# CS633: Assignment 2

Group 14


| Chitwan Goel | Aniket Suhas Borkar | Shrey Bansal |
|---|---|---|
| chitwang21@iitk.ac.in | aniketsb21@iitk.ac.in | shreyb21@iitk.ac.in |
| 210295 | 210135 | 210997 |


## 1 Data Structures

### 1.1 Data Arrays

We have used two $2-D$ data arrays `data` and `data_avg` to store the data for each process. At each time step, the processes use `data` array as the current data and store the average values after stencil computation in the array `data_avg`. The arrays are swapped after each iteration so the processes can use the new data.

### 1.2 Buffers

We have used 4 buffer send arrays `sendtop`, `sendbot`, `sendleft` and `sendright`, and 4 buffer receive arrays `recvbot`, `recvtop`, `recvleft`, `recvright` for the data exchange in interprocess communications. We also used 4 additional buffers `sendsubtop`, `sendsubbot`, `recvsubbot` and `recvsubtop` to gather data and scatter it across a sub-communicator when doing topology-aware communication.

### 1.3 Command line arguments

There are various arguments parsed from the command line, which are as follows:

1. `P_x`: The number of columns in the process placement matrix.
2. `N`: The matrix size for each process, i.e., $N = n^2$ where n is the number of rows and columns in the matrix.
3. `timesteps`: The number of times we have to run the stencil.
4. `seed`: Used for random number generation purposes.

### 1.4 Functions

The following functions are used

- `alloc_data()` - This function allocates the various buffers and data arrays mentioned in section 1.1 and 1.2.
- `init_data()` - This function takes the seed and rank as argument and initializes the data in `data` matrix using the `srand()` and `rand()`.
- `free_buf()` - This function frees the various dynamically allocated buffers and matrices.
- `clean_send_buf()` - This function re-initializes the various buffers for sending and receiving data to all zeroes.
- `do_halo_exchange()` - The main function that performs halo exchange and stencil computation. It has a flag `leader` in its arguments, which is used to decide whether to perform topology-aware communication or not.

- `getval()` - Given `i,j` and `data`, it returns `data[i][j]`. If $i$ or $j$ are out of index, it returns the value from the suitable buffer. For example, there is no upper neighbor for an element in the topmost row of the matrix. So when we do `getval(i-1, j)`, it gives the $j^{th}$ element from the data received from the upper process (if it exists, else 0 - as the receive buffer is initialized to 0, and never changed if no data is received).

## 2 Communication

We form sub-communicators when we have passed `leader = 1` argument, based on `color = rank / P_x`. We iterate over the `num_time_steps`. In each iteration of the loop, there are 4 `if-else` blocks, which are explained below:

1. `if (rank` $\geq$ `P_x)` - In this block, we pack the top-most 2 rows to send to the upper process. When not performing topology-aware communication, we have an `MPI_Recv` call after sending, which receives the bottom-most rows from the upper process. This condition ensures that this code is not executed for the processes in the top-most row of the process placement matrix. When performing topology-aware communication, we first gather the data to be sent at the $0^{th}$ process of the sub-communicators. After gathering the data, we have an `MPI_Send` call, which sends the data to the lower row's leader. After a corresponding `MPI_Recv` call to receive the gathered data from the upper row's leader, we scatter the data among the sub-communicator.

2. `if (rank + P_x <` `size)` - In this block, we pack the bottom-most 2 rows to send to the lower process. When not performing topology-aware communication, we have an `MPI_Recv` call, which receives the bottom-most rows from the upper process. After the receive call, we have an `MPI_Send` call, which sends the packed data to the lower process. This condition ensures that this code is not executed for the processes in the bottom-most row of the process placement matrix. When performing topology-aware communication, we first gather the data to be sent at the $0^{th}$ process of the sub-communicators. After gathering the data, we have an `MPI_Recv` call, which receives the data from the lower row's leader. After a corresponding `MPI_Send` call to send the gathered data to the upper row's leader, we scatter the data among the sub-communicator.

3. `if (rank % P_x` $\neq$ `0)` - In this block, we pack the left-most 2 rows to send to the left process. After sending, we have an `MPI_Recv` call, which receives the left-most rows from the right process. This condition ensures that this code is not executed for the processes in the left-most row of the process placement matrix.

4. `if (rank % P_x` $\neq$ `P_x - 1)` - In this block, we pack the right-most 2 rows to send to the right process. We have an `MPI_Recv` call, which receives the right-most rows from the left process. After the receive call, we have an `MPI_Send` call, which sends the packed data to the right process. This condition ensures that this code is not executed for the processes in the bottom-most row of the process placement matrix.

## 3 Computation

1. `count` - We have iterated over the grid points in two loops, one for columns (`j`) and one for rows (`i`). To account for the number of grid points used to compute the average, we have initialized the `count` variable to 9 and subtracted the number of points that cannot be accessed for that particular row and column.

2. `sum` - This stores the sum of values for each grid point `data[i][j]`.

3. `data_avg` - This array is updated by averaging `sum` over `count` i.e., `data_avg[i][j] = sum / count`.

4. `local_time` - This is an array of 2 elements. `local_time[0]` stores the time of topology-unaware stencil computation and `local_time[1]` stores the time of topology-aware stencil computation.

# 4 Results

We took the results by running the code on csews server. The data obtained is tabulated in `data.csv` file. We have used `python` to get the box-plot, the code for which is in `plot.py`.

## 4.1 Observations from the Boxplot

1. As the data size increases, the performance difference between the two cases becomes visible, and the topology-aware communication performs better due to the decreased number of inter-node communications. The inter-node communications are decreased since now only the leader process on every row performs inter-node communication. This is in contrast to unaware communication, where every process is involved in inter-node communication.

2. As the data size increases, the total time taken to execute the code also increases. This is due to increased computation time and communication time (more data is communicated per process).

3. In both the data sizes, the time taken to execute 12 processes is slightly higher.

4. Data size impacts timings more than whether the communication is topology-aware.

**Note:** All three group members contributed equally to every section of the Assignment.

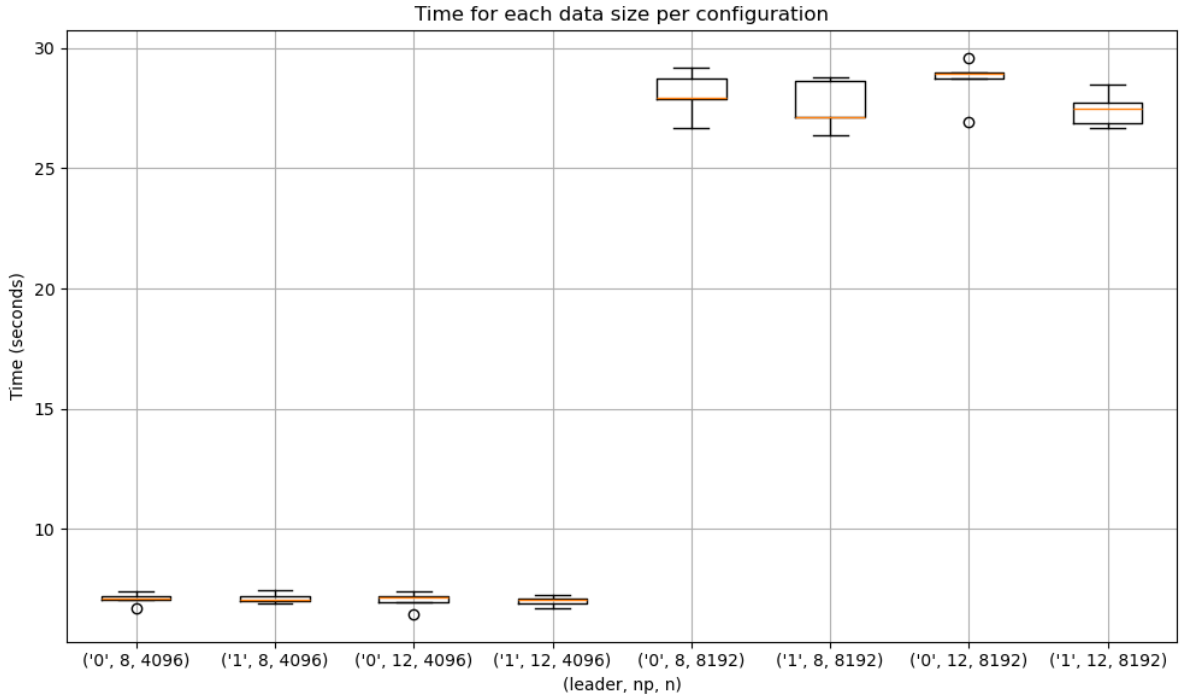


Figure 1: Combined boxplot of all configuations

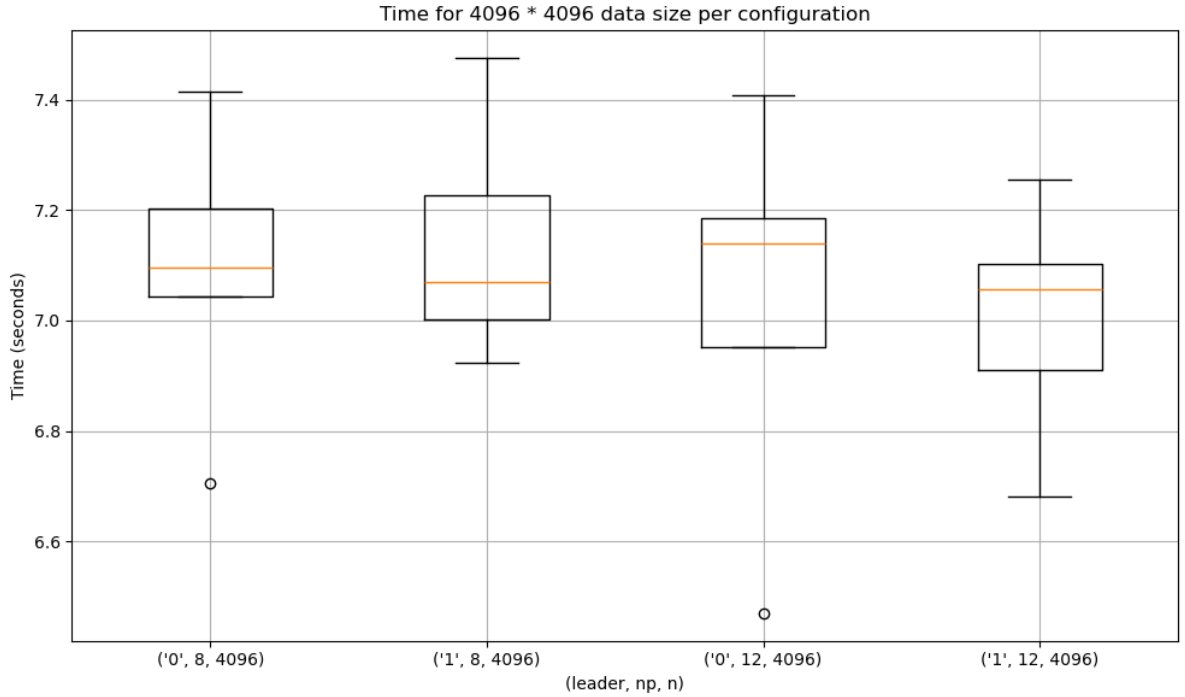


Figure 2: Boxplot of runs with $n = 4096$. Improvement in performance due to topology-aware communication is visible with a higher process count.

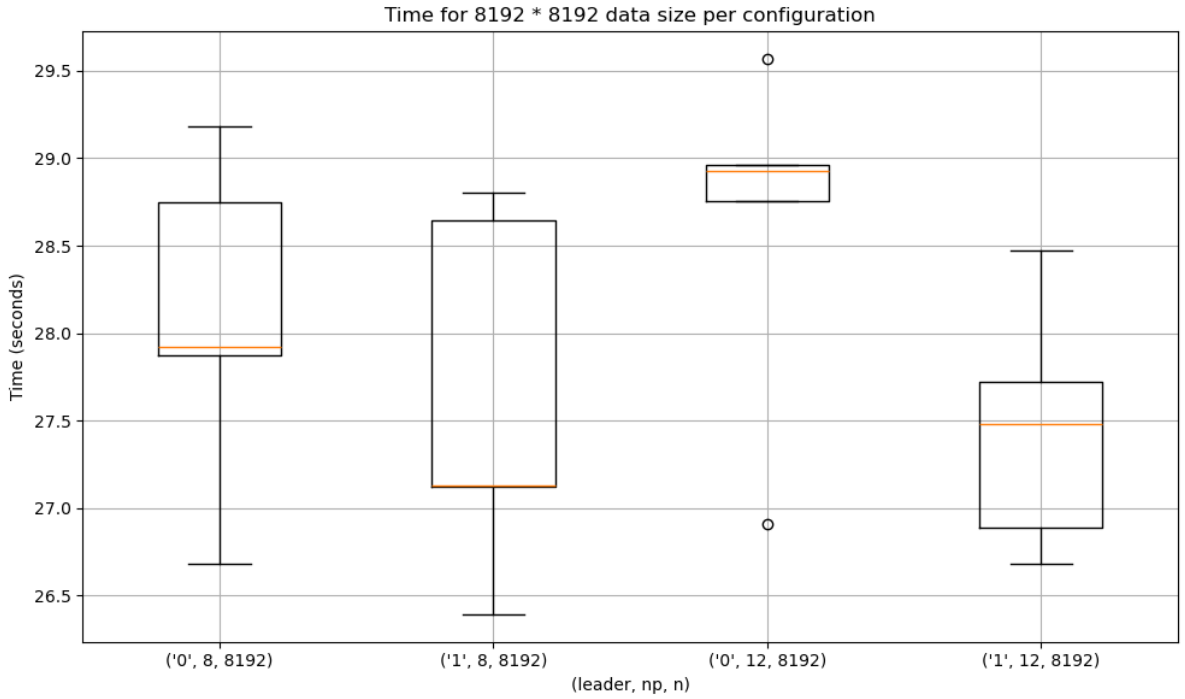


Figure 3: Boxplot of runs with $n = 8192$. Improvement in performance due to topology-aware communication is visible with a higher process count.