CME 323: Distributed Algorithms and Optimization

Instructor: Reza Zadeh (rezab@stanford.edu)

HW#3 - Due at the beginning of class May 18th.

1. Download the following materials:

   - Slides: `http://stanford.edu/~rezab/dao/slides/itas_workshop.pdf`
   - Spark and Data: `http://training.databricks.com/workshop/usb.zip`

   Now, answer the following questions:

   (a) Checkpoint on slide 11

   (b) Checkpoint on slide 55

   (c) Checkpoint on slide 60

   Submit your code and answers. **Solution.**

   (a)
   ```
   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)
   ```
   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))
   ```
   So there are 18 instances of the word "Spark" in `README.md`.

   (c) The easiest method is to build off part (b):

   ```
   val f2 = sc.textFile("docs/contributing-to-spark.md")
   val wc2 = f2.flatMap(l => l.split(" ")).
               map(word => (word, 1)).
               reduceByKey(_ + _)
   wc.join(wc2).map(x => (x._1,x._2._1 + x._2._2)).
               filter(_._1 == "Spark").
               collect()

   res3: Array[(String, Int)] = Array((Spark,20))
   ```

   The slides specifically tell you to join the two RDDs, but in class Professor Zadeh mentioned that operations like `join` and `groupBy` should be avoided in favor of `union` and `reduceByKey` when possible. This is partially because the function passed to `reduceByKey` must be associative, so it can be applied to the list of

(K, V) pairs on each machine before the tuples are sent over the wire. In this way, each node will only communicate at most one tuple for each key, whereas before it would communicate one tuple for each (K, V) pair.

```
val wc1 = sc.textFile("README.md").
            flatMap(line => line.split(" ")).
            map(word => (word, 1))
val wc2 = sc.textFile("docs/contributing-to-spark.md").
            flatMap(line => line.split(" ")).
            map(word => (word, 1))
wc1.union(wc2).
    reduceByKey(_ + _).
    filter(_._1 == "Spark").
    collect()
```

This solution is more readable and uses our preferred operations, but it is not actually more efficient. Note that in the previous solution we apply a `reduceByKey` to each RDD before joining them — that was so that in the join each RDD would have at most one value for each key. This has the side effect of applying the associative reduce operation at each node before sending the wordcount tuples over the network.

We can, however, do better than these two solutions. Note that we are computing more information than we need — we are only interested in the number of times "Spark" appears in our corpus, but we instead compute all word counts. So, we can filter our RDDs before we begin counting by removing all words that are not "Spark", thus reducing the communication costs for each machine from $O(|D|)$ to $O(1)$ (D is our dictionary, the set of all unique words that appear in the corpus). The resulting code is both readable and efficient:

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

res2: Long = 20
```

2. Warmup question. 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 you do not have to worry about these. You are now 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. Pseudocode is OK.

Solution:

Function 1 MAP (*userid-a, userid-b*)

```
1: if userid-a < userid-b then
2:     string ← "userid-a, userid-b"
3: else
4:     string ← "userid-b, userid-a"
5: end if
6: return (string, 1)
```

Function 2 REDUCE (*key, list of values*)

```
1: if sum(values) = 2 then
2:     return key
3: else
4:     return
5: end if
```

3. Warmup question. 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:

   Function 1 MAP (*document*)

   ```
   1: for each word in document
   2:     return (word, 1)
   ```

   Function 2 REDUCE (*key, list of values*)

   ```
   1: return sum(values)
   ```

   The canonical wordcount 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.

4. The *prefix-sum* operator takes an array $a_1, \ldots, a_n$ and returns an array $s_1, \ldots, s_n$, where $s_i = \sum_{j \leq i} a_j$. For example starting with an array 17 0 5 32 it returns 17 17 22 54. Describe 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 $\text{sum}(x[0 \ldots 3])$ and $\text{sum}(x[4 \ldots 7])$, then we can easily combine these to compute $\text{sum}(x[0 \ldots 7])$.

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 $\langle i, a_i \rangle$ is mapped to $\langle i/R, (i, a_i) \rangle$. 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 a_1 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.

5. For a given undirected graph $G = (V, E)$ with n vertices and m edges ($m \geq n$), we say that G is shallow if for every pair of vertices $u, v \in V$, there is a path from u to v of length at most 2 (i.e., using at most two edges).

   (a) Give an algorithm that can decide whether G is shallow in $O(n^{2.376})$ time.

   (b) Given an $n \times r$ matrix A an $r \times n$ matrix B where $r \leq n$, show that we can multiply A and B in $O((n/r)^2 r^{2.376})$ time. (Use the fact that we can multiply two $r \times r$ matrices in $O(r^{2.376})$ time.)

   (c) Give an algorithm that can decide whether G is shallow in $O(m^{0.55} n^{1.45})$ time. [Hint: consider length-2 paths that go through low-degree vertices and length-2 paths that go through high-degree vertices separately. Use (b)]

Solution

(a) Consider the adjacency matrix A for G. A_{ij} contains the number of paths of length 1 from node i to j. similarly, A^2_{ij} contains the number of paths of length 2 from node i to j. Thus $(A^2 + A)_{ij}$ contains the number of paths of length at

most 2 from node i to j. Our algorithm will compute $A^2 + A$ and return true if and only if all non-diagonal entries of $A^2 + A$ are non-zero. A^2 can be computed in $O(n^{2.376})$ using Strassen's algorithm. A be be computed in $O(n^2)$ time, for a total running time of $O(n^{2.376} + n^2) = O(n^{2.376})$.

(b) We simply split up the $n \times r$ matrix into n/r $r \times r$ matrices, and use block matrix multiplication. In the case that r does not divide n exactly, we can simply add rows of zeros to the left-hand multiplicand matrix, and add columns of zeros to the right-hand multiplicand matrix and then remove extraneous rows and columns from the result.

We perform $\lceil n/r \rceil \times \lceil n/r \rceil$ block matrix multiplications, each taking $O(r^{2.376})$ time.

The runtime will be $O(\lceil n/r \rceil^2 r^{2.376}) = O((n/r + 1)^2 r^{2.376}) = O((n/r)^2 r^{2.376})$.

(c) We will maintain a boolean matrix M that will have $M_{ij} = 1$ if and only if there is a path of length at most 2 between node i and j. We initialize $M = A$, the adjacency matrix for G, leaving only paths of length 2 to be considered. At the end, we check each entry of M and claim the graph is shallow if and only if all non-diagonal entries of M are positive. Since M is initialized to A, it already contains paths of length 1. We will continuously update M to take into account paths of length 2. To do that, we look at all possible ordered triples (u, v, w). Each triple defines a path of length 2 going from u to w, through v.

We split the vertex set into two sets:

$$V_H = \{v \in V | deg(v) > d\}, \quad V_L = \{v \in V | deg(v) \leq d\}$$

Consider each ordered triple (u, v, w) defining a path from u to v to w. Either $v \in V_L$ or $v \in V_H$.

Case: $v \in V_L$, i.e. the middle vertex is low degree

```
1: for edge (v, w) ∈ E do
2:     if v is low-degree then
3:         for each neighbor u of v do
4:             M_uw = 1
5:             M_wu = 1
6:         end for
7:     end if
8:     if w is low-degree then
9:         for each neighbor u of w do
10:            M_uv = 1
11:            M_vu = 1
12:        end for
13:    end if
14: end for
```

This step takes at most $O(md)$ time since for each edge we check at most d neighbors.

Case: $v \in V_H$, i.e. the middle vertex is high-degree. We construct a matrix B with dimensions $n \times r$ where $r = |V_H|$. Each row corresponds to a node in V and each column corresponds to a node in V_H. $B_{ij} = 1$ if and only if there is an edge between arbitrary node i and V_H-member j. Thus BB^T gives us the number of paths of length 2 from arbitrary node i to arbitrary node j that go through some high-degree node as the middle node. We can do the BB^T computation in $O((n/r)^2 r^{2.376})$ time. We then update M to $M = M + BB^T$.

Since $2m =$ sum of all degrees $\geq |V_H|d = rd$. Thus $r \leq 2m/d$. So the computation takes $O((n/r)^2 r^{2.376}) = O(n^2 r^{0.376}) = O(n^2 (m/d)^{0.376})$.

So now we've covered all cases, M accounts for all possible paths of length 2 going through high-degree or low-degree vertices.

Finally we traverse M and claim the graph is shallow if and only if all non-diagonal entries of M are non-zero. This $O(n^2)$ will be dominated by $O(n^{1.45} m^{0.55})$, since $m \geq n$.

Thus total running time is $O(md + n^2 (m/d)^{0.376})$. We now minimize this bound with respect to d. Setting $md = n^2 (m/d)^{0.376}$ gives $d^* = n^{1.45} m^{-0.45}$. Subbing back in gives a bound of $O(md^* + n^2 (m/d^*)^{0.376}) = O(n^{1.45} m^{0.55})$.