Parallelizing matrix operations using MPI

applied to Parallel Matrix Transportation problem

Ismaele Landini 235831
Computer science, communications and electronics engineering
University of Trento
Trento, Italy
ismaele.landini@studenti.unitn.it

***Abstract*—This paper focuses on analyzing the performance of sequential, distributed memory parallel programming (MPI), and shared memory parallel programming (OpenMP) within the context of High-Performance Computing (HPC) cluster architectures. The study emphasizes the implementation of matrix transpose operations across these approaches and evaluates their efficiency and scalability under varying workloads and conditions. By comparing these programming paradigms, the paper identifies the most suitable methods for performing matrix transpositions in different scenarios. The findings provide insights into optimizing multi-process applications using message-passing techniques for improved performance and scalability.**

***Index Terms*—1. Introduction, 2. State-of-the-Art, 3. Methodology, 4. Experimental and System Description, 5.Results and Discussion, 6.Conclusions.**

I. Introduction

Multi-process architectures are integral to modern computing systems, enabling large-scale data processing by leveraging multiple CPUs and cores. Parallel programming interfaces such as MPI (Message Passing Interface) provide scalable models for distributed memory programming in languages such as C, C++, and Fortran, enabling efficient communication between processes to decompose and solve computational problems. This paper investigates distributed memory parallel programming using MPI and compares its performance with sequential execution and shared memory parallel programming using OpenMP. The study focuses on matrix transpose operations, through two key functions, matrix transposition and symmetry verification are evaluated on metrics such as weak and strong scalability, efficiency, and bandwidth. Given the critical influence of factors such as problem size, hardware configuration, and communication overhead, this analysis highlights how MPI and OpenMP impact matrix operation performance. By comparing these methodologies, the paper aims to identify the most effective paradigm for different workloads and computational environments.

II. State-of-the-Art

Nowadays a basic operation in computer science, matrix transposition is used extensively in signal processing, numerical simulations, and machine learning. The effectiveness of transposition algorithms becomes essential for maximizing the performance of High-Performance Computing (HPC) applications in relation to dataset sizes grow. Dr. Rolf Rabenseifner's research highlights that **Distributed Memory Programming** with MPI remains a leading approach, effectively distributing workloads across multiple nodes while minimizing communication overhead. By using adaptive block and cyclic distribution techniques, MPI optimizes load balancing and reduces synchronization delays. Whereas the **Shared Memory Programming** via OpenMP is improvements with the advent of high-core count processors. However brings with it, issue of critical section non-uniform memory access persist as bottlenecks for large-scale datasets. Studies of ETH Zurich University demonstrate that Hybrid models combining MPI and OpenMP are gaining traction as they offer improved flexibility in exploiting all processors resources Although, ongoing research continues to refine these approaches, integrating emerging technologies such as quantum computing and AI-driven performance to push over the limitation of both parallel architecture.

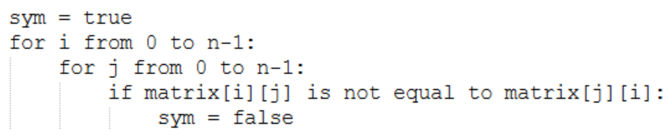
III. Methodology

This section outlines how the setup provides a clear and scalable framework for evaluating the effectiveness of various approaches under different workloads. The matrix transpose operations implemented include two key functions: checkSym and matTranspose. The *checkSym* function determines if the input matrix is symmetric, while *matTranspose* computes the transpose of a given input matrix. The matrix size, specified as a power of 2, is provided as an input parameter and populated with random values. This design ensures scalability, allowing the problem to adapt to increasing workloads and effectively test the capabilities of parallel methods. Performance evaluation is conducted using wall-clock time to measure the execution time of each function. This precise approach ensures reliable comparisons of efficiency and scalability across different implementations.

A. Sequential Implementation

Sequential implementation serves as a baseline for evaluating parallel programming techniques, offering a straightforward approach with basic function and algorithm design. The checkSym function (Fig. 1) iterates through all elements of the input matrix, verifying whether each element is equal to its corresponding element with inverted row and column indices.

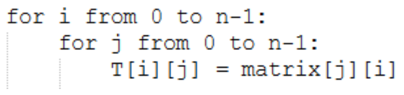
This simple structure provides a foundation for comparison against more complex parallel implementations.



```
sym = true
for i from 0 to n-1:
    for j from 0 to n-1:
        if matrix[i][j] is not equal to matrix[j][i]:
            sym = false
```

Fig. 1. CheckSym function iterates through the matrix with two nested loops, checking if each cell is equal to its corresponding transposed cell.

The matTranspose function (Fig. 2) computes the transpose of an input matrix, M, and stores the result in an output matrix, T. This is accomplished through two nested loops that iterate over the elements of M, swapping rows with columns and assigning the corresponding values to T. The straightforward design ensures clarity and serves as a foundation for assessing the efficiency of parallelized implementations.

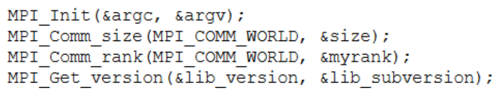


```
for i from 0 to n-1:
    for j from 0 to n-1:
        T[i][j] = matrix[j][i]
```

Fig. 2. MatTranspose function iterates through the matrix with two nested loops, swapping rows with columns and assigning the corresponding values to T.

B. Distributed memory programming

Distributed memory programming leverages multiple processors, each with its private memory, to execute tasks concurrently. This paradigm utilizes Message Passing Interface (MPI), where independent processes with separate memory spaces communicate through message passing. By dividing workloads across processes, MPI enables programs to fully utilize multi-process architectures, taking advantage of its powerful routines. Before utilizing MPI tools, it is essential to define the MPI environment (Figure 3).



```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Get_version(&lib_version, &lib_subversion);
```

Since memory is not shared among processes in distributed memory programming, a copy of the matrix M is distributed to each process using appropriate MPI routines. The MPI_Bcast routine [1] is utilized for this purpose, enabling the broadcasting of the matrix from one process (the root) to all other processes in a communicator, maintaining data consistency across the distributed environment.

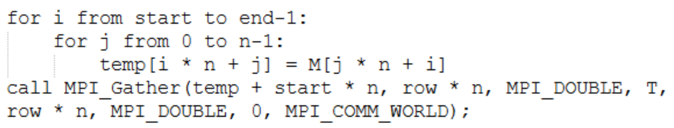
*[1] int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int source, MPI_Comm comm)*

For each function, workload distribution across processes is managed using a 2D Block Distribution. Specifically, the 2D (BLOCK, *) approach divides the matrix into row blocks, assigning each process a contiguous block of rows. This method simplifies both the distribution and the reassembly of data, making it efficient for implementation. In contrast, the 2D (BLOCK, BLOCK) distribution divides the matrix into smaller square blocks, optimizing data distribution by balancing workload more evenly across processes. However, this approach introduces significant complexity in both data distribution and reassembly, as it requires intricate data reorganization to restore the global matrix correctly.

The checkSymIMP function implements the same algorithm as its sequential counterpart, but each process iterates over its assigned set of rows independently. If the matrix is found to be non-symmetric, a local check variable (symPro) is set to false. Since this variable is private to each process, there is no risk of data races. Once all processes complete their iteration, the MPI routine MPI_Allreduce [2] is used to aggregate the check variables from all processes within the same communicator. This operation performs a logical AND across all local results, producing a global result (symRed) that determines whether the matrix is symmetric.

[2] MPI_Allreduce(symPro, symRed, 1, MPI_C_BOOL, MPI_LAND, MPI_COMM_WORLD)

The matTransposeIMP function (Fig. 4) distributes the computation of the matrix transpose across multiple processes, allowing them to work simultaneously. Each process calculates a block of the transpose by swapping the row and column indices of the matrix. Once a process has completed its assigned block, it uses the MPI routine MPI_Gather to send the transposed block back to the root process for consolidation. While MPI_Allgather could be used to gather the transposed data from all processes, MPI_Gather is preferred in this case for efficiency, as it reduces overhead by collecting the data only on the root process.



```
for i from start to end-1:
    for j from 0 to n-1:
        temp[i * n + j] = M[j * n + i]
call MPI_Gather(temp + start * n, row * n, MPI_DOUBLE, T,
row * n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Fig. 3. The matTransposeIMP function performs the matrix transpose across all processes and then collects the final result into the root process using the MPI_Gather routine

C. Shared Memory Programming

The same implementations as Delivery 1 (see reference [0])are used. Performance may vary slightly due to overhead from managing both multi-processing and multi-threading simultaneously. However, the results remain almost identical.

IV. Bottleneck problems

This section explores the various bottlenecks affecting the performance and scalability of matrix transposition, through Shared Memory Programming and the use of the Message Passing Interface (MPI): 1.*Adaptive Block Partitioning*: selecting an optimal data partitioning strategy allows to maximize the performance and avoiding workload imbalances. The program employs a 2D(BLOCK, BLOCK) distribution, ensuring a balanced workload and simplifying implementation across different problem sizes. 2.*Synchronization Delays*: as demonstrated by Professor William Gropp of University of Illinois subdividing and recollecting the dataset among each processes, increase time requirement to spread result at defined processes and if even one process lags, the entire operation is delayed. 3.*Process Overhead*: allocating more processes than necessary may overload the system, resulting to resource saturation and unpredictable performance. Since the system must manage both processes and their associated resources, excessive process creation introduce overhead that outweighs performance gains. Processes require more management and memory, which can further impact efficiency, unlike threads.In order to attain the greatest performance, the study determines the ideal ratio between the number of processes and dataset size. Regarding OpenMP architecture bottlenecks see Delivery.

V. Experiments and System Description

A. *Computing system and platform configurations*

This section describes the computing environment used to evaluate the performance of parallelization techniques in this project. High-Performance Computing (HPC) clusters serve as the backbone of the computational setup, consisting of interconnected nodes that execute tasks in parallel. Each node operates independently with its own processor and memory, making it well-suited for efficiently handling complex workloads. For this project, the system consists of four CPUs, each with one thread per core. The processors are based on the Intel Xeon Gold 5118 architecture, running at 2.30 GHz. For this project are requested 64 CPU core, 64 OpenMP threads and 64 MPI processes. Access to the HPC cluster is secured via SSH protocols, ensuring safe connections through the university network. Users connect using built-in SSH clients on Linux and macOS or external tools like PuTTY and MobaXterm on Windows, often in conjunction with Virtual Private Network (VPN) connections for secure remote access.The matrix transposition operations were implemented in C/C++, utilizing standard libraries such as stdio.h, stdlib.h, and time.h for input/output and timing functions. The mpi.h library played a crucial role in enabling multi-process programming through the Message Passing Interface (MPI). This carefully designed setup provided an optimized computational environment for testing and analyzing distributed memory programming techniques.

B. *Experiments design*

This project focuses on testing and exploring distributed memory programming using the Message Passing Interface (MPI). Employing all necessary MPI library functions to distribute and collect data among processes, with the goal of evaluating the performance of matrix transposition operations. As noted earlier, multi-process architectures are more challenging to manage compared to multi-thread implementations, often resulting in anomalous performance values. Which performance results on a shared computing cluster can vary due to several factors including dynamic CPU scheduling, contention among concurrent processes, variations in node performance due to simultaneous usage by other users. All these factors introduce variability and underscore the importance of carefully analyzing the computing environment when computing a multi-process programming. To mitigate these, multiple tests (10 times) were conducted to gather sufficient data and normalize results. We tested varying numbers of processes/threads, scaling from 2 up to 64, and analyzed speedup, scalability, efficiency, and bandwidth. This approach offers insights into the impact of parallelization on performance and resource utilization. Furthermore, the number of processes must be at least equal to the matrix size (N) to ensure proper workload distribution. This is necessary for effectively dividing the matrix into blocks of rows, allowing each process to handle an appropriate portion of the task.

VI. Results and Discussion

The analysis focuses on the matTranspose function across all implementations, highlighting the relationship between matrix size, the number of MPI processes, and the number of threads. Each technique is evaluated based on execution time, bandwidth, and, for parallel architectures, strong and weak scalability, as well as efficiency. Contrary to expectations, the multi-process architecture did not show a significant performance improvement. In contrast, the multi-thread implementation demonstrated a direct relationship between the number of threads and matrix size (Fig. 5).

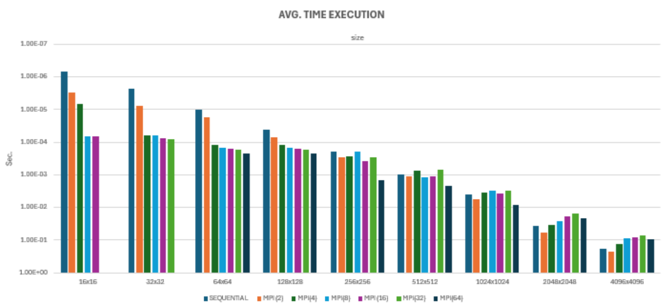


Fig. 4. The average execution time for sequential and distributed memory implementations measured across a range of matrix sizes, from 16x16 to 4096x4096. The taller the bar, the lesser execution time required

Even with larger matrices, MPI performance did not scale as expected. Only when processing very large datasets do multiple processes begin to offer communication benefits. However, these gains are marginal compared to sequential execution. As shown in the analysis (Fig. 5), only matrix sizes of 2048×2048 and 4096×4096 demonstrate a slight improvement in performance. For smaller datasets, the sequential implementation actually performs better. This confirms our initial thesis: managing distributed memory processes incurs significant resource overhead due to the time required for data distribution and collection among processes. Consequently, MPI-based approaches are best suited for applications involving very large datasets. However, when compared to OpenMP techniques, the improvements from MPI are less significant. As the matrix size increases, explicit parallelization methods like OpenMP show clear advantages, with configurations of 8 or 16 threads providing optimal performance.

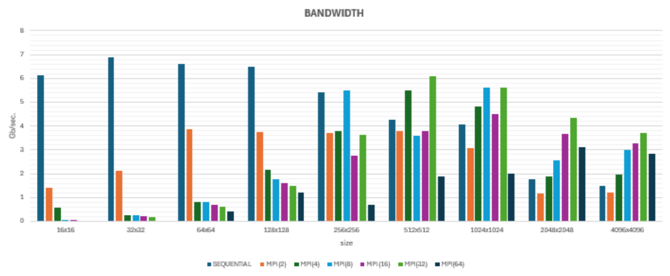


Fig. 5. Bandwidth (bandwidth = data transfered / time taken) for sequential, distributed, and shared memory implementations was measured across a range of matrix sizes, from 16x16 to 4096x4096.

This trend is particularly evident in the bandwidth measurements (Fig. 6). Initially, sequential implementations deliver impressive bandwidth efficiency due to their streamlined performance with smaller datasets. However, as the workload increases, sequential methods begin to fall behind, allowing multi-process programming with an optimal number of processes to take a lead. Although the MPI implementation maintain consistent performance across different data sizes with slightly increase for great workload, less competitive in peak bandwidth compared to the optimized multi-thread approach.

Weak scalability, which measures how effectively the system maintains performance as both the number of processes and the workload increase proportionally, presents a different picture compared to the analysis of bandwidth and execution time. In this case, as both the matrix size and the number of processes grow, we observe a degradation in performance, falling significantly below the expected threshold (Fig. 7). This performance decline is likely due to factors such as communication overhead or memory limitations, which hinder the efficiency of scaling with an increasing problem size.

AVG. SCALABILITY	SEQUENTIAL	MPI (2)	MPI(4)	MPI(8)	MPI (16)	MPI(32)	MPI(64)
16x16	6.68E-07						
32x32		0.000008					
64x64			0.000122				
128x128				0.000149			
256x256					0.000377		
512x512						0.000687	
1024x1024							0.008381
RESULT		8.35E-02	5.47E-03	4.48E-03	1.77E-03	9.72E-04	7.97E-05

Fig. 6. Average Weak Scalability (weak scalability = time sequential execution / (execution on p processor/threads) * size of matrix). Red text color defines if under threshold 1, which defines the system scales well, meaning additional processes handle the extra workload efficiently.

The analysis of speedup and efficiency metrics for MPI and OpenMP show that shared memory programming delivers a notable performance boost for matrix transposition tasks. For matrices of size 512×512 and larger, MPI based implementations show increased computational speed, though with some anomalous performance, such as with 4096×4096 matrices using 8 or 16 processes, likely due to the overhead associated with process management on the HPC system and operations of gather. In contrast, the OpenMP implementation, benefiting from the lightweight nature of thread-based parallelism, achieves good performance even for 256×256 matrices, although it tends to underperform for very small matrix sizes.

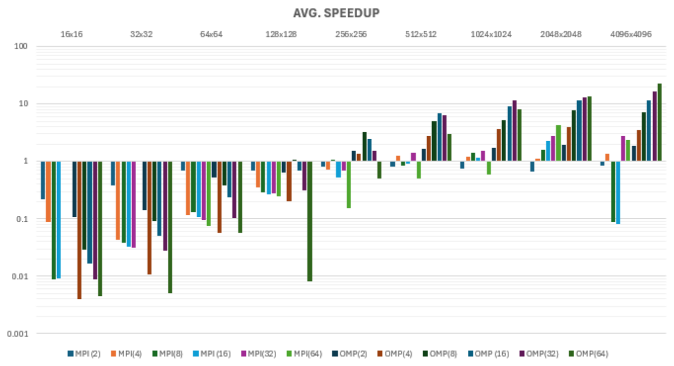


Fig. 7. Average Strong Scalability (scalability = time sequential execution / exution on p processor) for sequential, distributed, and shared memory implementations was measured across a range of matrix sizes, from 16x16 to 4096x4096. Where more MPI implementations perform notably worse, falling below a threshold of 1, which defines if there is an increase of performance compare to sequential implementation

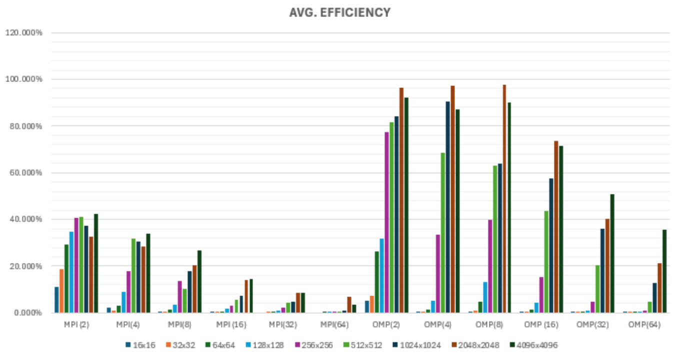


Fig. 8. Average Efficiency (efficiency = time sequential execution / (exution on p processor) * number of process/threads

VII. Conclusion

In summary, this study evaluated the performance of matrix transposition using three approaches: sequential, MPI-based distributed memory, and multi-threaded OpenMP implementations. Our findings decisively show that increasing the number of MPI processes does not translate into significant performance gains; in fact, using fewer processes enhances communication efficiency. OpenMP, on the other hand, consistently outperforms MPI by delivering superior scalability and efficiency, especially when an optimal number of threads is employed on larger matrices. While MPI remains a viable option for extremely large matrices in distributed systems, the evidence clearly favors OpenMP for most scenarios—offering better performance and scalability for small to medium-sized matrices. Consequently, OpenMP emerges as the recommended approach for achieving optimal matrix transposition performance in parallel computing environments.

REFERENCES

[1] Ismaele Landini (2024),Exploring Implicit and Explicit Parallelism with OpenMP, Delivery 1

[2] William Gropp (2003),Improving the Performance of MPI Derived Datatypes by Optimizing Packing Algorithms,

[3] Bruno Magalhães (2013), Distributed Matrix Transpose Algorithms,

[4] Marco Ferretti (2022), Parallel Computing with MPI

[5] Kale, V. (n.d.). High-performance parallel programming techniques*. University of Southern California / Information Sciences Institute. Retrieved from https://www.isi.edu/

[6] Laura del Rìo Martìn (2024). Introduction. Basic MPI Operations. Introduction to parallel computing. University of Trento.

[7] Laura del Rìo Martìn (2024). Collective Communication. Introduction to parallel computing. University of Trento.

[8] Laura del Rìo Martìn (2024). ollective Communication (Part II). Introduction to parallel computing. University of Trento.

[9] Laura del Rìo Martìn (2024). Advanced Collective Communication and Derived Data Types Introduction to parallel computing. University of Trento.

[10] Sol Ji Kang, Sang Yeon Lee, and Keon Myung Lee (2013), Performance Comparison of OpenMP, MPI, and MapReduce inPractical Problems, Department of Computer Science, Chungbuk National University

[11] Rabenseifner, R., Hager, G., Wellein, G. (2019). Advanced strategies for MPI workload distribution. International Conference on High-Performance Computing.

[12] Ben-Nun, T., Hoefler, T. (2020). GPU-accelerated hybrid models: Challenges and solutions. Computing Surveys.

[13] Zhonghe Zhou (2016), National Science Review: Volume 3 Issue 1 March 2016, Oxford Academy.

[14] William Gropp (2020), Challenges for MPI in Its Third Decade, University of Illinois.

[15] Ben-Nun & Hoefler (2020), Understanding performance bottlenecks in modern HPC systems. ETH Zurich University.