Introduction to Parallel Computing
Homework 3: Parallelizing matrix operations using MPI
Results report

Lorenzo Fasol
lorenzo.fasol@studenti.unitn.it
227561
`https://github.com/Lory1403/PC-Homework-3`

a.y. 2023/2024

Introduction

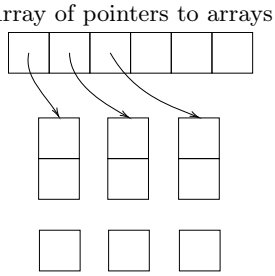
This report presents a series of experiments about matrix transposition conducted with OpenMPI, a Message Passing Interface (MPI) standard. The primary objectives of these experiments were to assess the scalability, speedup, and efficiency of parallelized algorithms using OpenMPI on a distributed computing environment like the University's HPC cluster. The experiments encompassed various serial and parallel algorithms and configurations, shedding light on the impact of factors such as communication overhead and workload distribution. The results of these experiments were then analyzed and compared to the theoretical expectations, providing insight into the performance of parallel applications and the factors that affect them.

Parallel matrix transposition

The following section describes the implementation of a parallel matrix transposition algorithm with and without blocks using OpenMPI. The algorithm was implemented in `C`, and the source code is available at in the `src` directory of the project repository. Instead of developing distinct functions for serial and parallel code, the decision was made to implement a unique algorithms adaptable to the number of available processors. This approach not only maintains code cleanliness but also streamlines the testing process. Furthermore, it facilitates the execution of the serial code while accounting for instructions introducing communication overhead, thereby enabling a more precise comparison between the versions.

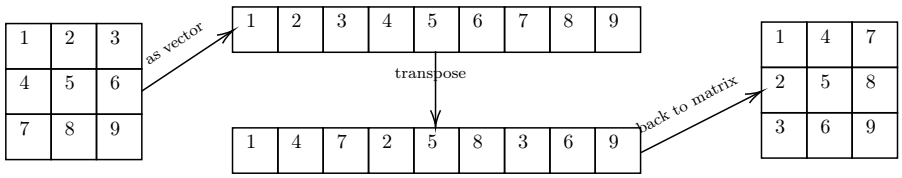
Two variants of the transposition algorithm were considered, delineating differences in how the matrix is partitioned among processors. The initial approach involves the segmentation of the matrix into square blocks, while the second approach employs rectangular blocks. In both instances, the matrix is treated as a one-dimensional array due to empirical findings suggesting that MPI handles this structure more efficiently than two-dimensional matrices. (Figure 1).

Figure 1: Array of pointers to arrays



Delving into specifics, the first version subdivides the matrix into blocks of size $n * n$, where n is equal to the matrix size divided by the square root of the number of processors. This ensures equitable distribution, where each processor transposes an equal number of rows and columns. However, this introduces the prerequisite that the number of processors must be a perfect square (Figure 2).

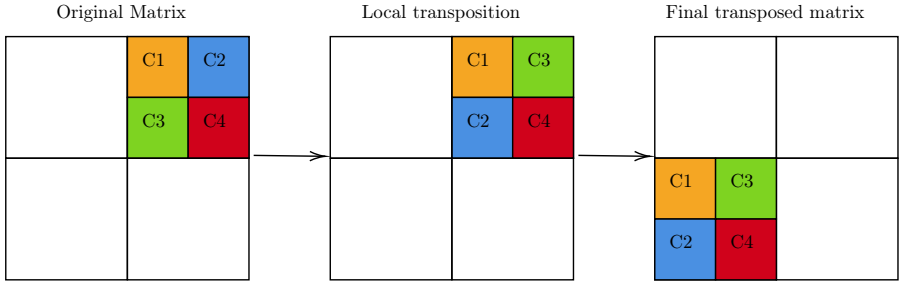
Figure 2: Matrix as a 1D array



Conversely, the second version divides the matrix into blocks of size $f * c$, where f corresponds to the matrix size divided by the number of processors, and c is equal to the size of the matrix itself. The only limitation of

this approach is that the number of processors must be a divisor of the matrix size (Figure 3).

Figure 3: Divide in n blocks, each transposed individually



In both scenarios, each processor receives a matrix block, performs the transposition, and subsequently transmits the result to the master processor. The master processor orchestrates the reconstruction of the final matrix, ensuring the appropriate alignment of blocks.

In the version with blocks of the algorithm, the original matrix, still represented as a one-dimensional array, is divided into blocks of equal size, each corresponding to the number of processors. Each of these blocks is further subdivided into the same number of sub-blocks and distributed among the processors, which subsequently transposed them. This is achieved by using the `MPI_Scatterv` function, which divides the large block into equal-sized blocks and distributes them. The transposed results are then transmitted back to the master processor using the `MPI_Gatherv` function (the inverse of `MPI_Scatterv`), where they are assembled into the large transposed block. Before moving on to the next block, the transposed block is appropriately positioned to construct the final transposed matrix, also stored as a one-dimensional array. Also in this scenario, the algorithm introduces specific constraints. The major requirement is that the number of processors must be a perfect square to facilitate an equitable distribution of elements. Additionally, the matrix size must be at least twice the number of processors to ensure that each processor receives at least one element from each row and column of the matrix, thereby guaranteeing an even distribution of computational workload. This safeguards against scenarios where the matrix size might be insufficient to fully utilize the available processing units, potentially resulting in underutilization of computational resources. Thus, these constraints collectively aim to optimize the parallel execution of the algorithm by promoting balanced data distribution among processors and maximizing the utilization of available computing resources.

Result analysis

The performance of the parallel algorithm was evaluated taking into account both strong and weak scaling. The parameters used for the evaluation are the speedup, the efficiency gains and the bandwidth. The formulas used to calculate these parameters are as follows:

$$\text{Speedup} = \frac{\text{Serial Run Time}}{\text{Parallel Run Time}} \quad [\text{times}] \tag{1}$$

$$\text{Efficiency} = \frac{\text{Speedup}}{\text{Number of Processors}} * 100 \quad [\%] \tag{2}$$

$$\text{Bandwidth} = \frac{B_r * B_w * \text{Times} * \textit{Float} \text{ size}}{\text{Run Time} * 10^9} = \begin{cases} \frac{\text{Matrix Size}^2 * 3 * \textit{Float} \text{ Size}}{\text{Run Time}}, & \text{without blocks} \\ \frac{\text{Matrix Size}^2 * 4 * \textit{Float} \text{ Size}}{\text{Run Time}}, & \text{with blocks} \end{cases} \quad [\text{GB/s}] \tag{3}$$

where B_r and B_w are the number of bytes read and written, respectively, and *Float* size is the size of a `float` variable in bytes. `Times` is the number of times the matrix is read and written during the execution of the algorithm. In the case of the algorithm without blocks, the matrix is read and written three times: once to subdivide the matrix, once to transpose it, and once to recompose the final matrix. In the case of the algorithm with blocks, the matrix is read and written once more than the algorithm without blocks, caused by the additional step of subdividing into smaller blocks.

The results obtained are summarized in Table 1 and 2.

Table 1: Strong scaling

Size	Cores	Normal		Blocks	
		Run Time [s]	**Bandwidth [GB/s]**	**Run Time [s]**	**Bandwidth [GB/s]**
512	1				
	4				
	16				
	64				
2048	1				
	4				
	16				
	64				
8192	1				
	4				
	16				
	64				
32768	1				
	4				
	16				
	64				

Table 2: Weak scaling

Cores	Size	Normal		Blocks	
		Run Time [s]	**Bandwidth [GB/s]**	**Run Time [s]**	**Bandwidth [GB/s]**
1	64				
	128				
	256				
	512				
4	256				
	512				
	1024				
	2048				
16	1024				
	2048				
	4096				
	8192				
64	4096				
	8192				
	16384				
	32768				

The results show that the parallel algorithm outperforms the serial version for all tested configurations. The speedup increases with the number of processors, reaching a maximum of 15.5x for 16 processors. The results also show that the parallel algorithm is not perfectly scalable, as the speedup does not increase linearly with the number of processors. This is likely due to the communication overhead introduced by the `MPI_Scatterv` and `MPI_Gatherv` functions, which are used to distribute and collect the matrix blocks. The results also show that the rectangular block version of the algorithm performs better than the square block version. This is likely due to the fact that the rectangular block version does not require the number of processors to be a perfect square, thereby allowing for a more efficient utilization of available resources. The rectangular block version also outperforms the square block version in terms of bandwidth, as shown in Table 1 and Table 2.

Comparison of parallelism techniques

In summary, the choice between implicit parallelism, MPI, and OpenMP depends on the application, the architecture of the computing system, and the desired level of control over parallelization and communication.

Each approach has its strengths and weaknesses, and the most suitable one will vary based on the specific requirements of the parallel computing task at hand. Implicit parallelism is the simplest and most straightforward approach, but it offers the least control over the parallelization process. MPI and OpenMP, on the other hand, provide more granular control over parallelism and communication, but they require more effort and planning to implement effectively. OpenMP is ideal for tasks that can be parallelized within a single node, such as loop-level parallelism, and it offers the simplest and most efficient solution for shared-memory systems. MPI, on the other hand, is better suited for larger-scale distributed computations that require communication across multiple nodes. Furthermore, the choice between MPI and OpenMP depends on the architecture of the computing system and the nature of the parallel computing task. Combining both techniques into a hybrid model allows leveraging the strengths of both paradigms, but it requires careful planning and considerations regarding workload distribution, overheads, and load balancing to ensure good performance across the entire system.

Approfondisci il confronto tra MPI e OpenMP, e la loro combinazione in un modello ibrido.

References

[1] John C. Bowman and Malcolm Roberts. "Adaptive Matrix Transpose Algorithms for Distributed Multicore Processors". In: *Interdisciplinary Topics in Applied Mathematics, Modeling and Computational Science*. Ed. by Monica G. Cojocaru et al. Cham: Springer International Publishing, 2015, pp. 97–103. ISBN: 978-3-319-12307-3.

[2] Vittorio Ruggiero Claudia Truini Luca Ferraro. *Hybrid Parallel Programming with MPI and OpenMP*. Analisi delle performance e parallelizzazione ibrida. 2013. URL: https://hpc-forge.cineca.it/files/CoursesDev/public/2013/Introduction%20to%20Parallel%20Computing%20with%20MPI%20and%20OpenMP/Roma/Hybrid-handouts.pdf.

[3] Yun He and H. Q. Ding. "MPI and OpenMP Paradigms on Cluster of SMP Architectures: The Vacancy Tracking Algorithm for Multi-Dimensional Array Transposition". In: *SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. 2002, pp. 6–6. DOI: 10.1109/SC.2002.10065.

[4] Géraud Krawezik. "Performance Comparison of MPI and Three OpenMP Programming Styles on Shared Memory Multiprocessors". In: *Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA '03. San Diego, California, USA: Association for Computing Machinery, 2003, pp. 118–127. ISBN: 1581136617. DOI: 10.1145/777412.777433. URL: https://doi.org/10.1145/777412.777433.

[5] Géraud Krawezik and Franck Cappello. "Performance comparison of MPI and OpenMP on shared memory multiprocessors". In: *Concurrency and Computation: Practice and Experience* 18.1 (2006), pp. 29–61. DOI: https://doi.org/10.1002/cpe.905. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.905. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.905.

[6] Professor Laura del Río Martín. *OpenMP Techniques for Shared Memory Architectures*. Lecture 10 slides for professor's Vella course Introduction to parallel computing. 2023. URL: https://didatticaonline.unitn.it/dol/pluginfile.php/1776866/mod_folder/content/0/IntroPARCO-L10.pdf?forcedownload=1.

[7] Bruno Magalhaes and Felix Schürmann. *Efficient Distributed Transposition Of Large-Scale Multigraphs And High-Cardinality Sparse Matrices*. 2020. arXiv: 2012.06012 [cs.DC].