Indian Institute of Technology, Kanpur
CS633 - Parallel Computing

Assignment 2

by

Yash Phirke (231010090, yashkp23)
Prashant Kumar (22201264, kumarp22)
Vishw Patel (231010087, vishwdipak23)

README

1 Code Explanation

1.1 Without leader

The purpose of this code is to demonstrate parallel communication using MPI for a 2D stencil computation. It uses the **MPI_Pack / MPI_Unpack** and **MPI_Send/MPI_Recv** for sending the halo regions to its neighbouring processors. It calculates the average value of each element in a 2D grid based on its neighboring elements, iteratively over multiple time steps and it prints the **maximum** of time taken by each processes.

Upon initialization, the code initializes MPI, establishing communication channels among different processes. Each process is assigned a rank within the MPI communicator MPI_COMM_WORLD, facilitating coordination and data exchange. Command-line arguments provide essential parameters such as the number of processes in x-direction (Px), the total grid size for each processors (N^2), the number of time steps (time_step), a random seed (seed), and the stencil size (stencil). These parameters define the problem's scope and execution settings.

The local grid size is determined based on the square root of the (N^2) based on that each processors get 2D grid of N x N points. The 2D array used for computation is initialized as $[N + 2 * overlap]^2$. This additional overlap region acts as ghost cells to copy-paste the halo region grid points of the neighbouring processes. The overlap is computed based on formula $overlap = (stencil - 1)/4$ which give us the overlap cells to be added in each direction. Additionally, temporary arrays and buffers are initialized to facilitate this data exchange between neighboring processes. The main 2D array is randomly initialized based on the given formula and the overlap regions are set to 0.0. The code uses boolean to determine the existence of neighboring processes which are true/false based on the the grid topology which is of Px x Py for P processors. Where Px is an input argument and Py = P / Px.

The main computational loop iterates over the specified number of time steps. Within each iteration, the code orchestrates several critical operations.

Firstly using the boolean, code checks for the existence of the processor in left/right/top/bottom. Based on that, it makes a temporary array **(temp_r, temp_l, temp_b, temp_t)** and populates it with the data from the local halo regions which is packed into buffer **(buff_r, buff_l, buff_t, buff_b)** using MPI_Pack . MPI communication functions, such as MPI_Send and MPI_Recv, facilitate this packed data exchange among neighboring processes. This Send/Recv operations are done

based on the Nearest Neighbour (NN) algorithm. The code switches the buffer names while this Send/Recv operations. For example, a processor sends its right buffer [buff_r] to the right processor which will be received and stored into the left buffer [recvbuff_l] of that processor and vice versa is done for left-right and top-bottom communications. The received buffers **(recvbuff_r, recvbuff_l, recvbuff_t, recvbuff_b)** are unpacked using MPI_Unpack library into the temporary recv-arrays **(recvtemp_r, recvtemp_l, recvtemp_b, recvtemp_t)** and copied into the respective overlap regions. This completes the communication part.

Following communications, each process performs the stencil computation on its local grid portion. This computation involves updating each grid point based on the average value of its neighboring points, as defined by the stencil pattern. The computed values are stored in temporary arrays. Then, temporary 2D array values are copied back into our main 2D array. This completes the loop.

The time is measured using **MPI_Wtime()** function, which includes the communication and computation time for the given number of time steps. The maximum execution time across all processes is determined using the **MPI_Reduce()** function, enabling efficient performance analysis.

1.2 With leader

The purpose of this code is to minimize the inter node communications. Host-wise process placement is used to keep processors on different nodes. With $ppn = px = 4$, the first four processes are placed on single node and the next four processes on the next node. This placement techniques make the top and bottom communications between the processors to be inter-node. In order to reduce this number of inter-node communications we have implemented the new algorithm which packs all the data of halo region used only for the top and bottom communications and sends it to the another node which unpacks the data and fill this communicated halo region data into the respective ghost cells.

We achieve this by first making the sub communicators using the **MPI_Comm_split()** of each Px processors which is made using the **color = myrank / Px** logic following the host wise process placement. This allows us to use **MPI_Gather()** in those new sub communicators to gather the halo region data which is to be communicated to the other processes in the subsequent node. After gathering the data in the leader process ($new\_rank = 0$) of each new sub communicator. We use **MPI_Pack()** to pack this top and bottom's gathered data into a buffer.
This packed data is then communicated between the leader processors of each new sub communicators using **MPI_Send()** and **MPI_Recv()** which takes place across nodes. Note, this communications still follows the NN-algorithm for left and right communications which happens the way they used to do for without leader case. Just the number of top and bottom communications are reduced by gathering the data in the new leader rank.
After receiving the packed data of the halo regions the unpacking of this data happens using the **MPI_Unpack**. The unpacked data is scattered in the following processors in the new sub communicator from the leader rank using the **MPI_Scatter**. The top, bottom, left, right boolean are still used to make sure the halo regions are transferred into the correct ghost cells of the each respective processors.
This completes the communication part of the code making sure the number of inter-node communication are reduced. The following computations of averaging using the 9 stencil is done the similar way done for the without leader case.

2 Plots

Observations:

- With increase in data size from 4096^2 to 8192^2 we can see substantial increase in time. This could be mostly due to increase in computation as well as the send/recv buffer size for communication cause of increase in the data size by 4 times which increase the overall time by 4 times.

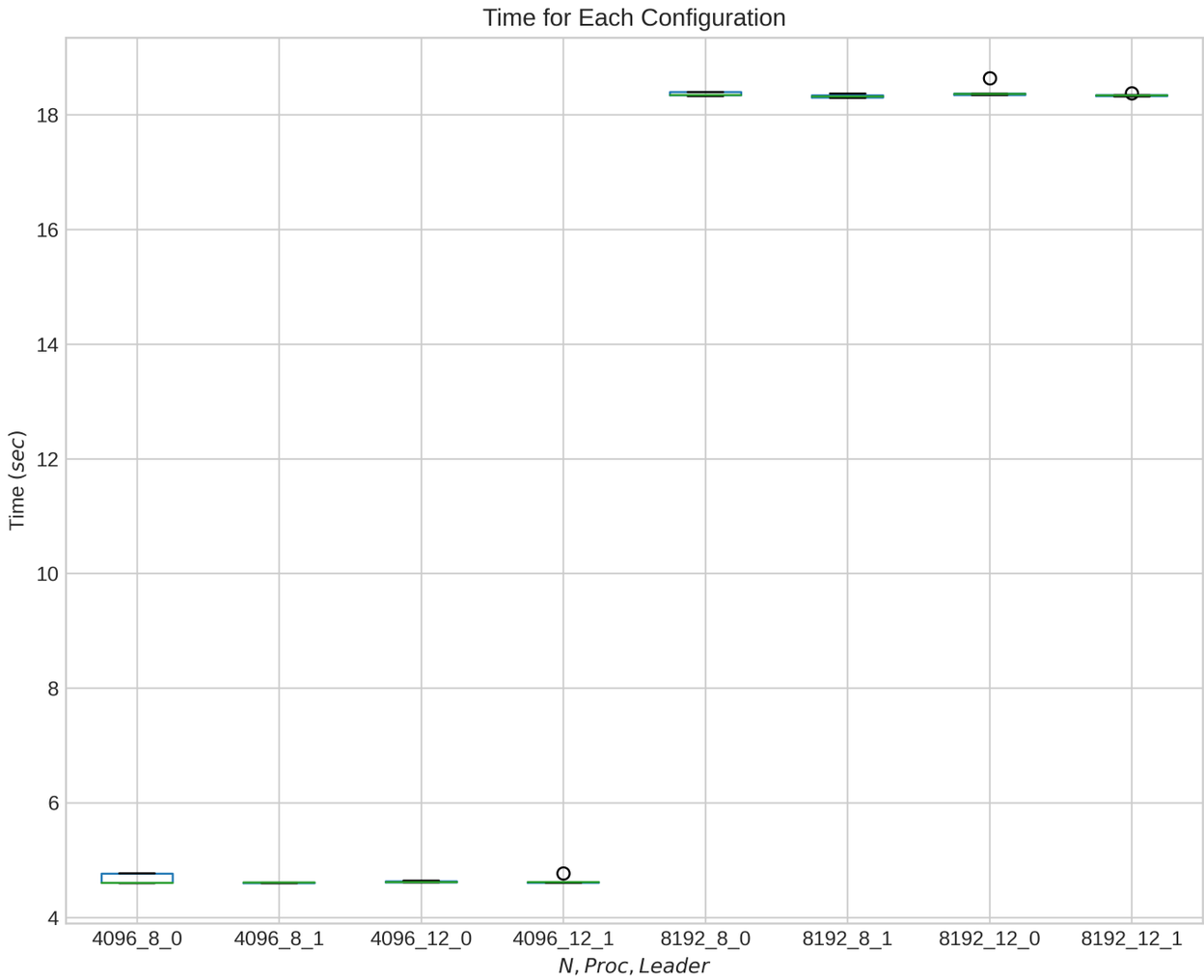

Figure 1: Execution Time vs N, Proc, Leader

- We can see that for the with and without leader cases for each data sizes of 4096 and 8192 we found very negligible decrease in time for all this cases.
- The with leader case has reduced the inter-node congestion in communication between two or more nodes. We didn't see much change in time. This could be due to the fact that overall communication data size still remains the same as only the number of inter-node communications are reduced. We might get to see the improvement for large number of processors.

3 Contribution and Acknowledgement:

P.K. conceptualization, code writing, optimization, debugging, verification, documentation, reviewing, editing and plotting. **Y.K.P.** documentation, reviewing, editing, plotting, conceptualization, code writing, optimization, debugging and verification. **V.D.P** conceptualization, code writing, documentation, reviewing, editing, plotting, optimization, debugging and verification.

We would like to express our sincere gratitude to Dr. Preeti Malakar, who assigned us this project which helps us to understand the concepts of parallel computing using this hands-on assignments.