Performance Analysis of Distributed SpMV using MPI: Scalability and Process Mapping

Samuele Morandi
Student-ID: 245067
Email: samuele.morandi@studenti.unitn.it
Github Repository: https://github.com/SamuMorandi/PARCO-Computing-2026-245067

***Abstract*—Sparse Matrix-Vector multiplication (SpMV) is a bandwidth-critical kernel in scientific computing, whose performance on distributed systems is often limited by irregular memory access patterns and communication overhead. This paper presents a distributed-memory implementation of SpMV using MPI, employing a 1D row-wise partitioning strategy on Compressed Sparse Row (CSR) data and a vector replication scheme via collective primitives. We evaluate the scalability of our approach through a dual analysis: a Strong Scaling study performed on real-world sparse matrices and a Weak Scaling study on synthetic random matrices. Crucially, we investigate the trade-off between intra-node resource contention and inter-node network latency by comparing "Dense" (compact) and "Distributed" (spread) allocation strategies. Our results reveal a regime-dependent performance crossover: under Strong Scaling, distributed configurations maximize aggregate memory bandwidth, outperforming dense ones at large scales (up to 128 processes). Conversely, under Weak Scaling, dense configurations prove superior by minimizing the number of communicating nodes and the associated collective latency. This study highlights the importance of balancing local memory saturation with network diameter in distributed iterative solvers.**

I. Introduction

Sparse Matrix-Vector multiplication (SpMV) is a fundamental computational problem in scientific computing and engineering. Its applications span diverse domains, from fluid dynamics simulations to large-scale graph analysis [1]. As datasets continue to grow in scale, the acceleration of SpMV via parallel processing on distributed-memory clusters is critical for achieving timely and efficient analysis.

However, achieving high performance on distributed systems remains notoriously difficult. This challenge stems from the irregularity of sparse data, which leads to major performance bottlenecks: 1) irregular memory access patterns that saturate memory bandwidth; 2) load imbalance; and 3) communication overhead, where the cost of exchanging vector elements across the network dominates the low arithmetic intensity of the kernel [4].

While extensive research has focused on shared-memory optimizations, a gap exists in the empirical analysis of how Process Mapping strategies impact performance on distributed architectures. Recent studies highlight that default mapping policies often fail to optimize for the non-uniform communication costs of modern HPC clusters [3]. Specifically, the choice between saturating a node's cores ("Dense") versus spreading processes across multiple nodes ("Distributed") involves a complex trade-off between maximizing local memory bandwidth and minimizing network latency.

To address these challenges, we designed a scalable distributed solver adopting Foster's design methodology [2]. The Partitioning stage decomposes the problem using a 1D cyclic row-wise distribution ($owner(i) = i \pmod P$) to mitigate load imbalance caused by clustered non-zeros. Communication is managed by replicating the dense input vector across processes using collective primitives, ensuring that data dependencies are satisfied before local computation.

This paper addresses the mapping gap by presenting a comparative performance analysis of allocation strategies for a CSR-based MPI implementation. We evaluate the scalability of our approach through a dual analysis:

- **Strong Scaling:** Benchmarking real-world matrices to assess speedup on fixed workloads [1].
- **Weak Scaling:** Benchmarking synthetic matrices to analyze system behavior under constant per-process load.

The main contribution of this work is to provide clear empirical evidence of a "performance crossover" point: while dense packing minimizes collective latency for weak scaling, distributed mapping significantly outperforms it at large scales for bandwidth-bound strong scaling tasks.

II. Methodology

The proposed solution implements a distributed SpMV solver leveraging the MPI standard. The implementation focuses on maximizing code modularity and handling irregular sparsity patterns through robust collective communications.

A. Matrix Parsing and Distribution

Rank 0 parses the input file (.mtx) line-by-line. To avoid memory bottlenecks, we implement a **Buffered Distribution Strategy**: non-zero elements are assigned to destination ranks based on cyclic partitioning and accumulated in per-process buffers (size 50,000). When a buffer fills, Rank 0 flushes it via `MPI_Send`, while receivers consume data via `MPI_Recv`, ensuring no single process holds the entire matrix.

B. Data Partitioning and Storage

We adopt a **1D Cyclic Partitioning** strategy to distribute the rows of the global matrix A among P processes. In this scheme, a global row index i is assigned to process p such that:

$$owner(i) = i \pmod P \tag{1}$$

This cyclic distribution improves load balancing for matrices with clustered non-zero elements.

Locally, each process stores only its assigned rows using the Compressed Sparse Row (CSR) format. To facilitate the conversion from unordered input triplets to the ordered CSR format, we utilize a temporary auxiliary structure named `Node`, defined as a tuple $\{row, col, value\}$. Incoming elements are first buffered into a vector of `Node` objects, then sorted based on row and column indices to ensure memory contiguity, and finally compressed into the standard `values`, `cols`, and `row_ptr` arrays. This intermediate step ensures the correctness of the CSR structure regardless of the arrival order of remote data.

C. Communication Strategy

The core SpMV operation $y = Ax$ requires access to the input vector x. Since the partitioning is cyclic and the matrix structure is irregular, a local row computation may require elements of x owned by any other process.

Instead of a complex dependency graph construction, we adopt a **Vector Replication Strategy** using `MPI_Allgatherv`. Algorithm 1 outlines the complete procedure.

- **Setup:** Before the iteration loop, processes exchange the size of their local portion of vector x using `MPI_Allgather` to calculate receive displacements.
- **Execution:** The SpMV kernel is executed over 10 iterations to measure performance under hot-cache conditions. At each iteration, `MPI_Allgatherv` is called to reconstruct the global vector x on every process (Line 12).
- **Computation:** Once the full vector x is available locally, the matrix-vector multiplication proceeds as a standard local CSR operation without further communication latencies inside the computation loop.

Algorithm 1 Distributed SpMV Execution

```
1:  Input: Local CSR Matrix ($A_{loc}$), Local Vector ($x_{loc}$)
2:  Output: Local Result Vector ($y_{loc}$)
3:  $P \leftarrow$ Total MPI Processes
4:  $N \leftarrow$ Global Matrix Columns
5:  $recv\_counts \leftarrow$ Array of size $P$
6:  Setup Phase:
7:  MPI_ALLGATHER($size(x_{loc})$, $recv\_counts$)
8:  $displs \leftarrow$ Calculate displacements from $recv\_counts$
9:  $x_{global} \leftarrow$ Allocate array of size $N$
10: Execution Loop:
11: for $iter = 1$ to $NUM\_ITERATIONS$ do
12:     MPI_ALLGATHERV($x_{loc}$, $x_{global}$, $recv\_counts$, $displs$)
13:                                         ▷ Local SpMV Computation
14:     for $i = 0$ to $rows_{loc}$ do
15:         $dot \leftarrow 0$
16:         for $k = row\_ptr[i]$ to $row\_ptr[i+1]$ do
17:             $col \leftarrow cols[k]$
18:             $dot \leftarrow dot + values[k] \times x_{global}[col]$
19:         end for
20:         $y_{loc}[i] \leftarrow dot$
21:     end for
22: end for
```

TABLE I
PROCESS MAPPING CONFIGURATIONS (NODES × PPN)

Procs (P)	Dense	Distributed
16	1×16	2×8
32	2×16	4×8
64	4×16	8×8
128	8×16	16×8

PPN = Processes per node.

III. EXPERIMENTAL SETUP

To ensure reproducibility and rigorous benchmarking, we detail the hardware environment, software toolchain, and the evaluation metrics used for the analysis.

A. Hardware and Software Environment

All benchmarks were executed on a high-performance computing (HPC) cluster. The scaling performance was measured across a range of 1 to 128 MPI processes.

To investigate the impact of resource contention, we systematically compared **Dense** (compact) and **Distributed** (spread) node allocation strategies across multiple process counts.

Table I shows every configuration used for each number of processes.

All experiments were executed with a uniform wall-time limit of 6 hours, utilizing compute nodes with RAM capacity ranging between 24GB and 32GB. The nodes are interconnected via Infiniband and Omni-Path technologies, ensuring low-latency communication for MPI primitives.

The source code was implemented in C++ and compiled using the GCC 9.1.0 compiler suite linked with the MPICH 3.2.1 library. The compilation command utilized the following flags: `-std=c++11` for language standard compliance, `-O3` to enable aggressive vectorization and architecture-specific optimizations, and `-g` to generate debug symbols.

B. Datasets and Workloads

Two distinct datasets were selected to evaluate different scaling regimes:

- **Real-World Matrices (Strong Scaling):** We utilized 5 matrices from the SuiteSparse collection [1]: *bmwcra_1, ML_Geer, msdoor, nlpkkt240, PFlow_742*. Table II reports the structural properties of each matrix.
- **Synthetic Matrices (Weak Scaling):** We employed pre-generated synthetic matrices where the global dimension increases linearly with the number of processes ($N = P \times 50,000$). These matrices maintain a fixed density of 40 non-zeros per row, ensuring that each

process is assigned an identical computational workload ($NNZ_{loc} = constant$).

In both experimental protocols, the dense input vector x was initialized with random double-precision values.

TABLE II
PROPERTIES OF REAL-WORLD MATRICES FROM SUITESPARSE [1]

Matrix	Rows	NNZ	Density	Domain
`bmwcra_1`	0.14M	10M	4.81×10^{-4}	Structural problem
`msdoor`	0.41M	19M	1.11×10^{-4}	Structural problem
`PFlow_742`	0.7M	37M	6.73×10^{-5}	Fluid dynamics
`ML_Geer`	1.5M	110M	4.89×10^{-5}	Machine learning
`nlpkkt240`	27.9M	760M	9.71×10^{-7}	Optimization

M = Millions. Values rounded for brevity.

C. Evaluation Metrics

Performance is evaluated using two standard scaling protocols:

1) **Strong Scaling:** Measures the speedup ($S = T_1/T_P$) keeping the *total* problem size fixed while increasing the number of processes P. This metric tests the solver's ability to reduce time-to-solution for a specific dataset. Additionally, we measured the **Load Imbalance**, defined as the ratio between the maximum and average number of non-zeros per process:

$$Imbalance = \frac{NNZ_{max}}{NNZ_{avg}} \tag{2}$$

A value close to 1.0 indicates a uniform work distribution.

2) **Weak Scaling:** Measures the execution time keeping the workload *per process* constant. Ideally, the execution time should remain constant ($T_P \approx T_1$) as both P and the problem size increase proportionally.

In both protocols, performance is reported in terms of execution time and throughput. The throughput is calculated as:

$$GFLOPS = \frac{2 \times NNZ_{loc}}{t_{avg} \times 10^9} \tag{3}$$

where $2 \times NNZ_{loc}$ represents the floating-point operations required (multiplication and addition per element) and t_{avg} is the average execution time over 10 iterations per process.

IV. RESULTS

This section presents the experimental results obtained on the HPC cluster.

A. Strong Scaling Analysis

The collection of measures gained from the strong scaling execution, demonstrate that the 1D-partitioning works well, keeping the Load Imbalance (Eq. 2) extremely low (under 1.03) in all the cases. Fig. 1 and Fig. 2, instead, illustrate the Speedup (T_1/T_P) obtained for the five real-world matrices listed in Table II, comparing the two mapping strategies.

The results highlight a correlation between matrix size and scalability potential. Small datasets like `bmwcra_1` (10M NNZ) reach saturation around 16 processes; adding more resources degrades performance as the problem size per core becomes trivial compared to the `MPI_Allgatherv` latency.

Fig. 1 clearly shows how performance degrades when adding a second node (transition from 16 to 32 processes). This implies that even if the aggregate computational power increases, the communication overhead between nodes outweighs the benefits. Conversely, in Fig. 2, the peak performance for each matrix is reached at different process counts, suggesting a stronger correlation between problem size and optimal scalability limits.

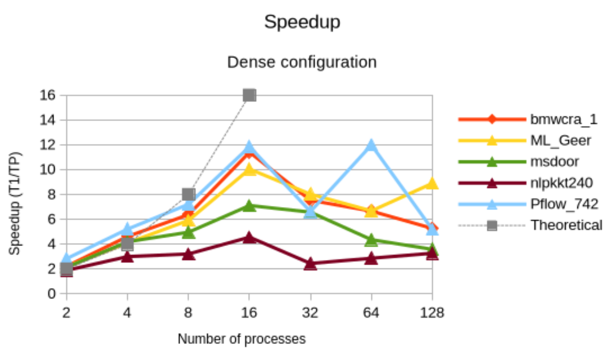

Fig. 1. **Dense Configuration.** Performance decreases for all matrices after reaching the peak at 16 processes (1 Node, 16 Cores), marking the transition from intra-node to inter-node communication.

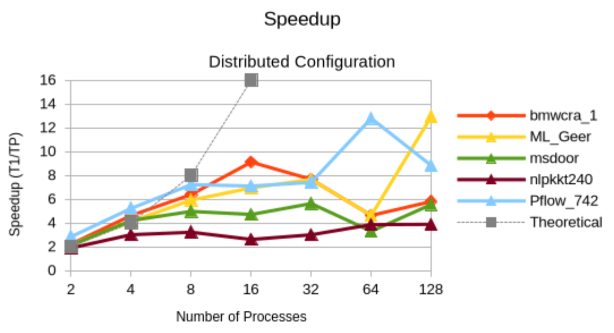

Fig. 2. **Distributed Configuration.** Shows superior scalability at large scales. The optimal number of processes varies depending on the matrix size.

Comparing the configurations reveals the critical impact of hardware resource contention. In the **Dense** case (Fig. 1), even large workloads like `ML_Geer` suffer from performance degradation at high process counts, as the aggregate memory bandwidth of the node becomes saturated. Conversely, in the **Distributed** configuration (Fig. 2), the same matrix maintains a positive scaling trend up to 128 processes (reaching $\approx 13\times$). This confirms that the distributed strategy effectively doubles the available memory bandwidth per process, unlocking performance that is otherwise throttled in fully populated nodes.

Interestingly, the largest matrix, `nlpkkt240` (760M NNZ), shows limited scalability (Speedup $\approx 3.8x$) in both cases. This anomaly supports the hypothesis that for extremely large sparse structures, scalability is intrinsically limited by the massive volume of data exchanged in the vector replication step.

B. Weak Scaling Analysis

In the Weak Scaling analysis, we evaluated the system's efficiency by keeping the workload per process constant ($50,000$ rows per core) while increasing the global problem size linearly with P.

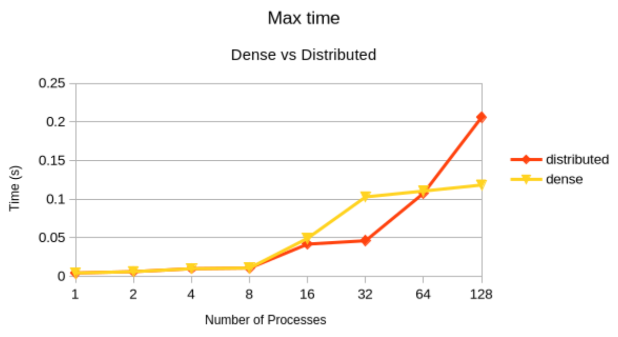

Fig. 3. **Weak Scaling Comparison.** Ideally, time should remain constant. The Distributed configuration (orange) is faster at medium scales ($P = 32$) but degrades at $P = 128$.

Fig. 3 compares the execution times of the two configurations. The ideal weak scaling behavior (constant time) is not achieved due to the growing cost of the `MPI_Allgatherv` operation, needed to exchange the vector x. Since the global vector size grows as $N = P \times 50,000$, the volume of data exchanged increases linearly, penalizing larger runs.

The comparison reveals an inversion relative to Strong Scaling. At **Medium Scale** ($16 \leq P \leq 32$), the **Distributed** configuration is faster (e.g., $0.046s$ vs $0.103s$ at $P = 32$) due to higher memory bandwidth. However, at **Large Scale** ($P = 128$), the **Dense** configuration becomes superior ($0.118s$ vs $0.206s$). The Distributed setup involves 16 physical nodes, creating higher network overhead compared to the Dense configuration (8 nodes). Thus, for Weak Scaling, minimizing the number of communicating nodes is critical. This suggests that for Weak Scaling, minimizing the number of communicating nodes (as in the Dense configuration) is more critical than maximizing memory bandwidth, as the bottleneck shifts from local memory access to inter-node collective communication.

C. Impact of Process Mapping: Synthesis

To provide a direct comparison between the allocation strategies, Fig. 4 isolates the performance of the `ML_Geer` matrix under **Strong Scaling conditions**.

The graph reveals a complex interaction between network latency and memory bandwidth. At **Small to Medium Scale** ($16 \leq P \leq 64$), the **Dense** configuration generally outperforms the Distributed one (e.g., $10\times$ vs $7\times$ speedup at $P = 16$), suggesting that minimizing network nodes is advantageous while the memory controller is not saturated. Conversely, at **Large Scale** ($P = 128$), a dramatic **crossover** occurs: the **Distributed** configuration surges to a peak speedup of

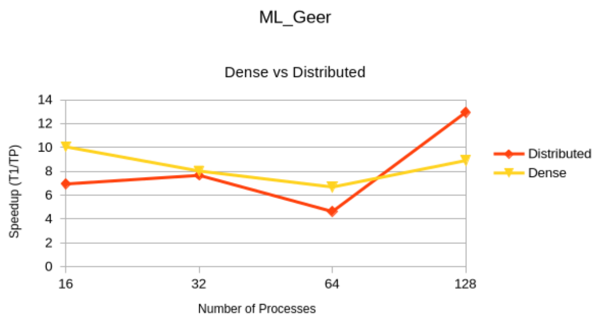

Fig. 4. Direct comparison on `ML_Geer` (Strong Scaling). Dense (yellow) maintains superiority up to $P = 64$, leveraging intra-node communication. However, at $P = 128$, the Distributed configuration (red) overtakes significantly, proving that memory bandwidth becomes the ultimate bottleneck.

$12.94\times$, while Dense only recovers to $8.91\times$. This bandwidth advantage is quantified by the system throughput (Eq. 3), where the Distributed configuration achieves 0.129 GFLOPS versus 0.086 GFLOPS for Dense. This gap underscores that for memory-bound kernels like SpMV, aggregate performance is strictly limited by the available memory channels per core.

This behavior confirms that SpMV is bandwidth-bound, but the "bandwidth wall" is hit only at large scales. Up to 64 processes, the benefits of compact placement (low latency) dominate. At 128 processes, the aggregate memory bandwidth provided by doubling the number of nodes (Distributed strategy) becomes the decisive factor, unlocking performance that the Dense configuration cannot reach.

V. Conclusion

We presented a distributed MPI implementation of the SpMV kernel, analyzing the trade-off between memory bandwidth and network latency. Our results identified a critical performance crossover dictated by the scaling regime. Under **Strong Scaling**, the **Distributed** strategy proves superior at scale ($P = 128$) by maximizing aggregate memory bandwidth to overcome local saturation. Conversely, under **Weak Scaling**, the **Dense** strategy prevails, as minimizing the number of communicating physical nodes effectively reduces the global vector exchange latency. These findings confirm that optimal process mapping is context-dependent: bandwidth-bound problems favor distributed allocation, while latency-bound global reductions favor dense packing.

A. Future Works

In future, it may be interesting to address the bottlenecks by implementing hybrid MPI+OpenMP parallelism to reduce the communication footprint and exploring 2D partitioning strategies. Also non-blocking collectives could be used to reduce the execution time. Finally, it may be useful to avoid the replication of the vector among all the processes: instead, it could be useful to allow a point-to-point communication, in order to exchange only the needed values.

References

[1] T. A. Davis and Y. Hu, ”The University of Florida SuiteSparse Matrix Collection,” ACM Transactions on Mathematical Software (TOMS), vol. 38, no. 1, pp. 1-25, 2011. [Online]. Available: https://sparse.tamu.edu/

[2] I. Foster, *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[3] S. R. Saufi et al., ”ZAKI+: A Machine Learning Based Process Mapping Tool for SpMV Computations on Distributed Memory Architectures,” in IEEE Access, vol. 7, pp. 77085-77100, 2019.

[4] N. Zhang et al., ”Balancing Computation and Communication in Distributed Sparse Matrix-Vector Multiplication,” in Supercomputing Frontiers and Innovations, 2023.

[5] A. Buluç et al., ”Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks,” in Proc. ACM SPAA, 2009.