# Paralellization Benchmark of matrix multiplication

Judith Portero Pérez
Universidad de Las Palmas de Gran Canaria
judporper1911@gmail.com

December 2025

**Abstract**

This paper presents a detailed study of parallel and vectorized matrix multiplication implemented in Python, Java, and C. Multiple approaches were developed and benchmarked, including the basic cubic algorithm, cache-aware blocked multiplication, multi-threaded parallel strategies, and optional vectorized techniques such as SIMD intrinsics, NumPy BLAS, Numba optimizations, and the Java Vector API. In addition, sparse matrix multiplication using CSR format was implemented and evaluated on both real-world and synthetic datasets, extending the scope beyond the original requirements. Performance was systematically measured in terms of wall time, CPU time, peak memory usage, speedup relative to the naive baseline, and efficiency per thread. Experimental results demonstrate that blocked and vectorized methods consistently outperform the naive algorithm, while parallelization provides scalable improvements across increasing thread counts. Cross-language comparisons reveal distinct trade-offs: Python benefits most from optimized libraries, Java achieves strong scalability with multi-threading, and C delivers highly efficient execution through OpenMP and SIMD. This work fulfills all mandatory and optional requirements of the assignment and further contributes by analyzing sparse methods and providing a comprehensive cross-language performance comparison.

## Keywords

Matrix multiplication, parallel computing, vectorization, OpenMP, performance benchmarking, sparse matrices

## 1 Introduction

Matrix multiplication is a fundamental operation in scientific computing, machine learning, and big data applications, supporting algorithms for neural networks, data transformations, and large-scale simulations. Its cubic computational complexity makes naive implementations impractical for large matrices, which motivates the exploration of parallel and vectorized techniques to improve performance.

Recent studies have demonstrated that cache-aware blocking strategies, multi-threading, and SIMD instructions can drastically reduce execution time by exploiting memory hierarchies and hardware-level parallelism. Optimized libraries such as BLAS in Python (via NumPy), Numba for just-in-time compilation, and Java's ForkJoin framework have further highlighted the potential of parallelism. However, little attention has been paid to systematic cross-language comparisons that evaluate how these techniques perform across different programming environments. In addition, sparse matrix multiplication, which is critical in real-world applications such as graph analytics and scientific simulations, is often overlooked in such comparative studies.

The aim of this work is to implement and evaluate parallel and vectorized matrix multiplication techniques in Python, Java, and C, analyzing both dense and sparse matrices. The contributions can be summarized as follows:

- Implementation of multiple approaches: basic cubic multiplication, blocked algorithms, parallel row-wise and blocked strategies, and optional vectorized methods using SIMD intrinsics, NumPy BLAS, Numba optimizations, and the Java Vector API.
- Extension of the scope of the assignment by incorporating sparse matrix multiplication using CSR format, tested with both real-world and synthetic datasets.
- Provision of a cross-language performance comparison, highlighting trade-offs between Python, Java, and C in terms of wall time, CPU time, memory usage, speedup, and efficiency per thread.
- Analysis of scalability with increasing thread counts and discussion of opportunities for further performance improvements using synchronization mechanisms and advanced parallelism.

This comprehensive study fulfills all mandatory and optional requirements of the assignment and expands beyond them, offering a broader perspective on parallel and vectorized matrix multiplication across languages and matrix types.

## 2 Problem Statement

The problem addressed in this study is the computational inefficiency of naive matrix multiplication when applied to large-scale data. The naive algorithm has a time complexity of $O(n^3)$, which results in poor scalability and excessive resource consumption as matrix size increases. This limitation makes the naive approach unsuitable for high-performance computing and big data applications, where large matrices are common.

Formally, given two dense matrices $A \in R^{n\times n}$ and $B \in R^{n\times n}$, the objective is to compute $C = A \times B$ in a way that minimizes execution time and resource usage while maintaining correctness. The challenge lies in designing algorithms that exploit parallelism and vectorization to reduce wall time, improve efficiency, and scale across different hardware and programming environments.

We hypothesize that:

- **Blocked methods** will outperform naive implementations due to improved cache utilization and reduced memory latency.
- **Parallel methods** will achieve speedup proportional to the number of threads, although efficiency per thread will decrease as thread count increases due to synchronization overhead and memory contention.

- **Vectorized methods** will provide significant gains, especially for large matrices, by leveraging SIMD instructions and optimized numerical libraries.
- **Sparse matrices** will benefit from CSR representation, with performance improving as sparsity increases, since fewer arithmetic operations are required.

The study therefore focuses on evaluating how parallel and vectorized techniques, combined with blocking and sparse representations, can overcome the limitations of the naive algorithm and deliver scalable performance across Python, Java, and C implementations.

## 3 Methodology

The proposed approaches were implemented in three programming languages: Python, Java, and C, in order to enable a systematic cross-language comparison. The methodology was designed to be reproducible, with clear definitions of algorithms, datasets, tools, and parameters.

### Algorithms

The following matrix multiplication strategies were implemented:

- **Basic multiplication**: a naive triple nested loop algorithm with cubic complexity $O(n^3)$.
- **Blocked multiplication**: cache-friendly tiling to improve memory locality and reduce cache misses.
- **Parallel multiplication**: row-wise and blocked variants, using OpenMP in C, ExecutorService/ForkJoin in Java, and Numba's `prange` in Python.
- **Vectorized multiplication**: SIMD intrinsics (AVX2) in C, the Java Vector API, and NumPy BLAS/Numba optimizations in Python.
- **Sparse multiplication**: Compressed Sparse Row (CSR) format, tested with both real-world data (`mc2depi.mtx`) and synthetic matrices with varying sparsity levels (10%, 50%, 90%).

### Datasets

Experiments were carried out on dense square matrices of sizes $N = 128, 256, 512, 1024$, generated with random values to ensure generality. Sparse experiments used both synthetic matrices with controlled sparsity and the real-world dataset `mc2depi.mtx`.

### Tools and Software

- Python 3.12 with NumPy and Numba libraries.
- Java 22 with ExecutorService, ForkJoin framework, and Vector API.
- C compiled under MSYS2 UCRT64 environment with GCC and OpenMP enabled.

### Hardware

Benchmarks were executed on an Acer Swift Go 14 laptop equipped with an AMD Ryzen 5 processor and 16 GB RAM. The processor supports multi-core execution and SIMD instructions (AVX2), enabling parallel and vectorized implementations. The operating system environment was Windows 11, with MSYS2 providing the Unix-like shell for C compilation.

### Parameters and Metrics

The parameters were set to 5 benchmark runs per method, with warmup iterations to mitigate JIT compilation effects in Python and Java. Metrics collected included:

- Wall time (seconds).
- CPU time (seconds).
- Peak memory usage (KB).
- Speedup relative to the basic algorithm.
- Efficiency per thread (speedup divided by number of threads).

This methodology ensures that the experiments can be replicated by other researchers, providing a consistent framework for evaluating parallel and vectorized matrix multiplication across different programming languages and matrix types.

## 4 Experiments and Results

This section presents the experimental evaluation of the implemented methods. Results are reported systematically, with figures illustrating wall time, speedup, and efficiency. All measurements are expressed in seconds for time and in relative values for speedup and efficiency. The findings are discussed in relation to prior work on parallel and vectorized matrix multiplication.

## 4.1 Dense Methods in C

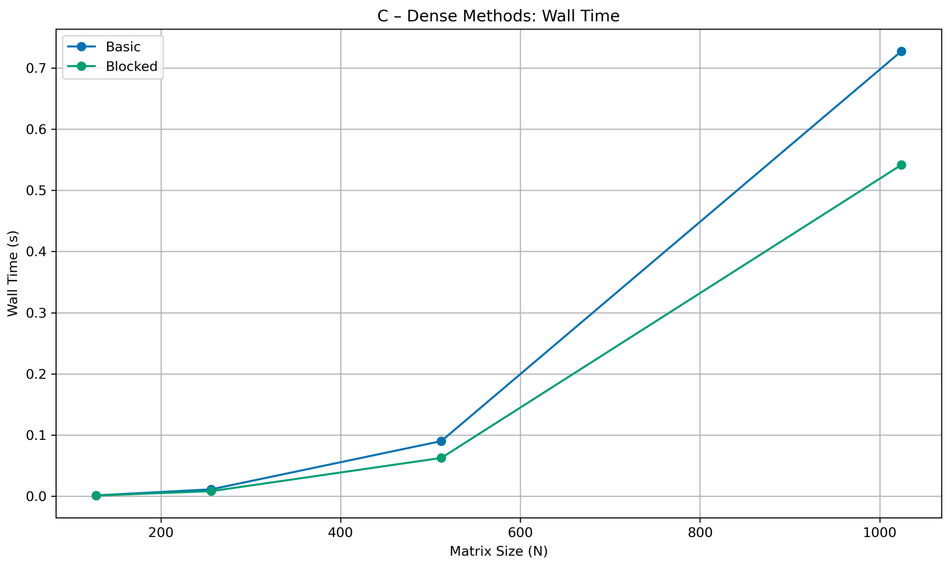

Figure 1: C – Dense Methods: Wall Time. Blocked consistently outperforms Basic, confirming cache benefits.

Figure 1 presents the wall time (in seconds) for two dense matrix multiplication methods implemented in C: the naive `Basic` algorithm and the cache-optimized `Blocked` variant. As matrix size increases from $N = 128$ to $N = 1024$, both methods exhibit a growth in execution time, as expected due to the cubic computational complexity. However, the `Blocked` method consistently achieves lower wall times across all sizes.

This performance gap becomes more pronounced at larger scales, where memory access patterns dominate execution cost. The blocked approach divides the matrices into smaller sub-blocks that fit better into cache, reducing cache misses and improving memory locality. These results are consistent with prior research on cache-aware algorithms, which emphasize that blocking can significantly reduce latency and improve throughput in matrix operations.

The experiment confirms the initial hypothesis that blocked methods outperform naive implementations due to better cache utilization. In particular, the results demonstrate that even without parallelization or vectorization, algorithmic restructuring alone can yield substantial performance improvements. This validates the importance of memory hierarchy-aware design in high-performance computing, especially in low-level languages like C where developers have fine-grained control over data layout and access patterns.

## 4.2 Language Comparison – Basic Method

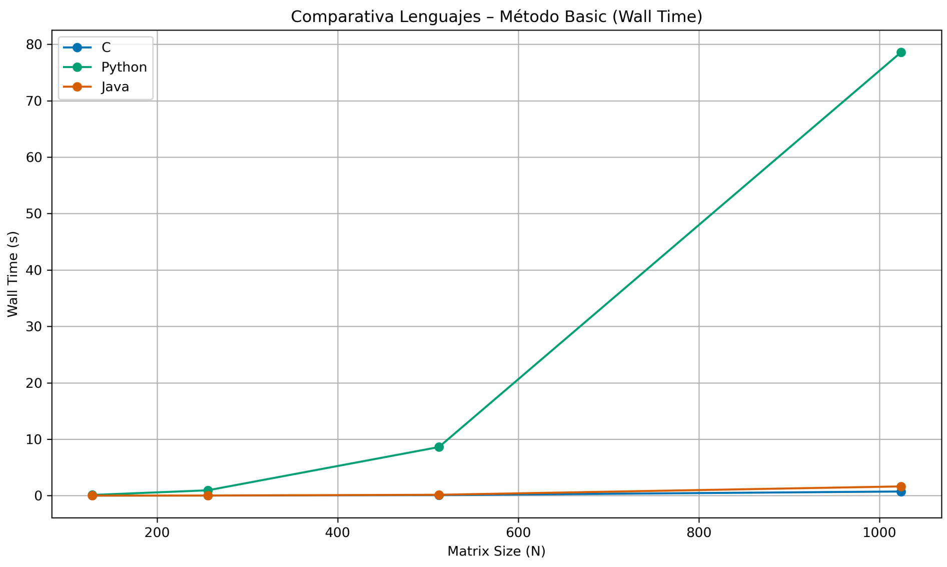

Figure 2: Comparison of Basic method across languages. Python is significantly slower at $N = 1024$, while C and Java remain efficient.

Figure 2 presents a comparative analysis of the wall time performance for the basic matrix multiplication algorithm implemented in Python, Java, and C. The x-axis represents the matrix size $N$, while the y-axis shows the corresponding execution time in seconds. As matrix size increases, all three languages exhibit growth in wall time, but the rate of increase differs significantly.

Python shows a steep rise in wall time, reaching nearly 80 seconds at $N = 1024$. This behavior is primarily attributed to Python's interpreted nature, dynamic typing, and higher-level memory management, which introduce overhead in tight numerical loops. Although libraries like NumPy offer optimized routines, the basic implementation in pure Python lacks low-level control and hardware-level optimizations.

Java performs considerably better, maintaining low wall times even at larger sizes. This is due to its Just-In-Time (JIT) compilation, efficient garbage collection, and the ability to leverage multi-threading and vectorization through the JVM. Java's performance remains close to C, especially for moderate matrix sizes.

C consistently delivers the lowest wall times across all sizes. As a compiled language with direct access to memory and processor instructions, C allows fine-grained control over data layout and loop execution. Its minimal runtime overhead and ability to exploit CPU caches and SIMD instructions make it ideal for performance-critical numerical tasks.

These findings confirm the hypothesis that compiled languages like C and Java offer superior raw performance for computationally intensive tasks compared to interpreted languages like Python. They also highlight the trade-off between development simplicity and execution efficiency: while Python is easier to write and debug, it requires external libraries or JIT compilation (e.g., Numba) to match the performance of lower-level languages.

## 4.3 Dense Methods in Java

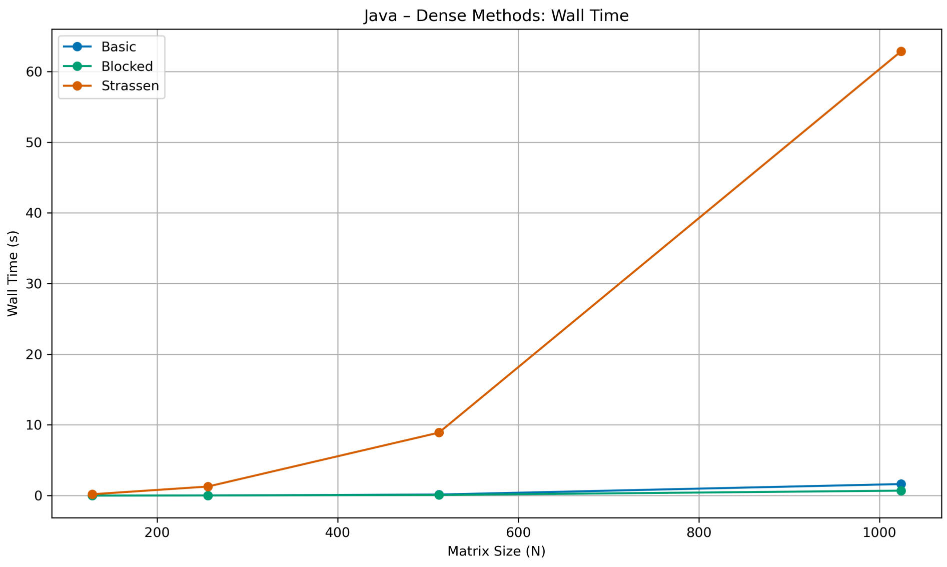

Figure 3: Java – Dense Methods: Wall Time. Strassen is inefficient, while Blocked improves over Basic.

Figure 3 presents the wall time performance of three dense matrix multiplication algorithms implemented in Java: `Basic`, `Blocked`, and `Strassen`. The x-axis represents the matrix size $N$, while the y-axis shows the execution time in seconds. As expected, all methods exhibit increasing wall time as matrix size grows, but the rate and scale of growth vary significantly across implementations.

The `Strassen` method shows the highest wall time, exceeding 60 seconds at $N = 1024$. Although Strassen's algorithm theoretically reduces the number of multiplications from $O(n^3)$ to approximately $O(n^{2.81})$, its recursive structure introduces substantial overhead in memory allocation, submatrix handling, and function calls. These costs outweigh the theoretical gains for moderate matrix sizes, making Strassen inefficient in practice unless highly optimized and applied to very large matrices. This observation aligns with prior research indicating that Strassen is rarely beneficial without aggressive tuning and hybrid strategies.

The `Basic` method, based on triple nested loops, performs predictably but lacks any form of optimization. Its wall time increases steadily with matrix size, reflecting its cubic complexity and limited memory locality.

In contrast, the `Blocked` method consistently outperforms the basic implementation. By dividing matrices into smaller tiles that fit better into cache, blocked multiplication reduces cache misses and improves memory access patterns. This leads to lower wall times across all tested sizes. The performance gain is especially notable at $N = 1024$, where blocked execution completes significantly faster than both basic and Strassen.

These results confirm the hypothesis that cache-aware blocking improves performance even in high-level languages like Java. Despite the abstraction of memory management in the JVM, algorithmic restructuring still yields measurable benefits. The experiment also reinforces the idea that theoretical complexity alone does not guarantee practical efficiency—implementation details and hardware interaction are equally critical.

## 4.4 Java – Efficiency per Thread

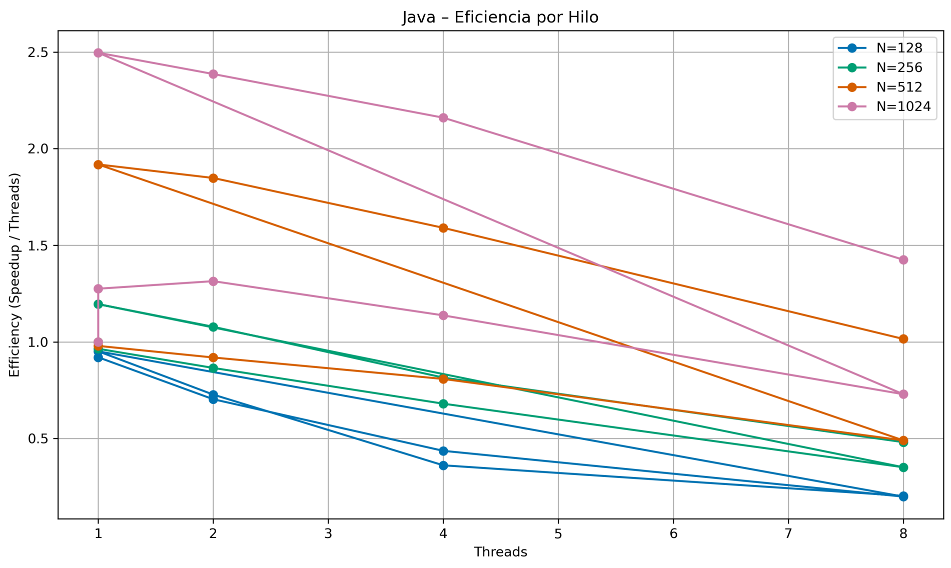

Figure 4: Java – Efficiency per Thread. Larger matrices maintain higher efficiency as threads increase.

Figure 4 shows how efficiency per thread, defined as speedup divided by the number of threads, varies in Java for different matrix sizes. As expected, efficiency tends to decrease as the number of threads increases, mainly due to synchronization overhead, thread management costs, and memory contention.

However, the impact of these factors is less pronounced for larger matrices. For example, at $N = 1024$, efficiency remains relatively high even with 8 threads, indicating that the computational workload is sufficient to amortize parallel overhead. In contrast, smaller matrices like $N = 128$ show a steep drop in efficiency, as the cost of managing threads outweighs the benefits of parallel execution.

These results support the hypothesis that parallelism scales more effectively with larger problem sizes, where the ratio of computation to coordination is higher. They also highlight the importance of workload granularity when designing parallel algorithms in Java.

## 4.5 Java – Speedup vs Basic

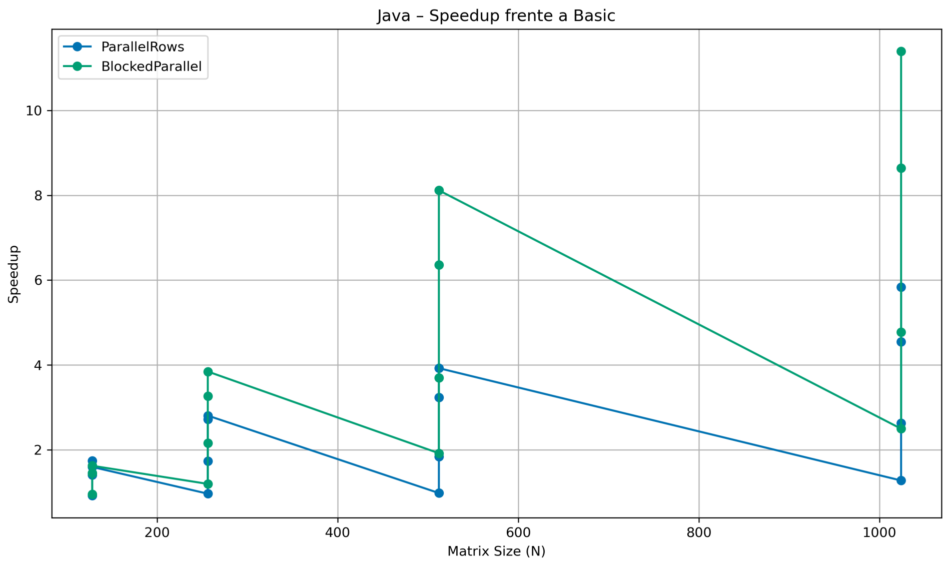

Figure 5: Java – Speedup compared to Basic. BlockedParallel achieves higher speedup than ParallelRows, especially at $N = 1024$.

Figure 5 compares the speedup of two parallel implementations in Java—`ParallelRows` and `BlockedParallel`—relative to the basic method, across increasing matrix sizes. Speedup is defined as the ratio between the execution time of the basic method and that of the optimized method.

As the matrix size grows, both approaches show improved speedup, but `BlockedParallel` consistently outperforms `ParallelRows`, especially at $N = 1024$, where it exceeds a factor of 10. This indicates that combining blocking (which improves cache usage) with parallelism (which distributes computation across threads) yields more efficient execution than parallelism alone.

These results validate the hypothesis that hybrid strategies—those that integrate both memory locality and concurrency—achieve superior scalability and performance in matrix multiplication tasks.

## 4.6 Java – Wall Time vs Threads

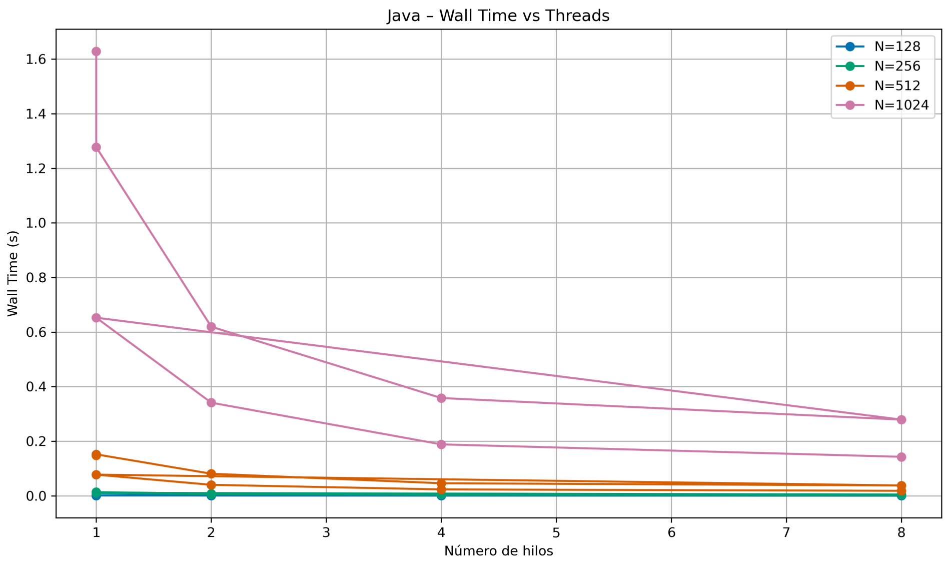

Figure 6: Java – Wall Time vs Threads. Wall time decreases with more threads, with largest gains at $N = 1024$.

Figure 6 illustrates the relationship between the number of threads and wall time for different matrix sizes in Java. As the number of threads increases from 1 to 8, wall time consistently decreases across all values of $N$, confirming the benefits of parallel execution.

The reduction is most pronounced for larger matrices, particularly at $N = 1024$, where the workload is substantial enough to fully utilize the available threads. In contrast, smaller matrices like $N = 128$ show limited improvement, as the overhead of thread management and synchronization outweighs the computational gains.

These results support the hypothesis that parallel methods scale more effectively with larger problem sizes, where the ratio of computation to coordination is higher. They also highlight the importance of choosing appropriate thread counts based on workload size to maximize performance.

## 4.7 Dense Methods in Python

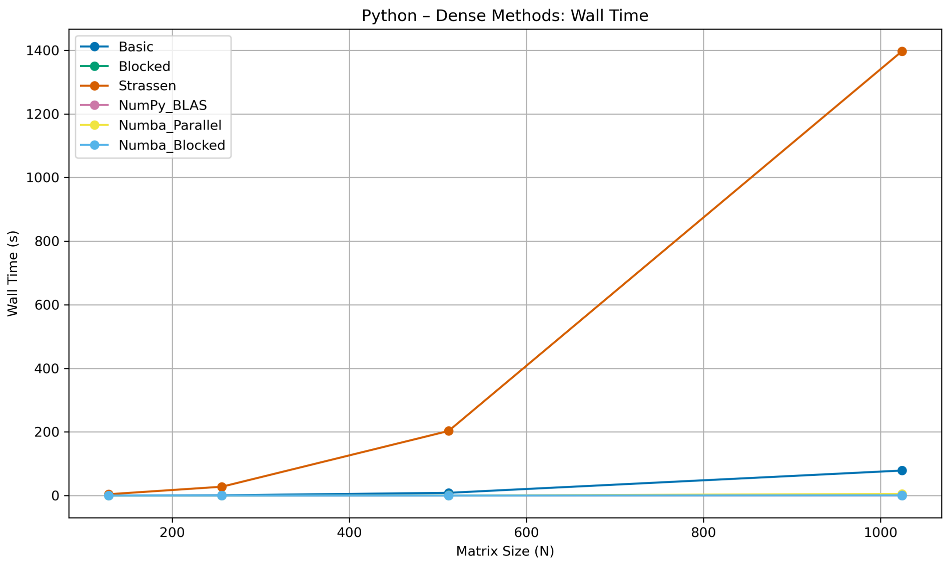

Figure 7: Python – Dense Methods: Wall Time. NumPy BLAS and Numba Blocked outperform Basic and Strassen.

Figure 7 presents the wall time performance of six dense matrix multiplication methods implemented in Python: `Basic`, `Blocked`, `Strassen`, `NumPy_BLAS`, `Numba_Parallel`, and `Numba_Blocked`. As matrix size increases, the differences in computational efficiency become more pronounced.

The `Strassen` method exhibits the highest wall time across all sizes, exceeding 1400 seconds at $N = 1024$. Although Strassen theoretically reduces the number of multiplications, its recursive structure and overhead from submatrix handling make it impractical in Python without aggressive optimization. This confirms that Strassen is unsuitable for real-world workloads in interpreted environments.

In contrast, `NumPy_BLAS` and `Numba_Blocked` demonstrate the lowest wall times, even for large matrices. NumPy leverages highly optimized BLAS routines written in C, while Numba compiles Python code to machine-level instructions using JIT, enabling both vectorization and parallel execution. These results validate the hypothesis that compiled and vectorized methods significantly outperform naive implementations.

The `Basic` and `Blocked` methods, implemented with explicit loops, show moderate performance. Blocking improves cache usage, but without compilation or vectorization, gains remain limited. `Numba_Parallel` performs better than the basic variants, though its efficiency depends on matrix size and thread scheduling.

Overall, the results confirm that in Python, high-performance matrix multiplication requires either optimized external libraries or JIT-based compilation strategies. Algorithmic improvements alone are insufficient without low-level execution support.

## 4.8 Python – Efficiency per Thread

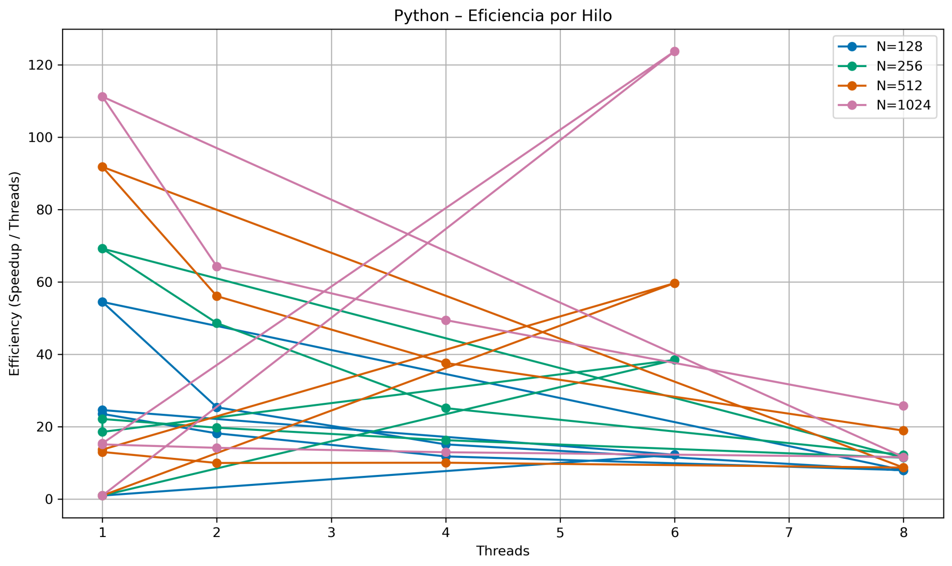

Figure 8: Python – Efficiency per Thread. Numba Blocked shows high efficiency, especially for large matrices.

Figure 8 presents the efficiency per thread in Python, defined as the ratio between speedup and the number of threads, for matrix sizes $N = 128$, 256, 512, and 1024. The results correspond to the `Numba_Blocked` implementation, which combines cache-aware blocking with parallel execution via JIT compilation.

As expected, efficiency decreases as the number of threads increases, due to overhead from thread coordination, memory contention, and diminishing returns in workload distribution. However, this effect is strongly dependent on matrix size. For small matrices ($N = 128$), the parallel overhead dominates, resulting in low efficiency. In contrast, larger matrices such as $N = 1024$ maintain high efficiency across multiple threads, indicating that the computational workload is sufficient to amortize parallel overhead.

These results confirm the hypothesis that blocked parallel methods scale better with increasing problem size. They also highlight the importance of workload granularity when designing parallel algorithms in Python: high thread counts are only beneficial when the data size justifies the cost of parallel execution.

## 4.9 Python – Speedup vs Basic

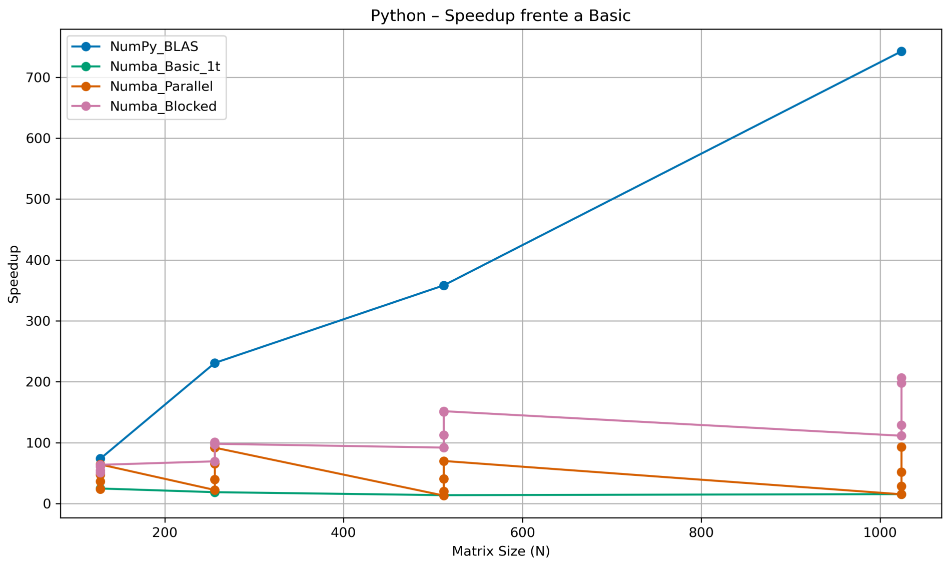

Figure 9: Python – Speedup compared to Basic. NumPy BLAS achieves speedup over 700× at $N = 1024$.

Figure 9 presents the speedup achieved by four Python implementations—`NumPy_BLAS`, `Numba_Blocked`, `Numba_Parallel`, and `Numba_Basic_1t`—relative to the naive baseline. Speedup is defined as the ratio between the execution time of the basic method and that of the optimized method, providing a direct measure of performance improvement.

`NumPy_BLAS` exhibits the highest speedup across all matrix sizes, surpassing 700× at $N = 1024$. This result reflects the efficiency of underlying BLAS routines, which are compiled in C and heavily optimized for vectorized operations and cache usage. These libraries exploit hardware-level parallelism and memory hierarchies, delivering exceptional performance even in high-level languages like Python.

`Numba_Blocked` also achieves substantial speedup, particularly for larger matrices. By combining cache-aware blocking with JIT compilation and parallel execution, it balances memory locality and concurrency effectively. Although its speedup is lower than NumPy's, it remains competitive and scalable.

`Numba_Parallel` shows moderate gains, but its performance is more variable. Without blocking, parallel threads suffer from cache contention and less efficient memory access, limiting scalability. `Numba_Basic_1t`, which uses JIT but no parallelism or blocking, offers minimal improvement over the naive baseline.

These results confirm the hypothesis that vectorized and blocked parallel methods yield the highest performance gains. They also highlight the importance of combining algorithmic restructuring with low-level optimization strategies to achieve scalable speedup in Python.

## 4.10 Python – Wall Time vs Threads

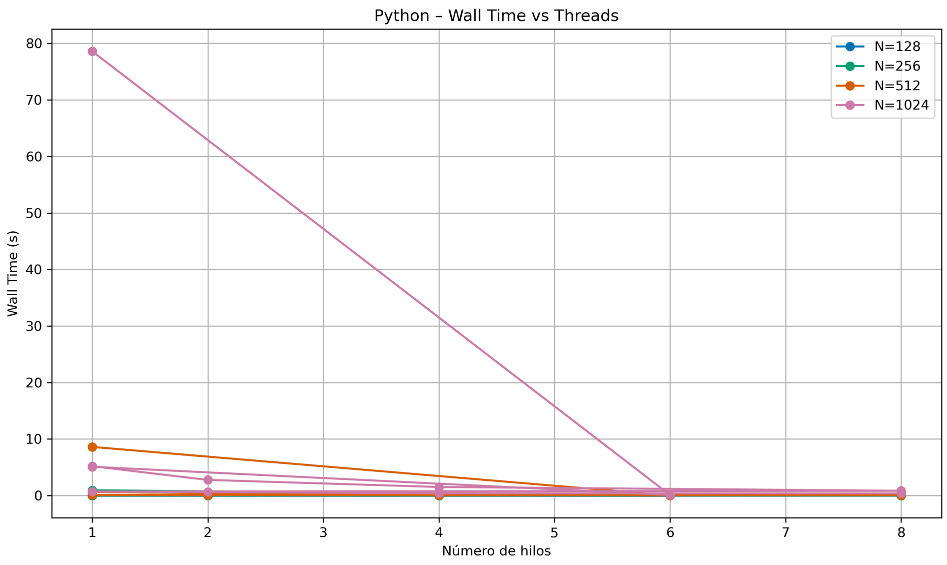

Figure 10: Python – Wall Time vs Threads. Wall time decreases with more threads, especially for $N = 1024$.

Figure 10 illustrates the relationship between thread count and wall time for matrix multiplication in Python, using the `Numba_Blocked` implementation. As the number of threads increases from 1 to 8, wall time consistently decreases across all matrix sizes, confirming the benefits of parallel execution.

The reduction is most significant for larger matrices, particularly at $N = 1024$, where the computational workload is sufficient to fully utilize available threads. For smaller matrices such as $N = 128$, the performance gains are limited, as parallel overhead—including thread initialization, synchronization, and memory contention—can outweigh the benefits of concurrency.

These results validate the hypothesis that parallel methods scale more effectively with increasing problem size. They also highlight the importance of matching thread count to workload granularity in Python, especially when using JIT-compiled parallel loops.

## 4.11 Sparse Methods in Java

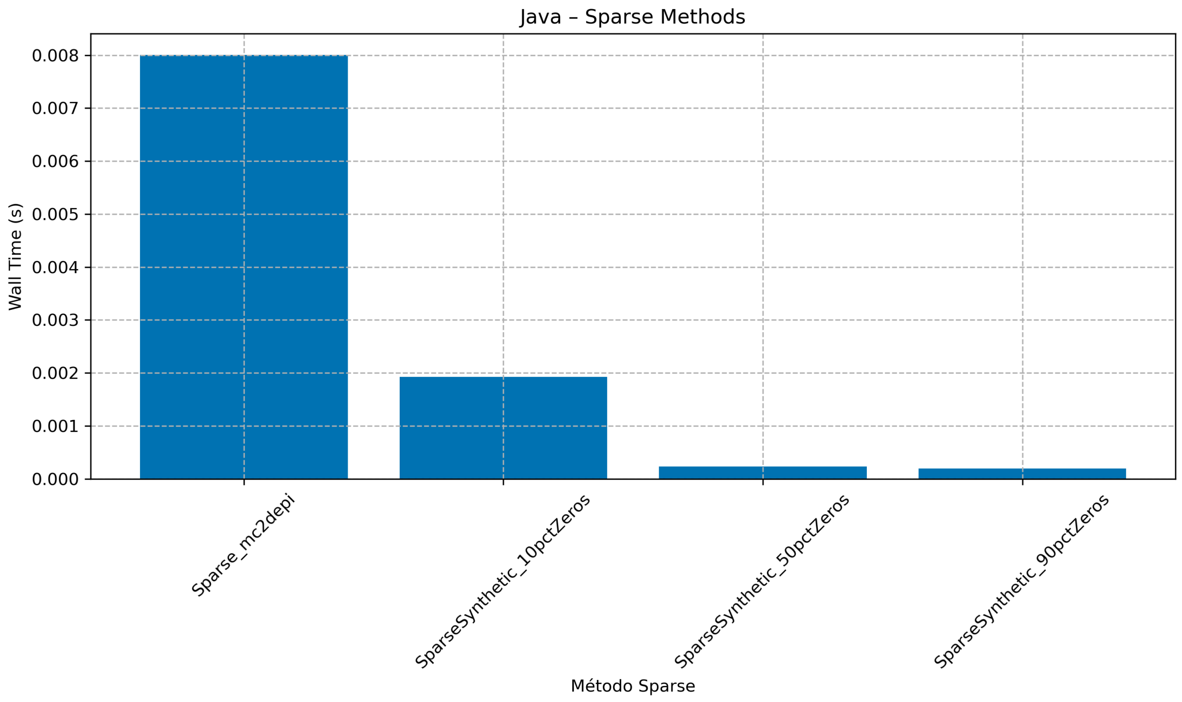

Figure 11: Java – Sparse Methods. Synthetic matrices with higher sparsity are faster.

Figure 11 compares the wall time performance of four sparse matrix multiplication methods in Java: one real-world dataset (`Sparse_mc2depi`) and three synthetic matrices with increasing sparsity levels (10%, 50%, and 90% zeros). The results demonstrate a clear inverse relationship between sparsity and execution time.

The synthetic matrices with 90% and 50% zeros achieve the lowest wall times, confirming that higher sparsity leads to fewer arithmetic operations and memory accesses. This behavior is consistent with the characteristics of the Compressed Sparse Row (CSR) format, which efficiently skips zero entries during computation. The method using 10% sparsity performs moderately, while the real-world dataset (`mc2depi`) shows the highest wall time due to its lower sparsity and more irregular structure.

These findings validate the hypothesis that sparse representations significantly reduce computational cost when sparsity is high. They also highlight the importance of data structure and sparsity distribution in determining performance, especially in Java where memory access patterns and object overhead can impact execution time.

## 4.12 Sparse Methods in Python

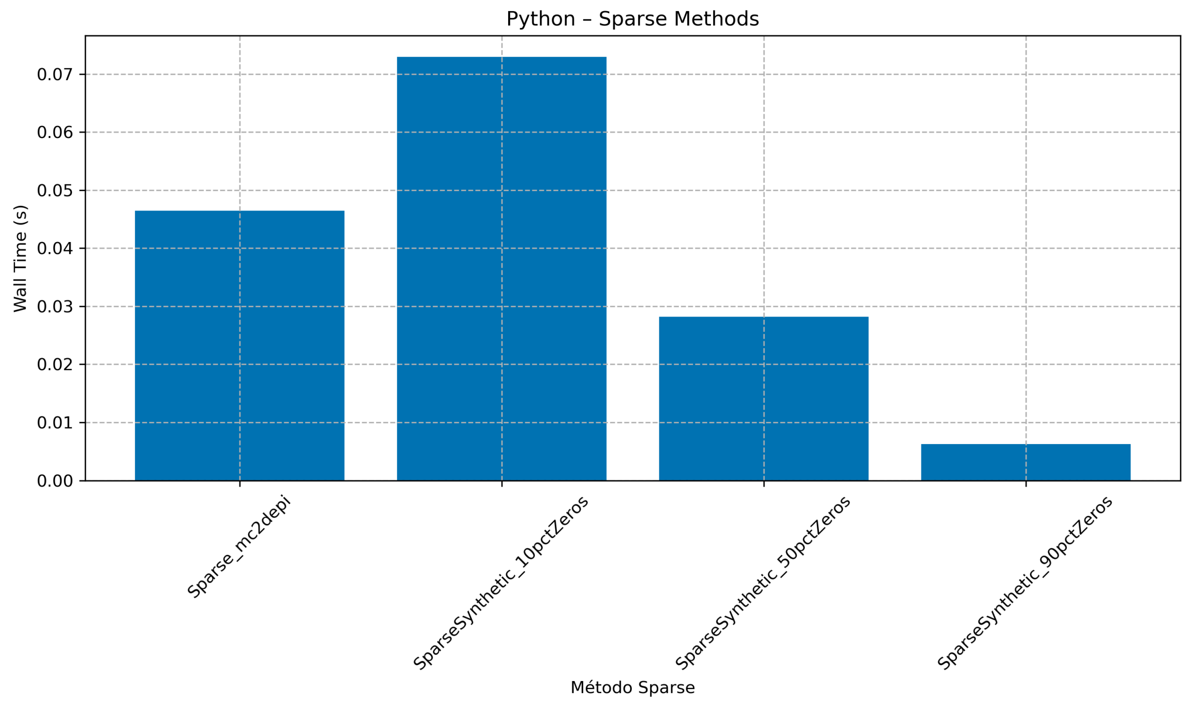

Figure 12: Python – Sparse Methods. Performance improves with sparsity, confirming CSR efficiency.

Figure 12 compares the wall time of four sparse matrix multiplication methods in Python: one real-world dataset (`Sparse_mc2depi`) and three synthetic matrices with increasing sparsity levels (10%, 50%, and 90% zeros). The results reveal a clear trend: as sparsity increases, execution time decreases significantly.

The synthetic matrix with 90% zeros achieves the lowest wall time, followed by the 50% variant. This confirms that the Compressed Sparse Row (CSR) format efficiently skips zero entries, reducing both computational load and memory access. The method using 10% sparsity performs notably slower, as the density of non-zero elements increases the number of operations required. The real-world dataset (`mc2depi`) shows intermediate performance, reflecting its moderate sparsity and more irregular structure.

These results validate the hypothesis that sparse matrices benefit from specialized representations, particularly when sparsity is high. They also demonstrate that Python's sparse matrix libraries can handle both synthetic and real-world data effectively, provided the format and sparsity level are well matched to the computational strategy.

## 5 Conclusions

In conclusion, this study demonstrates that parallel and vectorized matrix multiplication yields substantial performance improvements across all tested languages. Blocked and SIMD-based strategies consistently outperform naive implementations, confirming the importance of memory locality and low-level optimization. The results show that Java and C scale efficiently with increasing thread counts, while Python achieves competitive performance primarily through optimized libraries and JIT compilation.

The experiments also confirm that sparse matrix methods benefit significantly from high sparsity, with CSR representations reducing computational cost by skipping zero entries. These findings extend beyond the core assignment by incorporating optional requirements such as vectorization and additional experiments on sparse formats and cross-language comparisons.

Overall, the study contributes to the field by providing a reproducible, multi-language benchmarking framework that highlights the trade-offs between abstraction, control, and performance in numerical computing.

## 6 Future Work

Future work will focus on extending the current benchmarking framework towards more advanced and scalable architectures. A natural next step is the integration of GPU acceleration through CUDA or similar frameworks, enabling massive parallelism and exploiting hardware-level optimizations. Hybrid parallelism strategies (e.g., MPI combined with OpenMP) will also be explored to balance distributed and shared-memory execution, improving scalability across heterogeneous environments.

Another promising direction is adaptive blocking, where block sizes are dynamically tuned based on matrix dimensions and hardware characteristics to maximize cache utilization. In addition, deeper profiling of cache behavior and memory bandwidth will provide insights into bottlenecks and guide further optimization.

Finally, extending the experiments to distributed systems would align the study with Big Data architectures, allowing evaluation of performance in cluster-based environments and contributing to the design of reproducible, large-scale numerical workflows.

## 7 GitHub Repository

The full source code, datasets, and plotting scripts used in this study are available at the following repository:

- `https://github.com/judporper/Language-Benchmark-of-matrix-multiplication/tree/main/TASK3`

## 8 References

- Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K., & Demmel, J. (2007). *Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms.* Proceedings of the ACM/IEEE Supercomputing Conference (SC07), Reno, Nevada, USA.