SpMV 1D partitioning with MPI

Devid Troka 239293
Computer science, communications and electronics engineering
University of Trento, Trento, Italy
devid.troka@studenti.unitn.it
https://github.com/Devid663/PARCO-Computing-2026–239293-2.git

***Abstract*—In recent decades, the growing need to perform calculations on large data sets, combined with the achievement of the maximum power dissipable by cooling systems, have led to a more concentrated development on multicore architectures, parallelism, and optimization programs and algorithms. The purpose of this article is to analyze the performance of a sparse matrix-vector multiplication (SpMV) operation, which is considered an important operation for performance analysis and notoriously difficult to execute efficiently. This allows us to better visualize the performance of such a complex operation, with various choices and parallel design strategies, and to better understand its scalability. This makes SpMV an excellent case study for understanding the potential and limitations of parallel execution and strategies. The execution of this operation will be analyzed and performed using the MPI library, which is standard for exchanging messages on distributed memory on parallel architectures.**

***Index Terms*—1. Introduction, 2. Methodology, 3. Experiments, 4. Results, 5. Conclusions**

I. Introduction

The **sparse matrix-vector multiplication** (**SpMV**) operation is an important operation, where the operands of the matrix are sparse (typically more than 50% of the entries are 0). It is therefore a matrix whose number of non-zero elements is much smaller than the total number of elements, and, if the number of its non-zero elements grows only linearly with the size of the matrix. It's a fundamental operation that can have a broad impact, given its uses in scientific fields and operations that involve large data sets. It is well known, however, that SpMV is not a very simple operation, due to its structure and the fact that these algorithms are necessarily memory-bound because sparsity causes very low arithmetic intensity. Over the years, efficient implementations have been relatively scarce, leading to the development of various techniques and formats for storing the matrix. This means that there are different formats, more and less efficient, for storing only the non-null values of the matrix. One of the most commonly used formats, and the one used in this paper, is the CSR (Compressed Sparse Row) format. Using **MPI** can help optimize the operation by trying to distribute the workload through distributed parallelism. The entire process will be handled using **Foster's design methodology**, which allows for the creation of parallel programs following a well-defined strategy, with particular attention on Data partitioning.

II. Methodology

A. Problem definition

The Sparse Matrix-Vector Multiplication (SpMV) operation calculates $y = A \cdot x$, where A is a sparse matrix with *nnz* non-zero elements. As mentioned above, the most common data structure for storing sparse matrices in an SpMV calculation is the **Compressed Sparse Row** (**CSR**) format. Each row of matrix A is divided one at a time into a dense array *val*, together with the corresponding integer ind, which stores the column index of each stored element. A third integer array, *ptr*, keeps track of where each row begins in *val*, ind. In its most popular variant, CSR maintains a floating array *val[nnz]* and two integer arrays, *col_ind[nnz]* and *row_ptr[n]*, to store *A = (aij)*.

Listing 1
CSR SpMV (A) theorical code

```
for (int i = 0; i < n; i++) {
    y[i] = 0.0;
    for(int k=row_ptr[i]; k<row_ptr[i+1]; k++){
        y[i] = y[i]+val[k]*x[col_ind[k]];
    }
}
```

The *row_ptr* array stores the index of each row in val. This means that val[k] stores the element of the matrix *aij*, then $row\_ptr[i] <= k < row\_ptr[i+1]$. The *col_ind* array stores the column index of the elements in the val array. This means that if the element of the *val[k]* matrix is in *j*, then $col\_ind[k] = j[2]$. This algorithm will be implemented with MPI, following the Foster's Methodology, trying to optimize the performance.

B. Algorithms and data structures

As mentioned, Foster's methodology was followed for this implementation, a systematic approach to developing parallel programs in the best possible way. It is basically divided into four phases:

- **Partitioning:** dividing the entire problem into many small tasks. Crucially, good partitioning will determine the size of the communication.

There are two types:
  - Domain decomposition, which consists of dividing the matrix, used in our case;
  - **Functional decomposition:** dividing the work into tasks.
- **Communication:** essentially how the tasks defined above exchange the necessary data. As we will see, this phase is also very important and determines the efficiency of the implementation.
- **Aggregation:** grouping tasks to reduce the number of minor tasks.
- **Mapping:** assigning tasks.

Data partitioning

As mentioned above, this is the fundamental part of Foster's methodology, it is therefore a good idea to explore this part in more detail. Data partitioning consists of assigning data and operations to processes in such a way as to balance the workload (data, calculations), reducing inter-process communication. In the case of SpMV, both the matrix and the vectors must be distributed and considered in the data layout. **The simplest way to partition a sparse matrix is by rows (1D)**. Each process has a set of rows and the non-zero values of those rows. The elements of the vector are distributed in the same way as the rows of the matrix. The simplest 1D partition distributes the n rows evenly among processes in blocks, so that each process has approximately n/p consecutive rows. Although this partition balances the rows, it **does not balance the zeros**. The distribution of 1D (cyclic) data, per row, can be defined by the formula:

$$owner(i) = imodP$$

where P is the number of processes and i is the row index. Each process stores all non-zero values belonging to its rows. Data partitioning is used because by improving the layout of the data, it is possible to significantly reduce the SpMV time. MPI is therefore used not only to speed up SpMV calculation but also to partition the matrix by rows (1D, in fact), adapting to CSR. Essentially, each process manages a contiguous subset of matrix rows and calculates the corresponding SpMV elements locally. This approach reduces the computational load of each individual process, reducing the memory bottleneck. However, it must be used with caution. The algorithm implemented in testing works as follows:

- Rank 0 reads and scans the elements of the global COO (Coordinate format to then build the CSR), accumulating the local vectors, calculating, for example, the local *nrow* number and the *nnz* per rank, sending them to the other ranks.
- Each rank receives a buffer and builds the local COO.

Matrix distribution and Local Construction

After reading the matrix, rank 0 scans all arrays (*I_global[K]*, *J_global[K]*, *VAL_global[k]*), accumulates them in separate buffers by rank, and sends them. This could reduce overhead. The communication is divided into two phases for rank 0:

- rank 0 sends only one integer to each rank: *num_local_nnz*
- rank 0 sends the actual arrays
- Each rank then receives: *num_local_nnz*, allocates the buffers, and receives the data.

Each rank receives its own triples. Cyclic 1D partitioning is then performed by rows, followed by communication: *nz_local* counting and COO distribution. In the next phase of the algorithm, the row indices are converted and the local CSR is then constructed, where each rank effectively constructs a local CSR submatrix.

Now all that remains is to identify the **remote column indices** (**ghost entries**). Once the matrix has been read, the indices distributed by row and the matrix have been read locally, the actual product operation must be performed, but this last step is necessary to make it possible. Following the classic formula for calculating the SpMV:

$$y_i = \sum_j Aij \cdot x_j$$

Not all xj values are necessarily stored correctly on the correct rank. This is because, according to cyclic 1D partitioning, a global index *j* is local to rank if:

$$j \; mod \; P = rank$$

if this rule is not followed, then **index j is said to be a ghost entry**. In simple terms, rank 0 cannot access value *x[j]* because it is allocated to another rank at that moment. This obviously also applies to ranks other than 0. Therefore, before the product can actually be calculated, all columns used in the CSR must be checked, and for each global column (*j*), the formula must be checked to see if it is correct. If it is not, that index is added to an array *ghost_cols[]*. Ghost entries are checked by the function *check_ghost_entries()*. This function scans the structure of the local CSR and selects the global column indexes whose owner differs from the current rank.

At this point, as we understand, each rank unfortunately does not have all the elements of x, so the ghost check has been performed, but one fundamental thing is still missing, namely the exchange of these ghosts. As mentioned, a ghost is essentially a missing *x_j* value at the current rank, but without receiving it from the rank where it is currently located, it is not possible for the rank that requires it to know it. Without this MPI exchange, it is not possible to perform our SpMV correctly. Obviously, precisely because MPI is being used, each rank cannot read from the memory of the others. This communication is performed once per SpMV. In the case of iterated SpMV, it would be necessary to communicate it several times.

The *exchange_ghosts* function performs the actual exchange using an **MPI communication pattern**, initially designed as **point-to-point** directly on the cyclic property, where each rank directly requests the value it needs. However, this results in a simple pattern, but with too many MPI messages, one for each ghost. In fact, each rank communicates with every other rank, but doing so results in $\Theta(P)$ messages per rank, for a total of up to $\Theta(P^2)$ messages. This **makes it very poor in terms of scalability**, becoming **a communication bound**. To overcome this problem, an alternative has been devised using collective communication, allowing us to group the ghosts together, and each process provides and receives the ghosts it needs, using the *MPI_Alltoall* and *MPI_Alltoallv* operations. Essentially, each rank counts how many ghost indices are requested by each owner, exchanging this data. Subsequently, each rank exchanges the lists of requested indices, prepares the requested values, and exchanges them with each other. The exchanged values are stored in a vector aligned with the local sorting and then used by the SpMV kernel. However, a comparison between the two will be shown.
Once this is done, you have: local CSR, local indices and local vector, global ghost columns, and ghost values. All that remains is to implement the SpMV. Before performing the calculation, however, an additional function is needed, namely *get_x_value*(), to retrieve the value *x[j]*, this function is a ”shortcut”.

Ultimately, a distributed SpMV in 1D rows was implemented using MPI. The rank 0 process reads and distributes the matrix in COO format, each process constructs a local CSR representation, and the ghost vector elements are exchanged via point-to-point communication. Each process calculates the results of the rows it owns. If a local row requires remote *x_j*, the process contacts the owner directly and receives that value. More generally, all processes run the same program and differ only in rank. This correctly follows the SPMD model.

III. Experiments

High Performance Computing (HPC) clusters use powerful processors that work in parallel to process large multidimensional data sets and solve complex problems at extremely high speeds. Each cluster node operates independently with its own processor and memory, enabling efficient handling of complex workloads. For all experiments, a 16-core computing node with 16 GB RAM available was requested via pbs through the $short\_cpuQ$ queue. Accessible via SSH to the cluster (head node), via external software, such as MobaXterm, and a VPN. The code was entirely written in C language, divided into 3 files (main.c, utility.c, and utility.h). The Matrix Market library was used to read the matrices. For parallelization, the MPI library was used. All of this is then performed by a .pbs script on the node. The toolchain used is:

```
mpicc -std=c99 -O3 -fopenmp main.c utility.c mmio.c -o
```

Since MPI has higher noise, possible communication interference, and scheduling variability, multiple runs were performed for testing, from which various metrics were obtained for different numbers of processes: 1, 2, 4, 8, 16, 32, 64, and 128. The implementation only supports real general matrices from SuiteSparse (i.e., they have non-zero matrix values explicitly listed in the file, not symmetric, Hermitian, or other) to avoid storage issues. In addition, three randomly generated matrices were also analyzed to test weak scaling (only for p=1, p=8, and p=64). For each configuration, one warm-up run and five measured runs were performed, from which the average was then taken.
In general, several metrics were taken. The average execution time of the SpMV, the speedup, the efficiency, and the FLOPs.

IV. Results and discussion

As mentioned, the tests were carried out on seven matrices, the first four of which are real problem matrices (collected from SuiteSparse):

- matrix1: ≈ 300 x 300 with 8000 non-zero entries: small and structured, predictable ghosts
- matrix2: ≈ 50000 x 50000 with 380000, very irregular, ghosts potentially everywhere
- matrix3: ≈ 18000 x 18000 with 450000 non-zero entries, structured, famous in matrix operations
- matrix4: ≈ 1000000 x 1000000 with 3000000 non-zero entries, unstructured

and 3 randomly generated matrices, attempting to scale on 1, 8, and 64 processes, to also test weak scaling:

- randomP1: ≈ 50000 x 50000 with 50000 non-zero entries
- randomP8: ≈ 300000 x 300000 with 300000 non-zero entries
- randomp64: ≈ 100000 x 1000000 with 1000000 non-zero entries

A. Strong scaling performance and speedup

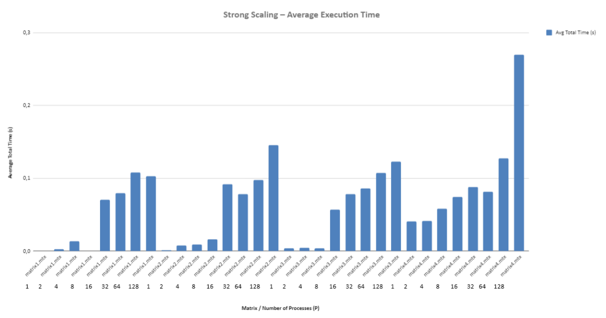

Fig. 1. All 4 matrices avg. execution time

Starting immediately with the data analysis, for the first matrix (Matrix1) it is immediately apparent that it is too small, there is no speedup, overhead dominates, and the average execution time is negligible.

In the case of the second matrix (Matrix2), the advantage of using "collective communication" for the distribution of ghosts, instead of point-to-point communication, which would have resulted in a very high volume of communication, is immediately apparent. And in fact, the data proves it: Despite this, matrix 2 has very poor speedup, undoubtedly due to communication.

TABLE I
POINT-TO-POINT TEST AVERAGE TIMES

	Average times of point-to-point	
matrix	number of P	avg time
matrix2.mtx	1	1.558e-03
matrix2.mtx	2	9.9998e-02
matrix2.mtx	4	3.1899e-01
matrix2.mtx	8	2.9336e-01
matrix2.mtx	16	1.6174e-01
matrix2.mtx	32	7.4936e-02
matrix2.mtx	64	4.4298e-02
matrix2.mtx	128	4.7355e-02

Matrix 3 shows more regular communication and a good computation/communication balance; it is definitely the matrix that scales best. However, as soon as communication increases with the number of processes, speedup worsens. Matrix4 is a huge unstructured matrix with very sparse ghosts.

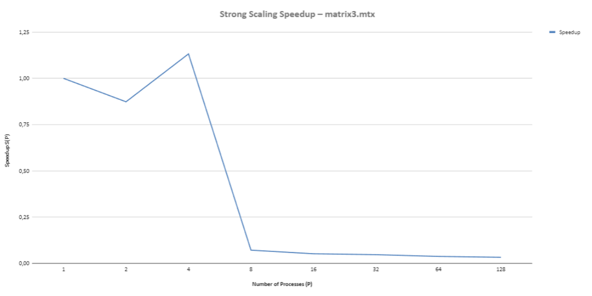

Fig. 2. Matrix 3 speedup

Once again, communication is dominant. Finally, efficiency decreases rapidly as the number of processors increases, confirming that the communication bottleneck dominates SpMV computation. Despite the speedup being less than 1 for most matrices, and despite the average execution times being good, the results clearly highlight the cost of communication and the limitations of 1D partitioning for distributed SpMV. Structured matrices perform better at low concurrency, while unstructured matrices suffer from a high number of ghost entries.

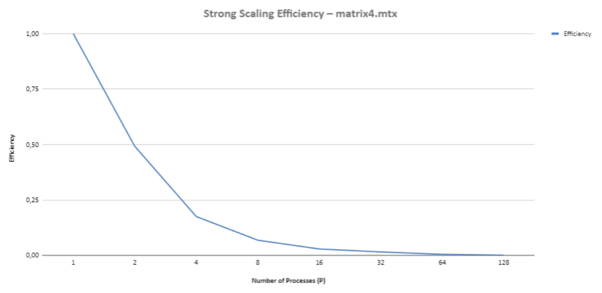

Fig. 3. Efficiency matrix 4

B. *Weak scaling*

While strong scaling is very limited by communication, caused by the irregularity of the matrices, weak scaling on random matrices shows averages that increase as the execution problem grows with the number of processes. This should comply with Gustafson's law.

TABLE II
WEAK SCALING TEST AVERAGE TIMES

	Average times of weak scaling	
matrix	number of P	avg time
randomP1.mtx	1	2.369e-03
randomP8.mtx	8	4.01e-03
randomP64.mtx	64	8.872e-02

C. *Bottleneck*

The tests clearly show the limitations of 1D partitioning in this implementation of SpMV. Despite the use of MPI, this mode is strongly dominated by the ghost entry exchange mode, which dominates execution time as the number of processes increases. All this is confirmed by the data reported in the strong scaling. So, we can say that **the communication makes the 1D partitioning strategy communication-bound** in this cases..

V. CONCLUSIONS

This work has seen the implementation of SpMV using an MPI-based implementation, following Foster's methodology and paying particular attention to data partitioning. For the latter, it was decided to adopt 1D partitioning, based on rows. The experiments clearly show that the implementation is severely limited by communication due to the strong presence of ghost elements, which require a high number of communications. In general, 1D partitioning works best on matrices with a regular structure, because unstructured matrices are matrices with more ghost entries, thus increasing the necessary communication. Therefore, it works best with matrices with uniform rows, low connectivity, and adequate locality. 2D would obviously be more advantageous with very large and irregular matrices, as they limit the number of messages per process. The best results would be obtained by using both partitions, 1D to reduce the volume of communication and then 2D to limit the number of messages. That said, 1D has proven to be a very valid partitioning method due to its simplicity and advantages, but it remains limited. Future work could certainly include optimization of the algorithm adopted for communication in 1D and definitely a version of 2D.

VI. References

References

[1] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. 2007. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In Proceedings of the 2007 ACM/IEEE conference on Supercomputing (SC '07). Association for Computing Machinery, New York, NY, USA, Article 38, 1–12. https://doi.org/10.1145/1362622.1362674

[2] Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. 2009. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures (SPAA '09). Association for Computing Machinery, New York, NY, USA, 233–244. https://doi.org/10.1145/1583991.1584053

[3] Devid Troka. Sparse Matrix-Vector Multiplication sequential vs OpenMP

[4] Performance portability of sparse matrix–vector multiplication implemented using OpenMP, OpenACC and SYCL Kinga Stec, Przemysław Stpiczyński Maria Curie-Skłodowska University, Institute of Computer Science, ul. Akademicka 9, 20-033, Lublin Poland

[5] Flavio Vella (2025). Introduction to parallel computing. University of Trento. Lecture 9.

[6] Flavio Vella (2025). Introduction to parallel computing. University of Trento. Lecture 10.

[7] Flavio Vella (2025). Introduction to parallel computing. University of Trento. Lecture 9.

[8] Scalable Matrix Computations on Large Scale-Free GraphsUsing 2D Graph Partitioning Erik G. Boman, Karen D. Devine, and Sivasankaran Rajamanickam. 2013. Scalable matrix computations on large scale-free graphs using 2D graph partitioning. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13). Association for Computing Machinery, New York, NY, USA, Article 50, 1–12. https://doi.org/10.1145/2503210.2503293

[9] Parallel sparse matrix-vector multiplication as a test case for hybrid MPI+OpenMP programming G. Schubert

[10] Algorithms for Parallel Shared-Memory Sparse Matrix-Vector Multiplication on Unstructured Matrices Kobe Bergmans, Karl Meerbergen, Raf Vandebril

[11] Introduction to Parallel Computing a.y. 2025 – 2026 Prof. Flavio Vella Recover Lecture 5

[12] Devid Troka. SpMV 1D partitioning with MPI

[13] Flavio Vella (2025). Introduction to parallel computing. University of Trento. Lecture 12.

[14] Flavio Vella (2025). Introduction to parallel computing. University of Trento. Lecture 13.

[15] Flavio Vella (2025). Introduction to parallel computing. University of Trento. Lecture 14.