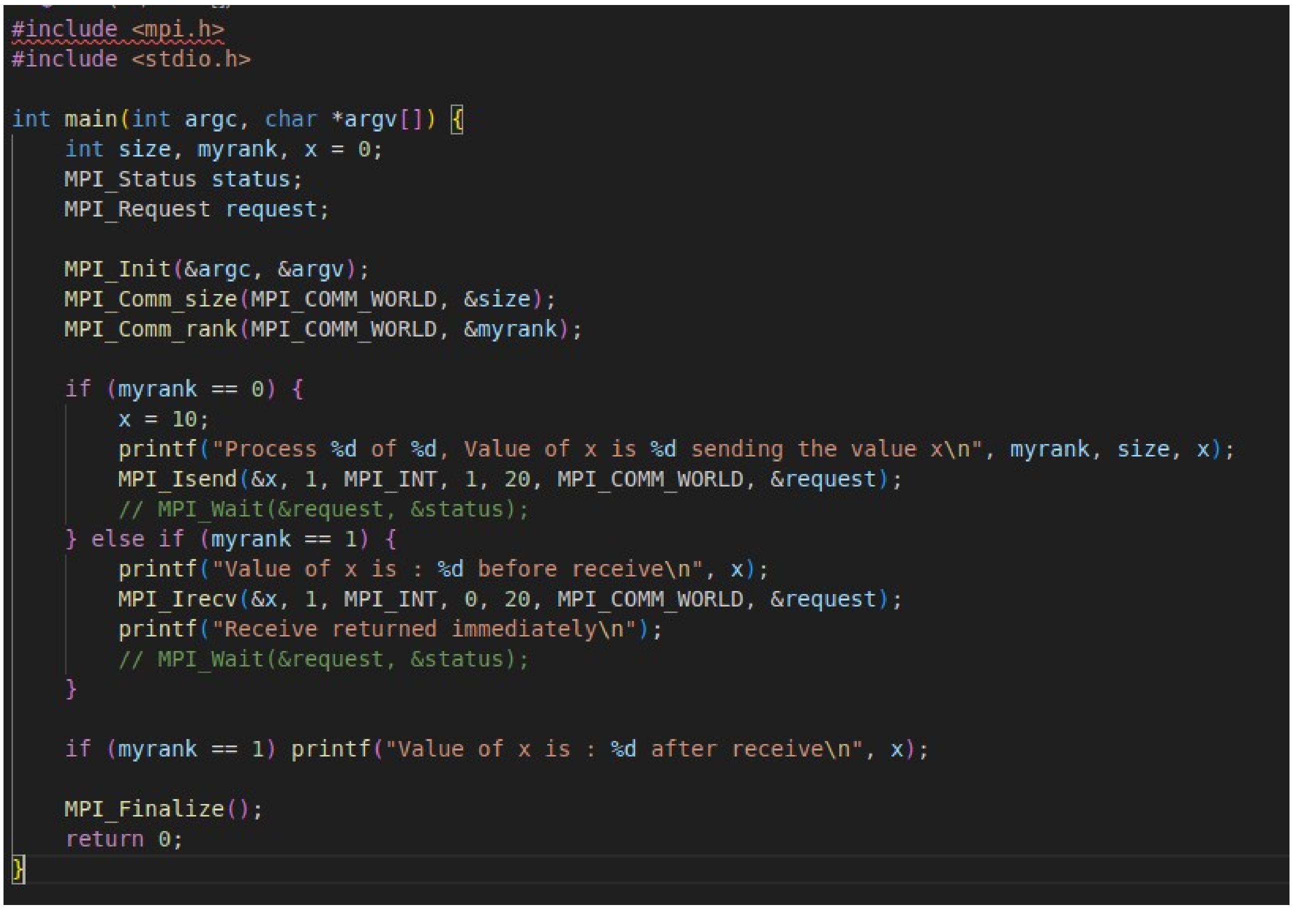
Nithin S
221IT085

IT301 Lab Assignment 7

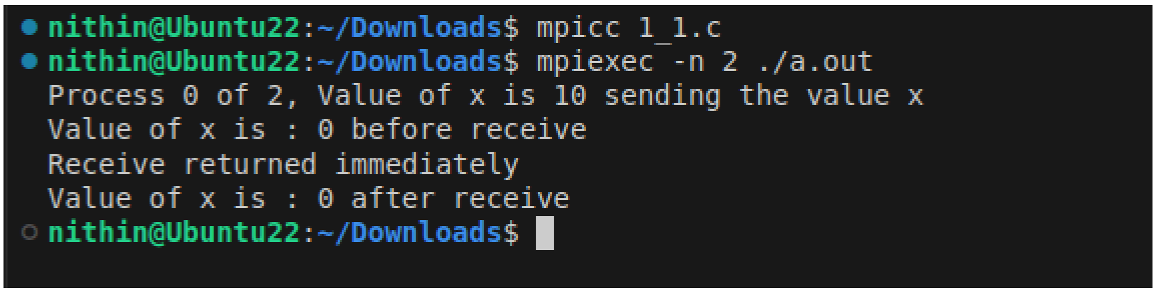
Q1. Program 1. MPI non-blocking Send and Receive (). Record the observation with and without MPI_Wait ()
a) Note down results by commenting MPI_Wait()

Code



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

int main(int argc, char *argv[]) {
    int size, myrank, x = 0;
    MPI_Status status;
    MPI_Request request;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    if (myrank == 0) {
        x = 10;
        printf("Process %d of %d, Value of x is %d sending the value x\n", myrank, size, x);
        MPI_Isend(&x, 1, MPI_INT, 1, 20, MPI_COMM_WORLD, &request);
        // MPI_Wait(&request, &status);
    } else if (myrank == 1) {
        printf("Value of x is : %d before receive\n", x);
        MPI_Irecv(&x, 1, MPI_INT, 0, 20, MPI_COMM_WORLD, &request);
        printf("Receive returned immediately\n");
        // MPI_Wait(&request, &status);
    }

    if (myrank == 1) printf("Value of x is : %d after receive\n", x);

    MPI_Finalize();
    return 0;
}
```

Output



```
nithin@Ubuntu22:~/Downloads$ mpicc 1_1.c
nithin@Ubuntu22:~/Downloads$ mpiexec -n 2 ./a.out
Process 0 of 2, Value of x is 10 sending the value x
Value of x is : 0 before receive
Receive returned immediately
Value of x is : 0 after receive
nithin@Ubuntu22:~/Downloads$ 
```

Analysis

Without `MPI_Wait`, the following happens in the code:

1. **Non-blocking Send and Receive**:
   - `MPI_Isend` and `MPI_Irecv` initiate communication but return immediately, without waiting for the data transfer to complete.

2. **Process 0 Behavior**:
   - **Process 0** sets `x = 10` and calls `MPI_Isend`. Since `MPI_Wait` is not used, the program does not wait for the send operation to complete and continues execution immediately after initiating the send.

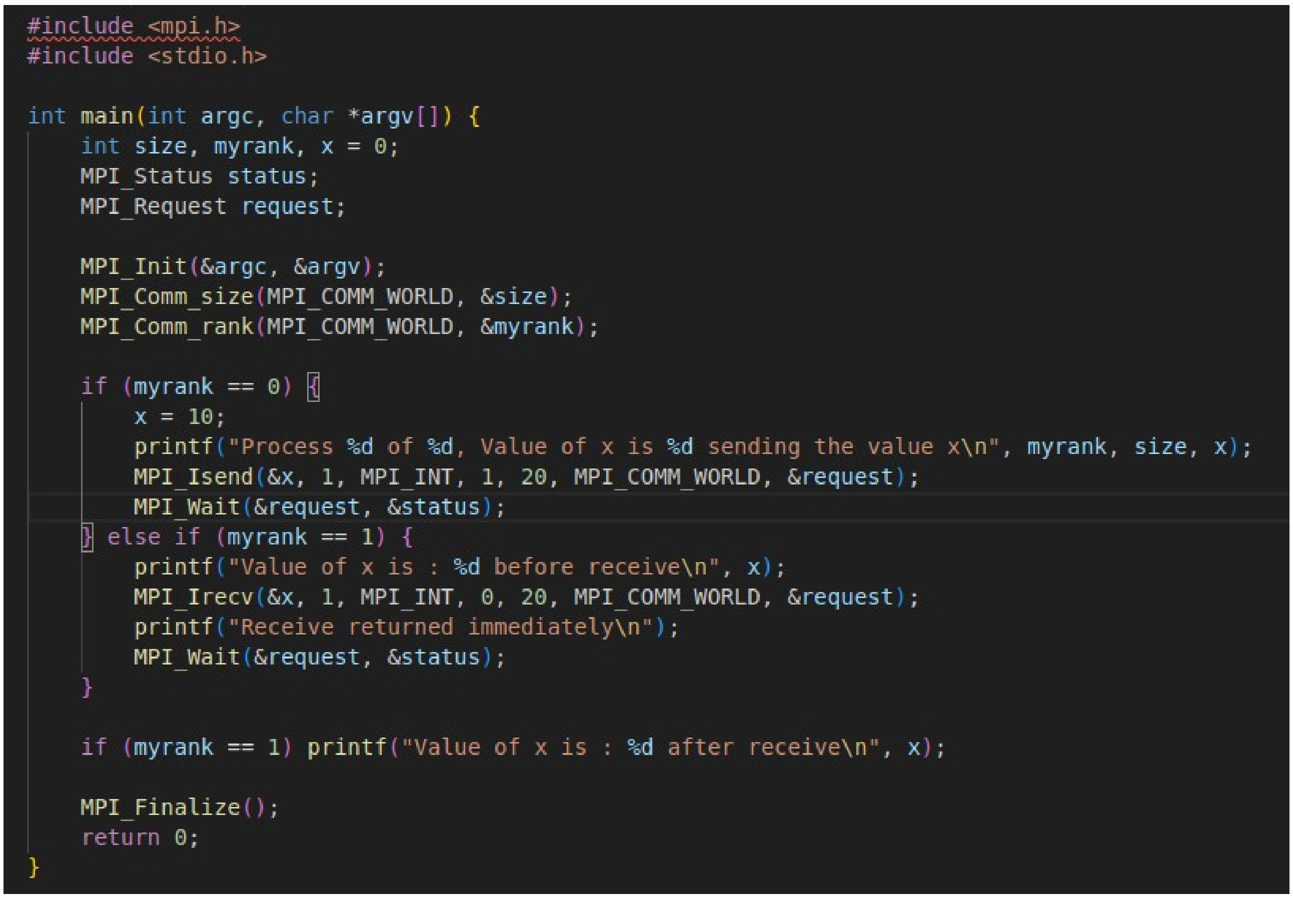
3. **Process 1 Behavior**:
   - **Process 1** prints the value of `x` (initialized to `0`) before calling `MPI_Irecv`. After calling `MPI_Irecv`, the receive operation is initiated but doesn't block, so the program moves on and prints the value of `x` again without waiting for the communication to complete.

4. **Result**:
   - On **Process 1**, the value of `x` remains `0` both before and after the `MPI_Irecv` call because the data transfer has not been completed by the time `x` is printed. The receive operation was initiated, but since no `MPI_Wait` is called, there's no guarantee that the message has actually been received.

This demonstrates that the non-blocking receive does not immediately update `x` without synchronization.

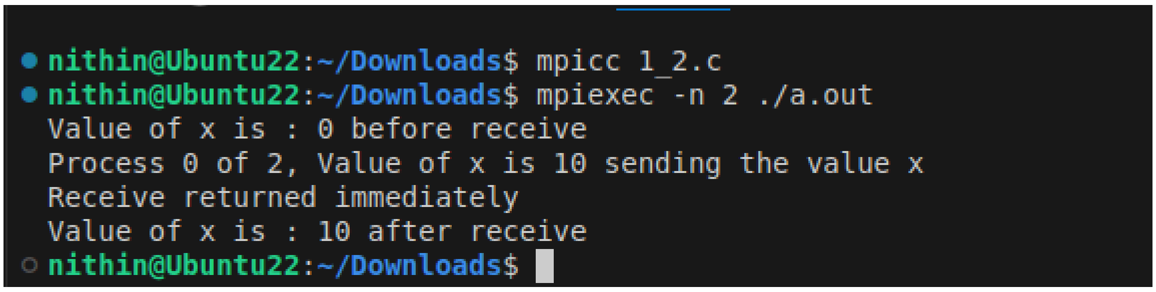
b) Note down result by uncommenting MPI_Wait()

Code



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

int main(int argc, char *argv[]) {
    int size, myrank, x = 0;
    MPI_Status status;
    MPI_Request request;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    if (myrank == 0) {
        x = 10;
        printf("Process %d of %d, Value of x is %d sending the value x\n", myrank, size, x);
        MPI_Isend(&x, 1, MPI_INT, 1, 20, MPI_COMM_WORLD, &request);
        MPI_Wait(&request, &status);
    } else if (myrank == 1) {
        printf("Value of x is : %d before receive\n", x);
        MPI_Irecv(&x, 1, MPI_INT, 0, 20, MPI_COMM_WORLD, &request);
        printf("Receive returned immediately\n");
        MPI_Wait(&request, &status);
    }

    if (myrank == 1) printf("Value of x is : %d after receive\n", x);

    MPI_Finalize();
    return 0;
}
```

Output



```
nithin@Ubuntu22:~/Downloads$ mpicc 1_2.c
nithin@Ubuntu22:~/Downloads$ mpiexec -n 2 ./a.out
Value of x is : 0 before receive
Process 0 of 2, Value of x is 10 sending the value x
Receive returned immediately
Value of x is : 10 after receive
nithin@Ubuntu22:~/Downloads$
```

Analysis

In this MPI program, the communication between Process 0 and Process 1 is managed using non-blocking operations (`MPI_Isend` and `MPI_Irecv`), but unlike the previous case, the communication is synchronized with `MPI_Wait`. Here's what happens:

1. **Non-blocking Communication with `MPI_Wait`**:
   - **Process 0** sends the value of `x = 10` using `MPI_Isend` and then waits for the operation to complete with `MPI_Wait`.
   - **Process 1** initiates a non-blocking receive (`MPI_Irecv`) and immediately proceeds with the next statement but uses `MPI_Wait` to ensure the receive operation is completed before accessing `x`.

2. **Correct Data Transfer**:
   - Due to the presence of `MPI_Wait`, both the send and receive operations are guaranteed to complete before the program proceeds further.
   - This ensures that by the time **Process 1** prints the value of `x` after the receive, it has correctly received the value sent by **Process 0**.

3. **Execution Flow**:
   - **Process 0** sets `x = 10`, prints the value, and sends it to **Process 1**. It waits for the send operation to complete.
   - **Process 1** initially prints `x = 0` (its default value), initiates the receive, and immediately returns from `MPI_Irecv`. It waits for the receive operation to complete, and after that, `x` is updated to `10`, as sent by **Process 0**.

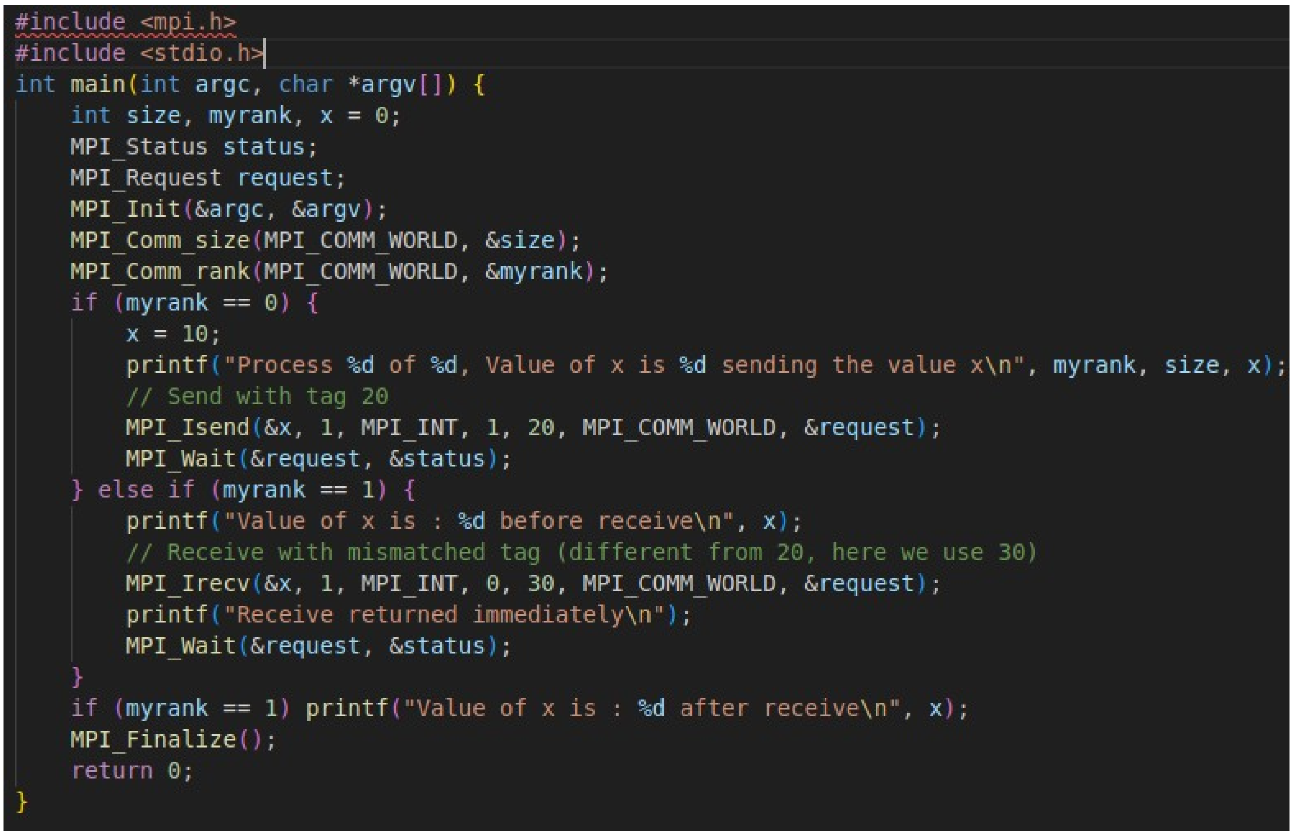
4. **Output Analysis**:
   - The output shows the non-blocking nature of the receive operation. The first print statement on **Process 1** shows `x = 0` because the receive has not completed yet. After `MPI_Wait`, the value of `x` is updated to `10`, confirming that the message has been successfully received.

The use of `MPI_Wait` ensures that the non-blocking communication completes, resulting in the correct value of `x` being printed by Process 1 after the receive. This demonstrates the importance of synchronization in non-blocking MPI operations to avoid using incomplete data.

c) Note down the result by having mismatched tag.

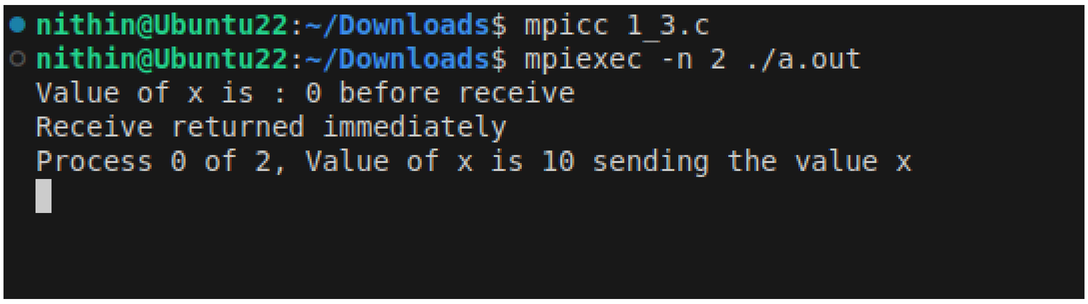
In each case observe whether the process was waiting for completing send/receive operation or if it is continuing its execution

Code



```c
#include <mpi.h>
#include <stdio.h>
int main(int argc, char *argv[]) {
    int size, myrank, x = 0;
    MPI_Status status;
    MPI_Request request;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    if (myrank == 0) {
        x = 10;
        printf("Process %d of %d, Value of x is %d sending the value x\n", myrank, size, x);
        // Send with tag 20
        MPI_Isend(&x, 1, MPI_INT, 1, 20, MPI_COMM_WORLD, &request);
        MPI_Wait(&request, &status);
    } else if (myrank == 1) {
        printf("Value of x is : %d before receive\n", x);
        // Receive with mismatched tag (different from 20, here we use 30)
        MPI_Irecv(&x, 1, MPI_INT, 0, 30, MPI_COMM_WORLD, &request);
        printf("Receive returned immediately\n");
        MPI_Wait(&request, &status);
    }
    if (myrank == 1) printf("Value of x is : %d after receive\n", x);
    MPI_Finalize();
    return 0;
}
```

Output



```
nithin@Ubuntu22:~/Downloads$ mpicc 1_3.c
nithin@Ubuntu22:~/Downloads$ mpiexec -n 2 ./a.out
Value of x is : 0 before receive
Receive returned immediately
Process 0 of 2, Value of x is 10 sending the value x
```

Analysis

Process 0 sends the value of `x` using `MPI_Isend` with a tag of `20`.

- **Process 1** tries to receive the value of `x` using `MPI_Irecv`, but it expects a message with the tag `30`, which does not match the tag `20` used by **Process 0**.
- Because the tags are mismatched, **Process 1** will not receive the message, and it will continue to wait indefinitely for a message with the tag `30` that will never arrive.

Behavior and Output:

1. **Process 0** will successfully send the value `10` with tag `20` and finish its execution normally.
2. **Process 1** will print the initial value of `x` (which is `0`), initiate a non-blocking receive with tag `30`, and then return immediately. However, because there is no message with tag `30`, `MPI_Wait` will block indefinitely, causing **Process 1** to hang and never print the updated value of `x`.

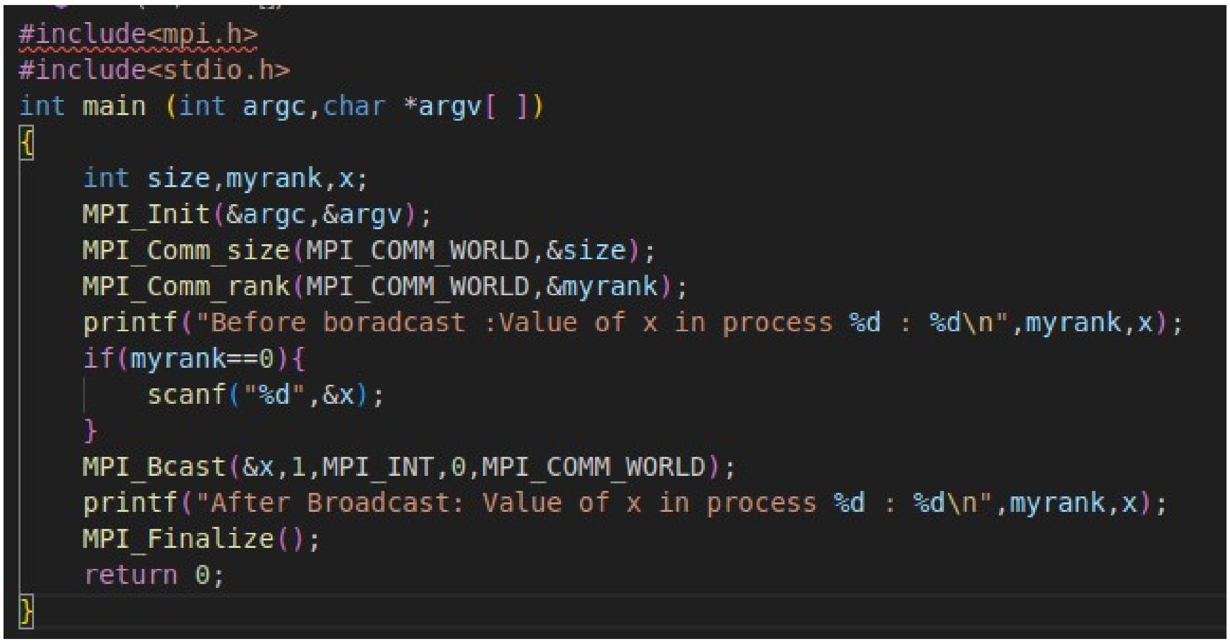
The output will look something like this, but it will not terminate

- **Tag Mismatch**: The communication fails because **Process 1** is waiting for a message with tag `30`, but **Process 0** sends the message with tag `20`. As a result, **Process 1** never receives the message, and `MPI_Wait` blocks indefinitely.
- **No Message Transfer**: The value of `x` on **Process 1** remains unchanged (still `0`) because the message with the expected tag is not received.
- **Blocking Behavior**: Without a matching tag, the program hangs indefinitely since **Process 1** is waiting for a message with the wrong tag.

Using mismatched tags in MPI results in communication failure, where the receiving process waits indefinitely for a message with the expected tag. This highlights the importance of ensuring that the tags in `MPI_Send` and `MPI_Recv` match correctly for successful communication.

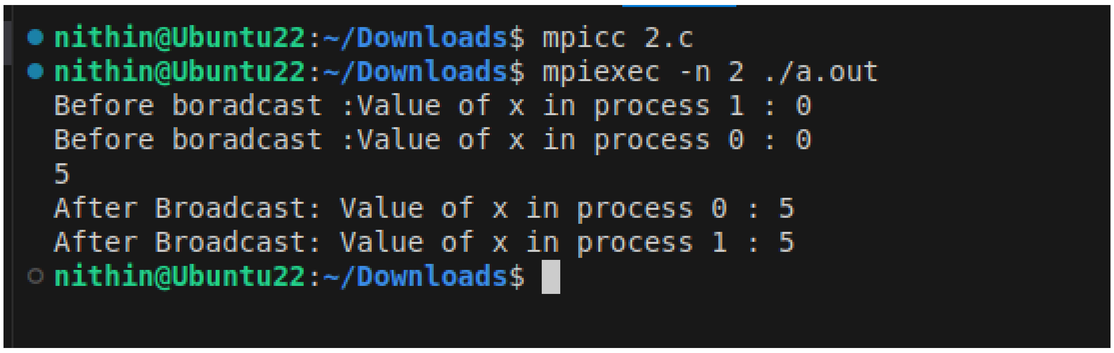
Q2. Demonstration of Bcast(). Record the observation and write the results

Code



```c
#include<mpi.h>
#include<stdio.h>
int main (int argc,char *argv[ ])
{
    int size,myrank,x;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
    printf("Before boradcast :Value of x in process %d : %d\n",myrank,x);
    if(myrank==0){
        scanf("%d",&x);
    }
    MPI_Bcast(&x,1,MPI_INT,0,MPI_COMM_WORLD);
    printf("After Broadcast: Value of x in process %d : %d\n",myrank,x);
    MPI_Finalize();
    return 0;
}
```

Output



```
nithin@Ubuntu22:~/Downloads$ mpicc 2.c
nithin@Ubuntu22:~/Downloads$ mpiexec -n 2 ./a.out
Before boradcast :Value of x in process 1 : 0
Before boradcast :Value of x in process 0 : 0
5
After Broadcast: Value of x in process 0 : 5
After Broadcast: Value of x in process 1 : 5
nithin@Ubuntu22:~/Downloads$
```

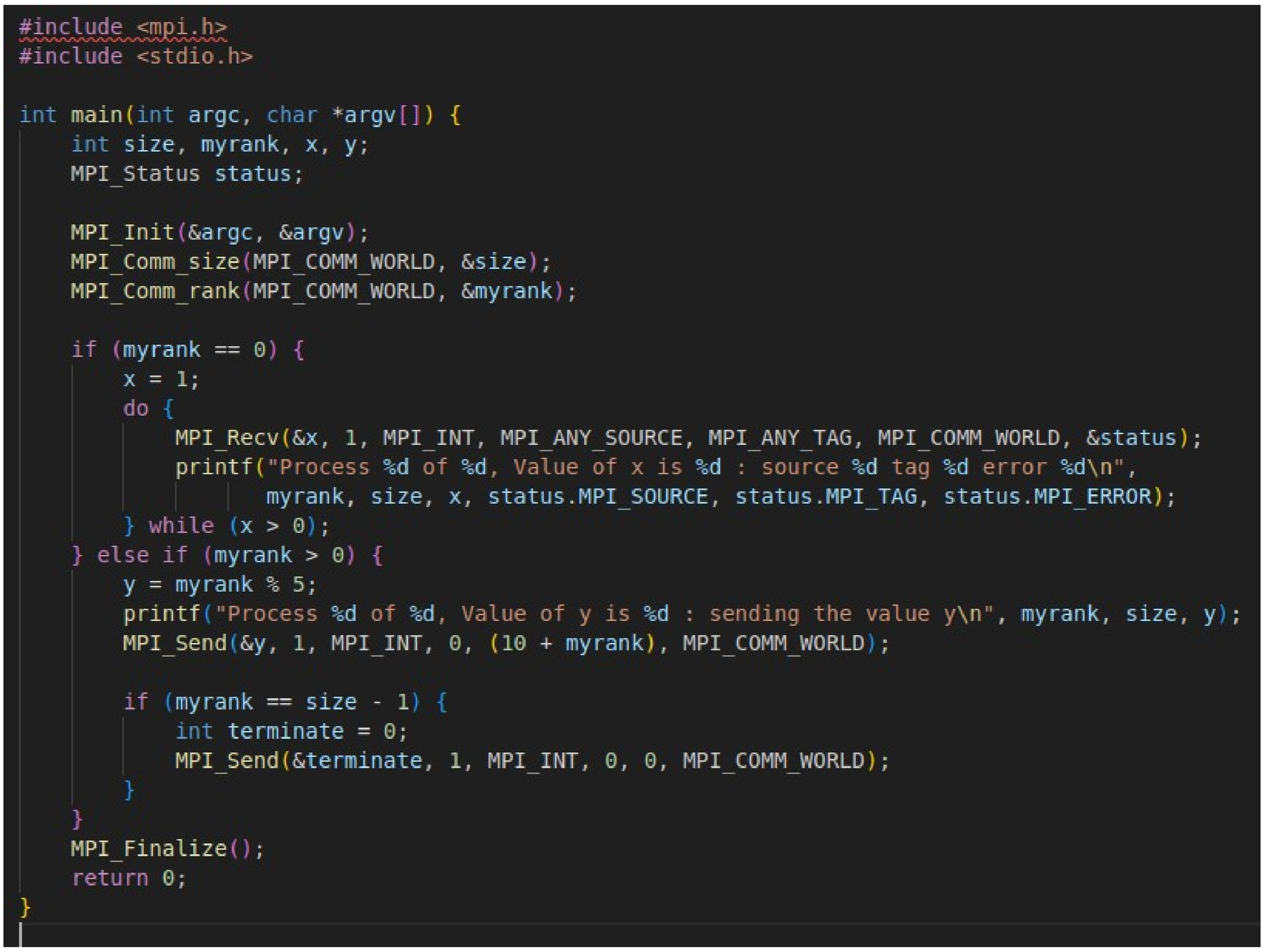
Analysis

This MPI program showcases the use of the `MPI_Bcast` function to synchronize data across multiple processes in a parallel computing environment. Initially, each process retrieves its rank using `MPI_Comm_rank`, and the uninitialized variable `x` is printed. Since `x` is not assigned a value before this point, it prints random or garbage data in each process. Process 0, designated as the "root" process, then takes input for `x` from the user using `scanf`. Once process 0 has the value of `x`, the `MPI_Bcast` function is used to broadcast this value from process 0 to all other processes. The parameters of `MPI_Bcast` include the memory address of `x`, the data type (`MPI_INT`), and the rank of the broadcasting process (in this case, process 0).

After the broadcast, all processes, including process 0, receive the value of `x` and print it. This ensures that regardless of their rank, all processes have the same value of `x`, demonstrating how broadcasting is used in MPI to share information efficiently across all processes. Finally, `MPI_Finalize` is called to clean up the MPI environment and end the program. This method of broadcasting is crucial in parallel applications where one process computes or receives a value and must share it with all other processes, ensuring data consistency in distributed computations.

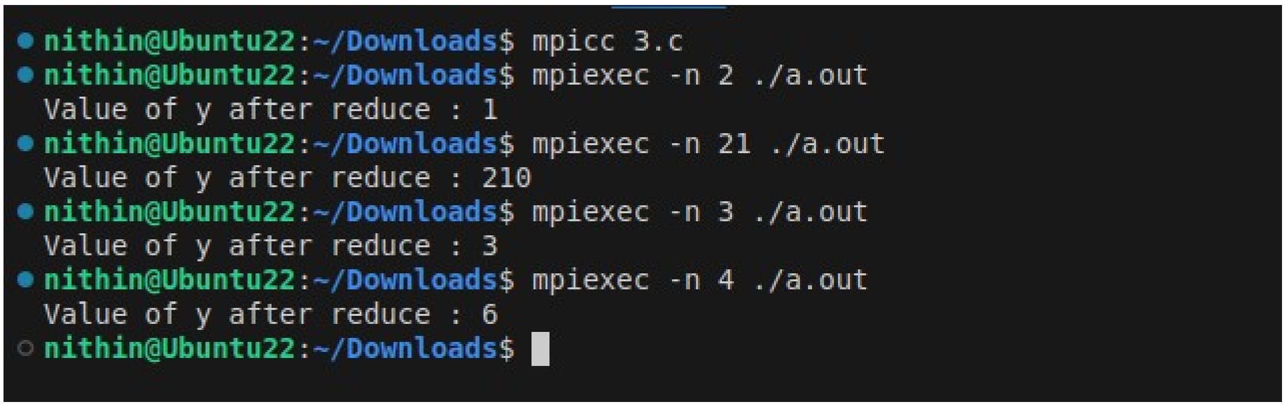
Q3. Demonstration of Reduce ();
Note down the observation and write the result

Code



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

int main(int argc, char *argv[]) {
    int size, myrank, x, y;
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    if (myrank == 0) {
        x = 1;
        do {
            MPI_Recv(&x, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
            printf("Process %d of %d, Value of x is %d : source %d tag %d error %d\n",
                    myrank, size, x, status.MPI_SOURCE, status.MPI_TAG, status.MPI_ERROR);
        } while (x > 0);
    } else if (myrank > 0) {
        y = myrank % 5;
        printf("Process %d of %d, Value of y is %d : sending the value y\n", myrank, size, y);
        MPI_Send(&y, 1, MPI_INT, 0, (10 + myrank), MPI_COMM_WORLD);

        if (myrank == size - 1) {
            int terminate = 0;
            MPI_Send(&terminate, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
        }
    }
    MPI_Finalize();
    return 0;
}
```

Output



```
● nithin@Ubuntu22:~/Downloads$ mpicc 3.c
● nithin@Ubuntu22:~/Downloads$ mpiexec -n 2 ./a.out
  Value of y after reduce : 1
● nithin@Ubuntu22:~/Downloads$ mpiexec -n 21 ./a.out
  Value of y after reduce : 210
● nithin@Ubuntu22:~/Downloads$ mpiexec -n 3 ./a.out
  Value of y after reduce : 3
● nithin@Ubuntu22:~/Downloads$ mpiexec -n 4 ./a.out
  Value of y after reduce : 6
○ nithin@Ubuntu22:~/Downloads$ 
```

Analysis

This MPI (Message Passing Interface) program demonstrates a simple example of parallel reduction across multiple processes. The key function used is `MPI_Reduce`, which performs a global reduction operation (in this case, summing) on the value of `x` (initialized as the rank of each process) and stores the result in `y`, only on the process with rank 0 (the root process).

Observations:

The program starts by initializing MPI with `MPI_Init` and retrieves the total number of processes (`size`) and each process's rank (`myrank`) using `MPI_Comm_size` and `MPI_Comm_rank`, respectively.

Each process assigns its rank (`myrank`) to variable `x`, which effectively creates a scenario where each process has a unique integer to contribute to the reduction operation.

MPI_Reduce: This function aggregates the values of `x` from all processes using the `MPI_SUM` operation. The result is stored in `y` on the root process (rank 0). The root process (rank 0) will then print the sum of the ranks of all processes.

The program concludes with `MPI_Finalize`, which cleans up the MPI environment.

The sum of all process ranks (i.e., 0 + 1 + 2 + ... + (size-1)) is computed and printed by process 0. For \(n \) processes, the result should be \((n-1) \times n / 2 \).

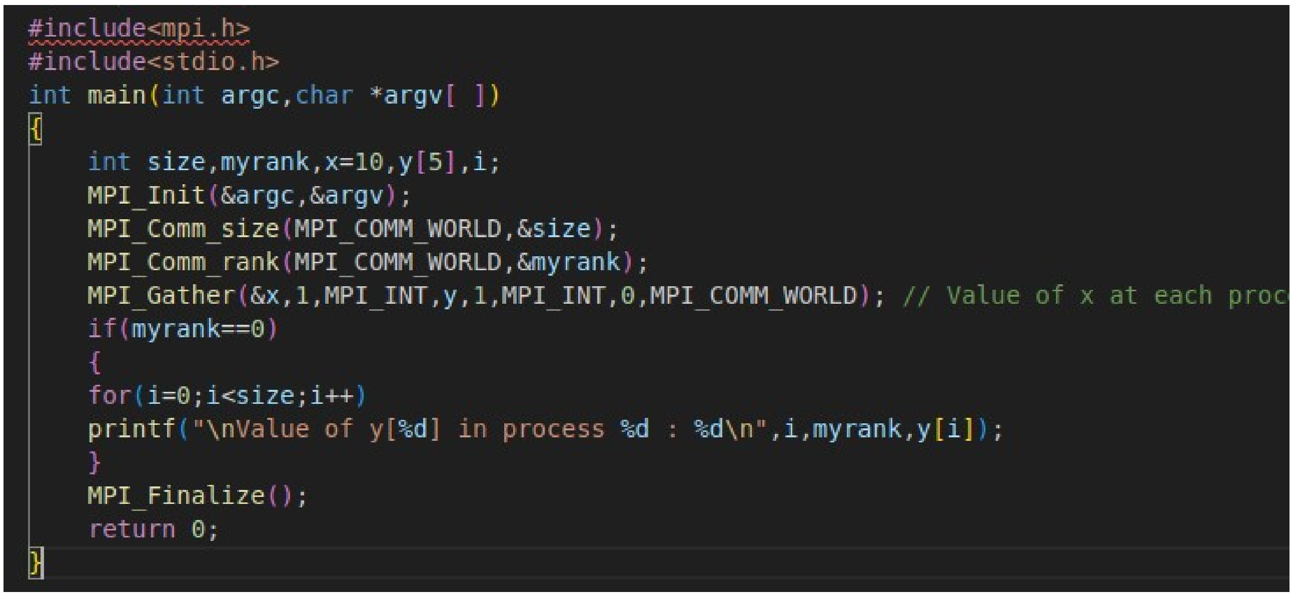
Only the root process (rank 0) has access to the final result (`y`), so the other processes remain unaware of it.

The program is scalable in terms of the number of processes, as the `MPI_Reduce` operation effectively handles aggregation over multiple processes. However, only one result is communicated to the root process.

In summary, this program demonstrates how MPI can be used to perform collective operations like summing values from all processes, and it highlights the use of reduction operations in parallel computing.

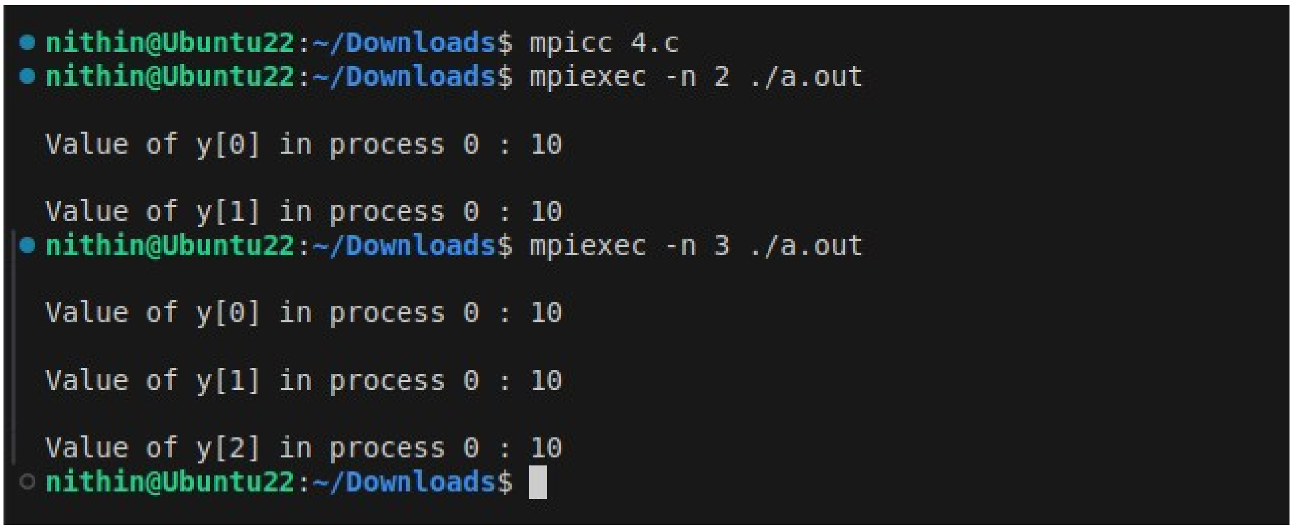
Q4. Program 4: Demonstration of MPI_Gather();
Note down the observation and explain the result.

Code



```
#include<mpi.h>
#include<stdio.h>
int main(int argc,char *argv[ ])
{
    int size,myrank,x=10,y[5],i;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
    MPI_Gather(&x,1,MPI_INT,y,1,MPI_INT,0,MPI_COMM_WORLD); // Value of x at each proc
    if(myrank==0)
    {
    for(i=0;i<size;i++)
    printf("\nValue of y[%d] in process %d : %d\n",i,myrank,y[i]);
    }
    MPI_Finalize();
    return 0;
}
```

Output



```
● nithin@Ubuntu22:~/Downloads$ mpicc 4.c
● nithin@Ubuntu22:~/Downloads$ mpiexec -n 2 ./a.out

  Value of y[0] in process 0 : 10

  Value of y[1] in process 0 : 10
● nithin@Ubuntu22:~/Downloads$ mpiexec -n 3 ./a.out

  Value of y[0] in process 0 : 10

  Value of y[1] in process 0 : 10

  Value of y[2] in process 0 : 10
○ nithin@Ubuntu22:~/Downloads$ 
```

Analysis

This MPI (Message Passing Interface) program demonstrates the use of `MPI_Gather`, which collects data from all processes and sends it to a designated root process. Here, each process has a value `x = 10` and uses `MPI_Gather` to collect these values into the array `y` on the root process (process 0).

The program starts by initializing the MPI environment with `MPI_Init`, after which it retrieves the total number of processes (`size`) and the rank of each process (`myrank`) using MPI_Comm_size` and

`MPI_Comm_rank`.Each process holds the integer variable x, which is initialized to 10 for every process.

MPI_Gather: This function is called with x (the value to be sent from each process), and it gathers these values into the array y on process 0 (root). Each process contributes one value (x), and all gathered values are stored in the array `y` of size 5 (or more, depending on the number of processes).

If the current process has rank 0, it prints the contents of the array y, showing the values collected from all processes. Since each process initializes x to 10, the root process will print the value 10 for every entry in y corresponding to the number of processes.

`MPI_Gather` gathers values from all participating processes and stores them in the root process's array. Since all processes send the same value (10), the root process will see an array where each element is `10`.

The array `y` in process 0 must be large enough to store data from all processes. In this case, it should have at least as many elements as there are processes in `MPI_COMM_WORLD`.

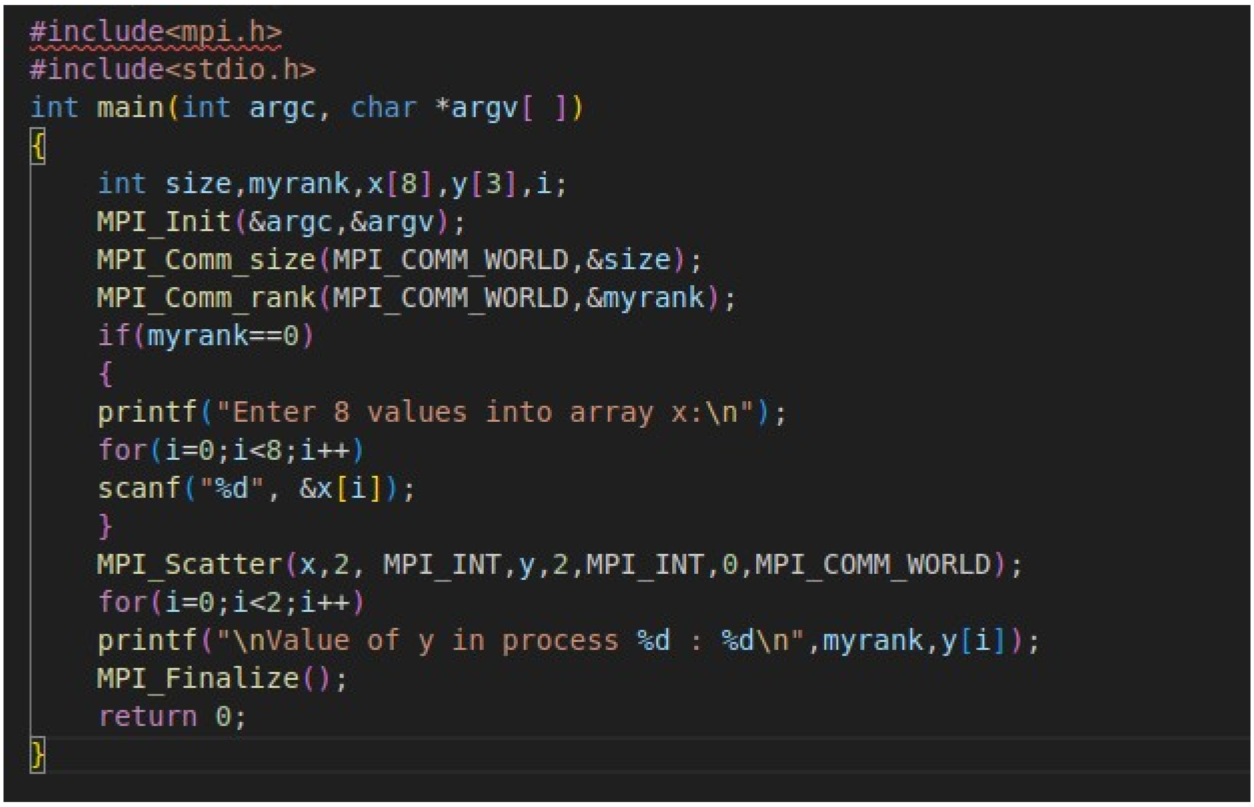
Only process 0 prints the gathered values (`y`), and for \(n \) processes, it will output the value `10` repeated \(n \) times.

The program can scale to any number of processes, with `MPI_Gather` collecting data from all processes efficiently.

In summary, this program highlights the `MPI_Gather` operation, which allows the root process (rank 0) to collect data from all other processes. In this specific case, it gathers the value `10` from every process and prints it from process 0.

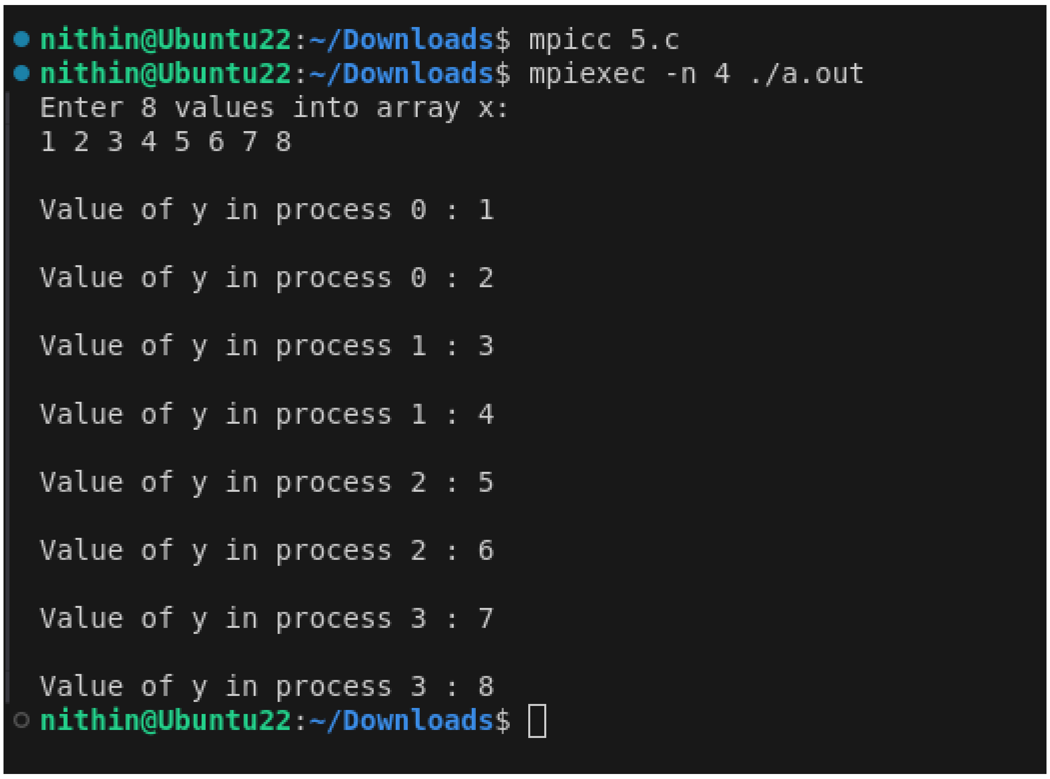
Q5. Demonstration of MPI_Scatter();
Note: Atleast 4 MPI processes are needed to scatter the input array to each process. i.e. each process receives two chunks from the array

Code



```c
#include<mpi.h>
#include<stdio.h>
int main(int argc, char *argv[ ])
{
    int size,myrank,x[8],y[3],i;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
    if(myrank==0)
    {
    printf("Enter 8 values into array x:\n");
    for(i=0;i<8;i++)
    scanf("%d", &x[i]);
    }
    MPI_Scatter(x,2, MPI_INT,y,2,MPI_INT,0,MPI_COMM_WORLD);
    for(i=0;i<2;i++)
    printf("\nValue of y in process %d : %d\n",myrank,y[i]);
    MPI_Finalize();
    return 0;
}
```

Output



```
nithin@Ubuntu22:~/Downloads$ mpicc 5.c
nithin@Ubuntu22:~/Downloads$ mpiexec -n 4 ./a.out
Enter 8 values into array x:
1 2 3 4 5 6 7 8

Value of y in process 0 : 1

Value of y in process 0 : 2

Value of y in process 1 : 3

Value of y in process 1 : 4

Value of y in process 2 : 5

Value of y in process 2 : 6

Value of y in process 3 : 7

Value of y in process 3 : 8
nithin@Ubuntu22:~/Downloads$ 
```

Analysis

This MPI program uses the `MPI_Scatter` function to distribute parts of an array from the root process (rank 0) to all other processes. The program starts by initializing the MPI environment and determining both the number of processes (`size`) and the rank of each process (`myrank`). The root process (process 0) is responsible for initializing an array `x` of 8 integers, which it fills with values entered by the user. This input step ensures that the root process has data that it can later distribute to the other processes.

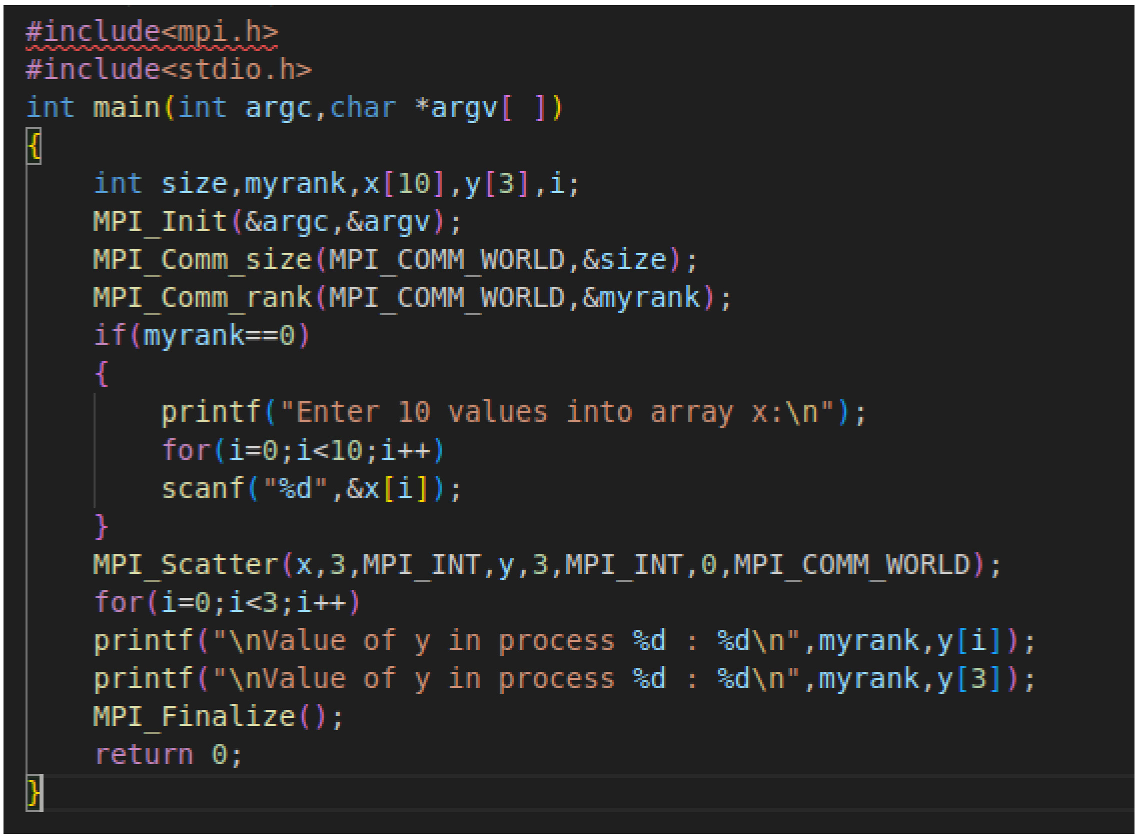
Once the array `x` is populated on the root process, the `MPI_Scatter` function is used to divide this array into smaller chunks. Specifically, it sends 2 integers from `x` to each process, including the root process itself. Each process then receives its own portion of the array into the local array `y`. The function works by assigning parts of `x` to each process; since there are 8 elements in total, and each process receives 2 elements, this setup supports up to 4 processes.

After the scattering operation, each process prints the values it received in its `y` array. The output shows that each process has its own segment of the original `x` array, distributed by the root process. For example, if process 0 has the full `x` array with 8 elements, it will send the first two elements to itself, the next two elements to process 1, and so on.

This code demonstrates how `MPI_Scatter` can be used to partition a larger dataset across multiple processes in parallel computing. It allows for efficient distribution of data, where each process operates on a subset of the total data. This distribution is particularly useful when tasks are independent and can be parallelized. The program ends by finalizing the MPI environment, signaling the end of parallel execution.

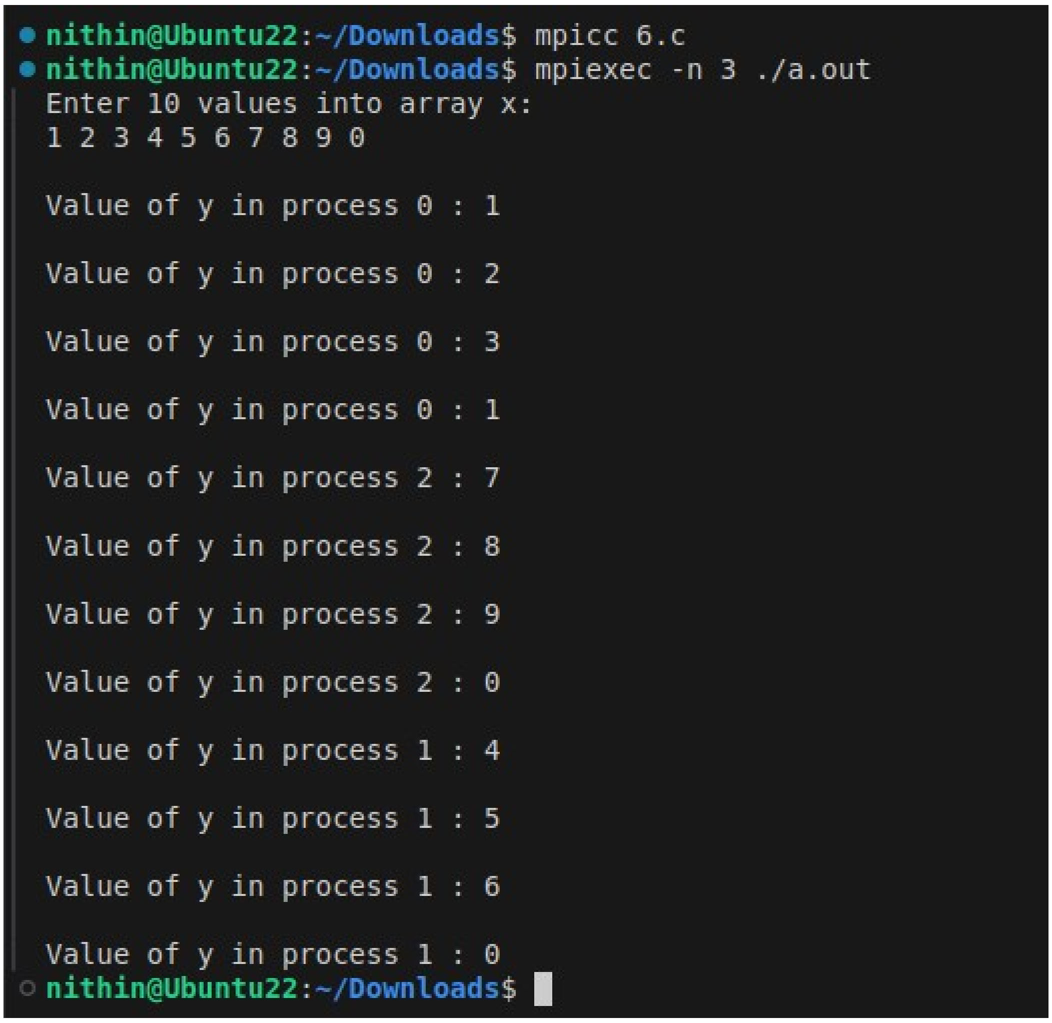
Program 6: Demonstration of MPI_Scatter() with partial scatter;
Note: Program is hardcoded to work with 3 processes receiving three chunks from the array.
Note down the differences between program 5 and program 6.

Code



```c
#include<mpi.h>
#include<stdio.h>
int main(int argc,char *argv[ ])
{
    int size,myrank,x[10],y[3],i;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
    if(myrank==0)
    {
        printf("Enter 10 values into array x:\n");
        for(i=0;i<10;i++)
        scanf("%d",&x[i]);
    }
    MPI_Scatter(x,3,MPI_INT,y,3,MPI_INT,0,MPI_COMM_WORLD);
    for(i=0;i<3;i++)
    printf("\nValue of y in process %d : %d\n",myrank,y[i]);
    printf("\nValue of y in process %d : %d\n",myrank,y[3]);
    MPI_Finalize();
    return 0;
}
```

Output



```
● nithin@Ubuntu22:~/Downloads$ mpicc 6.c
● nithin@Ubuntu22:~/Downloads$ mpiexec -n 3 ./a.out
  Enter 10 values into array x:
  1 2 3 4 5 6 7 8 9 0

  Value of y in process 0 : 1

  Value of y in process 0 : 2

  Value of y in process 0 : 3

  Value of y in process 0 : 1

  Value of y in process 2 : 7

  Value of y in process 2 : 8

  Value of y in process 2 : 9

  Value of y in process 2 : 0

  Value of y in process 1 : 4

  Value of y in process 1 : 5

  Value of y in process 1 : 6

  Value of y in process 1 : 0
○ nithin@Ubuntu22:~/Downloads$ 
```

Analysis

Differences Between Program 5 and Program 6

1. **Array Size and Number of Processes**:
    - In **Program 5**, an array of size **8** is scattered among up to **4 processes**, where each process receives **2 elements** (since `MPI_Scatter(x, 2, MPI_INT, y, 2, MPI_INT, 0, MPI_COMM_WORLD)` is used). The root process (rank 0) inputs values into the array.
    - In **Program 6**, the size of the array is **10**, and it is hardcoded to work with **3 processes**, where each process receives **3 elements** (due to `MPI_Scatter(x, 3, MPI_INT, y, 3, MPI_INT, 0, MPI_COMM_WORLD)`).
    -
2. **Input and Output Handling**:
    - In **Program 5**, the output is printed by each process to show the values of `y` that each has received after the scatter operation.

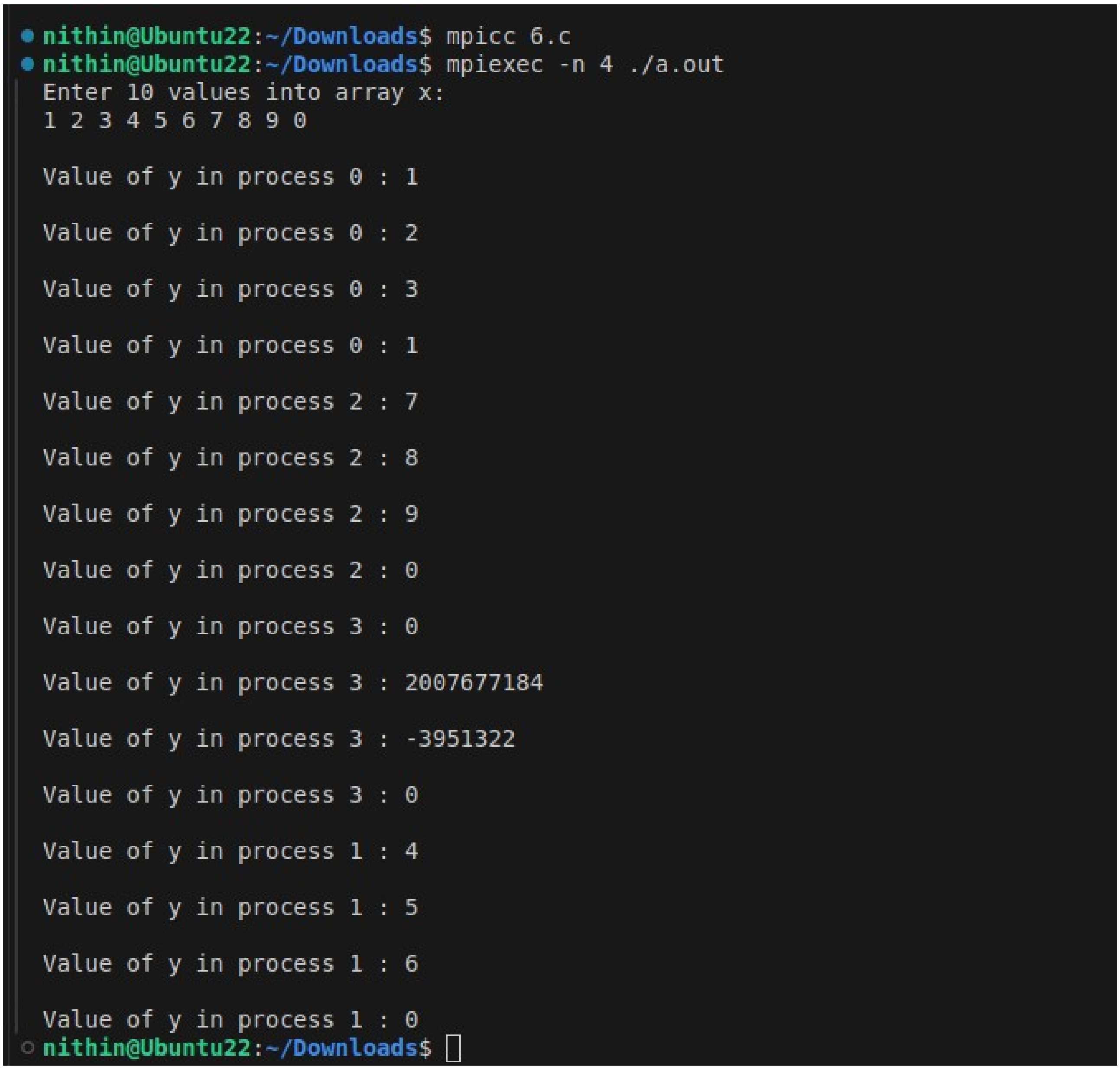
- In **Program 6**, after the `MPI_Scatter`, the program attempts to print four values from the `y` array, specifically `y[3]`, which does not exist within the allocated size for `y`. This leads to undefined behavior, as it tries to access an index out of bounds .

3. **Error in Accessing Out-of-Bounds Index**:

   - **Program 5** correctly allocates the size of the `y` array according to the number of elements received by each process (2 elements), preventing any out-of-bounds access.
   - In **Program 6**, when printing `y[3]`, this will lead to undefined behavior since the `y` array is only of size **3**. This could result in garbage values or a segmentation fault depending on the system and memory allocation.

What happens if the array values are scattered into 4 processes? Also try to note down the results.

Output



```
nithin@Ubuntu22:~/Downloads$ mpicc 6.c
nithin@Ubuntu22:~/Downloads$ mpiexec -n 4 ./a.out
Enter 10 values into array x:
1 2 3 4 5 6 7 8 9 0

Value of y in process 0 : 1

Value of y in process 0 : 2

Value of y in process 0 : 3

Value of y in process 0 : 1

Value of y in process 2 : 7

Value of y in process 2 : 8

Value of y in process 2 : 9

Value of y in process 2 : 0

Value of y in process 3 : 0

Value of y in process 3 : 2007677184

Value of y in process 3 : -3951322

Value of y in process 3 : 0

Value of y in process 1 : 4

Value of y in process 1 : 5

Value of y in process 1 : 6

Value of y in process 1 : 0
nithin@Ubuntu22:~/Downloads$ 
```

Analysis

When the code was run for 4 processes, the following outcomes were observed:

1. **Process 0** received values `1, 2, 3`, but oddly, it also printed `1` again for `y[3]`, which should not exist, as only 3 elements were scattered per process.
2. **Process 1** received the correct values `4, 5, 6`, but printed `0` for `y[3]`, which is an out-of-bounds value, reflecting undefined behavior.
3. **Process 2** correctly received values `7, 8, 9`, but also printed `0` for `y[3]`, again showing an access to an invalid index.

4. **Process 3** exhibited the most irregular behavior, where it received only `0` from the array and then printed unexpected values for `y[1]`, `y[2]`, and `y[3]`. These values (`2007677184`, `-3951322`, and `0`) are garbage values, indicating that `MPI_Scatter` did not assign valid values to `y` for process 3.

Explanation of the Behavior:

1. **Array Size Mismatch**:
    - The array `x` has **10 elements** and the scatter operation is trying to distribute **3 elements per process** (`MPI_Scatter(x, 3, MPI_INT, y, 3, MPI_INT, 0, MPI_COMM_WORLD)`).
    - For **4 processes**, this would require scattering a total of 4×3=12 elements, but the array only contains 10 elements. Therefore, the last process (process 3) does not receive valid data because there are not enough elements to scatter. This results in undefined or garbage values being stored in `y` for process 3.

2. **Out-of-Bounds Access**:
    - The program prints values of `y[3]`, which is out of bounds for the allocated array (`y[3]` is invalid since the array is defined to have only 3 elements, `y[0]`, `y[1]`, and `y[2]`). Accessing `y[3]` leads to the printing of garbage or random values, as observed in processes 0, 1, 2, and especially process 3.
    - In C, out-of-bounds access does not automatically cause an error, but leads to undefined behavior. This is why garbage values like `2007677184` and `-3951322` are printed for process 3.

3. **Process 3 Receiving Zeroes**:
    - When `MPI_Scatter` does not have enough data to send, it might default to sending zero or leave the values in the local memory unchanged. For process 3, it received `0` as valid data because there were not enough elements in `x` to send 3 values. The remaining elements after scattering (i.e., positions `y[1]` and `y[2]` for process 3) contained garbage due to uninitialized memory.

Why This Happened:

- **Partial Scatter Issue**: The code attempts to scatter **3 elements** per process, but with **10 elements in the array**, it's impossible to evenly distribute these among **4 processes**. `MPI_Scatter` expects to distribute exactly 3 elements to each process, but only 10 elements are available, causing process 3 to receive incorrect or garbage values.
- **Out-of-Bounds Access**: The attempt to access and print `y[3]` in all processes causes undefined behavior because `y` is only allocated to store 3 integers. This leads to garbage values being printed, which is common in C when accessing invalid memory.

Correcting the Issue:

1. **Ensure Proper Array Size**: Either adjust the size of the array `x` to have enough elements (e.g., 12 elements for 4 processes), or adjust the number of elements scattered per process. For example, scatter **2 elements** per process if you only have 10 elements (`MPI_Scatter(x, 2, MPI_INT, y, 2, MPI_INT, 0, MPI_COMM_WORLD)`).
2. **Avoid Out-of-Bounds Access**: Do not attempt to access or print `y[3]` when only 3 elements are received. The correct loop should only iterate over the valid indices (`y[0]` to `y[2]`).