Distributed Graph Processing Lab

Semantic Data Management

Muhammad Zain Abbas
Kainaat Amjid

25th April, 2022

Exercise No. 01:

Solution for *exercise 1* was provided. So, from the learning point of view, we referred to the official documentation for each step to get a better understanding of the logic and code flow. So, initially we define our data (using *Lists.newArrayList* - collection method, *Tuple2* and *Edge*) and create vertices and edges' RDDs using *JavaSparkContext* object (via *parallelize* method).

```
List<Tuple2<Object,Integer>> vertices = Lists.newArrayList(
    new Tuple2<Object,Integer>(1l,9),
    new Tuple2<Object,Integer>(2l,1),
    new Tuple2<Object,Integer>(3l,6),
    new Tuple2<Object,Integer>(4l,8)
    );

List<Edge<Integer>> edges = Lists.newArrayList(
    new Edge<Integer>(1l,2l, 1),
    new Edge<Integer>(2l,3l, 1),
    new Edge<Integer>(2l,4l, 1),
    new Edge<Integer>(3l,4l, 1),
    new Edge<Integer>(3l,1l, 1)
    );

JavaRDD<Tuple2<Object,Integer>> verticesRDD = ctx.parallelize(vertices);
JavaRDD<Edge<Integer>> edgesRDD = ctx.parallelize(edges);
```

Once we have our RDDs, we construct a graph (with appropriate *storage levels* for vertices and edges); and create a GraphOps instance for calling the Pregel *API*, in which we specified our Apply (VProg), Scatter (sendMsg) and Gather (merge) methods.

```
Graph<Integer,Integer> G = Graph.apply(
  verticesRDD.rdd(), edgesRDD.rdd(), 1, StorageLevel.MEMORY_ONLY(), StorageLevel.MEMORY_ONLY(),
  scala.reflect.ClassTag$.MODULE$.apply(Integer.class),
  scala.reflect.ClassTag$.MODULE$.apply(Integer.class)
);
GraphOps ops = new GraphOps(G,
  scala.reflect.ClassTag$.MODULE$.apply(Integer.class),
  scala.reflect.ClassTag$.MODULE$.apply(Integer.class)
);
Graph<Integer, Integer> output_graph = ops.pregel(INITIAL_VALUE, Integer.MAX_VALUE,
EdgeDirection.Out(), new VProg(), new sendMsg(), new merge(),
scala.reflect.ClassTag$.MODULE$.apply(Integer.class));
```

After execution, pregel API will return a graph as an output. We extracted all the nodes (vertices) and picked the first node (you can pick any node since all of them will have the same value) to see the maximum value.

```
VertexRDD<Integer> output_vertices = output_graph.vertices();
JavaRDD<Tuple2<Object,Integer>> output_rdd = output_vertices.toJavaRDD();
Tuple2<Object,Integer> max_value = output_rdd.first();
Utils.print("Vertex '" + max_value._1 + "' has the maximum value in the graph '" + max_value._2 + "'");
```

Also, we specified our initial value first

```
    // Initial value for pregel execution
    static final Integer INITIAL_VALUE = Integer.MIN_VALUE;
```

VProg (Apply):

In the first superstep, just return the node's value. Otherwise, in each superstep, just return the maximum among the node current value and the message.

```
private static class VProg extends AbstractFunction3<Long,Integer,Integer,Integer> implements
Serializable {
  @Override
  public Integer apply(Long vertexID, Integer vertexValue, Integer message) {
    Utils.print("[ VProg.apply ] vertexID: '" +  vertexID +  "' vertexValue: '" +  vertexValue + "'
message: '" + message + "'" );
    if (message.equals(INITIAL_VALUE)) { // First superstep
      Utils.print("[ VProg.apply ] First superstep -> vertexID: '" +  vertexID +  "'");
      return vertexValue;
    } else { // Other supersteps
      Utils.print("[ VProg.apply ] vertexID: '" +  vertexID +  "' will send '" + Math.max(vertexValue,
message) + "' value");
      return Math.max(vertexValue, message);
    }
  }
}
```

sendMsg (Scatter):

From the triplets' view, we can "peek" at the destination value before sending the message. This enabled us to only send messages if the destination node has less value than the message we are about to send.

```
private static class sendMsg extends AbstractFunction1<EdgeTriplet<Integer,Integer>,
Iterator<Tuple2<Object,Integer>>> implements Serializable {
  @Override
  public Iterator<Tuple2<Object, Integer>> apply(EdgeTriplet<Integer, Integer> triplet) {
    Long srcId = triplet.srcId();
    Long dstId = triplet.dstId();
    Integer srcVertex = triplet.srcAttr();
    Integer descVertex = triplet.dstAttr();

    if ( srcVertex <= descVertex ) {
      Utils.print("[ sendMsg.apply ] srcId: '" +  srcId +  " [" + srcVertex + "]' will send nothing to dstId: '" +
dstId + " [" + descVertex + "]'");
      return JavaConverters.asScalaIteratorConverter(new ArrayList<Tuple2<Object,Integer>>().iterator()).asScala();
    } else {
      Utils.print("[ sendMsg.apply ] srcId: '" +  srcId +  " [" + srcVertex + "]' will send '" + srcVertex + "' to
dstId: '" + dstId + " [" + descVertex + "]'");
      return JavaConverters.asScalaIteratorConverter(Arrays.asList(new Tuple2<Object,Integer>(triplet.dstId(),
srcVertex)).iterator()).asScala();
      }
    }
  }
```

merge (Gather):

Since, we don't need to merge multiple messages before receiving so, basically we need not to implement this method.

```
private static class merge extends AbstractFunction2<Integer,Integer,Integer> implements Serializable {
  @Override
  public Integer apply(Integer msg1, Integer msg2) {
    Utils.print("[ merge.apply ] msg1: '" +  msg1 + "' msg2: '" + msg2 + "' -- do nothing");
    return null;
  }
}
```

Output:

```
After spark context
[ log ] Create vertices and edges
[ log ] Create RDD for vertices and edges
[ log ] Create Graph from vertices and edges
[ log ] Create graph operations' object
[ log ] Run pregel over our graph with apply, scatter and gather functions

---------------------------------

[Stage 0:>                                                          (0 + 0) / 8][ VProg.apply ] vertexID: '3' vertexValue: '6' message: '-2147483648'
[ VProg.apply ] vertexID: '1' vertexValue: '9' message: '-2147483648'
[ VProg.apply ] First superstep -> vertexID: '1'
[ VProg.apply ] vertexID: '4' vertexValue: '8' message: '-2147483648'
[ VProg.apply ] First superstep -> vertexID: '4'
[ VProg.apply ] vertexID: '2' vertexValue: '1' message: '-2147483648'
[ VProg.apply ] First superstep -> vertexID: '2'
[ VProg.apply ] First superstep -> vertexID: '3'
[ sendMsg.apply ] srcId: '1 [9]' will send '9' to dstId: '2 [1]'
[ sendMsg.apply ] srcId: '3 [6]' will send nothing to dstId: '1 [9]'
[ sendMsg.apply ] srcId: '2 [1]' will send nothing to dstId: '4 [8]'
[ sendMsg.apply ] srcId: '2 [1]' will send nothing to dstId: '3 [6]'
[ sendMsg.apply ] srcId: '3 [6]' will send nothing to dstId: '4 [8]'
[ VProg.apply ] vertexID: '2' vertexValue: '1' message: '9'
[ VProg.apply ] vertexID: '2' will send '9' value
[ sendMsg.apply ] srcId: '2 [9]' will send '9' to dstId: '4 [8]'
[ sendMsg.apply ] srcId: '2 [9]' will send '9' to dstId: '3 [6]'
[ VProg.apply ] vertexID: '3' vertexValue: '6' message: '9'
[ VProg.apply ] vertexID: '3' will send '9' value
[ VProg.apply ] vertexID: '4' vertexValue: '8' message: '9'
[ VProg.apply ] vertexID: '4' will send '9' value
[ sendMsg.apply ] srcId: '3 [9]' will send nothing to dstId: '1 [9]'
[ sendMsg.apply ] srcId: '3 [9]' will send nothing to dstId: '4 [9]'

---------------------------------

[ log ] Get output graphs' vertices
Output graph has '4' vertices
Vertex '1' has the maximum value in the graph '9'
```

Exercise No. 03:

We decided to change the type for *vertices* and *messages*, so we can store a *label* string and shortest *path* list. For this purpose, we modified the code for exercise 2 to replace all types and other parameters like: *initial values* etc, in order to use our *vertex data type*. A snippet of our data type is shown below:

```java
package exercise_3;

import java.util.List;
import java.util.ArrayList;
import scala.Tuple2;

public class Vertex extends Tuple2<Integer, List<String>> {

    public Vertex() { super(0, new ArrayList<String>()); }
    public Vertex(Integer num) { super(num, new ArrayList<String>()); }
    public Vertex(Integer num, List<String> str_list) { super(num, str_list); }

    boolean equals(Vertex vertex) { return this._1.equals(vertex._1) && this._2.equals(vertex._2); }
}
```

Exercise No. 04:

We were curious to see the changes in result if we change the damping factor and max iterations for the page rank algorithm. So, we decided to check *pagerank* and *time taken* to get the sense of how the parameters affect the pagerank algorithm. We started off by decreasing the damping factor (by increasing reset probability in Spark GraphFrame API) by

0.05 and for each damping factor, we ran the pagerank algorithm for max iterations of [5, 10, 15, 20].

Based on the output, we observed that by decreasing the damping factor (increasing reset probability), the output result converges (top 10 vertices) relatively quickly on less max iterations but the pagerank values themselves decrease. And generally, the higher the damping factor, the more time it took to run the pagerank algorithm. And furthermore, as we expected, the more the iterations, the more time it took to finish the pagerank algorithm.

Link:

To our project on **Github.**