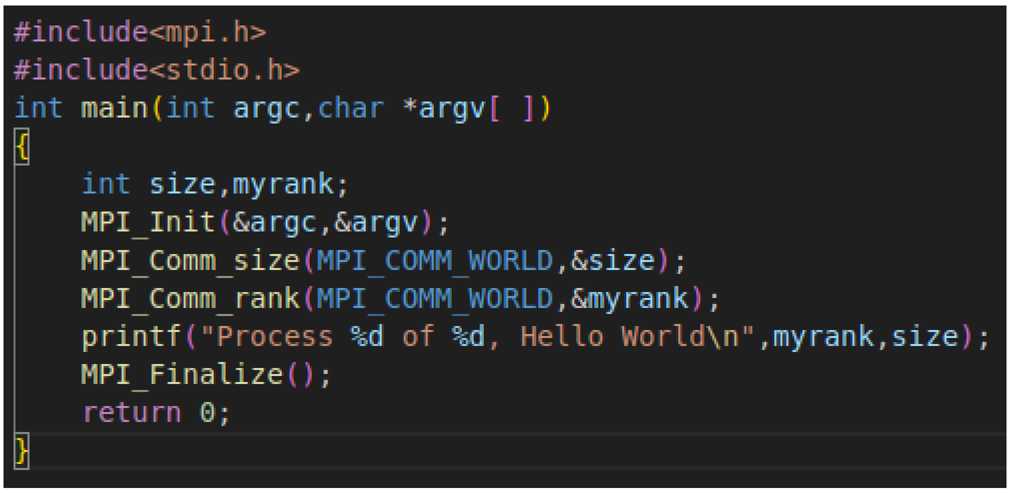
Nithin S
221IT085

IT301 Lab Assignment 6

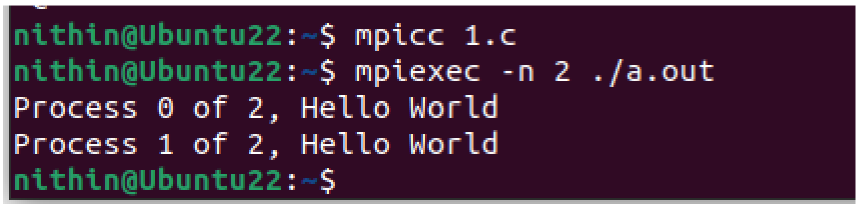
Q1. Simple Hello World program to find rank and size of communication world

Code



```c
#include<mpi.h>
#include<stdio.h>
int main(int argc,char *argv[ ])
{
    int size,myrank;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
    printf("Process %d of %d, Hello World\n",myrank,size);
    MPI_Finalize();
    return 0;
}
```

Output



```
nithin@Ubuntu22:~$ mpicc 1.c
nithin@Ubuntu22:~$ mpiexec -n 2 ./a.out
Process 0 of 2, Hello World
Process 1 of 2, Hello World
nithin@Ubuntu22:~$
```

Analysis

This C program demonstrates a simple use of the Message Passing Interface (MPI) to print a "Hello World" message from each process in a parallel computing environment. It begins by including the necessary headers for MPI and standard input/output functions.

The main function initializes the MPI environment and retrieves the total number of processes as well as the rank of each individual process. Each process then outputs its rank along with the total number of processes.

Finally, the program cleans up the MPI environment before exiting. When executed with multiple processes, this program effectively illustrates the parallel execution capabilities of MPI, with each process independently printing its message.

To run the program, it must be compiled using an MPI compiler and executed with a command that specifies the number of processes.

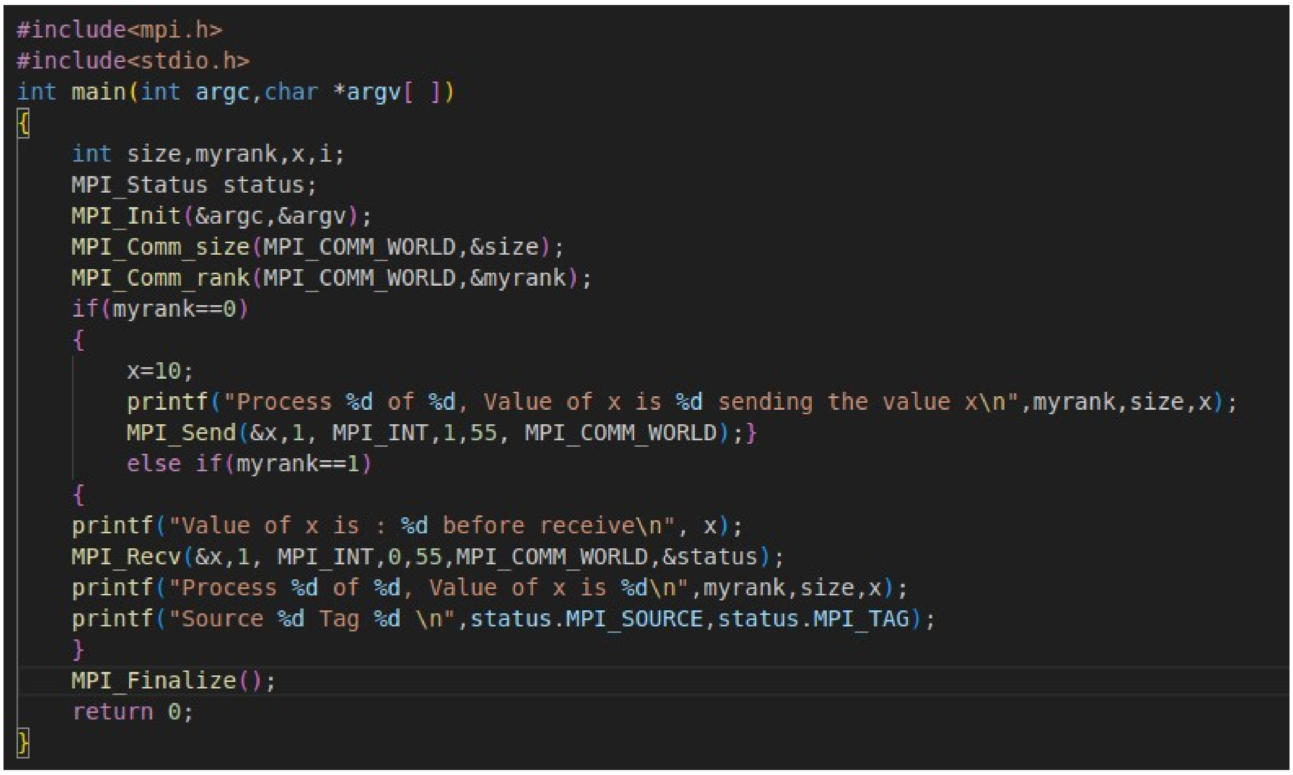
MPI_COMM_WORLD: This is the default communicator that includes all the processes initiated by the MPI environment. It is often used for collective communications.

Ranks: Each process has a unique rank within a communicator, which helps to identify and manage communications between processes.

Parallel Execution: When this program is run with multiple processes (e.g., using mpirun or mpiexec), each process will independently execute the code within main(), leading to multiple output lines indicating how many processes are running.

Q2. MPI_Send() and MPI_Recv() for sending an integer.
(a) Note down source, destination and tag.

Code



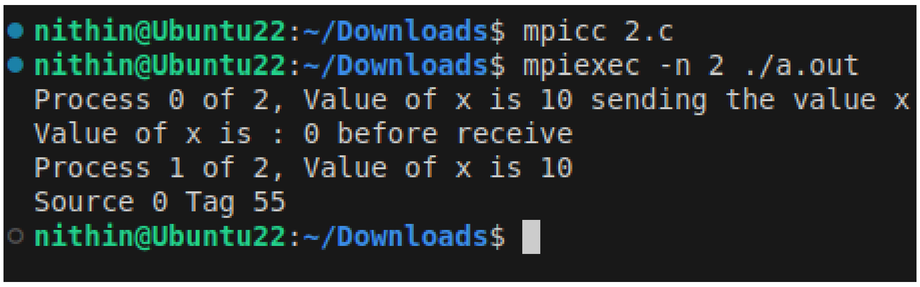
```c
#include<mpi.h>
#include<stdio.h>
int main(int argc,char *argv[ ])
{
    int size,myrank,x,i;
    MPI_Status status;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
    if(myrank==0)
    {
        x=10;
        printf("Process %d of %d, Value of x is %d sending the value x\n",myrank,size,x);
        MPI_Send(&x,1, MPI_INT,1,55, MPI_COMM_WORLD);}
        else if(myrank==1)
    {
    printf("Value of x is : %d before receive\n", x);
    MPI_Recv(&x,1, MPI_INT,0,55,MPI_COMM_WORLD,&status);
    printf("Process %d of %d, Value of x is %d\n",myrank,size,x);
    printf("Source %d Tag %d \n",status.MPI_SOURCE,status.MPI_TAG);
    }
    MPI_Finalize();
    return 0;
}
```

Source: The source of the message is process 0 (the sender).

Destination: The destination of the message is process 1 (the receiver).

Tag: The tag used for the message is 55.

Output



```
nithin@Ubuntu22:~/Downloads$ mpicc 2.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
Process 1 of 2, Value of x is 10
Source 0 Tag 55
nithin@Ubuntu22:~/Downloads$
```

Analysis

This MPI code provides a clear demonstration of basic inter-process communication using the Message Passing Interface. It begins by including the necessary headers for MPI and standard input/output functions.

In the `main` function, the program initializes the MPI environment, which is essential for any MPI application, and retrieves the number of processes and the rank of each process.

Process 0, which has the highest priority, initializes an integer variable `x` to 10. It then prints its rank, the total number of processes, and the value of `x`, followed by sending this value to Process 1 using the `MPI_Send` function.

Process 1, upon execution, initially prints an uninitialized value for `x`, illustrating how it starts with a default state. It then uses `MPI_Recv` to receive the value sent from Process 0.

After receiving the value, it outputs its rank, the total number of processes, the received value, and additional details about the source of the message and its tag.

The use of `MPI_Status` allows Process 1 to gain insights into the communication, such as the sender's rank and the tag associated with the message, which can be useful for managing multiple message types in more complex applications.

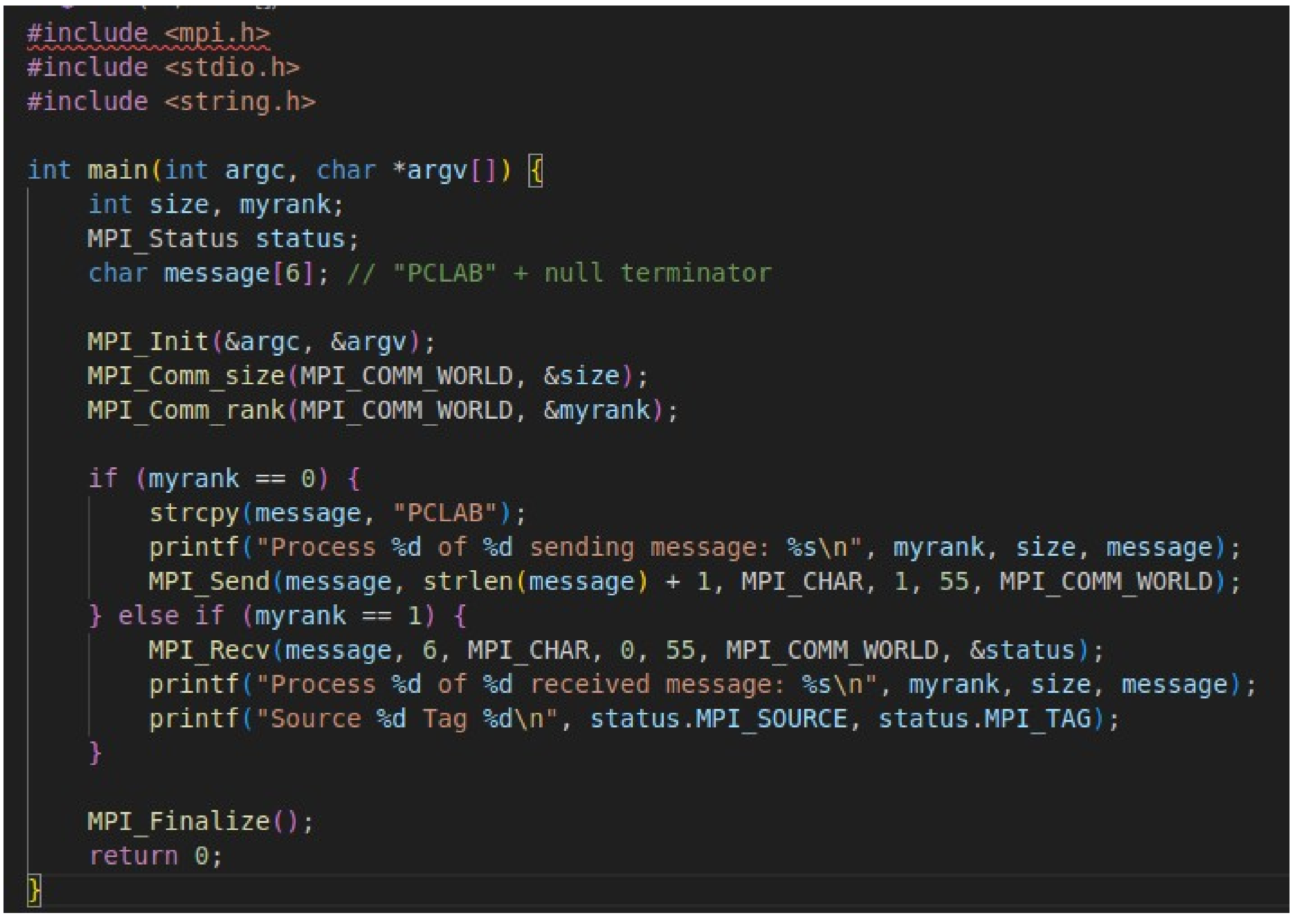
Finally, both processes call `MPI_Finalize` to clean up the MPI environment before exiting, ensuring proper resource management.

Overall, this code serves as a practical example of how MPI can facilitate communication between processes in a parallel computing setup. It highlights key MPI concepts such as ranks, message passing, and process coordination, making it a foundational exercise for anyone learning parallel programming.

By executing this code with at least two processes, users can observe the dynamic interaction between the processes, providing insights into the workings of parallel computing.

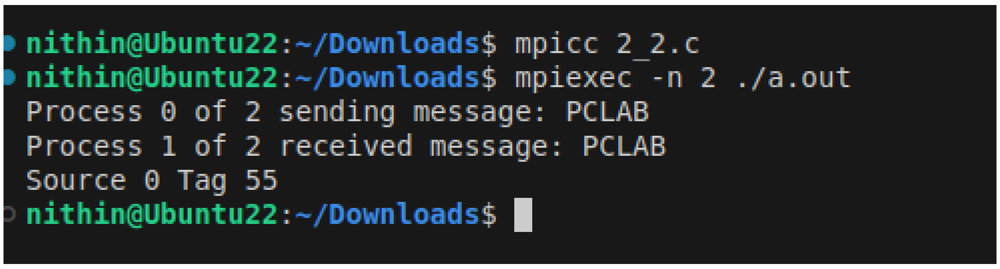
(b) Modify the program to send the string "PCLAB" and add screenshot of the result.

Code



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

int main(int argc, char *argv[]) {
    int size, myrank;
    MPI_Status status;
    char message[6]; // "PCLAB" + null terminator

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

    if (myrank == 0) {
        strcpy(message, "PCLAB");
        printf("Process %d of %d sending message: %s\n", myrank, size, message);
        MPI_Send(message, strlen(message) + 1, MPI_CHAR, 1, 55, MPI_COMM_WORLD);
    } else if (myrank == 1) {
        MPI_Recv(message, 6, MPI_CHAR, 0, 55, MPI_COMM_WORLD, &status);
        printf("Process %d of %d received message: %s\n", myrank, size, message);
        printf("Source %d Tag %d\n", status.MPI_SOURCE, status.MPI_TAG);
    }

    MPI_Finalize();
    return 0;
}
```

Ouput



```
nithin@Ubuntu22:~/Downloads$ mpicc 2_2.c
nithin@Ubuntu22:~/Downloads$ mpiexec -n 2 ./a.out
Process 0 of 2 sending message: PCLAB
Process 1 of 2 received message: PCLAB
Source 0 Tag 55
nithin@Ubuntu22:~/Downloads$
```

Analysis

This MPI code illustrates how to send and receive a string message between two processes in a parallel computing environment.

It begins by including the necessary headers for MPI and standard I/O operations. Within the `main` function, the MPI environment is initialized, and the total number of processes and the rank of each process are obtained.

Process 0, identified by its rank, creates a string message "PCLAB" and prints a notification indicating that it is sending this message to Process 1.

It uses the `MPI_Send` function to transmit the string, ensuring to include the null terminator by sending the length of the string plus one. On the other hand, Process 1 waits to receive this message using `MPI_Recv`, specifying the expected message length. Upon receiving the message, it prints out the received string along with its rank, the total number of processes, and additional information about the source of the message and its tag using the `MPI_Status` object.

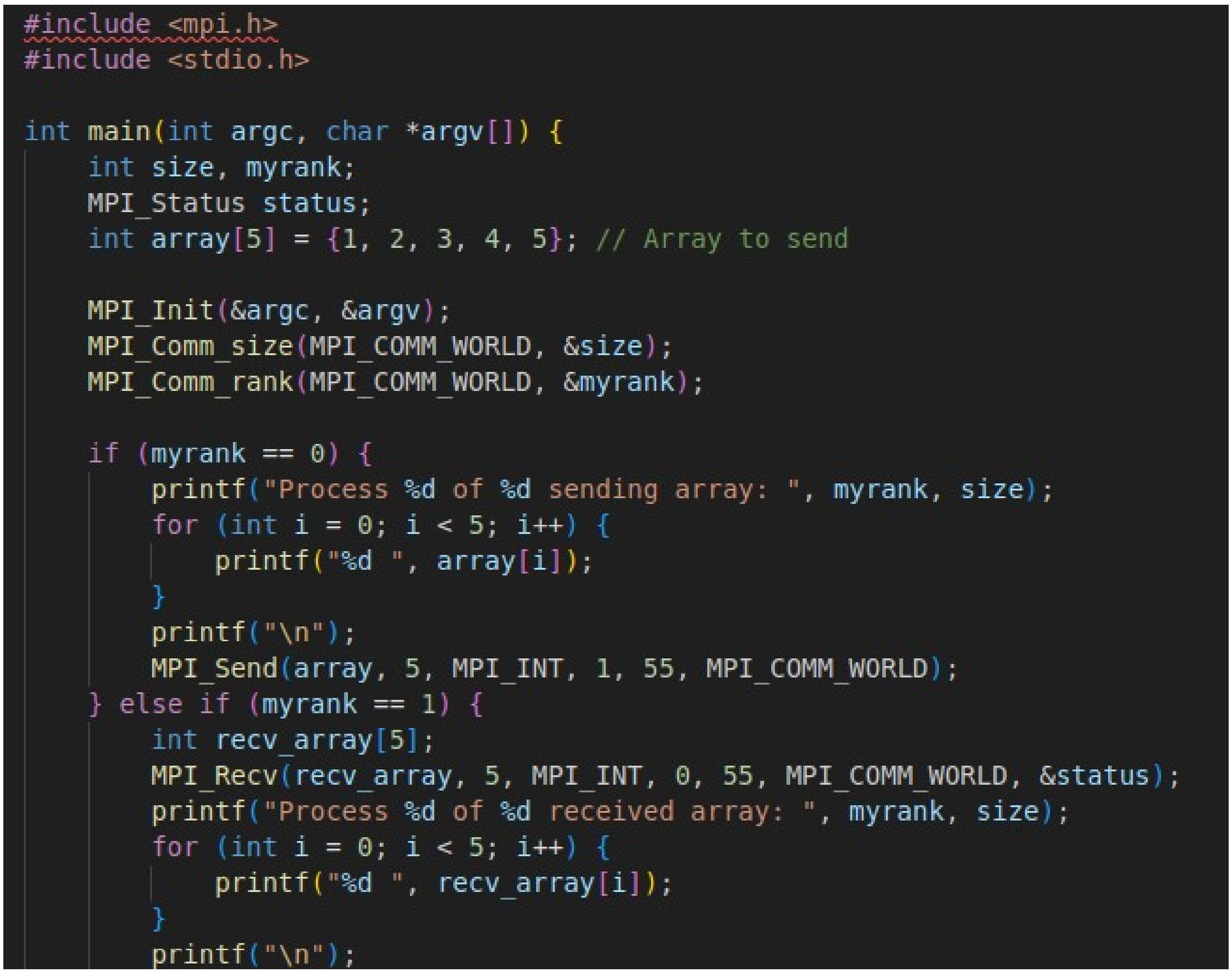
The program concludes with a call to `MPI_Finalize`, which cleans up the MPI environment, ensuring that all resources are properly released.

This example effectively demonstrates fundamental MPI concepts such as message passing, ranks, and process communication, making it an excellent exercise for understanding string handling and inter-process communication in a parallel programming context.

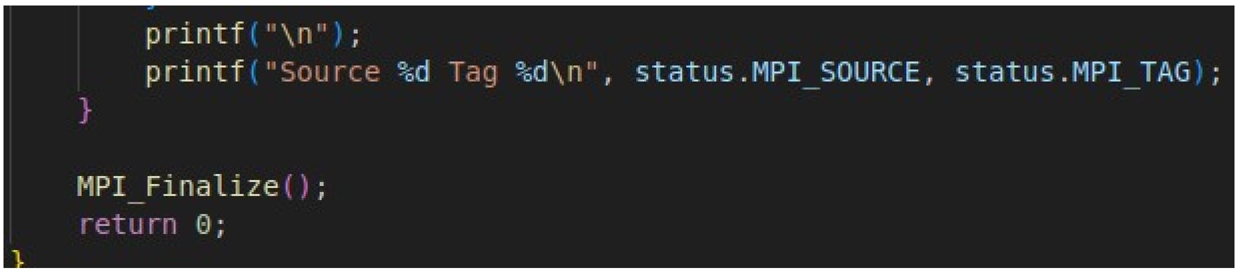
By executing this code with at least two processes, we can observe the straightforward exchange of messages and gain insight into the basics of using MPI for collaborative computing tasks.

(c) Modify the program to send array of elements and add screenshot of the result.

Code

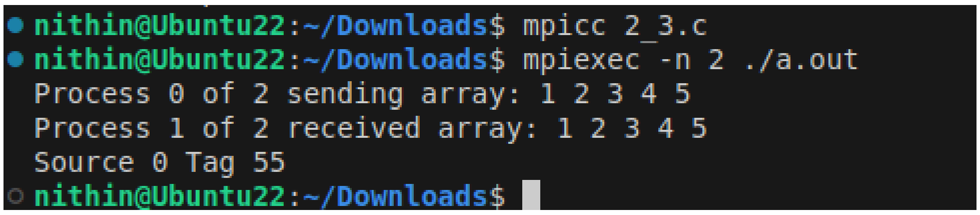


```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
    int size, myrank;
    MPI_Status status;
    int array[5] = {1, 2, 3, 4, 5}; // Array to send

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

    if (myrank == 0) {
        printf("Process %d of %d sending array: ", myrank, size);
        for (int i = 0; i < 5; i++) {
            printf("%d ", array[i]);
        }
        printf("\n");
        MPI_Send(array, 5, MPI_INT, 1, 55, MPI_COMM_WORLD);
    } else if (myrank == 1) {
        int recv_array[5];
        MPI_Recv(recv_array, 5, MPI_INT, 0, 55, MPI_COMM_WORLD, &status);
        printf("Process %d of %d received array: ", myrank, size);
        for (int i = 0; i < 5; i++) {
            printf("%d ", recv_array[i]);
        }
        printf("\n");
```



```c
        printf("\n");
        printf("Source %d Tag %d\n", status.MPI_SOURCE, status.MPI_TAG);
    }

    MPI_Finalize();
    return 0;
}
```

Output



```
nithin@Ubuntu22:~/Downloads$ mpicc 2_3.c
nithin@Ubuntu22:~/Downloads$ mpiexec -n 2 ./a.out
Process 0 of 2 sending array: 1 2 3 4 5
Process 1 of 2 received array: 1 2 3 4 5
Source 0 Tag 55
nithin@Ubuntu22:~/Downloads$
```

Analysis

This MPI code demonstrates the transmission of an integer array between two processes in a parallel computing environment. It starts by including the necessary headers for MPI and standard I/O.

Inside the `main` function, the MPI environment is initialized, and both the total number of processes and the rank of each process are determined.

Process 0 is responsible for sending data. It initializes an integer array with five elements and prints the array to the console, indicating that it is about to send this data to Process 1.

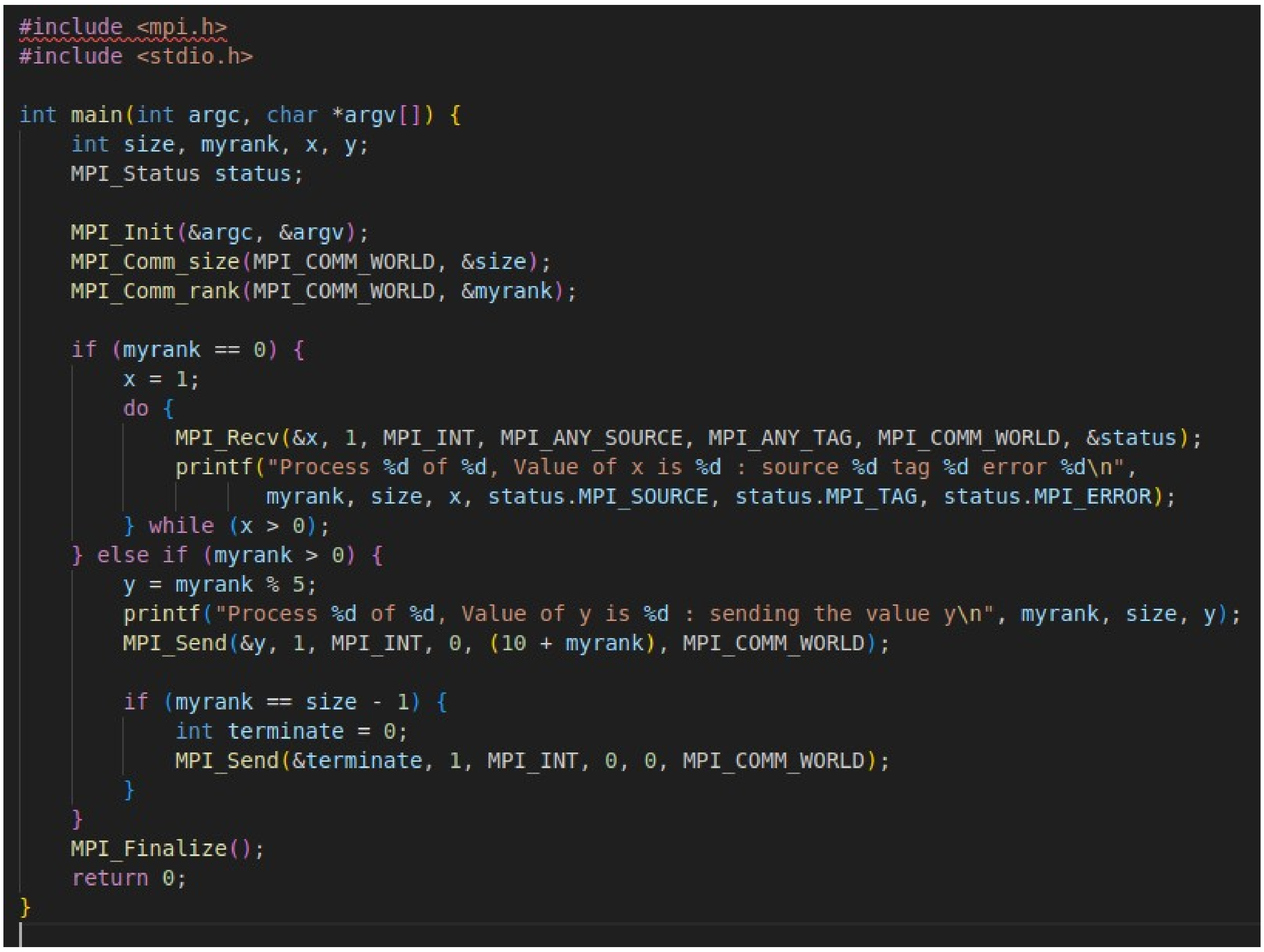
It uses the `MPI_Send` function to transmit the entire array, specifying the number of elements to send. Conversely, Process 1 waits to receive this data using `MPI_Recv`.

Upon successful reception, it prints the received array and also provides details about the source of the message and its tag through the `MPI_Status` object.

After the communication is complete, both processes call `MPI_Finalize` to clean up the MPI environment and free resources. This example effectively showcases fundamental MPI operations such as message passing, handling of arrays, and inter-process communication.

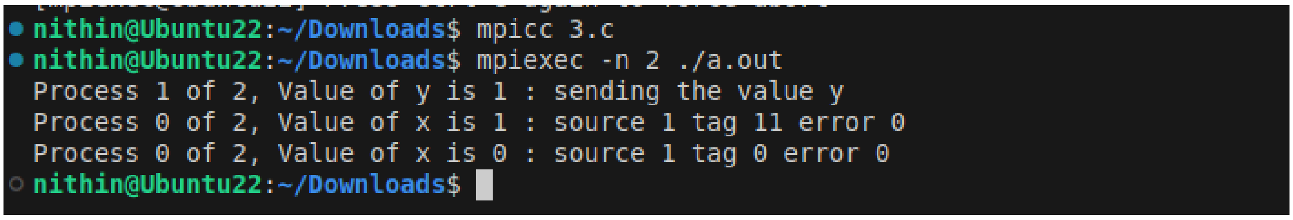
Q3. MPI_Send() and MPI_Recv() with MPI_ANY_SOURCE, MPI_ANY_TAG.
Note down the results and write your observation.

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
Process 1 of 2, Value of y is 1 : sending the value y
Process 0 of 2, Value of x is 1 : source 1 tag 11 error 0
Process 0 of 2, Value of x is 0 : source 1 tag 0 error 0
nithin@Ubuntu22:~/Downloads$
```

Analysis

This MPI code illustrates a simple implementation of a master-worker pattern where multiple processes send data to a designated master process (rank 0). It begins by including the necessary MPI and standard I/O headers. In the `main` function, the MPI environment is initialized, and the total number of processes and the rank of each process are determined.

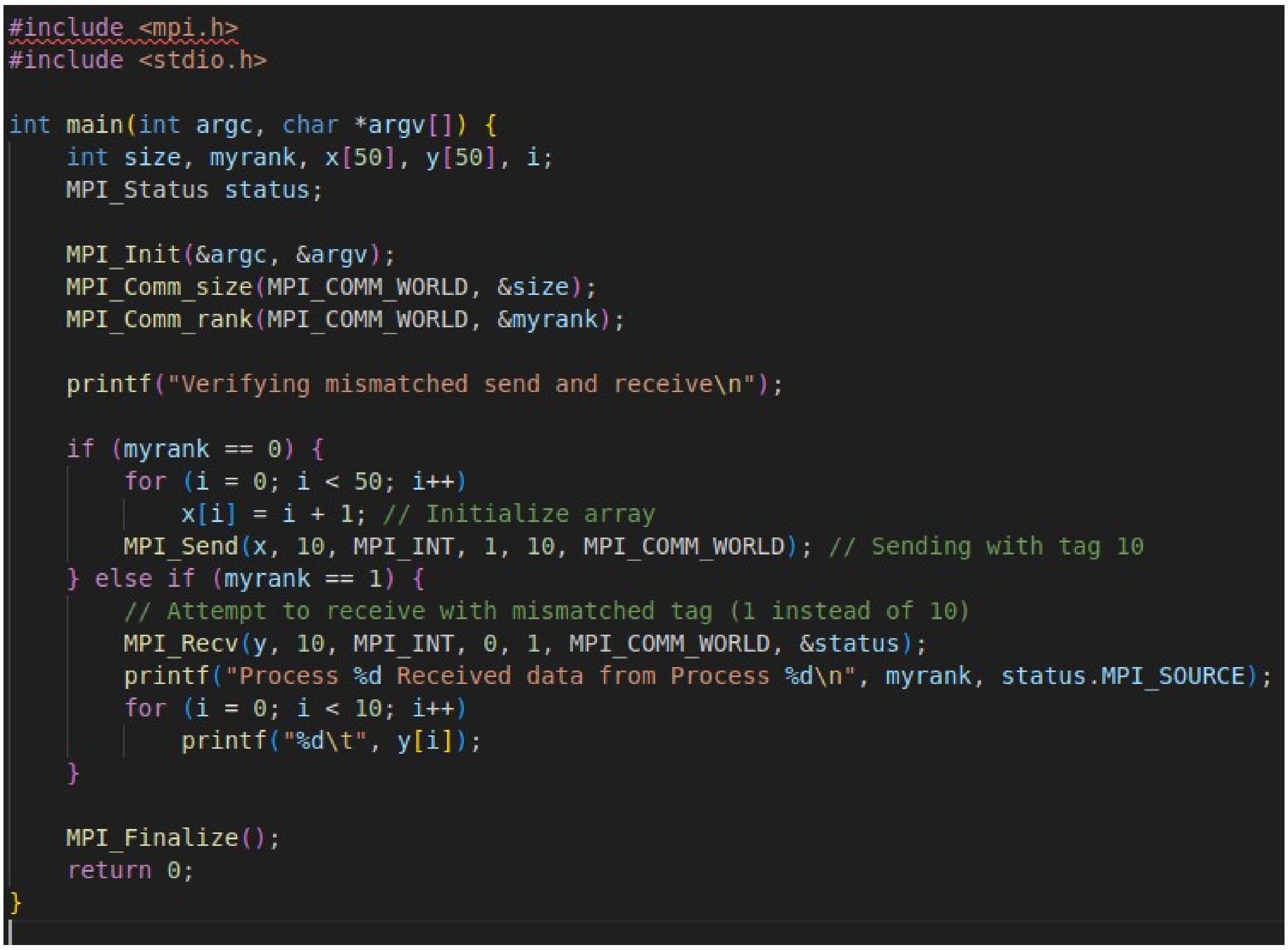
Process 0, serving as the master, initializes a variable `x` to 1 and enters a loop where it listens for incoming messages using `MPI_Recv`. It receives messages from any source and processes them until it receives a termination signal (when `x` is less than or equal to 0). Each time it receives a value, it prints the rank, total processes, the received value of `x`, the source of the message, and the message tag.

For all other processes (ranks greater than 0), the code calculates a value `y` based on the rank. Each process prints its rank and the value of `y` before sending this value to Process 0 using `MPI_Send`, along with a unique tag based on its rank. Additionally, the last process in the rank order sends a termination signal to indicate that no further data will be sent.

The program concludes by calling `MPI_Finalize` to clean up the MPI environment. This code effectively demonstrates key MPI concepts such as message passing, process communication, and termination signaling, highlighting the coordination between a master process and multiple worker processes.

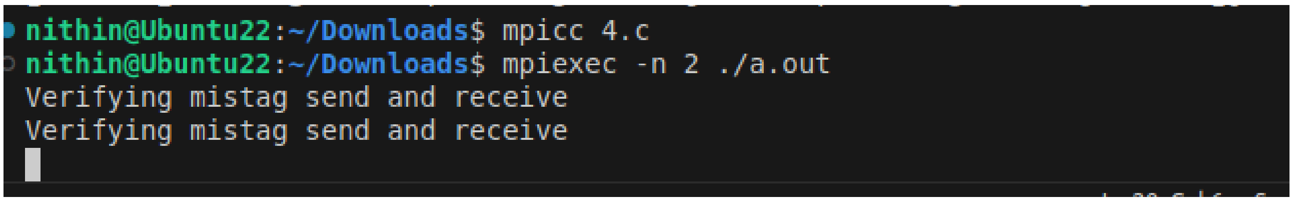
Q4. MPI_Send() and MPI_Recv() with mismatched tag. Record the result for mismatched tag and also after correcting tag value of send receive as same number.

Code



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

int main(int argc, char *argv[]) {
    int size, myrank, x[50], y[50], i;
    MPI_Status status;

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

    printf("Verifying mismatched send and receive\n");

    if (myrank == 0) {
        for (i = 0; i < 50; i++)
            x[i] = i + 1; // Initialize array
        MPI_Send(x, 10, MPI_INT, 1, 10, MPI_COMM_WORLD); // Sending with tag 10
    } else if (myrank == 1) {
        // Attempt to receive with mismatched tag (1 instead of 10)
        MPI_Recv(y, 10, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);
        printf("Process %d Received data from Process %d\n", myrank, status.MPI_SOURCE);
        for (i = 0; i < 10; i++)
            printf("%d\t", y[i]);
    }

    MPI_Finalize();
    return 0;
}
```

Output



```
nithin@Ubuntu22:~/Downloads$ mpicc 4.c
nithin@Ubuntu22:~/Downloads$ mpiexec -n 2 ./a.out
Verifying mistag send and receive
Verifying mistag send and receive
```

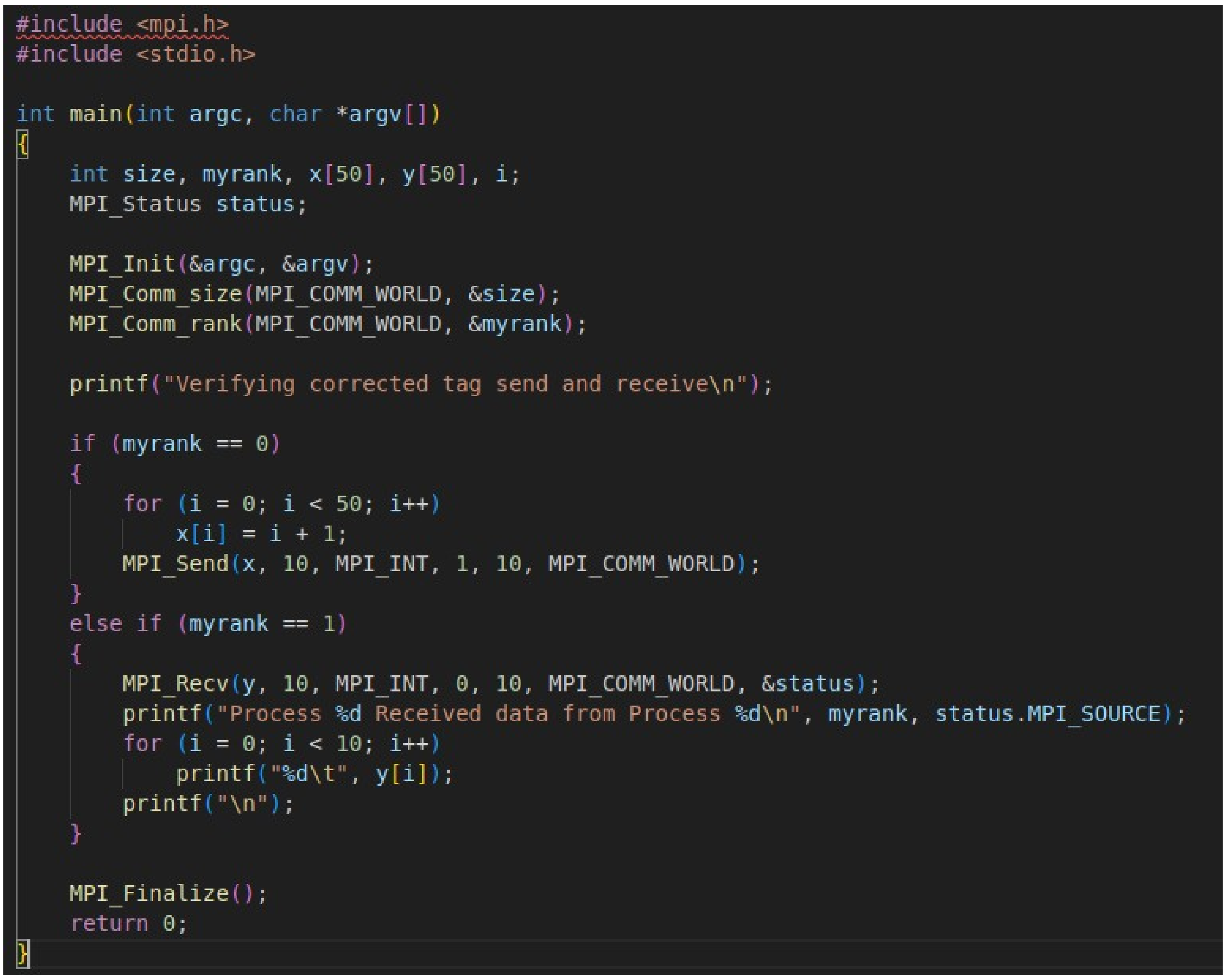
Analysis

In the current code, the tags are mismatched:

Process 0 sends with tag 10: MPI_Send(x, 10, MPI_INT, 1, 10, MPI_COMM_WORLD);
Process 1 receives with tag 1: MPI_Recv(y, 10, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);

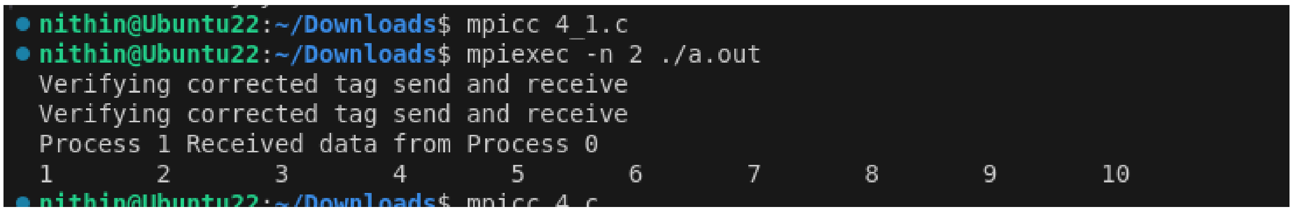
The program will **hang indefinitely**. Process 1 will wait for a message with tag 1, which Process 0 never sends. This is because MPI_Recv is a blocking call, and it won't return until it receives a message with the specified tag.

Corrected Code



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

int main(int argc, char *argv[])
{
    int size, myrank, x[50], y[50], i;
    MPI_Status status;

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

    printf("Verifying corrected tag send and receive\n");

    if (myrank == 0)
    {
        for (i = 0; i < 50; i++)
            x[i] = i + 1;
        MPI_Send(x, 10, MPI_INT, 1, 10, MPI_COMM_WORLD);
    }
    else if (myrank == 1)
    {
        MPI_Recv(y, 10, MPI_INT, 0, 10, MPI_COMM_WORLD, &status);
        printf("Process %d Received data from Process %d\n", myrank, status.MPI_SOURCE);
        for (i = 0; i < 10; i++)
            printf("%d\t", y[i]);
        printf("\n");
    }

    MPI_Finalize();
    return 0;
}
```

Output



```
nithin@Ubuntu22:~/Downloads$ mpicc 4_1.c
nithin@Ubuntu22:~/Downloads$ mpiexec -n 2 ./a.out
Verifying corrected tag send and receive
Verifying corrected tag send and receive
Process 1 Received data from Process 0
1       2       3       4       5       6       7       8       9       10
```

Analysis

After correcting tag values:
To fix this, we need to make the tags match. Let's change both to use tag 10

Result after correcting tag values:

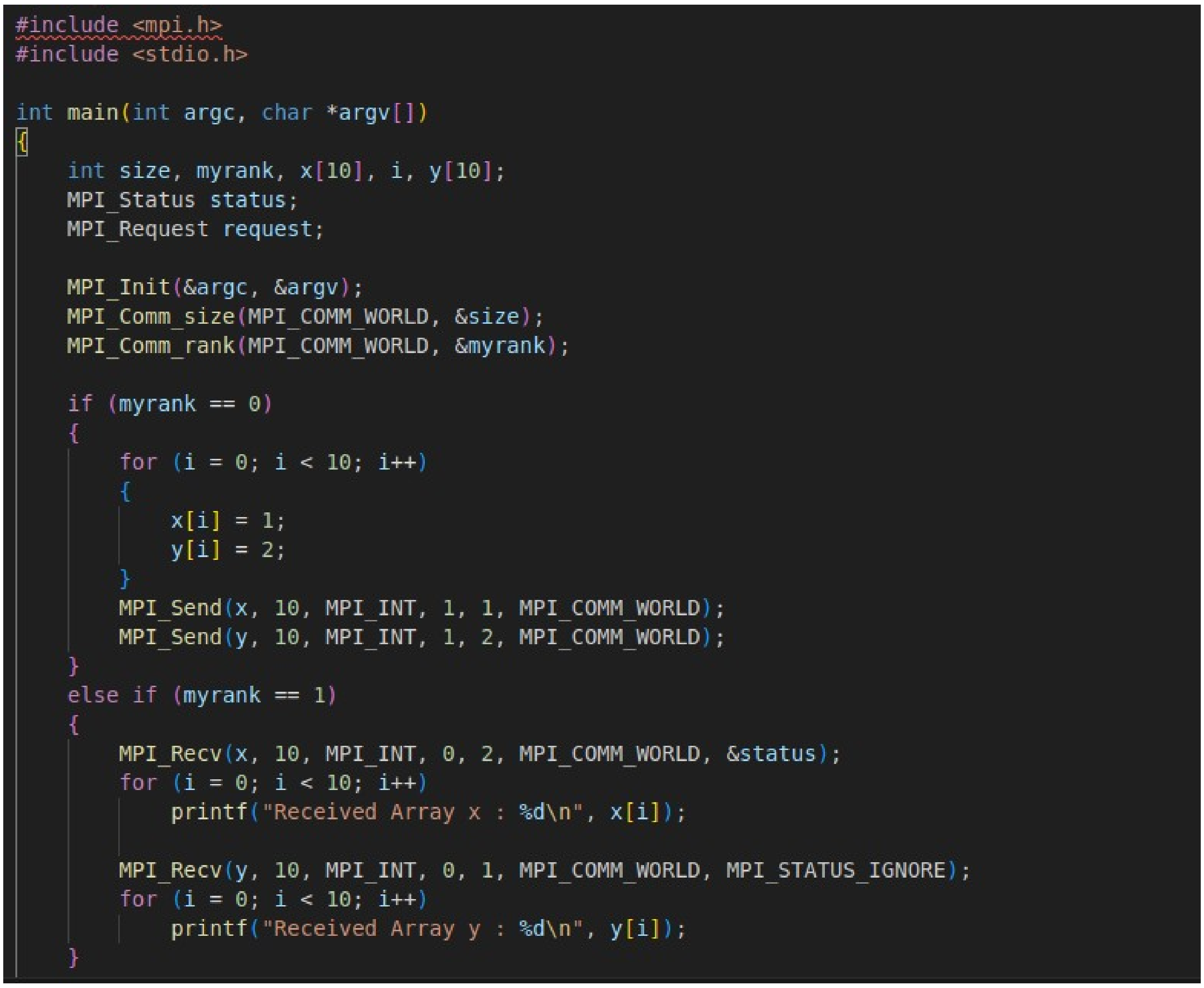
The program will run successfully.
Process 0 will send the first 10 elements of array x to Process 1.
Process 1 will receive the data and print it.

In summary
With mismatched tags: The program hangs due to Process 1 waiting for a message with tag 1 that never arrives.
After correcting tags: The program runs successfully, with Process 1 receiving and printing the first 10 integers sent by Process 0.

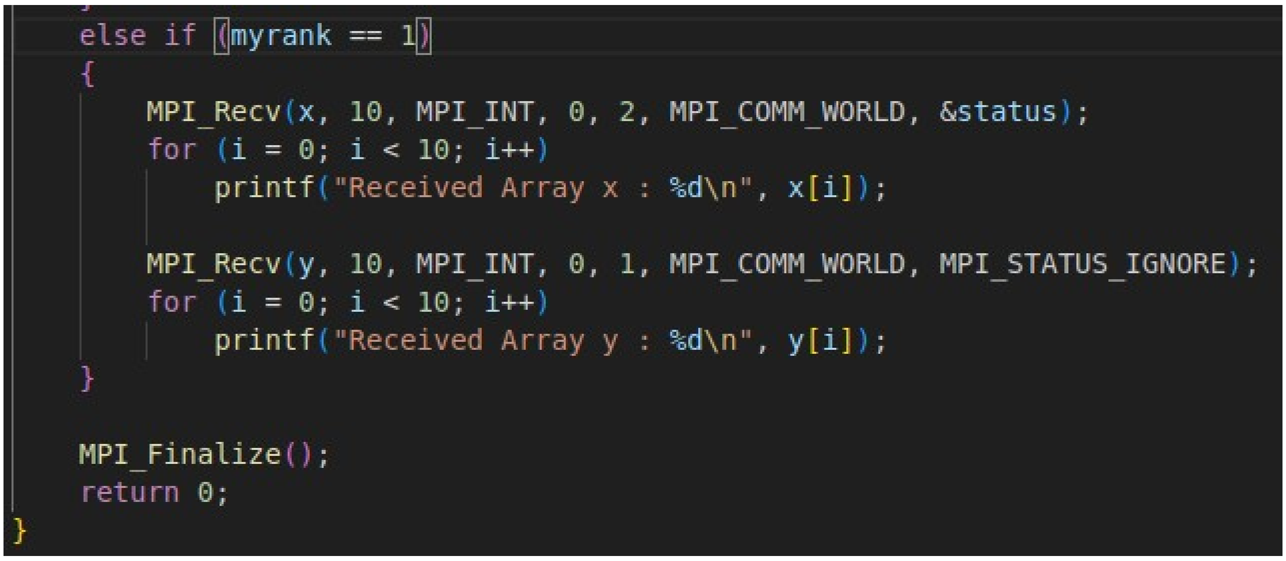
This example demonstrates the importance of matching tags in MPI communication to ensure proper message passing between processes.

Q5. MPI Program 5: MPI_Send() and MPI_Recv() standard mode: Note down your observation on the content of x and y at Process 1 and explain the importance of tag.

Code

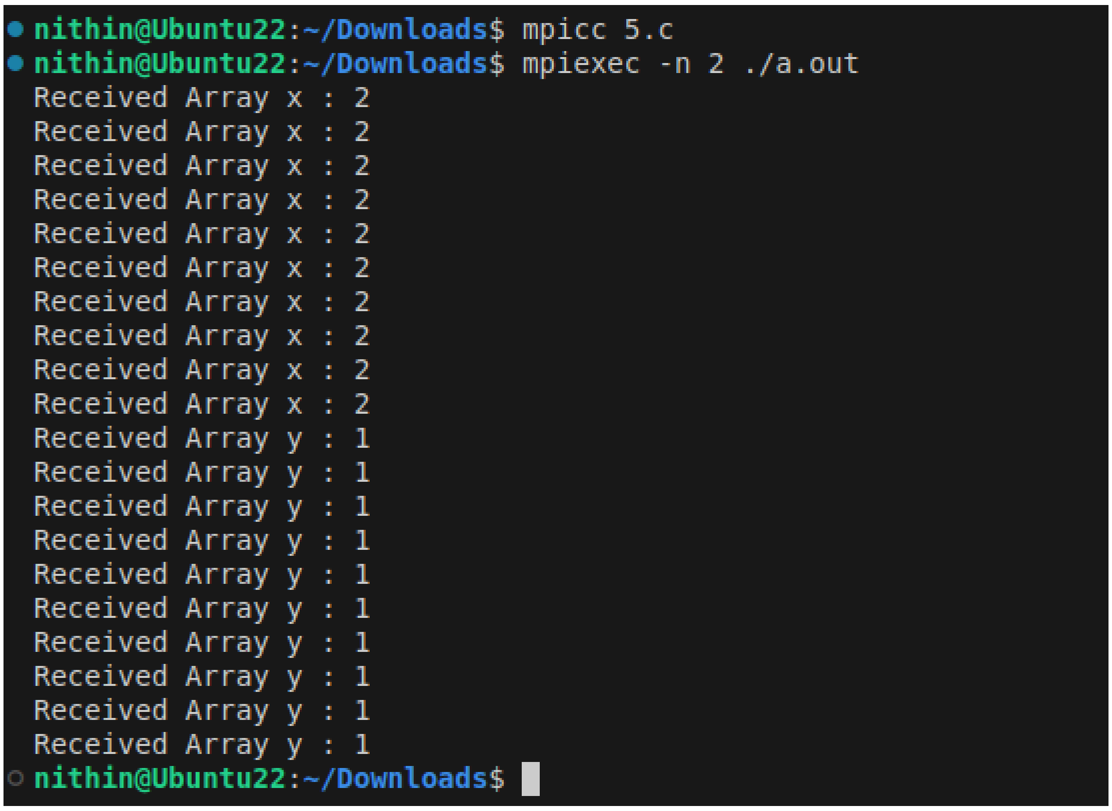


```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int size, myrank, x[10], i, y[10];
    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)
    {
        for (i = 0; i < 10; i++)
        {
            x[i] = 1;
            y[i] = 2;
        }
        MPI_Send(x, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
        MPI_Send(y, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
    }
    else if (myrank == 1)
    {
        MPI_Recv(x, 10, MPI_INT, 0, 2, MPI_COMM_WORLD, &status);
        for (i = 0; i < 10; i++)
            printf("Received Array x : %d\n", x[i]);

        MPI_Recv(y, 10, MPI_INT, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        for (i = 0; i < 10; i++)
            printf("Received Array y : %d\n", y[i]);
    }
```



```c
    else if (myrank == 1)
    {
        MPI_Recv(x, 10, MPI_INT, 0, 2, MPI_COMM_WORLD, &status);
        for (i = 0; i < 10; i++)
            printf("Received Array x : %d\n", x[i]);

        MPI_Recv(y, 10, MPI_INT, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        for (i = 0; i < 10; i++)
            printf("Received Array y : %d\n", y[i]);
    }

    MPI_Finalize();
    return 0;
}
```

Output



```
nithin@Ubuntu22:~/Downloads$ mpicc 5.c
nithin@Ubuntu22:~/Downloads$ mpiexec -n 2 ./a.out
Received Array x : 2
Received Array x : 2
Received Array x : 2
Received Array x : 2
Received Array x : 2
Received Array x : 2
Received Array x : 2
Received Array x : 2
Received Array x : 2
Received Array x : 2
Received Array y : 1
Received Array y : 1
Received Array y : 1
Received Array y : 1
Received Array y : 1
Received Array y : 1
Received Array y : 1
Received Array y : 1
Received Array y : 1
Received Array y : 1
nithin@Ubuntu22:~/Downloads$
```

Analysis

This MPI code demonstrates the transfer of two integer arrays between two processes in a parallel computing setup. It starts by including the necessary headers for MPI and standard I/O. Inside the `main` function, the MPI environment is initialized, and both the total number of processes and the rank of each process are determined.

Process 0, the sender, initializes two arrays, `x` and `y`, each containing ten elements. It fills array `x` with the value 1 and array `y` with the value 2. After populating these arrays, Process 0 sends them to Process 1 using `MPI_Send`, specifying unique tags for each array to differentiate the messages.

Process 1, the receiver, first waits for the array `y` using `MPI_Recv`, and once received, it prints the contents of array `x`. It then receives array `x` using another `MPI_Recv` call, and subsequently prints the contents of array `y`. The use of different tags allows Process 1 to correctly identify which array it is receiving at each step.

Finally, both processes call `MPI_Finalize` to clean up the MPI environment. This code effectively showcases essential MPI operations such as sending and receiving multiple messages, handling arrays, and utilizing tags for message differentiation. By executing this program with two processes, users can observe how data can be exchanged efficiently in parallel applications, making it a valuable example for understanding inter-process communication in MPI.

Importance of Tag here :

Tags are essential for managing the communication between processes effectively. Here’s how tags are important in this specific example:

1. Message Identification:
- Process 0 sends two different arrays (`x` and `y`) to Process 1, each using a distinct tag (1 for array `y` and 2 for array `x`). This allows Process 1 to identify which message it is receiving at any given time. When Process 1 calls `MPI_Recv`, it specifies the expected tag for the message it is trying to receive, ensuring that it processes the messages in the correct order.

2. Order of Reception:
- In the code, the order of sending and receiving is important. By using tags, Process 1 can first receive array `y` and then array `x`. The tags help maintain this order, making the code more readable and ensuring that the receiver knows what to expect when it receives a message.

3. Flexibility in Communication:
- Tags provide flexibility, allowing the same receiving process to handle multiple types of messages. In more complex applications, a single process might need to receive various data types or messages from multiple sources. By using different tags, it can selectively process messages based on their type or purpose.

4. Error Handling:

- Using tags can also help with error handling. If a process receives a message with an unexpected tag, it can handle this scenario appropriately, either by ignoring the message, logging an error, or taking corrective actions.

5. Debugging and Maintenance:
- Tags improve the clarity of communication in MPI programs. When debugging, it’s easier to track which messages are sent and received based on their tags, helping developers understand the flow of data and identify issues.

In summary, in the provided MPI code, tags are crucial for clearly distinguishing between the two arrays being communicated between processes, ensuring that messages are received and processed correctly, and providing flexibility and robustness in inter-process communication.