Performance Evaluation of a Single Core and a Multi-core Implementation

Ana Ramos
up201904969

Diogo Martins
up202108883

Linda Rodrigues
up202005545

1 Introduction

This project is structured in two phases: first, the evaluation of the performance of a single core and then, the evaluation of the performance of a multi-core implementation. The first phase aims to study the impact of memory hierarchy on processor performance when accessing large amount of data. By analyzing the performance metrics during program execution using the Performance API, we gain insights on how effectively the processor's cache memory works during matrix multiplication. Additionally, we implement and compare multiple algorithms with different memory allocation strategies to understand how memory management techniques impact algorithm efficiency. The second phase involves the implementation of parallel versions of the line matrix multiplication algorithm using Open Multi-Processing for us to analyze and compare its performance. In the following sections, we delve deeper into the problem description, explaining the algorithms used, and discussing the methodologies employed to study the impact of memory hierarchy on processor performance.

2 Problem Description and Algorithms Explanation

2.1 Problem

As mentioned before, the first phase of this project aims to study the effect on the processor performance of the memory hierarchy when accessing large amount of data. To achieve this, we utilized the product of two matrices, combined with the Performance API (PAPI) that allowed us to analyze and collect relevant performance metrics during the program execution. PAPI provided us the capability to track cache-related metrics such as the number of cache misses which helps with understanding how effectively the processor's cache memory is, particularly during matrix multiplication.

In the last phase, we used Open Multi-Processing (OpenMP) to implement parallel versions of the line matrix multiplication algorithm. OpenMP is a library designed for parallel programming in multi-processors. It employs a special directive, `omp pragma` , to mark sections of code to be executed in parallel. Upon encountering this directive, OpenMP facilitates the creation of slave threads to execute the designated parallelized code segments [1]. This allows us to explore the benefits of parallelization in optimizing performance and have a better understanding of memory hierarchy's impact on processor performance. [2]

2.2 Algorithms

For the first part of this project, we implemented three different algorithms to measure the performance of a single core when exposed to a large amount of data. These algorithms significantly vary in their performance due to differences in memory allocation strategies.

- Simple Multiplication (already provided)
- Line Multiplication
- Block Multiplication

For the Simple Multiplication and the Line Multiplication, it was asked to implement the algorithms in C/C++ and in another programming language of our choice. We selected Java as the alternative language due to its similar syntax and therefore easy translation. Additionally, Java offers a different memory management model, compared to C/C++. While the latter relies on manual memory allocation, Java memory management is system-controlled [3]. This distinction allowed us to understand how different memory management approaches impact the performance of the implemented algorithms.

2.2.1 Simple Matrix Multiplication

```

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

The algorithm represented above (in C/C++) represents a straightforward implementation of matrix multiplication in which one line of the first matrix is multiplied by each column of the second matrix. The algorithm has a time complexity of $O(n^3)$, where n represents the dimensions of the matrices being multiplied.

2.2.2 Line Matrix Multiplication

```

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

In the second algorithm, we implemented a version that multiplies an element from the first matrix by the correspondent line of the second matrix. Despite the only apparent change being the order of the loops, especially since both have the same time complexity of $O(n^3)$, it has performance implications such as cache efficiency as we will discuss in the following sections.

2.2.3 Block Matrix Multiplication

```

for (bki = 0; bki < m_ar; bki += bkSize)
    for (bkj = 0; bkj < m_br; bkj += bkSize)
        for (bkk = 0; bkk < m_ar; bkk += bkSize)
            for(i = bki; i < ((bki + bkSize) > m_ar?m_ar:(bki + bkSize)); i++)
                for(k = bkk; k<((bkk + bkSize) > m_ar?m_ar :(bkk+bkSize)); k++)
                    for(j = bkj; j<((bkj + bkSize)>m_br?m_br:(bkj+bkSize)); j++)
                        phc[i*m_ar+j] += pha[i*m_ar+k] * phb[k*m_br+j];
```

In the last algorithm, it was requested to adopt a block-based approach to matrix multiplication. The outermost loops divide the matrices into smaller blocks (size `bkSize`) and within these blocks, the subsequent nested loops iterate over the elements of the matrices, performing matrix multiplication operations. The use of a block multiplication approach, despite the additional complexity, aims to optimize cache efficiency and memory access patterns by exploiting spatial locality.

2.2.4 Linear Matrix Multiplication with Parallelism

In the second part of this project, we were tasked with employing the OpenMP software to enforce parallelism within our algorithm for matrix multiplication by line. We tested two distinct implementations for parallelism.

3 Performance Evaluation of a Single Core

3.1 Performance Metrics

As mentioned previously, we used the Performance API for the performance of algorithms made in C/C++, which is used to access and analyze performance metrics according to processor architectures and cache memory levels. To do the measurements, we used the faculty's computers during class to ensure consistent results across all the experiments. Each computer is equipped with 8 cores and we ran the tests using Ubuntu as the operating system.

The process for calculating the average time for each algorithm varied as follows:

- **Simple matrix multiplication and line multiplication:** 10 iterations each to determine the time average
- **Block matrix multiplication:** we did two iterations for matrices of sizes 4096x4096 and 6144x6144. However, due to time constraints, only one iteration was performed for larger values such as 8192x8192 and 10240x10240.

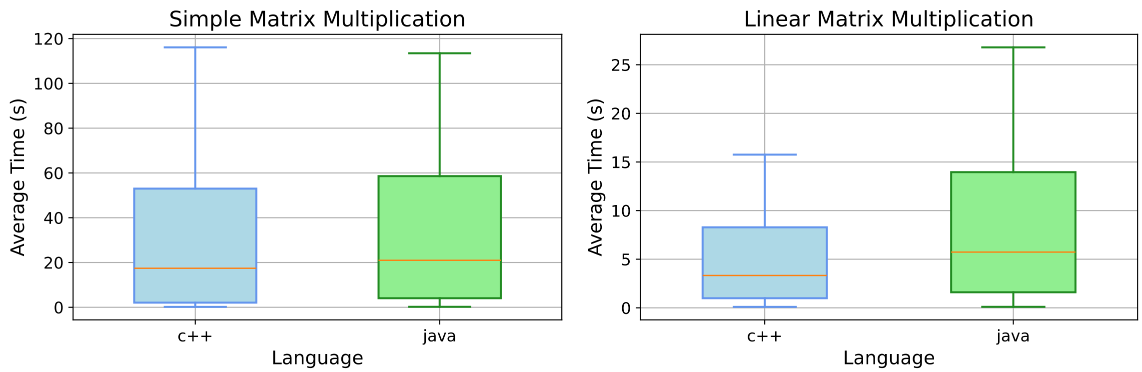
3.2 Results and Analysis

As discussed in the previous section, the results represent the average of 10 consecutive tests. In the following subsections, we present graphs, more precisely box plots, to display the results we obtained for all the algorithms.

The results of the comparison of the two programming languages, C++ and Java, are shown on Figure 1, utilizing the simple matrix multiplication algorithm (Figure 1a) and the linear matrix multiplication algorithm (Figure 1b).

When comparing the performance of algorithms implemented in C++ and Java, it is evident that C++ consistently outperforms Java in terms of execution time, which is more noticeable as the matrices dimensions grow larger. In terms of execution time between the two algorithms, simple matrix multiplication takes longer to execute when compared to linear multiplication. The latter iterates over each column first, then each row, exhibiting better cache locality by avoiding unnecessary cache misses. This rearrangement leads to better cache utilization and improved performance, specially when the matrices are larger.

The block matrix multiplication algorithm's performance is influenced by both the block size and the matrix dimensions. Smaller block sizes, such as 128 or 192, result in slightly higher average times compared to larger block sizes for certain matrix dimensions, due to increased overhead associated with managing smaller blocks. However, this relationship is not strictly linear, since in certain



(a) Simple Matrix Multiplication

(b) Matrix Line Multiplication

Figure 1: C++ vs Java - Average Execution Times

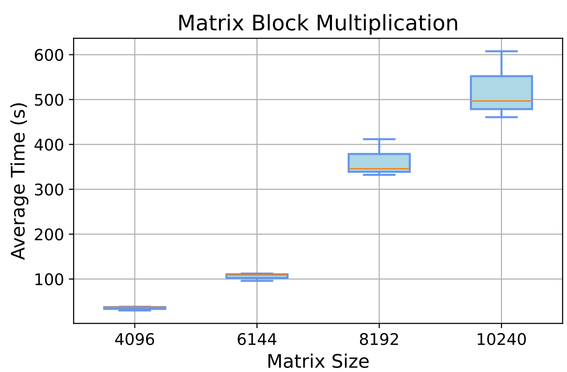


Figure 2: Block Matrix Multiplication Execution Times For Different Block Sizes

scenarios, a smaller block size may lead to better performance compared to a slightly larger block size. Additionally, as observed in the other two algorithms, the computation workload increases with larger matrices, leading to longer average execution times. By breaking down the multiplication into smaller computations, the algorithm exploits better cache locality and reduces cache misses, contributing to improved overall.

4 Performance Evaluation of a Multi-core Implementation

4.1 Performance Metrics

To assess the performance of our multi-core implementations, we calculated the MFLOPs, speedup, and efficiency, using the following formulas:

- $MFLOPs = 2N^3/time$, where N is the number of lines or columns in the matrix;
- $speedup = time_{sequential}/time_{parallel}$;

- efficiency = $speedup/\#cores$, where $\#cores$ represents the number of cores in the utilized computer.

4.2 Results and Analysis

For this second part, we had to use a different computer for the benchmarks, which has 6 performance-cores and 8 efficient-cores. We, then, decided to re-run the results from the Linear Matrix Multiplication execution times on the new device. This allows us to better compare the execution times from the sequential (single core) implementation to the results obtained from the parallelism (multi-core) implementations.

For these implementations, we ran each combination of size and implementation three times, calculating the average execution time. Additionally, we ran all previously considered matrix sizes, ranging from 600 to 10240. However, presenting the results in a box plot had its complications due to the extreme limits of our data distribution.

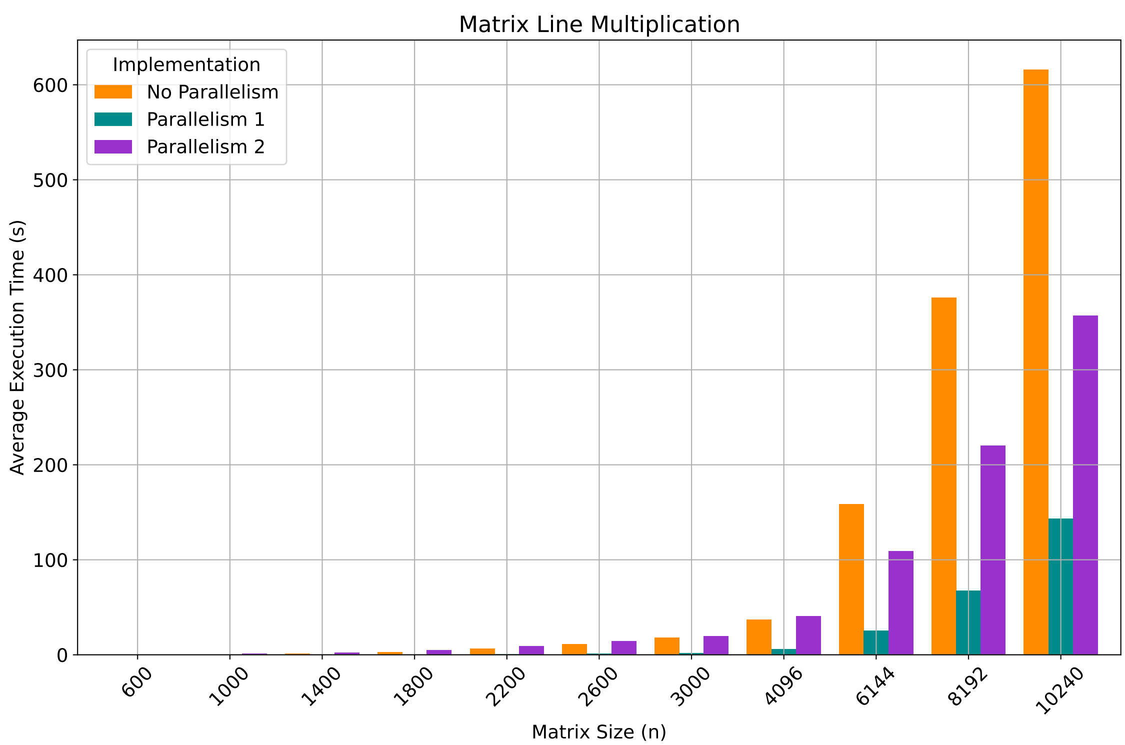


Figure 3: Linear Matrix Multiplication (Sequential vs Parallelism)

The average execution time results for the Matrix Line Multiplication algorithm are represented in a box plot, as shown in Figure 3.

We can see that the first parallel version of the algorithm, where the outermost loop is parallelized, is substantially better than the sequential version and the other parallel version, having a speedup superior to 4 for all matrix sizes, despite its efficiency reaching a peak around 2200x2200 and steadily declining starting on 4096x4096, having smaller gains the bigger the matrix (shown in Table 2). This is because the performance bottleneck is not the processor's speed anymore, but the memory bandwidth. The bigger the matrix, the more simultaneous accesses to memory there will be for all the different cores, and it becomes overloaded with requests causing the algorithm to slow down.

The second version of the algorithm only parallelizes the innermost loop, i.e., all threads execute the two outer loops in their entirety but divide the work of the inner one, and fares much worse because of it, even getting outstripped by the sequential version until the matrices are bigger than 4096x4096, as Figure 3 shows.

Matrix Dimension	Sequential MFLOPS	V1 MFLOPS	V2 MFLOPS
600	5538	30857	1014
1000	5917	43478	1749
1400	4170	46905	2298
1800	3914	42260	2279
2200	3228	35026	2307
2600	3146	31301	2472
3000	2951	28938	2730
4096	3705	23534	3363
6144	2920	18187	4256
8192	2923	16207	4995
10240	3484	14956	6014

Table 1: MFLOPS for the three algorithms

Matrix Dimension	V1 Speedup	V1 Efficiency (%)	V2 Speedup	V2 Efficiency (%)
600	5.571	39.8	0.183	1.3
1000	7.348	52.5	0.296	2.1
1400	11.248	80.3	0..551	3.9
1800	10.797	77.1	0.582	4.2
2200	10.849	77.5	0.715	5.1
2600	9.947	71.1	0.786	5.6
3000	9.804	70.0	0.925	6.6
4096	6.351	45.4	0.908	6.5
6144	6.228	44.5	1.457	10.4
8192	5.543	39.6	1.709	12.2
10240	4.292	30.7	1.726	12.3

Table 2: Speedup for the three algorithms

5 Conclusions

Through this project, we gained valuable insights on how the memory hierarchy influences processor performance, particularly when accessing large datasets. Through the use of tools such as PAPI and OpenMP, we can better understand and optimize the performance of matrix operations in computational tasks. Moving forward, these insights can serve as a foundation for further exploration of performance strategies in computational tasks.

References

[1] L. Toma, "Intro to parallel programming with OpenMP by Laura Toma, Bowdoin College," last Accessed on March 17, 2024. [Online]. Available: https://tildesites.bowdoin.edu/ ltoma/teaching/cs3225-GIS/fall17/Lectures/openmp.html

[2] "OpenMP Home," Nov 2023, last Accessed on March 17, 2024. [Online]. Available: https://www.openmp.org/

[3] "C++ VS Java: 20 key differences between C++ and Java in 2024," Nov 2023, last Accessed on March 17, 2024. [Online]. Available: https://www.mygreatlearning.com/blog/cpp-vs-java/

6 Annexes

6.1 Part 1

6.1.1 Default multiplication

Language	Matrix Dimension	Average
C++	600 x 600	0.184
	1000 x 1000	0.968
	1400 x 1400	3.195
	1800 x 1800	17.434
	2200 x 2200	37.063
	2600 x 2600	68.937
	3000 x 3000	116.089
Java	600 x 600	0.216
	1000 x 1000	1.471
	1400 x 1400	6.645
	1800 x 1800	20.989
	2200 x 2200	42.763
	2600 x 2600	74.386
	3000 x 3000	113.453

6.1.2 Line multiplication

For implementation done both in C++ and Java:

Language	Matrix Dimension	Average
C++	600 x 600	0.097
	1000 x 1000	0.461
	1400 x 1400	1.517
	1800 x 1800	3.326
	2200 x 2200	6.154
	2600 x 2600	10.408
	3000 x 3000	15.756
Java	600 x 600	0.102
	1000 x 1000	0.541
	1400 x 1400	2.662
	1800 x 1800	5.733
	2200 x 2200	10.517
	2600 x 2600	17.391
	3000 x 3000	26.790

For the implementation only asked in C++:

Language	Matrix Dimension	Average
C++	4096 x 4096	40.850
	6144 x 6144	137.469
	8192 x 8192	334.489
	10240 x 10240	644.092

6.1.3 Block Multiplication

Matrix Dimension	Block Size	Average
6144	768	112.181
	384	95.904
	192	108.999
4096	512	38.134
	256	29.960
	128	36.852
8192	1024	332.219
	512	345.764
	256	411.685
10240	1280	607.585
	640	496.695
	320	460.762

6.2 Part 2

6.2.1 Sequential

Language	Matrix Dimension	Average
C++	600 x 600	0.078
	1000 x 1000	0.338
	1400 x 1400	1.316
	1800 x 1800	2.980
	2200 x 2200	6.596
	2600 x 2600	11.171
	3000 x 3000	18.294
	4096 x 4096	37.091
	6144 x 6144	158.840
	8192 x 8192	376.052
	10240 x 10240	616.2777

6.2.2 First parallel implementation

Language	Matrix Dimension	Average
C++	600 x 600	0.014
	1000 x 1000	0.046
	1400 x 1400	0.117
	1800 x 1800	0.276
	2200 x 2200	0.608
	2600 x 2600	1.123
	3000 x 3000	1.866
	4096 x 4096	5.840
	6144 x 6144	25.504
	8192 x 8192	67.839
	10240 x 10240	143.5793

6.2.3 Second parallel implementation

Language	Matrix Dimension	Average
C++	600 x 600	0.426
	1000 x 1000	1.143
	1400 x 1400	2.388
	1800 x 1800	5.118
	2200 x 2200	9.230
	2600 x 2600	14.220
	3000 x 3000	19.779
	4096 x 4096	40.865
	6144 x 6144	108.984
	8192 x 8192	220.097
	10240 x 10240	357.025