Parallel Computing

Computação Paralela e Distribuída 2024-2025

1st Lab Work

3LEIC11

Diogo Salazar Ramos (up202207954)

Gonçalo Nuno Santos Pires Barroso (up202207832)

Guilherme Silveira Rego (up202207041)

March 21, 2025

Index

Problem Description

Throughout this report we will be analysing the performance of the memory hierarchy when accessing large amounts of data, and to simulate the use of these substantial amounts of memory we have used the product of two matrices, which is a known problem for high memory requirements and which we have implemented in C++ and in Python.

Execution time was our main comparative measure between these two different languages, we also used Performance API (PAPI) to know the level 1 and level 2 cache misses of each algorithm in C++.

The objective is to identify and understand the best practices of optimizing performance of algorithms.

Algorithms

We were asked to develop three matrice's product algorithms, **Simple Matrix Multiplication**, **Line Matrix Multiplication** and **Block Matrix Multiplication**, in C++ and another language of our choice, we chose Python because it was a language were familiar with and the algorithms were easier to implementate, even though we knew we would have much longer execution times.

Simple matrix multiplication was already implemented in C++, so we only implemented line and block matrix multiplication, and for Python we were only asked to develop simple and matrix multiplication, but we also developed block multiplication.

In the results and analysis section we will see that there are really big differences in performance between algorithms and execution time between languages and also algorithms.

Simple Matrix Multiplication

This algorithm is the most basic approach, basically iterating through each row of the first matrix (pha) and each column of the second matrix (phb) and computing the dot products for the element of the result matrix. The algorithm has a time complexity of $\mathbf{O(n^3)}$

```
// 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;
        }
}
```

Figure 1: Simple Matrix Multiplication in C++

which explains why it can be computationally intensive.

Line Matrix Multiplication

This algorithm is very similar to the first one, but by reordering the loops and doing a line-by-line multiplication, we were able to be more cache-friendly and improve memory access. The complexity is also **O(n^3)**, but it improves the execution time and reduces cache misses.

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

Figure 2: Line Matrix Multiplication in C++

Block Matrix Multiplication

In this last algorithm, we basically divided the matrix into 'blocks' and performed the multiplication on these blocks. With this strategy we tried to minimize cache misses and reduce the execution time. The complexity is also **O(n^3)**, but looking at the results after testing, we confirmed that the execution time and cache misses were greatly reduced.

```
// Block Multiplication
for (i0 = 0; i0 < m_ar; i0 += bkSize) {
        for (k0 = 0; k0 < m_ar; k0 += bkSize) {
                for (j0 = 0; j0 < m_br; j0 += bkSize)  {
                        for (i = i0; i < min(i0 + bkSize, m_ar); i++) {
                                for (k = k0; k < min(k0 + bkSize, m_ar); k++) {
                                        for (j = j0; j < min(j0 + bkSize, m_br); j++) {
                                                phc[i * m_ar + j] += pha[i * m_ar + k] * phb[k * m_br + j];
                                        }

                                }
                        }
                }
        }
}
```

Figure 3: Block Matrix Multiplication in C++

Performance Metrics

In order to evaluate the performance of the 3 algorithms that we implemented, we used the execution time and the Performance API (PAPI) in C++, which allowed

us to measure the number of cache misses for L1 and L2, and with the variation of these values we could evaluate the efficiency of our code.

Compilation in C++ was optimized with the flag -O2, and with all the results of execution time for both languages (Python and C++) annotated in an excel sheet we compared them and we will do an analysis on the next section.

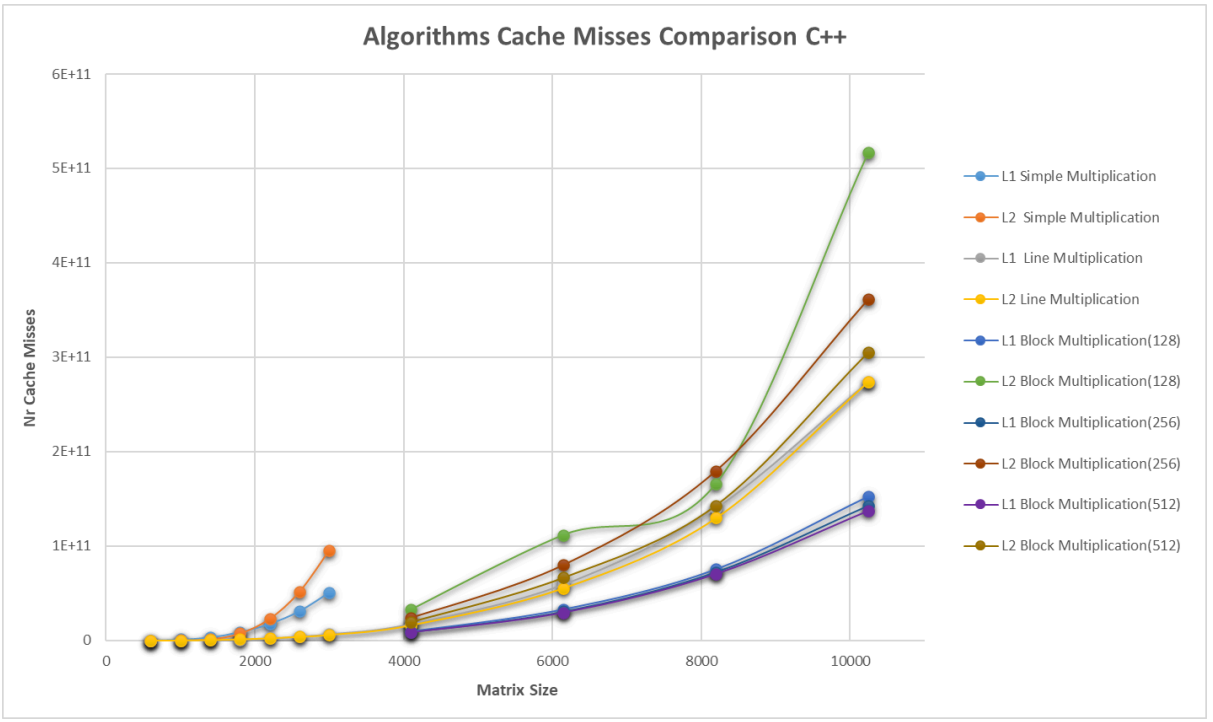
Also, to evaluate the difference between the 3 algorithms implemented in C++ we will use the execution time and number of cache misses to compare the improvement of performance.

Finally, to compare multi-core and single-core performance differences, we will also use execution time and number of cache misses to evaluate which parameters have improved from the single-core implementation to the multi-core implementation and thus understand the advantages or disadvantages of each implementation, also the speedup, Mflops and efficiency will be evaluated.

Results and analysis

● Algorithms Comparison in C++

1. **Execution Time**
    - As expected, **Simple Multiplication** had the worst execution time, increasing rapidly as matrix size grew, clearly due to its poor memory access pattern.
    - **Line Multiplication** significantly improved performance compared to the simple method. This is because it reuses loaded data more effectively, reducing unnecessary memory accesses.
    - **Block Multiplication** provided the best execution times overall, especially for larger matrices. By breaking the computation into blocks, it optimizes cache usage, reducing cache misses and improving speed,but will depend on hardware configuration and matrix sizes.
2. **Cache Misses**
    - As for execution time, **the Simple Multiplication** algorithm also suffered from the highest number of **L1 and L2 cache misses**.
    - **Line Multiplication** was an upgrade to the first algorithm, reducing L1 cache Misses and having fewer L2 cache misses than **Block Multiplication,** likely due to continuous row-wise access patterns.
    - **Block Multiplication** reduced **L1 cache misses** more effectively than other approaches, however for **L2** was worse than **Line Multiplication,** probably because of extra control overhead.

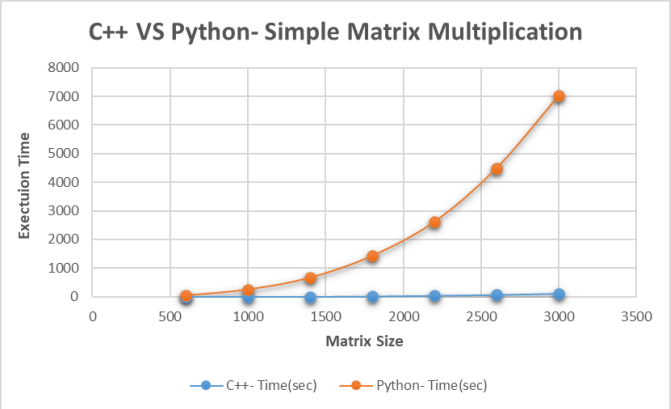


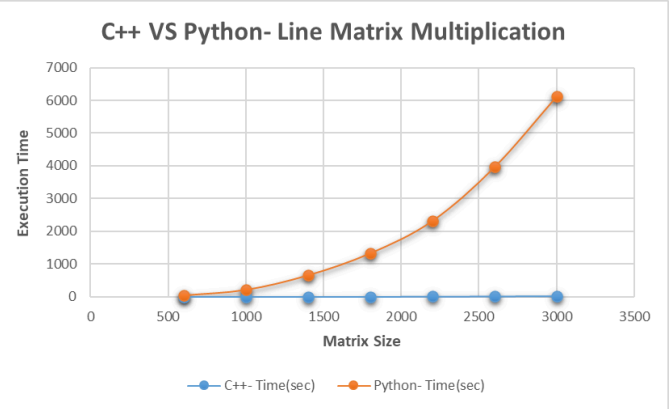
- **C++ vs. Python- Algorithms Execution Time**

To compare the two algorithms implemented in Python and C++, we used only the execution time. As we expected, C++ had much lower execution time than Python, this is due to several reasons, such as C++ being a compiled language, which means that the code is translated directly into machine instructions, leading to much faster performance.

One way to greatly improve the execution time of these two algorithms in Python and make it as strong as C++ would be to use optimized numerical libraries, like numpy, and with that approximate the execution time of C++ and Python, but

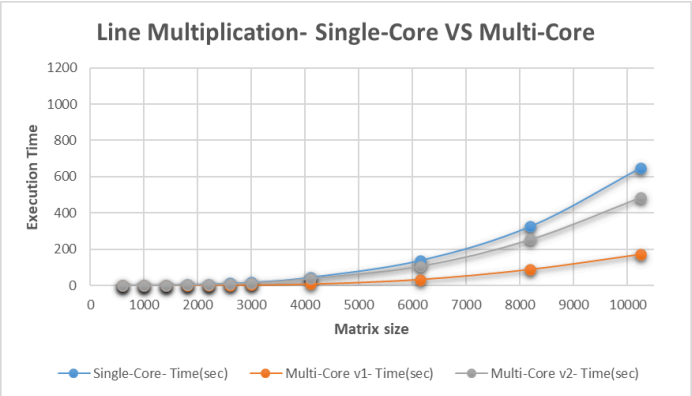
without these libraries we can clearly see through the graphs below that C++ is better than Python for these algorithms and Python algorithms are almost infeasible to large matrices.

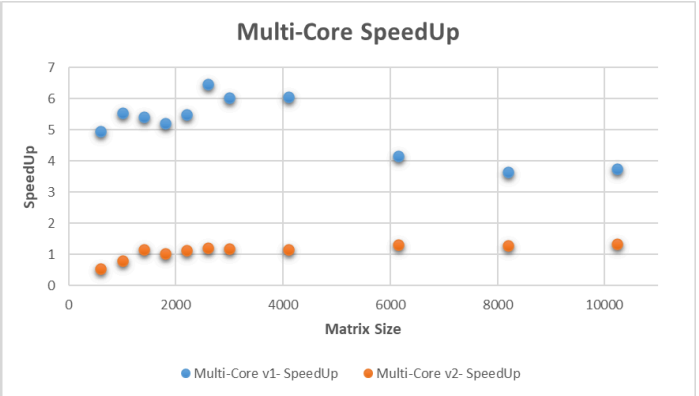


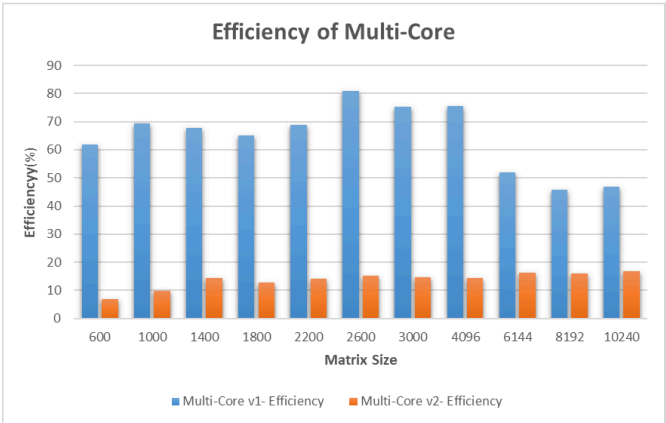


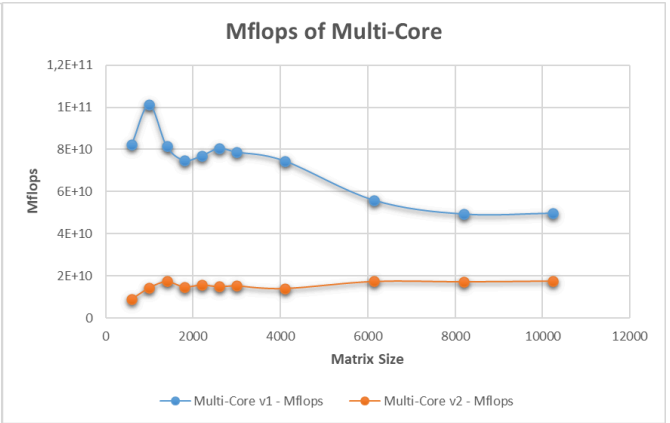
- **Single-Core VS Multi-Core Implementations**

Comparing the single-core and multi-core versions, as expected we could clearly see an improvement in performance for almost all matrix sizes, the parallel versions were much faster as we can see in the graphs below, which show the speedup and efficiency of each algorithm compared to the single-core version. Overall, the first parallel version was much better than the second, with an average speedup of 5 times that of the second, and consequently a higher efficiency. The Mflops of the first version were also better, as expected because the higher the Mflops, the faster the program will run for a given matrix size.









The difference in execution speeds between the two implementations is explained by the fact that the first takes better advantage of the potential of parallelism than the second.

The first algorithm parallelizes the outermost cycle and distributes all its iterations among the available threads, thus ensuring that each thread receives a unique row, while the second, despite parallelizing the block of code corresponding to the two outermost cycles, only distributes the iterations of the innermost cycle among the threads, creating a large overhead due to the repeated creation and destruction of threads.

Finally, the multi-core versions performed better overall than the single-core versions, and the first version really benefits from the full power of parallelism, while the second algorithm only takes partial advantage of a parallel implementation, so we can clearly see big differences in performance.

Conclusions

In summary, the first part of this project was to evaluate the performance of different algorithms, which allowed us to understand the importance and how memory access works, and it was great to try different algorithms and different approaches to compare for example execution time and number of cache misses.

In the second part, we had the opportunity to work with OpenMp and it was important to get some ideas about different approaches and therefore different performances, and also to understand new concepts about how parallelized areas work and their relation with execution time or cache misses for example.

Finally, we understood the importance of parallelism in a computational task and how to extract important metrics to compare different approaches and thus be more careful when choosing a particular algorithm.

Annexes

All data and graphics are also available at stats.xlsx.