Charles Zheng CME 323 HW 1

1.

Checkpoint on slide 11:

```
res0: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Checkpoint on slide 55:

```
(SparkCamp,4)
(Spark,3)
(spark,1)
(SparkSQL,1)
(../spark/bin/spark-submit,1)
```

Code for slide 60:

```
val rdd1 = sc.textFile("README.md").filter(_ contains "Spark")
val rdd2 = sc.textFile("spark/docs/contributing-to-spark.md").filter(_ contains "Spar
val wc1 = rdd1.flatMap(l => l.split(" ")).map(w => (w, 1)).reduceByKey(_ + _)
val wc2 = rdd2.flatMap(l => l.split(" ")).map(w => (w, 1)).reduceByKey(_ + _)
val joined = wc1.join(wc2)
```

Checkpoint on slide 60:

```
(Spark,(3,2))
```

2.

The mapper emits one key-value output pair for every input pair of vertices: the output key is the *sorted* vertices and the output value indicates the direction of the edge. The mapper takes directed edge $\langle a, b\rangle$: if $a < b$, it emits $\langle (a, b), 1\rangle$ and if $a > b$, it emits $\langle (b, a), 2\rangle$.

The reducer sees all the values $v_1, \ldots, v_n$ for a given key (c, d). If $\{1, 2\} \subseteq \{v_1, \ldots, v_n\}$ then it emits (c, d); otherwise it emits nothing.

No combiner is used; combiners are not likely to help in this problem.

Algorithm 1 Map

---

function MAP($\langle a, b\rangle$)
  if $a < b$ **then**
    Emit $\langle (a, b), 1\rangle$
  else
    Emit $\langle (b, a), 2\rangle$
  end if
end function

Algorithm 2 Reduce

```
function REDUCE(Key (c, d), Values {v_1, ..., v_n})
    if n < 2 then return
    end if
    for i = 2, ..., n do
        if v_i ≠ v_{i−1} then
            Emit ⟨c, d⟩
            return
        end if
    end for
end function
```

3.

The combiner is the same as the reducer. Let N be the total number of words.

Algorithm 3 Map

```
function MAP(String s)
    for Word w in s do
        Emit ⟨w, 1⟩
    end for
end function
```

Algorithm 4 Reduce/Combine

```
function REDUCE(Key w, Values {v_1, ..., v_n})
    s ← 0
    for i = 1, ..., n do
        s ← s + v_i
    end for
    Emit ⟨w, s⟩
end function
```

Without combiners– The shuffle size is N, and the reduce takes $N \pm O(B)$ operations.

With combiners– After the combine step, there are at most k key-value output pairs, since ther are at most k distinct keys. The shuffle size is kB, and the reduce takes $kB \pm O(B)$ operations.

4.

Let me briefly state the naive solution, which does not parallelize effectively. Run one map-reduce to count the number of elements N. Next, map each input pair $\langle i, a_i \rangle$ to $N-i+1$ output pairs $\langle i, a_i \rangle, \langle i+1, a_{i+1} \rangle, \ldots, \langle N, a_N \rangle$. Reduce by summing all values for a given key. With combiners, the shuffle size is N, and the number of reduce operations is NB where B is the number of mappers. The problem with this solution is that each mapper requires $O(N)$ storage to hold the output keys–but this is on the same order as the size of the entire data.

A better idea is to use divide-and-conquer. The idea is to divide the key-set into B equally-sized partitions: $P_1 = \{1, \ldots, n_1\}, P_2 = \{n_1+1, \ldots, n_2\}, \ldots, P_B = \{n_{B-1} + 1, \ldots N\}$. Accordingly define

$$\phi(i) = b \text{ such that } i \in P_b.$$

Then define the partial sums $p_1, \ldots, p_B$ by

$$p_b = \sum_{i \in P_b} a_i,$$

and define quantities

$$u_b = \sum_{c<b} p_b.$$

For $i = 1, \ldots, n$ define the within-partition prefix sums as

$$t_i = \sum_{j \in P_{\phi(i)} : j \leq i} a_j$$

Now observe that

$$s_i = t_i + u_{\phi(i)}$$

Therefore the procedure is as follows

1. (Map/Reduce 1) Count the number of keys N
2. (Map 2) Input: the original key-value pairs. Partition the keys sequentially into B workers
3. (Reduce 2) Each worker $b = 1, \ldots, B$ computes p_b and sends it to the driver
4. The driver computes $u_b = \sum_{c<b} p_c$ and sends u_b to each worker b for $b = 1, \ldots, B$.

5. (Reduce 3) Input: the output of step 2. Each worker $b = 1, \dots, B$ computes $s_i = u_b + t_i$ and emits $\langle i, s_i \rangle$ for each $i \in P_b$.

To formalize this procedure in the map/reduce framework we have to designate the partition number b as a key throughout steps 2-5. In steps 2 and 5, we have (i, a_i) as values. Note that step 5 uses the same input as step 3.

Algorithm 5 Step 2: Map 2

```
Parameters n_1, ..., n_{B-1} determined in Step 1, and n_B = N.
function MAP(⟨i, a⟩ from original inputs)
    for b ∈ 1, ..., B do
        if n_b > i then
            Emit ⟨b, (i, a)⟩
            return
        end if
    end for
end function
```

Algorithm 6 Step 3: Reduce 2

```
function REDUCE(Key b, values (i, a) from step 2)
    p ← 0
    for (i, a) in values do
        p ← p + a
    end for
    Emit ⟨b, p⟩
end function
```

Algorithm 7 Step 5: Reduce 3

```
Parameters u_1, ... u_B computed by driver in step 3.
function REDUCE(Key b, values (i, a) from step 2)
    Sort values (i, a) by i
    s ← u_b
    for Value (i, a) in sorted list do
        s ← s + a
        Emit ⟨i, s⟩
    end for
end function
```

The cost of the computation is dominated by step 2, when the data is partitioned: this requires a shuffle size of N. This is followed by a reduce in step 3 requiring $O(N/B)$ operations and $O(1)$ space. The driver has to complete $O(B)$ operations in step 4. Finally, each worker has to complete $O(N/B)$ operations in step 5, requiring $O(1)$ memory. The overall number of Map/Reduce iterations is 3, including the initial count.