Parallel and Distributed Computing – 1st Project Report

Guilherme Almeida, João Malva, Nuno Pereira

March 8, 2023

1 Introduction

The goal of this project was to develop a series of algorithms that perform matrix multiplication and analyze their performance using various techniques, namely:

- Normal Matrix Multiplication;
- "Line by Line" Matrix Multiplication;
- Block Matrix Multiplication;

2 Algorithm Analysis

The algorithms described were implemented using the C programming language, the Java programming language and the Performance API, PAPI for short.

PAPI is used in conjunction with the C code because it allows us to directly collect several metrics related to the performance of the algorithms developed. It is not used with the Java code since it runs on the Java Virtual Machine, and therefore the performance registers are virtualized, which inhibits us from

2.1 Normal Matrix Multiplication

This algorithm is the *naïve* implementation of the normal matrix multiplication procedure.

It iterates through each line of the first matrix and each column of the second matrix to perform the dot product between each vector, accumulating the results in the destination matrix.

The algorithm can be (grossly) coded as the following:

Algorithm 1 Naïve Matrix Multiplication Algorithm

```
1: function OnMult($a, b$)
2:     $mx\_size \leftarrow Lenght(a)$                ▷ $a$ should have the same side length as $b$
3:     $c \leftarrow matrix[mx\_size][mx\_size]$                ▷ $c$ is initialized with 0s
4:     for $i \leftarrow 0$ to $mx\_size$ do
5:         for $j \leftarrow 0$ to $mx\_size$ do
6:             for $k \leftarrow 0$ to $mx\_size$ do
7:                 $c_{i,j} \leftarrow c_{i,j} + a_{i,k} * b_{k,j}$
8:             end for
9:         end for
10:    end for
11:    return $c$
12: end function
```

2.2 ”Line by Line” Matrix Multiplication

This algorithm is an improvement over the normal matrix multiplication algorithm.
It exploits the properties of cache locality to speed up the matrix product computations.
The algorithm can be (grossly) coded as the following:

Algorithm 2 Line Matrix Multiplication Algorithm

```
1: function OnMultLine($a, b$)
2:     $mx\_size \leftarrow Lenght(a)$                    ▷ $a$ should have the same side length as $b$
3:     $c \leftarrow matrix[mx\_size][mx\_size]$          ▷ $c$ is initialized with 0s
4:     for $i \leftarrow 0$ to $mx\_size$ do
5:         for $k \leftarrow 0$ to $mx\_size$ do
6:             for $j \leftarrow 0$ to $mx\_size$ do
7:                 $c_{i,j} \leftarrow c_{i,j} + a_{i,k} * b_{k,j}$
8:             end for
9:         end for
10:    end for
11:    return $c$
12: end function
```

The main difference between this algorithm and the previous one is that, by changing the order of iteration of `j` and `k`, the processor does not need to load a new set of data into cache due to the multiplication order following the second matrix's lines, thus reducing drastically the number of cache misses resulting from the *naïve* implementation.

2.3 Block Matrix Multiplication

This reasoning behind this algorithm is to ”divide and conquer”: by dividing the original matrix into many sub-matrices and performing the multiplication on these, the process is sped up.

The algorithm used with each sub-matrix is the ”Line by Line” one since it offers better performance over the *naïve* implementation.

Since the code for this algorithm is pretty big, it is not included in the report.

This algorithm also has another input parameter: the block size. This value controls the size of the resulting sub-matrices, which affects the performance of the algorithm, as can be seen in the following section.

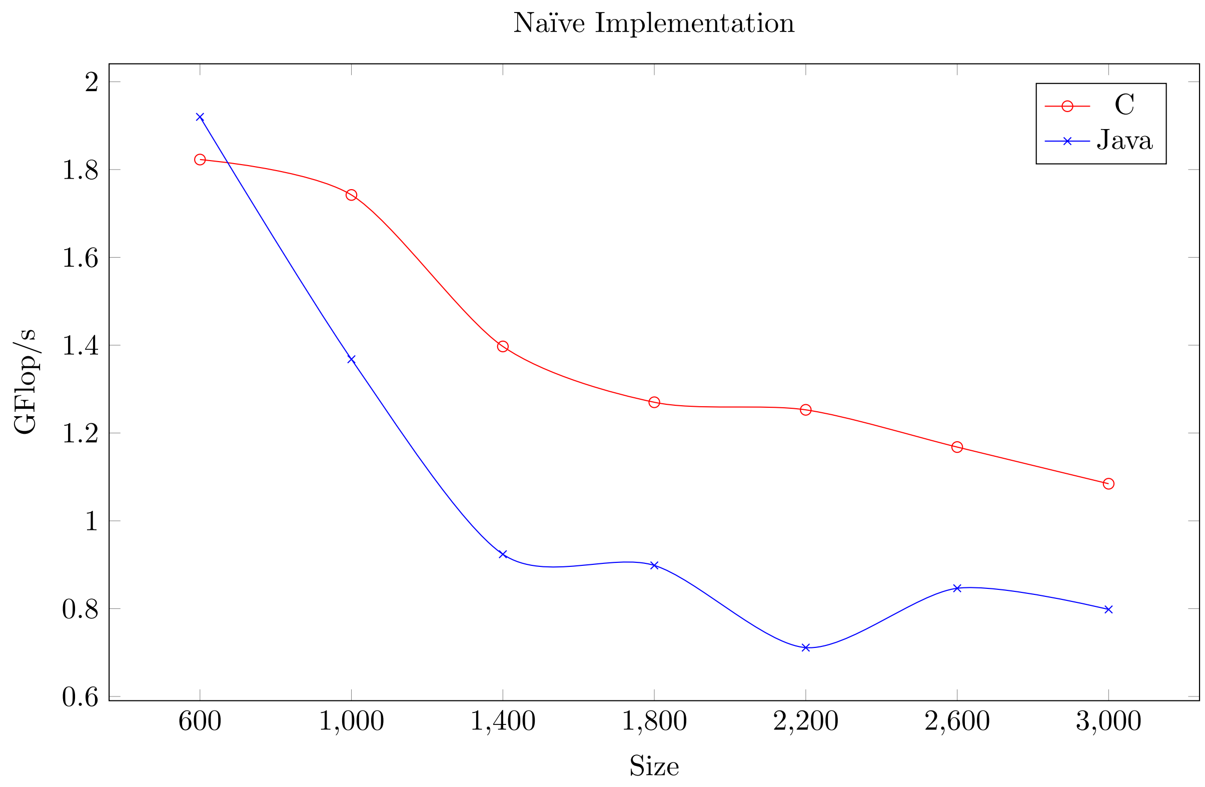

Figure 1: GFlop/s for both implementations of the naïve algorithm

3 Performance Analysis

The performance metrics collected to analyze the developed algorithms were:

- The execution time;
- The Level 1 Data Cache misses;
- The Level 2 Data Cache misses;

These metrics give us an overview of the behavior of the algorithms and allow us to derive another important metric: Gflops/s. This value indicates how many floating point operations were performed per second: (typically) the higher the value the more operations were performed and the faster the algorithm was.

3.1 Data graphics

The following graphics depict the differences between the two languages used across the various metrics measured:

3.1.1 GFlop/s

4 Conclusions

All of the project's goals were achieved -

A Appendix

A.1 Code

Versions of the algorithms in C and Java are available in folder `src/`.

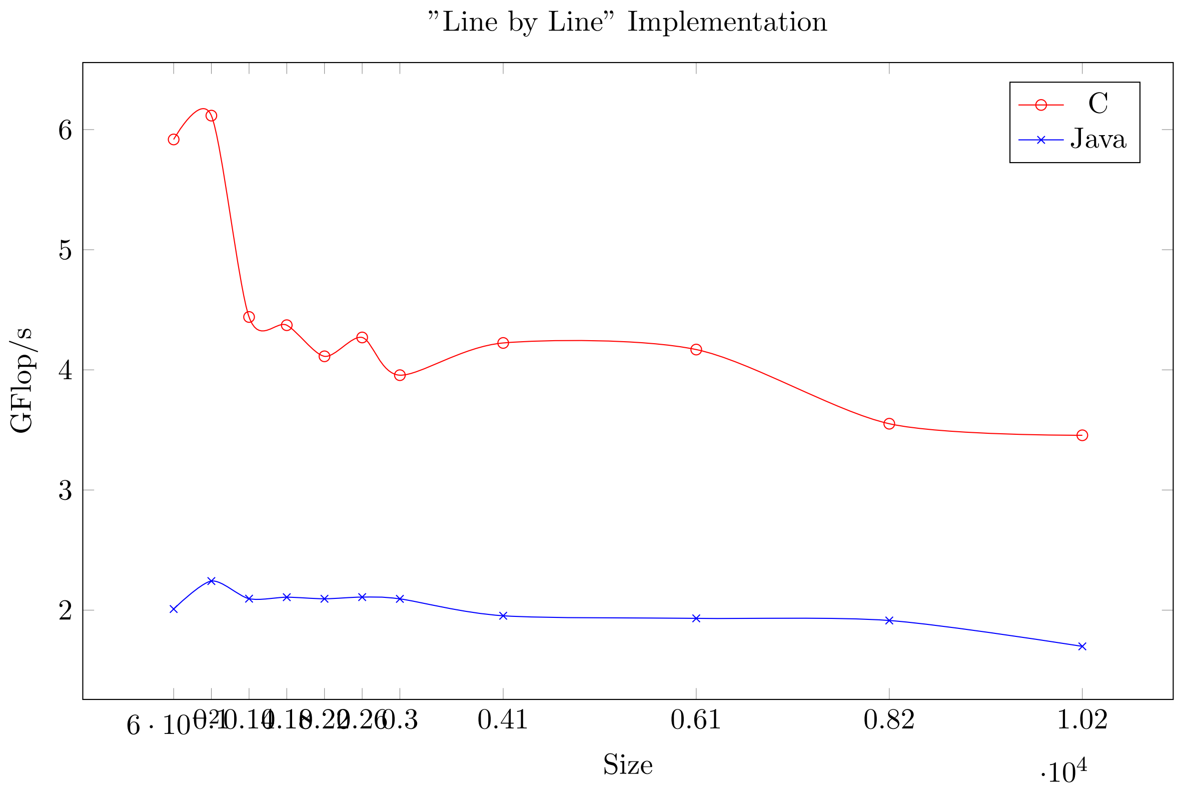

Figure 2: GFlop/s for both implementations of the "line by line" algorithm