CS3211 Parallel and Concurrent Programming – Week 13 tutorial

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

MPI program running over Ethernet (MPICH)
[user@access0]$ /opt/mpich/bin/mpicc -c cpi.c]
[user@access0]$ /opt/mpich/bin/mpicc -o cpi cpi.o]

For MPICH, create a machine file that looks like this:

```
# cat mynodes
access0
access1
access2
access3
access4
access5
access6
access7
access8
access9
```

Run binary MPI program (MPICH)
[user@access0]$ /opt/mpich/bin/mpirun -machinefile mynodes -np 8 /home/user/cpi

1. MPI Example Warmup– Management of groups. What will be printed by the following program (and why)?

   Assume MPI_Group_incl creates a new group by taking only the listed members (which is then used within a communicator).

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#define NPROCS 8

int main (int argc, char *argv[])
{
int         rank, new_rank, sendbuf, recvbuf, numtasks,
            ranks1[4]={0,1,2,3}, ranks2[4]={4,5,6,7};
MPI_Group   orig_group, new_group;
MPI_Comm    new_comm;

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

if (numtasks != NPROCS) {
  printf("Must specify %d tasks. Terminating.\n",NPROCS);
  MPI_Finalize();
  exit(0);
  }

sendbuf = rank;

/* Extract the original group handle */
MPI_Comm_group(MPI_COMM_WORLD, &orig_group);

/* Divide tasks into two distinct groups based upon rank */
if (rank < NPROCS/2) {
  MPI_Group_incl(orig_group, NPROCS/2, ranks1, &new_group);
  }
else {
  MPI_Group_incl(orig_group, NPROCS/2, ranks2, &new_group);
  }

/* Create new new communicator and then perform collective communications */
MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);
MPI_Allreduce(&sendbuf, &recvbuf, 1, MPI_INT, MPI_SUM, new_comm);

MPI_Group_rank (new_group, &new_rank);
printf("rank= %d newrank= %d recvbuf= %d\n",rank,new_rank,recvbuf);

MPI_Finalize();
}
```

Answer: Try to run it in tembusu with 8 processes. The program will split the processes into different communicators and assign new ranks to the processes.

2. What will happen when we run the following program?

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

int main (int argc, char *argv[])
{
int   numtasks, taskid, len, buffer, root, count;
char hostname[MPI_MAX_PROCESSOR_NAME];

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
MPI_Get_processor_name(hostname, &len);

printf ("Task %d on %s starting...\n", taskid, hostname);
buffer = 23;
root = 0;
count = taskid;
if (taskid == root)
   printf("Root: Number of MPI tasks is: %d\n",numtasks);

MPI_Bcast(&buffer, count, MPI_INT, root, MPI_COMM_WORLD);


MPI_Finalize();

}
```

Answer: Try it in tembusu. The program will ``hang". The count being used as argument in the broadcast library is wrong. Change the count to 1, and the problem should disappear.

3. Following is a simple example of the Scatter library in MPI, one of our collective communication primitives. Will the program progress to completion? What will be printed against each process? Try it on 4 processors and 4 processes in tembusu.

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#define SIZE 4

int main (int argc, char *argv[])
{
int numtasks, rank, sendcount, recvcount, source;
float sendbuf[SIZE][SIZE] = {
  {1.0, 2.0, 3.0, 4.0},
  {5.0, 6.0, 7.0, 8.0},
  {9.0, 10.0, 11.0, 12.0},
  {13.0, 14.0, 15.0, 16.0}  };
float recvbuf[SIZE];

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

if (numtasks == SIZE) {
  source = 1;
  sendcount = SIZE;
  recvcount = SIZE;
  MPI_Scatter(sendbuf,sendcount,MPI_FLOAT,recvbuf,recvcount,
              MPI_FLOAT,source,MPI_COMM_WORLD);

  printf("rank= %d  Results: %f %f %f %f\n",rank,recvbuf[0],
         recvbuf[1],recvbuf[2],recvbuf[3]);
  }
else
  printf("Must specify %d processors. Terminating.\n",SIZE);

MPI_Finalize();
}
```

Answer: Try it will 4 processors and 4 processes in tembusu.
The program will terminate. Print out is as follows. The per-process output may appear out-of-sequence.

rank= 0 Results: 1.000000 2.000000 3.000000 4.000000
rank= 1 Results: 5.000000 6.000000 7.000000 8.000000
rank= 2 Results: 9.000000 10.000000 11.000000 12.000000
rank= 3 Results: 13.000000 14.000000 15.000000 16.000000

4. This exercise implements a simple parallel data structure. This structure is a two dimension regular mesh of points, divided into slabs, with each slab allocated to a different processor. In the simplest C form, the full data structure is

```
double x[maxn][maxn];
```

and we want to arrange it so that each processor has a local piece:

```
double xlocal[maxn][maxn/size];
```

where `size` is the size of the communicator (e.g., the number of processors).

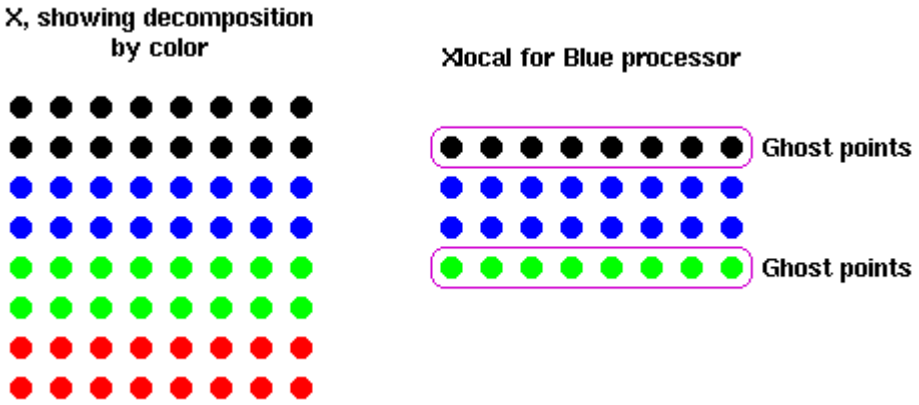
If that was all that there was to it, there wouldn't be anything to do. However, for the computation that we're going to perform on this data structure, we'll need the adjacent values. That is, to compute a new `x[i][j]`, we will need

```
x[i][j+1]
x[i][j-1]
x[i+1][j]
x[i-1][j]
```

The last two of these could be a problem if they are not in `xlocal` but are instead on the adjacent processors. To handle this difficulty, we define ghost points that we will contain the values of these adjacent points.

Write code to copy divide the array x into equal-sized strips and to copy the adjacent edges to the neighboring processors. Assume that x is maxn by maxn, and that maxn is evenly divided by the number of processors. For simplicity, you may assume a fixed size array and a fixed (or minimum) number of processors.

To test the routine, have each processor fill its section with the rank of the process, and the ghostpoints with -1. After the exchange takes place, test to make sure that the ghostpoints have the proper value. Assume that the domain is not periodic; that is, the top process (rank = size - 1) only sends and receives data from the one under it (rank = size - 2) and the bottom process (rank = 0) only sends and receives data from the one above it (rank = 1). Consider a maxn of 12 and use 4 processors to start with.



```
#include <stdio.h>
#include "mpi.h"
#define maxn 12

int main( argc, argv )
int argc;
char **argv;
{
    int rank, value, size, errcnt, toterr, i, j;
    MPI_Status status;
    double x[12][12];
    double xlocal[(12/4)+2][12];

    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    if (size != 4) MPI_Abort( MPI_COMM_WORLD, 1 );

    /* xlocal[][0] is lower ghostpoints, xlocal[][maxn+2] is upper */

    /* Fill the data as specified */
    for (i=1; i<=maxn/size; i++)
        for (j=0; j<maxn; j++)
            xlocal[i][j] = rank;
    for (j=0; j<maxn; j++) {
        xlocal[0][j] = -1;
        xlocal[maxn/size+1][j] = -1;
    }
    /* Send up unless I'm at the top, then receive from below */
    /* Note the use of xlocal[i] for &xlocal[i][0] */
    if (rank < size - 1)
        MPI_Send( xlocal[maxn/size], maxn, MPI_DOUBLE, rank + 1, 0,
                  MPI_COMM_WORLD );
    if (rank > 0)
        MPI_Recv( xlocal[0], maxn, MPI_DOUBLE, rank - 1, 0, MPI_COMM_WORLD,
                  &status );
    /* Send down unless I'm at the bottom */
    if (rank > 0)
        MPI_Send( xlocal[1], maxn, MPI_DOUBLE, rank - 1, 1, MPI_COMM_WORLD );
    if (rank < size - 1)
        MPI_Recv( xlocal[maxn/size+1], maxn, MPI_DOUBLE, rank + 1, 1,
                  MPI_COMM_WORLD, &status );

    /* Check that we have the correct results */
    errcnt = 0;
    for (i=1; i<=maxn/size; i++)
        for (j=0; j<maxn; j++)
            if (xlocal[i][j] != rank) errcnt++;
    for (j=0; j<maxn; j++) {
        if (xlocal[0][j] != rank - 1) errcnt++;
        if (rank < size-1 && xlocal[maxn/size+1][j] != rank + 1) errcnt++;
    }
    MPI_Reduce( &errcnt, &toterr, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD );
    if (rank == 0) {
        if (toterr)
            printf( "! found %d errors\n", toterr );
        else
            printf( "No errors\n" );
```

```
    }
    MPI_Finalize( );
    return 0;
}
```

5. As the last exercise, we take up an embarrassingly parallel activity. Write a MPI program to generate all prime numbers below a certain threshold (say all prime numbers lesser than `2500000`) All processes should distribute the work equally. Each process can take up every nth number where n == (rank*2) + 1. Even numbers should always be ignored. Comment on whether using strides is better than dividing the work among processes via contiguous blocks of numbers.

Answer:

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

#define LIMIT     2500000     /* Increase this to find more primes */
#define FIRST     0           /* Rank of first task */

int isprime(int n) {
int i,squareroot;
if (n>10) {
   squareroot = (int) sqrt(n);
   for (i=3; i<=squareroot; i=i+2)
      if ((n%i)==0)
         return 0;
   return 1;
   }
/* Assume first four primes are counted elsewhere. Forget everything else */
else
   return 0;
}


int main (int argc, char *argv[])
{
int   ntasks,               /* total number of tasks in partitiion */
      rank,                 /* task identifier */
      n,                    /* loop variable */
      pc,                   /* prime counter */
      pcsum,                /* number of primes found by all tasks */
      foundone,             /* most recent prime found */
      maxprime,             /* largest prime found */
      mystart,              /* where to start calculating */
      stride;               /* calculate every nth number */

double start_time,end_time;

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&ntasks);
if (((ntasks%2) !=0) || ((LIMIT%ntasks) !=0)) {
   printf("Sorry - this exercise requires an even number of tasks.\n");
   printf("evenly divisible into %d.  Try 4 or 8.\n",LIMIT);
   MPI_Finalize();
   exit(0);
   }

start_time = MPI_Wtime();   /* Initialize start time */
mystart = (rank*2)+1;       /* Find my starting point - must be odd number */
stride = ntasks*2;          /* Determine stride, skipping even numbers */
pc=0;                       /* Initialize prime counter */
foundone = 0;               /* Initialize */

/******************** task with rank 0 does this part ******************/
if (rank == FIRST) {
   printf("Using %d tasks to scan %d numbers\n",ntasks,LIMIT);
   pc = 4;                  /* Assume first four primes are counted here */
   for (n=mystart; n<=LIMIT; n=n+stride) {
```

```
      if (isprime(n)) {
         pc++;
         foundone = n;
         /***** Optional: print each prime as it is found
         printf("%d\n",foundone);
         *****/
         }
      }
   MPI_Reduce(&pc,&pcsum,1,MPI_INT,MPI_SUM,FIRST,MPI_COMM_WORLD);
   MPI_Reduce(&foundone,&maxprime,1,MPI_INT,MPI_MAX,FIRST,MPI_COMM_WORLD);
   end_time=MPI_Wtime();
   printf("Done. Largest prime is %d Total primes %d\n",maxprime,pcsum);
   printf("Wallclock time elapsed: %.2lf seconds\n",end_time-start_time);
}


/******************** all other tasks do this part ***********************/
if (rank > FIRST) {
   for (n=mystart; n<=LIMIT; n=n+stride) {
      if (isprime(n)) {
         pc++;
         foundone = n;
         /***** Optional: print each prime as it is found
         printf("%d\n",foundone);
         *****/
         }
      }
   MPI_Reduce(&pc,&pcsum,1,MPI_INT,MPI_SUM,FIRST,MPI_COMM_WORLD);
   MPI_Reduce(&foundone,&maxprime,1,MPI_INT,MPI_MAX,FIRST,MPI_COMM_WORLD);
   }

MPI_Finalize();
}
```

The method of using stride is preferred over contiguous blocks of numbers, since numbers in the higher range require more work to compute and may result in load imbalance. This program demonstrates embarrassing parallelism. Collective communications calls are used to reduce the only two data elements requiring communications: the number of primes found and the largest prime.