Parallel and Distributed Computing

U. PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Project 1: Parallel Computing

3LEIC02, Group 13
Duarte Gonçalves
Félix Martins
Marco Vilas Boas

Problem Description

In this project, we studied the performance of different matrix multiplication algorithms, using single core computing and multicore computing approaches. Using the Performance API (PAPI), we can study the effect of the memory hierarchy in the overall performance of the program.

The matrix multiplication problem has a temporal complexity of $O(N^3)$, where N is the size of a square matrix. The number of operations required to complete the matrix multiplication is the same for the 3 algorithms presented in this project. However, different algorithms access the computer's memory in different ways, which will lead to (significant) differences in performance, because of the memory layout and hierarchy. Basically, some algorithms are more *cache-friendly* than others.

Algorithms explanation

Naive Matrix Multiplication

In this algorithm, we multiply the two matrices exactly like the mathematical definition tells us to: each entry (i, j) in the result matrix is calculated by multiplying the i row of the first matrix with the j column of the second matrix. This is very simple to implement, however, since matrices are stored in memory row by row, this algorithm is very cache-unfriendly, because to access each column of the second matrix we must jump many memory locations at a time, implying that many cache misses will occur.

Line Matrix Multiplication

This algorithm is an improvement over the naive algorithm. If we contemplate the mathematical definition of matrix multiplication, we can see that, for example, for the first element in the first line of the first matrix, it must be multiplied with each element in the first line of the second matrix, and added to the correspondent place in the result matrix (which can start out as a matrix of zeros). This means that for every element in the first line of the first matrix, we may multiply it with the corresponding line in the second matrix, and add the result to the corresponding line in the result matrix. This way, we always access the matrices in a linear fashion, which is much more cache-friendly than the naive algorithm and therefore produces much better time results.

Block Matrix Multiplication

This algorithm is an improvement over the line algorithm. The idea is to divide the matrices into smaller blocks, and then multiply these blocks. This is a good idea since it allows us to multiply the blocks in a linear fashion. Additionally, it allows us to take advantage of the cache's spatial locality, because we are multiplying smaller blocks of the matrices, which are more likely to fit in the cache. This algorithm is the most cache-friendly of the three, and therefore produces the best time results.

Multicore implementations

We implemented two different multicore matrix multiplications, which are based on the Line multiplication algorithm.

In both of the implementations, there is a single fork and join of all the threads. This is good for performance, since these operations have some overhead. In fact, a naive pragma annotation inside the loops might lead to N or N^2 forks and joins, leading to a huge decrease in performance.

Another possible problem is the sharing of variables. Since our iteration counters were created before forking the threads, we had to make them private on the pragma annotation to prevent race conditions when reading variable values.

Outer loop parallelization

In the first implementation, the outer loop is parallelized. Note that we made the inner variables *j* and *k* private, since these were by default shared across the threads. **#pragma omp parallel for** divides the iterations of the outer loop across the processors available, by creating a thread for each processor.

Inner loop parallelization

In the second implementation, the inner loop is parallelized. We made the other variables *i* and *k* private, since they are by default shared. Even though we are only parallelizing the innermost loop, we cannot just annotate it directly with **#pragma omp parallel for**, since that would lead to N^2 joins and forks. So, we need to use **#pragma omp parallel** to create a group of threads just once, which will then execute a group of instructions. The other annotation, **#pragma omp for**, divides the inner loop across the already created threads. This avoids creating and joining the threads lots of times. However, it is expected that this implementation yields worse results in comparison to the previous one. This is because **#pragma omp for** forces synchronization between all threads, which basically means that all of them must be at the start of the innermost loop for any to enter it.

Determining the number of threads

Since we compiled the program with the flag **-fopenmp**, we know that the number of created threads will be equal to the number of processors available, which we can access with **omp_get_num_proc()**. We also verified that the threads used match this number by showing the thread numbers inside the loops. In FEUP's computers, the number of available cores always turned out to be 8.

Performance metrics

For performance metrics we measured various aspects of our program. The first, and most obvious, is the time of execution: it is a direct measure of how well the program performs. We used C++ **<chrono>** tools and Go's **time** package.

Another important measure that gives many insights on the program behavior is the L1 data cache misses. As we know, having a cache friendly program/algorithm is crucial for performance, as the access to cache (especially L1) is really fast. On the other hand, when a cache miss occurs, it is very costly and slow to get the data needed from the next cache level. This takes us to the third metric: L2 cache misses. If a L1 cache miss occurs, we may still find the data in the L2 cache, which is not ideal but also not bad. However, when an L2 cache miss occurs, we pay an even higher price for main memory data access.

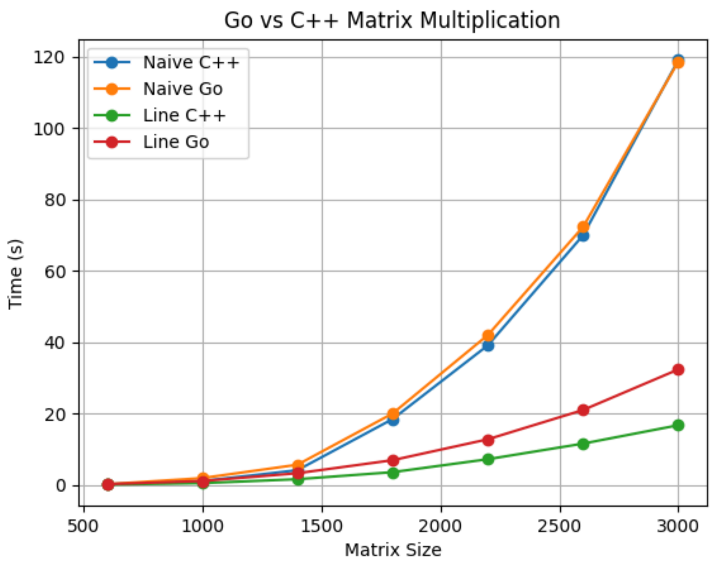
The final measures we present were not directly taken. They are derived from the time measurements. The first is *MFlops/s* (Millions of Floating Point Operations per second). As the name suggests, this measures the amount of arithmetic operations on floating numbers our CPU could make, on average, in a second. By operations we are talking about addition and multiplication.In the multicore implementations, we also derived the speedup and efficiency, which are given by:

$$Speedup = T_{single\ core}/T_{multicore}\ ;\ Efficiency = Speedup/NumberThreads$$

Note that we obtained all of our data using FEUP’s computers.

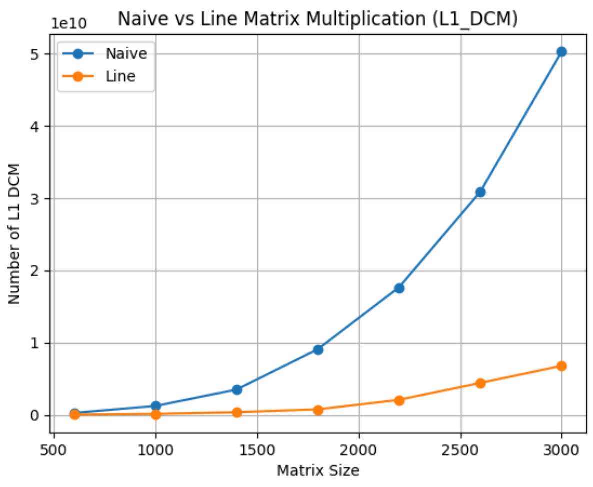
Results and analysis

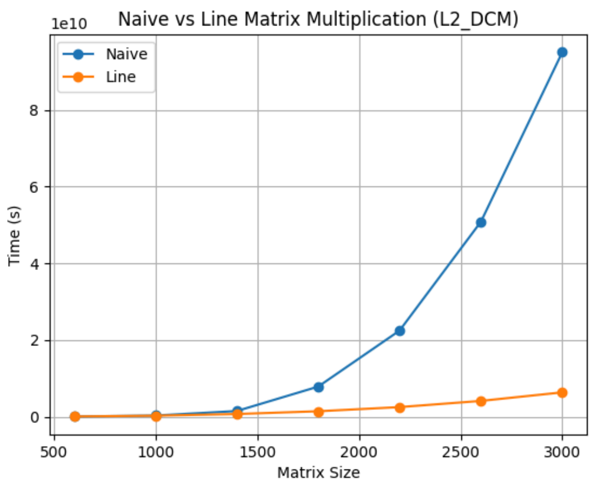
C++ and Go Implementations



When analyzing the difference between the C++ and GO implementations, we notice that, as expected, C++ holds a small advantage over GO. This difference is more noticeable for the Line multiplication algorithm. We can also use this graphic to visualize the difference in performance between the line matrix multiplication algorithm and the naive implementation. The naive version leads to, as we will see later on, unnecessary cache misses, which undoubtedly take a toll on performance.

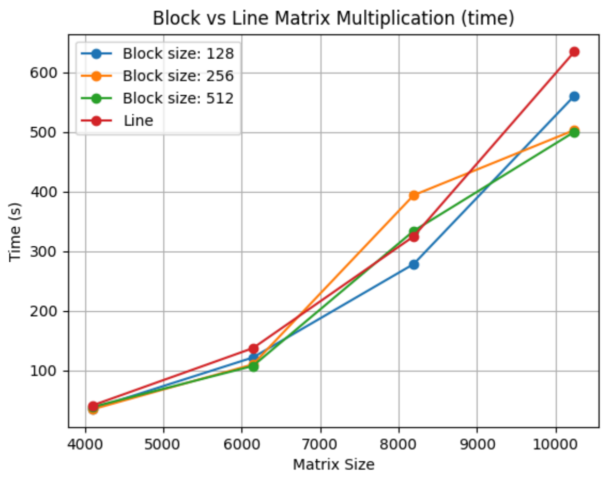
Naive and Line matrix multiplication





Indeed, as explained in the previous section, the line multiplication leads to a much lower number of cache misses, which will obviously lead to a better overall performance. This version was obtained by simply changing the order of the loops in the naive implementation. The key difference is that we are accessing the memory in a linear fashion, making much better use of the data locality property upon which caches rely on.

Line and Block matrix multiplication

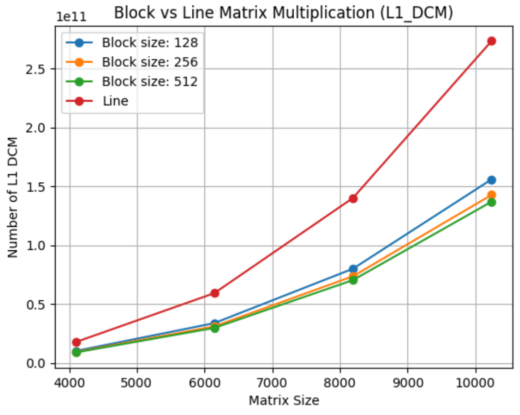


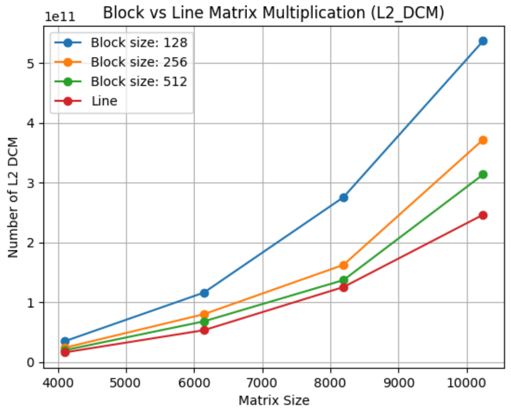
When comparing block against line matrix multiplication, we can see that the time measurements share their order of magnitude. Although this is the case, block matrix multiplication is favored for bigger sized matrices.

Despite the expectation that block matrix multiplication would consistently outperform line matrix multiplication due to its tendency to have fewer cache misses, this is not always the case. Actually, line matrix multiplication does not inherently result in a higher number of cache misses compared to the block version.

The reason for this lies in the behavior of both methods with respect to cache utilization. While block matrix multiplication generally reduces L1 data cache misses, it doesn't always guarantee a performance advantage. On one hand, in scenarios where a row of the matrix becomes sufficiently large, line matrix multiplication can lead to more L1 cache misses. On the other hand, while block matrix multiplication may excel in reducing L1 cache misses, it does not necessarily optimize cache usage at the L2 level, where line matrix multiplication holds an edge.

As a result, the superiority of one method over the other is not unequivocally established, with the effectiveness of each dependent on the specific characteristics of the matrices and the cache architecture.





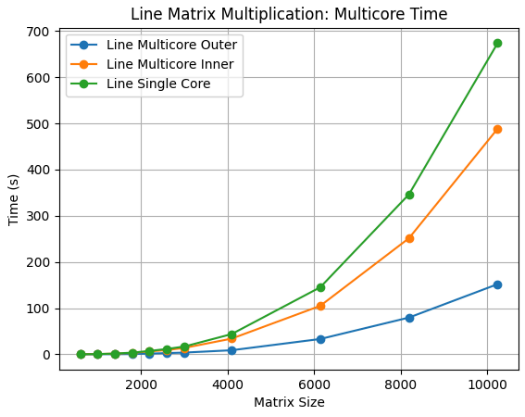
In the graphic at the left, it's apparent that L1 data cache misses are more common on the Line matrix multiplication algorithm. Opposingly, on the right, we conclude this algorithm has the least L2 data cache misses, although with a smaller disparity.

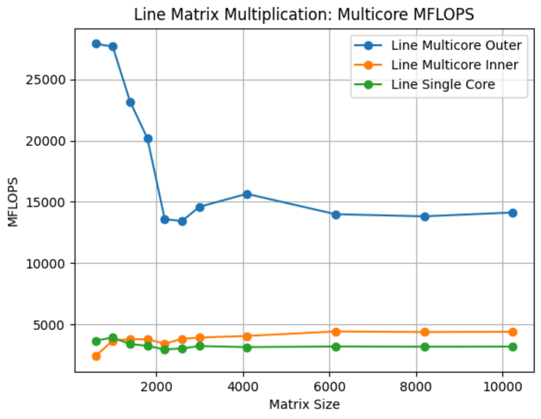
This discrepancy can be attributed to the fact that when dealing with sufficiently large matrices, a single row may exceed the capacity of the L1 cache, leading to L1 cache misses. However, this same row can still completely fit within the L2 cache, resulting in fewer L2 cache misses for this algorithm.

This characteristic contributes to a close comparison between the two methods. In fact, block matrix multiplication causes less L1 cache misses because of its reliance on smaller matrices and, consequently, rows. Additionally, dividing the matrix into smaller blocks prevents complete rows from being stored in the L2 cache. This happens because we only advance to the next segment of a row after computing the product of two smaller matrices. This will subsequently contribute to a higher frequency of L2 cache misses.

Another interesting take away from these graphics is that the smaller the block size, the more L2 cache misses. In fact, higher block sizes resemble the line matrix multiplication, which can be viewed as performing block matrix multiplication with a large block size.

Multicore analysis

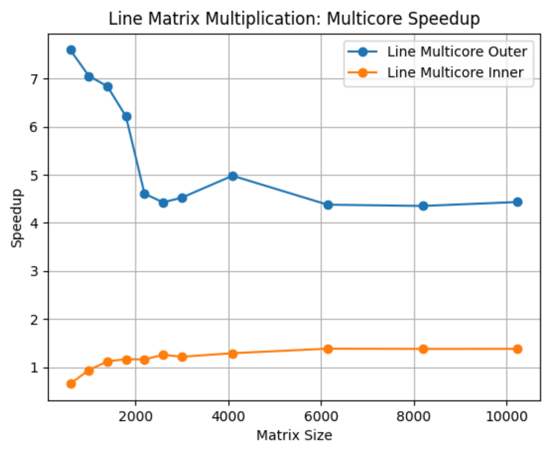


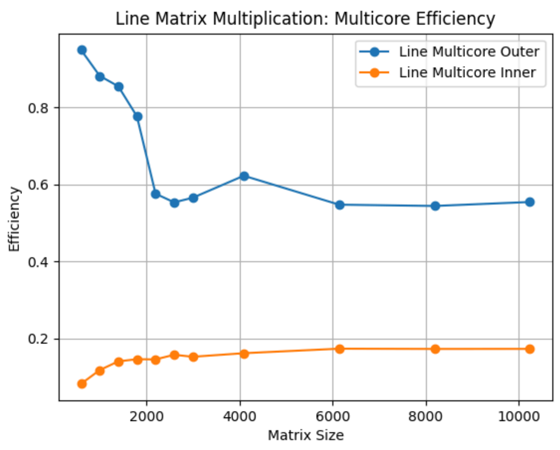


As expected, the parallelization of the outer loop leads to better performance. While the inner loop is still better than the single core implementation, especially for bigger matrices, it is quite the waste of processing power when compared to the outer loop parallelization. In fact, as explained before, the synchronization at the pragma annotation of the inner loop leads to a relatively big performance toll.

However, in the outer loop parallelization, our data shows a really big downfall of the MFlops measure around the 2000 matrix size. False sharing is definitely not the issue since it would cause performance to be worse for smaller matrices and not better.

Therefore, the performance deterioration on bigger matrices might be caused by the interaction of the many threads in memory accesses. Obviously, the single core implementation has no problems related to shared caches and memory, but the multithreading implementation could have troubles with this, especially after reaching a certain matrix size. The L2 cache is shared among all cores, thus a large enough matrix size might explain this phenomenon, since we cannot save all locally important values in faster cache(s).





The speedup and efficiency graphics also showcase this effect. Nonetheless, the performance of the outer loop parallelization is clearly above the inner loop parallelization.

Conclusions

Among the algorithms tested, it was found that the Line multiplication algorithm generally outperformed the Naive approach due to its better cache utilization, while the Block multiplication algorithm performed better than the Line multiplication algorithm on large matrices. The most important conclusion from this analysis is that slight variants of the same algorithm may produce really different performance results, in this case because of the different ways they access memory.

In the second part of the project, we analyzed two different ways of parallelizing the Line multiplication algorithm and their respective nuances. After learning how to correctly specify a parallel program without unnecessary overhead, we were able to conclude and deduce what possible inefficiencies could be happening given the performance results obtained.