Algorithm Engineering

Markus Andreassen - 201304194
Vinh Cong Nguyen - 201304216

Aarhus Universitet

15-03-2016

Algorithm Engineering
Markus Andreassen - 201304194
Vinh Cong Nguyen - 201304216

Introduction

This report consists of three projects, where we tackle a specific problem, and try to optimize and see pros and cons of using different algorithms. The computer used in all of these tests has the following core, which is the piece of hardware mainly utilised in the course of the projects. [1]

> Intel® Core™ i7-3630QM-processor 2.4 GHz.
> L1 cache: 256 KB
> L2 cache: 1024 KB
> L3 cache: 6144 KB

Our code can be seen at https://github.com/Borimino/dAlgoEngi.

Project 1 - Binary Search

The aim of this project is to thoroughly experiment with different forms of binary search algorithms to see differences, advantages and disadvantages brought by implementing said solutions. The data structures chosen should store a set S containing n integers and should all support the query:

> Pred(x) = return max{y ∈ S | y ≤ x}

Our first solution is a naive binary search algorithm, where we have a set S containing a large amount of integers. The set is contained within an array. We then utilise a standard binary search algorithm to support the Pred(x) query. The naive solution makes use of the general principles of the binary search algorithm, splitting the array in halves and comparing with the middle number to identify what half of numbers it must work with. The algorithm is implemented using a recursive approach.

Running a performance test using the linux program *perf*, we clearly see that a place where we can try to optimize, is regarding the cache. The following is a *perf*-test of our first solution, with S set to 2^{28}-1, and running the program with 100000 randomized input x.

```
598.520396      task-clock (msec)       #       0.991 CPUs utilized
85              context-switches                #       0.142 K/sec
2               cpu-migrations          #       0.003 K/sec
1,125           page-faults             #       0.002 M/sec
1,678,650,789   cycles                  #       2.805 GHz               [56.74%]
2,031,528,893   instructions            #       1.21  insns per cycle
405,724,462     branches                        #       677.879 M/sec
[72.14%]
1,875,470       branch-misses           #       0.46% of all branches   [72.18%]
27,976,735      cache-references
26,394,169      cache-misses            #       94.343 % of cache refs
```

Figure 1 - Naive binary search performance test

The second solution considered is an implementation of a BFS tree, where we try to utilise the general structure of the tree to optimize by reducing the amount of cache-faults occurring. Given the fact that cache misses are very heavy on computation time, since latency of RAM memory is usually several factors greater compared to cache memory. Running the same test with our second solution, we get the following data:

```
2910.893948     task-clock (msec)       #       1.000 CPUs utilized
56              context-switches                #       0.019 K/sec
4               cpu-migrations          #       0.001 K/sec
308,687         page-faults             #       0.106 M/sec
9,269,908,368   cycles                  #       3.185 GHz               [57.10%]
19,431,772,488  instructions            #       2.10  per cycle
3,548,621,152   branches                        #       1219.083 M/sec
[71.41%]
6,836,310       branch-misses           #       0.19% of all branches   [71.52%]
113,775,819     cache-references
82,003,291      cache-misses            #       72.074 % of cache refs
```

Figure 2 - Breadth first tree binary search performance test

Although we have decreased the amount of cache misses by approximately 22%, the amount of instructions run has been scaled by a factor of 10. The following plot shows the two perf tests, where the y-axis is the time taken measured in ms, and the x-axis is the number of the test.

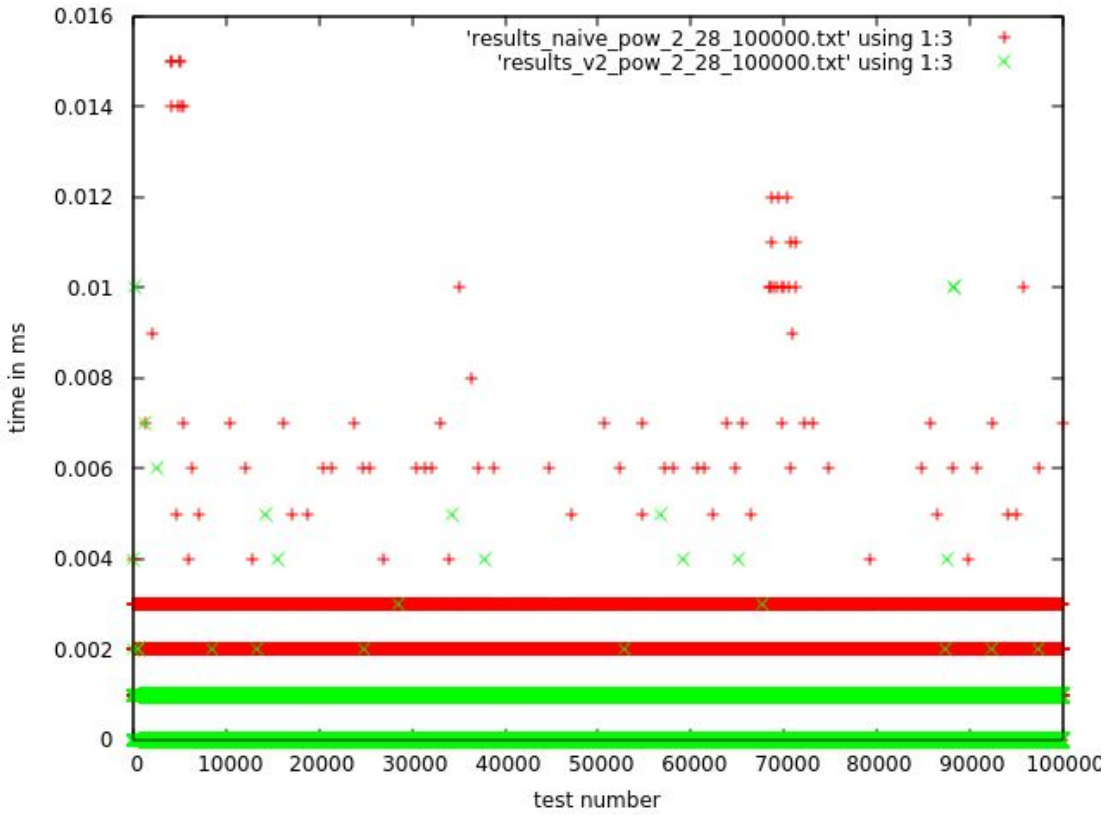


Figure 3 - Binary search tests. Naive vs BFS structure

We clearly see, that the binary search algorithm utilising the structure of a BFS tree is faster than the naive algorithm using a recursive approach, regardless of the fact that the total amount of instructions have been multiplied by a factor 10. This is due to the fact that the performance measurement in figure 2 includes our implementation of a BFS-tree initialization. To illustrate, the following figure shows the amount of instructions of respectively the initialization process and the search process of both algorithms.

```
Naive recursive approach:
         2,049,260,965 instructions     #       includes both initialization and
                                                searching
         1,367,822,980 instructions     #       initialization (66,7%)
           681,437,985 instructions     #       searching (33,3%)

BFS approach:
        19,431,772,488 instructions     #       includes both initialization and
                                                searching
        18,478,310,858 instructions     #       initialization (95,1%)
           953,461,630 instructions     #       searching (4,9%)
```

Figure 4 - Binary search. Amount of instructions pr. algorithm

As we can see, the percentage of instructions which includes the searching itself amounts to 4,9% of the program for our BFS-approach algorithm, compared to the naive recursive one, which is 33,3%. Using our implementations, it would be profitable to use the BFS-approach if we have a large amount of data which doesn't change often, as the searches are approximately twice as fast as the ones of the naive recursive approach, as illustrated in figure 3. However, if we often change data set, the naive recursive approach would be more effective.

Another test considered is the a test regarding the sizes of the data set. Here we run tests with data set S of sizes from 2^1-1 to 2^{28}-1. With each size of dataset containing ints, we search for a randomized integer x 100 times. The following plot illustrates our findings.

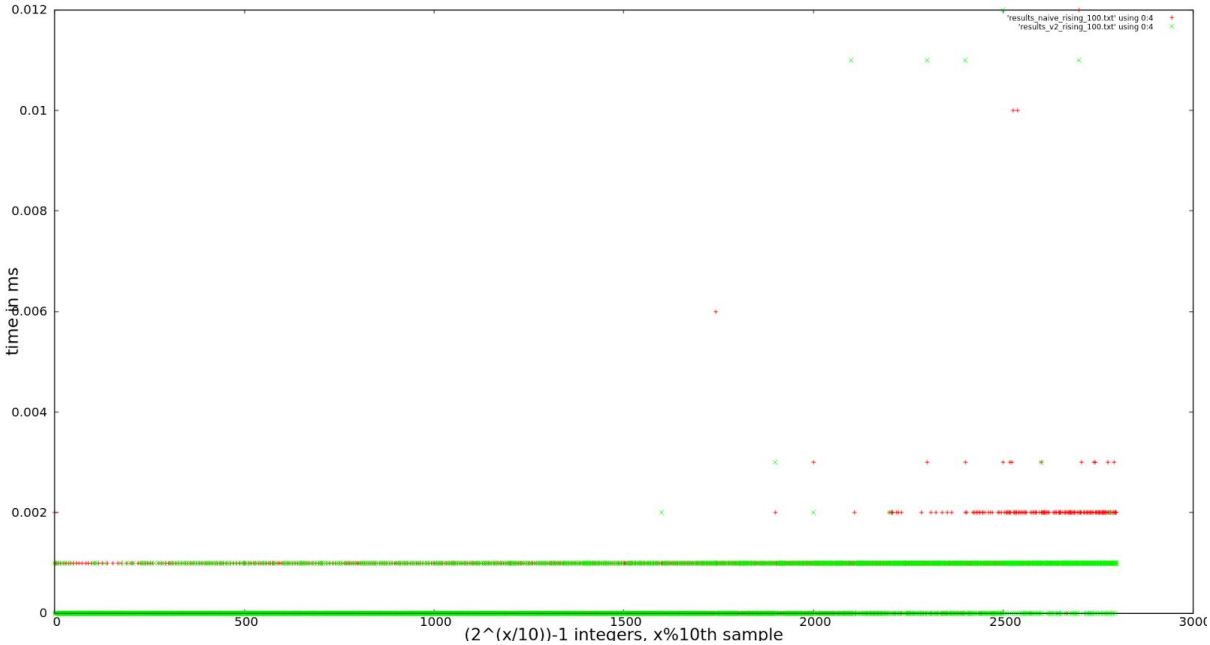


Figure 5 - Binary Search Rising Test Results

The y-axis consists of time measured in ms, and the x-axis contains a specific test. It is given by the power function currently tested multiplied with 100 with an addition of the test number. As an example; 1003 is the a test for power function 2^{10}-1, and it's the 3rd test in this series. The green dots depict the performance of our naive solution, the red dots depicts our solution utilising the BFS tree structure.

As illustrated by the plot, in the smaller data sets (<2^{21}-1), there is no significant difference between the time it takes to search for an integer using the two different algorithms. However, when the data set increases even further, we see an advantage of using the BFS-based algorithm. This might be because of the fact that the smaller sets of

data will not incur cache misses, as we see the larger sets of data does (figure 1), since the smaller sets of data can more easily be contained within the cache memory. This explanation also makes sense in regard to the specs of the computer, as we have 6 mb of cache memory available. At a dataset size of 2^{20} ints, we have approximately 4 mb of data. This can still be contained within the cache. When we go to the next level, we get 2^{21} bytes, which is approximately 8 mb of data. This means that the whole data set can no longer be contained within the cache memory, and this can result in cache misses when searching for specific integers within the dataset. Increasing the test size further would mean that we encounter even more cache-faults, as increasing the test size would mean decreasing the percentage of the whole dataset being in the cache memory at a given time.

Algorithm Engineering
Markus Andreassen - 201304194
Vinh Cong Nguyen - 201304216

Project 2 - Matrix Multiplication

The aim of this project is to explore the uses of optimizing the use of cache memory through cache-aware and cache-oblivious approaches.

First and foremost, we have implemented an unoptimized version of a matrix multiplication algorithm. We use an iterative approach by utilizing nested loops to calculate matrix C from two matrixes A and B, where A has dimensions [m,n] and [n,p]. To get a picture of the complexity of the algorithm, we experiment by having different ranges of m, n and p. This has yielded us the following results, shown as a plot of time-complexity and number q = m*n*p, which is the three dimensions multiplied.

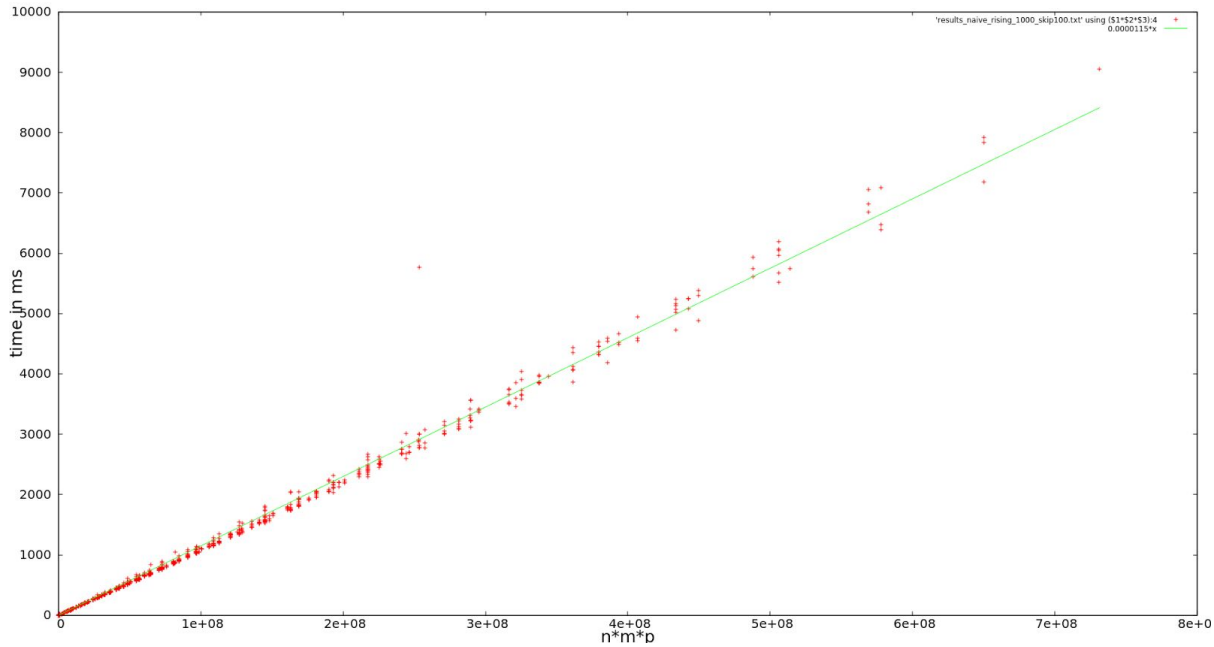


Figure 6 - Naive matrix multiplication algorithm

Here the x-axis contains the range of dimensions n,m and p going from 0 to 1000 multiplied by each other. With each step, we increase either n, m or p up towards the maximum amount, which is 1000. The y-axis contains the time spent in ms to multiply the two matrixes A and B with each other.

The time complexity of our unoptimized algorithm seems to be linear with the three dimensions multiplied, based on our graphs. This is in line with theory, as the expected time complexity is O(mnp). However, this only counts up to a certain point. When reaching sufficiently large values of m, n and p, the matrix multiplication algorithm's time complexity will be heavily influenced by cache-misses.
With the case of a fully associative cache and square matrixes, consisting of M cache lines of b bytes each, we will reach a point where

n will get so large, that every read from the second matrix will incur a cache miss, resulting in, in worst case, $O(n^3)$ cache misses. If we store the matrices in row-major order, the point will occur when $n > \frac{M}{b}$.[2] For the machine we have run tests on, we have the following specs.

Cache size = 6 megabytes.
Cache line size = 64 bytes.
Cache lines = cache size / cache line size = $\frac{6000000}{64} = 93750$ lines

Which means that the point where such behaviour is realized is when n is greater than:

$\frac{M}{b} = \frac{93750}{64} = 1464,85$ bytes

To test this behaviour, we create matrices with large values of m, n and p, and run our matrix multiplication algorithm with these matrices as input.

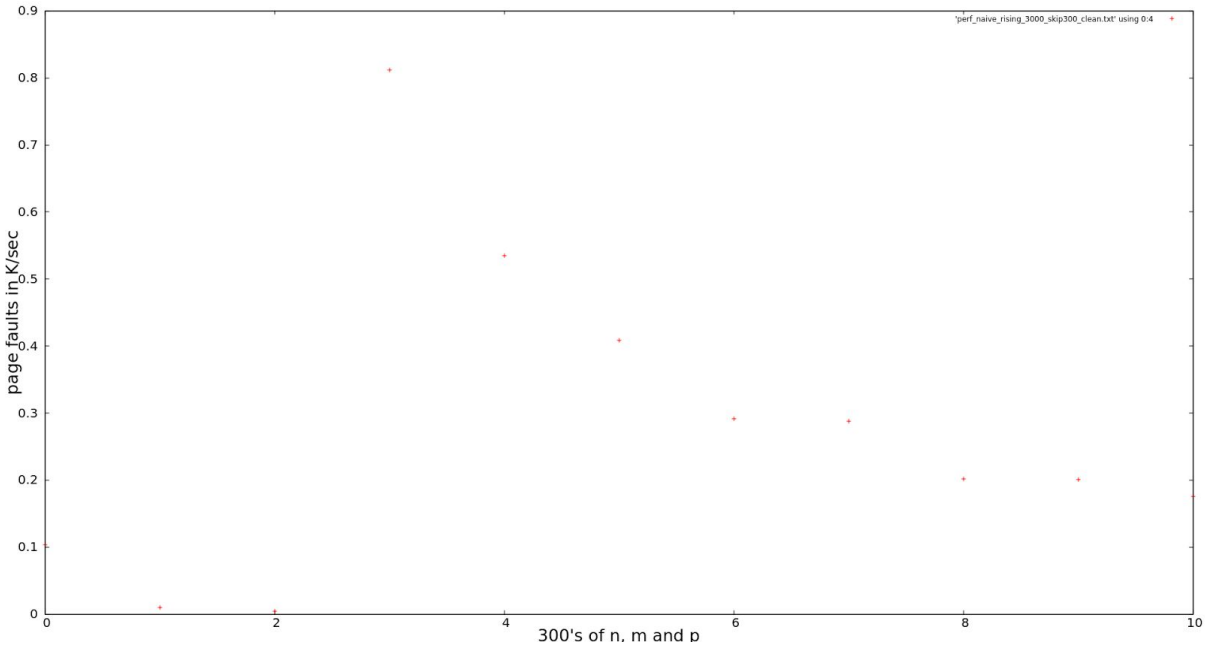


Figure 7 - Number of page faults over time encountered pr. test

The x-axis contains the index of the test. Test 1 uses dimensions 300x300x300, test 2 uses 600x600x600 and so on. The y-axis contains the amount of page-faults over time in K/sec on that specific test. As illustrated by our graph, the amount of page-faults encountered in test 1 and 2 are almost non-existent, whereas on test 3, there is a sudden spike of page-faults encountered. At first glance, this doesn't seem to extend what is stated in theory, as n is only 900 and not greater than 1464,85 in test 3. The spike should have happened in test 5, where the amount of page-faults encountered should have exceeded $O(900^3)$ worst case. There are two reasons for why this happens earlier for our implementation.

First of all, we are working with ints, and as an int is 4 bytes, we are working with a twice as big set of data, as opposed to working with bytes specifically.
Secondly, as we initialize our result matrix, we insert int 0 in all cells of the matrix, which also occupies space in the cache memory.
A calculation shows that we actually already exceed the maximum capacity able to be stored in the cache memory in test 3:

Matrix A = 900^2 cells
Matrix B = 900^2 cells
Matrix C = 900^2 cells
For each cell, we have an int:
($900^2 + 900^2 + 900^2$) * 4 = 9720000 bytes = 9.26 mb.

This means that almost every time we write to the result matrix in test 3, we would encounter a page-fault.

A possible way of handling page-faults in regard to matrix multiplication algorithms, is to utilise cache-oblivious algorithms. A cache-oblivious solution is presented in the paper “Cache Oblivious Algorithms”[3] by Frigo et. all. We have implemented the presented solution, which is a recursive matrix multiplication algorithm for rectangular matrices A and B with dimensions [m,n] and [n,p]. The general strategy is to halve the largest of the three dimensions, and react according to three different cases, until we reach the base case.
case base: If m = n = p = 1. Multiply the two elements with each other.

case 2: If $m \geq \max(n,p)$. Split matrix A horizontally and multiply both halves with matrix B.
case 3: If $n \geq \max(m,p)$. Both matrix A and B is split, and the resulting matrices are multiplied.
case 4: If $p \geq \max(m,n)$. Split matrix B vertically and multiply both halves with matrix A.

We have tried to implement a such solution, but our implementation of a cache-oblivious algorithm is a lot slower than our naive algorithm. It’s time complexity is still O(nmp) as it should be. The following is a plot of the performance of our cache-oblivious solution:

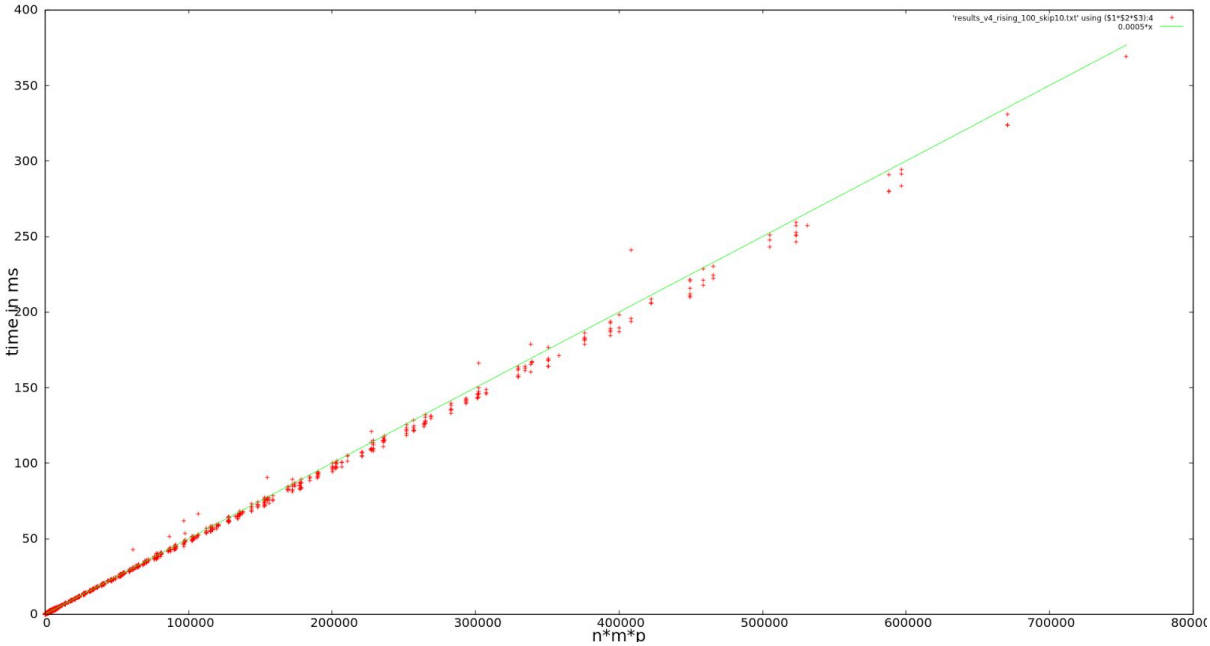


Figure 8 - Performance of cache-oblivious algorithm

As can be seen, the inclination on this is 0,0005x, compared to the naive one with an inclination of 0,0000115x. The naive one is approximately 50 times faster than our cache-oblivious one.
Even when increasing the ranges on the dimensions significantly, the naive algorithms still performs better than the cache-oblivious one, even though it should be heavily influenced by page faults when the ranges get sufficiently large. The following plot contains both algorithms, where we reach the sufficiently high values, which should result in page-faults dominating the time complexity of the naive algorithm.

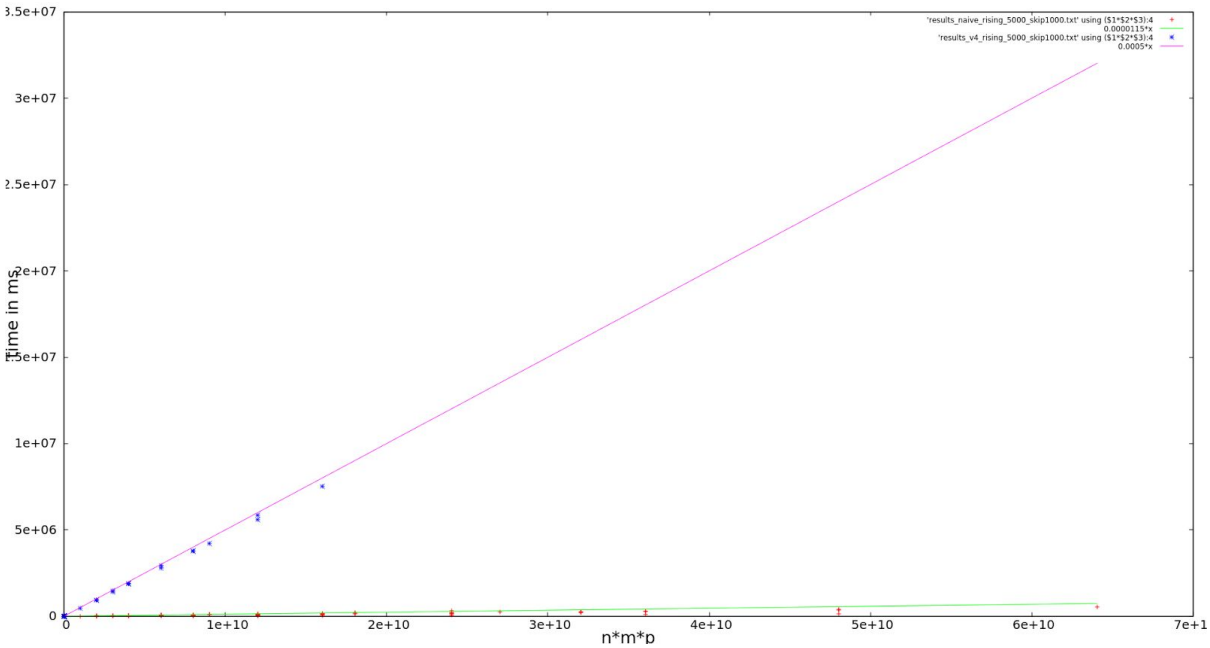


Figure 9 - Naive vs cache-oblivious algorithm

As can be seen on the graph, the time complexity of both algorithms are linear, but the cache-oblivious implementation performs a lot worse compared to the naive one. We could not even finish the same runs as we did with the naive one, as the larger dimensions would result in computation of several days. Although our implementation follows the guidelines explained in the article; “Cache Oblivious Algorithms”, our solution is far from optimal. We still have not been able to find possible reasons for the sub-optimal performance.

Project 3 - Radix sort

The aim of this project is to experiment with the radix sort algorithm in sorting 64-bit longs. In connection with this, we try to optimize the use of cache to speed up the, theoretically already fast, radix sort algorithm.
Our naive implementation of the algorithm consists of three modules and accepts input of N longs with key [0,M), where M is 10 in our regard.
The first module is simply a utility module which has the job of looking for a max value within an array.
The second module executes counting sort, first calculating key frequencies, thereafter determining new positions and lastly moving the elements accordingly.
The third module executes radix sort. First finding the maximum number to determine how many digits are present, and then executing counting sort for on every digit found starting with the least significant digit and ending at the most significant digit.
Using the perf linux tool to measure the performance of our algorithm, we have achieved the following data:

```
148.044494      task-clock (msec)       #       0.345 CPUs utilized
2,964           context-switches                #       0.020 M/sec
10              cpu-migrations          #       0.068 K/sec
147             page-faults             #       0.993 K/sec
250,304,338     cycles                  #       1.691 GHz
335,348,794     instructions            #       1.34  per cycle
71,282,268      branches                        #       481.492 M/sec
232,825         branch-misses           #       0.33% of all branches
301,261         cache-references
46,061          cache-misses            #       15.289 % of cache refs
```

Figure 10 - Perf of naive radix sort algorithm

To try and improve the algorithm, we employed techniques described in the paper: “Engineering a Multi-core Radix Sort”[4] by Wassenberg et. all. They present both a way to implement a radix sort algorithm able to make use of the nowadays oftenly seen multi-core processor, and a way to improve the second module of our algorithm. Namely the counting sort part of the program.
We have only been able to successfully implement the improvements to the counting sort module, which consists of making use of virtual memory to speed the process up.
The described method of achieving a faster counting sort lies in reducing the number of passes through the list of N elements to one instead of three. The way it is done in the article, is by creating a large list, able to store M times all N elements if necessary. The list is ordered by key frequencies, such that the ordering is still done correctly. This approach uses up a lot more storage space compared to the naive approach, specifically O(N*M). This is reduced to

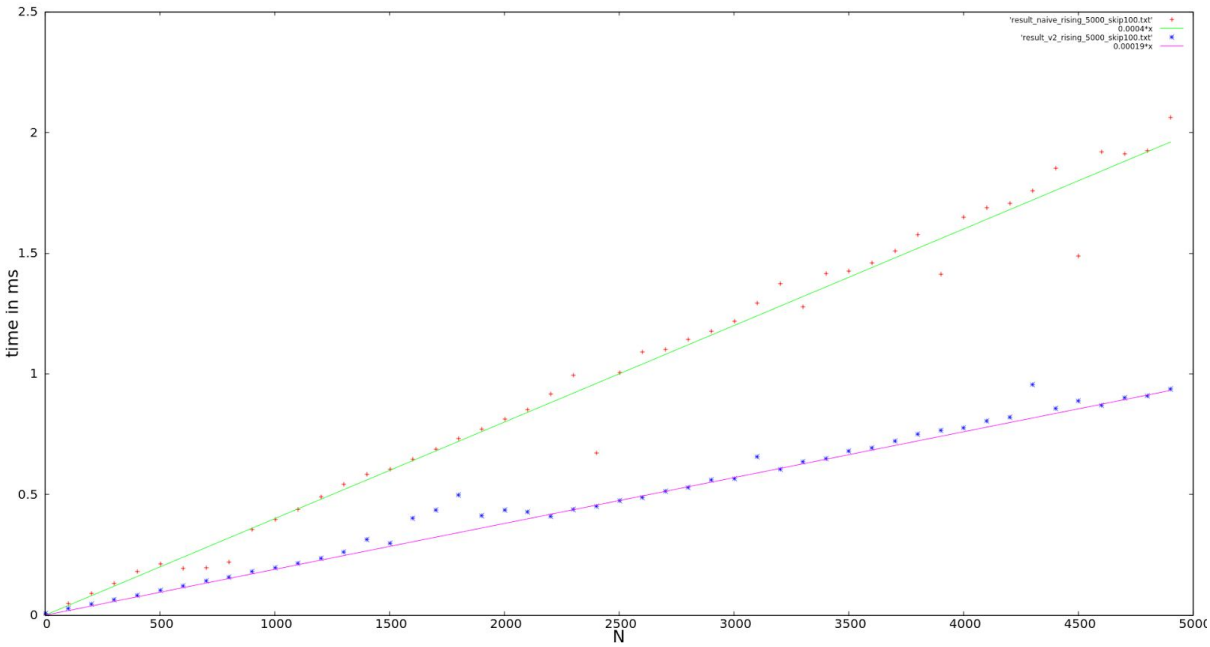
O(N+M*pageSize), as physical memory is only mapped when it is first addressed.

By making use of virtual memory and paging, we can also achieve a lower rate of cache-misses, which in turn improves performance.
Running perf on our improved algorithm yields us following data:

```
128.575247      task-clock (msec)       #       0.288 CPUs utilized
2,958           context-switches                #       0.023 M/sec
6               cpu-migrations          #       0.047 K/sec
797             page-faults             #       0.006 K/sec
215,921,025     cycles                  #       1.679 GHz
330,752,930     instructions            #       1.53  per cycle
73,055,562      branches                        #       568.193 M/sec
189,388         branch-misses           #       0.26% of all branches
1,689,140       cache-references
105,001         cache-misses            #       6.216 % of cache refs
```

Figure 11 - Perf of improved radix sort algorithm

Although it seems like there's more cache-misses compared to the naive algorithm, we make use of the cache more as well in the improved radix sort algorithm, which decreases time complexity of the algorithm, as cache memory is very fast.
The following graph represents both algorithms, with N of range 100 to 5000, skipping 100 at each step.



The x-axis represents N and the y-axis represents the time taken to sort N in ms.

As can be seen, the improved radix sort algorithm is faster than the naive one, which is also in line with what was stated in the paper. As earlier stated, this comes with the drawback of using more storage space.

References

[1]
http://www.notebookcheck.net/Intel-Core-i7-3630QM-Notebook-Processor.
80051.0.html

[2] https://en.wikipedia.org/wiki/Matrix_multiplication_algorithm

[3] Matteo Frigo, Charles E. Leiserson, Harald Prokop, Sridhar
Ramachandran. Cache-Oblivious Algorithms.
ACM Transactions on Algorithms, 8(1), Article No. 4, 2012.

[4] Jan Wassenberg, Peter Sanders. Engineering a Multi-core Radix Sort.
Euro-Par. Lecture Notes in Computer Science, volume 6853, 160-169,
Springer, 2011.