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

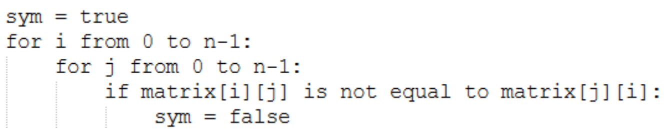
Matrix transposition is a vital operation in computing science, underpinning algorithms in fields such as machine learning, numerical simulations, and signal processing. Its efficiency greatly impacts the performance of High-Performance Computing (HPC) applications, especially for large-scale datasets. **Sequential implementations**, while simple and free of concurrency issues, become inefficient as matrix sizes grow. **Distributed memory programming** with MPI overcomes this by distributing workloads across nodes via message passing, using strategies like block and cyclic distribution to optimize load balancing and minimize communication overhead. However, effective workload partitioning remains a challenge. **Shared memory programming**, enabled by OpenMP, leverages multi-core processors by distributing tasks across threads in a shared memory environment. Though effective for medium-scale problems, it is constrained by memory architecture and lacks scalability for larger datasets. Hybrid models combining MPI and OpenMP offer potential for improved resource utilization but require further refinement to address challenges in dynamic workload balancing and inter-process communication.

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

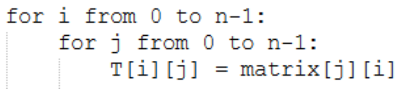
Sequential implementation serves as a baseline for evaluating parallel programming techniques, offering a straightforward approach with basic function and algorithm design. The checkSym function (Fig. 1) iterates through all elements of the input matrix, verifying whether each element is equal to its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



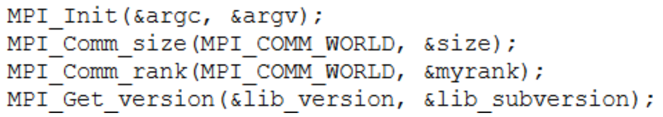
```
for i from 0 to n-1:
    for j from 0 to n-1:
        T[i][j] = matrix[j][i]
```

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. To initialize and manage the MPI environment, several key routines are implemented:

- MPI_Init: Initializes the MPI environment and must be invoked at the beginning of any MPI program. It accepts argc and argv as command-line arguments for program initialization.
- MPI_Comm_size: Determines the total number of processes within the specified communicator, providing a global view of the distributed system.
- MPI_Comm_rank: identifies the rank (a unique identifier) of the current process within the communicator, enabling task differentiation.



```
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);
```

Fig. 3. MPI environment

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.

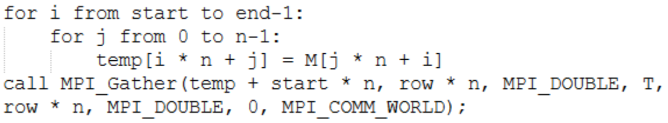
*[1] int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int source, MPI_Comm comm)*

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. 4. 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 are used. Performance may vary slightly due to overhead from managing both multi-processing and multi-threading simultaneously. However, the results remain almost identical.

D. Bottleneck problems

Several bottlenecks can affect the performance and scalability of matrix transposition implementations: 1.Communication Overhead: transferring blocks of the matrix between processes can introduce significant latency, especially for large matrices or when using a high number of processes. moreover collective operations, such as gathering data at the root process, can create bottlenecks if the number of processes is large or the volume of data is substantial. 2.Load Imbalance: uneven distribution of matrix blocks among processes can lead to some processes finishing early and remaining idle while others continue to work, reducing overall efficiency. 3.Hybrid Parallelism (MPI and Multi-Threading): combining MPI with multi-threading may result in contention and overhead due to the complexity of managing both types of parallelism. In this implementation, multi-threading is used exclusively on the root process (rank equal to 0), to avoid more multi-threading operation across processes. 4.Weak Scaling: when the matrix size is too small relative to the number of processes, communication overhead can dominate computation, negating the benefits of parallelism. In such cases, the operation may become inefficient and is likely to be avoided.

IV. Experiments and System Description

A. Computing system and platform configurations

Understanding the computing environment is fundamental for evaluating the performance of parallelization techniques in this project. High-Performance Computing (HPC) clusters form the backbone of the computational setup, comprising interconnected nodes that collaboratively execute tasks in parallel. Each node operates independently, with its own processor and memory, making it well-suited for handling complex workloads efficiently. For this project, the system utilized consists of a node equipped with 64 cores and configured to run 64 MPI processes, with each process allocated 1 MB of RAM. This setup enables scalable and efficient parallel execution. Access to the HPC cluster is secured through SSH protocols, ensuring safe connections via the university network. Users connect to the cluster 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 external access. The matrix transpose operations were implemented using C/C++ programming, leveraging standard libraries (e.g., ‘stdio.h‘, ‘stdlib.h‘, and ‘time.h‘) for input/output and timing operations. The ‘mpi.h‘ library was pivotal in enabling multi-process programming through Message Passing Interface (MPI) communication. This carefully designed setup provided an optimized computational environment for testing and analyzing distributed memory programming techniques.

B. Experiments design

This project explores distributed memory programming using MPI for communication among processes. The goal is to evaluate the performance of matrix transpose operations across multi-process and multi-thread implementations. Performance on shared computing clusters can vary due to factors like: 1.Dynamic CPU Scheduling: Task prioritization by the OS can impact performance consistency. 2.Processes are more complex than threads, requiring separate memory and communication, while threads share memory and are CPU-managed. 3.Node Contention: Resource competition with other users can affect results. 4.Memory Latency: Cache misses or memory fetch delays can cause performance variability. 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.

V. 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).

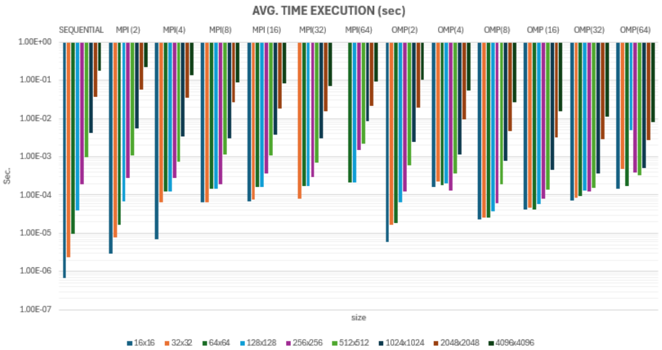


Fig. 5. The average execution time for sequential, distributed, and shared memory implementations was measured across a range of matrix sizes, from 16x16 to 4096x4096.

Even with larger matrices, MPI performance did not scale as anticipated; instead, fewer processes were preferred for communication. This may be due to the complexity of managing MPI, which introduces delays when handling smaller data volumes, even with large matrices. Additionally, threads share memory and do not require message-passing for communication, making them more efficient in data sharing compared to processes, which rely on more complex mechanisms for inter-process communication. Thus, results suggest that using fewer processes improves communication efficiency and reduces the number of processes needed for inter-process communication. However, when compared to OpenMP techniques, the improvements from MPI are less significant. As seen in Delivery 1, OpenMP outperforms MPI in many cases. As the matrix size increases, explicit parallelization methods like OpenMP show clear advantages, with configurations of 8 or 16 threads providing optimal performance.

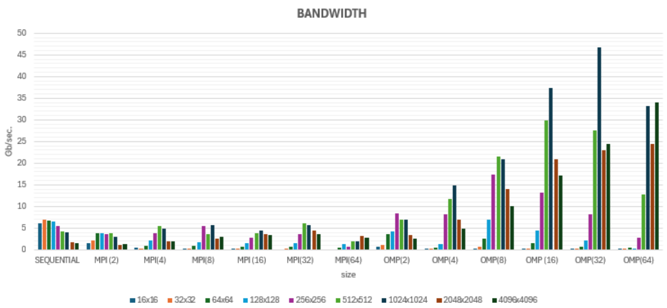


Fig. 6. Bandwidth (bandwidth = data transfered / time taken) for sequential, distributed, and shared memory implementations was measured across a range of matrix sizes, from 16x16 to 4096x4096.

This is particularly evident when comparing the bandwidth measurements (Fig. 6) between these implementations. OpenMP achieves significantly higher bandwidth than the MPI architecture, which struggles to outperform sequential programming in terms of bandwidth. These differences are directly linked to execution time. Initially, sequential methods demonstrate higher bandwidth due to their efficiency with smaller data sizes. However, as the workload increases, sequential methods begin to underperform, and multi-threaded programming with an optimal number of threads takes the lead. In contrast, MPI implementations maintain a consistent but less competitive performance.

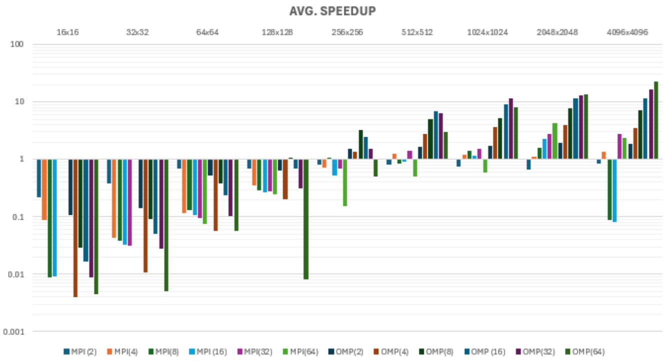


Fig. 7. Average Strong Scalability (scalability = time sequential execution / exution on p processor) for sequential, distributed, and shared memory implementations was measured across a range of matrix sizes, from 16x16 to 4096x4096. Where more MPI implementations perform notably worse, falling below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

This is further emphasized when examining weak (Fig. 8) and strong scalability, as well as efficiency (Fig. 9) measurements. The results show that MPI communication with a low number of processes maintains consistent behavior as the matrix size increases. However, with a high number of processes, performance becomes more oscillatory, especially with both small and large matrix sizes. This is particularly evident in the strong scalability measurements, where MPI techniques with 8 or 16 processes exhibit increasing oscillations as the matrix size grows (Fig. 7). As we have observed, multi-threading significantly improves performance as the workload increases, particularly when an optimal number of threads is assigned.

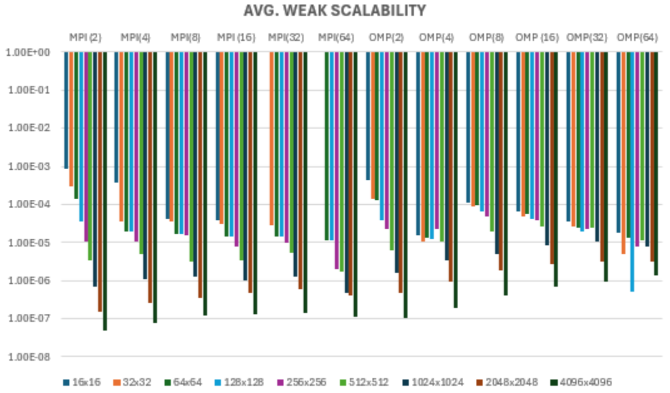


Fig. 8. Average Weak Scalability (weak scalability = time sequential execution / (exution on p processor/threads) * size of matrix)

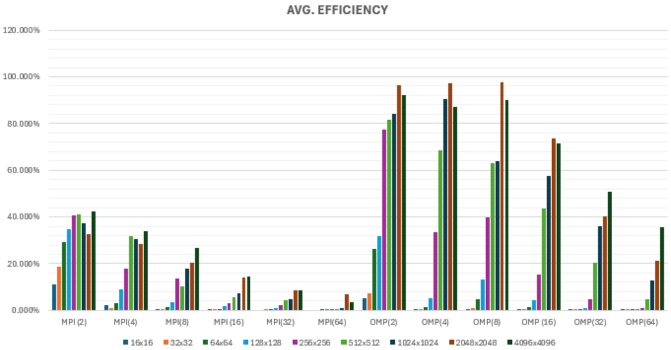


Fig. 9. Average Efficiency (efficiency = time sequential execution / (exution on p processor) * number of process/threads

VI. Conclusion

This study evaluates the performance of matrix transposition using three approaches: sequential, MPI-based distributed memory, and multi-threaded OpenMP implementations. The results indicate that MPI, particularly with a higher number of processes, did not yield significant performance gains. Instead, fewer processes enhanced communication efficiency. However, MPI's performance was generally inferior to OpenMP, which demonstrated superior scalability and efficiency, especially when an optimal number of threads was used for larger matrices. While MPI remains useful for very large matrices in distributed systems, OpenMP is the preferred approach for most scenarios, as it offers better scalability and performance for smaller to medium-sized matrices. This approach ensures optimal performance for matrix transposition tasks in parallel computing environments.

References

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

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

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

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

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

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

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

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

[9] 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