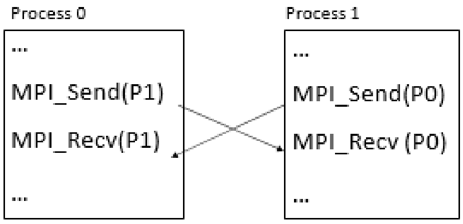
Lab Practice -II (19-12-2024)

1. Implement a MPI Program for deadlock .

Deadlocks

- A deadlock occurs when two or more processors try to access the same set of resources
- Deadlocks are possible in blocking communication
    - Example: Two processors initiate a blocking send to each other without posting a receive



43

a. Is it a deadlocked program? How?

```
#include <mpi.h>
void main (int argc, char **argv) {
int myrank;
MPI_Status status;
double a[100], b[100];
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if( myrank ==0){

MPI_Recv( b, 100, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD, &status );
MPI_Send( a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
//MPI_Recv( b, 100, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD, &status );

}
else if( myrank ==1){
MPI_Recv( b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD, &status );
MPI_Send( a, 100, MPI_DOUBLE, 0, 18, MPI_COMM_WORLD );
}
MPI_Finalize();
}
```

b.

```c
#include <stdio.h>
#include "mpi.h"
#define MSGLEN 2048         /* length of message in elements */
#define TAG_A 100
#define TAG_B 200

int main(int argc, char** argv )
{
  float message1 [MSGLEN],    /* message buffers                      */
        message2 [MSGLEN];
  int rank,                   /* rank of task in communicator          */
      dest, source,           /* rank in communicator of destination  */
                              /* and source tasks                      */
      send_tag, recv_tag,     /* message tags                          */
      i;
  MPI_Status status;          /* status of communication               */

  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  printf ( " Task %d initialized\n", rank );

  /* initialize message buffers */
  for ( i=0; i<MSGLEN; i++ )  {
    message1[i] = 100;
    message2[i] = -100;
  }

  /* ---------------------------------------------------------------
   * each task sets its message tags for the send and receive, plus
   * the destination for the send, and the source for the receive
   * --------------------------------------------------------------- */
  if ( rank == 0 )  {
    dest = 1;
    source = 1;
    send_tag = TAG_A;
    recv_tag = TAG_B;
  }
  else if ( rank == 1)  {
    dest = 0;
    source = 0;
    send_tag = TAG_B;
    recv_tag = TAG_A;
  }

  /* ---------------------------------------------------------------
   * send and receive messages
   * --------------------------------------------------------------- */
  printf ( " Task %d has sent the message\n", rank );
  MPI_Send ( message1, MSGLEN, MPI_FLOAT, dest, send_tag, MPI_COMM_WORLD );
  MPI_Recv ( message2, MSGLEN, MPI_FLOAT, source, recv_tag, MPI_COMM_WORLD,
             &status );
  printf ( " Task %d has received the message\n", rank );

  MPI_Finalize();
  return 0;
}
```

2. MPI program for Synchronous Send

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

int main(int argc, char *argv[])
{
    int rank, size, i;
    int buffer[10];
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (size < 2)
    {
        printf("Please run with two processes.\n");fflush(stdout);
        MPI_Finalize();
        return 0;
    }
    if (rank == 0)
    {
        for (i=0; i<10; i++)
            buffer[i] = i;
        MPI_Ssend(buffer, 10, MPI_INT, 1, 123, MPI_COMM_WORLD);
    }
    if (rank == 1)
    {
        for (i=0; i<10; i++)
            buffer[i] = -1;
        MPI_Recv(buffer, 10, MPI_INT, 0, 123, MPI_COMM_WORLD, &status);

                for (i=0; i<10; i++)
        {
                        printf("%d \t",buffer[i]);
            if (buffer[i] != i)
                printf("Error: buffer[%d] = %d but is expected to be %d\n", i, buffer[i], i);
        }
        fflush(stdout);
    }
    MPI_Finalize();
    return 0;
}
```

3. MPI Program for Isend and I Recv

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


///////////
//MPI_Isend
///////////
//
// int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag,
//              MPI_Comm comm, MPI_Request *request)
//
// This example uses MPI_Isend to do a non-blocking send of information from the root process to a destination process.
// The destination process is set as a variable in the code and must be less than the number of processes started.
//
// example usage:
//              compile: mpicc -o mpi_isend mpi_isend.c
//              run: mpirun -n 4 mpi_isend
//
int main(argc, argv)
int argc;
char **argv;
{
        int rank, size;
    int tag, destination, count;
    int buffer; //value to send

    tag = 1234;
    destination = 2; //destination process
    count = 1; //number of elements in buffer

    MPI_Status status;
    MPI_Request request = MPI_REQUEST_NULL;

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &size); //number of processes
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); //rank of current process

    if (destination >= size) {
        MPI_Abort(MPI_COMM_WORLD, 1); // destination process must be under the number of processes created, otherwise abort
    }

    if (rank == 0) {
        printf("Enter a value to send to processor %d:\n", destination);
        scanf("%d", &buffer);
        MPI_Isend(&buffer, count, MPI_INT, destination, tag, MPI_COMM_WORLD, &request); //non blocking send to destination process
printf("hello");
    }
```

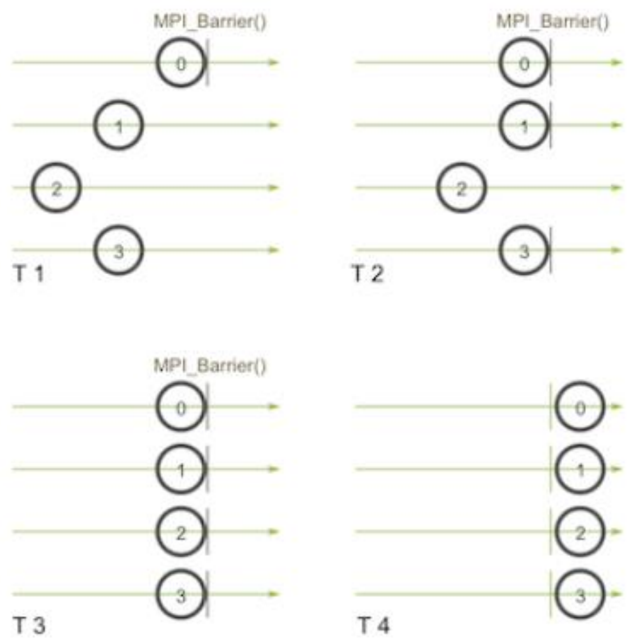
```
    if (rank == destination) {
        MPI_Irecv(&buffer, count, MPI_INT, 0, tag, MPI_COMM_WORLD, &request); //destination process receives
printf("done");
}

    MPI_Wait(&request, &status); //bloks and waits for destination process to receive data

    if (rank == 0) {
        printf("processor %d sent %d\n", rank, buffer);
    }
    if (rank == destination) {
        printf("processor %d got %d\n", rank, buffer);
    }

    MPI_Finalize();

        return 0;
}
```

4. MPI Collective Communication:
   Collective communication is a method of communication which involves participation of **all** processes in a communicator.

   Processor Synchrony (Barrier Synchronization)

   MPI_Barrier is to synchronize a program so that portions of the parallel code can be timed accurately.

```
MPI_Barrier(MPI_Comm communicator)
```



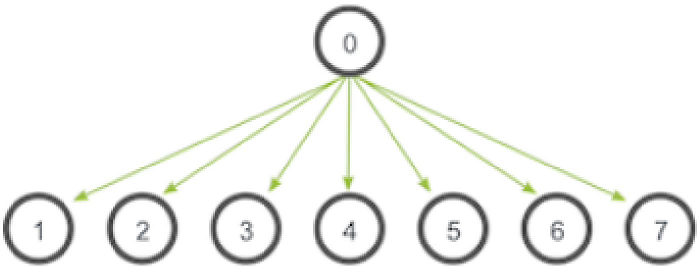

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

/**
 * @brief Illustrates how to use an MPI barrier.
 **/
int main(int argc, char* argv[])
{
    MPI_Init(&argc, &argv);

    // Get my rank
    int my_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    printf("[MPI process %d] I start waiting on the barrier.\n", my_rank);
    MPI_Barrier(MPI_COMM_WORLD);
    printf("[MPI process %d] I know all MPI processes have waited on the barrier.\n", my_rank);

    MPI_Finalize();

    return EXIT_SUCCESS;
}
```

5. Predict the output for the below MPI program

```
int token;
if (world_rank != 0) {
    MPI_Recv(&token, 1, MPI_INT, world_rank - 1, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Process %d received token %d from process %d\n",
           world_rank, token, world_rank - 1);
} else {
    // Set the token's value if you are process 0
    token = -1;
}
MPI_Send(&token, 1, MPI_INT, (world_rank + 1) % world_size,
         0, MPI_COMM_WORLD);

// Now process 0 can receive from the last process.
if (world_rank == 0) {
    MPI_Recv(&token, 1, MPI_INT, world_size - 1, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Process %d received token %d from process %d\n",
           world_rank, token, world_size - 1);
}
```

6. **<u>Broadcasting with MPI_Bcast</u>**

The communication pattern of a broadcast looks like this:




```
MPI_Bcast(
    void* data,
    int count,
    MPI_Datatype datatype,
    int root,
    MPI_Comm communicator)
```

7. Understand the code and execute and trace the output

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

void my_bcast(void* data, int count, MPI_Datatype datatype, int root,
              MPI_Comm communicator) {
  int world_rank;
  MPI_Comm_rank(communicator, &world_rank);
  int world_size;
  MPI_Comm_size(communicator, &world_size);

  printf("\n Inside mybcast : %d  %d \n",world_rank,world_size);

  if (world_rank == root) {
    // If we are the root process, send our data to everyone
    int i;
    for (i = 0; i < world_size; i++) {
      if (i != world_rank) {
        MPI_Send(data, count, datatype, i, 0, communicator);
      }
    }
  } else {
    // If we are a receiver process, receive the data from the root
    MPI_Recv(data, count, datatype, root, 0, communicator, MPI_STATUS_IGNORE);
  }
}

int main(int argc, char** argv) {

  int num_elements = 100;
  int num_trials = 10;

  MPI_Init(NULL, NULL);

  int world_rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

  printf("\n inside main: %d\n",world_rank);
  double total_my_bcast_time = 0.0;
  double total_mpi_bcast_time = 0.0;
  int i;
  int* data = (int*)malloc(sizeof(int) * num_elements);
  assert(data != NULL);

  for (i = 0; i < num_trials; i++) {
    // Time my_bcast
    // Synchronize before starting timing
    MPI_Barrier(MPI_COMM_WORLD);
    total_my_bcast_time -= MPI_Wtime();
    my_bcast(data, num_elements, MPI_INT, 0, MPI_COMM_WORLD);
    // Synchronize again before obtaining final time
    MPI_Barrier(MPI_COMM_WORLD);
    total_my_bcast_time += MPI_Wtime();

    // Time MPI_Bcast
    MPI_Barrier(MPI_COMM_WORLD);
    total_mpi_bcast_time -= MPI_Wtime();
    MPI_Bcast(data, num_elements, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Barrier(MPI_COMM_WORLD);
    total_mpi_bcast_time += MPI_Wtime();
  }

  // Print off timing information
  if (world_rank == 0) {
    printf("Data size = %d, Trials = %d\n", num_elements * (int)sizeof(int),
           num_trials);
    printf("Avg my_bcast time = %lf\n", total_my_bcast_time / num_trials);
    printf("Avg MPI_Bcast time = %lf\n", total_mpi_bcast_time / num_trials);
  }

  free(data);
  MPI_Finalize();
}
```