CS3211 Parallel and Concurrent Programming – Guidelines for tutorial (5-9 April 2010)

Sample Exercises:
[Please conduct these as an interactive discussion, rather than an evaluation. Please also make it clear to the students that they are not being evaluated for their performance in these exercises, so that they are not afraid to make mistakes while answering.]

MPI usage instructions See the file tembusu-MPI-access.pdf in Workbin\Assignments

1. MPI Example - This program should demonstrate a simple data decomposition. The master task first initializes an array and then distributes an equal portion of the array to the other tasks. After the other tasks receive their portion of the array, they perform an addition operation to each array element. They also maintain a sum for their portion of the array. The master task does likewise with its portion of the array. As each of the non-master tasks finish, they send their updated portion of the array to the master. Finally, the master task displays the global sum of all array elements.

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#define  ARRAYSIZE     16000000
#define  MASTER        0

float  data[ARRAYSIZE];

int main (int argc, char *argv[])
{
int   numtasks, taskid, rc, dest, offset, i, j, tag1,
      tag2, source, chunksize;
float mysum, sum;
float update(int myoffset, int chunk, int myid);
MPI_Status status;

/***** Initializations *****/
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
if (numtasks % 4 != 0) {
   printf("Quitting. Number of MPI tasks must be divisible by 4.\n");
   MPI_Abort(MPI_COMM_WORLD, rc);
   exit(0);
   }
MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
printf ("MPI task %d has started...\n", taskid);
chunksize = (ARRAYSIZE / numtasks);
tag2 = 1;
tag1 = 2;

/***** Master task only ******/
if (taskid == MASTER){

  /* Initialize the array */
  sum = 0;
  for(i=0; i<ARRAYSIZE; i++) {
    data[i] =  i * 1.0;
    sum = sum + data[i];
    }
  printf("Initialized array sum = %e\n",sum);

  /* Send each task its portion of the array - master keeps 1st part */
  offset = chunksize;
  for (dest=1; dest<numtasks; dest++) {
```

```
    MPI_Send(&offset, 1, MPI_INT, dest, tag1, MPI_COMM_WORLD);
    MPI_Send(&data[offset], chunksize, MPI_FLOAT, dest, tag2, MPI_COMM_WORLD);
    printf("Sent %d elements to task %d offset= %d\n",chunksize,dest,offset);
    offset = offset + chunksize;
    }

  /* Master does its part of the work */
  offset = 0;
  mysum = update(offset, chunksize, taskid);

  /* Wait to receive results from each task */
  for (i=1; i<numtasks; i++) {
    source = i;
    MPI_Recv(&offset, 1, MPI_INT, source, tag1, MPI_COMM_WORLD, &status);
    MPI_Recv(&data[offset], chunksize, MPI_FLOAT, source, tag2,
      MPI_COMM_WORLD, &status);
    }

  /* Get final sum and print sample results */
  MPI_Reduce(&mysum, &sum, 1, MPI_FLOAT, MPI_SUM, MASTER, MPI_COMM_WORLD);
  printf("Sample results: \n");
  offset = 0;
  for (i=0; i<numtasks; i++) {
    for (j=0; j<5; j++)
      printf("  %e",data[offset+j]);
    printf("\n");
    offset = offset + chunksize;
    }
  printf("*** Final sum= %e ***\n",sum);

  }  /* end of master section */



/***** Non-master tasks only *****/

if (taskid > MASTER) {

  /* Receive my portion of array from the master task */
  source = MASTER;
  MPI_Recv(&offset, 1, MPI_INT, source, tag1, MPI_COMM_WORLD, &status);
  MPI_Recv(&data[offset], chunksize, MPI_FLOAT, source, tag2,
    MPI_COMM_WORLD, &status);

  mysum = update(offset, chunksize, taskid);

  /* Send my results back to the master task */
  dest = MASTER;
  MPI_Send(&offset, 1, MPI_INT, dest, tag1, MPI_COMM_WORLD);
  MPI_Send(&data[offset], chunksize, MPI_FLOAT, MASTER, tag2, MPI_COMM_WORLD);

  MPI_Reduce(&mysum, &sum, 1, MPI_FLOAT, MPI_SUM, MASTER, MPI_COMM_WORLD);

  } /* end of non-master */


MPI_Finalize();

}   /* end of main */


float update(int myoffset, int chunk, int myid) {
  int i;
  float mysum;
```

```
  /* Perform addition to each of my array elements and keep my sum */
  mysum = 0;
  for(i=myoffset; i < myoffset + chunk; i++) {
    data[i] = data[i] + i * 1.0;
    mysum = mysum + data[i];
    }
  printf("Task %d mysum = %e\n",myid,mysum);
  return(mysum);
  }
```

2. This exercise presents a simple program to determine the value of pi. The algorithm suggested here is chosen for its simplicity. The method evaluates the integral of 4/(1+x*x) between 0 and 1. The method is simple: the integral is approximated by a sum of n intervals; the approximation to the integral in each interval is (1/n)*4/(1+x*x). The master process (rank 0) asks the user for the number of intervals; the master should then broadcast this number to all of the other processes. Each process then adds up every n'th interval (x = rank/n, rank/n+size/n,...). Finally, the sums computed by each process are added together using a reduction. Use **MPI_Reduce**

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

int main(argc,argv)
int argc;
char *argv[];
{
    int done = 0, n, myid, numprocs, i;
    double PI25DT = 3.14159265358979323846264 3;
    double mypi, pi, h, sum, x;

    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();
    return 0;
}
```

3. What will happen when the following program is run? Explain your answer.

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

int main (int argc, char *argv[])
{
int numtasks, rank, dest, tag, source, rc, count;
char inmsg, outmsg='x';
MPI_Status Stat;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
printf("Task %d starting...\n",rank);

if (rank == 0) {
  if (numtasks > 2)
    printf("Numtasks=%d. Only 2 needed. Ignoring extra...\n",numtasks);
  dest = rank + 1;
  source = dest;
  tag = rank;
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  printf("Sent to task %d...\n",dest);
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  printf("Received from task %d...\n",source);
  }

else if (rank == 1) {
  dest = rank - 1;
  source = dest;
  tag = rank;
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  printf("Received from task %d...\n",source);
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  printf("Sent to task %d...\n",dest);
  }

if (rank < 2) {
  rc = MPI_Get_count(&Stat, MPI_CHAR, &count);
  printf("Task %d: Received %d char(s) from task %d with tag %d \n",
         rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);
  }

MPI_Finalize();
}
```

Answer: The execution will hang, and the processes cannot progress to completion. Process 0 can post its blocking send (freeing the sendbuffer) and then wait for the receive. Process 1 cannot progress to perform its send, since its receive tag does not match with the tag of the message sent by process 0.