Faculty of Engineering of the University of Porto

U.PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

First Project

Performance Evaluation of a Single Core

Parallel and Distributed Computing

Bachelor's degree in Informatics and Computing Engineering

Class 2 - Group 4

André Tomás da Cunha Soares - up202004161

Carlos Alberto Ferreira Teles de Sousa - up202005954

Diogo Alexandre da Costa Melo Moreira da Fonte - up202004175

Index

1. Problem Description and Algorithms Explanation

The aim of this practical work is to study the effect of memory hierarchy on computer performance when accessing large amounts of data. To test it, we will use 3 algorithms for calculating products of two matrices. We collected performance indicators relevant to the execution of programs with PAPI (Performance API).

Basic Multiplication:

The first algorithm we implemented is the most basic of matrix multiplication, as it just consists of multiplying each row of the first matrix by each corresponding column of the second matrix.

Code:

```
for(i=0; i<m_ar; i++){
        for( j=0; j<m_br; j++){
                temp = 0;
                for( k=0; k<m_ar; k++){
                        temp += pha[i*m_ar+k] * phb[k*m_br+j];
                }
                phc[i*m_ar+j]=temp;
```

Line Multiplication:

This version is a variation of the Basic Multiplication algorithm, in which an element of the first matrix is multiplied by the corresponding line of the second matrix and accumulates the sum in the respective position of the result matrix.

Code:

```
for(i=0; i<m_ar; i++){
        for( j=0; j<m_br; j++){
                temp = 0;

                for( k=0; k<m_ar; k++){
                        phc[i * m_ar + k] += pha[i * m_ar + j] * phb[j * m_br + k];
                }
        }
```

Block Multiplication:

The third algorithm uses a block-oriented approach that divides the two matrices in blocks and calculates the values of the result matrix with Line Multiplication.

Code:

```
for (ii = 0 ; ii < m_ar ; ii += bkSize)
        for (jj = 0 ; jj < m_br ; jj += bkSize)
                for (kk = 0 ; kk < m_ar ; kk += bkSize)
                        for (i = ii ; i < ii + bkSize ; i++)
                                for (j = jj; j < jj + bkSize; j++)
                                        for (k = kk ; k < kk + bkSize ; k++)
                                                phc[i * m_ar + k] += pha[i * m_ar + j] *
phb[j * m_br + k];
```

2. Performance Metrics

To measure the performance of the processor in the given problem, we compared the results using two different programming languages, and some relevant performance indicators.

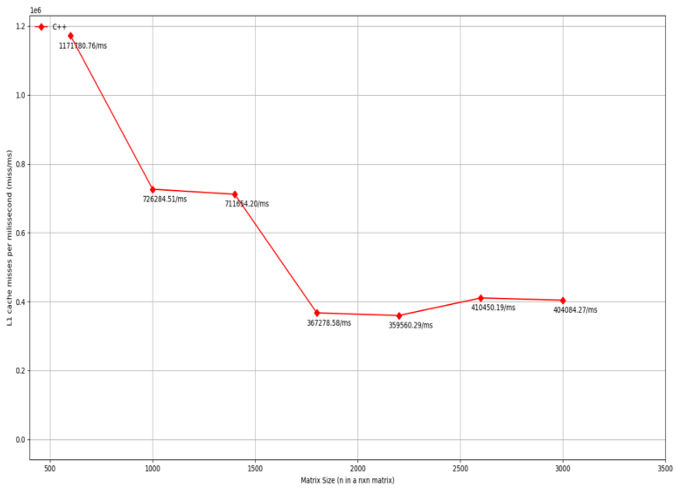
The central programming language studied is C++, supported by the Performance API (PAPI) to collect the values of **data cache misses per millisecond** in the Level 1 and Level 2 caches and we compared the number of **instructions per cycle**. As for our second programming language we opted for Java, for measuring and comparing the **time** performance, with System.nanoTime(), against C++ with Chrono. Both measure in nanoseconds for more precision.

So that our study is more consistent, we decided to take a considerable number of samples to have more reliable results. We tested the programs with several matrix sizes, as was required in the proposal.

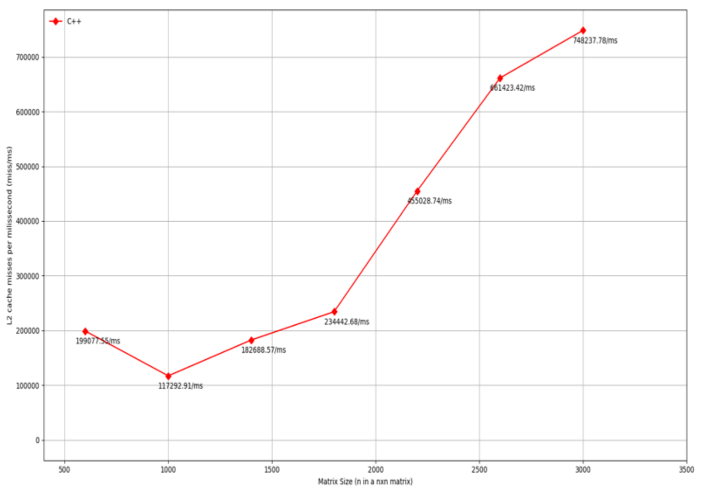
3. Results and Analysis

Basic Multiplication:

- **C++ Performance**

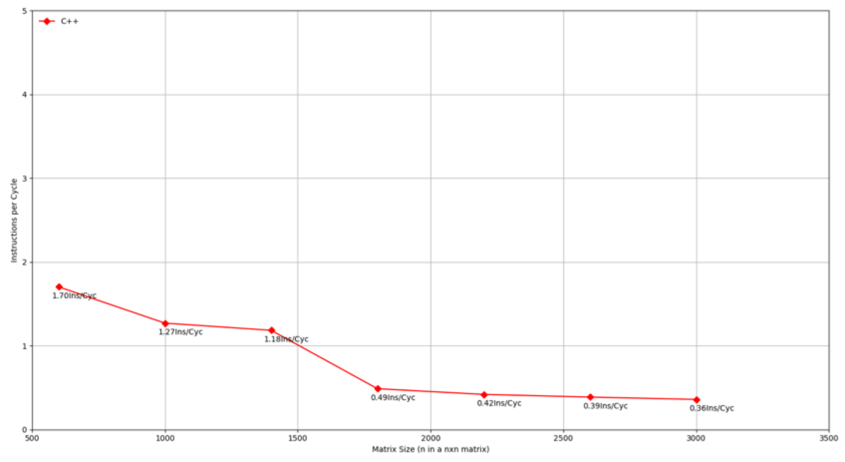


(DotMultL1Misses.png)



(DotMultL2Misses.png)

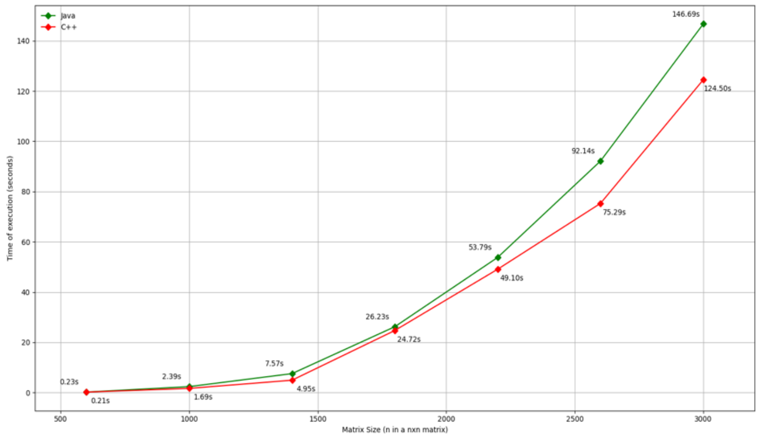
As expected, we noticed that the L1 Cache Misses per millisecond decrease along with the increase of the matrix size. However, the L2 Cache Misses per millisecond increase, so we conclude that they are inversely proportional. This happens because the L1 Cache is checked in first place and after the L1 miss, it is confirmed (or not) in the L2 Cache.



(DotMultInstructionsCycle.png)

Analyzing this graph, we can conclude that with the increase of the matrix size, the number of instructions per cycle decreases (lightly in each increment of the matrix size). This happens because the CPU is under a lot of stress to run the matrix product and reduces the number of instructions to solve that program instead of others.

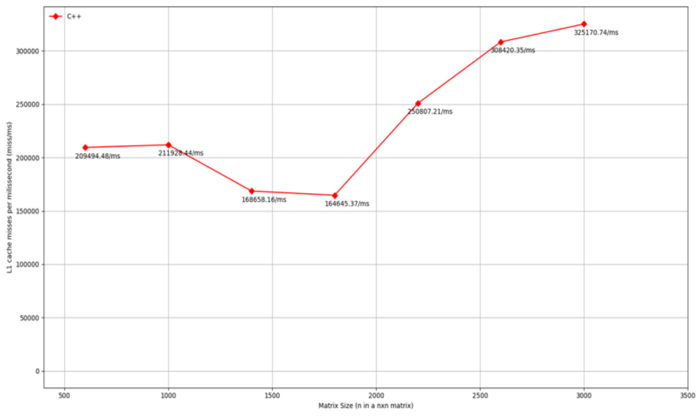
- **Comparison with Java Performance**



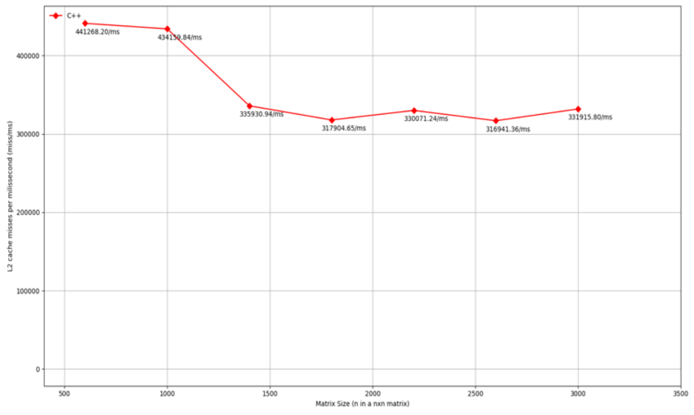
(DotMultC++Java.png)

Line Multiplication:

- **C++ Performance**

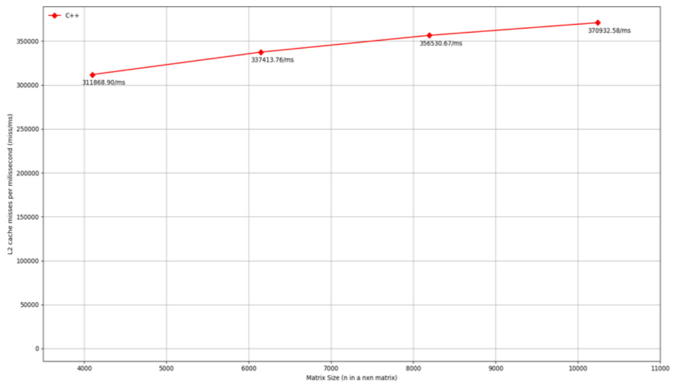


(LineMultL1Misses.png)

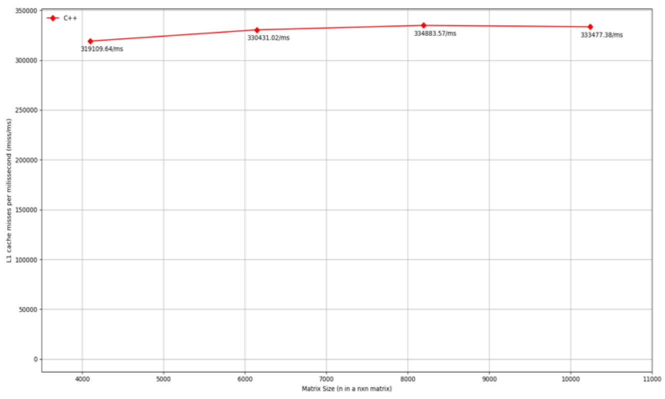


(LineMultL2Misses.png)

There is a significant improvement in all performance metrics with this algorithm compared to the Basic Multiplication algorithm. This happens since Line Multiplication takes advantage of C++ memory allocation.

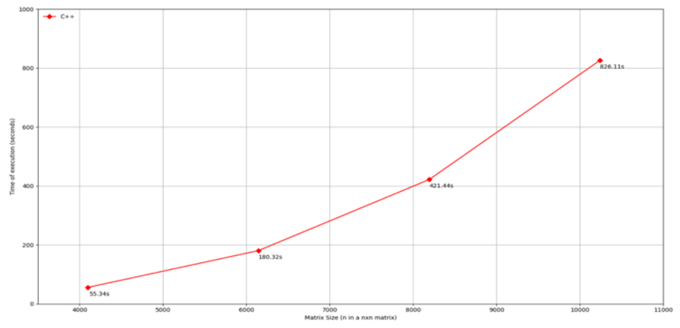


(LineBiggerL1Misses.png)

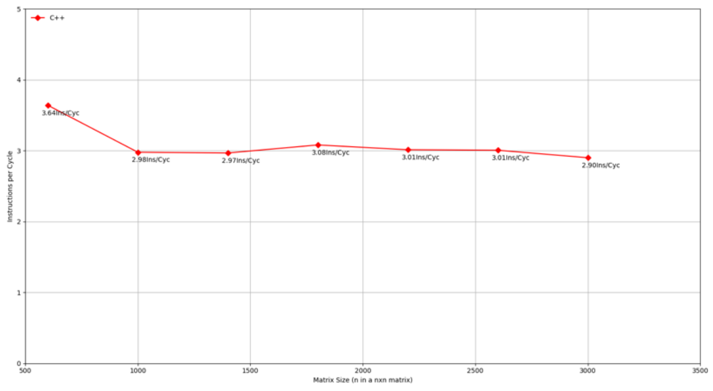


(LineBiggerL2Misses.png)

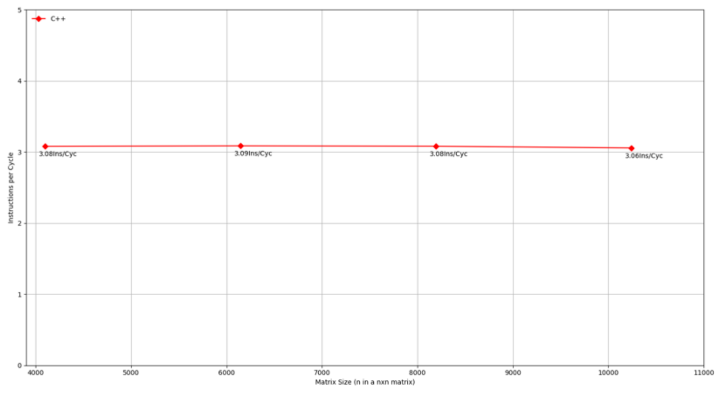
As expected, when we try the algorithm with bigger matrices, the Cache Misses increase, so we can conclude that the bigger the matrix the size, the more memory accesses.



(LineMultC++BiggerMatrixes.png)



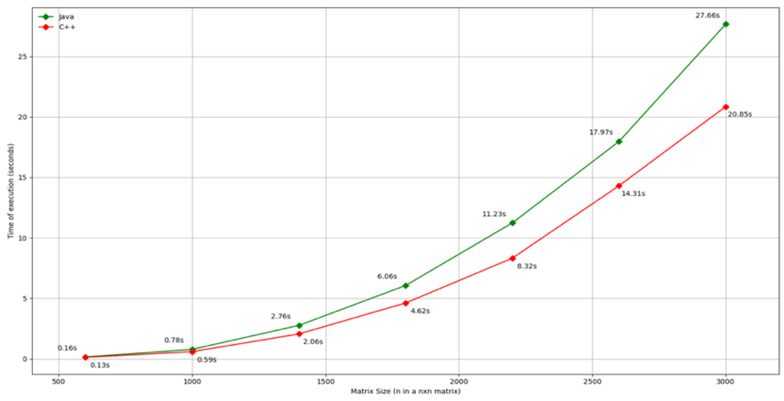
(LineMultInstructionsCycle.png)



(LineBiggerMultInstructionsCycle.png)

With the analysis of this graphics, we can conclude that the Instructions per Cycle doesn´t differ a lot from each other, by chance they are the same in the case where we use matrices of larger dimensions, as shown in the graph on the right.

- **Comparison with Java Performance**

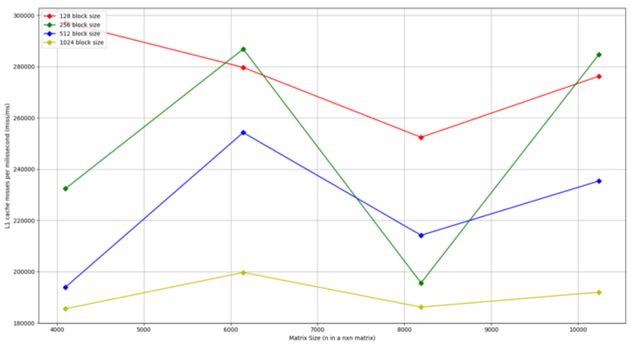


(LineMultC++Java.png)

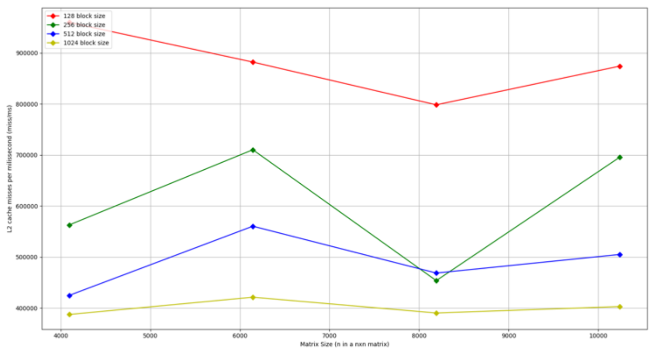
As expected, the times improved in both languages, but C++ continues to hold the lead.

Block Multiplication:

- **C++ Performance**

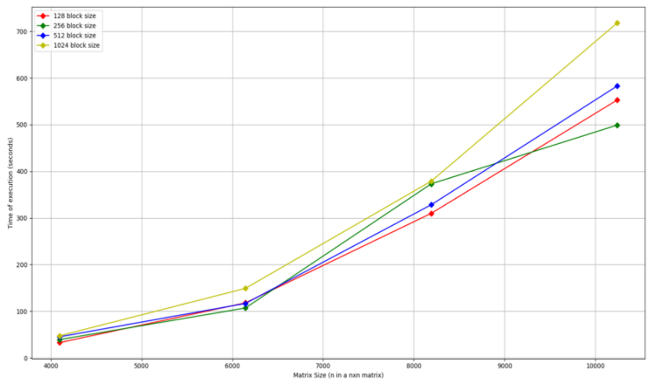


(L1MissesBlock.png)

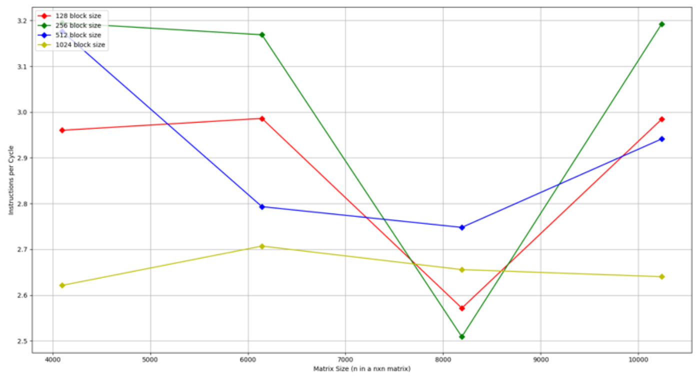


(L2MissesBlock.png)

With these graphs, we can conclude that the larger the block size, the smaller the number of Cache Misses, in both L1 and L2 Caches. Since when we read more data in each block read, we can have more Cache Hits.



(BlockMultSeconds.png)



(BlockMultInstructions.png)

In terms of time, the block size doesn't have a great impact, but the matrix size has. As expected, the run time increased with the matrix size, proportionally. With the block size increase, the number of instructions per cycle drops significantly, as we can see in the graphics on the right.

4. Conclusions

With this project we perceive various important metrics when we need to take program performance into consideration. Memory management in code is very important because we can take advantage of memory layout and locality to improve the performance of these algorithms. Since each of the three matrix multiplication algorithms has the same time complexity, the memory management makes the difference.