A Comparison of MPI Two-Sided and One-Sided Communication Performance

Junaid Dawood (1094837), Xongile Nghatsane (1110680), Marissa van Wyngaardt (719804)

***Abstract*—This report presents a performance comparison of MPI's two-sided and one-sided approaches to inter-process communication. This performance comparison is made with reference to implementations of a parallel block-based matrix transposition algorithm which scales with the number of processes used. Results from the testing conducted suggest that the two-sided approach offers superior performance for sufficiently large matrices. In addition, it is found that the use of a collective I/O based scheme is a viable alternative to the two main strategies presented within this report, regardless of the communication strategy used.**

I. Introduction

Message Passing Interface (MPI) is a message passing standard which is designed to function on a diverse range of parallel computing architectures [1]. Message passing typically achieves parallelism through messages passed between processes in order to transfer data. Popular implementations of the MPI standard for message passing include OpenMPI and MPICH [2], [3].

MPI typically facilitates the use of the 'single-program-multiple-data' (SPMD) parallelism model; as all nodes essentially run the same program. However, this is not required as many MPI implementations allow for multiple executable to exist within one MPI job.

This report presents a performance comparison of two MPI-based implementations of block-based matrix transposition [4], parallelised amongst many processes. These implementations differ in the type of communication used: the first makes use of MPI's typical two-sided communication, using *Send* and *Recv*. The second implementation uses MPI's one-sided communication [5], making use of *Windows* and *Get*.

II. Background

MPI's communication model is typically point-to-point in nature. That is, inter-process communication wherein both the receiver and sender are specified. That said, MPI also allows for *broadcast* and *receive-from-any* communication. MPI communication is typically two-sided i.e. it consists of distinct send and receive stages. For every send there must be an associated receive (if data is to actually reach its destination). Depending on the MPI implementation, a process may need its sent message to be acknowledged by a receiver, through a *Recv* call, before it can continue with the rest of its processing.

In addition, MPI does allow for one-sided communication, which was included in the MPI-2 standard [5], [6]. One-sided communication removes the necessity for the acknowledgement that exists within the conventional two-sided approach. Essentially, one-sided communication consists of making a *Window* of a process' address space accessible to other processors; with the host process facilitating the opening and closing of the *Window*. Other processes can then write or read from the *Window*, as long as it remains open. This removes the need for the host process to wait for acknowledgements from those processes which require data that it holds.

A. Problem Description and Success Criteria

The brief problem statement included in the introduction must be expanded so as to fully describe the nature of this research. The matrix transposition implementation is intended to work on large data sets; which are to be stored, out-of-core, in external files. Note: matrices are stored in row major order within binary files for all implementations; individual matrix items are of the *short* datatype.

This transposition implementation must be parallelised using the two MPI approaches mentioned earlier i.e. both one-sided and two-sided communication.

As described in the introduction, the algorithms discussed in this report pertain specifically to ‘block-based’ transposition algorithms. This essentially entails splitting a large matrix into equally sized submatrices, transposing these submatrices, and performing the appropriate submatrix swaps.

Assuming there is a $N \times N$ matrix A which can be written in terms of its $K \times K$ submatrices:

$$A = \begin{bmatrix} W & X \\ Y & Z \end{bmatrix} \tag{1}$$

it can be proved that:

$$A^T = \begin{bmatrix} W^T & Y^T \\ X^T & Z^T \end{bmatrix} \tag{2}$$

This can be generalised to any square matrix which can be split into submatrices of equal size. However, this research focuses on square matrices of dimension 2^x by 2^x, where x is an integer satisfying $x \geq 3$. The block-based approach is limited to a minimum submatrix dimension of 2×2, as there is no smaller meaningful block size. That said, it is likely that splitting input matrices into such small submatrices will involve larger communication overhead; leading to worse performance. However, this is more formally discussed later in this report.

III. Conventional Implementation

This implementation of the block-based transposition can be discussed in 4 stages: 1) Generation of matrix data, 2) Performing per-process transpositions, 3) Transmission of data to the master node, and 4) Writing of results to a file by the master node. Essentially, each process generates its own submatrix, transposes it, and passes the transposed matrix to the master process, which subsequently writes the entire transposed matrix to a file.

A. Generating Matrix Data

Matrix data is generated on a per-process basis, through typical random number generation techniques. Each process generates its own, equally sized, submatrix. The size of the submatrices is determined by the desired size of the overall matrix which is being transposed. If the number of available processes is so large that an even distribution would yield submatrices smaller than 2×2, then some processes are left out of the transposition entirely, in order to impose the 2×2 minimum submatrix size. Similarly, processes will be excluded to ensure that each process only ever performs a transposition on a single square submatrix.

B. Performing Submatrix Transpositions

Each process, upon completing its submatrix generation, then performs a transposition on said submatrix. This submatrix transposition is done using a naive approach, which is parallelised using OpenMP [7]. The naive approach entails the typical per-element row-to-column coordinate transformation swaps.

C. Data Transmission

Within this two-sided communication approach, all participating processes will send their submatrices to the master node (rank 0) upon completing their respective transpositions. This is done through typical *Send* and *Recv* semantics. Essentially, the master node will receive each submatrix as a contiguous buffer, storing all of the received submatrices within a single, larger, contiguous buffer which is held locally.

D. Writing of Transposed Matrix to File

The master node is responsible for writing the transposed matrix to an output file. This is implemented using the *File-writ*e and *File-write-at* functions. The master node performs the necessary calculations for transforming its continuous buffer representation of the matrix to the final row-major ordered contents of the output file. This involves calculating write offsets for each submatrix and writing these submatrices at the locations specified by said offsets. Each row of each submatrix is written consecutively; applying an offset equivalent to the size of the original matrix (N), after writing each row.

IV. One-Sided Approach

The implementation of the one-sided approach is largely the same as the conventional *Send* and *Recv* implementation. The one-sided implementation differs only in the way in which data is exchanged between processes and the way in which data is written to the output file.

A. Generating Matrix Data

Like the two-sided approach, submatrices are generated on a per-process basis. As before, processes are sometimes excluded to maintain a minimum block size of 2×2, or to maintain square submatrices.

B. Performing Submatrix Transpositions

For fairness, the one-sided approach does not attempt to improve on the parallelised naive approach used in the two-sided implementation.

C. Data Transmission

This implementation makes use of MPI's one-sided communications for data transmission. All participating processes open a *Window*; placing their local submatrix buffers within their respective *Windows*. The master node then performs *Get* operations on the contents of all these windows, such that it acquires all of the transposed submatrices required to construct the overall transposed matrix. The submatrices obtained through the Get operations are stored within a single contiguous buffer on the master node.

D. Writing of Transposed Matrix to File

Like before, the master node is responsible for writing the contents of a contiguous buffer to an output file. However, this approach makes use of an MPI derived type in order to do so. Specifically, the MPI *Vector* derived type is used to set a *file view* for the writing of each submatrix to the output file. The stride length used for this derived data type is equal to the row length of the overall matrix N, the block length is equal to the row length of the submatrices k, and the count is equal to $N \times k$, as shown in figure 1. Constructing the file view in this way allows each submatrix to be written to the file within a single write call; unlike the I/O strategy used in the two-sided approach.

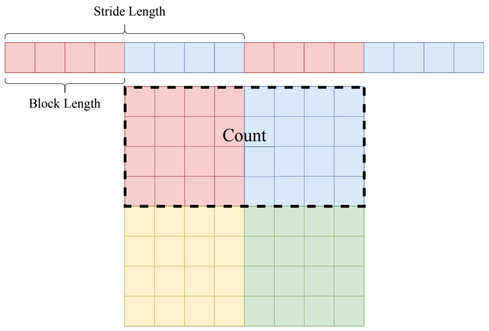

Fig. 1: Derived type file view mapping.

V. Alternative Collective I/O Implementations

Two alternative approaches are implemented to demonstrate the use of collective I/O in MPI [8], [9] i.e. *Read all* and *Write all*. It is thought that these implementations will be able to highlight the bottleneck which occurs when a single node is responsible for performing I/O. These two alternatives differ in that one makes use of one-sided communication and the other, two-sided. This is done to afford fair, independent, comparisons of the use of collective I/O with the implementations discussed earlier.

A. Input Data Generation

Both of the collective I/O alternatives source their matrix data from an external file. This file is read from using a non-contiguous *file view*: these *file views* are consistent across all involved processes, except for the offsets used. The involved processes make use of a single collective *Read all* call to populate their local submatrix buffers with values read from the file.

B. Submatrix Transposition

Submatrix transposition remains unchanged from the parallelised naive approach discussed earlier. This ensures fairness of comparison between the alternatives and the two original approaches.

C. Data Transmission

Since the goal of these alternatives is the use of collective I/O, neither of the two alternatives require the sending of data to a master node for I/O purposes. Thus, processes need only swap their transposed submatrices, with those of corresponding processes. The mapping of these swaps is described by the diagram shown below: there is essentially a block-level transposition which occurs in this process.

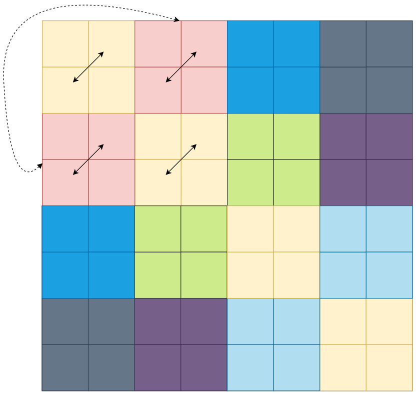

Fig. 2: Mapping of inter-process block swapping.

Of course, one of the alternatives does this through typical MPI *Send* and *Recv* commands whilst the other makes use of *Windows* and *Get* operations.

D. Writing Output Data

This step is very similar to the initial input data reading step: once a process has obtained its corresponding block data, it is able to write this block to the output file. This is done through the use of a derived data type (*Vector*) used to create a *file view* such that each process can write its entire submatrix to a file within a single collective *Write all* call.

It should be noted that the offsets and filetype used to define file views used when writing to the output file are the same as those used when reading data from the input file, for any given process. The following two diagrams contrast the nature of the collective I/O approaches and the two approaches discussed earlier.

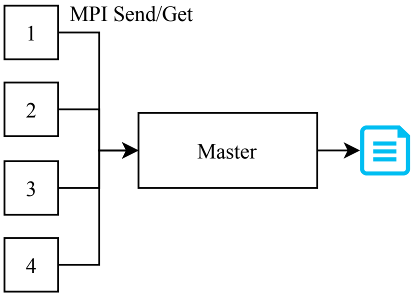

Fig. 3: General structure of two main approaches.

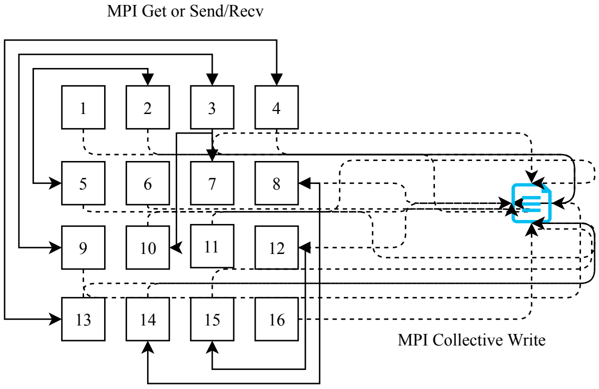

Fig. 4: General structure of two collective approaches.

E. Limitations

Since both of these approaches rely on evenly distributing blocks amongst threads, they are incapable of catering to scenarios in which the number of available processes is greater than the number of 2 × 2 blocks which can be yielded from the input matrix.

VI. Testing Environment and Procedure

The implementations were sequentially tested on a cluster with NFS support. The number of processes used includes 16, 32, and 64. Input matrices ranged in size from 8 × 8 to 128 × 128. Combinations of process count and matrix size were constrained to comply with the minimum 2 × 2 block size for the two collective I/O alternatives discussed earlier. In addition, work divisions which do not yield square submatrices were not used for these two approaches.

VII. Results and Discussion

This section contains the results of benchmarking the performance of the implemented approaches. This includes comparisons relating to the number of processes used for a single implementation, as well as inter-implementation comparisons. In addition, results are shown for some tests in which OpenMP compiler flags were not included i.e. tests in which intra-process transpositions ran serially.

It is clear from figures 5 and 6 that there is a lack of scaling of performance with respect to process count. This is thought to be as a result of the somewhat small input matrix sizes used. That is, the communication overhead incurred as a result of the higher process count is not outweighed by the extra computing power available, due to the limited complexity of the work done by each process.

In a similar vein, the use of OpenMP to parallelise the intra-block transpositions incurs a significant overhead considering the small size of the individual submatrices. This is confirmed by the graph in figure 7.

A performance comparison which considers slightly larger matrix sizes can be seen in the figures 8 and 9. These results surprisingly confirm the trend observed within the original result set with respect to scaling with the number of processes used. In addition, disabling OpenMP within these tests yielded superior performance, similar to earlier results.

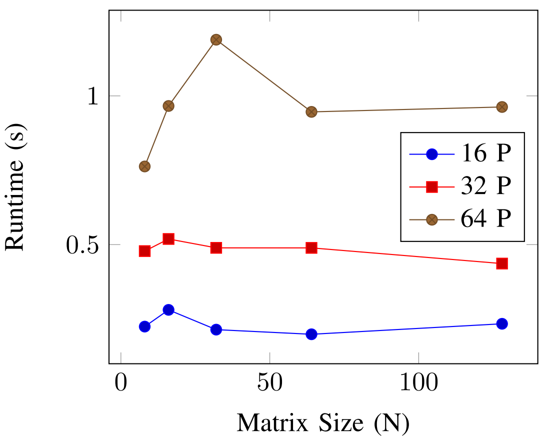

Fig. 5: Performance of two-sided approach (OpenMP enabled).

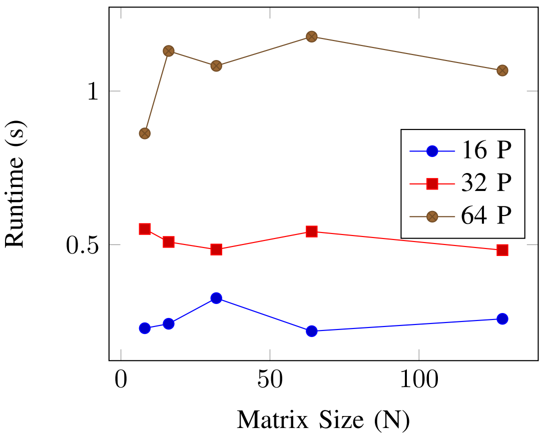

Fig. 6: Performance of one-sided approach (OpenMP enabled).

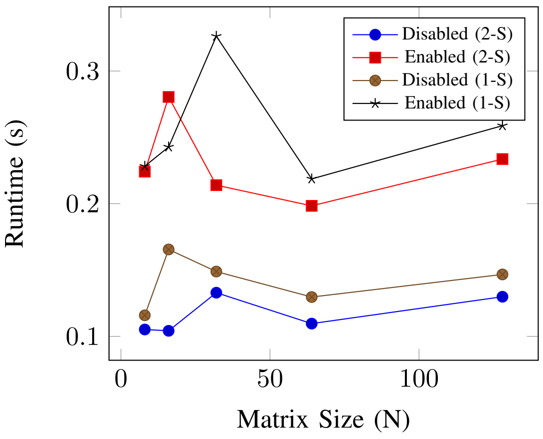

Fig. 7: Performance comparison between OpenMP disabled and enabled for 16 processes.

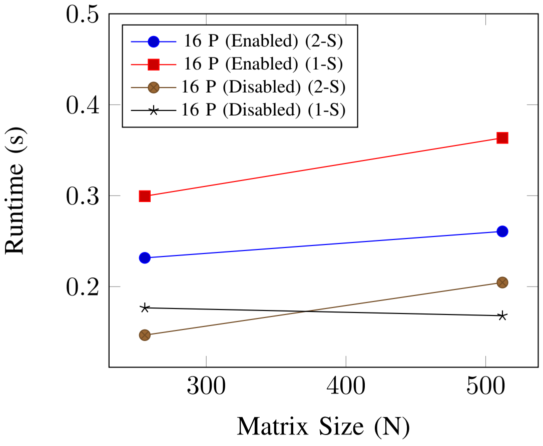

Fig. 8: Performance on larger input matrices on 16 processes.

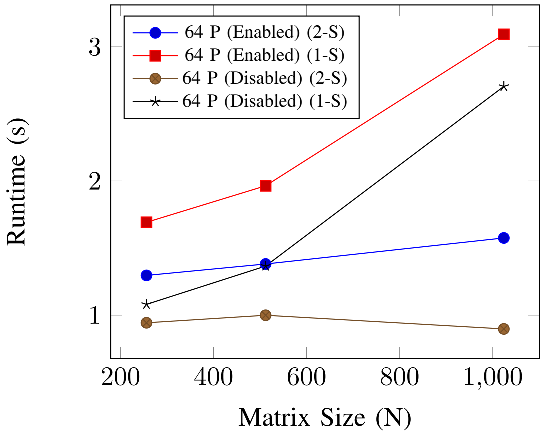

Fig. 9: Performance on larger input matrices on 64 processes.

From the gathered results, there is seemingly a significant performance increase granted by the use of a conventional two-sided approach, for sufficiently large matrix sizes. This is surprising given the motivation for the use of one-sided communication discussed earlier. However, this is thought to be due to the presented one-sided implementation not exercising the non-blocking nature of one-sided communication in a meaningful way. That is, the master node still waits for the submatrix data from all involved process before writing the submatrices to an output file.

The following two graphs contain the results of the final area of testing, with respect to the two collective I/O alternative implementations discussed earlier. These tests also involved considerations of the overhead incurred through the use of OpenMP.

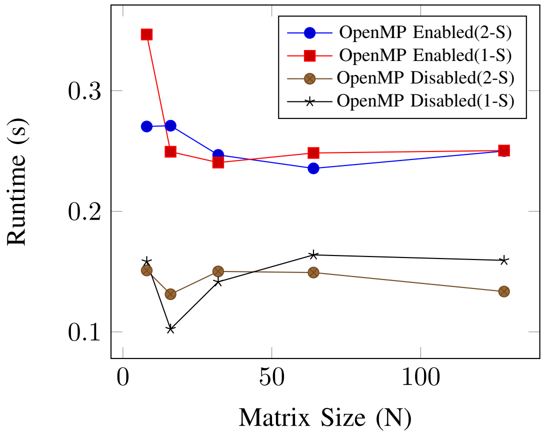

Fig. 10: Performance of Collective I/O alternatives for 16 processes.

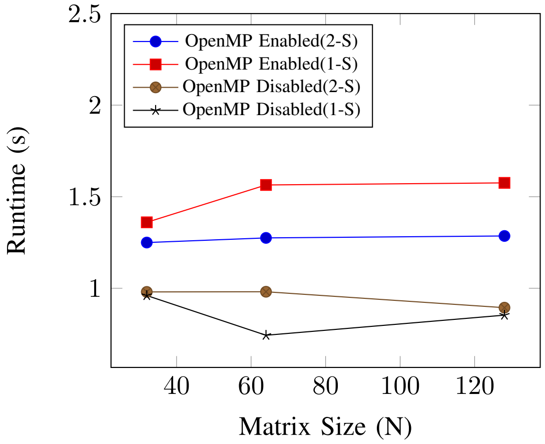

Fig. 11: Performance of Collective I/O alternatives for 64 processes.

The collective I/O alternative tests largely confirm the trends observed within the core performance analysis with reference to scaling, the use of OpenMP, and the communication method employed. One notable aspect is the similarity of the actual performance of the collective I/O approaches and the original two implementations. This is despite the fact that the collective I/O implementation reads input data from an external file, which would presumably involve greater latency.

VIII. Critique and Future Improvements

This section discusses the possible critiques that could be made of the presented implementations. Where possible, potential solutions or improvements are described alongside these critiques.

A. *No Redundancy in Implementations*

None of the presented implementations incorporates fail-safe functionality which would ensure that all jobs are completed despite the loss of a process or node. This is primarily because of the implicit mapping which has been used in all implementations; which links submatrices to individual processes. A more robust approach would be to make use of a job queue coordinated by a master process; wherein jobs are taken by idle processes and ultimately removed upon completion. If a process were to become unavailable, then the job of that process could be re-added to the queue, such that another process would perform said job. This redundancy would allow for the matrix transposition to be completed in the event of a loss of processes.

B. Inefficient Work-Sharing

The job queue strategy mentioned above would also be of help with respect to the ease with which fair work sharing can be achieved.

There are two drawbacks in the current work sharing implementation: 1) the exclusion of processes to maintain at least 2×2 submatrices, and 2) the exclusion of processes to maintain square submatrices. The exclusion of processes is, of course, only an inefficiency for significantly large matrices. That is, based on the results of the tests conducted this is not really a concern for small matrices.

A job queue strategy would do little to assist with the issue of maintaining a minimum submatrix size. However, with respect to the second issue, the use of a job queue would guarantee that more of the requested process actually performed a task within the MPI job. This would, however, be at the expense of the use of smaller submatrices, whereas excluding processes obviously has the opposite effect. Thus, further testing would be required to verify whether a performance increase is actually granted by this alternative, as there may be overhead incurred due to the use of smaller submatrices.

C. Inefficient Implementations of Submatrix Transpositions

As discussed earlier, the algorithm used to transpose the submatrices is a parallelised naive approach. Given the size of the input matrices tested, this approach achieves sufficient performance. Moreover, a more complex approach would likely involve increased overhead, outweighing the performance benefits for small submatrices. That said, for scalability, the submatrix transpositions could instead make use further block partitioning wherein the block transpositions and swaps are parallelised.

D. Scalability for Extremely Large Matrices

One of the common use cases for distributed computing in big data applications is the inability of any one process to store all of the input data at any point in time, due to the size of said data. The two main approaches presented in this report require the master process to gather all of the submatrices, thereby storing them in the memory associated with a single process. Obviously, there are concerns related to extremely large matrices as a result of this implementation.

The collective I/O approach does away with the need for this centralisation of the input data. Therefore, these approaches are more applicable for very large datasets. Of course, there may be cases in which the input data is so large that even storing entire submatrices is problematic. Such scenarios require increased partitioning of input data; splitting submatrix transpositions into multi-stage operations.

E. Unnecessary Data Sharing Between Processes

It could be suggested that the inter-process data sharing found in both of the collective I/O implementations is superfluous. Realistically, neither of these implementations actually require inter-process data sharing since processes could simply write their transposed submatrix to the output file, as collective I/O is used in any case.

That said, it would be hard to achieve a fair comparison between the collective I/O approach and the two main approaches if the collective I/O approach did not incorporate any data transfer between processes.

IX. Conclusion

Two matrix transposition implementations have been presented. These implementations make use of a block-based approach using MPI, incorporating either one-sided or two-sided communications. The two-sided implementation was found to achieving superior performance in the tests conducted. Additionally, a collective I/O approach was found to be a viable alternative to the two presented implementations, especially if reading matrix data from a file is required. The two main implementations which have been presented in this report could be improved in terms of the redundancy they achieve and, in addition, the efficiency of the submatrix transposition algorithms they incorporate.

References

[1] M. P. I. Forum, *Mpi: A message-passing interface standard*, https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf, (Accessed on 04/05/2019),

[2] *Getting help*, https://www.open-mpi.org/community/help/, (Accessed on 04/05/2019),

[3] *Mpich overview — mpich*, https://www.mpich.org/about/overview/, (Accessed on 04/05/2019),

[4] W. Gropp, *Lecture 7: Matrix transpose*, http://wgropp.cs.illinois.edu/courses/cs598-s16/lectures/lecture07.pdf, (Accessed on 04/05/2019),

[5] ——, *Lecture 34: One-sided communication in mpi*, http://wgropp.cs.illinois.edu/courses/cs598-s16/lectures/lecture34.pdf, (Accessed on 04/05/2019),

[6] W. Gropp and R. Thakur, "An evaluation of implementation options for mpi one-sided communication," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, Springer, 2005, pp. 415–424.

[7] *Home - openmp*, https://www.openmp.org/, (Accessed on 04/05/2019),

[8] W. Gropp, *Lecture 32: Introduction to mpi i/o*, http://wgropp.cs.illinois.edu/courses/cs598-s16/lectures/lecture32.pdf, (Accessed on 04/05/2019),

[9] R. Thakur, *Parallel i/o and mpi-io*, http://www.training.prace-ri.eu/uploads/tx_pracetmo/pio1.pdf, (Accessed on 04/05/2019),