CPD Assignment 1 Report

Performance evaluation of single-core and multi-core implementations of matrix product

Daniel Costa Basílio - up201806838
Diogo Maria Santos Venade - up202207805
Miguel Tomás Vieira Rodrigues - up202205749

Licenciatura em Engenharia Informática e Computação

1. Introduction

In this report we explore the differences between different algorithms for matrix multiplication, both in single-core and multi-core implementations. The main objective of the assignment was to determine the effects of these algorithms in processor performance.

2. Problem Description

This assignment can be divided into two parts. In the first part, we implemented three distinct, single-core algorithms for matrix multiplication: **standard multiplication**, **line multiplication** and **block multiplication**. The implementation was done in C++ for all three algorithms, and additionally in Java for the first two algorithms. The goal was to explore the differences in efficiency between the algorithms, and additionally between the two languages.

In the second part, we implemented two parallel versions of line multiplication, in C++, analysing the differences between the two and comparing them with the single-threaded version.

3. Algorithms Explanation

3.1. Standard Matrix Multiplication

In this algorithm, the first loop iterates over the rows of matrix A and the second loop over the columns of matrix B. The inner loop is used to iterate over the elements of the corresponding rows and columns. This follows the basic workings of matrix multiplication, where each element of the result matrix is the dot product of a row from the first matrix and a column for the second matrix.

```
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
```

***Fig. 1**: Nested loops of the standard matrix multiplication algorithm*

3.2. Line Matrix Multiplication

The main difference between this algorithm and the previous one is that the order of the last two loops is switched. Now, an element from matrix A will be multiplied by the corresponding line of matrix B. This results in matrix B being traversed over its rows, as opposed to its columns.

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

***Fig. 2**: Nested loops of the line matrix multiplication algorithm*

3.3. Block Matrix Multiplication

This algorithm iterates the two matrices over blocks of size chosen by the user, which are iterated over by the outer loops. When multiplying between two blocks, within the inner loops, the line multiplication algorithm is used. It should be noted that the *min()* function was used to ensure that, when the block size doesn't perfectly divide the matrices, we stay within the matrix boundaries.

```
for (i=0; i<m_ar; i+=bkSize)
   for (j=0; j<m_br; j+=bkSize)
      for (k=0; k<m_ar; k+=bkSize)
         for (l=i; l<min(i+bkSize, m_ar); l++)
            for (m=k; m<min(k+bkSize, m_ar); m++)
               temp = pha[l*m_ar+m]
                     for (n=j; n<min(j+bkSize, m_br); n++)
                     phc[l*m_ar+n] += temp * phb[m*m_br+n]
```

***Fig. 3**: Nested loops of the block matrix multiplication algorithm*

3.4. Line Matrix Multiplication - first parallel version

In this implementation, the outer loop is parallelized across multiple threads through the use of OpenMP and the *pragma* directive. This means that each thread will work with different rows of matrix A and the result matrix.

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

***Fig. 4**: Nested loops of the first parallel version of line matrix multiplication*

3.5. Line Matrix Multiplication - second parallel version

This algorithm starts by parallelizing the entire block of code, while ensuring that each thread has its own private copy of the *i*, *k* and *temp* variables. Then, the second *pragma* directive distributes the execution of the inner loop (which handles the columns of the result matrix) by the threads that had been previously created.

```
#pragma omp parallel private(i, k, temp)
for (i=0; i<m_ar; i++)
   for (k=0; k<m_ar; k++)
      temp = pha[i * m_ar + k]
      #pragma omp for
      for (j=0; j<m_br; j++)
         phc[i*m_ar+j] += temp * phb[k*m_br+j]
```

***Fig. 5**: Nested loops of the second parallel version of line matrix multiplication*

4. Performance Metrics

The tests were executed in a faculty computer equipped with an *Intel Core i7-9700* processor, which has 8 cores (1 thread each) and a single socket, along with a base frequency of 3 GHz (minimum of 0.8GHz and maximum of 4.7GHz). It runs Ubuntu 22.04.05 with kernel 6.5.0-15-generic and has 16 GB of RAM memory. The cache lines are 64 bytes long and the cache can hold 12 MB of data.

To collect information regarding the number of cache misses and floating operations, we used *PAPI*, in version 7.1.0.0. This required the compilation of the C++ using the *-lpapi* flag. We also used the *-O2* flag, for optimization, and when testing the multi-core algorithms, *-fopenmp* was also used. The version of the g++ compiler was 11.4.0. For the Java program, we used the javac compiler in version 21.0.5.
For the purposes of testing our algorithms, we extracted the following performance metrics: execution time, L1 and L2 cache misses (using *PAPI_L1_DCM* and *PAPI_L2_DCM*) and the number of floating operations performed, the latter using *PAPI_DP_OPS*. This was used instead of *PAPI_FP_OPS* because we always use the *double* data type in our code.

Using the number of floating operations, we calculated the *MFLOPS* (millions of floating operations per second) of the parallel implementations, dividing it by the elapsed time multiplied by 10^6. Additionally, the execution times allowed us to determine the speedup (Serial Execution Time / Parallel Execution Time) and efficiency (speedup / no. of cores) of the multi-core versions.

5. Results and Analysis

The first thing we did was compare the performance of the standard and line multiplications, both in C++ and Java. While both languages are highly efficient, C++ still manages to be slightly faster when executing the algorithms. This is due to the fact that while C++ compiles directly to machine code, Java is

compiled to bytecode, which is interpreted by JVM (Java Virtual Machine) at runtime. This adds an extra performance overhead, even though the JIT (just-in-time) compiler performs optimisation while running the program. It's important to point out that we used the same row-major array representation in both languages, in order to make the comparison more reliable.

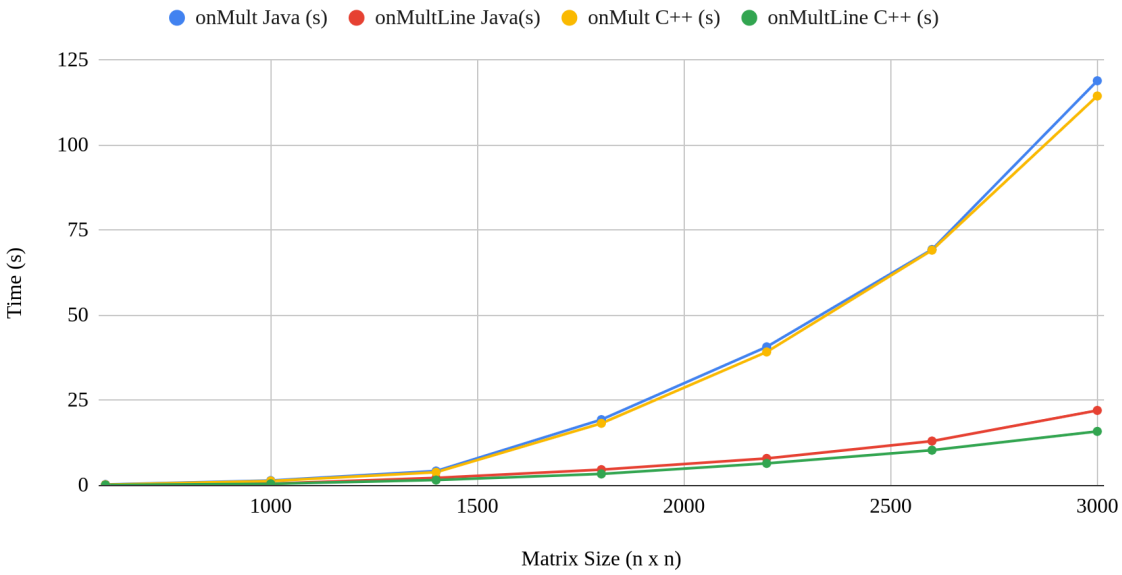

***Fig. 6**: Execution times for standard and line algorithms in C++ and Java*

Unsurprisingly, the execution time and cache misses were lower in the line algorithm. The arrays for the matrices are stored in row-major ordering, and this version traverses matrix B through its rows, therefore sequentially, which is much more cache-friendly and efficient. When we iterate over its columns, every time we get a cache miss when reading a value, we store a cache line, which however is not fully utilized when jumping to the next value in the column (due to the row-major order). This results in a much higher number of cache misses, and therefore higher execution time for the standard multiplication.

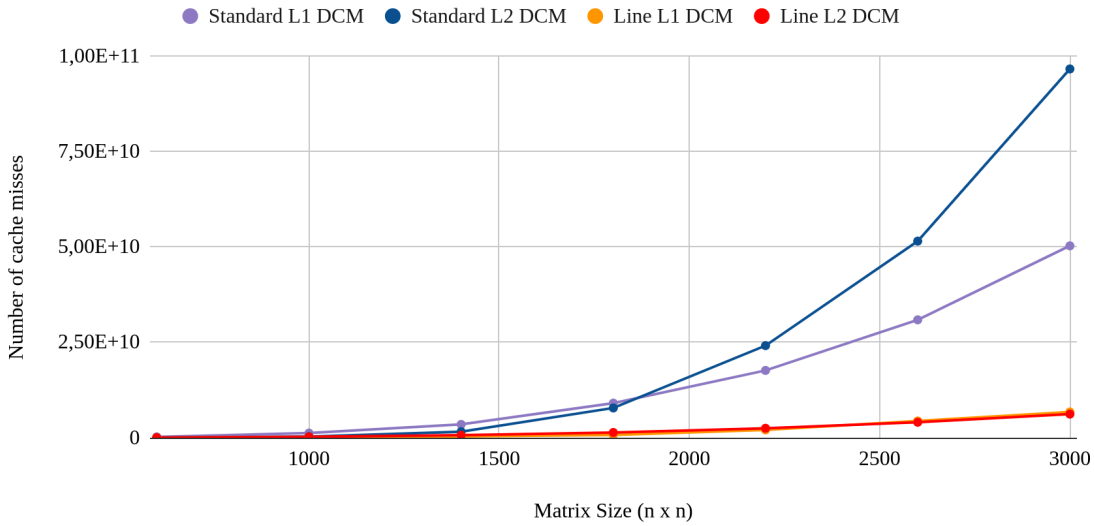

***Fig. 7**: Level 1 and Level 2 cache misses for standard and line algorithms*

The block multiplication algorithm managed to be even more efficient than the line algorithm, resulting in fewer cache misses and a lower execution time. This happens because, since they are smaller, blocks can fit more efficiently in the cache, minimising the need to fetch data from the slower main memory. As to which block size is more adequate, we can see that the number of cache misses decreased as the block size increased, making larger blocks more desirable, as long as they can still fit in the cache. A block of size 512 uses 512*512*8 = 2097152 bytes of memory, about 2 MB, which can still easily be kept in the 12 MB cache.

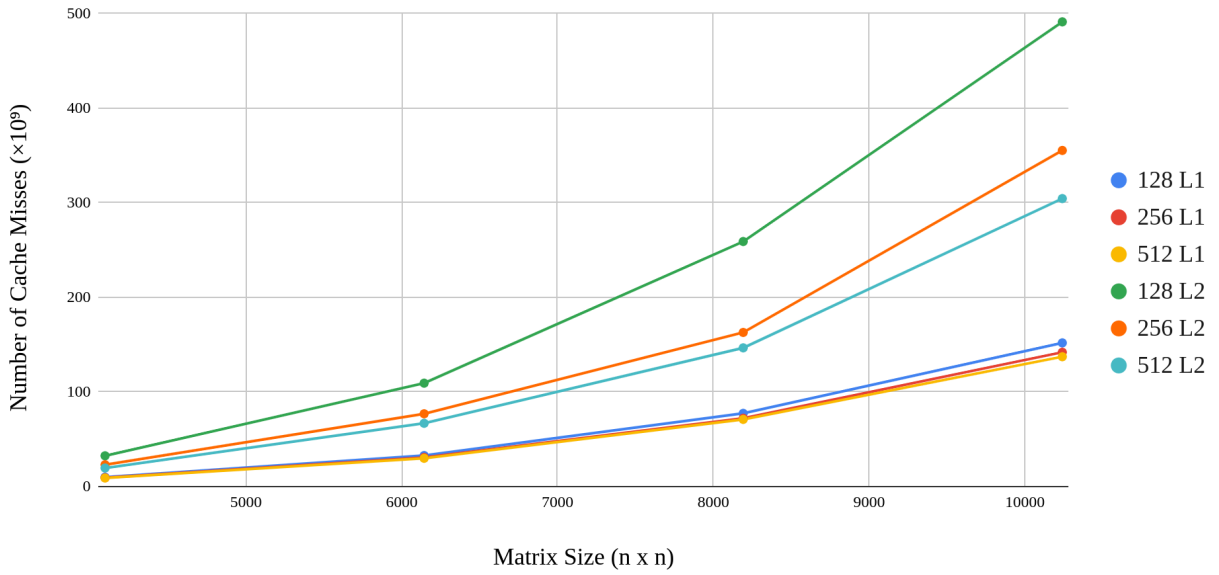

***Fig. 8**: Level 1 and Level 2 cache misses for block algorithm, with block sizes 128, 256 and 512*

Moving on to the parallel algorithms, we started by comparing them with the single-threaded line multiplication, by calculating the speedup and efficiency (as mentioned in section 4). As expected, the multi-threaded versions performed better.

Matrix size	Speedup (v1)	Efficiency (v1)	Speedup (v2)	Efficiency (v2)
4096	4.97	62%	1.17	15%
6144	4.32	54%	1.30	16%
8192	3.99	50%	1.29	16%
10240	3.56	45%	1.36	17%

***Fig. 9**: Speedups and efficiencies of parallel versions in relation to single-threaded algorithm*

As we can see, however, the first version of the parallel implementation is noticeably faster and more efficient than the second version. In the former, each thread independently handles a distinct row of the result matrix, which reduces the likelihood of memory contention. But in the latter, only the work of the inner loop is parallelised, meaning that threads may update the same row concurrently, causing more memory conflicts and slowing down calculations. This extra overhead reduces the number of MFLOPS, making the second version less efficient.

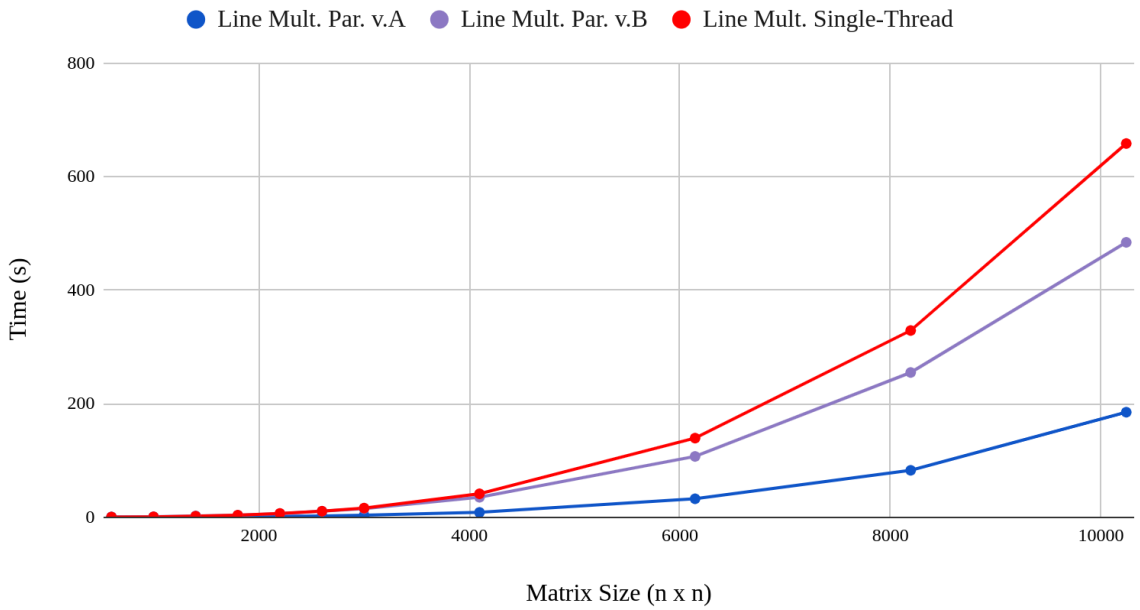

***Fig. 10**: Execution time of the parallel algorithms and the line multiplication algorithm*

6. Conclusions

Our experiments showed that optimising memory access patterns and using multi-threading significantly improve matrix multiplication performance. The line multiplication algorithm was much faster than the standard one due to better cache usage, and the block multiplication algorithm further reduced execution time by keeping data in the cache longer. Parallel implementations also boosted performance, but the first version was faster because it split the work more evenly across threads, while the second version was slower because the threads had to wait for each other more often. Therefore, we learned that choosing cache-friendly algorithms and properly balancing workloads in multi-threading are key to developing computationally efficient pieces of software.

Additional results and graphs can be found in GitLab (results.txt and results_java.txt), as well as in the following spreadsheet: CPD - Resultados Assignment 1.