Foundations of Parallel Computing Fall 2014 Midterm Exam

Prof. Alan Kaminsky — October 16, 2014

Name: _____

INSTRUCTIONS

1. 12 questions. 69 points. Read ***ALL*** questions ***CAREFULLY*** before starting!
2. Record your answers in the space provided. Use the back of the sheet if necessary.
3. Show your work in the space provided. If your answer is incorrect and you did not show your work, the question will get 0 points. If your answer is incorrect but you showed your work, the question might receive partial credit.

QUESTIONS

Questions 1–4. You have measured a certain parallel program's running time T on K cores computing a problem size of N for various values of K and N, and have derived this formula:

$$T = 400 + 400\,N + 4000\,N/K$$

1. (6 points) The program is run with $N = 100$. Under strong scaling, what are the program's running time, speedup, and efficiency for $K = 1$, $K = 2$, $K = 4$, and $K = 8$? Give your answers as numbers (not formulas) with three digits after the decimal point.

N	*K*	*T*	*Speedup*	*Efficiency*
100	1	440400	1.000	1.000
100	2	240400	1.832	0.916
100	4	140400	3.137	0.784
100	8	90400	4.872	0.609

2. (6 points) The program is run with $N = 50$ on one core. Under weak scaling, where the problem size is increased in proportion to the number of cores, what are the program's running time, sizeup, and efficiency for $K = 1$, $K = 2$, $K = 4$, and $K = 8$? Give your answers as numbers (not formulas) with three digits after the decimal point.

N	*K*	*T*	*Sizeup*	*Efficiency*
50	1	220400	1.000	1.000
100	2	240400	1.834	0.917
200	4	280400	3.144	0.786
400	8	360400	4.892	0.612

3. (3 points) For $N = 200$, what fraction of the program must be performed sequentially?

   Total T on one core = $400 + 400 \cdot 200 + 4000 \cdot 200 / 1 = 880400$
   Sequential portion of T = $400 + 400 \cdot 200 = 80400$
   Sequential fraction $F = 80400 / 880400 = 0.091$

4. (3 points) For $N = 200$, what is the maximum speedup the program can achieve?

   Maximum speedup = $1/F = 1 / 0.091 = 10.950$

Questions 5–8. The following is a fragment of a sequential Java program to compute a table of the square root function, $y = \text{sqrt}(x)$, for the values $x = 0.0, 0.1, 0.2, \ldots 19.8, 19.9, 20.0$, using Newton iteration:

$$ynext = (x/y + y)/2$$

Given an initial guess for y, the preceding statement is repeated, replacing y with $ynext$ at each iteration, until convergence. Each value of the square root function is to be computed with five digits of precision. Note that for different values of x, different numbers of iterations might be needed to achieve the required precision.

```
double[] sqrtTable = new double [201];
sqrtTable[0] = 0.0;
for (int i = 1; i <= 200; ++ i) {
   double x = i/10.0;
   double y = x/2.0;
   double ynext = (x/y + y)/2.0;
   while (Math.abs (2.0*(y – ynext)/(y + ynext)) >= 0.00001) {
      y = ynext;
      ynext = (x/y + y)/2.0;
   }
   sqrtTable[i] = y;
}
```

5. (3 points) In the preceding program, is it possible to parallelize the outer for loop? Explain why or why not.

   Yes, it is possible. The square root of each value of x is computed independently of all the others.

6. (3 points) In the preceding program, is it possible to parallelize the inner while loop? Explain why or why not.

   No, it is not possible. Each iteration of the inner loop needs the result of the previous iteration, so there is a sequential dependency.

7. (6 points) Write a multicore parallel program fragment using Parallel Java 2 to compute the square root function table. Your program fragment must use the correct Parallel Java 2 syntax.

```
double[] sqrtTable = new double [201];
sqrtTable[0] = 0.0;
parallelFor (1, 200) .exec (new Loop() {
   public void run (int i) {
      double x = i/10.0;
      double y = x/2.0;
      double ynext = (x/y + y)/2.0;
      while (Math.abs (2.0*(y – ynext)/(y + ynext)) >= 0.00001) {
         y = ynext;
         ynext = (x/y + y)/2.0;
      }
      sqrtTable[i] = y;
   }
});
```

8. (3 points) The parallel program is required to have a balanced load. To achieve a balanced load, do you have to add anything to the parallel program? If not, explain why not. If so, explain why, and describe what has to be added.

   Yes, a load balancing schedule (dynamic, proportional, or guided) must be added to the parallel for loop, because different iterations of the outer loop might take different amounts of time, because different *x* values might require different numbers of inner loop iterations to reach convergence.

   Questions 9–12. A multicore parallel program is given a point (a,b) and a series of points (x_i,y_i), $0 \le i \le n-1$. The program must find the index i such that the Euclidean distance d between (a,b) and (x_i,y_i) is minimized. The Euclidean distance between (x_1,y_1) and (x_2,y_2) is $((x_1 - x_2)^2 + (y_1 - y_2)^2)^{1/2}$.

9. (6 points) The program will use one and only one global reduction variable as well as one and only one per-thread reduction variable in each parallel team thread. A reduction variable class is needed. List the field or fields in the reduction variable class, and describe the information stored in each field.

   The reduction variable class needs two fields:
   – `minIndex` = index i such that the Euclidean distance d between (a,b) and (x_i,y_i) is minimized
   – `minDistance` = the minimized Euclidean distance d

10. (6 points) Write the Parallel Java 2 code for the reduction variable class. Include *only* the code for the class declaration, the field declaration(s), and the `reduce()` method. Do *not* include code for the rest of the class.

```
public class DistanceVbl implements Vbl {
   public int minIndex;
   public double minDistance;
   public void reduce (Vbl vbl) {
      DistanceVbl m = (DistanceVbl)vbl;
      if (m.minDistance < this.minDistance) {
         this.minIndex = m.minIndex;
         this.minDistance = m.minDistance;
      }
   }
}
```

11. (12 points) Write a Parallel Java 2 code fragment that does the program's computation. Assume that the following variables have been declared and initialized:

```
int n = ...;      // Number of points
double[] x = ...; // Array of X coordinates of the series of points
double[] y = ...; // Array of Y coordinates of the series of points
double a = ...;   // X coordinate of point being searched
double b = ...;   // Y coordinate of point being searched
```

    The program fragment must print the index i and the minimum distance d as defined above. The printout format is up to you.

    Do *not* write a complete program. Write *only* the piece that computes and prints the answer. However, if your program fragment needs additional variables, you must include the proper declarations and initializations for those variables. Also, your program fragment must use the correct Parallel Java 2 syntax.

```
   int n = ...;      // Number of points
   double[] x = ...; // Array of X coordinates of the series of points
   double[] y = ...; // Array of Y coordinates of the series of points
   double a = ...;   // X coordinate of point being searched
   double b = ...;   // Y coordinate of point being searched
```

```
DistanceVbl mindist = new DistanceVbl();

parallelFor (0, n – 1) .exec (new Loop() {
   DistanceVbl thrmindist;
   DistanceVbl dist;
   public void start() {
      thrmindist = threadLocal (mindist);
      thrmindist = Double.POSITIVE_INFINITY;
      dist = new DistanceVbl();
   }
   public void run (int i) {
      double dx = x[i] – a;
      double dy = y[i] – b;
      dist.minIndex = i;
      dist.minDistance = Math.sqrt (dx*dx + dy*dy);
      thrmindist.reduce (dist);
   }
});
System.out.printf ("minIndex = %d%n", mindist.minIndex);
System.out.printf ("minDistance = %.5f%n", mindist.minDistance);
```

12. (12 points) The preceding multicore parallel program is to be converted to run on a cluster of multicore nodes using Parallel Java 2. Describe the design of the cluster parallel version. State which class or classes are needed, describe what each class's purpose is, list the method or methods in each class, and describe what each method does.

Job class—The cluster parallel program itself

`main()` method—Partitions the `x` and `y` arrays into *K* equal-sized chunks, where *K* is the number of worker tasks; puts a chunk tuple for each worker containing the worker's rank, a chunk of `x`, a chunk of `y`, `a`, and `b`; sets up a task group with *K* worker tasks; sets up a reduction task to run in the job process when the other tasks finish

Worker task class—Performs a portion of the loop iterations

`main()` method—Takes the tuple corresponding to the worker's rank; does a parallel loop with reduction over the chunk of `x` and `y` to find the minimum index and minimum distance; puts a result tuple into tuple space containing the worker task's result

Reduction task class—Combines the worker tasks' results together

`main()` method—Repeat *K* times: take a worker task's result tuple and reduce it into the global minimum tuple; print the final result

Chunk tuple class—Contains the inputs needed by one worker task: rank, chunk of `x`, chunk of `y`, `a`, `b`

`matchContent()` method—Returns true (match) if template's rank equals tuple's rank

Result tuple class—Contains one worker task's results: minimum index, minimum distance

`reduce()` method—Combines two result tuples together, keeping the smallest distance and the corresponding index