Assignment 4 Solutions

Detian Deng

April 16, 2015

Analysis

1. Describe your Spark algorithm for solving the Three Musketeers problem

   The following is my very first algorithm, which implements the same idea from Hadoop Map/Reduce. An optimal algorithm is also implemented and described in question 2.

   Define a *FindTriad* procedure to get all triads from a single line of input.

```
FindTriad(line):
1. Parse line as an array of integers, A
2. Create an empty array of <Key, Value> pair, B
3. Set the first integer, A[1] as the key-person.
4. for i in 2 to A.length
5.   for j in (i+1) to A.length
6.     create an array of integers: B = [ key-person, A[i],A[j] ]
7.     key = Sort(B), value = 1
8.     B.append(<key, value>)
9. output B
```

   My Spark algorithm is:
   1) pass the FindTriad function to the flatMap transformation of the RDD
   2) pass the add function to the reduceByKey transformation of the resulted RDD
   3) apply the filter filter transformation to get the pairs whose value is greater than 1
   4) collect the results and write the keys to a file.

2. What is the Big-O complexity of the work your parallel implementation does (Instruction counting is ok)?

   Assume there are n people in the graph, then each individual has at most $n-1$ friends. In my implementation, there are n flatMap tasks, and at most $\binom{n}{3} = O(n^3)$ reduceByKey and filter tasks. In the worst case, each flatMap task takes $\binom{n-1}{2} = O(n^2)$, and each reduceByKey and filter task take $O(1)$. Since the reduceByKey transformation requires shuffling the keys, whose total work load can be assumed as $O(n^3 \log n^3) = O(n^3 \log n)$. Thus the overall complexity is $O(n^3 \log n)$.

   (a) What do you believe is the optimal complexity achievable within pure Spark? If your algorithm differs from the optimal, how could you improve your solution to get it to the optimal complexity solution?

   The optimal complexity achievable within Spark should be $O(n^3)$. Suppose this social network is an undirected graph G, $G.V$ is the set of vertices (people), and $G.E$ is the set of edges (friends pair), where $|V| = n, |E| = O(n^2)$, and the maximum degree of ant vertex of G is $Deg = O(n)$. The optimal algorithm should implement the following idea, whose complexity is $O(|E| \times Deg) = O(n^3)$:

```
ListTriangle(G):
1. for each v in G.V
2.   for each u in v.Adj
3.     if w in u.Adj intersect v.Adj
4.       output [v,u,w]
```

   In Pyspark, this algorithm can be implemented as below:

```
def makepair(text):
    words = text.split()
    k = words[0]
    v = set(words[1:])
    return (k,v)

def AintersectB(k1,k2,TheList):
    A = TheList.get(k1)
    B = TheList.get(k2)
    if A != None and B != None:
        return A.intersection(B)
    else:
        return {}

people = sc.textFile("friends1000")
AdjList = people.map(makepair)
DriverAdj = dict(AdjList.collect())
WorkerAdj = sc.broadcast(DriverAdj)
Edges = AdjList.flatMapValues(lambda x: x)
TriSet = Edges.map(lambda (k,v): ((k,v),
          AintersectB(k,v,WorkerAdj.value)))
Triangle = TriSet.flatMapValues(lambda x: x).map(lambda (k,v):
          tuple(sorted([int(v),int(k[0]),int(k[1])],reverse=True)))
print set(Triangle.collect())
```

3. Describe an optimization that can be achieved in a serial implementation that cannot be done within Spark for this problem. Why is this optimization not possible within Spark?

   Suppose this social network is an undirected graph G, $G.V$ is the set of vertices (people), and $G.E$ is the set of edges (friends pair), where $|V| = n, |E| = O(n^2)$. Based on Schank(2005), the following algorithm takes $O(|E|^{3/2}) = O(n^3)$ to list all triangles.

```
ListTriangle(G):
1. Sort G.V by its degree from high to low
2. Let A be a list of empty sets of length |V|
3. for s in 0 to |V|-1
4.   for t in G.V[s].Adj
5.     if s < t.index
6.       foreach v in A[s] intersect A[t.index]
7.         output [G.V[s],t,v]
8.       A[t.index] = A[t.index] union {G.V[s]}
```

   This optimization cannot be done within Spark because line 8 keeps updating A when $s < t.index$ and the iterations also depend on A. Since Spark parallelism is essentially done by data decomposition, this dependency makes Spark implementation not feasible.

4. Provide code to optimize the parallel performance of line 4 in the following program within Spark. Treat both xp and yp as though they were of equal length and each > 1 Million elements:

```
workerN = sc.broadcast(N)
result = xp.filter(lambda (k,v):k>workerN.value).join(yp.filter(lambda (k,v):
                    k>workerN.value)).collect()
```

5. Provide and briefly explain five reasons this code is suboptimal from the perspective of RDDs and the Spark runtime (Hint: Your coding solution should make answering this a bit easier).

   (a) Without filtering first, *xp.join(yp)* requires more data than needed (all the data vs. half of the data) shuffled over the network using more shuffle space.

   (b) Calling *.collect*() before filtering the arrays causes a large amount of data (more than needed) to be transferred from worker nodes to the driver node, which is slow and could lead to out-of-memory error.

   (c) The $filter()$ is a Python serial function that runs on the driver node instead of the worker nodes. There is no parallelism at this step. Spark $.filter()$ transformation should be used.

   (d) N calculated by *.count*() is returned to the driver node, so to avoid fetching N multiple times we should broadcast N to all workers.

   (e) The code does not make use of Spark lineage. Thus it does not have the advantage of fault tolerance.

6. We discussed k-means clustering in Spark in lecture. Here is Spark's implementation. Problem: You are given a machine with 2-cores (no hyper-threading) and 1GB of RAM. You are asked to perform k-means on a dense dataset with 1 Million records each with 100 elements of 16-byte floating-point precision. You are asked to compute k = 100,000 clusters.

   (a) Why can you not still use the same optimization(s) used to improve the performance of Spark over Hadoop! for this task? (Mathematically justify why this is true).

   The usual optimization used to improve the performance of Spark over Hadoop! is to persist the data in memory and do in-memory computation in each scheduled task. However, in this task, 1 Million records each with 100 elements of 16-byte will take about $1.6GB$ memory which is larger than the size of the RAM of this machine, and since by default Spark uses 60% of the configured executor memory (spark.executor.memory) to cache RDDs, a large fraction of the data cannot be cached in memory. And if $.cache()$ is used as in the given example code, the Storage Level is by default MEMORY_ONLY, therefore the part of data that cannot be cached will have to be recomputed on the fly each time they're needed, which is very inefficient. Therefore the same optimization does not work well.

   (b) How could you still achieve significant improvement over Hadoop! given the problem constraints?

      i. In terms of why Spark still has much better performance over Hadoop!
         - Spark has more efficient Shuffle operation.
         - Spark supports more than just Map and Reduce functions and flexible data structures (not just key-value pairs).
         - Lazy evaluation of big data queries which helps with the optimization of the overall data processing workflow.
         - Spark optimizes arbitrary operator graphs.

      ii. In terms of how we can optimize Spark when data cannot all fit in memory.
         - Use $.persist()$ with MEMORY_AND_DISK_SER as Storage Level using a fast serializer (e.g. Kryo library)to avoid recomputing too much data.
         - set the JVM flag -XX:+UseCompressedOops to make pointers be four bytes instead of eight.
         - Tune the Cache Size used for caching RDDs, and tune the JVM garbage collection.

7. Write a short program to optimally implement the same functionality as below within Spark. You may assume spark context already exists under the variable name sc (Hint: Doable in 2 lines):

```
a = sc.parallelize(xrange(1,N))
b = a.reduce(lambda x,y:x+y)
```

Reference

Schank, T., & Wagner, D. (2005). Finding, counting and listing all triangles in large graphs, an experimental study. In Experimental and Efficient Algorithms (pp. 606-609). Springer Berlin Heidelberg.

https://spark.apache.org/docs/latest/programming-guide.html