MATRIX TRANSPOSITION OF BIG-DATA

Lynch Mwaniki (1043475) Madimetja Sethosa (1076467) Teboho Matsheke 1157717
School of Electrical and Information Engineering, University of the Witwatersrand, Johannesburg 2050, South Africa
ELEN4020A: DATA INTENSIVE COMPUTING

Abstract: This report presents the detailed use of MPI and MPI_I/O to generate and transpose matrices held out of core. Matrices of different sizes were generated using MPI's read and write to file functions. The transpose algorithm successfully transposed the generated matrices using different numbers of threads. The testing was done using Wits EIE hornet01 Linux computer and the execution times were recorded. The execution times indicated that the transpose algorithm is scalable for large-scale parallel applications. The results indicated that speedup was observed when more than four threads were used to transpose the large matrices.

Key words: C, MPI, RMA, Matrix-transposition

1. INTRODUCTION

This report presents matrix transposition which uses MPI and MPI_I/O to generate matrices of different sizes in a file and uses a different number of threads to actually transpose the matrices. The matrix sizes ranges from 2^3 to 2^7 and 2^{10} to 2^{15} in successive powers of two. The transpose algorithm makes use of the Message Passing Interface (MPI) one-sided communication, commonly referred to as the Remote Memory Access (RAM). The Wits EIE Hornet01 Linux computer with 8 cores is used to run the algorithm. Consequently, different numbers of threads (1, 2, 4 and 8 threads) were used to transpose each file. The performance of the algorithm for each file size against the different number of threads was measured and recorded.

The report is structured as follows: brief knowledge about MPI and its operations is documented in Section 2. and the file generation method and the MPI matrix transposition algorithm is explained in Section 3. The timing results of the matrix transposition for different matrix sizes is critically analyzed in Section 4.

2. BACKGROUND

One of the most basic methods of programming for parallel computers is the use of Message Passing Interface (MPI) libraries. It is a library of functions in C or subroutines in Fortan that are used in source code to perform data communication between processors [1]. Sharing of data between processes is done through message passing: by explicitly receiving and sending data between processes. In MPI, the programmer explicitly divides data and work across the processors as well as manages the communications among them.

MPI offers a great deal of functionality: different types of communication (blocking and non-blocking), special routines for common collective operations and the ability to handle user-defined data types [2]. It also supports different types of parallel architectures. MPI was standardized in 1992 at the Super-computing conference by a group called MPI Forum [1]. It was further improved in 1994 by introducing Remote Memory Access (RMA), also referred to as one-sided communication since only one process is required to transfer data.

In RMA operations, there are two processors involved: the origin processor which originates the transfer ('put' or 'get') and the target processor whose memory is being accessed [3]. RMA communication is non-blocking. Consequently, it decouples data transfer from system synchronization; the remote process can continue to compute its operations while a target process sends data to it. Some of the RMA functions vital for communication are:

- MPI_Win_create declares an area of memory that is accessible to other processes
- MPI_Put transfers data from local to remote memory
- MPI_Get fetches data from remote to local memory

- MPI_Accumulate updates the remote memory using local values
- MPI_Win_fence to achieve synchronization
- MPI_Win_free de-allocates the memory window
- MPI_Finalize clears up all MPI states

The target has to declare a window, memory accessible to other processes which means that MPI is only limited to this window. This may make the implementation more efficient but tends to make programming slightly more cumbersome. We can overcome this issue by using distributed shared memory: distributed memory that acts as if it is shared.

Within MPI RMA, there are two modes; active and passive RMA [4]. In active RMA, the target limits the the time during which its window can be accessed. Active RMA acts like asynchronous transfer; the origin can execute many small transfers combined behind the scenes. In passive RMA, the target sets no limitations on the accessibility time period on its window. We can write and read from a target at any arbitrary time, but for this to not cause problems (strange deadlocks and debugging issues), a remote agent on the target is required.

3. MPI ALGORITHM

The MPI algorithm implemented for the project required files to be generated as input. The matrix files were generated using MPI this is detailed in Section 3.1. The MPI transposition algorithm is discussed in Section 3.2.

3.1 File Generation

The project required input files to be generated that would be used by the MPI Transpose algorithm. The authors did the matrix file generation using MPI. The code can be viewed in *mpiMatrixGenerator.c*, and the pseudo-code can be viewed in Algorithm 1. Figure 1 shows how the matrix generation is divided among 4 processors/threads. The algorithm made this division generic and divides the matrix evenly across processors/threads. Currently, this only works for 1 processor or where the number of processors specified are multiples of 2. Figure 1 illustrates how the implemented file generation divides the generation of a matrix across processors. Each process generates random numbers and the number of generated numbers is determined by the chunk size allocated which is in turn dependent on the number of processors specified to run the code. Equation 1 show the equation used to split the size.

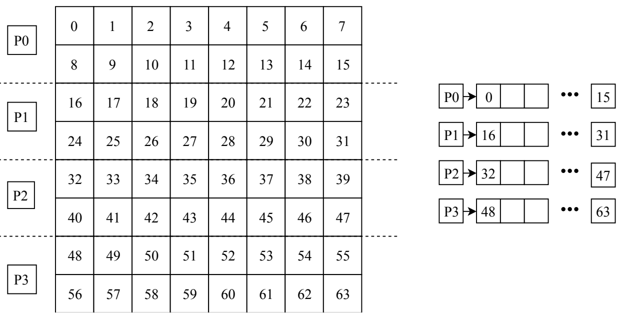

Figure 1: Figure showing File views for each process.

$$chunk\_size = MATRIX\_SIZE^2 \div num\_processors \quad (1)$$

Algorithm 1: MPI Matrix File Generation

```
Result: A NxN matrix stored in a file.
Input  : Size of square matrix N
1  Initialize MPI_COMM_WORLD
2  Create filename.
3  Open file for writing.
4  if error in opening file then
5      Abort MPI_COMM_WORLD
6  Generate seed using rank of processor.
7  if rank is equal to 0 then
8      MPI_File_write at displacement 1, size of matrix N.
9  Get size of MPI_Comm_World.
10 Divide matrix into chunks. (N*N)/size
11 Allocate buffer with chunk size.
12 Generate random numbers of chunk size and store them in
     the buffer.
13 Set displacement in file view before writing.
14 displace = ((rank*chunk_size)+1 )*sizeof(int).
15 Each processor writes buffer to file at view set using
     MPI_INT data type.
16 Close file.
17 Free buffer memory.
18 Finalize MPI.
19 return
```

The files generated contain randomly generated numbers in the range [0-99] as specified in the project brief [5]. The generated files store binary representations of the random numbers. The first element is the matrix size, N and the rest are the random numbers that make up the matrix. The output file is named $matrixFile\_ < N >$.

3.2 Matrix Transposition

The MPI algorithm implemented makes use of One-sided communication. Each process reads the first element in the file, N and stores this locally. The number of threads in the MPI_COMM_WORLD and the matrix size are then used to determine the chunk_size each processor should read from the file. A file view for reading is set for each processor. A MPI_Offset is used to displace the file view for each processor to ensure that they read in different rows of the matrix. The algorithm reads the matrix elements from a file in row major order. This process can be visualized by Figure 1. Each processor places the elements they read in, into a shared buffer. The algorithm makes use of MPI_Get() to access elements in a shared buffer [6, 7].

From here a simple approach was used to transpose the matrix. The threads make use of Remote Memory Access (RMA) calls with MPI_Get() to retrieve the elements in column major order which will result in the transposed matrix. A local buffer, with the size of the chunk read in, is allocated memory. The chunk size allocated to each processor determines how many columns must be read from the shared buffer into the local buffer. Equation 2 shows how the number columns is calculated.

$$number\_of\_columns = chunk\_size \div N \qquad (2)$$

Figure 2 illustrates an example of calls made to the shared buffer. After reading from the file, the shared buffer will look like the complete matrix spread across the threads. The algorithm implemented will use each processor to get elements from the shared buffer in column major order. From Figure 2, Processor 0 will make MPI_Get() calls to retrieve the first element. The next MPI_Get() call is made to the processor that has element 8. In this example, the processors read in chunk sizes of 16 elements thus the call made for the second element in column 0 is made to processor 0's shared buffer with a displacement of N. This process is repeated until the processor collects elements of the size of the chunk assigned or the number of columns

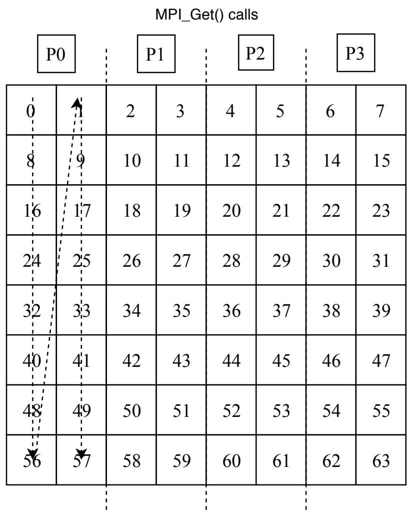

Figure 2: Figure showing MPI_Get calls for each thread from shared buffer using 4 threads

assigned for it to read. Appendix A Algorithm 2 gives the pseudo-code that performs the matrix transposition.

4. CRITICAL ANALYSIS

The environment used to test the algorithm was the Wits EIE Hornet01 Linux computer which has 8 cores running at 3.40 GHz and 16 GB RAM, running Ubuntu 18.04.2 LTS. The algorithm transposed the matrices stored in files generated by the File Generator algorithm as discussed in Section 3.1. The matrix transpose algorithm was executed using different number of processors for each file. The transposition of the matrices was timed and recorded. The timing of the algorithm includes reading from a file, transposing the matrix and writing the transposed matrix to a file.

The algorithm was initially tested with small files of size less than 66 Kb (matrix size N= 2^3 to 2^7) and the execution times are recorded in Table 1.

Table 1: Table showing Transpose time (in seconds) for small files.

Matrix size	Number of processors			
	1	2	4	8
8	0.006524	0.008469	0.009823	0.012312
16	0.006590	0.008661	0.010616	0.012131
32	0.006490	0.008106	0.012196	0.012920
64	0.006893	0.010437	0.013021	0.014107
128	0.007723	0.015426	0.018920	0.018007

The graph in Figure 3 graphically presents the execution results from Table 1. The second batch of files that

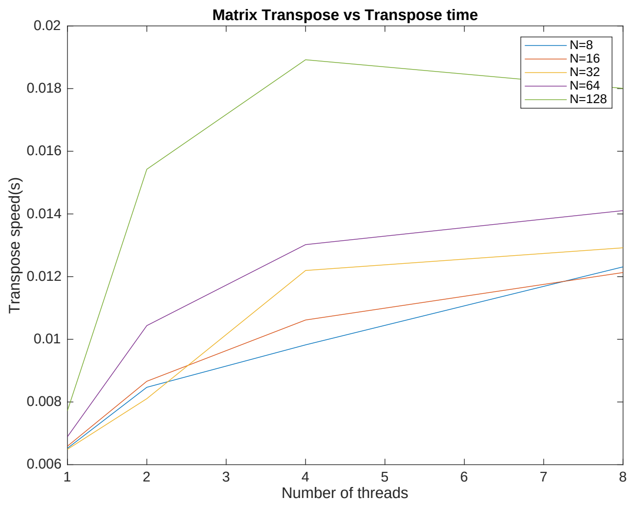

Figure 3: Figure showing Transpose time for small files.

contained matrices of size N= 2^{10} to 2^{15}, which were considered as large files since their size is ranging from 4 *MB* to 4.29 *GB*. The algorithm's execution times using these files as input are recorded in Figure 2.

Table 2: Table showing Transpose time (in seconds) for big files.

Matrix size	Number of processors			
	1	2	4	8
1024	0.043069	0.490143	0.532878	0.524006
2048	0.282899	1.894200	2.203462	1.898983
4096	1.548822	7.402981	8.139331	7.519005
8192	6.340876	32.26938	35.51283	33.70523
16384	25.76930	133.7979	154.5446	145.0108
32768	163.8462	624.2739	636.7037	689.7668

The graph in Figure 4 shows the relationship between the number of processors used and the transposition time for different files (big files) from Table 2. From both results (for small and big files), it is evident that the algorithm becomes faster as the number of threads is increased, but this is only visible for large files. As for small files (excluding that of $n = 128$), as the number of threads increase, the execution time increases. This shows that the benefits of parallelism does not get realized for small data sets. For big files, an increase in the number of threads improves the execution time of the algorithm. This indicates that the algorithm is scalable for large-scale

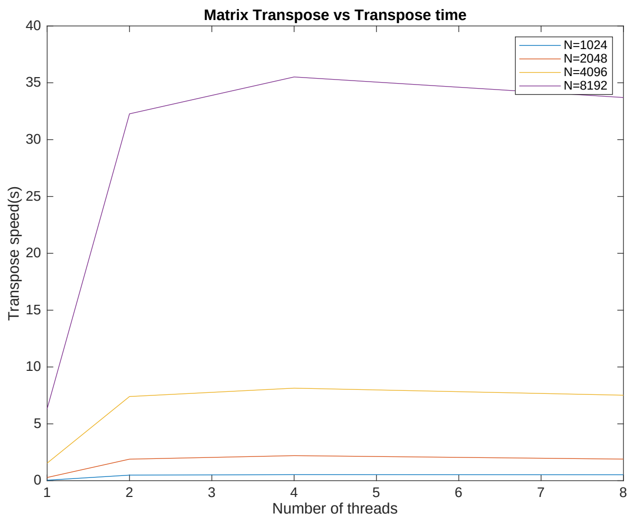

Figure 4: Figure showing Transpose time for big files.

parallel applications. Unfortunately, due to the limitations of the hardware used, the authors could not try doing the matrix transposition using processors = 16,32,64 to run the matrix files as these were not available at the time.

5. CONCLUSION

The report detailed the use of MPI to generate matrices in files and to transpose matrices of different sizes using different number of threads. They both make use of collective MPI_I/O. The file generation is able to generate files with matrices of different sizes. The transpose algorithm makes use of one-sided communication, and it is able to transpose matrices of different sizes. The execution times indicated that the transpose algorithm on large data sets tends to speed up when the number of processors used is increased. Although at some point this speed up will saturate indicating that adding more processors may not always improve the performance of a parallel computing algorithm.

REFERENCES

[1] W. Gropp, T. Hoefler, R. Thakur, and E. Lusk, *Using advanced MPI: modern features of the Message-Passing-Interface*. The MIT Press, 2014.

[2] N. Hjelm, "Optimizing one-sided operations in open mpi," *Proceedings of the 21st European MPI Users Group Meeting on - EuroMPI/ASIA 14*, 2014.

[3] "Mpi topic: One-sided communication." [Online]. Available: http://pages.tacc.utexas.edu/~eijkhout/pcse/html/mpi-onesided.html

[4] W. Gropp, E. Lusk, and A. Skjellum, "Using mpi," *Parallel Programming With MPI*, 1999.

[5] (2019) Project 2019 - Matrix Transposition of Big-Data. [Online]. Available: https://cle.wits.ac.za/access/content/group/ELEN4020A/Laboratory\%20Exercises/dicaProjects2019.pdf

[6] (2014) MPI One Sided Communication. [Online]. Available: https://software.intel.com/en-us/blogs/2014/08/06/one-sided-communication

[7] (2014) MPI One Sided Communication. [Online]. Available: https://www.mpich.org/static/docs/v3.1/www3/MPI_Get.html

A MATRIX TRANSPOSE ALGORITHM

Algorithm 2: MPI Matrix Transposition

```
Result: A NxN transposed matrix stored in a file.
Input  : File with a transposed square matrix N
Input  : Output file name
Initialize MPI_COMM_WORLD.
Get size of MPI_COMM_World.
Divide matrix into chunks. (N*N)/size
Create shared buffer of size chunk size.
Open file for reading.
if error in opening file then
    Abort MPI_COMM_WORLD
MPI_File_read at displacement 1, size of matrix (N).
Allocate local buffer with chunk_size.
Allocate shared buffer with chunk_size.
Set displacement in file view before reading.
displace = ((rank*chunk_size)+1 )*sizeof(int).
Read in matrix numbers of chunk size and store them in the shared buffer.
Call MPI_Win_Create to create window for shared buffer.
Synchronize RMA calls.
Set counter to zero.
for column = $rank \cdot num\_columns$ to $(rank \cdot num\_columns) + num\_columns$ do
    Loop through each processor and make calls to retrieve elements in each row at index equal to column.
    for processorID = 0 to num_processors do
        Set rowInChunk = 0
        for i = 0 to i = num_processors do
            Get from processID's shared buffer one element at displacement rowInChunk+column.
            Store this value in localbuffer at position counter.
            Increment rowInChunk by N
            Increment counter by 1
        Increment processorID by 1
    column += 1
Each processor writes buffer to file at view set using MPI_INT data type.
Close file.
Free buffer memory.
Finalize MPI.
return
```

B DIVISION OF WORK

Table 3: Work allocation

Group members work allocation		
Madimetja Sethosa	**Lynch Stephen Mwaniki**	**Teboho Peaceful Matsheke**
MPI transposition alg	MPI transposition alg	MPI transposition alg
Results recording	MPI file generation	transposition alg timer
Documentation	Documentation	Documentation