Stage 2 - Sparse Matrix-Vector Multiplication (SpMV)

Agustín Darío Casebonne

November 19, 2025

Abstract

Following the initial analysis of dense matrix multiplication, this second stage investigates the performance characteristics of **Sparse Matrix-Vector Multiplication (SpMV)** using the **Compressed Sparse Row (CSR)** format. This study shifts the focus from CPU-bound $O(N^3)$ algorithms to memory-bound operations with irregular access patterns. We benchmark implementations in **Rust, C++, Java, and Python** across consistent matrix sizes $N \in \{512, 1024, 1536\}$.

Our findings reveal a distinct trade-off between raw sequential speed and parallel scalability. While **Rust achieved the fastest sequential execution time** at the largest tested size ($N = 1536$), **C++** demonstrated superior parallel scaling, being the only language to achieve positive speedup ($> 1.0x$) within the tested range.

1 Introduction

Matrix multiplication remains a foundational operation in high-performance computing. However, many real-world matrices are sparse. The standard dense approach ($O(N^3)$) is computationally wasteful for these cases. This assignment implements the **CSR (Compressed Sparse Row)** format, which reduces storage and complexity to $O(N_{nz})$ (where N_{nz} is the number of non-zero elements).
Link to repository: https://github.com/Agustin-Casebonne/Individual-Assignments

1.1 Objectives

1. **Implementation:** Develop the SpMV kernel ($y \leftarrow y + Ax$) using CSR data structures in C++, Java, Python, and Rust.
2. **Benchmarking:** Execute parametrized runs to measure sequential vs. parallel performance.
3. **Overhead Analysis:** Quantify the cost of thread management and memory footprint.

2 Methodology

2.1 Experimental Setup

- **Algorithm:** SpMV using CSR format.
- **Sparsity:** Randomly generated matrices with approximately **5% density**.
- **Matrix Sizes (N):** $N \in \{512, 1024, 1536\}$.
- **Number of Runs (R):** Each experiment consists of $R = 5$ runs.

3 Results and Analysis

3.1 Performance Comparison

3.1.1 Sequential Execution

The following table aggregates the **Sequential** execution times. This highlights the raw efficiency of memory management and pointer arithmetic.

Table 1: Sequential Execution Times (seconds)

Language	N=512	N=1024	N=1536
Rust	**0.000018 s**	**0.000082 s**	**0.000171 s**
C++	**0.000017 s**	**0.000047 s**	0.000295 s
Java	0.000376 s	0.000195 s	0.000222 s
Python	0.000801 s	0.002694 s	0.006016 s

3.1.2 Parallel Execution

The table below shows the **Parallel** execution times using 4 threads.

Table 2: Parallel Execution Times (seconds)

Language	N=512	N=1024	N=1536
Rust	0.000481 s	0.000241 s	0.000320 s
C++	0.006950 s	**0.000044 s**	**0.000152 s**
Java	0.001669 s	0.000389 s	0.000359 s
Python	0.256507 s	0.389220 s	0.678378 s

Parallel Analysis:

- **C++ Dominance:** C++ was the only language to scale effectively, achieving a speedup of **1.94x** at $N = 1536$.
- **Rust Safety Overhead:** Despite being fast sequentially, Rust's parallel implementation struggled to surpass sequential speeds (Speedup 0.54x at $N = 1536$). This is due to the cost of memory safety checks (‘Arc‘/‘Mutex‘) in micro-latency tasks.
- **Python/Java Overhead:** The massive overhead of spawning threads/processes in managed languages resulted in negative scaling (parallel was slower than sequential).

3.2 Memory and CPU Usage Analysis

In this sparse stage, memory footprint is a critical metric. The following table summarizes the resource consumption observed during the largest benchmark ($N = 1536$).

Table 3: Resource Usage at N=1536

Language	Memory (MB)	CPU Usage	Notes
Rust	**3 MB**	N/A	Minimal footprint, aligns with theoretical CSR.
C++	$\approx$ 5 MB	N/A	Efficient raw pointer allocation.
Java	11 MB	6.3%	JVM Heap overhead.
Python	25 MB	79.3%	Significant `PyObject` overhead.

Memory Efficiency

Rust demonstrated the highest memory efficiency (3 MB), closely matching the theoretical minimum required to store the CSR structure ($2 \cdot N_{nz} + N_{rows}$). Java and Python consumed significantly more memory (up to **8x more** in Python) due to object headers, garbage collection metadata, and dynamic typing wrappers.

CPU Utilization

- **Python (High Usage, Low Speed):** Python showed high CPU usage (≈79%) but poor performance. This indicates "thrashing," where the CPU spends more time switching contexts and managing the Global Interpreter Lock (GIL) than performing actual calculations.
- **Java (Low Usage):** Java showed very low CPU usage (6.3%), suggesting that for such small matrices, the threads spent most of their time waiting for synchronization or initialization rather than computing.
- **Systems Languages:** Rust and C++ utilized the CPU efficiently during the brief computation windows, minimizing idle time.

4 Repository

I have uploaded all the code to the same repository where Stage 1 was previously uploaded. For better organization, I created two separate folders within the "Data" directory, each containing the results of its respective stage.

5 Conclusion

This comparative study leads to the following conclusions:

1. **Sequential Performance: Rust is the winner**, demonstrating the fastest raw execution time at $N = 1536$.
2. **Parallel Scalability: C++ is the winner**. It was the only language capable of effectively amortizing thread overhead at $N = 1536$.
3. **Efficiency:** Rust and C++ provide massive memory savings (3-5 MB) compared to Python (25 MB), validating their use for large-scale sparse linear algebra.