U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Computação Paralela e Distribuída

L.EIC028 2022/2023

CPD project1

Performance evaluation of a single core

Grupo T11G12

David dos Santos Ferreira - up202006302
Filipe de Azevedo Cardoso - up202006409
Luís Pedro Rodrigues de Morais - up200800621

1. Introduction

In this project, we will study the effect on the processor performance of the memory hierarchy when accessing large amounts of data, using three different algorithms of matrix multiplication and two different programming languages, such as C++ and Java.
The performance comparison between the algorithms will be done using time comparison for both programming languages and using an API (PAPI) to collect metrics and then analyze the results only for C++.

2. Algorithm Explanation and Implementation

2.1 Algorithm 1 - Simple Matrix Multiplication

For this problem, the C++ algorithm was already provided so we only have to implement the function ***onMult()*** in Java***.***
This algorithm is an implementation of the simple algebra matrix multiplication that multiplies one line of the first matrix by each column of the second matrix in Java.

Implemented on matrixproduct.java lines 36 to 44.

2.2 Algorithm 2 - Line Matrix Multiplication

For this problem, we implemented the function ***onMultLine()*** both in C++ and Java.
This algorithm implementation consists in multiplying an element from the first matrix by the correspondent line of the second matrix.

Implemented on matrixproduct.java lines 87 to 94
Implemented on matrixproduct.cpp lines 115 to 122

2.3 Algorithm 3 - Block Matrix Multiplication

For this problem, we implemented the function **onMultBlock()** in C++.
This algorithm implementation consists in a block-oriented algorithm that divides the matrices into blocks and uses the same sequence of computation as in **Algorithm 2**.

Implemented on matrixproduct.cpp lines 195 to 208

This way of multiplying matrices by blocks is expected to be the most efficient one, once it uses a block-based approach, which is very useful when matrices have high dimensions, so they cannot be stored as a whole in memory.
Therefore, we split the matrices into smaller blocks of smaller dimensions, so we can store data segments of the matrices in memory. In addition, we multiply the input matrices using **Algorithm 2** (product between an element and its corresponding line).
Although there is a slight increase in the amount of for cycle compared to previous algorithms, by dividing the matrices into smaller segments stored in memory and multiplying those segments using **Algorithm 2** we expect an increase in terms of performance.

3. Performance Metrics

3.1 Time

To evaluate and compare an algorithm's performance, time is one most valuable metrics. We used the time to compare the algorithm’s implementation between C++ and Java.

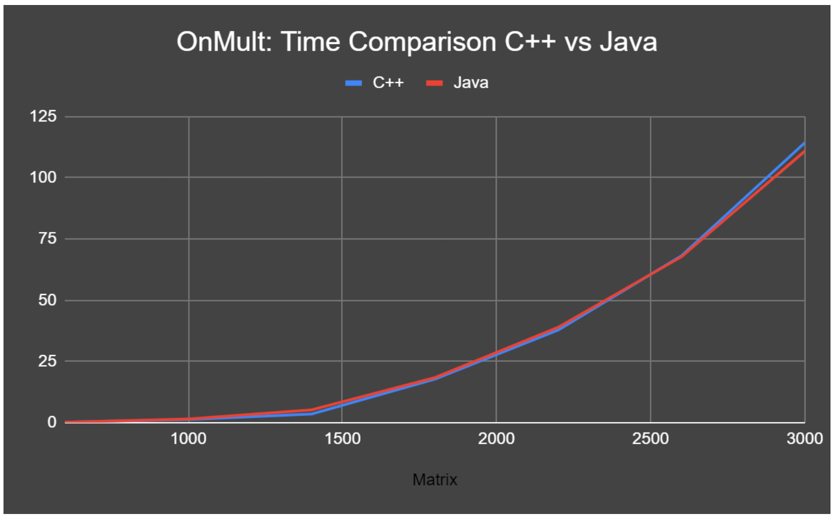
3.2 L1DCM and L2DCM

In order to evaluate the performance between the different C++ algorithms, Performance API (PAPI) was used to analyze cache misses.
To calculate the matrices it is necessary to access a certain position of the matrix, so it is important for the performance of the algorithm that the data is cached to allow faster access.
A cache miss occurs when the data the system requests does not exist in cache memory.
L1 cache is faster than L2 cache because it's in the CPU package, but L2 cache has more capacity.
For an algorithm to be faster, we must take advantage of the cache memory and try to reduce the number of cache misses.

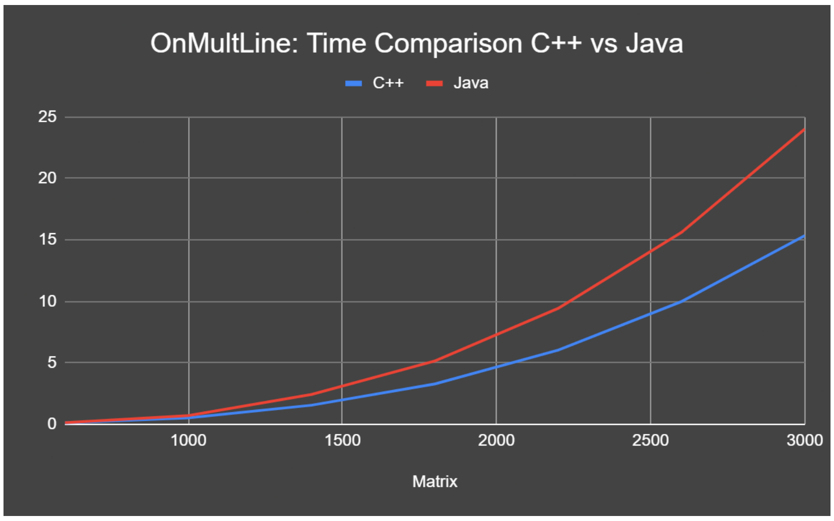
4. Results and Analysis

4.1 Time Comparison between C++ and Java

Simple Matrix Multiplication algorithm implementation C++ and Java languages are compared in the graph below. C++ an Java time performance is almost the same in this algorithm.

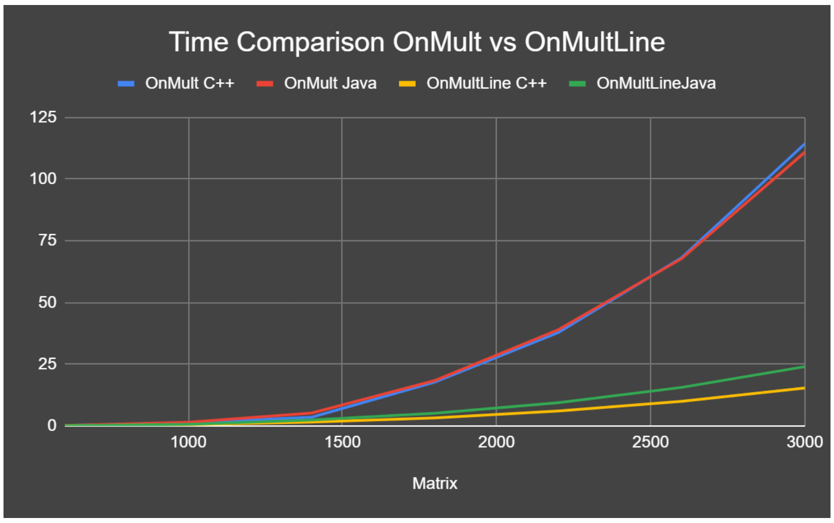


Line Multiplication algorithm implementation C++ and Java languages are compared in the graph below. In this algorithm C is significantly faster than Java.



4.2 Time Comparison between Algorithms

In the graph below we compare the time between Algorithm 1 and Algorithm 2 in bot Java and C++.
We can easily conclude that Line Multiplication is much faster than the Simple Multiplication Algorithm.
Further explanation will be discussed in L1DCM and L2DCM comparison.



4.3 Data Cache Misses Comparison

4.3.1 Algorithm 1 vs Algorithm 2

Analyzing the graph below that compares ***Algorithm 1*** and ***Algorithm 2*** L1DCM and L2DCM, we can conclude that Line Multiplication leads to significantly fewer data cache misses in both L1 and L2.
Using data cache misses and time comparison results, we can see that more data cache misses leading to more time for the algorithm to run.
This result is expected because the implementation of the Line Multiplication algorithm takes advantage of the data in the cache. In contrast to the simple matrix algorithm that, at each iteration, has to read all the values back into the cache.
We can conclude that the Line Multiplication algorithm is faster because it has fewer data cache misses.

On the second algorithm, execution time values are better than the first algorithm.
Considering that in memory access a phenomenon called Spatial Locality occurs, classified by the following parameters:

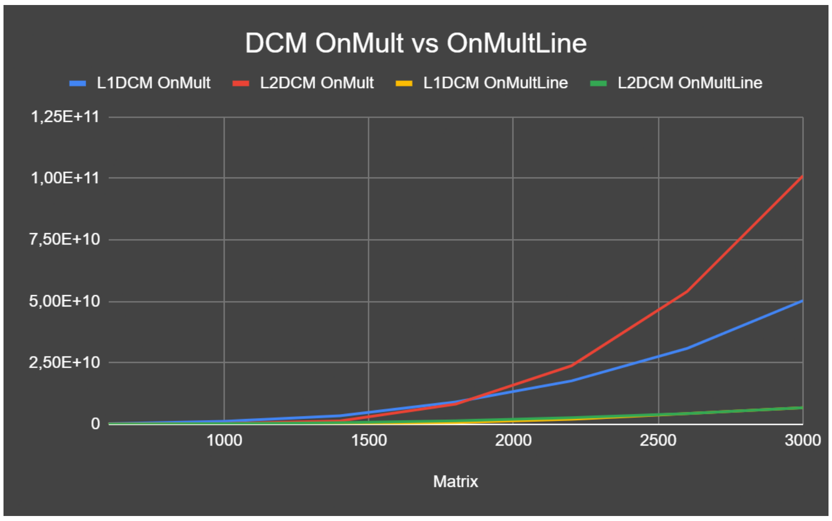
- When an element in memory is requested its neighbors will also be requested.
- A cache line is read in a single operation

- It is efficient to request elements that are on the same cache line.

Since a 1-dimensional array is used, elements in the same row will always be stored continuously, so accessing memory to get elements from a given row is faster than the process to get a given column. For this reason, matrix multiplication using the method of the product of an element by the corresponding row is more efficient than the product of a given row by a given column.

Indeed, we can conclude that thanks to this approach the number of cache misses is significantly reduced compared to algorithm one, thus improving performance, thanks to the reduction in execution time.

Regarding the performance in Java versus the performance in C++ it is possible to conclude that although both languages store the values of the arrays in contiguous memory positions, due to its low-level memory management and better compiler optimization the C++ language ends up being more efficient than JAVA.



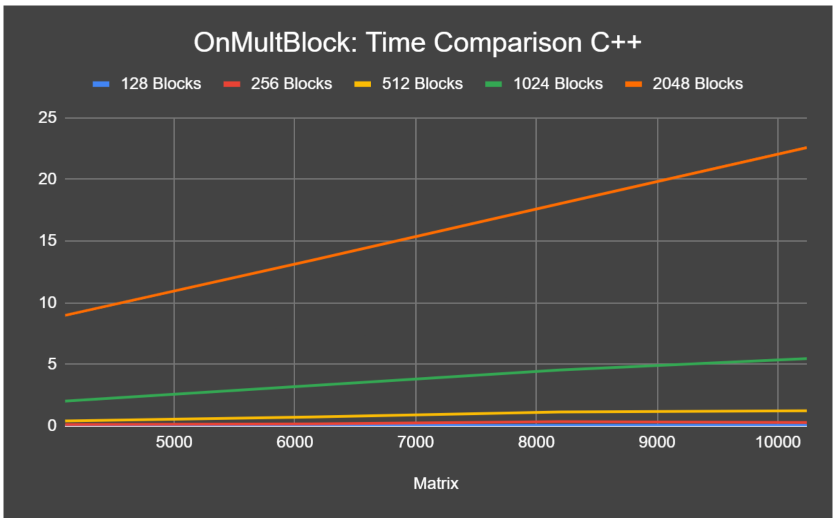
4.3.2 Algorithm 3 - Block Multiplication

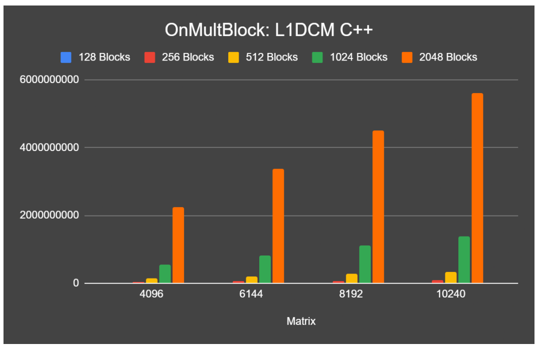
Analyzing the graph below that compares the time consumed by the execution of the Block Multiplication Algorithm, in different block sizes and matrix sizes, we can conclude that this algorithm is much faster than the other 2.

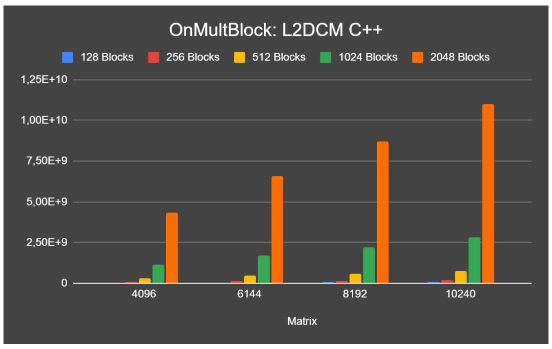
As can be seen, for a given size of an array the execution time is inversely proportional to the size of the blocks, that is, the larger the size of the blocks, the longer the execution takes. This is due to the fact that the number of data cache misses in caches L1 and L2 is greater as the block size increases. Still, since the cache misses in the L1 cache are resolved faster than the cache misses in the L2 cache and the number of data cache misses in L1 is always smaller than the number of data cache misses in L2 for any given block size, we can conclude that even if the block size increases we still have a shorter execution time than the algorithms in the previous paragraphs.

Although the instruction cache misses of L1 are larger than those of L2, these do not denote great expressiveness in the value of total cache misses, given their greatness in comparison to the greatness of the values of data cache misses. Furthermore, the fact that instruction cache misses in L1 are larger than in L2 is due to the fact that the cache misses counter in L1 is incremented at each fetch, while L2 is only incremented once, as can be concluded from the following article: https://sites.utexas.edu/jdm4372/2013/07/14/notes-on-the-mystery-of-hardware-cache-performance-counters/

In addition, it is also possible to verify that the execution time of this algorithm is lower than the execution time of the previous algorithms, as expected, this is due to the fact that the matrix multiplication algorithm reuses the same elements several times. Thus, if a given block does not fit in the cache it is necessary to access the main memory, thus increasing the execution time, since each access to the main memory is much slower than the access to memory, either in the L1 or L2 cache. However in case the block can fit in its entirety in cache memory, then it can be loaded into the cache, thus optimizing execution time, since several elements of a given array are loaded into memory at once, thus avoiding constant access to the main memory to get each element of a given array, minimizing cache misses. Finally, this approach coupled with the approach in exercise 2 makes matrix multiplication markedly improve its performance.


OnMultBlock: Time Comparison C++
128 Blocks
256 Blocks
512 Blocks
1024 Blocks
2048 Blocks
Matrix


OnMultBlock: L1DCM C++
128 Blocks
256 Blocks
512 Blocks
1024 Blocks
2048 Blocks
Matrix


OnMultBlock: L2DCM C++
128 Blocks
256 Blocks
512 Blocks
1024 Blocks
2048 Blocks
1,25E+10
1,00E+10
7,50E+9
5,00E+9
2,50E+9
Matrix