Assignment 5

CSE 332: Data Abstractions, Spring 2011
University of Washington
May 26, 2011
due: Friday, June 3, 2:15 p.m.

Instructions: This assignment will be collected but not graded in the usual way. Rather, each paper will be skimmed and evaluated for effort, and a score between 0 and 20 will be given. Solutions will be available after the due date, in advance of the final exam.

Create a PDF representation of your answers and submit it via Catalyst CollectIt.

1. (Graphing Amdahl's Law).

   Use a graphing program such as a spreadsheet to plot the following implications of Amdahl's Law. Turn in the graphs and tables with the data.

   (a) Consider the speed-up (T_1/T_P) where $P = 256$ of a program with sequential portion S where the portion $1 - S$ enjoys perfect linear speed-up. Plot the speed-up as S ranges from 0.01 (one percent sequential) to 0.25 (25 percent sequential).

   (b) Consider again the speed-up of a program with sequential portion S where the portion $1 - S$ enjoys perfect linear speed-up. This time, hold S constant and vary the number of processors P from 2 to 32. On the same graph, show three curves, one each for $S = 0.01$, $S = 0.1$, and $S = 0.25$.

2. (Pattern matching with fork-join parallelism). Given a (very long) string T called the "text" and a (short) string P called the "pattern", the *string-matching problem* is to find substrings of T that are equal to P. Let n be the length of T. You can assume that the length of P is a constant. All of your algorithms below should have work $O(n)$ and span $O(\log n)$. Be sure to explain for each algorithm why this is true.

   (a) Describe a fork-join parallel algorithm that outputs the index of the leftmost occurrence of the pattern P in T, using a sequential cutoff of 1.

   (b) Describe a fork-join parallel algorithm that outputs an array of the indices of all occurrences of the pattern P in T, using a sequential cutoff of 1. (Hint: use Pack.)

   (c) What changes do you need to make to your solution in part (a) to use a more sensible sequential cutoff?

3. (Debugging a "concurrent" queue). Consider this Java implementation of a queue with two stacks. We do not show the entire stack implementation, but assume it is correct. Notice the stack has synchronized methods but the queue does not. The queue is incorrect in a concurrent setting.

```
class Stack<E> {
  ...
  synchronized boolean isEmpty() { ... }
  synchronized E pop() { ... }
  synchronized void push(E x) { ... }
}
```

```
class Queue<E> {
  Stack<E> in  = new Stack<E>();
  Stack<E> out = new Stack<E>();
  void enqueue(E x){in.push(x); }
  E dequeue() {
    if(out.isEmpty()) {
      while(!in.isEmpty()) {
        out.push(in.pop());
      }
    }
    return out.pop();
  }
}
```

(a) Show the queue is incorrect by showing an interleaving that meets the following criteria:

i. Only one thread ever performs `enqueue` operations and that thread enqueues numbers in increasing order (1, 2, 3, ...).

ii. There is a thread that performs two `dequeue` operations such that its first `dequeue` returns a number larger than its second `dequeue`, which should never happen.

iii. Every `dequeue` succeeds (the queue is never empty).

Your solution can use 1 or more additional threads that perform `dequeue` operations.

(b) A simple fix would make `enqueue` and `dequeue` synchronized methods, but this would never allow an `enqueue` and `dequeue` to happen at the same time. To allow an `enqueue` and a `dequeue` to operate on the queue at the same time (at least when `out` is not empty), we could try either of the approaches below for `dequeue`. For each, show an interleaving that demonstrates the approach is broken. Your interleaving should satisfy the three properties listed in part (a).

```
E dequeue() {
  synchronized(out) {
    if(out.isEmpty()) {
      while(!in.isEmpty()) {
        out.push(in.pop());
      }
    }
    return out.pop();
  }
}
```

```
E dequeue() {
  synchronized(in) {
    if(out.isEmpty()) {
      while(!in.isEmpty()) {
        out.push(in.pop());
      }
    }
   }
   return out.pop();
}
```

(c) Provide a solution, based on two stacks as above, that correctly allows an `enqueue` and a `dequeue` to operate on the queue at the same time, at least when `out` is not empty. Your solution should define `dequeue` and involve multiple locks.