UNIVERSITÉ LIBRE DE BRUXELLES

INFO-H417 - Database Systems Architecture

External memory merge sort

Authors:

Yasin Arslan

Alexis Reynouard

Jacky Trinh

Professor:

Stijn Vansummeren

Contents

Chapter 1

Introduction

As part of the course INFO-H-417 Database System Architecture, we were asked to explore several different ways to read data from, and write data to secondary memory. We then have to implement an external-memory merge-sort algorithm and then examine the performance of all the implementation done. All this was written in the C++ language according to the C++11 standard.

Because our goal is to interpret the files as sequences of 32-bits integers to sorts, in the following we will denote by ***element*** four bytes viewed as a single 32-bits integer.

1.1 Environment

1.1.1 Computer specifications

We mainly use two computer to measure the performances of our implementations. We will refer to the first computer as computer1. Here are its specifications: The first computer used to run the benchmarks is:

- Computer model: Lenovo Thinkpad T440p
- Operating System: Manjaro Linux 17.1.0
- CPU: Intel core I7-4600M
- CPU speed: 2.90Ghz
- RAM: 8192 MB
- SSD: Samsung MZ7LN256HCHP (256GB)

Here are the specifications of computer2.

- Computer model: Dell Latitude E6410
- Operating System: Linux Mint Debian 2 (Kernel: Linux 3.16.0-4-amd64 #1 SMP Debian 3.16.51-3 (2017-12-13))
- CPU: Intel(R) Core(TM) i5 M 540
- CPU speed: 2.53GHz
- RAM: 3983 MB
- External memomy: Momentus 7200.4 ST9500420ASG (500GB)

1.1.2 Targeted systems

Our streams and merge-sort implementation should work on any (not too old) UNIX system with the glibc. The file are interpreted as sequence of 4 bytes integers, the endianness depending on the system. Because this can lead to ambiguities if one write a file on a, let's say, little-endian system then read on a big-endian system, we chosen to perform a test at compile-time that check that we are working on a little-endian system. This may be easily changed to work on other systems and is here to warn the user.

The tests and performance measures of our implementation were realised, however, with many other, but still common, softwares like gnu bash, gnu diff, python3, gnu time, *etc.*

Chapter 2

Streams

Since our merge sort algorithm will need to sort disk files, we obviously need to be able to read data from, and write data to disk. That's why we have to implement two types of streams: ***input streams*** and ***output streams***. Each type has its own operations: the input stream is able to open an existing file for reading, sequentially read the elements from the stream and close the stream, the output stream can create a new file, sequentially write elements to the stream and also close the stream.

There are so many ways to implement these two types of streams and we were asked to have four different implementations performing the same operations from above.

Cost formula parameter N is the number of elements that is read or written in each file, k is the number of files and B is the size of the buffer.

2.1 Expected behavior

2.1.1 First implementation

For the first one, we can read and write only one element at a time using the system calls read and write. This is the most basic implementation and we do not expect great performance from this. Indeed, since it will only read / write only one element at a time, we will have one system call per element, thus very poor performance.
The cost formula should be $N \times k$

2.1.2 Second implementation

This implementation is the same as the first one except that we do not specifically use the system call but instead we use the fread and fwrite functions from the C stdio library which has its own buffering mechanism.

Since it has its own buffer, we do expect a greater performance than the first one. The cost formula should be $k \times \lceil \frac{N}{B} \rceil$

2.1.3 Third implementation

We exactly have the same implementation as the first one but this time, we create our own buffer with a size B in internal memory. For the input stream, when the buffer is empty, the next B elements are read from the file. For the output stream case, when the buffer is full, the B elements in the buffer are written to the file. With the use of a buffer, we can make less system calls and therefore the performance should be greater than the first implementation. Compared to the second one, we have to find the optimal B parameter in order to achieve same or better performance.
We used template and C arrays in order to avoid overhead due to virtual calls. The cost formula should be the same as the second implementation: $k \times \lceil \frac{N}{B} \rceil$

2.1.4 Fourth implementation

In this fourth implementation, we use an alternative to standard file I/O called ***memory mapping***. This technique allows us to map a file into main memory, meaning that there's a correspondence between a memory address and a word in the file. In some ways, this is similar to the way the operating system load a program to execute.

This implementation should be one of the fastest since it will store data in the main memory and disk access is completely managed by the operating system.
The cost formula should be the same as the second implementation: $k \times \lceil \frac{N}{B} \rceil$
Note: B is equal to the page size multiplied by the quantity of mapped page.

2.2 Experimental observations

Our implementations was written with to good performance rather than beautiful Object Oriented code. Our streams implementation require a compiler which support the C++11 standard.
We used templates for the parameters of the streams (e.g. the buffer size) to allow the compiler to perform as many optimizations as possible. Indeed, in real cases our streams should be used with some predefined values for these parameters. However this is not convenient to test different values and it is the reason of some "strange code" in our project.

With our code we provide also a lot of other files and script which allow to compile some binaries for testing the streams. The reader may want to run `make`. By default, if all the dependencies are satisfied, this perform tests on the streams implementation as well as others implementation described later (the merge and the merge sort). Theses

steps ensure that the algorithms are well implemented and do what is expected. Then there will be some performances measures realized on the streams: first information on the hardware will be collected and stored in a new sub-directory of benchmark/. Then a file streams.csv will be written in this same sub-directory. This file contains result of performance measures.

The performance measure procedure for every implementation is done in the same way: we open / create k files and proceed to read / write one element from each file before read / write the next. The actual number of bytes read to / write from the disk depend on the internal mechanisms of the stream used.

When realizing performance measures for input, we close a file directly when its end is reached. Especially, for measures on files with different size (for example ranging from 3×10^6 to 4×10^6 integers), we do not wait for the end of all the files to reached to close any of them.

Symmetrically we performed writing tests either on files of the same size or of different sizes.

In order to time the executions, we used the GNU time program and pay specially attention to the time used to run the program in user and system mode. We also take care of cleaning the cache each time a file may be accessed many times for different tests.

2.2.1 First implementation

Here are the performance measures for input. The test was realized on files of size ranging from 12 to 16 MB. We have the time spent to read simultaneously different number of files.

Time - k	**1**	**2**	**5**	**10**	**30**	**100**	**1000**
user	0.844s	0.417s	0.763s	1.337s	4.120s	12.330s	107.040s
sys	0.583s	1.137s	2.840s	6.073s	19.410s	74.290s	979.980s
total	1.427s	1.554s	3.603s	7.410s	23.530s	86.620s	1087.020s

performed on computer1

As expected, those results are very slow. When we ran this implementation with 1000 files, we had to stop it midway because it took forever and was causing the computer to freeze. As we can see, it's mainly the system calls that take the most time.
Let's take a look for the output stream:

Time - k	**1**	**2**	**5**	**10**	**30**	**100**	**1000**
user	0.673s	1.510s	3.473s	6.147s	13.460s	38.463s	0.000s
sys	6.730s	15.233s	38.007s	73.927s	135.577s	366.810s	0.000s
total	7.403s	16.743s	41.480s	80.074s	149.037s	405.273s	0.000s

performed on computer1

For obvious reason, we did not try with 1000 files as it will take too long. Since the writing operation need more resource than the read operation, it was obvious from the start that the output stream would take longer than the input stream.

Note that the major difference is in time spent in kernel mode. This is explained because it is in this mode that each write is performed. The difference in user mode may be due to the generation of random data.

2.2.2 Second implementation

For the input stream:

Time - k	**1**	**2**	**5**	**10**	**30**	**100**	**1000**
user	0.287s	0.337s	0.597s	1.017s	2.843s	10.210s	136.150s
sys	0.000s	0.007s	0.020s	0.027s	0.107s	0.330s	6.043s
total	0.287s	0.344s	0.617s	1.044s	2.950s	10.540s	142.193s

performed on computer1

We did expect it to perform a lot better than the first implementation. Sadly, we couldn't find the information about the fread's own buffer mechanism. From these results, we can really say that there are so little system calls and then conclude that the buffer size must be really big.
And now for the output stream:

Time - k	**1**	**2**	**5**	**10**	**30**	**100**	**1000**
user	0.430s	0.543s	1.160s	2.140s	5.700s	19.367s	207.133s
sys	0.010s	0.007s	0.043s	0.053s	0.190s	0.757s	10.240s
total	0.440s	0.550s	1.203s	2.193s	5.890s	20.124s	217.373s

performed on computer1

We can have the same conclusion as the input stream one, since there's little time spent in these system calls, the buffer size of the fwrite function should be the same as the fread function. As we can see from these results, the output stream from this implementation also take longer than the input stream one but we can see that the difference between those two is small.

2.2.3 Third implementation

In this implementation, we use our own buffer that we've made. The buffer itself is implemented as a C array to have a minimal overhead. The size of the buffer is chosen by a parameter B. We decided to test with a B which is a number of the power of 2. So far we tried with 2, 4, 8, 16, 256, 512, 1024, 2048 and 4096. For this report we won't show

all the tests but we will show you the interesting ones.

We'll first show all the input stream result. For starter, we have $B = 2$ because it means that it's the double compared to the first implementation.

Time - k	**1**	**2**	**5**	**10**	**30**	**100**	**1000**
user	0.210s	0.320s	0.520s	0.953s	2.653s	8.933s	0.000s
sys	0.233s	0.430s	1.140s	2.320s	7.500s	33.790s	0.000s
total	0.443s	0.750s	1.660s	3.273s	10.153s	42.723s	0.000s

performed on computer1

For the same reason as stated above, we did not try to run it with 1000 files since it would also have taken extremely long for nothing.

As expected, the time consumed the more or less two times lesser than with the first implementation.
Now with a size $B = 256$:

Time - k	**1**	**2**	**5**	**10**	**30**	**100**	**1000**
user	0.183s	0.230s	0.367s	0.667s	1.583s	5.110s	54.397s
sys	0.000s	0.010s	0.017s	0.027s	0.157s	0.513s	10.397s
total	0.183s	0.240s	0.384s	0.694s	1.740s	5.623s	64.794s

performed on computer1

$B = 512$

Time - k	**1**	**2**	**5**	**10**	**30**	**100**	**1000**
user	0.183s	0.230s	0.370s	0.650s	1.660s	5.123s	63.957s
sys	0.000s	0.003s	0.010s	0.020s	0.063s	0.300s	8.437s
total	0.183s	0.233s	0.380s	0.670s	1.723s	5.423s	72.394s

performed on computer1

$B = 1024$

Time - k	**1**	**2**	**5**	**10**	**30**	**100**	**1000**
user	0.180s	0.237s	0.373s	0.710s	1.623s	5.577s	69.457s
sys	0.000s	0.000s	0.017s	0.003s	0.070s	0.227s	5.797s
total	0.180s	0.237s	0.390s	0.713s	1.693s	5.804s	75.254s

performed on computer1

$B = 4096$

Time - k	**1**	**2**	**5**	**10**	**30**	**100**	**1000**
user	0.183s	0.227s	0.367s	0.643s	1.713s	5.663s	90.943s
sys	0.000s	0.010s	0.013s	0.023s	0.073s	0.223s	4.307s
total	0.183s	0.237s	0.380s	0.666s	1.786s	5.886s	95.250s

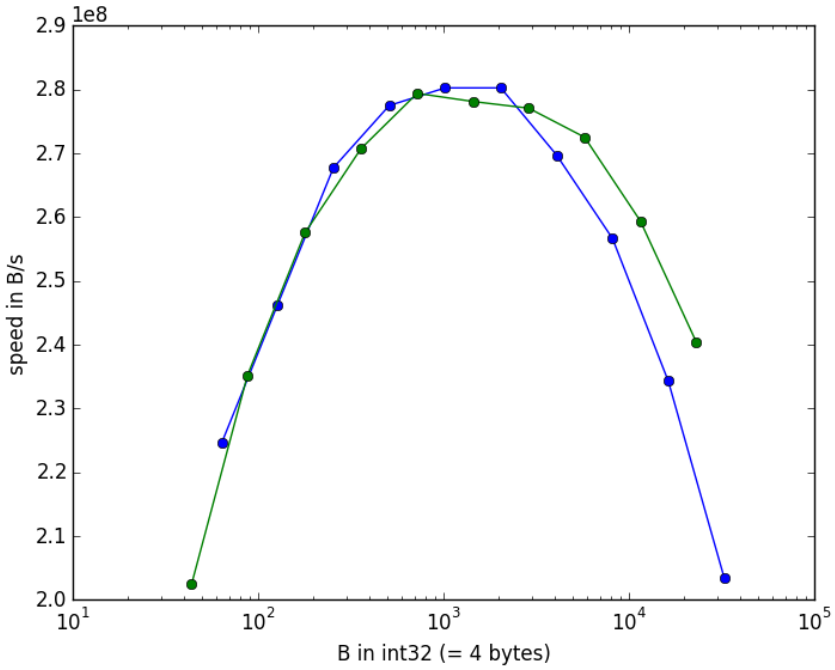
performed on computer1

As we can see, the optimal buffer size is 256 even though 512 was pretty too but when reading 1000 files, 256 is the fastest one. We also can notice that if the size is bigger, we spend a little less time in the system calls but a lot more time are spent in the user mode.

To improve our search of the best buffer size, we performed other measures whose the results are displayed on the next graph. To realize these tests, we read simultaneously k of 16 MiB each, with k varying between 1 and 30, and we compute the average speed for different size B of the buffer.

Here we saw that the best input speed is reached with a buffer of 4096 bytes (1024 integers). With this same tests we saw that the number of file read does not impact much the speed, since we read 2 or more files.

In blue are the power of two and in green the others.



Input Stream 3

performed on computer2

For the output stream now:

$B = 2$

Time - k	**1**	**2**	**5**	**10**	**30**	**100**	**1000**
user	0.527s	0.780s	1.637s	2.960s	7.810s	27.423s	0.000s
sys	1.610s	2.760s	7.137s	14.423s	44.303s	161.017s	0.000s
total	2.137s	3.540s	8.774s	17.383s	52.113s	188.440s	0.000s

performed on computer1

For same reason as above, we did not try with 1000 files with this B.

$B = 256$

Time - k	**1**	**2**	**5**	**10**	**30**	**100**	**1000**
user	0.287s	0.517s	1.067s	1.910s	4.853s	17.077s	176.880s
sys	0.020s	0.057s	0.130s	0.157s	0.557s	2.157s	32.653s
total	0.307s	0.574s	1.197s	2.067s	5.410s	19.234s	209.533s

performed on computer1

$B = 512$

Time - k	**1**	**2**	**5**	**10**	**30**	**100**	**1000**
user	0.367s	0.543s	1.090s	1.987s	5.527s	18.863s	174.450s
sys	0.030s	0.030s	0.070s	0.170s	0.437s	1.890s	20.227s
total	0.397s	0.573s	1.160s	2.157s	5.964s	20.753s	194.677s

performed on computer1

$B = 1024$

Time - k	**1**	**2**	**5**	**10**	**30**	**100**	**1000**
user	0.357s	0.663s	1.157s	2.193s	5.750s	19.270s	183.087s
sys	0.017s	0.027s	0.060s	0.093s	0.337s	1.573s	14.297s
total	0.374s	0.690s	1.217s	2.286s	6.087s	20.843s	197.384s

performed on computer1

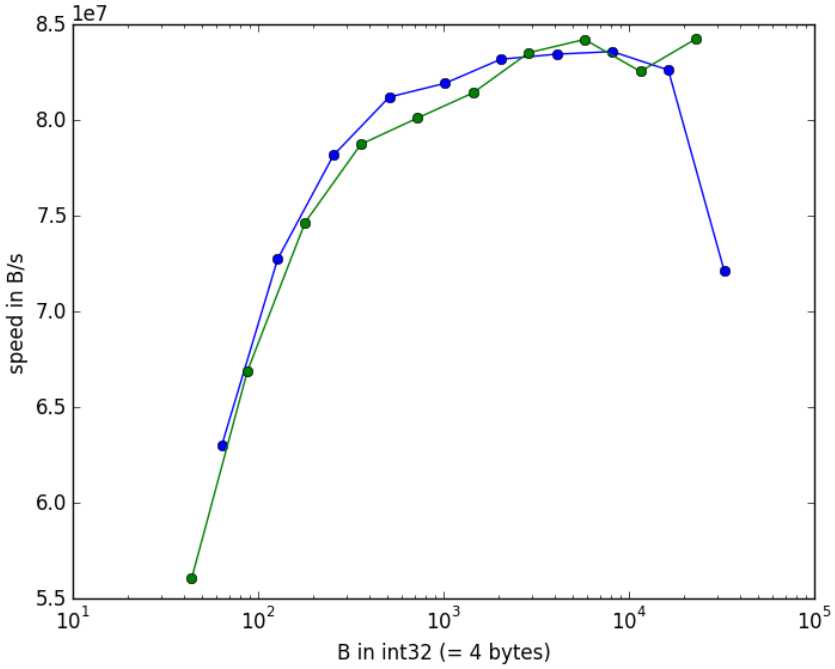
$B = 4096$

Time - k	**1**	**2**	**5**	**10**	**30**	**100**	**1000**
user	0.377s	0.630s	1.173s	2.160s	5.713s	17.583s	187.437s
sys	0.007s	0.010s	0.053s	0.127s	0.390s	1.223s	9.760s
total	0.384s	0.640s	1.226s	2.287s	6.103s	18.806s	197.197s

performed on computer1

From these results, we can see that having $256 \leq B \leq 4096$ for $k \leq 100$ does not change anything, but for $k = 1000$, the optimal value is from 512.

Again, we did some further research to better determine a best value for the buffer size. Theses tests are completely symmetric to those presented above and we get the following results:



Output Stream 3

performed on computer2

As we were wondering why the curve behave strangely with great values for B, we decide to distinguish between values that are power of two (in blue) and those which are not (in green). But there is no notable difference even if power of two seems a bit better (likely due to compiler optimization thanks to data alignment in memory – recall that the size of the buffer is take into account at compile time) and it seems that the best size for the buffer is 4096 bytes (1024 integers).

2.2.4 Fourth implementation

With the mapping system, we have to chose a B which was a multiple of our system page size.

$B = 4096$

Time - k	**1**	**2**	**5**	**10**	**30**	**100**	**1000**
user	0.187s	0.263s	0.400s	0.707s	1.863s	7.717s	154.503s
sys	0.003s	0.003s	0.017s	0.050s	0.127s	0.500s	9.617s
total	0.190s	0.266s	0.417s	0.757s	1.990s	8.217s	164.120s

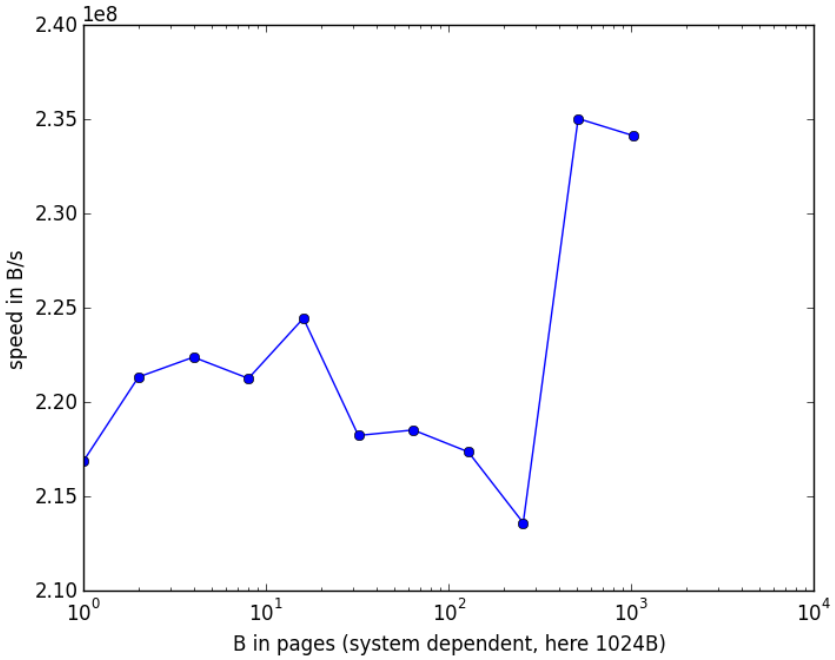
performed on computer1

$B = 8192$

Time - k	**1**	**2**	**5**	**10**	**30**	**100**	**1000**
user	0.187s	0.247s	0.377s	0.737s	1.883s	7.743s	151.967s
sys	0.003s	0.000s	0.040s	0.033s	0.107s	0.483s	9.767s
total	0.190s	0.247s	0.417s	0.770s	1.990s	8.226s	161.734s

performed on computer1

The result seems extremely similar although we tried different B. So, again we did further research. We tried to read simultaneously between 1 and 30 files of size 64MiB each on computer2 which has a smaller page size (1024 bytes). The next graph shows the average read speed as a function of the number of pages mapping the file.



Input Stream 4

performed on computer2

From these results it seems that the performance are independent of the mapping size, may be due to a look-ahead technique of the operating system. However The speed reached does not exceed that of the previous implementation.

Concerning the output:

$B = 4096$

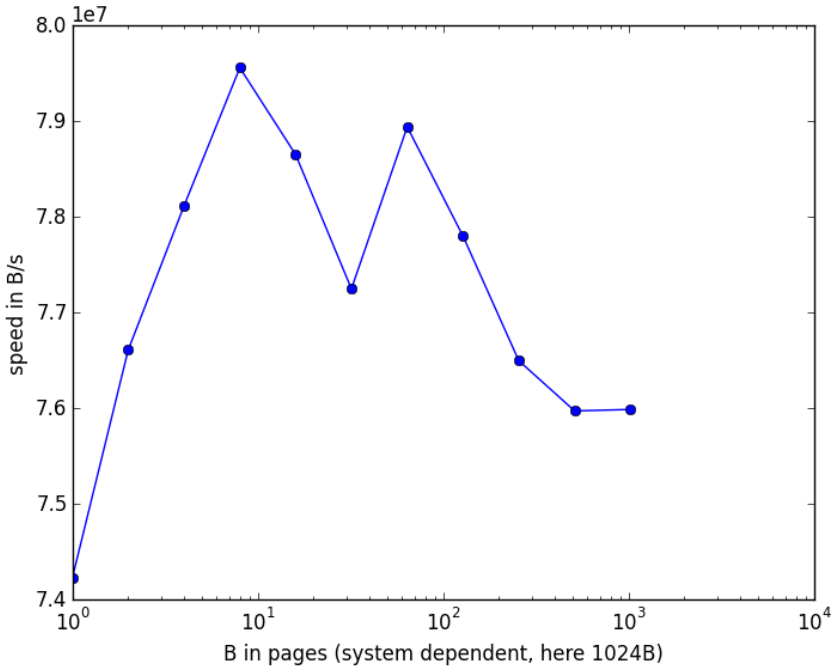
Time - k	**1**	**2**	**5**	**10**	**30**	**100**	**1000**
user	0.347s	0.560s	1.163s	2.167s	5.897s	20.420s	202.080s
sys	0.033s	0.050s	0.087s	0.200s	0.770s	3.027s	28.860s
total	0.380s	0.610s	1.250s	2.367s	6.667s	23.447s	230.940s

performed on computer1

$B = 8192$

Time - k	**1**	**2**	**5**	**10**	**30**	**100**	**1000**
user	0.350s	0.560s	1.113s	2.170s	6.077s	20.043s	196.700s
sys	0.017s	0.060s	0.117s	0.227s	0.683s	2.973s	28.573s
total	0.367s	0.620s	1.230s	2.397s	6.760s	23.016s	225.273s

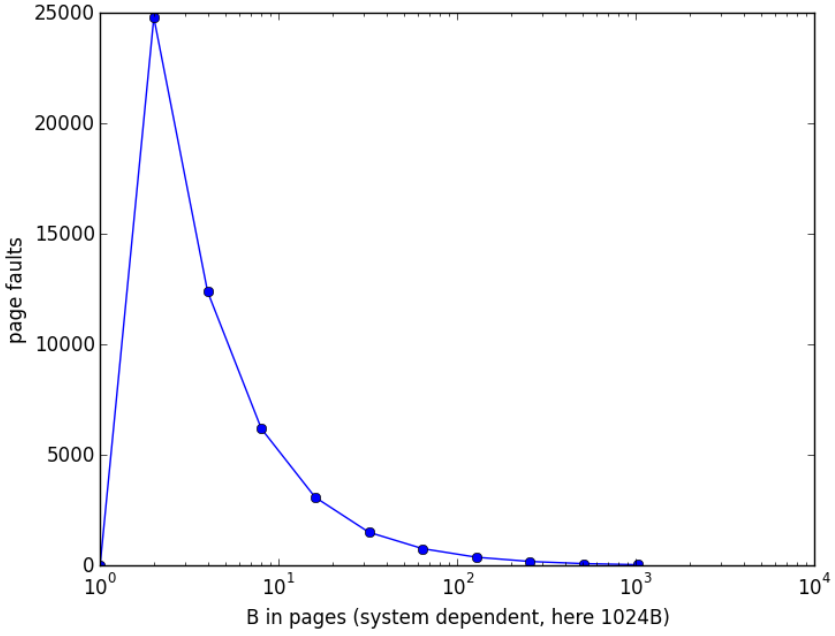
performed on computer1



Input Stream 4

performed on computer2

The output stream results seems very similar. However one interesting observation may be made concerning the number of (major) page faults as a function of B. The number of page fault decrease when the page size increase, as shown in the next graph.



Output Stream 4

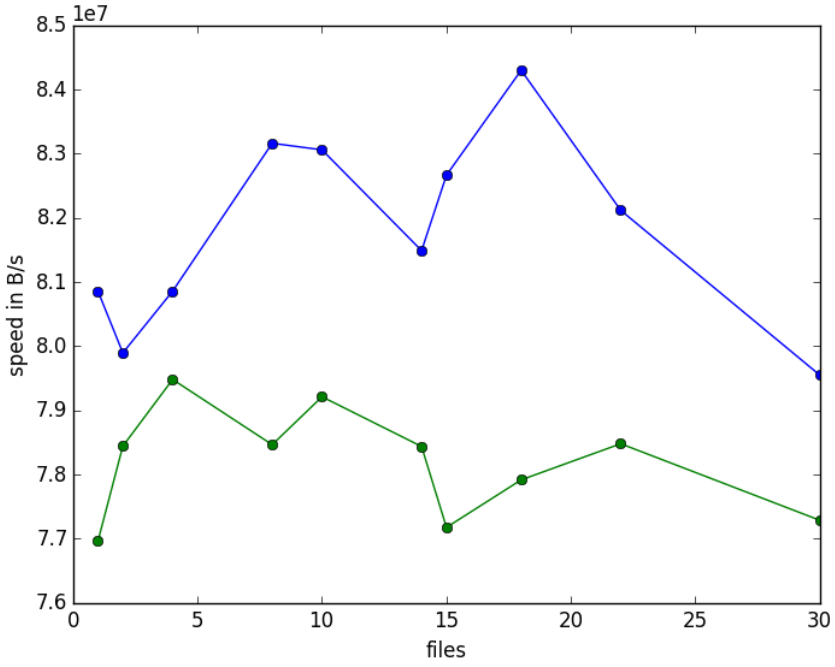
performed on computer2

2.3 Discussion

The first implementation's results are what we expected. Indeed, since it had to make a system call every time it reads or write only one element, we expected to be very slow. As for the second implementation, we did expect to perform a lot better than the first implementation but as we said before, we did not know its own buffer's size. Regarding the third implementation, it was also as expected, but we were quite surprised to see that the bigger the buffer, the more time it takes in user mode. As for the last one, we absolutely did not expect to behave this way. We thought that if B was bigger, it would obviously result in a faster way but the result did not seem to vary very much.

It could be interesting —especially because this implementation have results very close of those of the best implementation, and is more "statble" w.r.t its parameters— to try to improve its performances with the use of the madvise system call provided to tell to the kernel how the mapping will be used. This way the kernel may adapts its read-ahead and caching techniques. Unfortunately, we did not have time to realize such tests.

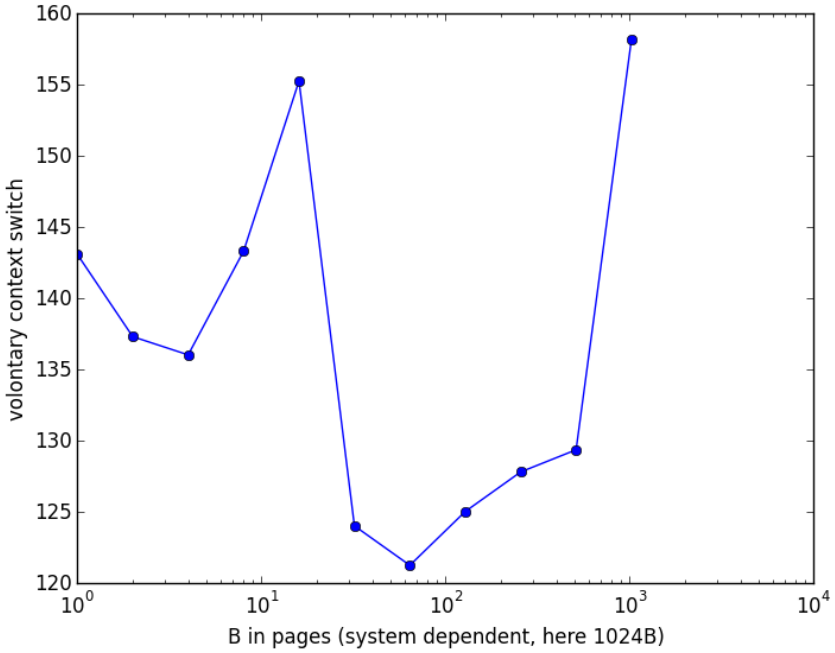
The following figure show, depending on the number of files, the speed of the output streams 3 and 4 with the best parameters: a buffer of 1024 integers for the streams 3 (in blue) and the average over 4 to 128 pages of mapping for the stream 4, as we can not determine a best value (in green).



Output Stream 3 (blue) and 4 (green)

performed on computer2

Another thing that could be interesting to analyze is the number of volontay context-switch performed by the output stream 4. This number seems to depend on the modulo of B on an other value (which one?) as we can see on the following graph:



Output Stream 4

performed on computer2

For the merge sort algorithm, we will use the third implementation with $B = 1024$ for the input stream and the same for the output stream as it seems optimal regarding the input and output streams.

Chapter 3

Merge Sort

The whole point of this project is to implement an external memory multi-way merge sort algorithm that will sort 32-bit integers.

3.1 Expected behavior

The merge-sort will first read the input file by chunks of M 32-bit integers and store them in a min-heap (that is a particular priority queue that keep the smallest element first and provide, constant time complexity to the min element, and, in our case, $O(\log n)$ complexity (amortized) for insertion and deletion). Each time one chunk is completely read, it is written on disk in a temporary file sorted in increasing order (thanks to the heap). The first written file is referenced by the number 1, the second 2... Theses numbers are stored in a queue (FIFO data-structure).

Then the d first files referenced in the queue are merged in a single one whose the reference is pushed at the end of the queue. The result We repeat the last step until we only have one stream. The first merge is referenced by the number 0, the next will reuse the number 1 as it does not reference any file anymore...

We expect it to be slow with non-optimal parameters, but we do expect great performance when the input file is big enough with optimal parameter. The worst case scenario should be when we have a big file with non-optimal parameter. Even with small file, it shouldn't be efficient if we have to create many stream for a small M.
Cost formula: $\frac{\lceil \frac{N}{M} \rceil}{d} * \lceil \frac{N}{M} \rceil * M$

3.2 Experimental observations

The benchmark is done with a single input file of 16 MB. We tried various M with various d. These results have been done by the computer1.
For a $M = 2^{18}$:

Time - d	**2**	**4**	**8**	**16**
user	3.54s	3.187s	2.86s	2.87s
sys	0.02s	0.01s	0s	0.03s
real	3.57s	3.18s	2.87s	2.91s

For a $M = 2^{19}$:

Time - d	**2**	**4**	**8**	**16**
user	3.14s	2.83s	2.82s	/
sys	0s	0s	0s	/
real	3.15s	2.85s	2.83s	/

Note: it is not possible to have more than 8 streams.
For a $M = 2^{20}$:

Time - d	**2**	**4**	**8**	**16**
user	2.72s	2.67s	/	/
sys	0s	0.01s	/	/
real	2.72s	2.69s	/	/

Note: it is not possible to have more than 4 streams.
For a $M = 2^{21}$:

Time - d	**2**	**4**	**8**	**16**
user	2.35s	/	/	/
sys	0s	/	/	/
real	2.36s	/	/	/

Note: it is not possible to have more than 2 streams.
For a $M = 2^{22}$:

Time - d	**1**
user	2.34s
sys	0s
real	2.34s

Note that for the last M, it can be considered as a simple heapsort since $M \geq N$ meaning that we read the input file and sort it without creating d streams.

As we can see from these results, the bigger the M, the faster it is. Also, the bigger de d, the faster it is.

3.3 Discussion

The merge sort algorithm combine the performance of our input and output streams. We did expect to be faster with bigger values. The only thing we did not expect was that the last test (which is only a simple heapsort) was equal to the test with $M = 2^{21}$. But it is not surprising since we don't have more I/O operation than for other tests. The performance is based on the input and output stream performances with a simple sort.

Chapter 4

Execution

In order to run the project, here are some command but first we have to be at the src/ directory.

4.1 Using the makefile

If the required dependencies are installed, run the make command should be a convenient way to compile, test and performs measurement on the project.

By default, make correspond to the follwing sequence:

1. make test-streams
2. make test-merge
3. make test-merge-sort
4. make benchmark-streams
5. make benchmark-mergesort

Theses steps are independents and may be run in any order. However, for brevity's sake, we will describe these steps as if they run in this order.

4.1.1 test-streams

Basically, if make test-streams is the first command run, this should compile the streams and the merge and merge-sort algorithms to object files .o, compile the binary tests/tests and generate many files in the tests/ directory.

After that, tests/tests streams is run. This generate various files from other files or from hard-coded and check that all the implementations of the streams performs well the required operations.

4.1.2 test-merge

make test-merge will produce some other files in tests/tests and run tests/tests merge to check if the merge of many sorted files really produce a sorted file that contains all the elements of the files to merge.

4.1.3 test-merge

Once again make test-merge-sort may produce some files in tests/. But this one will also produce the mergesort binary and will run it and check the result thanks to hard-coded data.

Now we can use the mergesort binary to sort any file thanks to the following command: `./mergesort <filepath> <M> <d>`, where M and d are described above.

4.1.4 benchmark-streams

This will create a sub-directory in benchmark/ to store some pieces of information on the current configuration as well as the result of the benchmarks.

To automatically collect this information, there will be a call to sudo, so sudo need to be installed and you should be a "sudoer". Otherwise you may write the config.txt file manually with the information that seems relevant to you. With the automated method, you will be required to enter your disk name, likely "sda".

This done, make will perform some measurements on streams executions. First streams is compiled then measurements are is done by running the benchmark-streams.sh bash script (you can edit it to choose what to test) and measurements are realized thanks to the gnu's time. The result are written in (appended to) streams.csv in your sub-directory. To have good measure the script should be run as root, as this, it can clear the cache when needed.

The fields of this CSV file are: the command run, the CPU time elapsed in user mode, the CPU time elapsed in kernel mode, the total elapsed time (wall clock), the number of minor page fault, the number of major page faults, the number of times that the program was context-switched voluntarily, an estimation on the number of input, an estimation on the number of output.

4.1.5 benchmark-mergesort

Basically, make benchmark-mergesort run the benchmark-mergesort.sh bash script. This script run the mergesort binary (should already be compiled at the test-merge-sort step), on some randomly generated file with various parameter for M and d, measure the performances like above and store the result in a merge.csv file in the same sub-directory as stated above.

To complete the benchmark with further tests, one can re-run these two make commands with the `-B` flag or run `./benchmark-mergesort.sh benchmark/<your_sub_directory>` if possible as super-user (were <your_sub_directory> is the same as stated above and should be your user name followed by an underscore followed by the host name of the machine.

Chapter 5

Conclusion

While doing this project, we learned many things. At first we did not entirely know how the input and output streams worked but by implementing the different streams, we've discovered how it actually worked and how faster it can be by testing it. We only implemented 4 ways but maybe there exists more. We did try to have a really stable code by doing inheritance, abstract class, template, C arrays, etc... but inevitably, that's where we probably spent more time.
As for the merge sort, it was not that hard to implement since we know how it worked. The only interesting thing was that we obviously had to use what we implemented for the streams and needed to figure which values were good enough.