Faculdade de Engenharia da Universidade do Porto

U.PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

CPD Project 1
Performance Evaluation of a Single Core

Computação Paralela e Distribuída

3º ano Licenciatura em Engenharia Informática e Computação

Turma 6 | Grupo 1

Data: 10/03/2023

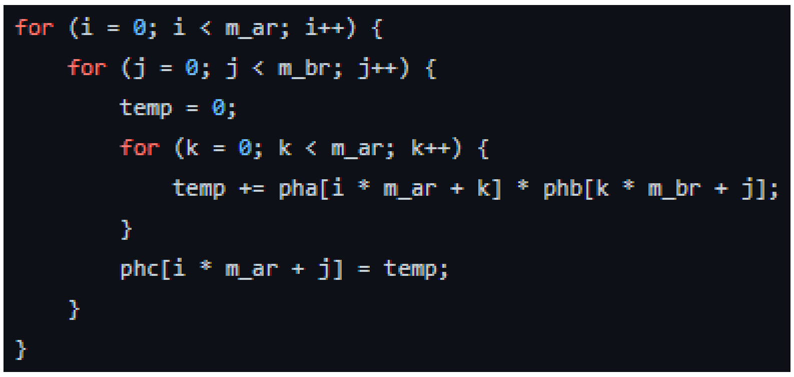
Project done by:
Diogo Filipe Ferreira da Silva (up202004288)
Henrique Oliveira Silva (up202007242)
Tiago Nunes Moreira Branquinho (up202005567)

Problem Description and Algorithms Explanation

This project consists in using the product of two matrices (with different algorithms) to study the effect on the processor performance of the memory hierarchy when accessing large amounts of data. The Performance API (PAPI) will be responsible for collecting relevant performance indicators of the program execution.
We decided to use Java as the secondary programming language for the first and second algorithms as the code syntax is very similar, following the same logic.
In most cases, C++ performs better than Java. This is justified by the fact that C++ programs are compiled directly into machine code, which can run faster than bytecode interpreted by a virtual machine (the case for Java). With that said, we expect better results from the C++ version but still close to Java.

1st Algorithm (Basic)

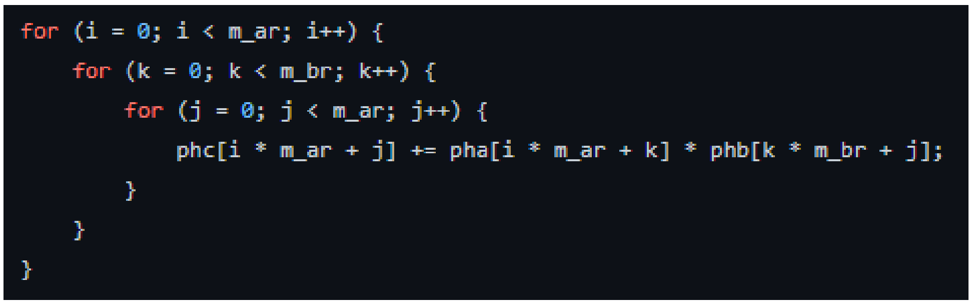
For the first algorithm, we simply need to iterate through each row of matrix A and multiply their elements with the corresponding ones in each column of matrix B, adding the results and filling matrix C. A temporary variable is used to store the sum of the products from row-column cell multiplication.



```
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;
    }
}
```

2nd Algorithm (Line Oriented)

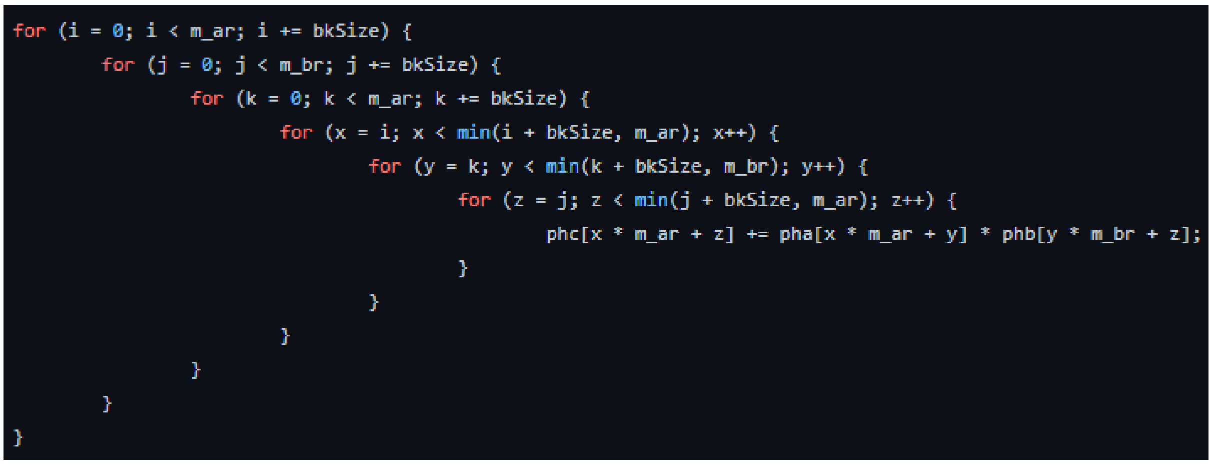
For the second algorithm, we accomplish the same as in the previous one by iterating over each element of matrix A, multiplying it by each element of the corresponding column in matrix B, and accumulating the result directly in the corresponding position in matrix C. We learned a more efficient way of applying this by also swapping the variables j and k as such (j would initially be used in the second “for loop”):



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

3rd Algorithm (Block Oriented)

For this last algorithm, we performed a variation of matrix multiplication called “block matrix multiplication”, which is optimized for large matrices by breaking them up into smaller, more manageable blocks. We achieve this by iterating through the elements that will be multiplied in each block and accumulating all the results in matrix C.



```
for (i = 0; i < m_ar; i += bkSize) {
        for (j = 0; j < m_br; j += bkSize) {
                for (k = 0; k < m_ar; k += bkSize) {
                        for (x = i; x < min(i + bkSize, m_ar); x++) {
                                for (y = k; y < min(k + bkSize, m_br); y++) {
                                        for (z = j; z < min(j + bkSize, m_ar); z++) {
                                                phc[x * m_ar + z] += pha[x * m_ar + y] * phb[y * m_br + z];
                                        }
                                }
                        }
                }
        }
}
```

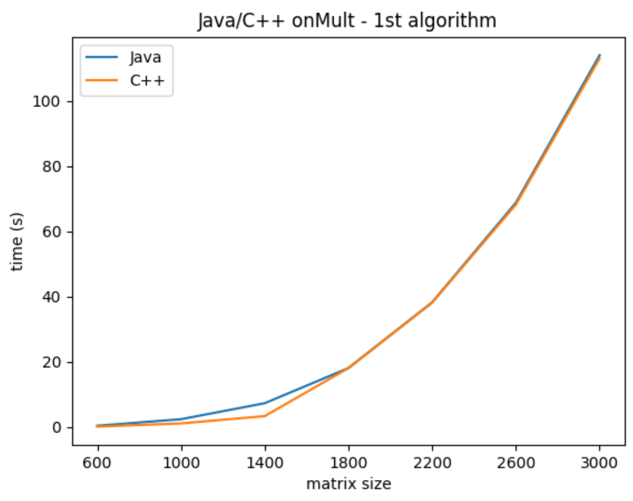
Performance Metrics

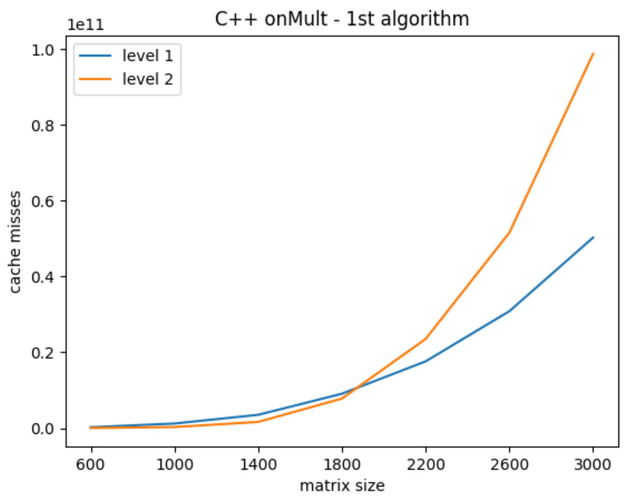
With the objective of measuring the processing performance, we used algorithms developed in two distinct languages, C++ and Java, for comparison terms, and gathered performance indexes we deemed relevant.
In both languages, elapsed time was registered (the faster the program, the better its performance); as for C++ only, we resorted to the Performance API (PAPI) to collect the amount of data cache misses (DCM) of the Level 1 (L1) and Level 2 (L2) caches. Data cache misses are relevant as they indicate how efficiently the program is using the CPU cache.

In order to realize these measurements, we ran the code in the computers from FEUP classrooms. In these machines, there are the L1 cache (32KB for data; 32KB for instructions) and the L2 cache (256KB), per core.

Results and Analysis

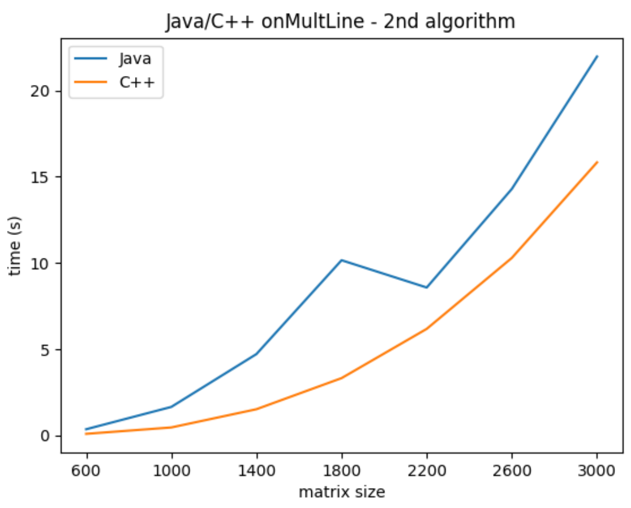
1st Algorithm (Basic)

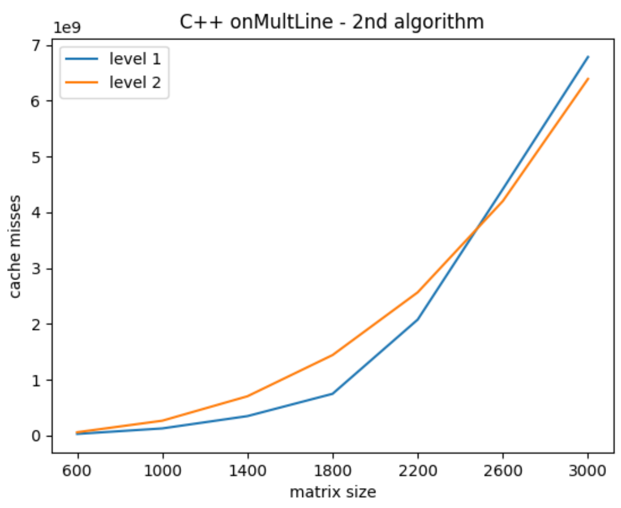




As we can observe, for higher values of matrix dimensions, the time it takes to execute the operations increases significantly. Also, as the matrices get bigger, there are obviously going to occur more memory accesses, which generally implies more data cache misses.
With this algorithm, each element of the resulting matrix is computed by summing the product of corresponding elements in the 2 input matrices over a row and column, respectively. This logic requires accessing multiple rows of one matrix and multiple columns of the other, which can result in poor memory locality and cache thrashing.
The results for C++ and Java were very similar as we can verify.

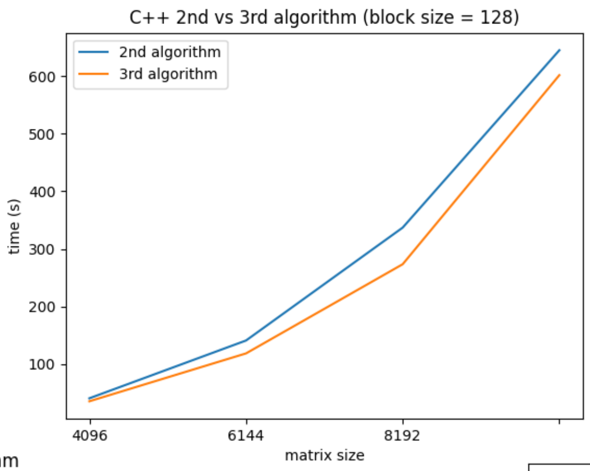
2nd Algorithm (Line Oriented)

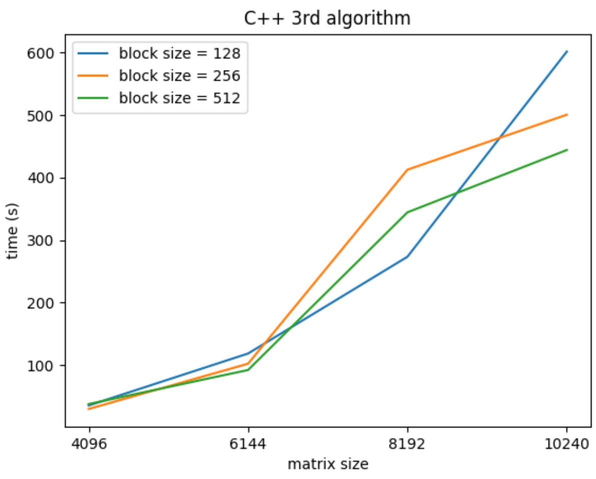


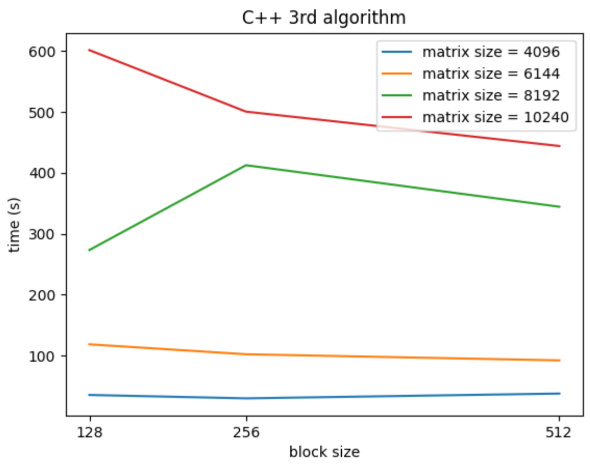


For the second algorithm, as we expected, the time of execution and the cache misses increase with larger matrix sizes. However, this line oriented algorithm produces better results than the previous one, having better memory allocation. For cache misses for instance, we can prove that by looking at the numbers magnitude (top of the graph), which is lower than for the first algorithm.
This is because here the program accesses the same memory location multiple times before moving to the next line. Also, we are accumulating the result in the final matrix directly rather than computing a temporary variable and then assigning it to the array. This eliminates unnecessary memory access, leading to improved performance.

3rd Algorithm (Block Oriented)



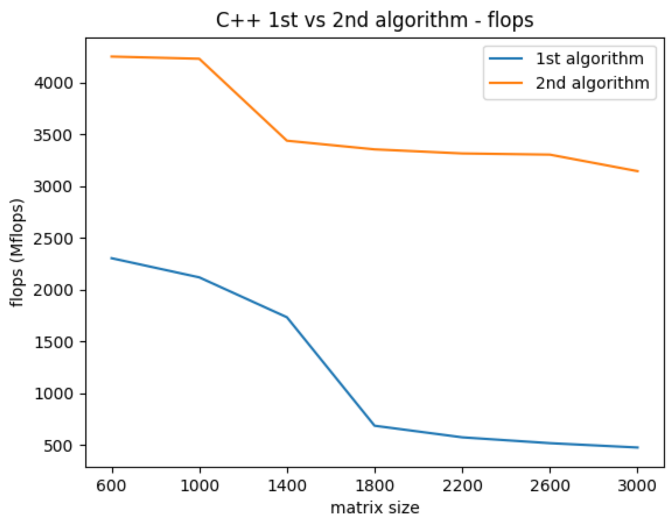




For the last algorithm, there’s again the correlation between execution time, cache misses and matrix sizes. We generally verify that the higher the number of blocks used to multiply (for the same matrix dimensions), the less time it takes to calculate the final matrix, and so the better the performance of our program. This would make sense as we are dividing our tasks more and more with each value increase.
As we expected, this algorithm, of all three, is the one that performs better. This can be explained by the improvement of the cache utilization, as the computation is performed on smaller submatrices which means the working set of data that the program needs to access at any given time is smaller.

Additionally, the data that the program needs to access is closer together in memory which reduces the time spent waiting for data to be loaded from memory.
Finally, there are less floating point operations required, as multiplying 2 submatrices can be performed using less operations than multiplying entire matrices.

Floating Point Operations



Concerning the number of floating point operations per second, algorithms exhibiting superior time performances, as previously observed, tend to perform better. However, improper memory usage can result in the processor expending significant effort on memory accesses and cache coherence, ultimately decreasing the useful work conducted.
Comparing the results obtained from the algorithms utilized, we can verify a decrease in operations per second as the matrix sizes increase due to the need for more load/store operations. Nevertheless, except for the first algorithm (where we need to access higher levels of memory more frequently), a leveling can be seen in that decrease due to an improved memory cache reuse, producing better results.

Conclusions

This project allowed us to understand the impact of data cache access on the processor's performance and how an algorithm with good memory management will be significantly beneficial in processing demanding applications.