Universidade do Porto

Faculdade de Engenharia

FEUP

Performance evaluation of a single core

Mestrado Integrado de Engenharia Informática e Computação

Parallel Computing

Authors:
João Carlos Eusébio Almeida - 201306301 - up201306301@fe.up.pt
João Gabriel Marques Costa - 201304197 - up201304197@fe.up.pt
Nuno Gonçalo Neto Silva - 201200642 - ei12187@fe.up.pt

13th of March 2017

Index

Introduction

For the Parallel Computing course, the class was proposed a study of the impact that the memory hierarchy has in a processor's performance, particularly when accessing the memory involves reading a considerable amount of data.

In this report, the problem regarding the product of the multiplication of two matrices, and its calculated solution through the use of two different algorithms, will be analysed, as well as the experimental results obtained when implementing these algorithms in two different programming languages.

Problem Description

The problem in analysis on this report is the product of the multiplication of two matrices. Given a matrix *A* of size *n* x *m* and a matrix *B* of size *m* x *p*, the matrix product of A and B should be a matrix *C* of size *n* x *p*, where the result of each element is given by the following equation:

$$C_{i,j} = \sum_{k=0}^{m-1} A_{i,k} \cdot B_{k,j}$$

In order to simplify the problem, only square matrices of size *N* x *N* will be considered (i.e. in this case, *n* = *m* = *p*) for this problem.

Algorithms Analysed

For this project, two different algorithms capable of solving this problem were analysed. Both algorithms present an $O(N^3)$ complexity and perform a total of $2N^3$ floating point operations (FLOP). The algorithms analysed in this report are as follows.

Algorithm 1 (onMult)

Algorithm 1 (colloquially named onMult in its code implementation) consists in the successive multiplication each row in matrix *A* with the columns of matrix *B*. An excerpt of its code implementation is as follows:

```
for(i = 0; i < m_ar; i++)
{   for(j = 0; j < m_br; j++)
    {   temp = 0;
        for(k = 0; k < m_ar; k++)
        {
            temp+= pha[i*m_ar + k] * phb[k*m_br + j];
        }
        phc[i*m_ar + j] = temp;
    }
}
```

Image 1: Excerpt from source code for *Algorithm 1* (onMult).

In this implementation, in a single iteration of the cycle indexed by the *i* variable, the row *i* of the matrix *A* and every element of the matrix *B* are accessed (all reads of the elements in matrix *B* are done by column), resulting in a row of the matrix *C* being calculated and written to memory, according to the equation presented in the Problem Description section.

Algorithm 2 (onMultLine)

Algorithm 2 (colloquially named onMultLine in its code implementation) begins by initializing all of the elements of the matrix *C* to 0, and consists in the successive multiplication of all the elements in a row of matrix *A* for all of the rows in matrix *B*. An excerpt of its code implementation is as follows:

```
for(i = 0; i < m_ar; i++)
{   for(k = 0; k < m_ar; k++)
    {   for(j = 0; j < m_br; j++)
        {
            phc[i*m_ar + j] += pha[i*m_ar + k] * phb[k*m_br + j];
        }
    }
}
```

Image 2: Excerpt from source code for *Algorithm 2* (onMultLine).

In this implementation, in a single iteration of the cycle indexed by the *i* variable, the row *i* of the matrix *A* and every element of the matrix *B* are accessed (all reads of the elements in matrix *B* are done by row), resulting in a row of the matrix *C* being calculated and written, according to the equation presented in the Problem Description section.

Evaluation Methodology and Results Analysis

Evaluation Methodology

In order to evaluate each of the algorithms, several experiences were conducted using the C++ and Java programming languages alongside the PAPI library (for the C++ portion of the experiments). The overall execution time of the algorithms and the number of data cache misses in the L1 and L2 caches were measured and used to calculate the number of data cache misses per FLOP for L1 and L2, through the use of the following equation:

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

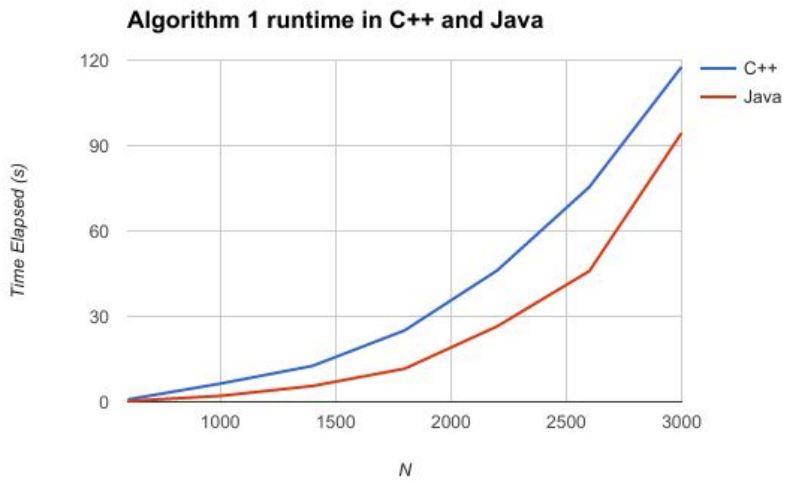
These measurements were also used for the calculation of the number of floating operations per second through the use of the following equation:

$$FLOPS\ =\ \frac{FLOP}{Time\ Elapsed\ (s)}\ =\ \frac{2N^3}{Time\ Elapsed\ (s)}$$

These experiments were conducted in a laptop with an Intel Core™ i7-4870HQ 4-core CPU with a base frequency of 2.50GHz (capable of reaching 3.70GHz through "Turbo Boost"). The CPU incorporates four L1 data caches of 32KB (one for each core), four L2 data/instructions caches of 256KB (one for each core) as well as one L3 data/instructions cache of 6MB (shared by the four cores).

Results Analysis

Algorithm 1 performance in C++ and Java



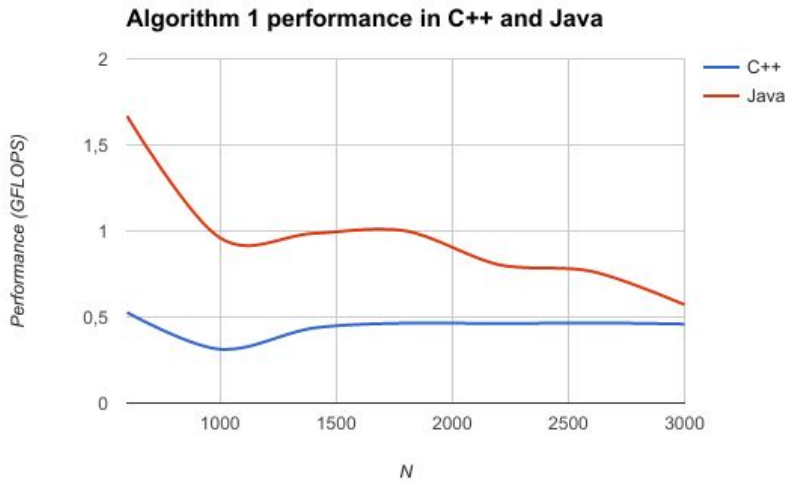
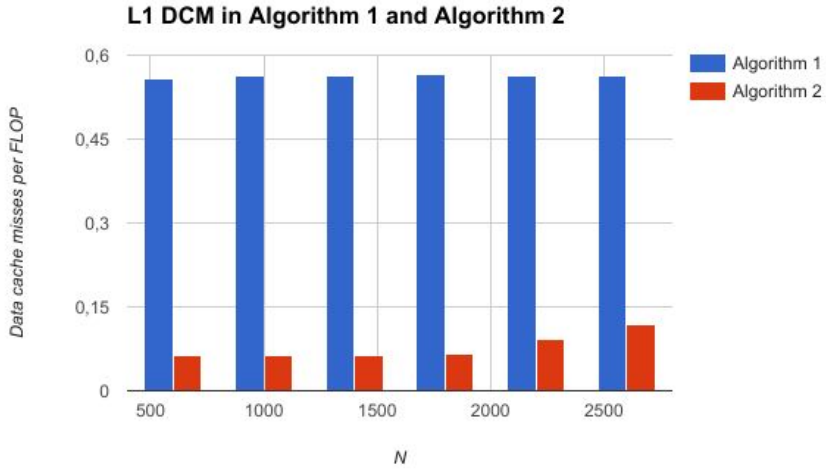


Image 3: Runtime (on the graph above) and performance (on the graph below) for *Algorithm 1* (onMult) in the C++ and Java programming languages.

Analysing the graphs in Image 3, it's possible to conclude that the implementation of *Algorithm 1* (on Mult) achieves significantly better performance in the Java programming language compared to its C++ counterpart when *N* is between the 600 and 3000 value. This observation is a bit surprising, seeing as how the Java programming language is considered a slower overall language than C++ in terms of performance, in spite of recent strides in this area made by recente Java updates. It's also important to note the decaying performance over the increasing *N* values in both languages, as to be expected taking into consideration the exponentially increasing number of calculations that need to be done for bigger *N* values.

Cache impact in the performance of the algorithms



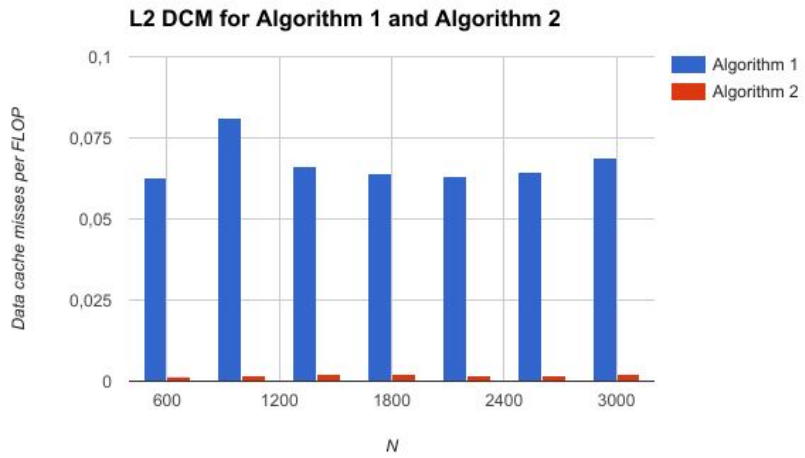


Image 4 - Data cache misses per FLOP in the L1 cache for *Algorithm 1* (onMult) and *Algorithm 2* (onMultLine) (in the graph above) and in the L2 cache for *Algorithm 1* and *Algorithm 2* (in the graph below).

Whenever a value stored in the RAM is read, a block of memory not only containing the value read, but also any values stored near it are copied to the cache (a phenomenon known as the “Principle of Locality”) which, in the context of this problem, will correspond to any elements adjacent to the value read positioned in the same row, or in immediate previous or following rows. The cache copy of this block of memory is stored with the purpose of ensuring faster read times, since any values near the value read are expected to be accessed themselves in the near future.

As such, it is to be expected that *Algorithm 2* (onMultLine), that reads each matrix row by row, to present better overall performance. This can be verified on the following graph:

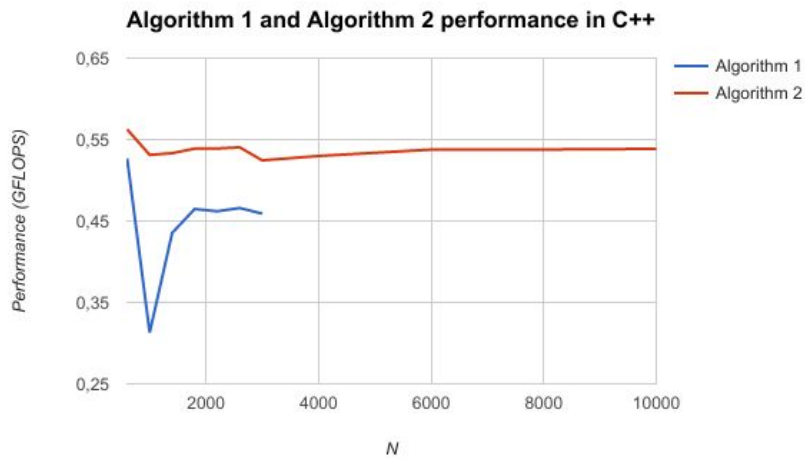


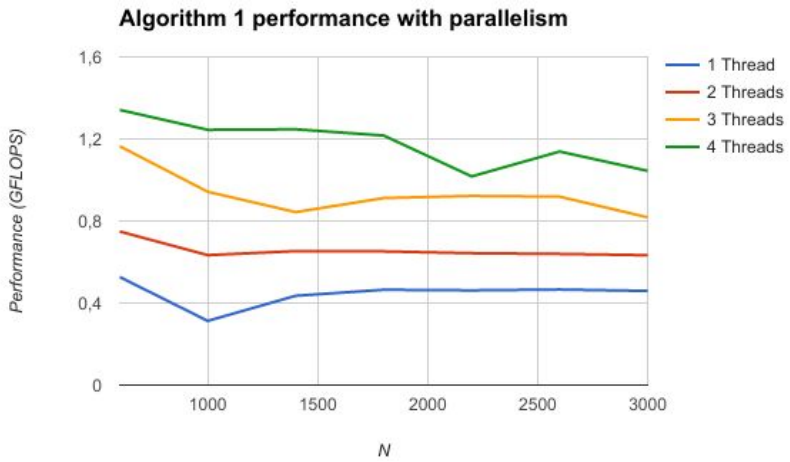
Image 5 - *Algorithm 1* (onMult) and *Algorithm 2* (onMultLine) performance in GFLOPS in the C++ programming language.

The reason for this performance improvement is visible on Image 4, where *Algorithm 2* (onMultLine) presents a significantly lower number of data cache misses per FLOP on both the L1 and L2 caches than *Algorithm 1*.

For *Algorithm 1*, starting at *N* = 1400, the data cache misses per FLOP in L1 and L2 stabilize at about 0.563 and 0.063 (Image 4), respectively, and its performance at 0.46 GFLOPS. In the case of *Algorithm 2*, the data cache misses per FLOP in L1 and L2 stabilize a bit later, when *N* = 2600, at about 0.125 and 0.0018 (not pictured), respectively, and its performance at 0.53 GLOPS.

Concluding this section, it's clear that *Algorithm 2* ends up being the superior algorithm across the board in the C++ programming language, as evidenced by its significantly lower number of data cache misses per FLOP on both the L1 and L2 caches and better performance over *Algorithm 1*. This is due in large part over the implementation of these algorithms following a row-major order in the storage of the matrices in the RAM alongside the aforementioned principle of locality.

Parallelism impact in the performance of the algorithms



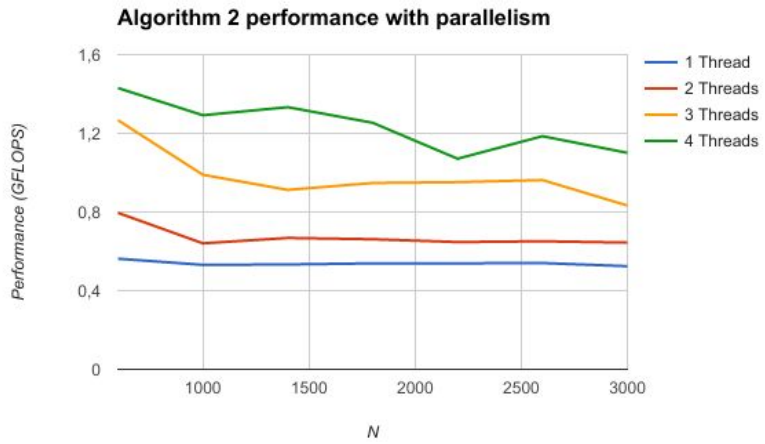


Image 6 - *Algorithm 1* (onMult) (graph above) and *Algorithm 2* (onMultLine) (graph below) performance in GLOPS from one to four threads.

In order to analyse the effect parallelism can have in the performance of both algorithms, the OpenMP library was used to create parallelism in the *for* cycle indexed by the *i* variable in the algorithms.

It's possible to conclude, from analysing the results displayed in the graphs in Image 5, the performance of both algorithms increases with the number of threads, reaching a maximum of roughly 1.3 GFLOPS for *Algorithm 1* and 1.4 GFLOPS for *Algorithm 2* when N equals 600.

The performance improvement over the sequential version of both algorithms to its parallel counterparts is fairly significant, hovering between 2 to 2.5 times better for both algorithms when N is between 400 and 3000 and the number of threads is equal to 4.

It's important to note here that the performance improvements for an increasing number of threads or even between algorithms is rather small here due to the limiting computer power available, and these improvements would be more obvious with a better performing CPU.

Conclusions

This project, which consisted in the analysis of two algorithms with different ways of accessing the same data, allowed us to study and deepen our knowledge on the influence of different hierarchical types of memory in different programming languages, on a fairly simple but nonetheless relevant problem of matrix multiplication.

We were able to conclude that the row-major order policy employed by the different programming languages analysed in the storage of arrays in RAM favor *Algorithm 2* (onMultLine), which benefits from a significant boost to its performance with parallelism (between 2 to 2.5 time better for 4 threads), achieving peak performance of 1.4GFLOPS.