Matrix Symmetry and Transpose: parallelizing matrix operations using MPI.

Valerio Cassone ID 236359
University of Trento
Trento, Italy
valerio.cassone@studenti.unitn.it

***Abstract*—This project aims to deeply explore the use of Parallel Programming techniques in the form of Message Passing Interface (MPI) in order to optimize two different problems, the symmetry check and the transpose computation of a matrix, and subsequently benchmark and analyze the performance of the implementations, comparing their efficiency and scalability with respect to the sequential baseline. Moreover, a comparison has been made between the MPI implementation and OMP parallelization techniques.**
The code and the results are available at the GIT repository [1].

I. Introduction

A. Problem identification

In linear algebra, the concepts of matrix symmetry and transposing are strictly correlated. Firstly, given $n \in \mathbb{N}$ and a square matrix M of $n \times n$ elements, M is symmetric if:

$$\forall i, j \in \{1, \dots, n\} \times \{1, \dots, n\},\ [M]_{ij} = [M]_{ji} \qquad (1)$$

Secondly, the transpose of a matrix is an operator which flips a matrix over its main diagonal: given $n, m \in \mathbb{N}$ and a matrix M of $n \times m$ elements, the transpose of M, denoted as M^T is a $m \times n$ matrix constructed by writing the rows of M as the columns of M^T or vice versa. In this way it is obtained that:

$$\forall i, j \in \{1, \dots, n\} \times \{1, \dots, m\},\ [M]_{ij} = [M^T]_{ji} \qquad (2)$$

By using the transpose operator definition, it is possible to state that a square matrix is symmetric if it is equal to its transpose, that is $M = M^T$.

The transpose operator is widely used in linear algebra for different purposes and computations, i.e. the inverse of an orthogonal matrix, and therefore it finds numerous applications in scientific fields.

B. Project objectives

The main objective of this project is creating a C program that sequentially computes the symmetry check and the transpose of a given square matrix M of floating-point numbers. Immediately after this, the proposed solution will be tested on a square matrix of $n \times n$ elements (where n is a power of two varying from 2^4 to 2^{12}) and the execution times of the two routines will be measured (in terms of wall-clock time). Thereafter there will be proposed optimizations of the sequential written code using Message Passing Interface (MPI) to solve the problem with the use of different processes, in order to measure the new execution times, benchmark and analyze the approaches and compare their efficiency and scalability with respect to the sequential implementation in terms of speedup, efficiency and bandwidth.

II. State of the art

Matrix symmetry and transposition are fundamental operations in computational science and engineering, with applications spanning numerical linear algebra, data analysis, and scientific simulations. As data sizes grow beyond the capacity of single-node architectures, distributed memory systems have become crucial. Leveraging tools like the Message Passing Interface (MPI), researchers have developed scalable and efficient approaches to handle these operations. In the recent years there have been notable efforts to enhance performance and scalability in specific contexts.

One of the modern matrix-related problems is dealing with sparse data structures and in 2016 H. Wang, W. Liu, K. Hou and W. Feng proposed the algorithms **`ScanTrans`** and **`MergeTrans`** to significantly increase speedup of matrix transposition on parallel CPUs [2].

In 2018 Eckstein and Matyasfalvi published a work focusing on matrix-vector multiplication. Their methodology includes efficient handling of matrix transposition in distributed-memory systems using MPI, achieving a good balance of work per processor and maintaining low communication costs, which is crucial for large-scale computations [3].

In 2020 by Magalhães and Schürmann addressed the challenges of transposing large-scale multigraphs and sparse matrices in distributed environments. Their approach involves an MPI-based implementation that demonstrates ideal scaling properties, even with heterogeneous datasets. This work is particularly relevant for applications dealing with complex graph models and high-cardinality data. The key operation is a series of collective communication functions which facilitate efficient data transfer between processes [4].

In summary, while the core principles of implementing matrix symmetry and transposition using MPI in distributed memory systems are well-established, recent studies have focused on optimizing these operations for specific data structures and application requirements. These advancements contribute to improved performance and scalability in parallel computing environments.

III. Methodology

A. Sequential Implementation

The solutions of the two proposed problems have been implemented in a C program. The sequential implementations follow the same logic and structure of the previous project "Matrix Symmetry and Transpose: exploring Implicit and Explicit Parallelism with OpenMP" [5].

B. Message Passing Interface

After implemented and tested the sequential solutions of the problems, it is possible to start searching for new approaches to reduce the execution times via distributing the workload among multiple processes using MPI. Message Passing Interface (MPI) is a standard messaging interface that allows communication between processes [6]. It provides a set of syntax and library routines to efficiently share and collect data among them.

The first step in order to setup the MPI environment is by calling **`MPI_Init(&argc, &argv)`** to subsequently obtain the values of **`size`** and **`rank`**. The first is the number of different processes involved in the execution of the C program, while the latter is a progressive number associated with each one. Among the processes, the one with rank 0 will be logically considered as the root process (the one which calls the other) and for this reason it is the only one which knows the initialized matrix, will spread the data to the other and retrieve the results.

Once the environment has been set correctly, all the processes will call the parallelized functions **`checkSymMPI`** and **`matTransposeMPI`** to solve the two problems.

The parallel symmetry check function has been implemented by firstly sharing the data to all processes. This can be done by calling **`MPI_Bcast(...)`** that, starting from the root process (rank 0), will send a copy of the whole matrix to each other process. The workload will then be divided as it follows: the matrix will be equally divided in **`chunk`** rows based on **`n`** and **`size`**; subsequently each process will execute the symmetry check on its rows and save the result in **`local_check`**. The last step will be collecting all the partial results and reduce them in a single **`check`** by calling **`MPI_Reduce(...)`** with the operation **`MPI_LAND`** that executes a logical AND in the reduction.

The matrix transposition can be done in a similar way. As with the symmetry check, the root process has to share the data and this time all the processes will transpose their **`chunk`** rows in a **`temp`** matrix of **`chunk`** $\times$ **`n`** dimensions. At the end the root process has to collect all the column matrices together in the final **`T`** matrix by calling **`MPI_Gather(...)`**. This function collects some data from each process and contiguously stores it in a destination buffer. For this reason, this has to be called for each row such that every process will send their row part of the final transposed matrix.

The first observation about this implementation is that all the processes receive the whole matrix while using only a part of it. For this reason an optimization that can be made is using **`MPI_Scatter(...)`** to equally divide the matrix among the processes. In this way every process will receive only the data it will need for its computation.

Moreover, the communication in the MPI standard is type-based, so up to this point every message passed containing **`double`** values needed to be declared as **`MPI_DOUBLE`** type in the functions. In fact, the MPI standard defines a series of basic built-in datatypes to exchange simple data. However, based on these it is possible to define new custom datatypes with a defined life cycle:

1) type declaration with **`MPI_Datatype dt`**;
2) type creation with different templates;
3) type registration with **`MPI_Type_commit(&dt)`**;
4) type use in the different functions;
5) type deallocation with **`MPI_Type_free(&dt)`**.

By using custom datatypes it is possible to define a type that will represents the matrix rows (using **`MPI_Type_contiguous(...)`** for contiguous data) or even the matrix columns (using **`MPI_Type_vector(...)`** for strided data). In this way not only the written code will become more interpretable, but operations like scattering the matrix by columns, which was previously obtained by scattering row by row among the processes, can be done with a single scatter too. Custom datatypes, however, are not always the best choice in terms of performance depending on the distribution of data (further considerations in the results).

C. Block Transposition

Instead of scattering and working on single rows or columns of the matrix, it is possible to try a different workload division by dividing the matrix in blocks in order to improve the performance of the matrix transposition especially when it comes to bigger matrices and memory and cache accesses. Starting from the sequential implementation extracted from the previous project "Matrix Symmetry and Transpose: exploring Implicit and Explicit Parallelism with OpenMP" [5], the main idea is to distribute some blocks of the matrix **`M`** among the processes, then the processes transpose the received blocks and finally the root process collects all the blocks in the matrix **`T`**. In order to divide **`M`**, a custom datatype **`block_type`** has been created and resized in order to represent a fixed size block of the matrix. In this way it is possible to efficiently use **`MPI_Scatter(...)`** to distribute **`M`** among the processes by assigning, for each row of blocks, an equal number of blocks for each process. After that, all the processes proceed in transposing each received block. At the end the root process can collect the results using **`MPI_Gather(...)`** in order to fill, column by column this time, the transposed matrix **`T`** with all the blocks.

IV. Experiments

The proposed solutions in the last section of this document have been collected in a GIT repository [1] and tested on the following configuration: Intel(R) Xeon(R) Gold 6252N CPU @ 2.30GHz, with 1024 MB of dedicated RAM and 16 selected cores and 16 available MPI processes. The compiler

used is **`gcc-9.1.0`** with **`mpich-3.2.1`** to compile the MPI programs with the 3.1 MPI standard. The experiments have been conducted following and combining the methodologies described in the previous section. In every simulation, the input $n \times n$ square matrix **`M`** has been initialized with random **`double`** values and then processed by the symmetry check and matrix transposer functions. The dimension n is a power of 2 varying from 2^4 to 2^{12}. The tested solutions, among with their identifier are the following:

Code	Description
S	sequential execution
SB	sequential block transposition
M	**S** with MPI parallel execution
MD	**M** with equally divided matrix
MC	**MD** with custom datatypes
MB	**SB** with MPI parallel execution

These simulations aim to test the proposed solutions in different combinations of methodologies, in order to study how they behave when increasing the input data in terms of FLOPS for the symmetry check and effective bandwidth for the matrix transpose. In addition to this, speedup, efficiency and scalability are compared for the MPI approaches.

V. Results

A. FLOPS and Bandwidth

Starting from the execution times obtained from the simulations it is possible to study the behavior of the proposed solutions in terms of FLOPS (GFLOPS) for the symmetry check and bandwidth (GB/s) for the transposition. In order to fully analyze the impact of the message passing between processes, two execution times have been recorded for each routine: one that includes the message passing, therefore measuring the execution time of the whole **`checkSymMPI`** and **`matTransposeMPI`** functions, and one considering only the computation part of the two functions, as it happens for the sequential implementation. To distinguish them, the first are referred with "message" times and the latter with "effective".

To facilitate the interpretation this report only shows the case $n = 12$, even though all the results can be visualized from the GIT repository. The results of the simulation have been organized in two different plots showing the FLOPS and the bandwidth over the number of processes (Fig 1, Fig 2). The red horizontal line shows the FLOPS and the bandwidth of the sequential solution, while, for each implementation, the continuous line shows the "message" metrics and the dashed line the "effective" ones. The main thing that stands out from both graphs is the impact of the message passing overhead. It is evident that dividing the workload significantly reduces the effective computation times, but when considering the overhead of distributing and collecting input and output data, the performance become fair less optimal, resulting in a slowdown rather than an acceleration of the overall computations. The same reasoning explains why the block transposition recorded the best effective bandwidth, but at the same time the worst real one.

Fig. 1. Symmetry Check: FLOPS (GFLOPS) for $n = 12$

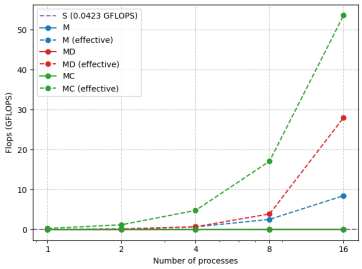
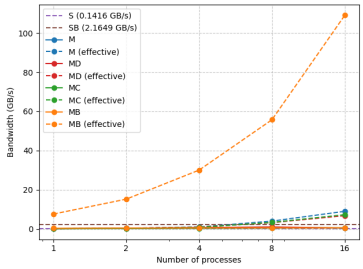


Fig. 2. Transpose: Bandwidth (GB/s) for $n = 12$



B. Speedup, Efficiency and Scalability

Going on with the analysis, it is possible to study the speedup, efficiency and scalability gains of the parallel solutions. Starting with speedup and efficiency, the results for the case $n = 12$ are plotted on the graphs Fig 3, Fig 4. The first piece of information that can be immediately read is the main difference between the two operations. In fact the symmetry check recorded really low efficiencies. The main reason behind this result is the fact that the required computation is relatively simple and the overhead of exchanging messages with matrix data is slowing down the whole process instead of accelerating it. On the other hand, the required computation for the transposition has been revealed to be worth the cost of message passing. Moreover, as mentioned before in the methodology, the use of custom datatypes didn't brought an increase of the efficiency. This is due to the fact that collecting entire columns with the use of vector types after the transposition resulted in a slowdown in the process of writing data in the memory, while gathering values row by row (exchanging n messages instead of a single one) reduced the root process' workload in terms of RAM and cache accessing. For the same reason, the block transposition recorded an almost null speedup and efficiency, even though it still remains the fastest solution in terms of effective computation times (excluding the message passing).

Regarding the scalability of the proposed solutions, the obtained results have been plotted on the graph Fig 5, starting with $n = 8$ and $P = 1$ and then increasing the problem dimensions along with the number of processes. It is evident that the recorded values for the weak scaling are really far from the ideal ones. However, this is due for the same reasons explained before for the strong scaling and the efficiency. Furthermore, the symmetry check routines still recorded worse values than the transposition ones.

TABLE I
SPEEDUP AND EFFICIENCY OF OMP AND MPI SOLUTIONS FOR $n = 12$

ID	O	OR	OBT	OB_S	OB_D	M	MD	MC	MB	O	OR	OBT	OB_S	OB_D	M	MD	MC	MB
Thrs/Procs	**Speedup (Symmetry Check)**									**Efficiency (Symmetry Check)**								
2	0.6	2.0	2.1	1.51	1.61	1.16	0.94	0.1	N/A	30.0	99.79	104.93	75.58	80.55	58.21	46.8	5.17	N/A
4	0.35	2.12	2.93	3.82	3.58	1.47	1.4	0.11	N/A	8.8	53.12	73.23	95.52	89.51	36.74	34.97	2.75	N/A
8	0.23	2.17	5.9	5.91	7.04	1.3	1.83	0.1	N/A	2.84	27.17	73.8	73.91	87.96	16.22	22.88	1.19	N/A
16	0.1	2.05	11.44	10.5	12.16	1.1	1.96	0.1	N/A	0.62	12.84	71.53	65.63	76.03	6.91	12.28	0.61	N/A
Thrs/Procs	**Speedup (Transposition)**									**Efficiency (Transposition)**								
2	2.02	1.92	2.09	2.14	1.64	2.54	2.05	1.44	0.06	101.04	96.15	104.47	107.14	82.13	126.77	102.4	71.87	2.9
4	4.07	3.36	3.68	4.37	3.4	4.94	4.52	2.13	0.07	101.82	83.89	91.99	109.33	84.92	123.4	113.06	53.31	1.66
8	6.63	4.96	6.46	6.32	5.54	5.97	6.93	3.03	0.07	82.82	61.98	80.79	79.0	69.24	74.6	86.62	37.82	0.89
16	9.98	8.1	8.85	7.72	8.16	2.51	3.42	3.77	0.07	62.36	50.65	55.31	48.23	51.0	15.66	21.37	23.53	0.43

Fig. 3. Speedup vs Number of processes

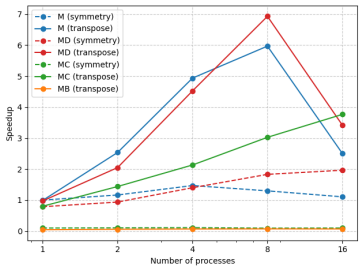


Fig. 4. Efficiency vs Number of processes

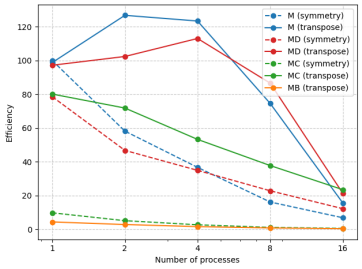
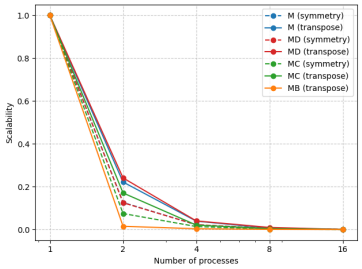


Fig. 5. Scalability vs Number of processes



C. Comparison of Parallelism Approaches

Starting from the results obtained with the simulations, it is possible to operate a qualitative and quantitative comparison between the MPI approach and the OMP solutions of the same problem proposed in "Matrix Symmetry and Transpose: exploring Implicit and Explicit Parallelism with OpenMP." [5]. Starting from the implementation part, it is clearly evident how the OMP approach is much more straightforward than the MPI one because of two main reasons: the first one is that OMP parallelization is easier to achieve because threads share a part of the memory of the program, so the parallelization does not require to explicitly share the data. Moreover, the use of pragmas delegates the workload division in the for loops to the compiler, so the developer doesn't have to explicitly and statically divide the workload among the spawned threads. On the other hand, even though MPI introduces the overhead (in time and lines of code) of sharing data among the processes, it does not impose a theoretical limit for the parallelization. When talking about scalability, in fact, the OMP techniques have a peak of performance that is represented by the maximum number of core available on a single CPU. MPI, however, allows to spread the problem on an arbitrary number of CPUs and different hardwares, as long as there is a connection between them for the message passing. Moreover, spreading the problem on different CPUs could lead also in exploiting both MPI and OMP to further optimize the performances.

When it comes to the results (speedup and efficiencies) collected in Table I, it is evident how the MPI implementations are significantly slower than the OMP ones. Besides the block symmetry check and transposition implementations, the computation part of the two approaches are perfectly identical, so for this reason the cost of the message passing stands out even more by observing the discrepancies in the values for the same number of threads/processes.

VI. CONCLUSIONS

This project successfully explored the application the Message Passing Interface (MPI) to optimize matrix symmetry checks and transposition operations. By comparing sequential implementations with various parallel approaches, significant insights were gained into the trade-offs between computational efficiency and communication overhead.

The experimental results highlighted that while parallelization can lead to substantial speedup in computational tasks, the message-passing overhead remains a critical factor that can diminish overall efficiency, especially for operations with relatively simple computational requirements like symmetry checks. The analysis also revealed that the use of custom MPI datatypes and block-based transpositions could further optimize performance, although these improvements depend heavily on matrix size and data distribution strategies.

Comparing MPI to OpenMP implementations demonstrated the strengths and limitations of each paradigm. OpenMP excels in simplicity and lower overhead for shared-memory systems, whereas MPI offers scalability for distributed environments, albeit with increased complexity and communication costs.

REFERENCES

[1] GIT Repository with the implemented code, reproducibility instructions and results. https://github.com/ValeMX/parco-h2-236359

[2] Hao Wang, Weifeng Liu, Kaixi Hou, and Wu-chun Feng. 2016. Parallel Transposition of Sparse Data Structures. In Proceedings of the 2016 International Conference on Supercomputing (ICS '16). Association for Computing Machinery, New York, NY, USA, Article 33, 1–13. https://doi.org/10.1145/2925426.2926291

[3] Jonathan Eckstein and Gyorgy Matyasfalvi. (2018). Efficient Distributed-Memory Parallel Matrix-Vector Multiplication with Wide or Tall Unstructured Sparse Matrices. 10.48550/arXiv:1812.00904. https://arxiv.org/abs/1812.00904

[4] Magalhaes, Bruno & Schürmann, Felix. (2020). Efficient Distributed Transposition Of Large-Scale Multigraphs And High-Cardinality Sparse Matrices. 10.48550/arXiv.2012.06012. https://arxiv.org/abs/2012.06012

[5] Valerio Cassone. 2024. Matrix Symmetry and Transpose: exploring Implicit and Explicit Parallelism with OpenMP. GIT Repository with paper and source code files. https://github.com/ValeMX/parco-h1-236359

[6] Message Passing Interface - an overview — ScienceDirect Topics