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

[1]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

[2]This project focuses on the matrix transposition and sym- metry problems, leveraging MPI parallelization technique. The primary goals include:

- Implement a sequential matrix transposition to serve as a baseline.
- Develop parallel implementations for both transposition and symmetry check using MPI.
- Evaluate the performance of the parallel implementations in terms of speedup, efficiency, and scalability.
- Compare MPI parallelization with OpenMP to highlight trade-offs between the two approaches like ease to use (OMP) vs scalability (MPI) and performance com- plexity.

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

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

[3]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

[4]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

[4]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. 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.

B. 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.

C. 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 96 CPU(s) and 24 core per socket with a total amount of 4 sockets. (each core with 1 thread). The model of CPU is Intel(R) Xeon(R) Gold 6252N CPU @ 2.30GHz.

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=64:mpiprocs=64: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}.

V. Result and discussion

Table and Graphics

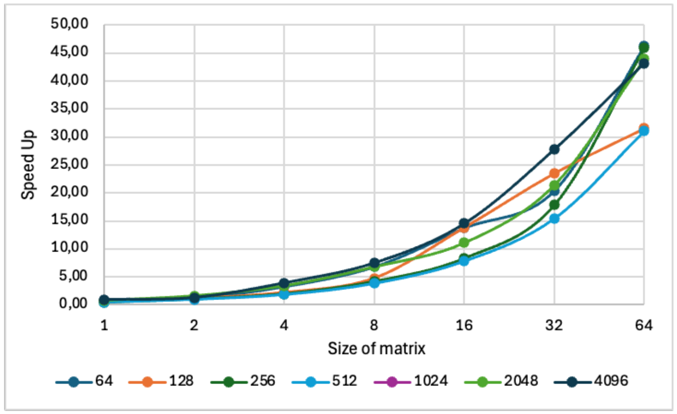
TABLE I
AVERAGE TIME SEQUENTIAL AND MPI IMPLEMENTATION

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 table 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



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:

$$SpeedUp = \frac{Ts}{Tp} \qquad (1)$$

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

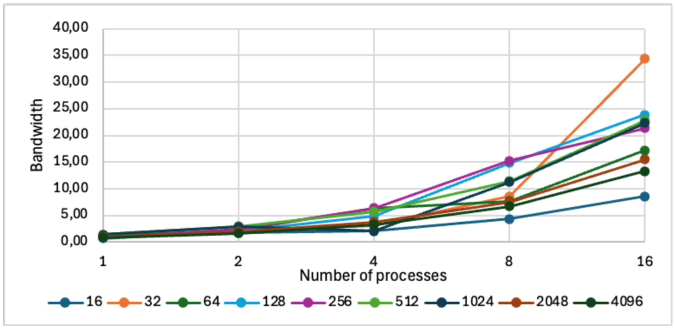
Efficiency has been calculated to know how many computational resources are actually used to perform the parallel transposition. This was done using the following formula:

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

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

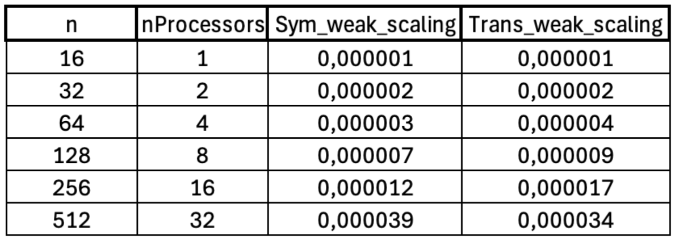
Bandwidth describe for each matrix size his effective data transfer rate from memory to CPU achieved during real operations and the comparison is done for each processes using the following formula:

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

where

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

TABLE V
WEAK SCALING



n	nProcessors	Sym_weak_scaling	Trans_weak_scaling
16	1	0,000001	0,000001
32	2	0,000002	0,000002
64	4	0,000003	0,000004
128	8	0,000007	0,000009
256	16	0,000012	0,000017
512	32	0,000039	0,000034

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.

DIFFERENCES BETWEEN APPROACHES

Sequential programs cannot take advantage of modern multi-core and multi-node architectures. Today's processors have multiple cores, and supercomputers have thousands of processors. A sequential approach wastes these resources, while parallel programming distributes the workload, improving efficiency.

OpenMP is designed for shared-memory systems, meaning that all threads operate within the same memory space. It is relatively easy to use since it relies on compiler directives (e.g., #pragma omp) to parallelize loops and sections of code. However, its scalability is limited because it depends on the memory bandwidth of a single machine. This makes OpenMP ideal for multi-threaded applications running on a single multi-core CPU but unsuitable for large-scale distributed computing.

On the other hand, MPI (Message Passing Interface) is specifically designed for distributed-memory systems, where each process runs independently and communicates with others through message passing. This model allows applications to scale efficiently across multiple machines in a cluster or supercomputer. However, it requires more effort to implement since programmers must explicitly manage data exchange between processes using functions like MPI_Send and MPI_Recv. The added complexity is justified by its ability to handle large-scale parallel workloads, making it the standard for high-performance computing (HPC).

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", Matrix Transposition. https://en.wikipedia.org/wiki/Transpose
[2] "sciencedirect.com", Importance of matrix transposition. https://www.sciencedirect.com/topics/computerscience/matrix-transposition
[3] "OpenMP library". https://en.wikipedia.org/wiki/OpenMP
[4] "MPI library". https://www.chpc.utah.edu/documentation/software/mpilibraries.php
[5] "Distributed memory system". https://en.wikipedia.org/wiki/Distributed_memory
[6] Pacheco, P., An Introduction to Parallel Programming, Morgan Kaufmann, 2011
[7] Dongarra, J., Van De Geijn, R., Reducing the Communication Overhead in Parallel Algorithms, Wiley, 2015

GITHUB

GIT repository Nicolò Bellè