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 27, 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 benchmark.

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 were 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.

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 marginally since we are not yet using parallelization.

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. This algorithm used the previous line multiplication optimization on each of these small matrices.

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,k} ← c_{i,k} + a_{i,j} + b_{k,k}
```

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

Results Analysis

The following tables present the data collected in regards to the performance metrics motioned above:

Matrix Size	Execution Time in C/C++ (s)	Execution Time in Java (s)	L1 Cache Misses	L2 Cache Misses
600	0.195	0.219	244653501	39881374
1000	1.178	1.572	1218513480	279800654
1400	3.370	4.028	3486142982	1446737049
1800	17.441	19.446	9047593753	6474783745
2200	38.367	40.977	17623177754	21085484639
2600	68.470	70.859	30872105198	47541682771
3000	115.008	119.743	50290435489	92834582308

Table 1: Performance for Normal Matrix Multiplication

As we can see C/C++ will always run faster than Java (this will persist throughout every algorithm). Additionally, the number of both L1 and L2 cache misses is very high hence the slow speed at which this algorithm runs.

Matrix Size	Execution Time in C/C++ (s)	Execution Time in Java (s)	L1 Cache Misses	L2 Cache Misses
600	0.102	0.152	27251081	56256654
1000	0.447	0.614	126128908	258714755
1400	1.476	2.734	347198530	712556517
1800	3.209	5.901	745250056	1455316787
2200	5.973	10.802	2074934678	2603551197
2600	10.039	17.933	4412661110	4245146040
3000	15.440	27.588	6780116699	6431098530
4096	40.751	-	17631509334	16662860353
6144	138.618	-	59479942150	56122394974
8192	339.714	-	140735675064	133737381570
10240	648.234	-	274931653237	263241362704

Table 2: Performance for Line Matrix Multiplication

This algorithm runs much faster than the last one dispute the same temporal complexity ($O(n^3)$) this is due to the change in order of calculation of the result matrix. This causes the program to load less memory each time it multiplies the data which will be saved save in cache and loaded later. This, in response, will cause less cache misses and the program does not need to load data from memory that often.

Matrix Size	Block Size	Execution Time in C/C++ (2)	L1 Cache Misses	L2 Cache Misses
4096	128	37.435	9969660010	33923604657
	256	33.541	9134545276	22396558533
	512	39.257	8759951347	19058265882
6144	128	121.318	33389228826	114150534019
	256	113.862	30804770621	77463393766
	512	116.091	29629739330	65190519908
8192	128	335.681	78645306193	252299480469
	256	329.338	72474408666	162485781627
	512	333.740	70361616713	144484233529
10240	128	571.861	153447617238	499714786829
	256	528.412	142455484130	350574408778
	512	525.005	137014601382	298485724101

Table 3: Performance for Block Matrix Multiplication

This algorithm runs marginally faster than the last since here we are just doing the same calculation on multiple smaller matrices. Consequently, this lowers the number of cache misses since we divide the big matrices into smaller ones so we load less memory on each matrix multiplication. This performance can be greatly enhanced thought parallelization.

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 out to be what was expected, with a language designed for speed like C/C++ beating out Java in every metric.

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.