Parallel Programming

Midterm Exam

Friday, April 17, 2015

First Name: ______________________________

Last Name: ______________________________

Your points are *precious*, don't let them go to waste!

Your Name Work that can't be attributed to you is lost: write your name on each sheet of the exam.

Your Time All points are not equal. Note that we do not think that all exercises have the same difficulty, even if they have the same number of points.

Your Attention The exam problems are precisely and carefully formulated, some details can be subtle. Pay attention, because if you do not understand a problem, you can not obtain full points.

Exercise	**Points**	**Points Achieved**
1	10	
2	10	
Total	20	

Exercise 1: Associativity (10 points)

Definition (1 pt)

Give the definition of associativity. What does it mean for an operator `f` to be associative?

Associativity and commutativity (1 pt)

Give an example operator that is associative but that is *not* commutative.

An associative operator (8 pt)

Given two types `A` and `B`, and two *associative* operators `f` and `g` with the following signatures:

- `f: (A, A) => A`
- `g: (B, B) => B`

Define a third operator `h` with the following signature:

- `h: ((A, B), (A, B)) => (A, B)`

such that `h` is associative. (1 pt)

Then, prove that `h` is indeed associative. (7 pt)

Be **very precise** in your proof. At each step, specify which axiom, hypothesis, or previous result you use.

Exercise 2: Parallel Merge Sort (10 points)

In this exercise, you are required to design and analyze a parallel version of the merge sort algorithm. Consider the function `msort` shown below that implements a *sequential* merge sort algorithm for sorting a list of objects of type `T`. Recall that a merge sort algorithm partitions the input list in two parts (whose sizes differ by at most one), sorts each of the parts independently, and finally assembles the two sorted parts into a single sorted list.

```
def msort[T](less: (T, T) => Boolean)(l: List[T]): List[T] = {
  if (l.length <= 1) {
    l
  } else {
    val (first, second) = partition(l)
    merge(less)(msort(less)(first), msort(less)(second))
  }
}
```

The parameter `less` defines an ordering on the instances of type `T`, and takes *constant time*. Given below are the implementations of the function `merge` that combines two sorted lists into a single sorted list, and the function `partition`.

```
def merge[T](less: (T, T) => Boolean)(xs: List[T], ys: List[T]): List[T] = {
   (xs, ys) match {
     case (Nil, _) => ys
     case (_, Nil) => xs
     case (x :: xtail, y :: ytail) =>
       if (less(x, y))
         x :: merge(less)(xtail, ys)
       else
         y :: merge(less)(xs, ytail)
   }
 }

 def partition[T](l: List[T]): (List[T], List[T]) = {
    val n = l.length / 2;
   (l take n, l drop n)
 }
```

You are given a `parallel` construct that can execute two function calls in parallel:

```
def parallel[A, B](taskA: => A, taskB: => B): (A, B)
```

(a) Which *one* of the above three functions can you parallelize to obtain a span of $O(n)$ for the function `msort`, where n is the size of the input list ? You are allowed to introduce one or more `parallel` constructs but you are not allowed to use a different data structure (other than the linked list). Recall that *span* is (informally) the time taken by an algorithm when the number of parallel processors is unlimited. See below for a more formal definition.

(b) Implement the parallel version of the function that you identified in part (a).

(c) Prove that the span of the merge sort algorithm is O(n) when the implementation you gave for part (b) is used ?

For your reference, a brief formal definition of span, which was explained in the lectures, is given below. For an expression $f(e_1, \cdots, e_n)$, where f is a function or a primitive operation, $S(f(e_1, \cdots, e_n)) = S(e_1) + ... + S(e_n) + S(f)(v_1, \cdots, v_n)$, where v_i denotes the values of e_i. If f is a primitive operation on integers, then $S(f)$ is constant, regardless of v_i. If f is a user-defined function, then $S(f)(v_1, \cdots, v_n)$ is equal to the span of the body of f when the values of the parameters are $v_1, \cdots, v_n$. For the parallel construct, $S($`parallel(c1, c2)`$) = c_p + \max(S(c1), S(c2)))$, where c_p is a constant. For the if-then-else construct, $S($`if(e1) e2 else e3`$) = S(e1) + S(e2) + c$ if the condition $e1$ holds, otherwise is equal to $S(e1) + S(e3) + c$, where c is a constant. Since the match construct can be reduced to if-then-else, its span can be derived using the span of if-then-else.