Performance Evaluation

Report

Mestrado Integrado em Engenharia e Computação

Computação Paralela

Author:
Diogo Serra Duque - up201406274@fe.up.pt

13th of March 2018

Introduction

For the CPAR course, the class was proposed a study on the effect on the processor performance of the memory hierarchy, giving a special focus to situations of accessing large amounts of data.

The operation chosen for the study was the calculation of the multiplication of two large matrices. First, this operation will be executed in different programming languages to measure and compare their time performance. Second, two different algorithms are attempted (in C language) as a way of solving the problem, while at the same time measuring some performance metrics for better understanding their differences.

Problem Description

The problem in analysis here is the result of the multiplication of two matrices. Given a matrix M1 of size *n* x *n* and a matrix M2 of size *n* x *n*, the result of the multiplication of M1 and M2 should be a matrix M3 of size *n* x *n* where each element's value is given by the following equation:

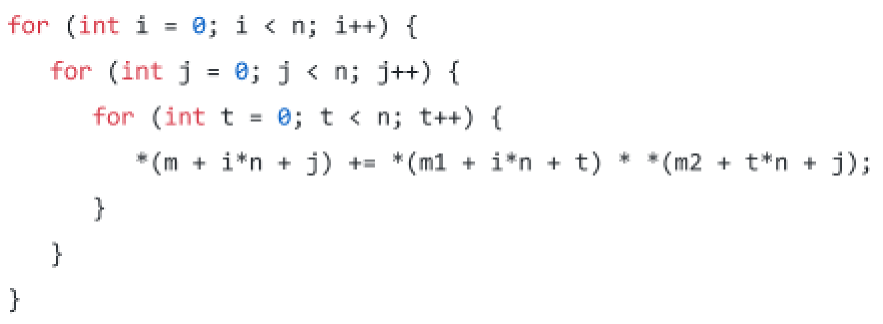
$$M3_{i,j} = \sum_{k=0}^{m-1} M1_{i,k} \,.\, M2_{k,j}$$

Algorithms Explanation

In this section, the two different algorithms used for solving the proposed problem are properly explained.

Naive Approach

The *Naive* approach on this problem leads to an algorithm that multiplies each row of matrix M1 for each column of matrix M2, thus computing the value of the cells of M3 one at a time. The central code of the algorithm is displayed right below, where *i* is the row of M1 being used in the multiplication and *j* is the column of M2. The inner loop with index variable *t* is used to sum the multiplications of each pair of elements.

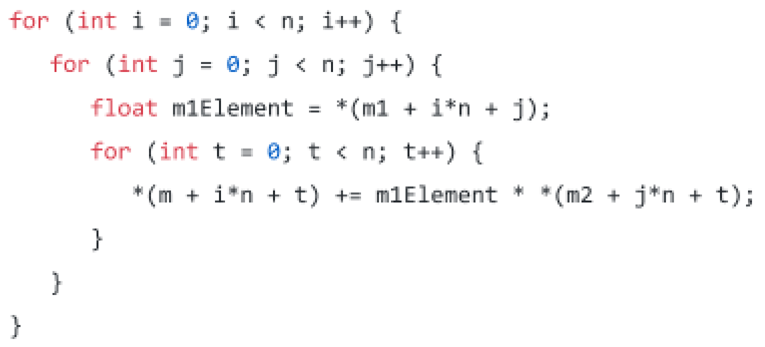


```c
for (int i = 0; i < n; i++) {
   for (int j = 0; j < n; j++) {
      for (int t = 0; t < n; t++) {
         *(m + i*n + j) += *(m1 + i*n + t) * *(m2 + t*n + j);
      }
   }
}
```

Fig.1 - Naive Algorithm

Suggested Approach

The approach suggested by the teacher consists in multiplying each element from M1 by the correspondent line of M2. Using this method, each M3 cell's value is not computed at once, but one line is gradually filled, then the next and so on. The central code of the algorithm is displayed right below, where *i* is the row of the element from M1 and *j* is the column of said element, while *t* is the index variable used to iterate the M2 line.



```
for (int i = 0; i < n; i++) {
   for (int j = 0; j < n; j++) {
      float m1Element = *(m1 + i*n + j);
      for (int t = 0; t < n; t++) {
         *(m + i*n + t) += m1Element * *(m2 + j*n + t);
      }
   }
}
```

Fig.2 - Suggested Algorithm

Evaluation Methodology

For evaluating the different performances among languages, the *Naive* algorithm was implemented in C, Java, C# and Python and the time each took to run was measured for different matrix sizes.

For evaluating the performance differences between the two algorithms, they were both implemented in C and compilation was made with gcc (with -O3 flag). Data Cache Misses (DCM) on caches L1 e L2 were recorded with the help of the PAPI library and also time of execution. These pieces of information were used for calculating the following performance metrics:

$$\text{DCM/FLOP} = \frac{Cache\ Misses}{FLOP} = \frac{Cache\ Misses}{2N^3}$$

$$\text{FLOPs} = \frac{FLOP}{Time\ (s)} = \frac{2N^3}{Time\ (s)}$$

All the previous mentioned experiments were run on a Ubuntu laptop with an Intel Core i5-6200U (2 cores, 4 threads, with 2.30GHz base frequency but capable of reaching 2.80GHz using Turbo Boost) and a 3 MB SmartCache.

Results Analysis

Languages Analysis

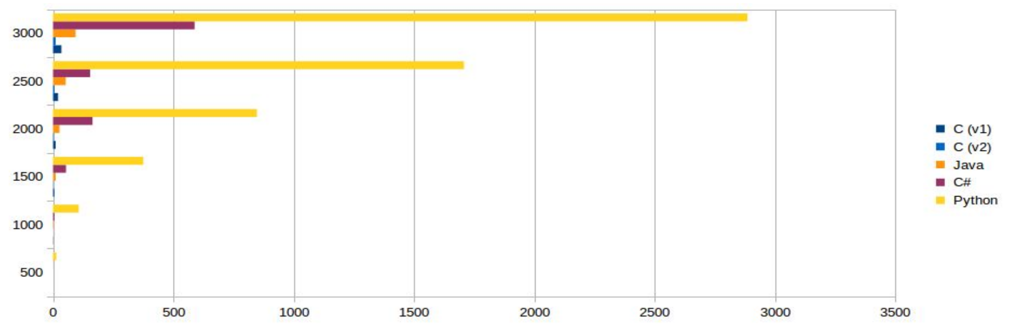


Fig.3 - Languages Comparison

As we can see, Python's execution time is a lot higher than the rest of the languages, as would be expected from an interpreted language.

C# comes next, followed by Java. Although it was expected for Java and C# to have more similar runtimes (as both are compiled to an intermediate language), the most likely reason for this is that C# is more optimized for Windows (.NET) than Linux (Mono).

C++ appears to be the fastest, as it compiles directly to binary code and is known to be a more low-level language than the above.

Algorithms Analysis

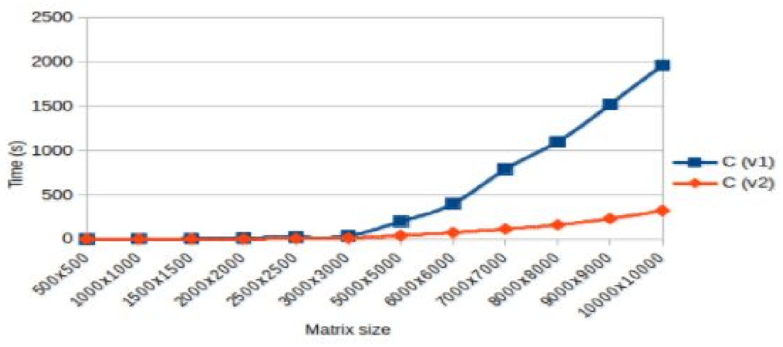


Fig.3 - Algorithms time comparison

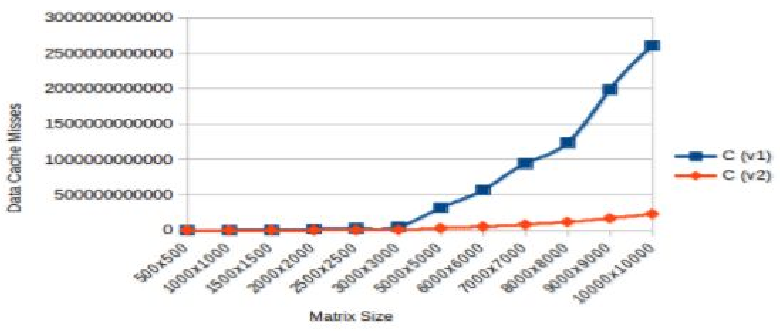


Fig.4 - Algorithms DCM comparison

At first glance, it appears safe to conclude from *Fig.1* that the suggested algorithm (v2) tends to be much faster than the naive (v1) one. The bigger the matrix size, the more discrepant they are from each other.

The reason for the discrepancy above is first found in the amount of failed accesses to cache (Data Cache Misses) during the 1st algorithm. This happens because multi-dimensional arrays are stored in memory in row-major order, which means that lines of the matrix are stored next to each other. So access should be made

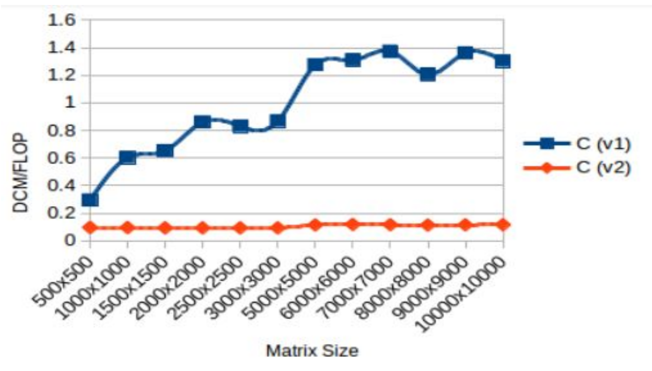


Fig.5 - Algorithms DCM/FLOP comparison

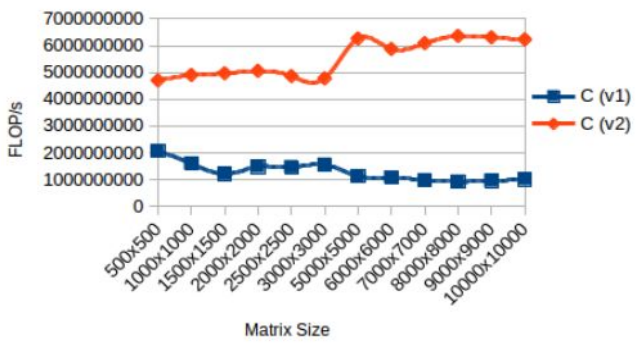


Fig.6 - Algorithms FLOP/s comparison

using that order, as doing it otherwise will require lots of unnecessary “long memory jumps”, which increases the chance of the cache not having the information readily available.
By finding the DCM/FLOP ratio (as shown in *Fig.5*), we understand the reason for such a steep rise of the execution times on the Naive algorithm. The Data Cache Misses per FLOP actually increases with the size of the matrix, which shows how inefficient this algorithm really is.

In *Fig.6*, we can see the actual FLOP/s the CPU performs, which again indicates how efficient the Suggested implementation is compared to the Naive one, since it performs 2.5 to 6 times faster!

Threads Analysis

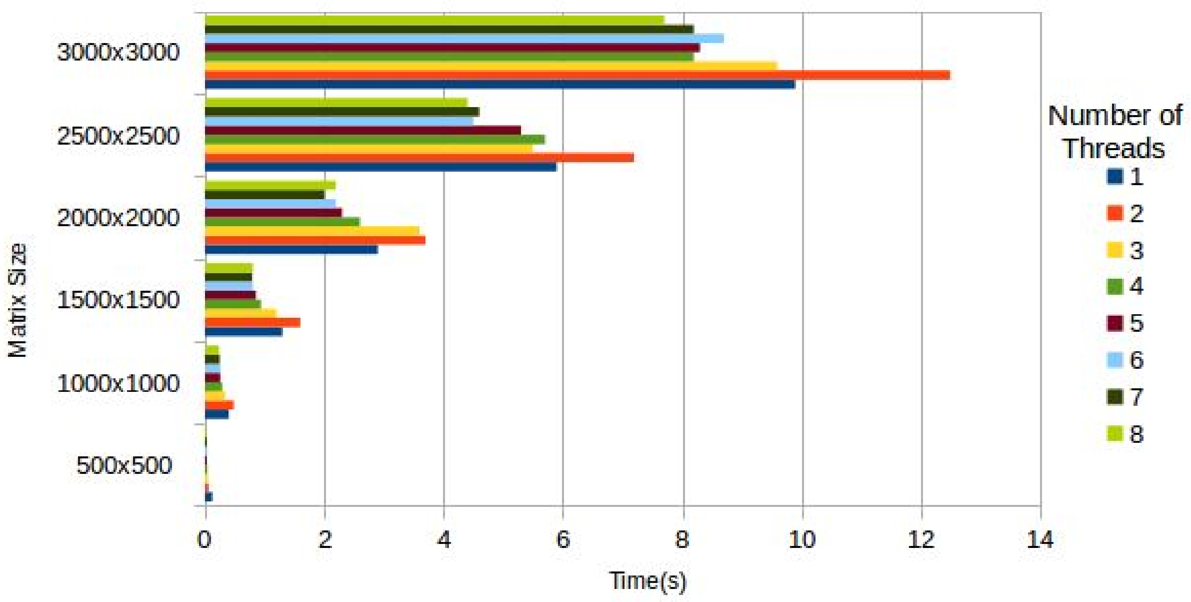


Fig.7 - Execution time using threads (on the suggested algorithm)

Naively, one might expect that a higher number of threads would **always** wield better execution times. However, as we can see on *Fig.7*, that’s not 100% true. As we can see, usually double-threading is actually worse even than single-threading, because resources are spent on managing the threads, but there are only 2, and therefore it's a lot of work for little profit (which doesn't even become profit). Even so, by taking a look at the results with a 1000x1000 matrix, multi-threading still doesn't seem to make so much difference; and even through the 1500x1500 to 2500x2500 matrices we don't see a big difference between 6 and 8 threads. This means that the amount of work we intend to has a heavy impact on how much threads are needed for a faster workflow, but sometimes more threads don't actually make the process faster.

Conclusions

As this assignment had three main sections, several conclusions were made regarding these sections.

Regarding the performance difference among languages, the charts show a clear difference between the levels of compilation of the languages. The closer the compilation is to binary code, the better performance they will have.

Regarding the difference between the 2 algorithms, the experiments showed that a naively-implemented algorithm may heavily reduce the number of FLOPS, thus directly influencing the execution time by forcing more direct access to memory.

Last but not least, regarding the threads comparison, it was possible to conclude that, although threads can increase performance, it depends a lot on the situation, as smaller operations will not benefit so much because a higher percentage of the time will be spend on managing threads. Also, too many threads may not provide a better execution time, exactly for the same reason as expressed before.

To sum it all up, when programming, we should always be careful on how we code our solutions, as minor and *naively-programmed* code sections may have a big impact on the final performance.