A Performance Evaluation of Programming Languages Operating in Single Core Instructions

Parallel and Distributed Computing
Bachelors in Informatics and Computer Engineering

3L.EIC01_G5
Joel Fernandes up201904977@up.pt
Mário Travassos up201905871@up.pt
Tiago Rodrigues up201907021@up.pt

March 24, 2022

Introduction

This project intends to show and evaluate the effect of processor performance when accessing large amounts of data, performing the same instructions multiple times. In this study, the product of two matrices was used as the base calculation.

Also, a comparison of how different programming languages interact with memory and impact the processor speed is shown. It is important to highlight that these tests were performed on a single core, so no parallelism optimizations are made.

Finally, performance measures were made using the Performance API (PAPI), which will be analyzed and discussed in further detail.

Problem Description

The problem used to evaluate the performance was the matrix multiplication. It was chosen because the amount of instructions does not impact performance tremendously, with the greatest bottleneck being memory access.

That way, we can measure more truthfully how much time does the processor spend accessing memory, and the impact that cache hits and misses have on a program.

Even though the main intention is to measure memory access performance, we also could see how some improvements in the algorithms used could make the processing time differ.

The first improvement was to multiply by line instead of the usual matrix multiplication. Then, a further improvement made was multiplying by block, which is shown to reduce running times.

Algorithm Analysis

Normal Multiplication

The first algorithm developed calculates the matrix product the way that students are traditionally taught in their Algebra class. In this algorithm, values are multiplied sequentially, with the first element being the dot product between the first row from the first matrix and the first column from the second matrix, and so on.

The following pseudocode details how the algorithm works. Here, a is the First Matrix and b the Second one, and they produce the result on Matrix c:

Algorithm 1 Regular Multiplication

```
1: procedure REGULAR MULTIPLICATION(a, b)
2:     for i = 0 to length(a) do
3:         for j = 0 to length(b) do
4:             temp ← 0
5:             for k = 0 to length(a) do
6:                 temp ← temp + a_{i,k} + b_{k,j}
7:             c_{i,j} ← temp
```

Although it is simple to implement and easy to understand as it follows the mathematical definitions, it is not very performant as matrix sizes increase drastically.

Line Multiplication

In this version of the algorithm, all that is done is a change in the order of operations. Instead of performing the calculations based on the result matrix, we perform them based on the rows of the second matrix. This means that first, all values of the first row of the second matrix are "used", and only then does it change to the second one, and so on.

To further clarify this, the following pseudocode shows how it works. Once again, a represents the first matrix, b the second one, and the output will be stored in c:

Algorithm 2 Line Multiplication

```
1: procedure LINE MULTIPLICATION(a, b)
2:     for i = 0 to length(a) do
3:         for j = 0 to length(b) do
4:             for k = 0 to length(a) do
5:                 c_{i,k} ← c_{i,k} + a_{i,j} + b_{j,k}
```

Block Multiplication

The final algorithm used was the block multiplication method. This takes advantage of the fact that a matrix can be partitioned into sections, called blocks, that can be multiplied together, yielding the same result as the regular multiplication.

The following pseudocode can help with understanding how such calculations are done. Again, a is the first matrix, b the second one, and c will store the result of executing the instructions. Also, this time, the size of the blocks is included, as it can affect speed of computations:

Algorithm 3 Block Multiplication

```
1: procedure FINDSMALLEST(index, blockSize, size)
2:     if index + blockSize > size then
3:         return size
4:     else
5:         return index + blockSize
6: procedure BLOCK MULTIPLICATION(a, b, blockSize)
7:     for jj = 0 to length(a) in blockSize increments do
8:         for kk = 0 to length(a) in blockSize increments do
9:             for i = 0 to length(a) do
10:                for j = jj to FindSmallest(jj, bkSize, length(a)) do
11:                    for k = kk to FindSmallest(kk, bkSize, length(a)) do
12:                        c_{i,j} ← c_{i,j} + a_{i,k} + b_{k,j}
```

Even though the number of loops increased, this proved to be the fastest algorithm, as will be later analyzed in the next section.

Performance Evaluation

Metrics Used

To evaluate the performance of each algorithm, 5 metrics were used to determine both speed and memory performance. To measure speed, the running time of the calculations was measured, in milliseconds. To measure memory access, the Performance API (PAPI) was used, to measure cache hits and misses, both to the L1 and the L2 cache.

Results Analysis

Cringe

Conclusion

The way and order in which we perform computations can have a great impact on performance. The use of the proper algorithms, even in single core programs, can have a significant impact on processing time as well. Even more, the programming language used and the way it handles memory can also affect the time, as it has been shown in this report.

The result turns what to be what was expected, with a language designed for speed like C++ beating out Java in every metric. **Mention cache hits and misses whenever they are available**

With this in mind, the report is satisfactory, as it concludes what was thought to be true beforehand, and it contributed to our knowledge of how machine instructions, order of operations and memory access can have a significant impact on the speed of programs.