Algorithms in Secondary Memory

Course: INFO-H-417 Database Systems Architecture

Project Report

Fernando Stefanini
Evgeny Pozdeev
Kunal Arora

Université Libre de Bruxelles

Université Libre de Bruxelles
http://www.ulb.ac.be

Title:
Algorithms in Secondary Memory

Course:
Database Systems Architecture

Project Period:
Fall Semester 2018

Project group:
Fernando Stefanini
Evgeny Pozdeev
Kunal Arora

Supervisor(s):
Stijn Vansummeren

Page Numbers: 42

Date of Completion:
January 6, 2019

Abstract:

This report presents the implementation of different stream classes of reading and writing from/to the file using Java standard streams and a special memory mapped I/O scheme. It includes the benchmark to understand the effect of different parameters on these stream implementations. We also implement external multi-way merge-sort algorithm using the best read/write implementation technique and compare the results received during the implementation and expected ones. The cost analysis is also taken into account for all the implementations in order to calculate the number of I/O's as it behaves as the bottleneck when implementing real-world applications.

Contents

Chapter 1

Introduction and Environment

1.1 Goal

The main objective of this project is to implement external-memory merge-sort algorithm and examine it's performance under different parameters.

To do so, we need to do following steps:

- Study four different ways of reading and writing to the file which have their own advantages and shortcomings. Also, we will benchmark them to find the best out of all the 4 different implementations.
  - Data Input/Output Stream
  - Buffered Input/Output Stream
  - Buffered Input/Output Stream with explicit buffer size
  - Memory Mapping
- Implement external multi-way merge-sort to get real-world experience with the performance of external-memory algorithms.
- Benchmark the external multi-way merge-sort algorithm using the best of the read/write implementation technique and compare the results received during the implementation correspond to our expectations or not.

1.2 Machine Environment

For implementation and testing purposes of this project, the following hardware configurations were used:

- Hard disk space: 55GB of SSD storage
- Main memory (RAM): 8GB DDR3
- CPU Processors: Intel Core i7 1.7 GHz with 2 cores
- Operating System: macOS Mojave 10.14.1

1.3 Programming Language

For this project, the Java programming language is used as it is most familiar to all the members of the group and is one of the most widely used programming language.
We did not introduce any additional external libraries for bench-marking purposes given the long execution time for simulations.

1.4 Testing method

In order to generate the test data a simple Java program was written (found in RandomGenerator.java in the project source code) to make use of Java's *java.util.Random* to generate an arbitrary number of integers in a file.
The generated sizes were:

- 1,000 integers: 4 kilobytes.
- 10,000 integers: 40 kilobytes.
- 100,000 integers: 400 kilobytes.
- 1,000,000 integers: 4 megabytes.
- 10,000,000 integers: 40 megabytes.

Those were used both for the I/O Stream benchmarks and the Multiway Merge-sort benchmark. Notice that one file for each size was created for the experiments, so all experiments claiming to use one of the file sizes is using the same file (or an exact copy with the same content in the case a single experiment needs more than one file of the same size).
For collecting runtime, macOS's command *time* was used.

Chapter 2

Streams

In this part, we are going to implement 4 different ways of reading data from the file and writing data to the file. Also, noting down their expected performance behavior and an expected cost formula that estimates the total I/O operations.

2.1 Implementation

To implement 4 different input/output stream, a common interface was created which contain method signatures and constant declarations.
In the project directory, they can be accessed inside the *src/ulb/dsa/io* folder with files named, *InputStream.java* and *OutputStream.java* file respectively for Input and Output stream classes.

InputStream.java - The input interface defines the following declarations.

- **void open(String filePath):** Open file with path defined in the *filePath* variable for reading purpose.
- **int readNext():** Read next integer value from the file.
- **boolean endOfStream():** Verify and denote the end of file stream.

OutputStream.java - The output interface defines the following declarations.

- **void create(String fileName):** Create a file named *fileName* for writing purpose.
- **void write(int number):** Write next integer value into the file.
- **void close():** Close the file and flush the streams.

Next we are going to explore 4 different implementation techniques of Input and Output in detail.

2.1.1 Data Input/Output stream

In this implementation technique, we are reading and writing one integer at a time in/from the file using Java utility classes *DataInputStream* and *DataOutputStream*.[1]

For this, initially a file input stream is created to read the data from file byte by byte. As we have to read Integers and the file input stream works with byte data we use the *DataInputStream* which is a wrapper over the *FileInputStream* class. Then, we are going to read an integer using the *readInt()* function of *DataInputStream* which provides functionality to read primitive data types. And then, the data is written in the file using, *writeInt()* function of *DataOutputStream*.

Cost formula: Number of I/O = k * N

where, **k** = Number of streams. **N** = Number of elements in the file.

Expected behavior:

- As in this implementation it is reading and writing one byte at a time, it is going to be very slow.
- Through the cost function we can deduce that the I/O cost increases as the number of streams or the file size increases.

2.1.2 Buffered Input/Output stream

As we saw that reading and writing one integer at a time is costly process so, we read and write data in chunks. This functionality is provided by *BufferedInput-Stream* and *BufferedOutputStream* [2] of java utility classes.

Initially, a file input stream is created to read the data from file. Then, we wrap the *FileInputStream* object by *Buffered Input Stream* in order to read the chunks of values. Now, in our final step in order to read int data type of values we wrap this buffered object by *DataInputStream* java stream. The same procedure is followed for output stream as well, which help us write int data using the *writeInt()* function of *DataOutputStream* in chunks as well.

In Java for the *Buffered Input/Output stream* the buffer size is *8192 bytes*. Consider, a file of *73,728 bytes* in size then, in order to retrieve all the data in the memory using, *Buffered Input Stream* we need *[73728/8192]* equals to, only 9 I/O operation as compared to 73728 I/O operation for the *Data Input Stream*.
This gives us the cost formula as,

Cost formula: Number of I/O = k * (N/B)

where, **k** = Number of streams. **N** = Number of elements in the file. **B** = Buffer size which is *8192 bytes* in this case.

[1]For basic understanding follow, https://docs.oracle.com/javase/8/docs/api/java/io/DataInputStream.html
[2]For basic understanding follow, https://docs.oracle.com/javase/8/docs/api/java/io/BufferedInputStream.html

Expected behavior:

- Perform faster than reading and writing one byte at a time.
- The number of elements read at once will remain the same for different filesize but, looking at the cost performance formula the I/O cost will increase as the filesize and number of streams increases.

2.1.3 Buffered Input/Output stream with explicit buffer size

The implementation concept of this part is very same as the previous implementation with an exception of providing explicit buffer size to read and write the data to/from the file. We also have to study the effect of varying buffer sizes to see the performance. The optimum buffer size is related on many factors: CPU, cache latency, file system block size.

Moreover, this implementation also relates a lot to the first implementation, because as it was offered in the project assignment description, read and writes should be "implemented as in step (1)", and the only difference is that reads and writes are completing sequentially in loops using Java object memory as buffer. Consequently, since this implementation didn't use any sort of buffer read and write supported by OS, we are expecting performance very close to the first implementation.

The cost formula for this implementation is as,

Cost formula: Number of I/O = k * (N/B)

where, **k** = Number of streams. **N** = Number of elements in the file. **B** = Size of the buffer.

Expected behavior:

- Similar performance as implementation 1.
- As we know that the optimum buffer size is related to many factors so, we can also expect a drop in performance in case of small or very large buffer sizes.

2.1.4 Memory Mapping

Memory mapping I/O [3] is a scheme where an I/O device is mapped into the address space of the system, as if it were RAM. Through memory mapping you can map the whole file or parts of large files into memory to have direct access of

[3] https://www.developer.com/java/other/article.php/1548681/Introduction-to-Memory-Mapped-IO-in-Java.htm

the data. Once the file is mapped, changes that you make to the memory map are automatically propagated to the file via underlying operating system.

In Java, the *java.nio* package provides all the functionalities for memory map including *MemoryMappedBuffer* to facilitate the reading and writing.

Following steps are followed to create a memory mapped input/output stream,

- A *FileChannel* object is obtained from the *RandomAccessFile* object.
  - Supplying it with **"r"** argument then, the file is opened for reading purpose.
  - But, a file is opened in read and write mode if we supply **"rw"** argument to it.
- Mapping is created for the file channel directly into the memory using **map** function of the *RandomAccessFile* object. It requires three arguments,
  - **mode:** The file can be mapped with *READ_ONLY*, *READ_WRITE*, *PRIVATE* mode.
  - **position:** Defines starting point in the file that is mapped into the buffer.
  - **size:** Number of bytes that will be mapped.

Advantages of the Memory-mapped I/O scheme [4]:

- Once the mapping is successfully completed the read and write operations are handled by underlying OS which is faster than, normal disk read and write operations.
- Map a smaller file into memory once and access it often.
- If your application often have write operations on bigger files, then you can map it in memory and write the data to the file synchronously as soon as you make changes to it in the memory.
- Multiple processes can share the same file in the memory.

Disadvantages of the Memory-mapped I/O scheme:

- Mapping a file into memory is a bit expensive process than reading or writing a few kilobytes of file, so performance point of view it is generally only worth mapping relatively large files into memory or a small file which is accessed often.
- As OS loads only a portion of file into memory so, if the data is not present in that portion it can result in significant page faults.

[4] `https://www.codeproject.com/Tips/683614/%2FTips%2F683614%2FThings-to-Know-about-Memory-Mapped-File-in-Java`

The cost formula for this implementation is as,

Cost formula: Number of I/O = k * (N/B)

where, **k** = Number of streams. **N** = Number of elements in the file. **B** = Size of the buffer.

The cost formula is same as the previous two implementations considering that we map B bytes of memory for N size file.

Expected behavior:

- We expect better performance as compared to other implementations as the file is mapped on memory and the changes are propagated directly to file.
- Although, the cost function is same as compared to the previous two implementations it is expected to perform better as, the read/write operations are handled by OS which is faster.

2.2 Experimental observations

In order to compare the performance of different I/O stream types, the following experiment was made:

- For a given value of N number of read/writes, 5 runs are made using k input streams and k output streams, where each iteration of N consists of reading one integer in each of of the k input streams and outputting it to the k output streams. E.g.: For a situation where k=5 we would have 5 read and 5 write operations, one for each input and output stream.
- After the 5 runs, the average time in seconds for such scenario is then recorded as the run time for the experiment.

This was done for the four I/O stream types, and for types 3 and 4 the following values for B (Buffer Size): 10 bytes, 100 bytes, 1 megabyte, 10 megabytes, 100 megabytes. In experiments the number of streams refers to how many input and output streams are used, hence if a graph says 10 streams it means this experiment uses 10 input streams and 10 output stream, and a total of 20 streams. Because of system and hardware capabilities, we were not able to run the experiment with more then 20 streams.
The first graphs below shows the log-based (for visualizing purposes) total time in seconds spent per each of the stream types (and its buffer size), when considering N operations. The legend entries can be read as:
$[Stream\_type]\_[buffer\_size\_in\_bytes]$.

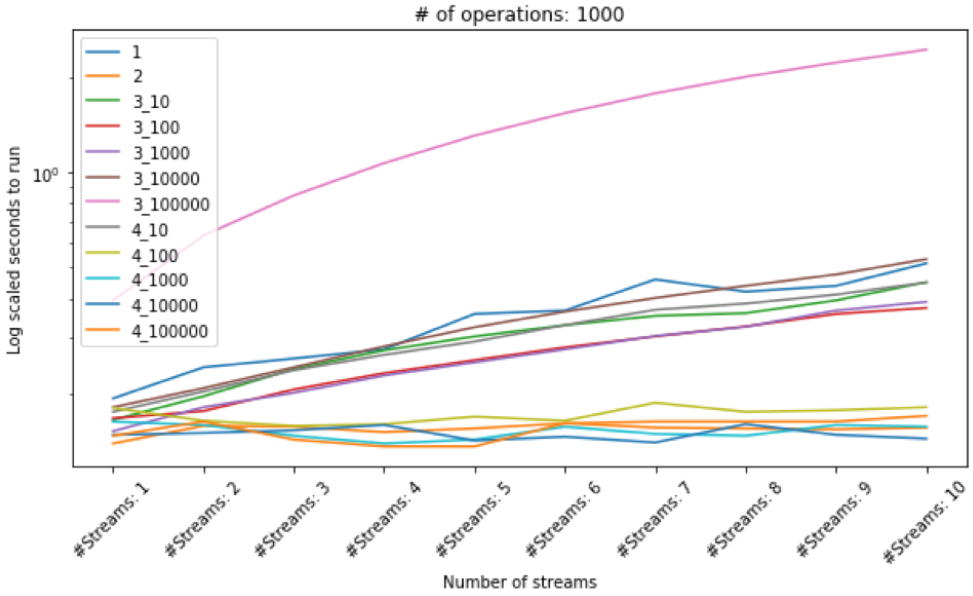

Figure 2.1: Results on 4 kilobyte file size for all I/O implementation

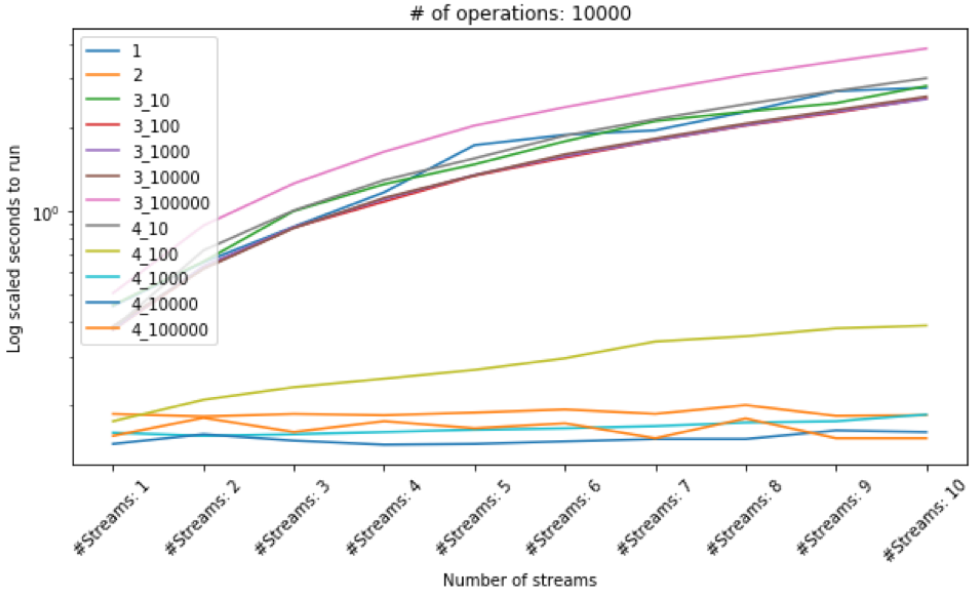

Figure 2.2: Results on 40 kilobyte file size for all I/O implementation

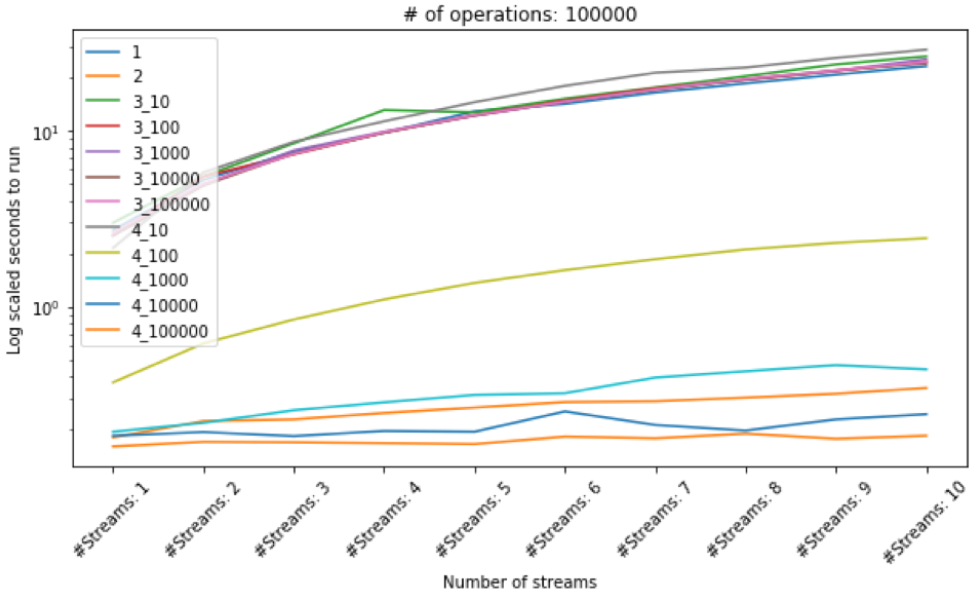

Figure 2.3: Results on 400 kilobyte file size for all I/O implementation

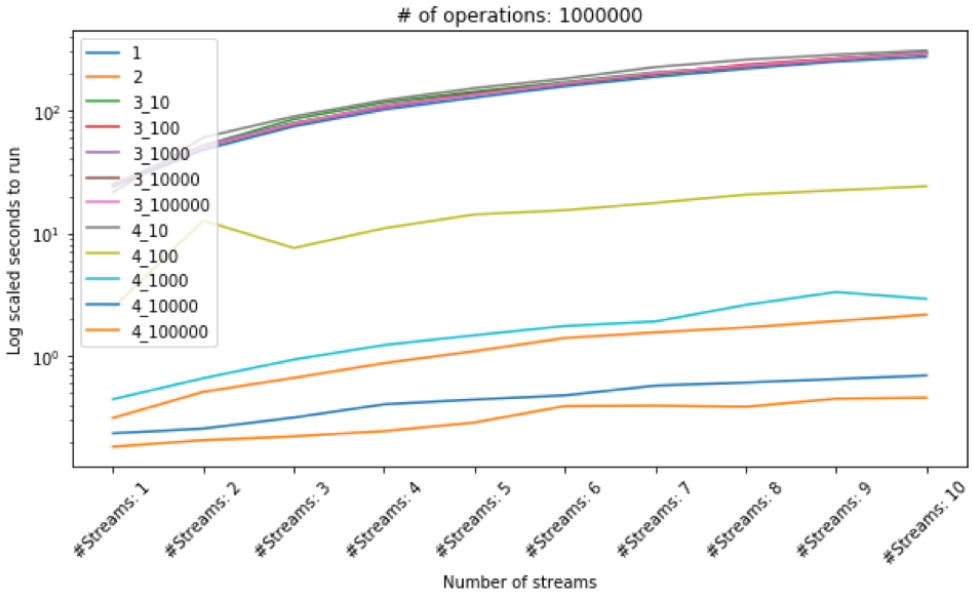

Figure 2.4: Results on 4 megabyte file size for all I/O implementation

From these observations it's already clear to see that some stream types outperforms others. In all the cases I/O type four with at least 1mb of buffer size is among the shortest running times, along with I/O type 2.

In our observations, we also observed that 3rd implementation performs poorly if we increase the buffersize more than *8192 bytes* because internally it is still performing read and write for each integer using the *DataInput/OutputStream* java class. This behavior contradicts our expected behavior. So, the optimal buffersize of *8192 bytes* for 3rd implementation is same as the one used as default in *BufferedInput/OutputStream* class for 2nd implementation.

For a clearer visualization of performance for 2nd and 4th stream implementation, the following graphs will help us to give better conclusion for these cases. Notice that the y-axis now display the total time spent in seconds **without log scale.**

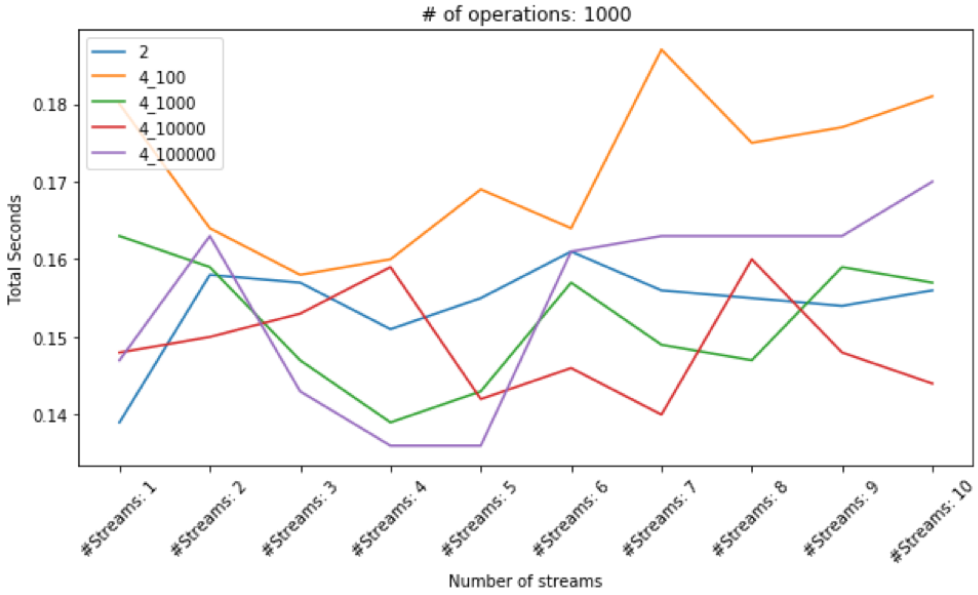

Figure 2.5: Results on 4 kilobyte file size for 2nd and 4th I/O implementation

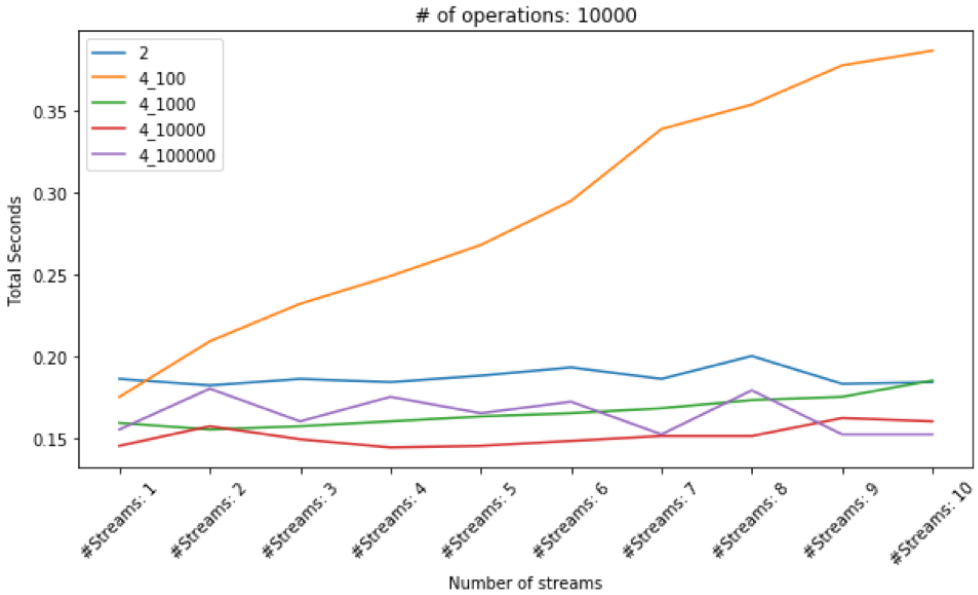

Figure 2.6: Results on 40 kilobyte file size for 2nd and 4th I/O implementation

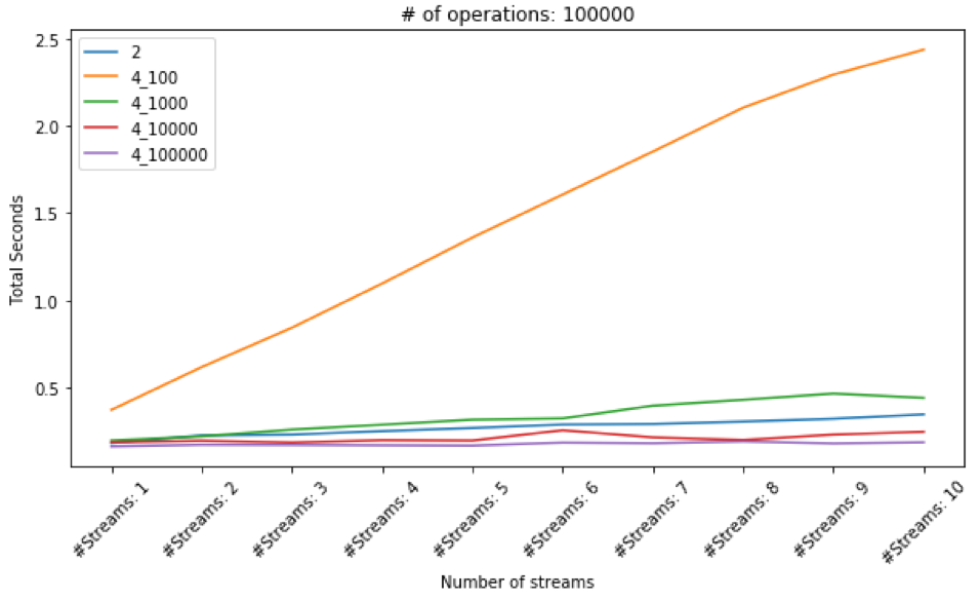

Figure 2.7: Results on 400 kilobyte file size for 2nd and 4th I/O implementation

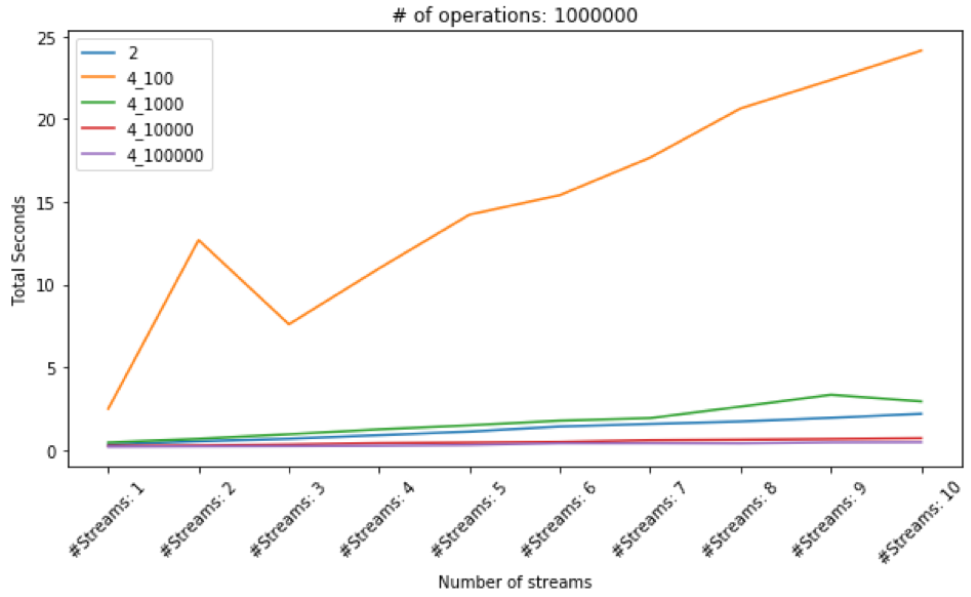

Figure 2.8: Results on 4 megabyte file size for 2nd and 4th I/O implementation

And here we can see that even though for a small number of operations N=1000 there is no clearer better stream type, for the cases that the buffer size of the forth stream type is bigger than the memory necessary to hold the whole dataset in the buffer it outperforms the second type of stream. It's important to notice that in real life usually you can't have a buffer that is big enough to hold all your data, we decided to run another experiment, this time considering 10,000,000 operations, to check if this pattern would still be seen. The following graph show the results for this run.

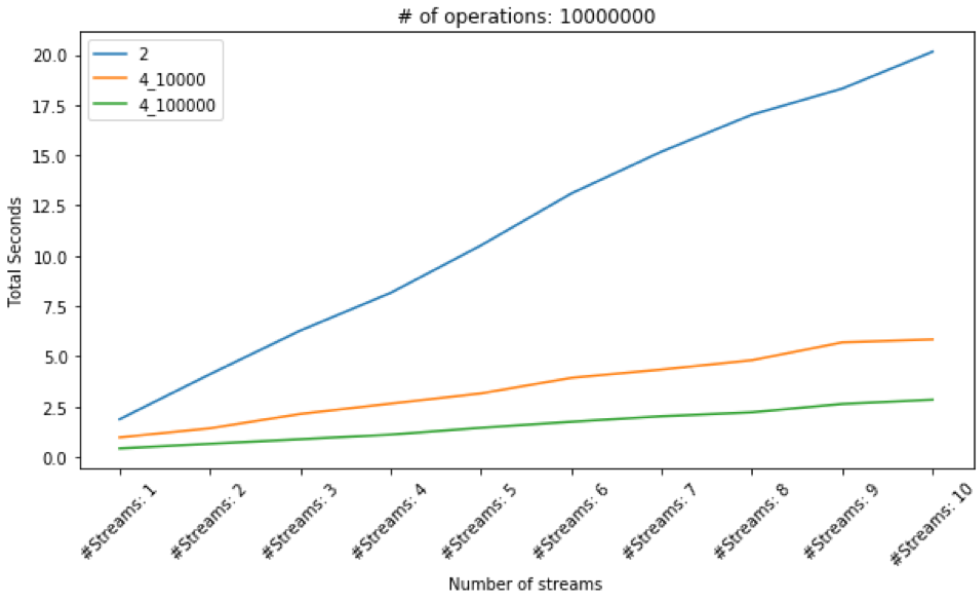

Figure 2.9: Results on 40 megabyte file size for 2nd and 4th I/O implementation

With this final graph, we can conclude that for a big enough number of operations, stream type 4 outperforms type 2 even if the buffer size is not big enough to fit the whole dataset in memory.

The complete detailed results that yielded this conclusion can be checked in the **AppendixA** for each type of stream and different number of operations.

2.3 Expected Behavior vs Experimental Observations

For this first benchmark we compared the different versions of the streams to check their performance on the different sizes of datasets. From the results we can see that as predicted before the first and third algorithms have the worse performance among them all, which is explained by the fact that they do the biggest number of I/O operations, being one for each integer in the file.

The second stream type displayed a good performance for smaller files, but as the file size grows it starts being outperformed by the fourth type with a increased buffer size. That happens even in the case where the buffer size of the fourth version of streams is not big enough to fit the whole loaded data into memory.

Given that the observed results complied with the predictions made previously, we thus decided to use the fourth type of stream which is implemented using memory mapping to run the benchmark for the multiway merge-sort algorithm.

Chapter 3

Multi-way merge-sort

3.1 Implementation

The multi-way merge sort algorithm was implemented in *MultiwayMerge.java* file, it consists of single class MultiwayMerge which has a single constructor, one public and one private method. In order to create MultiwayMerge object, it is necessary to define *streamResolver, initialMemAvailable (which refers to N), numberOfSortingStreams (which refers to d) and numberOfIOStreams (which refers to M)*.

The merge method takes *List<InputStream>* as an argument, it merges and writes them into a file (filePath also provided by arguments) simultaneously. Within this method, there were used 2 collections of java.util package:

- **PriorityQueue:** This was used to complete in-memory sorting of elements.
- **ArrayList:** This was used for keeping track of (top element)-(index of stream) mapping.

The method works like this:

- It looks up first elements of all input streams, it puts them into both priority queue (for sorting) and array (for keeping track of indexes)
- While there is data in the priority queue:
  - It removes the first element from the priority queue, writes it into the result file, and checks the index of input stream from where this element was derived and removes this record from the array.
  - If this input stream has more elements, then it reads the new one and puts in into the priority queue and array.

So assuming that all input streams for this method were sorted (which is provided by both sort and merge methods), this method will merge them into a single file.

The sort method takes a path to the source as an input and sorts it using merge method described above. This method also uses 3 collections of java.util package:

- **PriorityQueue:** This was used to complete in-memory sorting of elements
- **ArrayList:** This was used for keeping track of already sorted parts of the array.
- **LinkedList:** This is to keep track of new temporary files created during the sort

The method works like this:

- It reads the source array in portions defined by the list of arguments from constructor: it sequentially reads integers and put them into PriorityQueue. As soon as the number of integers read reaches initialMemAvailable it flushes the sorted array into a temporary file, saves the name of the temporary file into an array and continues the same procedure until it reaches the end of the input file.
- It iterates over temporary files names from the array, derives input streams from file names and passes them into the merge method, where they got merged in the way described above and saved into the next temporary file, which also gets added as the last record of the linked list. This loop ends when the linked list is empty, then the last file written is the result file.

All temporary files, so as the result file are written in tmp folder under the project root.

In this project, we used java.util.PriorityQueue because it was given as an example in the project documentation and, moreover, having the same asymptotic behavior for sorting - O(n log n) - it is more convenient to use compared to for example standard sorting algorithm (e.g. quick-sort): as soon as an element is added to priority queue it gets sorted, so at any given point of time the priority queue is sorted.

For I/O streams there might a be a problem with exceeding the number of possible streams opened in the system due to peculiar properties of our implementation and Java: if the number of opened I/O streams exceeds the limit, the program will crash. This is happening even if the restriction on max I/O provided on an object level and it is more likely to happen with large input size and small memory available size. In order to avoid it, we operate with paths to files, open streams only at those moments when it is essential and reuse the same I/O streams multiple times.

3.2 Expected behavior

For the current implementation the next conclusions were made:

- **N (size of the input file)** is a directly proportional relationship with the total number of I/Os: with growth of N the total number of I/Os will also grow.

- **d (number of streams to merge in one pass)** is an inverse proportional relationship with the total number of I/Os: with growth of d the total number of I/Os will decrease.
- **M (the size of main memory available)** is an inverse proportional relationship with the total number of I/Os: with growth of M the total number of I/Os will decrease.

Based on these conclusions formula that approximates the total number of I/Os was derived:
*2*N/M**[1]** + d*N/M**[2]** * 2*N/M/d**[3]***

where,

[1] - the first part of the formula estimates the number of I/Os for reading the source array, sorting and writing it back. Coefficient 2 comes from the fact that there we need to both read and write.

[2] - the second part of the formula relates to estimating the number of nodes in the merge tree plan: having N/M arrays and the lowest level and the fact that each node can have at most d children, the rough estimation of the number of elements in the tree (which relates to the number of merge method calls) is d*N/M.

[3] - the third part of the formula estimates the number of I/Os that need to be taken for each merge call. Coefficient 2 comes from the fact that there we need to both read and write.

The estimation of the total I/Os for a given formula can be found in the **AppendixA** of this document.

3.3 Experimental observations

In order to compare the performance Multiway merge-sort, the following experiment was made:

- For a given number of sorting streams d and an initial available memory M, 5 runs of the Multiway merge-sort are made over a file with 1,000,000 integers.
- After the 5 runs, the average time in seconds for such scenario is then recorded as the runtime for the experiment.

Given the conclusions reached in the previous "streams" sections, all experiments for Multiway merge-sort are ran using stream type number 4. Moreover, to avoid that the buffer size for the stream is big enough to fit all memory, a buffer size of 1 megabyte is used for all the experiments.

The following graph shows the results of the experiments, where the y-axis is the total seconds for each scenario, the x-axis is the number of sorting streams in the experiment and the legend shows the different sizes of available memory.

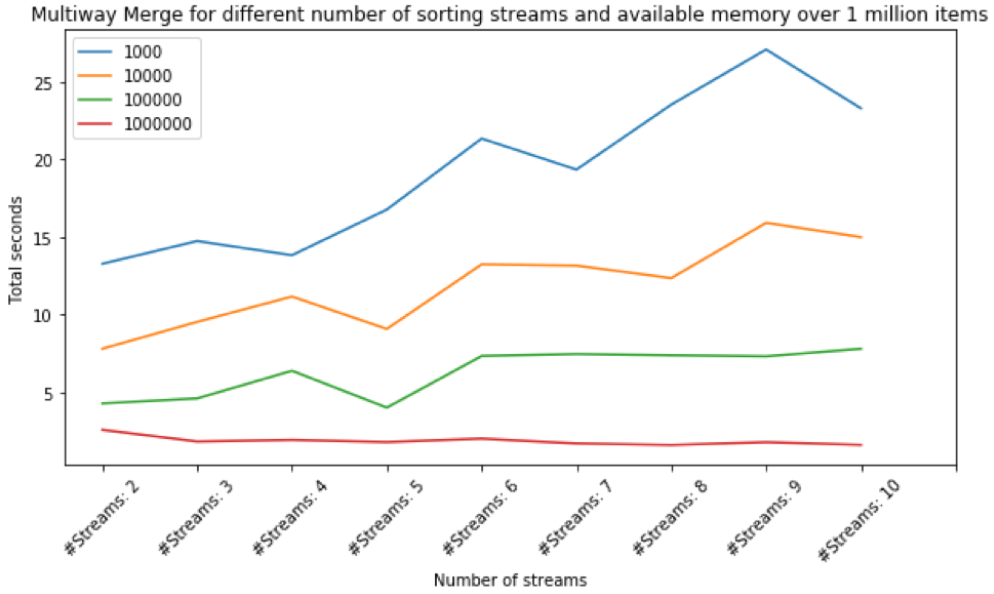

Figure 3.1: Results of Multiway merge on different parameters

3.4 Expected behavior vs Experimental observations

Given the function derived for multiway merge-sort, the following graph shows how the algorithm will behave in terms of the total number of I/Os, having a fixed N (=1000000) and variables M and d:

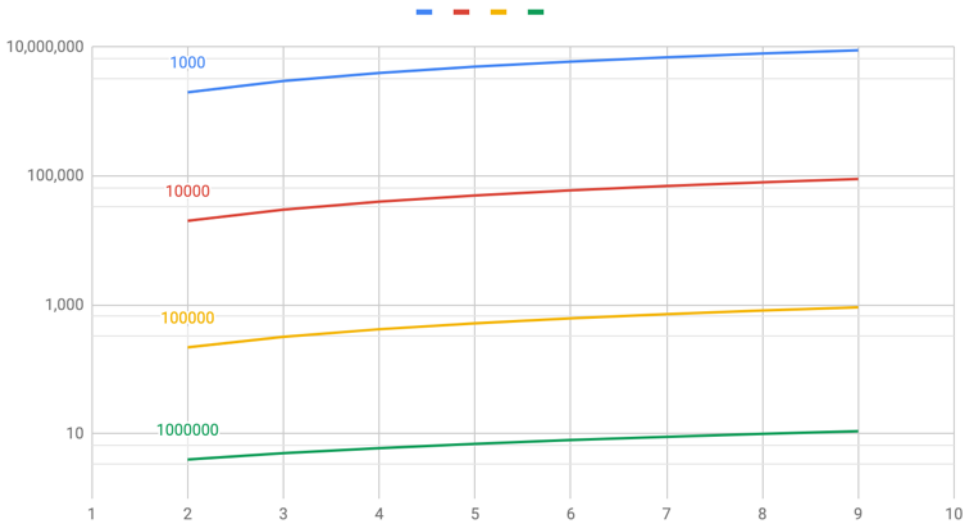

Figure 3.2: Multiway merge algorithm behavior in terms of total number of I/Os

As we can see, this graph correlates to the experiments' runtime in terms of the rank of each graph (e.g. the fastest and the slowest one) and also shows dependencies described in the Expected behavior chapter.

Chapter 4

Conclusion

I/O's are one of the most time-consuming operations, dealing with it sometimes can be challenging but essential to optimize a program. This research was aimed to investigate the performance of various types of IO streams and the sensitivity of the multiway merge-sort algorithm on its parameters. Based on the expected behavior estimation, implementation analysis, and experiments the following conclusions were derived for IOs streams:

- single reads and writes might work really well with small size files, but with size growth, the execution might time grow substantially;
- using buffers solve this problem at some point, but only by a constant multiplication factor (which is defined by buffer size); in any case, even this approach won't work really well with large files;
- memory mapping is proved to be the best choice since it behaves less sensitively to files of different sizes;

Based on the expected behavior estimation, implementation analysis, and experiments the following conclusions were derived for the merge sort algorithm:

- Free memory available for sorting looks like the most significant parameter, the total number of I/O's highly depends on it;
- Number of streams does affects performance, but has way less weight influence when compared with free memory available.

Appendix A

Appendix

A.1 Results for I/O Benchmarks

All complete result tables display the value obtained for the first to fifth run for a given I/O stream type and file size, along with the average of the five runs and the number of streams used for those runs. All values are in seconds.

A.1.1 Results for the first I/O stream:

- File size: 1,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	0.199	0.169	0.197	0.201	0.198	0.193
#Streams: 2	0.248	0.218	0.279	0.217	0.249	0.242
#Streams: 3	0.27	0.269	0.255	0.244	0.25	0.258
#Streams: 4	0.275	0.272	0.273	0.28	0.28	0.276
#Streams: 5	0.316	0.312	0.35	0.412	0.395	0.357
#Streams: 6	0.359	0.362	0.395	0.371	0.344	0.366
#Streams: 7	0.495	0.492	0.456	0.392	0.46	0.459
#Streams: 8	0.425	0.465	0.411	0.402	0.399	0.42
#Streams: 9	0.451	0.433	0.434	0.445	0.426	0.438
#Streams: 10	0.572	0.55	0.486	0.489	0.485	0.516

- File size: 10,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	0.407	0.408	0.509	0.439	0.511	0.455
#Streams: 2	0.669	0.624	0.659	0.661	0.655	0.654
#Streams: 3	0.934	0.878	0.862	0.859	0.864	0.879
#Streams: 4	1.139	1.141	1.188	1.206	1.17	1.169
#Streams: 5	1.984	2.124	1.672	1.448	1.415	1.729
#Streams: 6	1.784	2.057	2.254	1.706	1.616	1.883
#Streams: 7	1.937	1.876	1.904	2.006	2.057	1.956
#Streams: 8	2.281	2.417	2.285	2.22	2.192	2.279
#Streams: 9	2.797	2.827	2.582	2.916	2.408	2.706
#Streams: 10	2.669	3.031	2.69	2.845	2.693	2.786

- File size: 100,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	2.649	2.636	2.685	2.706	2.874	2.71
#Streams: 2	5.573	5.386	4.915	5.298	4.998	5.234
#Streams: 3	7.347	8.245	7.4	7.466	7.353	7.562
#Streams: 4	10.007	9.269	9.332	9.875	9.97	9.691
#Streams: 5	11.491	13.616	12.902	13.826	12.318	12.831
#Streams: 6	15.273	13.707	14.53	13.629	13.647	14.157
#Streams: 7	16.063	16.524	15.731	16.369	17.332	16.404
#Streams: 8	19.179	18.101	18.379	18.829	17.991	18.496
#Streams: 9	20.781	21.447	20.348	20.421	20.78	20.755
#Streams: 10	22.774	22.774	23.16	23.442	23.765	23.183

- File size: 1,000,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	22.408	24.09	23.597	23.145	26.895	24.027
#Streams: 2	48.774	47.613	46.098	47.615	48.977	47.815
#Streams: 3	72.149	71.916	71.839	80.265	74.542	74.142
#Streams: 4	97.527	96.783	110.749	95.893	106.101	101.411
#Streams: 5	132.002	123.482	130.539	122.494	127.195	127.142
#Streams: 6	164.449	152.072	158.102	150.969	160.116	157.142
#Streams: 7	186.774	183.78	181.297	180.923	204.595	187.474
#Streams: 8	232.844	225.078	212.388	209.372	210.86	218.108
#Streams: 9	244.74	250.836	254.576	248.623	251.361	250.027
#Streams: 10	265.719	268.247	288.733	270.847	271.969	273.103

A.1.2 Results for the second I/O stream:

- File size: 1,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	0.17	0.134	0.132	0.129	0.13	0.139
#Streams: 2	0.171	0.167	0.149	0.163	0.142	0.158
#Streams: 3	0.158	0.156	0.157	0.166	0.148	0.157
#Streams: 4	0.136	0.134	0.17	0.164	0.153	0.151
#Streams: 5	0.141	0.16	0.149	0.158	0.168	0.155
#Streams: 6	0.149	0.173	0.167	0.159	0.158	0.161
#Streams: 7	0.157	0.155	0.158	0.155	0.153	0.156
#Streams: 8	0.154	0.156	0.156	0.154	0.154	0.155
#Streams: 9	0.156	0.156	0.154	0.153	0.153	0.154
#Streams: 10	0.154	0.156	0.156	0.155	0.157	0.156

- File size: 10,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	0.169	0.177	0.181	0.192	0.209	0.186
#Streams: 2	0.18	0.173	0.19	0.185	0.181	0.182
#Streams: 3	0.194	0.184	0.172	0.18	0.202	0.186
#Streams: 4	0.184	0.171	0.164	0.199	0.203	0.184
#Streams: 5	0.172	0.191	0.183	0.176	0.219	0.188
#Streams: 6	0.194	0.164	0.223	0.179	0.203	0.193
#Streams: 7	0.177	0.186	0.168	0.181	0.217	0.186
#Streams: 8	0.222	0.185	0.21	0.199	0.184	0.2
#Streams: 9	0.201	0.198	0.169	0.169	0.177	0.183
#Streams: 10	0.183	0.187	0.184	0.185	0.182	0.184

- File size: 100,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	0.2	0.168	0.171	0.188	0.173	0.18
#Streams: 2	0.243	0.219	0.236	0.212	0.203	0.223
#Streams: 3	0.247	0.226	0.243	0.212	0.211	0.228
#Streams: 4	0.261	0.253	0.224	0.225	0.276	0.248
#Streams: 5	0.252	0.29	0.246	0.269	0.272	0.266
#Streams: 6	0.272	0.277	0.287	0.273	0.323	0.286
#Streams: 7	0.287	0.284	0.278	0.309	0.285	0.289
#Streams: 8	0.309	0.295	0.302	0.316	0.292	0.303
#Streams: 9	0.321	0.322	0.32	0.315	0.316	0.319
#Streams: 10	0.343	0.335	0.337	0.339	0.364	0.344

- File size: 1,000,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	0.302	0.283	0.363	0.298	0.323	0.314
#Streams: 2	0.56	0.517	0.5	0.505	0.476	0.512
#Streams: 3	0.682	0.658	0.662	0.664	0.659	0.665
#Streams: 4	0.86	0.875	0.885	0.847	0.918	0.877
#Streams: 5	1.049	1.107	1.086	1.113	1.129	1.097
#Streams: 6	1.428	1.443	1.523	1.309	1.315	1.404
#Streams: 7	1.501	1.629	1.617	1.572	1.483	1.56
#Streams: 8	1.728	1.69	1.736	1.698	1.692	1.709
#Streams: 9	1.939	1.933	1.928	1.937	1.9	1.927
#Streams: 10	2.135	2.113	2.326	2.131	2.179	2.177

- File size: 10,000,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	1.778	2.063	1.861	1.797	1.868	1.873
#Streams: 2	4.075	4.171	4.129	4.072	4.116	4.113
#Streams: 3	6.471	6.236	6.255	6.231	6.215	6.282
#Streams: 4	8.212	8.049	8.288	8.111	8.128	8.158
#Streams: 5	10.329	10.335	10.865	10.257	10.801	10.517
#Streams: 6	13.193	12.71	12.798	12.596	14.206	13.101
#Streams: 7	16.093	14.916	14.389	15.008	15.514	15.184
#Streams: 8	17.039	18.587	16.325	16.36	16.779	17.018
#Streams: 9	19.123	18.332	17.902	17.616	18.618	18.318
#Streams: 10	20.2	19.589	19.811	20.456	20.703	20.152

A.1.3 Results for the third I/O stream:

Buffer size: 10 bytes

- File size: 1,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	0.196	0.158	0.156	0.159	0.155	0.165
#Streams: 2	0.19	0.181	0.191	0.212	0.204	0.196
#Streams: 3	0.234	0.235	0.25	0.244	0.234	0.239
#Streams: 4	0.276	0.303	0.256	0.275	0.262	0.274
#Streams: 5	0.343	0.292	0.33	0.277	0.273	0.303
#Streams: 6	0.308	0.326	0.308	0.344	0.359	0.329
#Streams: 7	0.338	0.375	0.333	0.388	0.327	0.352
#Streams: 8	0.347	0.368	0.389	0.339	0.354	0.359
#Streams: 9	0.4	0.393	0.426	0.369	0.384	0.394
#Streams: 10	0.407	0.417	0.469	0.505	0.454	0.45

- File size: 10,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	0.471	0.506	0.49	0.392	0.437	0.459
#Streams: 2	0.72	0.657	0.635	0.637	0.642	0.658
#Streams: 3	0.887	1.04	1.133	0.962	0.981	1.001
#Streams: 4	1.207	1.375	1.205	1.215	1.254	1.251
#Streams: 5	1.48	1.503	1.472	1.453	1.467	1.475
#Streams: 6	1.676	1.794	1.892	1.809	1.754	1.785
#Streams: 7	2.137	1.971	2.066	1.945	2.441	2.112
#Streams: 8	2.542	2.135	2.205	2.335	2.201	2.284
#Streams: 9	2.379	2.407	2.49	2.427	2.548	2.45
#Streams: 10	2.967	2.612	2.735	2.79	3.022	2.825

- File size: 100,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	3.729	2.877	2.799	2.872	2.657	2.987
#Streams: 2	5.174	5.382	5.186	6.068	5.597	5.481
#Streams: 3	9.035	7.953	8.66	8.082	8.501	8.446
#Streams: 4	14.224	12.433	12.282	12.326	13.973	13.048
#Streams: 5	13.206	13.011	12.375	12.356	12.427	12.675
#Streams: 6	15.01	15.383	15.005	15.491	14.846	15.147
#Streams: 7	17.381	17.881	17.133	17.476	17.791	17.532
#Streams: 8	20.34	20.122	19.697	20.973	20.586	20.344
#Streams: 9	24.562	23.898	24.147	23.301	22.607	23.703
#Streams: 10	26.756	30.061	25.461	24.357	25.209	26.369

- File size: 1,000,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	25.347	25.839	25.2	24.585	24.443	25.083
#Streams: 2	50.437	54.215	52.965	51.757	49.332	51.741
#Streams: 3	78.565	82.777	87.33	85.1	90.286	84.812
#Streams: 4	112.923	107.645	119.418	118.939	122.615	116.308
#Streams: 5	146.315	140.685	145.706	134.887	144.605	142.44
#Streams: 6	171.701	172.973	159.482	180.637	165.731	170.105
#Streams: 7	196.539	211.244	203.648	212.314	198.387	204.426
#Streams: 8	223.589	224.831	229.067	233.982	220.885	226.471
#Streams: 9	269.817	264.542	254.851	260.804	269.807	263.964
#Streams: 10	296.234	305.128	293.763	302.317	304.762	300.441

Buffer size: 100 bytes

- File size: 1,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	0.229	0.155	0.158	0.149	0.146	0.167
#Streams: 2	0.176	0.177	0.174	0.176	0.177	0.176
#Streams: 3	0.2	0.2	0.203	0.211	0.216	0.206
#Streams: 4	0.227	0.25	0.234	0.224	0.227	0.232
#Streams: 5	0.257	0.271	0.252	0.248	0.248	0.255
#Streams: 6	0.279	0.291	0.277	0.276	0.276	0.28
#Streams: 7	0.308	0.298	0.301	0.3	0.307	0.303
#Streams: 8	0.324	0.326	0.331	0.324	0.324	0.326
#Streams: 9	0.354	0.345	0.346	0.392	0.35	0.357
#Streams: 10	0.381	0.371	0.367	0.38	0.367	0.373

- File size: 10,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	0.38	0.373	0.366	0.385	0.367	0.374
#Streams: 2	0.612	0.639	0.623	0.617	0.623	0.623
#Streams: 3	0.878	0.909	0.857	0.859	0.855	0.872
#Streams: 4	1.081	1.08	1.088	1.093	1.075	1.083
#Streams: 5	1.319	1.349	1.344	1.324	1.374	1.342
#Streams: 6	1.586	1.549	1.546	1.54	1.572	1.559
#Streams: 7	1.808	1.797	1.799	1.794	1.79	1.798
#Streams: 8	2.023	2.038	2.061	2.019	2.03	2.034
#Streams: 9	2.26	2.262	2.272	2.265	2.263	2.264
#Streams: 10	2.508	2.528	2.521	2.612	2.587	2.551

- File size: 100,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	2.544	2.508	2.493	2.609	2.507	2.532
#Streams: 2	4.861	5.816	6.582	4.935	5.157	5.47
#Streams: 3	7.265	7.191	7.31	7.585	7.205	7.311
#Streams: 4	9.847	9.537	9.91	9.577	9.693	9.713
#Streams: 5	12.447	11.984	12.724	11.936	11.993	12.217
#Streams: 6	14.485	14.218	14.817	14.76	15.771	14.81
#Streams: 7	16.937	16.815	17.649	17.379	18.176	17.391
#Streams: 8	19.097	19.441	19.198	19.05	20.119	19.381
#Streams: 9	21.682	21.437	21.758	22.299	21.595	21.754
#Streams: 10	24.009	24.27	25.269	24.114	25.056	24.544

- File size: 1,000,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	24.429	24.159	24.114	25.305	25.667	24.735
#Streams: 2	50.842	51.794	47.861	51.788	52.514	50.96
#Streams: 3	77.843	76.034	75.044	78.537	78.026	77.097
#Streams: 4	109.21	103.533	111.482	99.691	116.183	108.02
#Streams: 5	135.522	137.057	137.74	135.907	124.987	134.243
#Streams: 6	171.133	157.973	169.717	160.041	160.621	163.897
#Streams: 7	211.91	190.868	199.126	195.267	191.289	197.692
#Streams: 8	221.615	245.001	237.381	237.069	239.903	236.194
#Streams: 9	269.757	263.738	264.793	258.264	269.38	265.186
#Streams: 10	287.16	287.15	285.048	295.081	286.977	288.283

Buffer size: 1,000 bytes

- File size: 1,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	0.154	0.149	0.153	0.149	0.153	0.152
#Streams: 2	0.186	0.186	0.176	0.18	0.178	0.181
#Streams: 3	0.2	0.202	0.199	0.203	0.199	0.201
#Streams: 4	0.234	0.224	0.225	0.227	0.232	0.228
#Streams: 5	0.25	0.258	0.248	0.253	0.248	0.251
#Streams: 6	0.275	0.281	0.278	0.272	0.275	0.276
#Streams: 7	0.306	0.3	0.3	0.31	0.305	0.304
#Streams: 8	0.322	0.324	0.331	0.323	0.327	0.325
#Streams: 9	0.354	0.346	0.352	0.352	0.427	0.366
#Streams: 10	0.443	0.378	0.375	0.382	0.373	0.39

- File size: 10,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	0.373	0.376	0.369	0.384	0.373	0.375
#Streams: 2	0.615	0.615	0.651	0.646	0.636	0.633
#Streams: 3	0.878	0.901	0.876	0.862	0.857	0.875
#Streams: 4	1.111	1.096	1.113	1.101	1.101	1.104
#Streams: 5	1.334	1.381	1.333	1.37	1.331	1.35
#Streams: 6	1.587	1.558	1.592	1.566	1.591	1.579
#Streams: 7	1.802	1.803	1.798	1.797	1.792	1.798
#Streams: 8	2.04	2.033	2.021	2.024	2.081	2.04
#Streams: 9	2.289	2.282	2.28	2.272	2.283	2.281
#Streams: 10	2.496	2.539	2.563	2.562	2.514	2.535

- File size: 100,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	2.493	2.655	2.505	2.5	2.493	2.529
#Streams: 2	4.905	4.86	4.914	4.872	4.965	4.903
#Streams: 3	7.208	7.324	7.556	9.131	7.282	7.7
#Streams: 4	9.692	9.722	9.597	9.738	10.452	9.84
#Streams: 5	12.037	12.069	12.553	11.9	12.125	12.137
#Streams: 6	14.609	14.433	14.656	14.495	14.485	14.536
#Streams: 7	17.103	16.7	17.135	16.734	17.551	17.045
#Streams: 8	19.677	19.285	19.27	19.01	20.016	19.452
#Streams: 9	21.767	21.454	21.499	22.339	21.891	21.79
#Streams: 10	27.301	23.943	24.193	26.512	25.011	25.392

- File size: 1,000,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	24.738	24.726	24.257	25.634	26.039	25.079
#Streams: 2	52.867	51.893	52.225	49.343	53.248	51.915
#Streams: 3	84.703	77.013	82.698	74.395	77.718	79.305
#Streams: 4	98.497	109.299	102.418	110.929	98.378	103.904
#Streams: 5	138.375	133.947	143.744	129.681	128.831	134.916
#Streams: 6	163.831	157.008	164.351	173.011	163.033	164.247
#Streams: 7	204.297	196.346	191.747	211.928	195.287	199.921
#Streams: 8	232.379	230.349	215.082	220.986	231.176	225.994
#Streams: 9	252.558	250.513	260.311	254.162	265.995	256.708
#Streams: 10	291.606	295.86	288.52	302.785	297.985	295.351

Buffer size: 10,000 bytes

- File size: 1,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	0.248	0.164	0.169	0.165	0.16	0.181
#Streams: 2	0.21	0.212	0.206	0.206	0.204	0.208
#Streams: 3	0.243	0.241	0.242	0.24	0.242	0.242
#Streams: 4	0.287	0.281	0.286	0.279	0.279	0.282
#Streams: 5	0.32	0.332	0.325	0.319	0.322	0.324
#Streams: 6	0.372	0.356	0.364	0.368	0.353	0.363
#Streams: 7	0.4	0.393	0.404	0.411	0.398	0.401
#Streams: 8	0.429	0.447	0.448	0.432	0.432	0.438
#Streams: 9	0.478	0.474	0.484	0.472	0.472	0.476
#Streams: 10	0.513	0.54	0.549	0.531	0.528	0.532

- File size: 10,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	0.379	0.384	0.384	0.388	0.389	0.385
#Streams: 2	0.618	0.614	0.625	0.614	0.625	0.619
#Streams: 3	0.869	0.872	0.871	0.865	0.87	0.869
#Streams: 4	1.149	1.112	1.107	1.114	1.103	1.117
#Streams: 5	1.34	1.351	1.361	1.339	1.345	1.347
#Streams: 6	1.587	1.638	1.6	1.596	1.587	1.602
#Streams: 7	1.812	1.82	1.816	1.857	1.825	1.826
#Streams: 8	2.06	2.064	2.075	2.075	2.075	2.07
#Streams: 9	2.314	2.343	2.307	2.298	2.305	2.313
#Streams: 10	2.635	2.604	2.572	2.54	2.587	2.588

- File size: 100,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	2.515	2.509	2.518	2.549	2.553	2.529
#Streams: 2	4.842	4.853	4.883	4.821	4.937	4.867
#Streams: 3	7.442	7.481	7.356	7.204	7.419	7.38
#Streams: 4	9.564	9.601	10.044	9.632	9.763	9.721
#Streams: 5	12.114	12.334	11.867	12.217	12.12	12.13
#Streams: 6	14.523	14.366	14.842	14.38	14.858	14.594
#Streams: 7	16.849	17.238	16.625	17.399	16.858	16.994
#Streams: 8	18.997	20.747	18.964	19.076	19.839	19.525
#Streams: 9	21.524	21.965	22.643	22.2	21.384	21.943
#Streams: 10	23.855	23.991	24.582	24.042	23.866	24.067

- File size: 1,000,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	25.577	24.042	24.577	24.871	25.028	24.819
#Streams: 2	51.935	48.915	53.312	49.242	49.594	50.6
#Streams: 3	78.757	80.592	76.848	77.007	76.538	77.948
#Streams: 4	100.888	117.069	116.795	106.075	108.145	109.794
#Streams: 5	136.95	134.515	141.781	144.422	136.267	138.787
#Streams: 6	161.804	168.26	167.29	162.494	169.44	165.858
#Streams: 7	201.223	204.608	196.728	188.85	192.143	196.71
#Streams: 8	223.859	227.385	233.424	225.528	231.147	228.269
#Streams: 9	255.809	266.624	262.986	252.851	260.217	259.697
#Streams: 10	297.378	293.334	282.69	290.285	290.999	290.937

Buffer size: 100,000 bytes

- File size: 1,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	0.439	0.386	0.385	0.376	0.383	0.394
#Streams: 2	0.657	0.627	0.652	0.617	0.607	0.632
#Streams: 3	0.847	0.848	0.839	0.838	0.856	0.846
#Streams: 4	1.06	1.086	1.076	1.066	1.068	1.071
#Streams: 5	1.301	1.336	1.301	1.286	1.314	1.308
#Streams: 6	1.548	1.545	1.548	1.534	1.528	1.541
#Streams: 7	1.796	1.741	1.764	1.778	1.822	1.78
#Streams: 8	2.047	2.019	1.989	1.997	1.989	2.008
#Streams: 9	2.232	2.202	2.256	2.231	2.216	2.227
#Streams: 10	2.443	2.428	2.481	2.434	2.442	2.446

- File size: 10,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	0.505	0.504	0.514	0.503	0.511	0.507
#Streams: 2	0.877	0.881	0.899	0.887	0.893	0.887
#Streams: 3	1.264	1.25	1.242	1.257	1.277	1.258
#Streams: 4	1.637	1.624	1.649	1.617	1.66	1.637
#Streams: 5	2.029	2.002	1.996	2.044	2.098	2.034
#Streams: 6	2.386	2.371	2.348	2.359	2.381	2.369
#Streams: 7	2.72	2.702	2.719	2.721	2.74	2.72
#Streams: 8	3.087	3.107	3.099	3.118	3.099	3.102
#Streams: 9	3.495	3.453	3.466	3.487	3.444	3.469
#Streams: 10	3.89	3.882	3.813	3.851	3.831	3.853

- File size: 100,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	2.545	2.698	2.586	2.737	2.538	2.621
#Streams: 2	4.953	5.12	4.932	4.949	4.956	4.982
#Streams: 3	7.36	7.442	7.356	7.367	7.277	7.36
#Streams: 4	10.116	9.644	9.699	9.784	9.705	9.79
#Streams: 5	12.07	12.917	11.947	12.063	12.087	12.217
#Streams: 6	14.415	14.926	14.591	14.735	14.26	14.585
#Streams: 7	17.094	16.811	17.334	17.677	16.597	17.103
#Streams: 8	20.294	19.373	19.307	20.201	19.207	19.676
#Streams: 9	21.509	22.475	22.494	21.529	21.653	21.932
#Streams: 10	25.285	24.706	24.907	23.863	23.724	24.497

- File size: 1,000,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	25.411	25.874	25.572	24.73	23.84	25.085
#Streams: 2	49.821	52.859	51.468	50.785	49.172	50.821
#Streams: 3	74.458	79.941	78.032	77.805	78.992	77.846
#Streams: 4	103.172	117.508	108.128	103.726	114.69	109.445
#Streams: 5	133.799	127.189	139.65	129.888	139.148	133.935
#Streams: 6	162.978	166.462	164.547	170.256	168.903	166.629
#Streams: 7	199.051	208.936	206.731	199.91	193.878	201.701
#Streams: 8	216.532	223.911	236.953	247.223	232.079	231.34
#Streams: 9	259.538	262.699	260.613	263.579	267.571	262.8
#Streams: 10	290.581	294.758	297.216	293.689	289.259	293.101

A.1.4 Results for the fourth I/O stream:

Buffer size: 10 bytes

- File size: 1,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	0.197	0.17	0.168	0.168	0.17	0.175
#Streams: 2	0.203	0.205	0.203	0.203	0.203	0.203
#Streams: 3	0.244	0.239	0.234	0.231	0.237	0.237
#Streams: 4	0.267	0.263	0.267	0.259	0.269	0.265
#Streams: 5	0.299	0.285	0.29	0.292	0.293	0.292
#Streams: 6	0.327	0.331	0.328	0.325	0.332	0.329
#Streams: 7	0.377	0.358	0.371	0.379	0.356	0.368
#Streams: 8	0.404	0.379	0.39	0.379	0.374	0.385
#Streams: 9	0.406	0.416	0.414	0.413	0.406	0.411
#Streams: 10	0.447	0.455	0.441	0.443	0.457	0.449

- File size: 10,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	0.375	0.37	0.401	0.375	0.366	0.377
#Streams: 2	0.719	0.717	0.719	0.737	0.716	0.722
#Streams: 3	1.009	1.007	0.997	1.013	1.005	1.006
#Streams: 4	1.286	1.282	1.3	1.31	1.293	1.294
#Streams: 5	1.539	1.552	1.556	1.549	1.555	1.55
#Streams: 6	1.874	1.87	1.861	1.881	1.867	1.871
#Streams: 7	2.141	2.172	2.114	2.152	2.146	2.145
#Streams: 8	2.439	2.454	2.403	2.45	2.402	2.43
#Streams: 9	2.74	2.675	2.724	2.721	2.708	2.714
#Streams: 10	3.069	3.003	3.016	3.001	2.983	3.014

- File size: 100,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	2.102	2.128	2.244	2.171	2.096	2.148
#Streams: 2	5.796	5.804	5.774	5.717	5.738	5.766
#Streams: 3	8.622	8.614	8.647	8.587	8.611	8.616
#Streams: 4	11.316	11.093	11.256	11.216	11.432	11.263
#Streams: 5	14.123	14.261	14.476	14.836	14.619	14.463
#Streams: 6	17.647	17.831	18.353	17.557	18.192	17.916
#Streams: 7	21.57	21.914	20.542	21.05	21.239	21.263
#Streams: 8	22.482	23.162	22.738	21.759	23.591	22.746
#Streams: 9	26.403	24.826	26.12	25.909	25.831	25.818
#Streams: 10	29.736	27.676	28.168	29.226	28.98	28.757

- File size: 1,000,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	20.794	21.82	20.926	22.275	22.091	21.581
#Streams: 2	60.132	62.545	61.642	60.869	56.213	60.28
#Streams: 3	90.934	89.484	92.203	87.558	86.389	89.314
#Streams: 4	119.247	121.004	123.957	119.033	121.307	120.91
#Streams: 5	165.082	145.228	152.671	146.707	150.287	151.995
#Streams: 6	172.106	174.141	176.52	189.803	194.372	181.388
#Streams: 7	238.079	227.418	227.201	223.593	210.862	225.431
#Streams: 8	257.032	244.4	269.95	262.768	259.406	258.711
#Streams: 9	290.361	282.535	277.638	289.549	282.415	284.5
#Streams: 10	302.521	304.053	309.332	317.883	301.231	307.004

Buffer size: 100 bytes

- File size: 1,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	0.178	0.175	0.16	0.195	0.19	0.18
#Streams: 2	0.17	0.174	0.164	0.161	0.151	0.164
#Streams: 3	0.162	0.166	0.156	0.153	0.154	0.158
#Streams: 4	0.155	0.156	0.159	0.157	0.172	0.16
#Streams: 5	0.163	0.165	0.178	0.163	0.177	0.169
#Streams: 6	0.16	0.159	0.165	0.167	0.17	0.164
#Streams: 7	0.186	0.185	0.179	0.186	0.2	0.187
#Streams: 8	0.179	0.172	0.176	0.179	0.17	0.175
#Streams: 9	0.178	0.18	0.175	0.172	0.178	0.177
#Streams: 10	0.181	0.183	0.18	0.181	0.179	0.181

- File size: 10,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	0.173	0.17	0.164	0.169	0.2	0.175
#Streams: 2	0.203	0.221	0.205	0.204	0.213	0.209
#Streams: 3	0.224	0.231	0.227	0.24	0.239	0.232
#Streams: 4	0.248	0.256	0.255	0.246	0.241	0.249
#Streams: 5	0.271	0.268	0.27	0.267	0.265	0.268
#Streams: 6	0.291	0.297	0.3	0.291	0.297	0.295
#Streams: 7	0.341	0.344	0.335	0.343	0.33	0.339
#Streams: 8	0.343	0.347	0.366	0.374	0.34	0.354
#Streams: 9	0.361	0.376	0.388	0.376	0.39	0.378
#Streams: 10	0.384	0.381	0.388	0.392	0.389	0.387

- File size: 100,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	0.368	0.364	0.361	0.39	0.368	0.37
#Streams: 2	0.627	0.62	0.614	0.608	0.612	0.616
#Streams: 3	0.833	0.83	0.84	0.823	0.886	0.842
#Streams: 4	1.052	1.086	1.065	1.112	1.166	1.096
#Streams: 5	1.328	1.289	1.297	1.431	1.456	1.36
#Streams: 6	1.724	1.583	1.602	1.553	1.575	1.607
#Streams: 7	1.804	1.857	1.834	1.781	1.995	1.854
#Streams: 8	2.104	2.058	2.084	2.213	2.072	2.106
#Streams: 9	2.39	2.345	2.27	2.285	2.189	2.296
#Streams: 10	2.263	2.422	2.557	2.409	2.544	2.439

- File size: 1,000,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	2.153	1.851	1.975	3.023	3.405	2.481
#Streams: 2	9.543	13.799	16.846	14.389	8.849	12.685
#Streams: 3	7.706	7.554	7.482	7.415	7.777	7.587
#Streams: 4	11.764	12.896	10.342	10.073	9.803	10.976
#Streams: 5	13.147	15.411	16.847	13.044	12.675	14.225
#Streams: 6	16.606	14.985	14.969	14.994	15.46	15.403
#Streams: 7	17.412	16.826	15.782	19.688	18.667	17.675
#Streams: 8	22.32	20.538	20.302	20.205	19.863	20.646
#Streams: 9	22.642	22.46	23.458	21.725	21.562	22.369
#Streams: 10	24.612	25.018	25.475	23.145	22.487	24.147

Buffer size: 1,000 bytes

- File size: 1,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	0.154	0.192	0.173	0.154	0.141	0.163
#Streams: 2	0.193	0.163	0.141	0.151	0.145	0.159
#Streams: 3	0.146	0.146	0.149	0.15	0.142	0.147
#Streams: 4	0.14	0.14	0.137	0.138	0.14	0.139
#Streams: 5	0.141	0.145	0.145	0.141	0.141	0.143
#Streams: 6	0.161	0.164	0.152	0.16	0.147	0.157
#Streams: 7	0.143	0.145	0.155	0.157	0.146	0.149
#Streams: 8	0.147	0.15	0.147	0.146	0.147	0.147
#Streams: 9	0.162	0.157	0.16	0.159	0.157	0.159
#Streams: 10	0.157	0.162	0.146	0.158	0.163	0.157

- File size: 10,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	0.141	0.155	0.143	0.167	0.188	0.159
#Streams: 2	0.156	0.149	0.157	0.155	0.156	0.155
#Streams: 3	0.161	0.16	0.156	0.155	0.152	0.157
#Streams: 4	0.155	0.158	0.159	0.17	0.158	0.16
#Streams: 5	0.171	0.16	0.161	0.162	0.161	0.163
#Streams: 6	0.164	0.164	0.166	0.166	0.167	0.165
#Streams: 7	0.167	0.169	0.169	0.166	0.171	0.168
#Streams: 8	0.171	0.172	0.179	0.173	0.172	0.173
#Streams: 9	0.171	0.176	0.177	0.176	0.177	0.175
#Streams: 10	0.177	0.177	0.181	0.197	0.193	0.185

- File size: 100,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	0.191	0.189	0.188	0.193	0.209	0.194
#Streams: 2	0.201	0.229	0.224	0.213	0.223	0.218
#Streams: 3	0.244	0.246	0.302	0.246	0.247	0.257
#Streams: 4	0.259	0.331	0.256	0.264	0.317	0.285
#Streams: 5	0.339	0.333	0.296	0.297	0.305	0.314
#Streams: 6	0.333	0.315	0.313	0.325	0.32	0.321
#Streams: 7	0.376	0.356	0.407	0.396	0.437	0.394
#Streams: 8	0.466	0.458	0.405	0.389	0.422	0.428
#Streams: 9	0.423	0.482	0.454	0.498	0.464	0.464
#Streams: 10	0.452	0.431	0.431	0.437	0.443	0.439

- File size: 1,000,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	0.436	0.476	0.4	0.483	0.433	0.446
#Streams: 2	0.644	0.667	0.654	0.654	0.687	0.661
#Streams: 3	0.901	0.935	0.997	0.926	0.927	0.937
#Streams: 4	1.175	1.259	1.375	1.165	1.182	1.231
#Streams: 5	1.488	1.469	1.523	1.475	1.436	1.478
#Streams: 6	2.023	1.701	1.709	1.691	1.664	1.758
#Streams: 7	1.893	1.967	1.904	1.871	1.927	1.912
#Streams: 8	2.144	2.242	2.403	2.927	3.329	2.609
#Streams: 9	3.941	3.581	3.012	3.48	2.599	3.323
#Streams: 10	2.947	2.955	2.944	2.952	2.815	2.923

Buffer size: 10,000 bytes

- File size: 1,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	0.14	0.151	0.151	0.142	0.155	0.148
#Streams: 2	0.155	0.136	0.141	0.152	0.164	0.15
#Streams: 3	0.146	0.154	0.153	0.166	0.145	0.153
#Streams: 4	0.138	0.146	0.142	0.21	0.161	0.159
#Streams: 5	0.142	0.144	0.141	0.139	0.144	0.142
#Streams: 6	0.139	0.146	0.151	0.149	0.143	0.146
#Streams: 7	0.143	0.144	0.137	0.138	0.139	0.14
#Streams: 8	0.169	0.149	0.161	0.166	0.157	0.16
#Streams: 9	0.142	0.153	0.14	0.15	0.157	0.148
#Streams: 10	0.153	0.142	0.141	0.143	0.142	0.144

- File size: 10,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	0.135	0.136	0.15	0.153	0.152	0.145
#Streams: 2	0.154	0.141	0.135	0.209	0.148	0.157
#Streams: 3	0.147	0.148	0.162	0.144	0.145	0.149
#Streams: 4	0.142	0.145	0.144	0.145	0.143	0.144
#Streams: 5	0.145	0.144	0.146	0.147	0.145	0.145
#Streams: 6	0.144	0.148	0.148	0.153	0.145	0.148
#Streams: 7	0.148	0.149	0.151	0.149	0.156	0.151
#Streams: 8	0.151	0.155	0.151	0.149	0.148	0.151
#Streams: 9	0.15	0.156	0.193	0.16	0.151	0.162
#Streams: 10	0.152	0.161	0.168	0.165	0.156	0.16

- File size: 100,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	0.165	0.17	0.167	0.232	0.188	0.184
#Streams: 2	0.184	0.181	0.192	0.224	0.184	0.193
#Streams: 3	0.184	0.188	0.182	0.179	0.18	0.183
#Streams: 4	0.215	0.189	0.2	0.195	0.183	0.196
#Streams: 5	0.182	0.181	0.208	0.19	0.209	0.194
#Streams: 6	0.237	0.221	0.246	0.3	0.263	0.253
#Streams: 7	0.206	0.196	0.244	0.195	0.221	0.212
#Streams: 8	0.2	0.195	0.195	0.194	0.203	0.197
#Streams: 9	0.202	0.2	0.205	0.243	0.29	0.228
#Streams: 10	0.213	0.281	0.237	0.248	0.239	0.244

- File size: 1,000,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	0.216	0.267	0.206	0.262	0.224	0.235
#Streams: 2	0.264	0.261	0.255	0.252	0.251	0.257
#Streams: 3	0.299	0.296	0.302	0.347	0.334	0.316
#Streams: 4	0.384	0.424	0.411	0.385	0.421	0.405
#Streams: 5	0.45	0.472	0.45	0.426	0.418	0.443
#Streams: 6	0.485	0.476	0.485	0.478	0.472	0.479
#Streams: 7	0.584	0.534	0.579	0.612	0.567	0.575
#Streams: 8	0.613	0.616	0.617	0.615	0.584	0.609
#Streams: 9	0.641	0.66	0.664	0.64	0.65	0.651
#Streams: 10	0.684	0.693	0.693	0.705	0.701	0.695

- File size: 10,000,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	2.17	0.68	0.636	0.639	0.708	0.967
#Streams: 2	2.709	1.123	1.093	1.103	1.079	1.421
#Streams: 3	3.431	1.79	1.893	1.673	1.888	2.135
#Streams: 4	4.136	2.655	2.354	2.108	1.993	2.649
#Streams: 5	4.619	2.543	2.933	2.898	2.787	3.156
#Streams: 6	5.43	3.776	3.541	3.569	3.339	3.931
#Streams: 7	5.827	3.8	4.366	4.031	3.699	4.345
#Streams: 8	6.277	4.298	4.237	4.494	4.73	4.807
#Streams: 9	7.4	5.311	5.135	5.348	5.283	5.695
#Streams: 10	8.033	5.379	5.25	5.313	5.229	5.841

Buffer size: 100,000 bytes

- File size: 1,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	0.136	0.156	0.137	0.165	0.141	0.147
#Streams: 2	0.174	0.167	0.15	0.169	0.156	0.163
#Streams: 3	0.154	0.147	0.144	0.136	0.136	0.143
#Streams: 4	0.136	0.133	0.134	0.137	0.138	0.136
#Streams: 5	0.136	0.134	0.137	0.138	0.136	0.136
#Streams: 6	0.143	0.156	0.174	0.187	0.147	0.161
#Streams: 7	0.146	0.183	0.151	0.154	0.18	0.163
#Streams: 8	0.157	0.145	0.186	0.155	0.17	0.163
#Streams: 9	0.161	0.146	0.154	0.188	0.168	0.163
#Streams: 10	0.163	0.168	0.172	0.167	0.178	0.17

- File size: 10,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	0.148	0.171	0.149	0.149	0.156	0.155
#Streams: 2	0.157	0.167	0.188	0.194	0.193	0.18
#Streams: 3	0.186	0.175	0.146	0.145	0.146	0.16
#Streams: 4	0.146	0.158	0.189	0.194	0.186	0.175
#Streams: 5	0.167	0.176	0.177	0.147	0.157	0.165
#Streams: 6	0.184	0.19	0.182	0.152	0.151	0.172
#Streams: 7	0.153	0.151	0.149	0.15	0.159	0.152
#Streams: 8	0.195	0.186	0.149	0.177	0.19	0.179
#Streams: 9	0.153	0.15	0.15	0.151	0.155	0.152
#Streams: 10	0.152	0.15	0.155	0.151	0.152	0.152

- File size: 100,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	0.148	0.159	0.165	0.167	0.16	0.16
#Streams: 2	0.171	0.164	0.177	0.168	0.169	0.17
#Streams: 3	0.18	0.172	0.164	0.158	0.171	0.169
#Streams: 4	0.159	0.171	0.158	0.156	0.191	0.167
#Streams: 5	0.156	0.157	0.176	0.162	0.173	0.165
#Streams: 6	0.185	0.182	0.178	0.182	0.183	0.182
#Streams: 7	0.191	0.171	0.177	0.169	0.184	0.178
#Streams: 8	0.187	0.189	0.192	0.192	0.187	0.189
#Streams: 9	0.174	0.173	0.173	0.191	0.173	0.177
#Streams: 10	0.181	0.184	0.186	0.179	0.189	0.184

- File size: 1,000,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	0.178	0.174	0.175	0.201	0.185	0.183
#Streams: 2	0.195	0.214	0.208	0.211	0.207	0.207
#Streams: 3	0.218	0.222	0.224	0.228	0.217	0.222
#Streams: 4	0.243	0.246	0.249	0.244	0.243	0.245
#Streams: 5	0.281	0.28	0.272	0.279	0.33	0.288
#Streams: 6	0.44	0.421	0.392	0.37	0.336	0.392
#Streams: 7	0.386	0.397	0.427	0.417	0.352	0.396
#Streams: 8	0.377	0.385	0.396	0.389	0.39	0.387
#Streams: 9	0.451	0.454	0.464	0.44	0.441	0.45
#Streams: 10	0.415	0.442	0.424	0.507	0.506	0.459

- File size: 10,000,000 integers.

	1st	2nd	3rd	4th	5th	avg
#Streams: 1	0.39	0.397	0.425	0.49	0.395	0.419
#Streams: 2	0.668	0.793	0.593	0.614	0.594	0.652
#Streams: 3	0.861	0.953	0.851	0.901	0.813	0.876
#Streams: 4	1.087	1.049	1.153	1.084	1.15	1.105
#Streams: 5	1.411	1.393	1.352	1.657	1.421	1.447
#Streams: 6	1.853	1.67	1.785	1.723	1.701	1.746
#Streams: 7	2.18	1.953	1.977	2.045	1.946	2.02
#Streams: 8	2.178	2.252	2.188	2.291	2.175	2.217
#Streams: 9	2.451	2.522	2.679	2.72	2.788	2.632
#Streams: 10	2.834	3.084	2.747	2.765	2.784	2.843

A.2 Results for Multiway merge Benchmarks

All complete result tables display the value obtained for the first to fifth run for a given amount of available memory and number of sorting streams, along with the average of the five runs and the number of streams used for those runs. All values are in seconds and ran over a file with 1 million elements using I/O stream type 4 and a buffer size of 1000 bytes.

A.2.1 Memory available: 1000 bytes

	1st	2nd	3rd	4th	5th	avg
#Streams: 2	12.761	11.313	11.964	15.243	15.136	13.283
#Streams: 3	14.541	16.903	15.222	13.362	13.697	14.745
#Streams: 4	13.563	13.879	13.69	14.297	13.752	13.836
#Streams: 5	17.044	18.102	16.121	16.352	16.254	16.775
#Streams: 6	17.573	16.589	18.489	35.144	18.982	21.355
#Streams: 7	20.423	18.796	19.13	19.299	19.124	19.354
#Streams: 8	23.066	23.309	23.579	23.669	24.088	23.542
#Streams: 9	26.027	27.297	28.212	28.091	25.857	27.097
#Streams: 10	23.035	23.934	21.832	24.106	23.612	23.304

A.2.2 Memory available: 10,000 bytes

	1st	2nd	3rd	4th	5th	avg
#Streams: 2	7.262	8.0	8.139	7.909	7.71	7.804
#Streams: 3	9.284	8.272	8.785	11.197	10.154	9.538
#Streams: 4	11.954	11.08	10.698	11.772	10.35	11.171
#Streams: 5	8.699	8.756	8.669	8.824	10.465	9.083
#Streams: 6	12.87	12.707	12.646	13.423	14.571	13.243
#Streams: 7	13.473	13.42	12.966	13.189	12.757	13.161
#Streams: 8	12.439	12.991	12.335	12.128	11.905	12.36
#Streams: 9	16.456	16.267	15.897	14.755	16.224	15.92
#Streams: 10	16.567	14.633	13.612	15.342	14.854	15.002

A.2.3 Memory available: 100,000 bytes

	1st	2nd	3rd	4th	5th	avg
#Streams: 2	4.455	4.554	4.237	4.14	4.023	4.282
#Streams: 3	4.515	4.565	4.548	4.618	4.766	4.602
#Streams: 4	5.403	4.885	6.938	7.818	6.886	6.386
#Streams: 5	4.062	3.843	3.818	4.391	3.926	4.008
#Streams: 6	7.519	7.252	7.778	6.955	7.183	7.337
#Streams: 7	7.854	6.7	7.78	7.161	7.844	7.468
#Streams: 8	7.057	7.206	7.543	7.681	7.414	7.38
#Streams: 9	7.45	7.171	7.152	6.902	7.894	7.314
#Streams: 10	8.019	7.99	7.951	7.628	7.432	7.804

A.2.4 Memory available: 1,000,000 bytes

	1st	2nd	3rd	4th	5th	avg
#Streams: 2	2.861	2.71	2.957	2.329	2.023	2.576
#Streams: 3	1.714	2.197	1.806	1.693	1.726	1.827
#Streams: 4	1.722	1.929	1.859	2.034	2.09	1.927
#Streams: 5	1.707	1.656	1.768	1.835	1.999	1.793
#Streams: 6	2.085	1.823	2.056	2.209	1.864	2.007
#Streams: 7	1.786	1.558	1.591	1.698	1.87	1.701
#Streams: 8	1.543	1.756	1.549	1.564	1.598	1.602
#Streams: 9	1.665	1.77	1.867	1.812	1.804	1.784
#Streams: 10	1.766	1.591	1.555	1.542	1.591	1.609

A.3 Multiway merge-sort expected behavior with N = 1,000,000

	d								
M	2	3	4	5	6	7	8	9	10
1000	2,002,000	3,002,000	4,002,000	5,002,000	6,002,000	7,002,000	8,002,000	9,002,000	10,002,000
10000	20,200	30,200	40,200	50,200	60,200	70,200	80,200	90,200	100,200
100000	220	320	420	520	620	720	820	920	1,020
1000000	4	5	6	7	8	9	10	11	12

Figure A.1: Results of Multiway merge-sort expected behavior