1 MPI Hands-On; Collective Communications I

1. **Broadcasting an integer value to all of the MPI processes**, A program code10.c that reads an integer value from the terminal and distributes the value to all of the MPI processes.

    - Each process should print out its rank and the value it received. Values should be read until a negative integer is given as input.
    - You may find it helpful to include a **fflush(stdout)** to the code; after the **printf** calls in your program. Without this, output may not appear when you expect it.

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

int main( int argc, char **argv )
{
  int rank, value;
  MPI_Init( &argc, &argv );

  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  do {
    if (rank == 0)
      scanf( "%d", &value );

    MPI_Bcast( &value, 1, MPI_INT, 0, MPI_COMM_WORLD );

    printf( "Process %d got %d\n", rank, value );
    fflush(stdout);

  } while (value >= 0);

  MPI_Finalize( );
  return 0;
}
```

2. **Broadcasting the name of the master process**, A program code11.c that first broadcasts the name of the master process then each nodes send hello messages to master node.

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

#define TRUE 1
#define FALSE 0
#define MASTER_RANK 0

int main( int argc, char *argv[] )
{
  int count, pool_size, my_rank, my_name_length, i_am_the_master =
    FALSE;
  char my_name[BUFSIZ/2], master_name[BUFSIZ/2], send_buffer[2*BUFSIZ
    ],
    recv_buffer[2*BUFSIZ];
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &pool_size);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Get_processor_name(my_name, &my_name_length);

  if (my_rank == MASTER_RANK) {
    i_am_the_master = TRUE;
    strcpy (master_name, my_name);
  }

  MPI_Bcast(master_name, BUFSIZ, MPI_CHAR, MASTER_RANK, MPI_COMM_WORLD
    );

  sprintf(send_buffer, "hello %s, greetings from %s, rank = %d",
    master_name, my_name, my_rank);
  MPI_Send (send_buffer, strlen(send_buffer) + 1, MPI_CHAR,
      MASTER_RANK, 0, MPI_COMM_WORLD);

  if (i_am_the_master) {
    for (count = 1; count <= pool_size; count++) {
      MPI_Recv (recv_buffer, BUFSIZ, MPI_CHAR, MPI_ANY_SOURCE,
    MPI_ANY_TAG,
    MPI_COMM_WORLD, &status);
      printf ("%s\n", recv_buffer);
    }
  }

  MPI_Finalize();
}
```

3. **Computation of PI number with collective communications**. This example code12.c evaluates π by numerically evaluating the integral

$$\int_0^1 \frac{1}{1+x^2}dx = \frac{\pi}{4}$$

This code computes PI (with a very simple method) but does not use **MPI_Send** and **MPI_Recv**. Instead, it uses *collective* operations to send data to and from all of the running processes.

- The routine **MPI_Bcast** sends data from one process to all others.
- The routine **MPI_Reduce** combines data from all processes (by adding them in this case), and returning the result to a single process.

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

int main( int argc, char *argv[] )
{
  int done = 0, n, myid, numprocs, i, rc;
  double PI25DT = 3.141592653589793238462643;
  double mypi, pi, h, sum, x, a;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
  while (!done)
  {
    if (myid == 0) {
        printf("Enter the number of intervals: (0 quits) ");
        scanf("%d",&n);
    }
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if (n == 0) break;

    h   = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs) {
        x = h * ((double)i - 0.5);
        sum += 4.0 / (1.0 + x*x);
    }
    mypi = h * sum;

    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    if (myid == 0)
        printf("pi is approximately %.16f, Error is %.16f\n",
               pi, fabs(pi - PI25DT));
  }
  MPI_Finalize();
}
```