Parallel Project Report: Performance Analysis of Parallel Sparse Matrix Multiplication

Giovanni Todesco (ID: 244645)
Email: giovanni.todesco@studenti.unitn.it
Course: Introduction to Parallel Computing (2025-2026)
GIT: https://github.com/T00dd/PARCO-Computing-2026-244645.git

***Abstract*—This work presents a performance analysis of the SpMV algorithm (Sparse Matrix-Vector Multiplication) parallelized by the MPI standard in a distributed enviroment. The SpMV operation (y=Ax) remains a critical bottleneck in numerous scientific computing domains because of its low arithmetic density. We use the Compressed Sparse Row (CSR) format on a modern Non Uniform Memory Access (NUMA) architecture. The implementation uses a 1D decomposition of the domain and an optimize mechanism of 'ghost entries' to minimize the communication overhead during the exchange of data. The primary objective is to evaluate the scalability of the algorithm through a strong scaling and weak scaling tests. We also analyze the load balance in terms of non-zero elements (NNZ) and the communication volume for each rank.**

The results show strong scaling with speedups from 16x to 47x at 128 processes, but efficiency varies widely (12.5%-49.4%) based on matrix structure. We observe that communication accounts for only 2-13% of execution time, revealing kernel inefficiency and load imbalance as primary bottlenecks. Weak scaling fails dramatically, dropping to 1% efficiency at 128 processes due to round-robin distribution scattering local rows across distant ranks.

***Index Terms*—MPI processes, SpMV, CSR, NUMA, Weak scaling, Strong scaling, Load imbalance, Communication imbalance**

I. Introduction

Problem. SpMV despite its simplicity has two different problems that lead to weak performances: low arithmetic intensity, making the algorithm memory bandwidth-bounded (as we can see from the roofline model explained by Samuel Williams [1]) and indirect memory access (due to CSR format), which leads to high cache misses.

Objectives. (i) Implement a distributed SpMV using 1D row partitioning; (ii) design and evaluate an optimized Ghost Entries communication mechanism to handle vector element exchange; (iii) quantify performance through both Strong Scaling and Weak Scaling benchmarks; (iv) analyze load balancing by measuring NNZ distribution and communication volumes across ranks; (v) document a reproducible workflow.

II. State of the art

The optimization of SpMV is a very common field in High Performance Computing (HPC).

CPU and storage format. The CSR format is a standard due to its storage efficiency, but it suffers from irregular memory access patterns which degrade spatial locality [2]. Different researches proposed others storage formats, like Blocked-CSR or SELL-C-σ, to maximize vectorization and cache reuse. However, these type of formats often introduce lots of preprocessing overhead, making standard CSR the more efficient choice when the matrix structure is unknown or changes dynamically [3].

CPU and storage format. On shared-memory multicore architectures, the primary challenge for SpMV is load imbalance because the majority of the rows has few non-zero values and few rows has lots of non-zero values. Standard 1D row-wise partitioning is commonly used to distribute the matrix across MPI processes. The "Ghost entries" mechanism involves identifying and communicating only the specific subset of the input vector x required for local computations, drastically reducing communication volume compared to global collectives. Our work builds on these principles by combining optimized ghost exchange with direct index mapping to minimize computational latency.

III. Contribution and Methodology

A. Contribution

We made a distributed-memory SpMV implementation using pure MPI, trying to eliminate the performance noise associated with shared-memory threading in large-scale clusters. We developed an automated benchmarking workflow that scales up to 128 MPI processes, utilizing real-world matrices from SuiteSparse for strong scaling tests and synthetic matrices for weak scaling analysis. We provide also a load-balancing analysis that quantifies NNZ distribution and communication volume per rank, providing data into how matrix structure influences parallel efficiency.

B. Methodology

Firstly we convert the format from (COO) to Compressed Sparse Row format (CSR), to store more efficiently the non-zero elements. With the CSR format a matrix is compressed in three arrays: val (stores the non zero values), col_ind (stores the column indices corresponding to val) and row_ptr (stores the starting index of each row in val). Than we create the global vector with random values using the $rand$ function. To divide the matrices between all the processes we utilize a 1D cyclic partitioning strategy. The rule is `owner(i)=imodP` where `P` is the number of processes and `i` is the row of the matrix. Each process stores all non-zeros belonging to its rows. The global vector is also divided with the same method.

Ghost Entries and Communication. To compute $y_i = \sum_j A_{ij}x_j$, a process can require some elements of x that is stored in other ranks. We implemented a Ghost Entries mechanism to handle these cases:

- **Identification**: during the setup each rank scans its local columns and identifies witch indices j is out of its range
- **Mapping**: we perform a local index remapping. Global column indices are translated into a local workspace where local vector elements are followed by ghost elements. This allows to access vector data without expensive global-to-local lookups during multiplication.
- **Exchange**: we utilize MPI primitives (`MPI_Alltoall`) to exchange the ghost values creating the $x\_extended$ vector.

Parallel Kernel. Algorithm 1 calculates the local multiplication $y\_local = A\_local \times x\_extended$.

- **CSR iteration**: for each assigned row, the kernel iterates the row_ptr array to identify the range of non-zero elements.
- **Direct mapping**: it performs the product between the matrix value val and the extended vector.
- **O(1) access**: because of the remapping during the setup, the kernel accesses both local and ghost vector elements using a single, direct index lookup
- **Independent result**: each process computes its part of the result vector y independently.

Algorithm 1 OpenMPI Parallel SpMV Kernel

```
Require: $row\_ptr, col\_ind, val, x\_extended$
Ensure: $y \leftarrow Ax$
1: for $i = 0$ to $M - 1$ do
2:     $sum \leftarrow 0$
3:     for $j = row\_ptr[i]$ to $row\_ptr[i+1]$ do
4:         $sum \leftarrow sum + val[j] \times x\_extended[col\_ind[j]]$
5:     end for
6:     $y[i] \leftarrow sum$
7: end for
```

IV. Experiments and System Description

A. Computing system

The experiments were conducted on the University of Trento's cluster, in 4 nodes of a High-Performance Computing (HPC), designed with a Non-Uniform Memory Access (NUMA) architecture. The system specifications are:

- **CPU**: 128 logical cores distributed across multiple sockets and nodes
- **Memory**: 32GB of RAM for each node
- **Operating System**: CentOS Linux release 7.6.1810

B. Software configuration

The code was compiled using the GCC 9.1 compiler wrapper `mpicc` with the `-O3` optimization flag. We utilize the MPICH 3.2.1 library to manage the distributed memory enviroment.

Process Affinity and NUMA Memory Policy. To make the performance more stable and reducing operating system noise due to a NUMA architecture, we configure MPI process placement and memory allocation using the following strategies:

- `-bind-to core` binds each MPI process to a specific physical core, preventing process migration between sockets during execution
- `-ppn $PPN` controls the number of processes per node, calculated dynamically based on total process count and available nodes
- `numactl --interleave=all` interleaving distributes data evenly across all memory banks, preventing bandwidth saturation on any single NUMA node

On NUMA systems, memory access time depends on the memory location relative to the processor. If processes migrate between sockets, they may need to fetch data from remote memory banks, increasing the latency.

C. Dataset

In the benchmark we utilized 2 different types of matrices to evaluate the scaling behavior:

- **Strong scaling**: we select five sparse matrices from the SuiteSparse Matrix Collection with different sizes and sparsity, allowing us to stress our algorithm. We selected only matrices with Real values and Unsymmetric structures: $Twotone$, $HV15R$, $Cage15$, $Torso1$ and $Memchip$ as shown in Table I
- **Weak scaling**: to test the capability of the algorithm to handle increasing workloads, we generated, through a python script, 8 synthetic matrices designed so that the number of rows grows linearly with the number of processes, maintaining always the same number of NNZ

D. Benchmarking

To get accurate results, we follow this protocol:

- **Scaling range**: both strong and weak scaling tests are executed with $P = \{P = 1, 2, 4, 8, 16, 32, 64, 128\}$ MPI processes
- **Execution loop**: the ghost entries exchange and the local SpMV kernel is executed for 10 iterations considering only the 90 percentile
- **Time analysis**: the execution time is divided into communication time (ghost values exchange) and multiplication time (local CSR kernel). Since the bottleneck is the slowest process, we collect only the maximum time across all ranks using `MPI_Reduce` with `MPI_MAX` for both communication time and computation.
- **Load balance metrics**: each rank calculates his local number of NNZ and its communication volume. Than

we compute the iimbalance factor to analyze the impact of the matrix structure on the performance.

$$Imbalance\% = \frac{Max\_load - Avg\_load}{Avg\_load} \times 100$$

E. Design of Experiments

We designed our experiments to answer three main questions:

- **Strong scaling**: We check how much faster the code runs when we add more processes (from 1 to 128) on a fixed-size matrix. The primary metrics are speedup ($S_p = T_1/T_p$) and efficiency ($E = S_p/P$).
- **Weak scaling**: We test the ability to maintain constant the execution time under fixed local workload conditions.
- **Structural impact**: we correlate the matrix structure with the imbalance factor to explain the efficiency drops. This permits to quantify the penalty of 1D partitioning.

TABLE I
DATASET

Matrix	Rows	Columns	Non-zero	Sparsity(%)
twotone	120,750	120,750	1,206,265	0.0083%
HV15R	2,017,169	2,017,169	283,073,458	0.00007 %
cage15	5,154,859	5,154,859	99,199,551	0.0000037%
torso1	116,158	116,158	8,516,500	0.063%
memchip	2,707,524	2,707,524	13,343,948	0.00018%

V. Results and Discussion

A. Peak performance and speedup

As shown in Fig. 1, all the matrices show a clear performance improvement as we increase the number of processes (strong scaling). We can see from TABLE II that the we achieve a maximum speedup between $15.99x$ and $46.89x$, that is far from a linear speedup. While the system demonstrates scalability with all types of matrices, the parallel efficiency varies a lot, peaking at 49.4% for twotone (at 64 ranks) and dropping to 12.5% for cage15. The discrepancy between the high speedup and low efficiency tells us that: while adding resources reduces execution time, the return on investment diminishes rapidly due to algorithmic overheads.

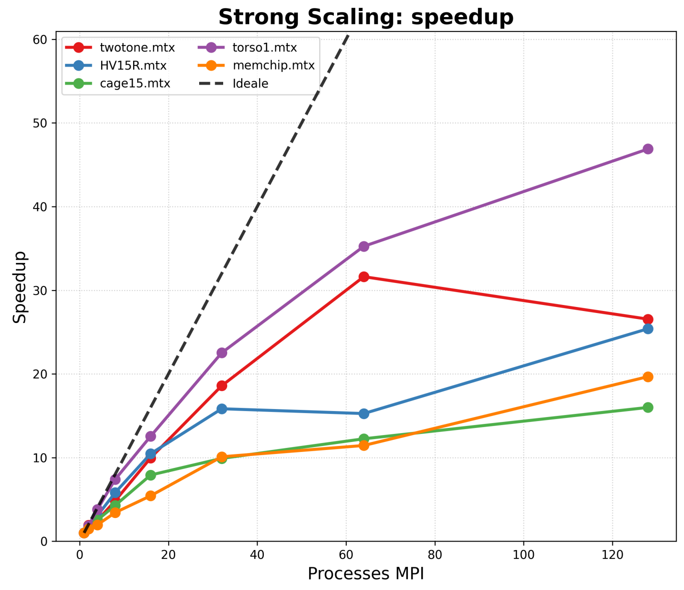
TABLE II
SPEEDUP

Matrix	Speedup	Processes	GFLOPS	Comm (ms)	Calc (ms)
twotone	31.62x	64	39.80	0.0038	0.0615
HV15R	25.40x	128	27.22	1.8050	20.7989
cage15	15.99x	128	15.49	1.9631	12.8055
torso1	46.89x	128	49.17	0.0077	0.3464
memchip	19.68x	128	18.74	0.2372	1.5807

B. Single matrix analysis

twotone. This matrix exhibits the classic ”latency-bound” behavior. At 64 processes, it reaches a peak speedup of 31.6x. However it slows down at 128 processes and this happens because the matrix is too small. At 128 ranks, the number of local NNZ for each rank became so low that the useful computation time is negligible. Consequently, the execution becomes dominated by the overhead of managing MPI message envelopes. The communication time increases drastically relative to the computation making it more inefficient.

HV15R. It’s a structured matrix and it’s the largest matrix in the dataset, successfully scaling up to 128 processes. The matrix remain computation-bounded and even at high number of processes the multiplication time dominates the communication overhead. We can also see (Fig. 2) how with 1D row partitioning the distribution of the load and volume is balanced due to the structural regularity of the matrix.



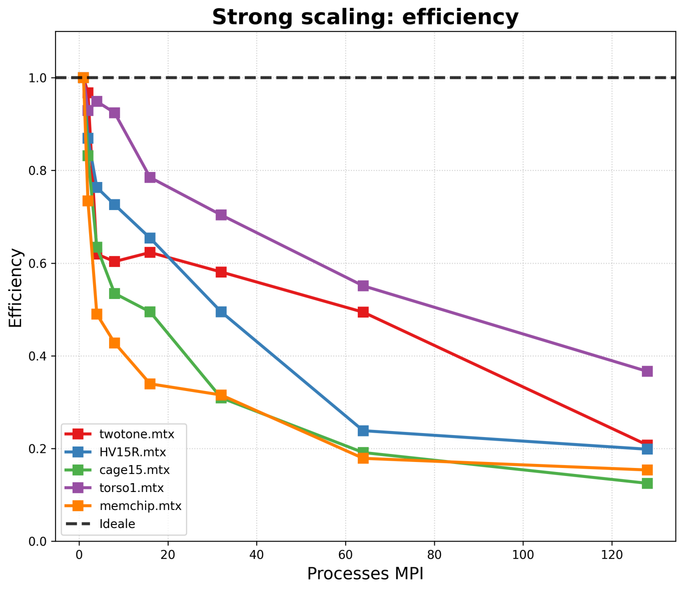


Fig. 1. Strong scaling

cage15. This matrix shows the lowest efficiency among the dataset, despite continuing to scale. Fig. 2 shows us a counter-intuitive behavior. As we increase the number of ranks, the communication imbalance decreases while the load imbalance grows only slightly. This is due to the fragmentation of communication. Because the matrix is highly interconnected, each process needs small pieces of data from almost every other process. Instead of exchanging a few large data packets (which is efficient), the network is flooded with thousands of tiny messages. The fixed "startup cost" (latency) of sending each small message accumulates, meaning the system spends more time establishing connections than actually transferring data.

torso1. This matrix is highly imbalanced but achieves the highest performance (46.9x) despite a clear anomaly shown in Fig. 2. Here the Load Imbalance grows significantly (the "Max NNZ" error bar at 128 ranks shows a 60% deviation from the average) and with it also the communication imbalance. Normally this would destroy the performance, however, the super-linear cache effect masks the problem. The dataset is small enough that even the overloaded rank fits its entire part into the cache. The speed boost is significant and it masks the penalty of the imbalance, allowing to run at high speed despite the poor distribution.

memchip. This matrix is a circuit simulation. The uniform distribution of circuit components results in excellent load balancing with less than 2% imbalance at all scales. Each process handles nearly identical workloads, with NNZ variance below 2% even at 128 processes. However, the overall efficiency remains low because communication costs grow significantly. At 128 processes, 13% of the execution time is spent on communication, compared to lower process counts. However, circuit matrices have long-range electrical connections that span across different regions. With 1D row partitioning, these cross-region dependencies force extensive inter-process communication. This structural characteristic causes communication overhead to grow up to 13% at 128 processes. The speedup reaches 19.7x but shows a visible slowdown between 32 and 64 processes where communication begins dominating the computation. Unlike HV15R's localized grid structure, memchip's structure requires widespread data exchange despite perfect balance, limiting efficiency.

C. Weak scaling analysis

Despite constant workload per process, as we can see from Fig. 3, efficiency drops to less than 1% at 128 processes. The round-robin row assignment (row i goes to rank $i\%P$) is the reason for this. This strategy scatters consecutive matrix rows across different processes. Since sparse matrices typically have non-zeros connecting nearby rows, this scattering turns local row interactions into remote communications. As we add more processes, neighboring rows become more spread out, forcing each rank to contact more distant processes for data.
The numbers confirm this effect: computation time stays likely constant, but communication explodes to 44% of total time at 128 processes. More processes mean more fragmented row distribution, which means more inter-process communication even if there is an identical per-process workload

VI. Conclusions

A. Summary of key findings

This work presented a performance analysis of an MPI-distributed SpMV algorithm with 1D round-robin partitioning. Our experiments lead to three primary conclusions:

Scaling limits: The algorithm shows strong scaling up to 128 processes, with speedups between 16x and 47x. However, efficiency varies widely ($12.5\% - 49.4\%$) due to the inefficiency of the kernel and load imbalance issues [4]. Communication overhead it's not the actual bottleneck because it accounts for only $2 - 13\%$ of the execution time [5].

Weak scaling failure: 1D row distribution causes bad weak scaling, dropping to less than 1% efficiency at 128 processes [6]. This strategy scatters neighboring rows across distant ranks, transforming local operations into remote communication with growth in ghost elements.

Structural dependency: matrix structure dominates performance [5]. Only regularly structured matrices like HV15R maintain reasonable efficiency. Irregular matrices suffer from either communication fragmentation (cage15) or load imbalance (torso1), though cache effects can partially compensate when working sets fit in cache.

B. Impact and future work

Our findings show that there isn't a "one-size-fits-all" approach to distributed SpMV. Simple round-robin partitioning works reasonably for structured matrices (HV15R) but fails when it meets irregular ones. The 1D distribution strategy proves fundamentally incompatible with sparse matrix locality, as evidenced by the weak scaling collapse to 1% efficiency. Future research should focus on structured aware partitioning [7] or 2D block partitioning [8]. We also plan to investigate on MPI+OpenMP approaches to exploit shared memory on the single node, reducing internode communication.

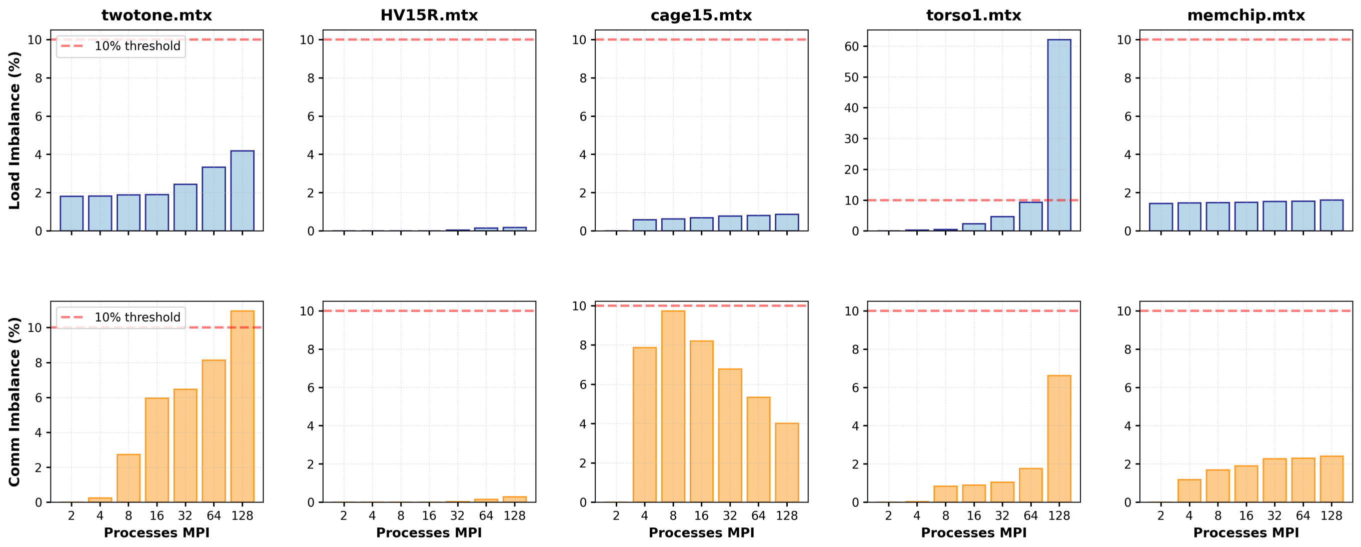


Fig. 2. Load and communication imbalance

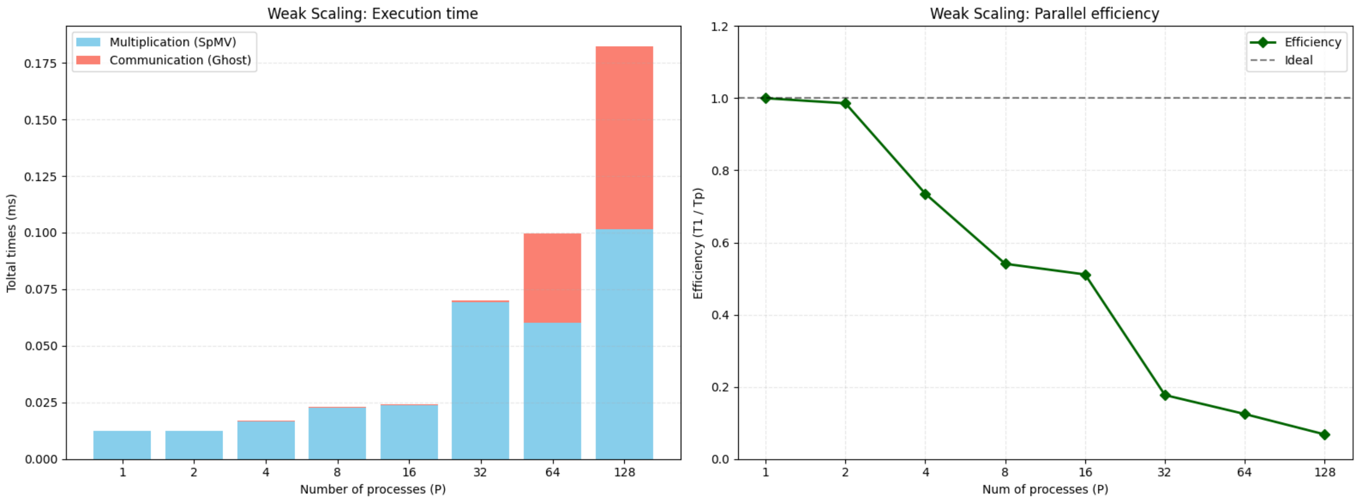


Fig. 3. Weak scaling

References

[1] Samuel Williams, Andrew Waterman, and David Patterson. 2009. ”Roofline: an insightful visual performance model for multicore architectures.” Commun. ACM 52, 4 (April 2009), 65–76. https://doi.org/10.1145/1498765.1498785

[2] J. L. Greathouse and M. Daga, ”Efficient Sparse Matrix-Vector Multiplication on GPUs Using the CSR Storage Format,” SC ’14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, New Orleans, LA, USA, 2014, pp. 769-780, https://doi.org/10.1109/SC.2014.68

[3] Kreutzer, Moritz, Hager, Georg, Wellein, Gerhard, Fehske, Holger, Bishop, Alan R., ”A Unified Sparse Matrix Data Format for Efficient General Sparse Matrix-Vector Multiplication on Modern Processors with Wide SIMD Units” SIAM Journal on Scientific Computing, 2014. https://epubs.siam.org/doi/abs/10.1137/130930352

[4] U. V. Catalyurek and C. Aykanat, ”Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication,” in IEEE Transactions on Parallel and Distributed Systems, vol. 10, no. 7, pp. 673-693, July 1999. https://doi.org/10.1109/71.780863

[5] A. Elafrou, G. Goumas and N. Koziris, ”Performance Analysis and Optimization of Sparse Matrix-Vector Multiplication on Modern Multi- and Many-Core Processors,” 2017 46th International Conference on Parallel Processing (ICPP), Bristol, UK, 2017, pp. 292-301,https://doi.org/10.1109/ICPP.2017.38

[6] H. M. Aktulga, C. Knight, P. Coffman, K. A. O’Hearn, T.-R. Shan, and W. Jiang, ”Optimizing the Performance of Reactive Molecular Dynamics Simulations for Multi-Core Architectures,” in Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2017, pp. 1-10. https://arxiv.org/pdf/1706.07772.pdf

[7] Y. Hong and A. Buluç, ”A Sparsity-Aware Distributed-Memory Algorithm for Sparse-Sparse Matrix Multiplication,” SC24: International Conference for High Performance Computing, Networking, Storage and Analysis, Atlanta, GA, USA, 2024, pp. 1-14, https://doi.org/10.1109/SC41406.2024.00053

[8] A. Buluc and J. R. Gilbert, ”Challenges and Advances in Parallel Sparse Matrix-Matrix Multiplication,” 2008 37th International Conference on Parallel Processing, Portland, OR, USA, 2008, pp. 503-510, https://doi.org/10.1109/ICPP.2008.45