CS633: Parallel Computing
Assignment 2

Group 14: Akshat Sharma (190090), Yash Gupta (190997)

Coding Methodology

1. `code.c:`
   Contains the MPI code to simulate the row subtraction operation mentioned in the assignment. The (0,0) entry of the matrix (A) is assumed to be the upper left entry. Each process initializes and owns its domain/division of the matrix (A). Any data points outside the process's domain that are required to update its domain points are received via P2P communication from the appropriate process. For both one-dimensional row-wise and two-dimensional decomposition, updating the last row of the domain of any process requires the first row of the domain of the process below it. For every process (except the process(es) not having any process below them), this data is received via P2P communication from the process below. We have used non-blocking MPI_Isend() and MPI_Irecv() for the communication.

   A note on performance : For the one-dimensional decomposition, processes do not need to communicate the entire first row in their domain but just the entries which fall inside the lower triangular matrix. Exploiting this reduces the number of P2P communications in the one-dimensional case. In the case of two-dimensional decomposition, to preserve the brevity of the code, we send/receive the entire row of the domain irrespective of whether it lies in the upper/lower triangular matrix. Thus, to ensure a fair comparison between one and two dimensional decomposition we transmit the entire row in both the decompositions. Then, we iterate through all the elements of the domain and update them only if they lie in the lower triangular matrix for both the decompositions.

   We insert an MPI_Barrier between the code segments for 1D/2D decomposition to ensure that the processes are synced before calculating the time for the two-dimensional decomposition.
   The runtimes are printed on the stdout console and manually inserted in the times.txt file.

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

int main(int argc, char *argv[]){
   int myrank;
   double stime, etime, proc_time1, proc_time2, max_time1, max_time2;

   MPI_Init(&argc, &argv);
```

```c
    assert(argc == 4);
    int px = atoi(argv[1]); // number of processes in x
    int py = atoi(argv[2]); // number of processes in y
    long long size = atoll(argv[3]); // total number of elements in a row/column
of the matrix

    int num_iter = 10; // number of iterations

    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    // 1D row-wise domain decomposition
    int num_procs = px * py;
    long long m = size / num_procs; // number of rows of matrix for each process
    double A1[m][size]; // sub-domain of the process

    srand(time(NULL));

    // initializing the matrix
    for(int i = 0; i < m; i++){
        for(int j = 0; j < size; j++){
            A1[i][j] = rand();
        }
    }

    long long sendcount = size;
    long long recvcount = size;
    double buf1[recvcount]; // receive buffer

    MPI_Request send_request1, recv_request1;
    MPI_Status send_status1, recv_status1;

    stime = MPI_Wtime();

    // updating the lower triangular matrix
    for(int t = 0; t < num_iter; t++){
        // sending elements of the first row to the above process
        if(myrank > 0){
            MPI_Isend(A1[0], sendcount, MPI_DOUBLE, myrank - 1, myrank - 1,
MPI_COMM_WORLD, &send_request1);
        }

        // receiving elements of the first row of the below process
        if(myrank < num_procs - 1){
```

```
            MPI_Irecv(buf1, recvcount, MPI_DOUBLE, myrank + 1, myrank, MPI_COMM_WORLD, &recv_request1);
        }

        if(myrank > 0){
            MPI_Wait(&send_request1, &send_status1);
        }

        // updating all rows except last row
        for(long long i = 0; i < m - 1; i++){
            for(long long j = 0; j < size; j++){
                long long elem_row = myrank * m + i; // row of the element in the complete matrix
                long long elem_col = j; // column of the element in the complete matrix

                // updating the element if it is in the lower triangular matrix
                if(elem_row >= elem_col){
                    A1[i][j] -= A1[i + 1][j];
                }
            }
        }

        if(myrank < num_procs - 1){
            MPI_Wait(&recv_request1, &recv_status1);
        }

        // updating the last row
        if(myrank < num_procs - 1){
            for(long long j = 0; j < size; j++){
                long long elem_row = myrank * m + (m - 1); // row of the element in the complete matrix
                long long elem_col = j; // column of the element in the complete matrix

                // updating the element if it is in the lower triangular matrix
                if(elem_row >= elem_col){
                    A1[m - 1][j] -= buf1[j];
                }
            }
        }
    }

    etime = MPI_Wtime();
    proc_time1 = etime - stime;
```

```
    MPI_Reduce(&proc_time1, &max_time1, 1, MPI_DOUBLE, MPI_MAX, 0,
MPI_COMM_WORLD);

    MPI_Barrier(MPI_COMM_WORLD);

    // 2D domain decomposition

    m = size / py; // number of rows of matrix for each process
    long long n = size / px; // number of columns of matrix for each process
    double A2[m][n]; // sub-domain of the process

    // initializing the matrix
    for(int i = 0; i < m; i++){
        for(int j = 0; j < n; j++){
            A2[i][j] = rand();
        }
    }

    double buf2[n]; // receive buffer

    MPI_Request send_request2, recv_request2;
    MPI_Status send_status2, recv_status2;

    int proc_row = myrank / px; // row of the process in the virtual topology
    int proc_col = myrank % px; // column of the process in the virtual topology

    stime = MPI_Wtime();

    // updating the lower triangular matrix
    for(int t = 0; t < num_iter; t++){
        // sending elements of the first row to the above process
        if(proc_row > 0){
            MPI_Isend(A2[0], n, MPI_DOUBLE, myrank - px, myrank - px,
MPI_COMM_WORLD, &send_request2);
        }

        // receiving elements of the first row of the below process
        if(proc_row < py - 1){
            MPI_Irecv(buf2, n, MPI_DOUBLE, myrank + px, myrank, MPI_COMM_WORLD,
&recv_request2);
        }

        if(proc_row > 0){
            MPI_Wait(&send_request2, &send_status2);
```

```
        }

        // updating all rows except last row
        for(long long i = 0; i < m - 1; i++){
            for(long long j = 0; j < n; j++){
                long long elem_row = proc_row * m + i; // row of the element in
the complete matrix
                long long elem_col = proc_col * n + j; // column of the element
in the complete matrix

                // updating the element if it is in the lower triangular matrix
                if(elem_row >= elem_col){
                    A2[i][j] -= A2[i + 1][j];
                }
            }
        }

        if(proc_row < py - 1){
            MPI_Wait(&recv_request2, &recv_status2);
        }

        // updating the last row
        if(proc_row < py - 1){
            for(long long j = 0; j < n; j++){
                long long elem_row = proc_row * m + (m - 1); // row of the
element in the complete matrix
                long long elem_col = proc_col * n + j; // column of the element
in the complete matrix

                // updating the element if it is in the lower triangular matrix
                if(elem_row >= elem_col){
                    A2[m - 1][j] -= buf2[j];
                }
            }
        }
    }

    etime = MPI_Wtime();
    proc_time2 = etime - stime;

    MPI_Reduce(&proc_time2, &max_time2, 1, MPI_DOUBLE, MPI_MAX, 0,
MPI_COMM_WORLD);

    if(!myrank){
```

```
        printf("configuration px = %d, py = %d, size = %lld, runtime 1D = %lf, 
runtime 2D = %lf\n", px, py, size, proc_time1, proc_time2);
    }

    MPI_Finalize();
    return 0;
}
```

2. `run.sh`:
   Compiles and executes the file (code.c) for all four configurations (for px = 2, py = 2,4, size = 4096, 32768) on two hand picked servers csews25 and csews32 (having less load compared to other servers). We have run 4 processes on each node.

```
mpicc ./code.c -o code
rm -f times.txt

for i in $(seq 1 5)
do
   mpirun -np 4 -hosts csews25:4 ./code 2 2 4096
done

for i in $(seq 1 5)
do
   mpirun -np 8 -hosts csews25:4,csews32:4 ./code 2 4 4096
done

for i in $(seq 1 5)
do
   mpirun -np 4 -hosts csews25:4 ./code 2 2 32768
done

for i in $(seq 1 5)
do
   mpirun -np 8 -hosts csews25:4,csews32:4 ./code 2 4 32768
done
```

   Note: the times for size = 4096 are taken by running the code on prutor and those for size = 32768 are taken by running the code on csews.

3. `plot_runtimes.py:`
   Reads the (times.txt) file and creates boxplots for the runtimes. The plots (runtimes.png) and (runtimes_size_4096.png) are generated in the (plots/) directory. They are described in the Plots section.

```
import numpy as np
import matplotlib.pyplot as plt
import re
import os
os.makedirs('./plots/', exist_ok=True)

def add_entry(runtime_dict, px, py, size, time):
   if (px, py, size) not in runtime_dict:
       runtime_dict[(px, py, size)] = []
   runtime_dict[(px, py, size)].append(time)

if __name__ == '__main__':
   with open('./times.txt', 'r') as f:
       content = f.readlines()

   runtimes1D = {}
   runtimes2D = {}
   for line in content:
       result = re.search(r"configuration px = (\d*), py = (\d*), size = (\d*),
runtime 1D = (\d*.?\d*), runtime 2D = (\d*.?\d*)", line)
       px, py, size, runtime1D, runtime2D = [float(i) for i in result.groups()]

       add_entry(runtimes1D, px, py, size, runtime1D)
       add_entry(runtimes2D, px, py, size, runtime2D)

   boxplot_1d_runtimes = []
   boxplot_2d_runtimes = []
   boxplot_small_1d = []
   boxplot_small_2d = []
   for i, (config, val) in enumerate(runtimes1D.items()):
       boxplot_1d_runtimes.append(val)
       if config[-1] == 4096:
           boxplot_small_1d.append(val)    # for just size 4096

   for i, (config, val) in enumerate(runtimes2D.items()):
       boxplot_2d_runtimes.append(val)
       if config[-1] == 4096:
           boxplot_small_2d.append(val)    # for just size 4096

   # plot of all values
```

```
    bp1 = plt.boxplot(boxplot_1d_runtimes, positions=[1,2,1,2],
patch_artist=True, boxprops=dict(facecolor="C0"))
    bp2 = plt.boxplot(boxplot_2d_runtimes, positions=[1,2,1,2],
patch_artist=True, boxprops=dict(facecolor="C2"))

    plt.xticks(ticks = range(3), labels = ["", (2,2), (2,4)], rotation = 30)  #
required to fix label alignment issue
    plt.xlabel("Configurations")
    plt.ylabel("Runtime in seconds")
    plt.legend([bp1["boxes"][0], bp2["boxes"][0]], ['1D', '2D'], bbox_to_anchor
= (1,1))
    plt.title("Configuration Vs Runtime")
    plt.tight_layout()

    plt.savefig('./plots/runtimes.png', dpi=300)
    plt.close()

    # plot of just size=4096 values
    bp1 = plt.boxplot(boxplot_small_1d, positions=[1,2], patch_artist=True,
boxprops=dict(facecolor="C0"))
    bp2 = plt.boxplot(boxplot_small_2d, positions=[1,2], patch_artist=True,
boxprops=dict(facecolor="C2"))

    plt.xticks(ticks = range(3), labels = ["", (2,2), (2,4)], rotation = 30)  #
required to fix label alignment issue
    plt.xlabel("Configurations")
    plt.ylabel("Runtime in seconds")
    plt.legend([bp1["boxes"][0], bp2["boxes"][0]], ['1D', '2D'], bbox_to_anchor
= (1,1))
    plt.title("Configuration Vs Runtime for data size = 4096")
    plt.tight_layout()

    plt.savefig('./plots/runtimes_size_4096.png', dpi=300)
    plt.close()
```

Plots

1. runtimes.png (Fig. 1) : The x-axis denotes the tuples (2, 2) and (2, 4) and the y-axis denotes the process runtime in seconds. The plot contains eight separate boxplots, for px=2, py=2,4, size=4096, 32768, decomposition type = 1D/2D. The blue/green plots indicate the runtimes for the 1D/2D decomposition respectively. The runtimes for size=4096 are relatively small and are not visible when plotting against the size 32768.

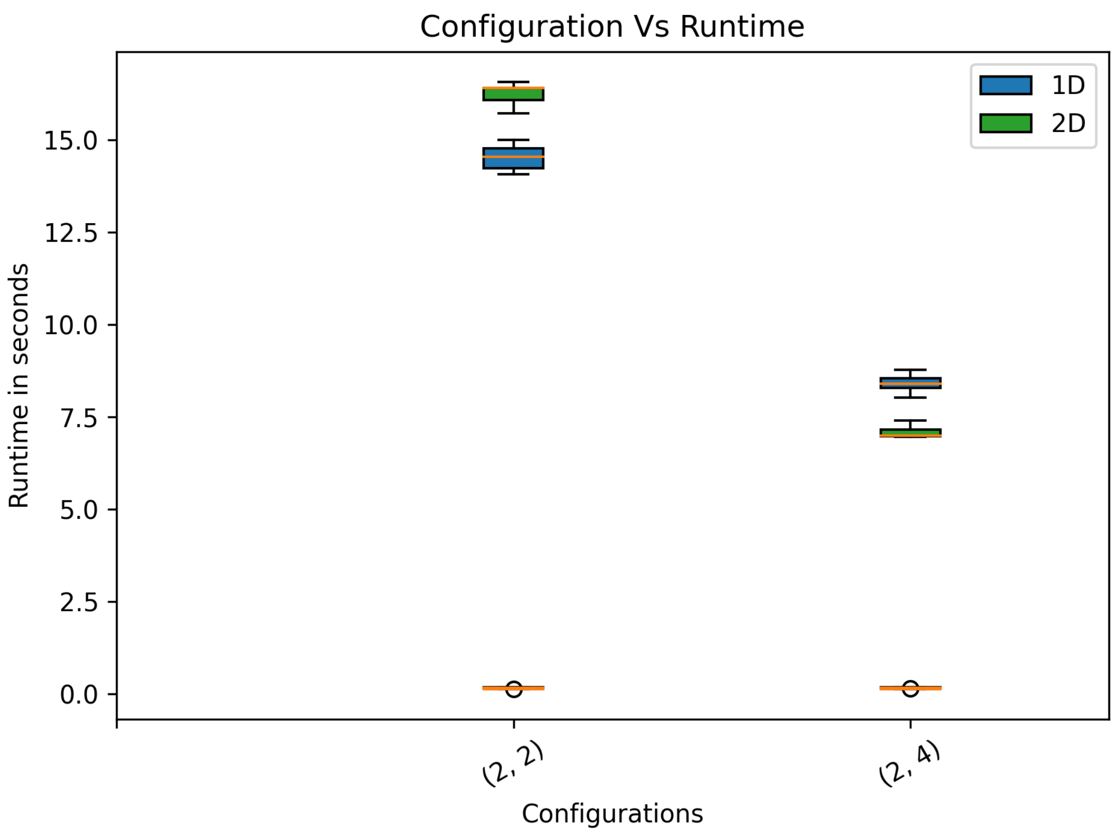

Figure 1

2. runtimes_size_4096.png (Fig. 2) : The plot contains four separate boxplots for just the size=4096.

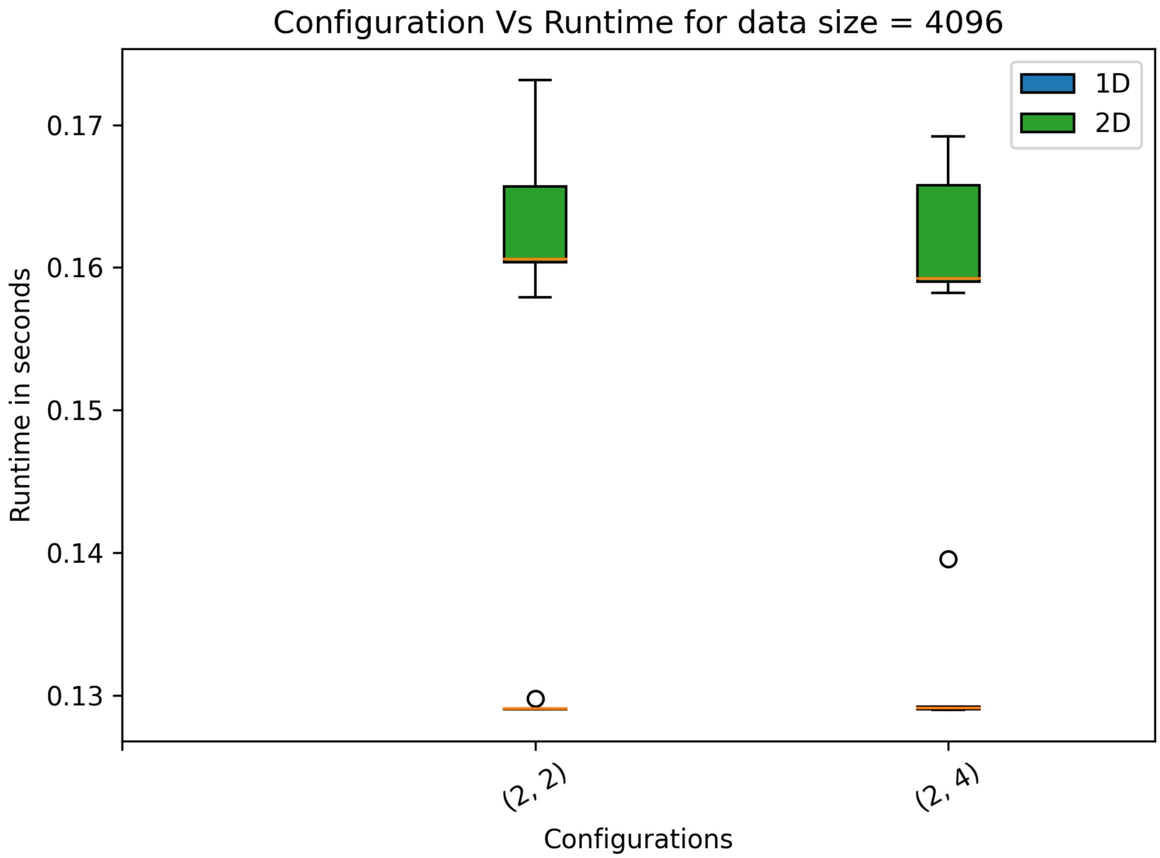
Configuration Vs Runtime for data size = 4096
1D
2D
0.17
0.16
0.15
0.14
0.13
Runtime in seconds
(2, 2)
(2, 4)
Configurations

Figure 2

Observations :

1. As the size increases from 4096 to 32768, the runtimes also increase for every configuration.
2. Note that the majority of the runtime can be divided into P2P communication between processes and the time required by a process to update its domain. Ignoring the number of processes and other constant factors the time for communication is $O(size\ of\ row)$ and the time for computation is $O(size\ of\ row\ *\ number\ of\ rows\ in\ each\ domain)$ which scales as $O(size\ of\ row^2)$ for a fixed number of processes. Therefore, for larger data sizes the computation part requires a majority of the time. <u>Hence, increasing the number of processes for larger data sizes is most beneficial</u>. This is apparent in the plots, where in Figure 2 moving from (px,py)=(2, 2) to (px,py)=(2, 4) gives little improvement as the data size is small whereas the same gives a significant performance boost for size=32768, as shown in Figure 1.

Appendix

1. Runtime data : Five runs per configuration

S. No.	px	py	size	1D time	2D time
1	2	2	4096	0.129069	0.173135
2	2	2	4096	0.129740	0.160600
3	2	2	4096	0.129048	0.157928
4	2	2	4096	0.128998	0.165666
5	2	2	4096	0.129006	0.160377
6	2	4	4096	0.129107	0.165751
7	2	4	4096	0.128984	0.159018
8	2	4	4096	0.129015	0.159216
9	2	4	4096	0.139565	0.169190
10	2	4	4096	0.129188	0.158215
11	2	2	32768	14.533213	16.401319
12	2	2	32768	14.991373	15.713689
13	2	2	32768	14.061208	16.075809
14	2	2	32768	14.223362	16.408631
15	2	2	32768	14.762262	16.561868
16	2	4	32768	8.019462	6.953642
17	2	4	32768	8.536447	6.967559
18	2	4	32768	8.387153	7.397684
19	2	4	32768	8.287314	6.987819
20	2	4	32768	8.765471	7.157647