CS633: Assignment 2

Group 5
Kajal Deep: 200483
Kumar Arpit: 200532

1 Coding Methodology:

- For n=4096, the readings were taken from prutor. Since prutor was not halting for n=32768, for the readings for those cases the program was evaluated on csews systems by SSHing onto them and using running parameters similar to the ones on prutor's evaluation script (as described below).
- The codes on csews were run using the following line:

  ```
  mpirun -np [px*py] -hosts csews26:4,csews28:4 ./a.x [px] [py] [n]
  ```

  where px, py and n were varied and chosen from [2,2], [2,4] and [32768] respectively (as on prutor's run script). The hosts csews26 and csews28 were chosen since they were under relatively light external load during evaluation period (verified using command htop) and the trend of 4 process per node was again observed on prutor (by printing hostname for various ranks).
- For the 1D case, for each input after printing the time for 2D case of $px \times py$, we also independently time the runs for a corresponding 1D decomposition as $1 \times px * py$.
- Since the value of times amongst various runs were coming extremely close, both on prutor and csews systems, we decided to take 15 readings each instead of 5 to obtain a clearer graph.
- The domain decomposition of the array was done as follows:

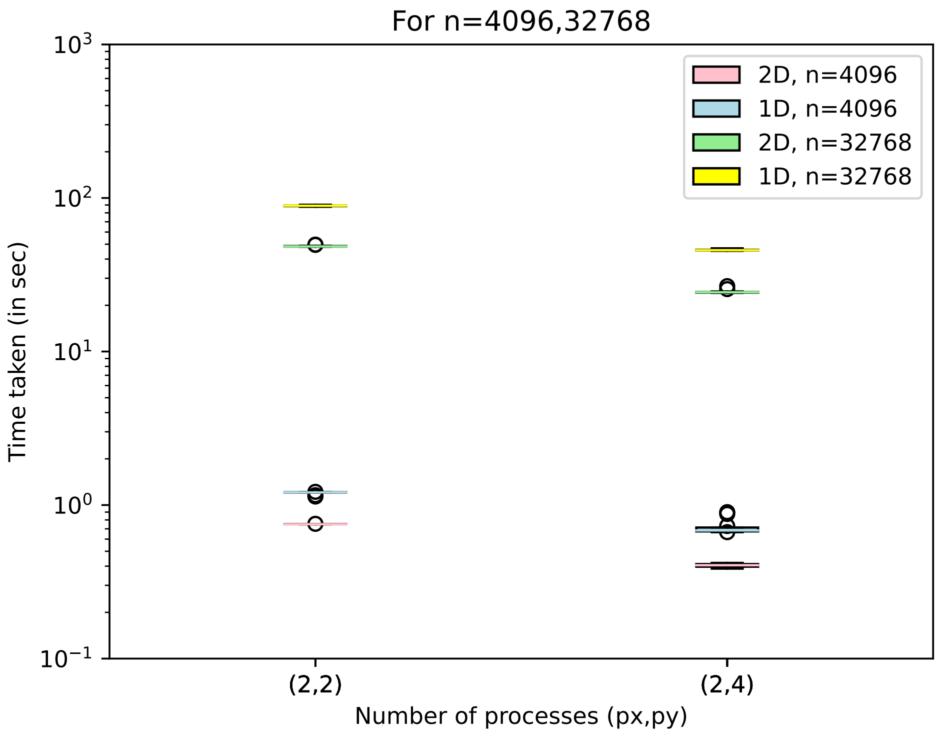
0	1	$\dots$	$px-1$
px	$px+1$	$\dots$	$2px-1$
$\vdots$	$\vdots$	$\ddots$	$\vdots$
$px*py-px$	$px*py+1-px$	$\dots$	$px*py-1$

  (The number in a cell denotes the rank of the process which owns it). Also, each process had full local ownership of its part of the domain.
- fun() function returns the time taken to do the computations and communication for the provided values of px, py and n. The main() functions initialises and finalises MPI calls and prints this time.
- The code is esentially contained in the function fun() which takes in the following parameters:
  - number of processes along X axis, px
  - number of processes along Y axis, py
  - the dimension of the array, n
- Each process initially
  - sets the number of iterations to 20,
  - gets its rank and total size,
  - finds its own position in the grid of processes,
  - initializes its own part of array randomly using rand() function on the heap using malloc(),
  - calculates the rank of process and size of data it is to send to and recieve from,
  - and initialises the receiving buffer.

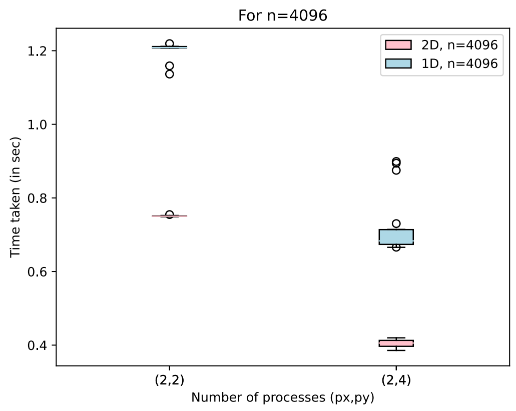
- Then each process starts its timer and goes inside the computation/communication loop.
- The processes in the even rows (as per decomposition) first send their relevant data to the process above it in the grid (if any) and after that receives data from the process below it (if any) while the processes in the odd rows do it in the opposite order. This is done to ensure there is no deadlock and to ensure maximum degree of parallelization at the same time.
- After receiving their respective data, each of the processes proceeds with its local computation and then moves over to the next iteration.
- Since it was not mentioned if all the array parts need to be gathered at a root process, we avoided doing it. After the 20 iterations, we find the end time and reduce it over all the processes which the root process prints in its main() function.

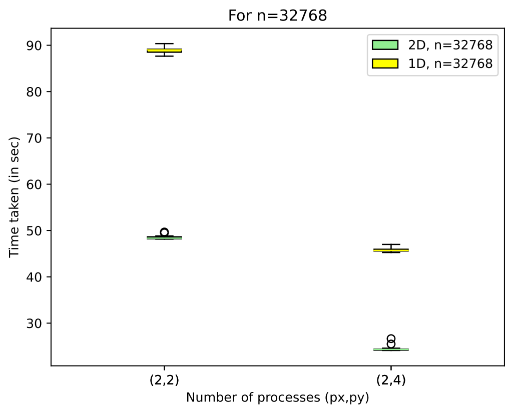
2 Plots:

The following are the observed Time(s) vs Number of processes(px,py) plots.



- We chose to plot the time taken on log scale due to the large stretch of it for the given domain size.
- For a slightly more zoomed in/clearer view of the above graph, we also plotted for the two sizes separately as below:





3 Observations:

- As mentioned above, we have taken 15 observations for each combination of px, py and n instead of 5. The mean of the observed times are:

px	**py**	**n**	**Mean 2D Time(s)**	**Mean 1D Time(s)**
2	4	4096	0.402865	0.723469
2	2	4096	0.750714	1.202131
2	4	32768	24.511907	45.841577
2	2	32768	48.546349	88.957410

- One important thing to keep in mind before looking at the following observations is that for the case of 4 processes, all processes were on the same node and in case of 8 processes, they were split between 2 nodes with 3 intra-node communications and 2 inter-node communications in case of 2D decomposition and 2 intra-node and 1 inter-node communications (albeit of different volumes).

- We notice that the amount of data communication was of the order n while the amount of computation was of the order n^2 for each process. Thus, we expect the communication portion of the ratio to be noticeably smaller compared to computation step especially for the larger array size where the computation dominates time taken.

- We observe the above in our data as well. We can see that going from 4 processes to 8, especially for the case of n=32768, the observed times almost gets halved. This is because the computation step is highly parallelizable as one process requires its own data only. However, in case of n=4096 we see that the times do not exactly get halved and there is a relatively less speedup. This is because the communication step is significant in comparison to computation there which does not get parallelized that well.

- As it goes from 2D decomposition to 1D row wise decomposition, we see that the times increase, almost double (not quite so, it hangs around 1.7-1.8 times for our runs). There are multiple factors involved here. The amount of work distribution between various ranks in much more uneven now. The nodes with lower ranks do significantly lesser computation as computation is done in only over the lower triangular matrix. Moreover, when we look at the communication to computation ration, in either case the computation part of the bottlenecking process is almost same while the amount of communication it needs to do is almost double for the 1D case.

- The variation observed in the time values have been quite low except for a few outliers throughout the 15 runs. In effect, the observed plots have very little deviation from the mean/median making the boxplots look almost like horizontal lines.

- There were other observations not visible here just from the plots. First, the time taken by different ranks during the same run was vastly different. For example, the rank owning the upper left portion of the domain has to do no computation or communication, while one owning the lower right has to compute on its entire domain and communicate n/px amount of data.

4 Code:

```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

long long int min(long long int a, long long int b) // Function to find minimum of two long long int numbers
{
    if (a < b)
        return a;
    else
        return b;
}

long long int max(long long int a, long long int b) // Function to find maximum of two long long int numbers
{
    if (a > b)
        return a;
    else
        return b;
}

double fun(long long int px, long long int py, long long int n)
{
    long long int NUMBER_OF_ITERATIONS = 20; // We set the number of iterations to 20
    MPI_Status status;
    int myrank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    double start_time, time, max_time;
    long long int myrow = myrank / px;                     // myrow is the row number of the process in the 2D grid
    long long int mycol = myrank % px;                     // mycol is the column number of the process in the 2D grid
    long long int realx = (n / px) * mycol;                // realx is the x coordinate of the first element of the process in the 2D grid
    long long int realy = (n / py) * myrow;                // realy is the y coordinate of the first element of the process in the 2D grid
    double **arr = malloc((n / py) * sizeof(double *)); // We allocate memory for the 2D array owned by the process in the following 7 lines
    arr[0] = malloc((n / py) * (n / px) * sizeof(double));
    for (int i = 0; i < (n / py); i++)
        arr[i] = &arr[0][i * (n / px)];
    for (long long int i = 0; i < n / py; i++)
        for (long long int j = 0; j < n / px; j++)
            arr[i][j] = (double)rand() / (double)RAND_MAX;              // We initialize the array with random double values
    long long int amount_to_send = min(max(realy - realx, 0), n / px); // amount_to_send is the number of elements to be sent to the process above
    if (myrow == 0)                                                     // If the process is in the first row, it does not send any elements
        amount_to_send = 0;
    long long int amount_to_recv = min(max(realy + (n / py) - realx, 0), n / px); // amount_to_recv is the number of elements to be received from the process below
    if (myrow == py - 1)                                                           // If the process is in the last row, it does not receive any elements
        amount_to_recv = 0;
    long long int send_to = myrank - px;   // send_to is the rank of the process to which the process sends elements
    long long int recv_from = myrank + px; // recv_from is the rank of the process from which the process receives elements
    double buff[amount_to_recv];           // buff is the buffer used to receive elements
    start_time = MPI_Wtime();              // start_time is the time at which the computation and communication starts
    for (long long int iter = 0; iter < NUMBER_OF_ITERATIONS; iter++)
    {
        if (myrow % 2 == 0) // We alternate the direction of communication, so that the processes do not deadlock. This is done by checking the parity of myrow
        {
            if (amount_to_send > 0 && myrow > 0) // Even rows send first if they have to send
                MPI_Send(&arr[0][0], amount_to_send, MPI_DOUBLE, send_to, iter, MPI_COMM_WORLD);
            if (amount_to_recv > 0 && myrow < py - 1) // Even rows receive second if they have to receive
                MPI_Recv(&buff[0], amount_to_recv, MPI_DOUBLE, recv_from, iter, MPI_COMM_WORLD, &status);
            for (long long int i = 0; i < n / py - 1; i++)
            {
                for (long long int j = 0; j < n / px; j++)
                {
                    if (realx + j <= realy + i)
                        arr[i][j] = arr[i][j] - arr[i + 1][j]; // We perform the computation in top-down fashion
                    else
                        break;
                }
            }
            for (long long int i = 0; i < amount_to_recv; i++) // We subtract the received elements from the last row
                arr[n / py - 1][i] = arr[n / py - 1][i] - buff[i];
        }
        else
        {
            if (amount_to_recv > 0 && myrow < py - 1) // Odd rows receive first if they have to receive
                MPI_Recv(&buff[0], amount_to_recv, MPI_DOUBLE, recv_from, iter, MPI_COMM_WORLD, &status);
            if (amount_to_send > 0 && myrow > 0) // Odd rows send second if they have to send
```

```c
                MPI_Send(&arr[0][0], amount_to_send, MPI_DOUBLE, send_to, iter, MPI_COMM_WORLD);
            for (long long int i = 0; i < n / py - 1; i++) // We perform the computation in similar fashion as above
            {
                for (long long int j = 0; j < n / px; j++)
                {
                    if (realx + j <= realy + i)
                        arr[i][j] = arr[i][j] - arr[i + 1][j];
                    else
                        break;
                }
            }
            for (long long int i = 0; i < amount_to_recv; i++)
                arr[n / py - 1][i] = arr[n / py - 1][i] - buff[i];
        }
    }
    time = MPI_Wtime() - start_time;                                         // time is the time taken for the computation and communication
    MPI_Reduce(&time, &max_time, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD); // We find the maximum time taken by any process
    free(arr[0]);                                                            // We free the memory allocated for the 2D array in the following 2 lines
    free(arr);
    return max_time; // We return the maximum time taken by any process
}

int main(int argc, char *argv[])
{
    double max_time;
    int myrank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    long long int px = atoi(argv[1]), py = atoi(argv[2]), n = atoi(argv[3]);
    max_time = fun(px, py, n); // We call the function with the given values of px, py and n
    if (myrank == 0)           // We print the maximum time taken by any process
        printf("px=%lld py=%lld n=%lld 2D Time=%lf\n", px, py, n, max_time);
    py = px * py;              // We set py to the total number of processes
    px = 1;                    // We set px to 1
    max_time = fun(px, py, n); // We call the function again with px=1, py=total number of processes and the same value of n
    if (myrank == 0)           // We print the maximum time taken by any process
        printf("px=%lld py=%lld n=%lld 1D Time=%lf\n", px, py, n, max_time);
    MPI_Finalize();
    return 0;
}
```