U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

April 2016

Performance evaluation of a single core

Parallel Computing – 1st assignment

Daniel Gomes and Diogo Gomes
{up201306839, up201106586}@fe.up.pt

Problem description

Improving performance is often thought of increasing CPU clock or, in a parallel manner, by increasing the number of CPUs. Yet, how much performance improvement could actually be achieved by just changing the algorithm itself to take advantage of the hardware characteristics?

As it is known, memory accesses are by far more expensive than CPU registry accesses and for that reason CPU CACHE memory is an important artifice that keeps some data quickly available. It should be emphasized that the size of CACHE memory is substantially smaller than main memory and, for that reason, required data will not frequently be accessible in it and a carefully thought algorithm is required to take the most possible advantage of it.

Therefore, in this project, it will be studied the effect of the memory (cache level 1, level 2 and main memory) on the processor performance when accessing large amounts of data by analysing performance metrics values for two slightly variations of the same operation, the algebraic product of two matrices. A first naive version, and a second, that more efficiently uses cache memory by accessing both multiplied matrices line by line. The first version will also be implemented in Ruby and compared with the C++ version. Later, multi thread versions of this previous two ones will be analysed , which were achieved using OpenMP API were executed in between 1 and 4 threads running on different CPU cores.

Algorithms explanation

The algebraic product matrices operations states that the value of each cell ***(i, j)*** of the result matrix ***C*** should be the sum of each element of the line ***i*** multiplied by the respective element on the column ***j*** of the matrix ***B***. For this reason it is implied that the number of columns of ***A***, A_{cols}, must be equal to the number of rows of **B**, B_{rows}, and the size of matrix ***C*** is $A_{rows} \times \mathbf{B}_{cols}$.

$$C(i,j) \;=\; \sum_{k=1}^{A_{cols}} A(i,k) \;*\; B(k,j)$$

Line by row – the naive algorithm

Following directly from the mathematical definition, the algorithm could be created by chaining two loops, controlling the two variables ***i*** and ***j*** to access to each position of ***C***. A third inner loop is used to walk true the line ***i*** of a and column ***j*** of ***B***. The sum of the pairs product is preformed and stored in $C_{(i,j)}$. As shown in the algorithm pseudo-code below:

```
for(i = 0; i < $A_{rows}$; i++)
        for(j = 0; j < $B_{cols}$; j++) {
                temp = 0;
                for(k = 0; k < $A_{cols}$; k++)
                        temp += A[i][k] * B[k][j];
                C[i][j] = temp;
        }
```

How cache memory works

By now, it is important to understand how matrices are stored in memory and how cache works, in order to comprehend what can be improved in this first algorithm.

Memory, at its lower level, is a vector of bytes being numbered sequentially. So in order to store two dimensional matrices in memory one of two approaches is followed: Row Major – stores contiguously the row elements and one column after another, and Column Major – column elements are continuously stored, and rows follow one another. In C/C++ matrices the Row major approach is followed.

When a memory access is performed, taking into assumption that if a byte is required in a near future the neighbor's should be to, the next following bytes are retained in CACHE. When accessing a byte if it already is cached (CACHE hit), the CACHE memory will serve it, otherwise the first procedure will occur (CACHE miss). Since CACHE memory is substantially smaller than main memory, recently accessed bytes are kept and the oldest ones are discarded.

It could be therefore deduced that accessing a matrix in an horizontally manner, column by column, row after row, it will result in a major CACHE hit rate versus accessing it in a vertically manner since, hypothetically, when accessing the first element of the row all the row is cached.

Also important to notice and that will be important later is that CACHE stack is divided by levels, the lower ones are faster, since are closer to the core of the CPU and are used to store more recently used data.

Naive implementation memory access analysis

Looking at this first implementation it can be seen that the ***A*** and ***C*** matrices are being accessed horizontally but ***B*** is being accessed vertically, and that for the calculation of each of $\mathbf{n^2}$ values[1] of ***C***, ***B*** is completely loaded to the CACHE memory resulting in a relevant waste of capacity given that only **n** of $\mathbf{n^2}$ elements of ***B*** are required and thus resulting in a potential bigger number of cache misses.

Line by line

Learning from the previous analysis it is possible to modify the algorithm so that CACHE memory is more efficiently used, for example, by accessing **B** in a horizontally manner. This would imply that a much higher number of cache hits would be achieved and thus, on average, faster memory accesses.

Just by swapping the two internal loops that control ***k*** and ***j*** this is achieved. Looking through the middle loop in, it can be understood as: given a line of **B**, all the multiplications involving this line are performed and only after using all the line elements, the next of **B** is considered. This way each cell of **C** is calculated incrementally and not at once (as in the first algorithm) but only after the ***i*** times that the outer loop repeats[2].

```
for(i = 0; i < A_rows; i++)
        for(k = 0; k < A_cols; k++)
                for(j = 0; j < B_cols; j++)
                     C[i][j] += A[i][k]*B[k][j];
```

[1] Where **n** is the number of rows and columns, supposing that **A** and **B** matrix follow $m\times n$, $n\times m$ dimensions.

[2] It is assumed that all elements in C are previously initialized to zero.

Performance metrics and evaluation methodology

Several experiments were performed in order to consolidate the theoretical analysis. At these experiments three main performance metrics were taken into account:

1. Percentage of L1 and L2 data cache misses in relation to the number of operations performed[3], measured using the Performance Application Programming Interface (PAPI)[4], which should have lower values for the second version given that the made improvements.
2. Performance, in Mega FLoating-point Operations Per Second (MFLOPS). As the same processor was used for all the tests, it could be expected that ideally the number of MFLOPS would keep constant that.
3. Execution time, in seconds.

Experiment tests were executed sequentially on an Intel® Core™ i7-3630QM processor and for the plotted results below were considered only one execution of each.

[3] Given that there are 3 operations being executed inside the 3 loops the algorithm executes ***$3n^3$*** operations. Initially was considered to also use PAPI to measure these values but later it was not done given incompatibilities with the computer where tests were run.

[4] In Ruby, it was used the gem 'PAPI' to be able to access PAPI through that programming language.

First experiment – C/C++ vs Ruby

For this experiment, the first version of the algorithm was implemented both in C/C++ and Ruby. Both were run with square matrices of dimensions from 600x600 to 3000x3000 with increments of 400.

As it should be predicted, looking at both implementation execution times values it can be seen that Ruby is considerably slower than C/C++ which is coherent with the fact that the percentage of cache misses in Ruby are much higher, L1 for example, is roughly 35% higher. This could being explained as Ruby is an interpreted language that takes intermediate steps to execute one operation and thus making more memory accesses during execution time and consecutively contributing to a faster out of date CACHE. This results in a much lower performance and consecutively faster growth on the polynomial curve that describes execution times.

Second experiment – first vs second version

For this experiment were considered the two versions of the algorithm implemented in C/C++. The results were registered for input matrices from 600x600 to 3000x3000 elements with increments in both dimensions of 400.

Here, as expected by the theoretical analysis, the number of cache misses for the second algorithm implementation is much lower. Specially when comparing level 1 cache (33% lower). Both cache level 2 reveal almost no misses.

Later, both versions of the algorithm were executed from 4000x4000 to 10000x10000 with intervals of 2000. This time L2 reveals also to have lesser misses on the second algorithm. This could be explained by the fact that if on the first tests CACHE (L1+L2) could be capable of storing all of the matrices resulting only in major misses at level 1, now given that the data involved is much larger, misses at level 2 are frequent and the differences between both algorithms are visible.

Third experiment – multithreading

Finally, at this third experiment, OpenMP was used to extend the single thread implementations to multi thread ones. This was achieved by adding

```
#pragma omp parallel for num_threads(n_threads) private(k) private(j)
```

at the beginning of each algorithm version, where $n_{threads}$ would vary between 1 and 4. This way, OpenMP splitted the outer loop into $n_{threads}$ threads and executed them in parallel. OpenMP *private(k)* and *private(j)* options are required to ensure that this two variables are not shared between threads.

Given that the outer loop is split in $n_{threads}$ it expected that the total amount of the execution time drops to $1/n_{threads}$ of the single threaded execution time, yet when looking at performance results that is only clearly seen for the 3 first iterations (1, 2 and 3 threads) but when using 4 threads the results are much similar to the 3 threads version. This could be explained by the fact that when dividing into more threads data sections tend to be smaller and the false sharing phenomena[5] starts occur more often.

And once again the percentage of cache misses at the second algorithm version are much lower, now being at the order of 1% at L1 and 0.1% at L2.

Conclusions

After all this experiments three main conclusions could be taken:

1. Ruby is a slower language than C/C++. This is not a novel conclusion since it is known that interpreted languages are in general slower than compiled languages and that is its biggest downfall.
2. An algorithm that manages well CACHE memory may represent huge improvements on the performance. Having impact both in single threaded versions and multi threaded versions.
3. Increasing the number of cores that execute the program infinitely does not bring infinitely performance improvements as solving data synchronization brings additional an overhead.

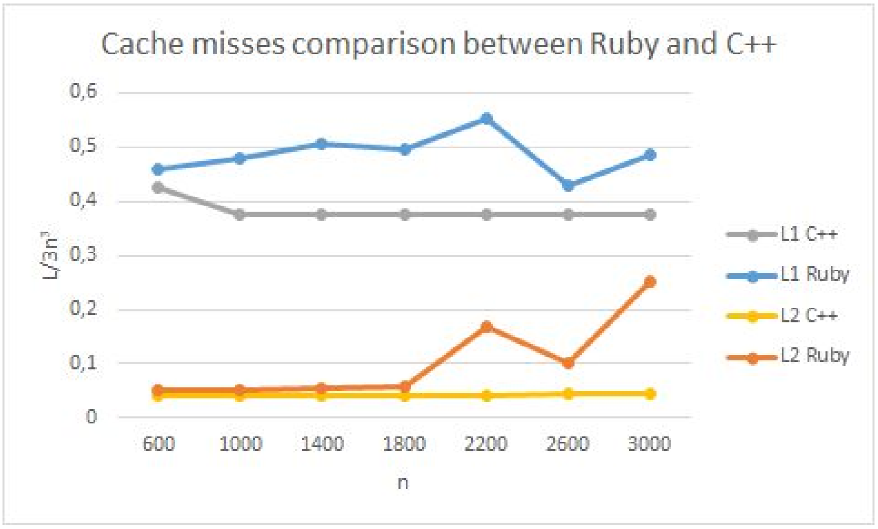
References

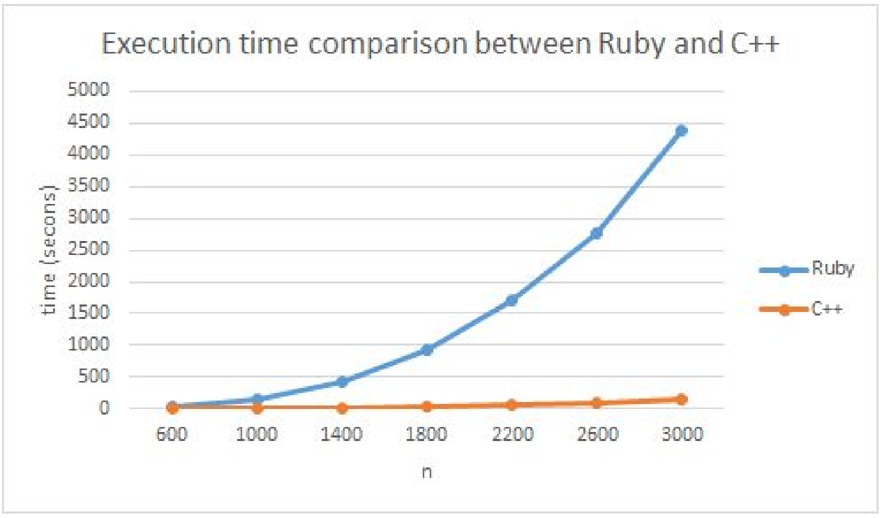
1. https://en.m.wikipedia.org/wiki/Matrix_representation
2. http://stackoverflow.com/questions/8620303/how-many-bytes-does-a-xeon-bring-into-the-cache-per-memory-accchunksess
3. http://stackoverflow.com/questions/7284179/is-ruby-a-scripting-language-or-an-interpreted-language
4. https://github.com/Nanosim-LIG/papi-ruby
5. https://icl.cs.utk.edu/projects/papi/wiki/PAPITopics:Getting_Started
6. http://openmp.org/wp/
7. http://icl.cs.utk.edu/papi/
8. https://github.com/Nanosim-LIG/papi-ruby

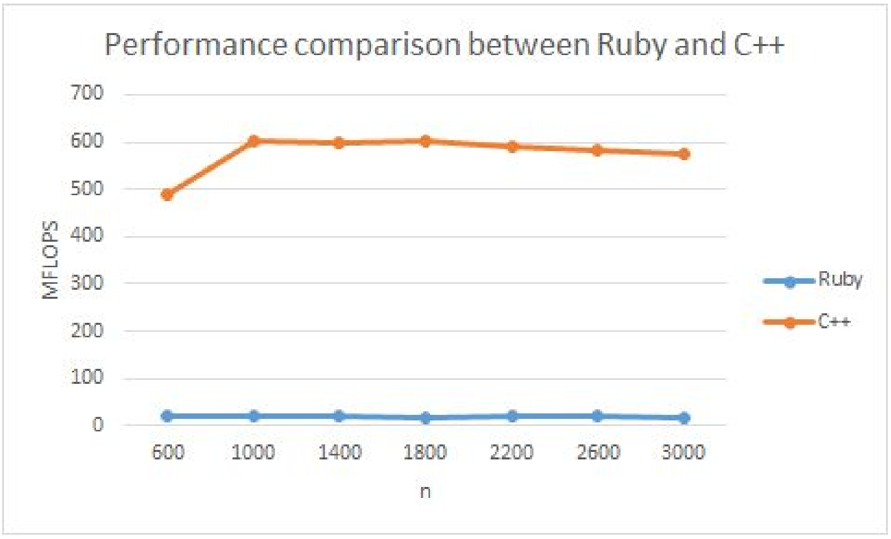
[5] Each core has its own CACHE. When multiple cores share the near data, given the way CACHE loads neighbours bytes in chunks, that data could exist replicated through the different caches. If one changes some byte value, the changes must be synchronized with remaining cores' CACHE that could potentially never be used there – also known as false sharing.

Appendix

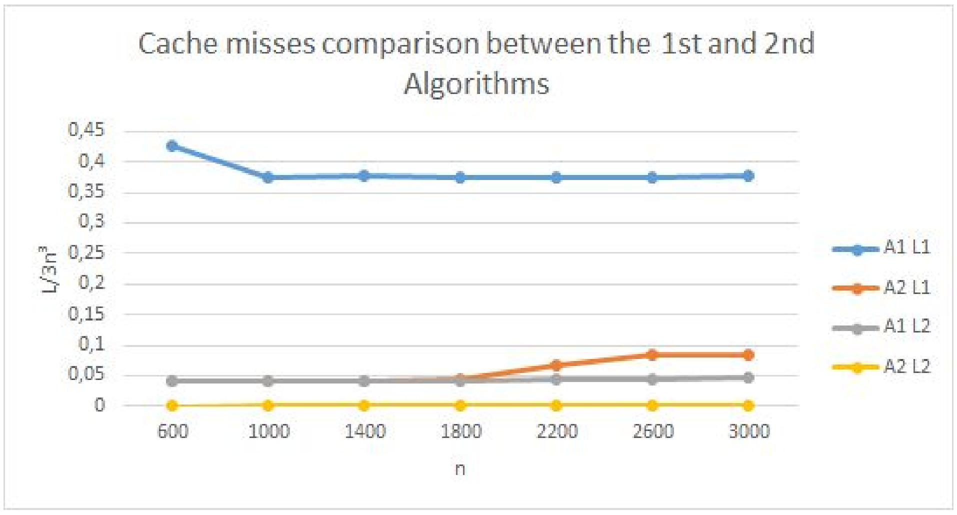
First experiment graphs

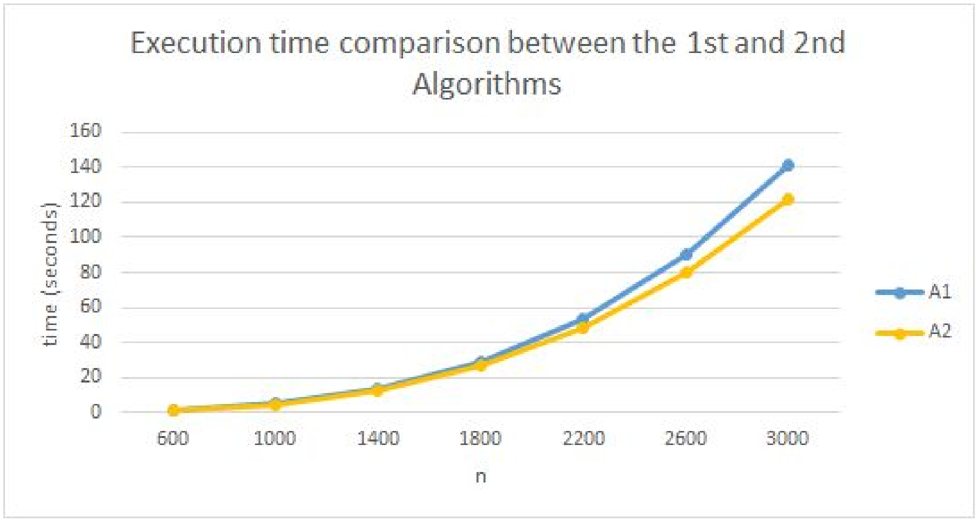


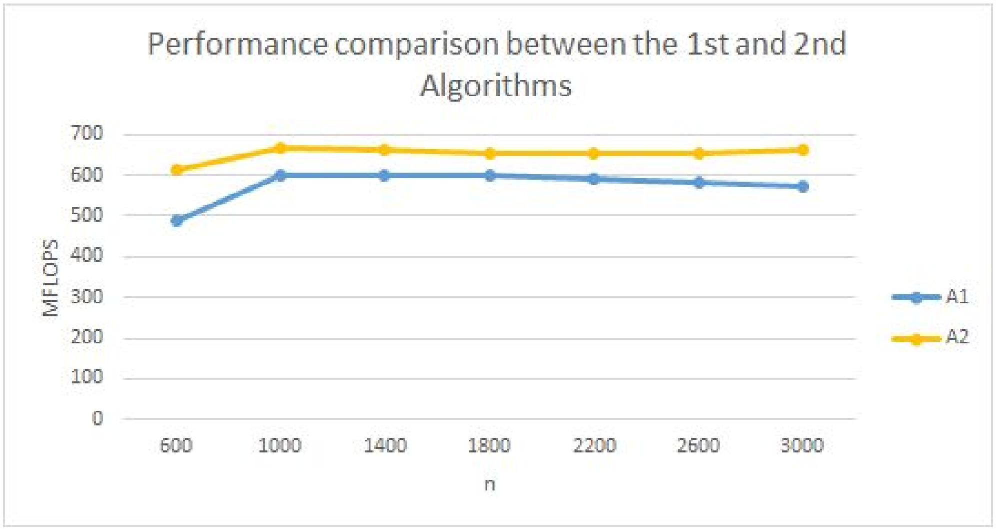


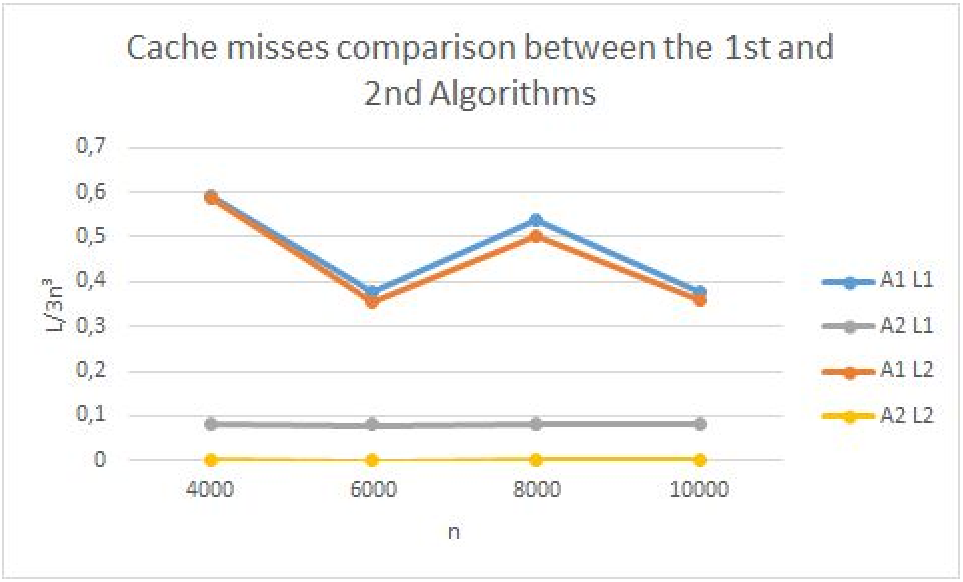


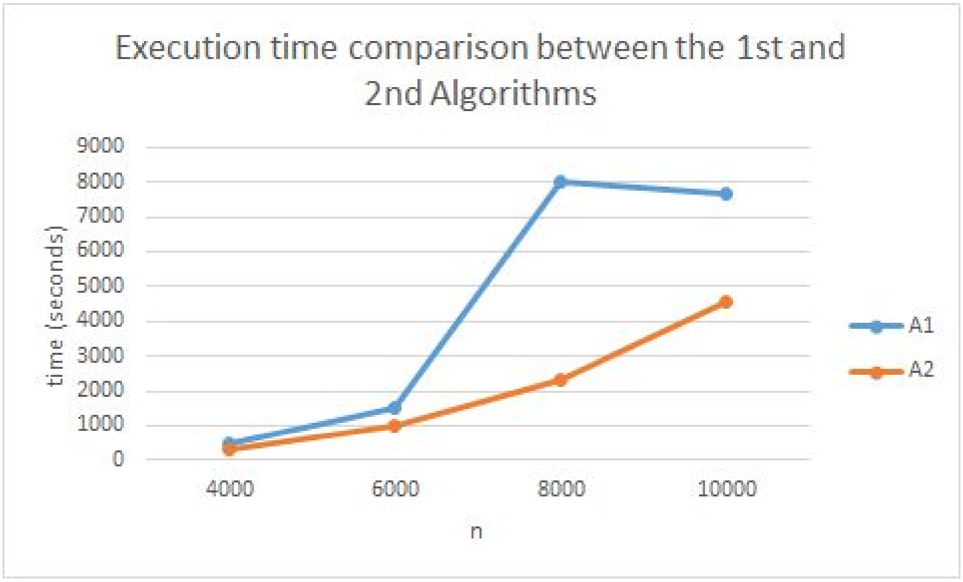
Second experiment Graphs

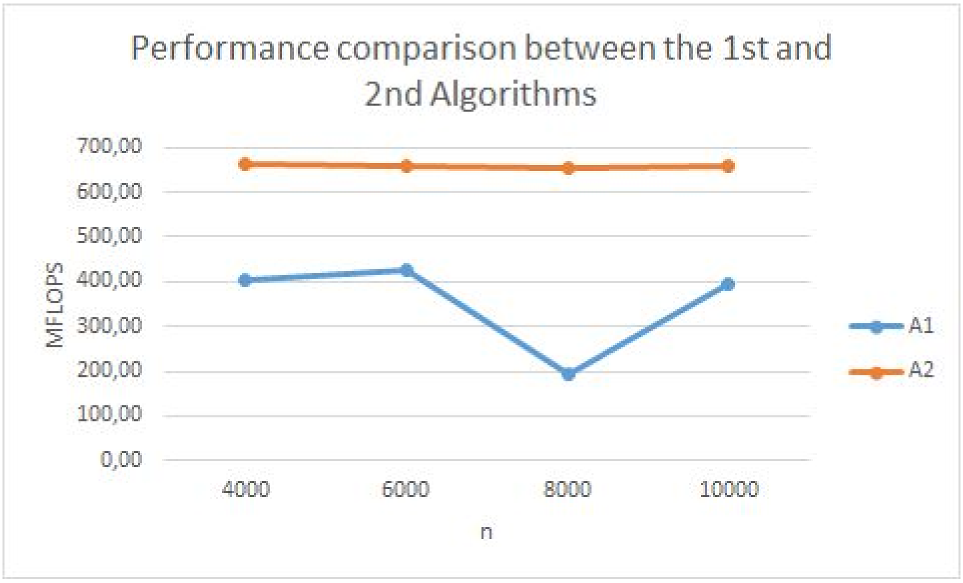




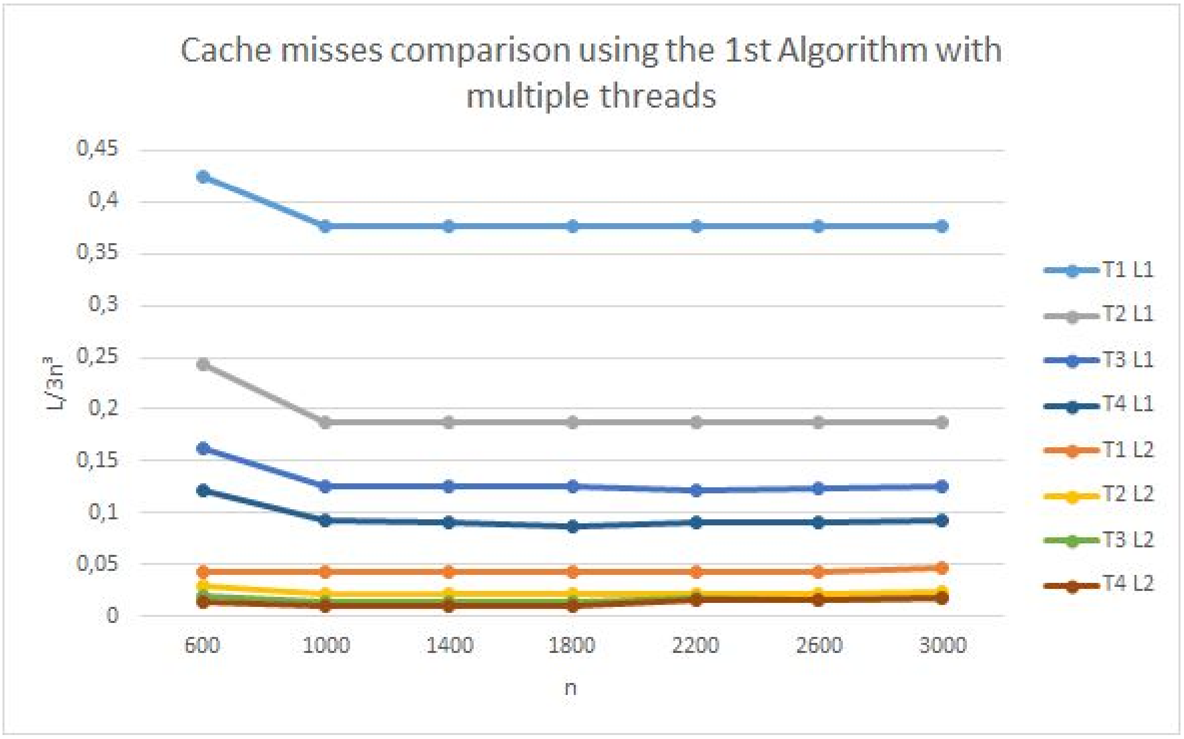


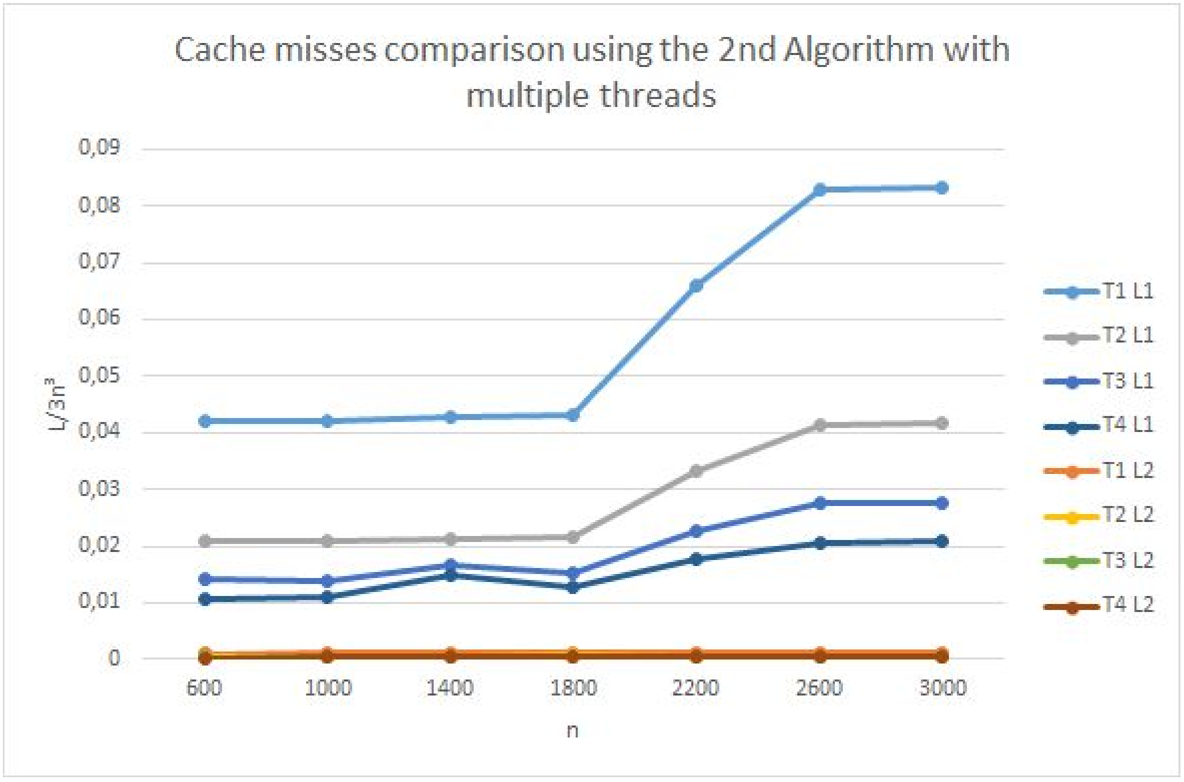

Cache misses comparison between the 1st and 2nd Algorithms
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0
$L/3n^3$
4000
6000
8000
10000
n
A1 L1
A2 L1
A1 L2
A2 L2

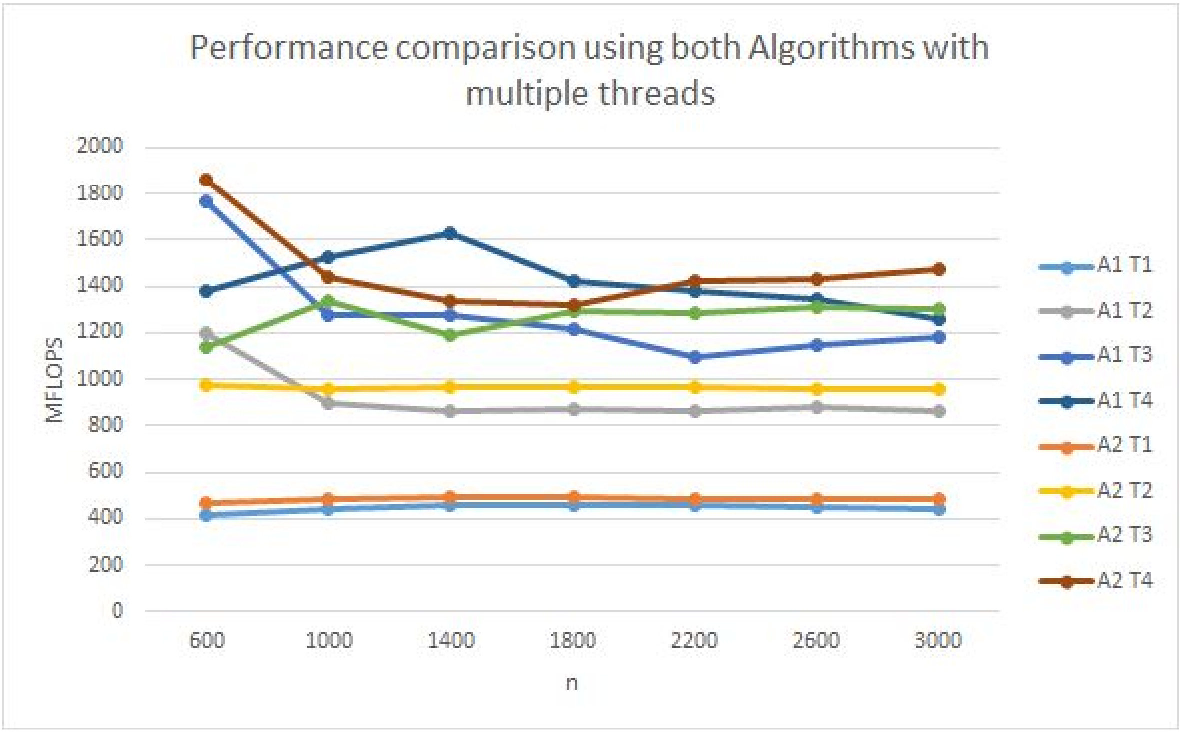

Execution time comparison between the 1st and 2nd Algorithms
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
time (seconds)
4000
6000
8000
10000
n
A1
A2

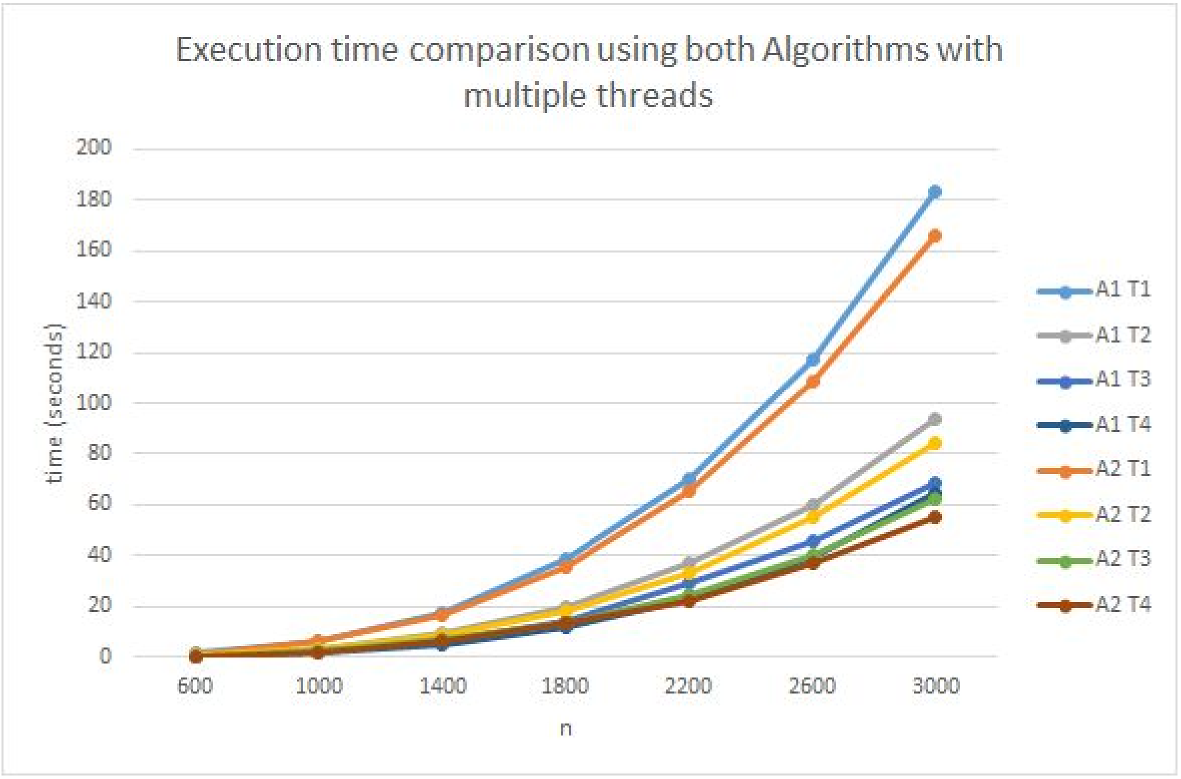

Performance comparison between the 1st and 2nd Algorithms
700,00
600,00
500,00
400,00
300,00
200,00
100,00
0,00
MFLOPS
4000
6000
8000
10000
n
A1
A2

Third experiment graphs






Performance comparison using both Algorithms with multiple threads
MFLOPS
2000
1800
1600
1400
1200
1000
800
600
400
200
0
600
1000
1400
1800
2200
2600
3000
n
A1 T1
A1 T2
A1 T3
A1 T4
A2 T1
A2 T2
A2 T3
A2 T4


Execution time comparison using both Algorithms with multiple threads
time (seconds)
200
180
160
140
120
100
80
60
40
20
0
600
1000
1400
1800
2200
2600
3000
n
A1 T1
A1 T2
A1 T3
A1 T4
A2 T1
A2 T2
A2 T3
A2 T4