Dated :
Assessment No. : 7

Function: MPI_Reduce ()

SCENARIO – I

Consider the following descriptive flow that break up the interval [A,B] into N subintervals, evaluate F(X) at the midpoint of each subinterval, and divide the sum of these values by N to get an estimate for the integral. Implement its logical structure Using MPI.

Assume, we have M processes available with MPI, then we can ask processes 0, 1, 2, ... M-1 to handle the subintervals in the following order:

```
0      1       2                    M-1  <-- Process numbers begin at 0
------ ------  ------  -----  ------
   1      2       3       ...       M
   M+1    M+2     M+3     ...     2*M
   2*M+1  2*M+2   2*M+3   ...     3*M
```

and so on up to subinterval N. The partial sums collected by each process are then sent to the master process to be added together to get the estimated integral.

Descriptive Flow:

```
*/
{
  double a = 0.0;
  double b = 1.0;
  int i;
  int id;
  int ierr;
  int n;
  int n_part;
  int p;
  double q;
  double q_diff;
  double q_exact = 2.0;
  double q_part;
  double wtime;
  double x;

/*
 Step1     : Initialize MPI.
*/

/*
Step2      : Get the number of processes.
*/
/*

  Step3    : Determine this processes's rank.
*/

/*
 Step4     : Record the starting time.
*/
```

Dated :
Assessment No. : 7

Function: MPI_Reduce ()

```
/*
  Step5  : The master process broadcasts, and the other processes receive,
  the number of intervals N.
*/

/*
Step6  :  Every process integrates F(X) over a subinterval determined by its process ID.
*/

  q_part = 0.0;
  n_part = 0;

  for ( i = id + 1; i <= n; i = i + p )
  {
    x = ( ( double ) ( 2 * n - 2 * i + 1 ) * a
        + ( double ) (       2 * i - 1 ) * b )
       / ( double ) ( 2 * n     );

    n_part = n_part + 1;
    q_part = q_part + f ( x );
  }

/*
Step7  : Each process sends its local result Q_PART to the MASTER process,
  to be added to the global result QI.
*/

/*
Step8  : Every process scales the sum and reports the results.
*/

/*
Step9  : Shut down MPI.
*/

}
/******************************************************************************/

double f ( double x )

/******************************************************************************/
/*
Purpose:

    F evaluates the function F(X) which we are integrating.

Parameters:

    Input, double X, the point at which to evaluate F.

    Output, double F, the value of F(X).
*/
{
  double value;
```

Dated :
Assessment No. : 7

Function: MPI_Reduce ()

```
 if ( x <= 0.0 )
 {
   value = 0.0;
 }
 else
 {
   value = 1.0 / sqrt ( x );
 }
 return value;
}
```

1. Implement in Visual Studio / Debian OS
2. Build and Execute Its Logical Scenario
3. Depict the screenshots along with proper justification

Linux Execution

To compile this code, type:

mpicc -o simple1 mpi_sample1.c.c //C Program

Or

mpif77 -o simple1 mpi_sample1.c.f // Fortran Program

To run this compiled code, type:

mpirun -np 4 simple1

MS Studio Execution

1. Build the Code Using Menu Option
2. Run the program on the command line

   mpiexec $\SomeDirectory mpi_sample1.c.exe

Dated :
Assessment No. : 7

Function: MPI_Reduce ()

SCENARIO – II

Consider the following program, called mpi_sample2.c. Most parallel codes assign different tasks to different processors.

For example, parts of an input data set might be divided and processed by different processors, or a finite difference grid might be divided among the processors available. This means that the code needs to identify processors. In this example, processors are identified by rank - an integer from 0 to total number of processors - 1.

```
#include < mpi.h>   /* PROVIDES THE BASIC MPI DEFINITION AND TYPES */
#include  < stdio.h>

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

  int my_rank;
  int size;
  MPI_Init(&argc, &argv); /*START MPI */

/*DETERMINE RANK OF THIS PROCESSOR*/
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

 /*DETERMINE TOTAL NUMBER OF PROCESSORS*/
  MPI_Comm_size(MPI_COMM_WORLD, &size);


  printf("Hello world! I'm rank (processor number) %d of size %d\n", my_rank, size);

  MPI_Finalize();  /* EXIT MPI */

}
```

1. Implement the above code in Microsoft Visual Studio

Dated :
Assessment No. : 7

Function: MPI_Reduce ()

2. Build and Execute Its Logical Scenario
3. Depict the screenshots along with proper justification

SCENARIO – III

Write a simple MPI program to print a word *'Greetings'* the processes. In order to explore its practical use, you are advised to read and understand the following statements.

1. MPI provides a set of send and receive functions that allow the communication of typed data with an associated tag.
2. Typing of the message contents is necessary for heterogeneous support - the type information is needed so a correct data representation conversion can be performed as data is sent from one architecture to anothers.
3. The tag allows selectivity of messages at the receiving endone can receive on a particular tag or one can wildcard this quantity allowing reception of messages with any tag.
4. Message selectivity on the source process of the message is also provided.

About Coding

For the example of process 0 sending a message to process 1. The code executes on both process 0 and process 1. Process 0 sends a character string using MPI_Send(). The first three parameters of the send call specify the data to be sent: 1. The outgoing data is to be taken from msg; 2. It consists of strlen (msg) + 1 entries each of type MPI CHAR (The string "Hello

There" contains strlen (msg) =11 significant characters. In addition, we are also sending the '\0' string terminator character. The fourth parameter specifies the message destination which

Dated :
Assessment No. : 7

Function: MPI_Reduce ()

is process 1. The fifth parameter specifies the message tag. Finally the last parameter is a communicator that specifies a communication domain for this communication.

Among other things a communicator serves to define a set of processes that can be contacted. Each such process is labelled by a process rank. Process ranks are integers and are discovered by inquiry to a communicator (MPI_Comm_rank ()). MPI_COMM_WORLD is a default communicator provided upon start-up that defines an initial communication domain for all the processes that participate in the computation.

The receiving process specified that the incoming data was to be placed in msg and that it has a maximum size of 20 entries of type MPI_CHAR. The variable status set by MPI_Recv() gives information on the source and tag of the message and how many elements were actually received. For example, the receiver can examine this variable to find out the actual length of the character string received.

Code Logic:

```
char Msg[20];
int myrank, tag = 99;
MPI_Status status;
….
….
MPI_Comm_rank (MPI_COMM_WORLD, &myrank); /* Find My Rank*/
if (myrank == 0)  {
strcpy(msg, “Hello there”);
MPI_Send( msg, strlen(msg)+1, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
else if (myrank == 1) {
MPI_Recv(msg, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &tatus);
}
```

Dated :
Assessment No. : 7

Function: MPI_Reduce ()