ELEN4020: Data Intensive Computing
Matrix Transposition of Big-Data

School of Electrical & Information Engineering, University of the Witwatersrand, Private Bag 3, 2050, Johannesburg, South Africa

Elias Sepuru 1427726 Boikanyo Radiokana 1386807 Lloyd Patsika 1041888

Abstract—**This report presents the implementation and analysis of matrix block transposition using straightforward MPI. The report divides the procedure into 4 main parts as follows: Input Data Generation of Matrices using Parallel I/O, Reading In of Data from Input Files using Parallel I/O and block transposition of matrices using MPI as well writing to files. The report anaylses the implementation through timing the various processes. It is noted that as the number of processors is increased, the time taken actually increases in most instances. Future recommendations include added code functionality which checks whether the transposition occurs properly, instead of human observation. Another improvement would be to provide another method of implementing parallelism in order to provide a more conclusive analysis.**

I. INTRODUCTION

Messaging-Passing Interface (MPI) is a versatile, standard interface for writing programs that run in parallel, using a distributed memory programming model. It is an interface which is generally used for composing parallel applications on systems such as laptops or clusters [**?**].

This report presents a block transposition implementation using Message-Passing-Interface in order to provide a comprehensive analysis of its performance. It involves the use of medium to large matrices being transposed through straight MPI with derived data types.

II. BACKGROUND

MPI was designed with large-scale systems in mind, and it is an architecture that supports highly parallel abstract machines through improving collective operations, scalable communicator and group operations. Reference [**?**] states that MPI can be used in order to take advantage of massive parallel computation as MPI provides a framework for inter-process communication. This project used this functionality in order to achieve Parallel I/O as well as parallel implementation of block matrix transposition.

A. Problem Description

The main problem that this report aims to solve is to develop and implement a parallel matrix transposition algorithm for very large data-sets that should be generated by a group of processors as sub-matrices but together form a large matrix file [**?**]. Furthermore, the matrix is required to be a large matrix A[N][N]. The input data, that is transposed after being read in, is initially in row major order after being generated by each submatrix using Parallel I/O. Another requirement is that the output data is also stored in row major order after transposition.

B. Success Criteria

The success of the project depends on achieving the following:

- Correct generation of the input data using Parallel I/O
- Correct storage of the input data in-out of core files
- Accurate Matrix Transposition using straight MPI
- Obtaining accurate time results for each important event, including input data generation, transposing of matrices and output data generation

III. DESIGN & IMPLEMENTATION

The Project is divided into 4 main parts: Generation of Matrices using random numbers, Reading of the generated input files, Transposition of the matrices and Writing of the transposed matrices to output files.

A. Generation of Random Numbers and Writing to Files

The required matrices to be generated as input data include matrix dimensions of A[N][N], where N = {8,16,32,64,128}. The number of processors that should be used to perform this generation are P = {16,32,64,128}.

In general, the procedure for generation of data involved each sub-matrix being generated by a single process. Every process is responsible for the generation of its sub-matrix random numbers as well as writing these to the respective file. The sub matrices for each process are all equal and this means that they are assigned the same amount of memory as an offset. The process with rank 0 is also responsible for writing the matrix dimension. Algorithm 1 shows the general pseudo code for this procedure.

B. Reading of the Input Data

An input text file is read in, in order to access the data, for transposition. The process is quite similar to the one for writing as each process is given a position in the input text file to read from. The processes then read their sub matrices and put them at specific positions in a 1D array. In order to have the data ready for transposition, the 1D array components are then transferred into a 2D array.

C. Matrix Transposition

Matrix transposition is performed using the `MPI_Alltoall` function together with the `localTrans()` function. `MPI_Alltoall` sends data from all processors to all processors. In other words, it sends distinct data to all the processes that are receivers [**?**]. This exchange of data from one process to another yields an outer matrix block transposition of elements. `MPI_Alltoall` achieves this process by sending a block of data (block *j*) from process *i* to the recieve buffer of process *j*. In the receive buffer of process *j*, the received block is placed in the ith block. The `MPI_Alltoall` function takes the data types of the sender and receiver, `MPI_COMM_WORLD` function, the number of elements sent and the elements that the receiver is accepting. For matrix transposition to be successful, the amount of data sent by process *i* must be the same as the amount of data received by process *j*, however the type of maps can be different. The `MPI_COMM_WORLD` argument is a communicator that groups together all processes at the beginning of the program to ensure that processes can communicate with each other [**?**]. Communication can be achieved in two ways, namely [**?**]:

- Point-to-point Communication
- Collective Communication

Collective communication was used because it allows all processes to communicate collectively. Figure 1 shows how the processes can communicate.

Following the transposition of the outer blocks with `MPI_Alltoall`, the `localTrans()` function transposes the local elements of the blocks within each process. This is done in parallel for all processes. Algorithm

```
input  : The dimension of the matrix to be
         generated
output: Text file with Output Matrix

MPI_File filehandler
MPI_Status status
MPI_Comm comm
MPI_Init
for i in MatrixDimension do
    Creating the filename file_name←
      ”file_”+”MatrixDimension”
      error←MPI_File_open(comm, n,
      MPI_MODE_RDONLY,
      MPI_INFO_NULL,&filehandler)
      MPI_Offset offset ← 0
    if error then
        MPI_Abort (MPI_COMM_WORLD,911)
    end
    MPI_Comm_size (comm,&size)
    MPI_Comm_rank (comm, &process_rank)
    Writing MatrixDimension at Position 0
    if process_rank EQUALS 0 then
        MPI_Offset temp ← 1
        error←MPI_File_write(comm, n,
          MPI_MODE_RDONLY,
          MPI_INFO_NULL,&filehandler)
    end
    Dividing Matrix into Chunks MPI_Offset
      chunk_memory ← $(MatrixDimension \times MatrixDimension/size)$
    randomNumbersBuffer
      ← $chunk\_memory \times sizeof(int)$
    randomNumberGenerator(randomNumbersBuffer
    ,chunk_memory)
    Calculating Offset based on memory
    offset ← $((process\_rank \times chunk\_memory) + 1) \times sizeof(int)$
    MPI_File_set_view
      (filehandler,offset,MPI$_INT, MPI_INT, ”native”$,
    MPI_INFO_NULL) error
      ← $MPI\_File\_write\_all(filehandler,$
    randomNumbersBuffer,chunk_memory,MPI_INT,
      &status$)error \leftarrow$
      $MPI\_File\_close(\&filehandler)$
end
MPI_Finalize
```

Algorithm 1: `Matrix Generation using MPI`

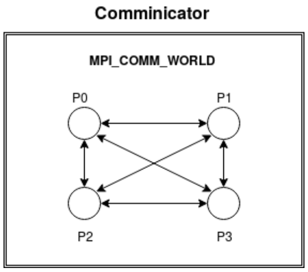

Fig. 1. Collective communication in MPI_COMM_WORLD

2 shows the pseudo code used to transpose the elements within each process.

```
input: A 2D matrix ”A” of size $n \times n$
otherCol ←0
otherRow ←0
for $i \leftarrow 0$ to $n-1$ do
    for $j \leftarrow 0$ to $n-1$ do
        for $k \leftarrow 0$ to $BLOCK\_SIZE$ do
            for $l \leftarrow 0$ to $l < k$ do
                if $i! = j$ then
                    otherRow ← $(i+l)$ otherCol
                      ← $(j+k)$
                end
                else
                    otherRow ← $(l+j)$ otherCol
                      ← $(k+i)$
                end
                swap $(A[k+i][l+j],$
                  $A$[otherRow][otherCol])
            end
        end
    end
end
```
Algorithm 2: `blockElementTranspose(A)`

D. Output Data Generation

Each process is responsible for writing its sub matrix to a file. The process is quite similar to the one for writing of random numbers to a file as each process is given a position in the output text file to write to. The processes then write their sub matrices at specific positions in the output text file. The output file then contains all matrix values which were contained in all the sub matrices of the process once the process is complete. These values are stored in row-major order, as required.

IV. CRITICAL ANALYSIS

A. Environment Used

The programs are coded in C. In order to allow for proper testing of the code, *hornet01.eie.wits.ac.za* is the cluster that is used. The cluster has a memory size of 16GB and a 3.4GHz. Although the cluster has 373 processors, a maximum number of 128 is used for this project. The environment has an MPI version called MPICH3.3.

B. Results

Figures 2-3 show graphs of program performances. A general trend shown is that the number of processors used does not correlate to a shorter time for execution. The results follow Amdahl’s Law. It re-affirms the idea that an increase in the processor number does not necessarily mean a faster execution time.

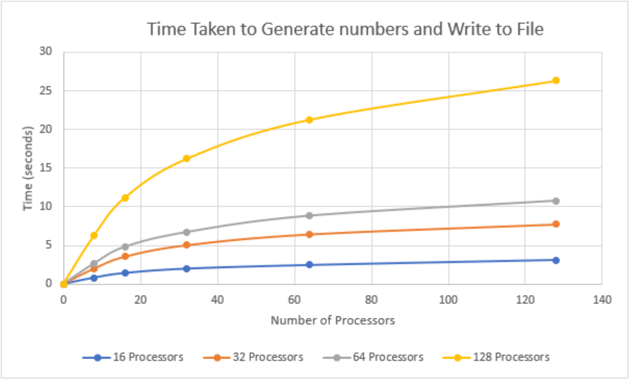

Fig. 2. Time Taken to Generate Input Data

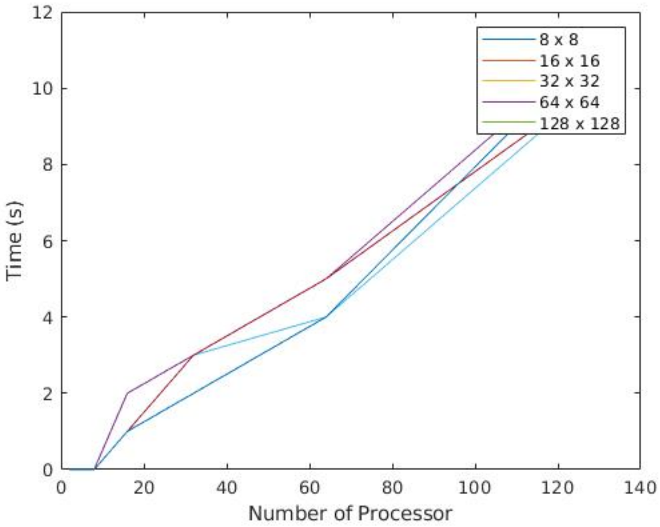

Fig. 3. Time Taken to Transpose Matrices and Write to File

C. Future Recommendations

As an improvement to the project, it is recommended that another method such as UPC or UPC++ be used in order to compare to the method used in this project. This is because, although this project makes use of timing in order to compare one algorithm at different processes and matrix sizes, it would be much better research if there was another method that is able to also transpose matrices for purposes of comparison. Another recommendation which would improve the functioning of the program would be the addition of functionality to check the validity of the transpose.

V. Work Division

The project was divided into manageable tasks and delegated amongst the members of the team. The team members,Elias Sepuru, Boikanyo Radiokana and Lloyd Patsika are final year Electrical and Information Engineering students. The following table shows the work division between the members mentioned:

TABLE I
Work division among team members

Work Division	
Elias Sepuru	Code: Matrix transposition, Integration of all codes and Timing Report: Section 3B, Section 4A, Section 4B
Boikanyo Radiokana	Code: Writing from a file, Integration of all codes and TimingReport: Section 3A, Section 5, Conclusion
Lloyd Patsika	Code: Random number generation and Reading to the file, Integration of all codes and Timing Report: Introduction, Background, Section 3C

VI. CONCLUSION

A straightforward MPI implementation is presented, where matrix block transposition is implemented in order to critically analyse its performance. The different size matrices are generated using Parallel I/O. This parallel implementation also extends to the reading of input files as well as the writing of the results generated by the transposition. Timing was used in order to analyse the performance of the program and it was noted that an increase in the number of processors does not necessarily correlate to an decrease in the time for execution of the program. Recommendations to improve the code include implementing another form of parallelism using UPC/UPC++ while necessary changes to the code include code tests to confirm accurate transposition.

Appendix

A. *Graphs showing various times of execution of code*

TABLE I
TIME TAKEN FOR INPUT GENERATION AND WRITING TO FILES

Matrix Sizes	**16 Pro.**	**32 Proc.**	**64 Proc.**	**128 Proc.**
8	0,829196s	1,997403s	2,675249s	6,30365s
16	1,425108s	3,55822s	4,831607s	11,183438s
32	1,989904s	5,049717s	6,731276s	16,201182s
64	2,465029s	6,466829s	8,855374s	21,271032s
128	3,126022s	7,772867s	10,802833s	26,308761s

TABLE II
TIME TAKEN FOR TRANSPOSITION AND WRITING TO FILES

Matrix Sizes	**2 Pro.**	**4 Proc.**	**8 Proc.**	**16 Proc.**	**32 Proc.**	**64 Proc.**	**128 Proc.**
8	0s	0s	0s	2s	3s	5s	11s
16	0s	0s	0s	1s	2s	4s	10s
32	0s	0s	0s	1s	3s	4s	10s
64	0s	0s	0s	1s	3s	5s	10s
128	0s	0s	0s	1s	2s	4s	11s

```
Initialize MPI
MPI_Init (&argc, &argv)
MPI_Comm (MPI_COMM_WORLD, &rank)
MPI_Comm_size (MPI_COMM_WORLD, &numProcs)
MPI Setup
MPI_File fh
MPI_Win win
MPI_Status status
MPI_Offset disp
MPI_Offset blocksize

Open file
 inputError←MPI_File_open(MPI_COMM_WORLD, in, MPI_MODE_RDONLY, MPI_INFO_NULL,&fh)

Validate file open successfully
if inputError then
    if rank ←0 then
        Display Error
        MPI_Finalize()
        exit()
    end
end

Get size of matrix
 sizeChecker← 1 byte
 MPI_File (fh, sizeChecker, 1, MPI_INT, &status)
 SIZE←sizeChecker [0]
 Display the SIZE
 Free memory of sizeChecker
 n← SIZE divide by numProcs

Condition buffer for each rank
 blockSize← SIZE $\times n$
 disp←(rank$\times blockSize) + 1 \times sizeof(int)$
 tempBuffer← (int*)malloc(sizeof(int))

Read file into Buffer
MPI_File_set_view (fh, disp, MPI_INT, MPI_INT, ”native”, MPI_INFO_NULL)
MPI_File_read (fh, tempBuffer, blocksize, MPI_INT, &status)
MPI_File_Close (&fh)

Confirm
for $i$← blocksize do
    if rank ←1 then
        Display rank and tempBuffer[i]
    end
end
```

Algorithm 3: `blockElementTranspose(A)`