Department of Computer Sience
Aarhus University

Spring, 2017

Algorithm Engineering

Professor: Brodal, Gerth Stølting

Luo, Ji
201511552
luo.ji@post.au.dk

Hvilshøj, Frederik
201206000
fhv@cs.au.dk

Henriksen, Rasmus L.
201303519
rlh@cs.au.dk

Contents

Project 1 - Binary Search

1.1 Introduction

The goal of this project is to analyse the efficiency of binary search under different memory layouts. The layouts being tested are inorder traversal layout (sorted array), preorder traversal layout (depth first searching order), breadth first searching layout and van Emde Boas layout [6].

It is known that van Emde Boas layout is cache oblivious and is an (asymptotically) optimal layout [9]. However, our implementation of binary search in this layout is not as fast as the BFS one. Further investigation revealed that the bottle-neck is the number of instructions. During the first few runnings, there were several anomalies in the data, which suggested interesting details hidden in the assembly code that we shall explain soon.

1.2 Implementation

1.2.1 Framework and overview

The framework consists of benchmarking and layout implementations, which are located in different files and namespaces.

Each layout is implemented in a namespace in the `layouts` namespace. The `layouts::name::build` and `layouts::name::destroy` are responsible for laying out the data from a sorted array and destroying the layout, i.e., reclaiming the memory allocated for the layout. There are several versions of predicates for each layout. Each predicate has the layout and the value `y` as input and returns the pointer to one of the largest `x`'s that is not greater than `y`. Generally, we have implemented stable and unstable predicates for each layout, where "stable" means that the predicate always returns the last element (the one with largest index in the sorted array) not greater than the input value, and where "unstable" means that the predicate returns any largest element satisfying the predicate. More precisely, unstable versions might return early if it encounters an element equal to `y`, while stable versions never return early. For van Emde Boas layout, we have considered more traits: recursive expansion of code and using BFS index as a helper variable. Detailed specification of layouts and predicates is available in the corresponding sections.

The `src/file_handlers.hpp` file contains the implementation of file handlers. In the framework, the layouts are generated and saved for loading later. The data generator is in `datagen/datagen.cpp`. It generates the layouts of $-2n, -2n+2, \ldots, 2n$ and specifies that the queries (used for benchmarking) should be drawn from the range $[-2n-10, 2n+10]$. There is also a test framework built with `Catch` in the `test` folder, which reads the data from `testcase` file and performs tests in the specified query range, assuming the plain binary search (in inorder layout) is correct.

The benchmark is located at `benchmark/benchmark_pcm.cpp`. It reads the parameters of benchmarking from command line arguments, including the layout files, the number of queries to execute and to average among, etc. A stable random source, `std::mt19937`, is used to generate consistent queries for all layouts and predicates, so that the time taken to generate queries and the queries themselves are the same and have good randomness. It is also noticeable that one should not store the queries in a large array as reading the queries could pollute the cache. The benchmark program measures L1, L2, L3 cache misses, instructions executed, branch mispredictions and CPU cycles per query, with Intel Performance Counter Monitor [11]. There is also another version the uses PAPI [10], `benchmark/benchmark_papi.cpp`, which is not used for some reason we will explain later.

1.2.2 Inorder layout

The layout is simply the sorted array itself. Algorithm 2 (in the appendix) describes the stable predicate. For the unstable version, return early if `array[mid]` $=$ `y`.

1.2.3 BFS layout

Traditionally, a binary search tree is embedded into a perfect binary tree, which is then laid out in BFS order in the memory. In this project, we choose the appropriate embedding scheme so that the elements from the input, when laid out in BFS order, appears before the dummy elements. This is equivalent to building a complete binary tree. Algorithm 3 (in the appendix) is the stable version of the predicate.

1.2.4 DFS layout

DFS layout is exactly the preorder layout. Given the inorder layout, there are many possible balanced trees, thus many possible preorder layouts. For an array of size n, we build the tree such that the left subtree consists of the first $\lfloor \frac{n}{2} \rfloor$ elements, and this rule is enforced recursively for each subtree. Algorithm 4 (in the appendix) is the stable version of the predicate.

1.2.5 Van Emde Boas layout

Implicit van Emde Boas layout is easy to implement for perfect binary trees. For an array of arbitrary size, we can build the tree as shown in Figure 1. This layout keeps the depth logarithmic to the number of elements while allowing simple implementation without space overhead.

The predicate is quite complicated and there are many versions of the predicate in the source code. Basically, the internal storage of a tree consists of the "nodes" as the first few consecutive elements, followed by the van Emde Boas layout of each "tree" ("node" and "tree" as in Figure 1). The searching first finds the "tree" of our interest by iterating over the "nodes" and then performs a regular binary search in van Emde Boas layout.

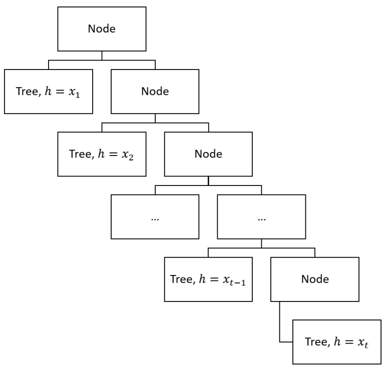


Figure 1: The tree for an array of size $N = 2^{x_1} + \cdots + 2^{x_t}$, where $x_1, ..., x_t$ are distinct natural numbers.

1.2.6 Summary of predicates

For exhaustive description of different versions of predicates we have implemented, see Table 4 in the appendix.

1.3 Experiments

1.3.1 Machine specifications

The experiments were conducted on a private Surface Pro 3. The hardware specifications are shown in Table 1. The computer was not used for anything else when testing. And the experiments have been conducted sequentially. Therefore, disturbance from other processes is expected to be low.

CPU	Processor Intel(R) Core(TM) i7-4650U CPU @ 1.70GHz, 2301 Mhz, 2 Core(s), 4 Logical Processor(s)
Memory	$2 \times$ 4GB DDR3
Cache	L1 $2 \times$ 32KB 8-way associative L2 $2 \times$ 256KB 8-way associative L3 4MB 16-way associative
OS	Windows 10 Version 1607 (Build 14393.953), 64-bit

Table 1: Hardware Specifications

1.4 Discussion

1.4.1 Theoretical analysis and prediction

We might compute the range of expected number of cache misses for each layout in a simplified model. In this model, there are two levels in the memory hierarchy. The low-level contains only one cache line and each cache line contains B consecutive words in the high-level, with certain padding requirement.

Moreover, we assume local variables are stored in the registers. It is noticable this assumption is not true if the searching algorithm requires non-constant additional space. In our implementation, searching in van Emde Boas layout requires $\mathcal{O}(\log \log n)$ additional space. However, asymptotically we can ignore the overhead of cache misses caused by accessing to additional space, which is infinitesimal to the cache misses caused by accessing the tree. In this project, each element is a word, a trivial upper bound for each layout is the number of elements accessed.

For inorder layout, if the array fits in the cache, there will be only constant number of cache misses, which depends on the padding. During the search, the range of the index shrinks by a half in each round. When the range of the index is small enough, it is the same case that the array fits in the cache. Otherwise, each round will create exactly one cache miss. Therefore, the number of cache misses is $\log_2(n/B) + \mathcal{O}(1)$.

For BFS layout, the first few rounds create constant number of cache misses, after which each round creates exactly one cache miss. The number of cache misses is also $\log_2(n/B) + \mathcal{O}(1)$.

For DFS layout, there is one or zero cache misses when one goes to the left subtree, and there is one cache miss when one goes to the right subtree, except for the last few levels. Whether there is a cache miss or not when going to the left subtree depends on the padding. On average, each time one goes to the left subtree there is $1/B$ cache misses. Therefore the expected number of cache misses is $\frac{1}{2}(\log_2(n/B) + \log_2 n/B) + \mathcal{O}(1)$.

For van Emde Boas layout, the expected number of cache misses is $\Theta(\log_B n)$ [6] and is optimal [9].

1.4.2 Choice of PAPI or PCM

At first, the project was compiled on Linux with PAPI for profiling. The results returned by PAPI is eccentric – it reports that searching in a set of size 2^{30} produces about 80 cache misses and this is recurring across different runs. For this reason, the project is migrated to use PCM, which gives reasonable data. Since the development environment is Windows, we also moved to Windows for benchmarking so that one does not have to switch machines for that.

1.4.3 Cycles per query

Practically, the most relevant measurement is cycles per query. As shown in Figure 2, for our data set, BFS layout is always the fastest, and the inlined version of van Emde Boas layout narrowly outperforms DFS layout when the size of the set gets large. The iterative (manual stack) version and the BFS index version of binary search in van Emde Boas layout and binary search in the inorder layout are among the slowest. We expected the BFS index version to be slow, since the conversion from BFS index to van Emde Boas layout index is not trivial. It is also expected that BFS layout and DFS layout are fast, since the searching logic in those layouts are simple, even simple enough to write assembly code instead of C++. However, it is also expected that van Emde Boas layout becomes that fastest when the set gets large enough, which is not observed from the data. We will explain that later.

1.4.4 Cache misses

The benchmark program also produced information about cache misses. Figure 3 gives L2 cache misses per query for all algorithms. Thanks to the clever choice of legends, it can be read directly from the plot that van Emde Boas layout is the most cache-friendly one. It is also obvious that stability makes little difference for cache performance. Reading into details, we find that the theory and the data match well for inorder layout and DFS layout, but the data for BFS layout is quite far from the expected number – it should perform as badly as inorder layout does, but in reality it is even better than DFS layout. [13] sheds hint to us: it might be caused by the prefetcher. An explanation could be that the prefetcher detected the pattern of accessing elements whose index is twice the current one. Another possible story is that due to pipelining, a mispredicted branch helps loading the correct cache line. A third one suggests that the branch predictor can provide hint for prefetching data. We shall check these hypotheses later.

1.4.5 Instructions and mispredictions

The idea of introducing unstable predicates is related to the instruction count and misprediction count. If the predicate is often given an element at shallow levels, it might exploit this to stop early if there is no requirement on the exact element (e.g., must be the last one in the original array) to be found. It is also imagineable that unstable predicates introduce more burden on branch mispredictions. In our setting,

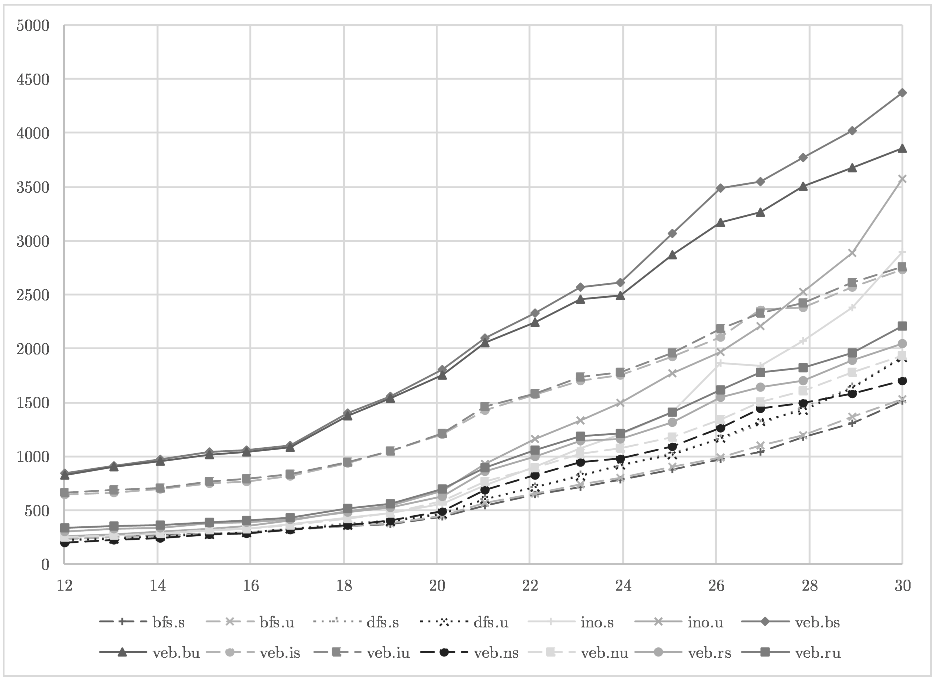


Figure 2: vertical axis: CPU cycles per query; horizontal axis: $\log_2 n$.

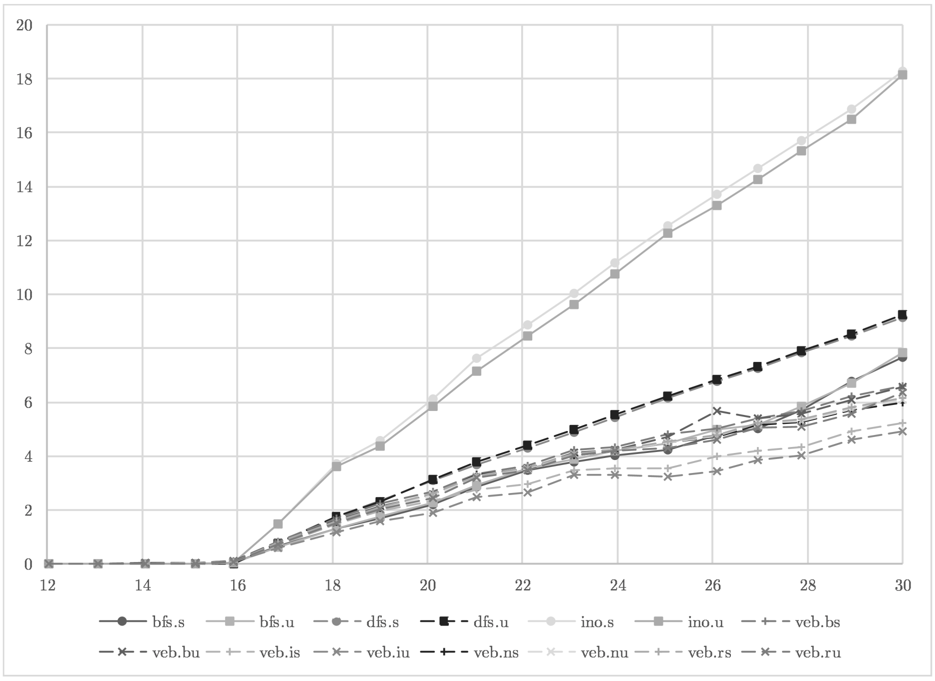


Figure 3: vertical axis: L2 cache misses per query; horizontal axis: $\log_2 n$.

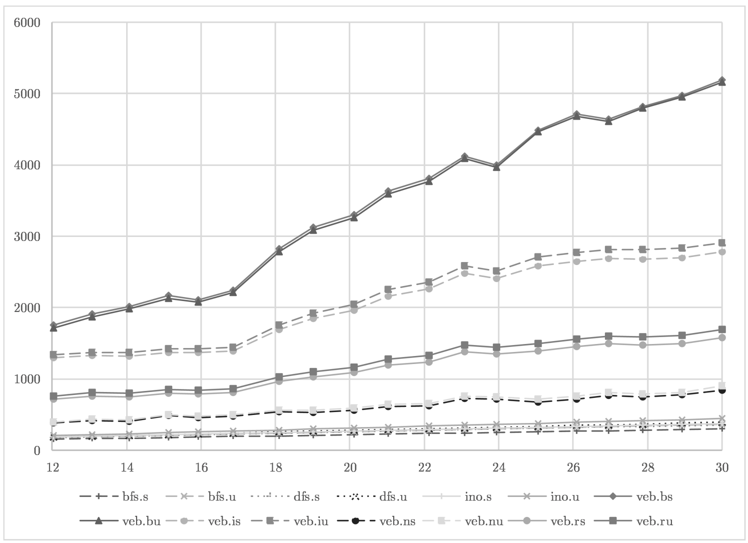
there is $\frac{1}{2}$ probability that the y is not in the set, and the probability that y belongs to k-th deepest level is $\Theta\left(2^{-h}\right)$. Figure 4 and Figure 5 give information of each algorithm on these two performance counters.

With the exception of `veb.b*`, each pair of stable/unstable predicates exhibits the same law that the stable version creates slightly less mispredictions that the unstable version. For `veb.b*`, converting BFS index to van Emde Boas index involves a fairly complicated computation, to which one might owe the extra branch mispredictions.

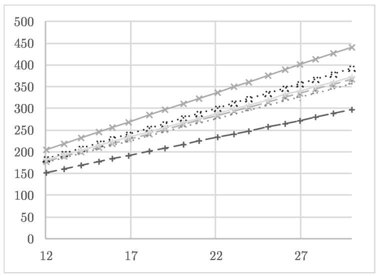
Again with the exception of `veb.b*`, unstable predicates use more instructions per query than their stable correspondent. The expectation of using less instructions with early stopping seems to fall flat. Let's study this problem in details. Suppose we are searching in a perfect tree of distinct elements of height h, with probability $1-p$ the element is not in the set and otherwise the element is chosen randomly. For our practical setting $p=\frac{1}{2}$. Roughly speaking, searching an element of height i requires $b+ki$ instructions with the unstable predicate for some $b>0, k>1$, and the cost of searching an element with the stable predicate requires $b+(k-1)\,h$ instructions. The expected number of instructions for am unstable query is

$$(1-p)\,(b+kh)+\frac{p}{2^{h+1}-1}\sum_{i=1}^{h}2^{i}\,(b+ki)\approx b+kh-k.$$

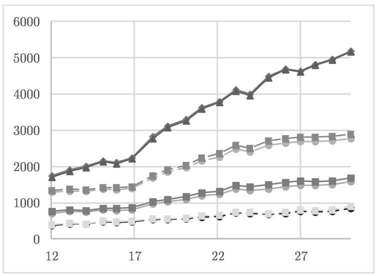
Therefore one benefits from early stopping of the unstable predicate if $h<k$, i.e., only for small h's, only for large k's. The model is not very accurate since as the depth gets larger, the cost of going one level down, k, also gets larger. For example, the cost of arithmetic operations will get larger. In our setting, because BFS index conversion is computationally intensive, using the unstable version can reduce the number of instructions.



(a) all algorithms in one plot.



(b) except van Emde Boas layout.

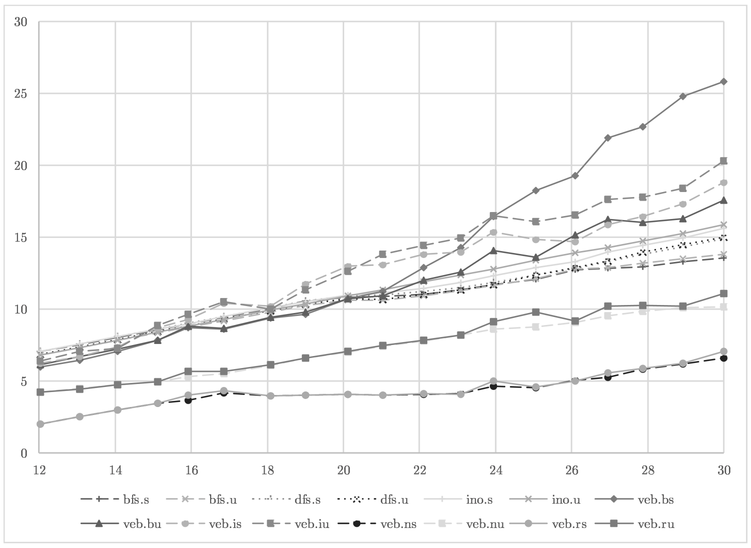


(c) for van Emde Boas layout only.

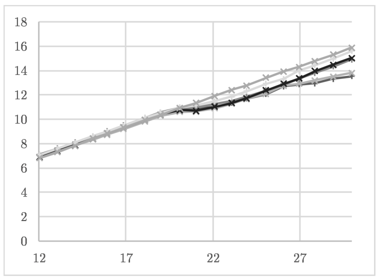
Figure 4: vertical axis: instructions per query; horizontal axis: $\log_2 n$.

1.4.6 Conditional move and prefetch

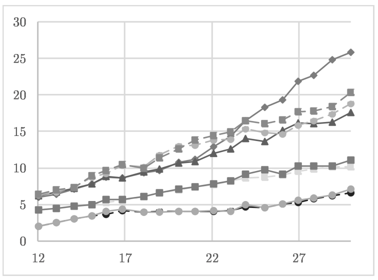
In this section we will explain why BFS layout has far less cache misses than the theory has predicted.



(a) all algorithms in one plot.



(b) except van Emde Boas layout.



(c) for van Emde Boas layout only.

Figure 5: vertical axis: branch mispredictions per query; horizontal axis: $\log_2 n$.

In the early stage of this project, we used GCC as our compiler (when we were still using PAPI). We tried compiling the project with `O2` and `O3`. Despite the obviously wrong number of cache misses, the results drew much attention. Figure 6 shows that the stable versions of BFS and DFS predicates generates little to no branch mispredictions under `O3` optimisation level. It is confirmed by reading the assembly code that GCC rewrites the code with conditional move instructions. The branch misprediction could be caused by the branch jump at the end of the loop. Another interesting phenomenon is that, as shown in Figure 7, for `O3` level of optimisation, BFS stable and unstable predicates diverge drastically. Reading the assembly code again, we find that in `O3`, GCC also rewrites `lea` instructions with (multiple) arithmetic instructions, which is surely one reason.

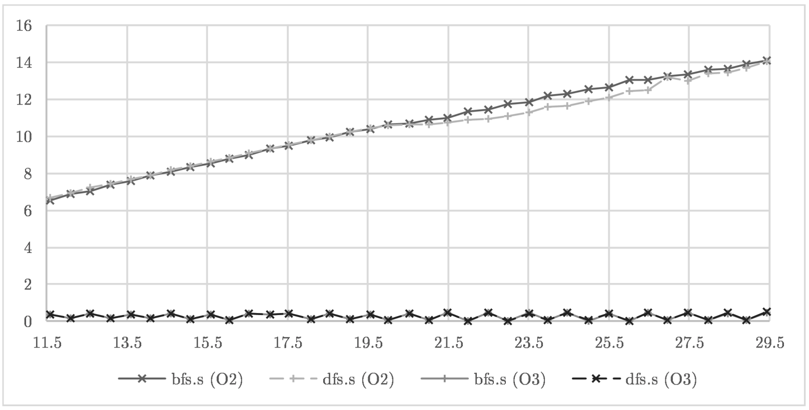


Figure 6: comparing `O2` and `O3`; vertical axis: branch mispredictions per query; horizontal axis: $\log_2 n$.

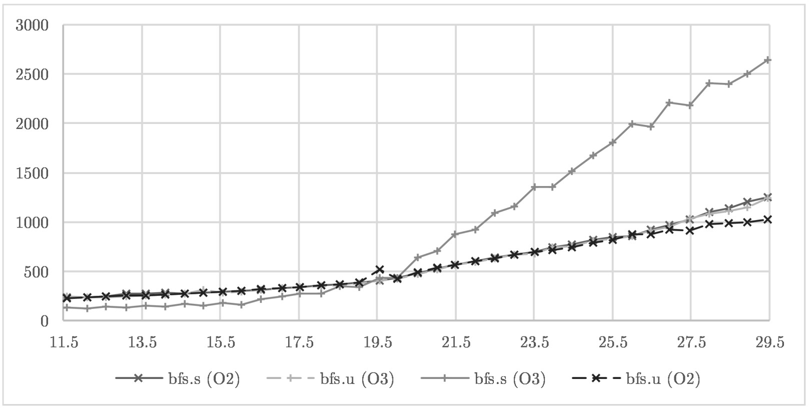


Figure 7: comparing `O2` and `O3`; vertical axis: CPU cycles per query; horizontal axis: $\log_2 n$.

Later, when programming and running under Windows, we did more experiements to investigate why BFS layout caused less cache misses. It turns out that those two problems are related. We implemented more stable BFS predicates, `bfs.cs`, `bfs.ps` and `bfs.cps`. Here, `c` stands for conditional (instructions) and `p` prefetch. Figure 8 shows the key measurements. Some observations:

1. If conditional move replaces branch jump, branch mispredictions cease to happen;
2. If conditional move is used but prefetch is not used, the number of cache misses coincides with theoretical estimation;
3. The `bfs.ps` is almost the same as `bfs.s` in terms of cycles.

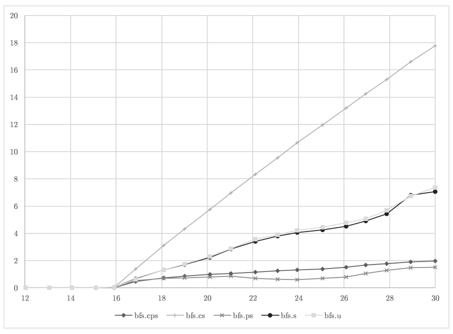
We conclude that the prefetcher does not learn the accessing pattern by itself, otherwise `bfs.cs` would have the same cache misses as `bfs.s`. However, it is possible that the counter does not count cache misses created by a mispredicted execution path, while such a path does help loading the cache. It is also possible that the prefetcher learns the pattern with the help of branch predictor. In either case, there is no point in doing manual prefetching. Also, the conditional move in `O3` code for `bfs.s` contributes to its slowness.

1.4.7 Decomposition of cycle contribution

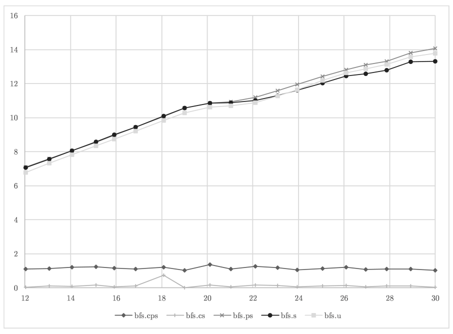
We also did linear regression on the data. We try interpolating CPU cycles per query from other measurements, in the hope that this could provide insight to bottlenecks in the implementation and the layout itself. After several trials, we found that L3 cache misses, number of instructions and number of mispredictions explain the CPU cycles best among the choices of a subset of variables, and the coefficient are sensible. The ratio of penalty (CPU cycles) brought by each factor is about 440 : 1 : 60. We also plotted doughnut charts for each algorithm to indicate the CPU cycles contributed by each selected factor. Each doughnut represents a data set, and the smaller the doughnut is, the smaller the data set is. It is straightforward from the plots that predicates for van Emde Boas layouts are slow because the computation (implementation) is too complicated, and the time are mostly spent on pulling data into cache for BFS layout.

1.5 Conclusion

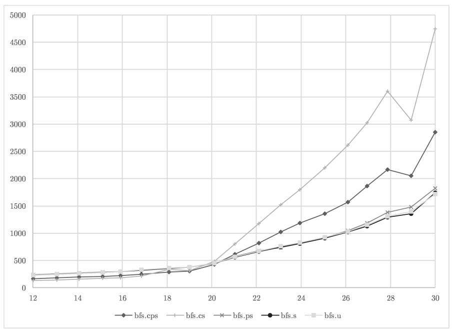
In this project we find that cache accessing time might not be dominant if the implementation is poor. We believe there is better implementation for van Emde Boas layout but we did not manage to implement it. From the investigation into BFS layout predicates, we find that today's processors have complex behaviours, and using slightly different instructions might tweak those behaviours, which deoptimises the program. It is necessary to write code in different ways (e.g., using tenary operators make Visual Studio generate conditional instructions, while equivalent `if` statement does not) and see if one of them outperforms the others, as well as using different optimisation options. It could be the case that one need to specify each single optimisation switch for each file to achieve maximum performance. There is one way to solve that – to write the assembly code, which gives up portability. There is a trade-off between the ease of programming and guaranteed performance, both at the level of selection of data structure and memory layout and at the level of programming language. Finally, it is foreseeable that if disk gets into the storage hierarchy of a binary search tree, van Emde Boas layout will almost surely be the fastest. It is thus, in practice, the best to combine van Emde Boas layout and BFS layout to attain maximum performance, which is a common strategy employed for sorting algorithm among others.



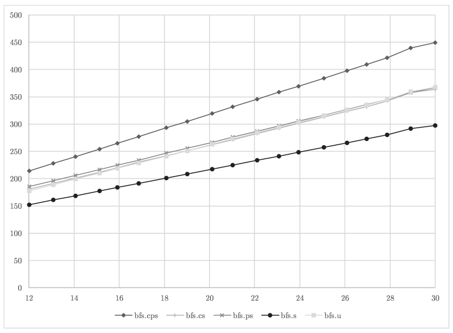
(a) L2 cache misses per query.



(b) branch mispredictions per query.

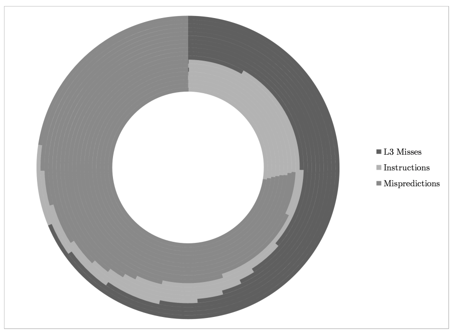


(c) cycles per query.

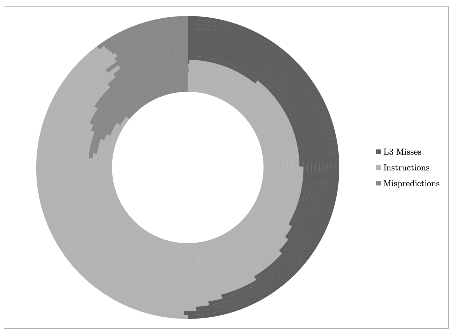


(d) instructions per query.

Figure 8: horizontal axis: $\log_2 n$.



(a) stable predicate for BFS layout.



(b) recursive stable predicate for van Emde Boas layout.

Figure 9: pie chart of contribution of cycles from select measurements.

Project 2 - Improving Quicksort in Java

2.1 Introduction

The goal of this project is to analyse and if possible improve the Dual-Pivot Quicksort implementation, that the standard Java library use to sort arrays with primitive types. The implementation was released in Java 7 [21]. The implementation in question is $Arrays.sort(int[])$. Note that a stable sorting method is not needed when sorting primitive types. The $Collections.sort(List < T >)$ method, which sorts arbitrary objects, relies therefore on mergesort and not quicksort since mergesort it is stable. Which also suggest that mergesort can be a worthy opponent to quicksort.

The rest of the introduction will first give pointers to work and analysis related to a Dual-Pivot Quicksort algorithm that was proposed by Vladimir Yaroslavskiy in 2009 [22]. The algorithm is now implemented as a part of the standard Java library. Then a study of the Java implementation reveals that the algorithm use: insertion sort, quicksort and mergesort, depending on the input size. Based on this, experiments have been conducted to verify some of the results from the related work and to see if the Java implementation can be beaten. The experiments have utilized on the JMH-framework for benchmarking the algorithms against each other.

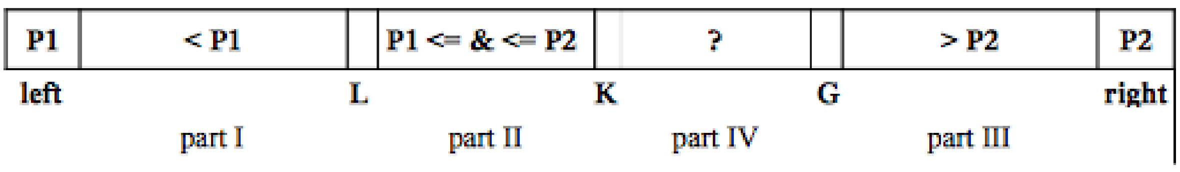
2.1.1 Java implementation

Previously the Java $Arrays.Sort(int[])$ method was based on a highly tuned standard quicksort implementation [4]. Furthermore, previous work have mostly been related to finding a good pivot element for the standard 1-pivot quicksort [8], whereas using two pivot elements have not been studied in depth before.

A Dual-Pivot Quicksort algorithm was proposed by Vladimir Yaroslavskiy in 2009 [22]. Yaroslavskiy stated the algorithm uses same amount of comparison compared standard quicksort [2] but 20% less swaps. This analysis was later improved by [20], showing that the performance gain, roughly is achieved by exploiting knowledge about the different position sets, thereby first making comparisons with a high probability to succeed. Whereas other similar dual-pivot algorithms does not exploit this fact [20]. The Yaroslavskiy-algorithm uses less comparisons ($1.9n \log n$ vs $2.0n \log n$) on average [14, 20]. It is furthermore shown that the algorithm's number of Cache Misses is lower ($1.6\frac{n+1}{B} \log \frac{n+1}{M+2}$ vs $2.0\frac{n+1}{B} \log \frac{n+1}{M+2}$), where M is the size of the cache and B is the size of a cache line compared to standard 1-pivot quicksort. [14].

Today, the Java $Arrays.Sort(int[])$ method is based on the Dual-Pivot Quicksort proposed by Yaroslavskiy. The algorithm, described in Algorithm 1 uses two pivot elements $P1$ and $P2$ and three pointers L, K, and G. The place for the pivot elements are then iteratively found. The algorithm holds three invariants:

- **Part I**: all elements in $(left, L) < P1$.
- **Part II**: $P1 <=$ all elements in $[L, K) <= P2$.
- **Part III**: all elements in $(G, right) > P2$.



Algorithm 1: Dual-Pivot Quicksort [22].

```
1  if $a.length < 17$ then
2      //Use Quicksort
3  $P1 = a[left]; P2 = a[right]$;
4  L, K, and G are initialized;
5  if $P2 < P1$ then
6      $swap(P1, P2)$;
7  while $K \leq G$ do
8      a[K] from part IV is compared to $P1$ and $P2$, and placed in part I, II, or III;
9      L, K, and G are updated accordingly;
10 $swap(P1, L); swap(P2, G)$;
11 Call recursively on part I, II, or III;
```

2.1.2 Java implementation

The Java implementation of Algorithm 1 can be found inside the *DualPivotQuicksort.java* class. Perhaps surprisingly does the implementation rely on several sorting algorithms.

First of all, looking into the constants defined in the top of the class, it is clear that at least three different sorting algorithms are used. When the input size is below 47, the data is sorted with insertion sort. When the input is less than 286 quicksort is preferred over mergesort. Furthermore, mergesort is only ran a constant number of times before the algorithm switch back to quicksort. This is especially the case if the array is nearly sorted.

Dwelving into the code reveals that two different versions of insertion sort is used. One version is used for elements that belongs to the leftmost part of a range. In other words; the first recursive call in the end of the quicksort procedure will pass a *true* flag indicating that is it the leftmost part. Whereas the next recursive step will pass *false*. When *true* is passed to the method; a simple traditional version of insertion sort is used, that should be optimized for the server Virtual Machine. Otherwise, according to the comments in the implementation: a more optimized algorithm called pair insertion sort is used. The variant should be faster in the context of quicksort compared to a traditional implementation of insertion sort.

Looking into the way quicksort is handled, the implementation uses two variants of quicksort. As default, the implementation uses a dual pivot approach. First, five indexes are calculated where e_3 is the midpoint of the part to be sorted. Then the rest indices are calculated as follows:

$$e_1 = e_3 - 2 * \frac{length}{7} \qquad e_2 = e_3 - \frac{length}{7}$$
$$e_4 = e_3 + \frac{length}{7} \qquad e_5 = e_3 + 2 * \frac{length}{7}$$

The five elements are then sorted using a kind of insertion sort, that swaps the elements around within the input array, until they are sorted. Now if the elements are distinct, i.e. $input[e_1] \neq input[e2]$, for e_i, e_{i+1}, then $input[e_2]$ and $input[e_4]$ will be used as pivot elements. Otherwise the algorithm will continue with a single pivot element, which will be $input[e_3]$ - effectively a median of five. The choice of spacing, $length/7$, have been empirically determined to work well on a wide variety of inputs by the developers.

It have been proposed picking the pivot indices: e_1, e_2, e_3, e_4, e_5 asymmetrically, e.g. e_1 and e_3 instead of e_2 and e_4 [21]. An experiment have been made to see if the results from the article is similar. In the experiments section 2.2.7, picking e_1 and e_3 as pivot indices will be denoted $(1, 3)$.

2.2 Experiments

This section introduces the experiments that have been conducted to test the different sorting implementations. First, the test setup is described and what one should be aware of when conduction timing experiments under the JVM. Then what test data have been used and finally the results of the actual experiments.

2.2.1 Pitfalls

When doing experiments in Java, one have to be very careful in order to get reliable results, since a large amount of pitfalls exists that will make benchmark results nonsense. Some of the most common pitsfalls are listed below:

Isolating runs mixing benchmarks within the same JVM run is wrong. Since, given enough runs, the virtual machine speculates that the sort() always runs the same method, and it can generate very efficient machine code. This assumption gets invalidated when the second benchmarks is run, because it runs a different sort() method. The virtual machine will then have to deoptmize the generated code. It will eventually generate efficient code to both sort() methods, but it will be slower than just optimizing for one of them. Similarly, the third benchmark introduces a third implementation, thus its execution gets even slower because Java HotSpot VM generate efficient native code for up to two different types at a call site, and then falls back to a more generic version for additional types.

Dead-code elimination if the return value of the method under benchmark is never used, the call will be removed. This will especially happen if the method has no side effects and have a simple control flow that does not use recursion.

Constant Folding is another common JVM optimization. Calculations which is based on constants will usually return the same result, regardless of how many times the calculation is performed. The JVM can detect this, and replace the instructions for the calculation with the result of the calculation. This can even extended to methods, such that method calls become optimized away if they always return the same constant.

Loop Optimizations it can be tempting to put benchmarking code inside loops, in order to repeat them multiple times. Perhaps also inside benchmark method calls, i.e. when using some testing framework. The caveat is that the JVM is good at optimizing loops, thus one may end up with different results than what was expected. Loops should therefore only be used as a part of the code that should be benchmarked and not the code around that is used to actually measure the code under test.

2.2.2 Testing Framework

A testing framework have been used to avoid these problems and likely more. But one still need to be very carefull, since each framework has it own intended way to be used in order to work correctly.

Initially, an attempt was made to use the MaliJAn framework [19] for testing purposes. Since this could be used to reproduce the results from [21]. The framework does unfortunately seems to rely on some external Mathematical service, which was unavailable when the experiments were conducted. Thus it was not possible to use the framework for this project.

2.2.3 JMH

Another framework - JMH - was found and used instead. JMH is a Java harness for building, running, and analysing nano/milli benchmarks written in Java [15]. A snippet of a benchmark method can be seen on: Listing 1.

```java
@Benchmark
public void benchmarkMethod() {
    // Code to benchmark.
}
```

Listing 1: Snippet of a benchmark method.

The framework allows the user to specify: number warmup and measurement iterations, the duration of those, if an experiment should be forked, different output metrics and much more. To use the framework one have to do special annotations to the class that contains the benchmark code. The source code must then be compiled to a .jar-file which then can be started with special parameters to make only a subset of the tests run, determine location of the output file and much more. The ant-script for these operations can be found through the Appendix A.7.

2.2.4 Machine used

The experiments were conducted on a private Mac computer. The hardware specifications are shown in table 2. The computer was not used for anything else when testing, conduction only one experiment at a time. Thus disturbance from other processes is expected to be low.

CPU	Intel Core i7-4850HQ, 4 cores @ 2,3 GHz
Cache	64 bytes Cache line L1 32KB L2 256KB L3 6MB (Shared)
RAM	16 GB
OS	macOS Sierra, Version 10.12.3

Table 2: Hardware Specifications, from *$ sudo sysctl -a | grep cache.*

2.2.5 Input

The algorithms have been tested on two different kinds on input. Both consisting of integers. The first version is random integers from $1, \dots, n$, where an array with n numbers is initialized, then a shuffle method is used to ensure randomness in the input data. This data is called *random data.* The second version assign a random number between $i - bound/2$ and $i + bound/2$ where i is the index in the array and *bound* is some user defined constant. In this project the bound was 20% of the input length. This data is called *almost sorted data.*

Notice, based on the hardware specifications on Table 2, the amount of elements (integers of 4 bytes) fitting into L1, L2, and L3 Cache are 8.000, 64.000, and 1.500.000 elements respectively.

2.2.6 Interesting experiments

We found two different directions of experiments interesting. It was found interesting to see if the results from: *Engineering Java 7's Dual pivot Quicksort Using MaliJAn* [21] would be similar to experiments conducted by us. Thus a replication of the experiental results from Table 1 in [21] should be made.

Furthermore, we found it interesting to see how close a homemade version of some sorting implementation is the to presumably highly optimized Java implementation. To work towards a solution for this, the following experiments are proposed:

Insertion sort When looking into the Java implementation, it seems reasonable to believe that some variant of insertion sort will be needed to compete against it.

Cutoff points Since it seems to be beneficial switching sorting algorithm at different input sizes, some results guiding towards a constants for when this should happen seems reasonable to make.

Best candidates Experiments with the best candidates against the native Java sorting algorithm.

When completing the above experiments, one would expect to see results that are somewhat in line with the Java Implementation. E.g. that some of the constants are in the same range.

2.2.7 Replicating [21]

In [21] is it proposed to pick the two pivot elements asymmetric, since it could give a faster sorting algorithm. Their results showed a small improvement when picking (1,3) as pivot for most inputs. Using the same benchmarks as the Java core library developers.

From Figure 10 it can be seen that almost the same results are obtained compared to [21]. The major difference seems to be the 2.5% difference between (1,3) and (1,4), where the running time was identically in the original paper. A similar experiment, but constraint to 100 warmpup and measuring rounds was conducted. It showed overall identical results, besides flipping the difference in favour of picking (1,3) as pivot elements. Thus fluctuations in the experiments seems to be the reason for this.

Another reason why the results might be slightly different is due to the setup of the experiments. In [21] the warmup rounds consisted of each algorithm sorting a fixed random list 12.000 times without measuring. This lets the JIT-compiler optimize the code. Then, each algorithm was run on 1.000 random permutations and the average time was reported. On Figure 10 a new random permutation is used at every round, also during warmpup. A solution to this have not been found, since it seems impossible to do with the JMH-framework.

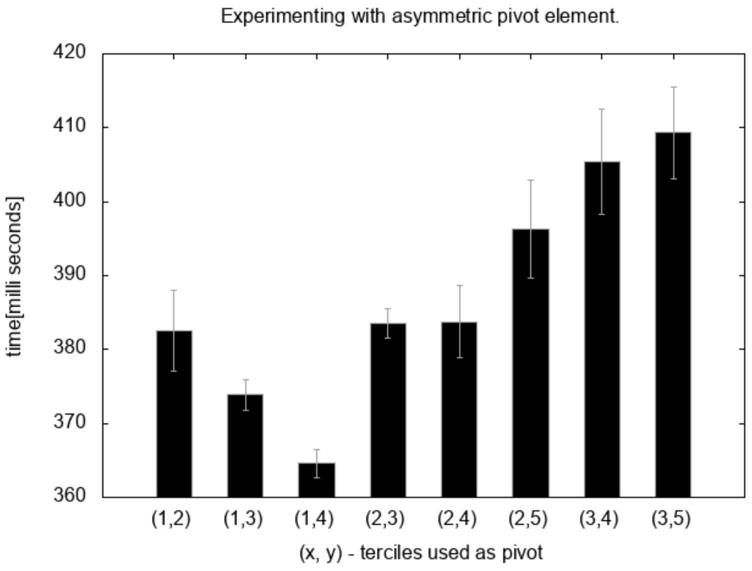


Figure 10: Replicating the experiment from [21]. The algorithm used is described in Appendix A[21]. The input was 4.000.000 randomly distributed elements. Each version was ran with 250 warmup rounds followed by 250 measuring rounds on new random input.

2.2.8 Suitable Insertion sort

Before determining the best cutoff point for the quicksort algorithm, if such a point exists, a good insertion sort candidate needs to be found. Three different implementations were taken into account. The first is taken from [5, 18] and called *Simple* on Figure 19 (Appendix A.2). The second is copied directly from the Java class: *DualPivotQuicksort.java* and is the variant used when the leftmost flag is set to true. The last implementation, called *Optimized* on Figure 19 (Appendix A.2), is shown as Algorithm 5 (Appendix: A.3).

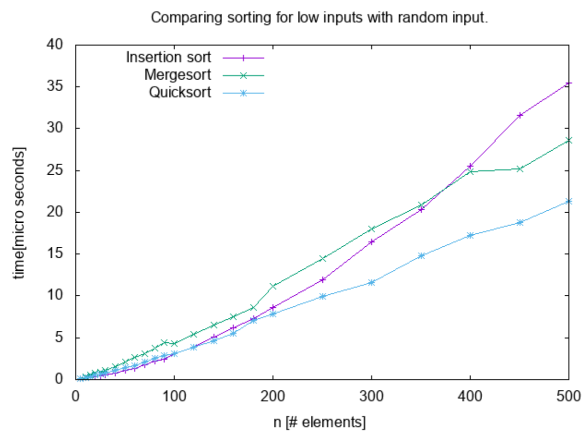
As seen on Figure 19 (appendix) the *Optimized* version clearly outperforms the other implementations, a tendency that also holds true for larger inputs. Though on almost sorted data, Figure 20 (Appendix A.2), the *Optimized* version is only slightly better for smaller input sizes, below 40. Though, since it is expected that it will not be used for larger input, this implementation of insertion sort is used further on.

2.2.9 Cutoff

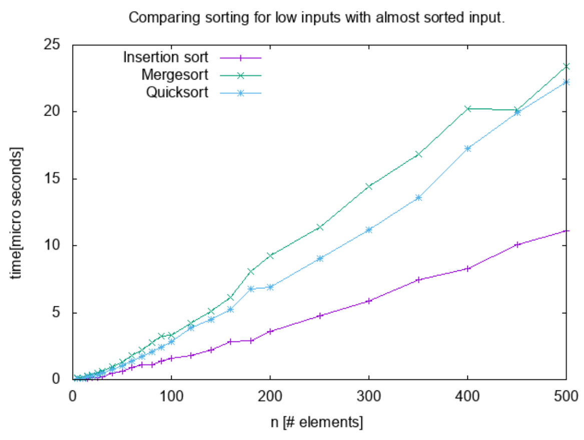
The next experiment shows the difference between insertion sort, quicksort, and mergesort when run on small inputs n, see Figure 11. The difference between the algorithms is small when sorting random data with $n = 200$, though mergesort is consistently slower, see Figure 11a. Insertion sort begins to experience slower running times at this point, $n = 200$, and catch up with merge sort at around $n = 400$. Quicksort is consistently be better compared to mergesort, also when the input size of the data grows, Figure 21 (Appendix A.2).

For the almost sorted data the picture is slightly different. Again mergesort performs worst compared to the two other algorithms, see Figure 11b. Insertion sort have a clear advantage, which continues to about 2.500 input elements, Figure 21 (Appendix A.2).

Thus for some types of small inputs, especially almost sorted data, it could be beneficial to use insertion sort. While for others kinds of inputs the difference seems to be none existing. This seems to fit fine with theory [5, 18], that the best case running time is $\mathcal{O}(n)$ and worst-case is $\mathcal{O}(n^2)$.



(a) Random input data.



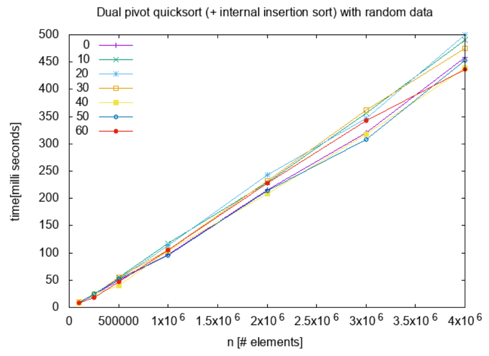
(b) Almost sorted data.

Figure 11: Experiment with small input values. The following algorithms was used: Algorithm 5 (insertion sort), Algorithm 6 (quicksort), and Algorithm 7 (mergesort). Each experiment was ran with 100 warmup rounds followed by 100 measured rounds. New input was generated at every iteration. The difference between each round with same parameters was minimal, thus no error bars have been added to the plot.

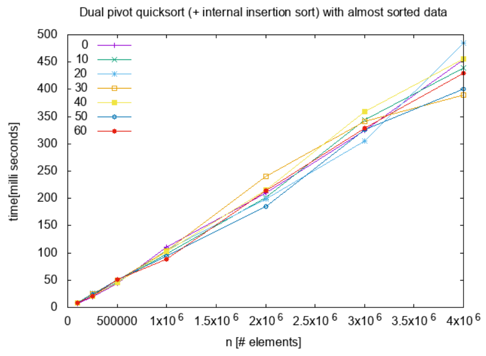
2.2.10 Optimizing Quicksort

The native Java sorting implementation uses, as earlier mentioned, insertion sort when the input is below as certain threshold. An experiment combining insertion sort and quicksort was made to give some estimate of the threshold for the implementation used in this project, Algorithm 6.

The experiments show that a threshold value around 50, seems to give a small performance boost, see Figure 12. The same holds true for smaller inputs, see Figure 23 (Appendix A.2). This is in line with the Java implementation that have its threshold at 47.



(a) Random input data.



(b) Almost sorted data.

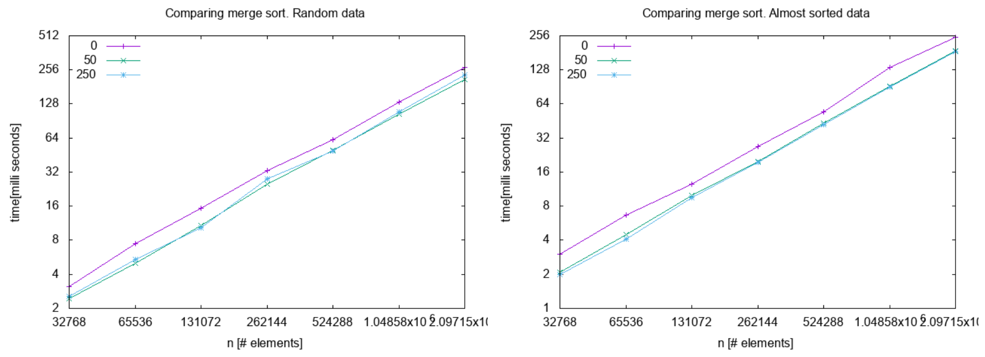
Figure 12: Experiment with different threshold values for when quicksort, Algorithm 6, should change to insertion sort, Algorithm 5. Each experiment was run with 100 warmup rounds followed by 100 measured rounds. New input was generated at every iteration. The difference between each round with same parameters was minimal, thus no error bars have been added to the plot.

2.2.11 Optimizing Mergesort

The native Java sorting implementation also uses a variant of merge sort when the input size is above 286. Thus it would be interesting to see if the same is true for this projects mergesort implementation, Algorithm 7.

The experiments does not reveal that a combination of mergesort and quicksort is an obvious better choice, see Figure 13. Choosing a threshold at either 50 or 250 seems to offer the best results. Setting the threshold at 50, is equal to use insertion sort, and leave quicksort completely out of the picture. Having the threshold at 250 will use a combination of quicksort and insertion sort.

Choosing a threshold of either 50 or 250 does not make much difference in this case, regardless of the data. Thus, to simplify things one would probably stick with a more *clean* version of mergesort and just use insertion sort.

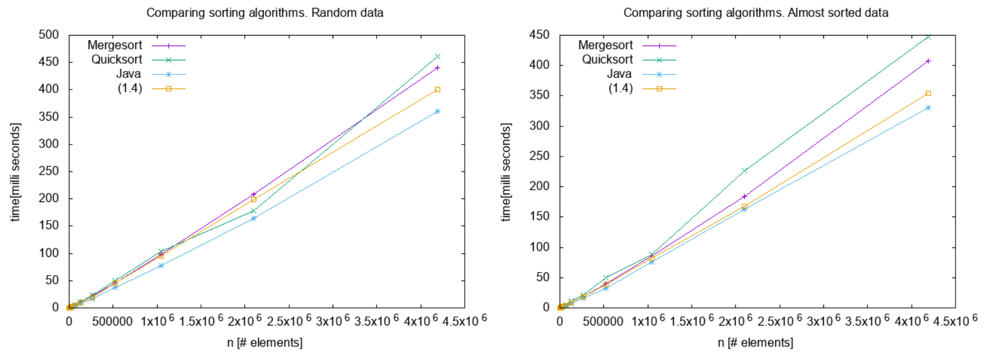


(a) Random input data. (b) Almost sorted data.

Figure 13: Experiment with different threshold values for when mergesort, Algorithm 7, should change to quicksort, Algorithm 6. Each experiment was ran with 100 warmup rounds followed by 100 measured rounds. New input was generated at every iteration. The difference between each round with same parameters was minimal, thus no error bars have been added to the plot. Furthermore, the threshold values: 100, 150, 200, 300, and 350 have been removed to increase clarity.

2.2.12 Comparison against native Java implementation

The last experiment compared three different implementations against the native Java implementation, Namely: (1,3) from [21], the other version of quicksort, and mergesort. The results can be seen on Figure 14. Not surprisingly is the native Java implementation best. The (1,3)-implementation is not that far behind, thus more experiments and better optimization seems very interesting to try out.



(a) Random input data. (b) Almost sorted data.

Figure 14: Experiment with different optimized algorithms. Each experiment was ran with 10 warmup rounds followed by 10 measured rounds. New input was generated at every iteration.

2.3 Discussion

As mentioned in the introduction of this report; the goal of this project was to analyse and if possible improve the Dual-pivot Quicksort implementation in Java. Thus the main focus have been to understand what should be conquered, the native Java sort implementation, but also to study how a reliable test setup should be made in Java, such that reliable results could be obtained if a better solution was made. Actual making and testing these algorithms have therefore been reduced to a less important factor.

Furthermore, did an astonishingly large amount of bugs (two major bummers) go through the testing. They were unfortunately first discovered after running the final test, thus the usefullness of more than a day of CPU-cycles was reduces to the heating of the location where the computer crunched numbers. All this due to a poorly made JUnit-test. This did unfortunately have a small impact on the experiments, since they eventually was run with fewer iterations in order to make it before the deadline.

The asymmetric experiments with the subset of the native Java implementation from [21], could have been explored in further depth. One interesting idea for future work is to copy the full Java implementation and make the changes directly to it, instead of just experimenting with a part of the code. Working with the full code at once, will likely make it harder to control the experiment. Though, should an improvement come to light, altering the code would be very easy. Another interesting thing would be a deeper study of cache misses and alike. Unfortunately did an attempt on a Linux-machine fail. According to the JMH-documentation, Linux is the only operating system with the required profiler, further attempts was not attempted. The overall results of the experiment seems to confirm those from [21].

The MaliJAn and JMH-framework was used as benchmarking frameworks, but only the JMH-framework went up an running. Great time was spend, trying to understand the framework and how to make it work. We believe that it has been set up properly, such the results presented is reliable. One way to check if the assumptions hold, was to make small isolated tests and see whether these gave meaningful results.

The results from all experiments have along with the actual time also returned the standard deviation from each run. It have been insured that the deviation have been kept low. Usually have 100 warmup and 100 measuring rounds have been sufficient to make the deviation below 0.25%.

Future work could also look into a parallel approach. Java did in version 1.8 introduce *Arrays.parallelSort()*, which sorts an array in parallel with at variant of mergesort. If the array length is below $2^{13} = 8.192$ the algorithm will just use *Arrays.sort()* studied in this paper.

2.4 Conclusion

The Dual-Pivot Quicksort algoroithm proposed by Yaroslavskiy have been studied, similarly have the Java implementation of the same algorithm. The JMH-framework have been used to ensure sensible benchmark results. Unfortunately have it only been possible to get timing results, since the profiler for getting cache-misses and alike on Linux did not seem to work. The experiments validated the finding in [21] and showed that the native Java sort implementation, as expected, was more optimized compared to what we was able to make.

Project 3 - Matrix Multiplication

3.1 Introduction

Matrix multiplication is a very fundamental and widely used mathematical operation where speed is often very important when calculated on computers. We seek to develop an efficient algorithm for multiplying matrices together. We consider a cache oblivious approach [7] as well as a loop based approach. The focus is put on cache misses and other hardware counts to improve running times. We find that transposing the second matrix is extremely beneficial. In fact it is the best solution we have come up with. Section 3.2 presents the algorithms used and our general setup. In Section 3.3 experiments and results are shown and finally a discussion of our approach and choices is provided in Section 3.4.

3.2 Implementation

We have implemented this project in C++14. It can be compiled with CMake version 3.6 or higher. Find details about running the program through Section A.7. The implementation contains the following executables that are all described in greater detail in the aforementioned section:

Testing: This executable runs automated tests for all our implementations.

Datagen: This executable generates random data for benchmark programs. It takes four arguments; m, n, p, *output* where the first three are arguments are the sizes of the two matrices to multiply i.e., A is a $m \times n$ matrix and B is a $n \times p$ matrix.

Benchmark: Benchmark program that uses PAPI [10] to measure hardware counts.

Benchmark2: Benchmark program that uses PCM [11] to measure hardware counts.

Our strategy was to implement different algorithms in functions all using the same method signature in order to easily benchmark and find the best algorithm. This made it possible to use function pointers to benchmark the different approaches with the same code for benchmarking.

We have chosen to layout our matrices as one consecutive array of random ints to ensure that scanning through a matrix row by row is laid out consecutively in the memory. This should give us the least cache faults when scanning. The random ints are generated by the `rand` function from `stdlibs` using current timestamp as seed.

The algorithms we have used in this project in order to benchmark and speed up matrix multiplication is a naive approach with three for-loops; a naive approach where the second matrix is first transposed in order to lower cache faults; a pure cache oblivious approach as described by Frigo et al. [7] as well as some mixed approaches using both the cache oblivious strategy and the naive in combination. We have also tried to use parallel computation to speed up computation time but it seems that the bottleneck is data access and not computation. We will get closer into this in Section 3.2. Finaly we have implemented a cache aware tiled approach as described by Frigo et al. [7]. The different algorithms are described in greater detail below. Note that the titles of the sections include coresponding titles in the figures where `T` means that B is first transposed and `x` is the variable size of a threshold. A complete list of variations can be found in Section A.5.

3.2.1 Naive (`Naive`)

The first algorithm for multiplying matrices is to do it using three for-loops iterating over the rows of the first matrix and the columns of the second. The pseudo code for this algorithm can be found in Algorithm 9 in Section A.4. We intended to use this implementation as a baseline for measuring the other algorithms.

3.2.2 Naive – transposed B (`Naive:T`)

The problem about the algorithm above is that it accesses the `B`-matrix in column order which gives a lot of extra cache misses. One easy solution to this problem is to transpose `B` before multiplying A and B. For this to work the indexing of B has to be flipped, ie. `B[k][j]` becomes `B[j][k]`. This solves the problem of not scanning consecutive elements. The complete pseudo code can be found in Algorithm 10 in Section A.4.

3.2.3 Cache oblivious (`Obl`)

We have implemented the solution suggested by Frigo et al. [7] to lower cache faults. This approach is recursively constructed in the following way. For two matrices A and B where A has size $m \times n$ and B has size $n \times p$ the recursion has three cases:

m > n, p: $C = \begin{bmatrix} C_1 \\ C_2 \end{bmatrix} = \begin{bmatrix} A_1 B \\ A_2 B \end{bmatrix}$ where $A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix}$

n > m, p: $C = A_1 B_1 + A_2 B_2$ where $A = [A_1 \ A_2]$ and $B = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}$

p > m, n: $C = [C_1 \ C_2] = [AB_1 \ AB_2]$ where $B = [B_1 \ B_2]$.

The base case of the recursion is when $m = n = p = 1$ where the product of A and B is added to C: `C[0][0] += A[0][0] * B[0][0]`.

3.2.4 Mixed approach (`Obl:x`)

From our initial experiments we discovered that `Naive` was fastest than `Obl` on small inputs. Therefor we decided to combine the two such that the recursion from `Obl` would only continue until the size of the largest dimension of the sub problem was below some threshold x and then the naive approach would take over.

3.2.5 Mixed approach – transpose B (`Obl:T:x`)

As well as the naive approach can benefit from transposing B; so can the mixed approach since it uses the naive algorithm in the bottom of the recursion.

3.2.6 Parallel computations (`Naive:x`)

For both the oblivious and the naive approach we have used OpenMP [16] to try to improve the running times. We used `#pragma omp parallel for` in order to parallelize `Naive`. For the `Obl` we used both a task based approach (`#pragma omp task`) by defining each partial computation in a recursion as a task and then making tasks dependent on each other by waits. We have also tried to divide the cores in the begining of the recursion with `#pragma omp parallel` hoping to gaining speed by the individual L1 and L2 caches on each core. For `Obl` we were not able to produce any valuable results and we will not cover more in this report.

3.2.7 Tiled approach (`Tile:s`)

Frigo et al. also mentions a tiled cache aware approach[7, p.4:3] which we implemented at the very end of the project to see if it could beat the the runningtimes of the other approaches. The algorithm tiles the matrices such that each tile can fit into the cache when processed. For concrete pseudo code see Algorithm 11 in the appendix.

3.3 Experiments

This section covers the experimental setup as well as the experiments conducted for this project. The order of the iterations is more or less chronological.

3.3.1 Experimental setup

The experiments conducted in this project were done on a private Lenovo Carbon X1 computer. The hardware specifications are shown in Table 3. The computer was not used for anything else when testing, conducting only one experiment at a time.

CPU	Intel Core i7-5600U 2 physical cores @ 2,6 GHz, 4 threads
Cache	64 bytes cache line L1 32KB 8 way associative L2 256KB 8 way associative L3 4MB 16 way associative (shared)
Memory	8 GB DDR3L-1600
OS	Linux Mint 17.3 (x64), Linux version 4.5.4-040504-generic

Table 3: Hardware Specifications. Further information can be found through reference [12].

The experiments conducted have been done with square matrices stored in row-major (as described in [7, Fig.2 p.4:3]) of equal size; mostly with sizes between (32×32) and $(10,000 \times 10,000)$. Note that it takes more than a gigabyte to keep three matrices (A, B and C) of size $10,000 \times 10,000$ in memory which is much larger than the L3 cache of size 4Mb.

The algorithms has been measured using both the `ctime` library, Intels Processor Counter Monitor (PCM)[11] and Performance API (PAPI) hardware counters[10]. PAPI supports counting cycles executed (`PAPI_TOT_CYC`) which have been used as a measure for running time through Iteration 1 and Iteration 2 (partially) but this counter was very misleading when measuring parallel executions which is why `ctime` was used for these experiments and PCM's `getTickCount` for the rest.

The reason why PCM and PAPI has been used for hardware counts is that we encountered multiple problems with PAPI. First PAPI only allows three counters at a time. This forced us to run tests multiple times to get all the counts needed. Secondly TLB misses were not supported in our environment. PCM supports both standard events such as L2 and L3 cache misses across all cores as well as a big set of other custom events including TLB misses. PCM also offers individual counts for each physical thread on the core – four counts per event in our case – which makes the counts less polluted by excluding events from the other cores. It should also be mentioned that it can run on both Windows, Linux, OS X and FreeBSD operating systems even though this project did not make use of this flexibility. Because of the change in measuring tools all graphs have a tag in the top right corner indicating with which tool the data was captured.

When measuring counts the multiplications were run twice without any measures to flush the caches before averaging over five iterations of the same multiplication to level out fluctuations. All measures are started just before and stopped just after the matrix multiplication; No initialization or resets were counted. All tests were run with highest priority on a singel core:

```
> sudo taskset -c 0 nice -n -20 ./benchmark ...
```

3.3.2 Iteration 1 – `Naive` and `Obl`

The first iteration concerns `Naive` and `Obl`. The complexity of `Naive` is $O(n^3)$ for computation as well as cache misses. `Obl` covered in Section 3.2.3 has the computational complexity $O(n^3)$ and the cache miss complexity is $O(n + n^2/B + n^3/B\sqrt{M})$[7, p.4:4]. Thus leading us to expect that the oblivious algorithm would be the fastest – especially on bigger inputs. `Naive` was implemented in an iterative manner which for smaller inputs should be faster than the recursive implementation of `Obl` since recursion incurs allocating space on the stack which takes time.

This iteration showed that `Naive` kept outperforming `OBl` – what *Naive* and *Obl* also shows in Figure 15.

Next step was to find a good threshold for `Obl:x`. Figure 15 shows running times for different thresholds chosen to cover a sensible range. We see that `Obl:x` is better than both `Naive` and `Obl` regardless of threshold. We also see that besides `Naive` the running times do follow the theoretical bound $O(n^3)$ since the curves are linear and has no slopes.

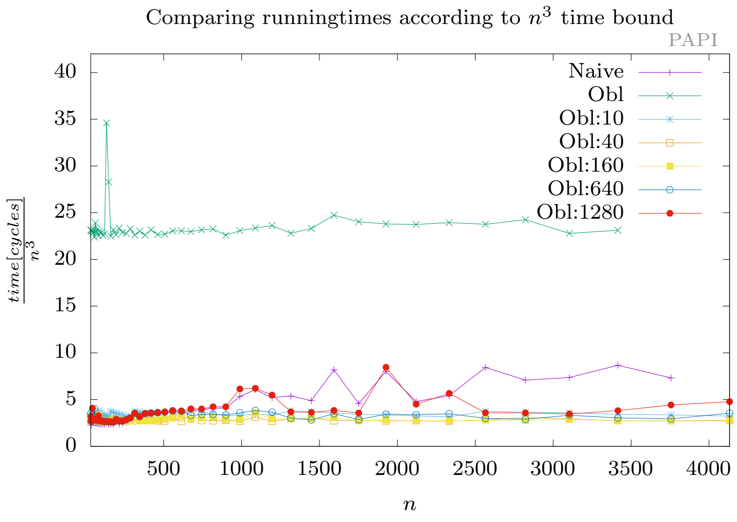


Figure 15: CPU cycles divided by theoretical computation time (n^3). This figure indicates that these algorithms all conforms to their computational bounds of $O(n^3)$ since they behave linearly in the graph.

For the thresholds choosen the performance is quite similar. `Obl:40` and `Obl:160` do seem to perform best though. Figure 26 in the appendix shows further envestigation of thresholds [130, 145, 160, 180] indicating that the performance is close to identical.

We considered the performance of the naive approach close to try to find the reason why it performed so different for different inputs.

First of all `Naive` performs significantly worse than otherwise (see Figure 25 in Section A.6). This can be explained by the associativity of the cache. When the matrices are powers of two then the first cache line of every row of a matrix most likely has the same least significant bits. This forces the cache lines to be associated with the same 8 slots in the L1 cache and the same 8 slots in the L2 cache resulting in cache faults at every single read or write when iterating through columns instead of rows. This trend can be verified by Figure 25 in the appendix which shows when running times increases significantly so do L1 cache and especially L2 cache misses.

Secondly even when the matrices do not have sizes that are powers of two, the naive algorithm will not be able to exploit all the data loaded in the cache lines when iterating through columns instead of rows. This is also indicated in Figure 28 in Section A.6 showing that as soon as the problem size exceeds L2 cache size (3 matrices of size 147 has appx size 256kb $\frac{147^2 * 4b * 3}{1024} \approx 256Kb$) the naive always has the most cache misses.

3.3.3 Iteration 2 – Parallelizing computations

The next thing we tried to do was to use parallelism to speed up the multiplications using OpenMP [16] to split the computations on multiple cores.

First we used `#pragma omp parallel for` in the naive algorithm and then benchmarked different numbers of cores. At first the results just seemed to get better and better the more cores used until we found out that the PAPI framework counts events in a way that is highly misleading when using multiple cores. An example of our misleading results is shown in Figure 29 in the appendix. Also measures such as branch mispredictions and cache misses were cut in half every time threads were doubled. When measuring the running time with `ctime` it gave the results shown in Figure 16 indicating that multiple threads did not improve performance for `Naive`.

The performance issues of this attempt could be caused by the data accesses. It seems like the bottleneck is the data access and not the computation since we gain no performance improvements from threading. Further more the architecture of the machine on which the tests were conducted have two cores but OpenMP acts like there are 4 because of the two physical threads on each core. If we run two physical threads but they run on the same core then we will gain nothing but conflicts when accessing data and using the same cache from different threads.

We did nothing to control which exact core the threads should run on when conducting these tests. So if we should have followed this path even further then we should have done so.

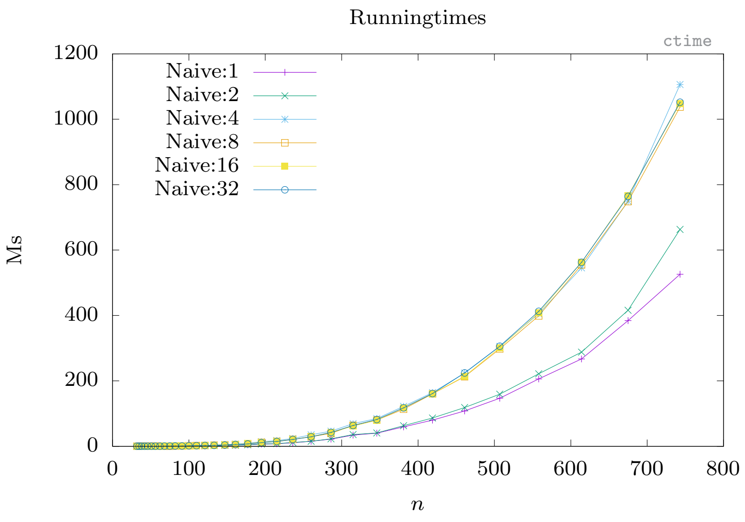


Figure 16: Runningtimes measured with `ctime` for a multi threaded version of the naive algorithm indicating that multiple threads does not improve performance of `Naive:T`

3.3.4 Iteration 3 – Transposing matrices

From the analysis of Iteration 1 an obvious step was transposing the second matrix (B) before multiplying the matrices by running through both A and B row by row. This should minimize the cache misses for B as well as make sure that every element in every cache line is utilized when a cache line is loaded. When scanning a matrix row by row it would give $O(\frac{n^2}{\mathcal{B}})$ cache misses where $\mathcal{B}$ is the cache line size so when we multipy A and B we scan B n times and get $O(\frac{n^3}{\mathcal{B}})$ cache misses instead of $O(n^3)$.

The improvement was implemented for both `Naive` and `Obl:x`. The results are shown in Figure 17. Transposing B is included in the running time. The figure shows that the running time of these approaches performs better than the other implementations for this range of inputs. Figure 28 in the appendix also shows that cache misses are improved significantly.

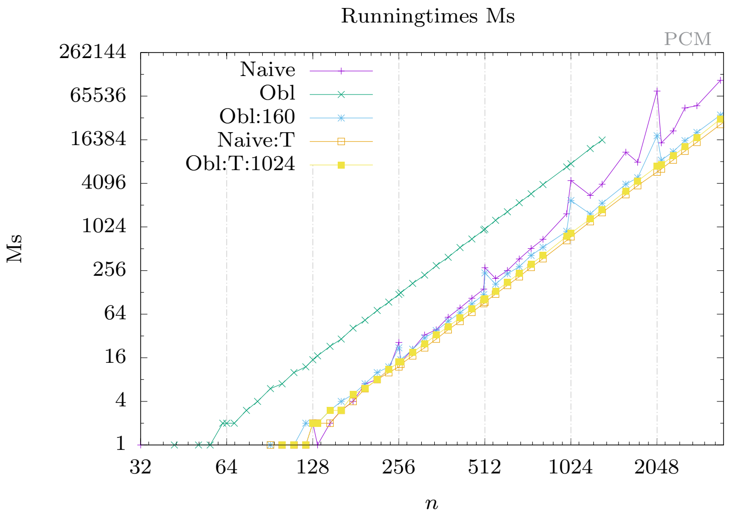


Figure 17: Runningtimes on logarithmic scales showing that the cache oblivious versions and versions that are transposing B performs best. Big inputs for `Obl` are skipped because of its long running times.

As we did for 'Obl:x' we also did a test with different thresholds for 'Obl:T:x'. The results are shown in Figure 27. No further detail will be given about this since it is straight forward and includes no further theoretical considerations. `Obl:T:1024` was found to be the best for the inputs that we used which is why it will be use throughout the report.

An interesting thing that we noticed was that for sufficiently large inputs (starting from matrices of size 4096×4096) L2 cache misses for `Naive:T` are accelerated by input sizes that are powers of 2 where as `Obl:T:1024` cache misses are not. This can be verified in Figure 28 in the appendix. This indicated that maybe `Obl:T:1024` could beat `Naive:T` by choosing the right input. So we did a test run with inputs of size 16384×16384 and $24,576 \times 24,576$ ($24,576 = 2^{14} + 2^{13}$). Sadly the test yielded no new result and `Naive:T` was still the fastest implementation with least cache misses as well as TLB misses.

3.3.5 Iteration 4 - Tiling

A last attempt to make something that could beat `Naive:T` was to use the `Tile:x` algorith, that have same bounds for computation and cache misses as `Obl`[7]. Figure 18 shows an excerpt of the whole diagram shown in Figure 30 in the appendix. The result is that we were able to get close to the running time of both `Naive:T` and `Obl:T:1024` using the tile strategy but we were never able to beat them.

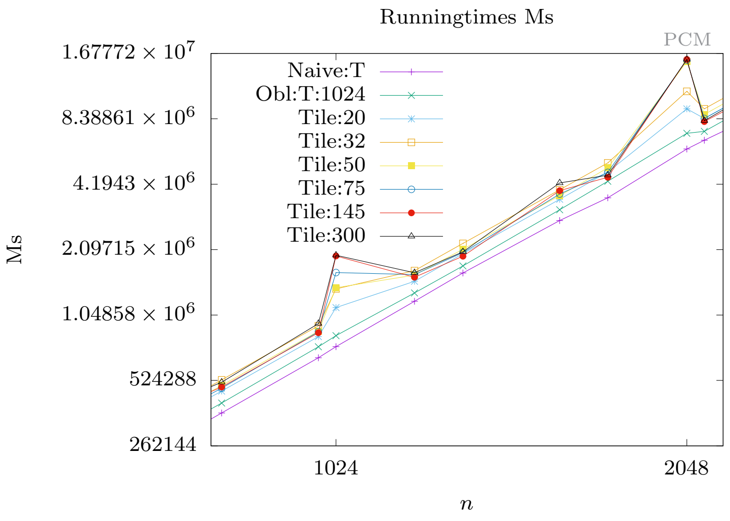


Figure 18: Excerpt from Figure 30 of running times for Tiled implementations showing that `Tile:x` only comes close to performing as well at `Naive:T` and `Obl:T:1024`. The result is similar for the whole range of Figure 30.

Tile sizes were calculated in terms of L1 and L2 cache sizes. 50×50 should fill appx. 30 Kb ($\frac{50\times 50 * 4b * 3}{1024}$) and thus be utilizing the whole L1 cache and similar for 145×145 tiles and L2 cache size. The rest of the sizes were chosen to surround the two calculations.

Results from benchmarks – not surprisingly – shows that whenever the tile size is within a cache size the algorithm has few cache misses on that level. More interesting is that for all those versions that are within some cache size the one with the largest tiles is the one with least cache misses.

An example is that for L1 cache misses `Tile:20` and `Tile:32` are within the L1 cache size since and `Tile:32` has the least L1 cache misses. The `Tile:50` that we calculated to be within L1 cache seemed to be a bit too big according to Figure 31 since cache misses for this tile size are varying back and forth from few to many compared to the other tile sizes. This is also the cace fot `Tile:145` and L2 cache misses in Figure 32.

If the cache misses were the only factor influencing the running time one would expect that the tile size that had the best running time would be either the largest one to fit into L1 or L2 cache. This is not the case as Figure 18 also indicates. From our measurements we did not find the reason why the smallest tile we tried was the fastest. We would have to investigated branch mispredictions, instruction counts etc. to explain this result.

3.4 Discussion

This section discusses choices and approaches for implementations, experiments and algorithms in general. It will explain the reason why we choose the things we did and what we could have done instead.

3.4.1 Implementation

For this project the assembly produced has not been considered. This means that the concrete implementations haven't been optimized more than what was done in the C++ code. What has been done though is to minimize multiplications and especially divisions in the implementations. This approach could result in compiler optimizations that would harm some concrete algorithms. So clearly this is not the best approach but since that data access has turned out to be one of the main factors we believe that we have still arrived at a quite good result.

The fact that the matrix representation used is just one int array with rows laid out consecutively gives us the ability to split matrices horizontally just by adding an offset to a pointer. Splitting vertically was a bit more costly of the need to maintain offsets. The alternative to a consecutive array would be an array for each row which would introduce more cache misses and slow down the algorithms.

3.4.2 Experiments

We chose to put a lot of effort in cache misses in this project because we saw from the very beginning how much effect it had on the running times but also because Frigo et al[7] focus a lot on this. More time could have been invested in branch mispredictions and other hardware counts but these measures didn't seem to have a significant effect on the running times.

This project has only included a single concrete comparison between theory and the actual results when dividing runningtimes with n^3 to ensure that the bounds held in the very beginning of the project. The reason for this is that more algorithms were favoured over optimization of the individual algorithms.

We would also have liked to averaged our measures over more iterations to level out fluctuations but because of the n^3 runningtime it was not feasible for the bigger matrices. This could have give us more accurate results.

A last thing that may have influenced the results is the fact that only square matrices were used. All of the algorithms that were used were implemented to handle arbitrary sizes of matrices. We restricted the usage to squares because it made the testing much easier. One could imagine that other results would appear with none square matrices.

3.4.3 Algorithms

The algorithms implemented for this project were only the ones presented by Frigo et al. and our immediate ideas such as transposing matrices and mixing the different approaches. There are more effecient sufisticated algorithms for multiplying matrices. One popular algorithm is the Strassen & Winograd algorithm that has a running time of $O(n^{\log_2(7)})$[1]. Also parallelism has been used to improve running times[3] as well as we heard about GPU usage in a lecture. So we could have duck much deeper into this project to make even better algorithms but our time constraints did not allow us to do so.

Also all the algorithms implemented worked *in place* meaning that they did not needed to keep track of temporary results. This would be nescessary for the Strassen & Winograd algorithm for instance. This would have complicated our measures as well as implementations considerably.

3.5 Conclusion

In conclusion this project presents experiments with multiple variants and mixes of the naive algorithm and the cache oblivious algorithm to find that the best algorithm that we could produce was `Naive:T` where the second matrix is transposed before multiplication. This gives the benefit of scanning the first and the second matrix. Both the naive algorithm and the cache oblivious algorithm by them selves were by far the slowest but mixing them by letting the naive algorithm handle the bottom of the recursion gave good better results.

References

[1] *20th Annual International Conference on High Performance Computing, HiPC 2013, Bengaluru (Bangalore), Karnataka, India, December 18-21, 2013.* IEEE Computer Society, 2013, ISBN 978-1-4799-0730-4. `http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6784161`.

[2] *Quicksort*, march 2017. `http://en.wikipedia.org/wiki/Quicksort`.

[3] Ballard, Grey, James Demmel, Olga Holtz, Benjamin Lipshitz, and Oded Schwartz: *Communication-optimal parallel algorithm for strassen's matrix multiplication.* In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 193–204, New York, NY, USA, 2012. ACM, ISBN 978-1-4503-1213-4. `http://doi.acm.org/10.1145/2312005.2312044`.

[4] Bentley, Jon L and M Douglas McIlroy: *Engineering a sort function.* Software: Practice and Experience, 23(11):1249–1265, 1993.

[5] Bentley, Jon Louis: *Programming pearls.* Addison-Wesley, 1986, ISBN 978-0-201-10331-1.

[6] Frigo, Matteo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran: *Cache-oblivious algorithms.* In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 285–, Washington, DC, USA, 1999. IEEE Computer Society, ISBN 0-7695-0409-4. `http://dl.acm.org/citation.cfm?id=795665.796479`.

[7] Frigo, Matteo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran: *Cache-oblivious algorithms.* ACM Trans. Algorithms, 8(1):4:1–4:22, 2012. `http://doi.acm.org/10.1145/2071379.2071383`.

[8] Hoare, Charles AR: *Quicksort.* The Computer Journal, 5(1):10–16, 1962.

[9] Hu, Haodong: *Cache-oblivious Data Structures for Massive Data Sets.* PhD thesis, Stony Brook, NY, USA, 2007, ISBN 978-0-549-90048-1. `http://dl.acm.org/citation.cfm?id=1571464`, AAI3336192.

[10] ICL Team, University of Tennessee, Innovative Computing Laboratory: *Performance application programming interface (papi)*, feb 2017. `http://icl.cs.utk.edu/papi/index.html`.

[11] @Intel: *Processor counter monitor (pcm)*, mar 2017. `https://github.com/opcm/pcm`.

[12] @intel: *Processor information, intel® core^{TM} i7-5600u processor*, mar 2017. `https://ark.intel.com/products/85215/Intel-Core-i7-5600U-Processor-4M-Cache-up-to-3_20-GHz`.

[13] Khuong, Paul-Virak and Pat Morin: *Array layouts for comparison-based searching.* CoRR, abs/1509.05053, 2015. `http://arxiv.org/abs/1509.05053`.

[14] Kushagra, Shrinu, Alejandro López-Ortiz, Aurick Qiao, and J. Ian Munro: *Multi-pivot quicksort: Theory and experiments.* In *2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2014, Portland, Oregon, USA, January 5, 2014*, pages 47–60, 2014. `http://dx.doi.org/10.1137/1.9781611973198.6`.

[15] OpenJDK: *Jmh: a java harness for building, running, and analysing nano/micro/milli/macro benchmarks written in java and other languages targetting the jvm.*, march 2017. `http://openjdk.java.net/projects/code-tools/jmh/`.

[16] OpenMP: *The openmp api specification for parallel programming*, mar 2017. `http://www.openmp.org/`.

[17] Sedgewick, Robert and Kevin Wayne: *Insertion sort*, march 2017. `http://algs4.cs.princeton.edu/21elementary/InsertionX.java.html`.

[18] Wikipedia: *Insertion sort — Wikipedia, the free encyclopedia*. `http://en.wikipedia.org/w/index.php?title=Insertion_sort&oldid=770857101`, 2017. [Online; accessed 19-March-2017].

[19] Wild, Sebastian: *Malijan: Maximum likelihood java bytecode analyzer*, march 2017. `http://wwwagak.cs.uni-kl.de/home/forschung/malijan`.

[20] Wild, Sebastian and Markus E. Nebel: *Average case analysis of java 7's dual pivot quicksort*. CoRR, abs/1310.7409, 2013. `http://arxiv.org/abs/1310.7409`.

[21] Wild, Sebastian, Markus E. Nebel, Raphael Reitzig, and Ulrich Laube: *Engineering java 7's dual pivot quicksort using malijan*. In *Proceedings of the 15th Meeting on Algorithm Engineering and Experiments, ALENEX 2013, New Orleans, Louisiana, USA, January 7, 2013*, pages 55–69, 2013. `http://dx.doi.org/10.1137/1.9781611972931.5`.

[22] Yaroslavskiy, Vladimir: *Dual-pivot quicksort*. Research Disclosure, 2009.

Appendix

A.1 Project 1 - Binary search

A.1.1 Description of predicates

Algorithm 2: pred_stable(array, size, y)

```
result ← null
while size ≠ 0 do
    mid ← $\lfloor \frac{size}{2} \rfloor$
    if array[mid] ≤ y then
        result ← array + mid
        array ← result + 1
        size ← size − mid − 1
    else
        size ← mid
return result
```

Algorithm 3: pred_stable(bfs, size, y)

```
result ← null
i ← 0
while i < size do
    if bfs[i] ≤ y then
        result ← bfs + i
        i ← 2 × i + 2
    else
        i ← 2 × i + 1
return result
```

Algorithm 4: pred_stable(dfs, size, y)

```
result ← null
while size ≠ 0 do
    if *dfs ≤ y then
        result ← dfs
        dfs ← dfs + ⌊size/2⌋ + 1
        size ← size − ⌊size/2⌋ − 1
    else
        dfs ← dfs + 1
        size ← ⌊size/2⌋
return result
```

Layout	Name	Method (`layouts::name` omitted)	Description
inorder	`ino.s`	`pred_stable`	Stable predicate for inorder layout.
inorder	`ino.u`	`pred_unstable`	Ditto but unstable.
BFS	`bfs.s`	`pred_stable`	Stable predicate for BFS layout.
BFS	`bfs.u`	`pred_unstable`	Ditto but unstable.
BFS	`bfs.cs`	`pred_cmov_stable`	Stable predicate for BFS layout, uses `?:` in the hope that Visual Studio generates `cmov*` instructions.
BFS	`bfs.ps`	`pred_pref_stable`	Stable predicate for BFS layout, uses `_mm_prefetch` to prefetch data into the cache line.
BFS	`bfs.cps`	`pred_cmov_pref_stable`	Mixture of the above two.
DFS	`dfs.s`	`pred_stable`	Stable predicate for DFS layout.
DFS	`dfs.u`	`pred_unstable`	Ditto but unstable.
vEB	`veb.rs`	`pred_recursive_stable`	Stable predicate for van Emde Boas layout. The binary search is a recursive method.
vEB	`veb.ru`	`pred_recursive_unstable`	Ditto but unstable.
vEB	`veb.is`	`pred_iterative_stable`	Stable predicate for van Emde Boas layout. The binary search is implemented with an explicit stack.
vEB	`veb.iu`	`pred_iterative_unstable`	Ditto but unstable.
vEB	`veb.ns`	`pred_inlined_stable`	Stable predicate for van Emde Boas layout. The binary search is recursive, but expanded for the first 5 recursive calls.
vEB	`veb.nu`	`pred_inlined_unstable`	Ditto but unstable.
vEB	`veb.bs`	`pred_bfs_stable`	Stable predicate for van Emde Boas layout. The binary search tracks the BFS index of the current node and uses a method to convert BFS index to the index in van Emde Boas layout.
vEB	`veb.bu`	`pred_bfs_unstable`	Ditto but unstable.

Table 4: Description of different versions of the predicates.

A.2 Project 2 - Extra diagrams

A.2.1 Suitable Insertion sort

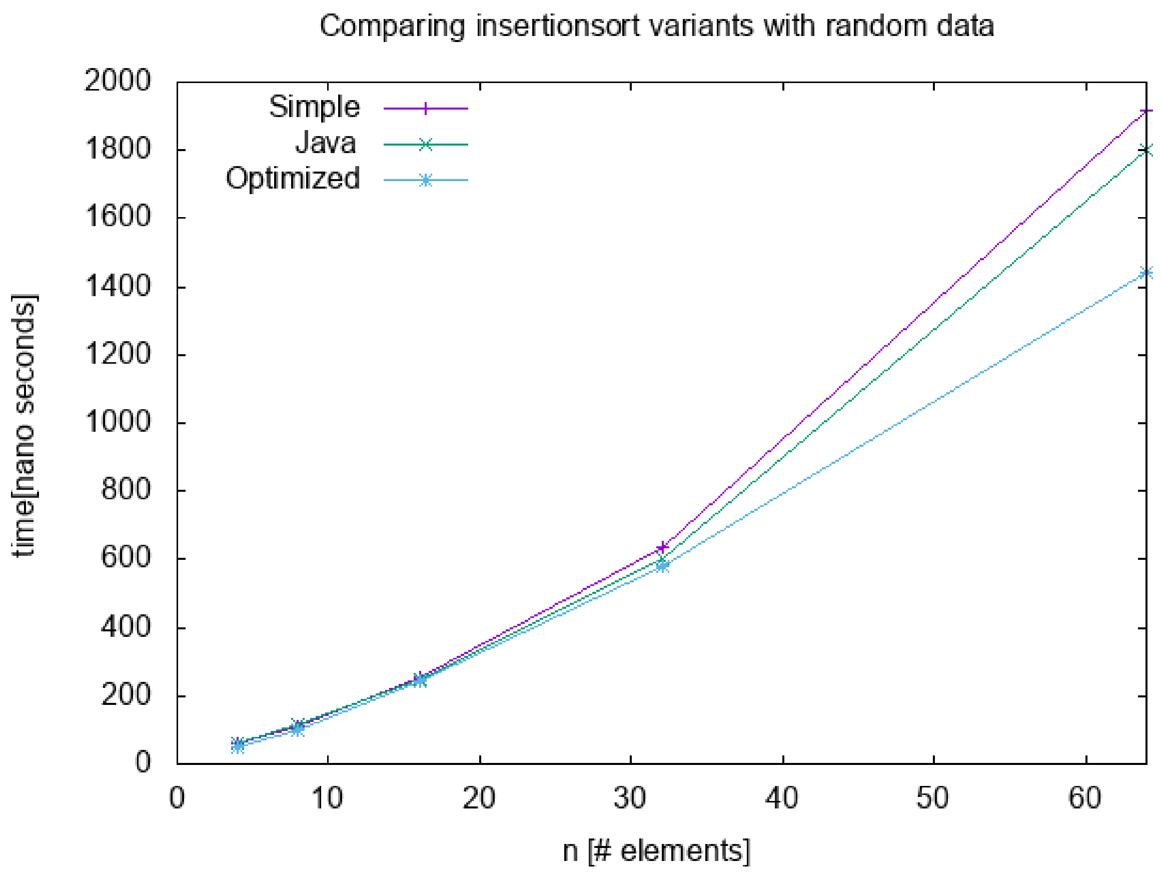


Figure 19: Different versions of insertion sort tested for increasing output with random data. Each experiment was ran with 100 warmup rounds followed by 100 test rounds. New input was generated at every iteration.

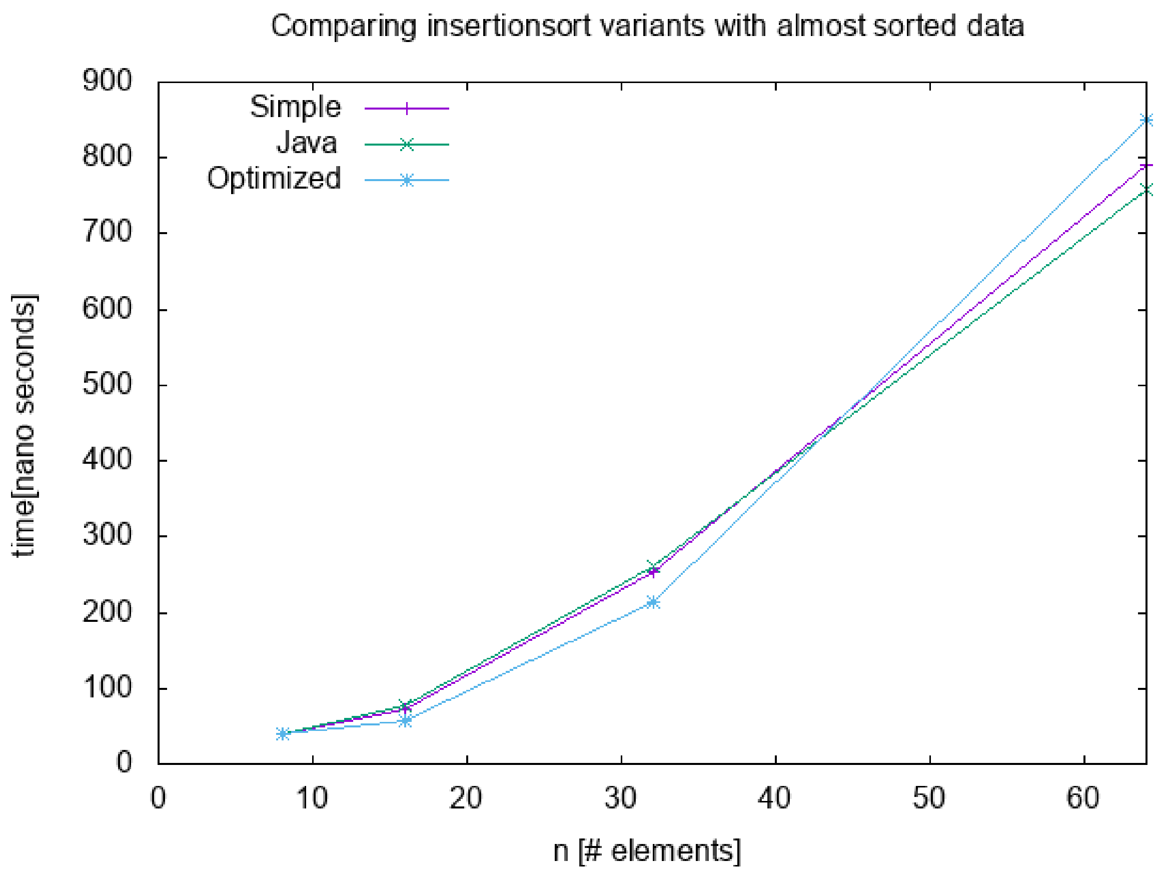


Figure 20: Different versions of insertion sort tested for increasing output with almost sorted data. Each experiment was ran with 100 warmup rounds followed by 100 test rounds. New input was generated at every iteration.

A.2.2 Cutoff

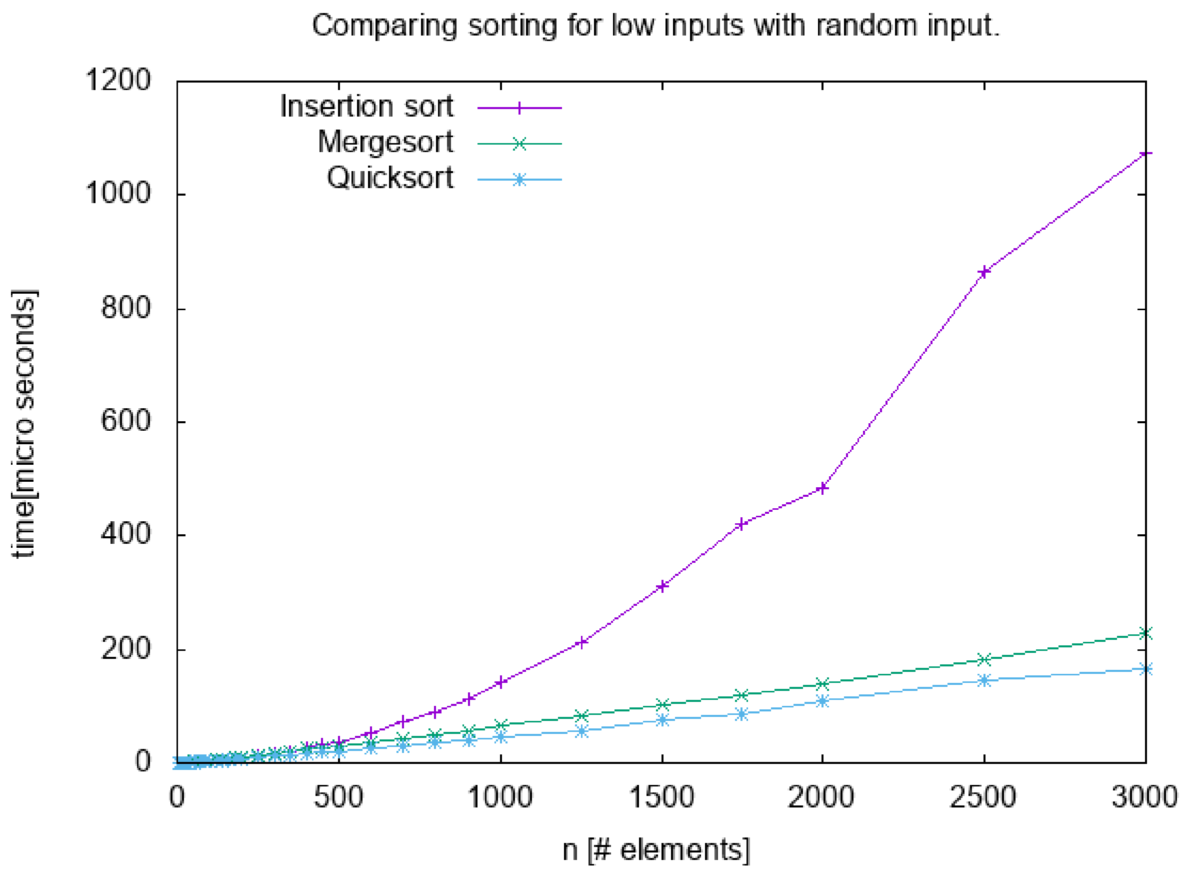


Figure 21: Experiment with small input values. The following algorithms was used: Algorithm 5 (insertion sort), Algorithm 6 (quicksort), and Algorithm 7 (mergesort). Each experiment was ran with 100 warmup rounds followed by 100 test rounds. New input was generated at every iteration.. The difference between each round was minimal, thus no error bars have been added to the plot. Random data was used.

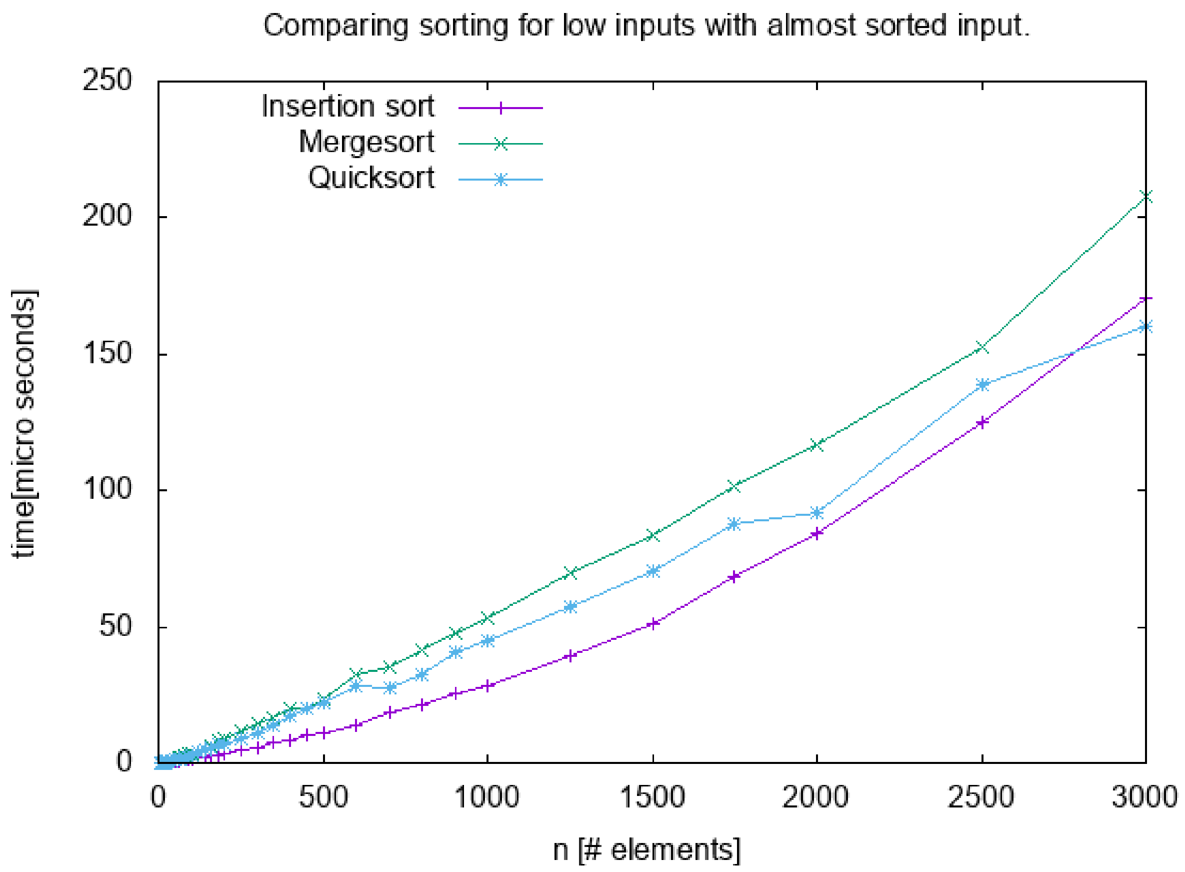
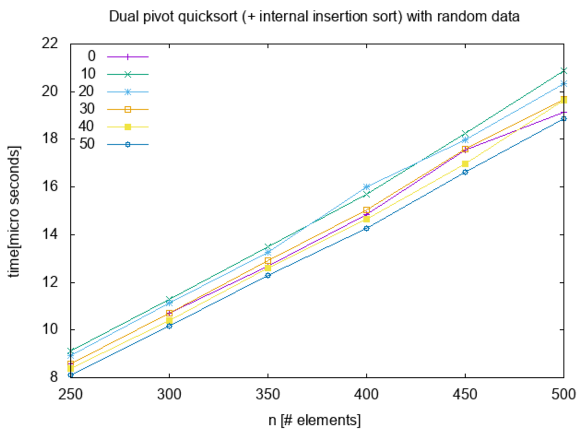
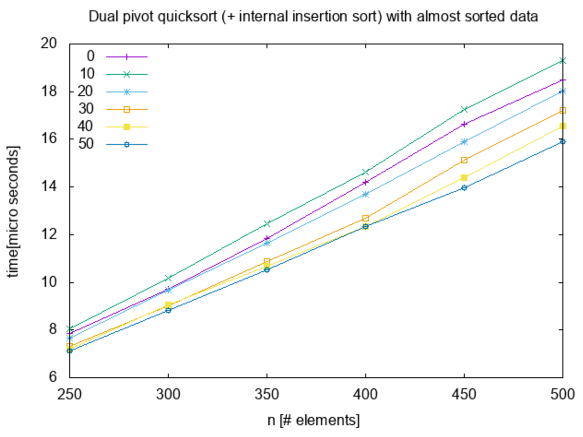


Figure 22: Experiment with small input values. The following algorithms was used: Algorithm 5 (insertion sort), Algorithm 6 (quicksort), and Algorithm 7 (mergesort). Each experiment was ran with 100 warmup rounds followed by 100 test rounds. New input was generated at every iteration.. The difference between each round was minimal, thus no error bars have been added to the plot. Almost sorted data was used.

A.2.3 Optimizing Quicksort



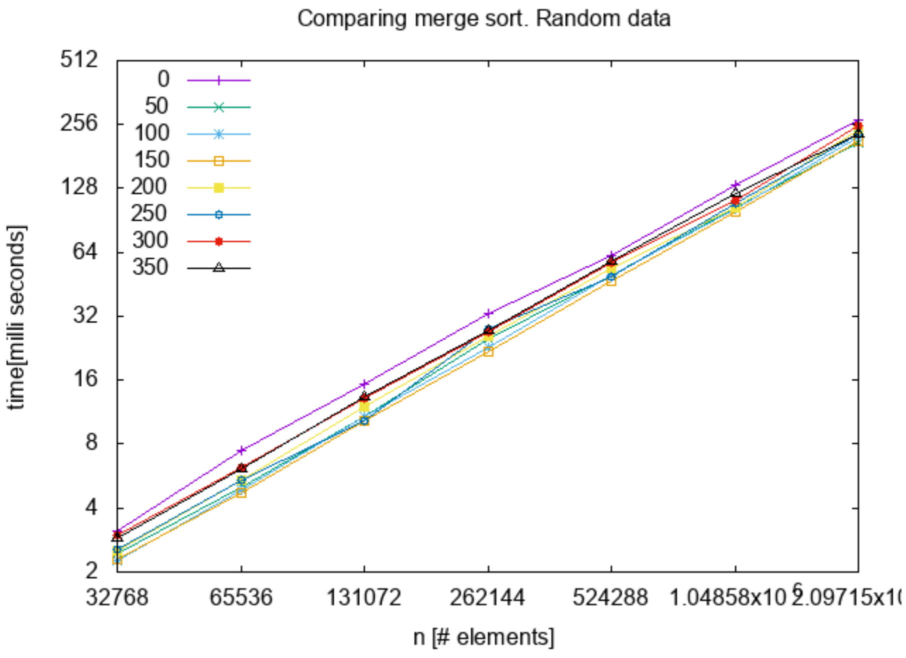
(a) Random input data.



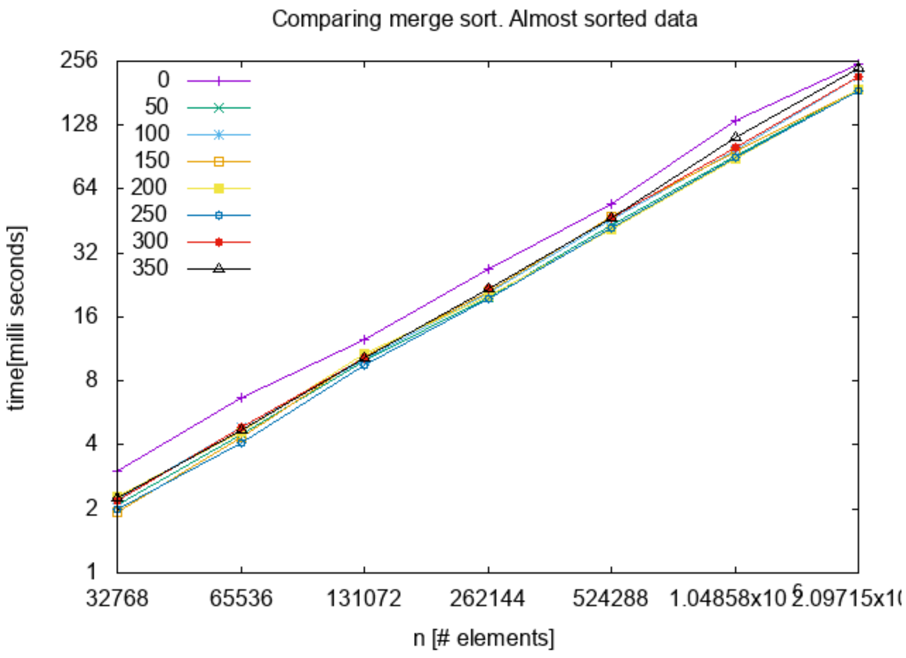
(b) Almost sorted data.

Figure 23: Experiment with different threshold values for when quicksort, Algorithm 6, should change to insertion sort, Algorithm 5. Each experiment was ran with 100 warmup rounds followed by 100 measured rounds. New input was generated at every iteration. The difference between each round with same parameters was minimal, thus no error bars have been added to the plot.

A.2.4 Optimizing Mergesort



(a) Random input data.



(b) Almost sorted data.

Figure 24: Experiment with different threshold values for when mergesort, Algorithm 7, should change to quicksort, Algorithm 6. Each experiment was ran with 100 warmup rounds followed by 100 measured rounds. New input was generated at every iteration. The difference between each round with same parameters was minimal, thus no error bars have been added to the plot.

A.3 Project 2 - Implementations

A.3.1 Optimized insertionsort

Algorithm 5 is based on the insertion sort algorithm from [17].

Algorithm 5: Optimized Insertionsort. sort(int[] input, int left, int right)

```
int n = right + 1, exchanges = 0;
for int i = n - 1; i > left; i−− do
    if input[i] < input[i - 1] then
        swap(input, i, i-1); exchanges++;
if exchanges == 0 then
    return;
for int i = left + 2; i < n; i++ do
    int v = input[i], j = i; while v < input[j - 1] do
        input[j] = input[j - 1];
        j−−;
    input[j] = v;
```

A.3.2 Dual Pivot Quicksort

Algorithm 6: QuickDualPivot. sort(int[] input, int left, int right)

```
if right <= left then
    return;
if input[right] < input[left] then
    swap(intput, left, right);
int lt = left = 1, gt = right - 1, i = left + 1;
while i < gt do
    if input[i] < input[left] then
        swap(input, lt++, i++);
    else if input[right] then
        swap(input, i, gt−−);
    else
        i++
swap(input, left, −−lt);
swap(input, right, ++gt);
sort(input, left, lt-1);
sort(input, lt+1, gt-1);
sort(input, gt+1, right);
```

A.3.3 Mergesort

Algorithm 7: Mergesort. sort(int[] input, int left, int right)

```
if right < left + QUICKSORT_CUTOFF then
    //Sort with Quicksort
if left < right then
    int middle = (left + right) >>> 1;
    mergesort(helper, input, left, middle); mergesort(helper, input, middle + 1, right); if
     input[middle] <= input[middle + 1] then
        System.arraycopy(helper, left, input, right - left + 1);
        return;
    merge(helper, input, left, midddle, right);
```

Algorithm 8: merge(int[] input, int left, int right)

```
int i = left, j = middle + 1;
for int k = left; k <= right; k++ do
    if i > middle then
        input[k] = helper[j++];
    else if j > right then
        input[k] = helper[i++];
    else if helper[j] < helper[i] then
        input[k] = helper[j++];
    else
        input[k] = helper[i++];
```

A.4 Project 3 - Algorithms

Algorithm 9: Naive for-loop implementation of matrix multiplication (`Naive`).

```
for i = [0..m] do
    for j = [0..p] do
        for k = [0..n] do
            C[i][j] += A[i][k] * B[k][j]
```

Algorithm 10: Navie for-loop implementation of matrix multiplication with transposed B (`Naive:T`)

```
// Call transpose first for i = [0..m] do
    for j = [0..p] do
        for k = [0..n] do
            C[i][j] ← C[i][j] + A[i][k] * B[j][k] // diff is here
```

Algorithm 11: Tiled for-loop implementation using s as cache width and height for tile (`Tile:s`). See `ord_mult` below.

```
a_row_idx, a_offset, b_offset ← 0
tiles ← $\lceil \frac{m}{s} \rceil$
for i = [0..tiles] do
    m_size ← min(s, m-a_row_idx)
    b_col_idx ← 0
    for j = [0..tiles] do
        p_size ← min(s, p-b_col_idx)
        for k = [0..tiles] do
            n_size ← min(s, n - a_offset)
            ord_mult(A + a_row_idx, B + a_offset, m_size, n_size, p_size, a_offset,
             b_offset, Dest + a_row_idx)
            a_offset ← a_offset + 1
        b_offset ← b_offset + s
        b_col_idx ← b_col_idx + s
        a_offset ← 0
    b_offset ← 0
    a_row_idx ← a_row_idx + s
```

Algorithm 12: `ord_mult`.

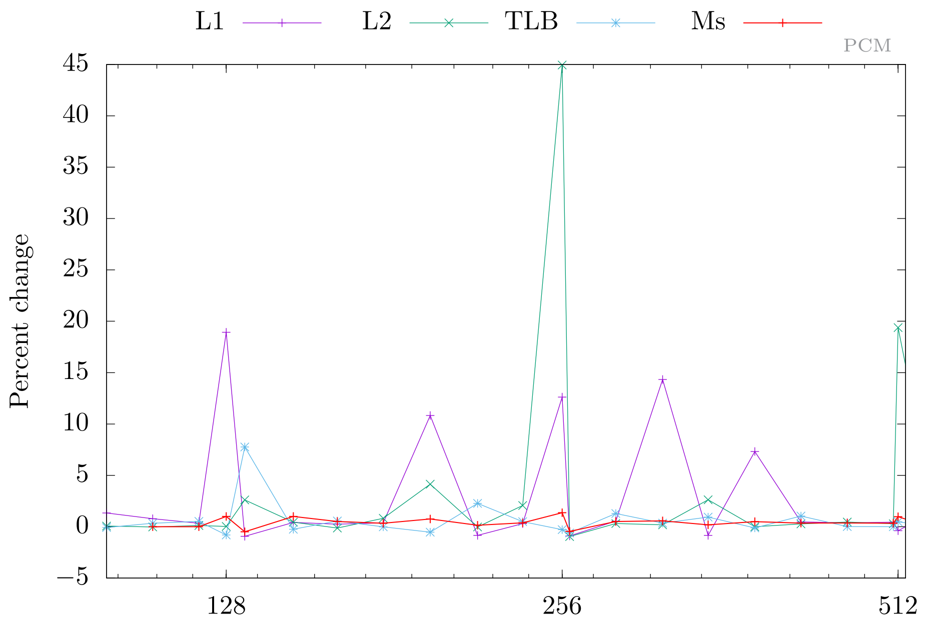
```
for i = [0..m] do
    for j = [0..p] do
        for k = [0..n] do
            C[i][c_offset + j] += A[i][a_offset + k] * B[k][b_offset + j]
```

A.5 Project 3 - Algorithm abbreviations

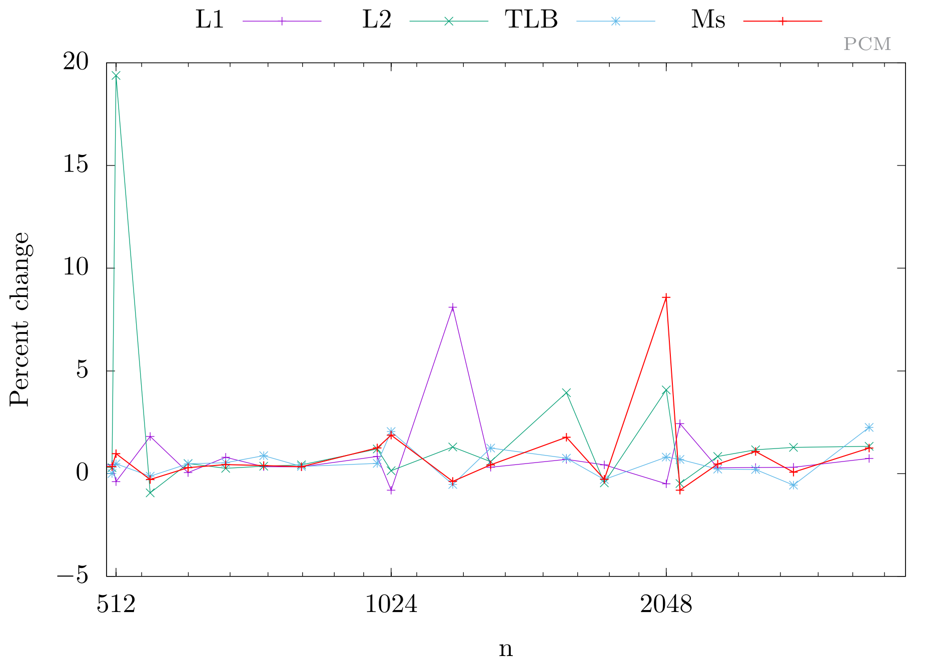
Abbreviation	**Specifications**
`Naive`	Naive approach
`Naive:T`	Naive approach and transposed B
`Naive:2`	Naive approach using two cores
`Naive:4`	Naive approach using four cores
`Naive:8`	Naive approach using eight cores
`Naive:16`	Naive approach using 16 cores
`Naive:32`	Naive approach using 32 cores
`Naive:64`	Naive approach using 64 cores
`Naive:128`	Naive approach using 128 cores
`Obl`	Cache oblivious only from [7]
`Obl:2`	Cache oblivious, threshold 2 (equivalent to `Obl`)
`Obl:10`	Cache oblivious, threshold 10
`Obl:40`	Cache oblivious, threshold 40
`Obl:130`	Cache oblivious, threshold 130
`Obl:145`	Cache oblivious, threshold 145
`Obl:160`	Cache oblivious, threshold 160
`Obl:180`	Cache oblivious, threshold 180
`Obl:640`	Cache oblivious, threshold 640
`Obl:1280`	Cache oblivious, threshold 1280
`Obl:T:8`	Cache oblivious, threshold 8 and transposed B
`Obl:T:16`	Cache oblivious, threshold 16 and transposed B
`Obl:T:32`	Cache oblivious, threshold 32 and transposed B
`Obl:T:64`	Cache oblivious, threshold 64 and transposed B
`Obl:T:128`	Cache oblivious, threshold 128 and transposed B
`Obl:T:256`	Cache oblivious, threshold 256 and transposed B
`Obl:T:512`	Cache oblivious, threshold 512 and transposed B
`Obl:T:1024`	Cache oblivious, threshold 1024 and transposed B
`Obl:T:2048`	Cache oblivious, threshold 2048 and transposed B
`Tile:20`	Tiled approach, tile size 20 by 20
`Tile:32`	Tiled approach, tile size 32 by 32
`Tile:50`	Tiled approach, tile size 50 by 50
`Tile:75`	Tiled approach, tile size 75 by 75
`Tile:145`	Tiled approach, tile size 145 by 145
`Tile:300`	Tiled approach, tile size 300 by 300

Table 5: Algorithm abbreviations and their specifications

A.6 Project 3 - Extra diagrams



(a) Range [100:520]



(b) Range [500:3750]

Figure 25: Change in percent for square matrices of size $n \times n$ for `Naive`. All data here is percentage-wise change between datasets of size n^2.

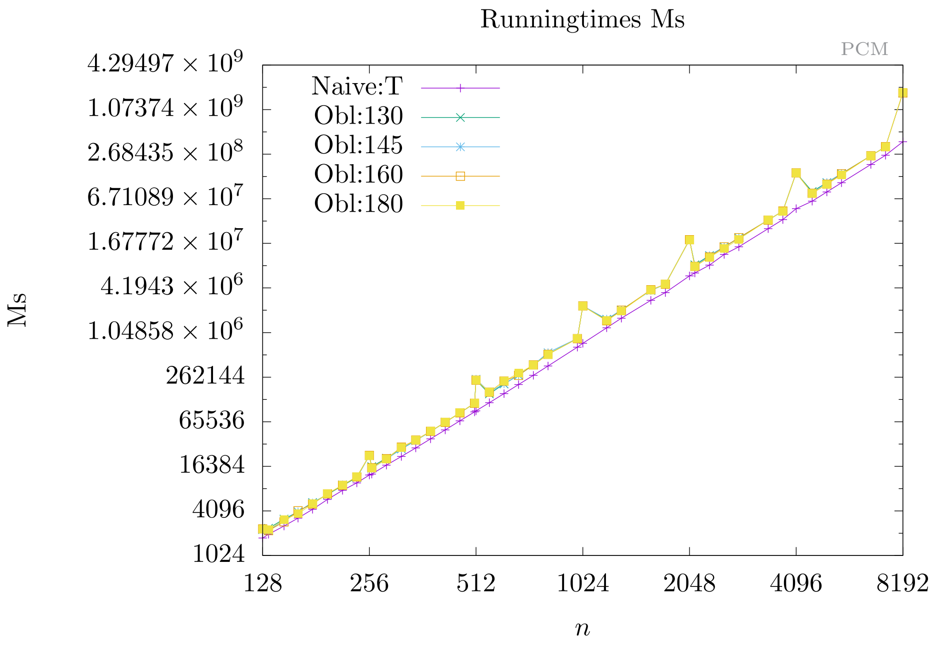


Figure 26: Running times for different thresholds for `Obl:x`. This run was conducted to narrow down the best threshold.

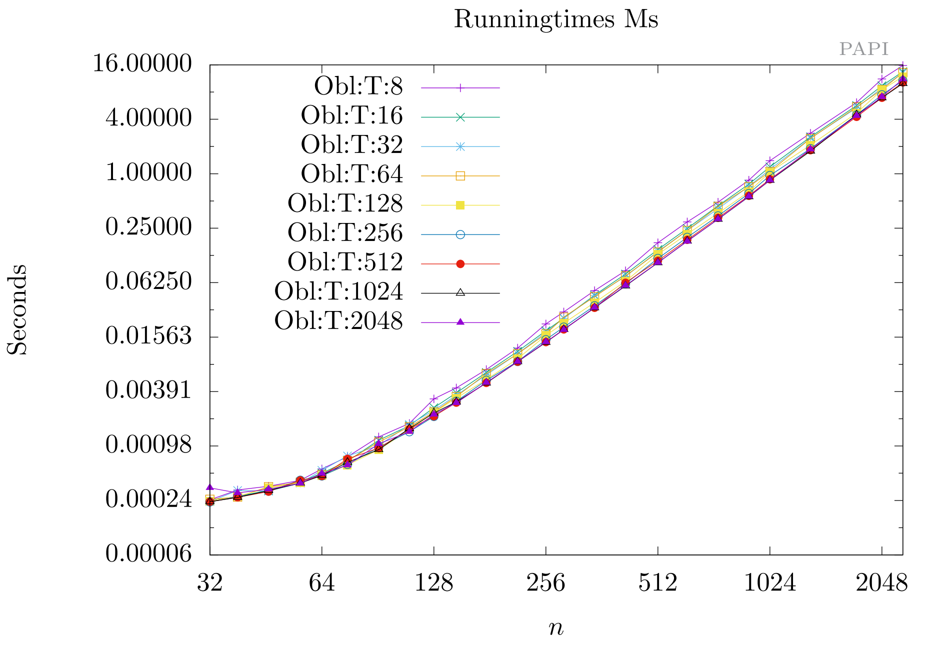


Figure 27: Running times for different thresholds of the Oblivious approach with transposed B. This run was conducted to find the best threshold.

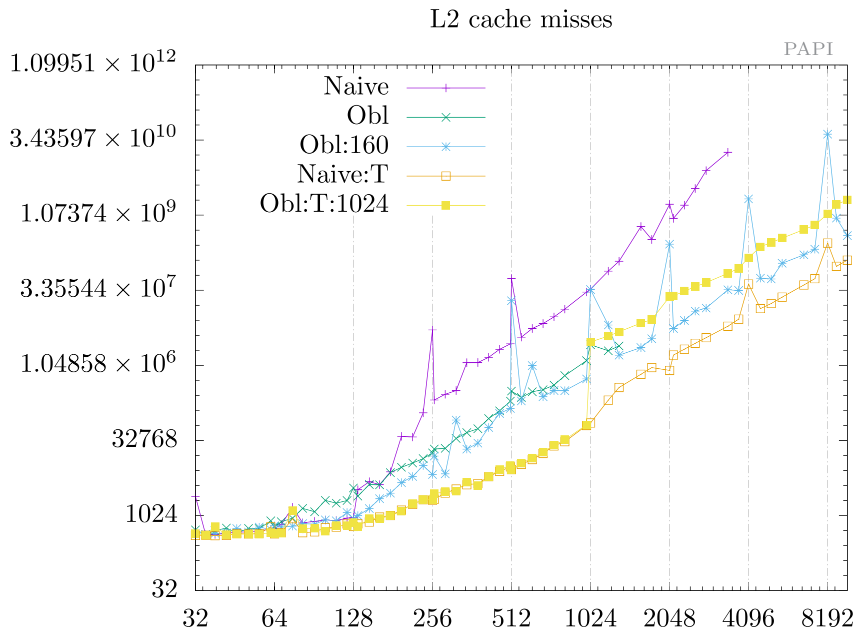


Figure 28: L2 cache misses on logarithmic scales. X-axis is n for $n \times n$ matrices. The graph shows that transposing B lowers cache misses a lot for `Naive:T` and that the cache oblivious versions are stable against matrices with sizes that are powers of 2.

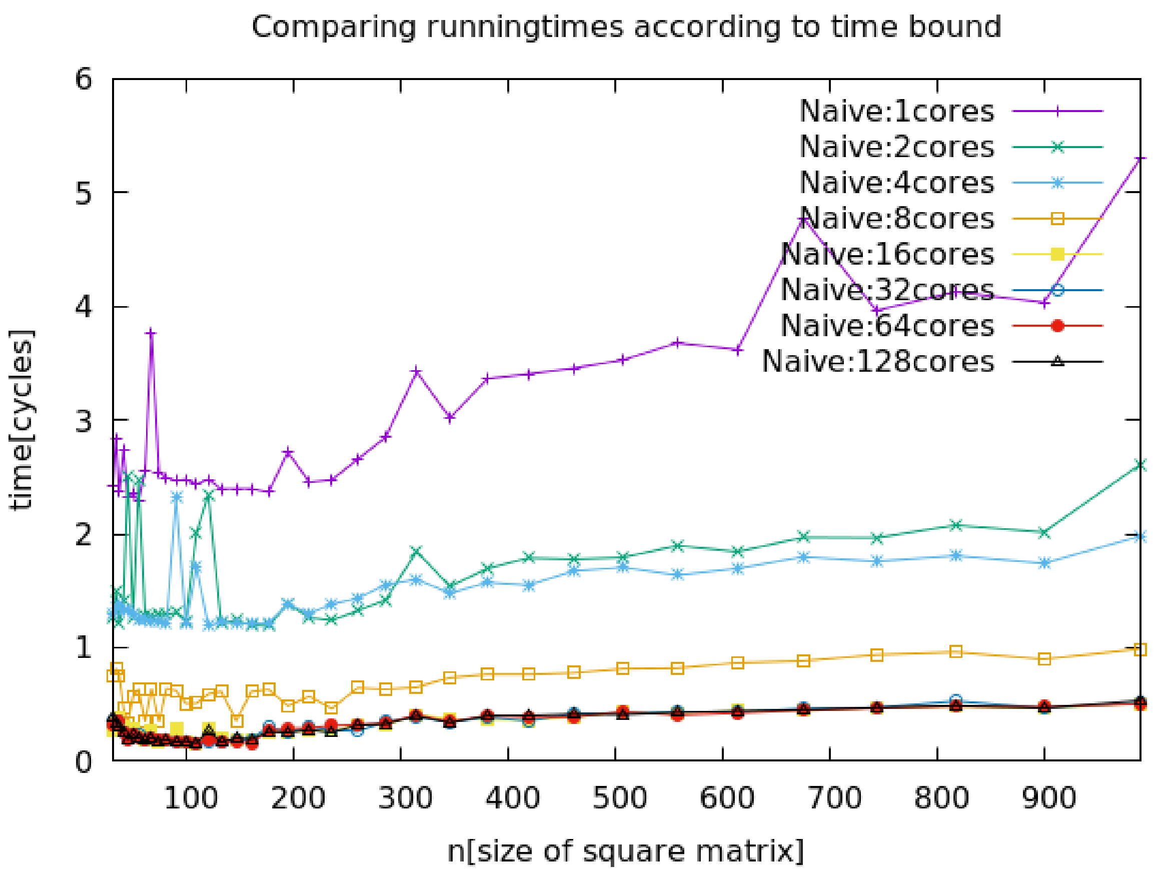


Figure 29: Misleading measures of cycles conducted with PAPI when using OpenMP to parallelize multiplications. These results are not valid.

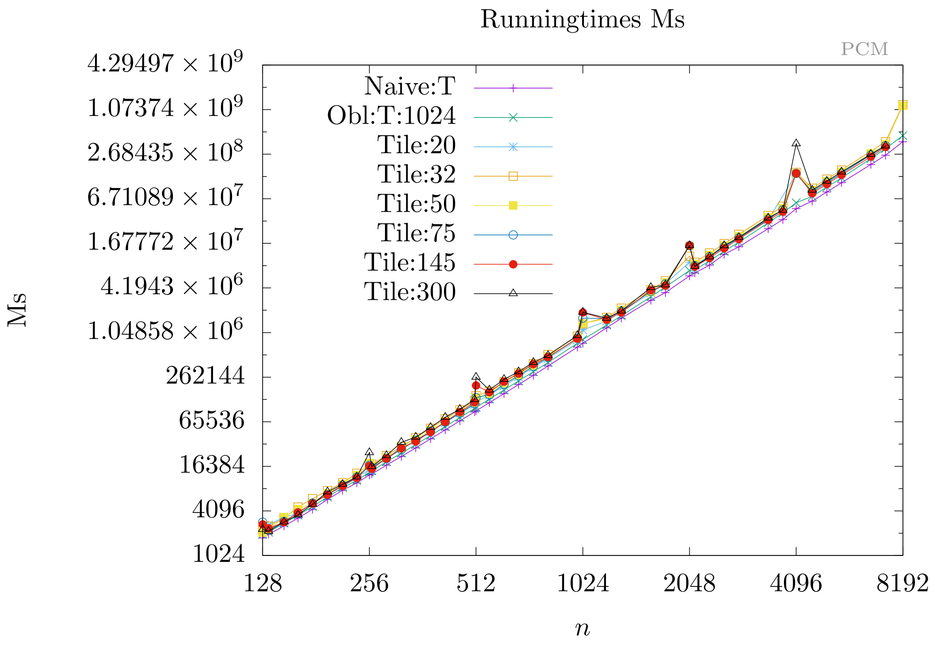


Figure 30: Running time for 'Naive:T' and tiled multiplication 'Tile:x' where x is the width and height of each tile. It shows that the performance of `Tile:T` is close to being as good as `Naive:T` but it is vulnerable to sizes that are powers of 2

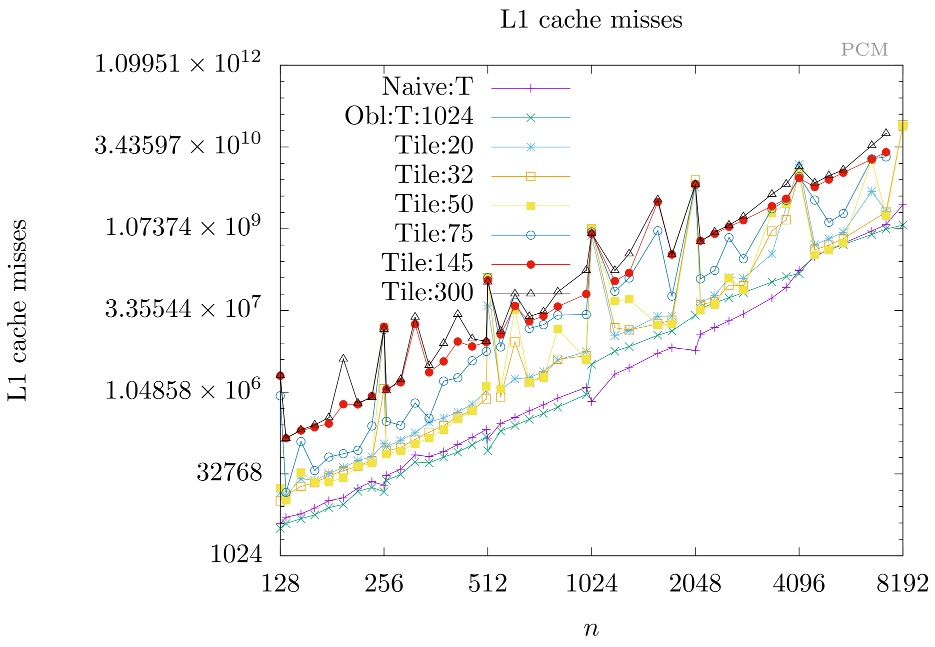


Figure 31: L1 cache misses associated with the running times shown in figure 30.

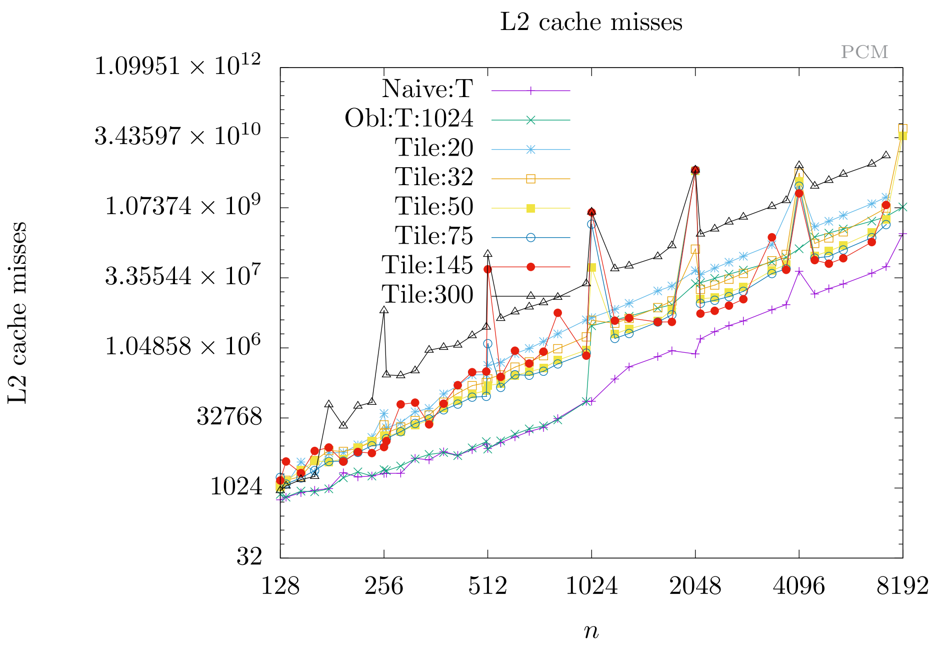


Figure 32: L2 cache misses associated with the running times shown in figure 30.

A.7 How to Run the Programs

The source code for the projects is available in the following repositories:

Binary search: `https://github.com/rlunding/alg_eng`.

Improving Quicksort in Java: `https://github.com/rlunding/alg_eng_quicksort`

Matrix multiplication: `https://github.com/fhvilshoj/alg_eng_matrix`

The `README` files for the projects explains everything necessary to build and run the projects.