CS633 Parallel Computing
Assignment-1

Shrey Bhatt, Roll no - 20111060
Saksham Jain, Roll no - 20111053

A brief overview of the problem

In the halo exchange problem, we are given N^2 data points per process and P processes. So, we consider a grid of $N \times N$ data points and $\sqrt{P} * \sqrt{P}$ processes and our task is to perform halo exchange with neighbouring processes. The boundary processes need not exchange data in case of non-existing neighbours. For e.g. consider the process grid with P = 4, process 0 does not have a top and left neighbour, process 1 has two neighbours(left and bottom) and so on. Each process has a subdomain of $N \times N$ data points. The halo regions need to be exchanged with four neighbouring processes(if exists) at every time step and do stencil computation repeatedly over some given timestamps. Next section describes how we have implemented this task in an efficient manner.

Code Execution and Explanation

Code Execution

In order to execute the setup, you need to follow two simple steps.

1. Start and stop the daemon process for Node Allocator.
2. Execute run.py script on terminal using the command python3 run.py

Since the run.py works on the basis of the hostfile generation process of **NodeAllocator**, it is necessary to run the NodeAllocator daemons beforehand. The executed job script i.e. **run.py** removes the already present data files and plot files. Then it compiles the **src.c** file using the make command, guided by the instructions present in **Makefile** to generate object file **halo**. Then it will compile the allocator.c file necessary to further produce the object file allocator.out. After creation of allocator.out, it will be called at beginning of every configuration i.e 5 times to generate a hostfile, namely **hosts**. Also, we have used a simple backup, in case if there is some issue with **NodeAllocator** i.e. one can also use a comparatively simple hostfile generator through the script **gen_hostfile.sh** by uncommenting the two lines mentioned in run.py. However, this script only checks for server liveliness using ping command and then randomly shuffles those servers and output them to hostfile hosts. So, it needs to be used only in case, if NodeAllocator is not working properly as a backup, otherwise no change is required in any files. After the hostfile is generated, the job script will execute the configurations of the halo object file by passing necessary arguments. The executions will produce and append data to the required data files **data_P_N.txt**. Since there are 5 executions, each data file will have 15 entries i.e. timings corresponding to 5 executions and 3 methods for the configuration. After the executions are over, the plot generator script **create_dplots.py** will be executed to accumulate the necessary data and generate required plots **plotP.jpg**. So, the important files generated and the files required for execution are as described above.

Code Explanation and Issues Faced

As mentioned above as well, the crux of the program i.e. the simulation for halo exchange is present in the file **src.c**. The program consists of several functions but below ones perform the important job -

1. **void compute_communicate_mus(double** arr,int t,int myrank,int nump,int subd_n)** - This function contains the logic for the case of multiple sends. Here, each MPI_Isend transmits only 1 element at a time. Initially, for each process the rank of neighbouring process are found using util functions i.e top, right, bottom, left neighbours using top_n(), right_n(), bottom_n(), left_n() functions respectively. Then, the process communicates with the neighbouring processes (if exists) using MPI_Isend and MPI_Irecv which are non-blocking calls for communication. Here, an issue was faced as to what should be the tag for each communication so that the messages are intercepted in the desired buffer position. This was done by forming the tag as $\mathbf{tag1} = \mathbf{t} * \mathbf{N} + \mathbf{index}$,

where N is the row/column length and index is the offset w.r.t row/column origin. This way since between every sender/receiver,N elements are communicated at a timestamp, every consecutive timestamp will have difference of N between it's first tag and every timestamp will have offset from 0 to N-1, thus utilizing the tag values from t*N to t*N + N-1. After receiving the required values, the stencil computation is done using **compute_Stencil_Mus_Derived()** function.

2. **void compute_communicate_pack(double** arr,int t,int myrank,int nump,int subd_n)** - This function does the communication after(before) packing(unpacking) of the data points to be sent(received) using the same communication methods i.e Isend and Irecv. However, since we are using PACKED data, an entire row/column is sent at a single pass. The program gets the rank of neighbouring process using the util function. After that, it utilizes MPI_PACK for packing and traverses along the border of subdomain, packing the necessary elements. After receiving the required values, we are doing a required computation using compute_Stencil_Pack() function, where MPI_Unpack is utilized. The tag values are used for this function by first initializing the values as **tag2 = N*Numts**, which are the tags utilized in the previous module. After that we simply add the current time stamp t, i.e. perform $\mathbf{tag2 + t}$ to form the required tag since at each timestamp there is only one communication (i.e. one send and one receive) between one neighbouring process pair.

3. **void compute_communicate_Derived(double** arr,int tag,int t,int myrank,int nump,int subd_n, MPI_Datatype rowm,MPI_Datatype colm)** - This function uses the derived datatype feature of MPICH library to send and receive the data points. Two datatypes **rowm** and **colm** are created. The Datatype rowm represents a single row of the subdomain matrix and is created using the function **MPI_contiguous**. The datatype colm represents a single column of the subdomain matrix and is created using the function **MPI_vector** by providing beginning value as column origin, blocklength and stride as the column length. The program finds the neighbouring proccess rank using the util function. Like above two cases, we have used MPI_Isend and MPI_Irecv to send and receive data points and used compute_Stencil_Mus_Derived() function for computation of stencil. Similar to PACK, at each timestamp only one send/receive is performed between a single pair of neighbour. So, the tag utilized here is also used by fixing tag as $\mathbf{tag3} = (\mathbf{N} + \mathbf{1}) * \mathbf{Numts} + \mathbf{t}$. However, a typical issue faced was that in order to utilize MPI_vector was that the 2D array needs to be contiguous. Now in order to utilize dynamic and 2D contiguous allocation, it was not possible to first initialize double pointer and then single reference pointers since they need not be contiguous. Hence, another strategy for coniguous allocation was applied which is demonstrated in the code for allocation of array **array3**.

4. **void compute_Stencil_Pack(double** arr,int pr_rank,int nump,int subd_n,double* buf[4])** - This function is used by compute_communicate_pack() to perform stencil computation. For every cell, it is inspected whether the neighbouring cell is part of the subdomain and if not, is there a neighbouring process that can provide it. Also, for over domain boundary cells, the neighbours are considered to be 2/3 depending on their position. The function also utilizes MPI_Unpack to extract values from received buffer.

5. **void compute_Stencil_Mus_Derived(double** arr,int pr_rank,int nump,int subd_n,double* buf[4])** - This function is used by compute_communicate_mus() and compute_communicate_Derieved() to perform stencil computation. Similar to pack counterpart, it also inspects whether the neighbouring cell is part of the subdomain and if not, is there a neighbouring process that can provide it. Also, for over domain boundary cells, the neighbours are considered to be 2/3 depending on their position. However, unlike Unpack, the received buffer values are extracted using a simple indexing based mechanism.

Some other noteworthy util functions are as shown below:

int top_n(int proc_rank,int nump) - This function is used to find top neighbour of a process and return -1 if does not exist.

int bottom_n(int proc_rank,int nump) - This function is used to find bottom neighbour of a process and return -1 if does not exist.

int left_n(int proc_rank,int nump) - This function is used to find left neighbour of a process and return -1 if does not exist.

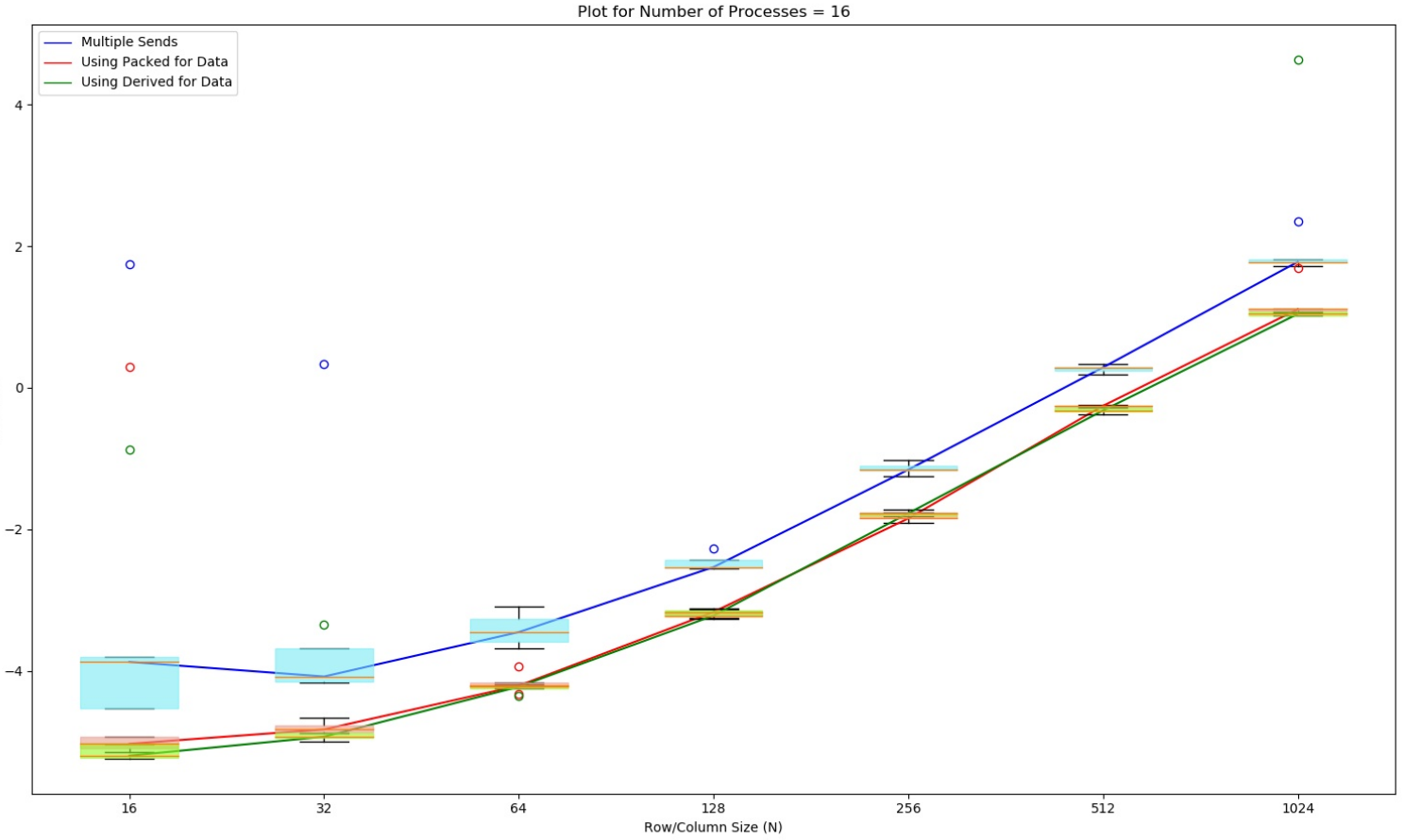
int right_n(int proc_rank,int nump) - This function is used to find right neighbour of a process and return -1 if does not exist.

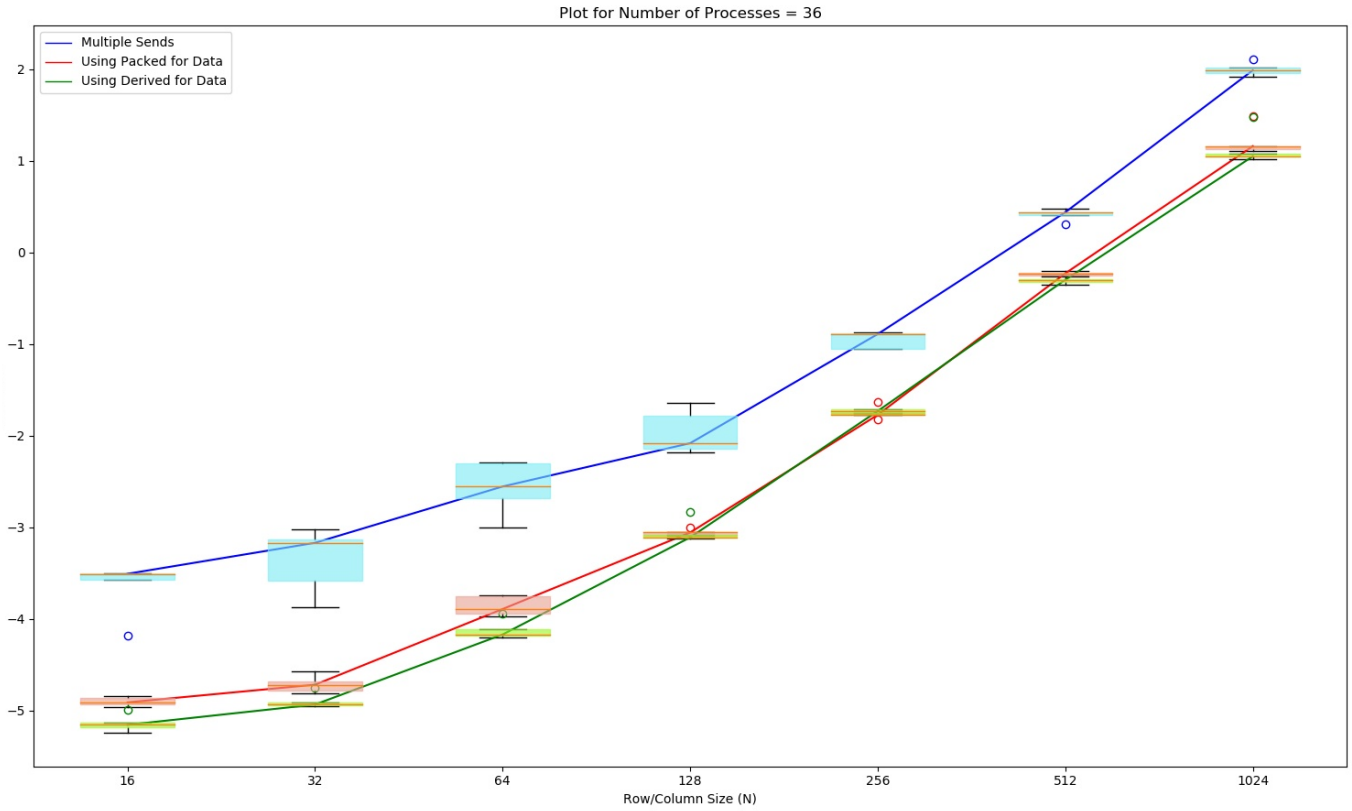
double **copy2Darray(double src,double** targ,int subd_n,int subd_n)** - This function is used to copy 2D array values from src to targ.

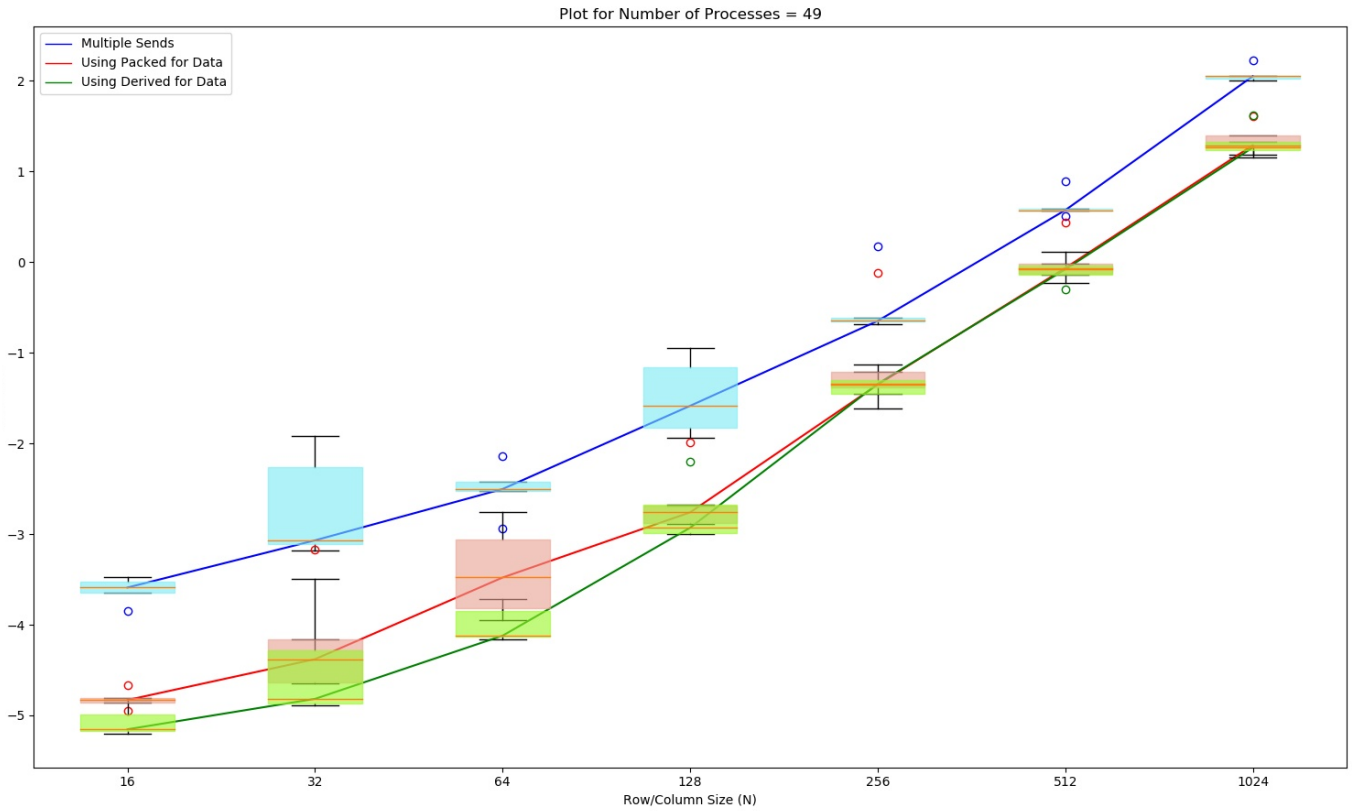
Time taken by each of the above function to run is saved in the data file which is later used to create boxplots.

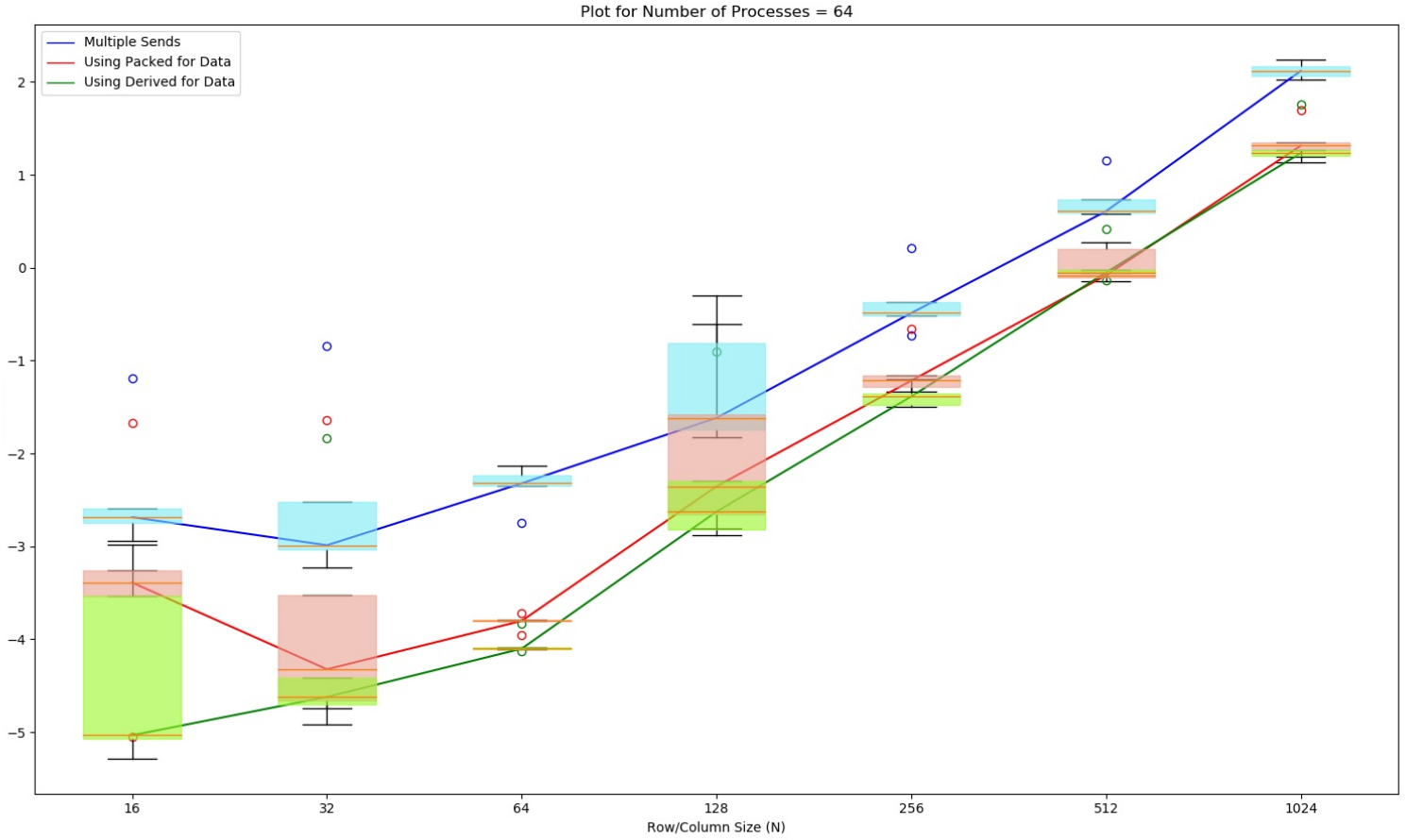
Experimental Setup, Plot Observations and Analysis

The experiment was setup in the csews cluster and through Node Allocator, few nodes from this cluster were assigned to execute the simulation. However, sometimes, the experiments need to be re-executed in case if a connection issue is faced during the on-going execution due to load imbalance of multiple executions conducted simultaneously. The plots obtained after execution are as displayed below.




Plot for Number of Processes = 36
Multiple Sends
Using Packed for Data
Using Derived for Data
2
1
0
−1
−2
−3
−4
−5
16
32
64
128
256
512
1024
Row/Column Size (N)


Plot for Number of Processes = 49
Multiple Sends
Using Packed for Data
Using Derived for Data
2
1
0
−1
−2
−3
−4
−5
16
32
64
128
256
512
1024
Row/Column Size (N)



The plot represents observed data between row length of sub-domain and the natural logarithm over time. As observed evidently and thought intuitively, as the size of data increases for a given number of process, the time taken also increases. Also, another thing that can be observed is that for a fixed size and comparing the time taken by different number of processes also increases slightly. This is because the problem size does not decrease with increasing number of process (every process has N x N sub-domain size). However, as the number of process increases, the the communication overhead increases overall since MPI runtime now has to manage more communications, due to which there is a slight rise observed in time taken. Also, as observed in general, the Multiple send method (in Blue) takes more time than the other methods throughout all executions. This is certainly understandable as Multiple communication means more overhead. Using MPI_PACK (in Red) also for most execution takes slightly more time than Derived Datatype methods (in Green). This can also be understood since PACK requires iterating over boundaries to PACK data which is not the case with Derived datatypes. However, for higher inputs, their execution duration generally overlap i.e. is closely similar as observed in plots. But, the duration for Multiple Sends exhibit significant difference relative to the other two methods.