Assignment 3

Student: Ubaid ur Rehman
Lecturer: Sir Ali Sayyad

This assignment is about PDC codes in which Open MPI is configured.

Task 1

- In MPI, if a non-root process changes the value before the broadcast, ***nothing happens to the broadcast***. The broadcast only sends the value from the root process, and that value overwrites the local values on all other processes, including any changes they made before the broadcast.
- Constraints must need before MPI_Bcast():
  1. Correctly initialize the buffers before sending or receiving data between them.
  2. All processes must be belongs to same group MPI_COMM_WORLD
  3. Data must be of same datatype.
- I think that's are some constraints that we should follow before using Broadcast.
- Broadcast is differ from Send/Recv is this sense:
  1. First, you don't need to worry of handling (send/recv) of data case of loops and all that. It efficiently handles all the process we do manually while using send()/recv() functions.
  2. You simply need to give correct buffers to the function and it handle all the headache, we faced it send and recv of send from a buffer and storing value into a buffer, on its own(user-friendly).
  3. Second send/recv function only give data to the specific or desired process which we want. Instead broadcast will spread data among all the processes.

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

int main(int argc, char** argv){    // passing arguments while exection
    MPI_Init(NULL,NULL);            // initialize MPI environment

    int rank,data;                  // neccessary variables

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);   // comm_rank to give each process unique rank

    if(rank == 0){                  // Master process initialize data variable
        data = 10;
    }
    MPI_Bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);    // Broadcasting of data to other processes
    printf("value on %d: after recieved: %d\n", rank, data); // Display the results over all processes

    MPI_Finalize();                 // Close the MPI environment before exit
    return 0;
}
```

Figure 1: Basic Understanding of Broadcast function.

```
(base) dr-pc@dr-pc-HP-EliteBook-840-G6:~/Desktop$ mpicc task_1.c -o t1
(base) dr-pc@dr-pc-HP-EliteBook-840-G6:~/Desktop$ mpirun -np 4 ./t1
value on 2: after recieved: 10
value on 3: after recieved: 10
value on 0: after recieved: 10
value on 1: after recieved: 10
(base) dr-pc@dr-pc-HP-EliteBook-840-G6:~/Desktop$
```

Figure 2: Output of above code.

Task 2

- Some several issues will faced not systematically but logically:

  1. What I mean is that, the array calculations and all things works fine but when the divisible is odd. It will only pick up the integer size value to scatter them on processors. **Example:** I want to run 6 processes and my data size is 16, so $16/6 = 2.67$. It disturbs array on each processes 2 2 index and left the remaining ones once each process get its job.

- We can easily do same code by a little change of using Scatterv() instead of Scatter() and Gatherv() instead of Gather(). MPI automatically handles the uneven the distribution of processors.

```c
#include<stdio.h>
#include<mpi.h>
int main(int argc, char** argv){

    MPI_Init(&argc,&argv);                  // Initialize MPI
    int rank,array_size = 16,size;          // Neccessary Variables

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);   // rank and size(no of processors)
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int snd_buffer[array_size],rcv_buffer[array_size/size]; // buffers (data storage)
    int chunk_size = array_size / size;     // 4

    if(rank == 0){
        // Initialization of array from Master
        for(int i = 0; i < array_size; i++){
            snd_buffer[i] = i+1;
        }
    }

    // Scatter the data among processors and
    MPI_Scatter(snd_buffer, chunk_size, MPI_INT ,rcv_buffer, chunk_size, MPI_INT, 0, MPI_COMM_WORLD);
    for (int i = 0; i < chunk_size; i++)
    {
        // Multipltication of values by 2
        rcv_buffer[i] *= 2;
    }

    // Gather all values and store back into snd_buffer
    MPI_Gather(rcv_buffer, chunk_size, MPI_INT, snd_buffer, chunk_size, MPI_INT, 0, MPI_COMM_WORLD);

    // Display values using Master
    if (rank == 0)
    {
        for (int i = 0; i < array_size; i++)
        {
            printf("%d ",snd_buffer[i]);
        }
        printf("\n");
    }

    MPI_Finalize();
    return 0;
}
```

Figure 3: Data Scattering and Gathering

```
(base) dr-pc@dr-pc-HP-EliteBook-840-G6:~/Desktop$ mpicc task_2.c -o t2
(base) dr-pc@dr-pc-HP-EliteBook-840-G6:~/Desktop$ mpirun -np 4 ./t2
2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32
(base) dr-pc@dr-pc-HP-EliteBook-840-G6:~/Desktop$
```

Figure 4: Output of above code.

Task 3

- MPI_Allgather() is expensive because we can say, it performs two operations:
  1. First gather all the data from the processes.
  2. Second Broadcast that data among the processes.

  This is the main reason of getting expensive with comparison of Gather() function.

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

int main(int argc, char** argv){
    MPI_Init(&argc,&argv);  // Initialize MPI

    int rank,size;          // Neccessary Variables

    MPI_Comm_rank(MPI_COMM_WORLD,&rank);    // rank and size declaration
    MPI_Comm_size(MPI_COMM_WORLD,&size);

    int buffer[size];           // data storage area
    srand(time(NULL)+rank);     // for taking each time different random value
    int value = rand() % 101;   // random value from 1-100
    int temp;

    MPI_Allgather(&value, 1, MPI_INT, buffer, 1, MPI_INT, MPI_COMM_WORLD); // Gather to collect all values from the processes

    MPI_Reduce(&value, &temp, 1, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD);// Collect and apply max function on those values

    // Just for displaying values by Master
    if (rank == 0)
    {
        for (int i = 0; i < size; i++)
        {
            printf("%d ",buffer[i]);
        }
        printf("\n");
        printf("Maximum value: %d \n",temp);
    }

    // For End-up MPI
    MPI_Finalize();
    return 0;
}
```

Figure 5: Distributed Reduction and All-Gather

```
(base) dr-pc@dr-pc-HP-EliteBook-840-G6:~/Desktop$ mpicc task_3.c -o t3
(base) dr-pc@dr-pc-HP-EliteBook-840-G6:~/Desktop$ mpirun --oversubscribe -np 6 ./t3
78 43 22 95 48 83
Maximum value: 95
(base) dr-pc@dr-pc-HP-EliteBook-840-G6:~/Desktop$
```

Figure 6: Output of above code.