Introduction to Parallel Computing
IntroPARCO 2024 H2: Matrix transposition

Nicolò Bellè
ID: 238178 - nicolo.belle@studenti.unitn.it

***Abstract*—The goal of this project is to evaluate the execution time for carrying out the transposition of a matrix by comparing sequential implementation and explicit parallelization using MPI library in order to find the most efficient for each one.**

I. Introduction

A. Background

Matrix transposition involves swapping the rows and columns of a matrix. In this process, each element in the matrix at position (i, j) is moved to position (j, i). This effectively "flips" the matrix along its diagonal, transforming rows into columns and columns into rows. While straightforward for small matrices, this operation becomes more complex and computationally demanding as the size of the matrix increases.

B. Importance of the problem

The goal of the project is to explore and evaluate different methods for optimizing the time it takes to perform matrix transposition by leveraging parallelization techniques (MPI in this case). By distributing the workload across multiple processes, these methods aim to handle larger matrices more efficiently and minimize computational delays.

II. State-of-the-art

Recent studies on matrix transposition have emphasized enhancing efficiency, given its essential role in advancing artificial intelligence, which has become increasingly prevalent in our daily lives.

The advancements focus on addressing computational challenges by implementing techniques for splitting operations into different processes that can perform their work in parallel, thus reducing latency when handling large-scale data.

In the following sections, we examine the current solutions to this problem, highlighting their strengths and identifying their limitations.

A. Parallelization with OpenMP

OpenMP is a standard programming model in C, C++ and Fortran that simplifies parallel programming for shared-memory architectures. It allows developers to parallelize code by dividing work across multiple threads with minimal code modifications.

However, for small workloads, the cost of thread creation and synchronization may outweigh performance gains. Moreover, improper synchronization can lead to race conditions and bottlenecks, so you have to be careful.

B. Parallelization with MPI

MPI is a standard library for parallel programming in C, C++, Fortran and more, focusing on distributed-memory systems which is favorable for its scalability and flexibility. Furthermore, this model offers fine-grained control over data sharing and synchronization.

Processes on distributed memory systems communicate via messages which introduce delays compared to shared memory.

The main disadvantages are that mpi requires explicit communication management and debugging can be challenging.

C. Gap to fill in this project

Matrix transposition in distributed memory systems involves exchanging data between processes located on different nodes. The project aims to optimize this communication to reduce bottlenecks and improve overall efficiency through the use of MPI library methods. In particular, by dividing the matrix into smaller ones, each process performs its assigned portion of the transposition independently. This parallelism ensures that all processes work simultaneously, maximizing resource utilization.

III. Contribution and methodology

The project is written using C programming language. The main file is called *deliverable2.c* and contains the following methods. (for loops of all *checksym* are not intentionally interrupted, even though *check* value is already set to false, to have a better wall-clock time comparison).

A. Input size of the matrix

There's two type of input function. The first one is *initMatrixSize* which implements the use of *scanf*, method contained within the library *stdio.h*. This method puts the program in a hold waiting for the user to enter the size of the matrix. Inside it, there are several controls on the typed value: this must be an integer greater than zero, therefore part of the range of integers and finally must be a power of two as required. If one of these points is not met, after a suitable warning, the input is requested again.

The previous function is commented on because the input is captured by another method called *checkInvalidInputAtoi*. It uses *atoi* method that gets the first parameter of the execute command *./deliverable2.out*. As before, the same controls have been implemented for this input but in this case, after the error message, the execution ends.

B. Allocation and deallocation of the matrix

Matrix has been dynamically allocated within *allocate_matrix* method by *malloc*:

```
float **Tc =(float**)malloc(r*sizeof(float*));
for (int i = 0; i < c; i++) {
    matrix[i]=(float*)malloc(c*sizeof(float));
}
```

and are subsequently deallocated with the *deallocate_Matrix* method:

```
for (int i = 0; i < r; i++) {
    free(matrix[i]);
}
free(matrix);
```

Moreover, the main matrix is initialized using the *rand()* function with float values from 0.0 to 10.0 as follows:

```
Mc[i][j] = ((float)rand() / RAND_MAX) * 10.0;
```

C. Sequential implementation

The *checksym* function has been implemented for sequential control of the symmetry of the matrix, which is equivalent to verify if the matrix is mirrored or not with respect to its diagonal. Pseudo code:

```
for (int i = 0; i < r; i++) {
    for (int j = i + 1; j < c; j++) {
        if (M[i][j] != M[j][i])
            check = false;
    }
}
```

In this method, as in the others *checkSym* explained below, a safety check is carried out in which the matrix must not be null, otherwise it leaves without performing any operation.

Instead, in the function *matTranspose* sequential matrix transposition is performed without any code optimization, so two nested for loops iterate all rows and columns starting from *i=0* and *j=0*. Pseudo code:

```
for (int i = 0; i < r; i++) {
    for (int j = 0; j < c; j++) {
        Tc[j][i] = Mc[i][j];
    }
}
```

where *Tc* is a matrix which will contain the result of the transposition and will be used as return for this method while *Mc* is the main matrix passed to the function as parameter.

The problem of this implementation arises when matrices reach large sizes because it becomes expensive to iterate and copy all values one after another in a sequential way.

D. MPI parallelization

By exploiting parallelization with MPI, in *checkSymMPI* and *matTransposeMPI* the efficiency has been further improved by assigning evenly some rows of the main matrix to each process. In this way, each process computes the same control as the sequential method with the difference that it does not have to scroll through the entire matrix but only the rows that are assigned to it.

Pseudo code:

```
int r_forProcess = rows / size;
int first_row_forProcess =rank * r_forProcess;
```

The power of this approach is that other processes can perform the same operation simultaneously without creating competition problems since they operate on different rows.

To perform *checkSymMPI*, in *rank==0*, also called root process, *check* variable is created and then for each process *local_check* variable is initialized. Both are initially set to 1, that is the matrix is assumed to be symmetrical. Then, each process sets the variable *local_check* to 0 if the local matrix is not symmetrical, otherwise the value is not changed. Implementing the following MPI method, the result of each individual process is synchronized in the variable *check*:

```
MPI_Reduce(&check_local, &check, 1, MPI_INT,
    MPI_LAND, 0, MPI_COMM_WORLD);
```

In function *matTransposeMPI*, the method which recomposes the local matrices transposed into the return matrix is as follows:

```
MPI_Gather(MPItransposedLocal, r_forProcess*c,
     MPI_FLOAT, transposed_contiguous_matrix,
    r_forProcess*c, MPI_FLOAT, 0,
    MPI_COMM_WORLD);
```

Since different processes execute the code, clause *rank==0* was used that allows only that specific process to run that part of the code to avoid for example *printf* repeated several times.

E. File .pbs

The file *deliverable2.pbs* is used to submit the job in the cluster specifying some parameters like the number of processes. Inside it you will find the following line needed to compile the *C* file:

```
mpicc deliverable2.c -o deliverable2.out
```

the code is then executed by changing the size of the matrix as required and setting the number of processes to be used. Pseudo code:

```
mpirun -np $nProcesses ./deliverable2.out 4096
```

where *$nProcesses* sets the number of processes.

IV. Experiments and System Description

A. Description of computing system

The simulations were performed by connecting via GlobalProtect vpn to the high-performance computing cluster, which is a collection of interconnected computers, called nodes, that work together as a single system to perform computationally intensive tasks. It's based on PBS queue management system. It's composed of 65 TB of ram and the computing nodes have 142 CPU (for a total of 7674 cores) and 10 GPU (for a total of 48128 CUDA cores).

Total theoretical peak performance: 478.1 TFLOPs

Theoretical peak performance CPU: 422.7 TFLOPs

B. Methodologies and libraries used in the project

To run the simulations a job has been started in the short_cpuQ queue specifying the following parameters:

```
-l select=1:ncpus=4:mpiprocs=16:mem=1gb
```

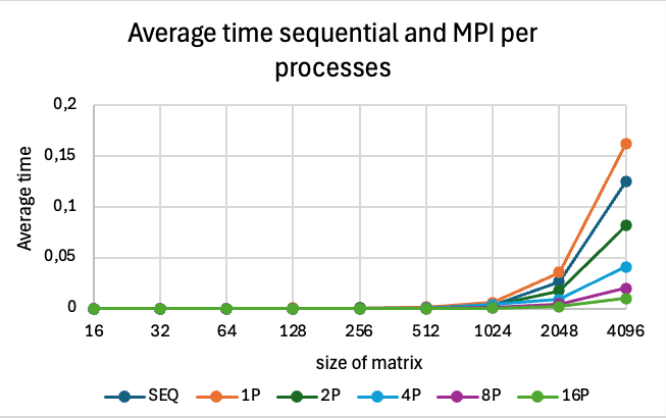
In the project it was thought to have each transposition and symmetry control function performed 10 times, so as to calculate the average time taken by each one and obtain a more accuracy result. The wall clock time function was used for calculating the time taken to transpose and check the symmetry of the matrix as required. The values were later saved in a *.csv* file for comparison between the different implementations. In addition, to analyze the MPI implementation more thoroughly were calculated speed up, efficiency and bandwidth for each thread and size of the matrix. This implementation was performed by 8 and 16 processes and each one with different size of matrix from 2^4 to 2^{12}. The libraries included in the *deliverable2.c* file are:

```
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <time.h>
#include <ctype.h>
#include <string.h>
#include <errno.h>
#include <limits.h>
#include <mpi.h>
```

V. Result and Discussion

TABLE I
AVERAGE TIME SEQUENTIAL AND MPI IMPLEMENTATION

SIZE_n	AVG_TIME_SEQ	1 PROC	2 PROC	4 PROC	8 PROC	16 PROC
16	0,000001478	0,000001669	0,000001192	0,000000954	0,000000477	0,000000238
32	0,000008512	0,000009537	0,000003576	0,000002623	0,000000954	0,000000238
64	0,000020814	0,000030279	0,000014067	0,000005245	0,000004292	0,000001907
128	0,000134492	0,000209808	0,00006175	0,000027418	0,000008821	0,000005484
256	0,000357103	0,000610828	0,000217676	0,000082016	0,000034332	0,000024557
512	0,000812221	0,001461267	0,00070405	0,000365734	0,000184298	0,00009203
1024	0,003279567	0,005919456	0,002969027	0,004046202	0,000747204	0,000375509
2048	0,026423788	0,035335541	0,017383099	0,008948803	0,004462719	0,002162218
4096	0,124832082	0,161854267	0,081532478	0,04108119	0,020124435	0,010127783

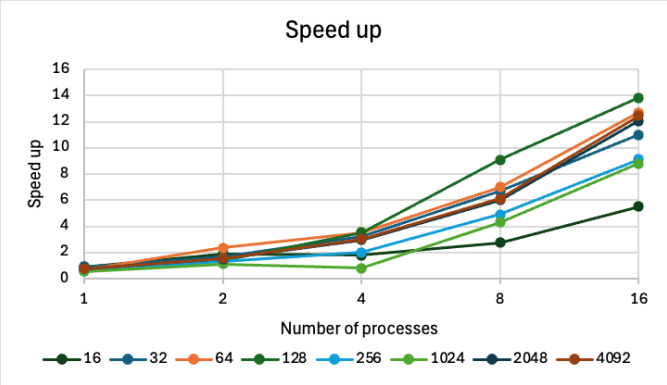


A. Table 1

In this graph is represented the average time taken by each process to carry out the matrix transposition for each size (in green) compared to the average time taken for sequential (in blue). You can see how efficient parallelization is when the size exceeds 1024.

TABLE II
SPEED UP

SIZE_n	1 PROC	2 PROC	4 PROC	8 PROC	16 PROC
16	0,89	1,88	1,80	2,75	5,50
32	0,89	1,66	3,23	6,67	11,00
64	0,69	2,36	3,51	6,99	12,68
128	0,64	1,38	3,51	9,07	13,83
256	0,58	1,31	1,99	4,91	9,13
512	0,56	1,09	1,97	3,79	7,72
1024	0,55	1,10	0,80	4,31	8,80
2048	0,75	1,51	2,97	6,02	12,07
4096	0,77	1,54	3,02	6,12	12,43



B. Table 2

Speed up has been calculated so that we have a data that measures how much the execution of a parallel application changes by using processes. This was done using the following formula:

$$SpeedUp = \frac{Ts}{Tp} \tag{1}$$

where *Ts* is the time of sequential implementation and Tp is the time of MPI parallelization.

TABLE III
EFFICIENCY

SIZE_n	1 PROC	2 PROC	4 PROC	8 PROC	16 PROC
16	88,57	94,00	45,00	34,38	34,38
32	89,25	83,00	80,68	83,44	68,75
64	68,74	89,19	87,73	87,43	79,22
128	64,10	69,11	87,83	113,38	86,44
256	58,46	65,28	49,77	61,36	57,09
512	55,58	54,49	49,29	47,32	48,24
1024	55,40	55,12	19,92	53,81	55,02
2048	74,78	75,52	74,20	75,31	75,43
4096	77,13	76,78	75,47	76,44	77,69

C. Table 3

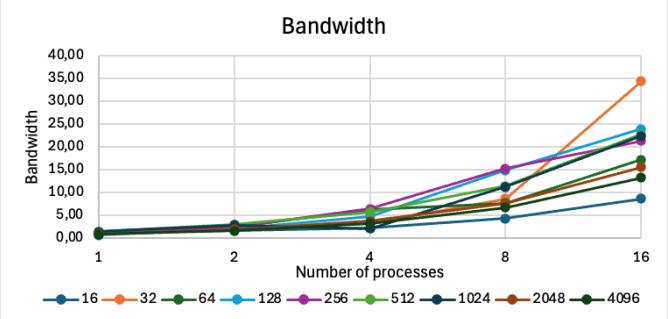
Efficiency has been calculated to know how many computational resources are actually used to perform the parallel transposition. This was done using the following formula:

$$Efficiency = \frac{SpeedUp}{Np} \times 100$$

where *SpeedUp* is the value computed before and *Np* is the number of processes.

TABLE IV
BANDWIDTH

SIZE_n	1 PROC	2 PROC	4 PROC	8 PROC	16 PROC
16	1,227	1,718	2,147	4,295	8,590
32	0,859	2,291	3,124	8,590	34,360
64	1,082	2,329	6,247	7,635	17,180
128	0,625	2,123	4,780	14,858	23,902
256	0,858	2,409	6,393	15,271	21,350
512	1,435	2,979	5,734	11,379	22,788
1024	1,417	2,825	2,073	11,227	22,339
2048	0,950	1,930	3,750	7,519	15,519
4096	0,829	1,646	3,267	6,669	13,252



D. Table 4

Bandwidth describe for each matrix size his effective data transfer rate from memory to CPU achieved during real operations and the comparison is done for each processes using the following formula:

$$Bandwidth = \frac{DataTransfered}{Tp}$$

where

$$DataTransfered = n \times n \times n \times size\_Of\_Float$$

TABLE V
WEAK SCALING

SIZE_n	1	2	4	8	16
16	0,885560216	0,62667785	0,32735939	0,14862261	0,05338600
32	0,892523854	0,42205161	0,30870231	0,18542468	0,02850158
64	0,687407114	0,53823482	0,22441963	0,16287209	0,06438195
128	0,641024174	0,39211489	0,26336354	0,10708321	0,03507556
256	0,584621203	0,40365741	0,04035117	0,03776577	0,02214976
512	0,555833397	0,25843584	0,08058564	0,03466850	
1024	0,554031823	0,18828283	0,07847390		
2048	0,747796333	0,32200443			
4096	0,771262224				

Weak scaling measures how the runtime of a parallel application is affected when both the number of processors or threads and the size of the problem increase proportionally. The formula for calculating it is:

$$Scalability = \frac{T1(N)}{Tp(size \times Np)}$$

where *T1* is the execution time on a single processor for a problem size *N*, *Tp* is the execution time on *p* processors for a problem size *p* * *N* and *p* is the number of processors.

CONCLUSION

MPI parallelization has proved to be a good method for greatly improving the efficiency of matrix transposition. The most difficult obstacles to overcome is to ensure, in addition to parallelism, also the synchronization and consistency of the operations that are performed. To solve this problem, the use of MPI was effective through the methods: *MPI_Reduce* and *MPI_Gather*.

REFERENCES

[1] "wiki.info", MPI library. https://hpc-wiki.info/hpc/MPI
[2] "rivier.edu", Matrix transposition. https://www2.rivier.edu/journal/ROAJ-Fall-2023/J1274_Malita%20and_Stefan_Fall-2023.pdf
[3] "Università degli studi di Bologna". https://typeset.io/pdf/universita-degli-studi-di-bologna-24t2zvdp7p.pdf

GITHUB

GIT repository Nicolò Bellè