FEUP - Engenharia Informática e Computação

Computação Paralela e Distribuída - 2º Semestre 2023/2024

Turma 7 Grupo 11

João Correia - up202005015@up.pt

Máximo Pereira - up202108887@up.pt

Pedro Estela - up202108892@up.pt

Index

1. **Problem description**
2. **Algorithms**
   - **2.1 Basic matrix multiplication**
   - **2.2 Line matrix multiplication**
   - **2.3 Block matrix multiplication**
3. **Performance metrics**
4. **Results and Analysis**
   - **4.1 Execution time and DCM comparison between Simple and Line Matrix multiplication**
   - **4.2 DCM comparison between Simple and Line Matrix multiplication**
   - **4.3 Sequential and Parallel Linear Matrix Multiplication**
5. **Conclusions**

1. Problem description

In this project, we will study the effect on the processor performance of the memory hierarchy when accessing large amounts of data, and the product of two matrices was used for this study. We made use of Performance API (PAPI) to collect performance indicators of the program execution in C++ and used the time taken for each algorithm to compare the execution in C++ and Java.

2. Algorithms

We tested the performance of a single core using three implementations of a matrix multiplication algorithm, and although they have the same time complexity, they will have different performances, because they use memory differently.
These implementations are:

1. Basic matrix multiplication
2. Line matrix multiplication
3. Block matrix multiplication

The first algorithm was already given to us in C++ and we were tasked with implementing the other two functions in C++ and then the first and second algorithms in another language. We chose to implement them in Java, since both C++ and Java are Object Oriented Languages and their syntaxes are similar.

2.1 Basic matrix multiplication

In this basic implementation, we were given an algorithm that multiplies each line of the first matrix by each column of the second matrix.
This is the pseudo code:

```
for i=0 to matrix1_size:
        for j=0 to matrix2_size:
                temp = 0
                for k=0 to matrix1_size:
                        temp += matrix1[i][k] * matrix2[k][j]
                resulting_matrix[i][j] = temp
```

2.2 Line matrix multiplication

In this algorithm, we implemented it to multiply an element from the first matrix by the corresponding line of the second matrix.
This is the pseudo code:

```
for i=0 to matrix1_size:
        for k=0 to matrix1_size:
                for j=0 to matrix2_size:
                        resulting_matrix += matrix1[i][k] * matrix2[k][j]
```

2.3 Block matrix multiplication

In this algorithm, we take a divide and conquer approach to split the matrices into smaller blocks that are calculated separately and added up.
This is the pseudo code:

```
for a=0 to matrix1_size, a+=blockSize:
  for c=0 to matrix1_size, c+=blockSize:
    for b=0 to matrix2_size, b+=blockSize:
      for i=a to a+blockSize:
        for k=c to c+blockSize:
          for j=b to b+blockSize:
            resulting_matrix[i][j] += matrix1[i][k] * matrix2[k][j]
```

3. Performance metrics

We made use of Performance API (PAPI) to evaluate the performance of the algorithms implemented using C++, allowing us to register the execution time and the number of cache misses for both L1 and L2. A cache miss implies considerable overhead when processing, so measuring the frequency at which they occur is very relevant in order to understand the performance of each algorithm.
We used the same computer to run all the tests and these are its specifications obtained through PAPI:

```
PAPI version : 7.1.0.0
Operating system : Linux 6.5.0-15-generic
Vendor string and code : GenuineIntel (1, 0x1)
Model string and code : Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz (158, 0x9e)
CPU revision : 13.000000
CPUID : Family/Model/Stepping 6/158/13, 0x06/0x9e/0x0d
CPU Max MHz : 4700 CPU
Min MHz : 800
Total cores : 8
SMT threads per core : 1
Cores per socket : 8
Sockets : 1
Cores per NUMA region : 8
NUMA regions : 1
Running in a VM : no
Number Hardware Counters : 10
Max Multiplex Counters : 384
Fast counter read (rdpmc): yes
```

We used the following line to compile the code:

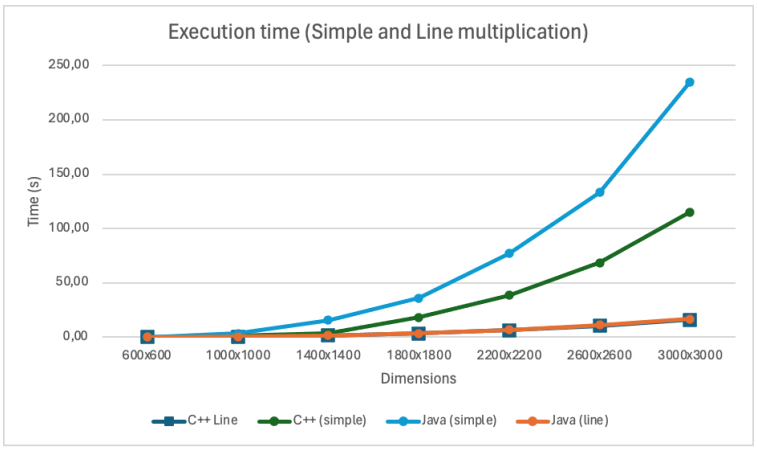
```
g++ -Wall -O2 matrixproduct.cpp -o matrix -lpapi
```

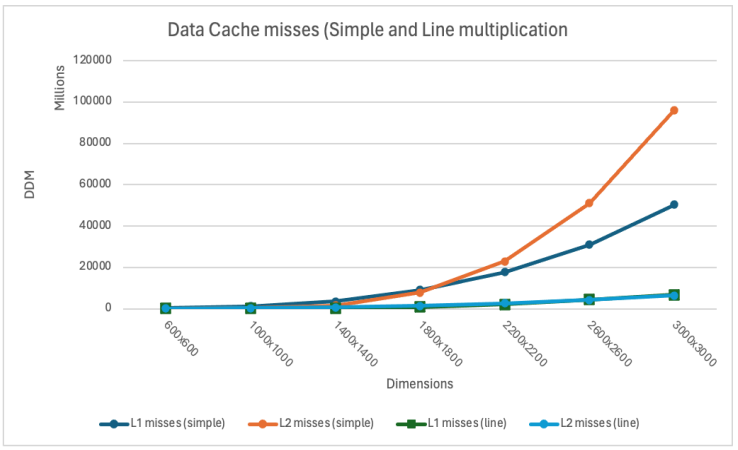
4. Results and Analysis

4.1 Execution time and DCM comparison between Simple and Line Matrix multiplication

For better precision, we've measured the results with four tests for each case. In the following charts, time and data cache misses are presented in relation to different dimensions. Full results will be attached in the Annexes.

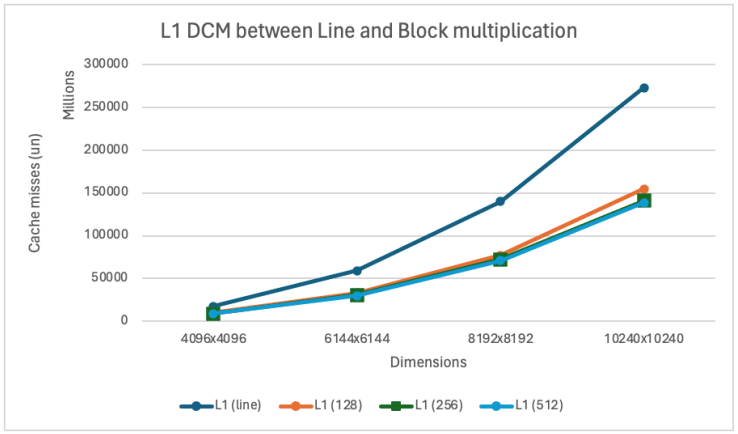
Execution Time and DCM comparison between Simple and Line Matrix Multiplication

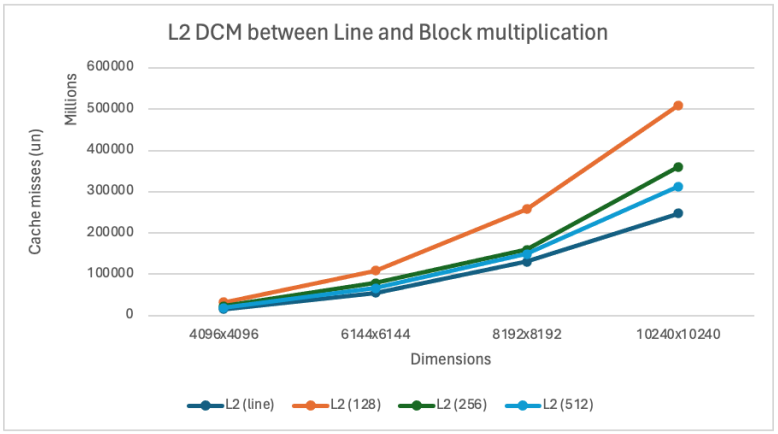




As we can see, the Line algorithm is similar in both languages but in the Simple multiplication, Java achieved a worse performance time. It was expected to have similar times in both algorithms because they have the same programming architecture/paradigm.
We can conclude that the Line multiplication algorithm is more efficient than the Simple multiplication algorithm, having fewer cache misses and better execution time too.

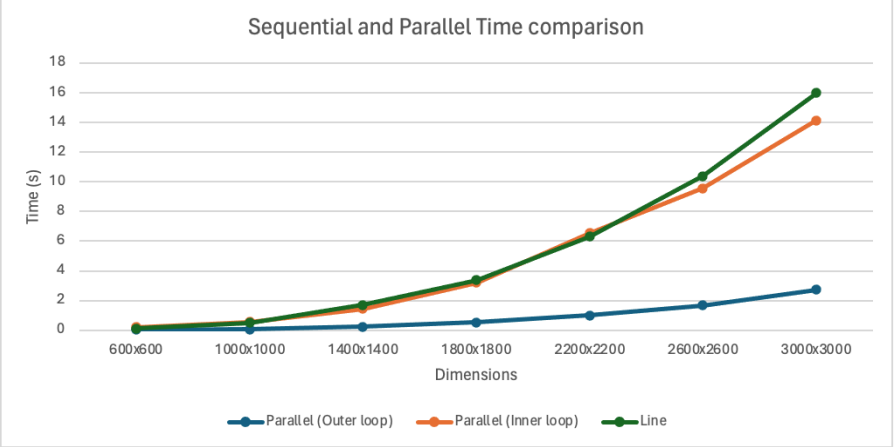
4.2 DCM comparison between Simple and Line Matrix multiplication

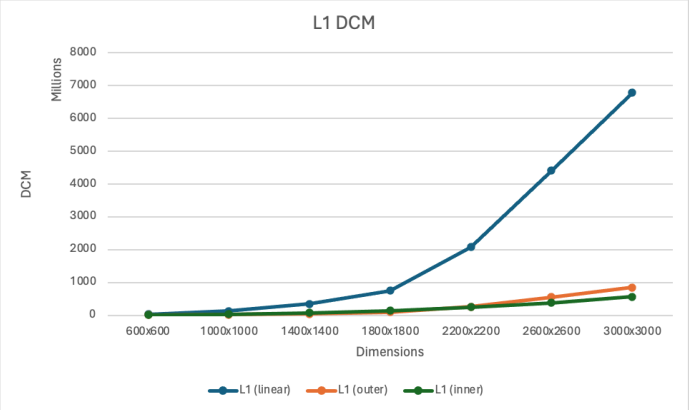


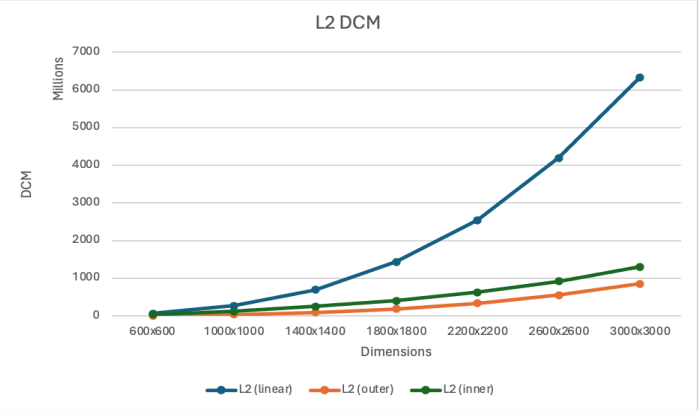


Through the observation of the charts, it is concluded that the Block Matrix algorithm had a lower number of cache misses in L1, in relation to the Line algorithm. However, L2 cache misses are bigger in Block algorithm due to having a worse cache locality than L1 cache. Since the misses in the Block algorithm were mostly in a lower level cache, that causes less overhead than in the Line algorithm, making it perform better.

4.3 Sequential and Parallel Linear Matrix Multiplication







Here, the outer loop refers to the algorithm that performs all three nested loops parallely, and the inner loop refers to the algorithm that only performs the innermost loop paralelly. We can see the outer loop algorithm is much more efficient than the other two, since it makes full use of the cores in the computer's processor, whereas the inner loop algorithm is slightly more efficient than the sequential one, performing much worse than the outer loop algorithm.

Comparison between sequential and parallel (outer loop) linear matrix multiplication

Matrix Dimension	Speed Up (%)	Mflops (seq)	Mflops (parallel)	Efficiency (%)
600x600	757,14	4075471698	30857142857	94,64
1000x1000	717,22	4085801839	29304029304	89,65
1400x1400	739,65	3250703391	24043811610	92,46
1800x1800	628,12	3446808511	21650116009	78,52
2200x2200	631,79	3374960380	21322653317	78,97
2600x2600	618,55	3386757232	20948748510	77,32
3000x3000	588,54	3381022446	19898664210	73,57

Comparison between sequential and parallel (inner loop) linear matrix multiplication

Matrix Dimension	Speed Up (%)	Mflops (seq)	Mflops (parallel)	Efficiency (%)
600x600	55,64	4075471698	2267716535	6,96
1000x1000	86,22	4085801839	3522677235	10,78
1400x1400	117,36	3250703391	3815085158	14,67
1800x1800	106,32	3446808511	3664755322	13,29
2200x2200	96,56	3374960380	3258760520	12,07
2600x2600	108,68	3386757232	3680837696	13,59
3000x3000	112,90	3381022446	3817198600	14,11

5. Conclusions

To sum up, the completion of this work allowed us to deepen our knowledge in benchmarking and efficient memory management for faster program execution time. Even sequential algorithms can be more efficient if proper techniques are applied, keeping in mind the allocation and reading of memory, for example.
The aforementioned graphs, data and results are proof of how paramount it is to correctly apply these techniques.