Parallel & Distributed Computing **Name:** _____
Midterm, Spring, 2014

Show Your Work! Point values are in square brackets. Point values are in square brackets. There are 30 points altogether.

1. Sally has written a Pthreads program that has the following efficiencies:

	Input Size				
p	1000	2000	4000	8000	16,000
1	1.0	1.0	1.0	1.0	1.0
2	0.9	0.9	0.9	0.9	1.0
4	0.8	0.9	0.9	0.9	0.9
8	0.8	0.8	0.8	0.8	0.9

On the basis of these data, is Sally's program

(a) Strongly scalable?

(b) Weakly scalable?

(c) Scalable?

Explain your answers. [3 points]

(a) The program is *not* strongly scalable. If it were, the efficiency would remain constant (or increase) when the input size is fixed and the number of processes/threads is increased. This is not the case for any of the input sizes.

(b) The program is *not* weakly scalable. If it were, the efficiency would remain constant (or increase) when the number of processes/threads and the input size are increased at the same rate. This is not the case for example, when $p = 4$ and $n = 2000$, $E = 0.9$, but when p and n are doubled in size, E drops to 0.8.

(c) The program *is* scalable. A program is scalable if for a given rate of increase in the number of processes/threads, there is a corresponding rate of increase in the problem size so that the efficiency remains constant. If the number of processes is doubled and the problem size is increased by a factor of 8, the efficiency remains constant.

Note that the case $p = 1$, is not included in any of the different formulations of scalability. Also note that the scalability of this program can only be commented on for the (very) limited amount of timing data.

2. Determine the output of the following MPI program if it is run with four processes. [5]

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

int main(int argc, char* argv[]) {
   int        my_rank, p, source, dest, i, q, n = 3;
   int        list[3], tlist[3];
   MPI_Comm   comm;
   MPI_Status status;

   MPI_Init(&argc, &argv);
   comm = MPI_COMM_WORLD;
   MPI_Comm_size(comm, &p);
   MPI_Comm_rank(comm, &my_rank);

   source = (p + my_rank - 1) % p;
   dest = (my_rank + 1) % p;
   for (i = 0; i < n; i++)
      list[i] = tlist[i] = my_rank + i;

   for (q = 1; q < p; q++) {
      /* Assume adequate buffering available */
      MPI_Send(tlist, n, MPI_INT, dest, 0, comm);
      MPI_Recv(tlist, n, MPI_INT, source, 0, comm, &status);
      for (i = 0; i < n; i++)
         list[i] += tlist[i];
   }

   if (my_rank == 0) {
      for (i = 0; i < n; i++)
         printf("%d ",list[i]);
      printf("\n");
   }
   MPI_Finalize();
   return 0;
}  /* main */
```

The output is:

```
6 10 14
```

A trace of the values in various program variables is on the following page.

(Solution to number 2, continued.)

my_rank	0			1			2			3		
source	3			0			1			2		
dest	1			2			3			0		
q	1	2	3	1	2	3	1	2	3	1	2	3
list	0	1	2	1	2	3	2	3	4	3	4	5
	3	5	7	1	3	5	3	5	7	5	7	9
	5	8	11	4	7	10	3	6	9	6	9	12
	6	10	14	6	10	14	6	10	14	6	10	14
tlist	0	1	2	1	2	3	2	3	4	3	4	5
	3	4	5	0	1	2	1	2	3	2	3	4
	2	3	4	3	4	5	0	1	2	1	2	3
	1	2	3	2	3	4	3	4	5	0	1	2

3. (a) State Amdahl's Law for the speedup of a parallel program.

   (b) Explain why Amdahl's Law can be misleading.

   [4]

   (a) Suppose a fraction r of the runtime of a serial program is "perfectly parallelized," and the remaining fraction $1 - r$ is "inherently serial." Then if the runtime of the serial program is T, the speedup of the parallel program will be

$$S = \frac{T}{rT/p + (1-r)T} = \frac{1}{r/p + 1 - r}.$$

   So the speedup is bounded above by

$$\frac{1}{1-r}.$$

   (b) This formula can be misleading because it implies that if an algorithm has any inherently serial code, a parallelization of the algorithm has a fixed, finite bound on the possible speedup. However, this is misleading, since Amdahl's Law doesn't take into consideration problem size. If we solve a problem of size n with 10 processors, we probably don't want to solve the same problem with 1000 processors: we want to solve a bigger instance of the problem, and the inherently serial fraction of a larger instance of the problem may be much smaller.

4. Lemuel has written a parallel MPI function `Lems_func`, and he wants to study its performance. So he adds the following code to his MPI program:

```
double start, finish, elapsed;
. . .
start = MPI_Wtime();
/* All processes call Lems_func */
Lems_func(. . .);
finish = MPI_Wtime();
elapsed = finish-start;
if (my_rank == 0)
   printf("Lems_func took %e seconds\n", elapsed);
```

(a) The time reported by this code can be misleading. Explain why. [2]

(b) Modify the above code so that it reports a more accurate time for `Lems_func`. [2]

(a) The runtime of a parallel program is the time that elapses between the time that the first process/thread starts executing and the time that the last process/thread finishes executing. Lemuel's code only reports the time that process 0 spends executing. So unless process 0 is the first to start and the last to finish, this won't represent the parallel runtime.

(b) Lemuel should add a barrier before the first call to `MPI_Wtime()` and he should find the maximum elapsed time after the second call to `MPI_Wtime()`:

```
double start, finish, elapsed, my_elapsed;
. . .
MPI_Barrier(comm);
start = MPI_Wtime();
/* All processes call Lems_func */
Lems_func(. . .);
finish = MPI_Wtime();
my_elapsed = finish-start;
MPI_Reduce(&my_elapsed, &elapsed, 1, MPI_DOUBLE, MPI_MAX,
      0, comm);
if (my_rank == 0)
   printf("Lems_func took %e seconds\n", elapsed);
```

5. Flynn's taxonomy classifies computing systems according to the number of instruction streams and the number of data streams. The von Neumann architecture is a single-instruction stream, single data stream (SISD) system. A typical multicore system (e.g., Intel Xeon) is a multiple instruction stream, multiple data stream (MIMD) system. Subsystems of typical GPU's (e.g., Nvidia Quadro) are single instruction stream, multiple data stream (SIMD) systems. Is it possible to construct a multiple instruction stream, single data stream (MISD) system? Explain your answer. [2]

   Suppose we have a collection of processors and each processor executes its own sequence of instructions. Furthermore, suppose the output of processor q is the input to processor $q+1$:

```
       +----+                  +----+                  +----+
0 in-->| P0 |-->0 out = 1 in-->| P1 |-->1 out = 2 in-->| P2 |--> etc.
       +----+                  +----+                  +----+
```

   So this looks like a computational pipeline, except that each processor is executing its own instruction stream.

6. In class we saw that for a *particular* input a parallel implementation of linear search could obtain superlinear speedup without overcoming a resource limitation. Suppose $T_\sigma(n)$ denotes the *worst case* serial runtime of linear search with input size n, and $T_\pi(n,p)$ denotes the *worst case* parallel runtime of linear search with input size n and p processors. Is it possible that

$$\frac{T_\sigma(n)}{T_\pi(n,p)} > p?$$

   Assume that n and p have been chosen so that the parallel implementation doesn't overcome a resource limitation. Explain your answer. [2]

   No. In the worst case, the element searched for will not be in the list. In this case, serial linear search will execute n comparisons, and an optimal parallel linear search will execute n/p comparisons on each process/thread. So if there is no parallel overhead, the best possible speedup is

$$S = \frac{n}{n/p} = p.$$

7. We've seen four "models" of parallel computation:

   (a) PRAM

   (b) BSP

   (c) LogP

   (d) The "three constants" model (which we discussed in class):

$$T_\pi(n,p) = x(n,p)t_a + y(n,p)t_s + z(n,p)t_w$$

   Choose two of the four models and identify (i) a weakness of the model, and (ii) a strength of the model. Explain your answers. [4]

   There are a *lot* of possibilities here. Here are a few.

   (a) PRAM

   - Strength: With the PRAM model it is relatively easy to estimate the runtime of a parallel program.
   - Weakness: No practical parallel system is accurately modeled by PRAM. So it's performance predictions are usually of little practical value.

   (b) BSP

   - Strength: Dividing execution of a parallel program into supersteps makes performance modeling easier than it is with LogP or Three Constants.
   - Weakness: Synchronization (which occurs in each superstep) is *very* expensive in large parallel systems. So division of a parallel program into supersteps can result in a huge performance penalty.

   (c) LogP

   - Strength: Provides a detailed model of network communication.
   - Weakness: Doesn't model computation, and reported models are not very accurate.

   (d) Three constants

   - Strength: Provides a relatively simple model of communication and computation.
   - Weakness: Empirically derived models are not very accurate.

8. The current MPI specification assumes that an MPI program is run on a "reliable" system, a system in which the hardware is not subject to failures. Is this a reasonable assumption? Explain your answer. [2]

   This isn't a reasonable assumption. In an exascale parallel system with millions of components, the mean time between failures can be measured in milliseconds, if not microseconds. So it's essential that the programming system provide simple and effective recovery methods.

9. Suppose that $A = (a_{ij})$ is an $n \times n$ matrix:

$$A = \begin{pmatrix} a_{00} & a_{01} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,n-1} \end{pmatrix}.$$

Recollect that the *transpose* of A, denoted A^T, is the matrix obtained from A by interchanging its rows and columns. So

$$A^T = \begin{pmatrix} a_{00} & a_{10} & \cdots & a_{n-1,0} \\ a_{01} & a_{11} & \cdots & a_{n-1,1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{0,n-1} & a_{1,n-1} & \cdots & a_{n-1,n-1} \end{pmatrix}.$$

If the one-dimensional array `A` stores the matrix A in row-major order (first row in first n elements, second row in next n elements, etc.) then the following serial code can be used to store A^T in the one-dimensional array `AT`.

```
for (i = 0; i < n; i++)
   for (j = 0; j < n; j++)
      AT[j*n + i] = A[i*n + j];
```

Write a parallel Pthreads function that takes as input a square matrix stored as a 1-dimensional array and returns the transpose also stored as a 1-dimensional array. The original matrix `A`, its transpose `AT`, and the order of the matrices `n` are stored as global variables. You can assume that n is evenly divisible by p, the number of threads. [4 points]

(If you need more space, continue your solution on the following page.)

```
void* Transpose(void* rank) {
   long my_rank = (long) rank;
   int i, j;
   int my_first_row = my_rank*n/thread_count;
   int my_last_row = my_first_row + n/thread_count - 1;

   for (i = my_first_row; i <= my_last_row; i++)
      for (j = 0; j < n; j++)
         AT[j*n + i] = A[i*n + j];

   return NULL;
}  /* Transpose */
```