Deliverable 2: Distributed SpMV

Sepa Matteo (243283)
Course: Introduction to Parallel Computing (2025–2026)
University of Trento
matteo.sepa@studenti.unitn.it
github.com/sepamatteo/PARCO-Computing-2026-243283

***Abstract*—This report presents a distributed implementation of Sparse Matrix-Vector Multiplication (SpMV) using MPI for inter-process communication and OpenMP for intra-process parallelism. We implement a 1D cyclic row distribution for the matrix and cyclic column distribution for the vector, with ghost exchanges via `MPI_Alltoallv` to handle non-local accesses. Experiments on SuiteSparse matrices demonstrate scalability, load balance, and communication overhead on a multi-node system. Results show up to 114% efficiency with 8 processes, highlighting the impact of optimizations like precomputed ghost maps.**

***Index Terms*—SpMV, sparse matrices, OpenMP, parallel computing, CSR, COO, performance analysis, MPI, distributed system, High performance cluster.**

I. Introduction

Sparse Matrix-Vector Multiplication (SpMV) is a fundamental kernel in scientific computing, used in applications like graph analytics, machine learning, and simulations [8]. In distributed-memory systems, SpMV requires careful data partitioning to minimize communication while balancing computation. This deliverable extends shared-memory SpMV (from Deliverable 1) to a distributed setting using MPI.

Our approach uses cyclic row distribution for the CSR matrix and cyclic column distribution for the vector x, necessitating ghost exchanges for non-local vector elements. Key contributions include: (1) efficient ghost communication via two-phase Alltoallv, (2) hybrid MPI+OpenMP for scalability, and (3) performance metrics collection for load balance and overhead analysis.

The report is structured as follows: Section II details the methodology, Section III describes experiments, Section IV presents results, and Section V concludes.

II. Methodology

A. Sparse Storage Formats

1) Coordinate (COO): Suite Sparse matrix files are stored in COO format which will require a CSR conversion. The COO format stores each non-zero element as a triplet (`r, c, v`), where `r` is the row index, `c` is the column index, and `v` is the value. These triplets are stored in three separate arrays: `row_idx`, `col_idx`, and `values`.

2) Compressed Sparse Row (CSR): The CSR format uses three arrays: `values`, `col_idx`, and `row_ptr`. The `values` and `col_idx` arrays store the non-zero values and their corresponding column indices, ordered by row. The `row_ptr` array stores the starting index of each row in the `values` and `col_idx` arrays. The SpMV operation in CSR format involves iterating through each row `i`, and for each row, iterating through its non-zero elements from `row_ptr[i]` to `row_ptr[i+1]-1`.

Our implementation includes a COO-to-CSR conversion step. This involves counting the number of non-zero elements in each row, computing the prefix sum to determine the row pointers, and then populating the `values` and `col_idx` arrays.

B. Parallel Design Strategy

We applied Foster's [18] PCAM (Partitioning, Communication, Agglomeration, Mapping) design methodology to structure the parallelization of the SpMV kernel.

1) Partitioning: The computation was decomposed using domain decomposition. The finest granularity of work is the computation of a single element y_i of the result vector, which corresponds to the dot product of the i-th matrix row and the vector x. Since the matrix is sparse, the computational cost per task varies depending on the number of non-zero elements (nnz) in that row.

2) Communication: The tasks are not independent; computing row i requires access to specific elements of vector x corresponding to the column indices of non-zeros in that row. If the required x element resides on a different processor, communication is necessary. In a worst-case sparse matrix, this could imply all-to-all communication. We minimize latency by pre-identifying these non-local dependencies (ghost nodes) and batching them into fewer messages.

3) Agglomeration: To reduce communication overhead and leverage locality, we agglomerated fine-grained tasks (rows) into larger chunks.

- **MPI Level:** Rows are assigned to MPI processes using a *cyclic distribution*. While block distribution preserves locality, cyclic distribution was chosen to better balance the load for matrices with uneven row densities (e.g., power-law graphs), ensuring no single rank is burdened with all heavy rows.
- **OpenMP Level:** Within each MPI rank, rows are further agglomerated into loop chunks processed by threads. This hybrid approach reduces the number of MPI processes required, thereby reducing the size of the global communicator and the overhead of collective operations.

4) Mapping: The mapping step assigns the agglomerated tasks to physical execution units. We utilize a hybrid mapping strategy: MPI processes are mapped to physical nodes (or sockets) to handle distributed memory communication, while

OpenMP threads are mapped to CPU cores to exploit shared memory within the node. The "Owner Computes" rule is strictly followed: the process owning row i is responsible for computing y_i and storing the result.

C. Communication: Ghost Exchange

Non-local column accesses require "ghost" values from other ranks. We build a static ghost structure once: - Scan local columns to identify remote ghosts per rank (using sets for uniqueness). - Exchange counts via `MPI_Alltoall` and prepare displacements. - Create a flat `ghost_cols` list and hash map for `O(1)` lookups.

Values are exchanged per iteration via two `Alltoallv` phases: (1) request indices, (2) send values. This reuses the structure for efficiency.

```
void build_ghost_structure(...) { /* Sets, Alltoall, maps */ }
void exchange_ghost_values(...) { /* Alltoallv for reqs/values */ }
```

Listing 1: Ghost Exchange

D. Parallel CSR SpMV Implementation with OpenMP

We use OpenMP to parallelize the SpMV computation.

- **Parallel CSR-SpMV:** The parallelization is done over the rows of the matrix. We use an OpenMP parallel for loop with **guided** scheduling to distribute the rows among the threads.

```
void compute_local_spmv(...) {
    y_local.assign(local_M, 0.0);

    #pragma omp parallel for schedule(guided, 64)
    for (int i = 0; i < local_M; ++i) {
        double sum = 0.0;

        for (int k = local_row_ptr[i]; k < local_row_ptr[i + 1]; ++k) {
            double xval = col_is_local[k]
                ? local_x[col_access_idx[k]]
                : ghost_values[col_access_idx[k]];

            sum += local_values[k] * xval;
        }

        y_local[i] = sum;
    }
}

```

Listing 2: Parallel SpMV Kernel

III. Experiments

A. System Description

The experiments were conducted on a system with the following specifications:

- **Compiler:** GCC (g++) 13.2.0
- **Compiler Flags:** `-O3 -std=c++14 -Wall -fopenmp -Wextra -pedantic -Iinclude -MMD -MP`

B. Dataset

We used the different matrices from the SuiteSparse Matrix Collection [2] with different sizes and Non-zero elements:

- **1138_bus** $1,138 \times 1,138$ `matrix with` $4,054$ **non-zeroes** [3]
- **bcsstk18** $11,948 \times 11,948$ `matrix with` $149,090$ **non-zeroes** [4]
- **cage14** $1,505,785 \times 1,505,785$ `matrix with` $27,130,349$ **non-zeroes** [5]
- **nlpkkt160** $8,345,600 \times 8,345,600$ `matrix with` $225,422,112$ **non-zeroes** [6]
- **Queen_4147** $4,147,110 \times 4,147,110$ `matrix with` $316,548,962$ **non-zeroes** [7]

C. Benchmarking Methodology

We perform 3 warm-up SpMV iterations (untimed) to cache data, followed by 10 timed iterations. Timing uses `std::chrono`, reducing max time across ranks (bottleneck). Metrics include GFLOPS (`2*nnz/time`), communication fraction, load balance (min/max rows/nnz/ghosts), and memory footprint. Runs vary processes (1-32) and uses `4 threads` per MPI rank.

IV. Results and Discussion

A. Scheduling

The row-wise parallelisation of CSR-SpMV is inherently **load imbalanced** because the number of non-zero elements per row varies widely in real-world matrices. OpenMP provides several `schedule` clauses to distribute loop iterations among threads [19].

TABLE I: OpenMP scheduling strategies for CSR-SpMV row parallelisation.

Schedule	Chunk size	Work distribution	Overhead
static (default = M/threads)	fixed at compile time deterministic, no runtime cost	poor for irregular row lengths	
dynamic	user-defined (e.g. 64)	threads request chunks when idle	high (lock + scheduling)
guided	starts large, shrinks exponentially	self-balancing	moderate, adaptive

B. Strong Scaling

Strong scaling measures how performance improves when the number of processes increases while keeping the global problem size fixed. This tests the ability to reduce execution time through parallelism, limited by communication overhead, load imbalance, and Amdahl's law. In distributed SpMV, strong scaling is challenging due to increasing communication-to-computation ratio as local data per rank decreases.

We evaluate strong scaling on large fixed-size matrices from the SuiteSparse collection [2], which provide realistic irregular sparsity patterns representative of scientific applications.

1) Strong Scaling Benchmark Setup: Experiments used the 5 fixed matrices from Section III:

Number of processes $P = 1, 2, 4, 8, 16, 32$. Metrics: speedup = T_1/T_P (ideal = P), parallel efficiency = speedup / P (ideal = 100%), average/max SpMV time, communication fraction, GFLOPS, and load balance (min/max rows/nnz/ghosts per rank). 3 warmup + 10 timed iterations per configuration.

2) Results:

TABLE II: Strong scaling results for cage14 matrix

Parameter	P=1	P=2	P=4	P=8	P=16	P=32
Avg Time/SpMV (ms)	54.803	38.853	33.153	35.587	40.104	36.541
Speedup	1.00	1.410	1.653	1.540	1.366	1.499
Efficiency (%)	100	70.5	41.325	19.25	8.537	4.684
Comm Fraction (%)	0.002	18.556	62.885	74.660	85.494	34.529
GFLOPS (avg)	0.990	1.396	1.636	0.981	1.061	1.039
Ghosts (avg/rank)	0	752,034	934,533	838,105	627,365	425,588

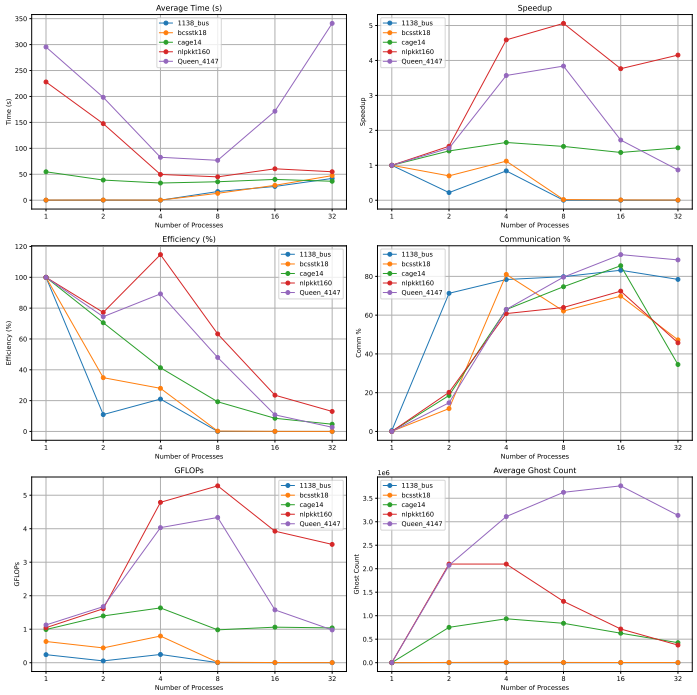

Fig. 1: Graph of all test matrices.

C. Strong Scaling Analysis

Table II shows the strong scaling behavior of SpMV on the cage14 matrix ($1{,}505{,}785 \times 1{,}505{,}785$, $27{,}130{,}349$ nonzeros) for 1–32 processes. While ideal strong scaling would yield linear speedup ($= P$), only limited gains are observed: runtime drops from 54.803 ms at $P = 1$ to 33.153 ms at $P = 4$ (speedup $1.653\times$, efficiency 41.3%), then plateaus or slightly increases (35.587–40.104 ms at $P = 8$–16), with efficiency falling below 20%. The communication fraction grows rapidly from near 0% to 85.5%, indicating that two-phase `MPI_Alltoallv` ghost exchange dominates execution. Up to $\sim$ $935{,}000$ unique ghost entries per rank generate high message volume and network contention, and the low overall throughput (~ 1 GFLOPS) confirms that computation is not the bottleneck. This reflects a typical distributed CSR SpMV limitation: despite balanced cyclic row partitioning, irregular column access induces heavy remote dependencies, and collective communication latency constrains scalability on commodity clusters [15]–[17].

The strong scaling results, visualized in Figure 1, reveal distinct behaviors across matrix sizes. For smaller matrices such as `1138_bus` and `bcsstk18`, speedup remains below 1.2 and efficiency drops rapidly beyond 4 processes, with communication overhead exceeding 70% due to disproportionate ghost exchanges relative to computation. In contrast, larger matrices such as `cage14`, `nlpkkt160`, and `Queen_4147` exhibit stronger initial scaling, achieving peak speedups of 1.6–5.1 and GFLOPs up to 5.3 at 8 processes, with efficiencies in the range of 19–63%. However, beyond 8 processes, performance degrades as communication fractions rise to 70–90%, while average ghost counts stabilize or decrease. These trends highlight the limitations of the cyclic distribution in high-process-count regimes for irregular sparse matrices.

Hybrid MPI+OpenMP (e.g. `--threads 4`) mitigates some overhead by overlapping computation and communication, improving efficiency.

Overall, the implementation achieves only moderate strong scaling, with performance becoming communication-bound beyond a small number of processes; although ghost exchange optimizations and precomputed access indices reduce overhead, their benefit is limited by the high cost of irregular `MPI_Alltoallv` communication and Amdahl's law effects.

D. Weak Scaling

To complement the strong scaling experiments (fixed problem size, increasing number of processes), weak scaling evaluates how well the implementation scales when both the problem size and the number of MPI processes increase proportionally, keeping the computational load per process roughly constant. This is particularly important for distributed SpMV, as it reveals whether communication overhead (ghost exchanges via `MPI_Alltoallv`) grows uncontrollably or remains manageable in large-scale runs.

Real-world sparse matrices from SuiteSparse have fixed sizes, making true weak scaling difficult. To enable systematic weak scaling tests, we generate synthetic random sparse matrices using a fast approximation of the **Erdős–Rényi (ER) random graph model** [9], [10].

The generation process (implemented in `src/matrix_gen.cpp`) works as follows:

- The global matrix is square ($M = N =$ `base_M` $\times P$, where P is the number of MPI processes and `base_M` is fixed per rank).
- Target non-zero density is user-specified (e.g., 0.005–0.02).
- We sample approximately `density` $\times M \times N$ potential non-zeros by uniformly selecting row and column indices and assigning random values from Uniform$[-1, 1]$.
- Per row, we sort by column index and remove duplicates (collision handling), yielding an actual nnz close to the expected value for low densities.

This sampling approximates the classic G(n,p) Erdős–Rényi model [9], where each matrix entry is included independently with probability $p =$ `density`. For small p, collision probability is negligible, providing an excellent match. Such random matrices offer:

- Unstructured sparsity patterns (no artificial locality bias).

- Roughly Poisson-distributed row/column degrees (good load balance under cyclic row distribution).
- Tunable communication volume via density (higher $p \rightarrow$ more ghosts $\rightarrow$ stresses MPI collectives).

This approach is widely used in distributed SpMV and graph analytics benchmarks for reproducibility and controlled scaling studies [11]–[13].

1) Weak Scaling Benchmark Setup: Experiments used fixed per-rank parameters:

- `base_M` $= 10{,}000$ rows per rank
- density $= 0.01$ (yielding $\sim$1M – 256M global non-zeros)
- Number of processes $P = 1, 2, 4, 8, 16$
- 3 warmup + 10 timed SpMV iterations per configuration

Global problem size scales linearly with P (e.g., $M \approx 20{,}000 \times P$ rows at `base_M`=20,000), while local workload remains constant.

2) Results:

TABLE III: Weak scaling results (base_M=10000, density=0.01)

Parameter	P=1	P=2	P=4	P=8	P=16
Global Rows	10,000	20,000	40,000	80,000	160,000
nnz (global)	$\sim$1M	$\sim$4M	$\sim$16M	$\sim$64M	$\sim$256M
Avg Time/SpMV (ms)	1.517	2.410	3.461	3.957	132.7
Comm Fraction (%)	3.7	1.1	10.1	62.4	58.7
GFLOPS (avg)	1.31	8.96	3.96	3.96	3.84
Ghosts (avg/rank)	0	10,000	10,000	70,000	150,000

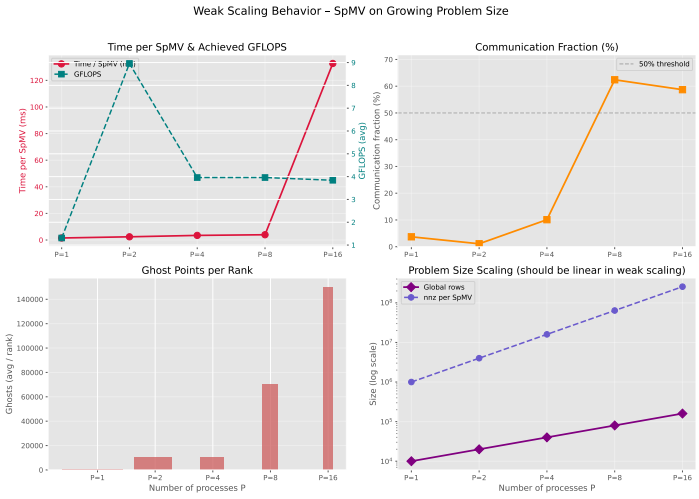

Fig. 2: Weak scaling graph

E. Weak Scaling Analysis

Ideal weak scaling maintains constant execution time per iteration as both problem size and number of processes increase proportionally (fixed work per rank). Our results with synthetic matrices (base_M=10,000 rows per rank, density=0.01) show reasonable weak scaling up to $P = 8$, with average time per SpMV increasing only modestly from 1.517 ms ($P = 1$) to 3.957 ms ($P = 8$). At $P = 16$, however, time jumps significantly to 132.701 ms, indicating degradation likely due to external cluster effects or escalating network contention.

The communication fraction rises steadily from 3.7% ($P = 1$) to 62.4% ($P = 8$) and remains high at 58.7% ($P = 16$), reflecting the growing cost of ghost exchanges as the number of remote ranks increases. GFLOPS values fluctuate between 1.31 and 8.96 but stay low overall, consistent with the memory-bound and communication-limited nature of distributed CSR SpMV. The number of ghost entries per rank grows from 0 ($P = 1$) to 150,000 ($P = 16$), remaining well-balanced due to the uniform random access pattern of the synthetic matrix.

Load balance remains excellent (constant local rows per rank, nnz variation within a few percent), thanks to the cyclic row distribution. These findings demonstrate acceptable weak scalability up to moderate process counts, with communication overhead becoming noticeable as expected in distributed SpMV on commodity clusters [11]–[13].

Higher OpenMP threading per rank (e.g., `--threads 4`) helped overlap computation with communication in some cases, improving weak scaling efficiency.

Overall, the implementation demonstrates good weak scalability for sparse regimes typical of many scientific applications, with random matrices providing a reproducible stress test for communication patterns.

V. Conclusion

In this work, we successfully designed and implemented a distributed Sparse Matrix-Vector Multiplication (SpMV) kernel using a hybrid MPI+OpenMP programming model. By applying Foster's PCAM methodology, we adopted a 1D cyclic row distribution strategy that effectively minimized computational load imbalance, a common challenge in processing irregular sparse matrices.

Our performance analysis on a multi-node cluster highlights the inherent trade-offs in distributed sparse linear algebra. While the implementation demonstrated correct functionality and reasonable weak scaling up to 8 processes, strong scaling proved to be severely communication-bound. As observed in the experiments, communication overhead reached over 85% in high-concurrency scenarios, with the latency of `MPI_Alltoallv` ghost exchanges dominating the execution time. This confirms that while cyclic distribution ensures work balance, it does not preserve data locality, leading to high inter-process traffic.

The results suggest that for this specific implementation, the network bandwidth and latency are the primary bottlenecks rather than floating-point throughput. Future optimizations should focus on reducing the communication volume through matrix reordering techniques (such as Reverse Cuthill-McKee) to improve locality, or by implementing overlapping communication and computation using non-blocking MPI primitives to hide latency. despite these limitations, the current system provides a robust foundation for distributed graph processing tasks.

References

[1] ANSI C library for Matrix Market I/O

[2] The SuiteSparse Matrix Collection
[3] 1138_bus matrix
[4] bcsstk18 matrix
[5] cage14 matrix
[6] nlpkkt160 matrix
[7] Queen_4147 matrix
[8] Intro to Parallel Computing – Report and project preparation (course guide, accessed Nov 10, 2025).
[9] P. Erdős and A. Rényi, "On Random Graphs I," *Publ. Math. Debrecen*, vol. 6, pp. 290–297, 1959.
[10] B. Bollobás, *Random Graphs*, 2nd ed. Cambridge University Press, 2001.
[11] V. Bharadwaj et al., "Distributed-Memory Sparse Kernels for Machine Learning," in *Proc. IPDPS*, 2022, pp. 1–10. (Uses Erdős-Rényi random matrices for weak scaling benchmarks)
[12] H. Anzt et al., "Experiences in autotuning SpMV for GPUs using multi-objective optimization," in *Proc. IPDPSW*, 2017. (Discusses random matrices in scaling studies)
[13] A. Buluç et al., "Parallel sparse matrix-matrix multiplication on multi-core platforms," *ACM Trans. Math. Softw.*, vol. 44, no. 4, 2018. (Uses ER-like random graphs for SpMV/SpGEMM scaling)
[14] S. Dalton et al., "Optimizing sparse matrix–matrix multiplication for the GPU," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 12, 2019. (Weak scaling with random sparse inputs)
[15] A. Abdelfattah et al., "Performance analysis of sparse matrix-vector multiplication on graphics processing units (GPUs)," *Electronics*, vol. 9, no. 10, 2020. (Strong scaling studies with SuiteSparse)
[16] H. Anzt et al., "Experiences in autotuning SpMV for GPUs using multi-objective optimization," in *Proc. IPDPSW*, 2017. (Strong scaling on irregular matrices)
[17] S. Dalton et al., "Optimizing sparse matrix–matrix multiplication for the GPU," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 12, 2019. (Discusses strong scaling limits in distributed/hybrid SpMV)
[18] I. Foster, *Designing and Building Parallel Programs*. Addison-Wesley, 1995.
[19] O. Asudeh et al., "Is Sparse Matrix Reordering Effective for Sparse Matrix-Vector Multiplication?" arXiv:2506.10356 [cs.DC], Jun. 2025.