ELEN4020A: Out-of-Core Matrix Transposition of Big Data using MPI-IO

Kopantsho Mathafa (849038)   Chizeba Maulu (900968)   James Phillips (1036603)

May 17, 2019

Abstract

This paper presents the out-of-core matrix transposition of Big Data using MPI-IO. The out-of-core input file contains a 2-dimensional, square matrix with elements in row-major order. The rank of the matrix is 2^n where $n \epsilon \{3, 4, 5, 6, 7\}$. The file is read using MPI collective IO functionality. This involves a group of processes reading a chunk of the matrix. The performance of the program is presented in the report. The run time of the program is computed for varying numbers of processes and matrix sizes. It is found that the performance of the program decreases. This is due to the use of blocking send and receive routines which await all processes to complete before the program continues. The running time increases exponentially as the size of the matrix changes. This is intuitive as the size of the matrix increases the complexity when reading the input file, transposing the matrix and writing the result to the terminal. The running time can be reduce by not writing the output to the terminal.

I. Introduction

Computational operations on data-sets are generally performed using core memory. Very large datasets which exceed core memory pose both a memory capacity and computation challenge. Out of core data, or that stored on a hard disk, solves the memory capacity problem [1]. This solution entails saving the data on a local or remote file which is accessed by the program which contains the algorithm(s) to be applied to the data. However, accessing this file, provided it is large, is computationally expensive if performed sequentially. Collective IO presents an elegant solution to this problem by accessing the data using multiple processes which divide the data among themselves [2]. Once accessed, this data needs be operated upon by these processes. This entails transfer of data between these processes. One- or two-sided communication can be used depending on whether both processes are required to participate in the communication or need be synchronized.

This paper presents the transposition of an out-of-core matrix using one-sided communication functionality and derived datatypes defined in Message Passing Interface (MPI). MPI is a standard interface used to write distributed memory parallel programs [3].

The rest of the paper is organized as follows, Section III presents the problem under study. A brief literature review is performed in Section II. The implemented MPI program as well as the experimental environment in which it is run used are described in Section IV. Section V critically evaluates the results of the algorithm.

II. Literature Review

Fu et al. propose a framework based on MPI one-sided communication for graph processing [1]. Given the size and rapid growth of the graphs, they are difficult to store on a single machine. The framework is formulated across the global address space. It uses MPI `Put` to transfer data to enhance scalability and efficiency. The computation processes is divided into portions that can be overlapped. The framework has 35% faster job completion time than frameworks that employ two-sided communication to the same problem. He et al. apply one-sided communication to the computationally-intensive Computed Tomography (CT) image reconstruction process [4]. The process is used to deduce a patient's internal structure using a CT scans projections. The underlying algorithm used in the process is the Katsevich algorithm.

Zhou et al. employ the one-sided communication functionality defined in MPI-2 with an asynchronous engine for non-blocking communication operations [5]. A scheme is developed that uses an arbitrary amount of processes with to drive asynchronous progression as opposed to an additional thread within the application. The scheme intends to improve the communication cost between processes and ensure efficient communication is overlapped efficiently.

Similarly, Dinan et al. develop an MPI-based runtime for programs with Partitioned Global Address Space (PGAS) models [6]. They assert that the one-sided communication defined in MPI-2 does not sufficiently support higher-level global array parallel programming models. Their design enhances interoperability and portability of MPI and GA applications.

III. Problem description

A group of processes is required to generate a large, two-dimensional square matrix. The elements of this matrix are randomly generated integers with range 0-99. This matrix must be written to a file which will serve at the input file to the transpose algorithm. The format of this file is required to have the rank of the matrix as the first value; the rest of the values are the elements of the matrix in row-major order starting from $A[0][0]$, where A is the matrix. This matrix is the global input matrix to the program. Reading the data requires dividing the matrix between several processes. These processes will then each have a sub-matrix of the size defined by the programmer. Each process will perform the transposition. Once complete, the transposed sub-matrices are assigned to different processes. The transpose must be written to an output file in row-major order. The program must be tested with matrix sizes of 2^n where $n = \{3, 4, 5, 6, 7\}$. The number of processes P should have values $P = \{16, 32, 64\}$.

IV. Algorithm

The program comprises two main parts: collective IO (reading and writing) and transposition. First, a 2D matrix with random elements in row major of range 0-99 is generated by a separate program implementation. This matrix is written to a binary file is row major. This file is used as the input file to the program. Algorithm 1 describes how the file is read by the program. The file is read using *MPI_Read*. The processes' view of the data is changed using `File_set_view`. The matrix size is deduced in the program by using `Count` to determine the number of integers each process has and multiplied by the number of processes. The file is read in a row order, depending on the number of processors available. For example, a *4x4* matrix read by 2 processors will produce two rows of 4x2. Each one of these rows is held in the that processors memory and operations can thus be performed.

Algorithm 1 Using MPI to read file

```
Input: Global matrix input file
Output: Global matrix divided among processes
Open File
 Get file size
 Set offset
 Set offset size to $offset/size\ of\ int$
 Set buffer to $offset/number\ of\ processes$
 Set view of file
 Read file
 Get count
 Determine matrix size
 ;          // Wait for all processes to complete
Close file
```

Once the rows have been collected, they are mapped using a struct. This struct holds the x and y coordinate and the value of that point at the matrix. The x and y coordinates are calculated based on the determined matrix size and the number of processors. This process proves to work quite accurately provided the number of processes is less than the order of the matrix. It was decided to use this struct due to the rows being read into the file. Due to the order of these rows, it would be difficult to perform a transpose based on the position of each element, thus the transpose is performed by merely swapping the x and y coordinates in the struct. Once this swapping has taken place, the structs need to be sent to a common process for collection. It was decided to that all of these structs should be sent to a single process for collection. To achieve this, first a contiguous MPI datatype is defined which will be used to send the struct's which are in array form in each process. The sending is achieved using *MPI_Send* and all data is sent to process zero. Process 0 does contain its own matrix data and has to be collected with all received data. This is achieved with an additional array which contains all the received data. The data is received using *MPI_Recv*, and a *for* loop is iterated based on the number of processes.

Once all data is collected, it is placed into a serial matrix and is sorted to try and obtain the correct positions for the transposed data. This implementation did not work unfortunately, and was unable to be sorted correctly with time constraints. It is required that the output be printed in an output file, but difficulty was encountered with *stack overflow* errors when trying to print the output to a file. Figures are thus presented from the various stages of the implementation.

For the demonstration, the input files are generated an submitted for demonstration. It is possible to view the output by changing each respective file name to ”matrix”.

Algorithm 2 Inverted index function implementation

```
Input: Global matrix
Output: Transposed global matrix
for $i \leftarrow 0$ to $N$ do
    for $j \leftarrow 0$ to $N$ do
        temp ←matrix[i].x
          matrix[i].x←matrix[i].y
          matrix[i].y←temp
    end
end
```

Algorithm 3 Perform transpose using MPI datat-ypes

```
Input: Chucked input matrix
Output: Chuck matrix sent to slave
for $i \leftarrow 1$ to numprocs do
    $MPI\_Recv(recv, bufsize, data, i, 0, comm, status)$
end
;                // Complete other tasks
for $i \leftarrow 1$ to numprocs do
    $MPI\_SEND(elements, bufsize, data, i, 0, comm)$
end
```

V. Evaluation of results

The program is run on an 4-core 8 hyper-threaded machine with 16 GB of memory. The CPU has a clock rate of 3.7GHz. The results of the simulations can be found in the Tables I to V. It can be seen that the time taken for the program to run to completion increases as the number of cores increases for a fixed matrix size. A possible explanation for this is the overhead caused by the communication between the cores. It can also be seen that the time increases exponentially as the size of the matrices increase, this can be clearly seen in the performance graphs shown below. The program implemented required the printing of the matrix which can account for the significant increase in time as the dimensions of the matrices increased.

Figures 4 and 5 show a section of a matrix before and after transposition. Creating binary output files proved challenging for this reason a compromise was found by displaying the matrix in the terminal.

Overall the program performed as expected although the efficiency could have been improved to obtain better performance results. The performance could be better improved by not printing the matrices to the terminal. This is likely the largest contributor to slower performance of the program. Blocking routines were used in the implementation of the program, this will also slow down the general performance of the program. This would not have been the case had non-blocking routines been used, although special care would need to be taken to ensure that all data is in a predictable and valid state.

TABLE I
TABLE INDICATING THE NUMBER OF PROCESSES USED AND TOTAL RUNNING TIMES FOR $A[8][8]$

No. of processes	Job completion time (sec)
4	0.00122
8	N/A
16	N/A

TABLE II
TABLE INDICATING THE NUMBER OF PROCESSES USED AND TOTAL RUNNING TIMES FOR $A[16][16]$

No. of processes	Job completion time (sec)
4	0.00196
8	0.00646
16	N/A

TABLE III
TABLE INDICATING THE NUMBER OF PROCESSES USED AND TOTAL RUNNING TIMES FOR $A[32][32]$

No. of processes	Job completion time (sec)
4	0.00861
8	0.02465
16	0.063486

TABLE IV
TABLE INDICATING THE NUMBER OF PROCESSES USED AND TOTAL RUNNING TIMES FOR $A[64][64]$

No. of processes	Job completion time (sec)
4	0.00815
8	0.1585
16	0.972193

TABLE V
TABLE INDICATING THE NUMBER OF PROCESSES USED AND TOTAL RUNNING TIMES FOR $A[128][128]$

No. of processes	Job completion time (sec)
4	1.1667
8	2.058
16	4.67502

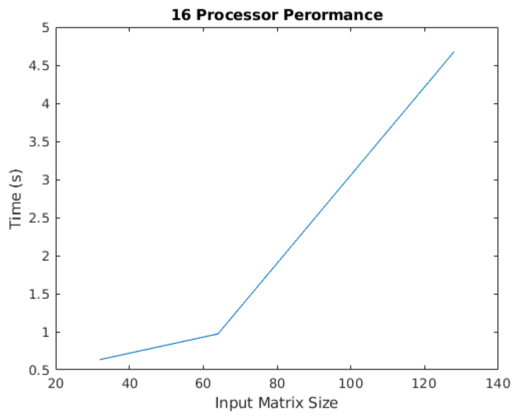


Fig. 1. Performance graphs of algorithm running on 16 cores

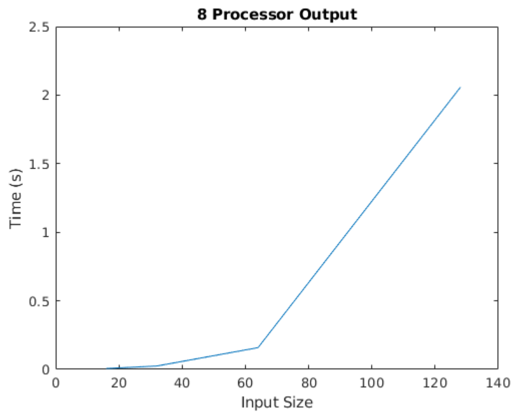


Fig. 2. Performance graphs of algorithm running on 8 cores

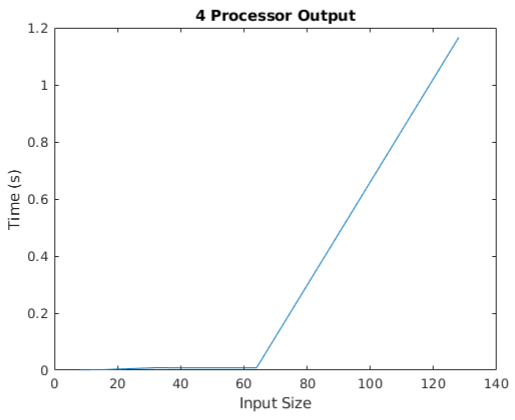
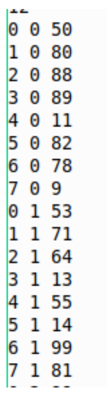
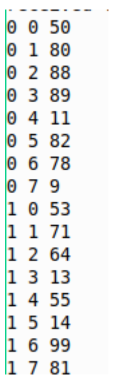


Fig. 3. Performance graphs of algorithm running on 4 cores



```
0 0 50
1 0 80
2 0 88
3 0 89
4 0 11
5 0 82
6 0 78
7 0 9
0 1 53
1 1 71
2 1 64
3 1 13
4 1 55
5 1 14
6 1 99
7 1 81
```

Fig. 4. Snippet of the matrix before transposition



```
0 0 50
0 1 80
0 2 88
0 3 89
0 4 11
0 5 82
0 6 78
0 7 9
1 0 53
1 1 71
1 2 64
1 3 13
1 4 55
1 5 14
1 6 99
1 7 81
```

Fig. 5. Snippet of the matrix after transposition

VI. Conclusion

The transpose of an out-of-core matrix is presented. MPI collective IO is used to use a group of processes read the file containing the matrix elements. This process creates various sub-matrices. These sub-matrices are used to generate a contiguous MPI derived datatype. The transpose algorithm is applied to this contiguous type. The resulting matrix is again divided between processes using send and receive methods. The root process is responsible for the send the sub-matrices to the other process. The output of the program is printed to the terminal. The time to run to completion is measured for each matrix rank size and process number. The running time increases exponentially as the size of the matrix increases, assuming the number of processes is kept stable. This is to be expected as the complexity is related to how long it takes to use processes to read sub-matrices, transpose them and finally write the result to the terminal. The running time increases as the number of processes increases. This is maybe due to the blocking send and receive routines which wait for all the processes to complete before moving to the next instruction.

References

[1] H. Fu, M. Gorentla Venkata, S. Salman, N. Imam, and W. Yu, “Shmemgraph: Efficient and balanced graph processing using one-sided communication,” in *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, May 2018, pp. 513–522.

[2] S. Seo, R. Latham, J. Zhang, and P. Balaji, “Implementation and evaluation of mpi nonblocking collective i/o,” in *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, May 2015, pp. 1084–1091.

[3] Byna, Gropp, Xian-He Sun, and Thakur, “Improving the performance of mpi derived datatypes by optimizing memory-access cost,” in *2003 Proceedings IEEE International Conference on Cluster Computing*, Dec 2003, pp. 412–419.

[4] T. He, J. Ni, J. Deng, H. Yu, and G. Wang, “Deployment of one-sided communication technique for parallel computing in katsevich ct image reconstruction,” in *First International Multi-Symposiums on Computer and Computational Sciences (IMSCCS’06)*, vol. 1, June 2006, pp. 416–423.

[5] H. Zhou and J. Gracia, “Asynchronous progress design for a mpi-based pgas one-sided communication system,” in *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*, Dec 2016, pp. 999–1006.

[6] J. Dinan, P. Balaji, J. R. Hammond, S. Krishnamoorthy, and V. Tipparaju, “Supporting the global arrays pgas model using mpi one-sided communication,” in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, May 2012, pp. 739–750.