Faculdade de Engenharia da Universidade do Porto

Performance evaluation of a single-core and a multi-core implementation

António Matos (up202006866)
Davide Teixeira (up202109860)
Emanuel Maia (up202107486)

Licenciatura em Engenharia Informática e Computação

Professor: Carlos Miguel Ferraz Baquero-Moreno

March 17, 2024

Contents

1 Introduction

This report showcases how different methods of accessing array elements can impact program execution time in array computations. We conducted two experiments: one comparing memory access times using the same algorithm with different access patterns, and another evaluating the effect of parallelization on execution time.

2 Problem description and algorithms explanation

In our investigation, we delved into three distinct algorithms aimed at optimizing sequential programs through memory manipulation. These algorithms (the traditional Matrix Multiplication, Line Matrix Multiplication, and Block Matrix Multiplication) were scrutinized for their performance on a dual core system. To further validate these performances, we conducted additional tests using various matrix sizes and configurations. Through these efforts, we sought not only to evaluate execution time but also to optimize memory utilization for enhanced sequential computing performance.

2.1 Matrix Multiplication

Matrix multiplication serves as the foundational algorithm in our study. It offers a straightforward method for computing the product of two matrices. The algorithm, detailed in Algorithm 1, has been implemented in both C++ and TypeScript. Its complexity is $O(n^3)$.

Algorithm 1 Matrix Multiplication

```
1: function MatrixMultiply(matrixA, matrixB)
2:     for $i = 0$ to len(matrixA) $- 1$ do
3:         for $j = 0$ to len(matrixB[0]) $- 1$ do
4:             temp $\leftarrow 0$
5:             for $k = 0$ to len(matrixA[0]) $- 1$ do
6:                 temp $\leftarrow$ temp + matrixA$[i][k] \times$ matrixB$[k][j]$
7:             end for
8:             matrixC$[i][j] \leftarrow$ temp
9:         end for
10:    end for
11:    return $matrixC$
12: end function
```

2.2 Line Matrix Multiplication

To enhance performance, we devised a modification to the traditional matrix multiplication algorithm. This enhancement, presented in Algorithm 2, differs from a traditional matrix multiplication by rearranging loops that iterate over the matrix indices, thus processing matrix elements line by line. Its complexity is $O(n^3)$.

Algorithm 2 Line Matrix Multiplication

```
1: function MatrixMultiply(matrixA, matrixB)
2:     for $i = 0$ to len(matrixA) $- 1$ do
3:         for $k = 0$ to len(matrixB[0]) $- 1$ do
4:             temp $\leftarrow 0$
5:             for $j = 0$ to len(matrixA[0]) $- 1$ do
6:                 temp $\leftarrow$ temp + matrixA$[i][k] \times$ matrixB$[k][j]$
7:             end for
8:             matrixC$[i][j] \leftarrow$ temp
9:         end for
10:    end for
11:    return matrixC
12: end function
```

2.3 Block Matrix Multiplication

Further optimizing our approach, we introduced the concept of block matrix multiplication. This technique, detailed in Algorithm 3, involves partitioning the matrices into smaller blocks, potentially reducing cache misses and improving performance by enabling more frequent and faster storage of data in cache memory, which leads to less time spent on retrieving data from slower higher-level memories. Its complexity is $O(n^3)$.

Algorithm 3 Block Matrix Multiplication

```
1: function BlockMatrixMultiply(matrixA, matrixB, blockSize)
       (rowsA, colsB)
2:     for $i = 0$ to rowsA $- 1$ step blockSize do
3:         for $j = 0$ to colsB $- 1$ step blockSize do
4:             for $k = 0$ to colsA $- 1$ step blockSize do
5:                 for $ii = i$ to min$(i +$ blockSize, rowsA$)$ do
6:                     for $jj = j$ to min$(j +$ blockSize, colsB$)$ do
7:                         temp $\leftarrow 0$
8:                         for $kk = k$ to min$(k +$ blockSize, colsA$)$ do
9:                             temp $\leftarrow$ temp + matrixA$[ii][kk] \times$ matrixB$[kk][jj]$
10:                        end for
11:                        matrixC$[ii][jj] \leftarrow$ matrixC$[ii][jj]$ + temp
12:                    end for
13:                end for
14:            end for
15:        end for
16:    end for
17:    return matrixC
18: end function
```

2.4 Multi-thread Implementation

In addition to those algorithms, two more variations of the second algorithm were done. Those variations are based on making a parallelized version of line matrix multiplication.

Those algorithms are implemented in C++ in Listing 1 and Listing 2, respectively.

```cpp
#pragma omp parallel for
for (int i=0; i<n; i++)
    for (int k=0; k<n; k++)
        for (int j=0; j<n; j++){}
```

Listing 1: C++ code with Parallel Instructions - Implementation 1

```cpp
#pragma omp parallel
for (int i=0; i<n; i++)
    for (int k=0; k<n; k++)
        #pragma omp for
        for (int j=0; j<n; j++) {}
```

Listing 2: C++ code with Parallel Instructions - Implementation 2

3 Performance Metrics

The performance evaluation of algorithms often relies on comprehensive testing conducted on specific computing systems. In our study, we aimed to assess the efficiency of various algorithms using two key performance metrics: execution time and cache misses. To achieve this, we utilized a computing system with specific hardware specifications and performance monitoring capabilities.

Our testing was carried out on a system running under Arch Linux, on the 6.7.8 kernel, equipped with an Intel(R) Celeron(R) CPU 1007U @ 1.50GHz. This CPU has a maximum frequency of 1500 MHz and a minimum frequency of 800 MHz. The system consists of 2 cores, each with 1 SMT thread, resulting in 2 total cores and 2 cores per NUMA region. It runs on a single socket and is not deployed within a virtual machine environment[2].

Additionally, the system features multiple levels of cache memory, including L1d cache with a capacity of 64 KiB (2 instances), L1i cache also with 64 KiB (2 instances), L2 cache with 512 KiB (2 instances), and L3 cache with 2 MiB (1 instance)[3].

Furthermore, our performance monitoring was facilitated by PAPI version 7.1.0.0, providing access to 10 hardware counters and allowing a maximum of 384 multiplex counters. Notably, the system supports fast counter read (rdpmc) functionality, enhancing the efficiency of performance data retrieval.

As for running the code itself, the C++ version was compiled with gcc version 13.2.1, using the `-O2`, `-lpapi` and `-fopenmp` flags (the last one being used only for the second part of this project, involving multiprocessing. For TypeScript, the code was compiled to JavaScript using tsc version 5.3.3, and then run with NodeJS version 21.6.2.

By measuring not only execution time and L1 cache misses, but also L2 cache accesses and FLOPS (floating point operations per second) across different algorithms, we gained valuable insights into their computational efficiency and memory utilization. This approach allowed us to make informed assessments of algorithmic performance and identify potential areas for optimization.

4 Results and analysis

4.1 Comparison between 3 algorithms in C++ and TypeScript

The first analysis that was done was tentative to relate the execution time of the algorithm with the matrix size for the different single algorithms (Algorithms 1, 2)—leading to the result on figure 1. In all of the three cases, we can see that the temporal complexity of the algorithm is clearly $O(n^3)$.

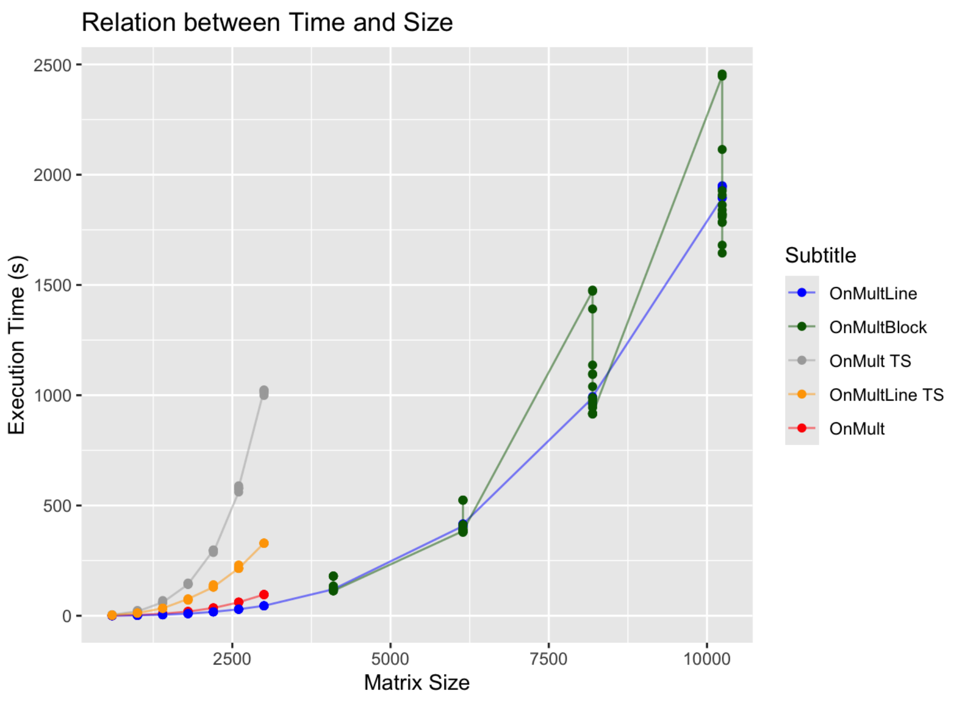

Figure 1: Relation between Matrix Size and Execution time for the different algorithms

The first conclusion we can check is that algorithm 1 is the slowest in most cases. As expected, algorithm 2 is clearly a cut above algorithm 1 when it comes to efficiency and execution time. That happens because of the way cache works. Cache works with the principle of spatial closeness[1, 4], which means that cache assumes that if a segment of memory is accessed at some time, the adjacent memory spaces have more probability of being accessed soon.

In algorithm 1, the memory access is not done sequentially, creating a great number of cache misses when trying to access memory. More cache misses mean a more number of accesses to the main memory, which will increase the latency of the algorithm.

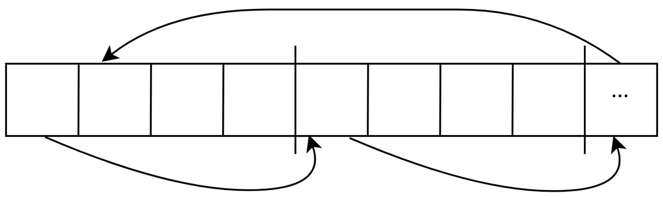

Figure 2: Algorithm 1 access memory

In algorithm 2, with the change of the for loop, the memory access is done in a sequential way, which creates less cache misses and, consequently, less time to execute.

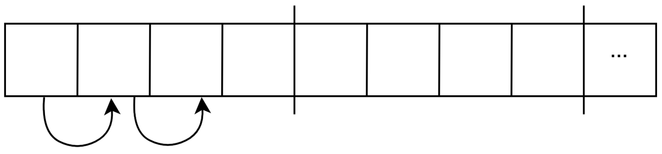

Figure 3: Algorithm 2 access memory

As the empirical values show, there exists a highly linear correlation between the number of cache misses and the execution time ($r^2 = 0.9998$), which makes sense with the theoretical values.

Another very obvious conclusion is that, even with the faster algorithm (as seen in the orange line), TypeScript takes significatively longer than C++ in the slowest algorithm (in the red line). Not only that, but execution time in TypeScript seems to grow at a higher rate from matrix size than C++. These results can be blamed on many factors.

Firstly, C++ is a low-level language, with little to no overhead such as runtime environments, and direct access to the memory, making matrix or general algebraic operations concise. TypeScript is a high-level language, focused on features that facilitate maintainability, but lacking the optimizations of C++. It is also important to note that C++ has better performance with direct compilation due using low level optimizations, while Typescript is typically executed in a browser, introducing some overhead when compared to native machine code. Finally there's the fact that C++ puts the programmer in charge of the memory management, while TypeScript handles it automatically, introducing further overhead in the form of a Garbage Collector.

4.2 Comparison between block size and execution time

Another test that was done to evaluate the correlation between the Block Size of the Data Cache Misses andnthe execution time. Figure 1 shows the output of that study.

The main takeaway is that the use of blocks decreases the number of cache misses and, consequently, the execution time depending on the block size. In general, the bigger the block size is, the smaller the amount of cache misses we have, as seen in figure 4.

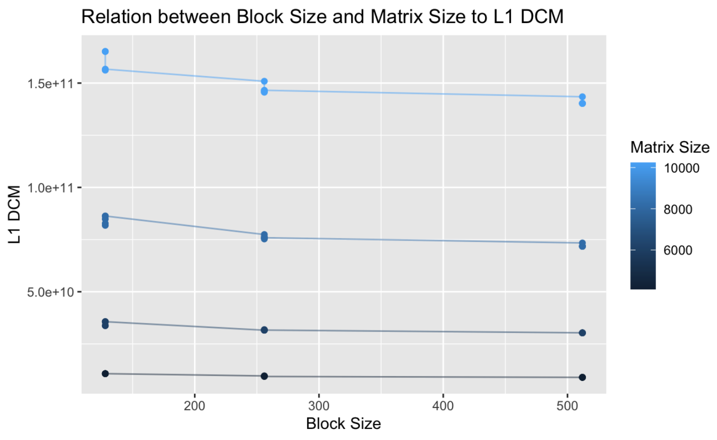

Figure 4: Relation between Block Size and Matrix Size to L1 DCM

4.3 Comparison between Single Core and Parallelized version

The final major test had the objective of seeing the influence of the parallelization in the matrix multiplication algorithm. For this purpose, we used 2 multi-core parallel algorithms and compared them to the multi-line algorithm as a benchmark.

As expected, the algorithm parallelization increased the performance and reduced the execution time to approximately a half of the original time. That measure was expected, as we are working with a dual-core machine.

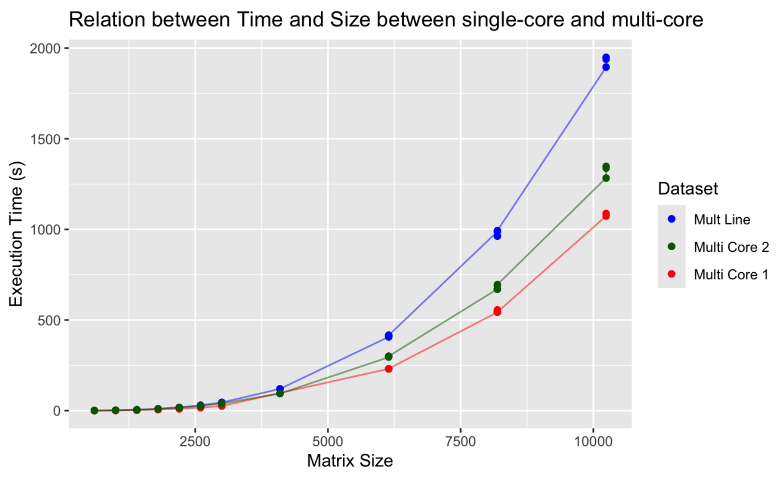

Figure 5: Relation between Time and Size between single-core and multi-core

We can observe that the 2^{nd} multi-core implementation is notably the slowest of the parallelized algorithms. This is because of the parallelization that will launch threads in $O(n^2)$. The sequential algorithm, however, remains the slowest, as we expected.

5 Conclusions

To sum it up, our research of memory management illustrates the value in boosting program efficiency within sequential computing. These findings highlight the importance of strategic memory manipulation, specially in maximizing the utilization of lower-level caches, thereby minimizing latency and enhancing overall performance. We can also have a better understanding of the advantages in performance of the memory hierarchy of a language like C++ compared to TypeScript.

Our results emphasize that, through techniques that leverage spatial locality and mitigate cache misses, developers can substantially enhance the responsiveness of sequential programs. As so, we conclude that taking hardware architecture into account when designing algorithms can lead to more efficient computing systems, with optimal performance.

References

[1] Jorge Barbosa. *Cache Memory and the impact on processor performance / data locality*.

[2] *Intel's webpage for the Intel® Core™ Celeron® 1007U Processor*. URL: `https://www.intel.com.br/content/www/br/pt/products/sku/72061/intel-celeron-processor-1007u-2m-cache-1-50-ghz/specifications.html`.

[3] *PAPI Standard Events By Architecture from the Innovative Computing Laboratory of the University of Tennessee*. URL: `https://icl.utk.edu/projects/papi/presets.html`.

[4] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, 2004. ISBN: 0071232656.

A Useful Links

Additional graphs and data sheets with our results can be found in this Google Sheets.

The code for our project is on GitHub and GitLab.