Parallel and Distributed Computing

1st Project - Parallel Computing

Afonso Dias (up202006721)
André Santos (up202108658)
Pedro Beirão (up202108718)

March 2024

1 Introduction

In this project, we studied the effect on the processor performance of the memory hierarchy when accessing large amounts of data. Matrix multiplication is frequently employed to analyse the impact of the memory hierarchy on processor performance due to its data-intensive nature, representative memory access patterns, and cache behaviour. Using the Performance API (PAPI) we collected relevant performance indicators of the program execution, mostly the L1 and L2 cache misses. It is also important to mention that the processor used for this measurements was an intel core i7-9700 with a frequency of 3Ghz, 8 cores and 8 threads, an L1 cache of 32KB for instructions and another chunk of 32KB for data, an L2 cache of 256KB, and a shared L3 cache of 12MB.

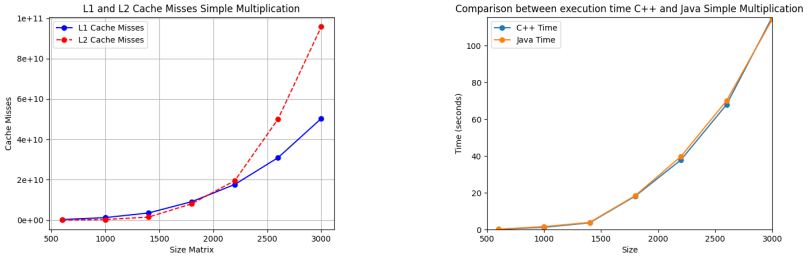
2 Performance Evaluation of a single core

This first part was divided into 3 different algorithms, which are basic matrix multiplication, line multiplication and block multiplication, were we looked to evaluate the performance of a single core using sequential programming.

2.1 Basic Matrix Multiplication

This is the most straightforward method, where each element of the resulting matrix is computed independently by taking the dot product of the corresponding row of the first matrix and the corresponding column of the second matrix.It involves three nested loops to iterate over the rows and columns of the matrices. While simple to implement, it has a time complexity of O(n3), where n is the size of the matrices, making it inefficient for large matrices.

We were asked to register the time and the L1 and L2 cache misses, using the already given algorithm written in C/C++ and writing it also on another language of our choice, we chose Java.The matrix sizes testes were: 600x600 to 3000x3000 elements with increments in both dimensions of 400.



(a) Cache Misses on naive method

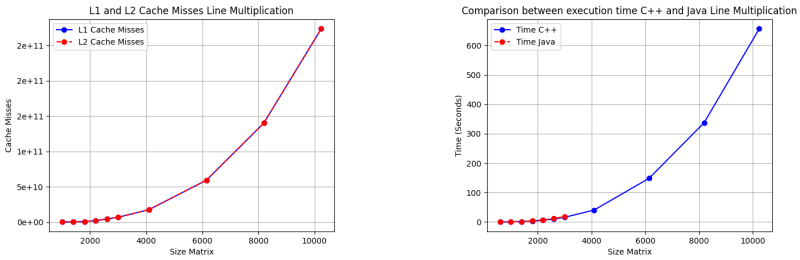
(b) Java vs C/C++

Figure 1: Results of the naive implementation

With these results we could conclude that the execution time between C/C++ and Java were mostly the same through out all the input matrices. Were we notice a big difference is between the recorded L1 and L2 cache misses on the C/C++ implementation, where we observe that the cache misses between the two caches are mostly the same till about a size of 2200, more or less, after that size we notice the gap between the cache misses increasing. This is a result of the cache size itself where the L1 cache is much smaller than the L2 cache, a difference of about 224Kb,per core, and as such the L1 cache can't handle the bigger input size matrices as well as the L2 cache.

2.2 Line Matrix Multiplication

This method optimizes the memory access pattern by traversing matrices differently to exploit locality of reference, which can improve cache performance. Instead of accessing the elements of the matrices in row-major or column-major order, this method traverses them in a linear fashion, accessing consecutive elements within a cache line. It can offer performance improvements over the simple method, especially for large matrices, by reducing cache misses and improving memory bandwidth utilisation. The time complexity remains O(n3), but it can be more efficient in practice due to mention better memory access patterns.



(a) Cache Misses on line method

(b) Java vs C/C++

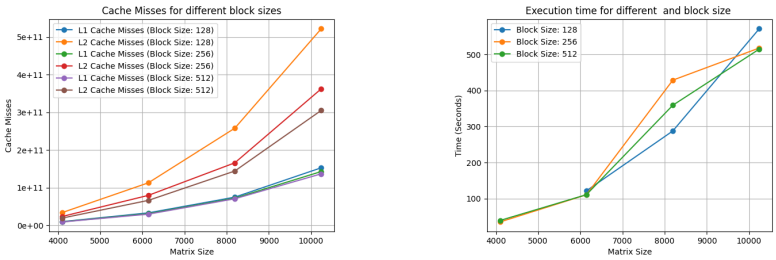
Figure 2: Results of the line implementation

As this algorithm is an improvement of the first one the can see relative better times, the time is less than 100 seconds for an input matrix of 4000 whereas the first algorithm took about that time to process a matrix with a size of 3000, and also much fewer cache misses as the gap between the L1 and L2 cache is mostly none. This is due to the aforementioned improvements above.

2.3 Block Matrix Multiplication

This method divides the matrices into smaller sub-matrices or blocks and performs matrix multiplication on these smaller blocks. By working with smaller blocks, it can improve data locality, reduce cache misses, and exploit parallelism

at a finer granularity. It involves recursive decomposition of the matrices into smaller blocks until they are small enough to use the simple matrix multiplication algorithm efficiently. Block matrix multiplication can significantly reduce the constant factors involved in the time complexity and can lead to better performance, especially on modern computer architectures with multiple levels of cache and parallel processing capabilities. Its time complexity is also O(n3), but it tends to have better performance in practice, particularly for large matrices, due to improved cache utilisation and parallelism.



(a) Cache Misses on block method

(b) C/C++ Time

Figure 3: Results of the block implementation

Analysing the results of this method was interesting because on we noticed that for our sample size at about a size 8000 the block that took the longest was the block with size of 256 followed by the one with 512 and lastly with the lowest time was the one with 128, this abnormality rapidly disappears as the time for the 128 block exponentially increases leaving the 512 and 256 sized blocks to be the best at a very large input matrix size at about 10000. With regards to the cache misses we observe that the higher the block size the lower is the number of the L2 cache misses and the L1 cache misses stay mostly the same for the tested block and matrices sizes. Overall this method shows the best results in terms of execution time than the others, mostly for large input matrices, where we can notice the most difference.

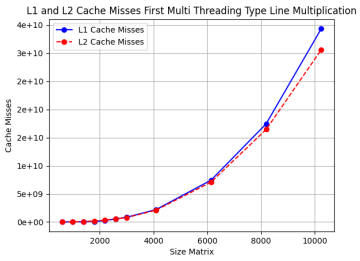
3 Part 2: Performance evaluation of a multi-core implementation

In this second part we were tasked to elaborate on the line multiplication method by implementing a parallel version of that algorithm using OpenMP (Open Multi-Processing), which is an API (Application Programming Interface) that supports shared-memory parallel programming in C, C++ (the language used in this evaluation), and Fortran. . This is where we start to take advantage of the multiple core/threads the processor has and start splitting the operations between them. We analysed two different versions of this algorithm, one using **#pragma omp parallel for** and the other using both **# pragma omp**

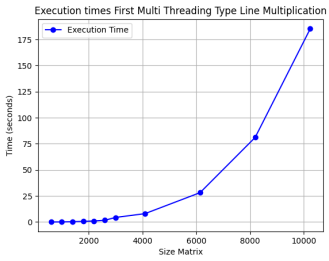
parallel for and # **pragma omp parallel for**, these are directives from the OpenMP API. Also there was a difference in this second part on the recorded time where we used the wall clock instead of a normal time function because when measuring the execution time of parallel programs, using a wall clock function provides a more accurate and comprehensive measurement of the total elapsed time, taking into account factors such as thread scheduling, load balancing, synchronization, and coordination across multiple cores or processors. We know also took into account to measure the effiency, MFlops and speedup because they provide insights into how effectively resources are utilised, how much performance improvement is achieved through parallelism, and the computational capabilities of hardware devices.

3.1 #pragma omp parallel for

In this version, the entire set of nested loops is parallelised using a single #pragma omp parallel for directive. This directive creates a team of threads, and the iterations of the outermost loop (i) are divided among these threads using automatic workload distribution. Each thread executes a portion of the outer loop, and within each iteration of the outer loop, the inner loops (k and j) are executed sequentially by each thread.



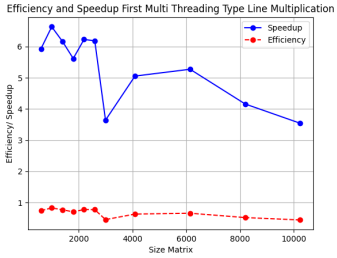
(a) Cache Misses



(b) Execution Time

We can already say that this method provides execution times 4 times better than the single core implementation as with a matrix with size 10000 we observe a time of 175 seconds and with the single core implementation it took almost 700 seconds. The same applies to the cache misses where the order of magnitude of the cache misses for both L1 and L2 for the single core implementation already surpasses by one obtained in the parallel implementation by 1, for example the first implementation makes about 5e+10 cache misses for both caches with an input matrix of 6000 for the same size matrix the parallel version makes less than 1e+10.

We are able to infer that the efficiency has almost perfect scalability, meaning that the parallel program achieves a linear speedup with the number of processors, for sizes between 600 and 3000 but then decrease significantly and maintains almost the same efficiency at about 0.5 to 0.6 and then continually to decrease as the size of the matrix increases. We can correlate the efficiency and



(a) Efficiency and Speedup

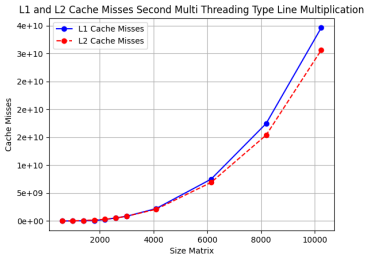
(b) MFlops

Figure 5: Results of the first parallel method

speedup as the lines drawn in the graph of the two are very similar, meaning we observe the same phenomena as the the already discussed efficiency. The mflops as we can see form a similar pattern as the other metrics where there is a big spike of performance with the size of about 1000 then it abruptly falls down very noticeable in the 3000 matrix size, goes up and then its a steady performance loss from 4000 onward.

3.2 #pragma omp parallel and #pragma omp parallel for

In this version, the parallelism is applied differently. The outer loop (i) is parallelised using #pragma omp parallel, which creates a team of threads. Inside the parallel region, the iterations of the outer loop (i) are divided among the threads, and each thread executes a portion of the outer loop independently. Within each iteration of the outer loop, the middle loop (k) is executed sequentially by each thread, as there is no parallelism directive applied to it. The innermost loop (j) is parallelised using #pragma omp for, which divides the iterations of this loop among the threads within the parallel region.

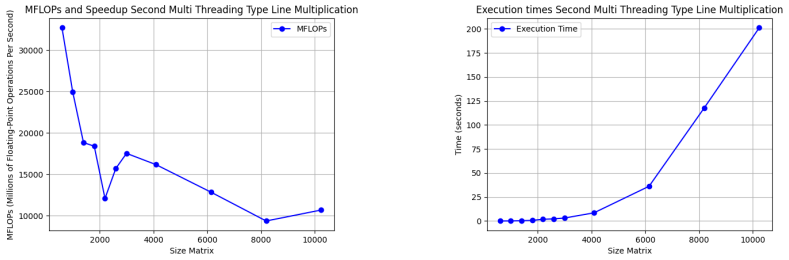


(a) Cache Misses

(b) Efficiency

The cache misses results are mostly the same between this second parallel implementation and the first parallel implementation, the main difference is that the L1 and L2 number of cache misses start to deviate first on this second

implementation. The same doesn't apply to the speedup, where there is a notable difference between the matrices of size up to 2000, where the second method can reach a speedup greater than 8 greater than what the max speedup was for the first implementation. As for the efficiency it follows mostly the same pattern as the first one for matrices of size above 2000.



(a) MFlops (b) Execution Time

Figure 7: Results of the second parallel implementation

Comparing the execution times we can see that this is implementation starts to have a greater execution from a input size of 4000 and onward. In comparison to the first implementation this one follows a more exponential loss where it starts higher with more than 30000 Mflops against the 26000 MFlops from the first method then it achieves the same drop, but this time closer to an input size of 2000 and then follows a steady decay from the 3000 input size till 8000, where it is significantly worse than the first one.

4 Conclusion

In conclusion we observed that there is a lot to gain from parallel computing when comparing to single core operations,where we see a huge improvement in execution time for higher sized matrices. Also we gained the knowledge to know how important good memory management is when programming, even when talking about sequential programming, where we notice very impactful differences in algorithms and practices that better make use of cache management.

5 References

- Intel core i7 specifications
- OpenMP reference guide