U. PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Computação Paralela e Distribuída

1º Trabalho Laboratorial

Computação Paralela e Distribuída 2023/2024
LEIC 3º Ano - Turma 4 - Grupo 17

João Tiago Morais Lourenço (up202108864)
Rodrigo Gonçalves Figueiredo (up202108870)
Simão Queirós Rodrigues (up202005700)

17 de março de 2024

1. Problem Description

In this project, our objective was to study the effect on the processor performance of the memory hierarchy when accessing large amounts of data. Our case study was the multiplication of two matrices that, with the help of the Performance API(PAPI), enabled us to collect relevant performance indicators during the program execution, to arrive at the proper conclusions.

2. Algorithms

In order to better understand the effects that a proper algorithm implementation can have, it was demanded the development of 3 different algorithms using sequential programming (one of them being posteriorly developed in parallel programming). The main factor behind what makes an algorithm better in the sequential programming scenario, in the case study, is the memory manipulation that, as we will show, can heavily impact the performance.

The implemented algorithms are the following:

2.1. Simple Matrix Multiplication

This is the simplest implementation of all, by simply multiplying each line from the first matrix by each column of the second matrix. The implementation of this algorithm was given in C/C++, but it was requested to be implemented in another language, from which we chose JAVA. Both languages were tested with square matrixes with sizes between 600 to 3000 (400 increments every time).

```
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;
    }
  }
```

2.2. Line Matrix Multiplication

This algorithm takes a step further by multiplying each cell from the first matrix by the corresponding line from the second matrix. At first glance, the code changes are minimal, each they are, but the impact that has on the performance is quite outstanding as we will show. This algorithm was also implemented by us in C/C++ and JAVA, being tested with square matrixes with sizes between 600 to 3000 (400 increments every time) and 4096 to 10240 (2048 increments every time). This algorithm is the one mentioned before that was also implemented using parallel programming to better understand the performance impacts that it implies.

```
//Sequential Version
 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];
      }
    }
  }
```

```
//First Parallel Version
#pragma omp parallel for
  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];
      }
    }
  }
```

```
//Second Parallel Version
 #pragma omp parallel
  for (i = 0; i < m_ar; i++) {
    for (k = 0; k < m_ar; k++) {
      #pragma omp for
      for (j = 0; j < m_br; j++) {
        phc[i * m_ar + j] += pha[i * m_ar + k] * phb[k * m_br + j];
      }
    }
  }
```

2.3. Block Matrix Multiplication

Lastly, this algorithm is distinct from the previous two, due to the fact that it takes both matrixes and subdivides them into blocks with fixed size. This algorithm was implemented only in C/C++ and tested with square matrixes with sizes between 4096 to 10240 (2048 increments every time).

```
  for (bi = 0; bi < m_ar; bi += bkSize)
  {
    for (bk = 0; bk < m_ar; bk += bkSize)
    {
      for (bj = 0; bj < m_ar; bj += bkSize)
      {
        // Perform mutiplication on blocks
        for (ii = bi; ii < bi + bkSize; ii++)
        {
          for (kk = bk; kk < bk + bkSize; kk++)
          {
            for (jj = bj; jj < bj + bkSize; jj++)
            {
              phc[m_ar * ii + jj] += pha[m_ar * ii + kk] * phb[m_ar * kk + jj];
            }
          }
        }
      }
    }
  }
```

3. Performance Metrics

As we said previously, we used the tool PAPI to collect the proper performance indicators, since it has access to a set of measures on the CPU level during the program's execution. During our analysis, we accounted for the execution time, the number of Floating Point Operations (FLOP) and the number of cache misses for L1 and L2, due to the fact that each cache miss has an impact on the performance.

To ensure the integrity of the collected data, the same computer was used throughout the process and each measurement was collected more than once to decrease the impact of random effects as much as possible. The computer used was running Ubuntu 22.04 and had an i7-9700 single clocked up to 4.7 GHz.

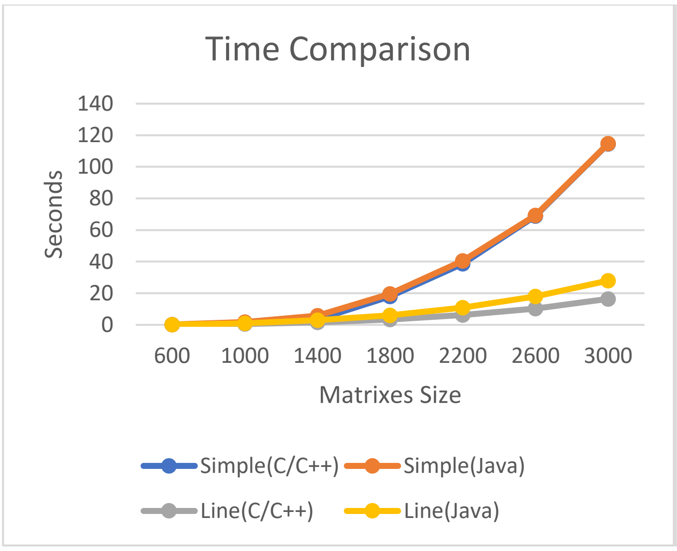
The metrics on the CPU level were only extracted every time the C/C++ version was running, the JAVA version was used to provide us with a comparison of execution time between the two languages. The C/C++ was always compiled using the -O2 optimization flag that, although it increased the compilation time (which wasn't important on our case study), it improved the performance of the generated code.

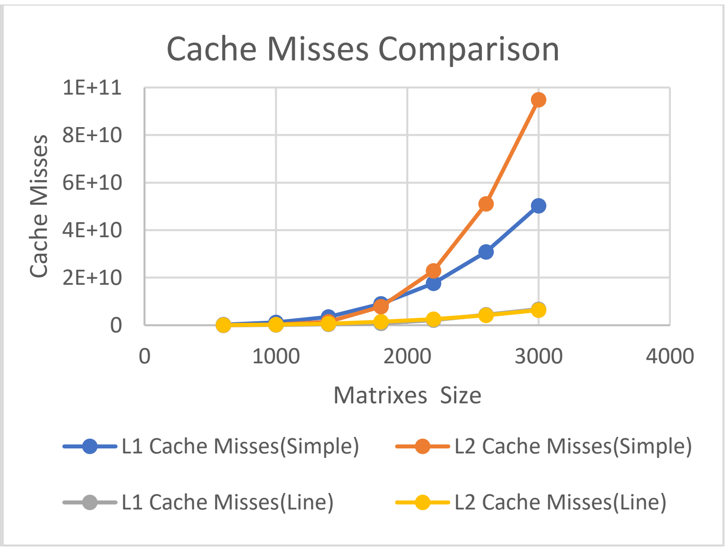
For the parallel versions, we only measured the execution time to allow us to compare to the respective sequential version by calculating the speedup and efficiency to conclude if the parallelism gives us the desired performance boost.

4. Results and Analysis

The following results are derived from the average of the collected data and are divided into the sections we concluded were more relevant to analyze.

4.1. Simple and Line Algorithms Comparison

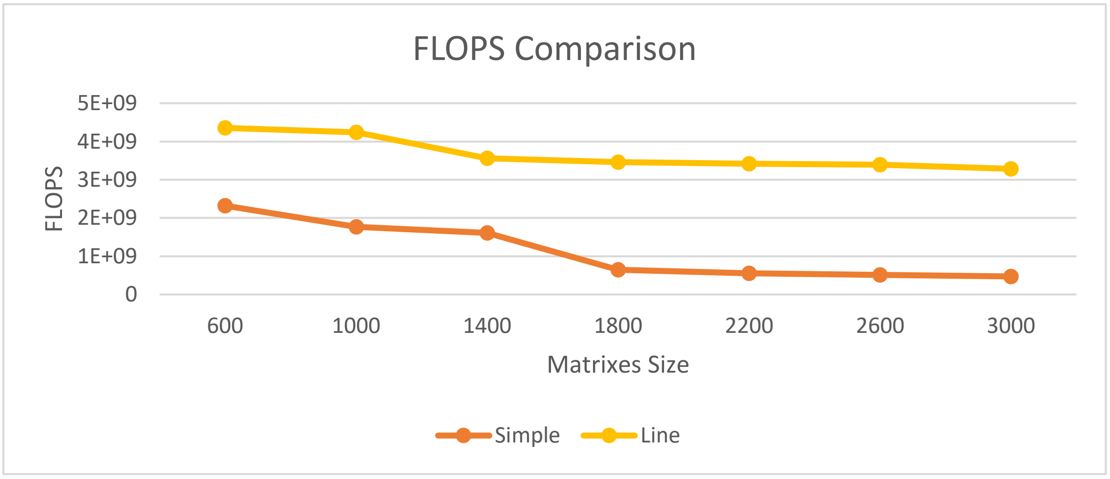




After analyzing both algorithms, the first thing that can be noticed is the significant difference between the execution time and the cache misses that both algorithms have. It is quite perceptible that one of the main factors that leads to the best performance of the line algorithm is the reduced number of cache misses that it has overall. This happens due to the fact that this algorithm goes accordingly to how the memory is organized, leading to visible time reduction.

Another visible aspect is the similarity in execution between C/C++ and Java, but this was to be expected since both are compiled and follow similar paradigms.

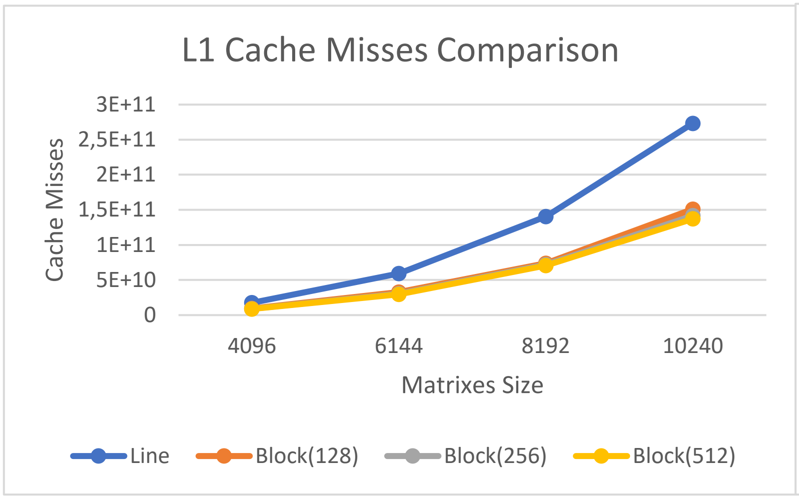
Our conclusion that the line algorithm is more efficient can be reinforced by looking at how the FLOPS behave when the matrixes size is increased:

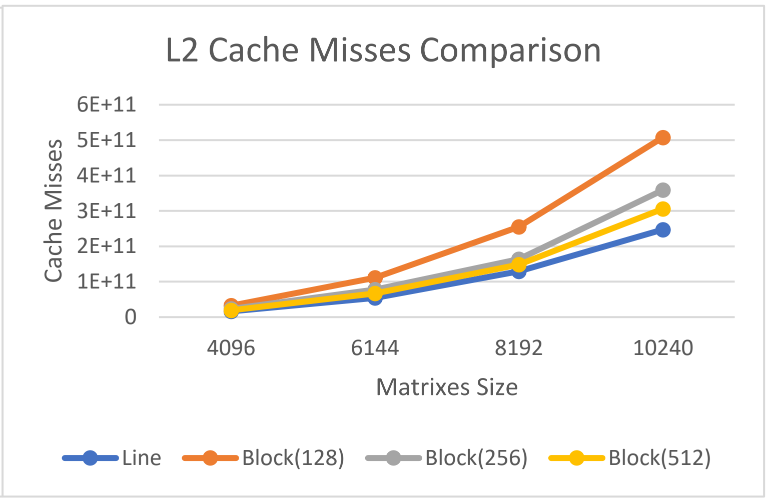


Since both algorithms perform the same operations, it is expected for both to have the same number of floating point operations (FLOP), but, as we can see, when it comes to floating point operations per second (FLOPS), it is a totally different story. We can see a more destabilized performance by the simple algorithm as the matrixes size increases, which goes along with what we theorized since this algorithm will "waste" more time fetching data on higher level memories, an action that can be proved by the higher number of cache misses.

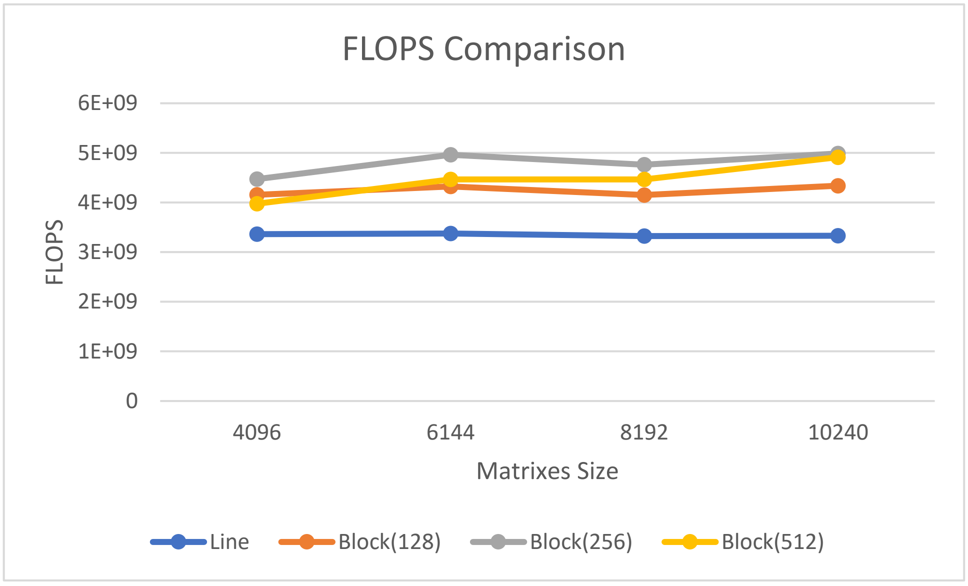
4.2. Line and Block Comparison

After comparing the two previous versions, we decided to compare the best from the previous ones (Line Algorithm) with the Block Algorithm to see if this new approach could significantly improve the performance:





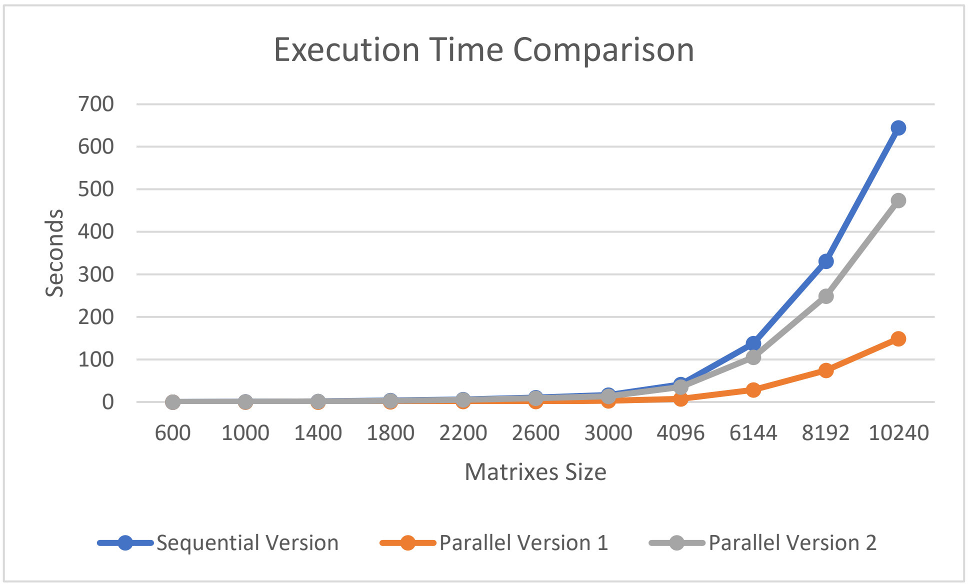
As we can observe, there are two main points that are immediately identified: the Line algorithm has a significantly higher amount of L1 cache misses, but it is the one that has lower L2 cache misses. Knowing this fact, what can we conclude about how that will affect the performance? Seeing that the L1 cache misses are more frequent and costly, it is to expect the block algorithm to have better performance, which can be verified by comparing the FLOPS, since both perform the same amount of floating point operations:



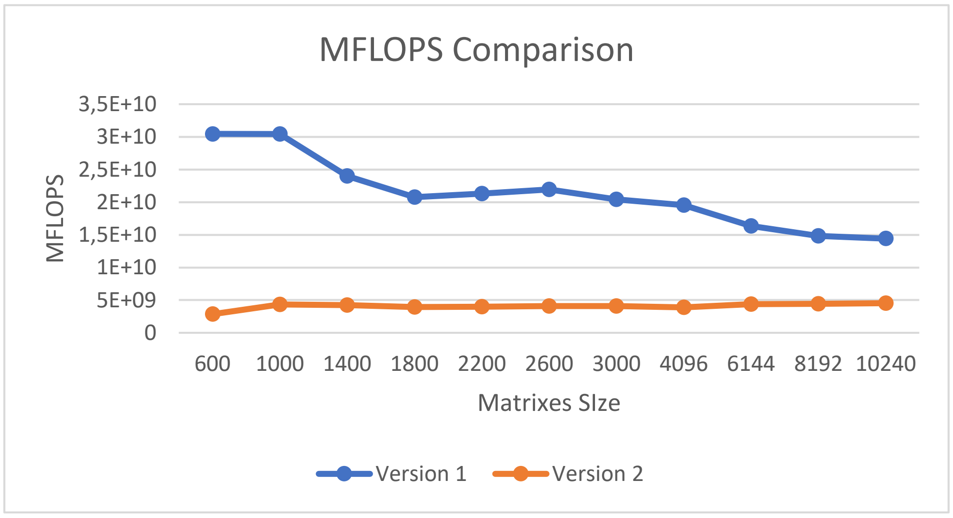
Verifying that the block algorithm is in fact the one with better performance, there's another question that must be answered, why does it have more L2 cache misses when it is able to have fewer L1 cache misses? This phenomenon occurs because of the logic behind this algorithm and the cache's specifications, since the algorithm divides the matrixes into blocks with fixed sizes, depending on the size we are working on, the algorithm can take most or less advantage of the cache available sizes.

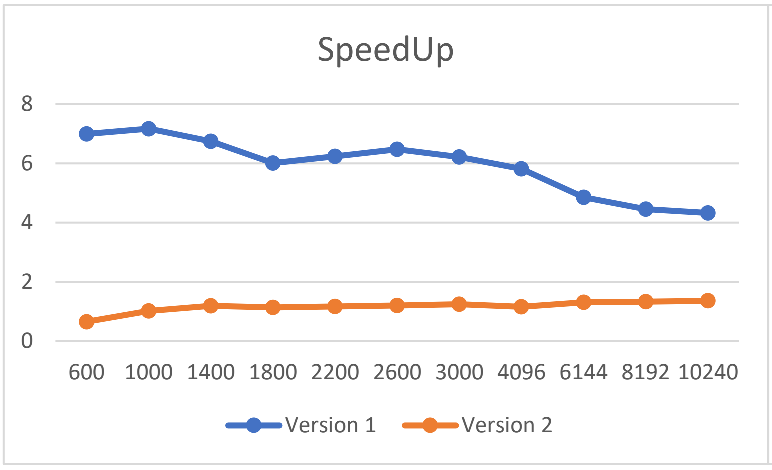
4.3. Line Sequential and Line Parallel

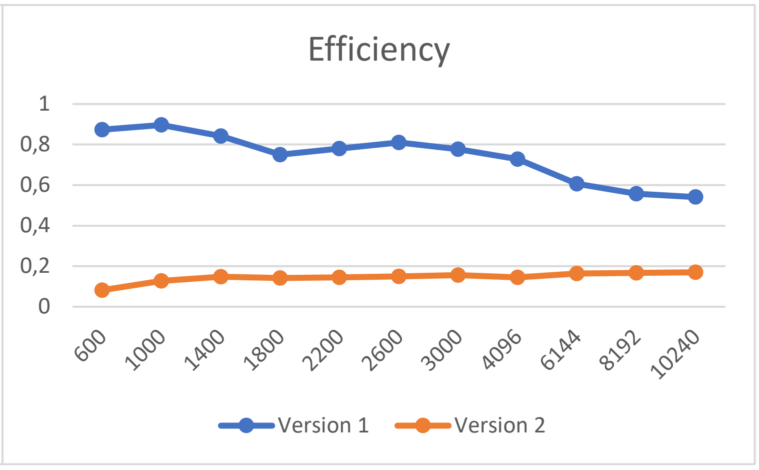
After comparing these different sequential approaches, it was asked to take a step further and enter the parallelization paradigm to see how much the performance could be improved. To help us with that, it was given to us two similar versions of the line algorithm we previously used so what was left to do was analyze and compare to the sequential version:



While measuring the time for the first version, nothing out of the ordinary was verified, as it is visible on the graphic, the parallel version proved to be much more efficient has it was expected. However, when we started measuring for the second version, the execution time didn't have much improvement. This lack of improvement compared to the first version kept being clearer as we started to analyze more performance metrics:







As observed, the second version really lacks the performance improvement to justify using it instead of the sequential version, since it can even achieve a speedup of 2. On the other hand, the first version pops off because it has a desirable speedup and efficiency, although not as stable, making the choice to use it more preferable than the sequential version.

Given the unexpected result with the second version, we decided to take a more careful look at it to see if we could find the reason behind the lack of improvements. Being the difference the additional parallelization of the inner loop, we concluded that this addition was unnecessary since it was parallelizing simple arithmetic operations in the context of the problem causing an overhead instead of making the process more efficient. This is a good example that, even though parallel programming is an extremely powerful and useful tool, it must be used with care making necessary not only a good understanding of the case study it is being used, but also a good theoretical knowledge to achieve the best outcomes.

5. Conclusions

This project allowed us to better understand the importance of memory management to improve the performance of a program, where what may seem like a small code change can have drastic results. In the case of sequential programming, the correct manipulation of memory has shown to be essential for better results by taking into account the hardware we are using and avoiding wastes of time with constant data fetching(especially when manipulating large amounts of data). On the other hand, the adoption of parallel programming has shown the great improvements it can bring when well applied, making, when possible, its use a great tool to improve performance. However, it was also quite clear that we must not overkill its use, making necessary a good analysis of when it must be applied to take the most of it.