Faculdade de Engenharia da Universidade do Porto

Computação Paralela e Distribuída

CPD Project 1

Computação Paralela e Distribuída

3.º Ano - Licenciatura em Engenharia Informática e Computação
2023/2024

Turma 13 - Grupo 08

Estudantes & Autores

João Miguel Vieira Cardoso (up202108732@up.pt)
João Miguel da Silva Lourenço (up202108863@up.pt)
Tiago Ribeiro de Sá Cruz (up202108810@up.pt)

Index

1. Problem Description

This project aims to analyze the matrix multiplication algorithm, its performance and results.

We will take into account only square matrices and we will also parallelize one of the algorithms in an effort to compare it with the previous sequential program and analyze its possible improvement.

2. Algorithm Explanation

2.1 Normal Matrix Multiplication

This is the default matrix multiplication algorithm. In this project we will only analyze squared matrices and they both need to have the same dimensions. We iterate through the elements of each row of the first matrix and we multiply with the corresponding element of the columns of the second matrix and we store the results in a resulting matrix. The formulation is as follows:

$$(\mathbf{AB})_{ij} = \sum_{k=1}^{m} A_{ik} B_{kj}$$

Where A is the first matrix, B is the second, AB is the resulting matrix, m is the size of the matrix, i to define the rows of the first matrix, j to define the columns of the second matrix and k to define the columns of the first matrix and rows of the second matrix.

2.2 Line Matrix Multiplication

In this algorithm, instead of, in each multiplication, having to iterate through the elements of each column of the second matrix, we can instead iterate through each element of each row of the second matrix and accumulate the results in the final matrix, adding them all up throughout the way, resulting in a better memory access than the first algorithm. With less cache misses, the performance increases.

2.3 Block Matrix Multiplication

In this algorithm we want to multiply two block matrices. In order to do this, we have to iterate through both matrices and calculate each block in order to get to the final result.

We have 3 outer loops that iterate through the matrices' blocks and three inner loops that perform the multiplication on each block.

Regarding the first 3 loops:

- The first loop goes through the rows of the blocks of the first matrix.
- The second loop goes through the columns of the blocks of the first matrix and the rows of the blocks of the second matrix.
- The third loop goes through the columns of the blocks of the second matrix.

Regarding the last 3 loops:

- The first inner loop goes through the rows of the current block on the first matrix.
- The second inner loop goes through the columns of the current block of the first matrix and the rows of the current block of the second matrix
- The third inner loop goes through the columns of the current block of the second matrix.

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

Then, we calculate the value of the current position.

3. Performance

To evaluate the performance we used **PAPI**.

PAPI allows us to analyze the processing time of the program, the amount of cache hits and misses of both the caches **L1** and **L2**. We can use the recorded times of the program to measure the **speedup** and compare it to another version of the program. To obtain the speedup we must first reproduce a more parallelized version of the C++ program.

We chose cache hits and misses as performance evaluators because, in the context of a program, there are 2 different kinds of **times**: **execution time** and **memory access time**.

Memory access time is, in most scenarios, higher than the execution time. Due to this, the memory access time serves as a bottleneck to how fast the program can go. To reduce memory access time we must perform more cache hits and less cache misses.

A cache hit is when an element is already in the cache. A cache miss happens when an element isn't in the cache and needs to be brought over from another memory location. Cache access times are much faster compared to accessing elements from the main memory

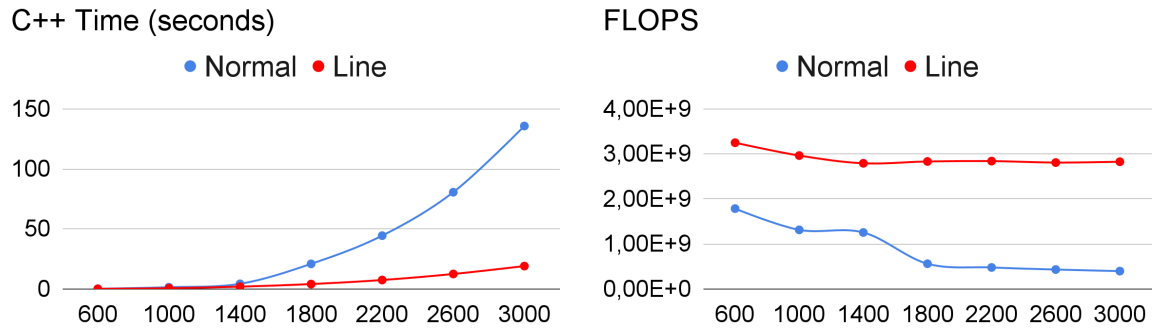
With this in mind, we are looking for algorithms which can **minimize** cache misses and, therefore, have better memory access performance, as to **maximize** overall program performance

4. Results and Analysis

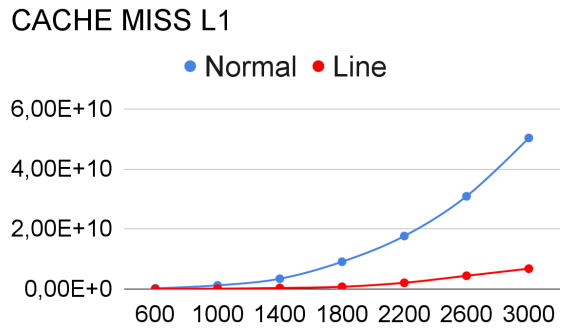
Here we will compare the algorithms with each other, each graph representing a performance evaluator discussed before, the horizontal axis representing the size of the matrix and the vertical axis representing the scale of the performance evaluator.

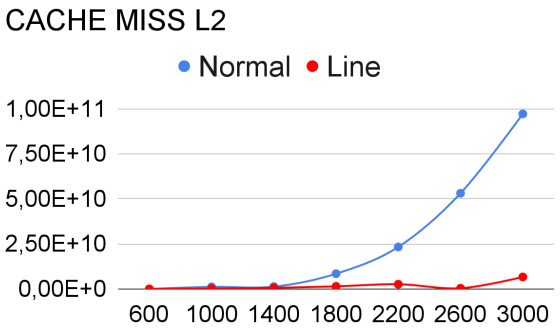
Analysis

4.1 Normal Matrix Multiplication vs Line Matrix Multiplication



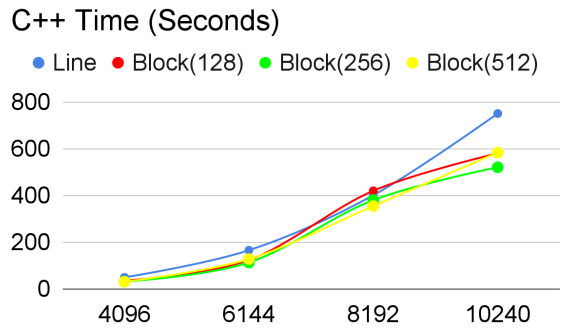
It is observable in the left graph that the **Line Matrix Multiplication** algorithm takes **overall less** time to compute compared to the **Normal** or **Naive Matrix Multiplication**. This is due to the fact that, in the **Normal Matrix Multiplication** algorithm, the **memory access pattern** is not optimized for **caches**, which leads to **poor cache utilization** and **frequent cache misses**. These **cache misses** decrease the performance of the program, because accessing data from the **main memory** is **more costly** compared to accessing it from the **cache**. The **Line Matrix Multiplication** algorithm, however, due to its more sequential approach to the computations, takes advantage of better **spatial locality** and is, therefore, more **cache-friendly.**
In the right graph we can see that there are more **floating point operations** being calculated **per second**. Because the computing time is overall better in the Line Matrix Multiplication algorithm, the program takes less time to finish, which, therefore, leads to more calculations being done per second, even though it's the same 2 operations being done every iteration: a multiplication of 2 doubles and a sum of 2 doubles.

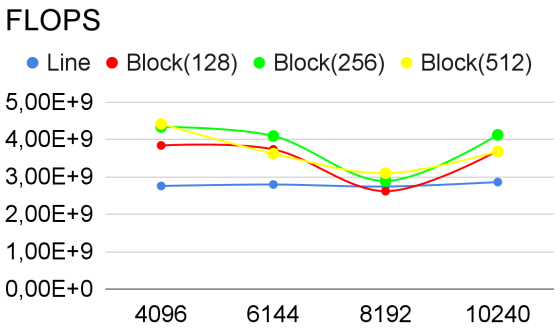




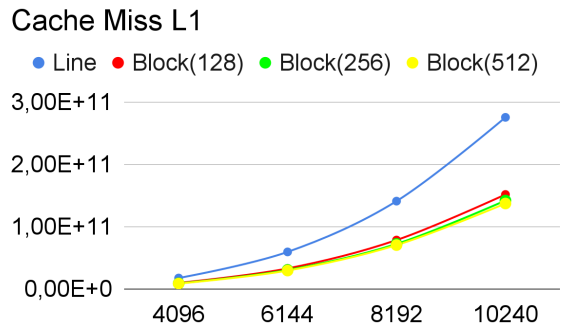
As we can see by these 2 graphs there are a lot **less cache misses** in the **Line Matrix Multiplication** algorithm than the **Normal Matrix Multiplication** algorithm. This happens, because every time we perform a cache miss on an array element, it brings that element and its neighbors. In the **Line Matrix Multiplication** algorithm we iterate the second matrix by each column, instead of each row, hitting the neighbors of each element instead of hitting an element that wasn't brought into the cache already. This **low amount of cache misses** results in a better performing time than the **Normal Matrix Multiplication** algorithm.

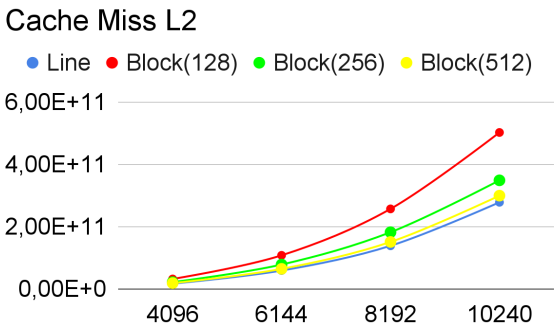
4.2 Line Matrix Multiplication vs Block Matrix Multiplication





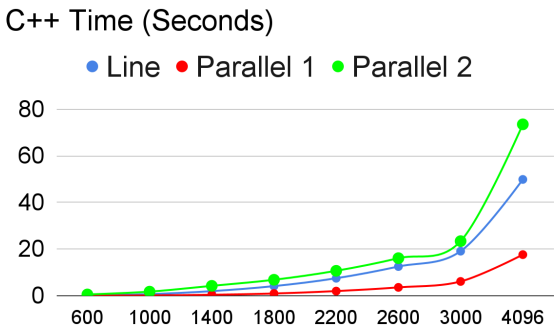
We can see by these graphs that the Block Matrix Multiplication takes less time but has more FLOPS. This happens because, although the amount of floating point operations are the same, since the time it takes to perform the Block Matrix Multiplication is less than the Line Matrix Multiplication algorithm, then we will be performing these floating point operations in less time resulting in a higher FLOPS. The time it takes to perform the Block Matrix Multiplication depends on the size of the block as we can see on the graph on the left, it seems that the sizes 256 and 512 are the better sizes to choose as block size, but which one of these to choose depends on the dimensions of the matrix.

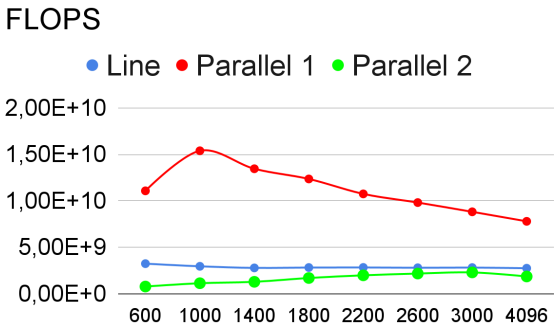




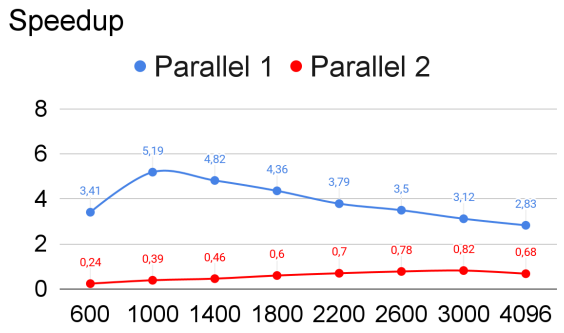
We can see that the Block Matrix Multiplication algorithm provides better cache locality in the L1 cache than the Line Matrix Multiplication algorithm. This happens because we are bringing blocks of elements into the cache and we are performing the operations on the elements on these blocks and then we move the block to another iteration. This makes it so that we are reusing a lot of elements instead of using it once and then coming back to it later (which would result in another extra cache miss). We also see that the Block Matrix Multiplication has more cache misses on the cache L2 the more the size of the blocks is raised. This happens because since the L1 cache is smaller than the L2 cache, we can assume that it can not hold that many elements compared to the L2 cache. Therefore, the more we increase the block size, the more elements can not be brought into the L1 cache and have to stay in the L2 cache. If we use a smaller block size we might not even bring elements into the L2 cache at all. Generally speaking, the higher the block size, the more likely the working set of data in the block will be located in the L2 cache, providing less L2 cache misses.

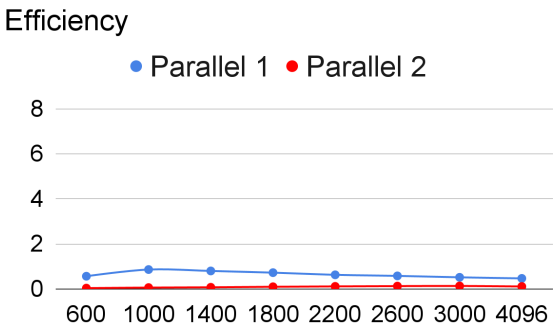
4.3 Line Matrix Multiplication vs Line Matrix Multiplication Parallelized





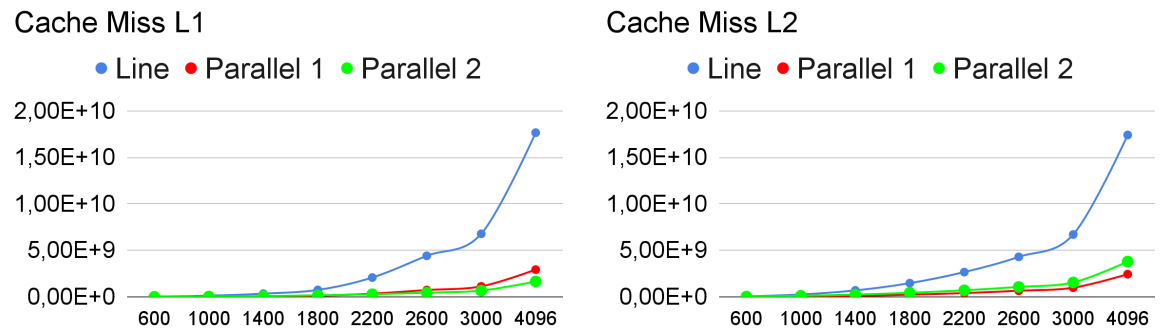
By analyzing this first graph we can see that the first parallelized version is faster than the other two and, despite being parallelized, the second parallelized version is actually slower than the sequential algorithm. We can also confirm this by looking at the second graph, where we can see that the first parallelized version has way more floating point operations per second than the other two while the second parallelized version did even worse than the sequential version. This may be due to some overhead due to distributing the loop among the threads or due to the waiting time for all threads to be synchronized.





We can also calculate the speedup by dividing the sequential time with the parallelized versions time. Just by looking, we can tell that the second parallelized version did not only not speed up the process, it actually slowed it down, having the worst speedup value at size 600 where speedup = 0.24. On the other hand, the first parallelized version did improve the time, having its highest speedup value with size 1000, where speedup = 5.19 and its lowest at size 4096 where speedup = 2.82. With these values, we can also calculate the efficiency

by dividing the speedup by the number of threads, where we confirm that the most efficient version of the code would be for the first parallelized version with size 1000.



We can see that the amount of cache misses is lower in the parallelized versions of the algorithm than the sequential one. This happens because when we parallelize the algorithm, we are using more processing units, which can be responsible for a subset of the data. Since we are essentially dividing the data into different processing units, it can now provide better cache locality since the data is more likely to be reused per unit because of the fact that the dataset is smaller for each unit.

5. Conclusion

In this project, we analyzed several approaches to the matrix multiplication problem. The focus of this project was to analyze the time, FLOPS, cache misses and speedup of the different algorithms and compare them with each other, varying the matrix size and/or block size. By analyzing graphs, tables and simulations we are able to reach conclusions regarding algorithms and the way they interact with memory, reinforcing what we have learned during the theoretical lessons.

Annexes

Values recorded

Normal Matrix Multiplication

Size	**C++ (Seconds)**	**C# (Seconds)**	**C++ (FLOPS)**	**C++ Cache Misses**
600	0.249	0.814	1 785 123 967	L1: 244784005
				L2: 40422476
1000	1.499	4.965	1 313 197 636	L1: 1215305413
				L2: 238459870

1400	4.330	14.631	1 255 260 750	L1: 3423853464
				L2: 1328147875
1800	20.948	36.229	561 498 098	L1: 9089498705
				L2: 8497682942
2200	44.433	70.487	478 529 537	L1: 17641166879
				L2: 23356297415
2600	80.589	119.643	432 618 702	L1: 30877098905
				L2: 53154946274
3000	135.716	187.866	397 675 806	L1: 50294280779
				L2: 97199287004

Line Matrix Multiplication

Size	**C++ (Seconds)**	**C# (Seconds)**	**C++ (FLOPS)**	**C++ Cache Misses**
600	0.133	0.846	3 248 120 301	L1: 27115987
				L2: 55800833
1000	0.675	3.899	2 962 962 963	L1: 125795976
				L2: 256223720
1400	1.966	10.752	2 791 454 730	L1: 346164533
				L2: 693940514
1800	4.119	22.792	2 831 755 280	L1: 745307091
				L2: 1475284528
2200	7.499	41.632	2 839 845 312	L1: 2076048892
				L2: 2664292160
2600	12.521	68.443	2 807 443 495	L1: 4413120510
				L2: 4300955949

3000	19.107	104.917	2 826 189 355	L1: 6780630736
				L2: 6705101650
4096	49.877		2 755 557 741	L1: 17654791881
				L2: 17403496604
6144	165.976		2 794 720 128	L1: 59542788098
				L2: 59309674223
8192	401.148		2 740 912 650	L1: 141000356272
				L2: 138740481955
10240	750.871		2 859 990 129	L1: 275440966195
				L2: 278301579435

Block Matrix Multiplication

Size	**Block Size**	**C++ (Seconds)**	**C++ (FLOPS)**	**C++ Cache Misses**
4096	128	35.801	3 838 969 680	L1: 9776635577
				L2: 32289821248
	256	31.720	4 332 879 995	L1: 9093628438
				L2: 22086374792
	512	31.143	4 413 157 161	L1: 8772705440
				L2: 19011036539
6144	128	124.226	3 733 972 501	L1: 33023026515

				L2: 108138371938
	256	113.354	4 092 104 980	L1: 30603240624
				L2: 78518636767
	512	128.147	3 619 721 631	L1: 29628330299
				L2: 64714880006
8192	128	420.257	2 616 283 911	L1: 78563791767
				L2: 257121470420
	256	380.471	2 889 869 734	L1: 72362900211
				L2: 182270718647
	512	355.111	3 096 247 730	L1: 70491395471
				L2: 151133840440
10240	128	580.705	3 698 062 954	L1: 151679927739
				L2: 502355221600
	256	520.953	4 122 221 482	L1: 141936565145
				L2: 348780721512
	512	583.723	3 678 943 005	L1: 137214141759
				L2: 299588595542

Line Matrix Multiplication Parallelized – Version 1

Size	**C++**	**C++**	**Speedup**	**Efficiency**	**C++ Cache**

	(Seconds)	(FLOPS)			Misses
600	0.039	11 076 923 077	3.410	0.568	L1: 4523304
					L2: 9557149
1000	0.130	15 384 615 385	5.192	0.865	L1: 21026749
					L2: 42443650
1400	0.408	13 450 980 392	4.819	0.803	L1: 58185519
					L2: 113495293
1800	0.944	12 355 932 203	4.363	0.727	L1: 124922082
					L2: 235053441
2200	1.981	10 750 126 199	3.785	0.631	L1: 345231621
					L2: 407965683
2600	3.581	9 816 252 443	3.497	0.583	L1: 733649988
					L2: 657984418
3000	6.121	8 822 087 894	3.122	0.520	L1: 1126931364
					L2: 982875854
4096	17.625	7 797 954 806	2.830	0.472	L1: 2928449820
					L2: 2438844504

Line Matrix Multiplication Parallelized – Version 2

Size	C++ Seconds	C++ FLOPS	Speedup	Efficiency	C++ Cache Misses
600	0.548	788 321 167	0.243	0.040	L1: 9147712

					L2: 35008673
1000	1.749	1 143 510 577	0.386	0.064	L1: 34648107
					L2: 118486231
1400	4.236	1 295 561 851	0.464	0.077	L1: 83080193
					L2: 244271028
1800	6.841	1 705 013 886	0.602	0.100	L1: 164944296
					L2: 440119663
2200	10.686	1 992 887 891	0.702	0.117	L1: 283479065
					L2: 704405652
2600	16.105	2 182 676 188	0.777	0.130	L1: 451726492
					L2: 1070981145
3000	23.407	2 307 002 179	0.816	0.136	L1: 675110604
					L2: 1551589307
4096	73.506	1 869 765 100	0.679	0.113	L1: 1654238383
					L2: 3795347923