IntroPARCO 2024 H2

January 19, 2024

Dennis Alberti
238067
dennis.alberti@studenti.unitn.it
https://github.com/Dennis-Alberti/IntroPARCO-2024-H2

***Abstract*—This report explores explicit parallelization techniques using Message Passing Interface (MPI) by implementing a matrix transposition operation. The project includes a sequential implementation as a baseline, followed by parallel implementations using MPI. Performance analysis evaluates the efficiency, scalability, and speedup of the parallel approach relative to the sequential version. Additionally, the results are compared to an OpenMP implementation to identify key advantages and limitations of each approach. These findings contribute to understanding the trade-offs and advantages of MPI demonstrating superior scalability for larger matrices, while OpenMP provided easier implementation for small to medium-sized problems.**

I. Introduction of the problem and importance

A. Background and Motivation

Efficient matrix operations are fundamental in numerous computational fields, such as scientific computing, machine learning, and graphics processing. Matrix transposition, a simple yet computationally intensive operation for large datasets, provides a foundational benchmark for evaluating algorithmic and system performance. As data sizes grow, optimizing such operations becomes increasingly critical, particularly in the context of parallel and distributed computing.

B. Objectives of the project

This project focuses on the matrix transposition and symmetry problems, leveraging MPI parallelization technique. The primary goals include:

1) Implement a sequential matrix transposition to serve as a baseline.
2) Develop parallel implementations for both transposition and symmetry check using MPI.
3) Evaluate the performance of the parallel implementations in terms of speedup, efficiency, and scalability.
4) Compare MPI-based parallelization with OpenMP to highlight trade-offs between the two approaches like Ease to Use (OMP) vs Scalability (MPI) or Control (MPI) vs. Abstraction (OMP) and Performance vs. Complexity.

By benchmarking and comparing these approaches, this project aims to provide insights into the advantages and disadvantages of parallelization strategies like MPI in respect to OpenMP for a fundamental computational problem.

II. State-of-the-Art

[1] This research addresses the challenge of achieving scalability in exascale HPC by overlapping communication and computation. Traditional MPI implementations use dedicated progress threads, which can interfere with application tasks.

This research proposes a novel approach that leverages idle time within application threads to perform MPI communication, improving performance while minimizing disruption. This is achieved by integrating MPI with OpenMP using the OpenMP Tools interface.

[2] MPI.NET for parallel SQL query processing.

A. Gap Addressed by This Project

This project compare sequential and explicit parallelization techniques for matrix transposition using CPUs. By benchmarking these implementations across a range of matrix sizes and number of processors, analyzing key performance metrics, the study aims to identify the trade-offs and best practices for optimizing memory-bound operations. Additionally, the project investigates how system-specific features, such as compiler pragmas and MPI algorithms, influence performance, providing a more comprehensive understanding of parallelization strategies and advantages.

III. Contribution and Methodology

This project investigates the matrix transposition and symmetry check problem through three distinct implementations: sequential, explicit parallelization with OpenMP and parallelization with MPI.

A. Sequential Implementation

The importance of the sequential implementation is related to the confront and extrapolation of the computation time with 1 processor and use that results to evaluate the speedup, efficiency and scalability of my MPI implementation.

- Matrix Initialization: A random $N \times N$ matrix $matrix$ of floating-point numbers is generated.
- Symmetry Check: A function **checkSym** verifies if $matrix$ is symmetric by comparing $matrix[i*N+j]! = matrix[j*N+i]$).
- Matrix Transposition: A function **matTranspose** computes the transpose of $matrix$ and stores the result in a separate matrix $transposed\_matrix$.

- Performance Measurement: Execution times for **checkSym** and **matTranspose** are measured using $MPI\_Wtime()$. The matrix size N is provided as an input parameter.

```
// Check symmetry no Optimization
bool checkSym(float* M, int N) {
int isSym = true;
for (int i = 0; i < N; i++) {
    for (int j = 0; j < i; j++) {
        if (M[i*N+j] != M[j* N+i]){
    isSym = false;
}
return isSym;

// Transpose for no Optimization
void matTranspose(float* M , float* T , int N){
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                T[j*N+i] = M[i*N+j];
```

B. Explicit Parallelization with OpenMP

I used the same pragma already tested from the first project (IntroPARCO-2024-H1) changing the logic like in the serial. Development of a thread-level parallel solution using OpenMP, employing techniques such as work-sharing.

```
#pragma omp parallel for collapse(n)
```

- The collapse(n) directive tells the compiler to merge n nested loops into a single loop for parallel execution.

```
#pragma omp parallel for shared(variables)
```

- The shared clause defines variables that are shared among all threads in a parallel region. This means that all threads can access the same memory locations for those variables.

```
#pragma omp parallel for num_threads(variable)
```

- The num_threads directive specifies the number of threads that should be used for a particular parallel region.

C. Parallel Implementation with MPI

The MPI-based implementation extends the sequential version. The code primarily utilizes MPI_Bcast and MPI_Gather communication patterns within the MPI framework. While not explicitly using MPI_Scatter, the data distribution for the symmetry check can be considered an implicit form of scatter. These communication operations are fundamental to the parallel execution of the matrix symmetry check and transposition algorithms.

1) Parallel Symmetry Check:

- Domain Decomposition: Dividing the matrix into chunks (rows in this case) and assigning each chunk to a different process.
- Data Communication: Using MPI_Bcast to distribute the matrix and MPI_Gather to collect results.
- Parallel Efficiency: Calculating performance metrics like speedup and efficiency and broadcast to assess the effectiveness of the parallelization.

2) Parallel Matrix Transposition:

- Task Decomposition: The matrix is divided into sub matrices, each assigned to a different MPI process.
- Data Communication: Using MPI_Bcast to distribute the matrix and MPI_Gather to collect results.
- Parallel Efficiency: Calculating performance metrics like speedup and efficiency and broadcast to assess the effectiveness of the parallelization.

```
// Broadcast the data to the different processors
MPI_Bcast(...);
// Gather the transposed data in the correct order
for (int i = 0; i < N; i++) {
int MPI_Gather (...)
}
```

3) Performance Optimization:

- Communication Overlap: Computation is overlapped with communication to reduce idle time.
- Load Balancing: Tasks are distributed to minimize imbalance among processes.

IV. Performance Analysis:

Ideal Speedup: $S(_p) = p$

Speedup: $S = \frac{T_1}{T_p}$

Efficiency: $E = \frac{S(_p)}{p} * 100$

Ideal Scalability: $S(_w) = 1$

Scalability (Weak Scaling): $S(_w) = \frac{T_1(N)}{T_p(N*p)}$

Bandwidth: $B = \frac{DataTransferred}{TimeTaken}$

Scalability: $B = \frac{4096*4096*4*2}{TimeTaken}$

V. Experiments and System Descriptions

A. Computing Platform and System Description

Processor: Intel(R) Core(TM) i7-8565U CPU 1.80Hz, 4 cores, 8 logical processors.
RAM: 8GB DDR4.
Operating System: Windows 11
Compiler: GCC 13.2.0, G++ 13.2.0 (MinGW Compiler)
However the code was executed entirely on the cluster through interactive sections, due to an inability to run on my system.
Cluster Compiler: GCC 9.1.0 on the cluster with MPI support.
Libraries: $< omp.h >$ for OpenMP compilation, $< mpi.h >$ for MPI compilation and and standard C libraries.

B. Experiment

The experiments for this project were conducted on a computing platform that includes a multi-core processor, sufficient memory for large matrix manipulations, and an operating system that supports both OpenMP and

MPI parallelization frameworks. The processor is capable of handling multi-threaded operations, and the memory configuration was chosen to accommodate the matrix sizes used in the tests, which ranged from smaller matrices (e.g., 16x16) to larger ones (e.g., 2048x2048).

Using a 1D initialization for your matrix in the serial, OMP and MPI implementation, while logically representing a 2D matrix, the underlying storage is a contiguous block of memory (1D array). The project involved three parallelization techniques: serial execution, OpenMP parallelization, and MPI parallelization with different matrix sizes that goes from 2^4 to 2^{12}. The serial implementation used a simple two-loop structure to transpose the matrix. This served as the baseline for performance comparison. The OpenMP version was parallelized by dividing the matrix rows across available threads, which was controlled by the OpenMP runtime environment based on the number of cores in the system. The MPI implementation, on the other hand, split the matrix into smaller chunks that were distributed across multiple processes from 1 to confront to 32 running either on a single machine or across a cluster of machines. Each process performed its portion of the transposition, with inter-process communication used to exchange necessary rows of the matrix.

To ensure fair testing, the same matrix sizes 2^{12} were used for each parallelization method. The main goal was to compare the execution time, speedup, efficiency, bandwidth and scalability of the OpenMP and MPI approaches. The experimental setup involved varying matrix sizes to test the scalability of the methods. For OpenMP, the number of threads was adjusted to determine how performance improved with larger matrix. In the MPI setup, the matrix was distributed across different numbers of processes to test how well the method scaled when more processes were used, both on a single machine and in a distributed environment.

VI. Results and Discussion

Execution Time:

- Table 1 shows the execution times (in seconds) for the sequential, OpenMP and MPI implementations across different matrix sizes with 1 processor. Highlighting the similarities in the times, despite the different architectures the execution time remains similar.

OpenMP Symmetry check and Transposition comparison:

- Table 2 and Table 3 presents the computation time, the bandwidth, the speedup and the efficiency comparing the one obtained from the Symmetry check respect to the Transposition for the matrix size 4096.

MPI Symmetry check and Transposition comparison:

- Table 4 and Table 5 presents the computation time, the bandwidth, the speedup and the efficiency comparing the one obtained from the Symmetry check and Transposition for the matrix size 4096.
- Plots 1, 2, 3, 4 shows efficiency, speedup, bandwidth and weak scaling.
  As we can notice the speedup is much better for the MPI for the symmetry check and also for the transposition respect to the OMP, with the gap growing with bigger matrix sizes.
  Efficiency have some strange but still expected results for the two operations but with greater sizes everything become better.
  As the number of processes increases, the execution time remains nearly constant up to a certain number of processes (indicating good weak scalability).

A. Analysis and Interpretation

The experimental design was structured to test several hypotheses regarding the scalability and performance of these parallelization methods. The first hypothesis was that OpenMP would offer significant performance improvements over the serial implementation for moderate matrix sizes, but would eventually hit scalability limits as threading overhead became substantial. The second hypothesis suggested that MPI would perform better with larger matrices, especially when distributed over multiple machines, though the communication overhead could limit its effectiveness for smaller matrices. The results of these experiments were aimed at determining how each parallelization technique scaled with increasing problem size and to identify the break-even point where the overhead of parallelization might outweigh its benefits.

N	**checkSym (s)**	**matTranspose (s)**	**S_OpenMP (s)**	**T_OpenMP (s)**	**S_MPI (s)**	**T_MPI (s)**
16	0.000001	0.000001	0.000002	0.000001	0.000001	0.000001
32	0.000002	0.000002	0.000003	0.000003	0.000003	0.000004
64	0.000006	0.00001	0.000008	0.000012	0.000012	0.000016
128	0.000024	0.000042	0.000028	0.000052	0.00005	0.000063
256	0.0001	0.000361	0.000104	0.000431	0.000184	0.000251
521	0.000494	0.00275	0.000517	0.002755	0.001228	0.002495
1024	0.002956	0.017567	0.00307	0.017533	0.00593	0.017276
2048	0.012842	0.068775	0.014065	0.070773	0.028629	0.071069
4096	0.137368	0.449684	0.144415	0.420532	0.294243	0.395946

TABLE I
Execution times of symmetric check and matrix transpose operations with SERIAL, OPENMP AND MPI with 1 processor for varying matrix sizes.

threads	**Compu (s)**	**Bandwidth (GB/s)**	**Speedup (s)**	**Efficiency (%)**
1	0.144415	0.9294	0.951203	95.12
2	0.137251	0.9779	1.000851	50.04
4	0.071478	1.8777	1.921808	48.05
8	0.061378	2.1867	2.238047	27.98
16	0.04369	3.0721	3.144179	19.65
32	0.036131	3.7147	3.801902	11.88
64	0.034089	3.9373	4.029681	6.3

TABLE II
SYM OPENMP 4096. Execution times for Sym_OpenMP, Bandwidth, Speedup and Efficiency across different thread counts.

threads	Compu (s)	bandwidth (GB/s)	Speedup (s)	Efficiency (%)
1	0.420532	0.3192	1.069323	106.93
2	0.262035	0.5122	1.716126	85.81
4	0.120834	1.1108	3.721495	93.04
8	0.092083	1.4576	4.883479	61.04
16	0.035751	3.7542	12.578177	78.61
32	0.020825	6.445	21.593431	67.48
64	0.010151	13.2223	44.30001	69.22

TABLE III

TRAN OPENMP 4096. EXECUTION TIMES FOR TRAN_OPENMP, BANDWIDTH, SPEEDUP AND EFFICIENCY ACROSS DIFFERENT THREAD COUNTS.

N	Processors	Compu (s)	Speedup (s)	Efficiency (%)	Bandwidth (GB/s)
4096	1	0.294243	0.483749	48.37	0.4561
4096	2	0.12856	1.769448	88.47	1.044
4096	4	0.053188	2.453503	61.34	2.5234
4096	8	0.025836	4.960101	62	5.1949
4096	16	0.012026	12.200691	76.25	11.1604
4096	32	0.005912	26.089078	81.53	22.7031

TABLE IV

MPI SYMM. EXECUTION TIMES FOR SYMM_MPI, SPEEDUP, EFFICIENCY AND BANDWIDTH ACROSS DIFFERENT NUMBER OF PROCESSORS.

N	Processors	Compu (s)	Speedup (s)	Efficiency (%)	Bandwidth (GB/s)
4096	1	0.395946	1.273657	127.37	0.339
4096	2	0.181767	2.66485	133.24	0.7384
4096	4	0.097089	4.978223	124.46	1.3824
4096	8	0.029088	16.409296	205.12	4.6142
4096	16	0.009144	57.718309	360.74	14.6785
4096	32	0.003692	135.71389	424.11	36.3505

TABLE V

MPI TRAN. EXECUTION TIMES FOR TRAN_MPI, SPEEDUP, EFFICIENCY AND BANDWIDTH ACROSS DIFFERENT NUMBER OF PROCESSORS.

N	Processors	S_weak_scaling (s)	T_weak_scaling (s)
16	1	0.000001	0.000001
32	2	0.000002	0.000002
64	4	0.000003	0.000004
128	8	0.000007	0.000008
256	16	0.000012	0.000017
512	32	0.000039	0.000033

TABLE VI

MPI WEAK SCALING. EXECUTION TIMES FOR WEAK SCALING_MPI.

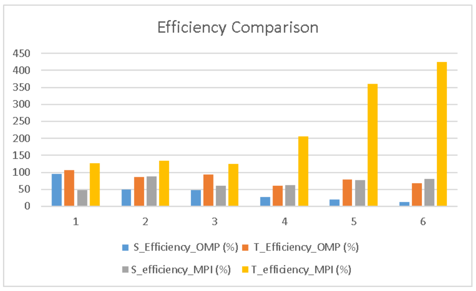

Fig. 1. Efficiency Comparison

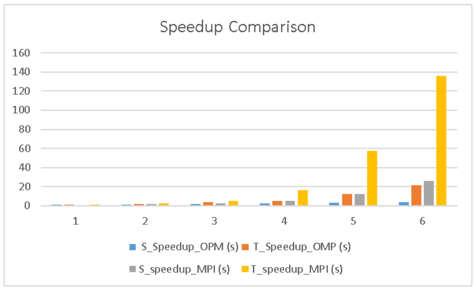

Fig. 2. Speedup Comparison

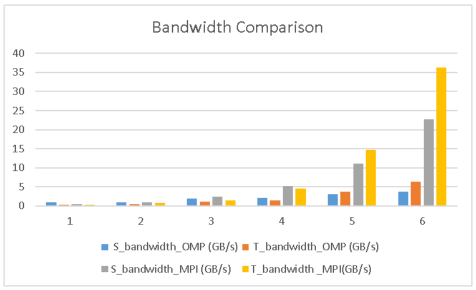

Fig. 3. Bandwidth Comparison

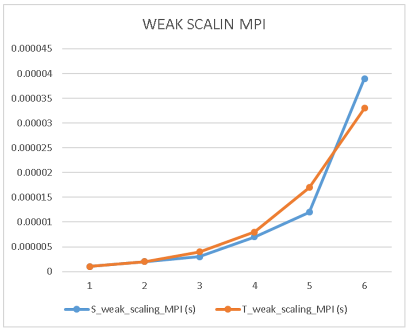

Fig. 4. WEAK SCALIN MPI

B. *Comparison with State-of-the-Art*

The state-of-the-art research emphasizes overlapping communication and computation in HPC by leveraging idle time within application threads and utilizing OpenMP integration for improved performance. This contrasts with my code which primarily uses basic MPI communication patterns without these optimizations, to be able to confront them separately.

C. *Key Insights*

This project compared sequential, OpenMP, and MPI approaches for matrix transposition, highlighting their performance and scalability. MPI showed superior scalability for large matrices, while OpenMP excelled for smaller problems due to lower overhead. Key bottlenecks, including communication latency and thread synchronization, were identified, with optimizations suggested for improvement. The findings emphasize MPI's strength in distributed systems and OpenMP's efficiency in shared-memory environments, offering valuable insights for optimizing parallel memory-bound operations.

References

[1] M. Sergent, M. Dagrada, P. Carribault, J. Jaeger, M. Pérache, and G. Papauré, "Efficient communication/computation overlap with mpi+openmp runtimes collaboration," in *Euro-Par 2018: Parallel Processing*, M. Aldinucci, L. Padovani, and M. Torquati, Eds. Cham: Springer International Publishing, 2018, pp. 560–572.

[2] N. A. Sabirov and R. F. Gibadullin, "Parallel processing of sql queries using mpi.net," in *2024 International Russian Smart Industry Conference (SmartIndustryCon)*, 2024, pp. 344–349.