Optimizing Square Matrix Symmetry Verification and Transposition Using OpenMP and MPI

Mosè Arcaro, ID:226616
Department of Engineering and Computer Science
Faculty of Engineering, Università degli studi di Trento
Trento, Italy
mose.arcaro@studenti.unitn.it

***Abstract*—This paper presents a comparative analysis of some implementations of square matrix operations (transposition and symmetry check functions): sequential, parallel using OpenMP, and parallel using MPI. The study explores different implementations and quantifies the performance improvements achieved through parallelization, demonstrating speedups and efficiency gains in both implementations compared to the sequential version. Furthermore, it presents an analysis of OpenMP and MPI across different implementations, highlighting their respective limitations.**

I. Introduction

Matrix operations are fundamental in numerous scientific computing applications, from image processing to numerical simulations. The performance of these operations can significantly impact overall system efficiency. [1]

The primary goal of this project is to compare four distinct implementations of square matrix operations:

- Sequential implementation
- Parallel implementation using OpenMP
- Parallel implementation using MPI
- Parallel implementation using MPI dividing matrix into Blocks

The comparison focuses on execution time to determine the effectiveness of each optimization approach.

II. State Of The Art

Matrix transposition optimization has been largely studied in literature [17] [15] [2] [3]. While extensive research exists on matrix transposition optimization, several gaps remain:

- Lack of systematic comparison between basic, OMP and MPI implementations
- Need for practical guidelines based on matrix size and hardware capabilities
- Study of the impact of thread and process numbers in addition to matrix size
- Detailed comparison of various techniques on distributed systems using MPI

Our work aim to evaluate and find the best combination of techniques in order to optimize the time performance of a matrix transposition and symmetry check

III. Contribution and Methodology

This project addresses these gaps by: Providing systematic performance comparison; Quantifying combined optimization effects and delivering practical implementation insights.

The sequential version consists of two simple for loops that traverse all matrix values (both for the symmetry check function and transpose calculation). We chose to traverse all elements to make the results independent of the matrix content. The parallel version on a single system uses combined OMP techniques [10] and memory access optimizations. The parallel version on distributed systems uses combined MPI techniques [4] [5] for data exchange. All versions were studied with float data type. Matrix dimensions: $n \in 2^4, 2^5, ..., 2^{12}$. Maximum number of processors: 96; Maximum number of threads: 96; The study was conducted on a single node to avoid network latency issues (which would be limited to 10 G/s), thus improving data locality and performance for moderately-sized matrices. The study was conducted on a system with 64-byte aligned memory [6]. An iterative approach was taken, testing different implementations and comparing results to identify the best version.

IV. Experiments and System Description

A. Hardware

UniTn Cluster [21] with: Ram: 32k DDR SRAM; [13] Processors: Intel® Xeon® Gold 6252N Processor 35.75M Cache, 2.30 GHz; [12] Number of cores: 96; Name of the processor: hpc-c11-node07.unitn.it

B. Software

Compiler: GCC 9.1; Libreries: stdio.h, stdlib.h, time.h, immintrin.h, sys/time.h, omp.h, math.h, getTimeOfTheDay [14]; OpenMP version 3.1; File PBS; File CSV; Python 3.7.2 with pandas, matplotlib, pyplot, sys, os; Mpich version 3.2.1

C. Sequential Version

The pragma option override(checkSym, "opt(level, 0)") was used to remove all possible compiler optimizations. The implementation consists of simple nested loops that traverse the entire matrix. Fig. 1 shows its time performace (SEQ lines)

D. OpenMP

It consists of using blocking and pragma to optimize and improve data access in memory. In Fig. 1 we can see its time performace (OMP lines)

E. MPI

A version has been implemented that shares the entire matrix with all processes; each process transposes its own rows and then sends them back to process 0 to create the transposed matrix. Simple `MPI_Recv` and `MPI_Send` were used. (MPI base) A version that uses Alltoall and Gather for transposition, which requires some checks because it doesn't work with all dimensions, as the use of Alltoall requires each process to work with exactly one row and no more. (MPI AlltoAll) [7] A final version using MPI datatypes, which improve performance. (MPI DT) [20] [8]

Processors	Dim	MPI base	MPI AlltoAll	MPI DT
2	16	0.0000196	-	0.000106
16	16	0.0003756	0.0002184	0.0006662
32	32	0.0004772	0.0003604	0.0004302
64	64	0.0009314	0.0009056	0.000746
26	128	0.0005418	-	0.0004648
53	256	0.0018184	-	0.0017326
87	512	0.0179904	-	0.0178608
44	1024	0.0096436	-	0.0087542
72	2048	0.037802	-	0.0371126
95	4096	0.2196158	-	0.2102584

TABLE I
COMPARISON TIME[S] FOR CHECK SYMMETRY TIME FOR VERSION AND PROCESSES

Processors	Dim	MPI base	MPI AlltoAll	MPI DT
2	16	0.0005334	-	0.000106
16	16	0.0003886	0.000176	0.0002614
32	32	0.0012018	0.0003638	0.0023944
64	64	0.0007288	0.0007936	0.0007432
24	128	0.0010518	-	0.0003068
55	256	0.0030182	-	0.0015578
81	512	0.0130306	-	0.010048
43	1024	0.03303	-	0.012872
78	2048	0.1464398	-	0.0546082
92	4096	0.7141268	-	0.214745

TABLE II
COMPARISON TIME[S] FOR TRANSPOSITION TIME FOR VERSION AND PROCESSES

Note how the DT version performs worse in cases of well-distributed work among processes (first rows), but performs better when the work is not evenly distributed. This is because MPI datatypes [8] provide a more efficient way to handle communications with irregular and non-contiguous workloads, reducing overhead and allowing optimization at the implementation level. Furthermore, notice how with DT, when the workload increases, the performance improves significantly. Finally, a last version was implemented by dividing the matrix into blocks [16], sharing and transposing only those blocks. Multiple experiments were conducted with this version: base version (MPI BB) using simple send and receive; using datatypes (MPI BOD); adding scatter, gather and collective communicators (MPI BNVD) [18]; using "variable" versions of collective communicators (MPI BDV) [19]. This last implementation only works with specific dimensions and number of processors, since dividing the matrix into blocks and sharing it requires the matrix to be exactly divisible by the number of processors.

Processes	Dim	MPI BB	MPI BDV	MPI BOD	MPI BNVD
4	16	0.0005008	0.000597	0.000351	0.0003992
16	16	0.000614	0.0006654	0.0003196	0.000313
16	32	0.000788	0.0009246	0.0006452	0.0002608
64	64	0.002114	0.0013482	0.0013552	0.0011696
16	128	0.0016578	0.0008668	0.0003506	0.0007334
4	256	0.0015676	0.0006844	0.0007632	0.0011232
64	512	0.0117464	0.0028946	0.0024816	0.0053942
64	1024	0.0274762	0.0085702	0.0114406	0.017472
16	2048	0.0844384	0.02403	0.0242252	0.060321
64	4096	1.2823244	0.08869	0.0954242	0.2753908

TABLE III
COMPARISON TIME[S] BETWEEN DIFFERENT BLOCK DIVIDING IMPLEMENTATIONS

As can be noted, for small-sized matrices BOD performs better, while when the workload increases BDV becomes better.

V. RESULTS AND DISCUSSION

The best versions were compared to analyze their execution times Fig.1 and Fig.2 .

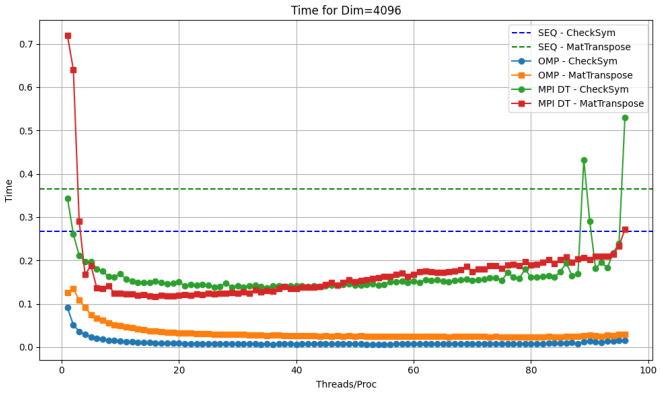

Fig. 1. time[s] performances for Dim = 4096

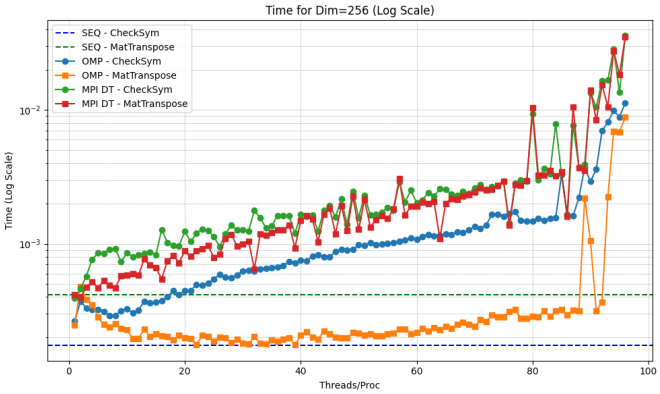

Fig. 2. time[s] performances for Dim = 256 (log scale)

As we can notice in Fig 2, the best MPI version, turns out to be very slow, even worse than the sequential version (this is true up to the matrix dimension of 2048); this is no longer true in the version that divides the matrix into blocks.

Let's now analyze speedup (1), efficiency (2), and scaling (3) for a better understanding of the results.

$$\frac{\text{Time On Sequential Version}}{\text{Time On N processors}} = \text{speedup} \tag{1}$$

$$\left(\frac{\text{speedup}}{\text{number Of Processors}}\right) \cdot 100 = \text{efficiency} \tag{2}$$

$$\left(\frac{\text{Single processor time for size: } (N)}{\text{P processors time for size: } (p \cdot N)}\right) = \text{scalability} \tag{3}$$

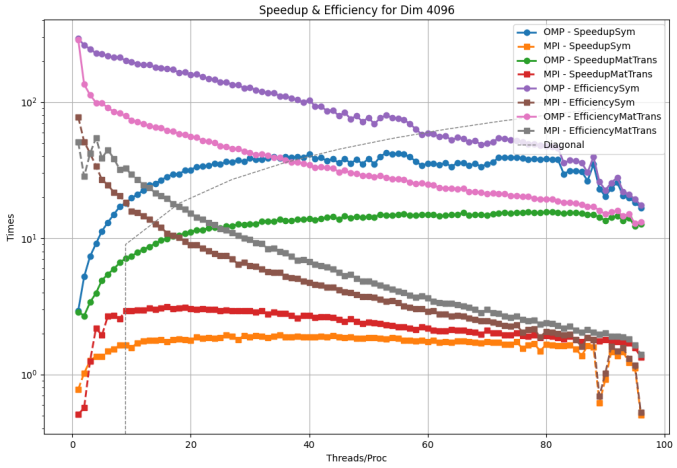

Fig. 3. efficiency and speedup for Dim = 4096 (log scale)

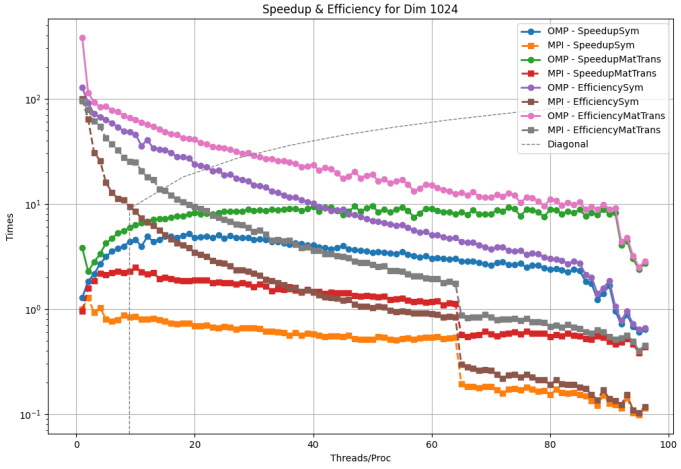

Fig. 4. speedup and efficiency for Dim = 1024 (log scale)

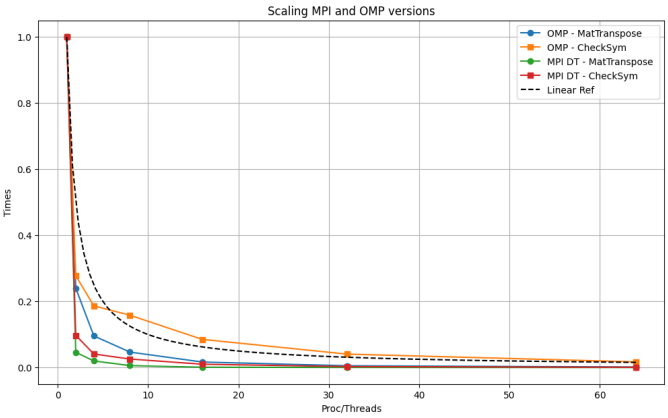

Fig. 5. scaling

As can be seen in Fig. 3 and Fig. 4 , the version using OpenMP performs better due to the lack of need to share information. Despite the elimination of network latency when using a single node, due to inefficiencies related to internal management and sharing the entire matrix with all processes, the MPI version achieves significantly lower results compared to the OpenMP version. Note how MPI however scales better Fig. 5.

A. *Block vs No-Block*

A comparison was then made between the best version that shares the entire matrix (MPI DT) and the best one that shares and works only on a single block (MPI BDV) and, as expected, the performance improves. As expected the performances improves and the efficeincy remains better respect to Non-Blocking version seen in Fig. 6 and Fig.7 Demonstrating that sharing the entire matrix with all processes represents an unnecessary bottleneck that worsens performance.

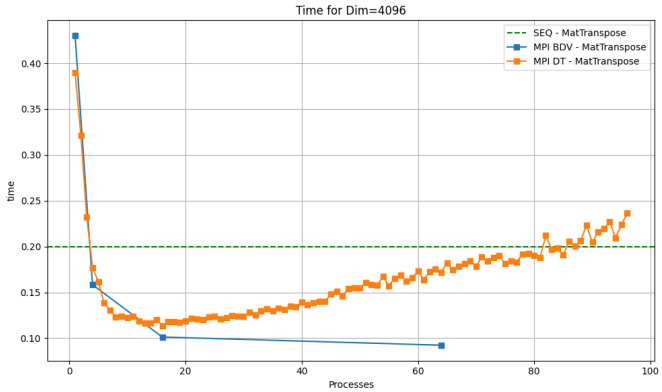

Fig. 6. time performances for Dim = 4096 (log scale)

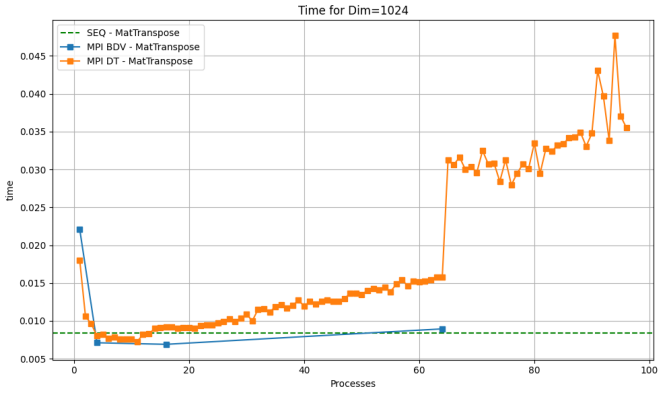

Fig. 7. time performances for Dim = 1024 (log scale)

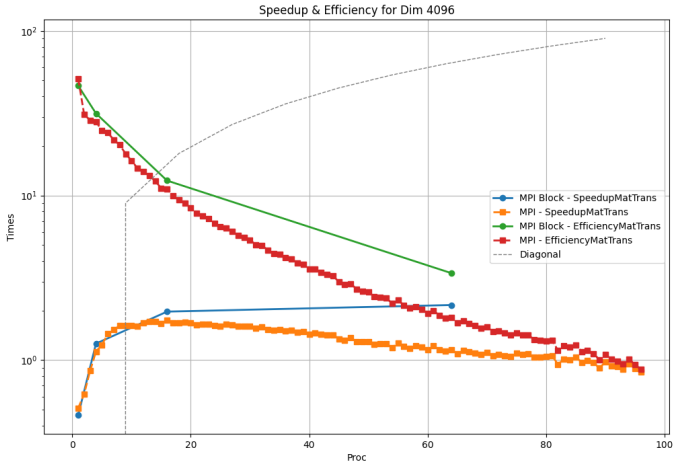

Fig. 8. speedup and efficiency for Dim = 4096 (log scale)

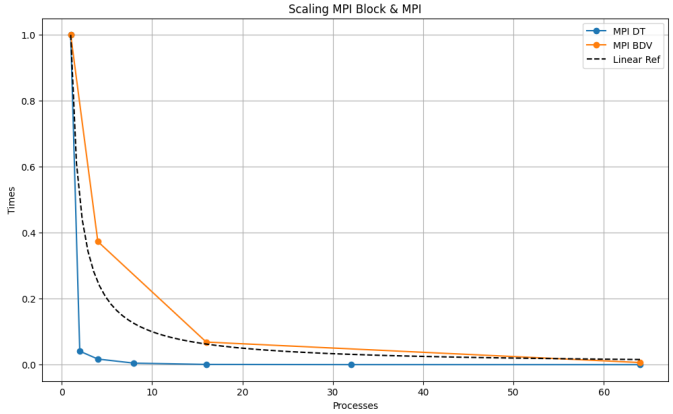

Fig. 9. scaling

However, as seen in Fig. 8 and Fig. 9 the speedups still deviate significantly from the ideal, and the version that divides the matrix into blocks scales slightly worse.

B. Problems Encountered

The synchronization of communication between processes required careful coordination, and debugging proved complex in a distributed environment. It was necessary to implement checks on dimensions and number of processes, given the particular functioning of some implementations that only work if the rows or matrix dimensions are divisible by the number of processes. Initially, we tried to abort superfluous processes, for example, by dividing processes into sets and subsets with global communicators [11], which proved to be a poor choice due to its implementation difficulty; therefore, we opted to abort the entire program. It was also necessary to divide the workload as uniformly as possible among processes while avoiding sharing the entire matrix with all processes.

VI. Conclusions

We have demonstrated various versions for performing matrix transposition operations and symmetry checking, showing different optimization methods through both OpenMP and MPI parallelization. We have shown how a multithreaded approach brings undisputable benefits both when working in monolithic environments with OpenMP and when using MPI, which brings significant benefits in distributed systems or when optimal scaling is needed. The two methods cannot be directly compared given their strengths on completely different physical systems, and our data shows that, using a single node, OpenMP is undoubtedly more efficient. Our experiments also confirm that avoiding sharing the entire matrix is highly recommended as it introduces a significant bottleneck. Furthermore, the choice to divide the matrix into blocks, share it, and then receive back the transposed blocks proved better in terms of performance but worse in terms of scalability. Finally, for very small matrices, a sequential approach is preferable.

VII. Further Works

An interesting additional analysis would be to compare the block versions with OpenMP and using a distributed hardware environment to show the true potential of MPI.

Git and Instructions for Reproducibility

You can find all complete data and codes here, with the instructions for reproducibility https://github.com/mosearc/ParallelProgramming.git

References

[1] https://en.wikipedia.org/wiki/Transpose
[2] https://www.researchgate.net/publication/221530749_On_the_Limits_of_Cache-Oblivious_Matrix_Transposition
[3] https://www.sciencedirect.com/science/article/abs/pii/016781919500016H https://brunomaga.github.io/Matrix-Transpose
[4] https://www.open-mpi.org/doc/v4.0/
[5] https://mpitutorial.com/
[6] https://stackoverflow.com/questions/794632/programmatically-get-the-cache-line-size
[7] https://www.researchgate.net/figure/Transposing-the-matrix-using-all-to-all-communica fig1_220073913
[8] https://stackoverflow.com/questions/19058839/whats-the-benefit-of-mpi-datatype
[9] https://it.wikipedia.org/wiki/Legge_di_Amdahl
[10] https://www.openmp.org/spec-html/5.0/openmpsu42.html#x65-1390002.9.3 https://www.openmp.org/spec-html/5.0/openmp.html
[11] https://stackoverflow.com/questions/13774968/mpi-kill-unwanted-processes

[12] https://ark.intel.com/content/www/us/en/ark/products/193951/intel-xeon-gold-6252n-processor-35-75m-cache-2-30-ghz.html https://docs.google.com/spreadsheets/d/1-n4q-nFSHiIugUh_PSmsFS1VwP_ZUitkCSyDvz2As0E/edit?gid=1254941127#gid=1254941127 https://www.cpu-world.com/CPUs/Xeon/Intel-Xeon%206252N.htm
[13] https://en.wikipedia.org/wiki/DDR_SDRAM
[14] https://www.delftstack.com/howto/c/gettimeofday-in-c/
[15] https://www.intel.com/content/www/us/en/docs/advisor/cookbook/2023-0/optimize-memory-access-patterns.html
[16] https://www.researchgate.net/figure/Dividing-a-large-data-matrix-into-submatrices-of-manageable-sizes_fig16_6751767 https://stackoverflow.com/questions/16134620/block-matrix-transpose https://www.researchgate.net/figure/Block-wise-matrix-transposition-The-matrix-is-optionally-distributed-over-several_fig3_284726052
[17] https://ieeexplore.ieee.org/abstract/document/995452 https://dl.acm.org/doi/abs/10.1145/3673038.3673159 https://ieeexplore.ieee.org/abstract/document/1639721
[18] https://www.sciencedirect.com/topics/computer-science/collective-communication
[19] https://www.mpi-forum.org/docs/mpi-1.1/mpi-11-html/node70.html https://www.mpi-forum.org/docs/mpi-1.1/mpi-11-html/node70.html https://www.mpi-forum.org/docs/mpi-1.1/mpi-11-html/node72.html
[20] https://enccs.github.io/intermediate-mpi/derived-datatypes-pt2/ https://arxiv.org/abs/1809.10778
[21] https://docs.google.com/spreadsheets/d/1-n4q-nFSHiIugUh_PSmsFS1VwP_ZUitkCSyDvz2As0E/edit?gid=1254941127#gid=1254941127