Name: Brijesh Mavani
CWID: A20406960
University: Illinois Institute of Technology
Course: Parallel and Distributed Processing
Final Project – Design Report

Fast Fourier Transformation:

Fast Fourier Transformation (FFT) is a highly parallel "divide and conquer" algorithm for the calculation of Discrete Fourier Transformation of single-, or multidimensional signals. It can be efficiently implemented using the parallel programming model such as MPI, CUDA etc. The basic outline of Fourier-based convolution is:

- Apply direct FFT to the convolution kernel,
- Apply direct FFT to the input data array (or image),
- Perform the point-wise multiplication of the two preceding results,
- Apply inverse FFT to the result of the multiplication

Following task subdivision provided in the project description:

A=2D-FFT(Im1)	(task1)
B=2D-FFT(Im2)	(task2)
C=MM_Point(A,B)	(task3)
C=Inverse-2DFFT(C)	(task4)

The parallel algorithm using MPI is as follows:

1. The processor 0 will read the input files and load the data into matrices A and B.

   The following steps are related to Task 1 and 2.
2. The processor 0 will distribute the rows of input matrices A and B to other processors. This will be the part of the communication.
3. All the processors will compute the forward 1D-FFT on the rows of matrices A and B belong to that processor. This will be the part of the computation.
4. All processors will send the result back to the processor 0. This will be the part of the communication.
5. The processor 0 will transpose the matrices A and B resulted in 1D-FFT operation on rows. This will be the serial computation.
6. Processor 0 will distribute the rows of transposed matrices A and B to other processors. This will be the part of the communication.
7. All the processors will compute the forward 1D-FFT on the rows of matrices A and B belong to that processor. This will be the part of the computation.
8. All processors will send the result back to the processor 0. This will be the part of the communication.

   The following step is for task 3.
9. Processor 0 will compute the point to point multiplication on matrices A and B and store result in matrix C. This will be the part of the computation.

   Following steps are for task 4.
10. Processor 0 will distribute the rows of matrix C to all other processors. This will be the part of the communication.

11. All the processors will compute the inverse 1D-FFT on the rows of matrix C belong to that processor. This will be the part of the computation.
12. All processors will send the result back to the processor 0. This will be the part of the communication.
13. The processor 0 will transpose the matrix C resulted with 1D-FFT operation on rows. This will be the serial computation.
14. Processor 0 will distribute the rows of transposed matrix C to other processors. This will be the part of the communication.
15. All the processors will compute the inverse 1D-FFT on the rows of matrix C belong to that processor. This will be the part of the computation.
16. All processors will send the result back to the processor 0. This will be the part of the communication.
17. Processor 0 will write the final output in a file.

As input size for given files is 512x512, the size of the matrices is hard-coded at the beginning of each program file. This can be changed to execute the code for any other size by modifying below line in program file:

#define N 512

This is located on line 22 in all files.
Following are the details of each implementation.

A. **MPI send and receive (Part A):**
The algorithm is provided above is used here. The MPI communication operations were performed using the MPI_Send and MPI_Recv functions.

The MPI function MPI_Wtime() has been used to capture the timing information. The communication and computation timing details are provided below:

Number of processors	Total Elapsed Time (ms)	Computation Time (ms)	Communication Time (ms)	Speedup (Ts/Tp)	Efficiency (Sp/P)
1	33.044	33.044	0	-	-
2	24.419	15.937	8.482	1.35	0.675
4	20.047	8.235	10.811	1.65	0.413
8	18.506	6.533	11.973	1.79	0.224

B. MPI Collective (Part B):
The implementation is same as of MPI send and receive the code. The only change is that MPI_send and MPI_Recv functions were replaced by the MPI_Bcast and MPI_Gather.

As we can see in both tables (part A and Part B), there is not much difference in timings while implementing the program using MPI point to point communication and MPI Collective communications. Some differences can be seen in the communication. From the experiments, it can be inferred that the collective calls improve communication performance when the number of processes grows, but as execution done with maximum 8 processors, this is a too small sample to reach conclusions.

Number of processors	Total Elapsed Time (ms)	Computation Time (ms)	Communication Time (ms)	Speedup(Ts/Tp)	Efficiency (Sp/P)
1	38.723	37.067	1.656	-	-
2	28.968	22.846	6.122	1.34	0.67

4	23.768	15.593	10.575	1.63	0.408
8	21.513	10.001	11.512	1.8	0.225

C. Task and Data Parallel Model (Part C):

In this model, we need to divide the processors into 4 groups where each group will be in charge of one task. As the number of the processors are 8 (as provided), each group will consist of the 2 processors. To follow the dynamic programming practice, nothing is hardcoded into the code (except input matrix size). The same code can be executed for 16, 32 processors where each group will consist of 4 and 8 processors respectively. As number groups are four, the number of processors should be multiple of 4. MPI_Group function has been used to group the processors. The 4 communicators are created for the intra-communications. The details are provided below:

Group	Communicator	Processor rank
P1	P1Comm	0,1
P2	P2Comm	2,3
P3	P3Comm	4,5
P4	P4Comm	6,7

The algorithm is changed as Processor 0 is no longer the one doing the sequential computations. The updated the algorithm is provided below:

1. The source process will read the input files and load the data into matrices A and B.

   Below steps will be the part of communication.
2. The source process will distribute the input matrix A to the processors in P1.
3. The source process will distribute the input matrix B to the processors in P2.

Task 1 on P1 processors:

1. P1 processors will perform forward 1D FFT on the matrix A. This will be the part of the computation.
2. P1 first processor will collect the resulted matrix A from the other P1 processors. This will be the communication part.
3. P1 first processor will transpose the matrix A resulted with 1D-FFT operation on rows. This will be the serial computation.
4. P1 first processor will distribute the transposed matrix A to other processors in P1. This will be the communication part.
5. P1 processors will perform forward 1D FFT on the matrix A. This will be the part of the computation.
6. P1 processors will send their respective data of matrix A to the corresponding processors in P3.This will be the communication part.

Task 2 on P2 processors:

1. P2 processors will perform forward 1D FFT on the matrix B. This will be the part of the computation.
2. P2 first processor will collect the resulted matrix B from the other P2 processors. This will be the communication part.
3. P2 first processor will transpose the matrix B resulted with 1D-FFT operation on rows. This will be the serial computation.
4. P2 first processor will distribute the transposed matrix B to other processors in P2. This will be the communication part.

5. P2 processors will perform forward 1D FFT on the matrix B. This will be the part of the computation.
6. P2 processors will send their respective data of matrix B to the corresponding processors in P3.This will be the communication part.

Task 3 on P3 processors:

1. P3 processors will compute the point to point multiplication on the part of matrices A and B they received from P1 and P2 processors and store the result in the matrix C. This will be the computation part.
2. P3 processors will distribute their part of matrix C to the corresponding P4 processors. This will be the communication.

Task 4 on P4 processors:

1. All the processors in P4 will compute the inverse 1D-FFT on the rows of matrix C belong to that processor. This will be the part of the computation.
2. P4 first processor will collect the resulted matrix C from the other P4 processors. This will be the part of the communication.
3. P4 first processor will transpose the matrix C resulted with 1D-FFT operation on rows. This will be the serial computation.
4. P4 first processor will distribute transposed matrix C to other processors in P4. This will be the part of the communication.
5. All the processors in P4 will compute the inverse 1D-FFT on the rows of matrix C belong to that processor. This will be the part of the computation.
6. All processors in P4 will send the part of matrix C belongs to that processor back to the source processor. This will be the part of the communication.

   At the end, the source processor will write the final result (matrix C) into the file.

From above algorithm, it can be inferred that it requires more communication. The processors in P3 needs the input from P1 and P2 both processor groups while earlier there wasn't any need for this communication. Also, some processors will be idle while other processors busy computing or communicating with another processor group.

Computation and communication timings for P1=P2=P3=P4=2 are as below:

Number of processors	Total Elapsed Time (ms)	Computation Time (ms)	Communication Time (ms)	Speedup (Ts/Tp)	Efficiency (Sp/P)
P1=P2=P3=P4=2	22.870	13.410	9.460	33.044/22.870 = 1.44	1.44/8 = 0.18

D. Comparison between cases A and C (Part D):

As we can see from the results of cases A and C, case A performs better in this particular implementation. This is due to the fact that in case A all processors are busy in either computing or communicating. Whereas in C some processors will be idle/free while other processors busy computing or communicating with another processor group. The performance of the case C could be different if there are multiple jobs to work on and all processor groups work on different jobs. This will make all processor busy with some task. However, this needs to be evaluated further but we can suppose the performance will be improved as it will make all processors to work on jobs and keep them busy.

Correctness for each implementation:

I have used MPI_Barrier function to ensure the synchronizartion of all processors. The MPI_Barrier has been used before computation and communication. I have also compared the output of my execution with the out file provided by professor on Blackboard. If the data in both file matches I concluded that my implementation is working as expected. So, by using MPI_Barrier and the comparing the output file, I ensure the correctness for my implementation.

Appendix:

The design document, result analysis and README files can be found in the zip file uploaded to blackboard. As the average number of lines for each implementation is around 550 lines, the total lines for all 3 types of implementation are around 1650 lines. Due to this program is not provided here. The code files are uploaded to blackboard also same files can be found on the comet system.

As the resulting file for 2D convolution is around 3MB, it is also not provided in zip file. Those files are present on the Comet system. The execution scripts are also present on the comet system.

Path on the Comet system : /home/bmavani/Project -- All c file, scripts, output files are present at this path.

You will find following C program files:

1. ProgrammingProject_2DConvolution_MPISendReceive_BrijeshMavani.c – Implementation with MPI point to point communication with MPISend and MPI_Recv functions for part A.
2. ProgrammingProject_2DConvolution_MPICollective_BrijeshMavani.c – Implementation of MPI collective communication with MPI_Bcast and MPI_Gather functions for part B.
3. ProgrammingProject_2DConvolution_MPITask_BrijeshMavani.c – Implementation with task parallelism with four groups for part C.

Scripts are created for the execution of the code:

1. MPIP2P<1/2/4/8>_job.sh – Scripts to execute the ProgrammingProject_2DConvolution_MPISendReceive_BrijeshMavani.c for number of processors mentioned in filename<1/2/4/8>.
2. MPICollective<1/2/4/8>_job.sh – Scripts to execute the ProgrammingProject_2DConvolution_MPICollective_BrijeshMavani.c for number of processors mentioned in filename<1/2/4/8>.
3. MPITask8_job.sh – Script to execute the ProgrammingProject_2DConvolution_MPITask_BrijeshMavani.c on the 8 processors.

Output files:

1. mpi_sendrecv_output – Output of MPI point to point communication implementation (part A). The format is same as the provided input files.
2. mpi_collective_output – Output of MPI Collective communication implementation (part B). The format is same as the provided input files.
3. mpi_task_output – Output of task parallelism implementation (part C). The format is same as the provided input files.