Parallel and Distributed Computing

First Project

Bruno Oliveira - up202208700
Mansur Mustafin - up202102355
Rodrigo Silva - up202205188

Bachelor's in Informatics and Computing Engineering
Parallel and Distributed Computing

Professor: João Resende

March 2025

Contents

1 Introduction

This report presents the algorithms, performance results and respective analysis of the first project in the Parallel and Distributed Computing course.

The aim of this project was to evaluate performance between single-core (in 2 languages) and multi-core (using OpenMP) approaches to a problem. Specifically, we wanted to implement and compare 5 different algorithms to perform the multiplication of two matrices (see Algorithms Explanation).

By developing the code for these tasks, we became more familiar with modern and high performance C++ and Lua, since we chose it as our alternative implementation language. Moreover, we also had to research about matrix multiplication algorithms and optimizations, a common benchmark in computing. Last but not least, in order to implement multi-core solutions, we explored and experimented with the OpenMP standard, allowing us to learn more about CPU multithreading.

2 Algorithms Explanation

To see the effects of cache locality and parallel computing on matrix multiplication implementations, we used five different matrix multiplication algorithms, described in this section. These algorithms perform the same operations, ending in the same results, only differing by the order of the operations done, which, as we will see, will have crucial impacts on each algorithm's performance.

2.1 Classic Matrix Multiplication

The *classic* matrix multiplication algorithm is closely related to its hand calculation. This method for multiplication of two matrices: $A \times B = C$, where $A(m, n)$, $B(n, p)$ and $C(m, p)$, obtains the sum of a cell (m, p) in the solution, by linear combination of row m of A and column p of C.

Implementing this algorithm is fairly straighforward, as we only need to iterate through every row and column of the solution matrix, and, for each cell, perform the linear combination of the values:

```
for (i = 0; i < m; i++)
    for (j = 0; j < p; j++)
        for (k = 0; k < n; k++)
```

Figure 1: Overview of Classic Matrix Multiplication Algorithm

As we can expect from the 3 nested loops, the overall time complexity for the matrix multiplication algorithm is (n^3). This time complexity is maintained among all the implemented algorithms, therefore performance impacts are mainle caused by cache accesses and parallelism.

2.2 Line Matrix Multiplication

The line matrix multiplication algorithm is very similar to the previous algorithm. The only difference is that the second and third loops switch places:

```
for (i = 0; i < m; i++)
    for (k = 0; k < n; k++)
        for (j = 0; j < p; j++)
```

Figure 2: Overview of Line Matrix Multiplication Algorithm

In practice, the difference between this and the previous approach is that instead of calculating the result matrix cell by cell, we instead focus on calculating and storing some terms of the linear combination line by line.

Although the correctness and overall time complexity of this algorithm remains the same, due to the way the matrices are stored and the memory is accesed, this allows for fewer cache misses, leading to performance improvements.

2.3 Block Matrix Multiplication

The last single-core algorithm implemented is the block matrix multiplication algorithm. This approach expands on top of the previous one and the same logic applied in calculating subresults line by line is now used to calculate the solution block by block. For this, we need 6 nested loops:

```
for (I = 0; I < m; I += block_size)
    for (K = 0; K < n; K += block_size)
        for (J = 0; J < p; J += block_size)
            for (i = I; i < min(I + block_size, m); i++)
                for (k = K; k < min(K + block_size, n); k++)
                    for (j = J; j < min(J + block_size, p); j++)
```

Figure 3: Overview of Block Matrix Multiplication Algorithm

The main idea is that now, we have divided the matrix into smaller blocks and the operations are performed block by block. Again, due to the way the memory is accessed, this leads to better performance and less cache misses because the local block information is already stored in cache.

2.4 Parallel Matrix Multiplication

Besides the previous three single-core algorithms, we also implemented two variations of the line algorithm using multi-core processing. For this, we used the OpenMP API for parallel programming in C++. The next sections show an explanation and overview of the implementations.

2.4.1 First Version

The first approach aims to parallelize the outermost loop (i) of the line algorithm. There will be a division of i values between the different threads so that each one works on a separate row at the same time (they also get their own k and j variables). Due to working on separate rows, the threads do not need synchronization.

For this, we used OpenMP's `parallel for` and `private` variable pragmas:

```
#pragma omp parallel for private(i, k, j)
for (i = 0; i < m; i++)
    for (k = 0; k < n; k++)
        for (j = 0; j < p; j++)
```

Figure 4: Overview of First Parallel Matrix Multiplication Algorithm

2.4.2 Second Version

The second approach aims to parallelize the innemost loop (j) of the line algorithm. The thread pool will work on the same row at the same time but on different columns. In this case, the threads need to be synchronized after finishing the row operations which can lead to some overhead.

For this, we used OpenMP's `parallel` region, `for` and `private` variable pragmas:

```
#pragma omp parallel private(i, k)
for (i = 0; i < m; i++)
    for (k = 0; k < n; k++)
        #pragma omp for private(j)
        for (j = 0; j < p; j++)
```

Figure 5: Overview of Second Parallel Matrix Multiplication Algorithm

3 Performance Metrics

The performance of the algorithms was evaluated using the following metrics:

- **Time:** The time required to execute the algorithm. This is a crucial metric for comparing the speed and efficiency of multicore implementations.

- **Speedup:** Defined by the ratio of the sequential execution time T_{seq} to the parallel execution time T_{par}:

  $$Speedup = \frac{T_{\text{seq}}}{T_{\text{par}}}.$$

  This indicates how much faster the parallel algorithm runs compared to the sequential version. A higher speedup indicates more effective utilization of parallel resources.

- **Efficiency:** Defined by the ratio of the speedup to the number of threads p:

$$Efficiency = \frac{Speedup}{p}.$$

  This measures how effectively the algorithm scales with an increasing number of threads. An efficiency of of 1 (or 100%) indicates perfect scaling.

- **FLOPS:** The number of floating-point operations per second, given by the formula

$$FLOPS = \frac{2 \times n^3}{time},$$

  where $2 \times n^3$ represents the total number of arithmetic operations (one multiplication and one addition per iteration) in the matrix multiplication, and time is the total execution time. Higher FLOPS values indicate better numerical performance.

- **Cache Misses:** To evaluate memory access patterns and their performance impact, we use number of cache miss, such as L1_DCM (Level 1 Data Cache Misses), L2_DCM (Level 2 Data Cache Misses), and L3_TCM (Level 3 Total Cache Misses). Each miss forces the processor to fetch data from a slower cache level or main memory, incurring extra clock cycles. Reducing cache misses can improve time execution by minimizing these wasted cycles.

4 Results and Analysis

5 Conclusion

References