Parallel Matrix Transposition Optimization with MPI

Alessandro Benassi
236027
alessandro.benassi@studenti.unitn.it
https://github.com/ale-bena/MPI-vs-OpenMP-implementations-for-a-parallel-matrix-transposition.git

Abstract— **This paper examines the performance of a square matrix transposition using MPI and compares this approach to the basic sequential one and to a previously developed [1] worksharing OpenMP one. Performance is measured as the effective bandwidth and also speedup and efficiency are computed to provide a better comparison. All of this is done using different matrix sizes and different numbers of threads, excluding the sequential one that has a single thread. This study is conducted at node level due to temporary limitations of the available cluster and this has a negative impact on the results, which show that OpenMP has better performance.**

1. Introduction of the problem and importance

Transposition of a matrix rearranges elements of a two-dimensional array so that rows become columns and columns become rows. As stated in [1] 2D matrix transposition operation is fundamental for many fields of study nowadays. The aim of this project is to investigate and confront the performance of three different implementations: a basic sequential one, an OpenMP and an MPI(Message Passing Interface) implementation. This is done at node level due to temporary cluster limitations and so the expectation is that pure MPI will not be able to exploit all of his potential. Running MPI between different processors on the same node is equivalent to communicating messages using inter-process communication (IPC) between different Unix processes under the control of a single operating system running on the SMP node[3]. So by utilizing it on one node it's important to have the awareness that it is very difficult to overcome the sequential and OpenMP implementations, which fit better on a single node application. So the aim is to exploit the potential of the Message Passing Interface providing a good implementation which then might be tested and developed on a multi-node architecture.

2. State of the art

This document [4] proposes an implementation with an all-to-all communication type, due to the fact that the considered data layout is distributed. This is inefficient for a non-distributed layout and causes performance decrement caused by the communication overhead. It would have been necessary only if the matrix was held in pieces by the process, which is not the case considered in this paper.

The vacancy tracking algorithm seems a very interesting implementation[3], because it avoids using a temporary buffer to compute the transpose operation. This obviously speeds up the program but I have chosen a different implementation since the beginning for the sequential and the OpenMP implementations. The aim is to provide a solid implementation, which avoids possible overlaps and data loss and minimizes the allocated space of the possible needed buffers.

3. Methodology

The program is structured to handle only a particular type of matrices, which are the square ones with dimensions varying between 16 and 4096 as a power of 2. The number of processes is designed also to be a power of 2 starting from 2 and going up to 64. This implementation provides a good range in both of the parameters allowing us to see the behaviour of the algorithm from lower dimension up to bigger ones, with more density in the lower part of the interval.

A. Sequential Implementation

The sequential main program takes in input the dimension of the matrix and the number of times to execute the function; some controls are performed to ensure they are valid values. The times_average file is created and opened in append mode and the header is written. An array for sorting the elements is allocated with size equal to the number of executions. Then there is the main part of the program, where the transpose is executed and the time taken is computed. At each iteration, the case is freed using the function cache_flush(), which allocates a number of elements high enough to ensure that equal elements won't be found and gathered in a faster way by the CPU. Then the matrix and the transpose are allocated in a contiguous way in memory using a pointer to pointer to float, and the first is initialized with random values between 0.0 and 99.99. matTranspose() is called and the time is taken with omp_get_wtime. For small matrices a correctness check is performed. The time is saved in the sort array and is printed on times_table.csv, at the end the memory is freed and the loop restarts. After the loop the array with the times is sorted and the time average is calculated over the central 40% of the values to eliminate outliers and the values are then printed on times_average.csv, which is created with a header line. For the MPI and OpenMP program there is another part that fetches from this file the average of the time for the used size of the matrix and uses it to compute the speedup and the efficiency. This last two parameters are computed with the formulas in 3-E.

The `checkSym()` function verifies if the matrix is symmetric by computing two nested for loops which analyze only the superior part of the matrix, except the central line. The input parameters for this function are a pointer to a pointer to a float, which passes the matrix and an integer that represents the size of this last one. The condition of the first for, which scrolls the rows, is set to $i < n$ where n is the size of the matrix; and the condition of the second one, which is for the columns, is set to $j < i$. Inside of the two loops there is an if construct that returns zero when the element selected is different from its symmetric:

```
if (M[i][j] != M[j][i]) {return 0;}
```

Otherwise, if the matrix is symmetric 1 is returned.
The function `matTranspose()` transposes the matrix. It takes two pointers to a pointer to a float and an integer as input; these are respectively the matrix to transpose, the matrix where it will be transposed and the dimension of the square matrices. The first thing it does is to check whether the matrix is symmetric or not by using the previously implemented `checksym()` function and if this is equal to zero we can proceed with the transpose operation. This process is based on sequentially scrolling all the elements of the matrix with two nested for loops with incremental indexes $i, j < n$ and assigning them to the other matrix with the indexes swapped to exchange the positions.

```
T[i][j] = M[j][i];
```

B. MPI

The program takes in input the same parameters as 3-A plus the number of processes.
The first thing to do when using MPI is to initialize the environment by using the function *MPI_Init()*[6], then it's also necessary to call *MPI_Comm_rank()*[7] to get the rank of each process and *MPI_Comm_size*[8] to get the total number of processes involved. The size is gathered directly from the command line, and the rank is calculated accordingly. See 4 for the explanation of how MPI code is compiled.
The *checkSymMPI()* function takes in input the matrix to check as a pointer to pointer to float, the size *n* and, in addition, it also takes the *rank* and the *size* to exploit the MPI functionalities. When entering the function only rank 0 has the matrix with the values so *MPI_Bcast()*[9] is used to send the matrix to all the other ranks so they can perform the control. To exploit parallelization each rank will check only a part of the matrix corresponding to the number of rows assigned to each rank calculated as $rows\_per\_process = n/size$ and starting from the position $rank * rows\_per\_process$. The variable *local_symmetry* is declared equal to 1 and is changed to 0 when a non symmetric element is found. The nested for loop has this variable in the condition so as when it changes value it exits the cycle. The element check is performed as it was done in 3-A. Then *MPI_Allreduce()*[10] is used to gather back the value from all the ranks, perform a logical AND operation between all the received values and send the result back to every rank on the variable *global_symmetry*, so each of them will have the same result.
For the function *matTransposeMPI()* the first to do is to calculate the rows per process doing $rows_p = n/size$. Then to avoid operating on a symmetric matrix, *checkSymMPI()* is performed, checking if it is equal to 0, meaning that it is not symmetric. After passing this control a temp pointer is declared as null and is allocated only for rank 0. Then *MPI_Scatter()*[11] is performed, passing $rows\_p * n$ elements of *M[0]* from rank 0 to the others, which will receive it on *T[0]*(also rank 0 will receive it). The transpose is then performed using a 1D offset implementation; this consists in a nested for loop, both with the indexes starting from zero, and with the outer one having the condition $i < rows_p$ and the inner one $j < n$. Doing that we are looping to assign the elements so that we have the rows, with a length of *rows_p* instead of *n* and we have *n* rows instead of *rows_p*. The operation inside is the following:
$M[0][j * rows_p + i] = T[0][i * n + j]$;
After that *MPI_Gather()*[12] is performed passing the transposed elements to the temp array of rank 0. So this last one will obtain all the arrays of the processes in order, containing the $n * rows\_p$ matrices in a contiguous way. Now the problem is to put the transposed elements in the correct place. To achive that a cycle with three nested for loops is used, the outer loop is to iterate over the received arrays and so has condition $r < size$, and then the other two iterates over the *n* rows and the *rows_p* elements per row of the submatrices. The elements are assigned to *T[0]* in this way:
$T[0][i * n + j + r * rows\_p] = temp[r * rows\_p * n + i * rows\_p + j]$
After that temp is freed and the function ends. This implementation with *MPI_Scatter()* was chosen because it divides the matrix in blocks of rows, so each process can compute the transpose operation.

C. OpenMP

The methodology for the OpenMP worksharing implementation is explained in [1], since the aim of this paper is to compare its result it is important to specify that the allocation of the matrix has been done like for the sequential and the MPI implementation and slightly differs to the function used in [1] as explained previously in 3-B.

D. Challenges

The first challenge was to avoid using buffers(2) to reduce the allocated space for the MPI, since there were already two allocations in the main for the initial and for the transposed matrices. This means that each rank has its own M and T matrix declared, but it was done in a way that only rank 0 detains the initialized values. The original matrix was reused,as explained in 3-B, it was used to store the transposed elements and send them to *MPI_Gather()*. This was a crucial step because otherwise another buffer would have been necessary in addition to the one used to gather back the values.
Another important challenge was how to retrieve the values back in the *checkSymMPI()* and be sure that all the ranks

had the same result. A first solution to this problem was to use *MPI_Reduce()* and then broadcast the result, but then the *MPI_Allreduce* function was discovered to fit perfectly for the task since it gathers the values, performs the operation and sends them to all the processes.
In *matTransposeMPI()*, after transposing the rows passed to the ranks with the *MPI_Scatter()*and gather them back with *MPI_Gather()*, there was the problem that the obtained matrix was an array with the transposed rows of size *rows_p* one after the other so the result was not correct and there was the need to reorder the matrix as explained in 3-B.
Another problem to solve was to allocate the matrix in a contiguous way in memory to provide to *MPI_Scatter()* an array, which can be easily splitted. This was done by changing the function *declareMatrix()* with respect to what was done in [1]. Now it allocates firstly the pointer to pointer to float, then *M[0]* is allocated with a space of $n * n$:
$M[0] = (float*)malloc(n * n * sizeof(float));$, using a sort of one dimensional logic. To keep the possibility to retrieve the elements of the matrix using the notation $M[i][j]$ a for loop implements the indices from 1 onwards: $M[i] = M[0] + i * n;$. The performance of this implementation is discussed in 5.

E. Metrics

The performance is evaluated for all the implementations with the following: parameters bandwidth, speedup and efficiency. These metrics are all based on the time taken to execute the transpose function, including also the check for eventual symmetric matrices and are computed with the following formulas:

$$\text{Speedup} = \frac{\text{Sequential Time}}{\text{Parallel Time}}$$

$$\text{Efficiency} = \frac{\text{Speedup}}{\text{Number of Threads}} \cdot 100$$

The bandwidth for the sequential and the OpenMP is computed with the same formula since the data transferred is the same:

$$\text{Bandwidth}_{seq/OMP}^{\text{Gb/s}} = \frac{(10n^2 - 4n) \cdot 8}{T_{\text{transpose}} \cdot 10^9}$$

The bandwidth for the MPI implementation is different because we have to take in account for all the transfer of data between the ranks, which happens with the functions: *MPI_Bcast(), MPI_Allreduce(), MPI_Scatter()* and *MPI_Gather*. The formula is the following, Where *n* is the size of the square matrix and *p* is the number of processes:

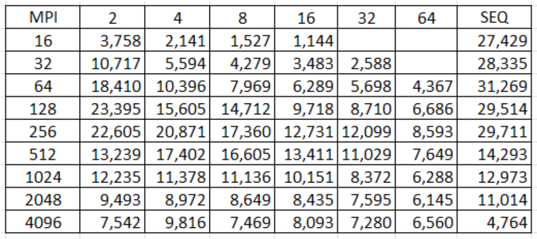
$$\text{Bandwidth}_{\text{MPI}}^{\text{Gb/s}} = \frac{(12 \cdot n^2 + 4 \cdot p) \cdot 8}{T_{\text{transpose}} \cdot 10^9}$$

4. Experimental Setup

Experiments are conducted on a system with the following specifications:

- Architecture: x86_64
- CPU op-mode(s): 32-bit, 64-bit
- Byte Order: Little Endian
- CPU(s): 96
- On-line CPU(s) list: 0-95
- Thread(s) per core: 1
- Core(s) per socket: 24
- Socket(s): 4
- NUMA node(s): 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 cache: 32K
- L1i cache: 32K
- L2 cache: 1024K
- L3 cache: 36608K
- NUMA node0 CPU(s): 0-23
- NUMA node1 CPU(s): 24-47
- NUMA node2 CPU(s): 48-71
- NUMA node3 CPU(s): 72-95

The file are organized in a way that each implementation type(seq, omp, mpi) has a typefunc.h file which stores the libraries and the declaration of the functions, typefunc.c which contains the implementation of these functions and typemain.c where the main function is implemented. This system provides clear division and improves readability compared to a single file approach. The drawback is that we need to include typefunc.c when compiling. To compile and execute the file a .pbs file has been used since it is more practical and compact compared to executing the instructions in an iterative session. The libraries used are the stdio.h, stdlib.h, string.h and omp.h for the sequential and OpenMP implementations, instead for MPI code the last library is substituted with mpi.h. The execution time is measured with *omp_get_wtime()* for the first two implementations and with *MPI_wtime()* for the last. This means that there can be a slight difference in time, but it should not affect the results much. The gcc91 version 9.1.0 toolchain is used to compile the files and there is also the mpich 3.2.1 toolchain for the MPI code. Both modules are loaded in the pbs file, as explained in the readme file. The sequential program is compiled with *-O0* flag to avoid possible improvement by the compiler and it also has *-fopenmp* flag for the time computation. The omp code is compiled with *-O2* and *-fopenmp* flag to have some optimization, *-O2* is used also for the MPI program, along with the *-np* to indicate the number of processors available. In the first section of the provided pbs there are the parameters to run on the cluster, then the modules are loaded and the result repository is created. After that the sequential is compiled so that the others can run correctly, otherwise they will not have the average to compute the speedup and the efficiency. Then the OpenMP and the MPI versions are executed and at the end the files are moved to the result directory. The file *times_table.csv* contains all the times of the executed programs, instead *times_avg.csv* contains only



MPI	2	4	8	16	32	64	SEQ
16	3,758	2,141	1,527	1,144			27,429
32	10,717	5,594	4,279	3,483	2,588		28,335
64	18,410	10,396	7,969	6,289	5,698	4,367	31,269
128	23,395	15,605	14,712	9,718	8,710	6,686	29,514
256	22,605	20,871	17,360	12,731	12,099	8,593	29,711
512	13,239	17,402	16,605	13,411	11,029	7,649	14,293
1024	12,235	11,378	11,136	10,151	8,372	6,288	12,973
2048	9,493	8,972	8,649	8,435	7,595	6,145	11,014
4096	7,542	9,816	7,469	8,093	7,280	6,560	4,764

Fig. 1: Bandwith computed with the formulas in 3-E in GB/s

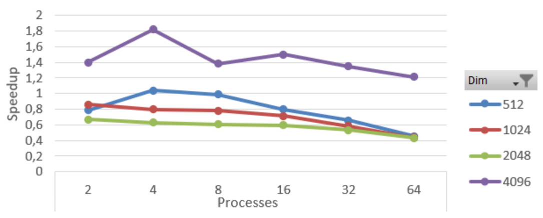


Fig. 2: Strong scaling MPI

the averages with the speedup and the efficiency for each of them, computed with the functions in 3-E. For the MPI the cases in which the process number is larger than the matrix size has been excluded from the execution to avoid errors due to an incorrect split of the matrix between the ranks.

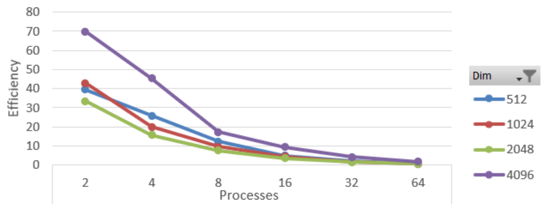


Fig. 3: Efficiency MPI

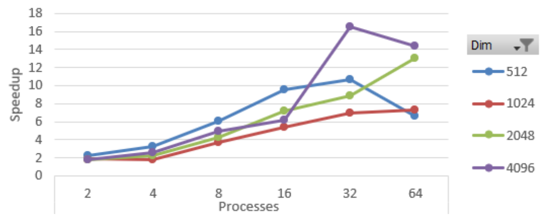


Fig. 4: Strong scaling OpenMP

5. Results and Discussion

From 1 it is possible to observe that the values of the bandwidth for the MPI implemetation are quite low compared to the sequential ones. In particular the fact that peak is in the middle left part of the table means that it works better with lower number of threads in the setup described in 4. Furthermore the fact that the values decreases as the number of processors increases for a fixed matrix size, indicates that communication overhead grows with the number of processors. The drops that happens for larger matrix dimensions can be related to increased data volume and maybe also cache inefficiency. From 2 we can observe the strong scaling with the speedup. The peak is for size 4096 with 4 processes. As the number of processors grows after 4, they all tend to degrade, probably due to communication overhead caused by functions like *MPI_Bcast()* and *MPI_Allreduce()*. For smaller dimensions: 512, 1024, 2048, the speedup remains below 1 even with an increasing number of processes. This indicates that the overhead from adding more processes outweighs the benefits of parallelism. Small problems may not have enough work to utilize efficiently the available processors, indeed for smaller dimensions the numbers were even lower, so the decision was not to include them to provide a clearer vision of the results.

Regarding the efficiency shown in 3, we can see that generally it declines as the number of processes increase and larger matrix tend to have slightly better results. This decline suggests that the program faces important communication costs as the number of processes increases. By looking at 4 we can notice that OpenMP has a different trend. Overall it has higher values and as the number of processes increases the values increases. Taking a look at 2 we notice that for both larger matrices scale better. For space issues I am not able to insert the bandwidth table of the OpenMP, but it has higher values overall and reaches the peak in the middle-right part of the table. Lastly comparing the results also to [3], it is easy to notice that the MPI has lower performance than OpenMP but the values obtained are lower. After analyzing and comparing the different implementation of the *declareMatrix()* function it appears that results for the sequential times compared to [1] are slower especially for bigger sizes of the matrix. This appears to improve the performance at dimension 512 and 4096, while decreasing them for 1024 and 2048(in 2 and 3 the lines for 1024 and 2048 are a little lower). An interesting thing to do would be to modify this function and see how the metrics change. A bottleneck that cannot be removed is the *MPI_Bcast()* in the *checkSymMPI()* function; other than remove it, it may be possibly to exploit it since the ranks will still hold the complete matrix in the *matTransposeMPI()*, so by correctly dividing the rows we could perform the transpose without using *MPI_Scatter()*. Another possible and interesting implementation can be the hybrid one[3][5].

6. Conclusion

The OpenMP implementation is more scalable and efficient, particularly for larger matrices, while the MPI implementation suffers from communication overhead that limits its strong scaling potential. So the poor performance of the MPI implementation is probably determined by the limitation of running on a single node. In this situation MPI works better with a smaller number of threads, while for OpenMP the performance improves with the number of processes.

References

[1] ale-bena, "*Matrix-Transposition-Optimization*," GitHub, https://github.com/ale-bena/Matrix-Transposition-Optimization.git.

[2] E. Bajrović and J. L. Träff, ”Using MPI Derived Datatypes in Numerical Libraries,” in *Recent Advances in the Message Passing Interface*, Y. Cotronis, A. Danalis, D. S. Nikolopoulos, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 29–38.

[3] Y. 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 https://ranger.uta.edu/~chqding/acpi/pubs/sc02e.pdf.

[4] B. R. C. Magalhães and F. Schürmann, ”Efficient Distributed Transposition of Large-Scale Multigraphs and High-Cardinality Sparse Matrices” , vol. abs/2012.06012, 2020. [Online]. Available: https://arxiv.org/abs/2012.06012.

[5] Lorna Smith and Mark Bull, ”Development of Mixed Mode MPI / OpenMP Applications,” in *Wiley online library*, 2001, doi: https://doi.org/10.1155/2001/450503.

[6] MPICH Documentation, ”MPI_Init function,” [Online], Available: https://www.mpich.org/static/docs/v3.3/www3/MPI_Init.html.

[7] MPICH Documentation, ”MPI_Comm_rank function,” [Online], Available: https://www.mpich.org/static/docs/v3.3/www3/MPI_Comm_rank.html.

[8] MPICH Documentation, ”MPI_Comm_size function,” [Online], Available: https://www.mpich.org/static/docs/latest/www3/MPI_Comm_size.html.

[9] MPICH Documentation, ”MPI_Bcast function,” [Online], Available: https://www.mpich.org/static/docs/v3.1/www3/MPI_Bcast.html.

[10] MPICH Documentation, ”MPI_Allreduce function,” [Online], Available: https://www.mpich.org/static/docs/v3.3/www3/MPI_Allreduce.html.

[11] MPICH Documentation, ”MPI_Scatter function,” [Online], Available: https://www.mpich.org/static/docs/v3.1/www3/MPI_Scatter.html.

[12] MPICH Documentation, ”MPI_Gather function,” [Online], Available: https://www.mpich.org/static/docs/v3.3/www3/MPI_Gather.html.

Appendix

A. Code Repository and Reproducibility

The code and instructions for reproducing the results are available at: `[https://github.com/ale-bena/MPI-vs-OpenMP-implementations-for-a-parallel-matrix-transposition.git]`