Faculdade de Engenharia da Universidade do Porto

Computação Paralela e Distribuída

Project 1

Turma 1, Grupo 11

Autores:
André Gonçalves Pinto 202108856
Luis Vieira Relvas 202108661
Rodrigo Moisés Baptista Ribeiro 202108679

17 de março, 2024

First Part

1. Problem Description

This project consists of exploring the performance optimization of matrix multiplication algorithms, with a specific emphasis on the impact of memory hierarchy on processor efficiency.

In the first part we are asked to run the following algorithms in the chosen languages: Standard Matrix Multiplication, Matrix Line Multiplication and Matrix Block Multiplications. The performance metrics in C++ must be done by PAPI, which is a library for detailed CPU performance analysis. The Matrix Sizes: 600x600 to 3000x3000 (increments of 400x400) for all algorithms in both languages, 4096x4086 to 10240x10240 (increments of 2048x2048) for Matrix Line Multiplication in C++. The Block Sizes: 128, 256, 512 with matrix sizes from 4096x4096 to 10240x10240 (increments of 2048).

In the second part we should analyze the performance enhancement of the Matrix Line Multiplication algorithm through parallelization techniques, making use of the OpenMP library.

2. Algorithms

For this project, and taking into account memory manipulation, we implemented three different algorithms to solve matrices, aiming to measure the performance of a single core, taking advantage of memory allocation in different ways. For the first two methods, we developed, and tested, both in C++ and Java.

2.1. Simple Matrix Multiplication

This implementation utilizes a straightforward C++ algorithm for matrix multiplication. The core computation multiplies each row of the first matrix with every column of the second matrix.

- Time Complexity: $O(n^3)$;
- Space Complexity: $O(n^2)$;

2.2. Line Matrix Multiplication

This improved algorithm still operates within an $O(n^3)$ time complexity. However, the core calculation differs: instead of multiplying a single element by a matrix column, it multiplies an element of the first matrix by an entire row of the second matrix. This modification aims to potentially leverage better memory access patterns and cache utilization.

- Time Complexity: $O(n^3)$

- Space Complexity: $O(n^2)$

2.3 Block Matrix Multiplication

This algorithm introduces the concept of dividing matrices into smaller submatrices (blocks). It performs matrix multiplication on these blocks and combines the results. While the overall time complexity remains $O(n^3)$, this approach aims to improve memory locality and cache performance by working with smaller chunks of data at a time.

- Time Complexity: $O(n^3/b^3*b^3) = O(n^3)$
- Space Complexity: $O(n^2)$

3. Performance

To evaluate the performance of the algorithms for C++ we used the Performance API (PAPI) so that we can access various measures of the CPU Cache memory as well as the number Floating Points Operations (FLOP). With PAPI we can also access the number of L1 and L2 cache misses.

To ensure that the tests were performed in the same environment, we used the computers from the classroom where we had classes. These computers were running on Ubuntu 22.04 with an Intel Core i7-9700 @ 3.00GHz with 8 cores. Lastly , in C++ version of the code we used the -O2 optimization flag when compiling as suggested for when using PAPI.

In PAPI we used the counters :

- L1 Cache Misses;
- L2 Cache Misses;

4. Results and analysis

To rigorously compare the three algorithms, we measured and plotted execution time and MFLOPS for each algorithm across increasing matrix sizes as requested. To ensure accuracy, we performed three measurements for each data point and averaged the results. For the parallel algorithms we took execution times, MFLOPS, speedup and efficiency.

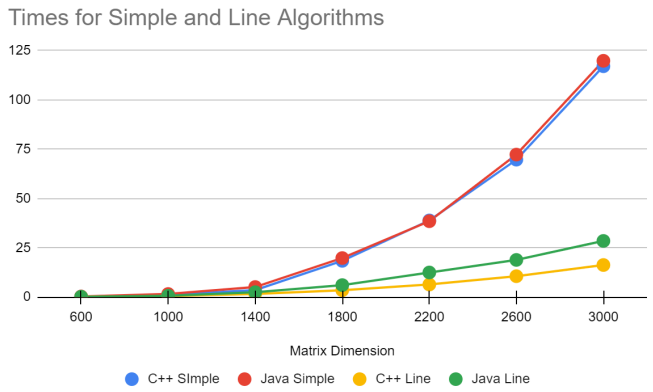
4.1. Execution time and cache misses between Line and Simple Matrix Multiplication

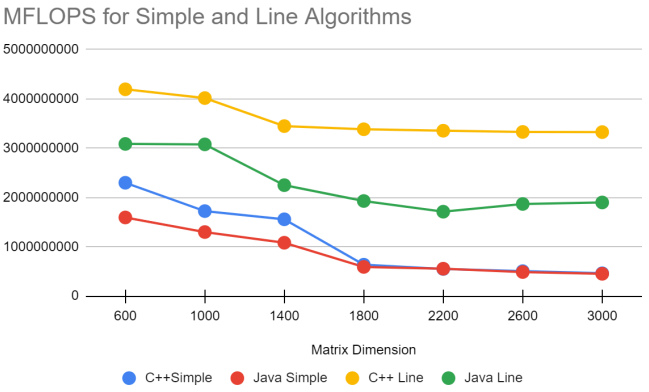
We compared Standard and Line Matrix Multiplication implementations in both C++ and Java. The execution times for both languages were similar, likely due to their compiled nature and the shared algorithmic logic.

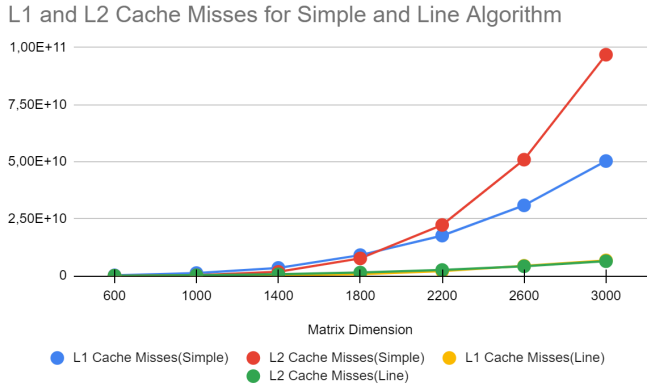
Importantly, the Line Matrix Multiplication algorithms significantly outperformed the Standard implementations. This performance improvement is attributed to reduced cache misses, by using the result matrix as an accumulator and

accessing elements in a line-by-line, the Line Algorithms ensure that frequently used data remains more readily available in faster levels of the memory hierarchy.

Analyzing the MFLOPS for both the Simple and Line Matrix Multiplication algorithms we can observe that the Line algorithm demonstrates a higher MFLOPS (floating-point operations per second) with agreement of the reduced number of L1 and L2 cache misses. Additionally, we conclude that the execution time of the Simple Matrix Multiplication algorithm increases significantly as the matrix size grows, likely due to increased cache pressure and the need to access memory to fetch data. In contrast, the Line Matrix Multiplication algorithm exhibits a relatively constant MFLOPS across matrix sizes, suggesting that its optimizations lead to better scaling behavior.

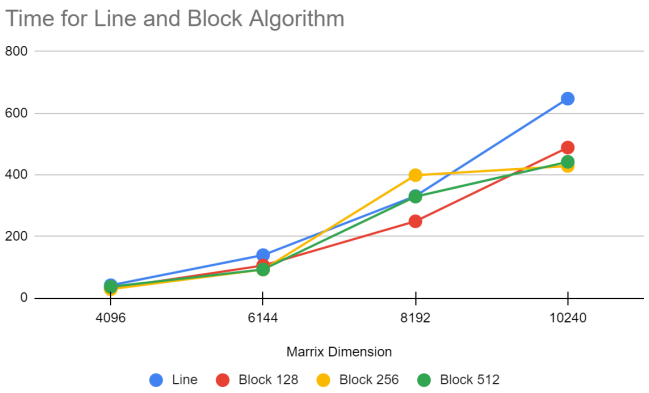


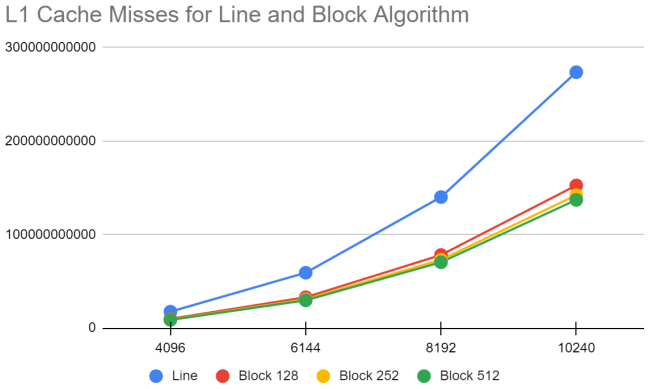


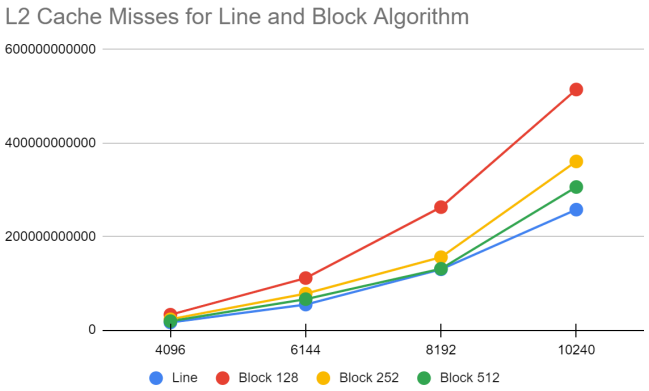


4.2. L1 and L2 Data cache misses and execution time between block sizes on a Block Matrix Multiplication Algorithm

Analysis of the Block Matrix Multiplication algorithm's time evolution plot reveals that an optimal block size exists for each matrix size. This optimum minimizes the execution time. As the block size increases, it approaches the matrix line used in the Line Multiplication Algorithms. This suggests that there's an ideal block size (smaller than the matrix size) that maximizes performance.







Second Part

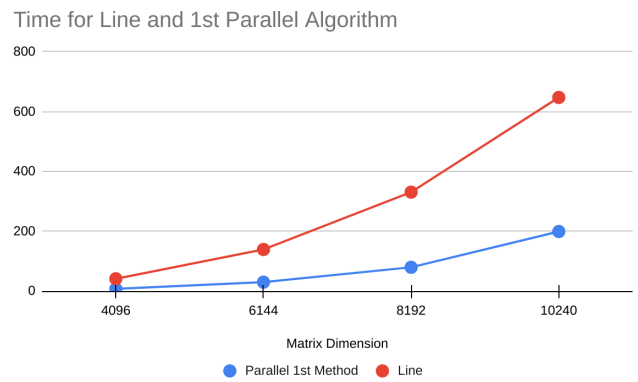
1. First Parallel Algorithm:

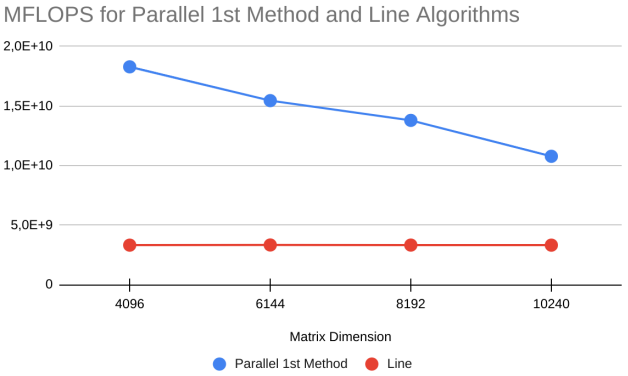
This first algorithm distributes the outermost loop across eight threads, one per available core. Each thread receives its own copies of k and j and independently executes the full inner loops. This approach effectively parallelizes the workload across the machine's cores.

1.1. Performance evaluation between Sequential Time and Parallelization Algorithm 1:

Our results demonstrate the clear performance advantage of parallelization over the sequential algorithm. As anticipated, the parallelized version distributes the computational workload across eight cores, leading to significant speed improvements. However, this speedup diminishes as matrix size increases. This slowdown likely stems from memory access conflicts, as multiple cores compete for the limited bandwidth of a sequential memory bus.

The parallelized version significantly reduced cache miss rates by optimizing cache utilization. Multiple threads working concurrently on distinct matrix portions allow for more efficient cache use. This minimizes cache misses and boosts overall performance compared to the sequential algorithm.





2. Second Parallel Algorithm

In the second parallelization approach, eight threads (one per core) execute the two outermost loops.

The inner loop, which involves multiplying a matrix value by a vector, is divided across threads. Each thread receives a subsection of the vector and performs calculations for its assigned range of “j” values. This effectively distributes the inner loop’s workload across multiple cores. At the end we will explain how this affects the whole execution.

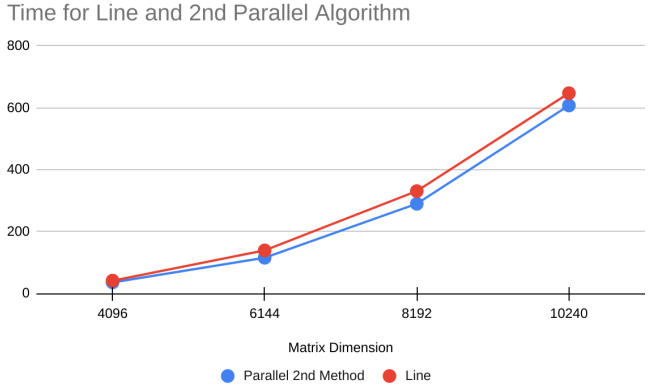
2.1. Performance evaluation between Sequential Time and Parallelization Algorithm 2:

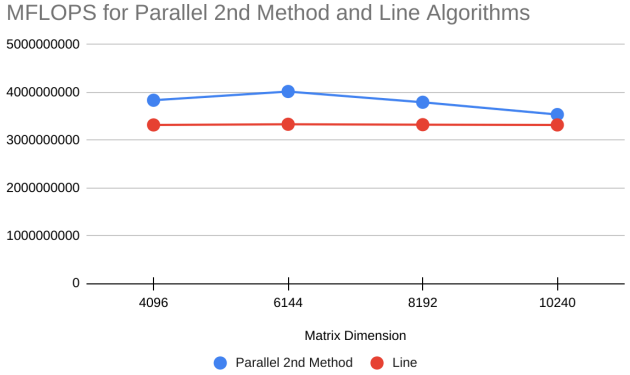
As observed, the second parallelization algorithm didn’t significantly improve performance over the sequential algorithm. This is likely due to two factors: Synchronization Overhead and Reduced Data Locality.

The Synchronization Overhead is due to threads must synchronize at each iteration of k introducing wait times, the Reduced Data Locality because the algorithm multiplies a single matrix value by a vector. This means each thread handles a smaller data portion, potentially worsening cache performance compared to the first parallel approach.

Consequently, the execution time improvement wasn’t proportional to the number of cores used, limiting speedup.

On the other hand, the algorithm still exhibits significantly lower cache misses than the sequential version, likely due to the same mechanisms described for the first parallel algorithm.

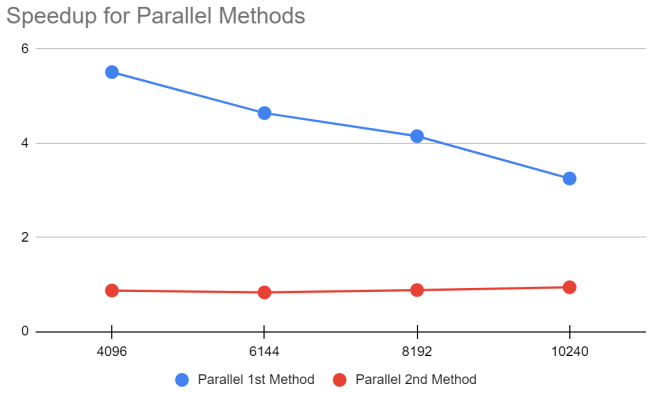


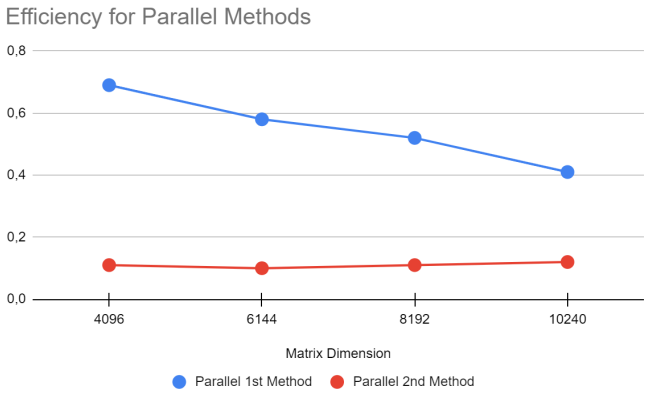


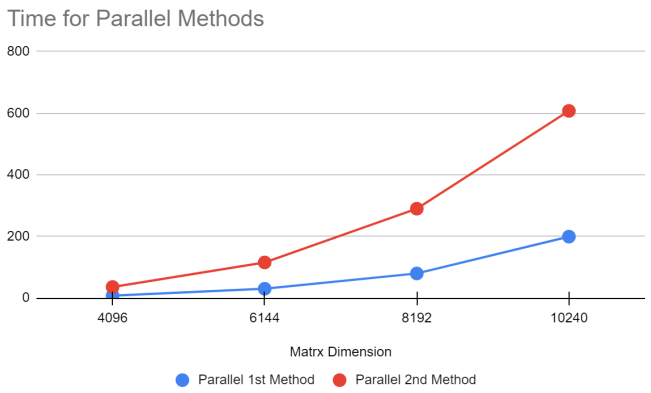
3. MFLOPS/Speedup/Efficiency comparison between parallel Algorithms

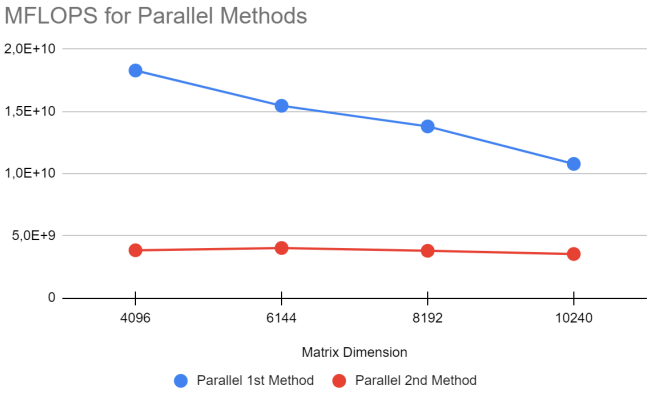
From the plots, it reveals that the first parallel algorithm method consistently outperforms the second one. Despite the second method's initially higher MFLOPS for smaller matrices, this advantage diminishes with increasing matrix size. The first method demonstrates superior execution time, speedup (sequential time / parallel time) and efficiency (speedup / number of cores).

The first method's efficiency stems from its reduced synchronization overhead. By parallelizing the outermost loop and assigning 25% of iterations to each thread (for example assuming 4 threads, the algorithm needs to synchronize after $N^3/4$), it minimizes synchronization points compared to the second method. In the second method, parallelizing the inner loop necessitates synchronization after every N iterations, increasing overhead.









5. Conclusion

Through this project, we developed practical memory management skills that significantly improve program speed. We learned that both thoughtful memory use in sequential programs and well-optimized memory handling in parallel algorithms are key to boosting performance.

6. Annexes

6.1. Folha de Cálculo

Folha de Cálculo

6.2. Código

6.2.1. Código Geral

Simple Matrix Multiplication:

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

Line Matrix Multiplication:

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

Block Matrix Multiplication:

```
for(x=0; x<m_ar;x+=bkSize){
        for(y=0;y<m_ar;y+=bkSize){
            for(z=0;z<m_ar;z+=bkSize){
                for(i=x;i< x+ bkSize; i++){
                    for(k = y ; k < y +bkSize;k++){
```

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

6.2.2. Código em c++

```
#include <stdio.h>
#include <iostream>
#include <iomanip>
#include <time.h>
#include <cstdlib>
#include <papi.h>

using namespace std;

#define SYSTEMTIME clock_t


void OnMult(int m_ar, int m_br)
{

    SYSTEMTIME Time1, Time2;

    char st[100];
    double temp;
    int i, j, k;

    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);



        Time1 = clock();

        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;
            }
        }


        Time2 = clock();
        sprintf(st, "Time: %3.3f seconds\n", (double)(Time2 - Time1) /
CLOCKS_PER_SEC);
        cout << st;

        // display 10 elements of the result matrix tto verify correctness
        cout << "Result matrix: " << endl;
        for(i=0; i<1; i++)
        {     for(j=0; j<min(10,m_br); j++)
                cout << phc[j] << " ";
        }
        cout << endl;

        free(pha);
        free(phb);
        free(phc);


    }

    // add code here for line x line matriz multiplication
    void OnMultLine(int m_ar, int m_br)
    {
        SYSTEMTIME Time1, Time2;

        char st[100];
        double temp;
        int i, j, k;

        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);



        Time1 = clock();

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


        Time2 = clock();
        sprintf(st, "Time: %3.3f seconds\n", (double)(Time2 - Time1) /
CLOCKS_PER_SEC);
        cout << st;

        // display 10 elements of the result matrix tto verify correctness
        cout << "Result matrix: " << endl;
        for(i=0; i<1; i++)
        {     for(j=0; j<min(10,m_br); j++)
              cout << phc[j] << " ";
        }
        cout << endl;

        free(pha);
        free(phb);
        free(phc);

     }
```

```
// add code here for block x block matriz multiplication
void OnMultBlock(int m_ar, int m_br, int bkSize)
{
   SYSTEMTIME Time1, Time2;

   char st[100];
   double temp;
   int i, j, k, x, y, z;

   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);



   Time1 = clock();

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

```
            }
        }



        Time2 = clock();
        sprintf(st, "Time: %3.3f seconds\n", (double)(Time2 - Time1) /
CLOCKS_PER_SEC);
        cout << st;

        // display 10 elements of the result matrix tto verify correctness
        cout << "Result matrix: " << endl;
        for(i=0; i<1; i++)
        {     for(j=0; j<min(10,m_br); j++)
                cout << phc[j] << " ";
        }
        cout << endl;

        free(pha);
        free(phb);
        free(phc);


    }



    void handle_error (int retval)
    {
        printf("PAPI error %d: %s\n", retval, PAPI_strerror(retval));
        exit(1);
    }

    void init_papi() {
        int retval = PAPI_library_init(PAPI_VER_CURRENT);
        if (retval != PAPI_VER_CURRENT && retval < 0) {
            printf("PAPI library version mismatch!\n");
            exit(1);
        }
        if (retval < 0) handle_error(retval);

        std::cout << "PAPI Version Number: MAJOR: " <<
PAPI_VERSION_MAJOR(retval)
                << " MINOR: " << PAPI_VERSION_MINOR(retval)
                << " REVISION: " << PAPI_VERSION_REVISION(retval) << "\n";
    }


    int main (int argc, char *argv[])
    {
```

```
char c;
int lin, col, blockSize;
int op;

int EventSet = PAPI_NULL;
long long values[2];
int ret;


ret = PAPI_library_init( PAPI_VER_CURRENT );
if ( ret != PAPI_VER_CURRENT )
    std::cout << "FAIL" << endl;


ret = PAPI_create_eventset(&EventSet);
if (ret != PAPI_OK) cout << "ERROR: create eventset" << endl;


ret = PAPI_add_event(EventSet,PAPI_L1_DCM );
if (ret != PAPI_OK) cout << "ERROR: PAPI_L1_DCM" << endl;


ret = PAPI_add_event(EventSet,PAPI_L2_DCM);
if (ret != PAPI_OK) cout << "ERROR: PAPI_L2_DCM" << endl;


op=1;
do {
    cout << endl << "1. Multiplication" << endl;
    cout << "2. Line Multiplication" << endl;
    cout << "3. Block Multiplication" << endl;
    cout << "Selection?: ";
    cin >>op;
    if (op == 0)
        break;
    printf("Dimensions: lins=cols ? ");
    cin >> lin;
    col = lin;


    // Start counting
    ret = PAPI_start(EventSet);
    if (ret != PAPI_OK) cout << "ERROR: Start PAPI" << endl;

    switch (op){
        case 1:
            OnMult(lin, col);
            break;
        case 2:
```

```
                OnMultLine(lin, col);
                break;
            case 3:
                cout << "Block Size? ";
                cin >> blockSize;
                OnMultBlock(lin, col, blockSize);
                break;

        }

        ret = PAPI_stop(EventSet, values);
        if (ret != PAPI_OK) cout << "ERROR: Stop PAPI" << endl;
        printf("L1 DCM: %lld \n",values[0]);
        printf("L2 DCM: %lld \n",values[1]);

        ret = PAPI_reset( EventSet );
        if ( ret != PAPI_OK )
            std::cout << "FAIL reset" << endl;



    }while (op != 0);

    ret = PAPI_remove_event( EventSet, PAPI_L1_DCM );
    if ( ret != PAPI_OK )
        std::cout << "FAIL remove event" << endl;

    ret = PAPI_remove_event( EventSet, PAPI_L2_DCM );
    if ( ret != PAPI_OK )
        std::cout << "FAIL remove event" << endl;

    ret = PAPI_destroy_eventset( &EventSet );
    if ( ret != PAPI_OK )
        std::cout << "FAIL destroy" << endl;

}
```

6.2.3. Código em Java

```
import java.util.Scanner;
public class Main {
    public static double OnMult(int m_ar, int m_br) {

        double[] pha = new double[m_ar * m_ar];
        double[] phb = new double[m_ar * m_ar];
        double[] phc = new double[m_ar * m_ar];
```

```
    for (int a = 0; a < m_br; a++) {
        for (int b = 0; b < m_br; b++) {
            phb[a * m_br + b] = a + 1;
        }
    }

    long Time1 = System.currentTimeMillis();

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

    long Time2 = System.currentTimeMillis();

    double Total_time = (Time2 - Time1) / 1000.0;
    System.out.printf("Time: %.3f seconds%n%n", Total_time);
    System.out.println("Result Matrix: ");
    for (int c = 0; c < Math.min(10, m_br); c++) {
        System.out.printf("%.2f ", phc[c]);
    }
    return Total_time;
}

public static double OnMultLine(int m_ar, int m_br) {
    double[] pha = new double[m_ar * m_ar];
    double[] phb = new double[m_ar * m_ar];
    double[] phc = new double[m_ar * m_ar];

    for (int a = 0; a < m_br; a++) {
        for (int b = 0; b < m_br; b++) {
            phb[a * m_br + b] = a + 1;
        }
    }

    long Time1 = System.currentTimeMillis();

    for (int i = 0; i < m_ar; i++) {
        for (int k = 0; k < m_br; k++) {
```

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

    long Time2 = System.currentTimeMillis();

    double Total_time = (Time2 - Time1) / 1000.0;

    System.out.printf("Time: %.3f seconds%n%n", Total_time);
    System.out.println("Result Matrix: ");
    for (int c = 0; c < Math.min(10, m_br); c++) {
        System.out.printf("%.2f ", phc[c]);
    }
    return Total_time;
}

public static double OnMultBlock(int m_ar, int m_br, int bkSize) {
    double[] pha = new double[m_ar * m_ar];
    double[] phb = new double[m_ar * m_ar];
    double[] phc = new double[m_ar * m_ar];

    for (int a = 0; a < m_br; a++) {
        for (int b = 0; b < m_br; b++) {
            pha[a * m_br + b] = 1;
            phb[a * m_br + b] = a + 1;
            phc[a * m_br + b] = 0;
        }
    }

    int i, ii, j, jj, k, kk;
    long Time1 = System.currentTimeMillis();

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

```
            }
        }
    }

    long Time2 = System.currentTimeMillis();

    double Total_time = (Time2 - Time1) / 1000.0;

    System.out.printf("Time: %.3f seconds%n%n", Total_time);
    System.out.println("Result Matrix: ");
    for (int c = 0; c < Math.min(10, m_br); c++) {
        System.out.printf("%.2f ", phc[c]);
    }
    return Total_time;
  }
  public static void main(String[] args) {

    if (args.length != 2) return;

    int dim = Integer.parseInt(args[0]);
    int bkSize = Integer.parseInt(args[1]);

    double total_time = 0;

    for(int i = 0; i < 3; i++)
        total_time += OnMult(dim, dim);
    total_time /= 3;
    System.out.printf("AVG Time: %.3f seconds%n%n", total_time);

    total_time = 0;
    for(int i = 0; i < 3; i++)
        total_time += OnMultLine(dim, dim);
    total_time /= 3;
    System.out.printf("AVG Time: %.3f seconds%n%n", total_time);

    total_time = 0;

    for(int i = 0; i < 3; i++)
        total_time += OnMultBlock(dim, dim, bkSize);
    total_time /= 3;
    System.out.printf("AVG Time: %.3f seconds%n%n", total_time);
  }
}
```