# 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**

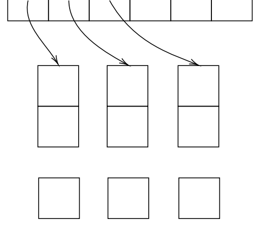
This report presents a series of experiments about matrix transposition conducted with OpenMPI, a Message Passing Interface (MPI) standard. The primary objectives of these experiments were to assess the scalability, speedup, and efficiency of parallelized algorithms using OpenMPI on a distributed computing 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 is 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 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.

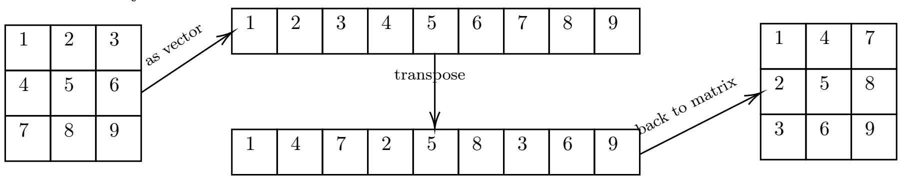
Two variants of the transposition algorithm were considered, delineating differences in how the matrix is partitioned among processors. The initial approach involves the segmentation of the matrix into square blocks, while the second approach employs rectangular blocks. In both instances, the matrix 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



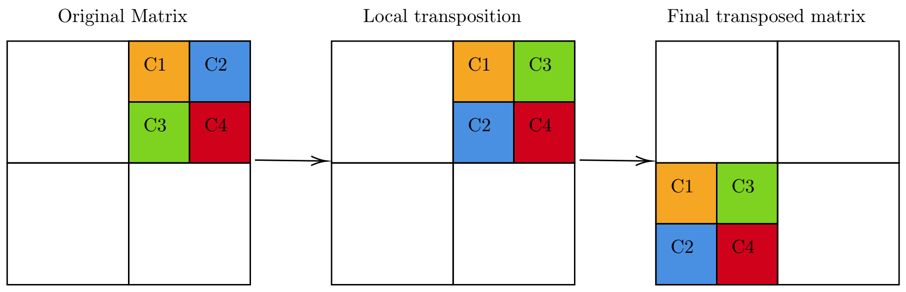
Delving into specifics, the first version subdivides the matrix into blocks of size $n * n$, where $n$ is equal to the matrix size divided by the square root of the number of processors. This ensures equitable distribution, where each processor transposes an equal number of rows and columns. However, 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 into $n$ 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 the 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 considering 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:

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

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

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

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.

I encountered challenges in obtaining results for the missing values in the following tables. The execution faced delays, eventually leading to termination by the system when the wall time was reached. This issue likely arises due to insufficient available memory on the assigned nodes.

The results obtained are summarized in Tables 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 | 0.002226 | 1,413 | 0.005520 | 0.760 |
| | 4 | 0.002251 | 1,397 | 0.003363 | 1,247 |
| | 16 | 0.005359 | 0,587 | 0.003376 | 1,242 |
| | 64 | 0.014420 | 0,218 | 0.009164 | 0,458 |
| 2048 | 1 | 0.064612 | 0.779 | 0.079998 | 0.839 |
| | 4 | 0.026691 | 1.886 | 0.044276 | 1.516 |
| | 16 | 0.025379 | 1.983 | 0.041785 | 1.606 |
| | 64 | 0.083068 | 0.606 | 0.049830 | 1.347 |
| 8192 | 1 | 3.729533 | 0.216 | 2.515364 | 0.427 |
| | 4 | 1.019834 | 0.790 | 0.901501 | 1.191 |
| | 16 | 1.176419 | 0.685 | 0.627510 | 1.711 |
| | 64 | 1.462280 | 0.551 | 0.765071 | 1.403 |
| 32768 | 1 | 57.112991 | 0.226 | 67.101906 | 0.256 |
| | 4 | - | - | 24.95976 | 0.688 |
| | 16 | - | - | 11.06617 | 1.552 |
| | 64 | - | - | 11.33467 | 1.516 |

**Table 2:** Weak scaling

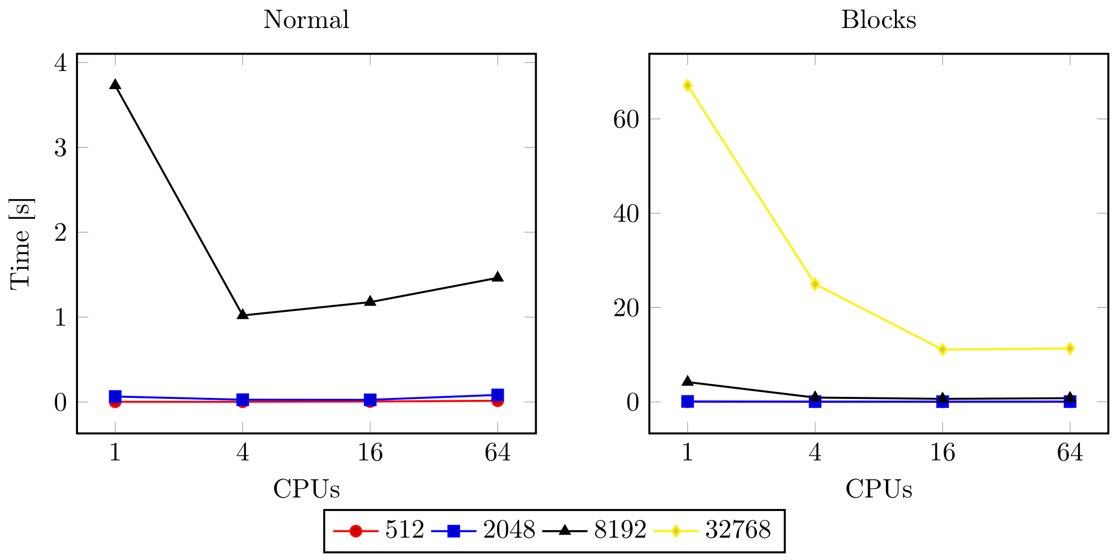
| Ratio | CPUs | Normal | | Blocks | |
|---|---|---|---|---|---|
| | | Run Time [s] | Bandwidth [GB/s] | Run Time [s] | Bandwidth [GB/s] |
| 64 | 1 | 0.00034 | 1.45 | 0.000097 | 0.68 |
| | 4 | 0.001141 | 0.69 | 0.001797 | 0.58 |
| | 16 | 0.013206 | 0.95 | 0.010417 | 1.61 |
| | 64 | 0.218325 | 0.92 | 0.171367 | 1.57 |
| 128 | 1 | 0.000111 | 1.77 | 0.000336 | 0.78 |
| | 4 | 0.002095 | 1.50 | 0.003130 | 1.34 |
| | 16 | 0.039099 | 1.29 | 0.042607 | 1.58 |
| | 64 | 0.650468 | 1.24 | 0.684192 | 1.57 |
| 256 | 1 | 0.000441 | 1.78 | 0.001309 | 0.80 |
| | 4 | 0.006750 | 1.86 | 0.010760 | 1.56 |
| | 16 | 0.107758 | 1.87 | 0.158777 | 1.69 |
| | 64 | 5.574380 | 0.58 | 2.789386 | 1.54 |
| 512 | 1 | 0.002263 | 1.39 | 0.005754 | 0.73 |
| | 4 | 0.025396 | 1.98 | 0.043814 | 1.53 |
| | 16 | 0.659142 | 1.22 | 0.636953 | 1.69 |
| | 64 | - | - | - | 1.51 |

In the weak scaling plot (Plot 2), can be observed that despite the constant workload per processor, the execution time tends to increase as the number of processors grows. This phenomenon is attributed to the fact that an increase in processors also augments the number of communications required for processor synchronization, becoming more frequent and costly.
In the strong scaling one (Plot 1), on the other hand, as the number of processors increases, the execution time tends to decrease, although not linearly. This behaviour is attributed to the same reason as in weak scaling. These results are primarily supported by efficiency graphs, demonstrating a decreasing trend in efficiency as the number of processors increases.
Plot 3 shows that the efficiency of the block-based program steadily decreases with an increasing number of processors, while the algorithm without blocks exhibits a more irregular trend, experiencing a sharp efficiency decline already at 4 processors.
Contrary to expectations, the bandwidth graph (Plot 5) shows irregular behaviour for the blockless algorithm with a noticeable decline at 64 processors. In contrast, the block-based algorithm exhibits a more regular trend in line with expectations. Nevertheless, a decline is observed with a matrix size of 512 due to its reduced

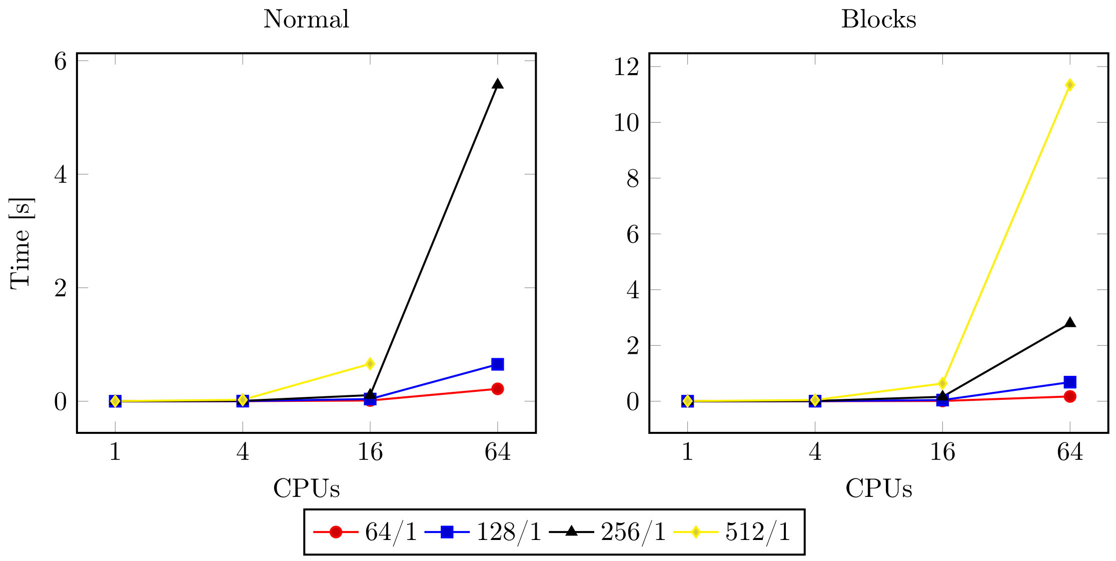
dimensions, preventing optimal utilization of available resources.
The speedup graphs (Plot 4) summarize the aforementioned observations, revealing that the blockless algorithm, despite an initial peak in performance at 4 processors, tends to decrease with an increasing number of CPUs. Meanwhile, the block-based algorithm displays a more consistent trend, experiencing a peak in performance at 16 processors for the larger matrix and maintaining a steady trend for others.
From the strong scaling graph, can be further observed that beyond 4 processors for the blockless version and 16 processors for the block-based version, the execution time tends to stabilize. This phenomenon could be attributed to the fact that as the number of processors increases, the execution time may decrease, but delays introduced by communication between processors offset the saved time.

From all these data, can be asserted that MPI may not be the optimal approach for the direct parallelization of algorithms. Indeed, MPI serves as a framework for inter-process communication rather than algorithm parallelization. MPI is better suited for applications that demand process communication, such as matrix multiplication, where computations are performed in parallel, but inter-process communication is essential for the summation of partial results. In such cases, inter-process communication is substantial and time-consuming, making MPI less suitable as the preferred framework.

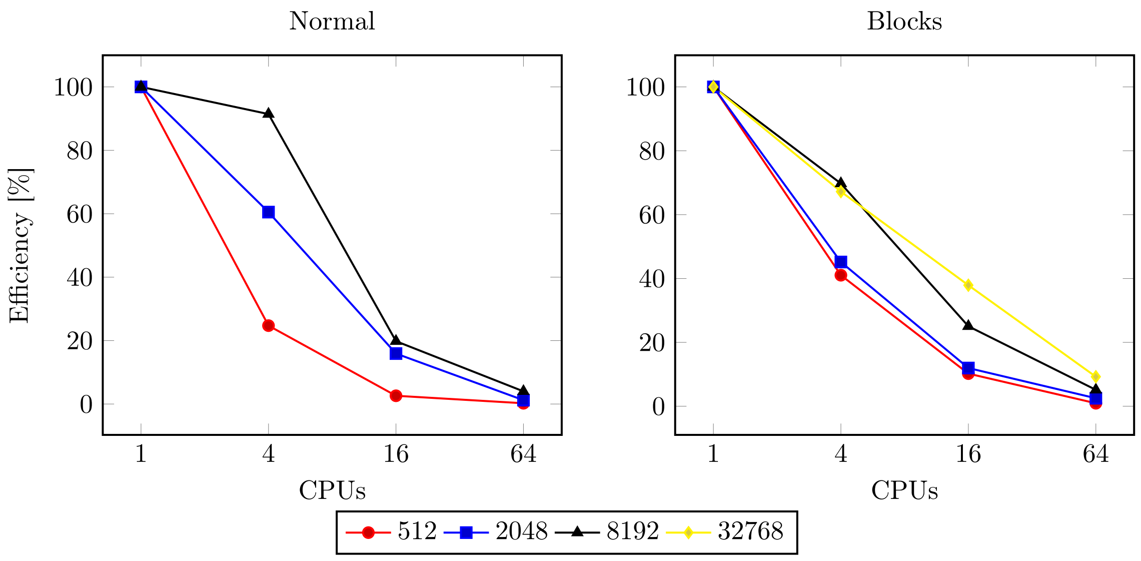
**Plot 1:** Strong scale graphs by size of matrices



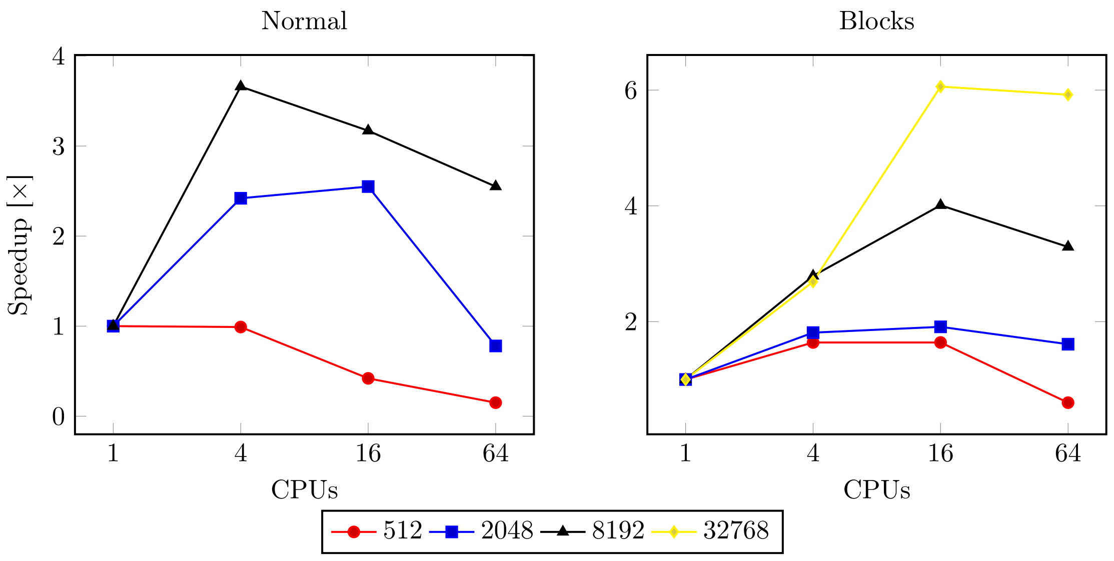
**Plot 2:** Weak scale graphs by matrix size-CPUs ratio



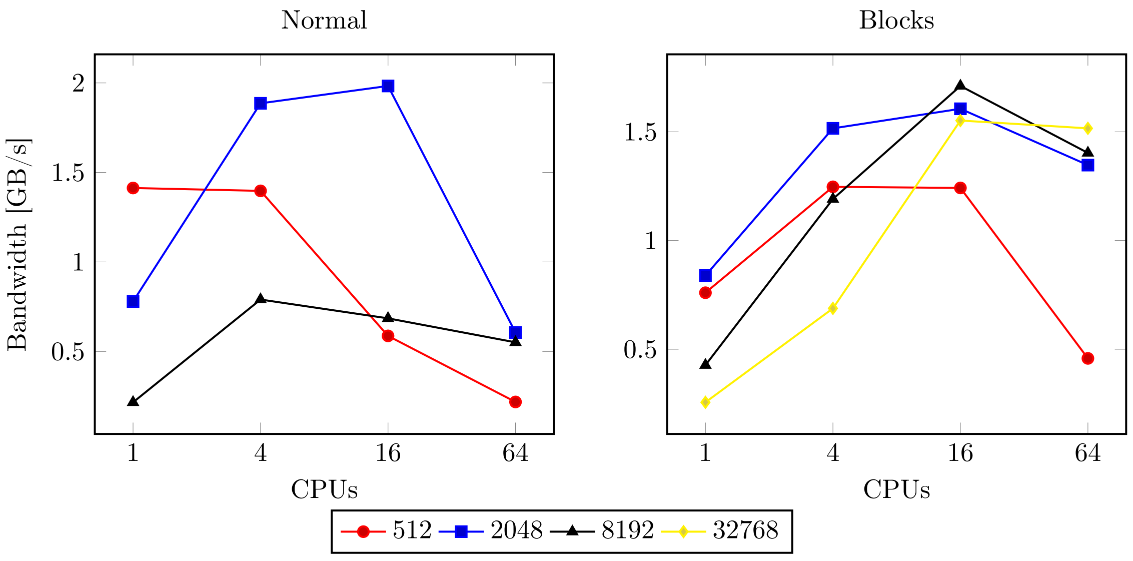
**Plot 3:** Efficiency graphs by size of matrices



**Plot 4:** Speedup graphs by size of matrices



**Plot 5:** Performance graphs by size of matrices



## 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.

The hybrid model made of the synergy between MPI and OpenMP, capitalizes on the advantages of both paradigms. In this hybrid approach, MPI manages communication and coordination between different nodes or processors, while within each node, OpenMP guides the parallelization of tasks through multiple threads. This hybrid model is particularly advantageous in applications demanding both inter-node and intra-node parallelism, effectively leveraging the strengths of both MPI and OpenMP.
However, this approach is also the most complex and challenging to implement, requiring a thorough understanding of both MPI and OpenMP. Effective utilization of this hybrid model demands meticulous planning and thoughtful considerations regarding workload distribution, potential overheads, and the crucial aspect of load balancing. These factors are instrumental in guaranteeing optimal performance across this entire system.

## 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.