Parallel computing - second deliverable

Daniele Pedrolli
DISI, University of Trento
Email: daniele.pedrolli@studenti.unitn.it

***Abstract*—In the first part of this research, available at [DEL1], we explored the optimization of matrix transposition mainly through OpenMP to employ more computational resources. In this second part of the project, we'll attempt to optimize the operation with MPI, by using a distributed memory model. The results will then be bench-marked against both a sequential and OpenMP implementation of the algorithm.**

I. Introduction

A. Importance of the project

Matrix transposition is a simple mathematical operation that is widely used across various fields like machine learning, data science, physics, engineering.

While the operation itself is straightforward, its impact on performance is significant, mainly for large datasets, which may exceed the capacity of on-chip caches. For this reason, optimizing such operation poses a relevant challenge, especially in terms of memory accessing patterns, requiring the CPU, and potentially GPU to handle them efficiently, as further explored in [CTZ14] "A decomposition for in-place matrix transposition" and [AWD23] "Analyzing and implementing GPU hash tables".

B. Outline of the project

The project aims at analyzing performance for both a symmetry check function and a transposition function (same as in [DEL1]), but I believe the symmetry check was slightly overlooked in the first part, as it was only implemented sequentially. This time it is implemented with MPI as well, even though only one approach is proposed, as opposed to the transposition, for which there are two.

II. State-of-the-art

A. Existing research and proposed solutions

The Message Passing Interface (MPI) is a widely used distributed memory parallelization technique that is by now extensively implemented and tested. Its implementation on the other hand is not trivial as, depending on the the solution scalability issues may arise, which are detrimental to performance, especially in large matrices.

An example of these issues is when using `MPI_Alltoall`, which shares the data from every processor to every other, meaning the number of communications scales as $O(P^2)$ (where P is the number of nodes). This can for example be improved by using `MPI_Bcast`, which only shares data from the main node to all other nodes, so it scales as $O(log\ P)$, making it significantly more efficient with only a small difference code-wise.

It is these memory access patterns and data sharing patterns that have been the main subject of study in distributed memory models: [FOX87] introduced 2D cyclic data distribution to make sure that the workload is balanced across all processors. Additionally, there are some very recent studies ([ZHO24]) that focus on mitigating the communication overhead introduced by MPI, by "overlapping communication and computation", using non-blocking MPI calls to make sure the data is being transmitted at the same time part of the matrix is already being processed. This would improve the strong scaling, as communication latency is partially hidden, while the total amount of data transferred isn't reduced, limiting weak scaling benefits.

Hybrid approaches combining OpenMP and MPI have also been proposed, as early as 2008 [LUS08] "Early Experiments with the OpenMP/MPI Hybrid Programming Model", and further analyzed in [APP22] "Parallelization of Array Method with Hybrid Programming: OpenMP and MPI". These solutions leverage shared-memory parallelism to reduce the number of MPI processes in every node, limiting inter-node communication. Such approaches succeed in minimizing MPI's overhead, but the weak scaling benefits are still poor, resulting in diminishing returns as more nodes are added.

More recent developments explore one-sided communication: it allows a process to directly read or write the memory of another, eliminating the need for explicit message passing. These approaches show promise in both weak and strong scaling, as explored in [BAL13] "An Implementation and Evaluation of the MPI 3.0 One-Sided Communication Interface", but at the cost of code complexity. Some scalability issues are still present especially with large systems or problems, due to the very same reduced synchronization requirement that allows one-sided communication to increase performance in the first place, as concluded by Xin Zhao in [ZHAO17] "Scalability Challenges in Current MPI One-Sided Implementations".

B. Project objective

The objective of the project is to analyze the performance of MPI, starting from the simplest implementation of a distributed memory algorithm. We will then try to optimize it by reducing the volume of communication, and finally analyze more advanced techniques like transposition in blocks.

III. Contribution and methodology

As mentioned in the introduction, we wanted to start from a sequential and OpenMP implementation of the algorithm, and although there is not much to be said about the sequential

approach, it is worth noting that the OpenMP one allows for different number of threads to be used at each execution.

A. Implementation MPI_one

This implementation is contained in the file *04_transposition_mpi_one.c*. It aims to be the most simple and not very optimized, in order to give a baseline for MPI. Considering the complexity of the problem, rather than using an `MPI_Alltoall` technique, we use a `MPI_Bcast` to share data only from the main processor to all the others, to avoid the massive overhead of the all-to-all scaling($O(n^2)$). The algorithm is structured as follows.

The number of rows and columns to be handled by each processor is computed (`matrix_size/num_processors`), alongside the interval in which each node will operate, then the whole matrix is shared among the processors using `MPI_Bcast`. This is the same for both the symmetry check and transposition functions, but from this point forth the implementation differs.

The symmetry check (function signature: `int checkSymMPI(float *matrix, int n, int rank, int num_processor))` initializes a local and a global flag for symmetry, then each processor checks the rows and columns it was earlier assigned; if asymmetry is found, the local flag is set to 0, but the loop is carried out, in order to provide a comparable benchmark to the sequential and OpenMP approach. This decision was made considering that the probability of a symmetric matrix is nearly 0, so the check would only cover a handful of cells at most. Once the `for` loop finishes, all the `local_sym` are reduced into a single `global_sym` to be returned as result. This is achieved using an `MPI_Allreduce`, which is only executed on the main processor and does an AND operation between all the flags. Lastly, the matrix is de-allocated from memory on all processors, but the main one.

The transposition (function signature: `void matTransposeMPI(float *matrix, float *transposed, int n, int rank, int num_processors))` allocates an extra variable in each node: `local_transposed`, used to store the result; the operation is then carried out in a simple nested `for` loop. Once each node finishes, `MPI_Gather` is executed only on the main processor to put each locally transposed row, that is now a column, into the `transposed` matrix (that was given as argument to the function). Ultimately, on all processors the `local_transposed` matrix is de-allocated from memory. Memory is also de-allocated for the non-transposed matrix, on all processors but the main one.

B. Implementation MPI_two

This implementation is contained in file *05_transposition_mpi_two.c*. The objective was to reduce the communication volume, only sending to each processor the data it is going to handle. It's quite an obvious optimization, considering with the previous approach we'd send the whole matrix and then only process a specific interval in each node. The symmetry check isn't explained in detail for this file, as it is kept the same as the previous one, considering that it would require sending a block of rows and a block of columns anyways. This decision would only have a noticeable impact in the case we have very many processors compared to the size of the matrix: instead of sending n lines, we'd be sending $n \times$ num_processors. With a large number of nodes this would result in significant communication overhead. The algorithm is structured as follows.

The transposition (function signature: `void matTransposeMPI(float *matrix, float *transposed, int n, int rank, int num_processors)` begins the same way as before: by defining the number of lines each processor will handle (`matrix_size/num_processors`). Then a `local_block` is allocated on all nodes, alongside a `send_row_buffer` which will be used to store the rows to be sent to the main processor. On the main node a buffer to receive transposed rows is also allocated in `recv_row_buffer`. At this point we use `MPI_Scatter` to distribute the rows in the respective processors. From here, the block of rows is put into the send buffer, ready to be sent to the receiving buffer in the root node using `MPI_Gather`: it will be transposed "on the fly" right before being appropriately inserted in the final `transposed` matrix. Just like before, memory is de-allocated on all nodes for the local block and sending buffer, and the receiving buffer on the main processor.

IV. Experiments and system description

A. Computing system and platform specifications

Differently from [DEL1], the code for this part of the project was only compiled and tested on the UniTN cluster, which is a Linux-based system. The relevant specifics for the platform follow.

- **CPU Model**: Intel Xeon Gold 6252N, 2.30GHz
- **CPUs**: 96 cores per node, 1 thread per core, 4 sockets, 24 cores per socket
- **Cache**: *L1*: 32KB, *L2*: 1MB, *L3*: 36MB
- **Architecture**: x86_64, NUMA: 4 nodes
- **Virtualization**: VT-x enabled

B. Project organization

The files related to MPI and its bench-marking are in the *del2* directory of the project: there are 6 code files, which are all explained in detail in the README. Regardless, a few details worth noting follow.

- Each average time presented is computed over 30 iterations to partially rule out any outliers that may occur. Since in practice they appear only sporadically, there is no need to additionally normalize the data or anything like that.
- Each time a matrix is transposed, a function `checkTranspose` is called. This is to make sure the transposition was successful, since especially for

OpenMP and MPI, the code could compile and run even if it isn't working as intended.

- As mentioned earlier, the symmetry check function for all approaches could execute much faster if it stopped after finding the first asymmetry, but in all cases it is forced to traverse the whole matrix to ensure consistent results. This is because the overhead for OpenMP or MPI would outweigh the real performance of their algorithms.
- Unless specified, MPI data uses 32 processors, OpenMP uses 8.
- Supplementary data and tables for the graphs provided below can be found here [DATA].

V. Results and Discussion

A. Matrix transposition performance

Fig.1 shows the raw performance of each of the four presented approaches. It is to be noted that the MPI approaches use 32 processors, as we will later find out that it is the optimal number to guarantee an efficient use of resources. OpenMP on the other hand, uses 8 CPUs.

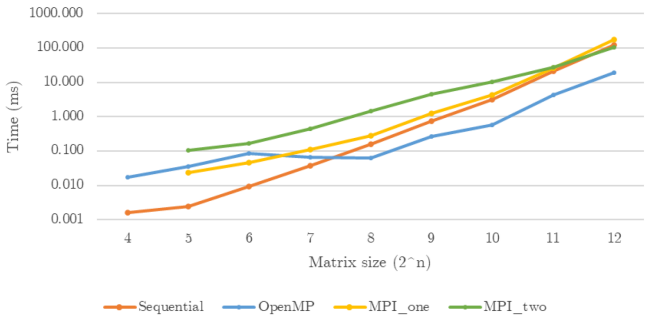


Fig. 1: Transposition performance

There are multiple observations to be done about this graph.

- OpenMP's performance is very similar in smaller matrices, due to the (small) overhead caused by the multithreading overhead.
- MPI's times are comprised of the communication time and computation time. It would be possible to show only the actual computation time, but I avoided doing it. The reason for this is somewhat trivial: in any real world case the impact of the overhead cannot be understated, and could be a deciding factor when choosing implementations. Additionally, considering the core difference and the times are still similar, we can deduce that the time added due to the scattering of the matrix is several-fold larger than the computation time.
- The sequential and OpenMP implementation appear to be following an exponential trend, so we can suppose that for even larger matrices, the performance won't be as relevant. Conversely, MPI's performance isn't all that good, but its graph looks to be on a converging or linear trend (better seen in fig.2, speedup). Due to this, we can suppose that on very large matrices, the performance will be far better than both sequential and OpenMP.

We can continue our analysis by looking at the results in terms of *Speedup*. These results are directly compared to the sequential approach, which will be our baseline from here on. The formula used to compute the speedup follows:

$$Speedup = \frac{\text{BaselineTime} - \text{TestTime}}{\text{BaselineTime}} \times 100$$

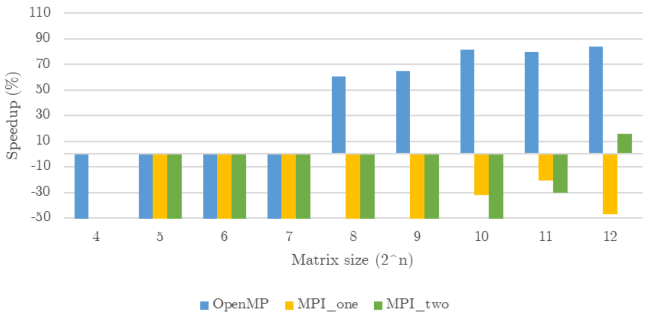


Fig. 2: Transposition speedup

It is worth noting that even though OpenMP largely outperforms the other approaches, as mentioned before MPI's performance seems to get much better with matrix size. This supports our hypothesis that on even larger matrices it would be the best approach by far.

At this point it is quite clear that the best way to fairly compare OpenMP and MPI is on the largest matrix at our disposal: 4096x4096. For smaller sizes, MPI's overhead becomes too relevant compared to the computation time. Thus, the following graphs will only show data for this matrix size (Additional data can be found here: [DATA]).

Let's now look at *efficiency* and *scaling*. We'll compute each using the formulae below (in order: efficiency, strong scaling, weak scaling).

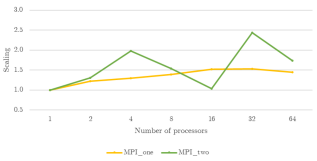
$$E(p) = \frac{S(p)}{p} \times 100 \qquad S(p) = \frac{T_1}{T_p} \qquad S_w = \frac{T_1(N)}{T_p(N \times p)}$$

TABLE I: 4096 Transposition efficiency

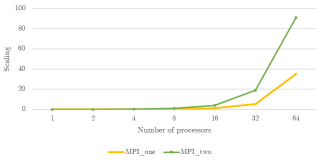
num_proc	MPI_one	MPI_two
1	1.000000	1.000000
2	0.613206	0.651572
4	0.324838	0.493720
8	0.173960	0.192426
16	0.095285	0.064825
32	0.047804	0.076284
64	0.022595	0.027142

Table 1 shows an interesting result: efficiency decreases as more cores are added. This will be discussed more later, but one conclusion we could draw is that the communication overhead is too relevant for the matrices we are handling, and perhaps MPI would be more suitable for larger, more complex problems.

To conclude our analysis, we look at strong and weak scaling. The strong scaling graph shows quite clearly that 32 is the optimal number of cores to ensure efficient use of resources. This is the reason we used this value all along.



(a) 4096 Strong scaling



(b) 32R weak scaling

Testing with 8 and 16 cores was also done, but the results were adjusted once this result was obtained. Having more than 32 cores does not have a relevant benefit on the matrices we are handling, as the scattering process would cause large delays. This is also visible in Fig.3c, and for that purpose the graph presented shows a constant workflow size of 32 rows per process (to mitigate communication volume). It is clear, however, that there is an ideal compromise between matrix size and number of cores: for MPI, it can most likely be found in much larger matrices, with a ad-hoc implementation and workload division.

B. Symmetry check performance

Most of the considerations to be made regarding the symmetry check performance are very similar to the transposition, so there isn't much left to be said. As mentioned before, there is only one symmetry check approach done with MPI, using 32 cores. Of course, the formulae are also identical.

For these reasons, the data shown is: symmetry check times and speedup (the rest can be found here: [DATA]).

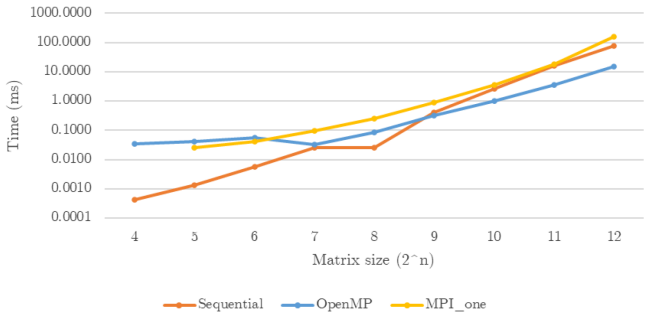


Fig. 4: CheckSym performance

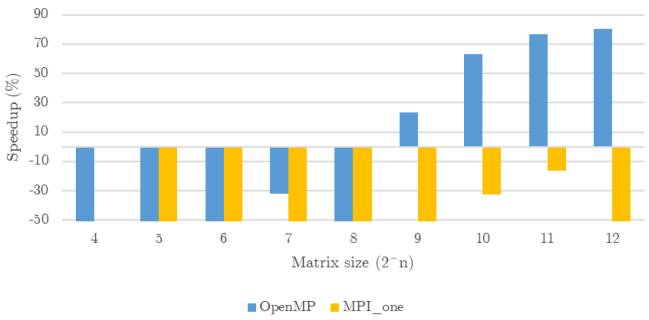


Fig. 5: CheckSym speedup

It is worth noting that the symmetry check times could be greatly reduced by stopping the function once the first asymmetry is found. As said before, I chose not to do this to ensure consistency in the results, rather than having some probabilistic inconsistency. Due to this, the times are very similar between transposition (MPI_one) and symmetry check, as the time to traverse the matrix is identical, and the transposition operation doesn't take very long at all.

VI. Conclusions

From the results obtained we understand two things: for small matrices, any of the algorithms proposed would do a good enough job, considering the amount of data is reduced. Operations with large matrices however, do need some optimization in order to run efficiently.

For that purpose, OpenMP substantially outperforms both sequential and MPI implementations. I believe that it is due mostly to the problem at hand: matrix transposition is memory bound, but MPI still has two main issues in communication overhead and poor cache reuse (due to scattering of data). OpenMP on the other hand, has direct memory access, better cache locality and far less synchronization overhead.

In different settings, with a different problem, MPI could easily perform better than OpenMP: for example compute-bound problems, problems that require very large memory capacity, or strong scaling problems with minimal communication overhead (like Monte Carlo simulations, for example, more here: [RAF18]).

Nonetheless, the objectives of the project were successfully achieved: the initial approach was optimized, the volume of communications was reduced, the performance was analyzed and bench-marked. While we had initially hoped MPI would "take the crown" and outperform the other approaches, we have gained valuable insights into its behavior, strengths, and limitations.

VII. GIT and reproducibility

The GIT repository can be found at this link, the relevant reproducibility instructions are contained in the README of the repository. The data that was collected but not directly presented in this paper can also be found in the repository, at `root/del2/utils/MPI_tables.xlsx`.

References

[DEL1] Daniele Pedrolli, "Parallel computing, Deliverable 1", GitHub repository link.

[DATA] Daniele Pedrolli, "Parallel computing, Supplementary MPI data", .XLSX file link

[CTZ14] B. Catanzaro, A. Keller, and M. Garland, "A decomposition for in-place matrix transposition," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Orlando, FL, USA, Feb. 2014. Available at: This link.

[AWD23] M. A. Awad, S. Ashkiani, S. D. Porumbescu, M. Farach-Colton, and J. D. Owens, "Analyzing and implementing GPU hash tables," in *Proceedings of the SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*, Jan. 2023, pp. 33–50. Available at: This link.

[ZHO24] H. Zhou, K. Raffenetti, Y. Guo, T. Gillis, R. Latham, and R. Thakur, "Designing and prototyping extensions to the Message Passing Interface in MPICH," *International Journal of High Performance Computing Applications*, vol. 38, no. 5, Aug. 2024. Available at: This link.

[FOX87] G. C. Fox, S. Otto, A. J. Hey, and D. Towsley, "Matrix algorithms on a hypercube I: Matrix transposition," *Parallel Computing*, vol. 4, no. 1, pp. 17–31, 1987. Available at: This link.

[LUS08] E. Lusk and A. Chan, "Early Experiments with the OpenMP/MPI Hybrid Programming Model", Argonne National Laboratory, 2008. Available at: This link.

[APP22] A. V. Martinez "Parallelization of Array Method with Hybrid Programming: OpenMP and MPI", *Applied Sciences*, vol. 12, no. 15, p. 7706, 2022. Available at: This link.

[ZHAO17] P. Balaji, X. Zhao, W. Gropp, "Scalability Challenges in Current MPI One-Sided Implementations", in *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 88–99, 2017. Available at: This link.

[BAL13] J. Dinan, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, R. Thakur, "An Implementation and Evaluation of the MPI 3.0 One-Sided Communication Interface", Argonne National Laboratory, 2013. Available at: This link.

[RAF18] S. Rafibakhsh, D. Hassani, "Parallelization and Implementation of Multi-Spin Monte Carlo Simulation of 2D Square Ising Model using MPI and C," Journal of Theoretical and Applied Physics, 2018. Available at: This link.