Parallel Computing

*Note: Sub-titles are not captured in Xplore and should not be used

1st Leonardo Falsarolo, student ID 237821
Department of Information Engineering and Computer Science
University of Trento (Trentino Alto Adige)
leonardo.falsarolo@studenti.unitn.it
https://github.com/FlsLeonardo/Parallel-computing-second-mid-term

***Abstract*—Matrix transposition is fundamental in scientific computing. This study evaluates serial, OpenMP, and MPI approaches, demonstrating significant speedups with parallel methods. OpenMP excels in shared memory, while MPI ensures scalability in distributed systems.**

***Index Terms*—Matrix Transpose, Parallel Computing, OpenMP, Data Parallelism, MPI Optimization, Threads, Processes .**

I. Introduction

Parallel computing is crucial for modern scientific and engineering applications as it enables high performance and scalability by using multiple processing units or thread concurrently, this is particularly important for matrix transposition. It is a fundamental operation that swaps rows and columns, it becomes challenging with large datasets, making it well-suited for parallel optimization.

This work explores three distinct approaches to parallelizing matrix transposition:

- **Serial Computation**: The baseline method, where the operation is performed sequentially on a single processing unit. [1]
- **OpenMP (Open Multi-Processing)**: A shared-memory parallel programming model that allows efficient utilization of multi-core processors through thread-based parallelism. [1]
- **MPI (Message Passing Interface)**: A distributed-memory model that enables parallel processing across multiple nodes, making it suitable for large-scale computing environments.

The primary goal of this project is to evaluate the effectiveness of these methods for matrices of varying sizes, identifying the optimal strategies for different scenarios. By comparing efficency, speedup and bandwidth, this study aims to provide practical insights into the trade-offs between shared-memory and distributed-memory parallelization techniques. These findings will guide developers in selecting the most efficient approach for matrix transposition based on their specific application requirements and hardware configurations.

II. State Of Art

A. Current research.

Recent advances in parallel computing have significantly enhanced the performance of numerical simulations across various scientific and engineering domains. Notably, Gunawan [3] explore the use of OpenMP to solve 1D shallow water equations with high-order time discretization, improving computational efficiency in hydrodynamic simulations. Their work demonstrates how parallelization with OpenMP can reduce computation time while maintaining accuracy. Another important contribution by Ban [4] presents an advanced parallelism technique for the Discontinuous Galerkin Time-Domain (DGTD) method in electromagnetic simulations. They introduce a novel MPI-based algorithm that unifies message-passing and shared-memory parallelism, resulting in enhanced speed and accuracy for time-domain simulations. Finally, Temuçin [5] propose a novel GPU-initiated MPI communication method, where the GPU manages communication traditionally handled by the CPU. This GPU-aware approach optimizes performance in high-performance computing tasks, including complex calculations like Jacobian matrix evaluations, by fully leveraging GPU computational power.

B. Limitations

Recent advancements in parallel computing, while promising, face some limitations: Gunawan [3] highlight scalability issues in OpenMP when dealing with higher-dimensional problems and non-uniform grids, Ban [4] encounter communication overhead in large-scale simulations due to MPI, and Temuçin [5] face limitations in MPI communication with GPUs, with limited applicability for certain algorithms.

C. Gap my project aims to fill

Unlike studies [3] [4] [5] , which explore parallelization methods specific to hydrodynamic, electromagnetic, and GPU-initiated communication tasks, my work analyzes the efficiency, speedup, and bandwidth of matrix transposition. This analysis provides insights into how these methods scale with increasing dimensions and computational resources, offering a broader perspective on the trade-offs between serial, OpenMP, and MPI-based solutions.

III. Methodology

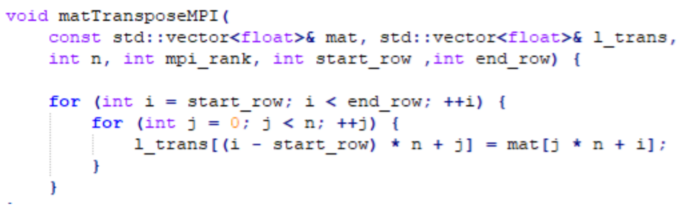
A. Methods and Implementations

The matrix transposition algorithm used in my project is based on Method 1 from my first report [1]. The first approach is the serial one, where I traverse the entire matrix to transpose it. In the OpenMP case, I use the directive `#pragma omp parallel for collapse(2)` to parallelize the transposition loop, where the number of threads is dynamically specified. [1] The novelty of this implementation is the use of MPI for distributed matrix transposition. In the `matTransposeMPI` function, each process is assigned a portion of the matrix, identified by the `start_row` and `end_row` indices, and computes the transposition only for its assigned portion. To do this, I used a 1D array (flattened) to represent the matrix, as MPI does not directly support passing multidimensional arrays.

In this implementation, the **rank 0** process contains the original matrix, which is broadcast to all other processes using the `MPI_BCAST` function. Through this operation, each process receives a complete copy of the matrix and can compute the transposition on its assigned portion. In the transposed matrix, I use a **column-major** [2] index to fill the destination array `l_trans`, while the data is fetched from the original matrix, which follows a **row-major** [2] index. This approach allows me to correctly access the elements of the original matrix and transpose them into the correct position in the destination array. In practice, to access an element in the original matrix, I use the index `mat[j * n + i]`, which follows the **row-major** [2] convention, while to store the element in the transposed array, I use the index `l_trans[(i - start_row) * n + j]`, which follows the **column-major** [2] convention.

Furthermore, the transposed data is gathered in the rank 0 process using `MPI_GATHER`. [IV-B]

The `matTransposeMPI` function is defined as follows:



```
void matTransposeMPI(
    const std::vector<float>& mat, std::vector<float>& l_trans,
    int n, int mpi_rank, int start_row ,int end_row) {

    for (int i = start_row; i < end_row; ++i) {
        for (int j = 0; j < n; ++j) {
            l_trans[(i - start_row) * n + j] = mat[j * n + i];
        }
    }
```

Fig. 1. Mpi local transposition

The code is organized across multiple files: the `main.cpp` file handles the execution, while the serial, OpenMP, and MPI implementations are split across three separate files. Common functions are declared in the `Functions.h` file. Execution times and data related to the implementations are saved in three separate CSV files, which are then analyzed by creating graphs using Python with the Matplotlib library to visualize performance metrics such as speedup, efficiency, and bandwidth. [1]

In summary, the matrix is divided among the various processes, each of which computes the transposition for its subset of rows, and then the results are gathered to form the complete transposed matrix.

B. Challenges faced

In parallel computing, especially when using MPI for matrix transposition, several challenges arise that must be addressed to ensure efficient performance. These challenges include:

- **Data Communication and Synchronization**: Ensuring efficient data communication and synchronization between processes is critical, particularly when dealing with large matrices, to prevent delays and race conditions that can cause significant performance bottlenecks.
- **Memory Management**: Effective memory management is essential in parallel environments to avoid excessive memory usage and inefficiencies, especially when broadcasting and gathering matrix data across processes.
- **Load Balancing**: Properly distributing the workload across processes is key to achieving optimal performance, as imbalances can result in idle times for some processes, ultimately reducing overall efficiency.

IV. Experiments and System Description

A. Description of the computing system and platform

For the tests I have used the UniTN's cluster . This experiments were conducted on a high-performance computing system equipped with 64 Intel Xeon Gold 5218 CPUs, each with 16 cores per socket and having 4 Sokets, with NUMA architecture to optimize memory access:

- **NUMA node0**: CPU(s) from 0 to 15
- **NUMA node1**: CPU(s) from 16 to 31
- **NUMA node2**: CPU(s) from 32 to 47
- **NUMA node3**: CPU(s) from 48 to 63

The performance of my parallel implementations, utilizing both OpenMP with shared memory and MPI with distributed memory, is compared against their respective sequential algorithms. The analysis focuses on evaluating the performance improvements, processing time reductions, and data transfer efficiency across different matrix sizes, considering how OpenMP leverages shared memory for thread-level parallelism [1] and how MPI handles distributed memory across multiple processes.

B. Relevant specifications or configurations.

For the MPI implementation, the following methods and configurations were used:

- **Bcast**: This method is used to broadcast the matrix from the process with rank 0 to all other processes, ensuring that every process has the same data.
- **Gather**: This method collects the transposed matrix from all processes and gathers the data back into rank 0.

For the serial and OpenMP implementations:

- The configurations and methods used are the same as in Method 1 [1].

The tests were conducted for matrices with sizes ranging from 2^4 to 2^{12}, and the number of processes and threads followed the pattern of 1, 2, 4, 8, 16, 32, and 64.

To run everything with the necessary versions, the following command was used:

```
module load mpich-3.2.1--gcc-9.1.0
```

C. Description of the experimental setup, procedures, and methodologies used in the project

Given the dimension of the matrix, it is possible to evenly distribute the work across all MPI processes. This ensures that each process handles a portion of the matrix proportional to the total size, leading to balanced workload distribution and optimal performance.The transposition is done in 3 steps:

- **Matrix Flattening:** The matrix is first flattened into a 1D array using **row-major indexing** [III]. This step is necessary to send the matrix data to the MPI processes, as MPI does not support multi-dimensional arrays. The flattened array is then distributed among the MPI processes for parallel computation.
- **Matrix Transposition:** Each process performs the transposition of its assigned portion of the matrix. The original matrix is accessed using **row-major indexing** [III] (i.e., `mat[j * n + i]`), while the transposed data is stored using **column-major indexing** [III] (i.e., `l_trans[(i - start_row) * n + j]`), ensuring the correct placement of the elements in the transposed matrix.
- **Matrix Deflattening and Gathering:** After the transposition, the data is gathered [IV-B] at the root process (rank 0) using `MPI_Gather`. The 1D array representing the transposed matrix is then **deflattened** [III] back into a 2D matrix, reconstructing the final transposed matrix for further analysis.

D. Discussion on how experiments are designed to test the hypotheses or achieve the objectives.

The experiments were designed to test hypotheses regarding the performance of parallel implementations, focusing on efficiency and bandwidth. Beyond confirming established hypotheses, they aim to uncover new behaviors.

- **Efficiency:**
  - For MPI, it is hypothesized that efficiency decreases as the number of processes increases, due to communication overhead.
  - For OpenMP, a slight decrease in efficiency is expected as the number of threads increases, caused by competition for shared resources.
- **Bandwidth:**
  - MPI is expected to exhibit higher bandwidth compared to shared-memory approaches, thanks to its distributed model. However, excessive processes might reduce bandwidth due to communication overhead.
  - OpenMP is anticipated to maintain stable bandwidth for moderately sized matrices, with potential limitations on larger matrices caused by the architecture used.

V. Results and Discussion

A. Presentation of result.

In addition to considering efficiency and speedup, which were discussed in the previous report [1], the performance of the parallel matrix transposition is also evaluated using weak scaling, strong scaling, and bandwidth.

a) Weak and Strong Scaling: **Weak scaling** examines how the performance changes when both the problem size and the number of processes increase proportionally. **Strong scaling** evaluates how the performance improves when the problem size remains constant and the number of processes increases. The formula for both weak and strong scaling is:

$$S = \frac{T_{\text{serial}}(n)}{T_{\text{parallel}}(n, P)}$$

where n is the problem size and P is the number of processes.

b) Bandwidth: The bandwidth B is the rate at which data is transferred between memory and processes. It can be calculated using the formula:

$$B = \frac{2 * \text{Data Transferred}}{T_{\text{execution}}}$$

where $T_{\text{execution}}$ is the time taken to complete the parallel operation, and the $2*$ needs to be there because we have to read and write the data, the "Data Transferred" refers to the amount of matrix data being transmitted during the execution taking into account that it's a float matrix.

B. Analysis and interpretation in context.

Looking at the first graph [Fig 2], it is evident that:

1. MPI has lower speedup compared to OpenMP**: Due to communication overhead, MPI shows a lower speedup than OpenMP, which achieves higher speedup, especially for smaller matrices. [IV-D]

2. Speedup decreases with increasing matrix size**: For both implementations, speedup is lower for larger matrices, indicating that performance is affected by matrix size. [IV-D]

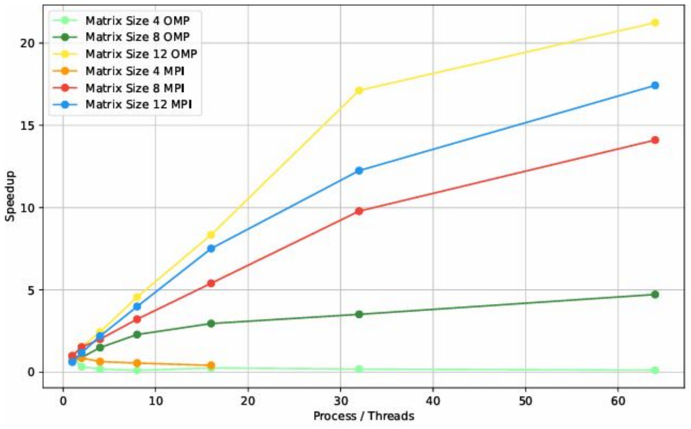


Fig. 2. speedup strong scaling

In the second graph [Fig 3], we can observe the following:

- **Efficiency of MPI and OpenMP**: Both show a decrease in efficiency as the number of processes/threads increases.

However, MPI loses efficiency more quickly due to communication overhead, while OpenMP maintains higher efficiency thanks to shared memory. [IV-D]

- **Strong Scaling**: Efficiency decreases with the increase in processes/threads, with a greater drop for larger matrices due to communication overhead (MPI) and resource contention (OpenMP). [IV-D]

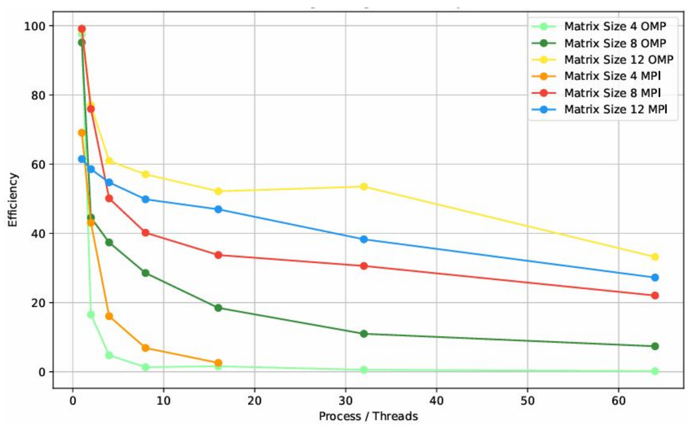


Fig. 3. Efficiency strong scaling

For MPI, as seen in the third graph [Fig 4], the weak scaling efficiency decreases as the number of processes increases due to communication overhead. For OpenMP, the weak scaling efficiency decreases as the number of threads increases, caused by competition for shared resources. [IV-D]

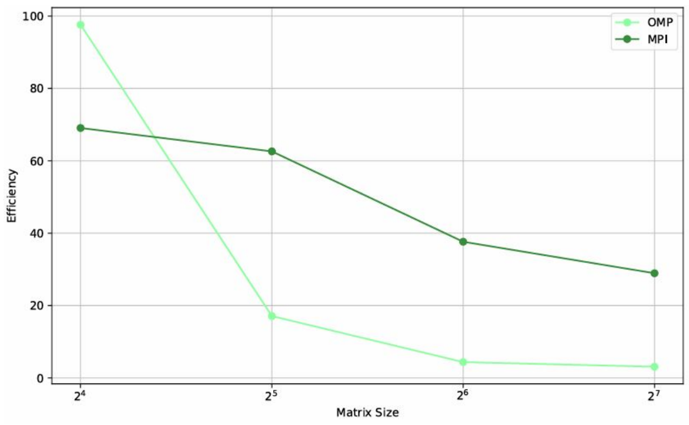


Fig. 4. Efficiency weack scaling

Note: Data collected with 1 thread and 1 process were compared to the serial implementation for all the graphs above.

For MPI, as seen from the final graph [Fig 5], bandwidth is generally higher due to its distributed model, but it decreases as the number of processes increases due to communication overhead. For OpenMP, as seen from the graph, bandwidth remains stable for moderately sized matrices, but it may decrease with larger matrices due to contention for shared memory resources. [IV-D]

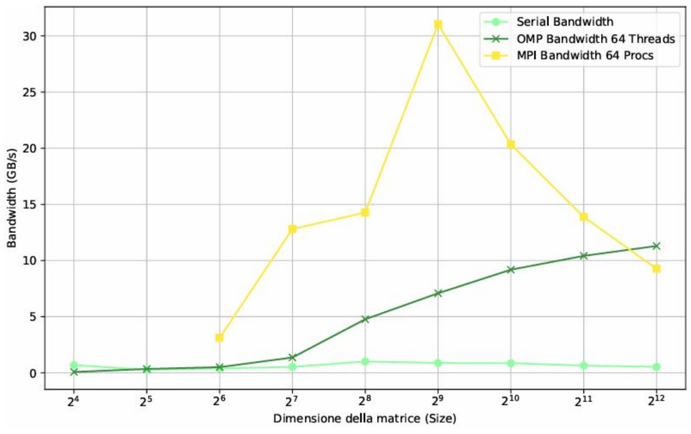


Fig. 5. Bandwidth comparison

Conclusion: The hypotheses on speedup, efficiency, and bandwidth are confirmed by the graphs.[IV-D] For MPI, speedup increases with processes but drops due to communication overhead, while bandwidth is higher but declines with more processes. Efficiency decreases as processes increase. For OpenMP, speedup grows with threads but diminishes at higher counts due to resource contention. Bandwidth remains stable for moderate matrix sizes but decreases for larger ones. Efficiency slightly drops as the number of threads rises. Overall, the results align with the expected trends for both parallel implementations.

C. *Comparison with state of art*

While recent advances in parallel computing, such as those by Gunawan [3], Ban [4], and Temuçin [5], focus on optimizing performance for specific problems using OpenMP, MPI, and GPUs, my project stands out for its performance analysis of matrix transposition using different parallel methods (MPI and OpenMP).

Unlike techniques that focus solely on optimizing performance for specific applications, my work provides a detailed comparison of speedup, efficiency, and bandwidth metrics, offering a statistical basis for improving parallel implementations.

VI. Conclusion

The project explored the performance of matrix transposition using parallel approaches with MPI and OpenMP, analyzing key metrics such as speedup, efficiency, and bandwidth. The results confirm the initial hypotheses regarding the performance of parallel implementations, highlighting the advantages and limitations of each approach. In particular, MPI showed higher bandwidth, but with a performance drop due to communication overhead, while OpenMP maintained stable bandwidth for moderately sized matrices but experienced a decrease for larger matrices.

The work also provided a comparative analysis that enriches the understanding of the behaviors of parallel implementations in various scenarios.

References

[1] Falsarolo Leonardo Parallel Computing first midterm

[2] https://www.mathworks.com/help/coder/ug/what-are-column-major-and-row-major-representation-1.html

[3] P. H. Gunawan, I. Iryanto, I. Palupi and N. Ikhsan, OpenMP Performance for 1D Shallow Water Equations with High Order of Time Discretization, 2023 3rd International Conference on Intelligent Cybernetics Technology & Applications (ICICyTA), Denpasar, Bali, Indonesia, 2023, pp. 501-505, doi: 10.1109/ICICyTA60173.2023.10428994.

[4] Z. G. Ban, Y. Shi and P. Wang, ”Advanced Parallelism of DGTD Method With Local Time Stepping Based on Novel MPI + MPI Unified Parallel Algorithm,” in IEEE Transactions on Antennas and Propagation, vol. 70, no. 5, pp. 3916-3921, May 2022, doi: 10.1109/TAP.2021.3137455.

[5] Y. H. Temuçin, W. Schonbein, S. Levy, A. Sojoodi, R. E. Grant and A. Afsahi, ”Design and Implementation of MPI-Native GPU-Initiated MPI Partitioned Communication,” SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis, Atlanta, GA, USA, 2024, pp. 436-447, doi: 10.1109/SCW63240.2024.00065