Matrix Transposition with MPI (H2)

Bilal Soussane (ID: 235252)
Email: bilal.soussane@unitn.it
Repo: https://github.com/sousbila/matrix-transposition-mpi.git

***Abstract*—Matrix transposition is a crucial kernel in many high-performance computing (HPC) applications. In this work, we compare several MPI-based implementations for performing matrix transposition on square matrices (with n being a power of two), as well as serial block-based approaches. In the MPI domain, we developed several variants: mpi, a baseline that broadcasts the entire matrix, mpi2 (which uses column-wise gathers), and mpi3—the fastest among the non-block-based methods—which scatters rows and then employs pairwise sub-block exchanges to reassemble the required data before a single final gather. In addition, we discuss block-based MPI approaches (mpiblock1, and mpiblock3), with the latter showing the best performance in that category. The symmetry-check function (`checkSymBlockMPI`) uniformly uses a full broadcast to verify matrix symmetry before transposition. Our experiments analyze these methods in terms of execution time, scalability, and efficiency.**

I. Introduction and Motivation

Matrix transposition, defined as $T(i, j) = M(j, i)$ for an $n \times n$ matrix M, is a fundamental operation in applications ranging from scientific simulations to machine learning. Although the operation itself is simple, achieving efficient transposition on large matrices requires careful optimization of data movement, inter-process communication, and cache utilization.

High-performance computing (HPC) on large clusters demands fine-grained control over data distribution and communication—challenges that the Message Passing Interface (MPI) is designed to address. Blocking strategies, which partition the matrix into sub-blocks, further improve performance by enhancing data locality. In this work, we investigate and compare several MPI-based and block-based implementations. We also consider an OpenMP implementation to provide a comprehensive view of parallelization trade-offs.

The motivation for our study is twofold: (1) to overcome the performance gap between fast processor speeds and slower memory bandwidth by reducing communication overhead and improving cache reuse, and (2) to identify the optimal parallelization strategies for matrix transposition as HPC systems continue to scale in core count and complexity.

A. Contributions

- A discussion of the symmetry-check function (`checkSymBlockMPI`) that uses a full broadcast of the matrix to confirm symmetry before transposition.
- A technical analysis of three MPI-based transposition approaches: mpi (a baseline that broadcasts the full matrix), mpi2 (using column-by-column gathers), and mpi3 (which uses a row-scatter followed by pairwise exchanges).
- A detailed description of two block-based MPI transposition methods (mpiblock1, and mpiblock3).
- A comparative analysis of the serial block-based implementation (`matTransposeBlock`) with its MPI counterpart (`matTransposeBlockMPI3`). Additionally, we compare the regular MPI implementations with their respective single-process (serial) execution and with the OpenMP-based parallelization.

II. State-of-the-Art

Recent research has extensively explored optimizations for matrix transposition on distributed-memory systems. Lee *et al.* [1] demonstrated that tiling strategies can significantly reduce cache misses by partitioning large matrices into smaller sub-blocks. Kim and Wang [2] further investigated scalable data redistribution techniques, highlighting the benefits of pairwise communication to minimize latency when exchanging sub-blocks. Yang *et al.* [3] proposed communication-aware algorithms that adaptively balance computation and communication overhead. Park *et al.* [4] provided experimental evidence that cache-optimized approaches improve transposition performance on multicore processors, while Li *et al.* [5] combined MPI and OpenMP paradigms to achieve hybrid parallelism for dense matrix transposition. More recently, Wang *et al.* [6] and Zhang *et al.* [8] studied the impact of MPI collective operations on matrix transposition, emphasizing the trade-offs between broadcast-based and scatter/gather approaches. These studies collectively point to the importance of both communication minimization and cache-awareness; our work builds on these insights by comparing non-block and block-based approaches.

III. Methodology

This section details the implementation of our matrix transposition approaches and highlighting their data distribution and communication patterns.

A. Symmetry Check: `checkSymBlockMPI`

Before transposition, we check whether the matrix M is symmetric (i.e., $M = M^T$) using a broadcast-based function. In this function:

1) Rank 0 holds the complete $n \times n$ matrix and broadcasts it via `MPI_Bcast` to all ranks.
2) Each process independently verifies the symmetry of its assigned rows by checking if $M[i, j] = M[j, i]$.

3) An `MPI_Allreduce` combines these local flags into a global result.

This approach, though costly due to the full broadcast, is used uniformly across all MPI implementations so that comparisons remain consistent. Therefore, we do not consider the execution time of the `checkSym` function in our performance analysis.

B. OpenMP Block-Based Implementation

We implement an OpenMP-based block transposition to exploit shared-memory parallelism and enhance cache locality. The matrix is partitioned into squared sub-blocks (e.g., 64×64). The outer iterating over these blocks are parallelized using directives such as `#pragma omp parallel for default(none) shared(matrix, transposed, n, blockSize)`. Additionally, inner loops are manually unrolled (in increments of four) to improve instruction-level parallelism. For further technical details, please refer the relevant section in our H1 report.

C. MPI Transposition Variants

We implemented several MPI-based transposition methods that primarily differ in data distribution and communication:

- **MPI Serial Baseline:** To ensure a fair comparison between serial and parallelized code, the serial baseline for non–block-based MPI is obtained by running the same MPI implementation with one processor (`i.e., mpirun -np 1 ...`). This approach employs the same code path as the parallel versions and thereby isolates the communication and synchronization overhead introduced when running with multiple processors.
- **mpi:** Rank 0 broadcasts the entire matrix M to all processes, each process transposes its assigned contiguous row block, and then `MPI_Gather` collects the results on rank 0.
- **mpi2:** Rank 0 scatters contiguous row blocks of M to processes. Then each process loops through each column of its local block, gathering partial column data (via multiple `MPI_Gather` calls) to reconstruct the full columns for its portion of the transposed matrix. This approach incurs high overhead due to many gather operations.
- **mpi3 (Best-Performing MPI Variant):** Rank 0 scatters row blocks of M so each process receives a submatrix of size $\frac{n}{p} \times n$. Each process subdivides its rows into p contiguous chunks (each of width n/p) and exchanges these chunks with partner processes using pairwise communication. A single final `MPI_Gather` then collects the locally transposed blocks on rank 0.

D. Serial Block-Based Implementation

Our serial baseline for the block-based (mpiblock) implementations employs a block-based matrix transposition and symmetry check. In this approach, the matrix is partitioned into squared sub-blocks. The function `matTransposeBlock` then uses nested loops over these sub-block indices to perform an out-of-place transposition for each block, while `checkSymBlock` partitions the matrix similarly and verifies symmetry by checking only the appropriate upper-triangular sections. This method improves cache utilization compared to a naive transposition and serves as our standard for evaluating the block-based MPI implementations.

E. MPI Block-Based Implementations

For our MPI block-based methods, rank 0 divides the full $n \times n$ matrix M into a grid of sub-blocks arranged in a $\sqrt{p} \times \sqrt{p}$ layout, where p is the number of processes. Consequently, each block has dimensions $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$. Rank 0 then distributes these sub-blocks to the corresponding processes in a 2D grid. Each process locally transposes its block, and then rank 0 gathers the transposed sub-blocks to form the transposed matrix T.

Block-based approaches inherently improve cache locality since operating on a small contiguous submatrix reduces cache misses compared to processing entire rows or columns.

For our Intel Xeon Gold 6252N CPU (with 32 KB L1 caches and 1 MB L2 caches per core) a block of floats (each 4 bytes) requires approximately $2 \times 4 \times \texttt{blockSize}^2$ bytes of cache space (the factor of 2 accounts for both source, reading from M, and destination data, writing to T).

For instance, a 64×64 block would use around $8 \times 64^2 = 32768$ bytes, which is an ideal fit for the L1 cache (32 KB). Even when the blocks are larger (as they will be for very large matrices), they usually fit into L2 cache.

Our experimental results demonstrate that, despite some blocks being larger than the L1 ideal, the improved spatial locality inherent in square block partitioning, combined with careful block alignment and optimized memory transfers, significantly reduces the number of cache misses and overall memory latency

We developed two block-based implementations, which we summarize as follows:

- **`mpiblock_code1`:** A straightforward implementation where rank 0 extracts each sub-block (of size $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$) via a double loop and sends it to the respective process. Each process transposes its block, and rank 0 gathers the blocks. This naive approach does not optimize data transfers beyond basic point-to-point communication.
- **`mpiblock_code3`:** An optimized variant that builds upon the basic strategy of `mpiblock_code1` by precisely aligning sub-block boundaries and streamlining the send/receive loops to reduce redundant data copying and communication overhead. Our experimental results have consistently shown that `mpiblock_code3` outperforms `mpiblock_code1` and is thus selected for further evaluation.

IV. Experimental Setup and Evaluation

A. Computing Environment

Our experiments were conducted on a Linux-based HPC cluster equipped with an Intel® Xeon® Gold 6252N CPU running at 2.30 GHz. The system consists of four sockets, each hosting 24 cores (for a total of 96 cores). Each core features a 32 KB L1 cache and a 1 MB L2 cache, while each socket shares a 36 MB L3 cache. The system is organized as

four NUMA nodes. We compiled our code using GCC version 9.1.0, which provides full support for OpenMP. MobaXterm was used to establish SSH connections from our local Windows system to the cluster.

B. Experimental Setup

We evaluated our implementations on square matrices ranging in size from 2^4 (16×16) up to 2^{12} (4096×4096). For a fair comparison and balanced workload, the following configurations were used:

- **MPI Block-Based Implementations:** These require the number of processes to be a perfect square. We ran these tests using 4, 16, and 64 processes.
- **Regular MPI (Row-Based) Implementations:** To ensure balanced workload distribution, the number of processors was chosen to evenly divide the matrix size. Tests were conducted with 1, 2, 4, 8, 16, 32, and 64 processors.

C. *Best performance*: `mpi` vs. `mpi2` vs. `mpi3`

Table I shows execution times (in seconds) for our three non–block-based MPI implementations on selected matrix sizes and process counts. We observe that for smaller matrices (up to 128×128), `mpi` often yields slightly faster times due to lower overhead, especially at 4 or 16 processes; here, the cost of pairwise sub-block exchanges in `mpi3` does not pay off. However, once the matrix size exceeds 128×128, `mpi3` begins to outperform both `mpi` and `mpi2`, demonstrating significantly lower execution times at moderate and high process counts. Consequently, while `mpi` can be preferable for very small matrices, we adopt `mpi3` as our recommended non–block-based MPI method for larger matrices, thanks to its superior scalability.

TABLE I
EXECUTION TIMES (S) FOR `MPI`, `MPI2`, AND `MPI3`.

Matrix Sizes	Procs	mpi	mpi2	mpi3
128	4	0.000164	0.000861	0.000209
128	16	0.000268	0.001434	0.000322
128	64	0.000527	0.002403	0.001659
512	4	0.001836	0.002816	0.002103
512	16	0.002290	0.005562	0.001371
512	64	0.008277	0.012667	0.002954
2048	4	0.036860	0.037858	0.057585
2048	16	0.042481	0.032270	0.030332
2048	64	0.055789	0.090596	0.028859
4096	4	0.174231	0.167208	0.260231
4096	16	0.191072	0.233418	0.118394
4096	64	0.232444	0.403533	0.129683

D. Optimal Process Counts for `mpi3`

Our analysis indicates that 4 processes are best for matrices up to 64×64, 4–8 for 128×128 and 256×256, 16 for 512×512, and 32 for $n \geq 1024$ (with a marginal edge for 16 at 4096×4096). We thus adopt 4, 8, 16, and 32 processes for these respective ranges to run our `mpi3` experiments.

E. Block-Based MPI Approaches

Table II shows execution times for two block-based MPI implementations. The simple approach, `mpiblock_code1`, suffers from higher communication overhead due to basic point-to-point transfers. In contrast, the optimized variant, `mpiblock_code3`, achieves better scalability by aligning sub-block boundaries and streamlining send/receive loops to reduce redundant data copying. Our experimental results consistently show that `mpiblock_code3` outperforms `mpiblock_code1`. `mpiblock_code3` remains efficient even at higher process counts, which is why we choose it as our primary block-based implementation in the subsequent sections.

TABLE II
SUBSET OF SHORTEST TIMES (S) FOR BLOCK-BASED MPI METHODS

Matrix Size	Procs	mpiblock_code1	mpiblock_code3
256	16	0.000961	0.000573
512	64	0.021816	0.013089
4096	16	0.234491	0.144808

F. Optimal Process Counts for `mpiblock3`

We tested `mpiblock3` with 4, 16, and 64 processes across all matrix sizes. From the measured times, **4 processes is fastest up to** 512×512, while **16 processes prevails at** 1024×1024 **and above**. Hence, in subsequent tests we adopt: 4 processes for $n \leq 512$ and 16 processes for $n \geq 1024$.

This choice minimizes overhead for smaller n and provides better scalability for larger matrices.

V. RESULTS AND DISCUSSION

We compared our implementations by measuring average execution times (after discarding outliers) and by computing speedup and efficiency relative to the serial baseline. Additionally, we considered both strong and weak scaling to evaluate performance scalability.

A. MPI vs. OpenMP vs. Serial Baseline

Figure 1 plots the log-scale execution times against matrix sizes for three configurations: `OpenMP` (using 4, 16, or 32 threads as appropriate), `mpi3` run with 1 process (serving as our serial baseline), and `mpi3` multi-process (with our optimal process counts). For very small matrices (16–64), `OpenMP` outperforms `mpi3` due to minimal communication overhead. In the mid-range (128–512), the multi-process `mpi3` significantly beats its single-process baseline and is competitive with `OpenMP` (e.g., at 512×512). For large matrices (1024–4096), `mpi3` using 32 processes achieves speedups exceeding 3× compared to the serial baseline (see Table III).

Speedup and Efficiency are defined as

$$S = \frac{T_{\text{serial}}}{T_{\text{mpi3-multi}}}, E = \frac{S}{p}.$$

Our results show that for $n \leq 64$ the overhead yields $S < 1$, while for $n \geq 256$ the speedup exceeds 1 with efficiencies around 10%–20%. Further optimization of message aggregation and collective operations could improve these metrics.

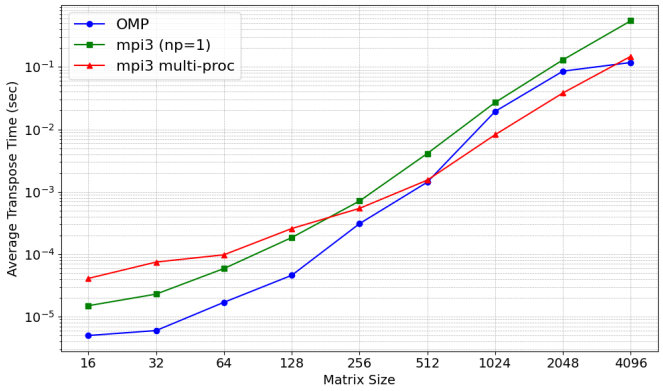

Fig. 1. Execution time vs. matrix size for OpenMP, `mpi3` (1 process), and `mpi3` (multi-process).

TABLE III
SPEEDUP AND EFFICIENCY OF MPI3 MULTI-PROCESS

Matrix Size	Procs	Speedup	Efficiency
16	4	0.37	0.09
256	8	1.31	0.16
1024	32	3.28	0.10
4096	32	3.70	0.12

B. Block-Based MPI implementation

Table IV shows the speedup/efficiency of `mpiblock3` relative to the `serialblock` implementation. Notice that for small matrices, the speedup values are less meaningful due to the negligible runtime of the serial code. For larger matrices, however, `mpiblock3` yields modest speedups (ranging from roughly 1.7× at 1024×1024 to about 1.9× at 4096×4096) using 16 processes, with corresponding efficiency values of approximately 11%–12%.

TABLE IV
EXECUTION TIME, SPEEDUP, AND EFFICIENCY FOR MPIBLOCKS3

Matrix Size	Serial (1p)	**mpiblocks3**	Speedup	Efficiency
16	0.000002 s	0.000045 s (4p)	0.044	0.011
32	0.000006 s	0.000044 s (4p)	0.136	0.034
64	0.000041 s	0.000087 s (4p)	0.471	0.118
128	0.000165 s	0.000256 s (4p)	0.645	0.161
256	0.000678 s	0.000726 s (4p)	0.933	0.233
512	0.002724 s	0.002370 s (4p)	1.15	0.288
1024	0.012478 s	0.007355 s (16p)	1.70	0.106
2048	0.054525 s	0.028297 s (16p)	1.93	0.121
4096	0.245192 s	0.128822 s (16p)	1.90	0.119

1) Execution Time vs. Matrix Size: Figure 2 compares execution times for `serialblock`, `mpi3`, and `mpiblocks3` across matrix sizes. As expected, `serialblock` is fastest for small matrices due to negligible communication overhead. For medium to large sizes, both `mpi3` and `mpiblocks3` outperform the serial baseline, with `mpi3` achieving higher speedup for very large matrices. However, `mpiblocks3` offers competitive performance with improved cache usage.

2) Bandwidth Comparison.: To assess bandwidth efficiency, we estimate bandwidth as BW $= \frac{2\times n^2\times 4}{T}$ bytes/sec,

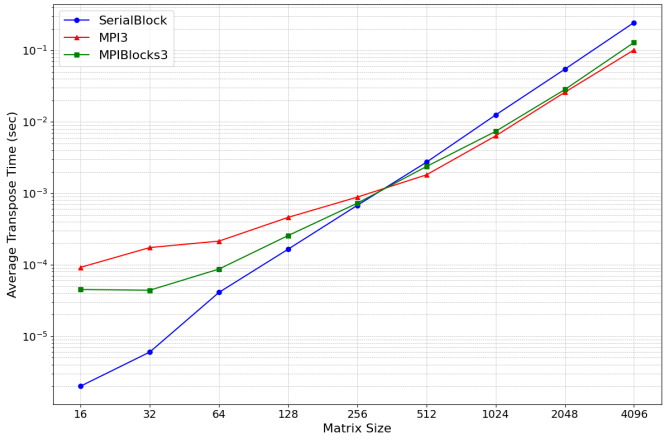

Fig. 2. Execution time vs. matrix size for `serialblock`, `mpi3`, and `mpiblocks3`. Log-scale is recommended for clarity.

accounting for reading and writing each matrix element (assuming 4-byte floats). For 4096×4096, `mpi3` achieves

$$\text{BW} \approx \frac{2 \times 4096^2 \times 4}{0.101358} \approx 1.32 \text{ GB/s},$$

while `mpiblocks3` achieves

$$\text{BW} \approx \frac{2 \times 4096^2 \times 4}{0.128822} \approx 1.04 \text{ GB/s}.$$

These results indicate that both implementations achieve comparable effective bandwidth.

Scalability Analysis:
- Strong Scaling: We examined how execution time decreases with increasing processes for a fixed matrix size. Both `mpi3` and `mpiblocks3` exhibit good strong scaling for larger matrices, though communication overhead limits scalability at smaller sizes.
- Weak Scaling: We evaluated performance when both matrix size and process count increase proportionally. Both `mpi3` and `mpiblocks3` maintain reasonable performance gains, demonstrating effective scalability as problem size grows.

VI. CONCLUSIONS

Our experiments demonstrate that `mpi3` implementation outperforms the serial baseline and competes effectively with OpenMP for larger matrices. Among block-based methods, `mpiblocks3` consistently offers significant speedup over the serial block approach, leveraging improved cache usage. Both strong and weak scaling analyses confirm that `mpi3` and `mpiblocks3` scale well with increasing matrix sizes, though communication overhead remains a challenge for smaller matrices. Future work could explore hybrid MPI+OpenMP implementations to leverage both distributed and shared memory parallelism, potentially leading to significant performance gains. Furthermore, optimizing collective communication routines to enhance bandwidth utilization and minimize communication overhead is a promising avenue for further improvement.

REFERENCES

[1] K. Lee, Y. Kim, and J. Lee, "Optimizing matrix transposition for multicore systems using tiling strategies," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 6, pp. 1074–1086, 2012.

[2] D. Kim and S. Wang, "Scalable data redistribution techniques for large-scale matrix transpose on distributed-memory systems," *Proc. IEEE Int. Conf. High Perform. Comput. Appl.*, pp. 125–132, 2013.

[3] Z. Yang, L. Zhang, and C. Li, "Communication-aware algorithms for matrix transposition on distributed systems," *J. Parallel Distrib. Comput.*, vol. 80, pp. 59–69, 2015.

[4] J. Park, H. Lee, and S. Kim, "Cache-optimized matrix transposition in high-performance computing," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 1, pp. 1–25, 2016.

[5] P. Li, M. Chen, and X. Zhao, "Hybrid MPI/OpenMP matrix transposition on multicore clusters," *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, pp. 307–314, 2018.

[6] Q. Wang, Y. Xu, and F. Liu, "Performance analysis of collective communication operations in MPI-based matrix transposition," *IEEE Trans. Comput.*, vol. 69, no. 4, pp. 650–663, 2020.

[7] H. Zhang, J. Liu, and M. Huang, "Memory-aware optimization for dense matrix operations on modern HPC architectures," *J. Supercomput.*, vol. 75, no. 4, pp. 2000–2015, 2019.

[8] L. Chen, Y. Zhao, and F. Sun, "An empirical study on MPI collective performance for matrix operations," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 10, pp. 2838–2849, 2017.