Parallelism and Concurrency

Final Exam - Solutions

Wednesday, May 29, 2019

Manage your time All points are not equal. We do not think that all exercises have the same difficulty, even if they have the same number of points.

Follow instructions The exam problems are precisely and carefully formulated, some details can be subtle. Pay attention, otherwise you will lose points.

Refer to the API The last page of this exam is a small API. Please consult it before you reinvent the wheel. Feel free to detach it. You are free to use methods that are *not* part of this API provided they exist in the standard library.

Exercise	**Points**	**Points Achieved**
1	15	
2	15	
3	15	
4	15	
Total	60	

Exercise 1: Prefix-Average (15 points)

In this exercise, you will be implementing a parallel version of the `averages` function. The function takes as input a mutable array of numbers and modifies its content so that all elements are equal to the average of input values up to that point. A sequential version of `averages` could be written as:

```
def averages(numbers: Array[Long]): Unit = {
  var i: Int = 0
  var sum: Long = 0
  var count: Long = 0
  while (i < numbers.length) {
    count += 1
    sum += numbers(i)
    numbers(i) = sum / count
    i += 1
  }
}
```

The parallel version of `averages` that you are implementing in this exercise consists of two separate phases: An *upwards* phase and a *downwards* phase. During the upwards phase, a binary tree containing counts and sums is built on top of the array. During the downwards phase, that tree is traversed and values of the input array are updated.

```
def averages(numbers: Array[Long]): Unit = {
  val tree: Tree = upwards(numbers, 0, numbers.length)
  downwards(numbers, tree, 0, 0)
}
```

The datatype `Tree` is defined as:

```
sealed abstract class Tree {
  val count: Int
  val sum: Long
}

case class Branch(left: Tree, right: Tree) extends Tree {
  val count = left.count + right.count
  val sum = left.sum + right.sum
}

case class Leaf(count: Int, sum: Long) extends Tree
```

Question 1.1 (8 points)

Implement the recursive function `upwards`. The function constructs, in parallel, a tree of type `Tree` that summarises a slice of the `numbers` array. The precise requirements of `upwards` are:

- The `count` field of the returned tree should be equal to the number of elements in the slice.
- The `sum` field of the returned tree should be equal to the sum of elements in the slice.
- If the size of the slice is strictly above the value `THRESHOLD` then the returned tree should be a `Branch`, otherwise it should be a `Leaf`.
- In case of a `Branch`, the two subtrees should not differ in `count` by more than 1.
- The parameter index `from` is inclusive, while the parameter `until` is exclusive.
- You should use the `parallel` construct to parallelise recursive calls to the function.
- The `numbers` array should not be changed by `upwards`.
- All calls should refer to the same `numbers` argument.

Solution:

```
def upwards(numbers: Array[Long], from: Int, until: Int): Tree = {
  val size = until - from
  if (size <= THRESHOLD) {
    var i: Int = from
    var sum: Long = 0
    while (i < until) {
      sum += numbers(i)
      i += 1
    }
    Leaf(size, sum)
  }
  else {
    val mid = from + size / 2
    val (left, right) = parallel(
      upwards(numbers, from, mid),
      upwards(numbers, mid, until))
    Branch(left, right)
  }
}
```

Question 1.2 (7 points)

Implement the recursive function `downwards`. In addition to the array to be updated and the tree, the function takes two parameters that correspond to the count and respectively the sum of all elements appearing to the left of the currently considered slice. The two parameters are initially set to zero, but are to be appropriately updated in the recursive calls. The function should follow the structure of the input `tree`. In the case of a `Branch`, recursive calls should be made in parallel.

Solution:

```
def downwards(numbers: Array[Long],
              tree: Tree,
              countLeft: Int,
              sumLeft: Long): Unit = tree match {
    case Leaf(size, _) => {
      var i = countLeft
      val until = countLeft + size
      var sum = sumLeft
      while (i < until) {
        sum += numbers(i)
        numbers(i) = sum / (i + 1)
        i += 1
      }
    }
    case Branch(left, right) => {
      parallel(
        downwards(numbers, left, countLeft, sumLeft),
        downwards(numbers, right, countLeft + left.count, sumLeft + left.sum))
    }
```

Exercise 2: Futures (15 points)

In this exercise you will need to use futures to concurrently execute several long operations at the same time.

Consider the following binary tree definition that only has values in the leaves.

```
sealed trait Tree[+T]
case object EmptyTree extends Tree[Nothing]
case class Leaf[+T](value: T) extends Tree[T]
case class Branch[+T](left: Tree[T], right: Tree[T]) extends Tree[T]
```

In this particular tree the order of the values in the leaves is important and goes from left to right. For example, in the following tree the order of the values is `a, b, c, d`.

```
Branch(
  Branch(
    Branch(Leaf("a"), EmptyTree),
    Branch(Leaf("b"), Leaf("c"))
  ),
  Branch(EmptyTree, Leaf("d"))
)
```

We want to get the first value in the tree that satisfies a predicate. Usually a predicate would be of type `T => Boolean`, but in this case we will use `T => Future[Boolean]`. We return a `Future[Boolean]` because we assume that this predicate will take a large amount of time to compute (but not necessarily computationally expensive).

```
sealed trait Tree[+T] {
  /** Returns a future of the first value in the tree that satisfies the future
   *  predicate `p`. The future should succeed with a `Some` containing the leftmost
   *  value for which the predicate holds. Or it should succeed with the value
   *  `None` if there is no value for which the predicate holds.
   *
   *  The implementation is non-blocking and will evaluate all predicates
   *  asynchronously at the same time. Any computation that does not depend
   *  the computation of a predicate should be eagerly computed and create
   *  a successful future directly.
   *  Futures may depend on several futures, in which case the future finishes
   *  as soon as enough information is available. It may spawn more futures
   *  than necessary that will just be ignored (not cancelled).
   */
  def first(p: T => Future[Boolean]): Future[Option[T]]
}
```

Hint: Read carefully the specification in the documentation above.

Note: Use the functions in the API to create futures.

Question 2.1 (3 points)

Implement `first` on an empty tree.

Solution:

```
case object EmptyTree extends Tree[Nothing] {
  def first(p: Nothing => Future[Boolean]): Future[Option[Nothing]] = {
    Future.successful(None)
  }
}
```

Question 2.2 (4 points)

Implement `first` on a leaf.

Solution:

```
final case class Leaf[+T](value: T) extends Tree[T] {
  def first(p: T => Future[Boolean]): Future[Option[T]] = {
    p(value).map(b => if (b) Some(value) else None)
  }
}
```

Question 2.3 (8 points)

Implement `first` on a branch.

```
final case class Branch[+T](left: Tree[T], right: Tree[T]) extends Tree[T] {
  def first(p: T => Future[Boolean]): Future[Option[T]] = {
    // Start both futures
    val leftFuture = left.first(p)
    val rightFuture = right.first(p)
    leftFuture.flatMap {
      case Some(v) =>
        // If the left one returns a result,
        // return it (without starting a new future)
        // and ignore ‘rightFuture‘
        Future.successful(Some(v))
      case None =>
        // If the left one returns no result,
        // return the future result in ‘rightFuture‘
        rightFuture
    }
  }
}
```

Exercise 3: Concurrency (15 points)

Consider a chat system implement using `Akka` actors. In this system, each *client* has a corresponding actor. The code for the clients' actors is not shown. Clients can message directly other clients.

In addition to client actors, the system also has *broadcast* actors. Clients can subscribe or unsubscribe to a broadcast actor by respectively sending it `Subscribe` or `Unsubscribe`. When a client sends a `Message` to a broadcast actor, the broadcast actor immediately sends it to all currently subscribed actors.

Messages

```
case object Subscribe
case object Unsubscribe
case class Message(text: String)
```

Question 3.1 (7 points)

Consider the following scenarios. In each case, answer the question and motivate your answer.

Scenario 1

Consider a system with only two client actors. The first actor sends two messages to the second. Can the messages be received in a different order they were sent? *Explain your answer.*

Solution: No, Akka guarantees that any two messages sent from an actor A to an actor B will be received in the same order they were sent.

Scenario 2

Consider a system with two client actors and a single broadcast actor. The two client actors are currently subscribed to the broadcast actor. The first client actor sends two messages to the broadcast actor. Can the messages be received by the second client actor in a different order they were sent by the first client actor? *Explain your answer.*

Solution: The answer is also no. The two messages sent by the first client to the broadcast actor will be received in order, and since the broadcast actor immediately broadcasts the messages it receives, it will forwards these two messages to the second client in order, who will then receive them also in order.

Scenario 3

Consider a system with two client actors and a single broadcast actor. The two client actors are currently subscribed to the broadcast actor. The first client actor sends a message to the broadcast actor and a message directly to the second client actor. Can the messages be received by the second client actor in a different order they were sent by the first client actor? *Explain your answer.*

Solution: Yes, the second client will receive one message from the broadcast actor and one message from the first client, because these two messages were sent from different actors, Akka does not guarantee anything about the order in which they are received.

Question 3.2 (8 points)

Your goal in this question is to implement the broadcaster actor class. The requirements are:

- Upon reception of `Subscribe`, the sender should be considered *subscribed* to this actor.
- Upon reception of `Unsubscribe`, the sender should no longer be considered *subscribed*.
- Upon reception of a `Message`, the message should immediately be sent to all actors currently *subscribed* to this actor.

You are free to implement the actor state using either variables or `become`.

Solution:

```
class Broadcaster extends Actor {

  var subscribed: Set[ActorRef] = Set.empty

  def receive: Receive = {
    case Subscribe => subscribed = subscribed + sender
    case Unsubscribe => subscribed = subscribed - sender
    case Message(text) => subscribed.foreach(_ ! Message(text))
  }
}
```

Exercise 4: Spark (15 points)

In this exercise, your goal is to estimate the runtime cost of Spark programs executed in a cluster consisting of 8 nodes. We assume that runtime cost is dominated by the amount of data that is being transferred over the network.

The data consists of a set of conference attendees, modeled by the following case class:

```
case class Person(id: Long, age: Int, affiliation: Option[Int])
```

Here is a sample of 20 entries from the input dataset. Please use this sample to infer characteristics about the whole dataset. For instance, we can infer that about 50% of the population is unaffiliated, and there are about 10 different affiliations.

```
Person(id =  5232, age = 30, affiliation = None)
Person(id =  4949, age = 40, affiliation = Some(4))
Person(id =  2909, age = 53, affiliation = None)
Person(id = 18436, age = 19, affiliation = None)
Person(id =  2955, age = 19, affiliation = None)
Person(id = 19700, age = 48, affiliation = Some(3))
Person(id =  3028, age = 29, affiliation = Some(5))
Person(id = 14451, age = 20, affiliation = None)
Person(id =  2679, age = 24, affiliation = None)
Person(id =  4613, age = 51, affiliation = None)
Person(id = 17301, age = 40, affiliation = Some(8))
Person(id =  5919, age = 56, affiliation = Some(3))
Person(id =  7146, age = 34, affiliation = Some(9))
Person(id = 11668, age = 36, affiliation = None)
Person(id = 13705, age = 50, affiliation = Some(0))
Person(id =  8468, age = 26, affiliation = Some(5))
Person(id =  4398, age = 50, affiliation = None)
Person(id =  1677, age = 22, affiliation = Some(0))
Person(id =  8307, age = 41, affiliation = Some(5))
Person(id =  2801, age = 18, affiliation = Some(2))
```

For each question, estimate how much data is transferred over the network when executing the given query with an input RDD that consists of 1 Terabyte of data partitioned uniformly at random across the 8 nodes of the cluster. Briefly justify your answer.

Please ignore all potential overhead due to serialization and any potential gains due to data compression. For instance, you can assume that Spark uses exactly 12 bytes to transfer an instance of `(Int, Long)` over the network. Note that we don't expect a bit-precise estimation, a rough estimation backed with a good explanation will be graded with the maximum score.

Question 4.1 (3 points)

```
// Counts the unaffiliated participants over 25 years old
def Q2(in: RDD[Person]): Long = {
  in.map(p => (p.age, p.affiliation))
    .filter(p => p._2.isEmpty)
    .filter(p => p._1 > 25)
    .count()
}
```

Solution: 7 Longs = 64 bytes

Question 4.2 (4 points)

```
// Counts the unaffiliated participants over 25 years old (parallelize)
def Q1(in: RDD[Person]): Long = {
  val s1 = in.map(p => (p.age, p.affiliation)).collect()
  val s2 = sparkContext.parallelize(s1).filter(p => p._2.isEmpty).collect()
  sparkContext.parallelize(s2).filter(p => p._1 > 25).count()
}
```

Solution:

1st collect 7/8 * 3/4 of a TB (assuming Tuple2[Int, Option[Int]] takes about 3/4 the memory of a Person)

1st parallelize 7/8 * 3/4 of a TB

2nd collect 1/2 * 7/8 * 3/4 of a TB (assuming 50% of the input is unaffiliated)

2nd parallelize 1/2 * 7/8 * 3/4 of a TB

Total About 2TB

Question 4.3 (4 points)

```
// Compute the average age per affiliation (groupByKey)
def Q3(in: RDD[Person]): Seq[(Int, Long)] = {
  in.filter(p => p.affiliation.nonEmpty)
    .map(p => (p.affiliation.get, p))
    .groupByKey()
    .map(p => (p._1, (p._2.size, p._2.map(_.age).sum)))
    .mapValues(s => s._2 / s._1)
    .collect()
}
```

Solution: groupByKey: 7/8 * 5/4 of a TB (assuming Tuple2[Int, Person] takes about 5/4 of the memory of a Person)

Question 4.4 (4 points)

```
// Compute the average age per affiliation (reduceByKey)
def Q4(in: RDD[Person]): Seq[(Int, Long)] = {
  in.filter(p => p.affiliation.nonEmpty)
    .map(p => (p.affiliation.get, (1, p.age)))
    .reduceByKey((x, y) => (x._1 + y._1, x._2 + y._2))
    .mapValues(s => s._2 / s._1)
    .collect()
}
```

Solution: reduceByKey: 8 * 10 * sizeOf(Tuple2[Int, Long]) = 80 * 64 = 640 bytes

Appendix

Future

Relevant API for `Future[T]`:

- `def onComplete(callback: Try[T] => Unit): Unit`: When this future is completed, either through an exception, or a value, apply the provided function.
- `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 new future by filtering the value of the current future with a predicate.
- `def map[S](f: T => S): Future[S]`: Creates a new future by applying a function to the successful result of this future.
- `def foreach(f: T => Unit): Unit`: Asynchronously processes the value in the future once the value becomes available.

Relevant API for the `Future` object:

- `def apply[T](body: => T): Future[T]`: Starts an asynchronous computation and returns a `Future` instance with the result of that computation.
- `def successful[T](result: T): Future[T]`: Creates an already completed Future with the specified result.

A `Future` represents a value which may or may not **currently** be available, but will be available at some point, or an exception if that value could not be made available.

Option

Relevant API for `Option[+T]`:

- `def isEmpty: Boolean`: Returns `true` if the option is `None`, `false` otherwise.
- `def get: T`: Returns the option's value. Throws an exception in case of `None`.
- `def orElse[R >: T](alternative: => Option[R]): Option[R]`: Returns `this Option` if it is nonempty, otherwise returns the result of evaluating `alternative`.

Constructors for `Option[+T]`:

- `case class Some[+T](value: T) extends Option[T]`: Represents existing values of type `T`.
- `case object None extends Option[Nothing]`: Represents non-existent values.

Set

Relevant API for `Set[T]`:

- `def +(elem: T): Set[T]`: Adds an element to the set.
- `def -(elem: T): Set[T]`: Removes an element from the set.
- `def foreach(action: T => Unit): Unit`: Executes an action for each element of the set.

Relevant API for the `Set` object:

- `def apply[T](elems: T*): Set[T]`: Creates a set with the specified elements.
- `def empty[T]: Set[T]`: Returns an empty set.

Spark

Relevant API for Spark `RDD[T]`:

- `def collect(): Array[T]`: Returns an array that contains all of the elements in this RDD.
- `def count(): Long`: Returns the number of elements in the RDD.
- `def filter(f: (T) => Boolean):  RDD[T]`: Returns a new RDD containing only the elements that satisfy a predicate.
- `def map[U](f: (T) => U): RDD[U]`: Returns a new RDD by applying a function to all elements of this RDD.

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

- `def groupByKey(): RDD[(K, Iterable[V])]`: Groups the values for each key in the RDD into a single sequence.
- `def mapValues[U](f: (V) => U): RDD[(K, U)]`: Passes each value in the key-value pair RDD through a map function without changing the keys.
- `def reduceByKey(func: (V, V) => V): RDD[(K, V)]`: Merges the values for each key using an associative reduce function.

Relevant API for `SparkContext`:

- `def parallelize[T](seq: Seq[T]): RDD[T]`: Distribute a local collection to form an RDD.