Project Performance Analysis

MPI with primitive operations-

Description: The FFT calculation operation is parallelized in this method to increase the performance. The matrix is scattered across all the processes using static block allocation (static interleaving was tried and discarded as it required too may communication operations). Each process calculates the row wise 1D fast Fourier transform on its own block. Once it is done, the intermediate block is transferred to process 0, which gathers the entire matrix, calculate the transpose of that, and re-scatters it to all processes once again. Repeating the same process once again gives us the fast Fourier transform of both A and B matrix.

The transpose is not parallelized as it is a cheap operation, same as the matrix point multiplication. After calculating the matrix point multiplication, inverse 2D FFT is calculated in parallel, to get the result.

Different Versions tried:

1. The first version of the program used static interleaving strategy for distributing the data. It required too many communication operations, and hence the performance was even worse. Hence it was replaced with static block strategy.

2. In the second and final version, the functionality to monitor performance (timings) using MPI_Wtimer() was added. Both computation and communication time are recorded and displayed.

Note: Jarvis environment was used for testing and performance analysis.

MPI with collective functions-

Description: It is exactly the same algorithm as discussed above. The only difference is the custom methods of scattering and gathering the data are replaced by the API provided MPI_Scatter() and MPI_Gather() methods. The MPI_Bcast() is also used for broadcasting the number of rows to all processes.

Different Versions tried:

1. Only one version was tried, which is the final version.

MPI with OpenMP (Hybrid)-

Description: The algorithm is same as the previous one. The only difference is that the serial operations like transpose, matrix point multiplication, conversion from real to complex, and vice versa are parallelized using the OpenMP.

Different Versions tried:

1. The first version had both the loops parallelized in transpose and matrix point multiplication. It was decreasing the performance.

2. In the second version, only outer loop was parallelized.

MPI with OpenMP (Hybrid)-

Description: The algorithm is same as the previous one. The only difference is that the serial operations like transpose, matrix point multiplication, conversion from real to complex, and vice versa are parallelized using the OpenMP.

Different Versions tried:

1. The first version had both the loops parallelized in transpose and matrix point multiplication. It was decreasing the performance.

2. In the second version, only outer loop was parallelized.

MPI with task and data parallelization-

Description: In this program, I have divided the communication world into 4X2 cartesian matrix. Each row has a couple of processes, which perform separate tasks simultaneously. The first row performs 2D FFT on matrix A. At the same time, the second row performs 2D FFt on matrix B. Once these two rows are done with their task, both the matrices are transferred to row 3 of processes, which perform the matrix point mltiplication. There is no improvement in performance by performning the matrix multipliation on separate pair of processes, but it was done so as it was advised in the problem statement.

After the third pair completes the matrix multiplication, the result is sent to pair 4, which performs the inverse 2D FFT and writes the result to file.

Different Versions tried:

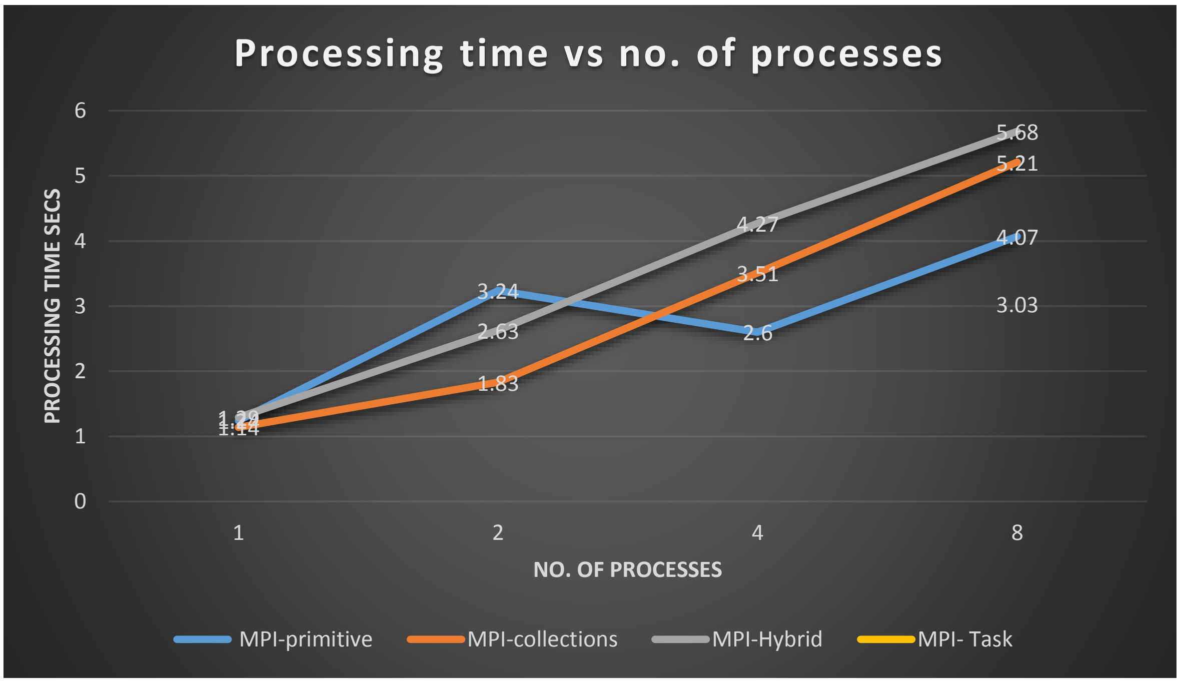
1. The first version used only first two pairs of processes for all calculations. It makes more sense this way as all the tasks can not be performed parallel, and all the time, at least 2 pair of processes are idle.

2. In the second version, the code was updated to use the third and fourth row of the processors, as instructed in problem statement. It reduced the performance due to increased communication cost.

Performance Comparison-

Note: The MPI-Task paralllel strategy was tested only for 8 processes, and hence is visile as a point on graphs.

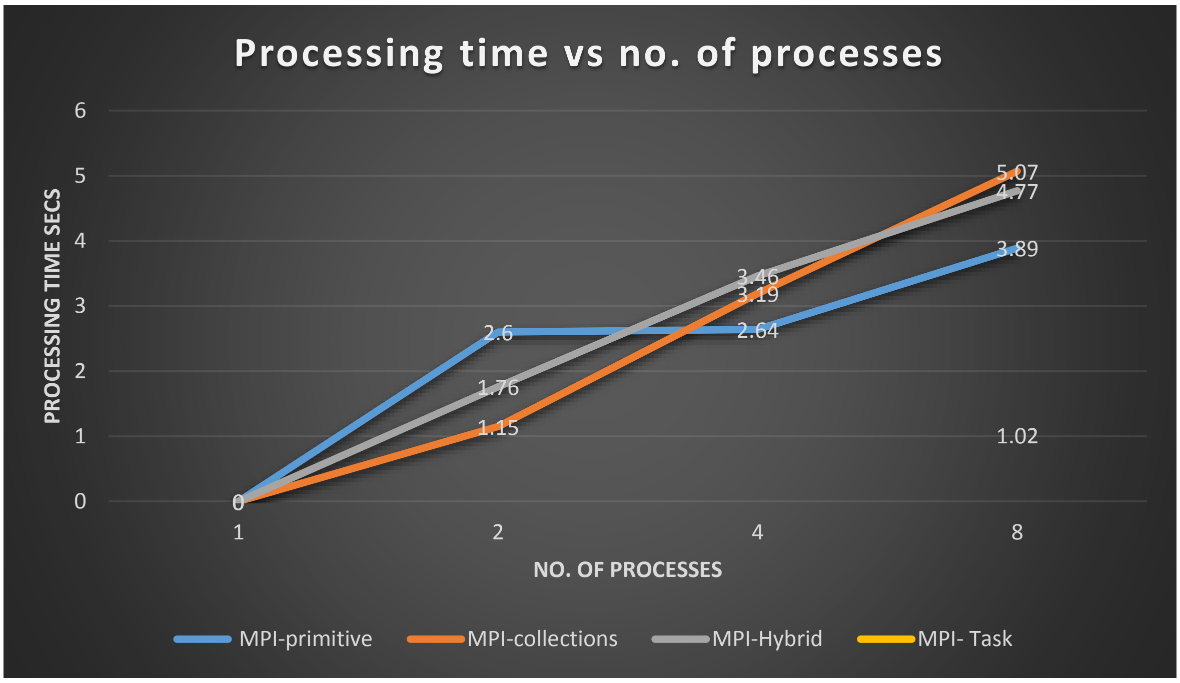
Performance analysis total processing time:



The above graph shows the total processing time taken by the system using various strategies, for different number of processes, for an input size of 512. The total time is increasing instead of decreasing, due to high communication cost as the number of processors increases. I tried but could not reduce the communication cost any further.

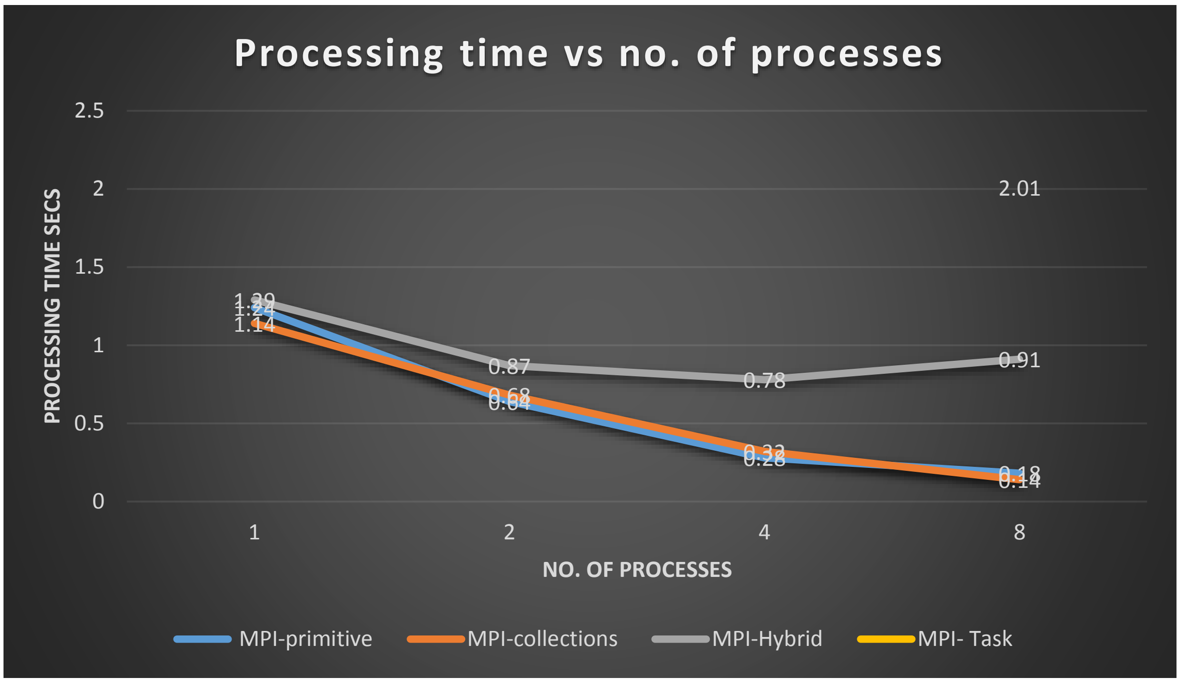
You can see that the total time is very less in the MPI-Task-parallelization approach due to reduced communication cost, because of the design. Most of the communication is only between the two processes which belong to same row, in the Cartesian system, and hence low communication cost. The performance improves and becomes better than the serial program when the input size is more than 2000, but for 512, the serial algorithm gives a better result.

Performance analysis total communication time:



The above graph shows the comparison of the communication time taken by each strategy. You can see here that the 90% of the total processing time (shown in above graph) is due to the communication overhead.

Performance analysis total computing time:



The above graph shows the computation time taken by each strategy, after removing the communication cost. You can see that the computation time reduces linearly as the number of processors increases, and gives a linear speedup.

	1			**2**			**4**			**8**		
	Total	**Comm**	**comp**	**Total**	**Comm**	**Comp**	**Total**	**Comm**	**comp**	**Total**	**Comm**	**comp**
MPI	1.24	0	1.24	3.24	2.6	0.64	2.92	2.64	0.28	4.07	3.89	0.18
MPI-Collective	1.14	0	1.14	1.83	1.15	0.68	3.51	3.19	0.32	5.21	5.07	0.14
MPI-Hybrid	1.29	0	1.29	2.63	1.76	0.87	4.27	3.46	0.78	5.68	4.77	0.91
MPI- task										3.03	1.02	2.01

Time in secs.

Total: total processing time.

Comm: total communication time.

Comp: total computing time.

Conclusion: There is a high communication cost associated with MPI, which drastically reduces the speedup for small input size. The MPI programs work well for very large input size (greater than 2000 in our case), but for small input size, serial program performs better.

<u>Total Time Speedup:</u>

MPI-

Sp = Ts/Tp

For input size N=512,

no. of process =8

Ts = 1.24 s

Tp = 4.07 s

Sp =.31

So it’s a sub linear speedup

MPI collectives-

Sp = Ts/Tp

For input size N=512,

no. of process =8

Ts = 1.14 s

Tp = 5.21 s

Sp =.22

So it’s a sub linear speedup

MPI Hybrid-

Sp = Ts/Tp

For input size N=512,

no. of process =8

Ts = 1.29 s

Tp = 5.68 s

Sp =.22

So it’s a sub linear speedup

MPI task parallel-

Sp = Ts/Tp

For input size N=512,

no. of process =8

Ts = 1.14 s

Tp = 3.03 s

Sp =.38

So it’s a sub linear speedup

<u>computation Time Speedup:</u>

MPI-

Sp = Ts/Tp

For input size N=512,

no. of process =8

Ts = 1.24 s

Tp = .18 s

Sp =6.88

So it’s a sub linear speedup

MPI collectives-

Sp = Ts/Tp

For input size N=512,

no. of process =8

Ts = 1.14 s

Tp = 0.14s

Sp =8.14

So it’s a linear speedup

MPI Hybrid-

Sp = Ts/Tp

For input size N=512,

no. of process =8

Ts = 1.29 s

Tp = .91s

Sp =1.41

So it’s a sub linear speedup

MPI task parallel-

Sp = Ts/Tp

For input size N=512,

no. of process =8

Ts = 1.14 s

Tp = 2.01 s

Sp =.56

So it’s a sub linear speedup