Performance Comparison of Serial and Parallelized Matrix Transposition with OpenMP and MPI

Cornel Grigor, id° 235986
Department of Information Engineering and Computer Science
University of Trento, Trento, Italy
cornel.grigor@studenti.unitn.it
https://github.com/cornelGrg/IPCdeliverable2

***Abstract*—The testing aims to revisit the original program of a matrix transposition [1] by implementing another version based on MPI's routines (*Message Passing Interface*) that parallelize the code through message passing. Multiple performance metrics were evaluated and a final comparison with OpenMP was conducted. Considering the program's behavior under specific circumstances, a discussion about the system's bottlencks was provided.**

***Index Terms*—Matrix Transposition, Parallel Computing, OpenMP, MPI, Performance Analysis**

I. Introduction of the problem and importance

It's important to remember the importance of the matrix tranposition operation and how making this base operation faster is very significant for many applications [1] . There are multiple ways to implement this algorithm but in this paper I'll be discussing a simple naive implementation that's been parallelized both with OpenMP and MPI to see how they compare and how they behave. While both are libraries used in HPC, OpenMP is more of a straight forward approach where we let the library manage most of the parallelism done with the machine's threads, MPI is completely the opposite as it leaves the actual process managing and inter-process communication up to the programmer who carefully needs to choose and implement the correct directives for the desired result (explicit parallelism) [2] [3]. MPI programs formally run with multiple processes but they're commonly referred to as processors since for maximum performance, typically the ideal ratio is one process per physical CPU core (processors) [3].

II. State-of-the-Art

The current state of the art of OpenMP was previously discussed in the last paper [1].

A. MPI

While technically MPI is considered a standard, there exist several open and closed source implementations of this standard. It is currently the industry's de facto standard for parallel programming and it is used in many supercomputer clusters to run any kind of numerical simulation to take advantage of the distributed architecture [3]. The recent introduction of MPI-4 improves GPU and heterogeneous system support [4] [5]similarly to what we previously saw with the newer versions of OpenMP [1].

MPI's success is also derived from the fact that it always maintains backward compatibility for codes even across multiple new architectures over the year. Although it's already succesful in this, a new level of MPI compatibility is arising thanks to recent research done on standard Application Binary Interface (ABI) that allows for even more flexibility, such as using the same MPI implementation for different software without the need to recompile. [7]

B. Limitations

Some limitations still remain and some are new with the up-trending use of heterogeneous systems that utilize GPUs as hardware accelerators, these environments require special data treatment like custom data types that allow for optimal conversions [5]. Custom data types also allow for further optimization especially in non-contiguous memory layouts [6].

C. Identifying the gap

My project aims at identifying what could be some of the main bottlenecks of the tested implementations, this is done by doing some performance analysis that test efficiency and scalability with both weak and strong scaling.

III. Contribution and Methodology

This project works on integrating the previous research [1] by providing real-world benchmarks that compare OpenMP with MPI and different approaches in the context of a squared matrix transposition across different test cases and configurations by providing performance data and analyzing it.

A. Methodology

For this project I used the Naive out-of-place transposition algorithm [1] from the last project as the base algorithm, the OpenMP implementation remained exactly the same but for MPI I have tested three different approaches:

1) Per row divided and gather approach [M1]
2) Scatter local transpose with gather on shared memory [M2]
3) Scatter local transpose with final reduction [M3]

Consistency between the approaches was kept in mind and to comply with this, each MPI version was tested by only tracking the time of the actual transpose computation without tracking the time that it takes to send the matrix data to each rank and the time to gather it in the end. This way we get a time that is comparable to the OpenMP and sequential version because otherwise it would be unfair for MPI since sending and receiving the data between ranks is very expensive and due to the simple nature of our program this ends up being very time consuming compared to the computation itself thus making the results unclear. In a real more complex MPI application this cost would be spread much better across the program. All versions utilize the same versions of the *checkSym()* function and the *matTranspose()* function in this form:

- *checkSymSEQ()*
- *matTransposeSEQ()*
- *checkSymOMP()*
- *matTransposeOMP()*
- *checkSymMPI()*
- *matTransposeMPI()*

The *checkSym()* checks if the matrix that was generated is symmetric since the transpose of a symmetric matrix is the matrix itself. Every MPI approach uses the same *checkSym()* function.

Before starting to explain the actual approaches it's important to state that for MPI, all the functions work by receiving flattened vectors as parameters instead of the Bi-Dimensional Vector that get used for SEQ and OMP methods. The same **M** matrix is used for all methods but before calling the MPI functions, the matrix gets flattened by row-major ordering. This is done because it allows us to store the matrix in a contiguous way in memory [8] thus we can send contiguous blocks with MPI which is usually faster (when possible) and simpler as it doesn't require the use of derived data types [9].

B. Symmetry check

As mentioned earlier, all the OpenMP implementations remain the same, including the algorithm used for the *checkSymOMP()* [1].

When running an MPI program with multiple processes, we refer to each process during run time as a *rank*. We assume rank 0 to be the *master* as most support functions of the program like saving data to csv, printing the results and initializing the matrix are executed by rank 0.

Before calling the function, rank 0 *broadcasts* the initialized matrix to all the other ranks, then every rank calls the functions where inside each one gets assigned a range of indexes, *start_row* and *end_row*, of the matrix on which to compute the actual check. The function then returns an integer that gets treated as a boolean, in the end a *"product all reduction"* is computed on the integer value, this way all the ranks know if the check resulted positive or not before calling the transpose function. The workload is equally divided in rows, each rank will have the same problem size to compute. Before executing, the program checks that the required configuration of threads and matrix size will allow for an equal distribution of the data, otherwise the test case gets skipped.

Algorithm 1 Checks if Matrix is Symmetric [MPI]

```
Ensure: true if the matrix is symmetric, otherwise false
  for $i = start\_row$ to $end\_row$ do
    for $j = 0$ to $n$ do
      if $mat[i*n+j] \neq mat[j*n+i]$ then
        return false
  return true
```

To keep data consistent with the last project, only the matrix transpose computation time was tracked.

C. MPI Transpose Implementation

For the MPI transpose, three main approaches were tested but only the *Per row divided and gather approach [M1]* was deemed successful as the observed performance of the other two, while still mathematically correct, were not as as efficient, especially when considering the total time (including scatter and gather times of the matrices).

For approach [M1], the same philosophy of workload division of the *checkSymMPI()* function was applied where every rank is assigned a range of indexes to compute the transpose. After every rank has completed their local transposed matrix, a *gather* operation on rank 0 is performed.

Algorithm 2 Per row divided and gather approach [M1]

```
  for $i = start\_row$ to $end\_row$ do
    for $j = 0$ to $n$ do
      $local\_trans[(i - start\_row) * n + j] \leftarrow mat[j * n + i]$
```

Approach *Scatter local transpose with gather on shared memory [M2]* works by *scattering* parts of the original matrix (equally distributed) from rank 0 to all the other ranks, each rank then writes on a *MPI shared window* it's piece of the matrix but with the column and rows reversed, the end result is a complete transposed matrix stored on the shared window.

The last approach *Scatter local transpose with final reduction [M3]* is similar to [M2] where rank 0 scatters pieces of the matrix to all the ranks, each rank computes and saves the local transposed matrix inside a vector, in the end a *sum reduction* between all the local transposed matrices is performed on rank 0, the result is a correct transposed matrix.

To ensure the transpose correctness, every **T** transposed matrix is checked by using a function [1].

IV. Experiments and System Description

A. System Specifications

The testing was done on the same HPC cluster that UniTN provided for this experience like last time but I ended up using different nodes due to availability, all the data for the sequential and OpenMP implemenetations was updated accordingly to the new nodes. The main data testing was performed on an Interactive session on the `short_cpuQ` queue. I used nodes that rely on 4x `Intel Xeon Gold 6140M` (72 cores total) processors from the "Skylake" family of Intel's processors running the x86-64 architecture connected using 4 NUMA nodes. Each processor has 24.75MB of L3 cache so in total 99MB of L3 cache then each core has 1MB of L2 and 32KB of L1 cache. The cluster runs on a linux environment and to compile the programs the *mpicxx* compiler wrapper based on *gcc-9.1.0* from the *MPICH 3.2.1* MPI's implementation was used. Stack memory limit is set to unlimited in the cluster's nodes.

B. Experimental Setup

The different configurations involved testing the matrix size from 2*2 to 4096*4096 so the testing scenarios remain the same [1]. For the OpenMP and MPI implementations data was acquired for multiple thread/processes configurations ranging from 2 and increasing in powers of two up to 64. Due to how the workload gets split in the MPI implementation, a maximum process number of 64 was chosen since most nodes of the *short_cpuQ* only went up to 96 cores (and not 128 to allow an equal division), this is to make the matrices equally divisible for simplicity.

C. Hypotesis

Since we already did a performance evaluation for OpenMP, we expect the MPI implementation to behave similarly. We will be looking at execution times, speedup and efficiency graphs as these will provide us with enough information to describe the actual parallel performance of these implementations, these performance indexes will get analyzed for strong scaling and weak scaling scenarios. Strong scaling speedup is expected to grow with larger matrices as the performance increase gained with multiple processes overcomes the extra overhead caused by the process management. Bandwidth is also computed using the following formula:

$$\text{Bandwidth (GB/s)} = \frac{\text{Total Data Transferred (GB)}}{\text{Execution Time (s)}}$$

$$\text{Total Data Transferred (GB)} = \frac{\text{Matrix Elements} \times 4\,\text{bytes} \times 2}{1024^3}$$

Note that 4bytes is the size of a float and we multiply by 2 because the data gets read and then written in the algorithm.

D. Performance Graphs and formula used

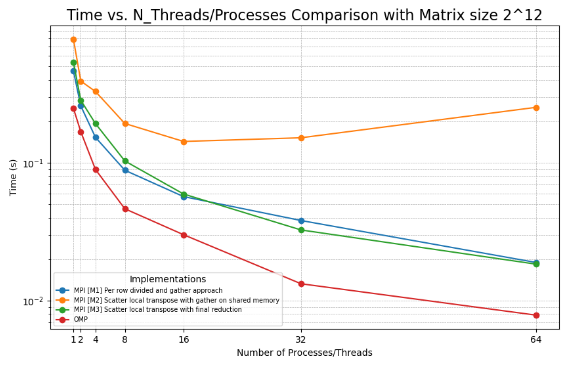

figure (a)

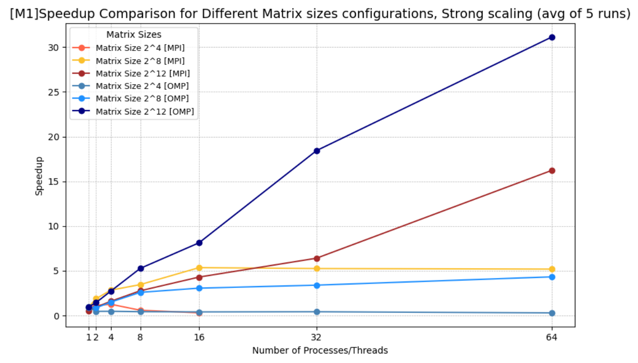

figure (b)

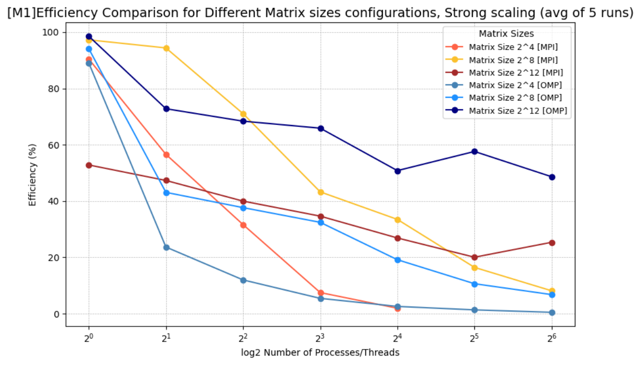

figure (c)

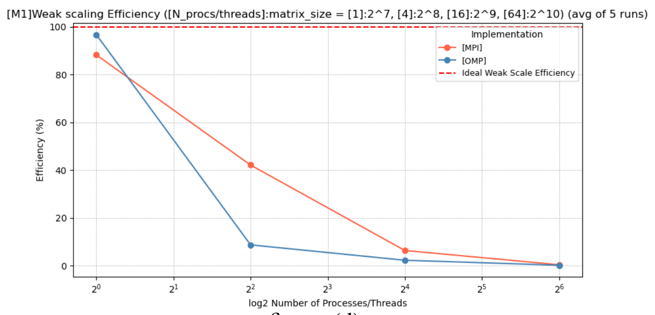

figure (d)

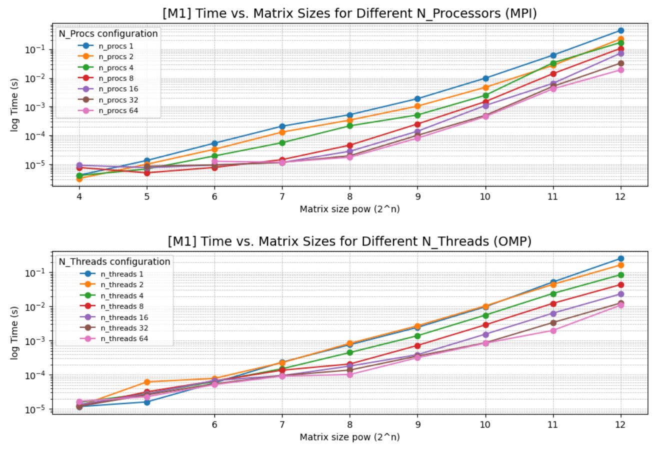

figure (e)

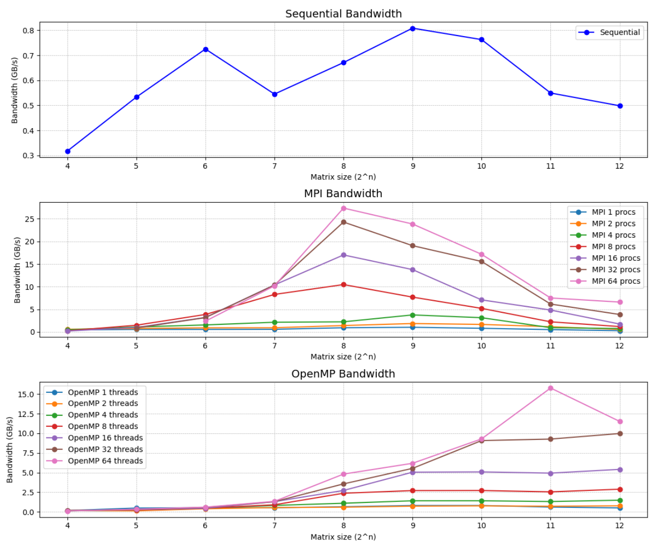

figure (f)

E. Analysis of the Performance Graphs and Comparison

Before analyzing the data it's important to state that all the data was plotted using a python plotting program and the data about speedup and efficiency is relative to the best serial time of each specific case.

The main analysis was done on method [M1] because, as we can see from the comparison graph, it was deemed the most efficient and scalable **[fig.(a)]**. In the case of [M2] we can assume that the shared memory has reached its maximum bandwidth limit when all the processes attempt to write on it. For [M3] we see a similar performance to [M1] because the main computation algorithm is very similar but when I tested for the entire time including the scatter and the final reduction it was a lot more inefficient than [M1]. With high amount of processes and big matrices the cost of this operation grows very fast (remember that the data used to generate the graphs was gathered only tracking the transpose computation time). Due to MPI's nature it's also important to point out that these data trends apply to this specific MPI implementation as small changes in the flow of MPI routines can lead to completely different results.

As expected, the strong scaling speedup for [M1] increases as the matrix size grows, as shown in **[fig.(b)]** . However, since the speedup does not scale linearly with the number of threads or processes, the efficiency decreases, as illustrated in **[fig.(c)]**. This indicates that while larger matrices benefit more from parallelization, the efficiency of resource utilization diminishes as more threads or processes are added. In strong scaling we know that the problem size per process grows proportionally to the number of processes, while having multiple processes is more costly to manage, when the problem size becomes big enough, this cost is overcome and we see an improvement in execution times **[fig.(e)]**.

For the weak scaling tests [M1], since the problem size per process needs to remain constant, due to the matrix sizes that I tested, the number of elements inside the matrix quadruples for every increase of its side expressed as a power of two, this means that to keep a constant amount of elements per process, we also need to quadruple the number of processes for each increase in matrix size to the next power of two. Below is a table showing the exact test cases I've done.

TABLE I: Weak Scaling Analysis Test Cases **[fig.(d)]**

Matrix Size ($2^n \times 2^n$)	**Number of MPI Processes**	**Elements per Process**
$2^7 \times 2^7$	1	16384
$2^8 \times 2^8$	4	16384
$2^9 \times 2^9$	16	16384
$2^{10} \times 2^{10}$	64	16384

In the graph **[fig.(d)]** we see that weak scaling efficiency degrades very quickly and is poor overall. Most likely this is caused by the increased communication overhead that we can see even with a small amounts of processes.

Analyzing the bandwidth graphs **[fig.(f)]** we can see that the bandwidth generally grows when increasing the number of processes/threads but it seems that, both for OpenMP and MP, once it reaches a maximum peak at a certain matrix size, it starts decreasing. This can be explained by the memory hierarchy since up to a certain point the data might fit into lower level cache, like L1 or L2 cache, but once it gets big enough it's possible that it doesn't fit in it anymore thus leading to lower bandwidth due to more frequent memory accesses. It's important to point out that adding more processes/threads will eventually stop the rising trend of bandwidth because of the added communication overheads that will add latency and thus result in worse resource usage. This is exactly what we see in recent CPUs developments where the growth of core counts in CPUs has exceeded the growth in memory bandwidth and this significantly limits performance [10].

V. Conclusions

This scenario demonstrates how the high flexibility of MPI programming can sometimes be counterproductive if not managed well. We should also consider that the ease of implementation with OpenMP is much higher than MPI at the cost of offering less fine-tuning to the programmer, in this particular case, OpenMP still remained more efficient in most tested scenarios. It's clear that MPI implementations are much more complex and take much longer to implement than OpenMP ones, in favor of better control over the multiple processes that hopefully lead to better performance thanks to the added flexibility and more granular control that can be used to overcome some of the main bottlenecks if addressed correctly.

Git and Instructions for Reproducibility

In the Git repository below you can find the code tested with any details about reproducibility:

https://github.com/cornelGrg/IPCdeliverable2

References

[1] Cornel Grigor, ”Performance Comparison of Serial and Parallelized Matrix Transposition Using Different Parallelization Techniques”, 2024.

[2] ”MPI: The Complete Reference” di Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, e Jack Dongarra.

[3] https://en.wikipedia.org/wiki/Message_Passing_Interface

[4] https://www.mpich.org/2022/01/21/mpich-4-0-released

[5] https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

[6] Byna, Gropp, Xian-He Sun and Thakur, ”Improving the performance of MPI derived datatypes by optimizing memory-access cost,” 2003 Proceedings IEEE International Conference on Cluster Computing, Hong Kong, China, 2003, pp. 412-419

[7] Jeff Hammond, Lisandro Dalcin, Erik Schnetter, Marc PéRache, Jean-Baptiste Besnard, Jed Brown, Gonzalo Brito Gadeschi, Simon Byrne, Joseph Schuchart, and Hui Zhou. 2023. ”MPI Application Binary Interface Standardization”. In Proceedings of the 30th European MPI Users' Group Meeting (EuroMPI '23). Association for Computing Machinery, New York, NY, USA, Article 1, 1–12.

[8] https://en.wikipedia.org/wiki/Row-_and_column-major_order

[9] Eijkhout, Victor. (2018). Performance of MPI sends of non-contiguous data. 10.48550/arXiv.1809.10778.

[10] J. Langguth, X. Cai and M. Sourouri, ”Memory Bandwidth Contention: Communication vs Computation Tradeoffs in Supercomputers with Multicore Architectures,” 2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS), Singapore, 2018, pp. 497-506, doi: 10.1109/PADSW.2018.8644601. keywords: Bandwidth;Multicore processing;Benchmark testing;Message systems;Computational modeling;Sockets;Instruction sets,