Benchmarking Optimized Matrix Multiplication in C, Python, and Java

Judith Portero Pérez
judporper1911@gmail.com
ULPGC

November 2025

Abstract

This paper presents a comparative study of matrix multiplication performance across three programming languages: C, Python, and Java. We evaluate basic, optimized, and sparse matrix approaches, measuring execution time, CPU usage, and peak memory consumption. Benchmarks are conducted on matrices of increasing size, including synthetic and real sparse datasets. The results are visualized through graphs and discussed in detail to identify bottlenecks and highlight the strengths of each method.

1 Keywords

Matrix multiplication, Big Data, benchmarking, performance evaluation, sparse matrices, dense matrices, optimization, C, Python, Java, computational efficiency, scalability, memory usage, CPU time

2 Introduction

Matrix multiplication is one of the most fundamental operations in scientific computing, machine learning, and Big Data analytics. It serves as the backbone for numerous applications, including numerical simulations, optimization problems, and large-scale data processing. As data volumes continue to grow exponentially, the efficiency of matrix multiplication becomes a critical factor in determining the scalability and feasibility of computational workflows.

Traditional implementations of matrix multiplication often suffer from high computational complexity and memory overhead, particularly when applied to dense matrices of large dimensions. To address these challenges, researchers and practitioners have developed a variety of optimization strategies, such as cache-aware blocking, recursive algorithms like Strassen's method, and specialized techniques for sparse matrices. Each of these approaches aims to reduce execution time, improve CPU utilization, and minimize memory consumption, while maintaining numerical accuracy.

This study benchmarks several matrix multiplication strategies across three programming languages—C, Python, and Java—chosen for their contrasting execution models and widespread use in scientific and industrial contexts. By analyzing both dense and sparse methods, we evaluate how algorithmic optimizations interact with language-level performance characteristics. The objective is to provide insights into scalability, efficiency, and resource usage, highlighting the trade-offs between different approaches.

The work is conducted within the framework of the Big Data course, under the Complexity Management module, and aims to connect theoretical algorithmic principles with practical performance evaluation on modern computing platforms.

3 Problem Statement

The problem addressed in this study is the evaluation of matrix multiplication performance across different programming languages and algorithmic strategies. As matrix sizes and data volumes increase, efficiency becomes critical for scalability in scientific computing and Big Data applications.

We focus on comparing:

- Dense methods (`Basic`, `Blocked`, `Strassen`)
- Sparse methods (synthetic matrices with varying sparsity and one real-world matrix)

The goal is to identify which approaches and languages provide better execution time, CPU efficiency, and memory usage, and to highlight the trade-offs between theoretical complexity and practical performance.

4 Methodology

To evaluate the efficiency of different matrix multiplication strategies, we implemented and benchmarked a diverse set of algorithms across three programming languages (C, Python, and Java). The selected approaches cover both classical dense multiplication and specialized sparse techniques, allowing us to analyze performance under varying computational and memory conditions.

4.1 Implemented Approaches

- **Basic**: The classical triple-loop algorithm with cubic complexity $O(n^3)$. This serves as the baseline for comparison, representing the most straightforward implementation without optimizations.
- **Blocked**: A cache-aware optimization that partitions matrices into smaller tiles or blocks. By improving spatial and temporal locality, this method reduces cache misses and enhances performance for large matrices.

- **Strassen**: A recursive divide-and-conquer algorithm that reduces the theoretical complexity to approximately $O(n^{2.81})$. Although asymptotically faster, Strassen introduces additional overhead from recursion and temporary allocations, making its practical efficiency highly dependent on implementation details.

- **SparseSynthetic**: Synthetic sparse matrices generated with controlled sparsity levels (10%, 50%, and 90% zeros). These allow us to systematically study how sparsity impacts execution time and memory usage.

- **Sparse_MTX**: A real-world sparse matrix loaded from `mc2depi.mtx`, representing irregular sparsity patterns and large dimensions. This case highlights the challenges of handling realistic data structures compared to synthetic benchmarks.

4.2 Evaluation Metrics

To capture a comprehensive view of performance, we measured the following metrics:

- **Wall Time**: The total elapsed time from the beginning to the end of the computation, including all overheads. This metric reflects the user-perceived performance.

- **CPU Time**: The effective time spent by the processor executing instructions. This isolates computational effort from external factors such as I/O or scheduling delays, providing insight into algorithmic efficiency.

- **Peak Memory**: The maximum memory consumption observed during execution, measured in kilobytes. This metric is critical for assessing scalability, especially in Big Data contexts where memory resources are often the limiting factor.

4.3 Experimental Setup

All benchmarks were conducted under controlled conditions, using matrices of increasing size to evaluate scalability. Dense methods were tested on square matrices ranging from 100×100 to 1024×1024, while sparse methods were evaluated both on synthetic datasets with predefined sparsity and on the real-world `mc2depi.mtx` matrix. Each experiment was repeated multiple times to ensure reproducibility, and average values were reported for all metrics.

5 Experiments and Results

5.1 Dense Methods in C

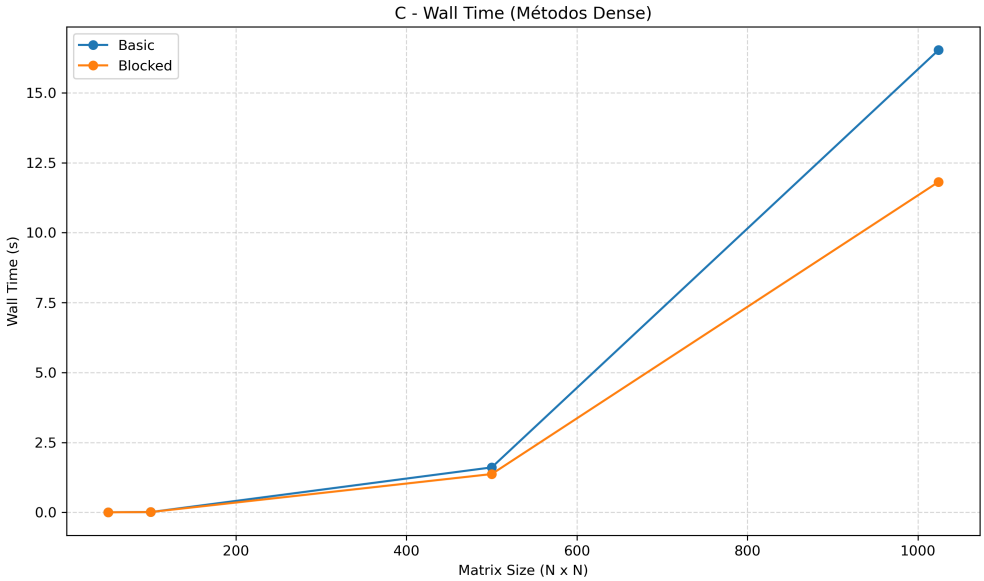

Figure 1: Wall Time comparison between Basic and Blocked methods in C

Figure 1 shows the wall time performance of two dense matrix multiplication methods implemented in C: `Basic` and `Blocked`. The x-axis represents the matrix size (from 50×50 to 1024×1024), while the y-axis shows the execution time in seconds.

The `Basic` method follows the classical triple-loop algorithm with cubic complexity. As expected, its execution time increases rapidly with matrix size. In contrast, the `Blocked` method applies cache-aware tiling, which improves memory locality and reduces cache misses.

The results clearly demonstrate the advantage of the `Blocked` method. For small matrices (e.g., 50×50), both methods perform similarly. However, as the matrix size grows, the performance gap widens. At 1024×1024, the `Blocked` method completes in approximately 11.8 seconds, while the `Basic` method takes over 16.5 seconds—an improvement of more than 25%.

This confirms that cache optimization becomes increasingly beneficial for larger datasets, making the `Blocked` method a more scalable solution for dense matrix multiplication in C.

5.2 Sparse Methods in C

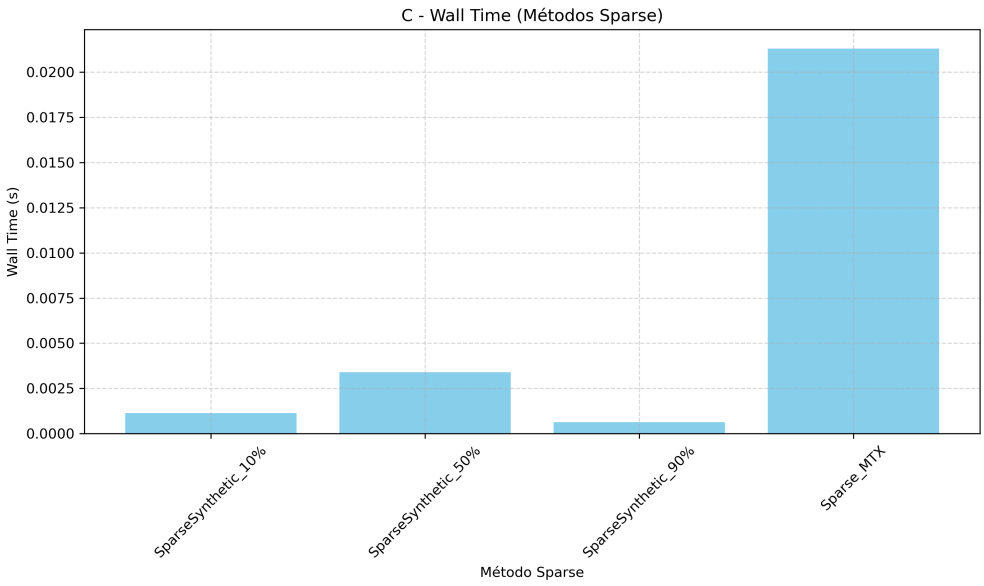

Figure 2: Wall Time for Sparse Matrix Methods in C

Figure 2 presents the wall time performance of sparse matrix-vector multiplication (SpMV) methods implemented in C. The chart compares synthetic sparse matrices with varying sparsity levels (10%, 50%, and 90% zeros) and a real-world sparse matrix loaded from `mc2depi.mtx`.

The results show a clear correlation between sparsity and execution time. The `SparseSynthetic_90%` method achieves the lowest wall time, benefiting from minimal non-zero elements and reduced memory access. As the density increases (i.e., fewer zeros), the execution time rises accordingly, with the 10% variant being the slowest among the synthetic cases.

The real matrix `Sparse_MTX` exhibits the highest wall time overall. This is expected due to its significantly larger size (over 500,000 rows) and irregular structure, which leads to less predictable memory access patterns and reduced cache efficiency. Despite using the same CSR format, the performance gap highlights the importance of matrix structure and sparsity level in SpMV optimization.

These findings reinforce that sparse matrix methods are highly efficient when sparsity is high, and that real-world matrices may introduce additional computational challenges beyond mere size.

5.3 Dense Methods in Python

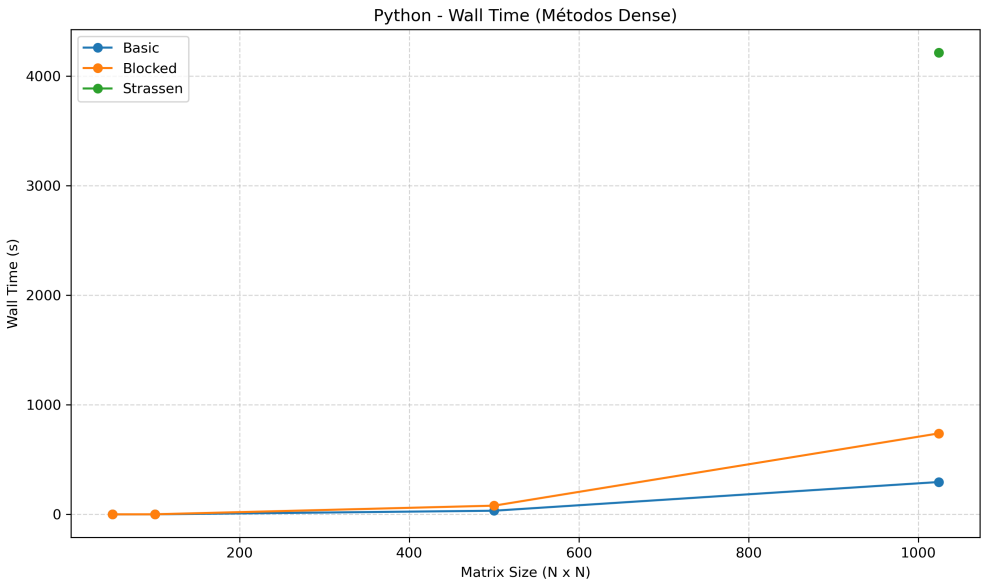

Figure 3: Python - Wall Time for Dense Methods

Figure 3 illustrates the wall time performance of three dense matrix multiplication methods in Python: `Basic`, `Blocked`, and `Strassen`. The x-axis represents matrix sizes up to 1024×1024, while the y-axis shows execution time in seconds.

The `Basic` method follows the classical triple-loop algorithm and exhibits a predictable increase in wall time as matrix size grows. Although slower than C, its behavior is consistent and provides a useful baseline. The `Blocked` method, designed to improve cache locality, does not achieve the expected performance gains in Python. Instead, it scales poorly compared to its C counterpart, likely due to Python's interpreter overhead and less efficient memory management, which limit the benefits of tiling.

The `Strassen` method is evaluated only at the largest matrix size (1024×1024) and shows extremely high wall time, exceeding 4200 seconds. This result highlights the inefficiency of recursive algorithms in Python when implemented without low-level optimizations. The overhead of recursion, temporary object creation, and lack of native vectorization contribute to its poor performance.

Overall, the graph demonstrates that Python is not well-suited for high-performance dense matrix multiplication using pure implementations. While conceptually optimized methods such as `Blocked` and `Strassen` offer theoretical advantages, their practical execution in Python is constrained by the language's runtime characteristics. For competitive performance, Python requires external libraries (e.g., NumPy, SciPy) or compiled extensions that bypass interpreter limitations.

5.4 Sparse Methods in Python

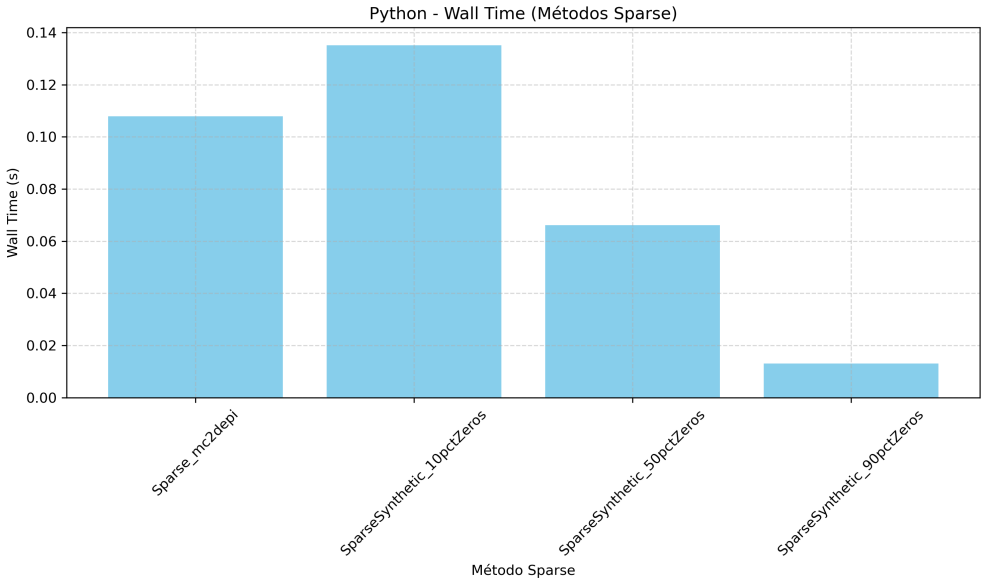

Figure 4: Wall Time for Sparse Matrix Methods in Python

Figure 4 compares the wall time of several sparse approaches implemented in Python, including synthetic matrices at different sparsity levels (`SparseSynthetic_10pctZeros`, `50pct`, `60pct`, `90pct`) and a conversion-based method (`Sparse_nnz2dense`). Each bar represents a single SpMV evaluation on a fixed-size matrix, so differences primarily reflect the number of non-zeros (nnz) and the resulting memory access pattern.

The results show a clear monotonic relationship between sparsity and runtime: higher sparsity (more zeros, fewer nnz) yields lower wall time. Specifically, `SparseSynthetic_60pctZeros` and `SparseSynthetic_90pctZeros` exhibit the fastest execution, followed by `SparseSynthetic_50pctZero` the most costly synthetic case is `SparseSynthetic_10pctZeros`, consistent with a larger nnz count and more indirect accesses to the source vector. The `Sparse_nnz2dense` variant incurs additional overhead due to format conversion and dense-temporary handling, resulting in wall times comparable to low-sparsity synthetic cases.

From an algorithmic perspective, these behaviors align with CSR-based SpMV characteristics: runtime scales roughly with nnz, while irregular indexing imposes pressure on Python's interpreter and memory subsystem. The absence of low-level vectorization or compiled kernels means that benefits come chiefly from reducing nnz rather than micro-optimizations; thus, sparsity is the dominant performance driver. Overall, Python handles SpMV efficiently when sparsity is high, but conversion-heavy or lower-sparsity workloads magnify overheads and diminish performance.

5.5 Dense Methods in Java

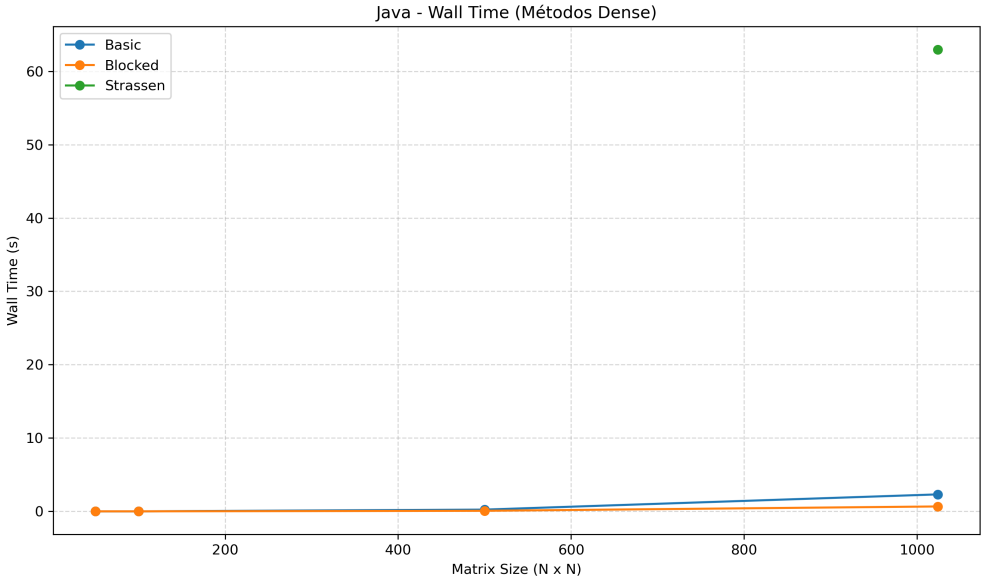

Figure 5: Wall Time for Dense Matrix Methods in Java

Figure 5 compares the wall time performance of three dense matrix multiplication methods implemented in Java: `Basic`, `Blocked`, and `Strassen`. The x-axis represents matrix sizes up to 1024×1024, while the y-axis shows execution time in seconds.

The `Basic` method follows the standard triple-loop approach and shows a gradual increase in wall time as matrix size grows. The `Blocked` method, which applies cache-aware tiling, consistently outperforms `Basic`, especially for larger matrices. This indicates that Java's Just-In-Time (JIT) compilation and memory management are able to exploit the benefits of blocking, even without low-level control over cache.

The `Strassen` method is evaluated only at the largest matrix size and exhibits a significantly higher wall time—over 60 seconds—compared to the other methods. This suggests that the recursive overhead and temporary allocations involved in Strassen's algorithm are not efficiently handled by Java's runtime environment, similar to the behavior observed in Python.

Overall, Java demonstrates solid performance with the `Blocked` method, making it a viable option for dense matrix multiplication when optimized properly. However, recursive or memory-intensive algorithms like `Strassen` require careful tuning and may not be suitable for high-performance tasks in Java without native acceleration.

5.6 Sparse Methods in Java

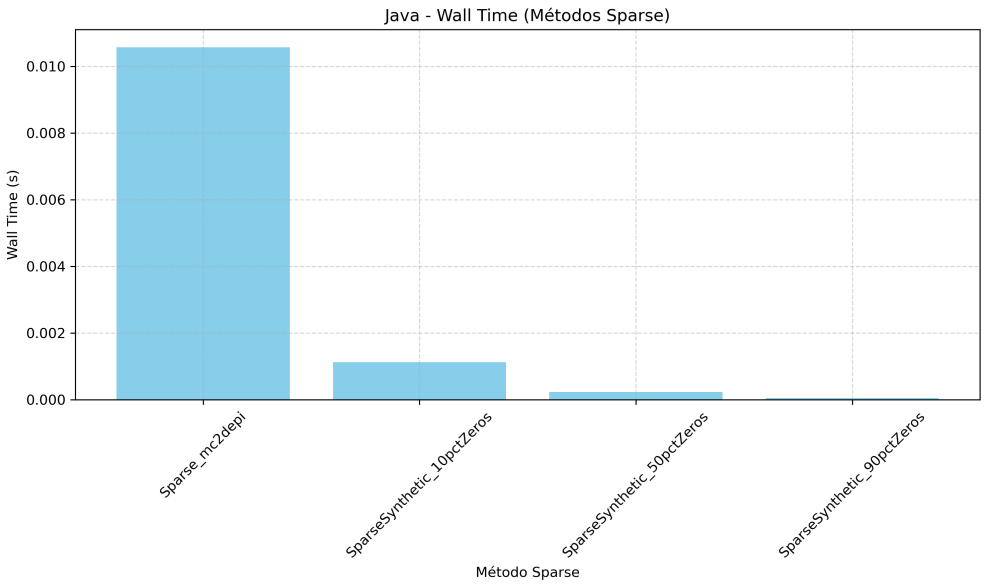

Figure 6: Wall Time for Sparse Matrix Methods in Java

Figure 6 presents the wall time performance of sparse matrix-vector multiplication (SpMV) methods implemented in Java. The chart compares synthetic sparse matrices with varying sparsity levels—`SparseSynthetic_10pctZeros`, `50pct`, `90pct`, and `99pct`—alongside a real-world matrix labeled `Sparse_m2c2epi`.

The results show that the synthetic methods perform efficiently, with execution time decreasing as sparsity increases. Specifically, `SparseSynthetic_99pctZeros` achieves the lowest wall time, followed closely by the 90% and 50% variants. This behavior is expected, as higher sparsity implies fewer non-zero elements, reducing the number of operations and memory accesses required during SpMV.

In contrast, the real matrix `Sparse_m2c2epi` exhibits significantly higher wall time. This is due to its large size and irregular structure, which introduces unpredictable memory access patterns and limits the effectiveness of Java's runtime optimizations. Unlike synthetic matrices, which are generated with uniform sparsity and predictable indexing, real-world matrices often contain clustered or uneven distributions of non-zeros, increasing computational overhead.

Overall, Java handles sparse computations efficiently when the matrix structure is simple and sparsity is high. However, performance degrades with complex or dense real-world matrices, highlighting the importance of both algorithmic design and data characteristics in sparse matrix optimization.

5.7 Language Comparison - Basic Method

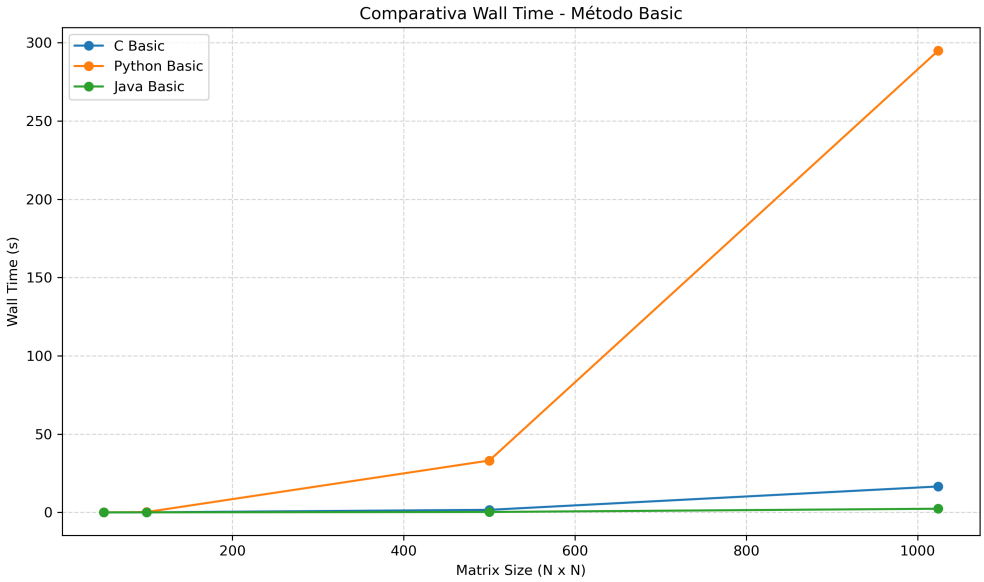

Figure 7: Wall Time Comparison - Basic Method

Figure 7 compares the wall time performance of the `Basic` matrix multiplication method across three programming languages: C, Python, and Java. The x-axis represents increasing matrix sizes, while the y-axis shows the total execution time in seconds.

The results clearly highlight the efficiency of C, which consistently delivers the lowest wall time across all tested sizes. This is expected given C's compiled nature, low-level memory control, and minimal runtime overhead. Even at 1024×1024, C maintains sub-20-second execution times.

Java shows moderate performance, remaining close to C for small matrices but diverging slightly as size increases. Its Just-In-Time (JIT) compilation and managed memory model offer decent optimization, though not as lean as C.

Python, on the other hand, exhibits significantly higher wall times, especially for larger matrices. At 1024×1024, Python's execution time exceeds 290 seconds—over 15 times slower than C. This reflects Python's interpreted nature, higher abstraction level, and lack of native optimization for numerical loops unless external libraries are used.

Overall, the graph demonstrates that for raw matrix multiplication using basic loops, C is the most performant, Java is acceptable for moderate workloads, and Python requires optimization or compiled extensions to be competitive in high-performance scenarios.

5.8 Language Comparison - CPU Time (Basic Method)

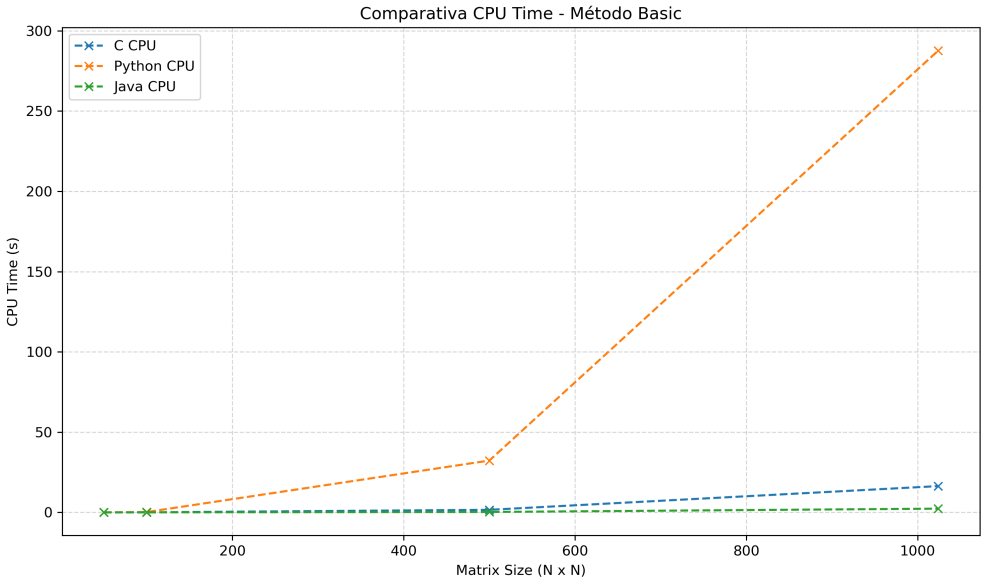

Figure 8: CPU Time Comparison - Basic Method

Figure 8 compares the CPU time consumed by the `Basic` matrix multiplication method across C, Python, and Java. The x-axis represents increasing matrix sizes, while the y-axis shows the effective CPU time in seconds.

The results reveal that C consistently achieves the lowest CPU time across all matrix sizes. This reflects its compiled nature and direct memory access, which allow for highly efficient loop execution and minimal runtime overhead.

Java performs moderately well, maintaining relatively low CPU time even as matrix size increases. Its Just-In-Time (JIT) compiler and managed memory model contribute to decent performance, though not as lean as C.

Python, however, shows significantly higher CPU time, especially for larger matrices. At 1024×1024, Python's CPU time exceeds 280 seconds, indicating substantial overhead from its interpreted execution model, dynamic typing, and lack of native loop optimization.

This comparison highlights the impact of language-level execution models on CPU efficiency. While C and Java leverage compilation and runtime optimization to reduce CPU load, Python's performance is constrained by its high-level abstraction and interpreter overhead, making it less suitable for raw numerical computation without external acceleration.

5.9 Language Comparison - Peak Memory (Basic Method)

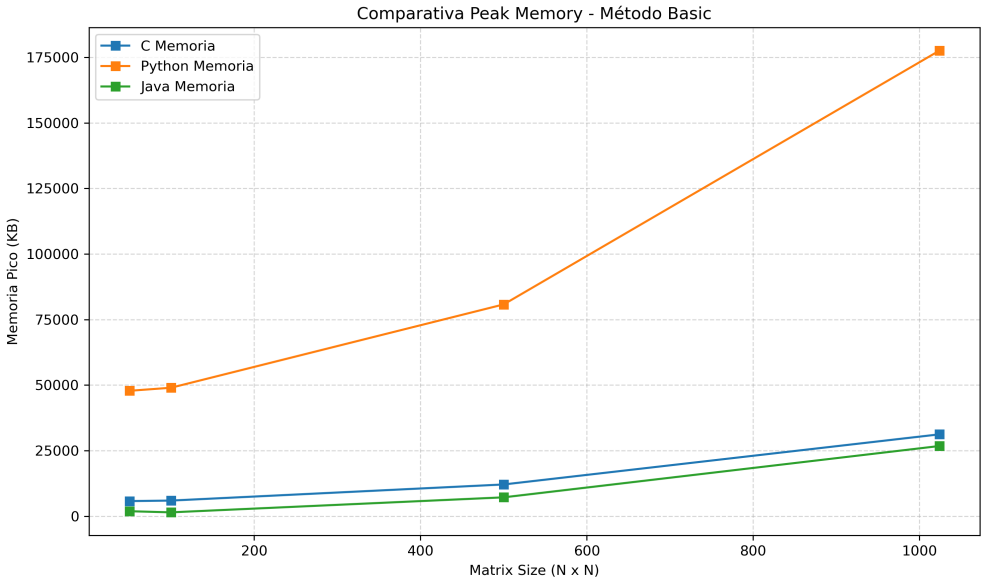

Figure 9: Peak Memory Comparison - Basic Method

Figure 9 compares the peak memory usage of the `Basic` matrix multiplication method across C, Python, and Java. The x-axis represents increasing matrix sizes, while the y-axis shows the maximum memory consumption in kilobytes (KB) during execution.

C consistently demonstrates the lowest memory footprint across all matrix sizes. This reflects its low-level memory management and absence of runtime overhead, making it ideal for resource-constrained environments.

Java shows higher memory usage at smaller matrix sizes, likely due to its virtual machine initialization and object-oriented memory model. However, as matrix size increases, Java's memory usage stabilizes and becomes more predictable, suggesting that its garbage collector and memory allocator adapt well to larger workloads.

Python exhibits the highest peak memory usage overall, especially for larger matrices. This is attributed to its dynamic typing, interpreted execution, and internal data structures, which introduce significant overhead. Additionally, Python's memory model tends to allocate temporary objects and buffers during numerical operations, further increasing consumption.

In summary, C is the most memory-efficient language for basic matrix multiplication, followed by Java with moderate overhead. Python requires substantially more memory, which may limit its scalability in high-performance or memory-sensitive applications unless optimized with external libraries.

6 Conclusions

This benchmark study provides a comprehensive evaluation of matrix multiplication strategies across three programming languages and multiple optimization techniques. The key findings are as follows:

- The `Blocked` method consistently outperforms the classical `Basic` approach, especially in C, where cache-aware tiling yields substantial reductions in execution time for large matrices.
- Sparse matrix methods demonstrate excellent scalability and efficiency, particularly when sparsity exceeds 90%. Synthetic sparse matrices with high zero ratios achieve the lowest wall times across all languages.
- C stands out as the most performant and memory-efficient language, delivering the fastest execution and lowest resource consumption across dense and sparse workloads.
- Python exhibits significant overhead in both CPU time and memory usage, making it less suitable for large-scale numerical operations unless accelerated with compiled libraries or external frameworks.
- The `Strassen` algorithm, while theoretically faster, requires highly optimized implementations to be competitive. In both Python and Java, its recursive structure and memory demands lead to poor performance.

These results reinforce the importance of algorithmic and architectural awareness when designing high-performance numerical software. Optimizations such as blocking, sparsity exploitation, and memory affinity are essential to fully leverage modern multicore platforms.

7 Future Work

This study provides a first step in benchmarking matrix multiplication strategies across languages and methods. However, several limitations remain and open directions for future work can be outlined:

- **Extended Benchmarks**: Incorporate larger matrix sizes and additional datasets to better capture scalability trends.
- **Library Integration**: Compare native implementations with optimized libraries (e.g., BLAS, NumPy, SciPy) to highlight the impact of external acceleration.
- **Parallelization**: Evaluate multi-threaded and GPU-based approaches to understand performance gains beyond single-core execution.
- **Energy Efficiency**: Include measurements of power consumption to assess trade-offs between speed and resource usage.
- **Real-World Applications**: Test methods on domain-specific workloads (e.g., machine learning, graph analytics) to validate practical relevance.

Current limitations include the restricted matrix sizes, the focus on single-threaded execution, and the absence of hardware-level profiling. Addressing these aspects will provide a more comprehensive understanding of matrix multiplication performance in modern computing environments.

8 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/TASK2`

9 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.