Computação Paralela e Distribuida

David Marques

Sofia Moura

José Ramos

March 10, 2023

Contents

1 Introduction

In this project we aim to compare the execution time of 3 algorithms for multiplying matrices: Line-by-Column Multiplication, Line-by-line multiplication and Block multiplication. We also want to study the difference in execution time of two languages C++ and Java.

2 Problem Description

The processor performance is highly affected by the memory hierarchy when we're dealing with large amounts of data.

In this project, we used Performance API (PAPI) to understand the effects of the memory hierarchy, which provided us with multiple performance counters to track the performance of a system that is significantly dependent on how data is stored in memory. This system is based on 3 matrix multiplication algorithms in both C++ and Java for increasing matrix inputs. Level 1 and 2 cache misses were also tested and analyzed.

3 Algorithms Explanation

For each algorithm used, we start by initializing the first matrix with 1.0 for every position and the second matrix starting with 1 and increasing 1 for each row.

```
double *pha, *phb, *phc;

 pha = (double *)malloc((m_ar * m_ar) * sizeof(double));
phb = (double *)malloc((m_ar * m_ar) * sizeof(double));
phc = (double *)malloc((m_ar * m_ar) * sizeof(double));

for(i=0; i<m_ar; i++)
   for(j=0; j<m_ar; j++)
      pha[i*m_ar + j] = (double)1.0;



for(i=0; i<m_br; i++)
   for(j=0; j<m_br; j++)
      phb[i*m_br + j] = (double)(i+1);
```

3.1 Line-by-Column Multiplication

This algorithm was the one that was given initially to us. Its implementation is pretty straightforward, all we are doing is multiplying each element of the column of the first matrix by each element of the line of the second matrix. In mathematical terms it looks like this:

$$\underset{\text{Matrix } \mathbf{A} \text{ is } 3x4}{\begin{bmatrix} 8 & 3 & 0 & 1 \\ . & . & . & . \\ . & . & . & . \end{bmatrix}} \underset{\text{Matrix } \mathbf{B} \text{ is } 4x4}{\begin{bmatrix} 5 & . & . & . \\ 4 & . & . & . \\ 3 & . & . & . \\ 1 & . & . & . \end{bmatrix}} = \underset{\text{Matrix } \mathbf{C} \text{ is } 3x4}{\begin{bmatrix} 53 & . & . & . \\ . & . & . & . \\ . & . & . & . \end{bmatrix}}$$

$$\text{because } c_{11} = \sum_{k=1}^{4} a_{1k} b_{k1} = 8 \cdot 5 + 3 \cdot 4 + 0 \cdot 3 + 1 \cdot 1 = 53$$

Figure 1: An example of matrix multiplication .

This algorithm is composed by two for statements that traverses both matrices. After that we use another for to traverse the pha matrix so that we can multiple each element of the column of the pha by each element of the line of phb, we then add the results of those multiplications in a temporary variable temp which will be then added to the result matrix phc.

```
 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;
  }
}
```

3.2 Line-by-line multiplication

Line-by-line multiplication is one of the methods used to multiply 2 matrices, where each result element in the final matrix is calculated by the sum of the multiplication between the corresponding line of the first matrix and column of the second one.

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

This algorithm is based on going through every position in the resulting matrix, by iterating through every line and column. The values stored in those positions are calculated by multiplication between the corresponding elements in the first and second matrices, and accumulated with the previous multiplications. Generally speaking, the result position from line i and column k of the result matrix is calculated by the product of the element in line i and column k of the first matrix with the element in line j and column k of the second matrix and accumulated with the previous multiplications from that same line and column.

3.3 Block-by-Block multiplication

This algorithm is very similar to the line-by-line algorithm but uses blocks so that we don't do all of the calculations at once. We use one block per matrix.

In this algorithm we first divide the matrices in blocks and then instead of multiplying the entire matrix we multiply the blocks, then we add the blocks to result in blocks of the resulting matrix.

```
 for (int ib = 0; ib<m_ar;ib+=bkSize)
     for (int jb = 0; jb<m_ar;jb+=bkSize)
     for (int kb =0;kb<m_ar;kb+=bkSize)
         for (i =ib;i<min(ib+bkSize,m_ar);i++)
             for (j =jb;j<min(jb+bkSize,m_ar);j++)
                 for (k=kb;k<min(kb+bkSize,m_ar);k++)
                 phc[i*m_ar+k] += pha[i*m_ar+k]*phb[j*m_ar+k];
```

3.4 Time and Space Complexity

All the three algorithms have the same time and space complexity being O(n^3) the time complexity and O3n the space complexity.

4 Performance Metrics

To analyze the results of our work we decided to use some parameters other them the execution time to compare the different algorithms. We used several performance metrics available in PAPI.

- L1 Data Cache Misses:
- L2 Data Cache Misses:
  - This two parameters gives the number of data misses of level 1 and level 2 cache respectively, we used this metrics to be able to see how the different algorithms can take advantage of the memory system.
- L1 Instruction Cache Misses:
- L2 Instruction Cache Misses:
  - This two parameters gives the number of instruction misses of level 1 and level 2 cache respectively, like the two metrics before this ones we used this to compare the usage of memory of the algorithms.
- L2 Total Cache Reads: We used this metric to see how many times the level 2 cache would be read from.
- L2 Total Cache Writes: We used this metric to see how many times the level 2 cache would be written to.
- Total Instructions: we used this metric to see the total number of instructions of each algorithm.

Unfortunately the Performance API is not available for java so we only have this data regarding the execution of the algorithms in C++.

5 Results

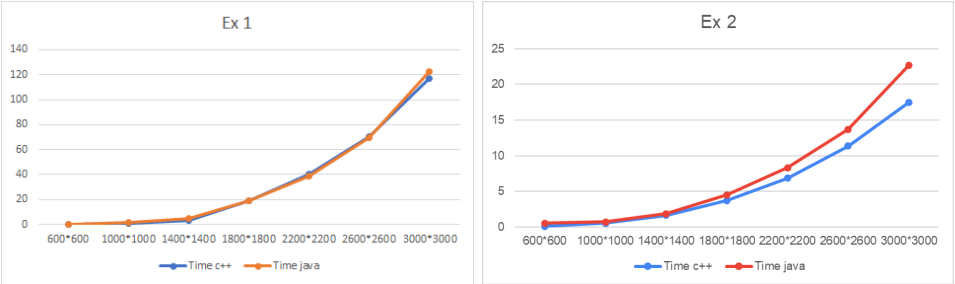

Figure 2: Time comparison of C++ and Java.

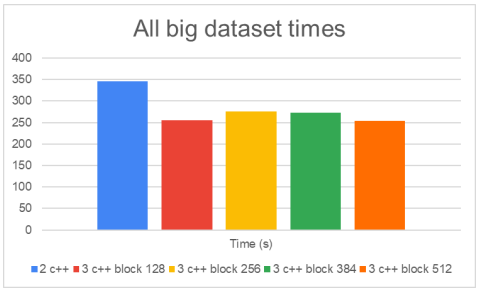

Figure 3: Time comparison of algorithms 2 and 3 in java.

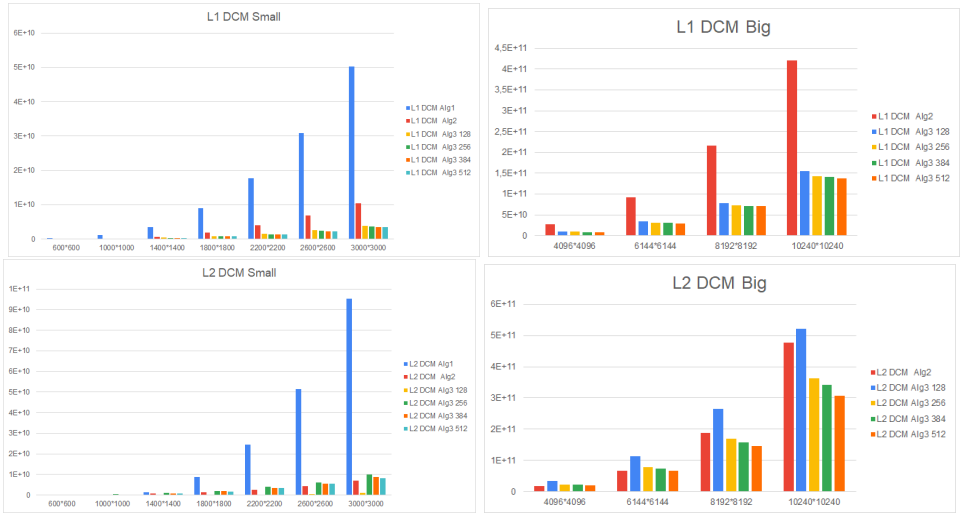

Figure 4: Comparing between the DCM metrics of the 3 algorithms.

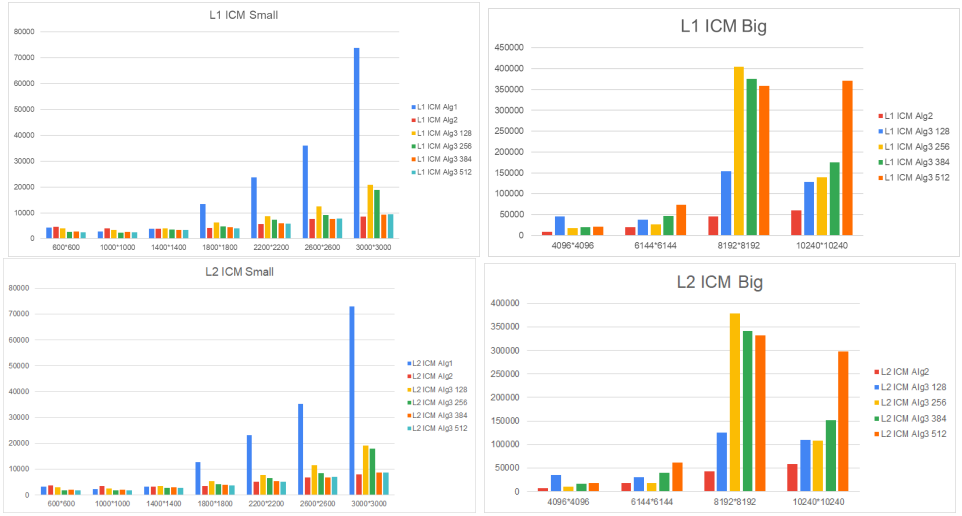

Figure 5: Comparing between the ICM metrics of the 3 algorithms.

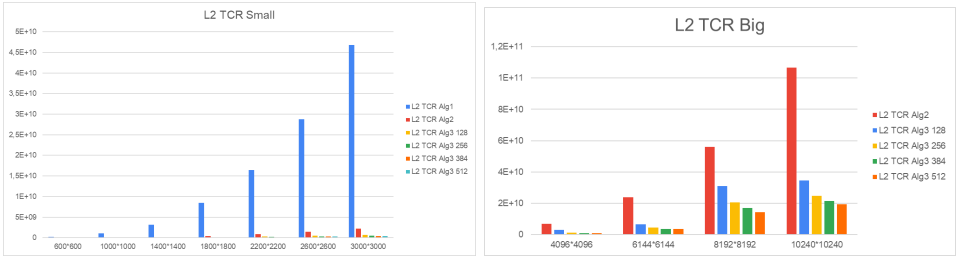

Figure 6: Comparing between the TCR metrics of the 3 algorithms.

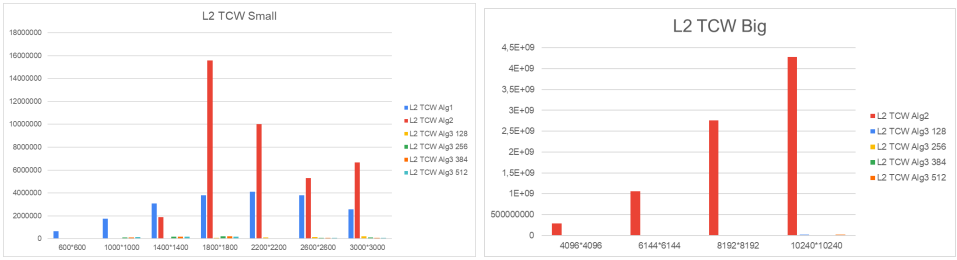

Figure 7: Comparing between the TCW metrics of the 3 algorithms.

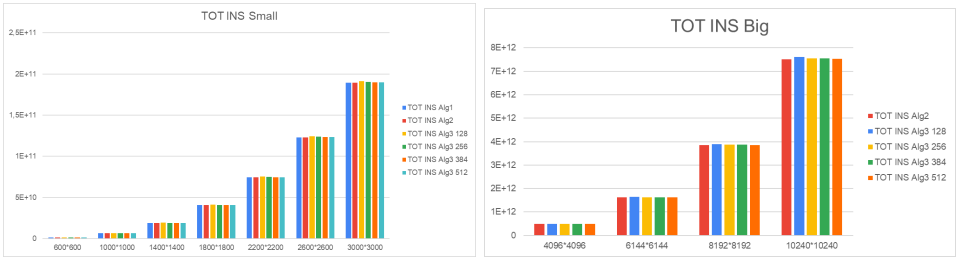

Figure 8: Comparing between the total instructions metrics of the 3 algorithms.

6 Results Analysis

Comparing the two languages that we used we can see that C++ has better execution times than java.

6.1 Time

Analysing the metrics that we collected we can see that overall the execution time of the algorithm 1 is the worst and the algorithm 3 is the best the only exception being in when comparing the 2nd algorithm with the 3rd when using a block size of 256 and 384 for matrices of size 8192*8192 in this case the time gets worse but because all the other results point in the other direction we can still confidently say that the 3rd algorithm is more efficient.

6.2 Data cache Misses

Comparing the data cache misses of the different algorithms we can see that the 1st algorithm is the worst in this metric. Focusing on the 2nd and 3rd algorithm,for the level 1 cache we see that regardless of the data set we always see an improvement from the 2nd to the 3rd, for the level 2 cache if used in small matrices (up to 3000*3000 in our testing) the 3rd has always a smaller rate of misses and the better performing version of that algorithm is the one where we use blocks of 128 the smaller that we tested, but for bigger matrices (4096*4096 and 6144*6144) we can observe the opposite, the 2nd having a smaller rate of misses than all of the instances of the 3rd, the story changes again if we look at bigger matrices (8192*8192 and 10240*10240) where the worst in this metric is the 3rd algorithm

when used with blocks of 128. With this we can conclude that overall the 3rd algorithm is the one who performed better looking at this metric and that we should increase the block size of this algorithm when we increase the size of the matrices.

6.3 Instruction Cache Misses

Looking at the instructions cache misses we can easily spot the worst algorithm, which is the 1st algorithm. Has for the 2nd and 3rd algorithm, the second algorithm is either the best or equal to the best third algorithm block size, and within the third algorithm there seams to have better results with the bigger blocks sizes, that is so for the smaller data set. As for the bigger data sets it's now obvious that the second algorithm is better than the rest. For the third algorithm we have some interesting results, has we can see the 128 block size starts off as having the worst results, but as we continue to increase the matrix it doesn't increase has much as the bigger block sizes, in the other hand the other block sizes have strange results on the 8192x8192 size matrix, being that they suddenly brutally escalated their size, only to have them diminished in the 10240x10240 size matrix.

6.4 Total Cache Writes

Analyzing this metric we see some weird results, the algorithm 1 and 2 data for the small matrices data set goes up and down as we increase the size of the matrices, but we can see that for the smaller matrices the 1st algorithm has the bigger values and the reverse happens in the bigger ones, for this data set the only constant data is that the 3rd algorithm is the one with the smaller values, looking at the bigger data set for the 2nd algorithm we see that in this case the value goes always up as we increase the size of the matrices and once again the 3rd algorithm has the smallest values. For the different block sizes of the 3rd algorithm we can see that the smallest size that we tested (128) is the one with the smallest TCW and as we increase the size the TCW varies.

6.5 Total Cache Reads

When we look at this metric we can see that the 1st algorithm is the one that rights more in the cache and the 3rd the one that rights the less, we can also observe that in the 3rd algorithm for bigger block sizes the TCR decreases. This makes sense because the data that we have is related to the level 2 cache and the 3rd algorithm is the one that has the better memory optimization meaning that it should require the use of the higher level caches less.

6.6 Total Instructions

When looking at the total instructions executed by each algorithm we can see that the number is almost constant throughout the entire testing with this information and the time comprising of the algorithms it is possible to conclude that the major factor in the extremely different execution times is the way each algorithm accesses the data in cache, proving once again that the 1st algorithm is the most inefficient and the 3rd the most efficient.

7 Conclusions

After analysing the results we can conclude that the third algorithm is the one that is better optimized to fully utilize the memory of the computer. We could also observer the differences in performance of Java and C++ seeing that C++ had better execution times overall.

8 Annexes

All of our results can be seem in this google sheets.
The code for our project can be found here.