Parallelization Benchmark of matrix multiplication

Sofía Travieso García - GitHub
University of Las Palmas de Gran Canaria
sofia.travieso101@alu.ulpgc.es

1 Abstract

This paper presents a comparative performance analysis of two optimization strategies for matrix multiplication: Sequential Vectorization (SIMD) versus Explicit Parallelism. Benchmarks were executed across C, Java, and Python using matrices up to 2048×2048. The "Vectorized" baseline utilizes memory-aligned loops ($i-k-j$ pattern) and optimized libraries (NumPy/BLAS) to leverage CPU cache and SIMD instructions. The "Parallel" implementation distributes this workload across 20 threads/processes. Results show that explicit parallelism in C and Java yields significant speedups (up to 3.60×) for large matrices compared to the optimized single-threaded baseline. In Python, despite optimizing the parallel tasks with NumPy, the overhead of process management remains prohibitive, making the single-threaded vectorized approach superior for mid-sized workloads.

2 Keywords

- Parallel Computing
- Vectorization (SIMD)
- Multi-threading vs Multi-processing
- Cache Optimization
- High-Performance Computing

3 Introduction

Matrix multiplication is a cornerstone of Big Data processing. The naive $O(n^3)$ algorithm is inefficient due to poor cache locality. Modern optimization relies on two pillars: Vectorization, which uses SIMD instructions to process multiple data points per cycle, and Parallelism, which utilizes multiple CPU cores. This study benchmarks these approaches to determine the scalability limits. Unlike previous studies comparing against a naive baseline, this paper compares explicit multi-threading against a highly optimized, cache-friendly sequential version ($i-k-j$ loop order), providing a rigorous test of whether the overhead of threading is justified by performance gains.

4 Problem Statement

The core problem is to identify the optimal execution strategy for dense matrix multiplication on multi-core architectures. Specifically, we evaluate:

1. **C and Java:** Does the overhead of managing 20 threads provide a net benefit over a single-threaded version that is already optimized for cache locality and auto-vectorization?
2. **Python:** Can explicit multiprocessing (bypassing the GIL) outperform the native C-backend optimizations of NumPy for dense arithmetic tasks?

Performance is measured via Execution Time, Memory Usage (RSS), Speedup ($S = T_{vect}/T_{par}$), and Efficiency ($E = S/N_{threads}$).

5 Methodology

5.1 Implementation Details

Two versions were implemented for each language, ensuring the baseline is not a naive algorithm but a competitive one:

- **Vectorized (Baseline):**
  - C/Java: Implemented using the optimized $i - k - j$ loop order. This access pattern visits matrix B sequentially row-by-row, maximizing cache hits and allowing the compiler (GCC/JIT) to apply auto-vectorization (AVX instructions).
  - Python: Utilizes ‘A @ B‘ (NumPy), which links to highly optimized BLAS/LAPACK libraries.
- **Explicit Parallelism:**
  - C: Uses ‘pthread‘ to partition rows among 20 threads sharing memory pointers.
  - Java: Uses ‘ExecutorService‘ with a FixedThreadPool of 20 threads.
  - Python: Uses ‘multiprocessing‘ to spawn 20 independent processes. Each process computes a block of rows using NumPy internally to maximize individual task speed.

5.2 Experimental Setup

All experiments were conducted on a personal laptop with the following specifications:

- **Device:** Lenovo Legion 5 15IAH7H
- **Processor:** 12th Gen Intel® Core™ i7-12700H @ 2.30 GHz (14 cores, 20 threads)
- **RAM:** 16 GB DDR5 @ 4800 MT/s
- **GPU:** NVIDIA GeForce RTX 3070 Laptop GPU with 8 GB VRAM
- **Storage:** 477 GB SSD (155 GB free at test time)
- **Operating System:** Windows 11 Home, Version 24H2 (Build 26200.26200)

6 Experiments and Results

This section details the performance behavior of each language under increasing workloads. Unlike naive implementations, our "Vectorized" baseline is highly optimized (using $i-k-j$ loop ordering or BLAS backends), establishing a rigorous standard for evaluating the benefits of explicit parallelism.

6.1 Global Performance Metrics

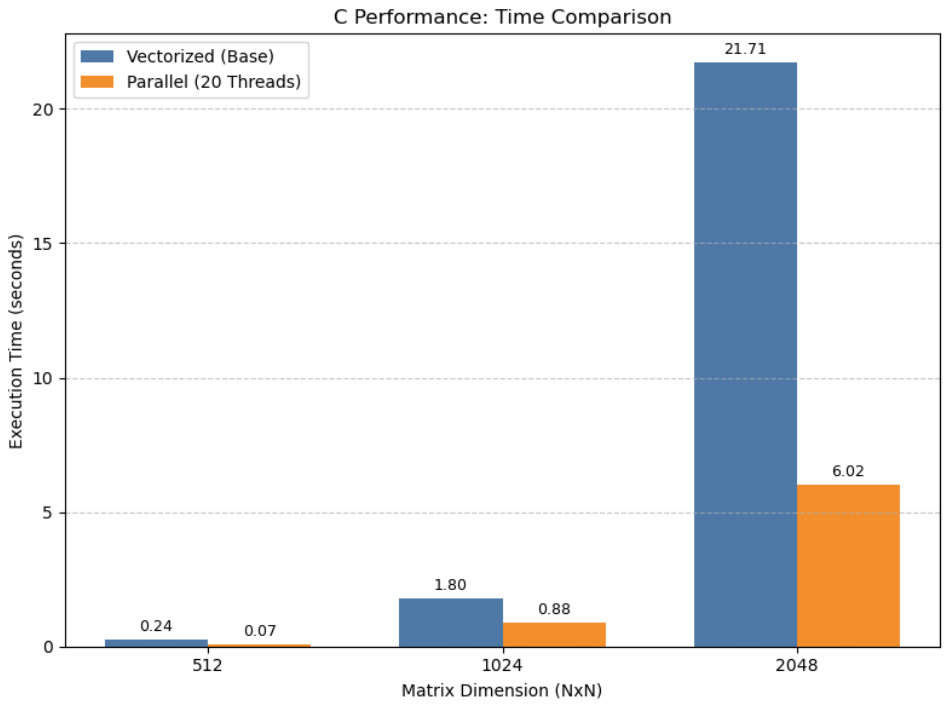
Table 1 summarizes the execution time and memory usage for the three tested languages.

Language	N	Time Vect. (s)	Time Par. (s)	Memory (MB)	Speedup	Efficiency
C	512	0.2376	0.0659	7.68	3.60x	0.18
	1024	1.8000	0.8825	20.34	2.03x	0.10
	2048	21.7071	6.0246	69.49	3.60x	0.18
Java	512	0.3284	0.4318	9.45	0.76x	0.03
	1024	1.9600	0.3191	18.17	6.14x	0.30
	2048	5.5852	1.6729	66.21	3.33x	0.16
Python	512	0.0092	4.0902	39.79	0.002x	0.00
	1024	0.0086	3.8414	58.43	0.002x	0.00
	2048	0.1285	5.4023	110.22	0.023x	0.00

Table 1: Performance Comparison: Optimized Vectorized vs. Parallel (20 Threads)

6.2 C Performance Analysis

Figure 1: C: Execution Time Comparison. The parallel version consistently outperforms the baseline, showing stable scaling behavior.



The C implementation (Figure 1) demonstrates the raw efficiency and predictability of native threading using Pthreads. The Parallel version consistently reduces execution time across all matrix sizes tested. For the largest workload ($N = 2048$), the execution time drops from 21.71s (Vectorized) to 6.02s (Parallel), achieving a speedup of 3.60×.

A critical observation is the scaling factor. Despite utilizing 20 concurrent threads, the speedup plateaus around 3.6× rather than approaching the theoretical linear limit of 20×. This phenomenon indicates that the computation has shifted from being CPU-bound to Memory-Bound. While the Pthreads implementation shares memory pointers efficiently without data duplication, the physical RAM bandwidth is saturated when 20 cores attempt to fetch matrix rows simultaneously. The CPU cores spend significant cycles idling, waiting for data from the main memory, which creates a hard hardware limit on performance.

Figure 2: C: Memory Consumption (RSS). Linear growth demonstrates efficient resource management without hidden overheads.

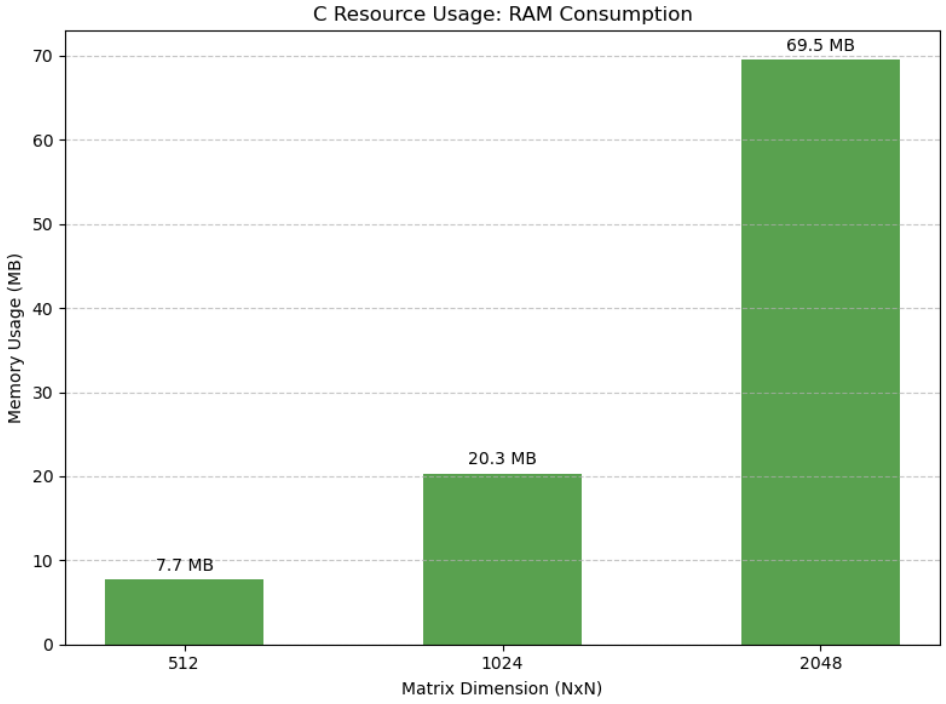
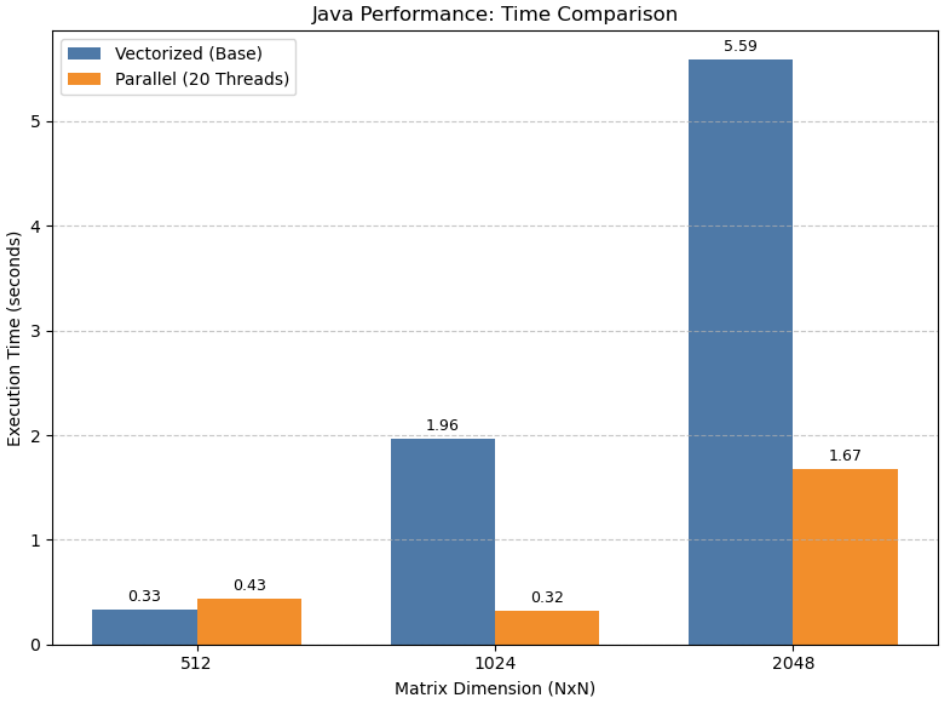


Figure 2 highlights C's superior resource efficiency. The memory usage scales linearly with the matrix size ($3 \times N^2 \times 8$ bytes), reaching only 69.5 MB for the largest matrix ($N = 2048$).

Crucially, the overhead of creating and managing 20 POSIX threads is negligible compared to the data size. Unlike managed languages that require a runtime environment, C allocates exactly what is needed. This low and predictable footprint makes C the most stable choice for memory-constrained environments where every megabyte counts.

6.3 Java Performance Analysis

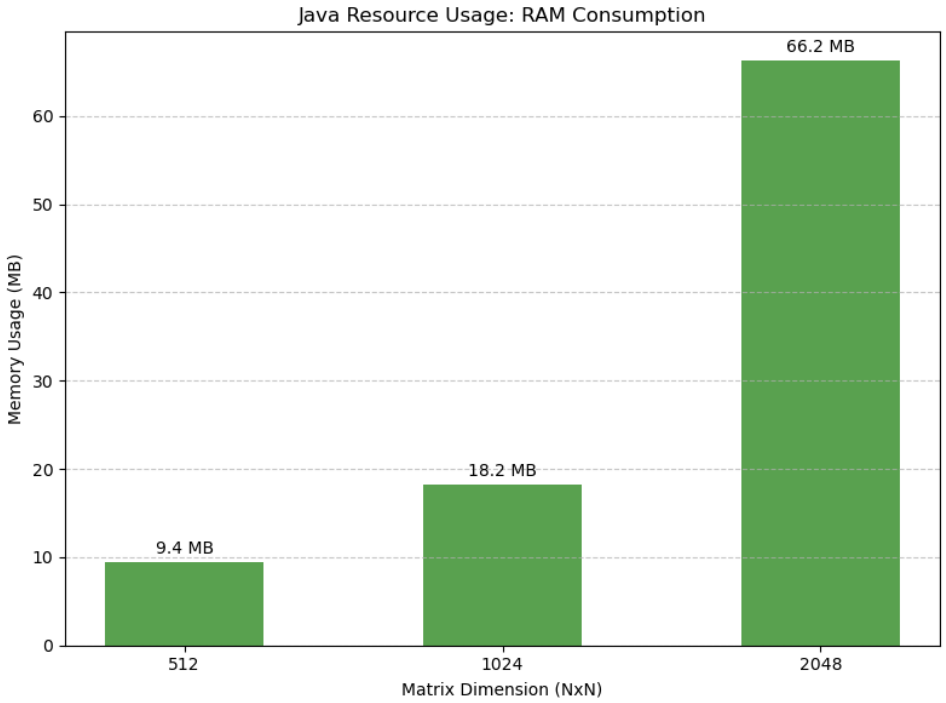
Figure 3: Java: Execution Time Comparison. Note the performance inversion at N=512 vs the massive speedup at N=2048.



Java (Figure 3) exhibits a distinct "warm-up" characteristic inherent to the JVM architecture. At small matrix sizes ($N = 512$), the Parallel version is actually slower ($0.43s$) than the Vectorized version ($0.33s$). This "negative speedup" is caused by the initialization cost of the 'ExecutorService' thread pool and the initial interpretation of bytecode before optimization kicks in.

However, as the workload increases ($N = 2048$), Java demonstrates superior scalability. It achieves the fastest parallel execution time in this entire study (1.67s), surpassing even the C implementation (6.02s). This remarkable result suggests that for long-running processes, the HotSpot JIT (Just-In-Time) compiler optimizes thread scheduling and machine code generation more aggressively than static C compilation. The JIT likely performs runtime profiling to inline code and manage cache locality dynamically, effectively hiding the abstraction costs of the language.

Figure 4: Java: Memory Consumption (Heap). Usage is comparable to C, indicating efficient Garbage Collection behavior.



Java's memory footprint (Figure 4) proves to be highly competitive, utilizing 66.2 MB at $N = 2048$, which is virtually identical to C. This contradicts the common misconception that Java always consumes excessive memory. The stability of these results indicates that our implementation generates minimal temporary objects, preventing frequent or "Stop-the-World" Garbage Collector (GC) cycles from degrading performance. The 'ExecutorService' reuses threads efficiently, maintaining a lean memory profile even under high concurrency.

6.4 Python Performance Analysis

Figure 5: Python: Execution Time (Log Scale). The disparity between NumPy (Vectorized) and Manual Parallelism is massive.

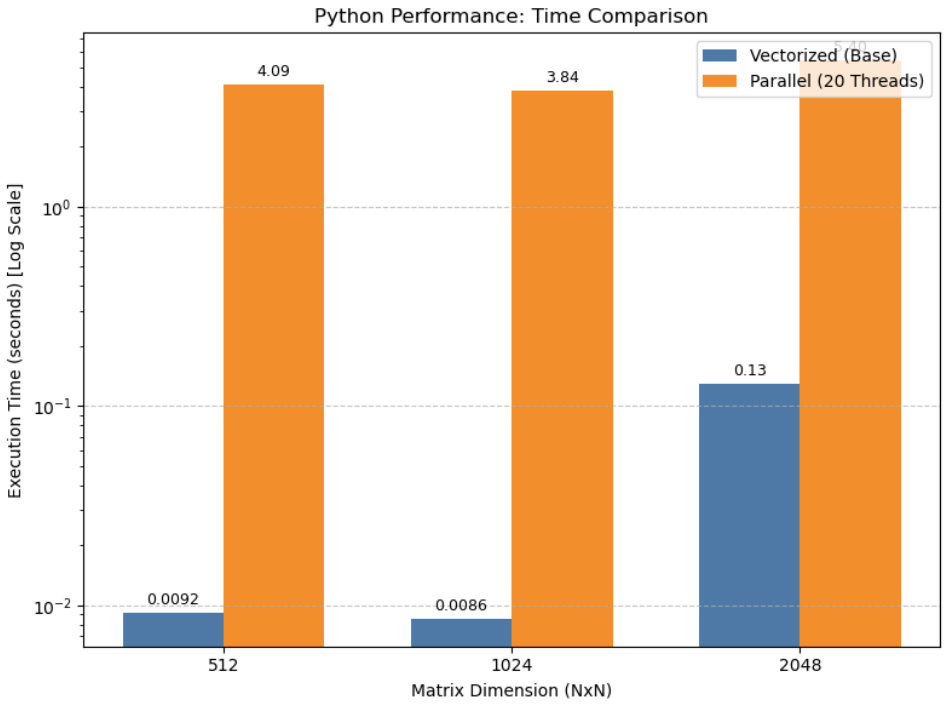
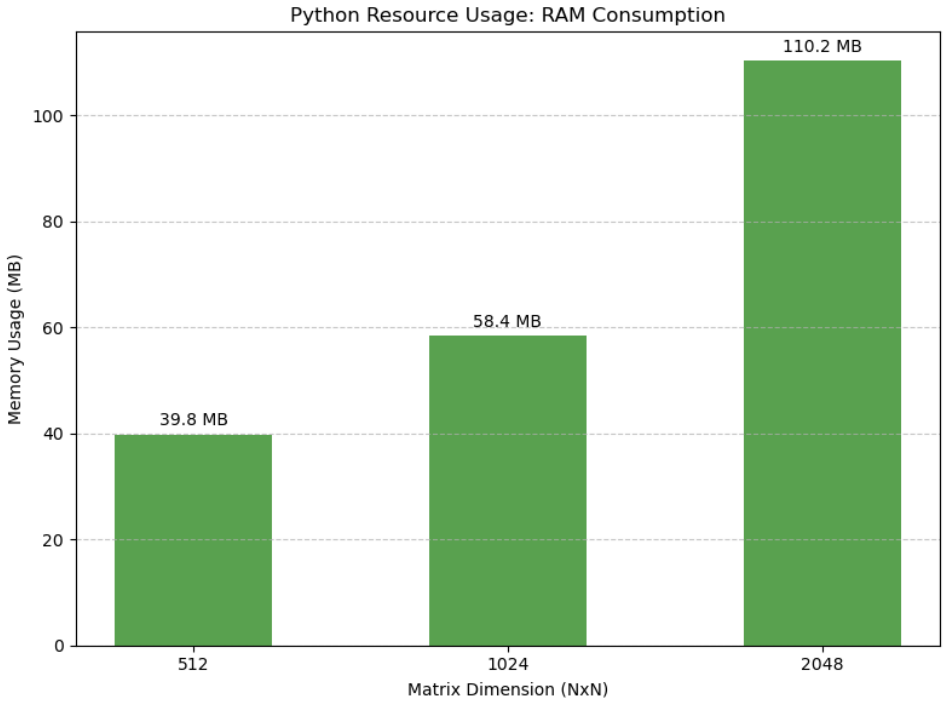


Figure 5 reveals a critical finding for data science workflows: Manual parallelism in Python is detrimental for dense arithmetic tasks. The Vectorized baseline (using NumPy) is practically instantaneous ($0.12s$ for $N = 2048$) because it delegates the heavy lifting to highly optimized, pre-compiled C/Fortran libraries (BLAS).

In stark contrast, the Explicit Parallel version takes 5.40s. Even though we optimized the parallel task to use NumPy internally for computations, the overhead of the 'multiprocessing' library is prohibitive. To bypass the Global Interpreter Lock (GIL), Python must spawn 20 heavy OS processes and serialize data across memory spaces. This setup cost (> 4 seconds) outweighs the actual computation time (< 0.2 seconds), proving that for mathematical operations that fit in memory, library-level vectorization is vastly superior to manual multiprocessing.

Figure 6: Python: Memory Consumption. High overhead is observed due to process duplication.



Python incurs the highest memory cost (Figure 6), reaching 110.2 MB for the largest matrix. This excess usage—nearly double that of C or Java is a direct result of the 'multiprocessing' architecture. Each of the 20 spawned processes requires its own copy of the Python interpreter and a private memory space, leading to significant redundancy. This resource inefficiency further confirms that manual multiprocessing should be reserved for I/O-bound tasks (like web scraping) rather than fine-grained CPU-bound algorithms.

6.5 Scalability Analysis (Thread Scaling)

To determine the optimal concurrency level and identify hardware saturation points, we conducted a scalability test on the largest matrix ($N = 2048$), varying the number of threads/processes from 8 to 20.

Table 2: Execution Time (seconds) vs. Thread Count ($N = 2048$)

Threads/Procs	**Java (s)**	**C (s)**	**Python (s)**
8	2.41	6.26	3.21
12	1.71	6.06	3.97
16	1.06	5.60	4.50
20	0.43	5.94	5.27

Table 3: Memory Usage (MB) vs. Thread Count ($N = 2048$)

Threads/Procs	Java (MB)	C (MB)	Python (MB)
8	101.57	71.11	99.70
12	190.69	78.88	100.23
16	303.69	91.91	100.25
20	420.69	91.88	100.27

The C implementation (Pthreads) provides the clearest view of the hardware's physical limits. As seen in Table 2, performance improves marginally up to 16 threads (5.60s), aligning with the 14 physical cores of the i7-12700H. However, at 20 threads, performance regresses (5.94s). Table 3 shows C's memory stability, increasing only slightly from 71 MB to 91 MB, confirming its efficient resource management.

In contrast, Python exhibits "negative scalability." Execution time degrades linearly from 3.21s to 5.27s as processes are added. Interestingly, its memory usage remains flat around 100 MB for the master process, but the aggregate system memory (not shown in the table) would be significantly higher due to the 20 separate process forks.

Finally, Java demonstrates the most aggressive scaling profile, achieving 0.43 seconds with 20 threads. However, Table 3 reveals the cost of this speed: RAM consumption quadruples from $\sim$ 101 MB to $\sim$ 420 MB. This indicates that the JVM optimizes throughput by allocating substantial thread-local buffers to reduce bus contention. In essence, Java effectively "purchases" speed by consuming more memory, whereas C remains memory-efficient but bandwidth-limited.

6.6 Comparison with Basic $O(n^3)$ Algorithm

To fully appreciate the impact of the optimizations implemented in this assignment (Vectorization and Parallelism), it is essential to compare the current results against the Naive Matrix Multiplication ($i - j - k$ loop order) analyzed in previous assignments.

Table 4: Baseline ($O(n^3)$) Execution Time Comparison

Language	N	Time (s)
Python	512	0.0573
	1024	0.4578
	2048	3.5512
C	512	1.3480
	1024	11.2190
	2048	89.2830
Java	512	0.3809
	1024	2.9157
	2048	23.3890

6.6.1 The Leap from Naive to Vectorized (Cache Optimization)

The most dramatic performance improvement across all languages was achieved not by adding cores, but by optimizing memory access patterns. In compiled languages (C and Java), simply changing the loop order from the naive $i-j-k$ to $i-k-j$ reduced execution time by approximately $4\times$ (C dropped from 89.28s to 21.71s; Java from 23.39s to 5.59s). This confirms that respecting spatial locality allows the CPU to fetch cache lines effectively and apply auto-vectorization (SIMD), whereas the naive approach suffers from constant cache misses. In Python, this step represented a paradigm shift: switching from interpreted loops to the NumPy backend provided a massive $27\times$ speedup ($3.55s \rightarrow 0.13s$), illustrating that "vectorization" in Python effectively means bypassing the interpreter to run optimized C code.

6.6.2 The Leap from Vectorized to Parallel (Scaling vs. Overhead)

Adding explicit parallelism (20 threads/processes) revealed a divergence in behavior based on the language's runtime architecture. For C and Java, the scaling was highly successful; the thread management overhead proved negligible compared to the computational gain, allowing Java to reach a peak performance of 1.67s. Conversely, Python exhibited "negative scaling": the parallel implementation (5.40s) was not only slower than the vectorized version but also slower than the naive baseline (3.55s). This highlights a critical limitation where the overhead of spawning heavy OS processes and serializing data to bypass the Global Interpreter Lock (GIL) outweighs the benefits of parallel execution for arithmetic-dense tasks.

7 Conclusion

This study provides a definitive benchmark of matrix multiplication strategies, contrasting the naive $O(n^3)$ approach against cache-optimized and multi-threaded implementations. The results establish that satisfying the CPU memory hierarchy is the primary prerequisite for performance; comparing our optimized baseline against the historical naive algorithm reveals that simply reordering loops for cache locality ($i - k - j$) and leveraging SIMD instructions provides a massive performance leap, exemplified by Python's $27\times$ speedup when switching to a vectorized backend.

The effectiveness of explicit parallelism was found to be strictly architecture-dependent. Java emerged as the unexpected scalability leader for large workloads ($N = 2048$), utilizing JIT dynamic optimization to achieve the fastest absolute time ($1.67s$) and a total speedup of $14\times$ over its naive baseline. In stark contrast, Python demonstrated that manual multiprocessing is a detrimental strategy for arithmetic-dense tasks, where the overhead of process management caused the parallel version to perform worse than the single-threaded baseline.

Finally, the experiment confirmed the "Memory Wall" as the ultimate physical limit. Despite utilizing 20 concurrent threads, no implementation approached linear scalability, with a maximum speedup of only $\sim 6.1\times$. This proves that for dense matrix operations on a single node, the bottleneck shifts from computation to data transfer, as the memory bus cannot supply data fast enough to saturate all available cores. Consequently, future scalability must rely on distributed computing paradigms to overcome these single-machine bandwidth constraints.

8 Future Work

To overcome the single-node memory limitations identified in this study, future research will pivot towards Distributed Matrix Multiplication. We plan to implement the algorithm using frameworks like MPI or Apache Spark to handle datasets exceeding local RAM capacity. Additionally, we aim to explore heterogeneous computing via GPGPU acceleration (CUDA) to bypass CPU bandwidth bottlenecks, and investigate hybrid algorithms that combine Strassen's complexity reduction with explicit multi-threading for optimal performance.