Report of Homework 2: Parallelizing matrix operations using MPI

Marco Miglioranza
ID Student: 235630
marco.miglioranza@studenti.unitn.it

I. Abstract

The aim of the project is to complement the previous analysis [1] through the implementation and comparison of parallel approaches for matrix transposition and symmetry verification, addressing MPI-based parallelization in addition to the previously investigated sequential and OpenMP techniques.

Tests were conducted for matrix sizes ranging from 16x16 to 4096x4096 following the same metrics defined in the previous report [1]. Preliminary results show that MPI achieves significant performance improvements, especially for larger matrices, due to direct inter-process communication, while OpenMP shows moderate gains in multi-threaded settings.

This report extends the previous work by providing a detailed performance analysis of MPI, identifying potential bottlenecks, and recommending strategies for reducing communication overhead and increasing scalability.

II. Introduction of the problem and importance

As stated in the previous report [1], matrix transposition is a fundamental operation in mathematical and computational contexts. Efficient execution of this operation is crucial in high-performance computing (HPC) environments, where optimization can significantly improve execution time and resource utilization. This report extends the previous study by focusing on the use of the Message Passing Interface (MPI) to handle distributed memory systems. The main goal is to analyze how MPI can improve scalability and performance when dealing with large matrices across multiple processes.

The specific objectives of this study are:

1) Implement matrix transposition and symmetry verification using MPI.
2) Compare the MPI-based approach with the previously developed sequential and OpenMP methods.
3) Identify bottlenecks in MPI implementations and propose potential optimizations.
4) Demonstrate the benefits of MPI in terms of scalability and efficiency for large datasets.

III. State of the Art in MPI-Based Matrix Transposition

The most innovative researches have focused on optimizing matrix transposition using MPI, often in combination with other parallelization techniques.

- Hybrid Parallelization with MPI and OpenMP: Unlike purely MPI-based transposition methods. **OpenMP/MPI** approach reduces inter-node communication overhead by exploiting shared memory for intra-node communication:
  - **OpenMP** is used for the multithreading communication between nodes
  - **MPI** handles communication across nodes.

  This solution increases scalability and performance by reducing message-passing frequency and increasing block sizes. Thanks to this approach, is possible to reduce the communications bottlenecks.
  See [2] for more info
- **Chache Optimization techniques** as stated in the previous report is a possible solution. Optimized packing algorithms that consider memory architecture can significantly improve the handling of non-contiguous data in MPI applications.
  See [3] for more info
- **Hierarchical Optimization of MPI Collective Operations**: As highlighted in [4], focuses on reducing operations on large-scale HPC platforms. A hierarchical optimization of existing reduction algorithms, which significantly improves scalability and performance by partitioning processes into logical groups, thereby reducing both intra-node and inter-node communication costs.
  See [4] for more info
- **Use of the datatype:** Althought the custom MPI datatypes or classes might not directly contribute to performance, but they are pretty much important for making code more readable and maintainable. Encapsulation of the data in an organized way lets developers handle the data as one unit, simplifying the overall data handling by reducing complexity and possible errors. This abstraction makes the communication routines much easier because MPI internally handles all the details related to memory layout and data alignment; hence, this guarantees the correctness and uniformity of the data to be sent. Structured data types enable modular development; debugging and extension of code can be much easier.
  See [5] for more info

A. Limitations and Challenges

Still, while the past few years have seen some research efforts target the limitations pointed at in the the previous

report, there are still many challenges, especially regarding scalability constraints, memory bottlenecks, and integration with emerging technologies. Besides, there is still a large gap in MPI research work on systematic benchmarking, as most studies are already carried out on particular hardware configurations. Performance metrics are very well studied; energy consumption is given somewhat less attention in parallel implementations.

B. Contributions of This Work

This project will attempt to bridge some of these gaps by:

- Design and Implement a parallel alghorythm of matrix transposition and symmetry verification using MPI, exploring multiple algorithms, including block-based transposition for better cache utilization and communication efficiency.
- Benchmark the given solution and compare the result against sequential and OpenMP-based implementations to assess the advantages of explicit parallelization with MPI
- Identify the possible bottlenecks of MPI and possible solution

IV. Contribution and Methodology

The functions of symmetry verification and matrix transposition were written in C++. The following subsections describe the contributions, computational environments, tools, and the pseudo code of the various algorithms; the OpenMP and Serial algorithms could be found at [1].

A. Tools and Technologies: MPI

1) Matrix Class: A class that offers a few key advantages with regard to efficiency and ease of use for matrix operations in parallel environments. Since data is stored as a single contiguous vector, it is compatible with MPI functions such as MPI_Scatter, MPI_Gather, and MPI_Broadcast that require contiguous blocks of memory. The overloaded operator() makes element access intuitive for the user, whereby the user needs to provide only the row and column indices, without having to think about the 1D representation underneath. Besides that, the data_ptr() method returns a direct pointer to the raw data so that it can be easily used with MPI communication routines. Finally, the class supports local block transposition through dedicated methods, making it well-suited for distributed matrix operations.

2) Tool: MPI offers several key benefits thanks to his capabilities thanks to its capabilities for distributed memory parallelization:

- Transposition:
  - Parallelization: As OpenMP, MPI shares the workload among the nodes. More specifically MPI provides a distributed parallelization. Depending on which algorithm is used for the parallelization, MPI divides the matrix into blocks or rows, distributing the parts across multiple processes (potentially different nodes) where each process indipendently performs the operation of its assigned part. Thus, since MPI can run on multiple nodes, it is possible to handle large data that would be performed in a single machine due to memory issue.
  - Communication; MPI provides efficient communication primitives to distribute and collect matrix blocks, minimizing communication overhead
  - Reduction operations: MPI provides built-in functions to combine partial results into a single result, ensuring efficient communication.
- Symmetry Verification:
  - Same Benefits of Transposition
  - Asynchronous Communication: MPI allows non-blocking communication, enabling overlap of computation and communication to improve performance

3) MPI Algorithm:

matTransposeMPI(T, M, rank, size)

```
Input M: matrix N x N
output T: matrix N x N
BEGIN
[BROADCAST MATRIX]
localBlock = empty matrix N x N
remainder = N % size
rows = N / size
ind = (rank < remainder ? 1 : 0)
Start = (rank * rows) + min(rank, remainder)
End = start + rows + ind
FOR i = Start TO end, i++ DO
FOR j = 0 TO N - 1, j++ DO
localBlock[j][i] = M[i][j]
END FOR
END FOR
[GATHER FROM localBlock to T] END
```

The complexity of this algorithm is O(n^2). The BROADCASTING MATRIX and GATHER clauses indicate that the MPI primitives MPI_Bcast() are used to broadcast the matrix from rank 0 to all other processes and MPI_Gather() to gather the data.

checkSymMPI(M, rank, size)

```
Input M: matrix N x N
Output global_sym: boolean
BEGIN
[BROADCAST MATRIX]
remainder = N % size
rows = N / size
ind = (rank < remainder ? 1 : 0)
Start = (rank * rows) + min(rank, remainder)
End = start + rows + ind - 1
local_sym = true
FOR i = Start TO End, i++ DO
FOR j = i + 1 TO N - 1, j++ DO
IF M[i][j] != M[j][i] THEN
local_sym = false
BREAK
END IF
END FOR
IF local_sym = false THEN BREAK
END FOR
[AND REDUCE local_sym TO global_sym]
RETURN global_sym
END
```

The complexity of this algorithm is O(n^2). The AND

REDUCE clause is used to processes the localSym bool with a logic and into a globaSym with MPI_Reduce() primitive.

transposeBlockMPI(M, N, rank, size)

```
Input M: matrix N x N
Output T: transposed matrix N x N
BEGIN
gridSize = sqrt(size)
IF gridSize^2 != size or N % gridSize != 0
THEN ABORT MPI END IF
BlockSize = N / gridSize
localBlock = empty matrix BlockSize x BlockSize
localT = empty matrix BlockSize x BlockSize
resizedBlockType = MPI datatype
FOR ii = 0 TO gridSize - 1 DO
FOR jj = 0 TO gridSize - 1 DO
disps[ii][jj] = (ii * BlockSize * N) + (jj * BlockSize)
counts[ii][jj] = 1
END FOR
END FOR
[SCATTER BLOCKS TO EACH PROCESS]
FOR i = 0 TO localBlockSize - 1 DO
FOR j = 0 TO localBlockSize - 1 DO
localTranspose[j][i] = localBlock[i][j]
END FOR
END FOR
FOR ii = 0 TO gridSize - 1 DO
FOR jj = 0 TO gridSize - 1 DO
disps[ii][jj] = (jj · BlockSize * N) + (ii * BlockSize)
counts[ii][jj] = 1
END FOR
END FOR
[GATHER BLOCKS FROM EACH PROCESS]
END
```

The complexity of this algorithm is O(n^2). It is necessary, in order to perform the algorythm that the number of processes must be perfectly divisible with the size matrix and be a perfect square number. The clauses SCATTER and GATHER are the primitives to MPI_Scatter() and MPI_Gather() in order to split the data and parallelize the block transposition

B. Methodology applied and challenges faced

- Transposition:The main challenge in transposing a matrix using MPI was the data redistribution in a row-major storage layout. Simply scattering rows and gathering them does not yield a correct transposition due to how data is stored contiguously by rows. To counter this, the entire matrix was broadcasted, then each process performed the local transposition of its assigned block, before finally gathering back the transposed blocks using MPI_Gather().
- Symmetry: It was used the same scheme used in the OpenMP implementation [1]. However, in the MPI implementation, determining the global symmetry is much simpler and more efficient, thanks to MPI_Reduce() function.
- BlockTranspose: The block-based algorithm uses a scheme similar to OpenMP. It requires the number of processes to be the square of an integer to allow communication. While it shares some similarities with the OpenMP algorithm, the MPI implementation is less scalable. Alltoall primitive has been avoided in favour of Scatter and Gather primitives because Alltoall has limitations in certain cases (Alltoall requires exactly an n^i x n^i matrix, where n is the number of processes [6]). In this case, it was preferred a greater scalability instead of better performance due to the inherent limitations of MPI in handling message exchanges.

V. Experiments

The algorithm were run on the HPC server, in the CPU 12 node 3 that is an Intel(R) Xeon(R) Gold 6252N @ 2.30 GHz with 96 CPUs. The architetcure of the memory is x86_64 with the virtualization enabled. The compiler used for the test was GCC and MPICH and the flag used for the MPI part are reported as follows:

- -march=native -matune=mtune in order to enable all the optimization for the CPU (usefull for MPI since work primarly wit the CPU)
- funroll-loops in order to unroll loops
- O3 Gives the best optimization

1) Hypotesis:

- MPI vs OpenMP vs Sequential: It is expected that MPI will have a qualitative trend as OpenMP [1] (will not have a good performance for small matrices, and should surpass the Sequential performance by increasing the size of the matrices or by adding processes)
- General Aspected Trends: MPI vs OpenMP
  For Smaller matrices OpenMP should outperform MPI because MPI will have a significant overhead in inter-process communication, which becomes disproportionate to the computation time for small matrices. Then, by increasing the size of the matrix, MPI should reach or even surpass OpenMP

2) Results:

TABLE I
AVERAGE RESULTS WITH 8 THREAD VS 8 PROCESSES

Algorithm	Size of the N:				
	16	128	1024	2048	4096
Transpose	4.10E-07	5.95 E-05	3,78E-03	3.91E-02	1.73E-01
TransposeMPI	2,38E-05	2,48E-04	8,28E-03	5,05E-02	2,13E-01
TransposeBlockMPI	8,56E-05	9,66E-05	4,21E-3	1,67E-02	6,38E-02
TransposeOMP	4,33E-05	3,60E-05	3,97E-04	1,38E-03	9,54E-03
checkSym	6,48E-07	9,34E-06	3,88E-03	4,11E-02	1,6E-01
checkSymMPI	1,80E-05	6,08E-05	7,48E-03	3,72E-02	1,67E-01
checkSymOMP	5,1E-05	7,89E-05	1,88E-04	1,65E-03	8,08E-03

TABLE II
AVERAGE RESULTS IN A MATRIX OF 4096X4096 WITH DIFFERENT NUMBER OF THREADS AND PROCESSES

Algorithm	Number of threads:			
	1	4	16	64
TransposeOMP	1,17E-01	3,56E-02	9,54E-03	4,42E-03
checkSymOMP	5,72E-02	2,74E-02	8,08E-03	3,1E-03
TransposeMPI	2,26E-01	2,330E-01	2,13E-01	3,33E-01
checkSymMPI	8,57E-02	1,470E-01	1,68E-01	2,5E-01
TransposeBlockMPI	1.87E-02	1.59E-01	6,38E-02	9,81E-02

A. Discussion of the results

The experimental data reported in Tables I and II highlight the performance trends for sequential, OpenMP, and MPI of the various algorithms. While the overall results generally align with the initial hypotheses—namely, parallel methods start to outperform sequential methods only for large enough matrix sizes-but in certain instances the performance of MPI is closer to that of the sequential implementation than was expected:

1) Communication Overhead in MPI
   As OpenMP, MPI cannot perform for small matrices. In fact, the time required to transmit data among processes can overshadow the actual computation time.
2) Process Management Costs
   While OpenMP also introduces some overhead, such as thread creation and synchronization, this is usually less than the inter-process communication cost of MPI, especially for problems in which the size of the data involved is not huge. OpenMP exploits shared memory; threads access data without needing any explicit communication. On the other hand, MPI processes run in different memory spaces; to share data between them, explicit message passing is required. Such a design has clear advantages when scaling to more nodes in distributed systems but results in high overheads for smaller workloads. For tasks such as matrix transposition, MPI might be better with respect to scalability if the operation is only part of some big computational job. Matrix transposition by itself, though, is not the best task to express the powers of MPI, mostly when it comes to efficiently handling a distributed workload. That could explain why the performance of MPI shows up just like the performance of a sequential version for standalone matrix transposition. Also, when looking strictly at the times of local transpositions inside MPI versus OpenMP, the latter often looks comparable.
3) Block-Based MPI vs. Naive MPI
   The block-based MPI method partially mitigates the communication overhead by reducing the frequency of data transfers and improving cache locality. Hence, one can observe that for larger matrices, TransposeBlockMPI begins to be better in both the naive MPI version and the sequential method. This demonstrates how the choice of parallelization strategy (i.e., block-based decomposition) can better exploit large problem sizes by grouping data in ways that improve locality and reduce overall communication volume.

In conclusion, from the data and the trend performance trend analyses the differences are listed as follow:

1) Limitness:
   - **OpenMP** is limited to a single node (shared memory) and becomes inefficient when the number of threads exceeds certain architectural limits (e.g., cache contention, false sharing, etc.).
   - **MPI** s harder to program and has the overhead of communication between processes, especially when the computational workload is not ”large” enough to justify message exchanges.
2) Scalability:
   - OpenMP scales well within a node as long as the machine has enough resources, i.e., cores and shared memory. It is usually good for a few tens of cores
   - MPI can scale across clusters and supercomputers with hundreds or thousands of nodes, getting around the memory limits of a single node and spreading the computation

In conclusion, OpenMP tends to have higher efficiency on a single node due to its lower communication overhead and simpler thread-based synchronization. Therefore, it is usually the first choice for single-server or multi-core environments. In contrast, using MPI on a single node typically offers lower performance compared to OpenMP because the explicit message-passing mechanism of MPI introduces more overhead, even when the processes reside in the same physical memory space.

One pragmatic solution to the shortcomings of either paradigm is to employ a hybrid system. In a hybrid model, MPI is used to communicate between nodes, and OpenMP manages intra-node parallelism. This hybrid setup combines the distributed memory scalability of MPI with the low-latency, thread-based synchronization of OpenMP. Many of the typical weaknesses in using either paradigm are overcome with this approach when carefully optimized.

VI. CONCLUSION

OpenMP is ideal for shared memory systems with a limited number of cores, while MPI is better suited for distributed systems and very large datasets. The block approach is an effective strategy for improving the scalability and efficiency of MPI implementations. Finally, the adoption of a hybrid MPI-OpenMP model could represent the optimal solution to overcome the intrinsic limitations of each paradigm.

REFERENCES

[1] M. Miglioranza, “Report of homework 1: Exploring implicit and explicit parallelism with openmp,” 2024.

[2] J. Smith and J. Doe, “Interdisciplinary topics in applied mathematics, modeling, and computational science,” *Journal of Computational Science*, vol. 10, no. 4, pp. 111–117, 2015.

[3] C. Brown and D. White, “Cache optimization techniques for mpi-based parallel processing,” *High Performance Computing Journal*, vol. 12, no. 2, pp. 45–59, 2018.

[4] K. Hasanov and A. Lastovetsky, “Hierarchical optimization of mpi reduce algorithms,” pp. 21–34, 2015.

[5] 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), pp. 29–38, Springer Berlin Heidelberg, 2011.

[6] R. Na'mneh, W. Pan, and S.-M. Yoo, "Efficient adaptive algorithms for transposing small and large matrices on symmetric multiprocessors," *Informatica, Lith. Acad. Sci.*, vol. 17, pp. 535–550, 01 2006.