Parallelism and Concurrency

Final Exam

Wednesday, May 31, 2017

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.

Closed-book The exam is closed book.

No communication You are not allowed to use internet, mobile phones, laptops, smart watches, etc.

Exercise	**Points**	**Points Achieved**
1	30	
2	30	
3	40	
Total	100	

Exercise 1: Water Level using Parallel Collections (30 points)

Consider a 2D hilly landscape, such as the one in Figure 1, made of rocks so solid that the water basically does not permeate the soil. After a heavy rain, the valleys of that landscape have been completely flooded, with the water level indicated by the dotted lines.

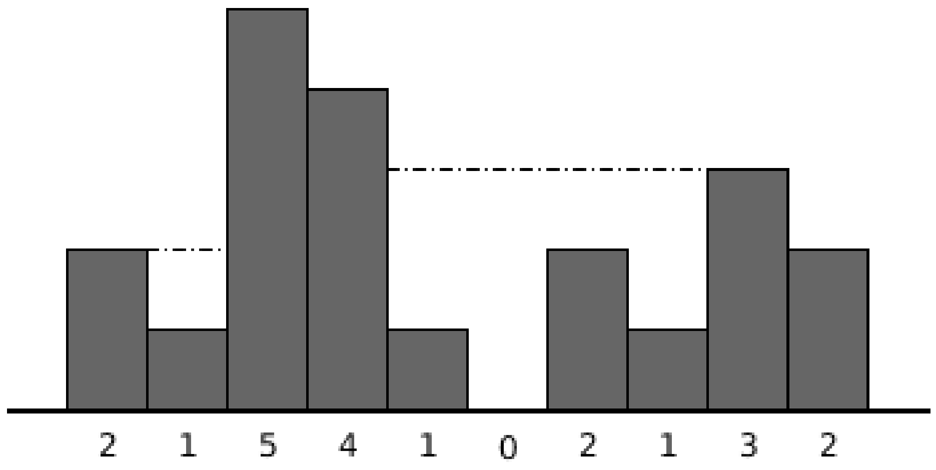

Figure 1: A hilly landscape with water

As you can see, water is retained as high as the borders on both sides will permit. Any excessive rain, however, flows on one or both sides, and eventually reaches the permeable soil outside of the hilly landscape, disappearing.

Question 1

We want to implement the method `waterLevels`, which takes a `ParSeq[Int]` indicating the heights of the hills on the landscape, and returning a `ParSeq[Int]` of the water levels at each position. You may assume that the heights given as input are always greater or equal to zero. The water levels are relative to the height of the hill at each point, so that they are indicative of how much water is retained (and not of how high above sea level it goes). The signature of the method is:

```
def waterLevels(hillHeights: ParSeq[Int]): ParSeq[Int]
```

For the above landscape, an invocation would be:

```
waterLevels(ParSeq(2, 1, 5, 4, 1, 0, 2, 1, 3, 2))
```

and its result should be:

```
ParSeq(0, 1, 0, 0, 2, 3, 1, 2, 0, 0)
```

Implement the method `waterLevels` so that it is parallel and efficient. To be precise, its *work* should be no more than $O(n)$ and its *depth* should be no more than $O(\log n)$, where n is the size of the input sequence. You do not have to actually prove nor even argue that those properties hold, though (only the code matters).

Hint: At any given position on the landscape, we can determine the height reached by the water (above sea level) with the following algorithm:

- Look left for the highest hill, note its height.
- Look right for the highest hill, note its height.
- Take the minimum of those two heights.

Naively repeating that algorithm for each point trivially gives a sequential algorithm in $O(n^2)$. That is clearly far from answering the question, but the property might be useful (probably will) to devise your own algorithm.

Hint: You might find useful to use a combination of `scanLeft`, `scanRight` and `zip`, as well as other more "common" operations on `ParSeq`, in your solution. The signature of those methods are given in the API at the end of this exam. You will also find example usage of those functions there.

Please answer on this page.

Question 2

In this second part, we are interested in counting how much water can be retained by the terrain. To be precise, we are interested in how many 1×1 squares are filled with water after a flood.

For instance, in the case of Figure 1, the number of water squares would be 9.

Complete the implementation of the `waterAmount` function, which returns the amount of water that can be retained by the terrain given as input. **You may assume that you have a correct implementation of the `waterLevels` function for the purpose of this question.**

Your implementation should be efficient and parallel. To be precise, it should have *work* $O(n)$ and *depth* $O(\log n)$, where n is the size of the input sequence.

```
def waterAmount(hillHeights: ParSeq[Int]): Int = ???
```

Hint: Note that the heights of the hills is given as a parameter, not the water levels.

Please answer on this page.

Exercise 2: Consensus using Futures (30 points)

In this exercise you'll be dealing with a consensus problem: a number of threads running in the same instance of a java virtual machine have to agree upon a conclusion together. Each thread may return a boolean value that says if they agree or disagree with a decision, the role of the consensus implementation is to return true if the **strict majority** of the results are true, and false otherwise.

You'll need to implement following function:

```
def consensus(futures: Seq[Future[Boolean]]): Future[Boolean] = ???
```

The input `futures` sequence contains all the decisions of the different threads, each wrapped in a `Future`.

The resulting `Future` should be successful, regardless how many of the input `Future`s fail. The value returned by the resulting future should be `true` if the strict majority of the input futures successfully returned the value `true`, and `false` otherwise.

Notes:

- In your solution you must never synchronously wait for all results to complete. The goal here is to complete the resulting `Future` as soon as possible, even if parts of the system are unreasonably slow.
- You do **not** have to worry about canceling the remaining `Future`s once you have a definite result.
- If one of the results completes with a failure, it's equivalent to completing with a `false` success.
- You may use mutable state in your implementation. If you do, make sure that your solution is not subject to concurrency problems. You may use any appropriate construct seen in this class to avoid the concurrency issues.

Additional help: The Promise:

One useful construct in Scala you may not have seen yet is the concept of a `Promise`. A `Promise` is an object which can be completed with a value at a later time. It is created as follows:

```
// Creating the promise.
val promise: Promise[Boolean] = Promise()
```

At any future time, the `Promise` can be given a value using the method `.success(value)`. This method can be called at most once! Calling this method multiple times will result in an exception.

```
// In this example, we give it the value ‘true‘.
promise.success(true)
```

A `Promise` can be very simply turned into a `Future`. The resulting `Future` will complete once a value is given to the `Promise`. Obviously, the call to `.future` can be done even if `.success` hasn't been called yet on the `Promise`.

```
// The future will complete once the promise is given a value using ‘success‘.
val myFuture: Future[Boolean] = promise.future
```

We strongly suggest that you create a `Promise` at the start of your function and return the corresponding `Future` using `.future`.

Please answer on the next page.

Exercise 3: Fighting Fake News using Spark (40 points)

In this exercise, you will use Spark to fight *fake news* on an imaginary social media website called *Bleeper*.

Bleeper is a social media platform on which users can publicly share messages and news articles, as well as follow other users. In the recent months, dubious news article have been spreading on the social media platform, which is detrimental to the brand image of *Bleeper*. You have been hired to end this problem.

In order to fight the propagation of fake news on the network, you decide to come up with a reputation system for users of the platform based on the following observation: **Following someone reputable tends to make you reputable.**

For this exercise, you have at your disposition the following types and (persisted) RDDs:

```
type UserID = Long
case class UserInfo(username: String, yearJoined: Int, locationCode: Int)

// All users of the website.
// Keys are distinct.
val users: RDD[(UserID, UserInfo)] = ...

// Contains a record (a, b) if and only if ‘a‘ is followed by ‘b‘.
// Does not contain duplicates.
// Has the same partitioner as ‘users‘.
val isFollowedBy: RDD[(UserID, UserID)] = ...
```

Question 1: Followers

In this first part, you are asked to efficiently compute the following RDD:

```
val allFollowers: RDD[(UserID, Iterable[UserID])] = ???
```

The RDD should contain, for each user that has at least one follower, the collection of all their followers. Note that if user `A` is followed by a user `B`, then we say that `B` is a follower of `A`.

Importantly, the `allFollowers` RDD should propagate the partioner of `isFollowedBy`, if any. Note also that **this RDD will be used multiple times during this exercise**.

Please answer below.

Question 2: Initial reputation

In this second question, you will have to efficiently compute the `initialReputations` RDD, that should assign an initial reputation to all users. The reputation of a user is either *preset*, which means that it was manually set, or *derived*, which means it was computed.

```
sealed abstract class Reputation {
  val value: Double
}
case class Preset(value: Double) extends Reputation
case class Derived(value: Double) extends Reputation

// Contains a predefined reputation value for a subset of users.
// Does not contain duplicates keys.
// Has the same partitioner as ‘users‘.
val presetUsers: RDD[(UserID, Double)] = ...


val initialReputations: RDD[(UserID, Reputation)] = ???  // Implement this.
```

If a user is part of `presetUsers`, it should have reputation `Preset(value)` in `initialReputations`, where `value` is the value assigned to that user in `presetUsers`. On the contrary, if the user is not part of `presetUsers`, it should have reputation `Derived(0.0)` in `initialReputations`.

The `initialReputations` RDD should propagate the partitioner of `users`, if any. This RDD will be used exactly once.

Please answer below.

Question 3: Deriving the reputation

In the last question, we gave an initial reputation to users. Using this initial reputation, you will have to compute a derived reputation, according to the following procedure:

- For users with a `Preset` reputation, their reputation should remain unchanged.
- For users with a `Derived` reputation, their new reputation should be `Derived(newValue)`, where `newValue` is **80% of the mean reputation value of all the people they follow**. If the user doesn't follow anyone, their reputation should be set to `Derived(0.0)`.

Complete the following `iterate` function, which takes a RDD of user reputations and returns the new user reputations according to the above procedure.

```
def iterate(oldReputs: RDD[(UserID, Reputation)]): RDD[(UserID, Reputation)] = ???
```

The `iterate` function will be used as follows to compute the final reputation of users:

```
var finalReputations = initialReputations
for (i <- 1 to 20) {
  finalReputations = iterate(initialReputations)
}
```

Note that the `finalReputations` RDD should have the same `Partitioner` as `oldReputs`, if any.

Please answer below.

Question 4: Most reputable users

Finally, you should compute the 200 people with the largest derived reputation (i.e. a reputation of the form `Derived`) in `finalReputations`, along with their reputation value.

```
val mostReputableUsers: Array[(UserID, Double)] = ???
```

Please answer below.

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. Assuming applications of parameter functions take constant time, all the below methods have *work* $O(n)$ and depth $(\log n)$, where n is the size of the collection.

- `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 scanLeft[B](z: B)(op: (B, A) => B): ParSeq[B]`: Produces a collection containing cumulative results of applying the operator **going left to right**. See below for example usage.
- `def scanRight[B](z: B)(op: (A, B) => B): ParSeq[B]`: Produces a collection containing cumulative results of applying the operator **going right to left**. See below for example usage.
- `def zip[B](that: ParSeq[B]): ParSeq[(A, B)]`: Produces a collection containing the pairs of elements from `this` and `that`. See below for example usage.
- `def tail: ParSeq[A]`: Returns the collection of all elements except the first.
- `def init: ParSeq[A]`: Returns the collection of all elements except the last.

Example usage of some methods

```
val xs = ParSeq(1, 2, 3, 4)

xs.scanLeft(10)(_ + _) == ParSeq(10, 11, 13, 16, 20)  // Returns true.
xs.scanRight(10)(_ + _) == ParSeq(20, 19, 17, 14, 10) // Returns true.

val ys = ParSeq("a", "b", "c")

xs.zip(ys) == ParSeq((1, "a"), (2, "b"), (3, "c"))  // Returns true.
```

Future API

Relevant API for `Future[+T]`:

- `def map[S](f: T => S): Future[S]`: Creates a new future by applying a function to the successful result of this future.
- `def flatMap[S](f: T => Future[S]): Future[S]`: Creates a new future by applying a function to the successful result of this future, and returns the result of the function as the new future.
- `def filter(p: (T) => Boolean): Future[T]`: Creates a future that will fail when the successful result of this future does not satisfy the given predicate.
- `def onComplete[U](f: Try[T] => U): Unit`: Registers a function to be executed when this future is completed, either successfully or not.

Spark API

Relevant API for Spark `RDD[T]`:

- `def collect(): Array[T]`: Return an array that contains all of the elements in this RDD.
- `def count(): Long`: Return the number of elements in the RDD.
- `def distinct(): RDD[T]`: Return a new RDD containing the distinct elements in this RDD.
- `def filter(f: (T) => Boolean): RDD[T]`: Return a new RDD containing only the elements that satisfy a predicate.
- `def flatMap[U](f: (T) => TraversableOnce[U]): RDD[U]`: Return a new RDD by first applying a function to all elements of this RDD, and then flattening the results.
- `def fold(zeroValue: T)(op: (T, T) => T): T`: Aggregate the elements of each partition, and then the results for all the partitions, using a given function and "zero value".
- `def groupBy[K](f: (T) => K): RDD[(K, Iterable[T])]`: Return an RDD of grouped items.
- `def map[U](f: (T) => U): RDD[U]`: Return a new RDD by applying a function to all elements of this RDD.
- `def reduce(f: (T, T) => T): T`: Reduces the elements of this RDD using the specified commutative and associative binary operator.
- `def sortBy[K](f: (T) => K, ascending: Boolean = true): RDD[T]`: Return this RDD sorted by the given key function.
- `def take(num: Int): Array[T]`: Take the first num elements of the RDD.
- `def union(other: RDD[T]): RDD[T]`: Return the union of this RDD and another one.

Additional methods available to RDDs of type `RDD[(K, V)]`:

- `def groupByKey(): RDD[(K, Iterable[V])]`: Group the values for each key in the RDD into a single sequence.
- `def join[W](other: RDD[(K, W)]): RDD[(K, (V, W))]`: Return an RDD containing all pairs of elements with matching keys in this and other.
- `def leftOuterJoin[W](other: RDD[(K, W)]): RDD[(K, (V, Option[W]))]`: Perform a left outer join of this and other.
- `def mapValues[U](f: (V) => U): RDD[(K, U)]`: Pass each value in the key-value pair RDD through a map function without changing the keys.
- `def partitionBy(partitioner: Partitioner): RDD[(K, V)]`: Return a copy of the RDD partitioned using the specified partitioner.
- `def reduceByKey(func: (V, V) => V): RDD[(K, V)]`: Merge the values for each key using an associative reduce function.
- `def rightOuterJoin[W](other: RDD[(K, W)]): RDD[(K, (Option[V], W))]`: Perform a right outer join of this and other.