CSE 332 Data Abstractions, Winter 2013
Homework 7

Due**: Friday, March 8, 2013** at the BEGINNING of lecture. Your work should be readable as well as correct. Homework Seven has four fun questions!! Please write your **section** at the top of your homework.

Problem 1: Amdahl's Law: Graphing the Pain

Use a graphing program such as a spreadsheet to plot the following implications of Amdahl's Law. For both part a and part b, turn in 1) the *graphs* and 2) *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 .01 (1% sequential) to .25 (25% 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 four curves, one each for S = .01, S = .1, S = .2, and S = .3.

Problem 2: Parallel Prefix and Filter

In this problem, the input is an array of strings and the output is an array of integers. The output has the length of each string in the input, but empty strings are filtered out.

For example, this input: `[ "", "", "cse", "rox", "", "homework", "", "7", "" ]`
Produces this output: `[ 3, 3, 8, 1]`

A parallel algorithm can solve this problem in O(log n) **span** and O(n) **work** by:

1) doing a parallel map to produce a bit vector,
2) doing a parallel prefix over the bit vector, and
3) doing a parallel map to produce the output.

Show a detailed walk through of the algorithm described above in the following way:

1) For step 1, give pseudocode for a fork join algorithm that does the mapping to a bit vector (Use an int array for the bit vector, one int per bit - do not do bitwise ops.)
2) For step 2, as done in lecture and section, in two phases, draw the actual prefix tree that would be created, including the fields in each node of the tree and their values for the specific example shown above. This can be two separate trees or done with different colors of ink and a careful description in the two phases ("up" and "down"). Note: because the input length is not a power of two, the tree will not have all its leaves at exactly the same height. (You do not have to show the fork-join code for this step, only the tree created and its contents.)
3) For step 3, give pseudocode for a fork join algorithm that does the mapping from the parallel prefix output to the final output as shown above.

It is fine to use .length() to find the length of Strings. Do not use a sequential cut-off for any of the three steps. The pseudocode in step 1 and step 3 should be general and applicable to other sample inputs. In step 1 and 3, be sure to describe each of the fields passed as parameters to your thread objects.

Problem 3: Parallel Quicksort

Lecture presented a parallel version of quicksort with best-case O($\log^2$n) span and O(n log n) work. This algorithm used parallelism for the two recursive sorting calls and the partition.

(a) For the algorithm from lecture, what is the asymptotic *worst-case* **span** and **work**. For both, state a recurrence and solve it – show your work solving the recurrence.
(b) Suppose we use the parallel partition part of the algorithm, but perform the two recursive calls *in sequence* rather than parallel. What is the asymptotic *worst-case* **span** and **work**? For both, state a recurrence and solve it – show your work solving the recurrence.

Problem 4: Another Wrong Bank Account

Note: The purpose of this problem is to show you something you should **not** do because it does **not** work. Consider this pseudocode for a bank account supporting concurrent access; assume that Lock is a valid locking class, although it is not in Java:

```
class BankAccount {
      private int balance = 0;
      private Lock lk = new Lock();
      int getBalance() {
            lk.acquire();
            int ans = balance;
            lk.release();
            return ans;
      }
      void setBalance(int x) {
            lk.acquire();
            balance = x;
            lk.release();
      }
      void withdraw(int amount) {
            lk.acquire();
            int b = getBalance();
            if(amount > b) {
                  lk.release();
                  throw new WithdrawTooLargeException();
            }
            setBalance(b - amount);
            lk.release();
      }
}
```

The code above is wrong if locks are not re-entrant. Consider the *absolutely horrible* idea of “fixing” this problem by rewriting the withdraw method to be:

```
void withdraw(int amount) {
      lk.acquire();
      lk.release();
      int b = getBalance();
      lk.acquire();
      if(amount > b) {
            lk.release();
            throw new WithdrawTooLargeException();
      }
      lk.release();
      setBalance(b - amount);
      lk.acquire();
      lk.release();
}
```

(a) Explain why this “fixed” code doesn't “block forever” (unlike the original code).
(b) Show this new approach is incorrect by giving an interleaving of two threads, both calling the new withdraw, in which a withdrawal is forgotten.