Exercise 1 : Parallel Encoding

Note: This exercise was already given last week. If you have already done it, briefly discuss the solution as a group before moving to the next exercise.

In this exercise, your group will devise a parallel algorithm to encode sequences using the run-length encoding scheme. The encoding is very simple. It transforms sequences of letters such that all subsequences of the same letter are replaced by the letter and the sequence length. For instance:

"AAAAATTTGGGGTCCCAAC" ⇒ "A5T3G4T1C3A2C1"

Your goal in this exercise is to come up with a parallel implementation of this algorithm. The function should have the following shape:

```
def rle(data: ParSeq[Char]): Buffer[(Char, Int)] =
  data.aggregate(???)(???, ???)
```

The `Buffer` class is already given to you. A buffer of type `Buffer[A]` represents sequences of elements of type `A`. It supports the following methods, all of which are efficient:

```
def isEmpty: Boolean  // Checks if the buffer is empty.
def head: A           // Returns the first element of the buffer.
def tail: Buffer[A]   // Returns the buffer minus its first element.
def last: A           // Returns the last element of the buffer.
def init: Buffer[A]   // Returns the buffer minus its last element.
def ++(that: Buffer[A]): Buffer[A] // Concatenate two buffers.

Buffer.empty[A]: Buffer[A]  // Returns an empty buffer.
Buffer.singleton[A](element: A): Buffer[A]  // Single element buffer.
```

A solution:

```
def rle(data: ParSeq[Char]): Buffer[(Char, Int)] = {

  def g(as: Buffer[(Char, Int)], bs: Buffer[(Char, Int)]) =
    if (as.isEmpty || bs.isEmpty || as.last._1 != bs.head._1) {
      as ++ bs
    }
    else {
      as.init ++
      Buffer.singleton((as.last._1, as.last._2 + bs.head._2)) ++
      bs.tail
    }

  def f(acc: Buffer[(Char, Int)], x: Char) =
    if (acc.isEmpty || acc.last._1 != x) {
      acc ++ Buffer.singleton((x, 1))
    }
    else {
      acc.init ++ Buffer.singleton((x, acc.last._2 + 1))
    }

  val z: Buffer[(Char, Int)] = Buffer.empty

  data.aggregate(z)(f, g)
}
```

Exercise 2 : Parallel Two Phase Construction

In this exercise, you will implement an array `Combiner` using internally a double linked list (DLL). Below is a minimal implementation of the `DLLCombiner` class and the related `Node` class. Your goal for this exercise is to complete the implementation of the (simplified) `Combiner` interface of the `DLLCombiner` class.

```
class DLLCombiner[A] extends Combiner[A, Array[A]] {
  var head: Node[A] = null  // `null` for empty lists.
  var last: Node[A] = null // `null` for empty lists.
  var size: Int = 0

  // Implement these three methods...
  override def +=(elem: A): Unit = ???
  override def combine(that: DLLCombiner[A]): DLLCombiner[A] = ???
  override def result(): Array[A] = ???
}

class Node[A](val value: A) {
  var next: Node[A]  // `null` for last node.
  var previous: Node[A]  // `null` for first node.
}
```

A solution:

```
class DLLCombiner[A] extends Combiner[A, Array[A]] {
  var head: Node[A] = null  // `null` for empty lists.
  var last: Node[A] = null // `null` for empty lists.
  var size: Int = 0

  override def +=(elem: A): Unit = {
    val node = new Node(elem)
    if (size == 0) {
      head = node
      last = node
      size = 1
    }
    else {
      last.next = node
      node.previous = last
      last = node
      size += 1
    }
  }
```

```
  override def combine(that: DLLCombiner[A]): DLLCombiner[A] = {

    if (this.size == 0) {
      that
    }
    else if (that.size == 0) {
      this
    }
    else {
      val combined = new DLLCombiner

      this.last.next = that.head
      that.head.previous = this.last

      combined.size = this.size + that.size
      combined.head = this.head
      combined.last = that.last

      combined
    }

  }

  // This is not implemented in parallel yet.
  override def result(): Array[A] = {
    val data = new Array[A](size)

    var current = head
    var i = 0
    while (i < size) {
      data(i) = current.value

      i += 1
      current = current.next
    }
    data
  }
}
```

Question 1

What computational complexity do your methods have ? Are the actual complexities of your methods acceptable according to the `Combiner` requirements ?

The complexity of += is constant, as well as the complexity of `combine`. This is obviously well within the desired complexity range. The `result` function takes time linear in the size of the data, which is acceptable according to the `Combiner` requirements. However, the `result` function should work in parallel according to the contract. This isn't the case here.

Question 2

One of the three methods you have implemented, `result`, should work in parallel according to the `Combiner` contract. Can you think of a way to implement this method efficiently using 2 parallel tasks ?

The idea is to copy the double linked list to the array from both ends at the same time. For this, we create a task that handle the second half of the array, while the current thread copied the first half.

```
override def result(): Array[A] = {
  val data = new Array[A](size)
  val mid = size / 2

  // This is executed on a different thread.
  val taskEnd = task {
    var i = size - 1
    var current = last
    while (i >= mid) {
      data(i) = current.value
      current = current.previous
      i -= 1
    }
  }

  // This is executed on the current thread.
  var i = 0
  var current = head
  while (i < mid) {
    data(i) = current.value
    current = current.next
    i += 1
  }
  taskEnd.join()
  data
}
```

Question 3

Can you, given the current internal representation of your combiner, implement `result` so that it executes efficiently using 4 parallel tasks ? If not, can you think of a way to make it possible ?

Hint: This is an open-ended question, there might be multiple solutions. In your solution, you may want to add extra information to the class `Node` and/or the class `DLLCombiner`.

Solution:

*The actual answer to this question is: **it depends**. Two see why, we first make the following observation:*

All implementations of the `result` function must consist of primarily two operations:

1. Moving to the next node in the list, and,
2. Copying the value of the node to the array.

Depending on the actual cost of the two operations, one may devise schemes that can make efficient use of more than two threads. For instance, assume for a moment that copying a value to the array is significantly costlier than moving to the next node in the list. In this case, we could execute the function efficiently in parallel by spawning multiple threads starting from the head of the list, each handling a disjoint set of indexes (for instance, one thread takes indexes of the form 4n*, another* 4n + 1 *and so on).*

On the other hand, if we assume that moving to the next node in the list has a cost comparable to the one of copying a value to the array, then finding such a strategy is more challenging, or even impossible.

However, there are ways to circumvent this problem by modifying the data structure used. One way could be to keep track of the middle of the double linked lists. The result function could then execute in parallel on 4 different threads by copying the array from both ends and from the middle (in both directions) simultaneously. The problem would then be to efficiently maintain the pointer to the middle of the list, which might not be a trivial task when combining arbitrary lists together. If you are interested in learning more about such data-structures, we encourage you to look up the skip list *data structure, which generalises on this idea.*

Another solution would be to modify the nodes so that they also point to their successor's successor and their predecessor's predecessor. This way, two threads could start from the start of the list and two from the end. In each case, one thread would be responsible for odd indexes and the other for even ones. This solution does not change at all the complexity of the various `Combiner` operations, but requires a bit more bookkeeping.

Exercise 3: Pipelines

In this exercise, we look at pipelines of functions. A pipeline is simply a function which applies its argument successively to each function of a sequence. To illustrate this, consider the following pipeline of 4 functions:

```
val p = toPipeline(Seq(_ + 1, _ * 2, _ + 3, _ / 4))
```

The pipeline `p` is itself a function. Given a value `x`, the pipeline `p` will perform the following computations to process it. In the above example,

```
p(x) = (((x + 1)        Application of first function
        * 2)            Application of second function
        + 3)            Application of third function
        / 4             Application of fourth function
```

In this exercise, we will investigate the possibility to process such pipelines in parallel.

Question 1

Implement the following `toPipeline` function, which turns a parallel sequence of functions into a pipeline. You may use any of the parallel combinators available on `ParSeq`, such as the parallel `fold` or the parallel `reduce` methods.

```
def toPipeline(fs: ParSeq[A => A]): A => A = ???
```

Hint: Functions have a method called `andThen`, which implements function composition: it takes as argument another function and also returns a function. The returned function first applies the first function, and then applies the function passed as argument to that result. You may find it useful in your implementation of pipeline.

Solution:

```
def toPipeline(fs: ParSeq[A => A]): A => A =
  if (fs.isEmpty) {
    (x: A) => x
  }
  else {
    fs.reduce(_ andThen _)
  }
```

Question 2

Given that your `toPipeline` function works in parallel, would the pipelines it returns also work in parallel ? Would you expect pipelines returned by a sequential implementation of `toPipeline` to execute any slower ? If so, why ?

Discuss those questions with your group and try to get a good understanding how what is happening.

Even though the pipeline is constructed in parallel, ***it will not itself execute in parallel****. The resulting pipeline still has to apply its argument to all the functions it contains sequentially. This is due to the fact that the `andThen` method simply returns a function that will apply the first function and then the second, sequentially.*

Question 3

Instead of arbitrary functions, we will now consider functions that are constant everywhere except on a finite domain. We represent such functions in the following way:

```
class FiniteFun[A](mappings: immutable.Map[A, A], default: A) {

  def apply(x: A): A = {
    mappings.get(x) match {
      case Some(y) => y
      case None    => default
    }
  }

  def andThen(that: FiniteFun[A]): FiniteFun[A] = ???
}
```

Implement the `andThen` method. Can pipelines of such finite functions be efficiently constructed in parallel using the appropriately modified `toPipeline` method ? Can the resulting pipelines be efficiently executed ?

Solution:

```
def andThen(that: FiniteFun[A]): FiniteFun[A] = {
  val newDefault = that(default)

  val newMappings = for {
    (x, y) <- mappings
    val z = that(y)
    if (z != newDefault)
  } yield (x, z)

  new FiniteFun(newMappings, newDefault)
}
```

Pipelines of such functions can be efficiently constructed in parallel, as was the case for "normal" functions also. Also, interestingly, the resulting pipeline can be executed efficiently. The execution time of the pipeline does not depend on the number of functions in the pipeline, only on the lookup time in the finite map `mappings` (which can be nearly constant time if the underlying map is a hashtable). The size of this map is upper bounded by the size of the mappings of the functions in the pipeline.

Question 4

Compare the *work* and *depth* of the following two functions, assuming infinite parallelism. For which kind of input would the parallel version be asymptotically faster?

```
def applyAllSeq[A](x: A, fs: Seq[FiniteFun[A]]): A = {
  // Applying each function sequentially.
  var y = x
  for (f <- fs) y = f(y)
  y
}

def applyAllPar[A](x: A, fs: ParSeq[FiniteFun[A]]): A = {
  if (fs.isEmpty) x
  else {
    // Computing the composition in parallel.
    val p = fs.reduce(_ andThen _)
    // Applying the pipeline.
    p(x)
  }
}
```

Solution:

To simplify the analysis, we will assume that lookup in the mappings takes constant time, and thus that applying a `FiniteFun` *also takes constant time. Let's also assume that* `fs` *is of size* n *for both functions.*

Since the function is purely sequential, the work and depth of `applyAllSeq` *are equal. They amount to n applications of a finite function, which is linear in* n.

For `applyAllPar`*, things are a bit more complex. Let's denote by* d *the size of the largest domain of all functions passed as argument.*

The depth of the function is simply the depth of computing the pipeline (`fs.reduce(_ andThen _)`*) plus a constant for applying the pipeline. Assuming infinite parallelism, this results in a depth that is in* $O(\log_2(n) \cdot d)$.

The work of `applyAllPar` *is significantly more than its depth, and can be upper bounded by* $O(n \cdot d)$. *Indeed, there are* n *applications of the* `andThen` *method, each of which takes* O(d) *time.*

When d *is a constant, then the parallel version will be asymptotically faster than its sequential counterpart. If* d *is exponentially larger than n, then the sequential version is expected to perform better.*