Parallel Project Report: Benchmarking of MPI Distributed Configurations and Hybrid Models for Sparse Matrix-Vector Multiplication

Student Leonardo Tartini (ID: 245119)
Email: leonardo.tartini@studenti.unitn.it
Course: Introduction to Parallel Computing (2025-2026)

***Abstract*—Sparse Matrix-Vector Multiplication (SpMV) is a cornerstone of scientific computing characterized by its memory-bound nature and irregular access patterns. In this project, I developed a code that makes the first process (rank 0) read sparse matrices *A* saved in COO notation and distribute them among other ranks, along with the corresponding multiplication vector *x* entries. After that, each rank converts the received data into CSR notation, shares its vector entries to help rebuild the full *x* and finally computes local SpMV to yield its entries of the result vector *y*. This report evaluates a distributed implementation using MPI and a hybrid MPI+OpenMP approach. By employing a 1D cyclic distribution, I achieve near-perfect load balancing for large-scale matrices, whereas for smaller matrices, the minor number of rows owned by each rank minimizes the averaging of NNZs distribution, losing effectiveness. During the test phase, I measured both communication and computation times, in order to find the best compromise between efficiency gain and communication overhead. Eventually, performance analysis reveals that collective communication via `MPI_Allreduce` becomes the primary bottleneck at scale and results show that a hybrid configuration (e.g. 4 ranks $\times$ 4 threads) provides a significant speedup (in my case up to 1.8x) over pure MPI by mitigating network contention and synchronization overhead.**

***Index Terms*—SpMV, MPI, OpenMP, Hybrid Programming, CSR, Load Balancing, HPC.**

I. Introduction

Problem: Sparse Matrix-Vector Multiplication ($y = Ax$) is a fundamental kernel in iterative solvers for linear systems. In High-Performance Computing (HPC), scaling SpMV involves addressing the low arithmetic intensity and the irregular distribution of non-zero elements (NNZ).

The efficiency of distributed SpMV depends on two main factors:

1) *Workload Imbalance:* Irregular matrices can lead to a "straggler" effect where one rank processes significantly more NNZs than others.
2) *Interconnect Latency:* Since each row might require any element of the input vector x, communication often outweighs computation.

Objective: Develop a code that:

1) Reads a matrix A saved in Matrix Market format
2) Distributes its COO representation and the random vector x among ranks
3) Rebuilds the full multiplication vector x to have it accessible to all ranks
4) Performs local SpMV inside each rank

For each execution, measure both communication and computation times to compare different configurations and find the best compromise between computation speedup and communication overhead.

Contribution: This study evaluates how the 1D cyclic distribution affects load balancing across different matrix scales and compares the performance of pure MPI (P ranks) versus a hybrid MPI+OpenMP (P ranks $\times$ T threads) approach to identify the optimal allocation of resources on a modern cluster. The whole process of running the code and analyzing the results is handled by a single PBS file that tests several combinations of the aforementioned parameters and returns a clear comparison of the obtained results.

II. State of the Art

SpMV is notoriously a memory-bandwidth bound kernel. Sparse matrices have a relatively small number of non-zero entries, so it is best to store them in more efficient structures rather than double-pointed row-columns. In this project I utilized standard row-wise partitioning in CSR (Compressed Sparse Row), which stores per-row indexes of non-zero elements and is one of the most widely used formats. There are many other data structures and tuning algorithms that optimize data accesses, such as ELLPACK, HYB, JAD and pOSKI, each having pros and cons for specific hardware.
Regarding data distribution, several strategies exist:

- **1D Block Distribution:** Simple to implement but often leads to severe load imbalance in matrices with irregular non-zero distributions.
- **1D Cyclic Distribution:** As implemented in this work, it improves load balancing by interleaving row ownership, which is particularly effective for matrices with power-law structures.

- **2D Partitioning:** Reduces the number of processes involved in each communication step but increases the algorithmic complexity [2].

However, in MPI distributed configurations, as the number of process increases, the main problem shifts from data access optimization to synchronization of the multiplication vector x. Schubert et al. [3], [4] demonstrate that standard MPI implementations often fail to overlap communication with computation effectively unless specific strategies, such as dedicated communication threads or explicit overlap, are employed. These studies [3] emphasize that performance is not only a matter of arithmetic throughput but is strictly tied to the hardware's interconnect latency and the ability to balance the workload: therefore hybrid models (MPI+OpenMP) are frequently proposed to reduce the number of MPI ranks, simplifying the communication graph and improving the efficiency of collective operations like `MPI_Allreduce`.

III. CONTRIBUTIONS & METHODOLOGY

A. Process overview

This is the flow of operations:

1) Rank 0 reads Matrix Market File and distributes COO triplets and vector entries to all ranks
2) Each rank converts its local data into CSR format and shares its vector entries with others via `MPI_Allreduce` to rebuild the whole multiplication vector
3) Each rank performs local SpMV (results are not gathered to avoid adding communication overhead that is not useful for benchmarking purposes)

The baseline of the project is a pure MPI implementation that tests performance with [1, 2, 4, 8, 16, 32, 64] processes to analyze how the computation-to-communication ratio changes. Then, the same tests are executed with a hybrid MPI+OpenMP approach running a few $P \times T$ combinations to find a hypothetical "sweet spot" balancing computation efficiency and communication overhead by mitigating network contention (less ranks) and exploiting row-level parallelism (OpenMP thread directives).

B. 1D Cyclic Data Distribution

To reduce imbalance in matrices with clustered non-zeros, I implemented a *1D cyclic distribution* both for the matrix A and the vector x. Each rank r is assigned rows i of A and entries j of x such that $i|j \pmod P = r$. Then, to split the multiplication vector among ranks I utilized `MPI_Scatterv`, a collective MPI operation (instead of innumerous MPI_Send) that avoids possible buffer overflows, segmentation faults and deadlocks, while also internally optimizing communication.
Note: although cyclic distribution yields optimal load imbalance for large-scale matrices, it might not be as effective with smaller ones, due to each rank having a minor number of rows and losing the averaging effect of the cyclic distribution itself.

C. Vector Synchronization

Vector synchronization is achieved through `MPI_Allreduce`. Each rank populates a local buffer of size N with its local elements of x at the correct positions (global indexes are kept track of) and receives the unknown values from all other ranks. While this requires $O(N)$ memory per rank, it ensures that all ranks have a consistent view of x for the multiplication, solving encountered problems of accesses to uninitialized addresses.

D. Hybrid Parallel Kernel

In hybrid MPI+OpenMP runs, the local computation is parallelized using OMP directive **#pragma omp parallel for schedule(dynamic, 64)**. This choice is specifically targeted at matrices with high variance in NNZ per row, ensuring that threads within a rank remain balanced.

Algorithm 1 Hybrid SpMV Kernel

```
1:  procedure SPMV_HYBRID(local_mat, full_x, local_y)
2:      #pragma omp parallel for schedule(dynamic, 64)
3:      for $i = 0$ to $local\_M - 1$ do
4:          $sum \leftarrow 0$
5:          for $k = row\_ptr[i]$ to $row\_ptr[i+1] - 1$ do
6:              $sum \leftarrow sum + vals[k] \times full\_x[col\_idx[k]]$
7:          end for
8:          $local\_y[i] \leftarrow sum$
9:      end for
10: end procedure
```

IV. Experimental Setup

A. Platform

- **CPU:** 16 x 4 nodes; Intel Xeon Gold 6252N @ 2.3GHz (64 cores total)
- **Cache:** L1d 32kB, L1i 32kB, L2 1024kB, L3 36608kB
- **Memory:** 16GB x 4 nodes (64GB total)
- **OS & Toolchain:** GNU/Linux, gcc9.1.0, OpenMPI, OpenMP
- **Libraries:** MMIO to read Matrix Market files

B. Dataset

Three SuiteSparse matrices were selected to cover different sparsity patterns and scales:

TABLE I
Benchmark Matrices Characteristics

Matrix	Rows (M)	NNZ	Density
af_shell8	504,855	9,046,865	17.92
audikw_1	943,695	39,297,771	41.64
fidapm37	9,152	765,944	83.69

In addition to these, I developed a simple code snippet that generates random synthetic matrices of a given size in order to evaluate weak scaling performance.

C. Metrics and Methodologies

For each matrix, for each number of processes, the code:

- registers NNZs distribution (load balance)
- measures the time taken for vector communication
- measures the time taken for multiplication

Each configuration runs 1(warm-up)+10 times to allow statistical analysis.

V. Results & Discussions

A. Load Balance and Matrix Scale

An analysis of the data in Table II reveals a significant discrepancy in load balancing efficiency between large and small matrices. For *audikw_1* (approx. 40M NNZ), the 1D cyclic distribution maintains an imbalance below 1% even at 64 ranks. Conversely, the *fidapm37* matrix exhibits an imbalance of 23.85% in the same configuration, as illustrated in Figure 1. This is due to its small size: with only 9,152 rows distributed across 64 ranks, each rank receives only ~ 143 rows. In such small samples, the cyclic distribution cannot average out the dense rows, leading to significant variance. This highlights that 1D cyclic distribution is a "large-scale" optimizer, while in other cases, different solutions (like 2D partitioning) might work best.

TABLE II
Load Imbalance Analysis (64 Ranks)

Matrix	Avg NNZ	Max NNZ	Imbalance %
audikw_1	614,027	617,164	0.94%
fidapm37	11,967	14,821	23.85%

Note: an uneven distribution leads to a "straggler effect" where the rank owning the densest rows dictates the execution time of the entire synchronization barrier.

B. Strong Scaling: The Communication Wall

Strong scaling analysis evaluates how the execution time decreases as the number of processors increases for a fixed total problem size. With my implementation, benchmarks on real-world matrices, such as *af_shell8* and *audikw_1*, reveal a significant performance bottleneck due to which increasing the number of MPI ranks becomes counterproductive, as illustrated in Figure 2. Beyond a very low threshold (typically $P \geq 4$), the computational work per rank drops below the overhead of the MPI interconnect. At 16+ ranks, the communication cost is two to four orders of magnitude higher than the actual computation, resulting in a parallel efficiency close to zero. This bottleneck is a direct consequence of the *all-to-all* nature of the vector synchronization via `MPI_Allreduce`. As the number of processes increases, the volume of data remains the same (N elements), but the number of messages and synchronization steps increases, leading to a saturation of the network bandwidth and increased latency.

Note: for sparse matrices with low arithmetic intensity, the cost of the all-to-all collective communication scales with the

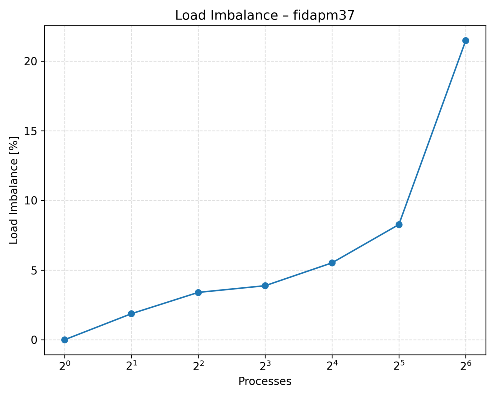

Fig. 1. NNZ distribution for fidapm37 on 64 ranks, showing high variance due to small matrix scale.

complexity of the interconnect topology rather than the matrix size.

C. Hybrid MPI+OpenMP

The hybrid model offers a solution to the communication wall. In Figure 3, I compare 16 cores used as 16 MPI ranks versus 4 MPI ranks with 4 threads each.

The **4x4 configuration** is 1.8x faster. Reducing the rank count decreases the number of participants in the `MPI_Allreduce` operation. This significantly lowers the pressure on the interconnect. Interestingly, the 1x16 configuration (single rank, 16 threads) is slower than 4x4, likely due to NUMA effects and memory contention within a single socket. The 4x4 setup provides the best trade-off between local shared-memory speed and global message-passing efficiency and it consistently outperformed both the pure MPI (16×1) and the fully multithreaded (1×16) approaches.

This "sweet spot" is explained by two factors:

1) **Reduced Network Contention:** By reducing the number of MPI ranks from 16 to 4, the number of messages in the `MPI_Allreduce` collective is reduced by 75%, significantly lowering the probability of network collisions.
2) **Memory Hierarchy Optimization:** Unlike the 1×16 setup, which may suffer from NUMA (Non-Uniform Memory Access) effects and cache thrashing, the 4×4 setup allows each rank to maximize local cache reuse while maintaining enough parallelism to hide memory access latencies.

D. Limitations

The current implementation is not as efficient as it may be, as I mainly focused on developing a basic but working code instead of making it perfectly optimized. One of the main performance limitation is certainly the `MPI_Allreduce` collective, used to share values of *x* among ranks. Though being of simpler implementation, this approach presents serious inefficiencies:

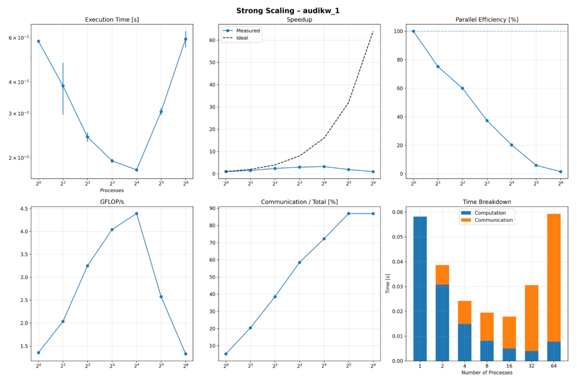
Fig. 2. Strong scaling evaluation on audikw_1 matrix using pure MPI configuration.

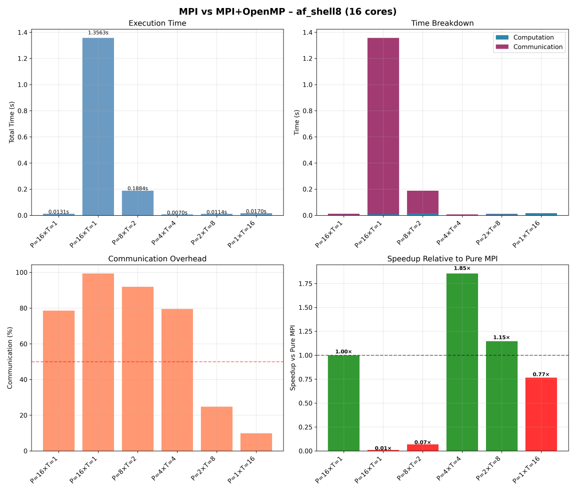
Fig. 3. Execution time comparison: Pure MPI (16x1) vs Hybrid (4x4) vs Multithreaded (1x16).

- **Communication volume:** instead of sharing only the needed values between owner and requester rank $O(non\_local\_nnz)$, it makes each rank send all its x values and receive all others $O(N \times P)$
- **Memory usage:** each rank now stores a local copy of the entire x vector, wasting a tremendous amount of memory ($O(N)$ instead of $O(ghost\_entries)$ per rank)
- **Limited scalability:** at the moment, with only 32-64 processes (a relatively small number) communication takes $\sim$ 90% of the whole process time, substantially wasting the computation speedup

Other aspects are not optimized, such as reading the Matrix Market File and the SpMV parallelization (the best configuration of parameters is strictly correlated to the matrix and hardware characteristics).

VI. Conclusions & Future Work

This study demonstrated that while 1D cyclic distribution effectively addresses load imbalance for large-scale sparse matrices, the way I implemented it introduces a significant bottleneck in distributed environments due to its reliance on global collective communication. These findings are highly consistent with the observations of *Schubert et al.* [3], [4], who identified that the communication-to-computation ratio is the primary limiting factor for distributed SpMV scalability on modern clusters. Experimental results corroborate the "Hybrid Advantage" discussed in [3]. Specifically, the 1.8x speedup achieved by my 4x4 configuration over the 16x1 pure MPI setup confirms that reducing the number of MPI ranks participating in collectives like `MPI_Allreduce` is more beneficial than increasing raw process parallelism. This aligns with Schubert's conclusion that hybrid models mitigate network contention and synchronization jitter. Furthermore, the high imbalance observed in the *fidapm37* matrix (23.85%) validates the concerns raised by *Williams et al.* [1] regarding the sensitivity of sparse kernels to matrix scale and local density patterns. For small-scale problems, the overhead of managing distributed buffers outweighs the benefits of parallel execution.

Future work will focus on two main optimization paths:

1) **Sparse Point-to-Point Communication:** transitioning from global `MPI_Allreduce` to localized `MPI_Isend/Irecv` patterns. Analysis of the *audikw_1* matrix suggests that each rank only requires data from a small subset of the total vector x.
Implementing a "halo-exchange" mechanism could reduce the communication volume and memory footprint by up to $\sim$ 93%, as each rank would only share and store the strictly necessary entries.
2) **2D Partitioning and MPI-IO:** to further improve scalability, 2D grid-based distributions will be investigated. As proposed by *Buluç and Gilbert* [2], 2D partitioning limits the communication to $\sqrt{P}$ processes, offering better asymptotic scaling. Additionally, **MPI-IO** could be implemented to eliminate the serial bottleneck of Rank 0 reading the Matrix Market File. This would allow each rank to perform independent, collective I/O operations on dedicated file offsets, significantly reducing the initialization time.

VII. Reproducibility & Artifact Availability

Github repository:
https://github.com/tartaretta2/PARCO-Computing-2026-245119.git

How to run: once the project folder is in the cluster, make sure to have some .mtx files in the data_matrices folder, then position yourself in the scripts folder and run the job with command: **qsub run_mpi_spmv.pbs**
This runs all test configurations on the matrices (pure MPI and hybrid MPI+OpenMP); the job also runs a python script to analyze results (load balance and time taken for vector synchronization and SpMV), which returns several plots addressing performance comparisons.
More detailed instructions are provided in the README.md file. Please take a look at that if you plan to run simulations.

References

[1] S. Williams, et al. ”Optimization of sparse matrix-vector multiplication on emerging multicore platforms.” (2007).

[2] A. Buluç, & J. R. Gilbert. ”Parallel sparse matrix-vector multiplication and indexing for 2D data distributions.” (2012).

[3] G. Schubert, G. Hager, et al. ”Parallel sparse matrix-vector multiplication as a test case for hybrid MPI+OpenMP programming.” (2011).

[4] G. Schubert, et al. ”Hybrid-parallel sparse matrix-vector multiplication with explicit communication overlap on current multicore-based systems.” (2011).