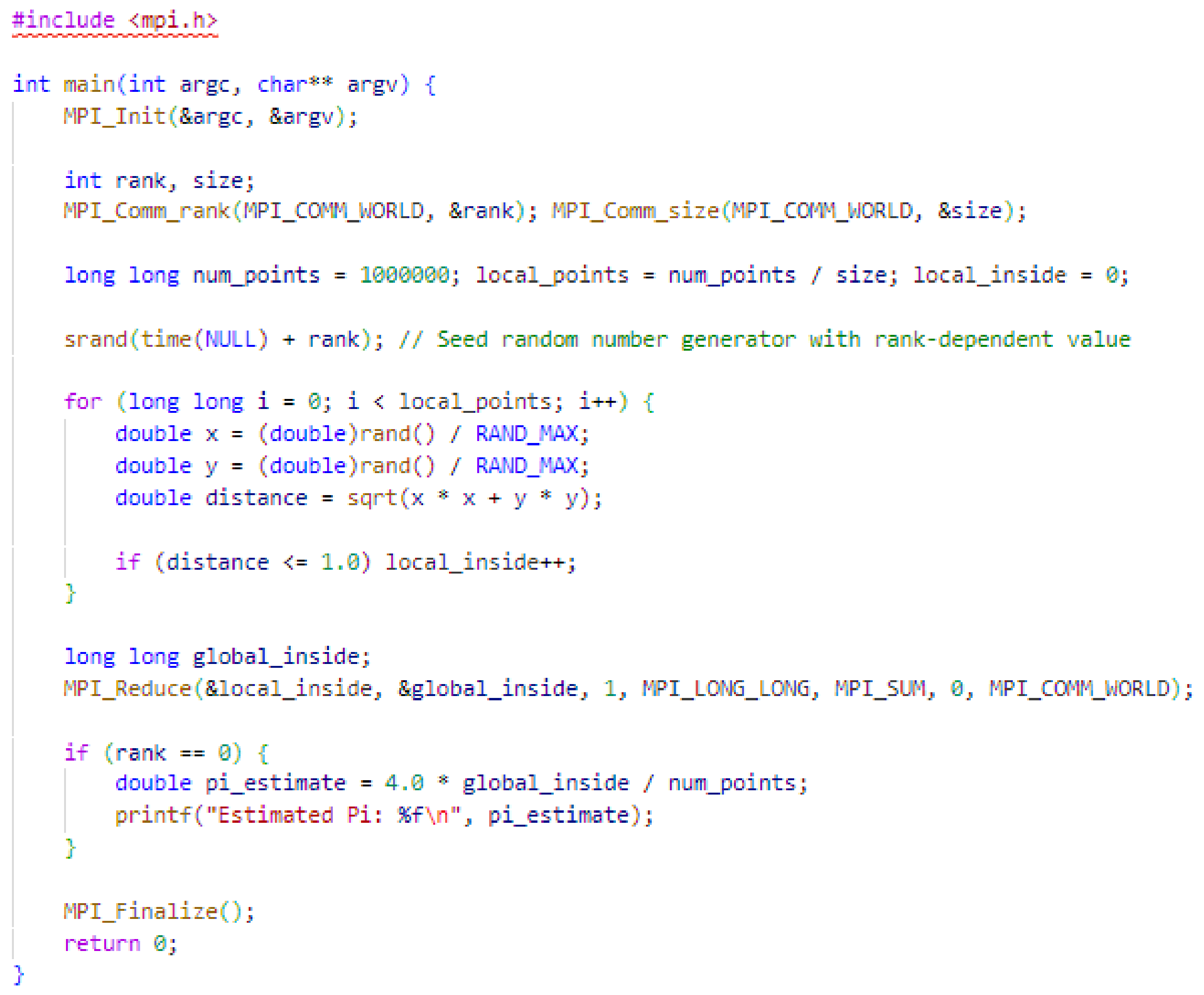
MPI PROGRAMMING

PDC – FALL 2023

Practice with MPI Programming

Estimate the value of Pi by generating random points inside the square and use Pythagorean theorem to determine whether the point lie inside the quarter circle or not.



```c
#include <mpi.h>

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

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); MPI_Comm_size(MPI_COMM_WORLD, &size);

    long long num_points = 1000000; local_points = num_points / size; local_inside = 0;

    srand(time(NULL) + rank); // Seed random number generator with rank-dependent value

    for (long long i = 0; i < local_points; i++) {
        double x = (double)rand() / RAND_MAX;
        double y = (double)rand() / RAND_MAX;
        double distance = sqrt(x * x + y * y);

        if (distance <= 1.0) local_inside++;
    }

    long long global_inside;
    MPI_Reduce(&local_inside, &global_inside, 1, MPI_LONG_LONG, MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        double pi_estimate = 4.0 * global_inside / num_points;
        printf("Estimated Pi: %f\n", pi_estimate);
    }

    MPI_Finalize();
    return 0;
}
```

Program: Write compact readable MPI C code to sum up all the numbers from 1 to 1000.

Use 1000 processes, where process 0 will print the sum.

Note: Explain all MPI functions after the code. Ensure that your code computes the correct sum.

```
#include<iostream.h>
#include<mpi.h>
int main(int argc, char ** argv){
int mynode, totalnodes;
int sum,startval,endval, accum;
MPI_Status status;
MPI_Init(argc,argv);
MPI_Comm_size(MPI_COMM_WORLD, &totalprocs); // get totalprocs
MPI_Comm_rank(MPI_COMM_WORLD, &myid); // get myid
sum = myid + 1; // rank is an integer ranging from 0 to totalprocs-1
if(myid != 0)
MPI_Send(&sum,1,MPI_INT,0,1,MPI_COMM_WORLD);
else
for(int j=1;j =j+1) {
MPI_Recv(&accum, 1, MPI_INT, j, 1, MPI_COMM_WORLD, &status);
sum = sum+accum;
}
if(myid ==0)
        cout<< "The sum from 1 to 1000 is :" << sum << endl;
MPI_Finalize();
}
```

Program for Trapezoid Rule implementation using MPI

```
/* File:     mpi_trap4.c
 * Purpose:  Use MPI to implement a parallel version of the trapezoidal
 *           rule.  This version uses collective communications and
 *           MPI derived datatypes to distribute the input data and
 *           compute the global sum.
 *
 * Input:    The endpoints of the interval of integration and the number
 *           of trapezoids
 * Output:   Estimate of the integral from a to b of f(x)
 *           using the trapezoidal rule and n trapezoids.
 *
 * Compile:  mpicc -g -Wall -o mpi_trap4 mpi_trap4.c
 * Run:      mpiexec -n <number of processes> ./mpi_trap4
 *
 * Algorithm:
 *    1.  Each process calculates "its" interval of
 *        integration.
 *    2.  Each process estimates the integral of f(x)
 *        over its interval using the trapezoidal rule.
 *    3a. Each process != 0 sends its integral to 0.
 *    3b. Process 0 sums the calculations received from
 *        the individual processes and prints the result.
 *
 * Note:  f(x) is all hardwired.
* IPP:   Section 3.5 (pp. 117 and ff.) */
#include <stdio.h>

/* We'll be using MPI routines, definitions, etc. */
#include <mpi.h>

/* Build a derived datatype for distributing the input data */
void Build_mpi_type(double* a_p, double* b_p, int* n_p,
      MPI_Datatype* input_mpi_t_p);


/*--------------------------------------------------------------------
 * Function:     Build_mpi_type
 * Purpose:      Build a derived datatype so that the three
 *               input values can be sent in a single message.
 * Input args:   a_p:  pointer to left endpoint
 *               b_p:  pointer to right endpoint
 *               n_p:  pointer to number of trapezoids
 * Output args:  input_mpi_t_p:  the new MPI datatype
 */
void Build_mpi_type(
      double*        a_p           /* in  */,
      double*        b_p           /* in  */,
      int*           n_p           /* in  */,
      MPI_Datatype*  input_mpi_t_p  /* out */) {
```

```
/* Get the input values */
void Get_input(int my_rank, int comm_sz, double* a_p, double* b_p,
      int* n_p);

/* Calculate local integral  */
double Trap(double left_endpt, double right_endpt, int trap_count,
   double base_len);

/* Function we're integrating */
double f(double x);

int main(void) {
   int my_rank, comm_sz, n, local_n;
   double a, b, dx, local_a, local_b;
   double local_int, total_int;

   /* Let the system do what it needs to start up MPI */
   MPI_Init(NULL, NULL);

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

   /* Find out how many processes are being used */
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

   Get_input(my_rank, comm_sz, &a, &b, &n);

   dx = (b-a)/n;           /* dx is the same for all processes */
   local_n = n/comm_sz;  /* So is the number of trapezoids  */

   /* Length of each process' interval of
    * integration = local_n*dx.  So my interval
    * starts at: */
   local_a = a + my_rank*local_n*dx;
   local_b = local_a + local_n*dx;
   local_int = Trap(local_a, local_b, local_n, dx);


  /* Add up the integrals calculated by each process */
  MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0,
        MPI_COMM_WORLD);

  /* Print the result */
  if (my_rank == 0) {
     printf("With n = %d trapezoids, our estimate\n", n);
     printf("of the integral from %f to %f = %.15e\n",
         a, b, total_int);
  }

  /* Shut down MPI */
  MPI_Finalize();

  return 0;
} /*  main  */
```

```
 int array_of_blocklengths[3] = {1, 1, 1};
 MPI_Datatype array_of_types[3] = {MPI_DOUBLE, MPI_DOUBLE, MPI_INT};
 MPI_Aint a_addr, b_addr, n_addr;
 MPI_Aint array_of_displacements[3] = {0};

 MPI_Get_address(a_p, &a_addr);


   MPI_Get_address(b_p, &b_addr);
   MPI_Get_address(n_p, &n_addr);
   array_of_displacements[1] = b_addr-a_addr;
   array_of_displacements[2] = n_addr-a_addr;
   MPI_Type_create_struct(3, array_of_blocklengths,
         array_of_displacements, array_of_types,
         input_mpi_t_p);
   MPI_Type_commit(input_mpi_t_p);
}  /* Build_mpi_type */

/*------------------------------------------------------------------
 * Function:     Get_input
 * Purpose:      Get the user input:  the left and right endpoints
 *               and the number of trapezoids
 * Input args:   my_rank:  process rank in MPI_COMM_WORLD
 *               comm_sz:  number of processes in MPI_COMM_WORLD
 * Output args:  a_p:  pointer to left endpoint
 *               b_p:  pointer to right endpoint
 *               n_p:  pointer to number of trapezoids
 */
void Get_input(
      int      my_rank  /* in  */,
      int      comm_sz  /* in  */,
      double*  a_p      /* out */,
      double*  b_p      /* out */,
      int*     n_p      /* out */) {
   MPI_Datatype input_mpi_t;

   Build_mpi_type(a_p, b_p, n_p, &input_mpi_t);

   if (my_rank == 0) {
      printf("Enter a, b, and n\n");
      scanf("%lf %lf %d", a_p, b_p, n_p);
   }
   MPI_Bcast(a_p, 1, input_mpi_t, 0, MPI_COMM_WORLD);

   MPI_Type_free(&input_mpi_t);
}  /* Get_input */
```

```c
/*------------------------------------------------------------------
 * Function:     Trap
 * Purpose:      Serial function for estimating a definite integral
 *               using the trapezoidal rule
 * Input args:   left_endpt
 *               right_endpt
 *               trap_count
 *               base_len
 * Return val:   Trapezoidal rule estimate of integral from
 *               left_endpt to right_endpt using trap_count
 *               trapezoids
 */
double Trap(
      double left_endpt  /* in */,
      double right_endpt /* in */,
      int    trap_count  /* in */,
      double base_len    /* in */) {
   double estimate, x;
   int i;

   estimate = (f(left_endpt) + f(right_endpt))/2.0;
   for (i = 1; i <= trap_count-1; i++) {
      x = left_endpt + i*base_len;
      estimate += f(x);
   }
   estimate = estimate*base_len;

   return estimate;
} /*  Trap  */
/*------------------------------------------------------------------
 * Function:    f
 * Purpose:     Compute value of function to be integrated
 * Input args:  x
 */
double f(double x /* in */) {
   return x*x;
} /* f */
```

Performance Evaluation

To evaluate we need to extract a single run time for the program. To do this, we synchronize all the processes before the timing begins with a call to MPI_Barrier followed by a call to MPI_Reduce to find the maximum.

```
double local_start, local_finish;
...
MPI_Barrier(comm);
local_start = MPI_Wtime();
/* code to be timed */
...
local_finish = MPI_Wtime();
local_elapsed = local_finish - local_start
MPI_Reduce(&local_elapsed, &elapsed, 1, MPI_DOUBLE,
   MPI_MAX, 0, comm);

if (my_rank == 0),
   printf("Elapsed time = %e seconds.\n",
      elapsed);
```

MPI_SEND & MPI_RECV Program

```
#include<stdio.h>

#include<mpi.h>

int main( int argc, char *argv[])

{

        int ierr, procid, numprocs;

        ierr = MPI_Init (&argc, &argv);

        ierr = MPI_Comm_rank(MPI_COMM_WORLD, &procid);

        ierr = MPI_Comm_size(MPI_COMM_SIZE, &numprocs);

if (numprocs ! = 2)

{

        printf("ERROR: NUMber of processes is not 2 ! \n");

        return MPI_Abort(MPI_COMM_WORLD, 1);

}
```

Program: In the given program the MPI code, array on each process is created, initialize it on process 0. Once the array has been initialized on process 0, then it is sent out to each process

```
#include<iostream.h>

#include<mpi.h>

int main(int argc, char * argv[])

{

int i;

int nitems = 10;

int mynode, totalnodes;

MPI_Status status;

double * array;

MPI_Init(&argc,&argv);

 MPI_Comm_size(MPI_COMM_WORLD, &totalnodes);

 MPI_Comm_rank(MPI_COMM_WORLD, &mynode);

 array = new double[nitems];

 if(mynode == 0)

 {

 for(i=0;i<nitems;i++)

 array[i] = (double) i;

 }

 if(mynode==0)

 for(i=1;i<totalnodes;i++)

MPI_Send(array,nitems,MPI_DOUBLE,i,1,MPI_COMM_WORLD);

else

MPI_Recv(array,nitems,MPI_DOUBLE,0,1,MPI_COMM_WORLD,

&status);

for(i=0;i<nitems;i++)

 {

 cout << "Processor " << mynode;
 cout << ": array[" << i << "] = " << array[i] << endl;
 }
MPI_Finalize();
}
```

Simultaneous Send and Receive, MPI_Sendrecv:

The subroutine MPI_Sendrecv exchanges messages with another process. A send-receive operation is useful for avoiding some kinds of unsafe interaction patterns and for implementing remote procedure calls.

A message sent by a send-receive operation can be received by MPI_Recv and a send-receive operation can receive a message sent by an MPI_Send.

```
MPI_Sendrecv(&data_to_send, send_count, send_type, destination_ID, send_tag, &received_data,
receive_count, receive_type, sender_ID, receive_tag, comm, &status)
```

Argument Lists

1 data_to_send: variable of a C type that corresponds to the MPI send_type supplied below

2 send_count: number of data elements to send (int)

3 send_type: datatype of elements to send (one of the MPI datatype handles)

4 destination_ID: process ID of the destination (int)

5 send_tag: send tag (int)

6 received_data: variable of a C type that corresponds to the MPI receive_type supplied below

7 receive_count: number of data elements to receive (int)

8 receive_type: datatype of elements to receive (one of the MPI datatype handles)

9 sender_ID: process ID of the sender (int)

10 receive_tag: receive tag (int)

11 comm: communicator (handle)

12 status: status object (MPI_Status)

It should be noted in above stated arguments that they contain all the arguments that were declared in send and receive functions separately in the previously.

Question: Analyze the commands of MPI, write their structures and use them in practice problems.