Université Libre de Bruxelles

Database Systems Architecture

Project Assignment

Algorithms in Secondary Memory

Author:
Pierre-Alexandre Bourdais
Matricule ULB: 000458467
Shafagh Kashef
Matricule ULB: 000455644
Keneth Ubeda
Matricule ULB: 000453317
Group: 12

Supervisor:
Prof. S.Vansummeren

January 7, 2018

UNIVERSITÉ LIBRE DE BRUXELLES

Contents

1 Introduction and Environment

1.1 Abstract

The goal of the project is to implement an external-memory merge-sort algorithm and examine its performance under different parameters (data volume, size of the buffer,..). Different read data and write data functions using such as memory mapping and buffer are implemented in this project and these implementations helped to implement the external-memory merge-sort algorithm.
The language used to implement the algorithms is Java and the IDE is NetBeans.In this Project very large relations with Two-Phase, Multiway Merge-Sort (TPMMS) algorithm is sorted.

The environment of the project:
Machine type: MacBook Air
Hard Disk Type: Macintosh HD
Operating System: MacOs
Memory : 8 Go 1600 MHz DDR3
Total Memory Available: 10Go
Programming Language: Java 8
Libraries : JMH

The merge sort implementation is able to sort disk files consisting of 32-bit integers. And reads data from, and writes data to disk.
Therefore stream classes are developed that can sequentially read and write a file consisting of 32-bit Integers.
Two types of streams are used: input streams and output streams.

The input stream supports the following operation:

- open : open an existing file for reading
- read next : read the next element from the stream
- end of stream : a Boolean operation that returns true if the end of stream has been reached

The output stream supports the following operations:

- create : create a new file
- write : write an element to the stream
- close : close the stream

2 Observations on streams

2.1 First I/O mechanism

2.1.1 Implementation

The first I/O mechanism is quite simple as it requires to read and write one element at a time using the read and write system calls.

- Read function mimic read system call by calling readInt() on a java.io.DataInputStream that is wrapped directly around a java.io.FileInputStream.
- Write function mimic write system call by calling write on a java.io.DataOutputStream that is wrapped directly around a java.io.FileOutputStream.

2.1.2 Expected behavior

Before running any test, a cost formula needs to be build for this implementation. The cost estimates the total number of I/O's that need to be done in function of N and k.

- k - the number of streams, where each stream is a file
- N - the data volumes
- B(R) - the number of blocks that R occupies on disk

The algorithm needs to read B(R) blocks, write B(R) blocks and open B(R) elements.

Cost(1st I/O) : N.B(R) + N.B(R) + k.B(R)
Cost(1st I/O) : 2.N.B(R)+k.B(R)
Cost(1st I/O) : (2.N+k).B(R)

2.2 Second I/O mechanism

2.2.1 Implementation

The second I/O mechanism need an other implementation of read and write functions. This time the read function has to read element by element each one with a size of 32-bit from a stream and stores them in a buffer. And the write function has to write one element (32-bit) from the buffer. More over this mechanism need also a new open function in order to use a buffer.

- Read function mimic fread system call by calling readInt() on a java.io.BufferedInputStream that itself is wrapped around a java.io.FileInputStream.
- Write function mimic fwrite system call by calling writeInt on a java.io.BufferedOutputStream that is wrapped directly around a java.io.FileOutputStream.

2.2.2 Expected behavior

Before running any test, a cost formula needs to be build for this implementation. The cost estimates the total number of I/O's that need to be done in function of N and k.

- k - the number of streams, where each stream is a file
- N - the data volumes
- N(R) - the number of blocks that R occupies on disk

The algorithm needs to read B(R) blocks, write B(R) blocks from a buffer and open B(R) elements.

Cost(2nd I/O) : N.B(R)+k.B(R)
Cost(2nd I/O) : (N+k)B(R)

2.3 Third I/O mechanism

2.3.1 Implementation

The third I/O mechanism is almost the same as the first one implemented. Indeed in this implementation of read and write function it's just need to add a buffer of size B in internal memory. Thus once the buffer becomes empty/full the next B elements are read/written from/to the file.

2.3.2 Expected behavior

Before running any test, a cost formula needs to be build for this implementation. The cost estimates the total number of I/O's that need to be done in function of N, B and k.

- k - the number of streams, where each stream is a file
- N - the size of the input file, measured in number of 32-bit integers.
- B(R) - the number of blocks that R occupies on disk
- b is the size of the buffer in internal memory

Cost(3rd I/O) : (N/b).B(R) + k.B(R)

2.4 Fourth I/O mechanism

2.4.1 Implementation

The fourth I/O mechanism needs to implement a read and write function performed by mapping and unmapping a B element portion of the file into internal memory through memory mapping. Every time the I/O mechanism needs to read and write outside of the of the mapped portion, the next B element of the file is mapped. In order to implement it the map method of the java.nio.channels.FileChannel class was used.

Memory mapped files are special, it allows Java program to access contents directly from memory, this is achieved by mapping whole file or portion of file into memory and operating system takes care of loading page requested and writing into file while application only deals with memory which results in very fast I/O operations. Memory used to load Memory mapped file is outside of Java heap Space. Java programming language supports memory mapped file with the java.nio package and has MappedByteBuffer to read and write from memory.

2.4.2 Expected behavior

Before running any test, a cost formula needs to be build for this implementation. The cost estimates the total number of I/O's that need to be done in function of N, B and k.

- k - the number of streams, where each stream is a file
- N - the size of the input file, measured in number of 32-bit integers.
- b - the size of the buffer in internal memory
- B(R) - the number of blocks that R occupies on disk

Cost(4th I/O) : k.B(R)

2.5 Experimental observations

In order to perform experimentation over our streams classes, two benchmark were created. One benchmark allows to test all the read functions from the four different implementations, and the other one allows to test all the write functions. With the JMH benchmark library the program can apply test for different value of the same parameter and also do warm up before running real test. 2 thread are doing the test in order to get efficient results.

2.6 Configuration

The configuration of the computer that ran the test is the following :

Figure 1: Computer configuration

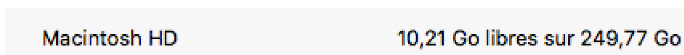

Figure 2: Free space on the computer

2.6.1 Write-Benchmark

So first the goal is to compare the implementations of each write functions. For the first two implementation of the I/O mechanism there is only two parameters to vary, N (size of the input file) and k the number of files. The two next implementation have one more parameter, B the size of the Buffer.

As the JMH benchmark library allow us the test will be done with various value for each parameters.

For the first run, k is taking is value in the following list {”1”,”2”,”3”,”4”,”5”, ”10”,”15”,”20”,”25”,”30”}, and for each value of k a test is done with an input file of 100.000 in 32-bit integers and a buffer of size 1000 in 32-bits integers. In order to test the second implementation the size of the buffer is set to 8192 for the buffer implementation.

```
Result "com.ulb.psk.MyBenchmark.writeIntFile":
  7,132 ±(99.9%) 0,326 ms/op [Average]
  (min, avg, max) = (6,867, 7,132, 7,402), stdev = 0,216
  CI (99.9%): [6,805, 7,458] (assumes normal distribution)


# Run complete. Total time: 00:13:58
```

Benchmark	(bufferSize)	(dataVolume)	(filesToWrite)	Mode	Cnt	Score		Error	Units
MyBenchmark.bufferedWriteIntFile	1000	1000	1	avgt	10	0,034	±	0,001	ms/op
MyBenchmark.bufferedWriteIntFile	1000	1000	2	avgt	10	0,040	±	0,002	ms/op
MyBenchmark.bufferedWriteIntFile	1000	1000	3	avgt	10	0,042	±	0,005	ms/op
MyBenchmark.bufferedWriteIntFile	1000	1000	4	avgt	10	0,045	±	0,006	ms/op
MyBenchmark.bufferedWriteIntFile	1000	1000	5	avgt	10	0,045	±	0,005	ms/op
MyBenchmark.bufferedWriteIntFile	1000	1000	10	avgt	10	0,045	±	0,009	ms/op
MyBenchmark.bufferedWriteIntFile	1000	1000	15	avgt	10	0,046	±	0,006	ms/op
MyBenchmark.bufferedWriteIntFile	1000	1000	20	avgt	10	0,047	±	0,006	ms/op
MyBenchmark.bufferedWriteIntFile	1000	1000	25	avgt	10	0,037	±	0,008	ms/op
MyBenchmark.bufferedWriteIntFile	1000	1000	30	avgt	10	0,032	±	0,004	ms/op
MyBenchmark.bufferedWriteIntFile	8192	1000	1	avgt	10	0,039	±	0,002	ms/op
MyBenchmark.bufferedWriteIntFile	8192	1000	2	avgt	10	0,035	±	0,001	ms/op
MyBenchmark.bufferedWriteIntFile	8192	1000	3	avgt	10	0,033	±	0,001	ms/op
MyBenchmark.bufferedWriteIntFile	8192	1000	4	avgt	10	0,031	±	0,002	ms/op
MyBenchmark.bufferedWriteIntFile	8192	1000	5	avgt	10	0,031	±	0,005	ms/op
MyBenchmark.bufferedWriteIntFile	8192	1000	10	avgt	10	0,032	±	0,004	ms/op
MyBenchmark.bufferedWriteIntFile	8192	1000	15	avgt	10	0,031	±	0,002	ms/op
MyBenchmark.bufferedWriteIntFile	8192	1000	20	avgt	10	0,031	±	0,003	ms/op
MyBenchmark.bufferedWriteIntFile	8192	1000	25	avgt	10	0,031	±	0,002	ms/op
MyBenchmark.bufferedWriteIntFile	8192	1000	30	avgt	10	0,032	±	0,001	ms/op
MyBenchmark.mmWriteIntFile	1000	1000	1	avgt	10	0,018	±	0,001	ms/op
MyBenchmark.mmWriteIntFile	1000	1000	2	avgt	10	0,025	±	0,001	ms/op
MyBenchmark.mmWriteIntFile	1000	1000	3	avgt	10	0,031	±	0,002	ms/op
MyBenchmark.mmWriteIntFile	1000	1000	4	avgt	10	0,037	±	0,002	ms/op
MyBenchmark.mmWriteIntFile	1000	1000	5	avgt	10	0,042	±	0,004	ms/op
MyBenchmark.mmWriteIntFile	1000	1000	10	avgt	10	0,067	±	0,007	ms/op
MyBenchmark.mmWriteIntFile	1000	1000	15	avgt	10	0,094	±	0,021	ms/op
MyBenchmark.mmWriteIntFile	1000	1000	20	avgt	10	0,121	±	0,031	ms/op
MyBenchmark.mmWriteIntFile	1000	1000	25	avgt	10	0,149	±	0,044	ms/op
MyBenchmark.mmWriteIntFile	1000	1000	30	avgt	10	0,153	±	0,017	ms/op
MyBenchmark.writeIntFile	N/A	1000	1	avgt	10	7,454	±	0,526	ms/op
MyBenchmark.writeIntFile	N/A	1000	2	avgt	10	7,577	±	0,605	ms/op
MyBenchmark.writeIntFile	N/A	1000	3	avgt	10	7,414	±	0,405	ms/op
MyBenchmark.writeIntFile	N/A	1000	4	avgt	10	7,230	±	0,316	ms/op
MyBenchmark.writeIntFile	N/A	1000	5	avgt	10	7,254	±	0,327	ms/op
MyBenchmark.writeIntFile	N/A	1000	10	avgt	10	7,080	±	0,402	ms/op
MyBenchmark.writeIntFile	N/A	1000	15	avgt	10	6,855	±	0,473	ms/op
MyBenchmark.writeIntFile	N/A	1000	20	avgt	10	6,398	±	0,440	ms/op
MyBenchmark.writeIntFile	N/A	1000	25	avgt	10	8,004	±	0,876	ms/op
MyBenchmark.writeIntFile	N/A	1000	30	avgt	10	7,132	±	0,326	ms/op

Figure 3: Test N = 1.000

The first run reveals that the first write mechanism (WriteInt) is from far away the one that take the longest time. The write mechanism with the memory mapping seems to be slowest than the two implementation with the buffer. More over the second I/O mechanism (using a buffer of size 8192) is faster than the third I/O mechanism (using a buffer of size 1000). The best combination here is to use memory mapping with 1 file to write and a buffer of 1000 32-bits integers.

For the second run the value of the input file (N) is set to 100.000 32-bits integers. In order to test the second implementation the size of the buffer is set to 8192 for the buffer implementation.

```
Result "com.ulb.psk.MyBenchmark.writeIntFile":
  576,078 ±(99.9%) 13,423 ms/op [Average]
  (min, avg, max) = (556,532, 576,078, 585,086), stdev = 8,878
  CI (99.9%): [562,655, 589,501] (assumes normal distribution)


# Run complete. Total time: 00:14:27
```

Benchmark	(bufferSize)	(dataVolume)	(filesToWrite)	Mode	Cnt	Score		Error	Units
MyBenchmark.bufferedWriteIntFile	8192	100000	1	avgt	10	3,415	±	0,195	ms/op
MyBenchmark.bufferedWriteIntFile	8192	100000	2	avgt	10	3,375	±	0,073	ms/op
MyBenchmark.bufferedWriteIntFile	8192	100000	3	avgt	10	3,162	±	0,061	ms/op
MyBenchmark.bufferedWriteIntFile	8192	100000	4	avgt	10	3,273	±	0,088	ms/op
MyBenchmark.bufferedWriteIntFile	8192	100000	5	avgt	10	2,790	±	0,113	ms/op
MyBenchmark.bufferedWriteIntFile	8192	100000	10	avgt	10	3,021	±	0,096	ms/op
MyBenchmark.bufferedWriteIntFile	8192	100000	15	avgt	10	2,413	±	0,081	ms/op
MyBenchmark.bufferedWriteIntFile	8192	100000	20	avgt	10	2,972	±	0,319	ms/op
MyBenchmark.bufferedWriteIntFile	8192	100000	25	avgt	10	2,684	±	0,545	ms/op
MyBenchmark.bufferedWriteIntFile	8192	100000	30	avgt	10	2,441	±	0,082	ms/op
MyBenchmark.bufferedWriteIntFile	10000	100000	1	avgt	10	3,331	±	0,215	ms/op
MyBenchmark.bufferedWriteIntFile	10000	100000	2	avgt	10	3,252	±	0,100	ms/op
MyBenchmark.bufferedWriteIntFile	10000	100000	3	avgt	10	3,431	±	0,112	ms/op
MyBenchmark.bufferedWriteIntFile	10000	100000	4	avgt	10	3,444	±	0,515	ms/op
MyBenchmark.bufferedWriteIntFile	10000	100000	5	avgt	10	2,793	±	0,098	ms/op
MyBenchmark.bufferedWriteIntFile	10000	100000	10	avgt	10	2,504	±	0,213	ms/op
MyBenchmark.bufferedWriteIntFile	10000	100000	15	avgt	10	2,311	±	0,065	ms/op
MyBenchmark.bufferedWriteIntFile	10000	100000	20	avgt	10	2,311	±	0,043	ms/op
MyBenchmark.bufferedWriteIntFile	10000	100000	25	avgt	10	2,292	±	0,029	ms/op
MyBenchmark.bufferedWriteIntFile	10000	100000	30	avgt	10	2,363	±	0,127	ms/op
MyBenchmark.mmWriteIntFile	10000	100000	1	avgt	10	1,463	±	0,204	ms/op
MyBenchmark.mmWriteIntFile	10000	100000	2	avgt	10	1,700	±	0,257	ms/op
MyBenchmark.mmWriteIntFile	10000	100000	3	avgt	10	1,575	±	0,100	ms/op
MyBenchmark.mmWriteIntFile	10000	100000	4	avgt	10	1,579	±	0,044	ms/op
MyBenchmark.mmWriteIntFile	10000	100000	5	avgt	10	1,514	±	0,091	ms/op
MyBenchmark.mmWriteIntFile	10000	100000	10	avgt	10	1,768	±	0,070	ms/op
MyBenchmark.mmWriteIntFile	10000	100000	15	avgt	10	1,931	±	0,149	ms/op
MyBenchmark.mmWriteIntFile	10000	100000	20	avgt	10	2,415	±	0,063	ms/op
MyBenchmark.mmWriteIntFile	10000	100000	25	avgt	10	2,521	±	0,042	ms/op
MyBenchmark.mmWriteIntFile	10000	100000	30	avgt	10	2,637	±	0,051	ms/op
MyBenchmark.writeIntFile	N/A	100000	1	avgt	10	666,993	±	30,079	ms/op
MyBenchmark.writeIntFile	N/A	100000	2	avgt	10	658,243	±	43,279	ms/op
MyBenchmark.writeIntFile	N/A	100000	3	avgt	10	534,820	±	15,031	ms/op
MyBenchmark.writeIntFile	N/A	100000	4	avgt	10	546,347	±	19,081	ms/op
MyBenchmark.writeIntFile	N/A	100000	5	avgt	10	549,830	±	18,702	ms/op
MyBenchmark.writeIntFile	N/A	100000	10	avgt	10	531,809	±	14,155	ms/op
MyBenchmark.writeIntFile	N/A	100000	15	avgt	10	551,015	±	30,062	ms/op
MyBenchmark.writeIntFile	N/A	100000	20	avgt	10	547,186	±	30,756	ms/op
MyBenchmark.writeIntFile	N/A	100000	25	avgt	10	569,422	±	21,278	ms/op
MyBenchmark.writeIntFile	N/A	100000	30	avgt	10	576,078	±	13,423	ms/op

Figure 4: Test N = 100.000

The results of the second run shows again that the writeInt function is the slowest. The best combination here is to use memory mapping with 1 file to write and a buffer of 1000 32-bits integers (same as the first run).

For the third run the value of the input file (N) is set to 1.000.000 32-bits integers. In order to test the second implementation the size of the buffer is set to 8192 for the buffer implementation.

```
Result "com.ulb.psk.MyBenchmark.writeIntFile":
  5243,696 ±(99.9%) 36,783 ms/op [Average]
  (min, avg, max) = (5221,530, 5243,696, 5288,271), stdev = 24,330
  CI (99.9%): [5206,912, 5280,479] (assumes normal distribution)


# Run complete. Total time: 00:28:55

Benchmark                         (bufferSize)  (dataVolume)  (filesToWrite)  Mode  Cnt     Score     Error  Units
MyBenchmark.bufferedWriteIntFile          1000       1000000               1  avgt   10    31,488 ±   2,583  ms/op
MyBenchmark.bufferedWriteIntFile          1000       1000000               2  avgt   10    31,625 ±   3,924  ms/op
MyBenchmark.bufferedWriteIntFile          1000       1000000               3  avgt   10    29,895 ±   2,061  ms/op
MyBenchmark.bufferedWriteIntFile          1000       1000000               4  avgt   10    30,093 ±   0,919  ms/op
MyBenchmark.bufferedWriteIntFile          1000       1000000               5  avgt   10    28,734 ±   1,137  ms/op
MyBenchmark.bufferedWriteIntFile          1000       1000000              10  avgt   10    28,576 ±   2,353  ms/op
MyBenchmark.bufferedWriteIntFile          1000       1000000              15  avgt   10    27,877 ±   1,394  ms/op
MyBenchmark.bufferedWriteIntFile          1000       1000000              20  avgt   10    37,313 ±   4,127  ms/op
MyBenchmark.bufferedWriteIntFile          1000       1000000              25  avgt   10    40,555 ±   5,169  ms/op
MyBenchmark.bufferedWriteIntFile          1000       1000000              30  avgt   10    38,937 ±   1,978  ms/op
MyBenchmark.bufferedWriteIntFile          8192       1000000               1  avgt   10    27,970 ±   1,387  ms/op
MyBenchmark.bufferedWriteIntFile          8192       1000000               2  avgt   10    28,949 ±   1,990  ms/op
MyBenchmark.bufferedWriteIntFile          8192       1000000               3  avgt   10    27,528 ±   0,733  ms/op
MyBenchmark.bufferedWriteIntFile          8192       1000000               4  avgt   10    30,770 ±   0,683  ms/op
MyBenchmark.bufferedWriteIntFile          8192       1000000               5  avgt   10    28,457 ±   4,273  ms/op
MyBenchmark.bufferedWriteIntFile          8192       1000000              10  avgt   10    28,249 ±   1,199  ms/op
MyBenchmark.bufferedWriteIntFile          8192       1000000              15  avgt   10    26,513 ±   1,031  ms/op
MyBenchmark.bufferedWriteIntFile          8192       1000000              20  avgt   10    26,397 ±   1,132  ms/op
MyBenchmark.bufferedWriteIntFile          8192       1000000              25  avgt   10    27,590 ±   2,264  ms/op
MyBenchmark.bufferedWriteIntFile          8192       1000000              30  avgt   10    28,284 ±   1,222  ms/op
MyBenchmark.mmWriteIntFile                1000       1000000               1  avgt   10    36,471 ±   6,052  ms/op
MyBenchmark.mmWriteIntFile                1000       1000000               2  avgt   10    58,679 ±   8,583  ms/op
MyBenchmark.mmWriteIntFile                1000       1000000               3  avgt   10    65,722 ±   4,653  ms/op
MyBenchmark.mmWriteIntFile                1000       1000000               4  avgt   10    71,769 ±   8,123  ms/op
MyBenchmark.mmWriteIntFile                1000       1000000               5  avgt   10    56,375 ±   6,772  ms/op
MyBenchmark.mmWriteIntFile                1000       1000000              10  avgt   10    57,165 ±   3,812  ms/op
MyBenchmark.mmWriteIntFile                1000       1000000              15  avgt   10    58,526 ±   4,676  ms/op
MyBenchmark.mmWriteIntFile                1000       1000000              20  avgt   10    61,822 ±   4,098  ms/op
MyBenchmark.mmWriteIntFile                1000       1000000              25  avgt   10    59,116 ±   5,044  ms/op
MyBenchmark.mmWriteIntFile                1000       1000000              30  avgt   10    61,558 ±   8,208  ms/op
MyBenchmark.writeIntFile                   N/A       1000000               1  avgt   10  5676,961 ± 917,509  ms/op
MyBenchmark.writeIntFile                   N/A       1000000               2  avgt   10  5195,480 ±  47,024  ms/op
MyBenchmark.writeIntFile                   N/A       1000000               3  avgt   10  5228,343 ± 108,065  ms/op
MyBenchmark.writeIntFile                   N/A       1000000               4  avgt   10  5198,916 ±  64,334  ms/op
MyBenchmark.writeIntFile                   N/A       1000000               5  avgt   10  5407,696 ±  59,048  ms/op
MyBenchmark.writeIntFile                   N/A       1000000              10  avgt   10  5214,788 ±  62,867  ms/op
MyBenchmark.writeIntFile                   N/A       1000000              15  avgt   10  5211,016 ±  79,283  ms/op
MyBenchmark.writeIntFile                   N/A       1000000              20  avgt   10  5252,365 ± 103,303  ms/op
MyBenchmark.writeIntFile                   N/A       1000000              25  avgt   10  5310,124 ± 142,772  ms/op
MyBenchmark.writeIntFile                   N/A       1000000              30  avgt   10  5243,696 ±  36,783  ms/op
```

Figure 5: Test N = 1.000.000

The third run shows again that the first I/O mechanism is very slow comparing to the other mechanisms. It tends to put away this implementation for the following of the project. This third brings something new, the memory mapping is not anymore the fastest. This time the best combination to use is to use buffered write with a buffer size of 8192 (second mechanism) and 20 files to write.

For the fourth run the value of the input file (N) is set to 5.000.000 32 bits integers. In order to test the second implementation the size of the buffer is set to 8192 for the buffer implementation.

```
Result "com.ulb.psk.MyBenchmark.writeIntFile":
  26077,913 ±(99.9%) 266,902 ms/op [Average]
  (min, avg, max) = (25833,601, 26077,913, 26374,008), stdev = 176,539
  CI (99.9%): [25811,011, 26344,816] (assumes normal distribution)


# Run complete. Total time: 01:45:18

Benchmark                          (bufferSize)  (dataVolume)  (filesToWrite)  Mode  Cnt      Score      Error  Units
MyBenchmark.bufferedWriteIntFile           1000       5000000               1  avgt   10    158,945 ±   10,611  ms/op
MyBenchmark.bufferedWriteIntFile           1000       5000000               2  avgt   10    186,286 ±   17,917  ms/op
MyBenchmark.bufferedWriteIntFile           1000       5000000               3  avgt   10    136,660 ±    6,781  ms/op
MyBenchmark.bufferedWriteIntFile           1000       5000000               4  avgt   10    137,560 ±    5,927  ms/op
MyBenchmark.bufferedWriteIntFile           1000       5000000               5  avgt   10    136,302 ±    3,211  ms/op
MyBenchmark.bufferedWriteIntFile           1000       5000000              10  avgt   10    137,602 ±    7,472  ms/op
MyBenchmark.bufferedWriteIntFile           1000       5000000              15  avgt   10    152,266 ±   11,195  ms/op
MyBenchmark.bufferedWriteIntFile           1000       5000000              20  avgt   10    170,588 ±   31,015  ms/op
MyBenchmark.bufferedWriteIntFile           1000       5000000              25  avgt   10    163,356 ±    6,142  ms/op
MyBenchmark.bufferedWriteIntFile           1000       5000000              30  avgt   10    157,701 ±   18,207  ms/op
MyBenchmark.bufferedWriteIntFile           8192       5000000               1  avgt   10    150,512 ±   11,115  ms/op
MyBenchmark.bufferedWriteIntFile           8192       5000000               2  avgt   10    176,720 ±   11,615  ms/op
MyBenchmark.bufferedWriteIntFile           8192       5000000               3  avgt   10    133,777 ±   13,793  ms/op
MyBenchmark.bufferedWriteIntFile           8192       5000000               4  avgt   10    135,368 ±   14,459  ms/op
MyBenchmark.bufferedWriteIntFile           8192       5000000               5  avgt   10    132,724 ±   22,567  ms/op
MyBenchmark.bufferedWriteIntFile           8192       5000000              10  avgt   10    141,408 ±    4,997  ms/op
MyBenchmark.bufferedWriteIntFile           8192       5000000              15  avgt   10    131,879 ±   21,411  ms/op
MyBenchmark.bufferedWriteIntFile           8192       5000000              20  avgt   10    133,963 ±    5,649  ms/op
MyBenchmark.bufferedWriteIntFile           8192       5000000              25  avgt   10    133,665 ±    3,733  ms/op
MyBenchmark.bufferedWriteIntFile           8192       5000000              30  avgt   10    134,609 ±    5,500  ms/op
MyBenchmark.mmWriteIntFile                 1000       5000000               1  avgt   10    218,697 ±   30,948  ms/op
MyBenchmark.mmWriteIntFile                 1000       5000000               2  avgt   10    309,910 ±   29,738  ms/op
MyBenchmark.mmWriteIntFile                 1000       5000000               3  avgt   10    259,758 ±    9,885  ms/op
MyBenchmark.mmWriteIntFile                 1000       5000000               4  avgt   10    254,948 ±    8,377  ms/op
MyBenchmark.mmWriteIntFile                 1000       5000000               5  avgt   10    256,975 ±   11,229  ms/op
MyBenchmark.mmWriteIntFile                 1000       5000000              10  avgt   10    268,300 ±   10,107  ms/op
MyBenchmark.mmWriteIntFile                 1000       5000000              15  avgt   10    287,143 ±   73,404  ms/op
MyBenchmark.mmWriteIntFile                 1000       5000000              20  avgt   10    284,672 ±   59,699  ms/op
MyBenchmark.mmWriteIntFile                 1000       5000000              25  avgt   10    280,410 ±   20,309  ms/op
MyBenchmark.mmWriteIntFile                 1000       5000000              30  avgt   10    283,780 ±   17,110  ms/op
MyBenchmark.writeIntFile                    N/A       5000000               1  avgt   10  29136,875 ± 3002,747  ms/op
MyBenchmark.writeIntFile                    N/A       5000000               2  avgt   10  29273,168 ±  642,624  ms/op
MyBenchmark.writeIntFile                    N/A       5000000               3  avgt   10  28798,192 ±  856,384  ms/op
MyBenchmark.writeIntFile                    N/A       5000000               4  avgt   10  29360,537 ±  953,454  ms/op
MyBenchmark.writeIntFile                    N/A       5000000               5  avgt   10  26579,117 ±  978,690  ms/op
MyBenchmark.writeIntFile                    N/A       5000000              10  avgt   10  25999,583 ±  293,830  ms/op
MyBenchmark.writeIntFile                    N/A       5000000              15  avgt   10  26013,073 ±  409,329  ms/op
MyBenchmark.writeIntFile                    N/A       5000000              20  avgt   10  25926,156 ±  161,744  ms/op
MyBenchmark.writeIntFile                    N/A       5000000              25  avgt   10  26040,508 ±  254,991  ms/op
MyBenchmark.writeIntFile                    N/A       5000000              30  avgt   10  26077,913 ±  266,902  ms/op
```

Figure 6: Test N = 5.000.000

The fourth run as the previous run is showing that the write int file is obsolete comparing to the other implementation. According to this run the best combination is to use a buffered write int with a buffer size of 8192 (second mechanism) and 15 files to write.

For the fifth run the value of the input file (N) is set to 7.000.000 32 bits integers. In order to test the second implementation the size of the buffer is set to 8192 for the buffer implementation.

```
Result "com.ulb.psk.MyBenchmark.writeIntFile":
  37479,286 ±(99.9%) 272,799 ms/op [Average]
  (min, avg, max) = (37190,127, 37479,286, 37739,204), stdev = 180,439
  CI (99.9%): [37206,487, 37752,084] (assumes normal distribution)


# Run complete. Total time: 02:23:25

Benchmark                          (bufferSize)  (dataVolume)  (filesToWrite)  Mode  Cnt      Score     Error  Units
MyBenchmark.bufferedWriteIntFile           1000       7000000               1  avgt   10    246,602 ±  25,101  ms/op
MyBenchmark.bufferedWriteIntFile           1000       7000000               2  avgt   10    343,961 ±  54,897  ms/op
MyBenchmark.bufferedWriteIntFile           1000       7000000               3  avgt   10    280,213 ±  35,998  ms/op
MyBenchmark.bufferedWriteIntFile           1000       7000000               4  avgt   10    226,426 ±  43,366  ms/op
MyBenchmark.bufferedWriteIntFile           1000       7000000               5  avgt   10    208,724 ±  12,784  ms/op
MyBenchmark.bufferedWriteIntFile           1000       7000000              10  avgt   10    238,669 ±  79,697  ms/op
MyBenchmark.bufferedWriteIntFile           1000       7000000              15  avgt   10    241,603 ±  32,532  ms/op
MyBenchmark.bufferedWriteIntFile           1000       7000000              20  avgt   10    243,502 ±  16,605  ms/op
MyBenchmark.bufferedWriteIntFile           1000       7000000              25  avgt   10    243,055 ±  58,140  ms/op
MyBenchmark.bufferedWriteIntFile           1000       7000000              30  avgt   10    222,433 ±   8,939  ms/op
MyBenchmark.bufferedWriteIntFile           8192       7000000               1  avgt   10    213,156 ±  34,732  ms/op
MyBenchmark.bufferedWriteIntFile           8192       7000000               2  avgt   10    215,379 ±  11,878  ms/op
MyBenchmark.bufferedWriteIntFile           8192       7000000               3  avgt   10    225,056 ±   8,466  ms/op
MyBenchmark.bufferedWriteIntFile           8192       7000000               4  avgt   10    179,298 ±  15,211  ms/op
MyBenchmark.bufferedWriteIntFile           8192       7000000               5  avgt   10    174,631 ±  16,059  ms/op
MyBenchmark.bufferedWriteIntFile           8192       7000000              10  avgt   10    190,051 ±  17,216  ms/op
MyBenchmark.bufferedWriteIntFile           8192       7000000              15  avgt   10    195,374 ±  94,714  ms/op
MyBenchmark.bufferedWriteIntFile           8192       7000000              20  avgt   10    174,947 ±  12,949  ms/op
MyBenchmark.bufferedWriteIntFile           8192       7000000              25  avgt   10    174,130 ±   5,611  ms/op
MyBenchmark.bufferedWriteIntFile           8192       7000000              30  avgt   10    175,708 ±   5,011  ms/op
MyBenchmark.mmWriteIntFile                 1000       7000000               1  avgt   10    322,388 ±  71,113  ms/op
MyBenchmark.mmWriteIntFile                 1000       7000000               2  avgt   10    477,561 ±  52,187  ms/op
MyBenchmark.mmWriteIntFile                 1000       7000000               3  avgt   10    403,982 ±  29,761  ms/op
MyBenchmark.mmWriteIntFile                 1000       7000000               4  avgt   10    360,681 ±  15,064  ms/op
MyBenchmark.mmWriteIntFile                 1000       7000000               5  avgt   10    367,399 ±  18,467  ms/op
MyBenchmark.mmWriteIntFile                 1000       7000000              10  avgt   10    383,425 ±  67,196  ms/op
MyBenchmark.mmWriteIntFile                 1000       7000000              15  avgt   10    375,352 ±  14,020  ms/op
MyBenchmark.mmWriteIntFile                 1000       7000000              20  avgt   10    387,260 ±  57,010  ms/op
MyBenchmark.mmWriteIntFile                 1000       7000000              25  avgt   10    438,789 ± 316,200  ms/op
MyBenchmark.mmWriteIntFile                 1000       7000000              30  avgt   10    400,781 ±  81,421  ms/op
MyBenchmark.writeIntFile                    N/A       7000000               1  avgt   10  38779,091 ± 407,859  ms/op
MyBenchmark.writeIntFile                    N/A       7000000               2  avgt   10  39088,631 ± 825,534  ms/op
MyBenchmark.writeIntFile                    N/A       7000000               3  avgt   10  38418,280 ± 940,858  ms/op
MyBenchmark.writeIntFile                    N/A       7000000               4  avgt   10  38389,754 ± 335,040  ms/op
MyBenchmark.writeIntFile                    N/A       7000000               5  avgt   10  38349,884 ± 237,246  ms/op
MyBenchmark.writeIntFile                    N/A       7000000              10  avgt   10  38312,227 ± 322,453  ms/op
MyBenchmark.writeIntFile                    N/A       7000000              15  avgt   10  38637,758 ± 368,924  ms/op
MyBenchmark.writeIntFile                    N/A       7000000              20  avgt   10  38333,828 ± 587,904  ms/op
MyBenchmark.writeIntFile                    N/A       7000000              25  avgt   10  37837,173 ± 275,679  ms/op
MyBenchmark.writeIntFile                    N/A       7000000              30  avgt   10  37479,286 ± 272,799  ms/op
```

Figure 7: Test N = 7.000.000

The best combination for the fifth run is to use a buffered write int with a buffer size of 8192 (second mechanism) and 25 files to write.

The value of N is becoming too big to keep on doing test on the machine, because of the free memory. So it will stop the test on varying the parameter N.

Now that different value of N and k have been tried there is just the size of the buffer to play with. In order to test the impact of the buffer size, B is set to multiple value; 8192, 10.000, 50.000, 100.000. More over in order to test this with a big size in input, N is set to 10.000.000 32-bits integers. This lead to the following results :

```
Result "com.ulb.psk.MyBenchmark.writeIntFile":
  53167,760 ±(99.9%) 1679,723 ms/op [Average]
  (min, avg, max) = (51077,540, 53167,760, 54485,088), stdev = 1111,033
  CI (99.9%): [51488,037, 54847,483] (assumes normal distribution)


# Run complete. Total time: 03:32:55

Benchmark                         (bufferSize)  (dataVolume)  (filesToWrite)  Mode  Cnt     Score       Error  Units
MyBenchmark.bufferedWriteIntFile          8192      10000000               1  avgt   10   295,083 ±    32,498  ms/op
MyBenchmark.bufferedWriteIntFile          8192      10000000               2  avgt   10   328,281 ±    20,370  ms/op
MyBenchmark.bufferedWriteIntFile          8192      10000000               3  avgt   10   299,121 ±    17,597  ms/op
MyBenchmark.bufferedWriteIntFile          8192      10000000               4  avgt   10   329,472 ±    21,563  ms/op
MyBenchmark.bufferedWriteIntFile          8192      10000000               5  avgt   10   239,296 ±     8,482  ms/op
MyBenchmark.bufferedWriteIntFile          8192      10000000              10  avgt   10   250,893 ±     8,840  ms/op
MyBenchmark.bufferedWriteIntFile          8192      10000000              15  avgt   10   234,207 ±    10,635  ms/op
MyBenchmark.bufferedWriteIntFile          8192      10000000              20  avgt   10   267,195 ±    11,562  ms/op
MyBenchmark.bufferedWriteIntFile          8192      10000000              25  avgt   10   288,307 ±    35,548  ms/op
MyBenchmark.bufferedWriteIntFile          8192      10000000              30  avgt   10   282,037 ±    49,212  ms/op
MyBenchmark.bufferedWriteIntFile         10000      10000000               1  avgt   10   404,009 ±   109,481  ms/op
MyBenchmark.bufferedWriteIntFile         10000      10000000               2  avgt   10   409,915 ±    82,194  ms/op
MyBenchmark.bufferedWriteIntFile         10000      10000000               3  avgt   10   363,684 ±    46,481  ms/op
MyBenchmark.bufferedWriteIntFile         10000      10000000               4  avgt   10   336,119 ±    45,109  ms/op
MyBenchmark.bufferedWriteIntFile         10000      10000000               5  avgt   10   226,898 ±     5,795  ms/op
MyBenchmark.bufferedWriteIntFile         10000      10000000              10  avgt   10   247,600 ±     8,024  ms/op
MyBenchmark.bufferedWriteIntFile         10000      10000000              15  avgt   10   228,914 ±    10,438  ms/op
MyBenchmark.bufferedWriteIntFile         10000      10000000              20  avgt   10   249,169 ±    20,951  ms/op
MyBenchmark.bufferedWriteIntFile         10000      10000000              25  avgt   10   251,125 ±    15,037  ms/op
MyBenchmark.bufferedWriteIntFile         10000      10000000              30  avgt   10   249,565 ±    13,146  ms/op
MyBenchmark.bufferedWriteIntFile         50000      10000000               1  avgt   10   460,588 ±    72,573  ms/op
MyBenchmark.bufferedWriteIntFile         50000      10000000               2  avgt   10   448,560 ±    53,377  ms/op
MyBenchmark.bufferedWriteIntFile         50000      10000000               3  avgt   10   396,207 ±    66,057  ms/op
MyBenchmark.bufferedWriteIntFile         50000      10000000               4  avgt   10   394,051 ±    37,873  ms/op
MyBenchmark.bufferedWriteIntFile         50000      10000000               5  avgt   10   233,242 ±     6,785  ms/op
MyBenchmark.bufferedWriteIntFile         50000      10000000              10  avgt   10   232,001 ±     8,559  ms/op
MyBenchmark.bufferedWriteIntFile         50000      10000000              15  avgt   10   251,956 ±    15,347  ms/op
MyBenchmark.bufferedWriteIntFile         50000      10000000              20  avgt   10   263,364 ±    22,866  ms/op
MyBenchmark.bufferedWriteIntFile         50000      10000000              25  avgt   10   265,834 ±    21,504  ms/op
MyBenchmark.bufferedWriteIntFile         50000      10000000              30  avgt   10   264,477 ±    23,185  ms/op
MyBenchmark.bufferedWriteIntFile        100000      10000000               1  avgt   10   673,708 ±    37,122  ms/op
MyBenchmark.bufferedWriteIntFile        100000      10000000               2  avgt   10   576,233 ±    60,068  ms/op
MyBenchmark.bufferedWriteIntFile        100000      10000000               3  avgt   10   511,549 ±    30,451  ms/op
MyBenchmark.bufferedWriteIntFile        100000      10000000               4  avgt   10   422,033 ±    44,640  ms/op
MyBenchmark.bufferedWriteIntFile        100000      10000000               5  avgt   10   241,867 ±    15,956  ms/op
MyBenchmark.bufferedWriteIntFile        100000      10000000              10  avgt   10   257,084 ±    76,225  ms/op
MyBenchmark.bufferedWriteIntFile        100000      10000000              15  avgt   10   251,288 ±    13,291  ms/op
MyBenchmark.bufferedWriteIntFile        100000      10000000              20  avgt   10   265,178 ±    25,387  ms/op
MyBenchmark.bufferedWriteIntFile        100000      10000000              25  avgt   10   258,564 ±    38,020  ms/op
MyBenchmark.bufferedWriteIntFile        100000      10000000              30  avgt   10   253,875 ±    20,630  ms/op
MyBenchmark.mmWriteIntFile               10000      10000000               1  avgt   10   141,667 ±    18,230  ms/op
MyBenchmark.mmWriteIntFile               10000      10000000               2  avgt   10   148,099 ±    16,673  ms/op
MyBenchmark.mmWriteIntFile               10000      10000000               3  avgt   10   153,998 ±    15,331  ms/op
MyBenchmark.mmWriteIntFile               10000      10000000               4  avgt   10   171,418 ±    33,659  ms/op
```

MyBenchmark.mmWriteIntFile	10000	10000000	5	avgt	10	148,646	±	4,586	ms/op
MyBenchmark.mmWriteIntFile	10000	10000000	10	avgt	10	198,393	±	24,786	ms/op
MyBenchmark.mmWriteIntFile	10000	10000000	15	avgt	10	197,663	±	15,479	ms/op
MyBenchmark.mmWriteIntFile	10000	10000000	20	avgt	10	193,992	±	9,119	ms/op
MyBenchmark.mmWriteIntFile	10000	10000000	25	avgt	10	201,246	±	15,086	ms/op
MyBenchmark.mmWriteIntFile	10000	10000000	30	avgt	10	201,055	±	6,947	ms/op
MyBenchmark.mmWriteIntFile	50000	10000000	1	avgt	10	126,795	±	33,164	ms/op
MyBenchmark.mmWriteIntFile	50000	10000000	2	avgt	10	124,025	±	14,798	ms/op
MyBenchmark.mmWriteIntFile	50000	10000000	3	avgt	10	122,004	±	4,075	ms/op
MyBenchmark.mmWriteIntFile	50000	10000000	4	avgt	10	132,651	±	17,042	ms/op
MyBenchmark.mmWriteIntFile	50000	10000000	5	avgt	10	128,199	±	4,210	ms/op
MyBenchmark.mmWriteIntFile	50000	10000000	10	avgt	10	189,370	±	5,171	ms/op
MyBenchmark.mmWriteIntFile	50000	10000000	15	avgt	10	213,757	±	7,496	ms/op
MyBenchmark.mmWriteIntFile	50000	10000000	20	avgt	10	211,339	±	6,367	ms/op
MyBenchmark.mmWriteIntFile	50000	10000000	25	avgt	10	212,452	±	7,987	ms/op
MyBenchmark.mmWriteIntFile	50000	10000000	30	avgt	10	212,879	±	12,983	ms/op
MyBenchmark.mmWriteIntFile	100000	10000000	1	avgt	10	114,845	±	12,983	ms/op
MyBenchmark.mmWriteIntFile	100000	10000000	2	avgt	10	122,372	±	9,270	ms/op
MyBenchmark.mmWriteIntFile	100000	10000000	3	avgt	10	121,497	±	3,754	ms/op
MyBenchmark.mmWriteIntFile	100000	10000000	4	avgt	10	120,861	±	5,530	ms/op
MyBenchmark.mmWriteIntFile	100000	10000000	5	avgt	10	124,593	±	4,297	ms/op
MyBenchmark.mmWriteIntFile	100000	10000000	10	avgt	10	197,331	±	7,427	ms/op
MyBenchmark.mmWriteIntFile	100000	10000000	15	avgt	10	223,016	±	4,756	ms/op
MyBenchmark.mmWriteIntFile	100000	10000000	20	avgt	10	224,349	±	5,632	ms/op
MyBenchmark.mmWriteIntFile	100000	10000000	25	avgt	10	230,758	±	8,388	ms/op
MyBenchmark.mmWriteIntFile	100000	10000000	30	avgt	10	234,787	±	17,118	ms/op
MyBenchmark.writeIntFile	N/A	10000000	1	avgt	10	54629,943	±	443,312	ms/op
MyBenchmark.writeIntFile	N/A	10000000	2	avgt	10	54446,142	±	267,360	ms/op
MyBenchmark.writeIntFile	N/A	10000000	3	avgt	10	54574,867	±	530,059	ms/op
MyBenchmark.writeIntFile	N/A	10000000	4	avgt	10	54809,896	±	842,873	ms/op
MyBenchmark.writeIntFile	N/A	10000000	5	avgt	10	55622,501	±	1685,965	ms/op
MyBenchmark.writeIntFile	N/A	10000000	10	avgt	10	58722,711	±	5686,321	ms/op
MyBenchmark.writeIntFile	N/A	10000000	15	avgt	10	55434,278	±	645,896	ms/op
MyBenchmark.writeIntFile	N/A	10000000	20	avgt	10	53263,151	±	2940,028	ms/op
MyBenchmark.writeIntFile	N/A	10000000	25	avgt	10	51678,350	±	1651,657	ms/op
MyBenchmark.writeIntFile	N/A	10000000	30	avgt	10	53167,760	±	1679,723	ms/op

Figure 8: 1st Test on buffer size

According to the previous run the buffered write int should be the best function to use here. But according to the result the best combination is to use the memory mapping with a buffer of 100.000 32-bits integers and 1 file to write.

In order to confirm the previous result, the data volume is set to 20.000.000 32-bits integers. More over the size of the buffer is also changed, now it is going to 1.000.000 and 2.000.000 32-bits integers.

```
# Run complete. Total time: 01:00:20

Benchmark                              (bufferSize)  (dataVolume)  (filesToWrite)  Mode  Cnt     Score     Error  Units
MyBenchmark.bufferedWriteIntFile               8192      20000000               1  avgt   10   511,067 ±  54,905  ms/op
MyBenchmark.bufferedWriteIntFile               8192      20000000               2  avgt   10   570,568 ±  51,070  ms/op
MyBenchmark.bufferedWriteIntFile               8192      20000000               3  avgt   10   628,796 ±  70,888  ms/op
MyBenchmark.bufferedWriteIntFile               8192      20000000               4  avgt   10   617,469 ±  53,087  ms/op
MyBenchmark.bufferedWriteIntFile               8192      20000000               5  avgt   10   646,213 ± 141,849  ms/op
MyBenchmark.bufferedWriteIntFile               8192      20000000              10  avgt   10   528,088 ±  24,170  ms/op
MyBenchmark.bufferedWriteIntFile               8192      20000000              15  avgt   10   476,849 ±  36,255  ms/op
MyBenchmark.bufferedWriteIntFile               8192      20000000              20  avgt   10   493,737 ±  19,791  ms/op
MyBenchmark.bufferedWriteIntFile               8192      20000000              25  avgt   10   530,713 ±  56,517  ms/op
MyBenchmark.bufferedWriteIntFile               8192      20000000              30  avgt   10   538,909 ±  89,318  ms/op
MyBenchmark.bufferedWriteIntFile              10000      20000000               1  avgt   10  1276,630 ± 609,513  ms/op
MyBenchmark.bufferedWriteIntFile              10000      20000000               2  avgt   10  1091,816 ± 411,150  ms/op
MyBenchmark.bufferedWriteIntFile              10000      20000000               3  avgt   10  1194,743 ± 462,344  ms/op
MyBenchmark.bufferedWriteIntFile              10000      20000000               4  avgt   10  1026,204 ± 409,809  ms/op
MyBenchmark.bufferedWriteIntFile              10000      20000000               5  avgt   10  1192,772 ± 263,295  ms/op
MyBenchmark.bufferedWriteIntFile              10000      20000000              10  avgt   10   548,092 ±  77,790  ms/op
MyBenchmark.bufferedWriteIntFile              10000      20000000              15  avgt   10   526,413 ± 157,298  ms/op
MyBenchmark.bufferedWriteIntFile              10000      20000000              20  avgt   10   541,929 ±  64,138  ms/op
MyBenchmark.bufferedWriteIntFile              10000      20000000              25  avgt   10   551,159 ±  44,987  ms/op
MyBenchmark.bufferedWriteIntFile              10000      20000000              30  avgt   10   607,091 ±  97,527  ms/op
MyBenchmark.bufferedWriteIntFile             100000      20000000               1  avgt   10  1471,175 ± 217,979  ms/op
MyBenchmark.bufferedWriteIntFile             100000      20000000               2  avgt   10  1309,037 ± 144,822  ms/op
MyBenchmark.bufferedWriteIntFile             100000      20000000               3  avgt   10  1274,185 ± 217,580  ms/op
MyBenchmark.bufferedWriteIntFile             100000      20000000               4  avgt   10  1152,132 ± 119,794  ms/op
MyBenchmark.bufferedWriteIntFile             100000      20000000               5  avgt   10   992,087 ± 169,667  ms/op
MyBenchmark.bufferedWriteIntFile             100000      20000000              10  avgt   10   514,825 ±  53,641  ms/op
MyBenchmark.bufferedWriteIntFile             100000      20000000              15  avgt   10   504,100 ±  38,010  ms/op
MyBenchmark.bufferedWriteIntFile             100000      20000000              20  avgt   10   529,546 ±  44,947  ms/op
MyBenchmark.bufferedWriteIntFile             100000      20000000              25  avgt   10   615,838 ± 299,228  ms/op
MyBenchmark.bufferedWriteIntFile             100000      20000000              30  avgt   10   559,687 ±  75,753  ms/op
MyBenchmark.bufferedWriteIntFile            1000000      20000000               1  avgt   10  1054,096 ±  87,645  ms/op
MyBenchmark.bufferedWriteIntFile            1000000      20000000               2  avgt   10   931,863 ± 100,612  ms/op
MyBenchmark.bufferedWriteIntFile            1000000      20000000               3  avgt   10   962,935 ± 114,415  ms/op
MyBenchmark.bufferedWriteIntFile            1000000      20000000               4  avgt   10   888,156 ±  57,201  ms/op
MyBenchmark.bufferedWriteIntFile            1000000      20000000               5  avgt   10   821,134 ±  42,465  ms/op
MyBenchmark.bufferedWriteIntFile            1000000      20000000              10  avgt   10   673,650 ±  50,082  ms/op
MyBenchmark.bufferedWriteIntFile            1000000      20000000              15  avgt   10   799,404 ±  65,330  ms/op
MyBenchmark.bufferedWriteIntFile            1000000      20000000              20  avgt   10   478,639 ±  52,549  ms/op
MyBenchmark.bufferedWriteIntFile            1000000      20000000              25  avgt   10   511,141 ± 121,880  ms/op
MyBenchmark.bufferedWriteIntFile            1000000      20000000              30  avgt   10   502,656 ±  53,691  ms/op
MyBenchmark.bufferedWriteIntFile           10000000      20000000               1  avgt   10   608,669 ±  16,118  ms/op
MyBenchmark.bufferedWriteIntFile           10000000      20000000               2  avgt   10   472,286 ±  19,499  ms/op
MyBenchmark.bufferedWriteIntFile           10000000      20000000               3  avgt   10   488,913 ±  36,866  ms/op
MyBenchmark.bufferedWriteIntFile           10000000      20000000               4  avgt   10   496,675 ±  15,782  ms/op
MyBenchmark.bufferedWriteIntFile           10000000      20000000               5  avgt   10   497,039 ±  36,294  ms/op
MyBenchmark.bufferedWriteIntFile           10000000      20000000              10  avgt   10   536,857 ±  88,897  ms/op
MyBenchmark.bufferedWriteIntFile           10000000      20000000              15  avgt   10   598,901 ± 326,812  ms/op
MyBenchmark.bufferedWriteIntFile           10000000      20000000              20  avgt   10   557,664 ± 117,861  ms/op
MyBenchmark.bufferedWriteIntFile           10000000      20000000              25  avgt   10   514,869 ± 142,716  ms/op
MyBenchmark.bufferedWriteIntFile           10000000      20000000              30  avgt   10   585,119 ± 173,860  ms/op
MyBenchmark.bufferedWriteIntFile           20000000      20000000               1  avgt   10   447,412 ±  32,828  ms/op
MyBenchmark.bufferedWriteIntFile           20000000      20000000               2  avgt   10   471,420 ±  51,054  ms/op
MyBenchmark.bufferedWriteIntFile           20000000      20000000               3  avgt   10   506,761 ±  83,750  ms/op
MyBenchmark.bufferedWriteIntFile           20000000      20000000               4  avgt   10   498,852 ±  86,463  ms/op
MyBenchmark.bufferedWriteIntFile           20000000      20000000               5  avgt   10   552,249 ±  88,789  ms/op
```

MyBenchmark.bufferedWriteIntFile	20000000	20000000	5	avgt	10	552,249	±	88,789	ms/op
MyBenchmark.bufferedWriteIntFile	20000000	20000000	10	avgt	10	545,704	±	144,435	ms/op
MyBenchmark.bufferedWriteIntFile	20000000	20000000	15	avgt	10	531,927	±	139,170	ms/op
MyBenchmark.mmWriteIntFile	10000	20000000	1	avgt	10	270,563	±	20,929	ms/op
MyBenchmark.mmWriteIntFile	10000	20000000	2	avgt	10	287,112	±	17,350	ms/op
MyBenchmark.mmWriteIntFile	10000	20000000	3	avgt	10	285,064	±	18,533	ms/op
MyBenchmark.mmWriteIntFile	10000	20000000	4	avgt	10	288,403	±	13,121	ms/op
MyBenchmark.mmWriteIntFile	10000	20000000	5	avgt	10	285,361	±	5,034	ms/op
MyBenchmark.mmWriteIntFile	10000	20000000	10	avgt	10	367,867	±	9,784	ms/op
MyBenchmark.mmWriteIntFile	10000	20000000	15	avgt	10	381,859	±	10,909	ms/op
MyBenchmark.mmWriteIntFile	10000	20000000	20	avgt	10	384,682	±	10,339	ms/op
MyBenchmark.mmWriteIntFile	10000	20000000	25	avgt	10	402,493	±	25,728	ms/op
MyBenchmark.mmWriteIntFile	10000	20000000	30	avgt	10	399,268	±	11,353	ms/op
MyBenchmark.mmWriteIntFile	100000	20000000	1	avgt	10	253,824	±	63,919	ms/op
MyBenchmark.mmWriteIntFile	100000	20000000	2	avgt	10	271,852	±	28,070	ms/op
MyBenchmark.mmWriteIntFile	100000	20000000	3	avgt	10	267,149	±	17,437	ms/op
MyBenchmark.mmWriteIntFile	100000	20000000	4	avgt	10	309,546	±	181,216	ms/op
MyBenchmark.mmWriteIntFile	100000	20000000	5	avgt	10	273,118	±	14,364	ms/op
MyBenchmark.mmWriteIntFile	100000	20000000	10	avgt	10	396,759	±	14,601	ms/op
MyBenchmark.mmWriteIntFile	100000	20000000	15	avgt	10	449,732	±	5,973	ms/op
MyBenchmark.mmWriteIntFile	100000	20000000	20	avgt	10	462,545	±	32,370	ms/op
MyBenchmark.mmWriteIntFile	100000	20000000	25	avgt	10	464,518	±	21,901	ms/op
MyBenchmark.mmWriteIntFile	100000	20000000	30	avgt	10	462,578	±	15,651	ms/op
MyBenchmark.mmWriteIntFile	1000000	20000000	1	avgt	10	589,498	±	60,801	ms/op
MyBenchmark.mmWriteIntFile	1000000	20000000	2	avgt	10	628,196	±	72,914	ms/op
MyBenchmark.mmWriteIntFile	1000000	20000000	3	avgt	10	574,631	±	79,179	ms/op
MyBenchmark.mmWriteIntFile	1000000	20000000	4	avgt	10	543,273	±	63,791	ms/op
MyBenchmark.mmWriteIntFile	1000000	20000000	5	avgt	10	510,464	±	52,149	ms/op
MyBenchmark.mmWriteIntFile	1000000	20000000	10	avgt	10	579,650	±	43,063	ms/op
MyBenchmark.mmWriteIntFile	1000000	20000000	15	avgt	10	729,632	±	28,133	ms/op
MyBenchmark.mmWriteIntFile	1000000	20000000	20	avgt	10	446,097	±	20,726	ms/op
MyBenchmark.mmWriteIntFile	1000000	20000000	25	avgt	10	455,381	±	18,740	ms/op
MyBenchmark.mmWriteIntFile	1000000	20000000	30	avgt	10	472,575	±	24,056	ms/op
MyBenchmark.mmWriteIntFile	10000000	20000000	1	avgt	10	236,929	±	12,579	ms/op
MyBenchmark.mmWriteIntFile	10000000	20000000	2	avgt	10	219,880	±	16,804	ms/op
MyBenchmark.mmWriteIntFile	10000000	20000000	3	avgt	10	213,703	±	18,547	ms/op
MyBenchmark.mmWriteIntFile	10000000	20000000	4	avgt	10	208,655	±	10,678	ms/op
MyBenchmark.mmWriteIntFile	10000000	20000000	5	avgt	10	209,069	±	13,405	ms/op
MyBenchmark.mmWriteIntFile	10000000	20000000	10	avgt	10	365,499	±	4,485	ms/op
MyBenchmark.mmWriteIntFile	10000000	20000000	15	avgt	8	459,802	±	75,592	ms/op
MyBenchmark.mmWriteIntFile	10000000	20000000	20	avgt	5	468,664	±	37,525	ms/op
MyBenchmark.mmWriteIntFile	10000000	20000000	25	avgt	4	501,647	±	206,111	ms/op
MyBenchmark.mmWriteIntFile	10000000	20000000	30	avgt	6	529,865	±	70,105	ms/op
MyBenchmark.mmWriteIntFile	20000000	20000000	1	avgt	10	262,690	±	81,253	ms/op
MyBenchmark.mmWriteIntFile	20000000	20000000	2	avgt	10	264,602	±	28,818	ms/op
MyBenchmark.mmWriteIntFile	20000000	20000000	3	avgt	10	265,337	±	26,037	ms/op
MyBenchmark.mmWriteIntFile	20000000	20000000	4	avgt	10	242,837	±	30,159	ms/op
MyBenchmark.mmWriteIntFile	20000000	20000000	5	avgt	6	286,358	±	111,618	ms/op
MyBenchmark.mmWriteIntFile	20000000	20000000	10	avgt	5	420,907	±	69,193	ms/op
MyBenchmark.mmWriteIntFile	20000000	20000000	15	avgt	3	471,738	±	372,433	ms/op

Figure 9: 2nd Test on buffer size

All of those previous benchmarks leads to the fact that the memory mapping mechanisms is the fastest and it doesn't required a lot of file in input in order to get a proper score.

2.6.2 Read-Benchmark

Now that all the implementation of the write function are done. The focus is done on the different implementations of the read function. After running a few benchmarks, it directly appears that once more the fourth mechanisms is the fastest and that the second and third one have almost the same time of execution.

For the first run, k is taking its value in the following list {"1","2","3","4","5", "10","15","20","25","30"}, and for each value of k a test is done with an input file of 100.000 in 32-bit integers and a buffer of size 1.000 in 32-bits Integers.

```
Result "com.ulb.psk.MyBenchmark.readIntFile":
  3,515 ±(99.9%) 0,149 ms/op [Average]
  (min, avg, max) = (3,406, 3,515, 3,712), stdev = 0,099
  CI (99.9%): [3,366, 3,664] (assumes normal distribution)


# Run complete. Total time: 00:07:13

Benchmark                   (bufferSize)  (dataVolume)  (filesToRead)  Mode  Cnt  Score   Error  Units
MyBenchmark.mmReadIntFile           1000          1000              1  avgt   10  0,019 ± 0,003  ms/op
MyBenchmark.mmReadIntFile           1000          1000              2  avgt   10  0,022 ± 0,004  ms/op
MyBenchmark.mmReadIntFile           1000          1000              3  avgt   10  0,024 ± 0,004  ms/op
MyBenchmark.mmReadIntFile           1000          1000              4  avgt   10  0,025 ± 0,006  ms/op
MyBenchmark.mmReadIntFile           1000          1000              5  avgt   10  0,026 ± 0,007  ms/op
MyBenchmark.mmReadIntFile           1000          1000             10  avgt   10  0,037 ± 0,020  ms/op
MyBenchmark.mmReadIntFile           1000          1000             15  avgt   10  0,049 ± 0,024  ms/op
MyBenchmark.mmReadIntFile           1000          1000             20  avgt   10  0,065 ± 0,045  ms/op
MyBenchmark.mmReadIntFile           1000          1000             25  avgt   10  0,061 ± 0,046  ms/op
MyBenchmark.mmReadIntFile           1000          1000             30  avgt   10  0,060 ± 0,034  ms/op
MyBenchmark.readIntFile              N/A          1000              1  avgt   10  3,439 ± 0,127  ms/op
MyBenchmark.readIntFile              N/A          1000              2  avgt   10  3,530 ± 0,305  ms/op
MyBenchmark.readIntFile              N/A          1000              3  avgt   10  3,440 ± 0,208  ms/op
MyBenchmark.readIntFile              N/A          1000              4  avgt   10  3,421 ± 0,062  ms/op
MyBenchmark.readIntFile              N/A          1000              5  avgt   10  3,403 ± 0,087  ms/op
MyBenchmark.readIntFile              N/A          1000             10  avgt   10  3,492 ± 0,149  ms/op
MyBenchmark.readIntFile              N/A          1000             15  avgt   10  3,512 ± 0,141  ms/op
MyBenchmark.readIntFile              N/A          1000             20  avgt   10  3,531 ± 0,172  ms/op
MyBenchmark.readIntFile              N/A          1000             25  avgt   10  3,684 ± 0,393  ms/op
MyBenchmark.readIntFile              N/A          1000             30  avgt   10  3,515 ± 0,149  ms/op
```

Figure 10: Test N = 1.000

The first run highlights the fact

For the second run, the data volume is set to 10.000 32-bits integers.

```
Result "com.ulb.psk.MyBenchmark.readIntFile":
  35,053 ±(99.9%) 1,465 ms/op [Average]
  (min, avg, max) = (33,596, 35,053, 36,401), stdev = 0,969
  CI (99.9%): [33,588, 36,518] (assumes normal distribution)


# Run complete. Total time: 00:07:12

Benchmark                     (bufferSize)  (dataVolume)  (filesToRead)  Mode  Cnt   Score    Error  Units
MyBenchmark.mmReadIntFile             1000         10000              1  avgt   10   0,051 ±  0,005  ms/op
MyBenchmark.mmReadIntFile             1000         10000              2  avgt   10   0,064 ±  0,012  ms/op
MyBenchmark.mmReadIntFile             1000         10000              3  avgt   10   0,078 ±  0,007  ms/op
MyBenchmark.mmReadIntFile             1000         10000              4  avgt   10   0,097 ±  0,010  ms/op
MyBenchmark.mmReadIntFile             1000         10000              5  avgt   10   0,116 ±  0,012  ms/op
MyBenchmark.mmReadIntFile             1000         10000             10  avgt   10   0,178 ±  0,003  ms/op
MyBenchmark.mmReadIntFile             1000         10000             15  avgt   10   0,161 ±  0,012  ms/op
MyBenchmark.mmReadIntFile             1000         10000             20  avgt   10   0,175 ±  0,033  ms/op
MyBenchmark.mmReadIntFile             1000         10000             25  avgt   10   0,189 ±  0,054  ms/op
MyBenchmark.mmReadIntFile             1000         10000             30  avgt   10   0,208 ±  0,067  ms/op
MyBenchmark.readIntFile                N/A         10000              1  avgt   10   3,521 ±  0,072  ms/op
MyBenchmark.readIntFile                N/A         10000              2  avgt   10   7,009 ±  0,179  ms/op
MyBenchmark.readIntFile                N/A         10000              3  avgt   10  10,444 ±  0,216  ms/op
MyBenchmark.readIntFile                N/A         10000              4  avgt   10  13,907 ±  0,186  ms/op
MyBenchmark.readIntFile                N/A         10000              5  avgt   10  17,129 ±  0,514  ms/op
MyBenchmark.readIntFile                N/A         10000             10  avgt   10  34,641 ±  0,962  ms/op
MyBenchmark.readIntFile                N/A         10000             15  avgt   10  34,300 ±  0,832  ms/op
MyBenchmark.readIntFile                N/A         10000             20  avgt   10  34,830 ±  1,022  ms/op
MyBenchmark.readIntFile                N/A         10000             25  avgt   10  34,581 ±  0,768  ms/op
MyBenchmark.readIntFile                N/A         10000             30  avgt   10  35,053 ±  1,465  ms/op
```

Figure 11: Test N = 10.000

For the third run, the data volume is set to 100.000 32-bits integers.

```
Result "com.ulb.psk.MyBenchmark.readIntFile":
  104,734 ±(99.9%) 1,479 ms/op [Average]
  (min, avg, max) = (103,371, 104,734, 106,095), stdev = 0,979
  CI (99.9%): [103,254, 106,213] (assumes normal distribution)


# Run complete. Total time: 00:07:06

Benchmark                     (bufferSize)  (dataVolume)  (filesToRead)  Mode  Cnt    Score    Error  Units
MyBenchmark.mmReadIntFile             1000        100000              1  avgt   10    0,237 ±  0,079  ms/op
MyBenchmark.mmReadIntFile             1000        100000              2  avgt   10    0,285 ±  0,016  ms/op
MyBenchmark.mmReadIntFile             1000        100000              3  avgt   10    0,344 ±  0,077  ms/op
MyBenchmark.mmReadIntFile             1000        100000              4  avgt   10    0,350 ±  0,039  ms/op
MyBenchmark.mmReadIntFile             1000        100000              5  avgt   10    0,347 ±  0,030  ms/op
MyBenchmark.mmReadIntFile             1000        100000             10  avgt   10    0,406 ±  0,022  ms/op
MyBenchmark.mmReadIntFile             1000        100000             15  avgt   10    0,495 ±  0,053  ms/op
MyBenchmark.mmReadIntFile             1000        100000             20  avgt   10    0,545 ±  0,048  ms/op
MyBenchmark.mmReadIntFile             1000        100000             25  avgt   10    0,643 ±  0,069  ms/op
MyBenchmark.mmReadIntFile             1000        100000             30  avgt   10    0,719 ±  0,065  ms/op
MyBenchmark.readIntFile                N/A        100000              1  avgt   10    3,673 ±  0,092  ms/op
MyBenchmark.readIntFile                N/A        100000              2  avgt   10    7,254 ±  0,200  ms/op
MyBenchmark.readIntFile                N/A        100000              3  avgt   10   10,811 ±  0,343  ms/op
MyBenchmark.readIntFile                N/A        100000              4  avgt   10   14,240 ±  0,396  ms/op
MyBenchmark.readIntFile                N/A        100000              5  avgt   10   17,713 ±  0,435  ms/op
MyBenchmark.readIntFile                N/A        100000             10  avgt   10   34,824 ±  0,988  ms/op
MyBenchmark.readIntFile                N/A        100000             15  avgt   10   52,464 ±  1,598  ms/op
MyBenchmark.readIntFile                N/A        100000             20  avgt   10   70,205 ±  2,520  ms/op
MyBenchmark.readIntFile                N/A        100000             25  avgt   10   90,564 ±  3,898  ms/op
MyBenchmark.readIntFile                N/A        100000             30  avgt   10  104,734 ±  1,479  ms/op
```

Figure 12: Test N = 100.000

For the fourth run, the data volume is set to 1.000.000 32-bits integers.

```
Result "com.ulb.psk.MyBenchmark.readIntFile":
  107,057 ±(99.9%) 2,537 ms/op [Average]
  (min, avg, max) = (104,926, 107,057, 109,801), stdev = 1,678
  CI (99.9%): [104,520, 109,594] (assumes normal distribution)


# Run complete. Total time: 00:07:12

Benchmark                   (bufferSize)  (dataVolume)  (filesToRead)  Mode  Cnt    Score   Error  Units
MyBenchmark.mmReadIntFile           1000       1000000              1  avgt   10    2,225 ± 0,444  ms/op
MyBenchmark.mmReadIntFile           1000       1000000              2  avgt   10    2,688 ± 0,228  ms/op
MyBenchmark.mmReadIntFile           1000       1000000              3  avgt   10    2,680 ± 0,234  ms/op
MyBenchmark.mmReadIntFile           1000       1000000              4  avgt   10    2,777 ± 0,212  ms/op
MyBenchmark.mmReadIntFile           1000       1000000              5  avgt   10    2,802 ± 0,466  ms/op
MyBenchmark.mmReadIntFile           1000       1000000             10  avgt   10    2,549 ± 0,053  ms/op
MyBenchmark.mmReadIntFile           1000       1000000             15  avgt   10    2,713 ± 0,174  ms/op
MyBenchmark.mmReadIntFile           1000       1000000             20  avgt   10    2,873 ± 0,199  ms/op
MyBenchmark.mmReadIntFile           1000       1000000             25  avgt   10    2,753 ± 0,092  ms/op
MyBenchmark.mmReadIntFile           1000       1000000             30  avgt   10    2,850 ± 0,061  ms/op
MyBenchmark.readIntFile              N/A       1000000              1  avgt   10    5,449 ± 0,120  ms/op
MyBenchmark.readIntFile              N/A       1000000              2  avgt   10    9,314 ± 0,256  ms/op
MyBenchmark.readIntFile              N/A       1000000              3  avgt   10   12,842 ± 0,393  ms/op
MyBenchmark.readIntFile              N/A       1000000              4  avgt   10   16,139 ± 0,234  ms/op
MyBenchmark.readIntFile              N/A       1000000              5  avgt   10   19,549 ± 0,573  ms/op
MyBenchmark.readIntFile              N/A       1000000             10  avgt   10   37,462 ± 1,151  ms/op
MyBenchmark.readIntFile              N/A       1000000             15  avgt   10   54,486 ± 1,582  ms/op
MyBenchmark.readIntFile              N/A       1000000             20  avgt   10   71,857 ± 1,964  ms/op
MyBenchmark.readIntFile              N/A       1000000             25  avgt   10   89,376 ± 1,911  ms/op
MyBenchmark.readIntFile              N/A       1000000             30  avgt   10  107,057 ± 2,537  ms/op
```

Figure 13: Test N = 1.000.000

For the fifth run, the data volume is set to 5.000.000 32-bits integers.

```
Result "com.ulb.psk.MyBenchmark.readIntFile":
  136,959 ±(99.9%) 2,997 ms/op [Average]
  (min, avg, max) = (132,980, 136,959, 138,862), stdev = 1,983
  CI (99.9%): [133,962, 139,956] (assumes normal distribution)


# Run complete. Total time: 00:07:18

Benchmark                    (bufferSize)  (dataVolume)  (filesToRead)  Mode  Cnt    Score    Error  Units
MyBenchmark.mmReadIntFile            1000       5000000              1  avgt   10   11,745 ±  0,874  ms/op
MyBenchmark.mmReadIntFile            1000       5000000              2  avgt   10   14,774 ±  1,135  ms/op
MyBenchmark.mmReadIntFile            1000       5000000              3  avgt   10   14,993 ±  1,169  ms/op
MyBenchmark.mmReadIntFile            1000       5000000              4  avgt   10   14,461 ±  0,863  ms/op
MyBenchmark.mmReadIntFile            1000       5000000              5  avgt   10   14,080 ±  0,507  ms/op
MyBenchmark.mmReadIntFile            1000       5000000             10  avgt   10   13,530 ±  0,579  ms/op
MyBenchmark.mmReadIntFile            1000       5000000             15  avgt   10   13,593 ±  0,484  ms/op
MyBenchmark.mmReadIntFile            1000       5000000             20  avgt   10   13,700 ±  0,497  ms/op
MyBenchmark.mmReadIntFile            1000       5000000             25  avgt   10   13,577 ±  0,252  ms/op
MyBenchmark.mmReadIntFile            1000       5000000             30  avgt   10   13,880 ±  0,350  ms/op
MyBenchmark.readIntFile               N/A       5000000              1  avgt   10   15,141 ±  0,280  ms/op
MyBenchmark.readIntFile               N/A       5000000              2  avgt   10   22,270 ±  0,537  ms/op
MyBenchmark.readIntFile               N/A       5000000              3  avgt   10   26,095 ±  0,716  ms/op
MyBenchmark.readIntFile               N/A       5000000              4  avgt   10   29,826 ±  0,280  ms/op
MyBenchmark.readIntFile               N/A       5000000              5  avgt   10   34,321 ±  1,144  ms/op
MyBenchmark.readIntFile               N/A       5000000             10  avgt   10   55,156 ±  2,156  ms/op
MyBenchmark.readIntFile               N/A       5000000             15  avgt   10   76,161 ±  1,877  ms/op
MyBenchmark.readIntFile               N/A       5000000             20  avgt   10   96,675 ±  1,870  ms/op
MyBenchmark.readIntFile               N/A       5000000             25  avgt   10  116,978 ±  2,687  ms/op
MyBenchmark.readIntFile               N/A       5000000             30  avgt   10  136,959 ±  2,997  ms/op
```

Figure 14: Test N = 5.000.000

Now that different value of N and k have been tried there is just the size of the buffer to play with. In order to test the impact of the buffer size, B is set to multiple value;10.000, 50.000, 100.000, 1.000.000 and 10.000.000. More over in order to test this with a big size in input, N is set to 10.000.000 32-bits integers. This lead to the following results :

```
Result "com.ulb.psk.MyBenchmark.readIntFile":
  18719,727 ±(99.9%) 297,961 ms/op [Average]
  (min, avg, max) = (18304,200, 18719,727, 18987,037), stdev = 197,083
  CI (99.9%): [18421,766, 19017,689] (assumes normal distribution)


# Run complete. Total time: 01:32:01
```

Benchmark	(bufferSize)	(dataVolume)	(filesToRead)	Mode	Cnt	Score		Error	Units
MyBenchmark.mmReadIntFile	10000	5000000	1	avgt	10	12,875	±	2,442	ms/op
MyBenchmark.mmReadIntFile	10000	5000000	2	avgt	10	15,441	±	1,542	ms/op
MyBenchmark.mmReadIntFile	10000	5000000	3	avgt	10	32,350	±	26,112	ms/op
MyBenchmark.mmReadIntFile	10000	5000000	4	avgt	10	17,152	±	2,315	ms/op
MyBenchmark.mmReadIntFile	10000	5000000	5	avgt	10	19,991	±	6,075	ms/op
MyBenchmark.mmReadIntFile	10000	5000000	10	avgt	10	18,611	±	5,316	ms/op
MyBenchmark.mmReadIntFile	10000	5000000	15	avgt	10	16,528	±	1,317	ms/op
MyBenchmark.mmReadIntFile	10000	5000000	20	avgt	10	17,380	±	0,564	ms/op
MyBenchmark.mmReadIntFile	10000	5000000	25	avgt	10	18,095	±	0,730	ms/op
MyBenchmark.mmReadIntFile	10000	5000000	30	avgt	10	18,845	±	0,543	ms/op
MyBenchmark.mmReadIntFile	50000	5000000	1	avgt	10	12,279	±	0,660	ms/op
MyBenchmark.mmReadIntFile	50000	5000000	2	avgt	10	15,800	±	0,218	ms/op
MyBenchmark.mmReadIntFile	50000	5000000	3	avgt	10	16,794	±	0,418	ms/op
MyBenchmark.mmReadIntFile	50000	5000000	4	avgt	10	17,507	±	0,409	ms/op
MyBenchmark.mmReadIntFile	50000	5000000	5	avgt	10	18,651	±	0,453	ms/op
MyBenchmark.mmReadIntFile	50000	5000000	10	avgt	10	22,391	±	0,704	ms/op
MyBenchmark.mmReadIntFile	50000	5000000	15	avgt	10	26,194	±	1,077	ms/op
MyBenchmark.mmReadIntFile	50000	5000000	20	avgt	10	31,405	±	0,746	ms/op
MyBenchmark.mmReadIntFile	50000	5000000	25	avgt	10	37,415	±	3,607	ms/op
MyBenchmark.mmReadIntFile	50000	5000000	30	avgt	10	40,810	±	2,984	ms/op
MyBenchmark.mmReadIntFile	100000	5000000	1	avgt	10	13,366	±	0,812	ms/op
MyBenchmark.mmReadIntFile	100000	5000000	2	avgt	10	17,991	±	1,183	ms/op
MyBenchmark.mmReadIntFile	100000	5000000	3	avgt	10	19,205	±	0,494	ms/op
MyBenchmark.mmReadIntFile	100000	5000000	4	avgt	10	21,072	±	0,637	ms/op
MyBenchmark.mmReadIntFile	100000	5000000	5	avgt	10	23,013	±	0,735	ms/op
MyBenchmark.mmReadIntFile	100000	5000000	10	avgt	10	31,194	±	1,199	ms/op
MyBenchmark.mmReadIntFile	100000	5000000	15	avgt	10	39,261	±	0,968	ms/op
MyBenchmark.mmReadIntFile	100000	5000000	20	avgt	10	49,326	±	1,684	ms/op
MyBenchmark.mmReadIntFile	100000	5000000	25	avgt	10	62,251	±	3,664	ms/op
MyBenchmark.mmReadIntFile	100000	5000000	30	avgt	10	77,889	±	11,909	ms/op
MyBenchmark.mmReadIntFile	1000000	5000000	1	avgt	10	31,471	±	1,911	ms/op
MyBenchmark.mmReadIntFile	1000000	5000000	2	avgt	10	54,484	±	2,044	ms/op
MyBenchmark.mmReadIntFile	1000000	5000000	3	avgt	10	74,664	±	4,196	ms/op
MyBenchmark.mmReadIntFile	1000000	5000000	4	avgt	10	98,325	±	7,811	ms/op
MyBenchmark.mmReadIntFile	1000000	5000000	5	avgt	10	118,656	±	9,817	ms/op
MyBenchmark.mmReadIntFile	1000000	5000000	10	avgt	10	110,955	±	10,422	ms/op
MyBenchmark.mmReadIntFile	1000000	5000000	15	avgt	10	111,139	±	24,063	ms/op
MyBenchmark.mmReadIntFile	1000000	5000000	20	avgt	10	118,878	±	21,220	ms/op
MyBenchmark.mmReadIntFile	1000000	5000000	25	avgt	10	108,990	±	10,534	ms/op
MyBenchmark.mmReadIntFile	1000000	5000000	30	avgt	10	124,856	±	23,112	ms/op
MyBenchmark.mmReadIntFile	10000000	5000000	1	avgt	10	103,828	±	16,509	ms/op
MyBenchmark.mmReadIntFile	10000000	5000000	2	avgt	10	119,951	±	14,541	ms/op
MyBenchmark.mmReadIntFile	10000000	5000000	3	avgt	10	139,568	±	32,350	ms/op
MyBenchmark.mmReadIntFile	10000000	5000000	4	avgt	10	124,137	±	23,244	ms/op
MyBenchmark.mmReadIntFile	10000000	5000000	5	avgt	10	127,802	±	13,124	ms/op
MyBenchmark.mmReadIntFile	10000000	5000000	10	avgt	10	127,057	±	8,898	ms/op
MyBenchmark.mmReadIntFile	10000000	5000000	15	avgt	10	107,448	±	24,572	ms/op
MyBenchmark.mmReadIntFile	10000000	5000000	20	avgt	10	124,500	±	18,767	ms/op
MyBenchmark.mmReadIntFile	10000000	5000000	25	avgt	10	160,425	±	58,940	ms/op
MyBenchmark.mmReadIntFile	10000000	5000000	30	avgt	10	169,808	±	74,346	ms/op
MyBenchmark.readIntFile	N/A	5000000	1	avgt	10	22195,341	±	2662,440	ms/op
MyBenchmark.readIntFile	N/A	5000000	2	avgt	10	20444,796	±	83,715	ms/op
MyBenchmark.readIntFile	N/A	5000000	3	avgt	10	24370,901	±	4289,356	ms/op
MyBenchmark.readIntFile	N/A	5000000	4	avgt	10	24041,832	±	1807,636	ms/op
MyBenchmark.readIntFile	N/A	5000000	5	avgt	10	23285,622	±	1473,546	ms/op
MyBenchmark.readIntFile	N/A	5000000	10	avgt	10	23823,957	±	4217,155	ms/op
MyBenchmark.readIntFile	N/A	5000000	15	avgt	10	20903,211	±	923,341	ms/op
MyBenchmark.readIntFile	N/A	5000000	20	avgt	10	21964,376	±	1567,025	ms/op
MyBenchmark.readIntFile	N/A	5000000	25	avgt	10	20506,977	±	811,356	ms/op
MyBenchmark.readIntFile	N/A	5000000	30	avgt	10	18719,727	±	297,961	ms/op

Figure 15: Test on buffer size

2.7 Discussion of expected behavior vs experimental observations

Those benchmarks seem to give results that are the same as the expected behavior. In fact the First implementation is from far away the one that take the more time to read and write a lot of data. The second and the third one have almost the same time of execution and the fourth one is almost all the time getting better result. Thus the stream implementation to use for the external multi-way merge-sort algorithm is the fourth mechanism. So the algorithm will use memory mapping mechanism to read and write faster.

3 Observations on multi-way merge sort

3.1 Implementation

First a multi way merging algorithm needs to be implemented. The algorithm consists of a d-way merging algorithm that, given d sorted input streams of 32-bit integer, creates a single output stream containing the elements from the input stream in sorted order.

In order to give to the algorithm the d sorted input streams, an ArrayList is filled up with ArrayList that represent each one an input stream. The merging is using a priority queue in order to obtain the next element to be output at all times.

Then once the d-way merging algorithm is done, a new implementation needs to be done. In fact the new algorithm to implement is an external memory multi-way merge-sort algorithm for sorting 32-bit integers. The algorithm takes as input an input file and the following parameters:

- M - the size of the main memory available during the first sort phase, in number of 32-bit integers;
- d - the number of streams to merge in one pass in the later sort phases

Moreover N is defined as being the size of the input file, measured in number of 32-bit integers. The program should be able to sort in a particular way.

- The program needs to read the input file and be able to split in into N/M streams. In our implementation, the program creates first N/M files then he fills each of them with a sorted stream. In order to sort all the stream the method sort from java.util.Collections is used. According to the documentation this sort algorithm offers guaranteed nlog(n) performance.
- In order to store the references to the N/M streams a queue is used.
- At this point the program needs to merge the d first streams in the queue and put the resulting stream at the end of the queue. This has to be

repeated until the number of streams that remain in the queue is under d.

TCB:
In this project two-pass algorithms are used, where data from the operand relations is read into main memory, processed, written out to disk again, and then reread from disk to complete the operation.Two passes are usually enough, even for very large relations and generalizing it to more than two passes is not hard. It begins with an implementation of the sorting operator r it divides the relation R for which B(R) greater than M into chucks of size M, sort them, and then process the sorted sublists.It requires only one block of each sorted sublist in main memory .
Two-Phase Multiway Merge-Sort (TPMMS) sorts very large relations in two passes.M is used,as the main memory buffers to sort. TPMMS fills repeatedly the M buffers with new tuples from R (Relation) uses main-memory sorting algorithm and sorts them and then writes out each sorted sublist to secondary storage after that it merges at most M — 1 sorted sublists to secondary storage, which limits the size of R. One input block to each sorted sublist and one block are allocated to the output. The smallest key among the first remaining elements of all the lists with is found and this comparison is done in main memory. A linear search is used for taking a number of machine instructions proportional to the number of sublists.A method based on "priority queues"2 is used that takes time proportional to the logarithm of the number of sublists to find the smallest element then the smallest element is moved to the first available position of the output block.If the output block is full, write it to disk and reinitialize the same buffer in main memory to hold the next output block.If the block from which the smallest element was just taken is now exhausted of records, read the next block from the same sorted sublist into the same buffer that was used for the block just exhausted.If no blocks remain, then leave its buffer empty and do not consider elements from that list in any further competition for smallest remaining elements.
In order for TPMMS to work, there must be no more than M — 1 sublists.Suppose R fits on B blocks. Since each sublist consists of M blocks, the number of sublists is J3 /M . thus it requires B / M ¡= M — 1, or B ¡= M (M — 1) (or about B ¡= M power 2). The algorithm requires us to read B blocks in the first pass, and another B disk I/O 's to write the sorted sublists. The sorted sublists are each read again in the second pass, resulting in a total of 3 B disk I/O 's. If, as is customary, The cost of writing the result to disk is not counted (since the result may be pipelined and never written to disk), then 2¿B is all that the sorting operator r requires. However, if storing the result on disk is needed to store the result on disk, then the requirement is 4 B.
2PMMS is sufficient to sort all but an incredibly large relation in two passes. However, if you have an even bigger relation, then the same idea can be applied recursively. Divide the relation into chunks of size M (M — 1), use 2PMMS to sort each one, and then treat the resulting sorted lists as sublists for a third pass. The idea extends similarly to any number of passes.

3.2 Expected behavior

Before running any test, a cost formula needs to be build for this implementation. The cost estimates the total number of I/O's that need to be done in function of N, d and M.

- N - the size of the input file, measured in number of 32-bit integers
- d - the number of streams to merge in one pass in the later sort phases
- M - the size of the main memory available during the first sort phase, in number of 32-bit integers

According to the course of Prof. S.VANSUMMEREN, a demonstration of the cost formula is build. Here is the demonstration of the cost formula :

The input that need to be sort is a relation R with a size of N 32-bit integers. Let's introduce N(R) as the number of blocks of the relations.

- In the first pass the program reads M blocks at the same time from the input relation R, sort these by means of the method sort from java.util.Collections, and write the sorted resulting sublist to disk. After the first pass there is $N(R)/M$ sorted sublists of M blocks each.
- In the following passes the program read d blocks from these sublists and merge them into larger sorted sublists. (After the second pass there is $N(R)/(M * d)$ sorted sublists of $M * d$ blocks each, after the third pass $N(R)/(M * d^2)$ sorted sublists, etc ..)
- The algorithm repeats until it obtains a single sorted sublist.

What is the complexity of this algorithms ?

- In each pass we read and write the entire input relation exactly once.
- There are $\lceil \log_M N(R) \rceil$ passes
- The total cost is hence 2N(R) $\log_M N(R)$ I/O operations.

3.3 Experimental observations

In order to test the external multi way merge algorithms with the three parameters; N, M and d, a benchmark has been implemented.

The first experiment is to vary only N and to keep M and d fixed. During the run N is taking those following values : 1.000.000, 2.000.000, 4.000.000, 8.000.000.

```
Result "com.ulb.psk.benchmark.MyBenchmark.externalMerge":
  90951,075 ±(99.9%) 66541,415 ms/op [Average]
  (min, avg, max) = (63867,396, 90951,075, 210048,026), stdev = 44013,026
  CI (99.9%): [24409,660, 157492,489] (assumes normal distribution)


# Run complete. Total time: 00:50:05

Benchmark                     (d)  (fileSize)  (firstPhaseMemory)  Mode  Cnt     Score       Error  Units
MyBenchmark.externalMerge      10    10000000               10000  avgt   10   7106,874 ±   610,026  ms/op
MyBenchmark.externalMerge      10    20000000               10000  avgt   10  15946,740 ±   859,903  ms/op
MyBenchmark.externalMerge      10    40000000               10000  avgt   10  39383,103 ±  2588,189  ms/op
MyBenchmark.externalMerge      10    80000000               10000  avgt   10  90951,075 ± 66541,415  ms/op
```

Figure 16: Test on N

The conclusion of this first experiment is quite obvious, because more big is the input file more it will take time to apply the algorithm.

The second experiment is to vary only M and to keep N and d fixed. During the run M is taking those following values : 10.000, 100.000, 1.000.000, 10.000.000.

```
Result "com.ulb.psk.benchmark.MyBenchmark.externalMerge":
  31086,351 ±(99.9%) 2715,593 ms/op [Average]
  (min, avg, max) = (28816,708, 31086,351, 33867,968), stdev = 1796,197
  CI (99.9%): [28370,758, 33801,944] (assumes normal distribution)


# Run complete. Total time: 00:52:11

Benchmark                     (d)  (fileSize)  (firstPhaseMemory)  Mode  Cnt     Score      Error  Units
MyBenchmark.externalMerge      10    40000000               10000  avgt   10  38638,161 ± 2913,278  ms/op
MyBenchmark.externalMerge      10    40000000              100000  avgt   10  28104,357 ± 4896,591  ms/op
MyBenchmark.externalMerge      10    40000000             1000000  avgt   10  25602,758 ±  736,116  ms/op
MyBenchmark.externalMerge      10    40000000            10000000  avgt   10  31359,117 ± 1190,794  ms/op
MyBenchmark.externalMerge      10    40000000            10000000  avgt   10  31086,351 ± 2715,593  ms/op
```

Figure 17: Test on M

The result of the second experiment highlights that the best value for M is not the biggest one. In fact if M has a size of 100.000 it is faster than if has a size of 10.000.000. In this experiment the best size for M is 1.000.000.

The third experiment is to vary only d and to keep M and N fixed. During the run M is taking those following values : 2, 3, 4, 5, 10, 15, 20, 30, 40. The value of M is set regarding the last experiment, by taking the value that gave the best result. So M is set to 1.000.000.

```
Result "com.ulb.psk.benchmark.MyBenchmark.externalMerge":
  25763,539 ±(99.9%) 1953,576 ms/op [Average]
  (min, avg, max) = (24380,601, 25763,539, 27776,064), stdev = 1292,170
  CI (99.9%): [23809,963, 27717,116] (assumes normal distribution)


# Run complete. Total time: 01:29:02

Benchmark                   (d)  (fileSize)  (firstPhaseMemory)  Mode  Cnt      Score      Error  Units
MyBenchmark.externalMerge     2    40000000             1000000  avgt   10  41562,686 ± 3558,276  ms/op
MyBenchmark.externalMerge     3    40000000             1000000  avgt   10  34105,524 ± 5112,707  ms/op
MyBenchmark.externalMerge     4    40000000             1000000  avgt   10  30946,335 ± 2335,822  ms/op
MyBenchmark.externalMerge     5    40000000             1000000  avgt   10  28549,205 ± 1511,995  ms/op
MyBenchmark.externalMerge    10    40000000             1000000  avgt   10  26583,125 ± 1431,166  ms/op
MyBenchmark.externalMerge    15    40000000             1000000  avgt   10  26878,014 ± 1022,072  ms/op
MyBenchmark.externalMerge    20    40000000             1000000  avgt   10  26447,133 ±  518,501  ms/op
MyBenchmark.externalMerge    30    40000000             1000000  avgt   10  26986,284 ±  601,210  ms/op
MyBenchmark.externalMerge    40    40000000             1000000  avgt   10  25763,539 ± 1953,576  ms/op
```

Figure 18: Test on d

The third experiment shows that more there is streams to merge in one pass in the later sort phases more faster is the algorithm to execute.

3.4 Discussion of expected behavior vs experimental observations

According to the expected behavior and the experimental observations the good choice for the parameters is to set a big buffer but not too big, 1.10^6 seems to be a good value. More over the number of streams to merge in one pass in the later phases must also be big order to reduce the time of execution.

4 Overall conclusion

During this project we learned about "Java NIO FileChannel","JAVA NIO – Memory-Mapped File","Testing the code performance with JMH tool", "Memory Mapped File in Java", "Revisiting File InputStream and Reader instantiation", "Priority Queue insert with priority" and "Implement K Way Merge Algorithm". We also learned to implement proper benchmarks, and so to use the JHM library.

List of Figures

5 References

http://tutorials.jenkov.com/java-nio/file-channel.html
https://examples.javacodegeeks.com/corejava/nio/filechannel/java-\
nio-channels-filechannel-example/
https://tutorials.techmytalk.com/2014/11/05/java-nio-memory-mapped-files/
https://www.codeproject.com/Tips/683614/Things-to-Know-about-Memory\
-Mapped-File-in-Java
https://www.developer.com/java/other/article.php/1548681/Introduction\
-to-Memory-Mapped-IO-in-Java.htm
https://www.javacodegeeks.com/2013/05/power-of-java-memorymapped-file.
html
http://www.mathcs.emory.edu/~cheung/Courses/554/Syllabus/4query-exec/
2-pass=TPMMS.html/
https://sortingalgorithms.quora.com/K-way-Merge-Algorithms
https://www.programcreek.com/2014/05/merge-k-sorted-arrays-in-java/
http://www.sanfoundry.com/java-program-k-way-merge-algorithm/
http://www.geeksforgeeks.org/merge-k-sorted-arrays/
http://www.mathcs.emory.edu/~cheung/Courses/554/Syllabus/4-query-exec/
2-pass=TPMMS.html/
https://en.wikipedia.org/wiki/K-Way_Merge_Algorithms/
https://www.javamex.com/tutorials/collections/priorityqueue_example_
heapsort.shtml
https://stackoverflow.com/questions/29380093/priorityqueue-insert-with-priority
http://tutorials.jenkov.com/java-nio/file-channel.html
https://tutorials.techmytalk.com/2014/11/05/java-nio-memory-mapped-files
https://www.javacodegeeks.com/2013/05/power-of-java-memorymapped-file.
html
http://tutorials.jenkov.com/java-performance/jmh.html
https://arnaudroger.github.io/blog/2017/03/20/faster-reader-inpustream-in-java.
html
http://cs.ulb.ac.be/public/_media/teaching/infoh417/slides-lect6.pdf
https://blog.goyello.com/2017/06/19/testing-code-performance-jmh-tool/
http://s-j.github.io/getting-started-with-jmh/