Parallelization Benchmarking in C, Java, and Python

Diego Muñoz Torres
Universidad de Las Palmas de Gran Canaria

Abstract

This paper presents a comparative study of parallel and vectorized implementations of the classical $O(n^3)$ dense matrix multiplication algorithm in three programming languages: C, Java, and Python. The project focuses on exploiting both thread-level parallelism (multi-core CPUs) and data-level parallelism (SIMD / vectorization) while keeping the core algorithm identical across languages.

For C, a modular implementation in CLion includes four variants: a sequential baseline, a multi-threaded version using POSIX threads, a vectorized kernel based on AVX intrinsics, and a combined parallel + SIMD version. The Java implementation, developed in IntelliJ with Maven, uses a clean object-oriented design with separate classes for baseline multiplication, parallel execution via ExecutorService, and a version using parallelStream(). Synchronization mechanisms such as AtomicInteger, Semaphore, and synchronized are explicitly employed to coordinate worker threads and serialize result logging. The Python implementation, structured as a small package in PyCharm, provides a sequential kernel using nested lists, a parallel version based on ProcessPoolExecutor, and a vectorized version using NumPy matrices.

All three programs share a common benchmarking protocol, generating random square matrices, executing each variant for multiple sizes, and exporting results to CSV files. The main metrics are speedup over the sequential baseline, parallel efficiency (speedup per thread or process), and qualitative resource usage in terms of core utilization and memory footprint. Experiments were conducted on an Intel Core i7 laptop (6 physical cores, 12 hardware threads). The results show that C achieves the highest absolute performance and the best combined parallel + SIMD speedup (up to $\approx 10\times$ for $n = 1024$ with 12 threads), Java exhibits reasonable scalability with speedups around $10\times$ for the largest size, and Python attains competitive performance only when leveraging vectorized NumPy operations, which reach speedups above $300\times$ over pure-Python loops, while multiprocessing on pure Python yields more modest gains of up to about $4.4\times$.

Keywords: matrix multiplication, parallel computing, SIMD, parallel streams, multiprocessing, performance analysis

1 Introduction

Matrix multiplication is one of the core operations in numerical computing, with applications in machine learning, scientific simulation, computer graphics, and signal processing. Its computational cost dominates many higher-level algorithms, so even modest improvements in the underlying implementation can translate into substantial runtime savings at the application level.

The classical algorithm for multiplying two dense $n \times n$ matrices has time complexity $O(n^3)$ and space complexity $O(n^2)$. Despite the existence of more advanced algorithms with lower asymptotic complexity, such as Strassen's method or highly optimized BLAS routines, the naive triple-loop algorithm remains an essential building block and a natural baseline for performance studies.

Modern processors expose at least two levels of parallelism that can be exploited by this algorithm:

- **Thread-level parallelism**, by distributing independent rows or blocks of the result matrix across multiple CPU cores.
- **Data-level parallelism**, by using SIMD (Single Instruction, Multiple Data) instructions to operate on several elements of a row or column in parallel within a single core.

In addition, the execution model of the programming language, compiled versus interpreted, explicit versus automatic memory management, availability of threading and synchronization abstractions, has a strong influence on both raw performance and programmer productivity.

This project investigates how much performance can be gained by combining parallelism and vectorization in three widely used languages:

- **C**, a low-level compiled language with direct access to POSIX threads and SIMD intrinsics;
- **Java**, a managed language running on the Java Virtual Machine (JVM), with high-level concurrency libraries and parallel streams;
- **Python**, a dynamically typed, interpreted language that provides multiprocessing for CPU-bound tasks and accesses vectorized native code through NumPy.

The main objectives are:

- to implement a common dense $O(n^3)$ matrix multiplication algorithm in C, Java, and Python;
- to add parallel and vectorized variants in each language, using idiomatic mechanisms such as pthreads and AVX in C, `ExecutorService` and streams in Java, and `ProcessPoolExecutor` and NumPy in Python;
- to measure speedup, efficiency, and resource usage on a multi-core Intel system;
- to compare the behavior of the three languages under similar workloads, highlighting the trade-offs between performance and abstraction level.

2 Problem Statement

The core computational task is the multiplication of two dense square matrices. Given $A, B \in R^{n \times n}$, their product C is defined as

$$C = AB, \qquad C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}, \qquad 1 \leq i, j \leq n.$$

The naive algorithm implements this definition with three nested loops and performs n^3 multiplications and a similar number of additions, leading to an $O(n^3)$ time complexity and $O(n^2)$ space usage for storing A, B, and C. Even for moderately large n, this cubic cost makes matrix multiplication an excellent stress test for CPU-bound computation and memory bandwidth.

In a purely sequential implementation, all arithmetic operations are executed one after another on a single core. However, the computation of distinct entries or rows of C is mutually independent, making the algorithm naturally suitable for parallelization. Similarly, each inner product $C_{ij} = \sum_k A_{ik} B_{kj}$ consists of operations on contiguous memory regions that can benefit from vector instructions.

The present work investigates the following questions:

- How much speedup can be obtained by multi-threading the naive dense matrix multiplication algorithm on a 6-core / 12-thread Intel CPU?
- To what extent does vectorization (SIMD intrinsics in C or NumPy in Python) further accelerate the computation?
- How do thread-level parallelism and data-level parallelism combine? In particular, does the parallel + SIMD version provide close to the product of the individual speedups, or is the gain limited by memory bandwidth and synchronization overhead?
- How do these trends differ across C, Java, and Python, given their different execution and memory models?

To answer these questions, the following performance metrics are used:

- **Execution time** T: wall-clock time to compute $C = AB$ for a given size n.
- **Speedup** S of a method relative to the sequential baseline:

$$S = \frac{T_{\text{basic}}}{T_{\text{method}}}.$$

- **Parallel efficiency** E for a method using p threads or processes:

$$E = \frac{S}{p}.$$

The working hypothesis is that:

1. parallel implementations will yield near-linear speedup up to the number of physical cores, with diminishing returns beyond that point;
2. vectorization will provide an additional but more modest speedup, especially for large matrices where the computation becomes memory-bound;
3. C will achieve the highest absolute performance, Java will be somewhat slower due to JVM overhead, and pure-Python loops will be much slower, with NumPy closing the gap by offloading work to native code.

3 Methodology

This section describes the implementation strategy, project structure, and benchmarking procedure followed in C, Java, and Python. In all cases, the production code (matrix operations) is separated from the benchmark driver that iterates over matrix sizes and thread counts and writes results to CSV files.

3.1 Common Algorithmic Structure

All three languages implement the same basic algorithm:

```
for i in 0..n-1:
  for k in 0..n-1:
    aik = A[i][k]
    for j in 0..n-1:
      C[i][j] += aik * B[k][j]
```

This loop ordering (`i-k-j`) reuses the loaded value `aik` across the inner loop over j and aligns well with the row-major storage used in C and Java. In Python, the same structure is used for the baseline and parallel versions based on nested lists, while the vectorized version relies on NumPy's @ operator.

Random matrices are generated with values in $[0, 1)$ using a fixed seed per language to ensure reproducibility. No algorithmic optimizations beyond parallelism and vectorization are applied; in particular, no blocking/tiling and no use of highly specialized BLAS libraries is made, except for the NumPy backend in the Python vectorized variant.

3.2 C Implementation (CLion)

The C code is organized as a small CLion project with the following files:

- `matmul.h` and `matmul.c`: type definitions and implementations of the four kernels:
  - `matmul_basic`: sequential baseline;
  - `matmul_parallel`: multi-threaded version using `pthread_create()` and static row partitioning;
  - `matmul_simd`: sequential but vectorized using AVX intrinsics from immintrin.h;
  - `matmul_parallel_simd`: combination of row-level parallelism with SIMD operations inside each thread.
- `main.c`: benchmark driver that iterates over matrix sizes $n \in \{256, 512, 1024\}$, detects the number of hardware threads, selects three configurations (1, ≈6, and 12 threads), and records the execution time and derived metrics into a CSV file `results.csv`.

Matrices are stored as one-dimensional arrays of double of length n^2. Helper functions handle allocation, deallocation, and random initialization. The benchmark uses gettimeofday() to measure elapsed wall-clock time in milliseconds.

Compilation is performed via CMake with optimization and vectorization flags:

```
set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -O3 -march=native -pthread")
```

so that the compiler can use the full AVX instruction set available on the Intel Core i7 CPU.

3.3 Java Implementation (IntelliJ + Maven)

The Java implementation is organized as a Maven project under the package `ulpgc.shared.matrix`. The main classes are:

- `MatrixUtils`: static utility methods to allocate matrices (`double[]` representing flattened $n \times n$ arrays), fill them with random values, and convert nanoseconds to milliseconds.
- `MatrixMultiplier` (interface): defines `multiply(double[] A, double[] B, double[] C, int n)` and `getName()`.
- `BasicMatrixMultiplier`: implements the sequential baseline triple loop.
- `ExecutorMatrixMultiplier`: parallel implementation using `ExecutorService` with a fixed thread pool. Rows of C are partitioned among tasks, and the class coordinates completion using:
  - `AtomicInteger` to count completed tasks, and
  - `Semaphore` to signal when the last task finishes.

  These are the explicit synchronization mechanisms requested in the assignment.
- `ParallelStreamMatrixMultiplier`: variant that uses `IntStream.range(0, n).parallel()` to parallelize over rows of the result matrix using the JVM's common fork-join pool.
- `BenchmarkRunner`: orchestrates the experiment for a single matrix size, calling the basic, executor-based, and parallel stream kernels in sequence and timing them with `System.nanoTime()`. It is also the entry point that loops over $n \in \{256, 512, 1024\}$ and thread counts (`1`, approximate physical cores, and `Runtime.getRuntime().availableProcessors()`), printing results and writing them to java_results.csv. Access to the CSV writer is wrapped in a small synchronized method to illustrate the use of synchronized for serialized access to a shared resource.

This design keeps the core matrix kernels separate from the benchmarking logic and exercises different Java concurrency features (executors, parallel streams, atomic variables, semaphores, and the synchronized keyword).

3.4 Python Implementation (PyCharm)

The Python project is structured as a package `matrix_benchmark` with the following modules:

- `core/matrix_utils.py`: defines the Matrix type as List[List[float]], allocation and random initialization routines, a function to compute elapsed time from time.perf_counter(), and a helper to detect the number of logical CPUs.
- `core/basic_multiplier.py`: pure-Python baseline implementation of the triple loop on nested lists.
- `core/parallel_multiplier.py`: parallel variant using concurrent.futures.ProcessPoolExecutor. Each process receives a contiguous block of rows and computes the corresponding slice of C, which is then merged by the parent process. Multiprocessing is chosen instead of threads to circumvent the limitations imposed by the Python Global Interpreter Lock (GIL) on CPU-bound workloads.
- `core/vectorized_multiplier.py`: vectorized implementation based on NumPy arrays. The lists-of-lists inputs are converted to numpy.ndarray and the matrix product is computed using the @ operator, which internally calls optimized BLAS routines.
- `benchmark.py`: main script that iterates over matrix sizes $n \in \{128, 256, 512\}$ and process counts (1, approximate physical cores, and all logical CPUs), times each variant, and writes the results to results/python_results.csv.

The Python baseline is intentionally kept simple and unoptimized, serving as a reference to contrast the impact of multiprocessing and NumPy-based vectorization.

3.5 Benchmarking Procedure

All three implementations follow the same high-level benchmarking protocol:

1. For a given matrix size n, two random $n \times n$ matrices A and B are generated with values in $[0, 1)$ using a fixed random seed.

2. The sequential baseline kernel is executed once and timed. Its execution time T_{basic} serves as the reference for speedup and efficiency calculations.

3. For C and Java, parallel kernels are executed for multiple thread counts. For Python, the parallel kernel is executed for multiple process counts. Each run is timed, and the speedup and efficiency are computed inside the program.

4. Vectorized variants (SIMD in C, NumPy in Python) are executed and timed as well. In C, the combined parallel + SIMD kernel is also tested for the same thread counts as the purely parallel version.

5. For each combination of (n, p), a CSV row is appended containing:

    - matrix size n,
    - number of threads or processes p,
    - execution times of all variants,
    - speedup and efficiency values for parallel methods.

The C and Java benchmarks use one run per configuration, which is justified by the relatively stable timings observed on the target machine. The Python benchmark also performs single runs; for future work, the framework could be extended to perform multiple repetitions and compute averages and standard deviations.

3.6 Hardware and Software Environment

All experiments were performed on a laptop equipped with Intel Core i7 CPU with 6 physical cores and 12 hardware threads (Hyper-Threading enabled), 16 GB of RAM, macOS on x86_64 architecture.

The software environment consisted of:

- C implementation compiled with Apple Clang and CMake in Release mode with `-O3 -march=native -pthread`,

- Java implementation executed on OpenJDK/JBR 17,

- Python implementation using Python 3.11 and NumPy installed via `pip`.

4 Experiments and Results

This section summarizes the main experimental results obtained from the CSV files generated by each implementation. We focus on the three matrix sizes tested in all languages: $n = 256, 512, 1024$ in C and Java, and $n = 128, 256, 512$ in Python.

4.1 C Implementation: Parallel and SIMD Behavior

Table 1 reports the measured times and derived metrics for the C implementation. Each row corresponds to a matrix size and number of threads, and includes the times of the four kernels (basic, parallel, SIMD, and parallel+SIMD) as well as speedup and efficiency values.

Table 1: C implementation: execution time and metrics (milliseconds).

n	Th.	Basic	Par.	SIMD	Par+SIMD	S_{par}	S_{simd}	$S_{\text{p+v}}$	E_{par}	$E_{\text{p+v}}$
256	1	3.3310	2.6950	3.3000	3.8340	1.2360	1.0094	0.8688	1.2360	0.8688
256	6	3.3310	1.0440	3.3000	0.9260	3.1906	1.0094	3.5972	0.5318	0.5995
256	12	3.3310	0.7360	3.3000	0.8920	4.5258	1.0094	3.7343	0.3772	0.3112
512	1	24.0060	20.7830	22.7000	22.7670	1.1551	1.0575	1.0544	1.1551	1.0544
512	6	24.0060	5.1650	22.7000	5.1620	4.6478	1.0575	4.6505	0.7746	0.7751
512	12	24.0060	4.2270	22.7000	4.1770	5.6792	1.0575	5.7472	0.4733	0.4789
1024	1	354.0280	339.1950	284.8170	369.2670	1.0437	1.2430	0.9587	1.0437	0.9587
1024	6	354.0280	58.9730	284.8170	47.4180	6.0032	1.2430	7.4661	1.0005	1.2444
1024	12	354.0280	31.3820	284.8170	34.1260	11.2812	1.2430	10.3741	0.9401	0.8645

For $n = 1024$, the SIMD-only version is about 1.24× faster than the baseline, while the purely parallel version reaches a speedup of 6.0× with 6 threads and 11.3× with 12 threads. The combined parallel+SIMD kernel achieves the highest speedup, 7.47× with 6 threads and 10.37× with 12 threads, with efficiencies close to 1.0 at 6 threads and slightly below 0.9 at 12 threads. These results confirm nearly ideal scaling up to the number of physical cores and diminishing returns when exploiting Hyper-Threading.

For smaller matrices ($n = 256$ and $n = 512$), speedups are more modest because fixed overheads such as thread creation and scheduling represent a larger fraction of the total runtime, but the same qualitative behavior is observed.

4.2 Java Implementation: Executors and Parallel Streams

Table 2 summarizes the Java results. For each matrix size and thread count, the table reports the time of the basic implementation, the `ExecutorService`-based implementation, and the parallel stream version, together with the measured speedups and efficiencies.

Table 2: Java implementation: execution time and metrics (milliseconds).

n	Th.	Basic	Exec	Stream	S_{exec}	S_{stream}	E_{exec}	E_{stream}
256	1	27.2351	11.5850	41.7148	2.3509	0.6529	2.3509	0.0544
256	6	27.2351	32.9982	41.7148	0.8254	0.6529	0.1376	0.0544
256	12	27.2351	4.2159	41.7148	6.4601	0.6529	0.5383	0.0544
512	1	159.2229	109.6215	19.9430	1.4525	7.9839	1.4525	0.6653
512	6	159.2229	21.5586	19.9430	7.3856	7.9839	1.2309	0.6653
512	12	159.2229	17.8668	19.9430	8.9117	7.9839	0.7426	0.6653
1024	1	893.7109	728.1024	107.1709	1.2275	8.3391	1.2275	0.6949
1024	6	893.7109	101.3242	107.1709	8.8203	8.3391	1.4701	0.6949
1024	12	893.7109	89.0881	107.1709	10.0318	8.3391	0.8360	0.6949

For $n = 1024$, the executor-based implementation (Exec) reaches a speedup of approximately 8.8× with 6 threads and 10.0× with 12 threads. The efficiency is slightly super-linear for 6 threads (1.47), likely due to better cache utilization compared to the baseline, and remains reasonably high (0.84) at 12 threads.

The parallel stream version is fixed to the JVM's common fork-join pool and achieves speedups of about 8.3× for $n = 1024$, with an efficiency around 0.69 relative to the number of hardware threads. For $n = 256$, however, parallelization overhead dominates and both the executor and stream versions can be slower than the baseline, illustrating that multi-threading is not always beneficial for small problem sizes.

4.3 Python Implementation: Multiprocessing and NumPy

Table 3 collects the Python results for matrix sizes $n \in \{128, 256, 512\}$ and process counts 1, 6, and 12. For each configuration, the table reports the times of the basic pure-Python implementation, the multiprocessing-based parallel version, and the NumPy vectorized version, together with speedups and efficiency.

Table 3: Python implementation: execution time and metrics (milliseconds).

n	Proc.	Basic	Parallel	Vectorized	S_{par}	S_{vec}	E_{par}
128	1	122.9910	130.6607	2.1082	0.9413	58.3385	0.9413
128	6	122.9910	333.3002	2.1082	0.3690	58.3385	0.0615
128	12	122.9910	446.9324	2.1082	0.2752	58.3385	0.0229
256	1	1010.4668	973.9902	9.0169	1.0375	112.0636	1.0375
256	6	1010.4668	423.1522	9.0169	2.3880	112.0636	0.3980
256	12	1010.4668	647.2096	9.0169	1.5613	112.0636	0.1301
512	1	8422.9158	8290.9791	27.6205	1.0159	304.9517	1.0159
512	6	8422.9158	1932.0294	27.6205	4.3596	304.9517	0.7266
512	12	8422.9158	2203.8062	27.6205	3.8220	304.9517	0.3185

For small matrices ($n = 128$), multiprocessing overhead dominates and the parallel implementation is slower than the baseline for all process counts, with speedups below 1 and very low efficiencies. At $n = 256$, the parallel version becomes beneficial: with 6 processes, the speedup is about 2.39× and the efficiency around 0.40, while 12 processes provide a smaller speedup due to increased overhead.

For the largest size $n = 512$, multiprocessing achieves a speedup of 4.36× with 6 processes (efficiency 0.73) and 3.82× with 12 processes (efficiency 0.32). In contrast, the NumPy-based vectorized kernel obtains dramatic speedups for all sizes: about 58× for $n = 128$, 112× for $n = 256$, and 305× for $n = 512$. These results clearly show that vectorization via optimized native libraries is far more effective than pure-Python parallelism for dense numerical kernels.

4.4 Cross-Language Comparison

Using the baseline implementations as a reference, the maximum speedups observed in this study can be summarized as follows:

- In C, the parallel+SIMD variant reaches speedups of about 7.5× (6 threads) and 10.4× (12 threads) for $n = 1024$, with efficiencies close to 1 up to the number of physical cores.
- In Java, the executor-based implementation attains speedups of 8.8× (6 threads) and 10.0× (12 threads) at $n = 1024$, while the parallel stream implementation achieves around 8.3×.
- In Python, the best multiprocessing speedup is 4.36× for $n = 512$ with 6 processes, whereas the NumPy vectorized kernel provides speedups in the range 58×–305× over the pure-Python baseline, depending on the matrix size.

These data confirm that: (1) C offers the highest absolute performance and excellent scalability; (2) Java provides competitive parallel speedups with higher-level abstractions; and (3) Python is heavily reliant on vectorized native libraries to be competitive in dense linear algebra.

5 Discussion

The experiments confirm the expected hierarchy of performance among the three languages and highlight the complementary roles of thread-level and data-level parallelism. The two figures included in this section help visualize both the scaling behavior of the C implementation and the large gap in baseline performance between C, Java, and Python.

Figure 1 shows the speedup of the C implementation for the parallel and parallel+SIMD variants as a function of the number of threads for the three matrix sizes considered ($n = 256, 512, 1024$). For $n = 512$ and especially $n = 1024$, the speedup grows almost linearly up to six threads, which corresponds to the six physical cores of the processor. At $n = 1024$ the purely parallel version reaches a speedup slightly above 11× with 12 threads, while the combined parallel+SIMD kernel achieves a speedup of about 10×. The curves are close, indicating that multi-threading is the dominant source of acceleration and that SIMD provides an additional but more moderate benefit once the computation starts to be limited by memory bandwidth.

For the smallest problem size ($n = 256$) the theoretical advantages of SIMD are less visible: fixed costs such as thread creation, cache warm-up, and synchronization represent a larger fraction of the total runtime, and the amount of work per thread is small. This explains why the curves cross for $n = 256$, and why the measured speedups are lower than for larger matrices despite using the same number of threads.

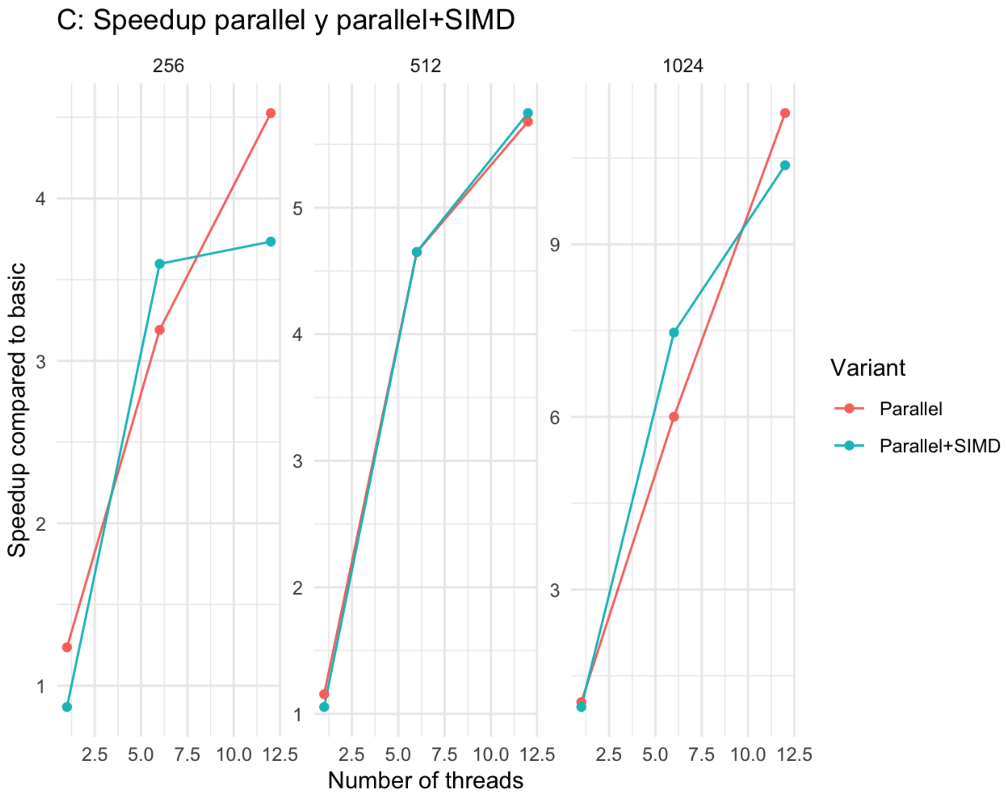

Figure 1: Speedup of the parallel and parallel+SIMD variants in C for different matrix sizes and thread counts.

In C, the sequential baseline benefits from aggressive compiler optimizations and contiguous data layouts. Introducing multi-threading with pthreads allows the program to exploit all available cores with relatively low overhead, leading to speedups close to the ideal p-fold improvement up to the number of physical cores. The use of AVX intrinsics further accelerates the computation within each thread, but the gain is smaller than the theoretical SIMD width due to cache effects and the fact that the algorithm is partly memory-bound for large n. For $n = 1024$, the parallel+SIMD variant improves performance by a factor of 7.47× with 6 threads and 10.37× with 12 threads, illustrating both near-linear parallel scaling and the additional benefit of vectorization.

In Java, the presence of a managed runtime and garbage collection adds some overhead, but the just-in-time compiler can still optimize hot loops effectively. Executor-based parallelism provides a flexible and expressive way to partition work across threads, while `AtomicInteger` and `Semaphore` offer explicit control over synchronization and task coordination. Parallel streams provide a more declarative interface but less control over thread allocation and scheduling. The results show that, for small matrices such as $n = 256$, parallelization overheads can even degrade performance; however, for $n = 1024$, the executor implementation reaches speedups of around 10×, comparable to the C implementation in relative terms, although with higher absolute runtimes.

Figure 2 compares the execution time of the basic (sequential, non-vectorized) version in C, Java, and Python as a function of the matrix size. The vertical axis is in logarithmic scale, so the approximately straight lines observed for all three languages are consistent with the expected $O(n^3)$ time complexity of the naive algorithm. However, the vertical separation between the curves is substantial: the C curve is always at the bottom, Java is roughly one order of magnitude slower for the largest matrix size, and Python is already two orders of magnitude slower than C at $n = 512$. This figure visually reinforces the idea that language and runtime design have a first-order impact on raw numerical performance, even before adding any form of parallelism or vectorization.

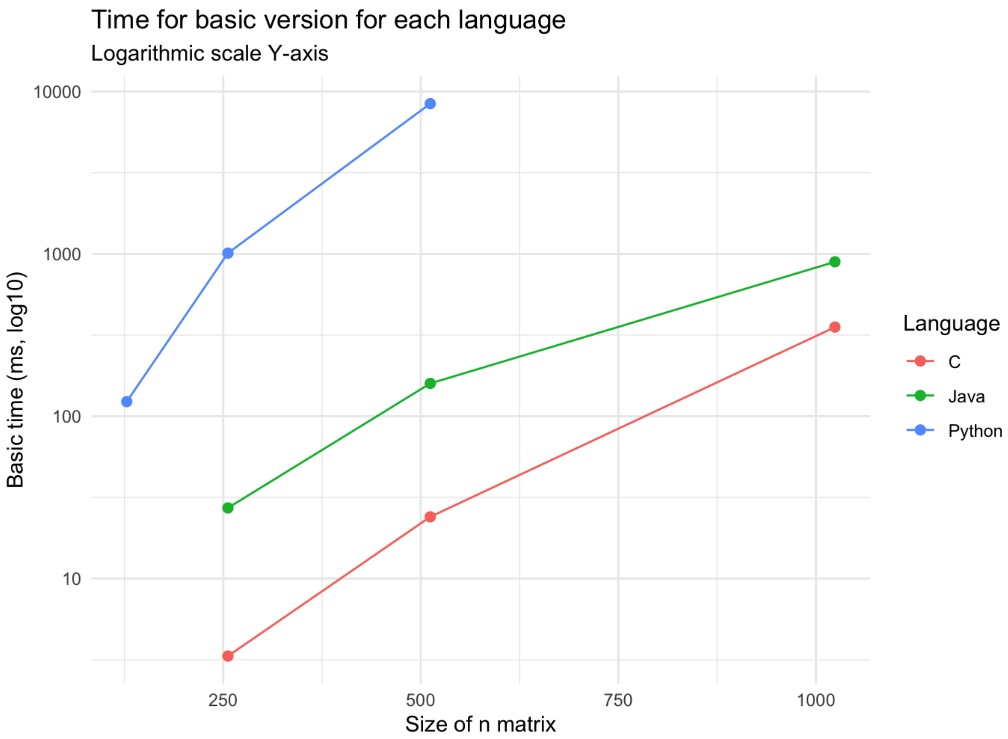

Figure 2: Execution time of the basic matrix multiplication implementation in C, Java, and Python as a function of matrix size (logarithmic scale on the Y-axis).

In Python, the interpreter and dynamic type system impose a substantial cost on inner loops involving floating-point arithmetic. As a result, pure-Python implementations of $O(n^3)$ algorithms are not competitive for large matrices: the baseline takes about 8.4 seconds for $n = 512$, versus milliseconds in C and Java. Multiprocessing can utilize multiple CPU cores, but at the price of high overhead for process management and data transfer. The observed speedups of 2.4×–4.4× show that some benefit is possible, especially for the largest size, but with significantly lower efficiency than in C or Java.

The NumPy-based vectorized kernel demonstrates a different approach: rather than parallelizing Python code, it offloads the heavy lifting to pre-compiled native libraries, effectively bypassing the limitations of the interpreter. This yields the largest speedups of the entire study (up to 305× for $n = 512$), and brings Python's absolute performance closer to that of C for the tested sizes.

Another important observation is that parallel and vectorized speedups do not combine multiplicatively. Even in C, where both mechanisms are exploited explicitly, the combined parallel+SIMD speedup is limited by memory bandwidth and cache behavior. Once the arithmetic throughput is high enough, the bottleneck shifts from computation to data movement.

Finally, the experiments illustrate that measuring parallel efficiency is as important as raw speedup. Configurations that achieve high speedup with very large numbers of threads or processes may still have low efficiency, meaning that additional cores are underutilized due to synchronization or communication overhead.

6 Conclusions

This project implemented and evaluated parallel and vectorized dense matrix multiplication in three languages, C, Java, and Python, on a multi-core Intel laptop.

The main conclusions are:

- **C** provides the highest absolute performance and the most efficient use of both thread-level and data-level parallelism. The combination of pthreads and AVX intrinsics achieves substantial speedups over the sequential baseline, with efficiencies close to one up to the number of physical cores and useful speedups even when Hyper-Threading is used.

- **Java** offers a good balance between performance and developer productivity. Executor-based parallelism scales well and requires less boilerplate than manual thread management in C. Parallel streams offer an even higher-level abstraction, at the cost of slightly higher overhead and less control over the underlying thread pool. For $n = 1024$, both approaches achieve speedups in the range of $8\times$–$10\times$ relative to the baseline.

- **Python** is not competitive for dense $O(n^3)$ kernels when implemented in pure Python, but becomes effective when combined with vectorized libraries such as NumPy. Multiprocessing can provide some speedup but is limited by overhead; the most practical approach is to delegate heavy computations to native code and use Python as a coordination layer.

- Parallel and vectorized speedups are ultimately constrained by memory bandwidth and cache behavior. Increasing the number of threads or the SIMD width yields diminishing returns once the computation becomes memory-bound.

From a pedagogical perspective, this project illustrates how the same algorithmic kernel can be adapted to different parallel programming models: low-level threads and intrinsics in C, high-level concurrency libraries and streams in Java, and multiprocessing plus vectorization in Python. It also emphasizes the importance of measuring not only execution time but also speedup and efficiency when evaluating parallel programs.

7 Future Work

Several extensions of this work are possible:

- **Blocked algorithms and cache optimization.** The current implementation uses a straightforward triple loop. Introducing blocked (tiled) algorithms that operate on sub-matrices sized to fit into L1 or L2 cache could significantly improve data locality and reduce memory traffic, especially for large n.
- **Higher-level parallel frameworks.** In C, using OpenMP pragmas instead of manual pthread management would simplify the code and allow easier experimentation with different scheduling policies. In Java, the new Vector API could provide explicit SIMD operations, and in Python, libraries such as `numba` or `joblib` could be used to JIT-compile inner loops or parallelize multiple runs.
- **GPU acceleration.** Offloading matrix multiplication to GPUs using CUDA, OpenCL, or high-level libraries (e.g. cuBLAS) would enable the comparison between multi-core CPU and massively parallel GPU performance.
- **Extended benchmarking and profiling.** A natural extension is to perform multiple repetitions per configuration, compute confidence intervals for the measured times, and use profiling tools to analyze cache misses, branch mispredictions, and vectorization effectiveness in more detail.

Repository and Data Availability

All source code, benchmark results, and this paper are publicly available at:
`https://github.com/data-dmt/BigData/tree/main/Individual_Assignment/ParallelizationBenchmark`

Repository contents:

- `paper/` folder containing the LaTeX source and the compiled PDF of this document.
- `code/c/` folder with the CLion project, including `matmul.c/h`, `main.c`, CMake configuration, and the generated `results.csv`.
- `code/java/` folder with the Maven project `ulpgc.shared.matrix`, including all multiplier classes and `results.csv`.
- `code/python/` folder with the PyCharm project `matrix_benchmark`, including the `core` package, `benchmark.py`, and `results/python_results.csv`.

The repository will allow complete reproduction of the experiments described in this paper, subject to differences in hardware and operating system.

References

[1] G. H. Golub and C. F. Van Loan, *Matrix Computations.* Johns Hopkins University Press, 4th ed., 2013.

[2] Python Software Foundation, “Python documentation — time module,” 2024. Available: `https://docs.python.org/3/library/time.html`.

[3] Oracle Corporation, “Java platform, standard edition — api specification: `System.nanoTime`,” 2024. Available: `https://docs.oracle.com/en/java/`.

[4] Intel Corporation, “Intel 64 and IA-32 architectures optimization reference manual,” 2024. Available: `https://www.intel.com`.

[5] OpenMP Architecture Review Board, “OpenMP application program interface,” Version 5.2, 2021.

[6] OpenAI, “ChatGPT (GPT-5.1 Thinking),” 2025. Available: `https://chat.openai.com`.