Parallel Matrix Transposition Using MPI With Derived Datatypes

Sanele Ndlovu-716411
Knowledge Dzumba - 813137
School of Electrical and Information Engineering, University of the Witwatersrand, Johannesburg
ELEN4020A:Data Intensive Computing in Data Science

Abstract—**A parallel matrix transposition algorithm was partially implemented using MPI with derived datatypes. Data for the matrix was randomly generated by different processes and later combined by the master (rank 0) process to give the full matrix. The algorithm presented makes use of the MPI_Type_create_subarray derived datatype to create subarrays of a larger matrix that are further transposed individually and then merged back together to form a transposed matrix. Due to difficulties encountered during the implementation phase, the transposed subarrays could not be merged into one output array.**

Index Terms—**MPI, MPICH, Parallel transposition, collective I/O**

I. Introduction

This paper presents the development and partial implementation of parallel matrix transposition using MPI derived datatypes and collective I/O. Performing operations on large datasets using traditional procedural methods of programming can have a significant impact on the efficiency and scalability of the program. With the shift in computer processor architecture and the current dominance of multi-core processors, parallel programming brings a solution to the problem of inefficient processing of large datasets. Parallel computing is divided into two categories, shared-memory computing and distributed memory computing. In shared memory computing, a collection of autonomous processors is connected to the main memory via an interconnection network and each processor has direct access to this main memory[1]. Shared memory computing facilitates communication between these processors through accessing shared data structures. Distributed memory computing comprises of multiple processors, which are each paired with their own private memory. The processors communicate with each other via an interconnection network, and this is usually done explicitly through message passing or through the use of functions that are specifically implemented to facilitate this type of communication. MPI (Message Passing Interface) is a standard for message passing in distributed memory systems, and it is mostly used with C, C++ and Fortran programming languages. Different implementations of this standard exist, the major ones being OpenMPI and MPICH. In this paper, the code was implemented using MPICH3.3 implementation of MPI. MPI derived data types are that standard's way of attempting to consolidate data that might otherwise require multiple messages so that they can be communicated between different processes. They are necessary because they make communication between processes cheaper since fewer messages get to be passed between processes. This is because in distributed memory systems, the cost of sending messages between processes is greater than the cost of performing computations in the local process[1].

Matrix transposition is one of the basic fundamental operations that can be performed on matrices, and they have a lot of applications in areas where data is to be processed and manipulated in a particular way, including having applications in many numeric algorithms such as FFT and the conversion of storage layout for arrays[2]. The focus of this paper is on parallel transposition of square matrices using MPI. The problem of a square matrix transposition can simply be explained as follows: Given a square $N \times N$ matrix, a new matrix has to be constructed such that row i of the constructed matrix is similar to column i of the original matrix. This means that the transposition operation exchanges the rows and columns of some matrix M.

II. Problem Description

It was required that a parallel matrix transposition algorithm be developed and implemented using MPI with derived datatypes. The matrix was to be made up of random short integers that are generated simultaneously by a collection of processes which could either be 16, 32 or 64. The matrices generated were to be of different sizes, specifically 2^N where N is in the set $3, 4...7$. The implementation was required to make use of collective I/O for reading and writing matrix data to files and it also had to make use of MPI's derived datatypes. Collective I/O is in this paper referred to as a type of communication in which all processes belonging to a particular MPI communicator are involved.

III. Input Generation

To generate the contents for the matrices, the random number generator provided by the C-language standard library was used. Given the size of the matrix that was to be generate, all processes started by the program generated a sequence of N_local random short integers. In this case, N_local was chosen according to a simple partitioning scheme that divided the total number of elements the matrix is to store by the number of processes that have been started by the running program, that is

$$N\_local = \frac{N}{comm\_size} \quad (1)$$

After generating N_local random short integers, each process stored them in a local buffer of the same size. Each process then wrote the content of its buffer to a shared file, and no order of writing to the file was followed in this operation. The code that was used to achieve this can be seen in Listing 1.

```

    //Each process should generate enough elements to fill a squre submatrix of main matrix
    local_N = (N * N / size);

    offset = rank * local_N * sizeof(int) + sizeof(int);

    int local_Buffer[local_N];

    for (int i = 0; i < local_N; i++)
    {
        local_Buffer[i] = rand() % 100;
    }

    //All processes write their randomy generated matrix elements to the same file
    MPI_File_write_at_all(fhw, offset, local_Buffer, local_N, MPI_INT, &status);
```

Listing 1. input matrix file generation

Although the different processes wrote their randomly generated integers to the input matrix file, its content still had to be made available to the code for further processing of an actual 2d matrix. To do this, the master process (rank 0) was used to read the contents of the input matrix file into a global 2d array representing the matrix to be transposed. Listing 2 shows a snippet of the code that was used to achieve this part of the input matrix generation.

```

     //Creating the input matrix to be transposed. This is done by the master process
    if (rank == 0)
    {
        //Start reading actual matrix data at an offset of sizeof(int) to account for the
        //added matrix size
        MPI_File_read_at(fhw, sizeof(int), &inputMatrix[0][0], N, MPI_INT, MPI_STATUS_IGNORE);

        for (int i = 1; i < size; i++)
        {
            //Fill the global matrix from elements in the inputMatrixFile
            MPI_File_read_at(fhw, i * sizeof(int) * local_N + sizeof(int), &inputMatrix[i][0], N, MPI_INT,
                                            &status);
        }
```

Listing 2. input matrix generation

IV. Transposition Algorithm

The input matrix that was previously generated was chunked using MPI's derived datatype for generating subarrays. This was done so that a block transposition algorithm could be implemented, in which chunks of the original matrix are transposed separately and the overall transposed matrix acquired through combining these transposed chunks together. Although we failed to implement the combination of transposed chunks into an overall transposed matrix, an algorithm of how this was implemented as well as an argument for its correctness is discussed in this section. Given the sample matrix in Fig. 1. to be the matrix to be transposed. Suppose 16 processes are to be used in the transposition of this matrix. Then the matrix can be chunked into 16 smaller 2×2, examples of which are shown in blue and brown. Each of these smaller chunks are transposed independently by the processes that have been assigned to them. The assignment of processes to different subarray is performed by the master process, which also participates in the transposition of its self-assigned subarray. Given that the block length for the subarrays is 2 in this example, the master process assigns the block starting at index $[0; 1 \times blockSize]$ to the successor process 1, followed by the assignment of the block starting at $[0; 2\times blockSize]$ to process number 2. This process of assignment continues in this order as long as $i \times blockSize$ does not exceed the length of one dimension of the original matrix to be transposed (N). In the event that $i \times blockSize$ exceeds the length of the matrix size N, the assignment is moved to the block starting at index $[blockSize; 0]$ and so on until all the blocks of the same size have been assigned to a process in the communicator. Once the assignment of blocks has been done, each process performs a transpose operation to the smaller chuck that it has been assigned to by applying a trivial transpose function as shown in Listing 3.

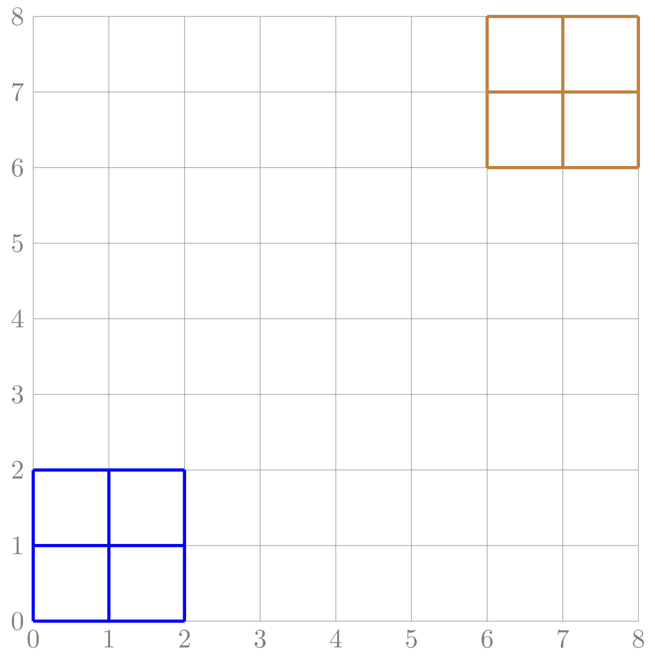

Fig. 1. Sample 8×8 matrix

```
int **transposeMatrix(int **matrix, int n)
{
    int **transposed = allocArray(n);
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            transposed[i][j] = matrix[j][i];
        }
    }

    return transposed;
}
```

Listing 3. transpose computation function

Once the transposition of the chunks is complete, they have to be merged back into a complete matrix that is itself a transposed version of the original matrix. This requires a second transpose, but this time, instead of the numbers within the chunks being interchanged, the subarrays are the ones that get interchanged. This means that once process 0 is done transposing the blue subarray shown in Fig. 1. and process 12 is done transposing the brown subarray, these two subarrays are then interchanged such that process 0 now handles the transposed brown subarray while process 12 handles the transposed blue subarray.

A. Programming Environment

The implemented code was compiled using GCC-7.0.4 on a Lenovo Ideapad 320 laptop with a RAM of 4GB and a processor speed of 2.5 GHz. MPICH3.3 was used to execute the program. Although MPI is a standard for distributed memory programming and allows for the use of different nodes to run the tasks, only one laptop was used to test and run the program implemented. Since no other

V. Observation and Recommendations

It was observed that using the $rand()$ and the current time for random number generation for parallel programming is not ideal because in the generation of the input array, most processes generated the same numbers because they were being run exactly at the same time, hence the random number generator seed kept producing the same results. Difficulties that led to the combination of chunks of matrices back into a fully transposed matrix were identified to be due to the memory layout which was chosen for representation of arrays and subarrays. They made it difficult to combine the chunks back together into one array, hence it would be recommended that for future work, better care be taken in the development stage of the algorithm so that problems like this can be avoided.

VI. Conclusion

Worked done on the development and implementation of a parallel algorithm for matrix transposition has been presented. The algorithm is a block transposition algorithm in which a process is assigned a smaller chunk of the main matrix to be transposed and performs the transpose operation on this smaller chunk. On completion of this transposition, the blocks that have been assigned to different processes are again transposed using the same idea of interchanging rows and columns, but this time around, the chunks of transposed data and not the individual elements within the subarrays. The implementation of combining back the chunks into a complete matrix was not implemented, but an overview of how the algorithm was intended to operate has been described. The use of $rand()$ together with $srand(time(NULL))$ has been found to be non-ideal for programming parallel systems because one ends up with the same numbers generated by most of the processes involved.

References

[1] P Pacheco, *An Introduction to parallel programming*. Morgan Kaufmann: Fourth Edition

[2] J Gomez-Luna, I-J Sung, L Chang, J M Gonzalez, N Guil, W. W Hwu, *In-Place Matrix Transposition on GPUs*, IEEE Transactions on Parallel and Distributed Systems, vol.27, No.3, March 2016

[3] https://www.merriam-webster.com/dictionary/risetime Accessed 12 April 2019