Parallelizing Matrix Operations using MPI

Davide Facchinelli
Mat: 237302
Email: davide.facchinelli@studenti.unitn.it

***Abstract*—The focus of this research is exploring MPI (Message Passing Interface) by optimizing the matrix transposition operation. The results will then be compared with the sequential and the OpenMP versions of the algorithm to highlight the strengths and the weaknesses of this interface. Besides looking at the matrix transposition algorithm, the research will also study, although with less emphasis, the symmetry check function.**

I. Introduction

A. Before starting

This research is the continuation of the one reported here: [Fac24], which objective was to study and explore the effect of Implicit and Explicit parallelization through GCC directives and OPENMP. It is important to have a look at it since it represents the foundations for this project.

B. Importance of the project

The matrix transposition operation itself is trivial, however, the scalability of it, the way computers' memories work and the amount of applications in which it is needed, bring to it a lot of attention and efforts in optimizing it. In real-world applications, a single matrix can contain millions of elements and take entire GBs of space to be stored in, since RAMs are not usually that big, driving CPUs to fetch chunks of memory in a clever way, is a crucial challenge. The problem is furthermore empathized by it being the core of almost every application. For example, it is used in cryptography to encode and decode messages (an example is the Hill cipher that uses it to find the adjugate matrix, Brian Worthington explains the process in the paper: [Wor10]) , by Artificial Intelligence networks to transpose dataset and analyze errors, in image processing to flip images, in signal processing as a way to simplify filtering, and much many others.

C. Outline of the project

The projects is logically into two main parts: matrix transposition and matrix symmetry check (still, the first one, is a little bit more empathized than the second one). That's what I also did in my previous project and I decided to keep up with my choice because the chance of randomly allocating a symmetric matrix are near to 0, and then it would be impossible to study the symmetry check function performances.

II. State of the Art

A. Current Researches and Implementations

The Message Passing Interface (MPI) has become a standard when talking about distributed memory systems. The most common and simple MPI technique is `MPI\_Alltoall` directive: it implies all the processors to share data with all the other ones, allowing them to reorder the matrix elements. However, it is immediately clear that this type of methods do not present good scalability results: the number of communications quadratically increases with the number of nodes, the bandwidth required increases with the matrix size and the efficiency is affected by whether the matrix is sparse or not (and square or not). Current researches works exactly on these problems, and below I will present some of them.

The issue of efficiency lack in sparse matrix is specifically addressed by a research of Magalhães and Shürmann [BRCM20]. They approach the problem to "find the way back" in graph connectivity (represented by sparse matrices in which the cells are the connection between two points/location). Their solution to this problem is an algorithm presenting linearly increasing weak scaling and a constant strong scaling. Moreover, the runtime was not dependent on the matrix size.

In order to reduce the bandwidth required by the communications there are many researches (a little less recent) presenting a hybrid use of MPI and OpenMP. The idea here is to exploit the capabilities of multi-core architectures by reducing the communication between nodes and using the node itself to carry out some computations. The research of Yun He and Chris H.Q. [YH02] shows that implementing an hybrid model can improve the time computation by a factor of 4.4.

Most recent researches are looking at way to approach the problem by taking in consideration the physical network topology. These type of algorithms are called topology-aware algorithms. A group of researchers of the National University of Defense Technology in China published an article [Xin24] presenting a the MTS library that can be used as massage passing interface between different levels of the networks reducing latency and bandwidth usage.

B. Objective of the project

From an algorithm perspective, the objective is to implement a code by using a simple directive such as `MPI_Bcast` to explore and understand how basic MPI works, and then try to look at some more advanced feature to subdivide the matrix in chunks to share equally among the processes.

III. Contribution and Methodology

A. The Methodology behind the project

My initial idea was to produce 2 codes for both the matrix transposition and the symmetry check processes, with a total of 4 programs. The reasons being having a first, easier version of the code, and a second, more structured one, presenting better performances. That's due to the fact that directive such as `MPI_Bcast` are simpler to use, but present some clear problems in scalability. In fact, for each new process, the matrix has to be sent an additional time, which, for big sizes represents a huge deal breaker in term of execution time.

B. What went wrong

In the repository, however, it is possible to find only 3 of the codes promised above. The missing one is the matrix transposition performed by using broadcast techniques. This lack is due to the little amount of time I had, and my decision to focus myself on the subdivision of the data in chucks. Moreover the symmetry check code with the "more advanced technique" does not work, still you can find it in the GitHub repository since I explored three different techniques in trying to complete it; you will find a little section below talking about the problem that I encountered here.

C. The Challenges and the Techniques

The first challenge that I encountered was in the Matrix Transposition in block algorithm. The method that I adopted was dividing the rows among the processes using `MPI_Scatterv`; the original idea was to use `MPI_Scatter`, which is more simple, but leaves you less flexibility in deciding the access patterns. Gathering back the transposed rows was not a straightforward step: the columns are not contiguously allocates, so they are tricky to move around. The solution I came up with consisted in transposing the 1D rows, received by a process, in a 2D local_transposed_matrix, in a way that it was more easy to collect the sparse column directly in a global 2D matrix using the `MPI_Gatherv` directive. To make the process clearer, the snipped of code 1 tries to emulate what I explained.

Algorithm 1 Matrix Transposition with MPI

```
Require: float *GlobalMatrix, float
    *LocalMatrix, float **LocalTransposed,
    float **GlobalTransposed
1: MPI_Scatterv(GlobalMatrix, Local
   Matrix)
2: Matrix_Tranpose(LocalMatrix, Local
   Transposed)
3: for i=0 to matrix_size do
4:   MPI_Gatherv(LocalTransposed[i],
     GlobalTransposed[i])
5: end for
```

For what concerns the symmetry check I was talking about before, the issue was given by the fact that MPI directives do not accept as input non contiguous pieces of memory. Since the idea was to scatter a set of rows with the corresponding set of columns to let a process check the symmetry on that section of the matrix, I had to find a solution to select the columns. On the repository it is possible to find three different methods that I tried to develop; each one based on the use of MPI custom data types combined with `Scatterv`. The easier one is `MPI_Type_vector`; it allows to specify a pattern that has to be used to access a vector, which means that by setting the right parameters, it is possible to access the columns. However, in my case, after correctly extracting and sending the columns for the first process, it started fetching the columns for the second process (and all the other) in a weird pattern that I was not able to interpret.

IV. Experiment and System Description

A. Computer System and platform description

All the codes were tested and run on the UNITN cluster, here below I report the characteristic of it for reproducibility purposes.

- Architecture: x86_64;
- CPU op-modes: 32-bit, 64-bit;
- CPUs: 96;
- Thread per core / Core per socket / Sockets: 1 / 24 / 4;
- NUMA nodes: 4;
- Vendor ID: GenuineIntel;
- CPU family: 6;
- Model: 85;
- Model name: Intel(R) Xeon(R) Gold 6252N CPU @ 2.30GHz;
- Stepping: 7;
- CPU MHz: 2300.000;
- BogoMIPS: 4600.00;
- Virtualization: VT-x;
- L1d / L1i / L2 / L3 caches: 32K / 32K / 1024K / 36608K;

B. Project Setup

In the GitHub repository there are 20 files. They are already in-depth explained in the README.md, however, to avoid any confusion, I will give a glimpse on the new ones, added for this project:

- 2 .pdf files, which are the deliverables that I submitted for the university research. I decided to push them in GitHub to give the possibility to cite them for whoever wants to.
- The MPI.pbs file that should be used to run on a single shot (on the cluster) all the files used in this project to collect data.
- The transposition_MPI_blocks.c file that uses MPI to transpose a Matrix by subdividing it in rows to scatter equally among the processes.
- The sym_check_MPI.c file that uses MPI to check the symmetry of a Matrix and the sym_check_MPI_blocks.c which does not work, but gives a view on the techniques that I tried to develop to scatter among the processes both rows and columns.

The list below points out a set of suggestions useful to have a correct code comprehension:

- To test whether the code actually transposes the matrix (or correctly checks its symmetry) it's possible to remove the comments that print the matrices, moreover, when matrices are too big to be visually checked, it is possible to use the function `matrix_actually_transposed`.

- The time showed at the end of each execution is the average time over 50 iterations.
- If needed, to have a complete vision of the graphs and the data collected during the project, at the end of the report I added the link to the Google Sheet containing all the measures I did.

V. Result and Discussion

A. *Presentation and analysis of Matrix Transposition*

In the graph 1 (notice that the graph is in logarithmic scale) I presented three different solution at the problem, the first one being the sequential, the second one being the OpenMP (with 8 threads) and the third one being MPI (with 32 processors). The choice on the amount of threads and processors to use was based, in both the cases, on the best performances that I get. The only reason why the first measure of the MPI algorithm is not present is that, with the logic that I implemented, it is not possible to divide a 16*16 matrix among 32 processors.

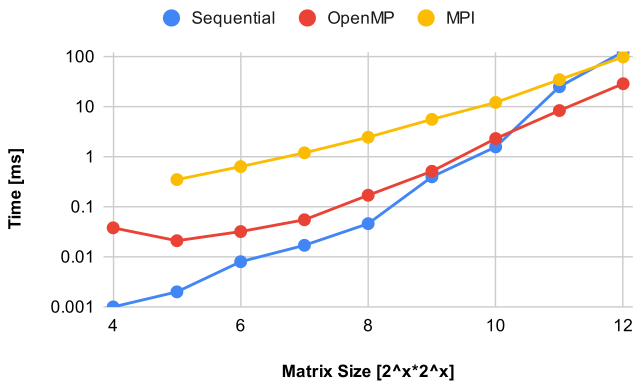


Fig. 1: Performance Analysis of Sequential, OpenMP and MPI-Based Matrix Transposition Algorithms

.

It is possible to see that MPI does not perform well. That is probably due to the fact that scattering the matrix among the processes implies a huge quantity of message passing, which increases the final time. In order to try to have better results, it is technically possible to remove the time needed to share the matrix, however, I decided to avoid that because for a Message Passing Interface it would not make any sense to remove it. On the other hand, looking at the graph as a whole, we can see that the MPI algorithm increases linearly, while the other two algorithms are not that promising. From that we can infer that, by assuming a similar trend with bigger matrices, MPI might actually present a feasible solution.

To have a better understanding of the results in plotted in the graph 1, we can compare the MPI and OpenMP solutions in terms of speedup by using as baseline the sequential code. The formula (1) documents how I processed the data.

$$Speedup = \frac{BaseTime - TestTime}{BaseTime} \cdot 100\% \tag{1}$$

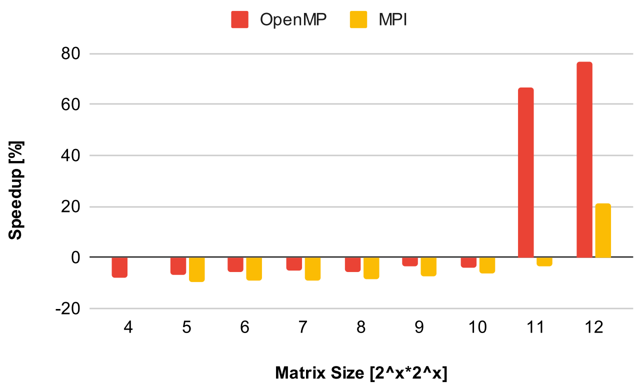


Fig. 2: Speedup of the OpenMP and MPI codes with respect to the sequential baseline

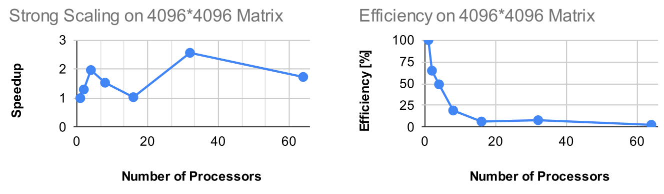
.

From the graph 2 (Please note that the negative part of the graph is in logarithmic scale, so for those values, the speedup is actually much worse than what visually appears) it is possible to see that, as expected, MPI works better than the sequential code only 4096*4096 matrices.

At this point it does make sense to look at the strong scaling and efficiency tests to have a better understanding on how MPI acts by increasing the number of processors and keeping the matrix size fixed. To compute the Strong Scaling I used the equation 2 which puts in relation the base time of the code run with 1 processor and the time needed to perform the computation with increasing amount of processors (namely 2, 4, 8, 16, 32, 64). From that, to obtain the efficiency I divided that result by the amount of processors used as the equation 3 documents.

$$S_{n\_processors} = \frac{BaseTime}{Time_{n\_processors}} \tag{2}$$

$$Efficiency = \frac{S_{n\_processors}}{n\_processors} \tag{3}$$



(a) Strong Scaling (b) Efficiency

Fig. 3: Strong and Weak Scaling for the MPI-Based Matrix Transposition Algorithm

In the image 3 I plotted both the strong scaling graph and the efficiency deriving from it. From the first one 3a it is possible to see that, even if not regular, for small matrices, by increasing the number of processors, we have and actual improvement in the timings. However, from the graph 3b it is immediately clear how the improvement is not worth the fatigue.

Finally I also decided to present the graph for the weak scaling performances. The idea here is to maintain the workload assigned to a processor equal, which translates as increasing the amount of processors as the matrix size increases. In the plot 4 the workload is set constant at 32 rows per processor, and we can see that for small matrices the algorithm performs well, but after a certain point the time quickly increases, which is probably due to the fact that the amount of time needed to scatter the rows and gather them back, again, overweight the time needed to carry out the transposition.

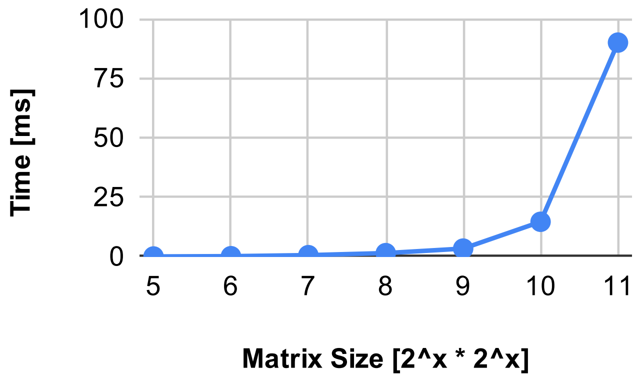


Fig. 4: Speedup of the OpenMP and MPI codes with respect to the sequential baseline

.

B. Presentation and analysis of Matrix Symmetry Check

For what concerns matrix symmetry check, a lot of considerations are actually pretty similar to what I just said for matrix transposition, so the analysis will be much shorter. In the graph 5 is possible to see the comparison between the sequential, the OpenMP and the MPI implementation of the algorithm. Again, as before, for the OpenMP I used 8 threads and for the MPI 32 processes since these two configuration gave be the best results.

Since in this case I used a simple `B_Cast` directive to share the matrix among the processes, I expected much worst results, and in fact, I was right. It is clear, by looking at the plot, that the MPI code is much worse than the other implementations, and additionally it does not seem to converge neither for big matrices.

The strong scaling analysis, that is represented here 6a, has a regular behavior for small matrices suggesting that in this case the computational speed of the processes is able to make up for the mesagge passing required time.

VI. Conclusion

Overall the aim of the project, namely, understanding and exploring MPI, successfully found good result. On the other hand, I have to say that MPI does actually require a lot of

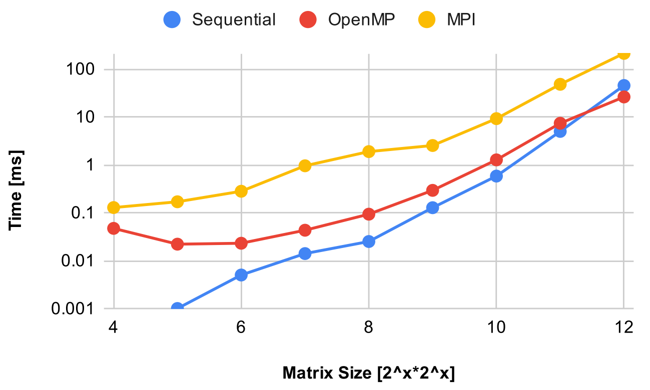
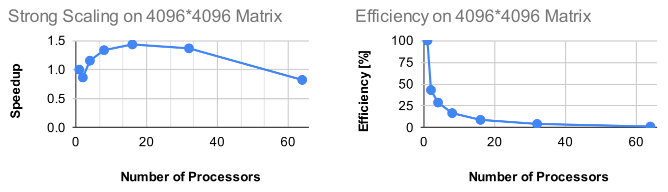


Fig. 5: Performance Analysis of Sequential, OpenMP and MPI-Based Matrix Symmetry Check Algorithms

.



(a) Strong Scaling (b) Efficiency

Fig. 6: Strong and Weak Scaling for the MPI-Based Matrix Symmetry Check Algorithm

planning in advance, before writing the code, and you cannot really tell whether the solution will actually give good results, so when dealing with these type of project, and with little experience, it is maybe better to choose OpenMP, which is much more easy to use, require basically no planning and gives immediate good results.

VII. Git and Instruction for Reproducibility

Here you can find the link to all codes and also the link to all the data I collected.

- You can find the Git Repository **here** with all the instruction to reproduce the code both on the cluster and on a windows machine.
- You can find the Google Sheet **here** with all the results collected throughout the project.

References

[BRCM20] Felix Schumann Bruno R. C. Magalhaes. Efficient distributed transposition of large-scale multigraphs and high-cardinality sparse matrices. https://brunomaga.github.io/assets/Matrix-Transpose/matrix-transposer.pdf, 2020. Online, 03-01-2024.

[Fac24] Davide Facchinelli. Exploring implicit parallelism and explicit parallelism with OPENMP. https://github.com/IGINOI/Matrix_Transposition/blob/main/Facchinelli_Davide_

Matrix_Transposition.pdf, 2024. Online, 21-12-2024.

[Wor10] Brian Worthington. An introduction to hill ciphers using linear algebra. https://sites.math.unt.edu/~tushar/S10Linear2700%20%20Project_files/Worthington%20Paper.pdf, 2010. Online, 21-12-2024.

[Xin24] Xinbiao Gan, Tiejun Li, Feng, Xiong, Bo, Yang, Xinhai Chen, Chunye Gong, Shijie Li, Kai Lu, Quiao, Li, Yiming Zhanhg. Mst: Topology-aware message aggregation for exascale graph processing of traversal-centric algorithms. https://dl.acm.org/doi/pdf/10.1145/3676846, 2024. Online, 04-01-2024.

[YH02] Chris H.Q. Ding Yun He. Mpi and openmp paradigms on cluster of smp architectures: the vacancy tracking algorithm for multi-dimensional array transposition. https://ranger.uta.edu/~chqding/acpi/pubs/sc02e.pdf, 2002. Online, 04-01-2024.