Matrix Transposition using Parallel Computing

Anita de Mello Koch - 1371116 **Nicholas Kastanos - 1393410** **Brendon Swanepoel - 601949**

School of Electrical & Information Engineering, University of the Witwatersrand, Private Bag 3, 2050, Johannesburg, South Africa
ELEN4020 Data Intensive Computing in Data Science

Abstract: This paper investigates the use of MPI Derived Data Types to parallelise a matrix transposition algorithm using MPI-3. The developed algorithm was run on two different machines for multiple matrix sizes. Each run of the algorithm was timed to test its efficiency. It was found that slightly over subscribing the number of ranks within the machine produced the optimal results. This is evidenced by transposing a square matrix of dimension 5120 took 42.725550 s when run on 9 ranks and 29.130977 s when run on 5 ranks for a 4 physical-core machine.

Key words: Big Data, Matrix Transposition, MPI, Parallel Computation

1. Introduction

Matrix transposition is an important part of many calculations performed on large data-sets and is used in several different applications within the fields of Science and Engineering. For very large matrices, this operation is expensive to compute due to the number of elements within the matrix. This poses a problem for calculations involving very large data-sets as the transposition is only a single step in the calculation. The speed of this transposition can be increased through parallelisation, particularly over several computers.

Background information on parallel matrix multiplication can be seen in Section 2. An example showcasing the use of parallel matrix transposition can be seen in Section 3. An analysis discussing the performance of the implemented algorithm is given in Section 5. Finally a conclusion on the findings is presented.

2. Background

This project deals with the use of parallelisation to compute the transpose of square matrices of varying sizes. These matrices are generated and stored in a file using a separate matrix generation algorithm which populates them with random values ranging from 0 to 99. Two ways in which this problem can be solved are discussed below.

2.1 Literature Review

Two methods in which the problem can be solved are by using a Message Passing Interface (MPI) or UPC++, which implements the PGAS model.

MPI defines a model for which message passing interface libraries should be developed. MPI-3 is not a library but rather a specification for libraries, a message-passing parallel programming model. Data in one process is sent to a separate process through cooperative operations on each process [1].

Currently, MPI is a popular choice for parallel programming. It is the only message passing library that can be considered a standard, so programs written using MPI can be compiled for different platforms without rewriting the code. Vendors are able to optimise the running of programs by taking into consideration native hardware features. It also has a large functionality base with over 403 routines defined in MPI-3 [1].

There are a number of libraries that have implemented the MPI standard. Three of these libraries are shown below in Table 1.

UPC++ is a C++ library which supports Partitioned Global Address Space (PGAS) programming. This model is a Single Program, Multiple Data (SPMD) architecture in which each thread of execution has access to its own private memory as well as the global address space [2].

It was designed with three main objectives in mind. The first was to provide an object-orientated PGAS model in the context of C++. The second was to increase the parallel programming idioms onto the functionality of its predecessor UPC. The final objective was to provide a simplified interface between existing parallel programming systems [3].

Table 1: MPI Libraries

MPI Library	**System**	**Compilers**
Open MPI	Linux	GNU, Intel, PGI, Clang
Intel MPI	Linux	GNU, Intel
MPICH	Linux	GNU, Intel, PGI, Clang

Both MPI and UPC++ allow for parallel communication by supplying methods which are able to both send and receive data between processes on single, or multiple, machines. This property can be used to develop a matrix transposition algorithm that runs across multiple processes.

2.2 Constraints

This project is constrained to the use of a native MPI-3 algorithm using MPI Derived Data Types.

2.3 Assumptions

The matrix to be transposed will always be square and stored in row-major order in an input file. The first element in the file will represent the dimensions of the matrix (N). The number of processors (N_P) needed to run the program must satisfy the condition:

$$N \bmod (N_P - 1) = 0$$

2.4 Success Criteria

This project will be considered a success if the following conditions are met:

- An input-data generator is implemented.
- An MPI-3 program is implemented which transposes a square matrix.
- Results are obtained from the algorithm and analysed.

3. Algorithms

Two algorithms are implemented. The first consists of the generating function for the input matrix. The second consists of an MPI implementation of matrix transposition. Each algorithm will be explored in detail below.

3.1 Generator

The generator algorithm creates a 2D, square matrix of dimension N and stores it in row-major order in a file. The elements of each matrix is randomly generated in the range of 0 to 99. Due to the manner in which MPI-IO reads and writes to files, the generated integer matrix must be stored in binary. The pseudo-code for this algorithm is shown in Figure 1.

```
input  : Integer for matrix dimension
output: File with generated matrix in row-major order
Open output file "inputMatrix.txt";
Output matrix dimension as first file input;
Initialise random number generator for number between 0 and 99;
for Matrix dimenson do
    for Matrix dimenson do
        Output random number followed by a space;
    end
end
Close output file;
```

Figure 1: Generator algorithm pseudo-code

3.2 MPI

The second algorithm makes use of MPI Derived Data Types to implement a parallel matrix transposition program. An MPI Derived Data Type is defined as a new data type which is comprised of a combination native MPI data types.

The algorithm uses MPI-IO to read and write the files for the input and output data. This requires that the input is stored in a binary format. The algorithm assumes that the input matrix will be perfectly divisible by the number of ranks (N_R) -1. If this condition is not met the program will abort with an error. It is specified to be $N_R - 1$ because rank 0 does not participate in the transpositions, however it completes all the necessary I/O.

The matrix to be transposed is read into memory by rank 0 and stored as a 1D array using row-major ordering. The program then uses the block transposition algorithm to transpose the matrix. The matrix is divided into $(N_R - 1)^2$ blocks. These blocks are sent to a rank which transposes the block and sends the result back to rank 0 to be processed. These blocks are created using a MPI vector derived data type.

Each computing rank is sent a block to transpose using the blocking function `MPI_Send()`. The transposed blocks are accepted by each rank with the `MPI_Recv()` function. Once the block is transposed, the result is sent back to rank 0. Rank 0 writes the result to a file using `MPI_Write_to_file_at()`. While writing to the file, rank 0 also transposes the blocks of the matrix, as per the block transposition method. The final output file is written in binary.

The pseudo-code for this algorithm is shown in Figure A.1 in Appendix A.

4. Results

The matrix transposition algorithm was run for multiple matrix sizes on two separate machines. The first consisted of a laptop with an Intel®

Core$^{\text{TM}}$ i7-8550U Processor, 8 GB of RAM, and a 8 MB cache . The second machine consisted of a cluster node (Hornet01) having an Intel® Xeon® Processor E3-1270, 16GB of RAM, and a 8 MB cache. The running time of the program was obtained in order to assess the efficiency of the algorithm.

Table 2 indicates the timing results obtained from the laptop. Table B.1 in Appendix B indicates the timing results obtained from the cluster node.

5. Analysis of Results

As can be seen from comparing Tables 2 and B.1, the cluster node out-performs the laptop, as is expected due to the superior processor specifications.

It is also observed that increasing the number of ranks on which the transposition is to be applied does not decrease the speed, as seen in Figure 2. This is as expected due to both machines having 4 actual cores and 4 virtual cores provided by hyper-threading. As the number of ranks increases, the more parallelised the execution of the code becomes. However, the ranks are idle while waiting for a processor to execute on, resulting in longer wait times for each rank and an overall drop in performance.

Table 2: Laptop Results

N (Dimension)	Ranks	Time (s)
2000	11	54.17
1000	11	9.038
800	11	5.883
5000	6	255.33
2000	6	37.290
1000	6	6.519
800	6	4.516
5000	5	237.48
2000	5	29.975
2000	3	28.177
800	3	3.644

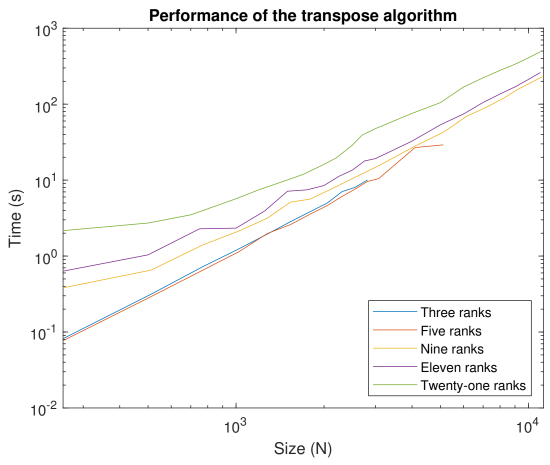


Figure 2: Log-Log graph showing the performance of the MPI algorithm on Hornet01.

Although increasing the number of physical processors increases the overhead time of message passing, it reduces the wait time of ranks. For large data-sets, the benefits of using more physical cores outweighs the idling time of waiting for messages to be passed. For a single machine run the optimal results are obtained from only slightly over subscribing the number of cores in the machines.

This is especially evident when looking at the results obtained from Hornet01 (see Table B.1 and Figure 2). When a matrix with a dimension 5120 was transposed, it took 42.725550 s when run on

9 ranks and 29.130977 s when run on 5 ranks.

Overall the developed program is found to run efficiently on a single node and testing on multiple nodes is expected to improve the performance of the algorithm significantly.

6. Conclusion

A randomised matrix generator algorithm was developed and used to create square matrices of different sizes. A parallelised MPI matrix transposition was implemented, making use of MPI Derived Data Types, and run on the matrices obtained from the generator algorithm.

The results obtained from running the transposition algorithm on multiple matrix sizes and with various numbers of ranks were obtained and analysed. It was found that the optimal run times were obtained from slightly over subscribing the number of threads available in the machine. Increasing the number of ranks far past the number of available threads decreased the performance of the algorithm significantly.

The algorithms developed were found to run efficiently on a single node and improved results are expected if the algorithms are run on multiple nodes. The project is considered a success as per the criteria set out in Section 2.4.

REFERENCES

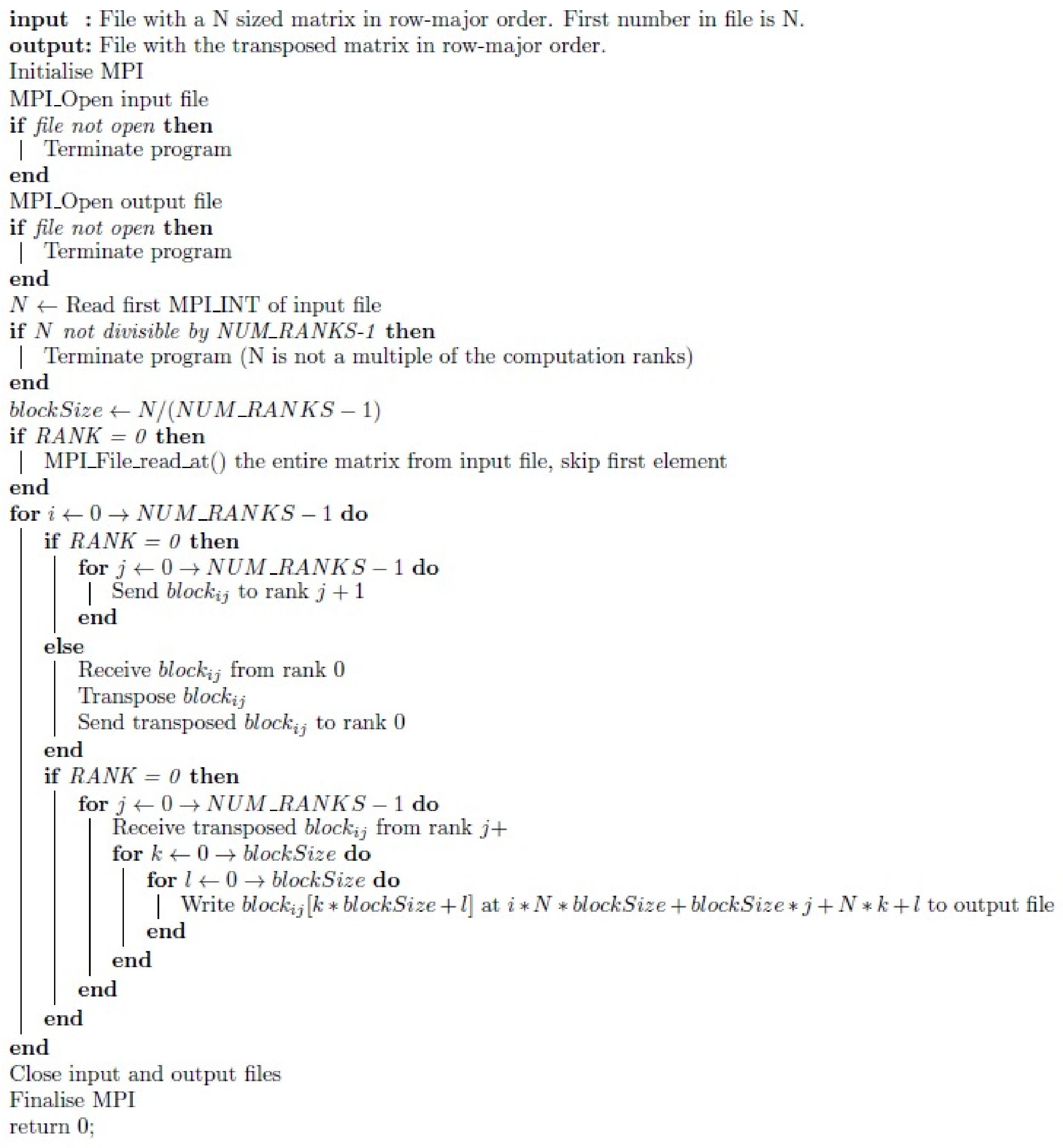
[1] Barney, B. (2019). Message Passing Interface (MPI). [online] Computing.llnl.gov. Available at: `https://computing.llnl.gov/tutorials/mpi/#What` [Accessed 4 May 2019].

[2] Bitbucket.org. (2019). berkeleylab/upcxx/wiki/ Home Bitbucket. [online] Available at: `https://bitbucket.org/berkeleylab/upcxx/wiki/Home` [Accessed 2 May 2019].

[3] Zheng, Y., Kamil, A., Driscoll, M., Shan, H. and Yelick, K. (2019). UPC++: A PGAS Extension for C++. [online] People.eecs.berkeley.edu. Available at: `https://people.eecs.berkeley.edu/~driscoll/pdfs/ipdps2014.pdf` [Accessed 8 May 2019].

Appendix

A MPI Algorithm Pseudo-Code



```
input  : File with a N sized matrix in row-major order. First number in file is N.
output: File with the transposed matrix in row-major order.
Initialise MPI
MPI_Open input file
if file not open then
    Terminate program
end
MPI_Open output file
if file not open then
    Terminate program
end
$N \leftarrow$ Read first MPI_INT of input file
if N not divisible by NUM_RANKS-1 then
    Terminate program (N is not a multiple of the computation ranks)
end
$blockSize \leftarrow N/(NUM\_RANKS - 1)$
if RANK = 0 then
    MPI_File_read_at() the entire matrix from input file, skip first element
end
for $i \leftarrow 0 \rightarrow NUM\_RANKS - 1$ do
    if RANK = 0 then
        for $j \leftarrow 0 \rightarrow NUM\_RANKS - 1$ do
            Send $block_{ij}$ to rank $j + 1$
        end
    else
        Receive $block_{ij}$ from rank 0
        Transpose $block_{ij}$
        Send transposed $block_{ij}$ to rank 0
    end
    if RANK = 0 then
        for $j \leftarrow 0 \rightarrow NUM\_RANKS - 1$ do
            Receive transposed $block_{ij}$ from rank $j+$
            for $k \leftarrow 0 \rightarrow blockSize$ do
                for $l \leftarrow 0 \rightarrow blockSize$ do
                    Write $block_{ij}[k * blockSize + l]$ at $i * N * blockSize + blockSize * j + N * k + l$ to output file
                end
            end
        end
    end
end
Close input and output files
Finalise MPI
return 0;
```

Figure A.1: MPI Transpose Algorithm Pseudo-code

B Cluster Node Results

Table B.1: Hornet01 Results

N	Ranks	Time (s)
256	3	0.081803
512	3	0.314701
768	3	0.722276
1024	3	1.248483
1280	3	1.950576
1536	3	2.838312
1792	3	3.840273
2048	3	4.975502
2304	3	7.044930
2560	3	8.064399
2816	3	10.068998
256	5	0.076931
512	5	0.292581
768	5	0.649454
1024	5	1.144604
1280	5	1.983275
1536	5	2.600120
1792	5	3.562047
2048	5	4.606892
2304	5	6.091390
2560	5	7.664860
2816	5	9.613604
3072	5	10.430917
4096	5	26.846469
5120	5	29.130977
256	9	0.381528
512	9	0.653554
768	9	1.397556
1024	9	2.141547
1280	9	3.161548
1536	9	5.133554
1792	9	5.617548
2048	9	7.181551
3072	9	15.441533
4096	9	28.089569
5120	9	42.725550
6144	9	69.245538
7168	9	90.690636
8192	9	118.673579
9216	9	157.177573
10240	9	193.793559
11264	9	233.550491
250	11	0.621512
500	11	1.037532
750	11	2.285528
1000	11	2.333531
1250	11	3.901606
1500	11	7.141535
1750	11	7.449510
2000	11	8.497504
2250	11	11.237517
2500	11	13.549535
2750	11	17.869507
3000	11	19.214463
4000	11	32.633572
5000	11	53.479704
6000	11	74.282461
7000	11	105.253534
8000	11	136.101531
9000	11	168.913530
10000	11	211.477531
11000	11	261.534481
200	21	1.986277
500	21	2.722253
700	21	3.490252
1000	21	5.686448
1200	21	7.490337
1500	21	10.062351
1700	21	11.937452
2000	21	16.138182
2200	21	19.561442
2500	21	28.857471
2700	21	39.230316
3000	21	47.694203
4000	21	75.645445
5000	21	104.769460
6000	21	168.109479
7000	21	222.450419
8000	21	278.645461
9000	21	335.997481
10000	21	407.234299
11000	21	490.861490
5000	5	27.580720
5000	6	39.954623
5000	9	44.409543