Introduction to Parallel Computing. Homework 2: Parallelizing matrix operations using MPI

Riccardo Bassan
Università di Trento
238793
riccardo.bassan@studenti.unitn.it

***Abstract*—This project aims to explore explicit parallelization techniques using Message Passing Interface (MPI). The main objectives are to implement and analyze the performance of a parallel matrix transposition and compare them with sequential and OpenMP implementations studying the program's bandwidth, speedup, efficiency, and scalability. Experiments are performed for varying matrix sizes and different numbers of processes.**

I. Introduction

The transposition of a matrix is a frequently used operation in computational tasks, and in many scientific and engineering applications, obtaining efficient performances in a parallel context could be very useful. Still, because of synchronization and data partition management, it can be challenging to do.
In this assignment, I explored Message Passing Interface (MPI) which is a standard library for message passing in distributed-memory systems. It is portable and scalable and offers efficiency for high-performance computers like clusters, where communication speed between nodes is fundamental.

II. State-of-the-art

Parallelizing matrix operations is a well-explored area in the field of high-performance computing (HPC), and achieving good performance in parallel implementations is a critical research topic.
There are two models of parallelism used in this context:

- Shared memory: allows multiple threads to work on the same memory space, it is effective for multi-core systems but its scalability is limited by the number of available cores in a single node.
- Distributed memory: it uses MPI and requires explicit communication between processes that do not share memory, message passing makes MPI really powerful for scaling across multiple nodes in a cluster, but it is more difficult to implement for a developer that must manage explicitly this communication.

Current research:

- Block-Based Decomposition: divides the matrix into smaller blocks reducing communication overhead, each process handles a subset of blocks minimizing data exchange.
- Hybrid Models: combines MPI with shared memory techniques like OpenMP (that I have studied in the report done for the first deliverable), in this model, MPI is used for inter-node communication, while OpenMP manages intra-node parallelism.
- Communication-Avoiding Algorithms: aim to minimize data exchanged between processes overlapping communication and computation.

This project aims to use these models to implement matrix transposition using MPI evaluating performances to find the best implementation.

III. Methodology

A. Sequential implementation

1) Symmetry check: As explained in the first deliverable report, this function verifies if all data are equal to their corresponding on the opposite side of the main diagonal by scrolling through the given matrix.

2) Matrix transposition: It swaps rows and columns of the given matrix and creates a transposed one as described in the previous report.

B. Parallel implementation with MPI

For every implementation of an MPI function, I created two values to save the identifier of each process (rank) and the total number of processes (size) using the MPI functions MPI_Comm_rank and MPI_Comm_size.
Then, I did a work distribution that divides the matrix rows among processes, each process is assigned $local\_rows = n/size$ rows that start from $start\_row = rank \times local\_rows$ and end at $end\_row = start\_row + local\_rows$.

1) Symmetry check MPI: After the work distribution, each process checks the symmetry of its assigned rows using two nested loops like in the sequential implementation where a local variable is set to false if there is any mismatch between the two corresponding values that are compared.

```
//symmetry check
for(int i=0; i<local_rows; i++){
  for(int j=i+1; j<n; j++){
    if(matrix[i][j]!=matrix[j][i]){
      local_check = false;
```

```
    }
  }
}
```

The local results from all processes are then aggregated using

```
MPI_Reduce(&local_check, &global_check, 1,
    MPI_C_BOOL, MPI_LAND, 0, MPI_COMM_WORLD);
```

that performs a logical AND across all processes (MPI_LAND). I choose MPI_Reduce because this function distributes the workload in a balanced way, it can use a logical AND to combine data, and aggregation occurs in parallel during communication (differently from other approaches like MPI_Gather), reducing the total number of messages exchanged.
In the end, if the rank is equal to zero (root process) the global result is returned, and elapsed time is calculated.

2) Matrix transposition MPI: A local buffer is created on each process to store its assigned rows for transposition, the local transposition is made by scrolling through each process's local rows and storing the result in the local buffer.

```
//local transposition
for(int i=0; i<local_rows; i++){
  for(int j=0; j<n; j++){
    local_buffer[i*n+j] = matrix[j][start_row+i];
  }
}
```

After that, each process sends its portion of the transposed matrix to the root process that assembles the complete matrix in a buffer using the function

```
MPI_Gather(local_buffer,local_rows*n,MPI_FLOAT,
    transpose_buffer, local_rows*n, MPI_FLOAT,
    0, MPI_COMM_WORLD);
```

I used this function because it is more efficient compared to some manual implementations like MPI_Send and MPI_Recv, and it is more suitable for this type of work instead of other functions like MPI_Allgather that would have been inefficient or MPI_Reduce that is not designed for this type of problems. The transposed matrix that was gathered into a single-dimensional buffer is then saved into a bi-dimensional matrix that is returned to the main function. The functions also calculate bandwidth, speedup, efficiency, and scalability which are useful for studying its performance.

IV. Experiments and System Description

A. System description

As in the first deliverable, the code was executed in the UniTN HPC cluster that consists of:

- 2 head nodes
- computing nodes: 142 CPU for a total of 7674 nodes, 10 GPU for a total of 448128 CUDA nodes
- Ram of 65 TB
- Ethernet, Infiniband, and Omni-Path networks
- shared storage
- total theoretical peak performance of 478.1 TFLOPs (CPU: 422.7 TFLOPs, GPU: 55.4 TFLOPs)

B. Experiments

The objectives of the experiments were to evaluate the execution time of the matrix transposition function and measure its speedup, efficiency, bandwidth, and scalability. The experiments were carried out by submitting the job in the short_cpuQ queue with 1 node, 16 CPUs, 16 processes, and 1 GB of memory.
The code is written in C programming language using some libraries like:

- <mpi.h>: allows the use of MPI functions for the parallelization and calculation of time.
- <stdbool.h>: allows the use of boolean data type.

For every symmetry check and matrix transposition function, I used the MPI function MPI_Wtime() to save the time at which the function starts and ends to estimate its execution time, which is useful for computing performance metrics.
I ran the program several times using different numbers of processes and matrix sizes to make a comparison between them. For each matrix size, I created a code that checks if it is a power of two, and then a random matrix is generated using the rand() function.
To correctly initialize and finalize the execution of MPI processes I use the functions MPI_Init() and MPI_Finalize() between which symmetry check and matrix transposition implementations are called. Some functions I wrote to help me verify the correctness of my experiments are:

- checkTrans(): checks the correctness of a transposition.
- print(): prints a matrix given.
- makeSymmetric(): makes a matrix symmetric to check if the symmetry check works.

Furthermore, in the matTransposeMPI() function I wrote a code that saves all the metrics that are calculated into a .csv file, the file is opened in append mode so every time I run the code the values are added to the file, this helps me with the study of performance writing all the data I need to do graphs and tables.

V. Results and Discussion

TABLE I
Execution time

Size	*Sequential Time*
16x16	***0.0000021***
32x32	***0.0000071***
64x64	***0.0000286***
128x128	***0.0001158***
256x256	***0.0003726***
512x512	***0.0011475***
1024x1024	***0.0038411***
2048x2048	***0.0350339***
4096x4096	***0.1712646***

Table 1 shows the quadratic increase of the execution time for the matrix transposition function, while in graph 1 the time

for the parallel implementation decreases as the number of processes increases, demonstrating strong scaling. This reduction becomes less significant at a higher number of processes due to communication overhead. Comparing this with the OpenMP execution time graph discussed in the previous report we can notice that MPI outperforms OpenMP for larger matrices due to its distributed memory while for smaller ones OpenMP performs better thanks to the lower communication overhead of the shared memory model.

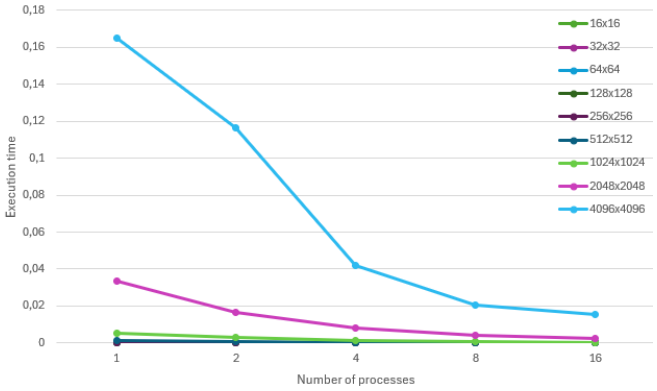

Fig. 1. Execution time

The speedup formula is:

$$Speedup = \frac{T_s}{T_p} \tag{1}$$

where T_s is the time of the sequential implementation and T_p is the execution time of the MPI parallelization. In Figure 2 we can see that it increases with the number of processes and for larger matrices it is almost linear indicating effective parallelization. Compared to OpenMP, MPI scales better with increasing processes for larger matrix sizes.

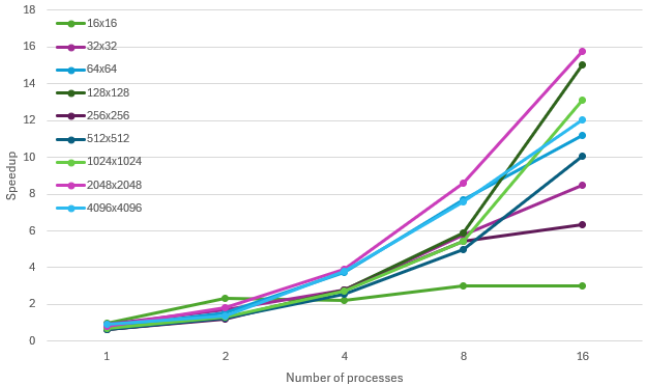

Fig. 2. Speedup

The graph in Figure 3 shows the bandwidth achieved for the matrix transposition operation across different matrix sizes and numbers of processes, its formula is:

$$Bandwidth = \frac{Data\ Transferred}{Time\ Taken} \tag{2}$$

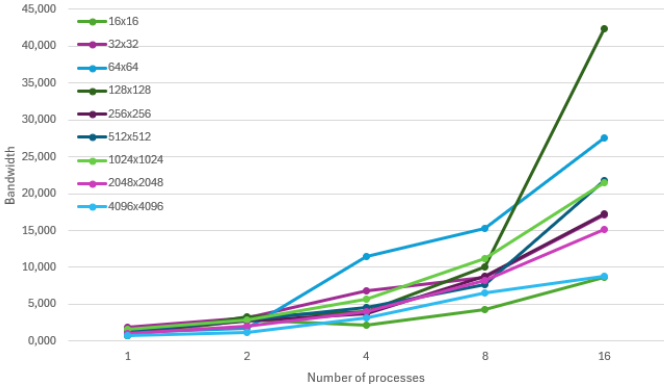

Fig. 3. Bandwidth

where $Data\ Transferred$ is $2 \times n \times n \times sizeof(float)$. We can notice that the bandwidth increases as the number of processors increases, reflecting the benefits of parallelization even if, for smaller matrices, this increase decreases due to communication costs.

size	1 proc	2 proc	4 proc	8 proc	16 proc
16x16	100%	100%	56%	38%	19%
32x32	94%	86%	70%	72%	53%
64x64	74%	78%	94%	96%	70%
128x128	67%	64%	70%	74%	94%
256x256	63%	61%	70%	68%	40%
512x512	67%	62%	64%	63%	63%
1024x1024	71%	65%	68%	68%	82%
2048x2048	79%	92%	98%	99%	98%
4096x4096	94%	69%	95%	95%	75%

TABLE II
EFFICIENCY

Efficiency is a measure of how effectively the processes are used as parallelism increases, it is calculated as:

$$Efficiency = \frac{Speedup}{N_p} \times 100 = \frac{T_s}{N_p T_p} \times 100 \tag{3}$$

where N_p is the number of processes. In Table II we see that for small matrices this value drops significantly as the number of processes increases, while for larger ones, a higher efficiency is maintained. Comparing these data with those collected in the previous report regarding OpenMP, we can see that OpenMP performs better for small matrix sizes as its shared-memory model avoids the communication overhead that is present in MPI. For larger matrices, instead, MPI scales better across nodes due to its distributed-memory architecture.

Table III represents scalability that evaluates how well the program can handle increasing problem sizes while proportionally increasing the number of processes. Its formula is:

$$Scalability = \frac{T_1(N)}{T_p(N \times p)} \tag{4}$$

where T_1 is the execution time on a single processor for a problem size N and T_p is the execution time on p processors for a problem size $p \times N$. In this case, it decreases for smaller matrices as the process count increases, and remains relatively high for larger matrices.

Size	1 proc	2 proc	4 proc	8 proc	16 proc
16x16	1.000000	1.000000	0.916084	0.199847	0.086526
32x32	1.000000	0.230645	0.129451	0.071691	0.04465029
64x64	1.000000	0.701279	0.194592	0.099745	0.07025545
128x128	1.000000	0.896076	0.380686	0.228903	0.07799425
256x256	1.000000	0.552384	0.271422	0.097723	0.02608116
512x512	1.000000	0.546526	0.191101	0.077488	
1024x1024	1.000000	0.320403	0.127759		
2048x2048	1.000000	0.287925			
4096x4096	1.000000				

TABLE III
SCALABILITY

VI. CONCLUSIONS

At the end of this project, we can say that MPI demonstrates a significant advantage for large matrices due to its distributed-memory model, which allows to effective exploit clusters nodes. It scales better for large matrices, due to the ability to distribute the computational load among distinct nodes without shared-memory limitation, however, for small matrices, scalability drops rapidly with increasing processor, due to the unfavorable ratio between communication and computation.

REFERENCES

[1] Pacheco, P., An Introduction to Parallel Programming, Morgan Kaufmann, 2011

[2] Dongarra, J., Van De Geijn, R., Reducing the Communication Overhead in Parallel Algorithms, Wiley, 2015

GIT repository link: https://github.com/RicBas03/Parallelizing-matrix-operations-using-MPI