U.PORTO FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

U.PORTO FC FACULDADE DE CIÊNCIAS UNIVERSIDADE DO PORTO

Parallel and Distributed Computing

Assignment 1

Authors:

João António Teixeira Coelho up202004846

João Guimarães Mota up202108677

Pedro Jorge Mendes Jesus Landolt up202103337

Bachelor in Informatics and Computing Engineering
March 2024

Contents

1 Problem description and algorithms explanation 1

2 Performance metrics 2

3 Results and analysis 2

4 Conclusions 6

Chapter 1 - Problem description and algorithms explanation

This assignment was divided into two parts: the performance evaluation of a single core; and the performance evaluation of a multi-core implementation.

We will first address the first part.

We were given a file, **matrixproduct.cpp**, that contained ***C++*** code that multiplied square matrices of **n*n** dimensions, where n is an input given by the user. This algorithm works by multiplying a line of the first matrix by each column of the second matrix. Before that, it allocates memory to create the two matrices it will multiply and the matrix that will be the result of the multiplication. That process is timed and used to determine efficiency, using ***PAPI***. After creating the square matrices and multiplying them, 10 elements of the resulting matrix are printed to ensure that the correct result was attained, and the memory is freed.

Our first task was to implement this exact algorithm in a programming language of your choice. We chose ***Java***. Since there is no dynamic memory allocation in Java, we opted to represent the matrices as arrays of arrays, which led to differences in accessing the matrix indexes when multiplying. However, the result was the same for both.

The second task consisted of implementing a different matrix-multiplication algorithm that multiplied an element from the first matrix by the correspondent line of the second matrix, in both languages. In this algorithm, the outer loop (i) still iterates over the rows of the first matrix (pha), but the middle loop (k) now iterates over a different dimension, and the innermost loop (j) iterates over the columns of the second matrix (phb). The differences between the C++ and Java implementations are like the ones from the previous task.

For the last task of part 1, we were asked to implement a Block Matrix Multiplication algorithm. This algorithm divides the square matrix into 2x2 matrices and multiplies them by each other. Our implementation breaks the matrices down in blocks of ***bkSize***, and then the inner loops iterate, respectively, over the rows of the result matrix, and the columns within the current block. The inner-most loop performs multiplication. Our implementation of this algorithm was based on our implementation of the line algorithm, as it not only divides the matrix in blocks, but also performs line multiplication.

These algorithms have different performance characteristics. This can be seen in chapter 3. In general, the line algorithm is more efficient than the first algorithm, because the values are closer to each other, which better utilizes the cache memory. Another reason is that the naïve algorithm needs to perform bigger leaps in memory to multiply row by column, while the line multiplication algorithm reads the contiguous memory at the start of each matrix and progresses uniformly for both matrices without ever going backwards. A problem arises when the dimension of the matrix is too big, as the lines are too extensive, and every value cannot be stored. That's when the block matrix becomes preferrable, since dividing the matrix into smaller blocks resolves that memory issue. For the line and block multiplication algorithms, we also initialized the result matrix, which allowed us to save memory (no need to allocate it as the loops occur) and unnecessary instructions that could have possibly resulted in more cache misses.

Part 2 consisted of the implementation and performance analysis of two parallel versions of the line multiplication algorithm. For that, we were given two solutions: one that employed ***#pragma omp parallel for*** before the first for loop; and one that employed ***#pragma omp parallel*** before the first loop and ***#pragma omp for*** before the last loop. The first approach parallelizes the outermost loop, creating threads based on the available processor cores and distributing the loop iterations among them. The second solution, however, creates threads before the first loop according to the number of available cores in the processor and parallelizes the innermost loop, distributing its iterations across previously created threads. This solution performs the two outermost loops on all threads.

Chapter 2 - Performance metrics

All measurements were taken more than once, to ensure that we obtained the best possible data for our analysis. The variables between each experiment are the size of the matrix, the algorithm chosen to perform multiplication, and in the case of block multiplication, the size of the block.

In the Java implementations, we only measured the time it took for the algorithms to execute.

In the C++ implementations, we used the Performance API (***PAPI***) to obtain the values of counters regarding ***data cache misses (DCM)***, ***instruction cache hits (ICH)***, ***instruction cache misses (ICM)***, ***double precision floating point operations per second (DP OPS)***, ***total instructions (TOT INS)***, and ***total cache misses (TCM)***, for the 3 different layers of the processor's cache (**L1, L2, L3**). The ***TOT INS*** and ***DP OPS*** measurements allowed us to calculate the number of ***MIPS*** and ***MFLOPs***, respectively. Lastly, the ***speedup*** and ***efficiency*** for part 2 were calculated, respectively, by dividing the time of the execution of the sequential algorithm by the time of execution of the parallel algorithm; and by dividing the speedup by the number of threads of the processor: in this case, 8.

Chapter 3 – Results and analysis

We ran tests according to the specifications of the statement. The results were the following:

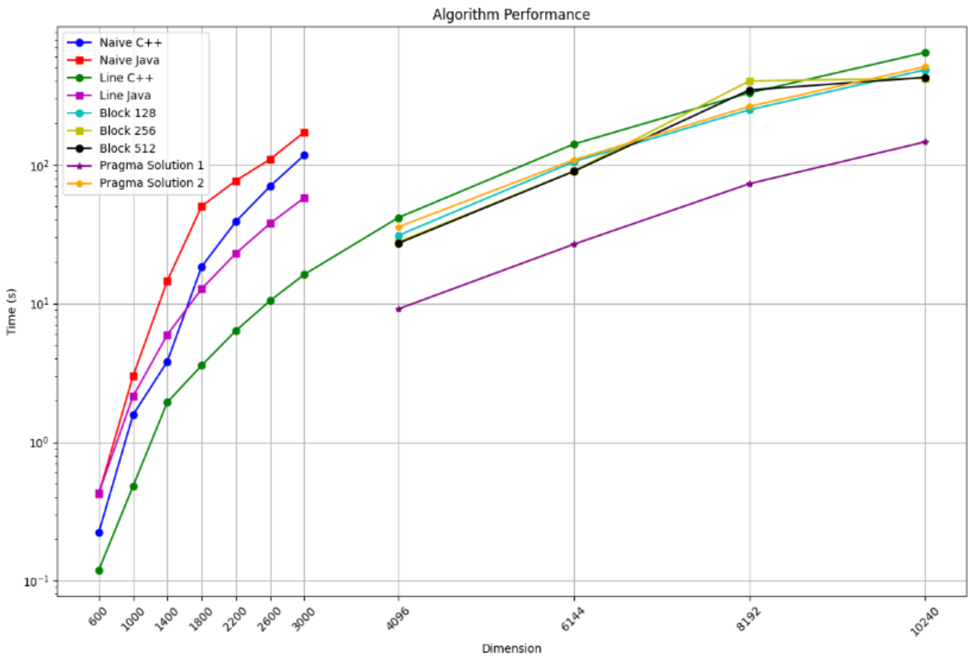
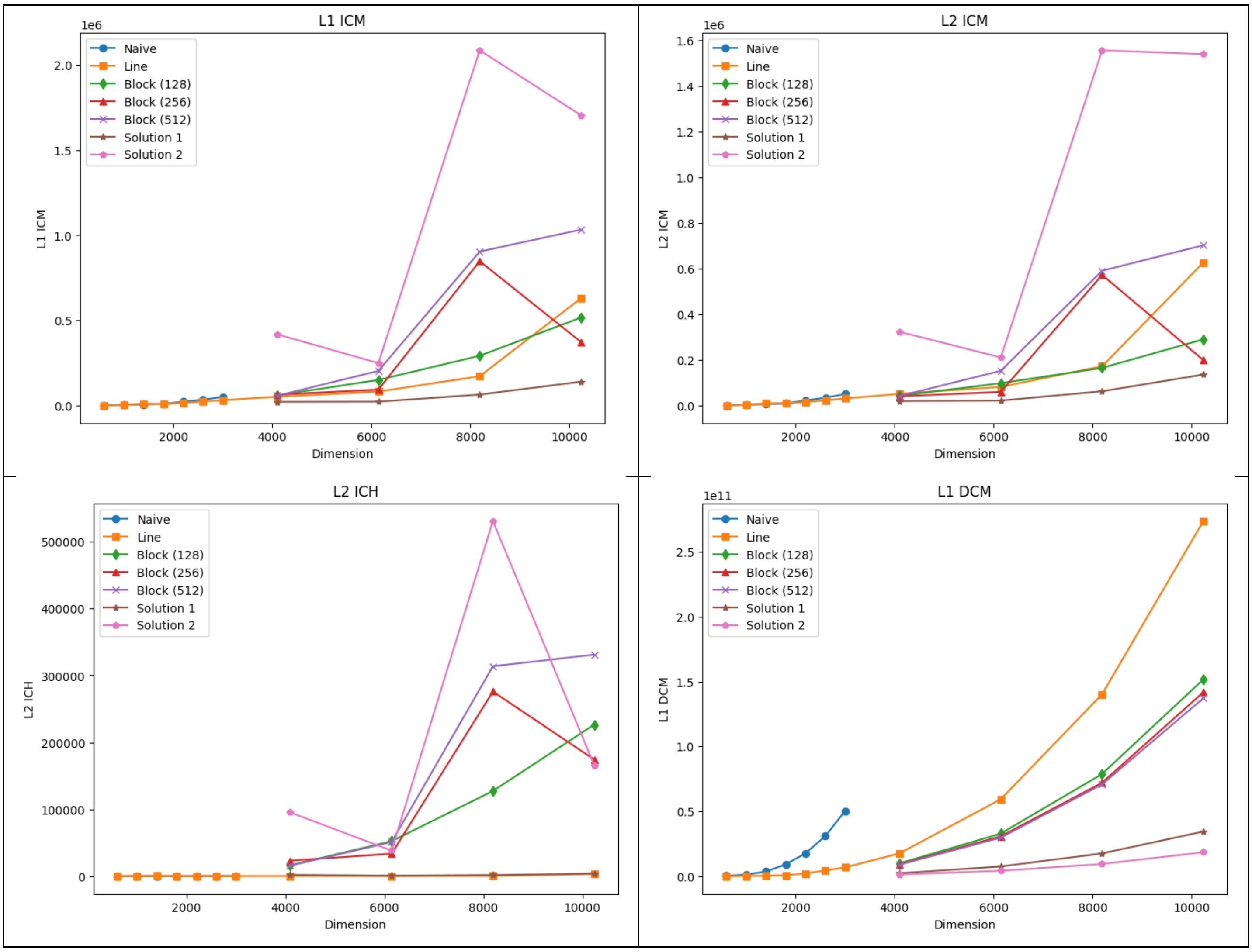
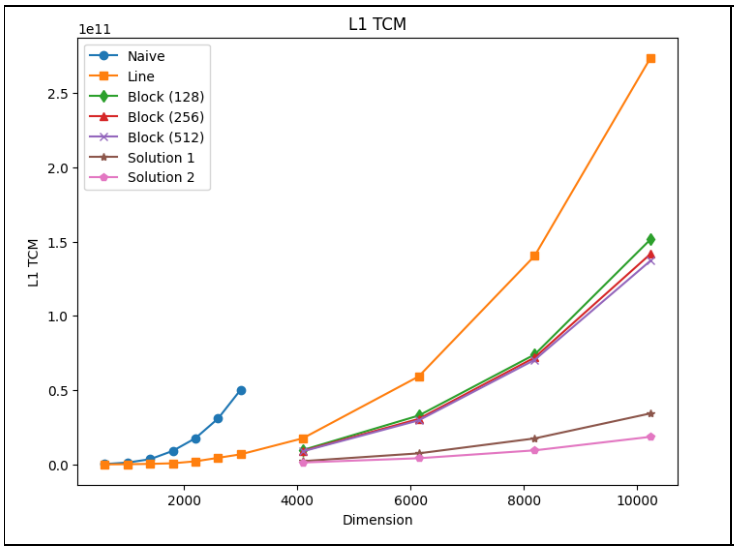
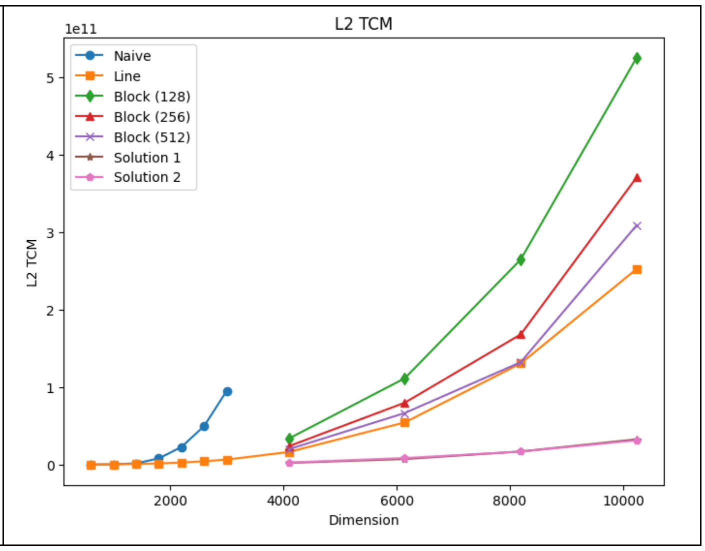


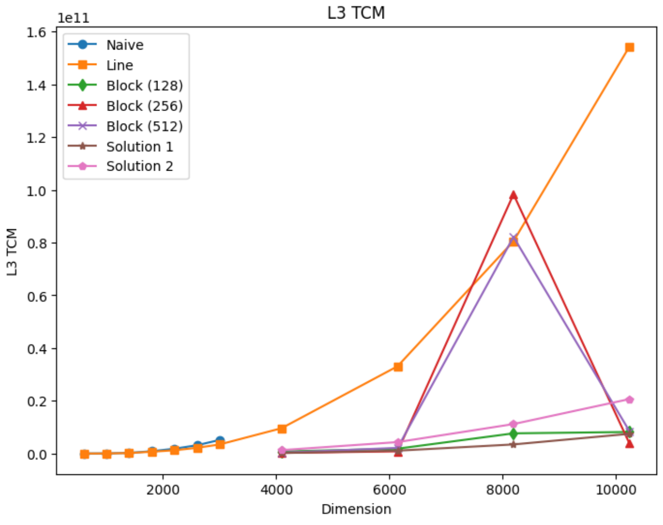
Figure 1 – Time comparison between different algorithms

As visible in the graph, the C++ algorithms are faster than the corresponding java algorithms. However, the difference between the line algorithms is bigger than the one between the naïve ones. This stems from the memory-management capabilities of C++ - a language that distinguishes itself for being faster and lighter, when memory is correctly allocated and used. Other noticeable data: the block-matrix algorithm is faster than the line algorithm, for the test cases, and the speed is higher if the block-size is bigger.



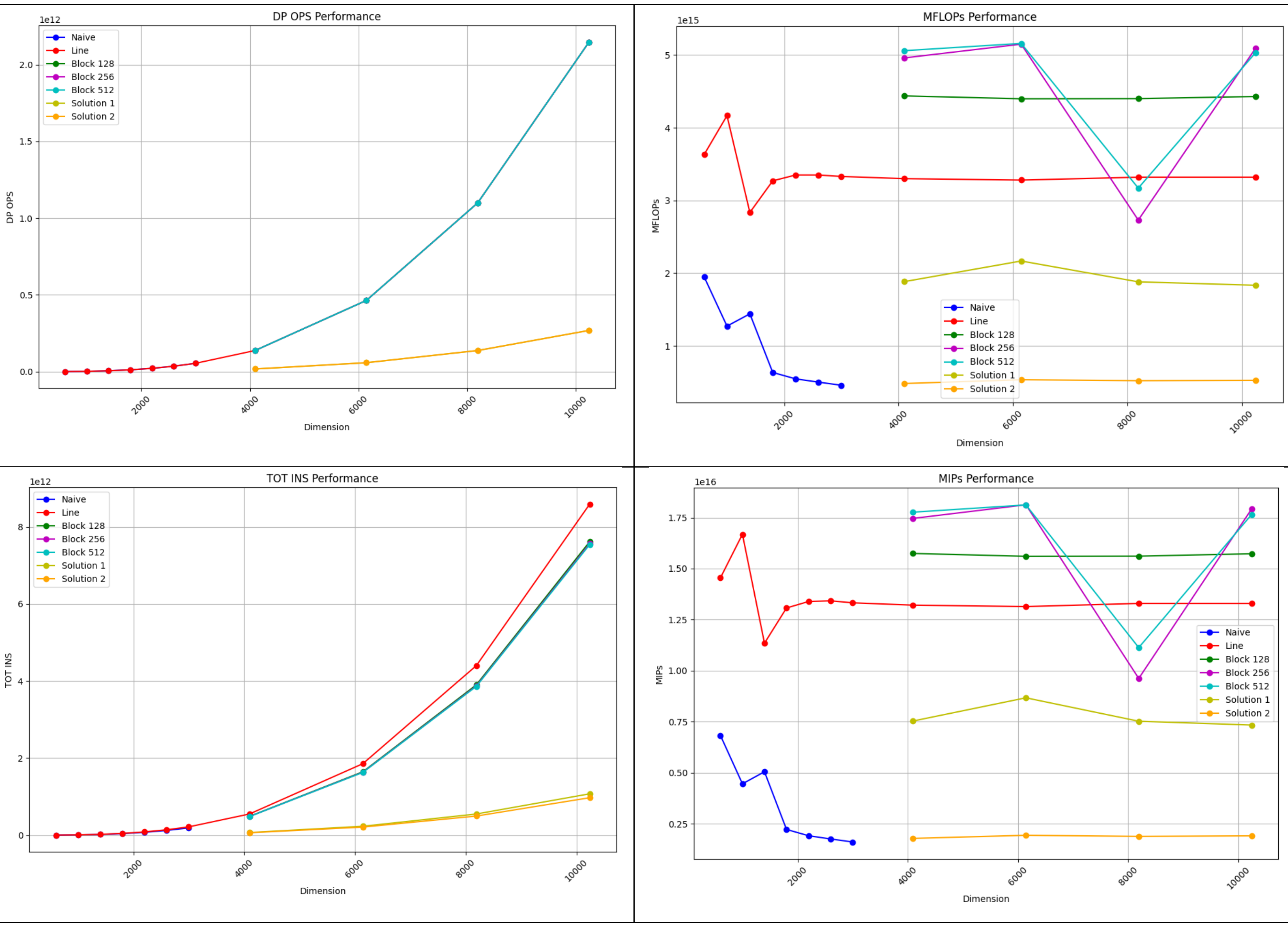






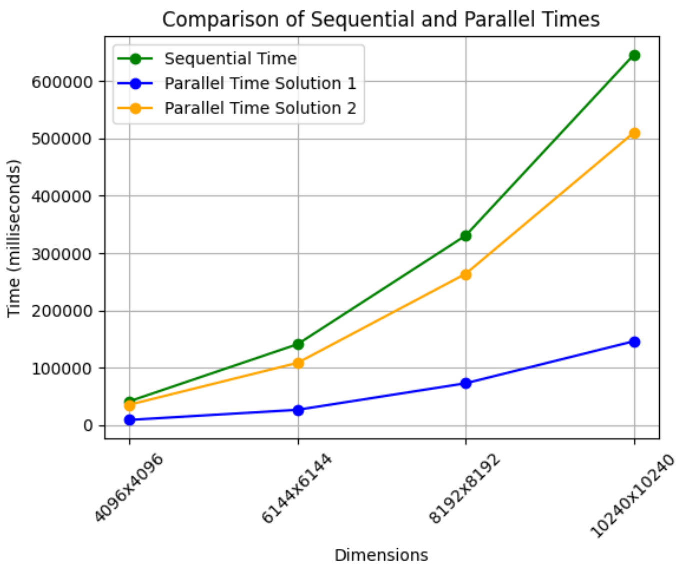
Figures [2,8] – Processor cache counter measures

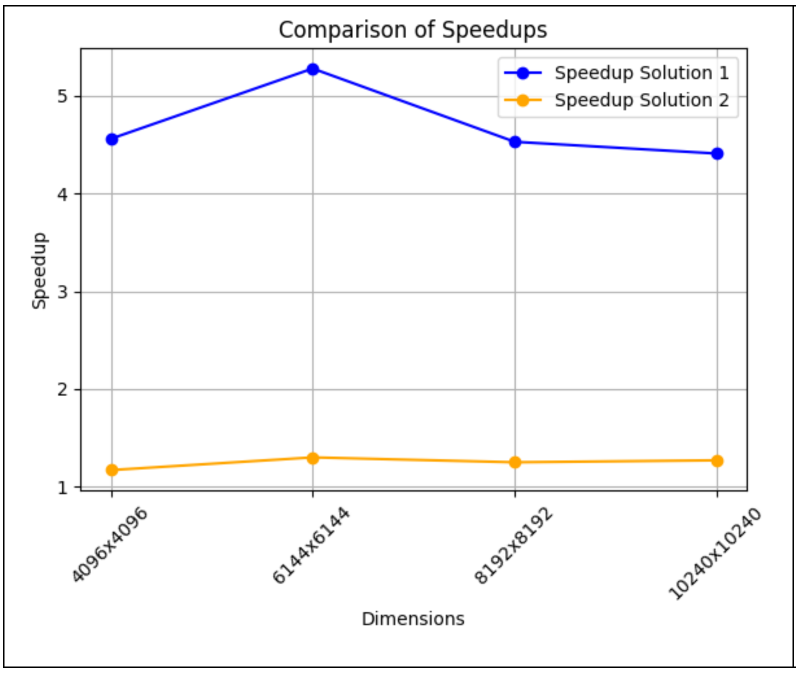
These graphs translate the information about cache hits and misses (***T***otal, ***D***ata, ***I***nstructions). In general, both cache hits and misses increase alongside the dimension of the matrices. Considering that lower cache misses and higher cache hits indicate better cache utilization, which can contribute to better performance, the block-matrix algorithms are generally more efficient than the line or naïve algorithms. Concerning the pragma solutions, solution 1 tends to have fewer cache misses than solution 2 for the same levels of cache. Since cache misses lead to the access of the next level of cache, it is expected that solution 2 has more cache hits for the next layer (L2 ICH graph). This is because solution 1 parallelizes the outermost loop, leading to better cache utilization as threads work on contiguous blocks of data.

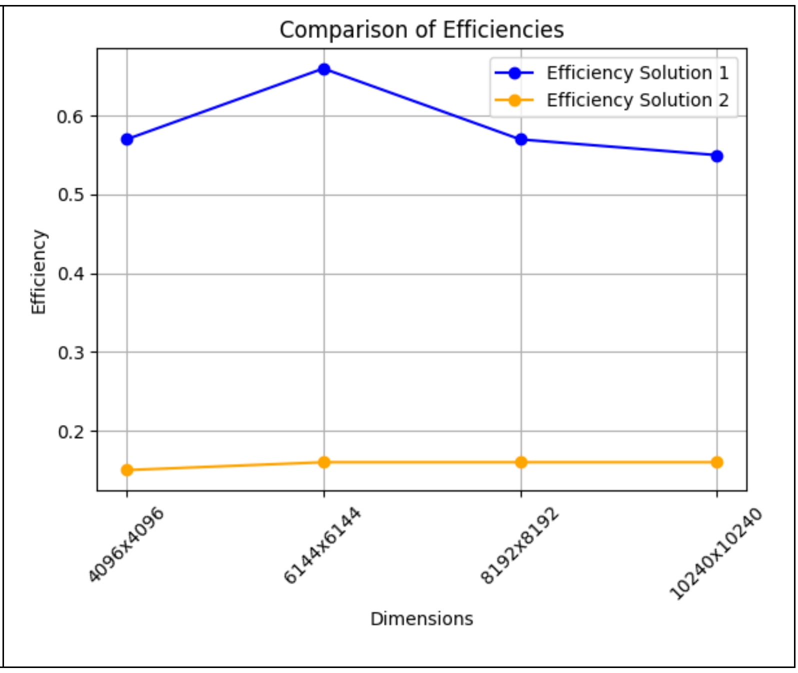


Figures [9, 12] – Instruction count measures

These graphs translate the difference between the number of instructions and operations of each algorithm. The general rule is that the faster the algorithm is, the more operations per second it can realize. The line and block algorithms increase their total instruction count, but maintain their MIPs values in a similar threshold, independent of the size of the matrix, while the naïve algorithm sees a drop in performance as the size of the matrix increases, due to reasons already mentioned. The same logic applies to DPOPS and MFLOPs. Regarding the pragma solutions, solution 1 stands out yet again.







Figures [13, 15] – pragma speedup and efficiency

These graphs translate the difference between both parallel solutions. Solution 1 is faster and more efficient. We can conclude that parallelizing the outermost loop and attributing iterations to those threads is, in terms of performance, superior to parallelizing the innermost loop.

Chapter 4 - Conclusions

This assignment allowed us to realize how important it really is to correctly manage and use the CPU's parallel processing capabilities when writing programs. Correct usage of said capabilities allows for scalability (bigger inputs/size of data) and better performance, as the resources of the computer are used in a more distributed way, promoting resource efficiency by minimizing idle time and maximizing CPU utilization.

References

• Block Matrix Multiplication slides from moodle;

Appendix – Source code

C++:

Setting up:

```
#include <stdio.h>
#include <iostream>
```

```cpp
#include <iomanip>
#include <time.h>
#include <cstdlib>
#include <papi.h>
#include <omp.h>
#include <chrono>

using namespace std;

#define SYSTEMTIME clock_t

// Command to run papi -> g++ -O2 filename.cpp -o filename -lpapi

using Clock = std::chrono::high_resolution_clock;
using TimePoint = std::chrono::time_point<Clock>;

TimePoint Now(){
    return Clock::now();
}

double DurationInSeconds(const TimePoint& start, const TimePoint& end){
    return std::chrono::duration<double>(end - start).count();
}

long long values[3];
int EventSet = PAPI_NULL;
int ret;

// Available papi inst ->
void PappiStart()
{

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

    ret = PAPI_add_event(EventSet, PAPI_L2_TCM);
    if (ret != PAPI_OK) cout << "ERROR: create eventset" << endl;

    ret = PAPI_add_event(EventSet, PAPI_L3_TCM);
    if (ret != PAPI_OK) cout << "ERROR: create eventset" << endl;
    // ret = PAPI_add_event(EventSet,PAPI_L1_ICM );
    // if (ret != PAPI_OK) cout << "ERROR: PAPI_L1_ICM" << endl;

    // ret = PAPI_add_event(EventSet,PAPI_L2_ICM);
    // if (ret != PAPI_OK) cout << "ERROR: PAPI_L2_ICM" << endl;

    // ret = PAPI_add_event(EventSet,PAPI_L2_ICH);
    // if (ret != PAPI_OK) cout << "ERROR: PAPI_L2_ICH" << endl;

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

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

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

}

void PappiEnd(){
    ret = PAPI_stop(EventSet, values);
    if (ret != PAPI_OK) cout << "ERROR: Stop PAPI" << endl;
    printf("L1 TCM: %lld \n",values[0]);
    printf("L1 TCM: %lld \n",values[1]);
    printf("L1 TCM: %lld \n",values[2]);
    // printf("L1 ICM: %lld \n",values[0]);
    // printf("L2 ICM: %lld \n",values[1]);
    // printf("L2 ICH: %lld \n",values[2]);
    // printf("DP OPS: %lld \n",values[3]);
    // printf("TOT INS: %lld \n",values[4]);
    // printf("L1 DCM: %lld \n",values[5]);
    // printf("L2 DCM: %lld \n",values[6]);

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

    ret = PAPI_remove_event(EventSet, PAPI_L1_TCM);
    if ( ret != PAPI_OK )
        std::cout << "FAIL remove event" << endl;
    ret = PAPI_remove_event(EventSet, PAPI_L2_TCM);
    if ( ret != PAPI_OK )
        std::cout << "FAIL remove event" << endl;

    ret = PAPI_remove_event(EventSet, PAPI_L3_TCM);
    if ( ret != PAPI_OK )
        std::cout << "FAIL remove event" << endl;

    // ret = PAPI_remove_event( EventSet, PAPI_L1_ICM );
    // if ( ret != PAPI_OK )
    //   std::cout << "FAIL remove event" << endl;

    // ret = PAPI_remove_event( EventSet, PAPI_L2_ICM );
    // if ( ret != PAPI_OK )
    //   std::cout << "FAIL remove event" << endl;

    // ret = PAPI_remove_event( EventSet, PAPI_L2_ICH );
    // if ( ret != PAPI_OK )
    //   std::cout << "FAIL remove event" << endl;

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

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

    // 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 eventset" << endl;
    }

}
```

Naïve algorithm:

```
void OnMult(int m_ar, int m_br)
{
    char st[100];
    double temp;

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



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



    TimePoint startTime = Now();
    PappiStart();
    for(int i=0; i<m_ar; i++)
    {   for(int j=0; j<m_br; j++)
        {   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;
        }
    }

    PappiEnd();
    TimePoint endTime = Now();

    double duration = DurationInSeconds(startTime, endTime);

    sprintf(st, "Time: %3.3f seconds\n", duration);
    cout << st;

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

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


}
```

Line Multiplication:

```
// add code here for line x line matriz multiplication
void OnMultLine(int m_ar, int m_br)
{
    char st[100];
    double temp;

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



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

    for(int i=0; i<m_br; i++)
        for(int j=0; j<m_br; j++)
            phc[i*m_br + j] = (double)0.0;



    TimePoint startTime = Now();
    PappiStart();
    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_br+j];
    PappiEnd();
    TimePoint endTime = Now();
    double duration = DurationInSeconds(startTime, endTime);
    sprintf(st, "Time: %3.3f seconds\n", duration);
    cout << st;

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

    free(pha);
    free(phb);
    free(phc);
}
```

Pragma solution 1:

```
// add code here for line x line matriz multiplication with parallel for
void OnMultLineParFor(int m_ar, int m_br)
{
    char st[100];
    double temp;

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



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

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

```
        for(int j=0; j<m_br; j++)
            phc[i*m_br + j] = (double)0.0;



    TimePoint startTime = Now();
    PappiStart();
    #pragma omp parallel for
    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_br+j];

    PappiEnd();
    TimePoint endTime = Now();
    double duration = DurationInSeconds(startTime, endTime);
    sprintf(st, "Time: %3.3f seconds\n", duration);
    cout << st;

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

    free(pha);
    free(phb);
    free(phc);
}
```

Pragma solution 2:

```
// add code here for line x line matriz multiplication parrallel for j
void OnMultLineParForJ(int m_ar, int m_br)
{
    char st[100];
    double temp;

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



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

    for(int i=0; i<m_br; i++)
        for(int j=0; j<m_br; j++)
            phc[i*m_br + j] = (double)0.0;


    TimePoint startTime = Now();
    PappiStart();
    #pragma omp parallel
    for(int i=0; i<m_ar; i++)
        for(int k=0; k<m_ar; k++)
            #pragma omp for
            for(int j=0; j<m_ar; j++)
                phc[i*m_ar+j] += pha[i*m_ar+k] * phb[k*m_br+j];

    PappiEnd();
    TimePoint endTime = Now();
    double duration = DurationInSeconds(startTime, endTime);
    sprintf(st, "Time: %3.3f seconds\n", duration);
    cout << st;

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

    free(pha);
    free(phb);
    free(phc);
}
```

Block-matrix multiplication:

```
// add code here for block x block matriz multiplication
void OnMultBlock(int m_ar, int m_br, int bkSize)
```

```
{
    char st[100];
    double temp;

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



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

    for(int i=0; i<m_br; i++)
        for(int j=0; j<m_br; j++)
            phc[i*m_br + j] = (double)0.0;

    TimePoint startTime = Now();
    PappiStart();
    for(int jj = 0; jj < m_ar; jj += bkSize)
        for(int kk = 0; kk < m_ar; kk += bkSize)
            for(int i = 0; i < m_ar; i++)
                for(int k = kk; k < min(kk + bkSize, m_ar); k++)
                    for(int j = jj; j < min(jj + bkSize, m_br); j++)
                        phc[i*m_br+j] += pha[i*m_ar+k] * phb[k*m_br+j];

    PappiEnd();
    TimePoint endTime = Now();
    double duration = DurationInSeconds(startTime, endTime);
    sprintf(st, "Time: %3.3f seconds\n", duration);
    cout << st;

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

}

/*

    Flops, papisstart - papiend (ver configs no avaliable), tempos,  Cronos
para o paralelismo

*/
```

Tests, error handling and papi initialization:

```

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[])
{
    freopen("test.txt","w",stdout);

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


    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

        switch (op){
            case 1:
                for(int i = 600; i <= 3000; i+=400){
                    cout << endl;
                    cout << "Dimension: " << i << " x " << i << endl;
                    OnMult(i, i);
                    cout << endl;
                }
                break;
            case 2:
                for(int i = 600; i <= 3000; i+=400){
                    cout << endl;
                    cout << "Dimension: " << i << " x " << i << endl;
                    OnMultLine(i, i);
                    cout << endl;
                }
                // for (int i = 4096; i <= 10240; i+=2048){
                //  cout << endl;
                //  cout << "Dimension: " << i << " x " << i << endl;
                //  OnMultLine(i, i);
                //  cout << endl;
                // }
                // for (int i = 4096; i <= 10240; i+=2048){
                //  cout << endl;
                //  cout << "Dimension: " << i << " x " << i << endl;
                //  OnMultLineParFor(i, i);
                //  cout << endl;
                // }
                // for (int i = 4096; i <= 10240; i+=2048){
                //  cout << endl;
                //  cout << "Dimension: " << i << " x " << i << endl;
                //  OnMultLineParForJ(i, i);
                //  cout << endl;
                // }
                break;
            case 3:
                for(int i = 4096; i <= 10240; i+=2048){
                    cout << endl;
                    int blockSize = 512; // 128, 256, 512
                    cout << "Dimension: " << i << " x " << i << endl;
                    cout << "Block size: " << blockSize << endl;
                    OnMultBlock(i, i, blockSize);
                    cout << endl;
                }
                break;

        }

    }while (op != 0);

}
```

Java:

Menu:

```
import java.util.Scanner;

public class matrixproduct_java {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("1. Multiplication");
        System.out.println("2. Line Multiplication");
        System.out.println("Selection?: ");

        String selection = scanner.nextLine();

        System.out.println("Dimensions: lins=cols ?");

        String userInput = scanner.nextLine();

        try {
            int intValue = Integer.parseInt(userInput);
            if (selection.equals("1"))
                OnMult(intValue, intValue);
            else if (selection.equals("2"))
                OnMultLine(intValue, intValue);
            else
                System.out.println("Invalid selection.");
        } catch (NumberFormatException e) {
            System.out.println("Invalid input. Please enter a valid
integer.");
        }
        finally {
            scanner.close();
        }
    }
```

Naïve algorithm:

```
    public static void OnMult(int a, int b) {
        double temp;
        int i, j, k;

        double[][] pha = new double[a][a];
        double[][] phb = new double[a][a];
        double[][] phc = new double[a][b];

        for(i=0; i<a; i++){
            for(j=0; j<a; j++){
                pha[i][j] = 1.0;
            }
        }

        for (i=0; i<a; i++){
            for(j=0; j<a; j++){
                phb[i][j] = i + 1.0;
            }
        }

        long startTime = System.nanoTime();

        for(i=0; i<a; i++){
            for(j=0; j<b; j++){
                temp = 0.0;
                for(k=0; k<a; k++){
                    temp += pha[i][k] * phb[k][j];
                }
                phc[i][j] = temp;
            }
        }
        long endTime = System.nanoTime();
        long duration = (endTime - startTime);

        System.out.println("\nResult: ");
        for(i=0; i<1; i++){
            for(j=0; j<Math.min(10, b); j++){
                System.out.print(phc[i][j] + " ");
            }
        }
        System.out.println("\n\nTime: " + String.format("%.3f",
duration/1_000_000_000.0) + " second(s)");
```

```
    }
```

Line algorithm

```
    public static void OnMultLine(int a, int b){

        int i, j, k;

        double[][] pha = new double[a][a];
        double[][] phb = new double[a][a];
        double[][] phc = new double[a][b];

        for(i=0; i<a; i++){
            for(j=0; j<a; j++){
                pha[i][j] = 1.0;
            }
        }

        for (i=0; i<a; i++){
            for(j=0; j<a; j++){
                phb[i][j] = i + 1.0;
            }
        }

        long startTime = System.nanoTime();

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

        long endTime = System.nanoTime();

        long duration = (endTime - startTime);

        System.out.println("\nResult: ");
        for(i=0; i<1; i++){
            for(j=0; j<Math.min(10, b); j++){
                System.out.print(phc[i][j] + " ");
            }
        }
        System.out.println("\n\nTime: " + String.format("%.3f", 
duration/1_000_000_000.0) + " second(s)");
    }
}
```