Project Report

The project requirement is to perform 2D Convolution on two 512x512 images using a parallel algorithm.

As per the requirement, I am following the below steps to perform 2D Convolution:

a. Read image file and store it in a matrix
b. Perform 1D FFT on each row of the matrix
c. Transpose the input matrix
d. Again, perform 1D FFT on each row of the matrix
e. Follow the above steps for the next image file
f. Use point wise multiplication to multiply both input matrices
g. Perform row wise 1D FFT on output matrix
h. Transpose the output matrix
i. Again, Perform row wise 1D FFT on output matrix
j. The output matrix will be the result matrix

I have implemented different functions for file read and write along with 1D FFT. File read/write functions are called once initially from the main function. Data distribution and computation logic are written in main function.

Sequential Code:

I have implemented the above steps in a sequential fashion without any parallelism. The code compiles and runs perfectly without any errors. The execution time of the sequential code is around 6000 ms.

Parallel Code:

Now, I have implemented the above algorithm parallely in four different ways:

- Using MPI Send and Receive
- Using MPI Collective Operations
- Using MPI-OpenMP
- Using MPI Communicator

I am using a similar logic for first three programs with specific modifications on each ones. Algorithm:

1. Call file read function to read the two image files
2. Distribute the input matrices among all processors equally (Since size is even number)
3. Each processor performs 1D FFT on chunk no. of rows
4. Processor 0 collects the input matrices from all the processors, performs transpose and redistribute among all processors.

5. Each processor again performs 1D FFT on chunk no. of rows (transposed columns).
6. Then each processor perform Point Multiplication and 1D Inverse FFT on chunk no. of rows of output matrix.
7. Processor 0 collects the chunk rows of output matrix from rest of the processors and store in its self-copy.
8. Processor 0 then performs transpose on output matrix and distributes them across all processors for another round of 1D Inverse FFT.
9. Each processor performs 1D Inverse FFT on chunk no. of rows of output matrix and sends back to processor 0.
10. Processor 0 Calls the file write function to display the function

Part A - Using MPI Send and Receive

I have used the MPI_Send and MPI_Recv functions to distribute and collect matrix from the processors. The code is compiled and run on 1, 2, 4 and 8 processors.

<u>Performance Evaluation:</u>

As you can see from the below table, with a single processor there will be zero communication time but high computation time. And as number of processors increase, communication time increases and computation time decreases. Also, speedup is decreasing due to rise in communication cost. So I am getting super linear speedup with 2 no. of processors and linear speedup with 4 processors.

	Total Time (ms)	Communication Time (ms)	Computation Time (ms)	Speedup
p=1	1035.38596	0.00596	1035.38	-
p=2	1068.4	463.61	604.79	5.615874204
p=4	1569.1	1282.38	286.72	3.823848066
p=8	2525.02	2346.05	178.97	2.376218802

Part B - Using MPI Collective Operations

In this program as per my algorithm broadcasting the entire matrix is not required. Also, MPI_SendRecv won't work since it is very costly. So I have used MPI_Scatter and MPI_Gather to distribute and collect the input output matrices.

<u>Performance Evaluation:</u>

In this case, the computation time is better than Part A but communication time is almost similar or rising. Hence I am getting better speedup with 2 & 4 processors.

	Total Time (ms)	Communication Time (ms)	Computation Time (ms)	Speedup
p=1	994.332	2.352	991.98	-
p=2	767.512	342.002	425.51	7.817467349
p=4	1987.54	1666.07	321.47	3.018807169
p=8	3095.33	2937.8	157.53	1.938403983

Part C - Using MPI-OpenMP

In this program I am using the combination of MPI and OpenMP since most of the 'for' loops can be parallelized without adding any restrictions. So Pthreads is not much of requirement. Hence, using one of the above methods I am parallelizing for loops in all the operations.

Performance Evaluation:

Addition of extra level of parallelism via openMP has definitely improve the computation cost from the A & B but the communication time remains the same. I have declared 4 threads to perform "for" loop operations parallely.

	Total Time (ms)	Communication Time (ms)	Computation Time (ms)	Speedup
p=1	1008.505	0.005	1008.5	-
p=2	1034.74	498.47	536.27	5.798558092
p=4	2238.84	1963	275.84	2.679959265
p=8	2951.84	2854	97.84	2.032630495

Part D - Using Task & Data Parallel Model

In this program, I have to use MPI Communicator to create group of processors and assign separate tasks to each group. As per the code goes, there will be 4 groups each containing some processor(s) depending on the number of processors. I am creating four communicators one for each group to communicate within the group. For Inter group communication I am using default communicator.
Group 1 & 2 each will perform 2D FFT on one of the images.
Group 3 will perform Matrix Multiplication
Group 4 will perform 2D FFT on output matrix and display the result.

Algorithm:

a. Initially, group 1 will send matrix 2 to group 2.
b. Group 2 will receive the matrix and both 1 and 2 will scatter their matrices inside the group amongst their fellow processors.
c. Group 1 & 2 will then perform 2D FFT on their matrix.
d. After that, group 1 & 2 will send the intermediary results to group 3 for matrix multiplication.
e. Group 3 will receive matrices from group 1 and group 2 and scatter amongst his own fellow processors.
f. Group 3 will then perform matrix multiplication of both the matrices.

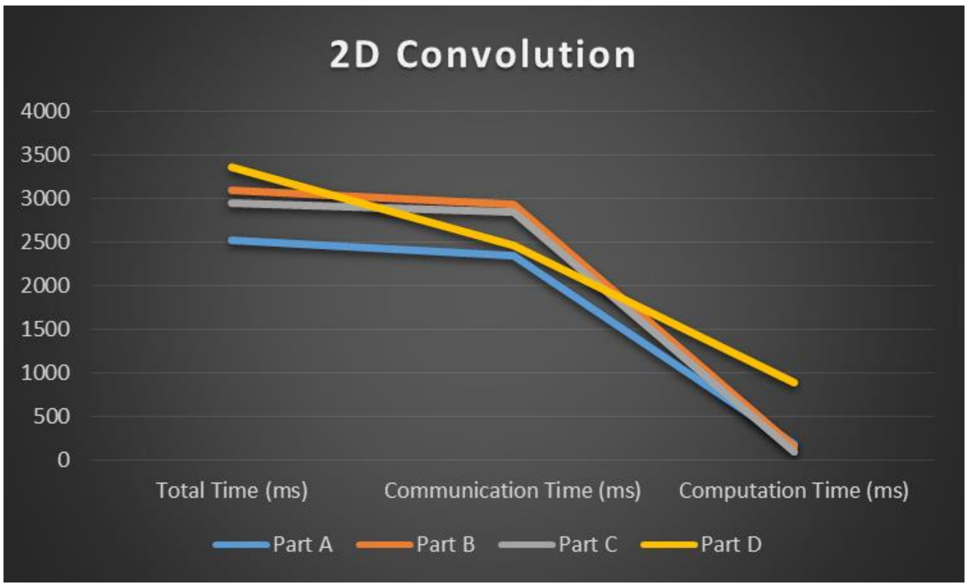
g. After that group 3 will send the output matrix to group 4 for 2D IFFT.
h. Group 4 will receive the output matrix from group 3 and scatter amongst his own fellow processors.
i. Group 4 will then perform 2D IFFT on the output matrix and write the output matrix to file.

The computation and communication time comes down to 890ms and 2471ms with 8 processors in 4 groups of 2. Hence, I am getting a sub-linear speedup of around 1.78.

Part E - Comparison

From the below table and figure, I can deduce that despite different computation time in all the parts the total time is almost of the same range due to communication time. Considering only computation time Part C – Using MPI-OpenMP gives me better performance followed by Part A – Using MPI Send Recv. Overall Part A has the highest speedup amongst all because of slightly less communication time.

	Total Time (ms)	Communication Time (ms)	Computation Time (ms)	Speedup
Part A	2525.02	2346.05	178.97	2.376218802
Part B	3095.33	2937.8	157.53	1.938403983
Part C	2951.84	2854	97.84	2.032630495
Part D	3362.03	2471.1	890.93	1.784636068



To reduce the communication time, I tried performing the transpose at individual processors to avoid multiple send receive across different processors but I wasn't getting the desired output. If we can do that, we can bring the communication cost to zero since only once we need to perform send and receive among processors.

Note: All the codes are compiled and tested on Jarvis.

Appendix – Source Code

The source code of all the four programs are attached with this document. The codes are also uploaded on Jarvis under my username – shorabhd inside Project folder.

There are five source codes:

Part.c - This is the sequential version of the code. I used this code to calculate the execution time for sequential code to calculate speedup.

PartA.c: This is part A parallel version using MPI Send/Recv

PartB.c: This is part B parallel version using MPI Scatter/Gather

PartC.c: This is part B parallel version using MPI along with OpenMP

PartD.c: This is part D parallel version using MPI Communicator

Scripts:

runA.sh: Script file to run Part A on Jarvis Cluster

runB.sh: Script file to run Part B on Jarvis Cluster

runC.sh: Script file to run Part C on Jarvis Cluster

runD.sh: Script file to run Part D on Jarvis Cluster

I have used 1_im1, 1_im2 as input files provided in sample to test my code. A file with the name output will be generated after successful execution of the code.

Readme.txt contains the steps to run the code.