CS633: PARALLEL COMPUTING

ASSIGNMENT 1

HALO EXCHANGE WITH NEIGHBORING PROCESSES

Group Members - Group 28

Name	Roll Number	Email ID
Mehta Shrey Kartik	200580	mehtask20@iitk.ac.in
Priyanka Jalan	190649	prianka@iitk.ac.in
Arpit Kumar Rai	200190	arpitkr20@iitk.ac.in

Problem Statement

Write a program to perform halo exchange (refer to Lectures 12 and 13) with neighbouring processes. The boundary processes need not exchange data in case of non-existing neighbours (for e.g. consider the process grid on the right, process 0 does not have a top and left neighbour, process 1 has three neighbours, and so on).

Assume Px X Py decomposition of processes. Every process owns a subdomain of data (doubles). For example, a sub-domain is shown in the right figure, every process has 16 data points and 4 halo regions (shown in green). The halo regions need to be exchanged with four (or less as the case may be) neighbouring processes (top, bottom, right, left) at every time step.

Every process performs communication for the boundary points/cells followed by a stencil computation. Next time step value at a point/cell P is computed as the average of the four neighbouring points/cells of P (left, right, top, bottom) in case of 5-point stencil. E.g. value (P, t+1) = [value (P_left, t) + value (P_right, t) + value (P_top, t) + value (P_bottom, t) + value (P, t)] / 5. Similarly for 9-point stencil, the average of the eight neighbouring points are used.

The assignment is to compare the performance of data exchange for the given domain sizes and process counts for (i) 5-point stencil and (ii) 9-point stencil using MPI_Pack/MPI_Unpack and MPI_Send/MPI_Recv for communication

Code Explanation

1) The code initializes a 2D array data buffer using the random initialization method provided in the question.

   srand(seed*(myrank+10)); data[i][j] = abs(rand()+(i*rand()+j*myrank))/100;

2) Odd Even Nearest Neighbour Exchange algorithm is employed instead of sequential Nearest Neighbour approach. The corner cases are appropriately handled by checking the existence using ranks of left and right processes.

```
// Intra-Rows NN-2
if(my_rank%2==0 && (my_rank+1)%Px!=0){  // Right send/recv…
}
else if(my_rank%2!=0 && (my_rank)%Px>0){  // Left send/recv…
}

if(my_rank%2!=0 && (my_rank+1)%Px!=0){  // Right send/recv…
}
else if(my_rank%2==0 && (my_rank)%Px!=0){  // Left send/recv…
}
```

```
// Intra-Columns NN-2
int col_rank = my_rank/Px;
if(col_rank%2==0 && col_rank<Py-1){  // Down send/recv…
}
else if(col_rank%2!=0 && col_rank>0){  // Up send/recv…
}
if(col_rank%2!=0 && col_rank<Py-1){  // Down send/recv…
}
else if(col_rank%2==0 && col_rank>0){  // Up send/recv…
}
```

3) Overall the computation of the average can be accounted from 4 different parts Send/Recieves i.e. Up, Down, Left and Right

   Left/Right Data Send and Recieve:

The forwarding of inter-process data to the left and right neighbour requires packing of the rightmost and leftmost columns into a derived structure. For this case, we pack the Columnar data using MPI_Pack() function call, send to the right neighbour using MPI_Send(), receive at the the adjacent neighbour using MPI_Recv() and at last unpack using the MPI_Unpack() function call.

Up/Down Data Send and Recieve

The forwarding of inter-process data to the top and bottom neighbour requires sending of the first row and last row respectively. This data is directly available in the 2D array data buffer and is sent directly using MPI_Send() and MPI_Recv().

4) The denominator for average calculation is appropriately handled using the rank of the process and the data buffer index.

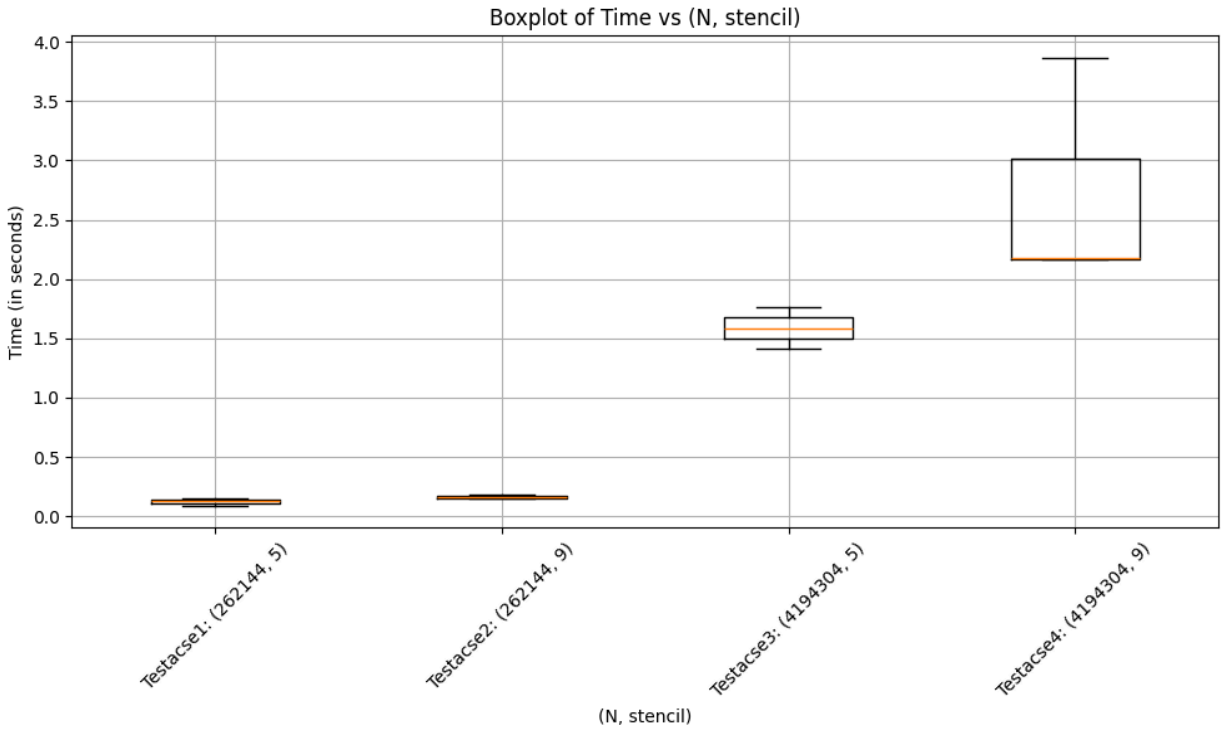
Optimizations

1) **Employing Odd Even Nearest Neigbour (NN-2) Algorithm**: NN-2 gives better performance as compared to NN-1 for large number of processes as the blocking Send and Recieves ultimately lead to sequential completion of the communication.

Timing Plot

The timing plot shows the time taken by the main halo exchange on Prutor, including the computations and communications and excluding the initializations. The x-axis represents the test cases (#datapoints, stencil) on which the program was run and the y-axis denotes the time in seconds.

Test Case 1 (262144, 5)	Test Case 2 (262144, 9)	Test Case 3 (4194304, 5)	Test Case 4 (4194304, 9)
0.090432	0.145697	1.585833	2.161219
0.145618	0.177314	1.410929	2.175656
0.129551	0.165461	1.763972	3.862782



Performance Observation

We analyze the parallel performance by observing the effect of **data size** and **stencil size**.

1. **Data Size** : As the data size per process increases, the time taken by both 5 and 9 stencil increases. This change can be attributed to the increase in the stencil computation time as well as the increase in the communication time between neighboring processes. Another interesting observation is that the change in time for 9 stencil on increasing data size is relatively higher than that for 5 stencil. This change may be because communication time becomes the major time taking part of the program on larger data sizes and 9 stencil performs about twice as many sends and receives as 5 stencil.
2. **Stencil Size** : From the table above, in every execution, the time taken for a given data size by 9 stencil is more than that taken by 5 stencil. This can be easily explained by the fact that 9 stencil involves more computation and communication as compared to 5 stencil.