Project Report

Project 1: Parallel and Distributed Computing

Class 3 Group 1
up202108740 Lara Bastos
up202108741 Lia Sobral
up202108678 Miguel Barros

March 2024

Contents

1 Introduction

In this project, the main goal is to evaluate the CPU performance on both single core and multi core computing. The former part focuses on studying the effect that different matrix multiplication algorithms have on the processor performance when accessing large amounts of data. The latter intends to implement parallel versions of the Line Matrix Multiplication algorithm and verify their performance.

2 Performance evaluation of a single core

In the first part of the project we aimed to understand how memory management and consequent CPU performance can be improved even when constrained to a single core. For that goal, we started by implementing three different matrix multiplications algorithm in C++. The first two were also implemented in Java.

2.1 Algorithms Explanation

The three implemented algorithms perform the multiplication of matrices A and B and store the result on matrix C. The matrices are always square and are represented as a two-dimensional array.

2.1.1 Normal Matrix Multiplications

The first algorithm was already given by the project proposal and consists of multiplying two matrices using the typical algorithm, by multiplying each line of the first one by each column of the second one.

The algorithm uses three nested loops:

```
1 for(i=0; i<m_ar; i++)
2     for( j=0; j<m_br; j++)
3         temp = 0;
4                 for( k=0; k<m_ar; k++)
5                         temp += pha[i*m_ar+k] * phb[k*m_br+j];
6                 phc[i*m_ar+j]=temp;
```

Algorithm 1: Pseudocode of Normal Matrix Multiplication

The **time complexity** is $\mathbf{O(N^3)}$, for a NxN matrix. The **space complexity** is $\mathbf{O(N^2)}$, where N is also the side of the matrix.

2.1.2 Line Matrix Multiplications

The second algorithm is similar to the previous one but with a slight improvement: it multiplies each element from A by the corresponsing element of a line in B:

```
1 for(i = 0; i < m_ar; i++)
2     for(k = 0; k < m_ar; k++)
3         for(j = 0; j < m_br; j++)
4             phc[i * m_br + j] += pha[i * m_ar + k] * phb[k * m_br + j];
```

Algorithm 2: Pseudocode of Line Matrix Multiplication

The time complexity and space complexity are $\mathbf{O(N^3)}$ and $\mathbf{O(N^2)}$ respectively. Although these are equal to the previous algorithm, this version reduces the number of cache misses performed, consequentially reducing execution time.

2.1.3 Block Matrix Multiplication

Lastly, the third algorithm consists of splitting the matrix A and B into sub matrices (blocks) A' and B', multiplying them line by line and adding them up.

Similarly to the previous ones, this algorithm has a **time complexity** of $\mathbf{O(N^3)}$ and **space complexity** of $\mathbf{O(N^2)}$, where N is the side of the matrices. Again, despite having the same complexity, the execution time of this version is improved as it increases the chances that the blocks will fit into the cache, reducing the number of times data needs to be fetched from the much slower main memory.

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

Algorithm 3: Pseudocode of Block Matrix Multiplication

3 Performance evaluation of a multi-core implementation

For the second part of the project, we implemented two parallel versions of the Line Matrix Multiplication Algorithm and analysed how taking advantage of the multiple cores available can improve CPU performance.

3.0.1 First Version of Parallel Line Matrix Multiplication

The first version parallelizes the outer loop of the algorithm which means that each thread handles a portion of the matrix rows, distributing the workload evenly across threads:

```
#pragma omp parallel for private (i, j, k) shared(pha, phb, phc, m_ar, m_br)
for(i = 0; i < m_ar; i++)
    for(k = 0; k < m_ar; k++)
        for(j = 0; j < m_br; j++)
            phc[i * m_br + j] += pha[i * m_ar + k] * phb[k * m_br + j];
```

Algorithm 4: C++ Implementation Solution 1 of Parallel Line Matrix Multiplication

3.0.2 Second Version of Parallel Line Matrix Multiplication

Contrarily, by parallelizing the innermost loop in the second version, each thread is responsible for computing a subset of the matrix elements within the inner loop.

```
#pragma omp parallel private (i, j, k) shared(pha, phb, phc, m_ar, m_br)
for(i = 0; i < m_ar; i++)
    for(k = 0; k < m_ar; k++)
        #pragma omp for
        for(j = 0; j < m_br; j++)
            phc[i * m_br + j] += pha[i * m_ar + k] * phb[k * m_br + j];
```

Algorithm 5: C++ Implementation Solution 1 of Parallel Line Matrix Multiplication

Note: The "private (i, j, k)" clause ensures each thread has its own copies of loop variables, while "shared(pha, phb, phc, m _ ar, m _ br)" declares variables accessible to all threads in OpenMP

4 Performance metrics

For each version of each algorithm we measured the **processing time**, which allowed the calculation and posterior analysis of the following metrics:

- **Mega Floating-Point Operations per Second** (**MFLOPS**): $\frac{Complexity of Algorithm}{Processing Time}$. The complexity of the used algorithms is $2 * n^3$.
- **Speedup**: $\frac{TSequential}{TParallel}$ (for the second part)
- **Efficiency**: $\frac{Speedup}{Cores}$ (for the second part)

Additionally, to better understand how different parallelization techniques and algorithms impacted CPU performance, we took advantage of the functionalities provided by PAPI (Performance API) that gathers information on the processor's performance in the form of events to measure:

- **Data cache misses L1**: PAPI_L1_DCM
- **Data cache misses L2**: PAPI_L2_DCM
- **Data accesses L2**: PAPI_L2_DCA

We considered these metrics to be relevant as a cache miss can imply significant variances in processing time and explain why certain implementations, despite having the same time complexity, are less efficient.

We measured each of these twice and used the arithmetic average for analysis. By averaging multiple measurements, we aimed to minimize the impact of outliers on our conclusions, thus providing a more accurate assessment of the performance.

All the execution were performed in an Intel Core i7-10510U, with 4 cores, 8 threads and a size of 64KB and 256MB for cache L1 and L2 respectively (per core).

5 Results and Analysis

5.1 Normal Matrix Multiplication and Line Matrix Multiplication

For both the **Normal Matrix Multiplication** and **Line Matrix Multiplication** algorithms we measured their execution times across square matrices ranging from 600x600 to 3000x3000, with increments of 400. Furthermore, the **Line Matrix Multiplication** algorithm was analyzed for matrices ranging from 4096x4096 to 1024x1024, with increments of 2048.

5.1.1 Execution Time

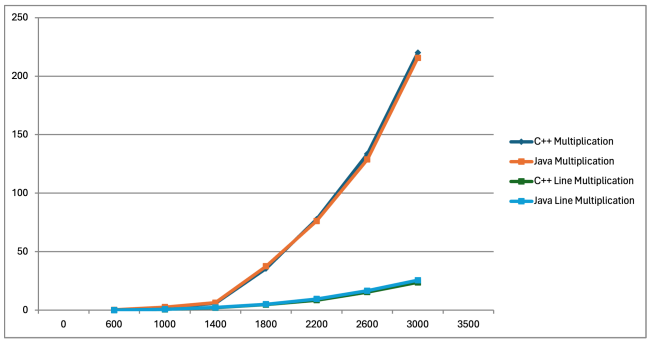

Figure 1: Execution time for Normal and Line Matrix Multiplication for C++ and Java

From an increasing matrix size, it's evident that the execution times remains consistent between C++ and Java implementations, which can be attributed to the fact that both languages are compiled, resulting in a similar performance.

The **Line Matrix Multiplication** algorithm consistently exhibits lower execution times compared to the Normal Matrix Multiplication. While the time complexity remains the same for both, the **Normal Matrix Multiplication** demonstrates an exponential increase in execution time as the matrix size grows.

5.1.2 L1 and L2 Data cache misses

It's notable that the **Line Matrix Multiplication** algorithm exhibits fewer cache misses compared to the **Normal Matrix Multiplication** algorithm, as hypothesized in section 2.1.2 and clarifying why the execution time is lower for the former. While in the **Normal Matrix Multiplication** algorithm the processor necessitates to access main memory access for each iteration of the inner loop to fetch a new line, in the **Line Matrix Multiplication** it only requires fetching a new line only in the two outer loops. This optimization leads to a reduction in cache misses from N^3 to N^2.

The Normal Matrix Multiplication algorithm has more L2 cache misses than L1, leading us to believe that the L1 cache may not be able to hold all the data being accessed. As a result, when the required data is not found in the L1 cache, the processor needs to fetch it from the larger and slower L2 cache.

An increase in matrix size correlates with a higher number of cache misses, as it would be expected.

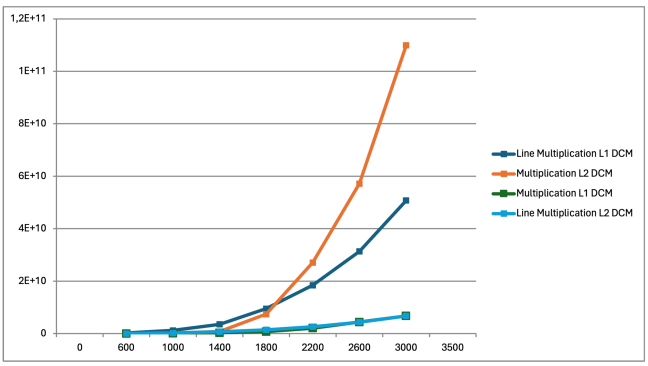

Figure 2: L1 and L2 Cache Misses for Normal and Line Matrix Multiplication for C++ and Java

5.1.3 MFLOPS

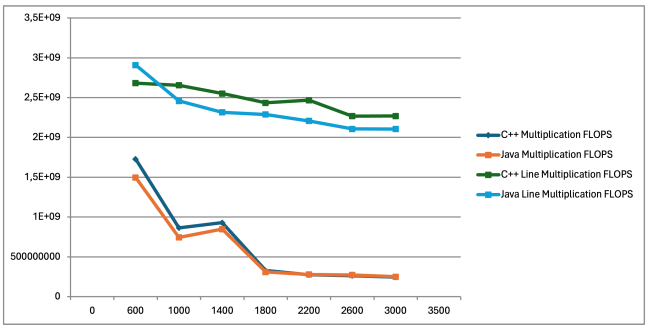

Figure 3: MFLOPS for Normal and Line Matrix Multiplication for C++ and Java

The number of MFLOPS on the **Line Matrix Multiplication** algorithm is significantly higher that in the **Normal Matrix Multiplication** one, as the algorithm is more efficient and requires less cache misses.

In the **Line Matrix Multiplication** algorithm, the number of floating-point operations remains relatively constant across different matrix sizes. Conversely, in the **Normal Matrix Multiplication**, the number of floating-point operations per second decreases as the matrix size increases. This observation leads us to infer that larger matrices necessitate more frequent main memory access, which consequently slows down the algorithm.

Both languages exhibited similar performance in the first algorithm, but Java's potentially worse performance in the second algorithm could be attributed to its less optimized memory management.

5.2 Different Block Sizes for The Block Matrix Multiplication

For the **Block Matrix Multiplication** algorithm, we measured execution times for matrices ranging from 4096x4096 to 1024x1024, with increments of 2048, using block sizes of 128, 256, and 512.

5.2.1 Execution Time

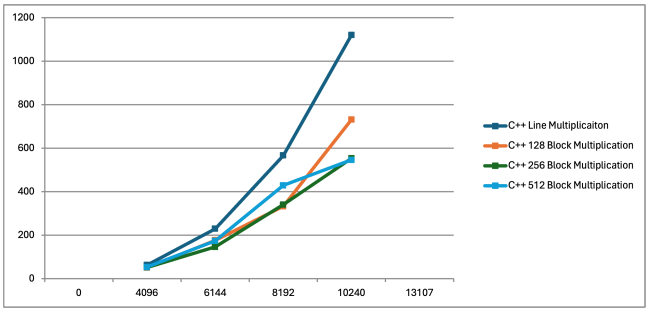

Figure 4: Execution time for Line and Block Matrix Multiplication

The analysis of the execution time for the **Block Matrix Multiplication** algorithm, compared to the **Line Matrix Multiplication**, demonstrates superior performance. Additionally, upon examining the impact

of different block sizes on the algorithm, we observed that the execution time is lowest with a block size of 256x256, leading us to believe this might be the ideal block size for the computer in use.

5.2.2 MFLOPS

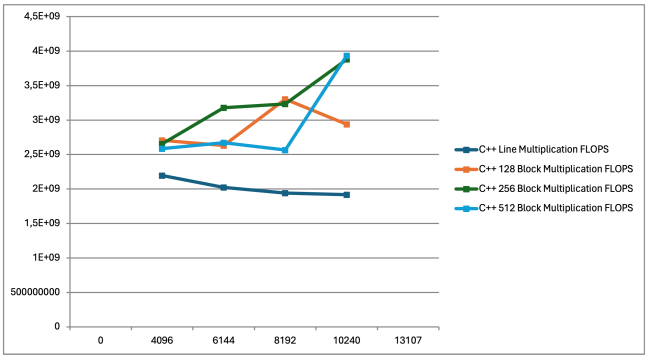

Figure 5: MFLOPS for Line and Block Matrix Multiplication

We can state the **Block Matrix Multiplication** is more efficient as there is a higher number of flops, meaning faster computations.

This data proved the theory discussed in the previous subsection, as the 256x256 seems to be the block performing the best in terms of operation per second as well.

5.2.3 L1 and L2 Data cache misses

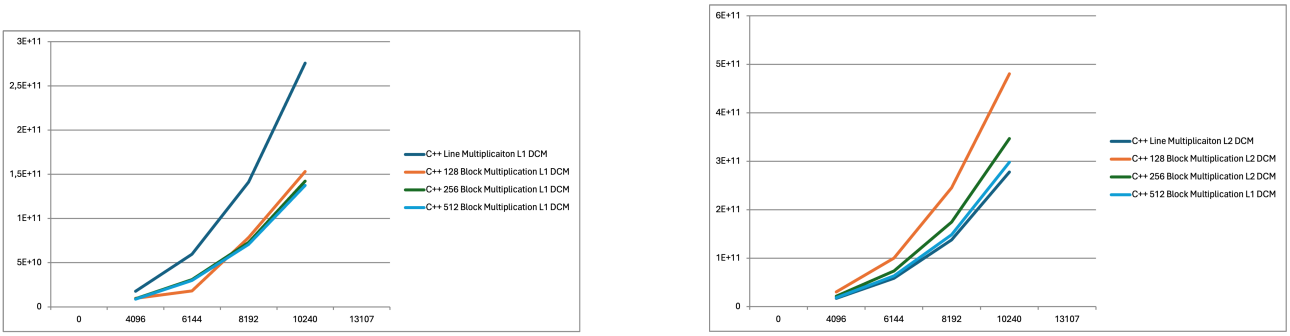

Figure 6: L1 and L2 cache misses for Line and Block Matrix Multiplication

We can observe that the **Line Matrix Multiplication** algorithm had a marginally higher number of data cache misses at level 1, suggesting the algorithm loses plenty of time accessing memory explaining once again why is this algorithm is less efficient. Therefore, the difference between L2 cache misses for **Block Matrix Multiplication** compared **Normal Matrix Multiplication** is not as relevant, so the higher number of cache misses for **Block Matrix Multiplication** doesn't decrease its performance.

At L2 we can see that for an increasing matrix size the number of cache misses is smaller for 256x256 and 512x512 blocks, corroborating with the conclusions taken in section 5.2.1 and 5.2.2.

Additionally, it was possible to conclude that at level 2 the number of cache misses decreases with the increase of the block size.

We owned this phenomenon to the fact that utilizing a block size of 128x128 with on cells of 8 bytes each (doubles), exceeds the capacity of the L1 cache used (64KB), necessitating the use of the L2 cache. Consequently, algorithms employing this block size may not effectively minimize cache misses. Larger block sizes however, particularly 512x512, prove to be more efficient in mitigating cache misses. This explains why the 128x128 block consistently performs worst.

5.3 Parallel Version of the Line Matrix Multiplication

For the second part of the project we measured both versions of parallelization for matrices from 600x600 to 3000x3000, with increments of 400.

5.3.1 Speedup and Efficiency

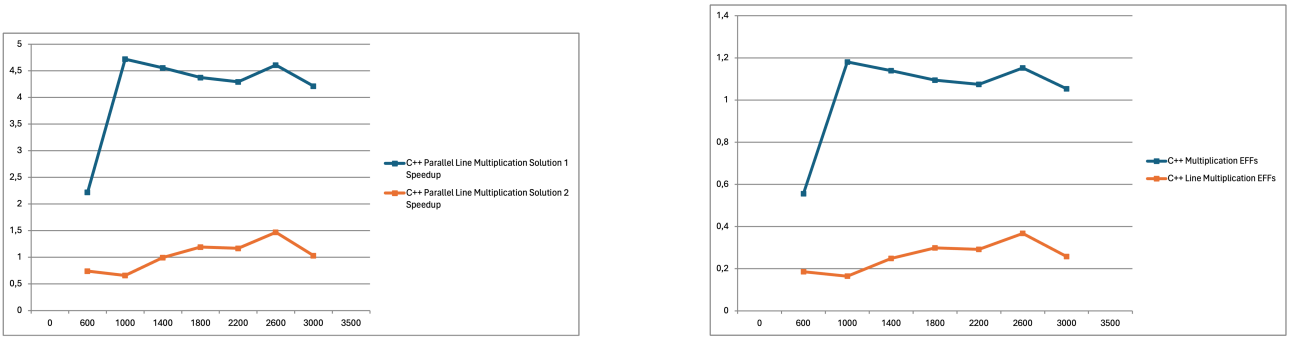

Figure 7: Speedup and Efficiency for the two versions of Parallel Line Multiplication

From the data, it's evident that the speedup is higher in the **First Version of the Parallel Line Multiplication** algorithm . In this version, parallelization is applied at the outermost loop, allowing each thread to handle a larger portion of the overall computation, thereby achieving better workload distribution. Because the parallelization encompasses the entire computation, threads in the first version can work concurrently on multiple iterations of the outer loop, maximizing parallelism. Particularly for matrix sizes exceeding 1000x1000, the parallelized version demonstrates significant efficiency gains compared to the sequential implementation, as for larger matrices there are more iterations of the computation to be distributed among threads

Conversely, the **Second Version of the Parallel Line Multiplication** performs even worse than the sequential one, as evidenced by its speedup below 1. By restricting parallelization to the innermost loop, threads don't have enough independent work to perform concurrently, leading to inefficient utilization of computational resources.

The efficiency assessment of both algorithms confirms the speedup results, as expected, affirming the superiority of the first version of parallelization over both the sequential and the second parallelization versions.

5.3.2 MFLOPS

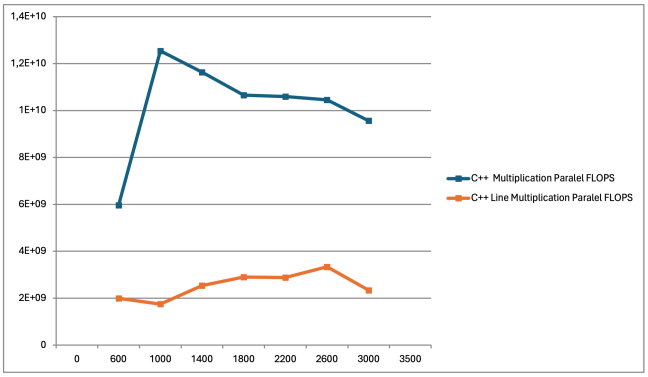

Figure 8: MFLOPS for Line and Block Matrix Multiplication

The MFLOPS metric further supports the superiority of the first version of the line multiplication algorithm, as it demonstrates that the **First Version of the Parallel Line Multiplication** executes more floating-point operations per second compared to its counterpart, attributed to its effective parallelization strategy at the outermost loop level.

6 Conclusion

In conclusion, we believe the goal proposed for this project were successfully met. We were able to apply the knowledge gained from the course lectures and deepen our understanding of Parallel and Distributed Computing concepts.

By analyzing various matrix multiplication algorithms, we understood how factors beyond time complexity can influence CPU performance. Moreover, the use of parallelization techniques demonstrated how taking advantage of the cores available can optimize computational performance.

7 References

1 Lecture slides.

2 Introduction to parallel algorithms CS300. Available at: https://www.stolaf.edu/people/rab/pdc/text/alg.htm

3 Intel Core i7-10510U Available at: https://www.techpowerup.com/cpu-specs/core-i7-10510u.c2236