Parallel Matrix Multiplication with Sparse Matrices

Mattias Trettel
University of Trento
Email: mattias.trettel@studenti.unitn.it

Ivan Donà
University of Trento
Email: ivan.dona@studenti.unitn.it

***Abstract*—Sparse matrix multiplication is a key operation in high-performance computing, with applications in scientific simulations, machine learning, and graph processing. This project explores efficient parallel algorithms for multiplying large sparse matrices using MPI, OpenMP, and hybrid approaches. We evaluate different sparse formats (COO and CSR) and distribution strategies (Direct and Reverse), focusing on performance, memory usage, and scalability. Results show that the Reverse distribution method, combined with the CSR format and hybrid parallelism, achieves the best balance between computation speed and resource efficiency. The analysis highlights critical trade-offs in parallel sparse matrix multiplication and provides insights into optimizing performance on multi-core and distributed systems.**

Github repository:https://github.com/ivandona/optimized-matrix-multiplication.

I. Introduction

Matrix multiplication is a core operation in many scientific and engineering applications. When dealing with large datasets, matrices are often sparse (containing mostly zero element), which allows for specialized algorithms that reduce both memory usage and computation time.

This project focuses on the parallel multiplication of sparse matrices, using compressed formats such as Coordinate (COO) and Compressed Sparse Row (CSR). Efficient parallelization is essential to achieve scalability, especially on modern multi-core and distributed systems.

We explore multiple implementation strategies using MPI, OpenMP, and hybrid approaches. Two main data distribution methods are compared: *Direct*, which assigns rows of one matrix and the full second matrix to each process; and *Reverse*, which distributes portions of both matrices to reduce redundant computation.

The goal is to evaluate the performance of each combination in terms of execution time, scalability, and memory efficiency. By analyzing trade-offs between computation and communication, we identify the most effective strategies for high-performance sparse matrix multiplication.

II. Sparse Matrices

Traditional matrix multiplication algorithms, which treat matrices as dense structures, are inefficient when applied to sparse matrices. Not only do they waste computational effort on zero-valued entries, but they also incur unnecessary memory and communication overhead. This inefficiency becomes increasingly significant as the problem size grows.

We implement and test algorithms able to handle this sparsity. The algorithms used in this project differ w.r.t. the format used to represent the matrices (COO vs CSR) and the data structures used for their implementation (array vs struct).

A. COO & CSR

When dealing with sparse matrices, we want to utilize a compressed representation of the matrix. We do this to avoid representing all the zeros inside of it. Two main formats are used for this project: COO and CSR.

1) COO: The COOrdinate format, also known as *ijv* or *triplet* format [1], stores the non-zero elements of the matrix as a triplet (`data`, `row index`, `column index`). In practice, we store these values in two different ways:

- COO with structs - the matrix is stored as an array of `struct`, where each `struct` has these three values.
- COO with arrays - the matrix is stored in three arrays, one for each of the values.

2) CSR (or CSC): The Compressed Sparse Row (or Column) format uses three arrays: one for the non-zero values (`data`), one for the column (or row) indices of those values (`indices`), and one (`indptr`) that marks where each row's (or column's) data starts in the other two arrays, therefore it is shorter. Since the third array is shorter, we do not use structs with CSR.

B. Implementations

For each of these approaches:

- COO with structs
- COO with arrays
- CSR with arrays

we developed a series of implementations. Each implementation is the result of a combination of the formats (CSR vs COO), the way in which the matrices are divided among the processes (Direct vs Reverse) and the type of parallelization (MPI vs OpenMP, vs Hybrid). In Fig. 1 we can see a schema of the possible combinations.

1) Direct: The Direct mode is the "standard" way to distribute the matrices among the processes. Given two matrices A and B, each process receives some rows of A and all of B. Each process can now compute a part of the rows of the result matrix C. After aggregating each part, we end up with the complete result matrix. In Fig. 2 we can see of how the division is done.

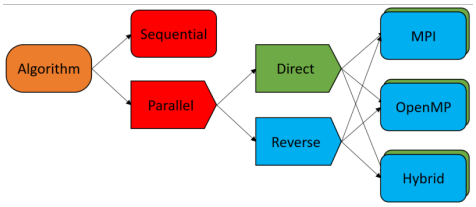

Fig. 1: Implementations of algorithms for sparse matrices

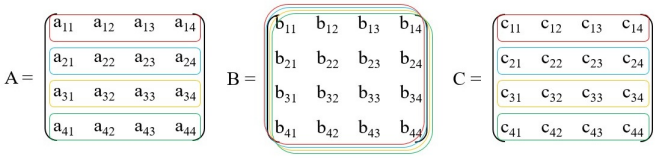

Fig. 2: Direct matrix division

2) Reverse: The Reverse mode is aimed to minimize the amount of duplicated data distributed among the different processes and avoid doing extra checks for redundant multiplication steps. In this case, each process will receive a part of A' columns and a part of B's rows. By doing so, we avoid duplicate data. The downside is that each process will get, after the multiplication, a matrix C_i of the same size of the final one. To obtain the final result matrix C, we just have to sum all these partial results. The division is shown in Fig. 3.

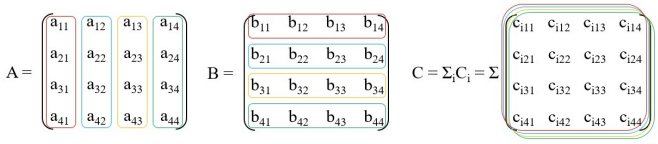

Fig. 3: Reverse matrix division

TABLE I: Pros and Cons of Direct and Reverse mode

Mode:	Direct	Reverse
Distribution phase	Data duplication	No data duplication
Multiplication phase	Comparisons needed	No comparisons needed
Aggregation phase	Simple aggregation	Sum of matrices needed

3) Direct vs Reverse: Pros and Cons: In Tab. I we can see a summary of the general differences between Direct and Reverse mode. Although all the aspects mentioned in the table influence the execution time, the most impacting one is for sure (not) having to make comparisons for the multiplication.

To understand why this is the case, let's analyze these two approaches using the multiplication between two sparse vectors as an example.

Direct case: Since the vectors are sparse, to find the correct pairs of elements to multiply, we need to go through both vectors and compare all their indices. In the example in Fig. 4, the arrows represent the comparisons made. Red arrows show correct pairs. Black arrows show incorrect ones.

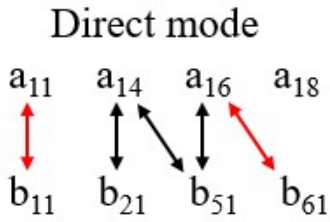

Fig. 4: Example of Direct mode

Reverse case: Since the two vectors represent a column of A and a row of B, we can directly compute the product between each pair of elements (as seen in Fig. 5). This allows us to not only avoid **needless comparisons**, but to **completely remove them**.

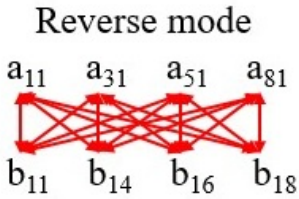

Fig. 5: Example of Reverse mode

Lastly, it is important to know that the **total number of red arrows is the same** for both approaches, i.e. the necessary products to compute the result are the same (given two matrices). The example with Direct mode has less red arrows per iteration because **each process has to go through more iterations**. Intuitively, the process handling the "red" part in Fig. 2 has to four iterations, one for each column of B, while in Fig. 3 has to do just one.

C. Results

Note: All the tests were run on the HPC cluster of the University of Trento.

1) Preliminary comparisons: There are a lot of possible combinations to test. For this reason, we do some preliminary tests to know which are the most promising. To compare the different variations of algorithms, we use two sparse square matrices of dimension 16384 (2^{14}) and non-zero element density of 15%. The tests are performed on 64 cores, divided across processes and threads. Now we compare the execution times of our 3 algorithms, each in its two modes: Direct and Reverse. Each algorithm uses the type of parallelization that matches the resource distribution used. The total time is calculated as the average of ten runs. To make the results more robust, we used the "`=pack:excl`" option. This ensures that all processes run on the same node, so they are physically close, and there are no other users' programs that could affect the results. The results of these tests are shown in Fig. 6. In the graph, shortened notations are used (a=array, s=struct,

d=direct, r=reverse, p=parallel). For example, “COO struct parallel reverse” is abbreviated as “COO_spr”.

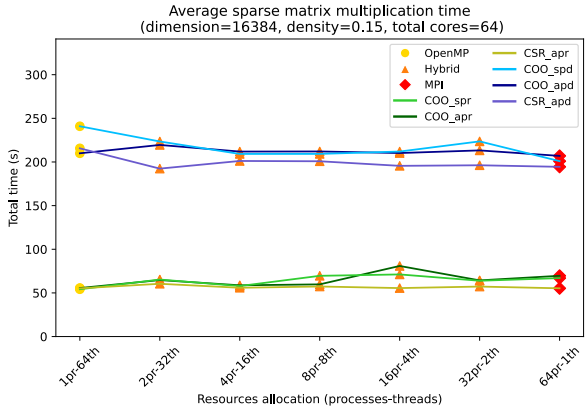

Fig. 6: Preliminary comparison

The graph in Fig. 6 clearly shows the performance difference between the Direct and Reverse modes. By analyzing the timing of the individual sections (shown in Fig. 7), we can understand what causes this gap. We can observe three points:

- The distribution time is nearly the same. This indicates that data duplication in the Direct mode, due to the broadcast of a matrix, does not significantly impact performance.
- The aggregation time is higher in the Reverse mode, because it requires summing a large number of matrices.
- The multiplication time is where the biggest difference lies. This happens because, as explained earlier, multiplication in the *Direct* mode requires a large number of comparisons, while the *Reverse* mode uses none.

2) Benchmark (Best algorithms): At this point, we can focus only on the algorithms related to the *Reverse* mode. Now we compare performance based on the number of nodes used, forcing each chunk to be allocated on a different node using the option `place=scatter`. As before, we will use various combinations of processes and threads, for a total of 256 cores.

From the graph shown in Fig. 8, we can see that even when varying the resource distribution across nodes, all programs show similar performance. Some interesting patterns also emerge:

- Given a certain number of nodes, in general, the best distribution is one MPI process per node, with the remaining cores used as threads.
- The performance follows a parabolic trend, with the minimum at 32 nodes. Due to limited resources, we stopped at 64 nodes, but it is likely that this trend continues as the number of nodes increases. This highlights the importance of having a proper balance between nodes, MPI processes, and threads.

In conclusion, the CSR format is statistically the one with the best performance. It is also always preferred over the COO

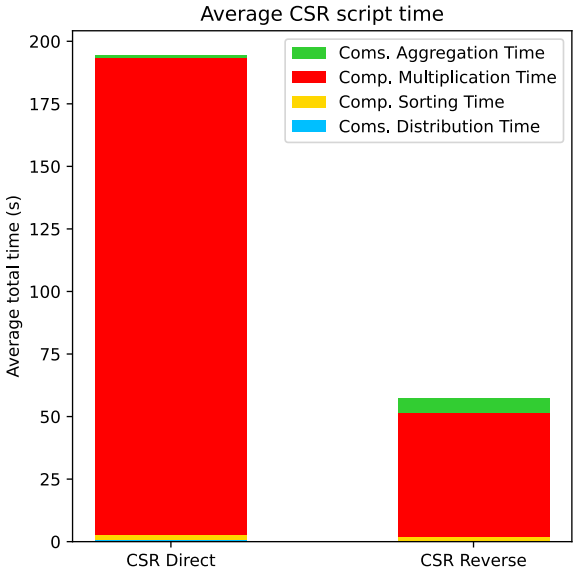

Fig. 7: Time comparison between individual sections

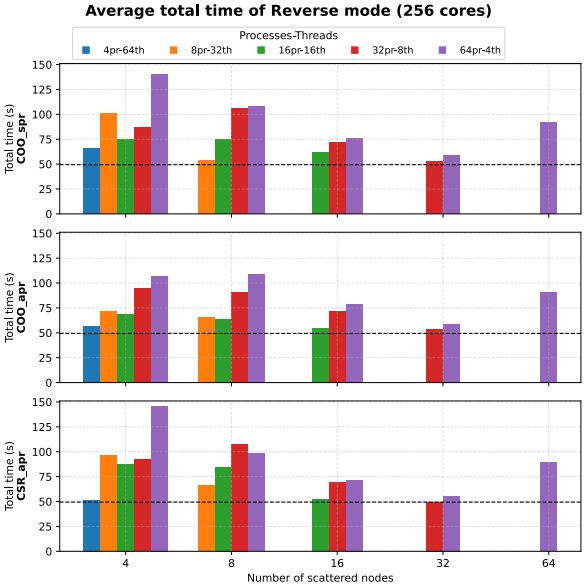

Fig. 8: Time comparison of algorithms using Reverse mode

format in terms of memory usage, as seen in Fig. 9. A sparse matrix in COO format requires the allocation of three vectors, each with a size equal to the number of non-zero elements. In the CSR (or CSC) format, instead of the row (or column) index vector, there is a vector whose size is equal to the number of rows plus one. This vector represents the offsets of each row, thus saving memory. The gap between the two formats increases as the size of the matrix grows.

D. Data dependency

Let’s analyze the dependencies observed in the CSR-based algorithm, since it is representative of all the dependencies

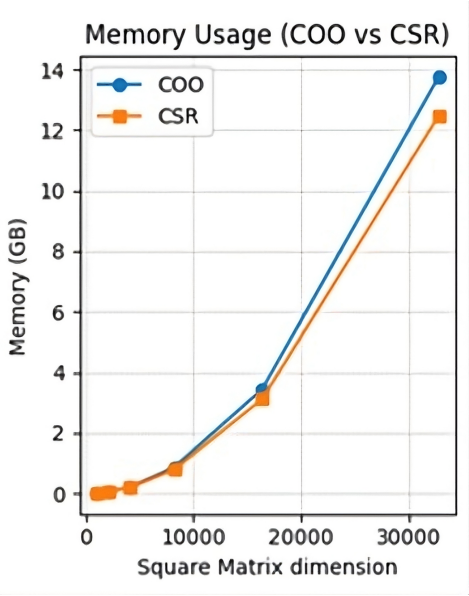

Fig. 9: COO vs CSR memory usage

observed in the other two algorithms (COO with array/structs) and implement ways to solve them only for the Reverse mode, because for the Direct mode is trivial. These dependencies can be divided into 4 groups, each tied to a certain memory location:

- `col_counts[]`: A temporary support array of size n (number of columns) to sort the matrix. Initially used to count how many non-zero elements appear in each column of the original CSR matrix. Later reused (after computing `col_ptr_csc`) as a decrementing counter to determine the correct position for each element in the CSC arrays.
- `col_ptr_csc[]`: The column pointer array of the resulting CSC matrix. For each column j, `col_ptr_csc[j]` stores the index in the CSC data arrays where that column starts. Computed by prefix-summing the values in `col_counts[]`.
- `col_counts[col]`: During the filling phase of CSC arrays, this specific access to `col_counts` is used to compute the next insertion position in the column `col`. Decremented after each use to ensure that multiple entries in the same column are written contiguously from bottom-up.
- `C[...]`: Represents the final matrix as an array.

The summary of these dependencies can be seen in Tab. II.

To resolve these dependencies, we used different strategies. For groups 1 and 3, an `#pragma omp atomic` directive was simply used. For group 2, nothing was done, as the impact on performance was negligible and it was reasonable to assume that removing the dependency could even worsen the situation. For group 4, the problem occurs when writing to the array C (see code in Fig. 10). In fact, it is possible for two threads to manipulate the data at the same time, making it inconsistent.

To solve this problem, we have two possible approaches:

TABLE II: Data dependency analysis results for CSR version.

Data Dependency Analysis (CSR)								
Memory Location	Earlier Statement			Later Statement			Loop-carried?	Kind of Dataflow
	Line	Iter.	Access	Line	Iter.	Access		
col_counts[]	101	i	read	101	i	write	no	anti
col_counts[]	101	i	read	101	i+1	write	yes	anti
col_counts[]	101	i	write	101	i+1	read	yes	flow
col_counts[]	101	i	write	101	i+1	write	yes	output
col_ptr_csc[]	106	i	write	106	i+1	read	yes	flow
col_counts[col]	113	i	read	113	i	write	no	anti
col_counts[col]	113	i	read	113	i+1	write	yes	anti
col_counts[col]	113	i	write	113	i+1	read	yes	flow
col_counts[col]	113	i	write	113	i+1	write	yes	output
C[...]	150	i	read	150	i	write	no	anti
C[...]	150	i	read	150	i+1	write	yes	anti
C[...]	150	i	write	150	i+1	read	yes	flow
C[...]	150	i	write	150	i+1	write	yes	output

```
int k, i, j;
for (k = 0; k < n; k++) {
  for (i = A_col_ptr[k]; i < A_col_ptr[k+1]; i++) {
    for (j = B_row_ptr[k]; j < B_row_ptr[k+1]; j++) {
      C[A_row[i]*n + B_col[j]] += A_value[i] * B_value[j];
    }
  }
}
```

Fig. 10: Serial code of the matrix multiplication

- *Parallelize the outer loop*. This allows us to initialize the threads only once and use an `atomic` directive to serialize access to the matrix, thus avoiding collisions. However, this introduces a significant bottleneck, since memory writes are the slowest operations, and this approach serializes a number of operations that grows with the cube of N.
- *Parallelize the inner loop*. This avoids serializing access to the matrix, but we must initialize the parallel section multiple times, which introduces a non-negligible overhead that grows linearly with N. This approach is possible due to a "property" of this mode (Reverse): after the outer loop, the remaining two loops iterate over a column of A and a row of B, multiplying each pair of elements. For a collision to occur, there would need to be two distinct pairs that simultaneously have the same row index from the element in A and the same column index from the element in B, since these two values are the destination cell indices. However, this is not possible, because in a column of A there cannot be two elements on the same row, and likewise, in a row of B there cannot be two elements in the same column.

Both implementations are shown in Fig. 11 and 12. After some tests (by comparing the timings), parallelizing the inner loop proved best.

E. Speedup and Efficiency

We now focus on analyzing in more detail the performance of the best-performing program: the CSR format, using arrays as the representation and the *Reverse* mode. Since no optimal

```
int k;
#pragma omp parallel for num_threads(num_threads)
for(k=0; k<n; k++){
    int i;
    for(i=A_col_ptr[k]; i<A_col_ptr[k+1]; i++){
        int j;
        for(j=B_row_ptr[k]; j<B_row_ptr[k+1]; j++){
            int index = A_row[i]*n + B_col[j];
            double val = A_value[i] * B_value[j];
            #pragma omp atomic
            C[index] += val;
        }
    }
}
```

Fig. 11: Matrix multiplication with parallelization of the outer loop

```
int k;
for(k=0; k<n; k++){
    int i;
    #pragma omp parallel for num_threads(num_threads)
    for(i=A_col_ptr[k]; i<A_col_ptr[k+1]; i++){
        int j;
        for(j=B_row_ptr[k]; j<B_row_ptr[k+1]; j++){
            int index = A_row[i]*n + B_col[j];
            double val = A_value[i] * B_value[j];
            C[index] += val;
        }
    }
}
```

Fig. 12: Matrix multiplication with parallelization of the inner loop

configuration emerged from the previous study, we proceed by analyzing all of them, with a maximum of 64 total cores. We also used both chunk distribution options discussed earlier: `pack:excl` and `scatter`, which should represent, respectively, the best and worst options among those available. The results can be seen in Tab. III and IV.

TABLE III: Average Speedup CSR Array Parallel Reverse

Average Speedup CSR Array Parallel Reverse								
Place Option	**Cores**	**1**	**2**	**4**	**8**	**16**	**32**	**64**
Pack Exclusive	OpenMP	1.00	1.70	3.03	6.69	12.17	16.98	25.29
	MPI	1.00	1.95	3.89	5.64	11.54	14.38	29.93
	Hybrid (2 pr)	1.00	-	3.33	5.45	11.93	16.68	33.63
	Hybrid (4 pr)	1.00	-	-	6.56	11.11	16.90	29.28
	Hybrid (8 pr)	1.00	-	-	-	12.27	18.55	30.05
	Hybrid (16 pr)	1.00	-	-	-	-	15.68	31.11
	Hybrid (32 pr)	1.00	-	-	-	-	-	28.01
Scatter	MPI	1.00	1.48	2.81	4.88	8.47	12.21	20.73
	Hybrid (2 pr)	1.00	2.62	4.41	7.00	12.45	12.45	21.73
	Hybrid (4 pr)	1.00	-	3.90	7.02	12.00	12.00	20.07
	Hybrid (8 pr)	1.00	-	-	6.76	11.32	11.32	18.04
	Hybrid (16 pr)	1.00	-	-	-	-	11.26	17.99
	Hybrid (32 pr)	1.00	-	-	-	-	-	20.47

1) Weak and strong scalability: Let's analyze the scalability of our algorithms. As seen from Fig. 13, the program has not good strong scalability. This aspect emerges also from the table on efficiency (Tab. IV), where it decreases with the increase of cores. To better understand what is causing this drop in performance, we take a closer look at CSR Reverse Hybrid algorithm. In particular, let's consider the timings of

TABLE IV: Average Efficiency CSR Array Parallel Reverse

Average Efficiency CSR Array Parallel Reverse								
Place Option	**Cores**	**1**	**2**	**4**	**8**	**16**	**32**	**64**
Pack Exclusive	OpenMP	1.00	0.85	0.76	0.84	0.76	0.53	0.40
	MPI	1.00	**0.98**	**0.97**	0.70	0.72	0.45	0.47
	Hybrid (2 pr)	1.00	-	0.83	0.68	0.75	0.52	0.53
	Hybrid (4 pr)	1.00	-	-	0.82	0.69	0.53	0.46
	Hybrid (8 pr)	1.00	-	-	-	0.77	0.58	0.47
	Hybrid (16 pr)	1.00	-	-	-	-	0.49	0.49
	Hybrid (32 pr)	1.00	-	-	-	-	-	0.44
Scatter	MPI	1.00	**0.74**	**0.70**	**0.61**	**0.53**	0.38	**0.32**
	Hybrid (2 pr)	1.00	-	0.65	0.55	0.45	**0.39**	**0.34**
	Hybrid (4 pr)	1.00	-	-	0.49	0.44	0.37	0.31
	Hybrid (8 pr)	1.00	-	-	-	0.42	0.35	0.28
	Hybrid (16 pr)	1.00	-	-	-	-	0.35	0.28
	Hybrid (32 pr)	1.00	-	-	-	-	-	0.32

each core, normalized by workload (Fig. 15. On the MPI side, the aggregation time increases due to the ever growing number of matrices to sum. On the OpenMP side the increase in time is caused by the fact that we initialize the parallel region N times.

For the same reason, weak scalability is also lacking.

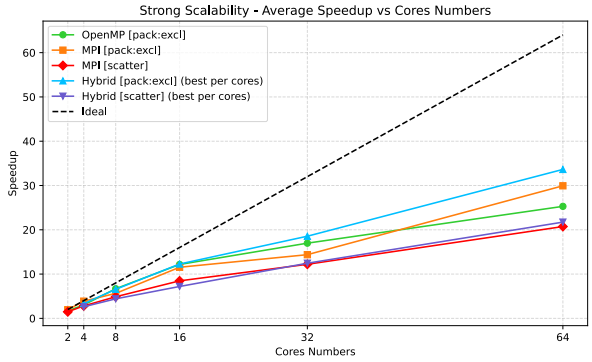

Fig. 13: Strong scalability analysis

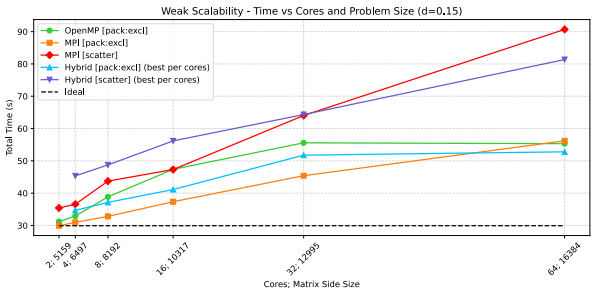

Fig. 14: Weak scalability analysis

III. Conclusion

In this project, we investigated various strategies for parallel sparse matrix multiplication using different data formats (COO and CSR), parallelization models (MPI, OpenMP, and hybrid), and distribution methods (Direct and Reverse).

Our results show that the *Reverse* distribution method consistently outperforms the *Direct* approach in multiplication time due to its ability to eliminate index comparisons. Despite

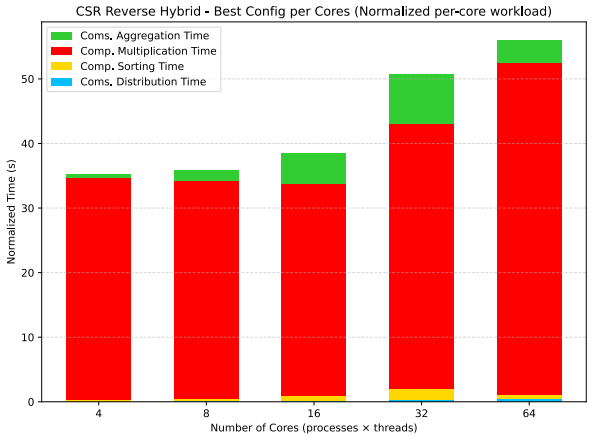

Fig. 15: Weak scalability analysis

a higher aggregation cost, Reverse mode proves more efficient overall, especially for large matrices.

Among the tested formats, the CSR representation offered the best performance in both computation speed and memory usage. Hybrid parallelization, combining MPI and OpenMP, provided the best scalability when properly balanced across nodes and cores.

While strong and weak scalability were not ideal in all configurations, performance trends highlighted the importance of minimizing synchronization overhead and optimizing memory access patterns.

Overall, the study demonstrates that careful selection of data structures, partitioning schemes, and parallelization strategies can significantly enhance the efficiency of sparse matrix multiplication on modern computing architectures.

REFERENCES

[1] SciPy community, *"coo_matrix — SciPy v1.16.0 Manual"*, SciPy Documentation;

[2] S. C. Eisenstat, M. C. Gursky, M. H. Schultz, and A. H. Sherman, *Yale Sparse Matrix Package: II. The Nonsymmetric Codes*, Research Report No. 114, Department of Computer Science, Yale University, New Haven, CT, Aug. 1977.