Matrix Transposition of Big-Data

Sbonelo Mdluli:1101772, Heemal Ryan:792656, Haroon Rehman:1438756
ELEN4020A Project Due date: 17 May 2019

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

***Abstract*—This project includes code written using MPI in order to solve a matrix transposition, this report showcases and explores a performance comparison when using 16, 32 and 64 parallel processes. Using more process was found to be immediately faster however eventually became unnecessary as explained by Amdahls Law. Parallels I/O is used to access file simultaneously from different processes in order to maximize access speed.**

I. Introduction

Big Data is often characterized by the 3 V's; Volume, extremely large data sets. Velocity, speed at which the data needs to be processed. Variety, referring to the different types of data that may need processing [1].

A very popular method of solving big data problems is parallel programming. Simply put, parallel programming allows for actions to be carried out simultaneously, in doing so, implementations are able to solve much larger problems than otherwise possible [2]. Efficient processing in usually required in numerical and scientific computations such as FFT and differential equations.

The following report focuses on a performance test using MPI (Message Passing Interface), a straight forward parallel programming method. The test entails recording the time taken to compute a matrix transposition.The following topics will be explored:

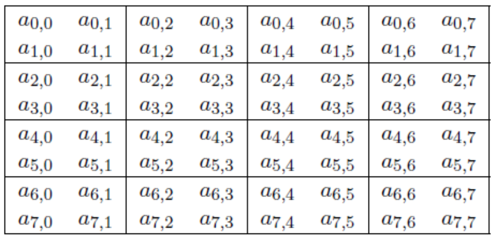
Problem Description What is MPI? Program Environment Code Structure Results Recommendations Conclusion

II. Background

A. Problem Description

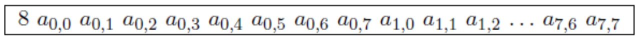
The problem entails showcasing a performance comparison when using 16, 32, 64 parallel processes to do a task. The task consists of performing a matrix transposition using MPI. The transposition should be done for a several sized matrices (see Tables 4-6). The input data is provided as per Tables 1 and 2

Table 1: Matrix A[8][8]



$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{0,5}$	$a_{0,6}$	$a_{0,7}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$	$a_{1,6}$	$a_{1,7}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$	$a_{2,6}$	$a_{2,7}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$	$a_{3,6}$	$a_{3,7}$
$a_{4,0}$	$a_{4,1}$	$a_{4,2}$	$a_{4,3}$	$a_{4,4}$	$a_{4,5}$	$a_{4,6}$	$a_{4,7}$
$a_{5,0}$	$a_{5,1}$	$a_{5,2}$	$a_{5,3}$	$a_{5,4}$	$a_{5,5}$	$a_{5,6}$	$a_{5,7}$
$a_{6,0}$	$a_{6,1}$	$a_{6,2}$	$a_{6,3}$	$a_{6,4}$	$a_{6,5}$	$a_{6,6}$	$a_{6,7}$
$a_{7,0}$	$a_{7,1}$	$a_{7,2}$	$a_{7,3}$	$a_{7,4}$	$a_{7,5}$	$a_{7,6}$	$a_{7,7}$

Table 2: Data *matrixFile_N*.



8 $a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$ $a_{0,4}$ $a_{0,5}$ $a_{0,6}$ $a_{0,7}$ $a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $\cdots$ $a_{7,6}$ $a_{7,7}$

III. Message Passing Interface

MPI is a standardized message passing structure used in parallel computing (message passing refers to sending an instruction to a process such as an object, function or thread). MPI provides parallel hardware vendors with a set of clearly defined routines/instructions that may be efficiently implemented. Table 3 contains a summary of MPI's strengths and weaknesses [3], [4], [5].

Table 3: Strengths vs Weaknesses

Strengths	Weaknesses
Highly efficient Scalibility Easily available Large volume of resources Reliable Stable communication	No support for multi-threading Inefficient dynamic process spawning No debugging facilities

IV. Program Environment

Code is to be tested using hornet01.eie.wits.ac.za, IP Address: 146.141.116.172, the host contains 16GB of available memory and a 3.4GHz CPU. A maximum of 8 threads is allowed per host. The environment is running on MPICH3.3 which is an MPI implementation. The maximum number of processors available on hornet01 is 373.

V. Code Structure

Before exploring the programs code structure, we will explain how an MPI application works: MPI programs consist of multiple processes running at the same time. These processes have a unique identifiers(ranks) starting from 0 to (No. of Processes - 1) which run asynchronously. MPI also contain the notion of 'communicators', communicators describe a collection of processes, at initialization MPI_COMM_WORLD is made and contains all processes that were started upon running of the application [6].

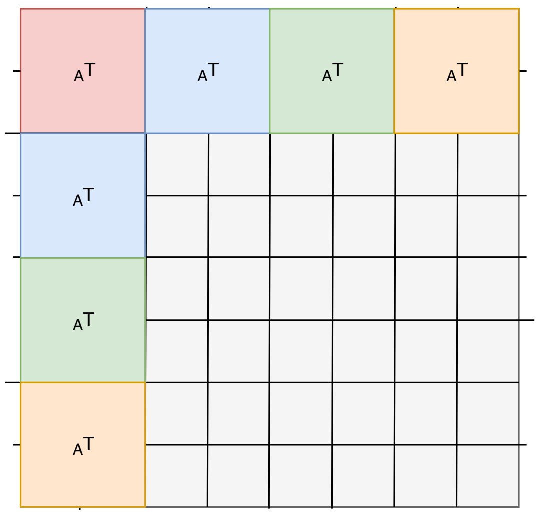


Fig. 1: Transposition Algorithm

The transposition algorithm is performed internally in blocks of 4 and those blocks are they put in to the overall encompassing matrix for further transposition. The final transposition is done by swapping the blocks. The blocks are put in a continuous rowtype where they are scattered to different processors, alltoall is used to perform the transpose of the bigger blocks.

The file generation and reading is done using parallel I/O which is one technique used to access data simultaneously from different application processes to maximize bandwidth and speed access time. This uses collective I/O which is used to move data between processors.

VI. Results

The following results were found during testing of code writing using the hornet01.eie.wits.ac.za host. Appendix B contain line graph plots for each performance test. The timing includes processing and communication time, these are fairly similar with most of the inputs, so only the processing time is reported.

Table 4

No. of Processes = 16			
Array Size	Read(ms)	Write(ms)	Transpose(sec)
2^3	74.59	0.119	0.930
2^4	70.02	0.294	0.080
2^5	86.54	3.360	0.079
2^6	96.50	23.98	0.101
2^7	112.9	26.38	0.094

Table 5

No. of Processes = 32			
Array Size	Read(ms)	Write(ms)	Transpose(sec)
2^3	70.55	0.129	0.177
2^4	77.21	0.208	0.213
2^5	80.49	1.567	0.221
2^6	83.41	11.08	0.154
2^7	95.32	10.39	0.210

Table 6

No. of Processes = 64			
Array Size	Read(ms)	Write(ms)	Transpose(sec)
2^3	66.77	0.093	0.293
2^4	67.61	0.120	0.321
2^5	70.56	0.255	0.329
2^6	72.16	2.782	ONCLUSION0.329
2^7	95.32	2.752	0.285

The results are according to Amdahl's law which can be used to estimate performance. The principle states that increasing the number of processors does not necessarily increase speed up time.

VII. Recommendations

The current transposition algorithm waits for rank 0 to do the internal block matrix locally and sends the results the different processors. This is inefficient because this causes a block/barrier. To increase the computation time the block transpose should be done in other processors as not to cause a barrier. Further optimizations can be made by making use of the shared memory of each distributed memory. This can be done by using Pthreads in each node. The send can be made asynchronous and non-blocking.

VIII. Conclusion

This report presents an investigation of matrix transpose and parallel I/O across multiple processors. A $N \times N$ square matrix of $N = 2^n$is where $N = 3, 4, 5, 6, 7$ is transposed using block transposition and MPI_alltoall. The input data in generated into a binary file using parallel I/O. The results from the investigation are consistence with theoretical studies namely Amdahl's law.

REFERENCES

[1] SearchDataManagement. (2019). What is big data? - Definition from WhatIs.com. [online] Available at: https://searchdatamanagement.techtarget.com/definition/big-data [Accessed 16 May 2019].

[2] OReilly — Safari. (2019). Patterns for Parallel Software Design. [online] Available at: https://www.oreilly.com/library/view/patterns-for-parallel/9780470697344/9780470697344_advantages_and_disadvantages_of_paral.html [Accessed 16 May 2019].

[3] GeeksforGeeks. (2019). MPI - Distributed Computing made easy - GeeksforGeeks. [online] Available at: https://www.geeksforgeeks.org/mpi-distributed-computing-made-easy/ [Accessed 16 May 2019].

[4] Cecs.wright.edu. (2019). [online] Available at: http://cecs.wright.edu/ schung/ceg820/mpi-book.pdf [Accessed 16 May 2019].

[5] Rosul, C. (2019). Message Passing Interface (MPI) Advantages and Disadvantages for applicability in the NoC Environment by. [online] Semanticscholar.org. Available at: https://www.semanticscholar.org/paper/Message-Passing-Interface-(-MPI-)-Advantages-and-in-Rosul/6c5dd34a693f72adaebd1d02d521c5b7cfdba589 [Accessed 16 May 2019].

[6] Userinfo.surfsara.nl. (2019). MPI in two hours — userinfo.surfsara.nl. [online] Available at: https://userinfo.surfsara.nl/systems/shared/mpi/mpi-intro [Accessed 16 May 2019].

Appendix

A. Pseudo-code

Algorithm 1: Function that reads input data

```
Function name: Main Function
Input/Output Files
char *in = argv [1], char *out = argv[2],
Setup
int rank, numProcs, inputError, SIZE,
int *sizeChecker, *tempBuffer, n
Initialize MPI
MPI_Init(argc, argv)
MPI_Comm_rank(MPI_COMM_WORLD, rank)
MP_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)
if There is an input error = true then
    exit;
else
    Get size of Matrix
    sizeChecker = (int*)malloc(sizeof(int))
    MPI_File_read(fh, sizeChecker, 1, MPI_INT, status)
    SIZE = sizeChecker[0]
    n = (int)(SIZE/numProcs)
    Condition the buffer for each rank blockSize = SIZE*n
    disp = ((rank*blockSize)+1)*sizeof(int)
    tempBuffer = (int*)malloc(SIZE*n*sizeof(int))
     Read the file into the 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)
end
return 0
```

Algorithm 2: Function performs matrix transpose

```
Function name: Main Function
Initialize
int i, world_rank, world_size
double process_time, process_time_to_avg, comm_time = 0
int **rand_matrix, **trans_matrix = NULL
int *recv_buffer = (int*) malloc(mat_size * sizeof(int))
int *recv_buffer2 = (int*) malloc(mat_size * sizeof(int))
MPI_Init(NULL, NULL)
MPI_Comm_rank(MPI_COMM_WORLD, world_rank)
MPI_Comm_size(MPI_COMM_WORLD, world_size)
Scatter rows
if world_rank == 0 then
    MPI_Scatter((rand_matrix[0][0]), mat_size, MPI_INT,
    (recv_buffer[0]), mat_size, MPI_INT, 0, MPI_COMM_WORLD)
else
    Send different rows
    MPI_Scatter(NULL, 0, MPI_INT, (recv_buffer[0]), mat_size, MPI_INT, 0, MPI_COMM_WORLD)
end
Do transpose of scattered rows MPI_Alltoall(recv_buffer, 1, MPI_INT, recv_buffer2, 1, MPI_INT,
  MPI_COMM_WORLD)
Gather to form matrix if world_rank == 0 then
    comm_time += MPI_Wtime()
    trans_matrix = create_matrix(mat_size, mat_size)
    comm_time -= MPI_Wtime()
    MPI_Gather(recv_buffer2, mat_size, MPI_INT, (trans_matrix[0][0]), mat_size, MPI_INT, 0,
      MPI_COMM_WORLD)
else
    MPI_Gather(recv_buffer2, mat_size, MPI_INT, NULL, 0, MPI_INT, 0, MPI_COMM_WORLD)
end
return 0
```

Algorithm 3: Write to output file

```
Function name: Main Function
Initialize
int i, j, rank, size, N=128
srand(time(0))
MPI_File fhw
MPI_Status status
MPI_Offset offset
MPI_Offset chunk
MPI_Init(argc, argv)
MPI_Comm_rank(MPI$_C OMM_W ORLD, rank)$
MPI_Comm_size(MPI_COMM_WORLD, size)
int buf[(N*N/size)]
int tmp
MPI_File_open(MPI_COMM_WORLD, ”data”,MPI_MODE_CREATE—MPI_MODE_WRONLY,
  MPI_INFO_NULL, fhw)
offset = rank*(N*N/size)*sizeof(int)
chunk = N*N/size
for i=0; i less than (N*N)/size ; i++ do
    int tmp = rand()%1000
    for j=0; j less than i; j++ do
        if tmp =buf[j] then
            tmp = rand()
        else
    end
    buf[i] =tmp
end
MPI_File_write_at_all(fhw,offset,buf,chunk,MPI_INT, status); MPI_File_close(fhw)
MPI _Finalize()
return 0
```

Algorithm 4: Void Transpose Function: Transposes all inner blocks of matrix

```
Function name: *void transpose-blocks
Initialization :int thread-counter = 0
long local = thread-counter
thread-counter++, int pos = 0
int posY = 0 , int tmp = size-mat/block-len
int x = 0 , tensor block

while 1 do
    pos = (block-len*local)
    x = local/tmp
    posY = x*block-len
    if posY ¿= size-mat then
        break
    else
        block = getSubMatrix(posY,pos)
        transpose(block)
        setSubMatrix(posY,pos,block)
        local= next-pos
        next-pos++
    end
end
```

B. Performance Tests

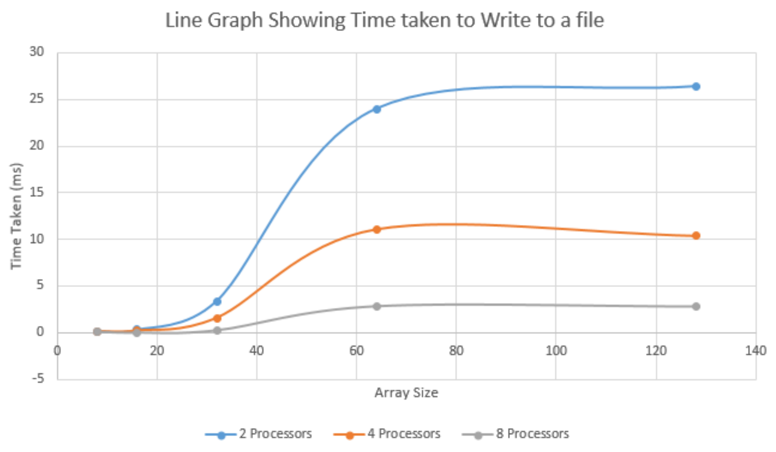


Figure 1: Time to write to File

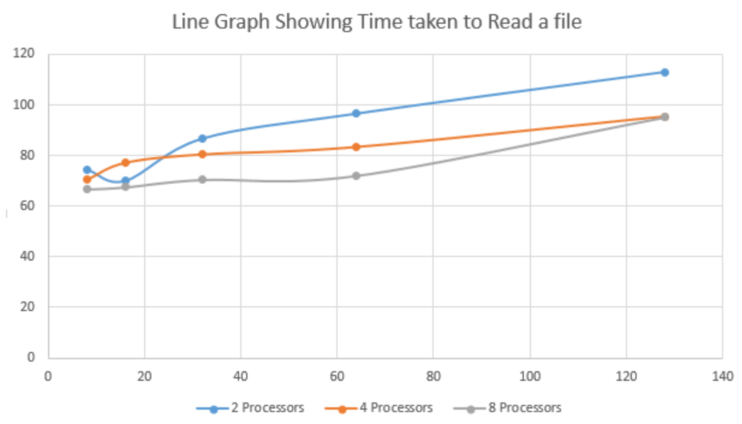


Figure 2: Time to Read from a file

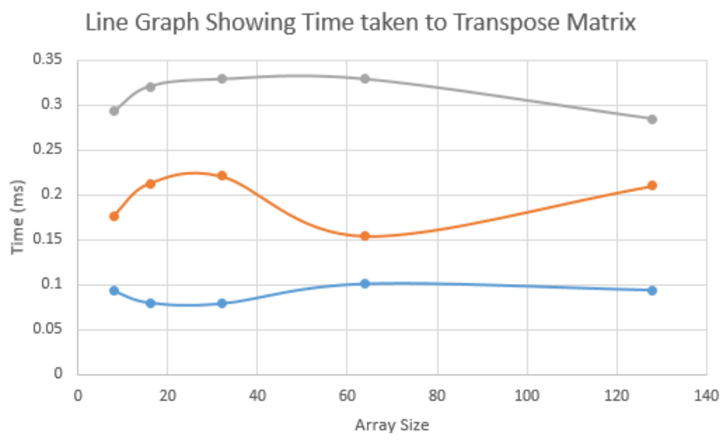


Figure 3: Time to Transpose Matrix