ECE 563 Spring 2016, Second Exam

DO NOT START WORKING ON THIS UNTIL TOLD TO DO SO. LEAVE IT ON THE DESK.

THE LAST PAGE IS THE ANSWER SHEET. TEAR IT OFF AND PUT ALL ANSWERS THERE. TURN IN BOTH PARTS OF THE TEST WHEN FINISHED.

You have until 3:20PM to take this exam. The total number of points should be 100, with three awarded for getting your name on this test and the answer sheet. After taking the test turn in both the test and the answer sheet.

Your exam should have 6 (six) pages total (including this cover page). ***YOU SHOULD ALSO HAVE A SEPARATE ANSWER SHEET!*** As soon as the test begins, check that your exam is complete and *let the proctors know immediately if it is not.*

Assume all programs are correct unless the question asks about errors. Assume all include files are included. The "include" statements are not shown to make the program text shorter.

This exam is open book, open notes, but absolutely no electronics. If you have a question, please ask for clarification. If the question is not resolved, state on the test whatever assumptions you need to make to answer the question, and answer it under those assumptions. *Check the front board occasionally for corrections.*

I have neither given nor received help during this exam from any other person or electronic source, and I understand that if I have I will be guilty of cheating and will fail the exam and perhaps the course.

Signature (must be signed to be graded):

Name (printed, worth 1 pt):

Q1, 5 points: Is there a race on *sharedInt* in the program below? Answer "yes" or "no".

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

  int i;

  omp_lock_t locks[omp_get_num_threads( )];
  for (i=0; i < omp_get_num_threads( );i++) {
     omp_init_lock(&locks[i]);
  }

  int sharedInt = 0;

  #pragma omp parallel for
  for (i=0; i < 10000; i++) {
     omp_set_lock(&locks[omp_get_thread_num( )]);
     sharedInt = sharedInt + omp_get_thread_num( );
     omp_unset_lock(&locks[omp_get_thread_num( )]);
  }
}
```

The next three questions (Q2 – Q4) use the program below.

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

  int i;

  #pragma omp parallel
  for (i=0; i < 10000; i++) {
     #pragma omp critical
     printf("first print.\n");
  }

  #pragma omp parallel
  for (i=0; i < 10000; i++) {
     #pragma omp master
     printf("second print.\n");
  }

  #pragma omp parallel for
  for (i=0; i < 10000; i++) {
     #pragma omp critical
     printf("third print.\n");
  }
}
```

Q2, 4 points: how many lines are printed by the first print?

Q3, 4 points: how many lines are printed by the second print?

Q4, 4 points: how many lines are printed by the third print?

The next five questions (Q5 – Q9) use the program below.

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

  int i;
  int sharedInt1, sharedInt2, sharedInt3, sharedInt4, sharedInt5;
  sharedInt1 = sharedInt2 = sharedInt3 = sharedInt4 = sharedInt5 = 0;

  #pragma omp parallel
  for (i=0; i < 10000; i++) {
     #pragma omp critical
     sharedInt1 = sharedInt1 + omp_get_thread_num( ); // LINE 1
  }

  #pragma omp parallel
  for (i=0; i < 10000; i++) {
     #pragma omp single
     sharedInt2 = sharedInt2 + omp_get_thread_num( ); // LINE 2
  }

  #pragma omp parallel
  for (i=0; i < 10000; i++) {
     #pragma omp master
     sharedInt3 = sharedInt3 + omp_get_thread_num( ); // LINE 3
  }

  for (i=0; i < 10000; i++) {
     #pragma omp parallel
     for (i=0; i < 10000; i++) {
        #pragma omp single
        sharedInt4 = sharedInt4 + omp_get_thread_num( ); // LINE 4
     }
  }

  for (i=0; i < 10000; i++) {
     #pragma omp parallel
     for (i=0; i < 10000; i++) {
        #pragma omp master
        sharedInt5 = sharedInt5 + omp_get_thread_num( ); // LINE 5
     }
  }
}
```

Q5, 2 points: Is there a race on *sharedInt1* (LINE 1)?

Q6, 2 points: Is there a race on *sharedInt2* (LINE 2)?

Q7, 2 points: Is there a race on *sharedInt3* (LINE 3)?

Q8, 2 points: Is there a race on *sharedInt4* (LINE 4)?

Q9, 2 points: Is there a race on *sharedInt5* (LINE 5)?

There is an MPI program with a cyclic (i.e., round-robin) distribution of the elements of arrays A. The program will execute on 4 processes, and A has 24 elements.

Q10, 4 points: In the answer sheet, show the elements, in the global space of the array, that reside on P_2. Note that there are four processor, overall (P_0, P_1, P_2 and P_3) but you only need to show what P_2 contains. Note also that there may be more positions in the vector on the answer sheet than there are elements of A on P_2.

Q11, 4 points: Assume the time spent processing an element of A is $1/i$, where i is the index of the element in the global space. Will the cyclic distribution give better, worse or the same performance, on average, as a block distribution of the array?

Two programmers develop communication code for the same routine. Programmer A writes the code, with the `MPI_Ssend` before the `MPI_IRecv`:

```
// VERSION A
if (pid == 0) left = numProcs-1;
else left = pid - 1;

if (pid == numProcs-1) right = 0;
else right = pid + 1;

int rc =  MPI_Ssend(sendBuf, 100, MPI_Double, left, NULL, MPI_Comm_World);
rc = MPI_IRecv(recBuf, 100, MPI_Double, right, NULL, MPI_Comm_World, request);
. . .
MPI_Wait(request, status)
```

and programmer B writes the code, with the `MPI_IRecv` before the `MPI_Ssend`:

```
// VERSION B
if (pid == 0) left = numProcs-1;
else left = pid - 1;

if (pid == numProcs-1) right = 0;
else right = pid + 1;

int rc = MPI_IRecv(recBuf, 100, MPI_Double, right, NULL, MPI_Comm_World, request);
rc =  MPI_Ssend(sendBuf, 100, MPI_Double, left, NULL, MPI_Comm_World);
. . .
MPI_Wait(request, status)
```

Q12, 4 points: Pick the best answer of which version is better.

a. Version A, since resources will not be tied by up the IRecv operation while waiting for the send to complete.

b. Version B, because deadlock cannot occur since a buffer to receive data will be available as soon as the IRecv executes.

c. Both are equally good.

A program takes 40 minutes to run sequentially, and 4 minutes to run in parallel on 40 processors.

Q13, 4 points: What is its speedup?

Q14, 4 points: What is its efficiency?

Amdahl's Law

A program whose sequential execution time runs for 100 minutes is examined and it is determined that 10 minutes of the execution time is inherently serial, i.e., cannot be executed in parallel, and the code consuming the remaining 90 minutes of the execution time can be executed in parallel.

Q15, 5 points What is sequential fraction of the program?

Q16, 5 points Ignorning communication costs, and keeping the problem size the same as above, what will the speedup be on an infinite number of procesors?

Q17, 5 points Ignorning communication costs, and keeping the problem size the same described above, what will the speedup be on 9 procesors?

Gustafson-Barsis

Q18, 5 points A program is executed on 101 processors. The speedup is 91. What fraction of time is spent in the serial code in the scaled problem?

Q19, 5 points A program is executed on 100001 processors. The desired speedup is 100000. What fraction of time can be spent in the serial portion of the program in a scaled program.

Karp-Flatt

Q20, 5 points. A program has an experimentally determined serial fraction that is increasing linearly with the number of processors. The amount of sequential work is staying the same and the distribution of the work leads to a good load balance. Would a faster network (select the best answer)

a. Increase the experimentally determined serial fraction

b. Not affect the experimentally determined serial fraction

c. Reduce the experimentally determined serial fraction

Q21, 5 points. Timings of a program executing on 10 processors show that it gets a speedup of 5. What can we say about the fraction of time spent in serial code caused by communication, load balance and increasing serial work? You can show your answer as a fraction.

Isoefficiency

Sam was not paying attention in class the day the MPI_Reduce, MPI_BCast and MPI_AllReduce collective communication primitives were discussed. To have every processor get the result of doing a reduce on the array A he has processor 0 do a MPI_Reduce and then sends the results to every other process using MPI_Send instructions. His code looks like:

```
int rc = MPI_Reduce(A, &result, 1, MPI_Int, MPI_Add, 0, MPI_Comm_World);

if (pid != 0) rc = MPI_Recv(result, 1, MPI_Int, 0, MPI_Int, MPI_ANY_TAG,
                            MPI_Comm_World, status)

for (int i=1; i<numProcs; i++) {
   rc = MPI_Send(result, 1, MPI_Int, i, MPI_ANY_TAG, MPI_Comm_World);
}
```

The work, W, done by the program is n.

Q22, 5 points. Show the formula for T_p (the parallel time).

Q23, 5 points. Show the formula for T_O (the total overhead).

Q24, 5 points. To maintain the same efficiency, does the work need to be

a. As fast, or faster than, T_O

b. Slower than T_O

c. The efficiency is related to the experimentally determined serial fraction and not T_O, so T_O is irrelevant to determing the efficiency as the program is run on larger numbers of processors.