Parallelism and Concurrency

Midterm Exam

Wednesday, April 18, 2018

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

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	20	
2	20	
3	20	
Total	60	

Exercise 1: Parallel Reduce-By-Key (20 points)

In this exercise you are tasked to implement a version of the reduce-by-key operation using `ParSeq.aggregate`. The function takes a sequence `data` of tuples consisting of keys and values, and a function `h` that *reduces* two values into one. As its output, the function returns the mapping from keys to the reduced subsequences of values in `data` which have their key in common.

Example

Given the following input sequence of tuples

```
(1, "do"), (2, "re"), (1, "mi"), (4, "fa"), (4, "so"), (2, "la"), (1, "ti")
```

and the reduction function `+` (i.e., string concatenation) we want to produce the mapping:

```
1 -> "domiti"
2 -> "rela"
4 -> "faso"
```

Question 1

Your task is to implement the reduce-by-key operation in a way that is *generic*, *efficient* and *parallel*. The operation takes as its inputs a `ParSeq` of `(K, V)` tuples, where the `K` component is the key, along with a reduction function `h` that takes two values of type `V` and returns a reduced `V`. Your implementation should reduce all the values in `data` that have the same key in the order that they occur in `data`, and produce a `Map` going from the keys to the reduced values.

Your method should be implemented in terms of the `ParSeq.aggregate` method. Remember that the `aggregate` method has certain requirements on its parameters, i.e., `z` and the two functions `f` and `g`.

Hint: You can refer to the appendix for an API of `Map` and other useful classes and methods.

Please answer on the next page.

```
def reduceByKey[K, V](data: ParSeq[(K, V)])(h: (V, V) => V): Map[K, V] =
```

Question 2

In general, which of the following properties should hold for the parameters `z`, `f` and `g` of `ParSeq.aggregate[S]`?

1. `f(g(u, v), x) == g(f(u, x), v)`
2. `g(u, z) == z`
3. `g(u, z) == u`
4. `f(f(z, x), x) == f(g(f(z, x), z), x)`
5. `(xs ++ ys).foldLeft(z)(f) == g(xs.foldLeft(z)(f), ys.foldLeft(z)(f))`
6. `xs.foldLeft(z)(f) == xs.foldRight(z) { (x, u) => f(u, x) }`

where `u, v: S`, `x: T` and `xs, ys: ParSeq[T]` are arbitrary.

Please circle the correct answer(s).

Exercise 2: Concurrency (20 points)

In concurrent systems, there is a common category of resources that require the system to satisfy the following two properties:

- **Property 1**: The resource can be read in parallel by multiple readers, however,
- **Property 2**: Writing to the resource requires exclusive access.

In another word, read-read may be overlapped, but read-write and write-write must be mutually exclusive.

One solution to implement the two properties is to use a read lock `rl`, a write lock `wl`, and a number `rn` to remember the number of concurrent readers. A tentative algorithm is given below:

```
val rl = new Lock   // read lock
val wl = new Lock   // write lock
var rn = 0          // readers count

def read(): Int = {
  rl.acquire()
  rn = rn + 1
  if (rn == 1) wl.acquire()

  // read resource, compute result, omitted

  rn = rn - 1
  if (rn == 0) wl.release()
  rl.release()

  result
}

def write(param: Int): Unit = {
  wl.acquire()
  rl.acquire()

  // write resource using param, omitted

  rl.release()
  wl.release()
}
```

Question 1

Does the algorithm above satisfy the two properties? Briefly explain.

Question 2

Is the code vulnerable to deadlocks ? If it is the case, describe an execution schedule that leads to a deadlock. Otherwise, motivate your answer.

Question 3

Improve `read` and `write` so that the two properties are satisfied and there is no deadlock. You may introduce extra variables.

Exercise 3: Parallel String Match (20 points)

Your task in this exercise is to count the number of appearance of a given word in a sentence. The function to implement has the following signature:

```
def count(word: Array[Char], sentence: Array[Char]): Int
```

Here are two examples where count returns 2:

```
count("you".toArray, "you know what you're doing".toArray) == 2
//                    ^^^ 1st one   ^^^ 2nd one

count("lala".toArray, "lalala".toArray) == 2
//                     ^^^^   1st one
//                       ^^^^ 2nd one
```

Instead of implementing `count` directly, you will have to implement two helper methods, `countWithin` and `countWithinPar`, which in addition to the two input arrays take a start index and an end index to define the slice of the `sentence` where the count should be performed. Afterwards, `count` can simply be implemented as:

```
def count(word: Array[Char], sentence: Array[Char]): Int =
  countWithin(word, sentence, 0, sentence.length)
```

Or, for a parallel implementation:

```
def countPar(word: Array[Char], sentence: Array[Char]): Int =
  countWithinPar(word, sentence, 0, sentence.length)
```

Please see the next pages for the questions.

Question 1

Implement a sequential version of `count` called `countWithin`, which takes as additional parameters the array indices at which to `start` and `end` the search within `sentence`. The index `start` is inclusive and `end` is exclusive. Your implementation should perform at most $O(n^2)$ work, where n is the size of the of the array slice, i.e., $n =$ `end` $-$ `start`.

```
def countWithin(word: Array[Char], sentence: Array[Char], start: Int, end: Int): Int =
```

Question 2

Implement an efficient parallel version of `countWithin` called `countWithinPar` using the `parallel` method presented in class (and in the appendix). Your implementation should be sequential for chunks of `THRESHOLD` characters or less. You can assume that the length of the `word` will always be strictly shorter than the `THRESHOLD`. Also, remember that the index `start` is inclusive and `end` is exclusive.

For this question, you can also assume that you have a correct sequential implementation of `countWithin` from question 1. Make sure to use it!

```
def countWithinPar(word: Array[Char], sentence: Array[Char], start: Int, end: Int): Int =
```

Question 3

What is the *depth* of `countPar` in terms of Big-O notation, assuming unbounded parallelism and constant size words and `THRESHOLD`? Briefly explain.

Remember that depth is defined as the length of the longest series of operations that have to be performed sequentially.

Appendix

The provided API is not meant to be exhaustive. You can, of course, use methods and classes that are not presented in this API.

Parallel API

- `def parallel[A, B](op1: => A, op2: => B): (A, B)`: Executes the two given computations in parallel and returns the pair of results.

Array API

Relevant API for `Array[T]`:

- `def apply(index: Int): T`: Gets the element at a given index. Indexes start at 0.
- `def length: Int`: Returns the length of the array.

ParSeq API

Relevant API for `ParSeq[A]`. You can assume that the implementations of the following methods are similar to what was seen during the lectures.

- `def map[B](f: (A) => B): ParSeq[B]`: Applies, in parallel, a function on all elements of the sequence, and returns the sequence of results.
- `def reduce(op: (A, A) => A): A`: Reduces the elements of this sequence using the specified operator.
- `def aggregate[B](z: => B)(f: (B, A) => B, g: (B, B) => B): B`: Aggregates the results of applying an operator to the elements.
- `def foldLeft[S](z: S)(f: (S, T) => S): S`: Folds the elements of this sequence sequentially from left to right using the specified operator `f`.
- `def foldRight[S](z: S)(f: (T, S) => S): S`: Folds the elements of this sequence sequentially from right to left using the specified operator `f`.

Map API

Relevant API for `Map[K, V]`. A `Map[K, V]` is an immutable collection mapping keys `K` to values `V` and supports the following operations efficiently:

- `def ++(other: Map[K, V]): Map[K, V]`: Concatenates two mappings. The second mappings has priority over the first in the case of duplicate keys.
- `def +(elem: (K, V)): Map[K, V]`: Appends an entry in the mappings. If the key previously existed, it is overridden.

- `def contains(key: K): Boolean`: Checks whether a given key exists in this mapping.
- `def apply(key: K): V`: Retrieve the value associated to the key if it exists, otherwise throws an exception.
- `def get(key: K): Option[V]`: Retrieve the value associated to the key, wrapped in `Some`, if it exists, otherwise returns `None`.
- `def keys: Seq[K]`: Returns all the map's keys in some unspecified order.

To create an empty `Map`, you can use the following method:

- `Map.empty[K, V]: Map[K, V]`: Creates an empty map.

Seq API

Relevant minimal API for `Seq[A]`.

- `def map[B](f: (A) => B): ParSeq[B]`: Applies, in parallel, a function on all elements of the sequence, and returns the sequence of results.
- `def foldLeft[S](z: S)(f: (S, T) => S): S`: Folds the elements of this sequence sequentially from left to right using the specified operator `f`.
- `def foldRight[S](z: S)(f: (T, S) => S): S`: Folds the elements of this sequence sequentially from right to left using the specified operator `f`.