Sample Exam Problems For the Material in "A Sophomoric Introduction to Shared-Memory Parallelism and Concurrency"

Solution:
Solutions Included

For more information, see http://www.cs.washington.edu/homes/djg/teachingMaterials.

Name:_____

1. In Java using the ForkJoin Framework, write code to solve the following problem:

   - Input: A `String[]`
   - Output: A `Pair` of (a) the number of words starting with the letter `'c'` and (b) the length of the longest word starting with the letter `'c'`. (If no words start with `'c'`, the length is irrelevant.)

   Your solution should have $O(n)$ work and $O(\log n)$ span where n is the array length. Do *not* employ a sequential cut-off: the base case should process one `String`.

   We have provided some of the code for you. You only need to provide one class definition:

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;
class Pair {
  int count;
  int longest;
  Pair(int c, int l) { count = c; longest = l; }
}
class Main{
  static final ForkJoinPool fjPool = new ForkJoinPool();
  Pair processCWords(String[] array) {
     return fjPool.invoke(new ProcessCWords(array,0,array.length));
  }
}
```

   Solution:

```
class ProcessCWords extends RecursiveTask<Pair> {
   String[] arr;
   int lo;
   int hi;
   ProcessCWords(String[] a, int l, int h) { arr=a; lo=l; hi=h; }
   public Pair compute() {
     if(hi==lo+1) {
       if(arr[lo].charAt(0)=='c')
         return new Pair(1,arr[lo].length());
       else
         return new Pair(0,0); // length < any word starting with 'c'
     } else {
        ProcessCWords left = new ProcessCWords(arr,lo,(hi+lo)/2);
        ProcessCWords right = new ProcessCWords(arr,(hi+lo)/2,hi);
        left.fork();
        Pair ans1 = right.compute();
        Pair ans2 = left.join();
        return new Pair(ans1.count+ans2.count,
                        Math.max(ans1.longest,ans2.longest));
     }
   }
}
```

It's also fine to mutate one `Pair` from a subproblem and return it. The code above will throw an exception if the array contains empty strings. We didn't take off for not noticing this, but we did give +0.5 for explicitly checking this case, without allowing more than a total of 10 points for the problem.

Name:_____

2. You are at a summer internship working on a program that currently takes 10 minutes to run on a 4-processor machine. Half the execution time (5 minutes) is spent running sequential code on one processor and the other half is spent running parallel code on all 4 processors. Assume the parallel code enjoys perfectly linear speedup for any number of processors.

   Note/hint/warning: This does *not* mean half the work is sequential. Half the *running time* is spent on sequential code.

   Your manager has a budget of $6,000 to speed up the program. She figures that is enough money to do only one of the following:

   - Buy a 16-processor machine.
   - Hire a graduate of this course to parallelize more of the program under the highly dubious assumptions that:
     - Doing so introduces no additional overhead
     - Afterwards, there will be 1 minute of sequential work to do and the rest will enjoy perfect linear speedup.

   Which approach will produce a faster program? Show your calculations, including the total *work* done by the program and the expected running time for both approaches.

   Solution:
   This is an application of Amdahl's Law. Let S be the running time for the sequential portion, L be the running time for the parallel portion on 1 processor and P be the number of processors. Then $T_p = S + L/P$. Initially T_P is 10 minutes, S is 5 minutes and $P = 4$, which means L is 20 minutes. So the total work is 20+5=25 minutes.

   Therefore, $T_{16} = 5 + 20/16$ is 6.25 minutes and that's the "buy a 16-processor" option.

   Under the "hire an intern" option S becomes 1 and L becomes 24 minutes. So the total time is $1+24/4$ which is 7 minutes.

   So the better choice is to buy the computer — apologies to those of you hoping for the job.

Name:_____

3. In Java using the ForkJoin Framework, write code to solve the following problem:

   - Input: An `int[]` (though the element type happens to be irrelevant)
   - Output: A new `int[]` where the elements are in the reverse order. For example, the element in the last array index of the input will be at index 0 in the output.

   Your solution should have $O(n)$ work and $O(\log n)$ span where n is the array length. Do *not* employ a sequential cut-off: the base case should process one array element.

   We have provided some of the code for you.

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveAction;

public class Main {
    static final ForkJoinPool fjPool = new ForkJoinPool();
    static int[] reverse(int[] array) {
        // ADD A LINE HERE
        fjPool.invoke(new Reverse(answer,array,0,array.length)); // DO NOT CHANGE
        // ADD A LINE HERE
    }
}

// DEFINE A CLASS HERE
```

   Solution:

```
    static int[] reverse(int[] array) {
        int [] answer = new int[array.length];
        fjPool.invoke(new Reverse(answer,array,0,array.length));
        return answer;
    }
class Reverse extends RecursiveAction {
    int[] out;
    int[] in;
    int lo;
    int hi;
    Reverse(int[] o, int[] i, int l, int h) {
        out = o; in = i; lo = l; hi = h;
    }
    public void compute() {
        if(hi==lo+1) {
            out[in.length-lo-1] = in[lo];
        } else {
            Reverse left  = new Reverse(out,in,lo,(hi+lo)/2);
            Reverse right = new Reverse(out,in,(hi+lo)/2,hi);
            left.fork();
            right.compute();
            left.join();
        }
    }
}
```

Name:_____

4. Suppose a program uses your solution to the previous problem to reverse an array containing 2^{27} elements.

   (a) In English, about how big is 2^{27}? For example, "one thousand" is an answer in the right form, but is the wrong answer.

   (b) When the program executes, how many fork-join threads will your solution to the previous problem create? Give both an exact answer and an approximate English answer.

   (c) Suppose you modify your solution to use a sequential cut-off of 1000. When this modified program executes, how many fork-join threads will it create? Give both an exact answer and an approximate English answer.

   Solution:

   (a) one hundred million (or one hundred twenty-eight million)

   (b) 2^{27}, one hundred million (If your answer to the previous question calls `right.fork` instead of `right.compute`, then the answer is $2^{28} - 1$, which is about two hundred fifty (six) million.)

   (c) 2^{18}, two hundred fifty (six) thousand (Note a very common mistake is answer twice as small, diving by 2^{10} instead of 2^9. Since 1000 is a bit less than 2^{10}, the code will solve problems of size 512 sequentially. This "off by 2x" error is a fairly minor error.)

Name:_____

5. In class we learned an algorithm for parallel prefix sum with $O(n)$ work and $O(\log n)$ span. In this problem, you will develop similar algorithms for parallel *suffix* sum, where element `i` of the output array holds the sum of the elements in indices greater than or equal to `i` in the input.

   (a) Describe in English or high-level pseudocode a two-pass algorithm to solve the parallel suffix sum problem. Assume the reader is familiar with the parallel prefix sum problem, so for any parts that are *exactly* the same, you can just say they are the same. Any parts that are different need a full explanation.

   (b) Suppose you needed an implementation of parallel suffix sum, but you already had available a library for parallel prefix sum. Describe an easier-to-implement algorithm for parallel suffix sum that uses parallel prefix sum and one or more other algorithms as subroutines while maintaining $O(n)$ work and $O(\log n)$ span.

   Solution:

   (a) Use an "up pass" and a "down pass" just like for parallel prefix. The "up pass" is identical, creating a binary tree with the sum for a range of the array at each node. For the "down pass" we pass a "fromRight" argument to each node. For the root, this value is 0. An internal node passes this value to its children as follows: The right child gets the same fromRight value and the left child gets the sum of the same fromRight value and the stored sum at the right child. At the leaf for range [lo,hi+1) (a one-element range), the fromRight value is the output for index lo.

   (b) Use three steps. First, reverse the array using your solution to problem 4. Second, perform parallel prefix sum on this array. Third, reverse the output of the second step and use this as the result. Each step is $O(n)$ work and $O(\log n)$ span, so the overall asymptotic work and span is the same.

   There is an alternate unanticipated solution for which we gave full credit if explained well: First do parallel prefix sum. Now perform a parallel map over the result where we take the total sum of the array and subtract from it the value in the array from the prefix sum (and then add back the value in the original input to produce an inclusive suffix sum). Note the total sum can be computed via its own reduce operation or just as the rightmost value from the prefix sum. Also note that for the map step, we can produce an inclusive sum by subtracting just the item one to the left (with a special case for index 0).

Name:_____

6. (a) Suppose we have an algorithm for finding the second-smallest element in a binary search tree. The code is somewhat complicated to account for all possible tree shapes, but it always descends to the correct node without making any modifications to the tree. Suppose the code is in a method `secondSmallestMidterm`. Would it be correct for two threads to execute `secondSmallestMidterm` concurrently? Explain briefly.

   (b) Here is a simpler algorithm for finding the second smallest element in a binary search tree in terms of some other operations:

```
synchronized E deleteMin() { ... }
synchronized E findMin() { ... }
synchronized void insert(E x) { ... }
E secondSmallestFinal() {
  E min = this.deleteMin();
  E ans = this.findMin();
  this.insert(min);
  return ans;
}
```

   Notice `deleteMin`, `findMin`, and `insert` are synchronized methods.

   i. Suppose two threads call `secondSmallestFinal` concurrently. Demonstrate how one of them can get the wrong answer.

   ii. Does the code above have any *data races*? Explain briefly.

   iii. What is the easiest way to fix `secondSmallestFinal`?

Solution:

(a) Yes, `secondSmallestMidterm` does no writes to shared memory, so multiple concurrent executions would not interfere. (Interleaving with other tree operations like `insert` would be a problem, but that is not the question here.)

(b) i. If thread 1 deletes the minimum element and then thread 2 runs `secondSmallestFinal`, it will actually get the 3rd smallest element (or raise an exception if there were only 2 elements to begin with).

ii. No, there is never unsynchronized access to the same field by two threads because all field access is by synchronized helper methods.

iii. Make it `synchronized` as well.

Name:_____

7. You are designing a new social-networking site to take over the world. To handle all the volume you expect, you want to support multiple threads with a fine-grained locking strategy in which each user's profile is protected with a different lock. At the core of your system is this simple class definition:

```
class UserProfile {
   static int id_counter;
   int id; // unique for each account
   int[] friends = new int[9999]; // horrible style
   int numFriends;
   Image[] embarrassingPhotos = new Image[9999];
   UserProfile() { // constructor for new profiles
     id = id_counter++;
     numFriends = 0;
   }
   synchronized void makeFriends(UserProfile newFriend) {
      synchronized(newFriend) {
        if(numFriends == friends.length
           || newFriend.numFriends == newFriend.friends.length)
          throw new TooManyFriendsException();
        friends[numFriends++] = newFriend.id;
        newFriend.friends[newFriend.numFriends++] = id;
      }
   }
   synchronized void removeFriend(UserProfile frenemy) {
      ...
   }
}
```

(a) The constructor has a concurrency error. What is it and how would you fix it? A short English answer is enough – no code or details required.

(b) The `makeFriends` method has a concurrency error. What is it and how would you fix it? A short English answer is enough – no code or details required.

(c) Rather than throwing an exception in `makeFriends` if an array is full, give two alternatives. Describe them only at a high level – a sentence or two is enough – without getting into any code details. One alternative should be easy and have nothing to do with concurrency. The other should involve concurrency.

Solution:

(a) There is a data race on `id_counter`. Two accounts could get the same `id` if they are created simultaneously by different threads. Or even stranger things could happen. You could synchronize on a lock for `id_counter` or mark `id_counter` `volatile`.

(b) There is a potential deadlock if there are two objects `obj1` and `obj2` and one thread calls `obj1.makeFriends(obj2)` when another thread calls `obj2.makeFriends(obj1)`. The fix is to acquire locks in a consistent order based on the `id` fields, which are unique.

(c) First, you could resize the array. Second, you could use *condition variables* to `wait` until there is room in the array and have `removeFriend` call `notifyAll`.

Note from instructor, not necessary for the exam: Getting this right is rather difficult since you need room in *both* arrays but it would be inefficient and deadlock-prone to wait on one object while still holding the lock for the other. Something like this pseudocode should work:

```
while(true) {
   synchronized(acct1) { if(acct1.friends is full) acct1.wait(); }
   synchronized(acct2) { if(acct2.friends is full) acct2.wait(); }
   synchronized(acct1) {
   synchronized(acct2) {
      if(acct1.friends is full || acct2.friends is full) continue;
      ... add to arrays ...
      break;
   }
 }
```

Name:_____

8. Answer "always" or "sometimes" or "never" for each of the following.

   (a) A program with data races also has bad interleavings.

   (b) A program with bad interleavings also has data races.

   (c) A program that correctly uses consistent locking (all thread-shared data is associated with a lock that is always held when accessing the data) has data races.

   (d) A program that correctly uses consistent locking (all thread-shared data is associated with a lock that is always held when accessing the data) has bad interleavings.

   (e) Java's `synchronized` statement will block if the thread executing it already holds the lock that is being acquired.

   (f) This code snippet using a condition variable `foo` is a bug: `if(someCondition()) foo.wait()`.

   Solution:

   (a) Sometimes

   (b) Sometimes

   (c) Never

   (d) Sometimes

   (e) Never

   (f) Always

Name:_____

9. Consider an adjacency-list representation of an undirected graph where if edge (u, v) is in the graph then v is in u's adjacency list and u is in v's adjacency list. (This supports efficient operations for "find all nodes adjacent to a given node.") Suppose we wish to support multiple threads accessing the graph concurrently such that every thread always sees a consistent state of the graph. Further suppose we use a locking strategy where each adjacency list uses a different reentrant lock (one lock for the whole adjacency list).

   (a) This pseudocode for deleting an edge from the graph — and doing nothing if the edge is not already present — has a concurrency error. Write down an interleaving that exhibits a concurrency error and describe what happens when the interleaving occurs. Assume `array` holds the adjacency lists, vertices are represented by ints, and `contains` and `delete` are methods on lists (the latter throwing an exception if the item is not present).

```
void deleteEdge(int u, int v) {
  synchronized(array[u]) {
    if(!array[u].contains(v))
      return;
  }
  synchronized(array[u]) { array[u].delete(v); }
  synchronized(array[v]) { array[v].delete(u); }
}
```

   (b) This pseudocode also has a concurrency error. Repeat the previous problem with this code:

```
void deleteEdge(int u, int v) {
  synchronized(array[u]) {
    synchronized(array[v]) {
       if(!array[u].contains(v))
         return;
       array[u].delete(v);
       array[v].delete(u);
    }
  }
}
```

   (c) Write a correct version of `deleteEdge` in the same style as the broken versions above. For simplicity, you can assume no other operations modify the graph (or that they are written using the same approach as your method).

Solution:
See next page.

Name:_____

(a) (Other answers are possible.) Here a bad interleaving causes an exception because we try to remove an edge that has already been removed. The exception occurs when Thread 1 does `array[u].delete(v)`.

```
Thread 1 (deleteEdge(u,v))          Thread 2 (deleteEdge(u,v))
--------------------------          --------------------------
  synchronized(array[u]) {
    if(!array[u].contains(v))
      return;
  }
                                     all of deleteEdge(u,v)
  synchronized(array[u]) {
    array[u].delete(v);
  }
```

(b) Here a bad interleaving causes a deadlock, so neither thread will run again and no other threads can get the two locks for these adjacency lists.

```
Thread 1 (deleteEdge(u,v))          Thread 2 (deleteEdge(v,u))
--------------------------          --------------------------
synchronized(array[u]) {            synchronized(array[v]) {
  synchronized(array[v]) { // blocks  synchronized(array[u]) { // blocks
```

(c)
```
void deleteEdge(int u, int v) {
  int a = u < v ? u : v;
  int b = u > v ? u : v;
  synchronized(array[a]) {
    synchronized(array[b]) {
       if(!array[a].contains(b))
         return;
       array[a].delete(b);
       array[b].delete(a);
    }
  }
}
```