MPI Role in Matrix Transposition

Implementation of two C++ ideas and comparison with other techniques

Deliverable 2 – Introduction to Parallel Computing 146209

Alessandro Luzzani
Dip.Ingegneria e Scienza
dell'Informazione
University of Trento (IT)
Bachelor Degree Student cod.235065
Lodrone, Tn, Italy
alessandro.luzzani@studenti.unitn.it

Abstract— **The aim of this document is to explain and analyse new ways of transposing a matrix in C++ using unique parallelization techniques offered by Message Passing Interface (MPI). The idea starts by considering outcomes included in recent and personal research, as it was evident how in many implicitly parallelized or OMP parallelized projects no possibility of adequate memory management led to noticeable loss in term of effective throughput, due to local memory bound. For this reason, we would like to try redesign some solutions considering new MPI features and see what kind of results we will face, with the goal of reorganizing memory usage and functions trying to obtain and see a difference, if possible.**

I. INTRODUCTION

MPI (Message-Passing Interface) is a message-passing library interface specification. Not only it allows programmers to open parallel environments in their C/C++/Fortran code, but also guarantees a flexible management of data involved in the execution, leaving the possibility of clear and well-structured communication between ranks. MPI has been used lastly in many relevant projects. MPI-2 was fundamental for tracking and simulating stars 'position in the universe (using CINECA Linux Cluster) optimizing calculus of position and accelerations.[3] It was also used for data analysis purposes in healthcare project by KU Leuven University, to process information while developing an algorithm to determine gastric motility [1]. MPI is also versatile, and can run on different domain, like the case of MMPI, which is a derived version of MPI for mobile purposes.[2]
Given this great potential, we want to integrate MPI with global and personal research already presented, in order to find new possibilities for matrices 'parallelization. MPI seems the perfect specification to go on with the development, as it goes beyond other libraries, changing memory management possibilities. MPI contains a set of functions that could be a help in our case. In this project, we have only used square matrixes that can contain floats for simplicity. We will see two MPI transposition methods as well as many other functions that revealed to be fundamental in performance and correctness evaluations.

II. CODE AND FUNCTIONS' IMPLEMENTATION

A. Introduction to the code

I decided to split my job into different parts that all co-operate for the final execution. I developed two .h files, one dedicated to time management, the other (nouned Matrice.h) to matrices creation and matrices' functions. Both .h files can be included in cpp files and their functionalities are widely used in different ways by the main function.

B. Matrice.h , the heart of the executable

Matrice.h is the key file of my solution. It contains everything useful to work on matrixes. Actually Matrice.h is a header file which contains just one class called Matrice, whose attributes are a *static const int* that defines the size and a statically allocated 2d array *m* of floats. The default constructor *Matrice()* and the function *createM()* both are used to create a new matrix when needed, with the first one that allocates the necessary space in the stack and the second one that assigns pseudo-random float numbers in each slot. Also the *createMsymmetric(float x)* function acts similarly, but instead of random numbers, assigns a float number taken as an input. This last function creates matrixes with all same values and can be called for specific reasons.

As matrixes were not dynamically allocated, the compiler must know the size before launching the program. Size can be changed only by opening the code in the .h file in which it is defined. Memory for pbs file customization can be calculated considering each float occupies 4 bytes.

Then there are other general functions that can be used for various reasons such as *stampamatrice()* that, as the name suggests, prints matrixes; a functions that indicates if one matrix is the transpose of another *checkTransposeOf(const Matrice& m2)*, and other two functions that copy the value of one matrix into another given one. Furthermore, there is a relatively big set of functions coming from previous research, that involve serial matrix transposition, implicitly parallelized or OMP parallelized transposition. For details about their implementation or working aspects are the same defined in previous analysis. **[D1]**

C. Timecheck_h and main files MPI managment

The header file timecheck.h is used to compute time by giving a struct named *Timer* with default tools that can be really useful in many occasions. The main file can be personalized and structured in many ways, and it is programmers' responsibility to guarantee the best user experience using desired function calls. * I tried to integrate MPI in the job in the most adequate way: in main files the library *Mpi.h* is included, MPI environment is initialized and closed always in that file too. In Matrice.h, new functions were written in an MPI-friendly logic, who separated cases related to ranks calling the same functions, so that we could avoid MPI issues in the header included that could affect other users including it.

**for further and more specific information see the README on Github*
https://github.com/luciangorie/int.ParallelComputingDeliverable2.git

D. *Transposition with MPI: two ideas and implementations*

I decided to study and develop two different ways of transposing matrixes in MPI that could be working for different necessities.

D.1 DISCOVERING MPI: a base case

In the first transposition experiment, we want to explore MPI general features and well-known functions and see how can be suitable for our context. In a file it was developed a possible solution of distributed matrix transposition. After opening the MPI environment with *Mpi_Init()*, the main process is split in ranks (decided in the pbs by the user), that should be the same as the number of matrix rows. Rank 0 is the only rank to own the full matrix m1. Each rank receives a row of the matrix in a buffer by rank 0. Q is used as index for destination ranks and for splitting the matrix in linear arrays. *Send and Recv* functions are used with *MPI_COMM_WORLD* as communicator. Then, it is called *Mpi_Alltoall*, a communication function that automatically manages to change data between all ranks in a specific way, which can remind a matrix transposal. The job is however not over, as we need to send back again all data in rank 0, as *AllToAll* leaves to each process floats in a dedicated buffer accessible just from owner rank. Then, after sending back rows to the starting point, the matrix is composed again with some nested loops. Int *s* takes at the beginning the size of the matrix and avoid access to not allocated memory.
Although the solution works, this implementation's aim was mainly to explore MPI functionalities, as it is immediately clear how flexibility could not be great, considered that we take one rank and so also one processor, for each row. However, we got a good MPI impression, as it allows us to explore programming aspects that I previously did not think about, and customize communication between ranks.

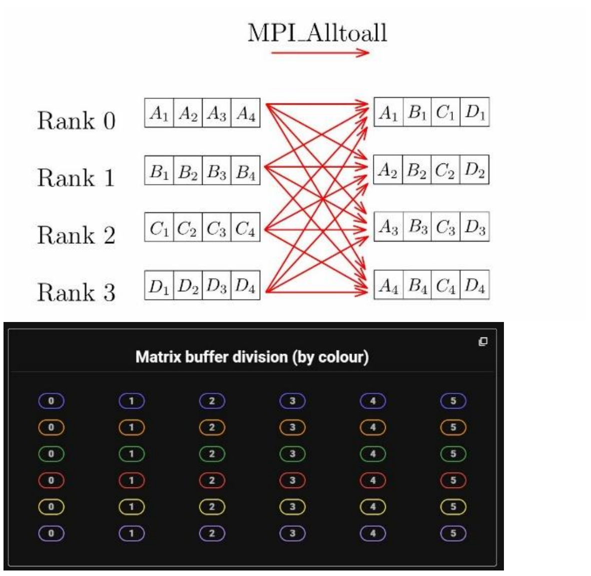
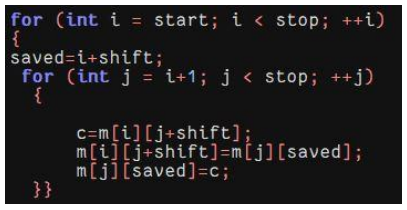


Figure 1:visualization of Mpi AlltoAll and matrix buffer division[8]

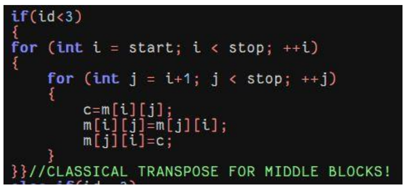
D.2 EXPLORING MPI POTENTIAL: block-based transposition

One of the many well-known transposition methods is the block-based technique. It basically considers a matrix as a matrix of sub-matrices and job of transposing them is relegated to different processes. Block Based transposition can work well with the usage of MPI library, and lets ranks independently transpose their submatrix before collecting data all in one rank (0). The logic idea behind block-based transposition is definitely more refined than the one we have seen before, as it requires more analysis. First of all, the count of blocks can be just a number whose square root is an integer, as they need to build a square in the proper way. So, blocks can be 4, 9,16,25 and more. For implementation reasons, we will just consider 4, 9 and 16 block division, I desired to remove N>=64 as their cost of management could be high, and 36 and 25 as the management could be risky and verbose, although it could be implemented. Size of blocks should also be defined and decided in relationship with matrix size. It has to be a multiple of the square root of the number of blocks. For 4 and 16 we do not face problems when size = 2^n. With the other number of blocks however, we should pay a little bit more attention, by decreasing or decreasing the size to the closest multiple.
Block-Based implementation sees as key point the function call in a main file by all ranks. Considered a matrix m known by all ranks (already shared or after distribution), all parallel executions call the function passing as parameters their id (number) and the block size. Then, inside the function, which is called *BlockBasedN* (with N number of blocks) a series of if clauses customize starting and ending point of nested loops for transposition. In this way, with the same function, each rank transposes its part. Rank 0 is always the one upper left, then the following numbers are assigned to the ranks covering the main diagonal. Other numbers are left for the remaining blocks. *A shift variable is used to change column access. Shift depends on matrix position of the block that should be transposed. The logic inside the functions is the same, regardless the number of blocks involved. Shift is proportional to the distance of the diagonal. It is calculated as **distance*block_size.** It is negative if the block stays in the left side of the matrix and positive in the right. When all functions are over it means that each process transposed its matrix chunk. The main file manages data collections thanks to two functions: *sendsubMatrix()* and *recvsubMatrix()* that contain *send* and *recv* MPI commands to help collecting distributed data in rank 0.



```
for (int i = start; i < stop; ++i)
{
saved=i+shift;
 for (int j = i+1; j < stop; ++j)
  {

      c=m[i][j+shift];
      m[i][j+shift]=m[j][saved];
      m[j][saved]=c;
  }}
```



```
if(id<3)
{
for (int i = start; i < stop; ++i)
{
   for (int j = i+1; j < stop; ++j)
   {
      c=m[i][j];
      m[i][j]=m[j][i];
      m[j][i]=c;
   }
}}//CLASSICAL TRANSPOSE FOR MIDDLE BLOCKS!
```

Figure 2: classic and shifted nested loops for matrix distributed transposition

Matrixes parts are converted into linear vectors and then copied in the matrix staying in rank 0 in the expected position. Ranks that transposed blocks containing the main diagonal are put exactly in the same position they were, while the other blocks are swapped using the classic transposition logic.

E. *CheckSymMPI()*

The function used to check symmetry of a matrix in MPI follows the same main idea of the block-based transposition, as the work of checking symmetry can be divided into blocks. In this case, not all blocks have the same size. Two blocks

**for further and more specific information see the README on Github https://github.com/luciangorie/int.ParallelComputingDeliverable2.git*

including the main diagonal can execute simultaneously (rank 0 and 1) while rank 2 analyses the remaining slots. If at least one function call returns false, the matrix is not symmetric.

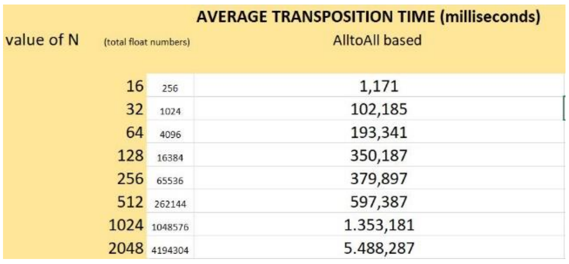
III. EXECUTION AND RESULTS ANALYSIS

Although the portability of MPI should allow the program to run quite everywhere, testing procedures and comparisons reported in this deliverable are all based on different runs on the same system, distributed on multiple nodes of the HPC Cluster open for research purposes to members of the University of Trento. The cluster has plenty of space available for tests as it is composed totally by 65 TB of total RAM, as well as a stunning total theoretical peak performance evaluated to be 478.1 TFLOP/s,. In order to run programs on that platform, we need to open an interactive session or put jobs in queue with a pbs file. They can be specifically customized and adapted for the necessity. In all my pbs there is an additional control and bash print on output files when there is one or some compilation mistakes. Moreover, the Pbs file prints information related to the node.

A. Measuring time and speedup

The first part of testing consists in evaluating individual time of my new developed functions. The first comparison we need is just between mean times of the implementations for both functions (more tries, removing outliers), studying their behaviour while increasing N from 2^4 to 2^{12}. The goal is to have a general overview about the two approaches, and of different block division possibilities. We follow implementations' requests while choosing nprocs and CPUs used during execution: for block based transpose we choose to give one block for each CPU, while for AlltoAll based function we take nprocs as the number of the rows and ncpus at 16.

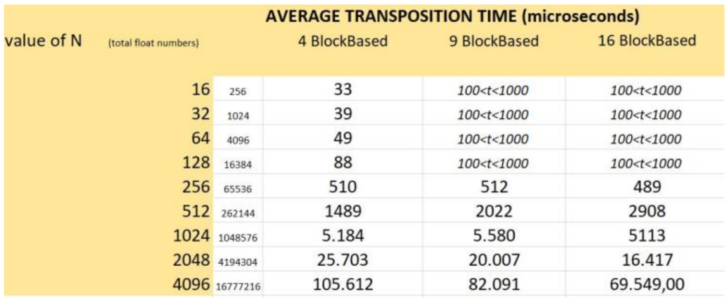
Let's analyse results for *the AlltoAll version* Here is a chart displaying average execution time for transposition calculated from rows divided into ranks.



		AVERAGE TRANSPOSITION TIME (milliseconds)
value of N	(total float numbers)	AlltoAll based
16	256	1,171
32	1024	102,185
64	4096	193,341
128	16384	350,187
256	65536	379,897
512	262144	597,387
1024	1048576	1.353,181
2048	4194304	5.488,287

Figures 3: Average transposition time for AlltoAll technique

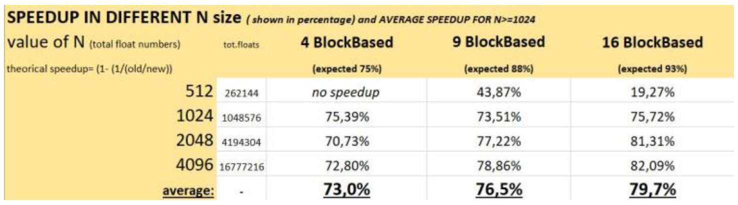
We can clearly see that in all cases time taken is very high. The advantage to have a lot of parallel process does not pay off in small matrices for the high cost of opening and closing MPI environment and in bigger matrices too, as AlltoAll computation cost grows. For this reason, is seems that altough this implementation could be considered intweresting in term of usage of MPI features, it lacks of concreteness, making the function very unlikely to adapt to real application. However, as stated in implementation's description, it was a good possibility in order to explore MPI functionalities, but nothing more in term of results. Execution time of the Block Based Transposition goes better for sure. In fig. 4, we see a chart with the arithmetic mean time taken for transposition in the three functions for different sizes.



		AVERAGE TRANSPOSITION TIME (microseconds)		
value of N	(total float numbers)	4 BlockBased	9 BlockBased	16 BlockBased
16	256	33	*100<t<1000*	*100<t<1000*
32	1024	39	*100<t<1000*	*100<t<1000*
64	4096	49	*100<t<1000*	*100<t<1000*
128	16384	88	*100<t<1000*	*100<t<1000*
256	65536	510	512	489
512	262144	1489	2022	2908
1024	1048576	5.184	5.580	5113
2048	4194304	25.703	20.007	16.417
4096	16777216	105.612	82.091	69.549,00

Figures 4: average transposition time in microseconds for block-based transposition

What jumps immediately into eye, is that there is a huge difference between two N size sectors. Before N=1024, there is no great difference between division in 4, 9 or 16 blocks. That happens mainly for two reasons: environment overhead (specifically impacting in very small sizes, we do not even report some time measurements) and copying procedure, which is quite complex and impacts each block based transposition. The advantage of 9 and 16 blocks division starts to be clear in bigger sizes, but does not strikingly emerge due to the very low arithmetic complexity of transposition procedures. From time measurement we can immediately find effective speedup of all cases. To do so, we need to take serial average time for every size for previous research. [D1] However, the way transposition works with MPI pushes us to change some data we got: in D1 no time for copying the calculated matrix was counted as timer was always stopped before it. For this reason, after taking codes attached to D1 in GitHub, I decided to change the code inside proj1.cpp, stopping the timer after storing the transposed matrix in the new one, and then estimating by outcomes new data. This choice allows us to treat information and re -use it for our research. With new calculated adequate values, speedup is obtained **1-(1/(serial_time / parallelized_time))** and expressed in percentage on chart in fig.5. On the chart we can see an average calculation of speedup for the three cases, as well as speedup expected outcome. Although N=512 is reported, it is not considered for speedup average computation, as it is still too much affected by ranks' management latency.



SPEEDUP IN DIFFERENT N size *(shown in percentage) and AVERAGE SPEEDUP FOR N>=1024*				
value of N (total float numbers)	tot.floats	**4 BlockBased**	**9 BlockBased**	**16 BlockBased**
theorical speedup= (1- (1/(old/new))		**(expected 75%)**	**(expected 88%)**	**(expected 93%)**
512	262144	*no speedup*	43,87%	19,27%
1024	1048576	75,39%	73,51%	75,72%
2048	4194304	70,73%	77,22%	81,31%
4096	16777216	72,80%	78,86%	82,09%
average:	-	**73,0%**	**76,5%**	**79,7%**

B. Obtaining Efficiency

Efficiency is directly calculated from speedup (in integers in this case) analysis results and it is **exp speedup / theorical_speedup.** Most efficient implementation is the one with just 4 blocks, around 92%, followed by the 16 blocks one at 49% and the 9 blocks 47%. It is not surprising, as speedup for 4 block-based implementation was close to the expected one. Results also here come from N>=1000 cases.

IV. COMPARING RESULTS AND FURTHER ANALYSIS

The goal of this last paragraph is to understand the state of what was developed in MPI and leave space of comparison with other techniques using scientifically recognized instruments. If before we had a look on data collection and how to visualize them, now it is time to focus on their interpretation. In total, we have 4 different scenarios: serial, implicitly parallelized, OMP-Parallelized (8 threads) [D1], MPI Parallelized (4 ranks, 9 ranks).

A. *Intervals with T-Student table*

The t-student distribution is very similar to the gaussian one, but it can be distinguished for its heavier tails. It is more suitable in cases in which number of findings is limited, like in our research. .Using N=1000 as costant given parameter , we would like to compare ranges of execution time including 80% of tries.[6]

T-Student Table 80% intervals		
serial	-	**[6084 - 6185] µs**
implicitly parallelized	-	**[4509 - 4626] µs**
OMP parallel	-	**[4360 - 4501] µs**
MPI parallel (4)	-	**[4438 - 4589] µs**
MPI parallel (9)	-	**[5242 - 5802] µs**

Figures 6 :T-Studrnt intervals

The OMP with 8 threads and MPI 4 Parallel versions are two functions that seem to work in best time ranges, sufficiently under 5 ms most of the times. While the serial takes very longer, the implicitly parallelized code is not too far from fully parallel built executions, which is noticeable. MPI version with 9 blocks has some execution difficulties, probably given by the part of converting all blocks into linear arrays.

B. *speedup and efficiency*

		speedUp	efficiency
serial	-	**-**	-
implicitly parallelized	-	**36,80%**	-
OMP parallel	-	**70,50%**	0,42372881
MPI parallel (4)	-	**73,00%**	0,92592593
MPI parallel (9)	-	**76,50%**	0,47281324

All speedup results are not drastically far from expectations. It is interesting to note how MPI version with 4 blocks has more speedup than the one OMP parallelized with 8 threads. As noted before, efficiency of both MPI transposition is not bad, with the one with 4 blocks to be more than 90% making it a very competitive approach.

C. *Scalability and peak performance*

We would like to see the comparison between programs' scalability both strong and weak.[4][5] Scalability testing applies just to parallelization techniques and is the best way to evaluate them. We calculate strong scaling taking a constant N for size and increasing number of parallel processes. With a transformation with **log e ()** of both x (time taken in µs) and y (number of parallel processes) values, we plot a linear descending slope. [7] Closer it is to -1, more strongly scalable is our program. OMP parallelization technique tests very well for strong scalability, with a slope of -0.60. Also MPI parallelization's strong scaling does not totally disappoint, but it works only for high value of N. Its graph shows a slope of ~ -0.25.

Weak scaling testing increases N and parallel processes simultaneously proportionally. Ideal output's slope is very close to 0. N is chosen strategically, considering parallel processes count. * MPI does not have a good weak scalability and it suggests, as seen in previous paragraphs, that less (4) blocks implementation is affected by less latency being a good balanced solution, differently to what 16 and 9 blocks do.

To guarantee visibility, all graphs related to scaling can be found on GitHub.

D. *Memory management and user – related aspects*

Another substantial difference that does not have to be forgotten involves aspects related to memory utilization and the relationship between programs and users' necessities. Speaking in term of memory management complexity, the MPI version is the most difficult to be executed in a homely environment. MPI requires more CPUs distributed among systems with their own RAM and cache, all connected by a performing path. It is something that can cost a lot, being a smart choice just for niche scenarios. OMP parallelization just needs more cores inside the single and same CPU and it operates on shared memory, although execution on multiple cores could block or push to the limit systems on laptops with non-professional architectures. Implicitly parallelized solution involves just one core at a time. However, compilation flags help to reach an impressive speedup still being very user friendly for average commercial devices. The serial version can be the best for old systems or tasks that do not have high priority.

E. *Peak performance*

To calculate best empirical peak performance of all versions we take best-working case scenario and divide FLOPs (which are **3 x number of loops)** by execution time in seconds. Just for MPI implementation we have to consider also copies of data before and after sending floats to rank 0, adding to FLOPs **2*(m.size)^2**.

Peak Performance (in GFLOPs/sec)		
serial	-	**0,51**
implicitly parallelized	-	**1,08**
OMP parallel	-	**2,8**
MPI parallel (4)	-	**0,55**
MPI parallel (9)	-	**0,72**

Peak performance analysis shows how the OMP parallel version gets the best result. [D1] All implementations remain still very far from the system theoretical peak performance, which is estimated to be 59,75 GFLOPs /s per core (478 TFLOPs/s total). This is a clear signal that executions are not affected by computational limit.
On the other hand, it is very likely that we find ourselves in a memory bounded area, in which fast caches are not spacious enough to satisfy all our requests. Sadly, MPI functions do not work as desired, but it reflects what seen before as for bigger blocks (which are functions that should perform the best), whose efficiency did not impress at all.

V. FINAL CONSIDERATIONS AND CONCLUSION

MPI shows clearly great potential and appreciated variety of functionalities that can let programmers chose best options leaving possibilities that are not available with other libraries. However, speaking about our case, results' analysis and interpretation does not amaze in term of performances, especially when multiple ranks are opened. Best and smartest MPI function is actually the one with 4 blocks. Transposition by blocks starts paying off just for very big sizes of N. This suggests, that MPI might be working well when complex or very big amount of data (best If known by all ranks) is processed. For just simple matrix transposition, other methods can be close or overtake MPI solution proposed.

**for further and more specific information see the README on Github https://github.com/luciangorie/int.ParallelComputingDeliverable2.git*

REFERENCES

BETWEEN BRACKETS NUMBERS REFERRED TO REFERENCES.

[1] HPC support in innovative MedTech, Pierre Beaujeana, Carl Menschb and Geert Jan Bexc, Laboratory of Theoretical Chemistry, Namur Institute of Structured Matter, University of Namur, Namur, Belgium

Department of Mathematics, Faculty of Science, University of Antwerp, Antwerp, Belgium

Data Science Institute, Hasselt University, Hasselt, Belgium https://pmc.ncbi.nlm.nih.gov/articles/PMC10357138/#Sx2

visited on 14/01/2024

[2] Parallel Computing on a Mobile Device, https://www.researchgate.net/publication/274373517_Parallel_Computing_on_a_Mobile_Device

[3] FLY: MPI-2 High Resolution code for LSS Cosmological Simulations, U. Becciani, V. Antonuccio, M. Comparato, Cornell University

https://arxiv.org/abs/astro-ph/0703526

[4] Luxe Quality, How to Do Scalability Testing: And Why It Matters? https://luxequality.com/blog/how-to-do-scalability-testing/, visited on 27/11/2024

[5] Gartner, Scalability, https://www.gartner.com/en/information-technology/glossary/scalability#:~:text=Scalability%20is%20the%20measure%20of,application%20and%20system%20processing%20demands , visited on 26/11/2024

[6] Table of T-Student distribution, https://image.slidesharecdn.com/t-distributiontable-170723122602/95/t-distribution-table-1-638.jpg?cb=1500812899

[7] *Following as an example:* Natalie Perlin, Joel P. Zysman, Ben P. Kirtman, Rosenstiel School of Marine and Atmospheric Sciences, University of Miami, Miami, FL, 33149 2 - Center of Computational Science, University of Miami, Coral Gables, FL, 33146, Practical scalability assessment for parallel scientific numerical applications, section III letter E, https://arxiv.org/pdf/1611.01598

[8]

MPI All to All Visualizer https://naeruru.github.io/mpi-vis/ visited on 15/01/2024

[9] (EXTRA SOURCES USED FOR THE DELIVERABLE)

Vella Flavio, L5-Modeling and Benchmarks, Introduction to parallel Computing, Bachelor Degree in ICT Engineering, University of Trento,2024.

[D1]
https://github.com/luciangorie/int.ParallelComputingDeliverable1.git and related PDF and references.

**for further and more specific information see the README on Github https://github.com/luciangorie/int.ParallelComputingDeliverable2.git*