Parallelism and Concurrency

Exam Solution

Wednesday, May 31, 2017

Exercise 1: Water level (30 points)

```
def waterLevels(levels: ParSeq[Int]): ParSeq[Int] = {
  val leftMaxs = levels.scanLeft(0)(Math.max)
  val rightMaxs = levels.scanRight(0)(Math.max)

  val mins = leftMaxs.tail.zip(rightMaxs).map {
    case (maxLeft, maxRight) => Math.min(maxLeft, maxRight)
  }

  levels.zip(mins).map {
    case (level, min) => min - level
  }
}


def waterAmount(hillHeights: ParSeq[Int]): Int =
  waterLevels(hillHeights).fold(0)(_ ++ _)
```

Exercise 2: Exercise 2 (30 points)

```
import scala.concurrent._
import scala.concurrent.ExecutionContext.Implicits.global
import scala.util._

def consensus(futures: Seq[Future[Boolean]]): Future[Boolean] = {
  val lock = new AnyRef {}

  val promise: Promise[Boolean] = Promise()

  var nSuccess = 0
  var nFailure = 0

  val successThreshold = futures.size / 2 + 1
  val failureThreshold = futures.size - successThreshold + 1

  futures.foreach { future =>
    future.onComplete {
      case Success(true) => lock.synchronized {
        nSuccess += 1
        if (nSuccess == successThreshold) {
          promise.success(true)
        }
      }
      case _ => lock.synchronized {
        nFailure += 1

        if (nFailure == failureThreshold) {
          promise.success(false)
        }
      }
    }
  }

  promise.future
}
```

Exercise 3: Exercise 3 (40 points)

Question 1

```
val allFollowers = isFollowedBy.groupByKey().cache()
```

Question 2

```
val initialReputation: RDD[(UserID, Reputation)] = users.leftOuterJoin(presetUsers).mapValues {
  case (_, Some(reput)) => Preset(reput)
  case (_, None) => Derived(0.0)
}
```

Question 3

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

  val meanReputationOfFollowed = allFollowers.join(oldReputs).flatMap {
    case (user, (followers, reput)) => followers.map {
      follower => (follower, (reput.value, 1))
    }
  }.reduceByKey {
    case ((r1, n1), (r2, n2)) => (r1 + r2, n1 + n2)
  }.mapValues {
    case (r, n) => r / n
  }

  oldReputs.leftOuterJoin(meanReputationOfFollowed).mapValues {
    case (Preset(v), _) => Preset(v)
    case (Derived(_), None) => Derived(0.0)
    case (Derived(_), Some(mean)) => Derived(0.8 * mean)
  }
}
```

Question 4

```
val mostReputableUsers: Array[(UserID, Double)] = finalReputations.flatMap {
  case (_, Preset(_)) => Seq()
  case (_, Derived(v)) => Seq(v)
}.top(Ordering.by {
  case (id, score) => score
})
```