Homework 2: Parallelizing matrix operations using MPI.

Dimitri Corraini
237963
dimitri.corraini@studenti.unitn.it

***Abstract*—This secondo project explores distributed parallelization techniques by implementing a matrix transpose operation and a symmetry verification. We will focus our analysis on square matrices of size $n \times n$ with n ranging from 2^4 to 2^{12}.**

I. Introduction and Background

A. Matrix Transposition and Its Importance

Matrix transposition is a fundamental operation that plays a critical role in various fields, including machine learning, computational fluid dynamics and computer graphics. A clear example of its application is in problems involving large-scale multigraphs and high-cardinality sparse matrices [2]

B. Importance of Matrix Symmetry

Verifying matrix symmetry is as important, if not more, than transposition. This is due to the fact that if a matrix A is symmetric, its transpose will follow $A^T = A$, which saves valuable computation time by eliminating the need to swap values along the diagonal.

C. Project Objectives

The primary objectives of this project are to develop an efficient and robust foundation for distributed matrix transposition, designed to scale when integrated with more computationally intensive operations.

D. Project Repository

All the code and additional resources can be found on GitHub: Project Repository.

II. State-of-the-Art

The state-of-the-art approaches for matrix transpositions involve offloading computations to GPUs using CUDA [1] and distributing the workload across multiple nodes using MPI. An example of distributed matrix transposition is demonstrated in [2], where MPI is utilized to distribute the ranks of large-scale multigraphs and high-cardinality sparse matrices.

III. Contribution and Methodolgy

To tackle this problem, the matrix was split into a number of rows evenly divisible by the number of processes. The splitting was done logically, meaning that each process received the entire matrix M. This was accomplished using the MPI_Bcast command in MPI.

While this approach was chosen for simplicity, it is important to note that for larger matrices, a more efficient solution would involve distributing individual rows or submatrices. This would significantly reduce the amount of data transferred between processes or nodes. Looking at the pseudocode, we can see that each process is responsible for a specific range of rows. The transposition is performed by taking the columns of the matrix and transposing them into the rows handled by the corresponding process. This approach ensures that when the results are gathered, each process contributes an array composed of the rows of the transposed matrix. This simplifies the task of combining the results into the final transposed matrix.

Matrix Transpose with MPI

Input: 1D array M of size $N \times N$ (row-major order)
Output: Transposed 1D array T of size $N \times N$
Initialize:
Number of processes, P
Current process rank, rank
Rows per process, rows_per_process $= N/P$
Broadcast matrix M to all processes
Compute row range for current process:
start $=$ rank $\times$ rows_per_process
stop $=$ (rank $+ 1$) $\times$ rows_per_process
Allocate buffer b of size rows_per_process $\times N$
for each i from start to stop $- 1$ **do**
  for each j from 0 to $N - 1$ **do**
    $b[(i - \text{start}) \times N + j] = M[j \times N + i]$
  end for
end for
Gather b from all processes into matrix T on root process

A similar approach is employed for the symmetry check, where the entire matrix is broadcasted to all processes. This broadcasting is essential because each process needs simultaneous access to the rows and columns of the matrix to verify its symmetry. Each process then compares the values of the rows assigned to it with their corresponding column counterparts. If the difference between every corresponding cell is smaller than 10^{-5}, the group is considered symmetric. The results are then gathered using the MPI_Reduce function with the MPI logical AND reduction operator.

Matrix Symmetry with MPI

Input: 1D array M of size $N \times N$ (row-major order)
Output: Boolean symmetry, indicating if M is symmetric
Broadcast the matrix M to all processes
Initialize local_symmetry as `true`

Compute row range for the current process:
start = rank × rows_per_process
stop = (rank + 1) × rows_per_process
for each i from start to stop $- 1$ **do**
  for each j from 0 to $N - 1$ **do**
    if $|M[j \times N + i] - M[i \times N + j]| > 1e{-}5$ **then**
      local_symmetry ← false
    end if
  end for
end for
Reduce local_symmetry using MPI_LAND to compute the global symmetry on the root process

IV. Experiments and System Description

The experiments for this project were conducted on a high-performance computing (HPC) system, specifically on the node `hpc-c11-node16`. This system is built on a 64-bit x86_64 architecture and utilizes an Intel(R) Xeon(R) Gold 6252N CPU operating at a base frequency of 2.30GHz. The system is equipped with four sockets, each containing 24 physical cores, for a total of 96 CPU cores. It has a theoretical FLOPS of 441.6 GFLOPS. The number of FLOPS can be calculated using the number of cores × clock speed × FLOP per clock cycle.

A. Experimental Setup and Procedures

For the purpose of these experiments, a maximum of 32 cores were utilized, with a total of 200 MB of memory allocated to the tasks. The code was compiled using g++ version 9.1.0 and Mpi version 3.2.1. The code is all implemented in a single `.cpp` file, and the matrix size can be decided by giving the side length as an argument. To test our functions, a job is submitted using a PBS file. This file handles the compilation of all tests and their execution. The results are then compared against a test function to verify correctness and are saved on convenient `.csv` files, to ease automatic data processing.

B. Experiment Design

The Experiment design for this project is quite straighforward. We begin by recording a time instance before invoking each function, once the function concludes its execution, a second time instance is recorded. The total run time is then determined by the different between the two timestamps. After collecting rouglhy 100 executions, the times are averaged together and visualized through graphs to present the project results.

V. Results and Discussion

Due to limited space we will discuss only the most significant results, specifically those concerning a matrix size of 4096×4096 for strong scaling and a weak scaling spanning from $n = 128$ to $n = 4096$. The results presented will follow this structure:

- MPI
- state-of-the-art comparison and limitations

A. MPI

We first analyze the results obtained for the matrix transposition function As shown in Figure 1, our strong scaling closely follows the ideal speedup curve up to a peak at 4 processors, after which it gradually decreases with the addition of more processors. This decrease in performance could be caused by sending the entire matrix the every processor.

Let's now analyze the results obtained for matrix symmetry. As shown in Figure 2, we observe a peak at 4 processors. However, in this case, the curve deviates more significantly from the ideal line, and the performance declines more rapidly as the processor count increases. This decline is likely due to a bottleneck caused by redundant checks in the implementation, where two processors unnecessarily verify the same elements. Reducing the number of duplicate checks could mitigate this issue and improve performance.

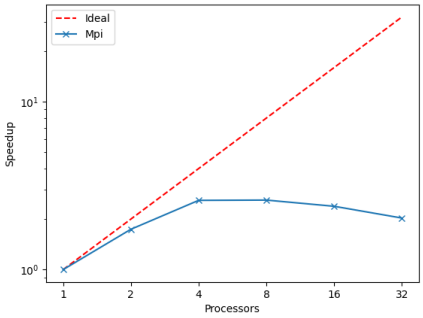

Fig. 1. Matrix Transposition MPI Strong Scaling $n = 4096$

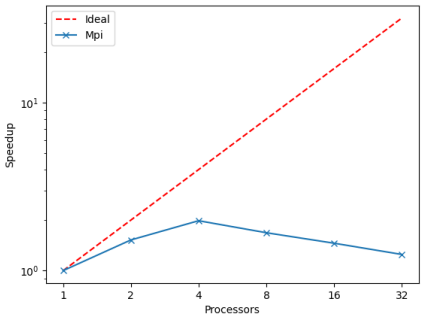

Fig. 2. Matrix Symmetry MPI Strong Scaling size $n = 4096$

We can now look at the weak scaling of the two functions shown in Figure 3 and in Figure 4. As we can see from the start the scaling of our approach as matrix grows has not a high level of efficiency this clearly visible when looking at Figure 3, in which performance plumets nearly instantly and stays constant to a low value. The opposite happens for Figure 4 where our efficiency decreases more gradually. This types of results as previously stated can be caused by

missed optimization leading to unnecessary computations and therefore a waste of resources as the matrix size grows.

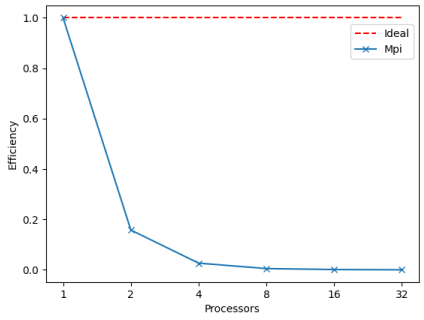

Fig. 3. Matrix Transposition MPI weak Scaling n from 128 to 4096

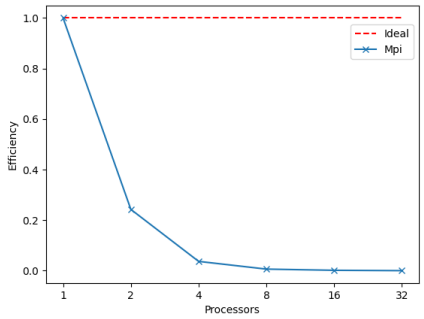

Fig. 4. Matrix Symmetry MPI weak Scaling n from 128 to 4096

To calculate the strong scaling the following formula has been used:

Strong scaling $= \frac{T(1)}{T(N_p)}$

To calculate the strong scaling the following formula has been used:

Weak scaling $= \frac{T(1)}{T(N_p)}$ with the size of the matrix doubling with every change of N_p

B. state-of-the-art comparison and limitations

When comparing the obtained results with state-of-art we can clearly see that my approach not evenly close to something groundbreaking and cannot rival in terms of performance when scaling up to bigger sizes.

VI. Conclusion

In conclusion the main limitations surely are the lack of refined optimization leading to some missed performance that could be extracted by a closer study at the problem, using other techniques such as block division and less intensive data transfer techniques.

References

[1] Khan, A., Al-Mouhamed, M., Fatayar, A., Almousa, A., Baqais, A., and Assayony, M., "Padding free bank conflict resolution for CUDA-based matrix transpose algorithm," in 15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2014, pp. 1-6, doi: 10.1109/SNPD.2014.6888709.

[2] B. Magalhaes and F. Schürmann, "Efficient Distributed Transposition Of Large-Scale Multigraphs And High-Cardinality Sparse Matrices," arXiv preprint arXiv:2012.06012, 2020.