CME 323: Distributed Algorithms and Optimization

Instructor: Reza Zadeh (rezab@stanford.edu)

HW#3 – Due Thursday May 23 11:59pm (on Gradescope)

1. **Intro to Spark** Download the following materials.

   - Slides
   - In this question, you will need to use two files, `README.md` and `contributing-to-spark.m` You can get these files from either build usb or Google Colab.
   - For Spark, you can either install Spark locally (refer to the instructions on the Spark Website), or use Google Colab colab link. We have provided the configuration in the Colab file. However, Colab currently does not support Scala, the language used in the slides, so you will need to write Python code to achieve the same tasks.

   Now, answer the following questions.

   (a) Checkpoint on slide 11.

   Solution

   ```
   val data = 1 to 10000
   val distdata = sc.parallelize(data)
   distdata.filter(_ < 10).collect()

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

   (b) Checkpoint on slide 55.

   Solution

   ```
   val f = sc.textFile("README.md")
   val wc = f.flatMap(l => l.split(" ")).
               map(word => (word, 1)).
               reduceByKey(_ + _)
   wc.filter(_._1 == "Spark").collect()

   res0: Array[(String, Int)] = Array((Spark,18))
   ```

   (c) Checkpoint on slide 60. Note: slide 59 references the file `CONTRIBUTING.md` which is in the provided zip file. Instead, use the file `website/getting-started.md`

Solution

```
sc.textFile("README.md").
    union(sc.textFile("docs/contributing-to-spark.md")).
    flatMap(_.split(" ")).
    filter(_ == "Spark").
count()

res2: Long = 20
```

2. Write a Spark program to find the *least squares fit* on the following 10 data points. The variable y is the response variable, and X1, X2 are the independent variables.

```
              X1         X2           y
 [1,] -0.5529181 -0.5465480 0.009519836
 [2,] -0.5428579 -1.5623879 0.982464609
 [3,] -1.3038629  0.5715549 0.499441144
 [4,]  0.6564096  1.1806877 0.495705999
 [5,] -1.2061171  1.3430651 0.153477135
 [6,]  0.2938439 -1.7966043 0.914381381
 [7,] -0.2578953  0.2596407 0.815623895
 [8,]  0.9659582  2.3697927 0.320880634
 [9,] -0.4038109  0.9846071 0.488856619
[10,]  0.6029003 -0.3202214 0.380347546
```

More precisely, find w_1, w_2, such that $\sum_{i=1}^{10}(w_1 X1_i + w_2 X2_i - y_i)^2$ is minimized. Report w_1, w_2, and the Root Mean Square Error and submit code in Spark. Analyze the resulting algorithm in terms of all-to-all, one-to-all, and all-to-one communication patterns.

Solution

```
import breeze.linalg.DenseVector

val x1 =  Array(-0.5529181, -0.5428579, -1.3038629,  0.6564096,  -1.2061171, 0.2938439, -0.2578953,  0.
val x_2 = Array(-0.5465480, -1.5623879, 0.5715549,  1.1806877,  1.3430651,-1.7966043,  0.2596407,  2.36
val y = Array(0.009519836, 0.982464609 , 0.499441144, 0.495705999 , 0.153477135 , 0.914381381 , 0.81562

val pts_cent = Array(x1, x2, y).transpose
val pts = sc.parallelize(pts_cent).cache

val w = DenseVector(0.0, 0.0)
val w_bc = sc.broadcase(w)
val step = 0.1
val max_iter = 1000

for (i <= i to max_iter){
```

```
val grad_x1 = pts.map(x => 2*(w_bc.value(0)*x(0) + w_bc.value(1)*x(1) - x(2))*x(0)}.reduce(_+_)
val grad_x2 = pts.map(x => 2*(w_bc.value(0)*x(0) + w_bc.value(1)*x(1) - x(2))*x(1)}.reduce(_+_)
w_bc.value(0) = w_bc.value(0) - step*grad_x1
w_bc.value(1) = w_bc.value(1) - step*grad_x2
}
```

Computing each of the gradients requires an all-to-one communication (due to the `.reduce(_+_))`. There are two of these per iteration. Broadcasting the updated `w_bc` requires one to all communication.

3. **Intro to Map Reduce** Assume you are given a typical MapReduce implementation where you only have to write the Map and Reduce functions. The Map function you will write takes as input a (key, value) record and returns either a (key, value) record or nothing. The Reduce function you will write takes as input (key, list of all values for that key) and returns either a record or nothing. The framework already takes care of iterating the Map function over all the records in the input file, key-based intermediate data transfer between Map and Reduce, and storing the returned value of Reduce. For all the following questions, provide algorithms at the level of pseudocode.

   (a) Given as set of records (for example, movie names and ranking), provide a MapReduce algorithm to output the top K movies of the set.

   Solution The algorithm is summarized in algorithms 1 and 2.

Algorithm 1: MAP(name, ranking)

```
1 $queue \leftarrow PriorityQueue()$;
2 Function MAP(name, ranking):
3     emit($(ranking, name)$);
```

   (b) Suppose you are given an input file which contains comprehensive information about a social network that has asymmetrical (directed) links, i.e., a network where users follow other users but not necessarily vice-versa (e.g., Twitter). Each record in this input file is (userid-a, userid-b), where userid-a follows userid-b (i.e., points to it). Note that this record tells you nothing about whether or not userid-b follows userid-a. Write a MapReduce program (i.e., Map function and Reduce function) that outputs all pairs of userids who follow each other.

   Solution The algorithm is summarized in algorithms 3 and 4.

4. **Connected Components with MapReduce** Finding out the number of connected components in a graph is a key subroutine in many graph algorithms. Provide and prove the correctness of a MapReduce algorithm to count the number of connected components in a graph (represented as an edge list).

Algorithm 2: Reduce(ranking_pair)

```
queue ← PriorityQueue();
Function Reduce(pair):
    if $queue.size < K$ then
        queue.insert(pair)
    else
        min = queue.getMin();
        if $pair.ranking > min$ then
            queue.removeMin();
            queue.insert(pair);
        end
    end
return queue
```

Algorithm 3: MAP($userid - a, userid - b$)

```
if $userid - a < userid - b$ then
    string ← "$userid - a, userid - b$";
else
    string ← "$userid - b, userid - a$";
end
```

Algorithm 4: REDUCE($key, listofvalues$)

```
if $sum(values) = 2$ then
    return key
else
    return
end
```

Solution This MapReduce algorithm emulates a BFS algorithm, and is summarized in Algorithm 7. The `NeighborSearch(E; L)` function refers the the MapReduce steps summarized in Algorithms 5 and 6, with the edge set E given as input, and L (the list of found nodes) as a parameter. Without loss of generality, we assume that $V = [n]$, so the driver can easily select $u \in V$, and $u \in V$, $u \notin L$. However, straightforward modifications to the algorithm can account for the case where the vertex set is not known in advance.

5. **Sampling from multiple streams** Suppose we have numerous sub-streams of data (say $S_1, \ldots, S_n$), provide and prove the correctness of an algorithm to generate k random samples from the aggregate stream.

Algorithm 5: MAP: Neighbor Search

```
$N \leftarrow \emptyset$;
Function $MAP((u, v); L)$:
    if $u \in L$ then
        $N \leftarrow N \cup \{v\}$
    end
    if $v \in L$ then
        $N \leftarrow N \cup \{u\}$
    end
for $n \in N$ do
    emit(1, n)
end
```

Algorithm 6: Reduce: Neighbor Search

```
Function $REDUCE(key, [values])$:
    return unique(values)
```

Algorithm 7: Count Connected Components

```
$count \leftarrow 0$;
$L \leftarrow \{u\}$ random node in $G = (V, E)$;
while $L \neq V$ do
    if NeighborSearch(E; L) $\neq \emptyset$ then
        $L \leftarrow L \cup$ NeighborSearch(E; L)
    else
        count += 1;
        $L \leftarrow L \cup \{u\}$ for $u \in V$, $u \notin L$
    end
end
```

Solution We note that the solution here is sampling with replacement (as the length of the stream increases, the difference between with and without replacement becomes negligible). On each of the substreams, we run k independent copies of the reservoir sampling algorithm presented in class. We also maintain a counter for each of the substreams. Let $\{r_j^{(i)}\}_{j=1,\dots k}$ be the sample generated from $S^{(i)}$, then $\mathbb{P}[r_j^{(i)} = S_l^{(i)}] = \frac{1}{n_i}$ (this comes from directly applying the result from class). Since each of the $r_j^{(i)}$ are independent, this generates k elements that are uniformly sampled from the stream $S^{(i)}$.

Each of these samples $\{r_j^{(i)}\}_{j=1,\dots k}$ is sent to the driver, along with a count n_i of the number of elements $S^{(i)}$ has seen. The driver selects k elements, $t_1, \dots, t_k$ independently from these nk elements, where $\mathbb{P}[t_{(\cdot)} = r_j^{(i)}] = \frac{n_i}{k \sum n_i}$.

6. **Word Count Shuffle** Consider counting the number of occurrences of words in a collection of documents, where there are only k possible words. Write a MapReduce to achieve this, and analyze the shuffle size with and without combiners being used (assuming B mappers are used).

Solution The algorithm is summarized in Algorithms 8 and 9. The canonical word-count example will have a shuffle size equal to sum of the sizes of all documents if combiners are not used. i.e. if there are n documents of length L then shuffle size is nL, where as if we combine, each mapper outputs at most k counts, for a total of Bk

```
Algorithm 8: MAP(document)
1 for word in document do
2     return (word, 1)
3 end
```

```
Algorithm 9: REDUCE(key, listofvalues)
1 return sum(values)
```

7. **Prefix Sum** The *prefix-sum* operator takes an array $a_1, \dots, a_n$ and returns an array $s_1, \dots, s_n$ where $s_i = \sum_{j \le i} a_j$. For example, starting with an array `[17 0 5 32]` it returns `[17 17 22 54]`. Describe (in detail) how to implement *prefix-sum* in MapReduce, where the input is stored as $\langle i, a_i \rangle$. That is, the key is the position in the array, and the value is the value at that position. Analyze the shuffle size and the reduce-key space and time complexity.

Solution Intuition: If we compute the partial sums $sum(x[0...3])$ and $sum(x[4...7])$, then we can easily combine these to compute $sum(x[0...7])$. Using this intuition, we can split up the input into intervals that fit onto a single machine by emitting keys (Map step) that hash the input into their assigned interval. For example if we have R

reducers, input $<i, ai>$ is mapped to $(\lfloor iR/n \rfloor, (i, ai))$. Then the reducers compute the partial sum for each of their assigned intervals. The result will be R partial sums.

Since the number of reducers can't be too large, we can fit R numbers into memory. So we use a second Map to put the sum of each interval into the memory of all machines. Knowing the sum of all previous intervals, we can compute the partial sum from a1 for all intervals following the second Reduce.

At worst the shuffle size of the first MapReduce is n if the a_i with the same index are not stored on the same machine. If the array is already stored in intervals then the first MapReduce can have zero shuffle size. The shuffle size of the second MapReduce is R^2 as each machine sends the sum of its interval to every other machine.

The reduce-key space is the maximum amount of data assigned to a single key. In the first MapReduce the reduce-key space is n/R and in the second, the reduce-key space is R.

The time complexity of the first MapReduce is $O(n/Rlog(n/R))$ as each machine sorts its interval and then iterates over it once computing the (partial) prefix-sum. The complexity of the second interval is $O(n/R)$ as each machine computes the sum of all the intervals before it and carries out a single pass over its own interval.