Parallel and Distributed Computing 2024/2025

Performance evaluation

1st Project

3LEIC12 Group 5

Maria João Vieira (up202204802@up.pt)
Marta Cruz (up202205028@up.pt)
Duarte Marques (up202204973@up.pt)

1. Problem description

This project explores different approaches to matrix multiplication, focusing on how memory access patterns influence execution time. By implementing and analyzing the algorithm in two distinct programming languages, C++ and Java, we assess the impact of language-specific optimizations. Furthermore, we investigate alternative computation strategies, including element-wise and block-based multiplication, and extend the study to parallel implementations using OpenMP.

Through performance measurements, this report tries to compare the different algorithmic approaches. By understanding how computation and memory work together, we can learn more about how to improve numerical operations for modern processors.

2. Algorithms

2.1. Basic Multiplication

The approach follows the traditional method of taking each row from the first matrix and computing its product with every column from the second matrix. This method results in a new matrix where each element represents the sum of products of corresponding elements from the row and column being considered. The computational complexity of this method is **$O(n^3)$**, due to the use of three nested loops. Written in C++:

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

2.2. Line Multiplication

This version of the matrix multiplication algorithm improves memory access patterns by reorganizing the loops to compute the result line by line. Instead of computing each element independently, it accumulates contributions to an entire row of the result matrix during each iteration of the inner loops. This approach can offer better cache performance on some systems, although it maintains the same computational complexity. The overall time complexity remains **$O(n^3)$**, as all three nested loops still iterate over the matrix dimensions. Written in C++:

```
for (int i = 0; i < n; i++) {
  for (int k = 0; k < n; k++) {
    for (int j = 0; j < n;  j++) {
      phc[i*n+ j] += pha[i*n+ k] * phb[k*n+ j];
```

2.3. Block Multiplication

This version of the algorithm implements block matrix multiplication, where the input matrices are divided into smaller submatrices (blocks), and the multiplication is performed block by block. This technique aims to enhance cache efficiency by increasing data locality, as smaller blocks can better fit into cache memory, reducing cache misses. The algorithm maintains the same asymptotic complexity of **$O(n^3)$**, but often results in significantly improved performance in practice for large matrices, especially when the block size is well chosen for the target architecture. Written in C++:

```
for (ii = 0; ii < n; ii += bkSize)
  for (kk = 0; kk < n;  kk += bkSize)
    for (jj = 0; jj < n; jj += bkSize)
      for (i = ii; i < min(ii + bkSize, n); i++) {
        for (k = kk; k < min(kk + bkSize, n); k++)
          double temp = pha[i * n + k];
          for (j = jj; j < min(jj + bkSize,n); j++) {
            phc[i * n + j] += temp * phb[k * n + j];
```

2.4. Parallel versions of the line multiplication

To improve the execution time of the line multiplication algorithm, parallelization using OpenMP was introduced. The parallelization is applied to distribute the matrix product calculation across different threads, leveraging multi-core processors, which results in reduced execution time.

In the first parallel version, parallelization is performed on the first loop iteration using the `#pragma omp parallel for` directive. This allows each row of the resulting matrix to be computed by a different thread, effectively distributing the workload across multiple cores. The parallelization is simple and efficient, significantly reducing execution time compared to the sequential version. Each thread computes a part of the product, maximizing the use of available cores, but without introducing significant additional complexity.

The second parallel version introduces an additional optimization by also parallelizing the innermost loop using the `#pragma omp for` directive within the loop. This allows each element of the resulting matrix to be computed independently by different threads. The inclusion of this additional parallelization aims to further improve the utilization of available cores, especially on machines with many cores. However, this also increases thread management and synchronization requirements, which may result in greater overhead compared to the previous version.

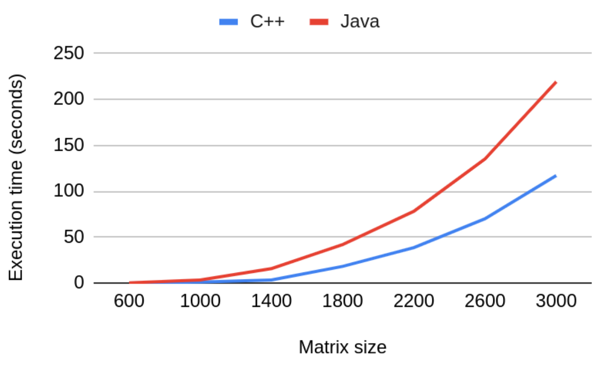
3. Performance metrics

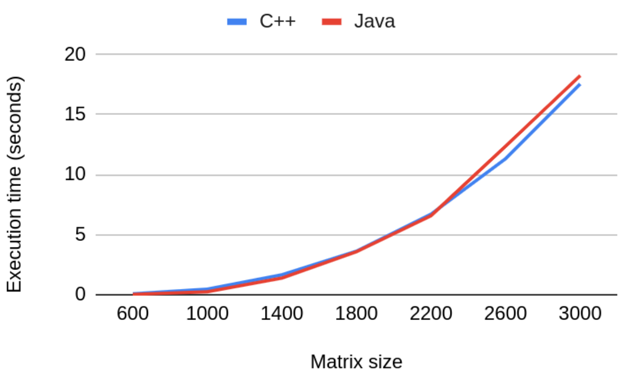
The performance evaluation of our implementations is based on three key metrics. **Execution Time** measures the duration required to complete the matrix multiplication, serving as a direct indicator of computational efficiency. **MFLOPS (Millions of Floating-Point Operations Per Second)** quantifies the processing power of each implementation, allowing for comparisons in terms of computational throughput. Lastly, **Speedup** is determined by the ratio between the execution time of the sequential and parallel implementations, reflecting the efficiency of parallelization.

4. Results and analysis

4.1. Comparison of execution times between different programming languages

Our evaluation showed that C++ consistently outperformed Java in execution time for standard and line multiplication, mainly due to lower-level memory management and better optimizations with **-O2**. C++ benefits from direct memory access and efficient cache usage, while Java's runtime and garbage collection add overhead. However, in the line multiplication, the difference was minimal, likely due to sequential memory access, reducing cache misses.





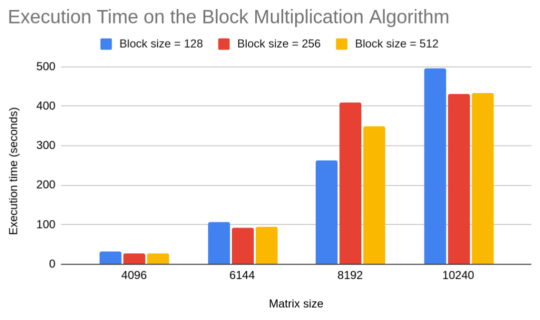
4.2. Comparison of execution times between different block sizes

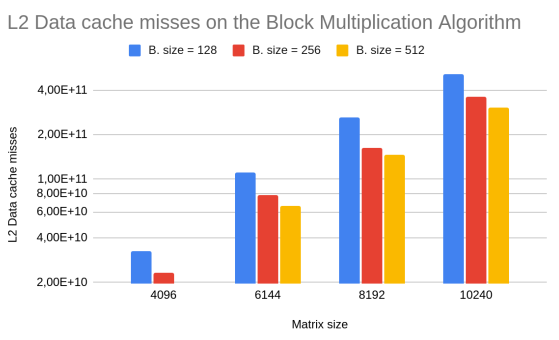
The performance analysis of block-oriented matrix multiplication shows that block size plays a key role in execution time and cache efficiency, especially with large matrices. Matrix multiplication has $O(n^3)$ complexity, causing execution time to grow exponentially with matrix size. However, the block size choice impacts performance more as the matrix size increases.

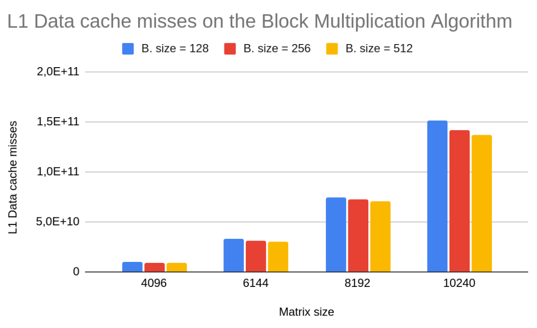
For smaller matrices, differences between block sizes are minimal, but for larger matrices, block size 512 tends to perform better than 256. This is because larger block sizes make better use of cache memory, reducing cache misses and speeding up execution.

Smaller block sizes, like 128, lead to higher cache misses, causing slower performance. This is due to inefficient cache use, requiring more memory accesses, which are slower. Cache performance data supports this, showing that larger blocks improve cache locality and reduce memory access costs.

In summary, block-oriented multiplication improves performance by optimizing cache usage. For large matrices, block size 512 strikes the best balance between cache efficiency and computational performance.





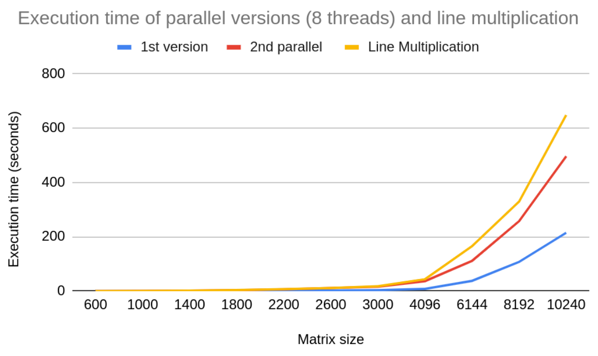


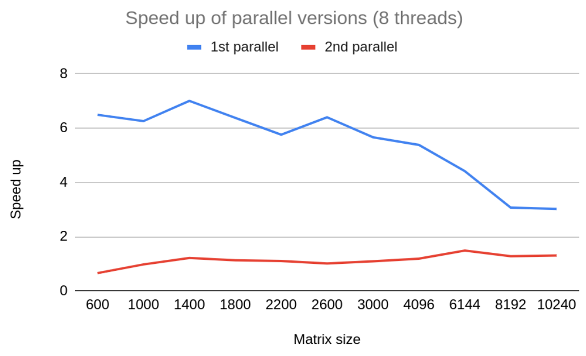
4.3. Performance Analysis of Multi-Core Implementations: Execution Time and Speedup

The performance analysis showed that the first version outperformed the second, which was faster than Basic Line Multiplication. The first version, using the #pragma omp parallel for directive, achieved the best performance by efficiently distributing the workload across multiple cores. Each thread handled a portion of the task independently, minimizing idle time and maximizing resource utilization, leading to significant execution time reduction and high speedup compared to the sequential approach.

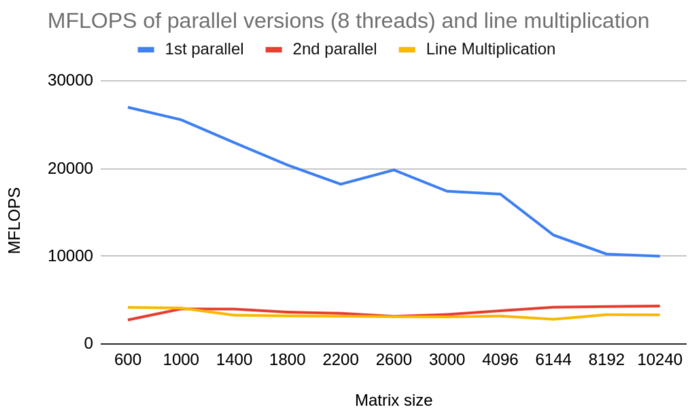
In contrast, Parallel 2, using #pragma omp parallel for for the outer loop and #pragma omp for for the inner loop, did not distribute the work as effectively. The overhead from managing multiple threads and a less optimal parallelization strategy resulted in a smaller speedup. The additional synchronization and thread management slowed performance, leading to less efficient resource use.

Line Multiplication, the sequential version, was the slowest because it did not benefit from multi-core processing, performing all operations on a single thread and resulting in longer execution times. The performance gap between Parallel 1 and Parallel 2 lies in the workload distribution and parallelization overhead. Parallel 1 minimized overhead and efficiently used available cores, while Parallel 2 incurred more overhead, reducing its speedup. Line Mult, being sequential, could not exploit parallel execution, resulting in significantly slower performance.





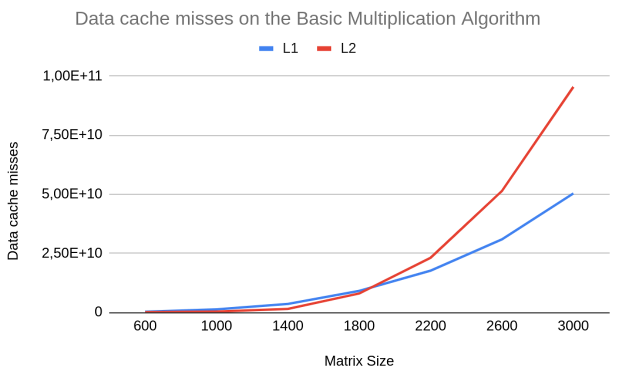
The MFLOPS graph shows that the first parallel version achieved significantly higher throughput compared to the second parallel version and Line Multiplication. This is because it better utilized multiple cores, reducing idle time and processing more operations. In contrast, the second parallel version and Line Multiplication had similar MFLOPS, as both faced higher overhead and less efficient workload distribution, leading to lower throughput.

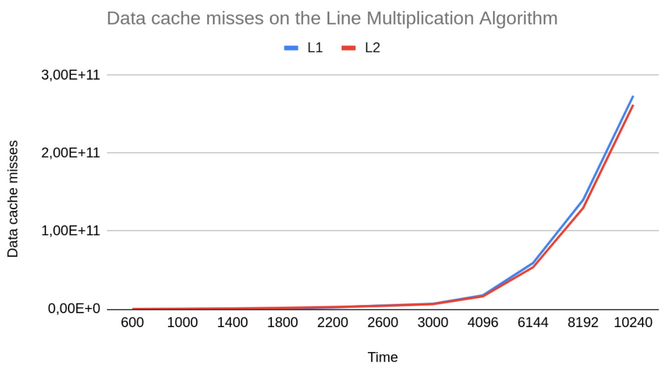


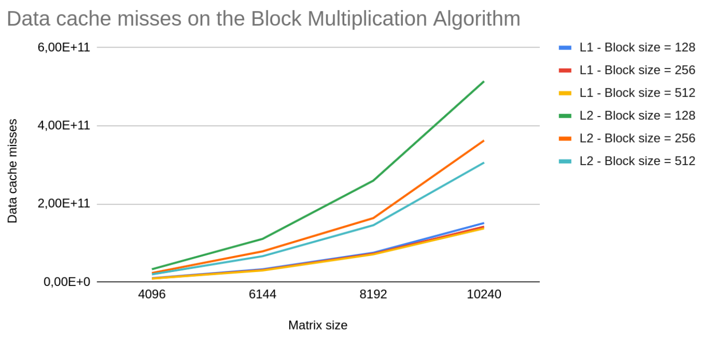
4.4 Comparison of cache misses across different algorithms

The cache misses grow significantly as the matrix size increases, with the basic and line multiplication algorithms showing similar trends. In both cases, L2 cache misses are higher than L1, indicating that data frequently moves out of the L1 cache, causing performance degradation. The exponential growth suggests that these implementations are not optimized for cache usage, leading to inefficient memory access patterns as the matrix size scales.

The block multiplication algorithm demonstrates better cache efficiency by reducing the overall growth rate of cache misses. Smaller block sizes tend to cause more L1 misses, while larger blocks shift the burden to L2 cache. This suggests that block-based approaches improve spatial locality, keeping data in the cache for longer periods. The results highlight the importance of selecting an optimal block size to balance L1 and L2 cache utilization, ultimately reducing memory bottlenecks and enhancing performance.







5. Conclusions

This project explored different matrix multiplication approaches, focusing on performance, cache efficiency, and parallelization. We learned that block size plays a crucial role in optimizing performance, with block size 512 offering the best balance for large matrices. C++ outperformed Java due to better memory management and optimizations, especially in line multiplication.

Parallelization showed that the first version was more efficient than the second, as it minimized overhead and utilized cores better. We also observed that the sequential line multiplication was the slowest, as it couldn’t leverage multi-core processing.

Overall, we gained valuable insights into how algorithm structure and parallelization can significantly improve performance, especially for large-scale computations.