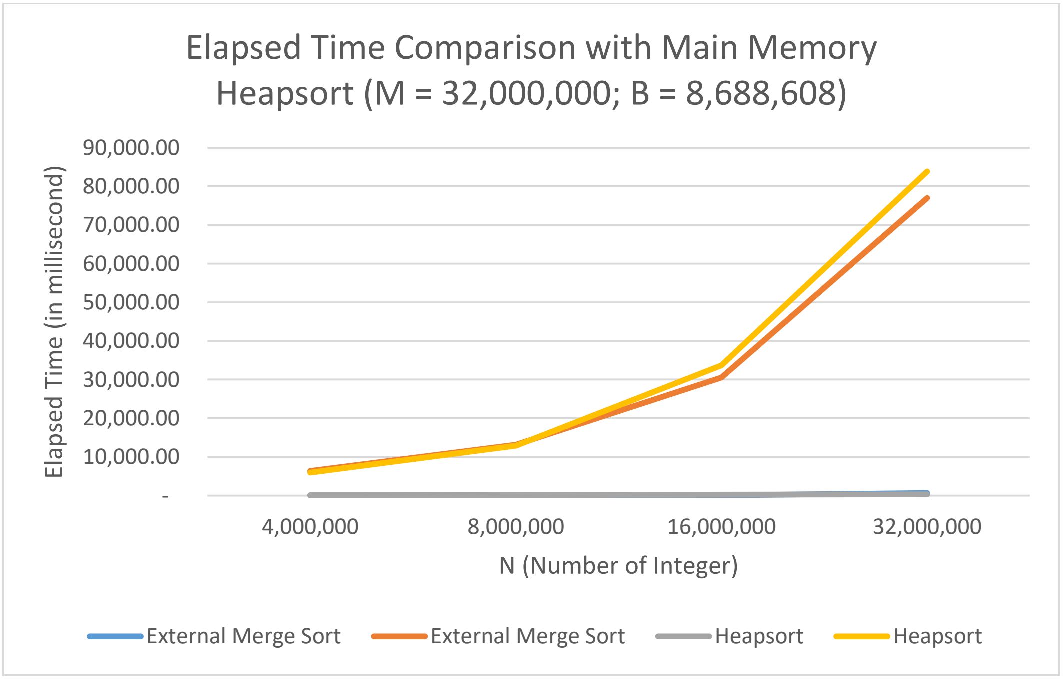
For comparison with main memory heapsort, we configure the external memory sort so that all of the element can fit into main memory using single pass, that is N ≤ M. For this test, we use Small Input (N ≤ 2,000,000; M = 2,000,000) and Large Input (4,000,000 ≤ N ≤ 32,000,000; M = 32,000,000). The value of B = 8,388,608 for both inputs.

The information below is the result of our test using Small Input:



N	External Merge Sort		Heapsort	
	SystemTime	CPUTime	SystemTime	CPUTime
15,625	1.56	14.06	3.13	6.25
31,250	-	10.94	1.56	15.63
62,500	1.56	43.75	1.56	23.44
125,000	3.13	179.69	1.56	48.44
250,000	6.25	407.81	1.56	125.00
500,000	14.06	898.44	3.13	398.44
1,000,000	9.38	1,462.50	4.69	840.63
2,000,000	17.19	2,765.63	10.94	2,031.25

The information below is the result of our test using Large Input:



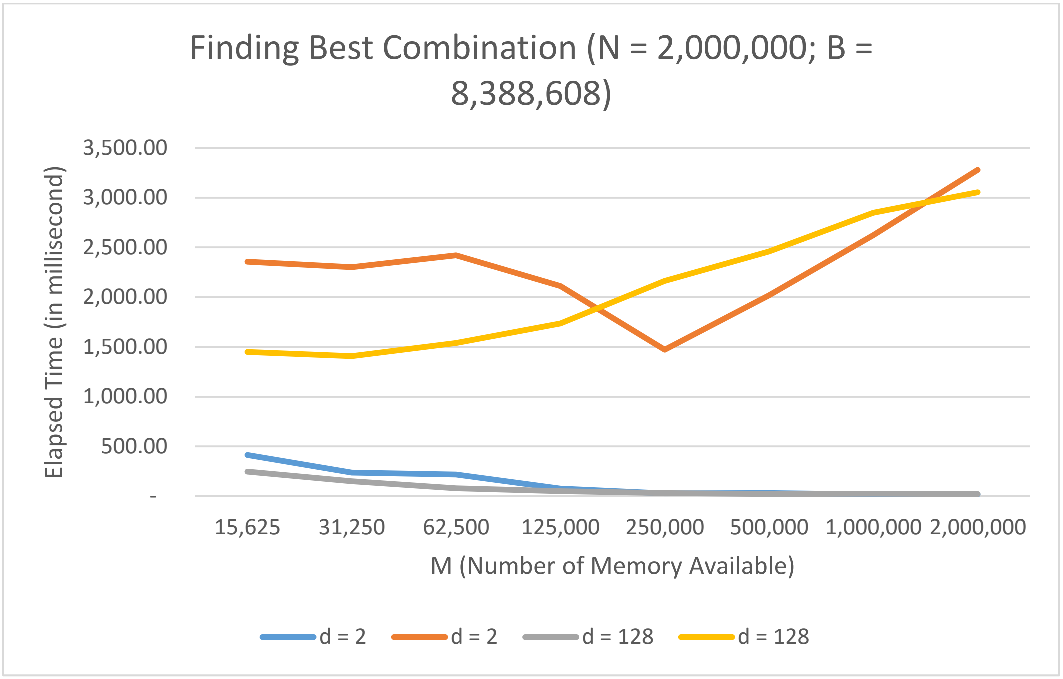
N	External Merge Sort		Heapsort	
	SystemTime	CPUTime	SystemTime	CPUTime
4,000,000	37.50	6,362.50	57.81	5,971.88
8,000,000	57.81	13,171.88	135.94	12,909.38
16,000,000	112.50	30,512.50	278.13	33,707.81
32,000,000	634.38	76,923.44	360.94	83,812.50

Cell highlighted in green color is the faster time of the two sort algorithm. Based on our experiment, we found out that both algorithms are comparable with each other. Heapsort performs faster than external memory sort as it has more green color than external memory sort using Small Input. However, using Large Input, external multi-way merge sort has more green color. Also, the time difference between the two algorithms is not that high. Hence, we conclude that external multi-way merge sort is comparable to main memory heapsort if the input is small enough to fit into the main memory.

3.2.5 Finding the Best Combination of M and d

After experimenting with variable N, M, and d, we also perform testing to find best combination of M and d. In this test, we use N from Small Input (N = 2,000,000) and Large Input (N = 32,000,000). We also reuse the data from testing variable M (with d = 2) and perform testing with those data with d = 128 (obtained from N / smallest M = 2,000,000 / 15625 = 128) for Small Input and d = 2048 for large input (obtained from N / smallest M = 32,000,000 / 15625 = 2048). We chose this maximum d value to ensure that the algorithm will use the smallest number of pass to complete the sort. From the result, we can approximate the optimum value of used d by calculating N / used M.

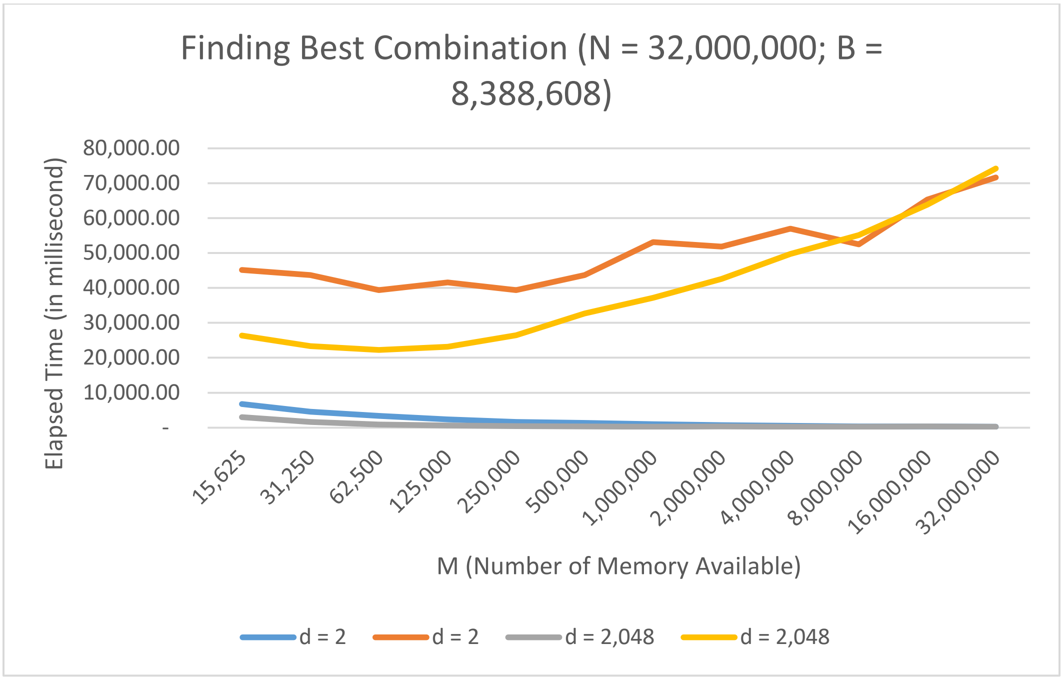
3.2.5.1 Small Input



M	d = 2		d = 128		Used d (N / M)
	SystemTime	CPUTime	SystemTime	CPUTime	
15,625	414.06	2,356.25	245.31	1,448.44	128
31,250	234.38	2,303.13	150.00	1,407.81	64
62,500	215.63	2,421.88	78.13	1,540.63	32
125,000	75.00	2,110.94	50.00	1,735.94	16
250,000	26.56	1,471.88	29.69	2,162.50	8
500,000	31.25	2,018.75	21.88	2,459.38	4
1,000,000	18.75	2,623.44	25.00	2,850.00	2
2,000,000	15.63	3,278.13	20.31	3,053.13	1

From this experiment, we can conclude that the optimum M and d value for Small Input (N = 2,000,000) is M = 31,250 and d = 64.

3.2.5.2 Large Input



M	d = 2		d = 2,048		Used d (N / M)
	SystemTime	CPUTime	SystemTime	CPUTime	
15,625	6,746.88	45,137.50	2,978.13	26,339.06	2048
31,250	4,512.50	43,673.44	1,595.31	23,296.88	1024
62,500	3,390.63	39,340.63	904.69	22,271.88	512
125,000	2,398.44	41,596.88	585.94	23,189.06	256
250,000	1,603.13	39,334.38	400.00	26,467.19	128
500,000	1,325.00	43,698.44	354.69	32,678.13	64
1,000,000	1,020.31	53,095.31	246.88	37,209.38	32
2,000,000	667.19	51,807.81	326.56	42,593.75	16
4,000,000	512.50	56,946.88	256.25	49,717.19	8
8,000,000	335.94	52,493.75	231.25	55,171.88	4
16,000,000	359.38	65,256.25	337.50	63,853.13	2
32,000,000	268.75	71,581.25	248.44	74,154.69	1

From this experiment, we can conclude that the optimum M and d value for Large Input (N = 32,000,000) is M = 62,500 and d = 512. From the result of Small Input and Large Input, we conclude that we need to find a good balance between M and d to obtain the optimum performance of external multi-way merge sort.

3.3 Analysis between Expected Behavior and Experimental Observations

Comparing the expected behavior and experimental observation, we found out that for variable N and d the experimental observation is in-line with the expected behavior, however for variable M the experimental observation is not in-line with the expected behavior.

For variable N, increasing variable N will decrease the overall performance as the time needed to complete the sort is increased. For variable d, increasing variable d will increase the overall performance as the time needed to complete the sort is decrease. Both variable N and d experiment result is in-line with the expected behavior.

For variable M, the experiment result is not in-line with expected behavior. This is because the time needed to perform the first pass sorting is higher if we use higher M despite using *O (n log n)* sorting algorithm as explained in the experiment result section. Hence, for variable M, increasing the value of M is improving the overall performance until an optimum point, but then increasing M again will decrease the overall performance of external multi-way merge sort.

For comparison with main memory heapsort, we found out that external multi-way merge sort and main memory heapsort is comparable to each other.

From the result of our testing regarding best combination for variable M and d, we found out that for Small Input (N = 2,000,000) the best combination is M = 31,250 and d = 64; and for Large Input (N = 32,000,000) the best combination is M = 62,500 and d = 512. From this result, we conclude that we need to find a good balance between M and d to obtain the optimum result using external multi-way merge sort.

3 Overall Conclusion

To sum up, we have implemented external multi-way merge sort algorithm using Java programming language and tested the implementation. To support the input/output of the sort, we also have implemented the stream implementation using four input/output stream classes. The four input/output classes have been compared and the best stream implementation (Stream 4) is chosen as input/output stream for the external multi-way merge sort.

For the Stream Test, increasing the value of N (number of integer) and increasing the value of k (number of stream) will decrease the overall performance. For Stream 3, increasing variable B (number of buffer element) will increase the performance until the point of diminishing return (B = 2048 integer) then increasing B even further won't increase the performance even more. For Stream 4, increasing the number of B will increase the performance, from our experiment the optimum number of B = 32,768 for Small Input (N = 2,000,000) and B = 8,388,768 for Large Input (N = 32,000,000). Stream 4 is the best stream implementation because it uses memory mapping method hence making the input/output operation much faster.

For the External Multi-Way Merge Sort Test, increasing variable N (number of integer) will decrease the performance while increasing variable d (number of streams to merge) will increase the performance. For variable M, the expected behavior is increasing the number of M will increase the performance, however the experiment result shows that increasing the number of M will increase the performance until an optimum point, then improving M even more is decreasing the performance. This is due to the time needed to perform the first sub-list sorting for the first pass is much higher using the higher M even when using *O(n log n)* sorting algorithm, hence affecting the overall performance. In the condition that the input file is small enough to fit in the memory ($N \leq M$), external multi-way merge sort is comparable to main memory heapsort. Our experiment found out that the best combination to sort for Small Input ($N = 2,000,000$) is $M = 31,250$ and $d = 64$; and for Large Input ($N = 32,000,000$) the best combination is $M = 62,500$ and $d = 512$.

Overall, external multi-way merge sort is a good choice if all of the element cannot fit in the main memory at the same time. To use external multi-way merge sort optimally, a good balance of parameter M and d has to be used. To support the input/output stream of the sorting algorithm, memory mapping method is the best choice if the input is very large.

4 References

[1] "Java tip: How to get CPU, system, and user time for benchmarking," Nadeau software consulting, 20 3 2008. [Online]. Available: http://nadeausoftware.com/articles/2008/03/java_tip_how_get_cpu_and_user_time_benchmarking. [Accessed 30 12 2016].

[2] "Java™ Platform, Standard Edition 8 API Specification," Oracle, [Online]. Available: docs.oracle.com/javase/8/docs/api/index.html?java/io/package-summary.html. [Accessed 31 12 2016].

[3] N. Coffey, "How big should my input stream buffer be?," Javamex, 2012. [Online]. Available: http://www.javamex.com/tutorials/io/input_stream_buffer_size.shtml. [Accessed 31 12 2016].

[4] "Memory-mapped file," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Memory-mapped_file. [Accessed 31 12 2016].

[5] B. Swaminathan, "Memory Mapped I/O in Java," PolarSPARC, 20 4 2011. [Online]. Available: http://www.polarsparc.com/xhtml/MemoryMappedIOinJava.html. [Accessed 31 12 2016].