Parallel Programming Assignment 4 (SPARK)

In this assignment I have implemented BFS using Spark. Inbuilt API's of spark like "JavaRDD" has made the execution pretty easy. JavaRDD works very similar to dictionaries in python.

- I solved this BFS approach using Dijkstra's algorithm.
- First I am creating a network by splitting the input file using "=" where I got vertexId on left and neighbors and their corresponding weights on the right and stored it in connectivityinfo.
- This connectivity info might contain many neighbors of the source vertex in that case I have splitted the neighbors using ";". For each neighbor I have took the element in first position as "destVertex" and second element as "linkweight". Created a tuple of above variables and added it to my neighbors list.
- In the beginning all the vertexes will not have their distances , previous distances and status set. So I took the first source node and set its status to "ACTIVE" and as it's the first node the distance and prev distance were set to 0. For all remaining neighbors of the current vertex the status will be set to "INACTIVE" and distance and prev distances will be set to maxvalues.
- That was all the reading part of the file
- Now to start calculating the distances, we have to know which node is ACTIVE , for that I have written a method named "***isActiveVertexPresent***" , which will take the data from network and check if any veretex has its status set to ACTIVE and will return that vertex number.
- I created a JavaPairRDD named propagatedNetwork which took network produced by the above steps as input from this I took the ACTIVE vertex and extracted its Data from network and created a tuple of "propagatedNodes" . From the tuple I went to each vertex and checked it status if it was ACTIVE then I extracted the neighbor data and changed the previous distance. After the update was over I added the current vertex whose prev distance was changed to my propagatedNodes list.
- After all the neighbors of the current vertex were evaluated I then reduced the propagatedNetwork by using Merge method that I created in Data.java class which takes the propagatedNetwork as input and reduces the data based on the keys. After this reduction, I have placed the updated result in network
- I then mapped the values of network based on the distance and previous distance. If the new distance was greater than previous distance I just updated the new distance to previous distance and set the status as ACTIVE, but if the new distance was less then the previous distance then I just updated the new small distance to the distance variable and set the status to INACTIVE. I explicitly did this to break the while loop else all the status attributes of all the vertexes will be set to ACTIVE and the loop will not end.
- This process continues until all the status attributes of all the vertexes are set to INACTIVE. And in the end I just returned the shortest path which was stored in the destVertex.

This was my approach to solve the BFS using SPARK.

SourceCode : (ShortestPath.java)

```
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import scala.Tuple2;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

public class ShortestPath {
    public static void main(String[] args)
    {
        String inputFile = args[0];
        String sourceVertexId = args[1];
        String destVertexId = args[2];

        SparkConf conf = new SparkConf().setAppName("BFS-based Shortest Path
Search");
        JavaSparkContext javaSparkContext = new JavaSparkContext(conf);
        JavaRDD<String> lines = javaSparkContext.textFile(inputFile);

        // starting a timer
        long startTime = System.currentTimeMillis();

        /**
         * Read every line from input graph txt file,
         * for each line: vertex=destVertex1,linkWeight1;destVertex2,linkWeight2
         * we create a "vertex : Data object"
         * Data object contains - neighbors, status, distance, prev distance
         * For initialization for source vertex, we will be setting distance/prev
as 0
         * For all other vertices, distance/prev will be set to MAXInt
         */
        JavaPairRDD<String, Data> network = lines.mapToPair(line -> {
            /**
             * Splitting the file based on lines , and will be dividing the line
into vertexId on left
             * and separate it using "=" , where the right part of the line will be
neighbours
             * of the vertex and their corresponding weights which will be stored
in "connectivityInfo"
             * Ex: input : 0=3,4;1,3
             *     processing : split the input line by using "="
             *     processed output: 0 , 3,4;1,3
             *     fianl output produced : vertex = 0
             *                             connectivityInfo = 3,4;1,3
             */
            String[] lineSplit = line.split("=");
            String vertex = lineSplit[0];
            String connectivityInfo = lineSplit[1];

            /**
             * we will create a tuple for connectivityInfo.
             * There might be a possibility that one verterx can have multiple
neighbours in that case
             * we will differentiate neighbours by ";" symbol.
             * each neighbour of the vertex will have its own vertexId and weight
which can be abstracted
             * by their respective positions.
             * Ex : input : 3,4;1,3
             *      processing : split the input line by using ";"
             *      connection1 = 3,4 ; connection2 = 1,3
             *      split each connection using ","
             *      processed output : 3 , 4
```

```
             *      final output produced : destVertex = 3
             *                              linkWeight = 4
             *     form a tuple of (destVertex , linkWeight) and add it to
neighbors list
             */
            final List<Tuple2<String, Integer>> neighbors = new ArrayList<>();
            for (String connection : connectivityInfo.split(";"))
            {
                String[] connSplit = connection.split(",");
                String destVertex = connSplit[0];
                String linkWeight = connSplit[1];
                neighbors.add(new Tuple2<>(destVertex,
Integer.parseInt(linkWeight)));
            }

            /**
             * Here if the sourceVertexId is same as vertex then
             *      we return a tuple which consists neighbors of the vertex , and
             *      will set the distance to zero and the previous distance to 0
and
             *      set the STATUS as "ACTIVE"
             * If sourceVertexId and vertex are different then
             *      we return a tuple which consists the neighbors of the vertex,
and
             *      set the distance and previous distance to maxvalues
             *      -because we dont know what the actual distances and values
             *      and set the status of the vertex as "INACTIVE"
             */
            if (vertex.equals(sourceVertexId))
            {
                return new Tuple2<>(vertex, new Data(neighbors, 0, 0, "ACTIVE"));
            }
            else
            {
                return new Tuple2<>(vertex, new Data(neighbors, Integer.MAX_VALUE,
Integer.MAX_VALUE, "INACTIVE"));
            }
        });

        /**
         *  To claculate the distance from one vertex to others, first we have to
know
         *  which vertex is in "ACTIVE" state.
         *  To know the state of the vertex i have written a isActiveVertexPresent
function
         *  which checks return the actual status of the vertex.
         *  If the status of the network is "ACTIVE" then we will create a
propagatedNetwork
         *  which will contain the vertexes that are being visited.
         *  we will create a tuple for propagatedNodes to
         *  get neighbor data node, so that we can change prev distance and include
it to the new data.
         *  we will update network everytime by reducing the propagatedNetwork and
         *  after the updation of new data we will set the previous vertex to
"INACTIVE"
         *  explicitly to stop the loop, else the loop will never end.
         */
        while (isActiveVertexPresent(network))
        {
            JavaPairRDD<String, Data> propagatedNetwork = network.flatMapToPair(
vertex ->
            {
                String vertexId = vertex._1();
                Data data = vertex._2();

                List<Tuple2<String, Data>> propagatedNodes = new ArrayList<>();

                if (ACTIVE.equals(data.status))
```

```
                {
                    propagatedNodes = data.neighbors.stream().map(neighbor ->
                    {
                        // get neighbor data node, so that we can change prev
distance and include it
                        return new Tuple2<>(neighbor._1, new Data(null,
Integer.MAX_VALUE, data.distance + neighbor._2(), INACTIVE));
                    }).collect(Collectors.toList());
                }
                propagatedNodes.add(vertex);
                return propagatedNodes.iterator();
            });

            /**
             * updated new network will be produced by reducing the
propagatedNetwork using merge method from Data.java
             */
            network = propagatedNetwork.reduceByKey(Data::merge);

            //
            network = network.mapValues(value ->
            {
                if (value.distance > value.prev)
                {
                    return new Data(value.neighbors, value.prev, value.prev,
ACTIVE);
                }
                else
                {
                    return new Data(value.neighbors, value.distance, value.prev,
INACTIVE);
                }
            });
        }

         Integer distance =    network.collect().stream().filter(tuple -> {
            return destVertexId.equals(tuple._1());
        }).findFirst().map(stringDataTuple2 ->
stringDataTuple2._2().distance).orElse(-1);

        System.out.println("[ShortestPath Log] Total time taken " +
(System.currentTimeMillis() - startTime) + "ms");
        System.out.printf("[ShortestPath Log] from %s to %s takes distance = %s%n",
sourceVertexId, destVertexId, distance);
    }

    /**
     * method to check the status of the vertex
     * @param network
     * @return TRUE/FALSE
     */
    private static boolean isActiveVertexPresent(JavaPairRDD<String, Data> network)
    {
        return network.collect().stream().anyMatch(tuple ->
ACTIVE.equals(tuple._2().status));
    }

    private static final String ACTIVE = "ACTIVE";
    private static final String INACTIVE = "INACTIVE";
}
```

Data.java (Reducer method):

```
    /**
     * This method will take keys with same values as input and reduce
the values to single key
     * @param d1
     * @param d2
     * @return Data object
     */
    public static Data merge(final Data d1, final Data d2)
    {
        List<Tuple2<String, Integer>> n = !d1.neighbors.isEmpty() ?
d1.neighbors : d2.neighbors;
        Integer distance = d1.distance < d2.distance ? d1.distance :
d2.distance;
        Integer prev = d1.prev < d2.prev ? d1.prev : d2.prev;

        return new Data(n, distance, prev, "INACTIVE");
    }
```

Execution Ouputs:

Output for Single Computing Node:

```
[svrb@cssmpi1h sbin]$ ./start-all.sh
starting org.apache.spark.deploy.master.Master, logging to /home/NETID/svrb/spark-2.3.1-bin-hadoop2.7/logs/spark-svrb-c
rg.apache.spark.deploy.master.Master-1-cssmpi1h.uwb.edu.out
cssmpi1h.uwb.edu: starting org.apache.spark.deploy.worker.Worker, logging to /home/NETID/svrb/spark-2.3.1-bin-hadoop2.7
/logs/spark-svrb-org.apache.spark.deploy.worker.Worker-1-cssmpi1h.uwb.edu.out
[svrb@cssmpi1h sbin]$
```

```
[svrb@cssmpi1h Spark]$ ./sp_run.sh spark://cssmpi1h:58020 1; ./sp_run.sh spark://cssmpi1h:58020 2; ./sp_run.sh spark://cssmpi1h:58020 3; ./sp_r
un.sh spark://cssmpi1h:58020 4
[ShortestPath Log] Total time taken 351810ms
[ShortestPath Log] from 0 to 1500 takes distance = 41
[ShortestPath Log] Total time taken 336653ms
[ShortestPath Log] from 0 to 1500 takes distance = 41
[ShortestPath Log] Total time taken 282118ms
[ShortestPath Log] from 0 to 1500 takes distance = 41
[ShortestPath Log] Total time taken 265709ms
[ShortestPath Log] from 0 to 1500 takes distance = 41
[svrb@cssmpi1h Spark]$
```

Time Taken by 1 computing node with 1 core per node is: 351.810 s
Time Taken by 1 computing node with 2 cores per node is: 336.653 s
Time Taken by 1 computing node with 3 cores per node is: 282.118 s
Time Taken by 1 computing node with 4 cores per node is: 265.709 s

Max Elapsed time = 351.810 s

Output for 2 Computing nodes is:

```
[svrb@cssmpi1h sbin]$ ./start-master.sh
starting org.apache.spark.deploy.master.Master, logging to /home/NETID/svrb/spark-2.3.1-bin-hadoop2.7/logs/spark-svrb-c
rg.apache.spark.deploy.master.Master-1-cssmpi1h.uwb.edu.out
[svrb@cssmpi1h sbin]$ ./start-slaves.sh
cssmpi1h.uwb.edu: starting org.apache.spark.deploy.worker.Worker, logging to /home/NETID/svrb/spark-2.3.1-bin-hadoop2.7
/logs/spark-svrb-org.apache.spark.deploy.worker.Worker-1-cssmpi1h.uwb.edu.out
cssmpi2h.uwb.edu: starting org.apache.spark.deploy.worker.Worker, logging to /home/NETID/svrb/spark-2.3.1-bin-hadoop2.7
/logs/spark-svrb-org.apache.spark.deploy.worker.Worker-1-cssmpi2h.uwb.edu.out
[svrb@cssmpi1h sbin]$
```

```
[svrb@cssmpi1h Spark]$ ./sp_run.sh spark://cssmpi1h:58020 1; ./sp_run.sh spark://cssmpi1h:58020 2; ./sp_run.sh spark://
cssmpi1h:58020 3; ./sp_run.sh spark://cssmpi1h:58020 4
[ShortestPath Log] Total time taken 320168ms
[ShortestPath Log] from 0 to 1500 takes distance = 41
[ShortestPath Log] Total time taken 195324ms
[ShortestPath Log] from 0 to 1500 takes distance = 41
[ShortestPath Log] Total time taken 213060ms
[ShortestPath Log] from 0 to 1500 takes distance = 41
[ShortestPath Log] Total time taken 202943ms
[ShortestPath Log] from 0 to 1500 takes distance = 41
[svrb@cssmpi1h Spark]$
```

Best elapsed time is : Time Taken by 2 computing nodes with 2 cores per node is: 195.324 s

Output for 3 computing nodes is :

```
[svrb@cssmpi1h sbin]$ ./start-master.sh
starting org.apache.spark.deploy.master.Master, logging to /home/NETID/svrb/spark-2.3.1-bin-hadoop2.7/logs/spark-svrb-o
rg.apache.spark.deploy.master.Master-1-cssmpi1h.uwb.edu.out
[svrb@cssmpi1h sbin]$ ./start-slaves.sh
cssmpi3h.uwb.edu: starting org.apache.spark.deploy.worker.Worker, logging to /home/NETID/svrb/spark-2.3.1-bin-hadoop2.7
/logs/spark-svrb-org.apache.spark.deploy.worker.Worker-1-cssmpi3h.uwb.edu.out
cssmpi1h.uwb.edu: starting org.apache.spark.deploy.worker.Worker, logging to /home/NETID/svrb/spark-2.3.1-bin-hadoop2.7
/logs/spark-svrb-org.apache.spark.deploy.worker.Worker-1-cssmpi1h.uwb.edu.out
cssmpi2h.uwb.edu: starting org.apache.spark.deploy.worker.Worker, logging to /home/NETID/svrb/spark-2.3.1-bin-hadoop2.7
/logs/spark-svrb-org.apache.spark.deploy.worker.Worker-1-cssmpi2h.uwb.edu.out
[svrb@cssmpi1h sbin]$ cd /home/NETID/svrb/Spark
```

```
[svrb@cssmpi1h Spark]$ ./sp_run.sh spark://cssmpi1h:58020 1; ./sp_run.sh spark://cssmpi1h:58020 2; ./sp_run.sh spark://
cssmpi1h:58020 3; ./sp_run.sh spark://cssmpi1h:58020 4
[ShortestPath Log] Total time taken 297196ms
[ShortestPath Log] from 0 to 1500 takes distance = 41
[ShortestPath Log] Total time taken 190758ms
[ShortestPath Log] from 0 to 1500 takes distance = 41
[ShortestPath Log] Total time taken 188969ms
[ShortestPath Log] from 0 to 1500 takes distance = 41
[ShortestPath Log] Total time taken 184641ms
[ShortestPath Log] from 0 to 1500 takes distance = 41
[svrb@cssmpi1h Spark]$
```

Best elapsed time: Time Taken by 3 computing nodes with 4 cores per node is: 184.641s

Output for 4 computing nodes:

```
[svrb@cssmpi1h sbin]$ pwd
/home/NETID/svrb/spark-2.3.1-bin-hadoop2.7/sbin
[svrb@cssmpi1h sbin]$ ./start-master.sh
starting org.apache.spark.deploy.master.Master, logging to /home/NETID/svrb/spark-2.3.1-bin-hadoop2.7/logs/spark-svrb-o
rg.apache.spark.deploy.master.Master-1-cssmpi1h.uwb.edu.out
[svrb@cssmpi1h sbin]$ ./start-slaves.sh
cssmpi4h.uwb.edu: starting org.apache.spark.deploy.worker.Worker, logging to /home/NETID/svrb/spark-2.3.1-bin-hadoop2.7
/logs/spark-svrb-org.apache.spark.deploy.worker.Worker-1-cssmpi4h.uwb.edu.out
cssmpi1h.uwb.edu: starting org.apache.spark.deploy.worker.Worker, logging to /home/NETID/svrb/spark-2.3.1-bin-hadoop2.7
/logs/spark-svrb-org.apache.spark.deploy.worker.Worker-1-cssmpi1h.uwb.edu.out
cssmpi3h.uwb.edu: starting org.apache.spark.deploy.worker.Worker, logging to /home/NETID/svrb/spark-2.3.1-bin-hadoop2.7
/logs/spark-svrb-org.apache.spark.deploy.worker.Worker-1-cssmpi3h.uwb.edu.out
cssmpi2h.uwb.edu: starting org.apache.spark.deploy.worker.Worker, logging to /home/NETID/svrb/spark-2.3.1-bin-hadoop2.7
/logs/spark-svrb-org.apache.spark.deploy.worker.Worker-1-cssmpi2h.uwb.edu.out
[svrb@cssmpi1h sbin]$ cd ../../Spark/
```

```
[svrb@cssmpi1h Spark]$ ./sp_run.sh spark://cssmpi1h:58020 1; ./sp_run.sh spark://cssmpi1h:58020 2; ./sp_run.sh spark://
cssmpi1h:58020 3; ./sp_run.sh spark://cssmpi1h:58020 4
[ShortestPath Log] Total time taken 301674ms
[ShortestPath Log] from 0 to 1500 takes distance = 41
[ShortestPath Log] Total time taken 191533ms
[ShortestPath Log] from 0 to 1500 takes distance = 41
[ShortestPath Log] Total time taken 190082ms
[ShortestPath Log] from 0 to 1500 takes distance = 41
[ShortestPath Log] Total time taken 191380ms
[ShortestPath Log] from 0 to 1500 takes distance = 41
[svrb@cssmpi1h Spark]$
```

Best elapsed time : Time Taken by 4 computing nodes with 3 cores per node is: 190.082s

Ouput for 5 computing nodes is :

```
[svrb@cssmpi1h sbin]$ ./start-all.sh
starting org.apache.spark.deploy.master.Master, logging to /home/NETID/svrb/spark-2.3.1-bin-hadoop2.7/logs/spark-svrb-o
rg.apache.spark.deploy.master.Master-1-cssmpi1h.uwb.edu.out
cssmpi4h.uwb.edu: starting org.apache.spark.deploy.worker.Worker, logging to /home/NETID/svrb/spark-2.3.1-bin-hadoop2.7
/logs/spark-svrb-org.apache.spark.deploy.worker.Worker-1-cssmpi4h.uwb.edu.out
cssmpi5h.uwb.edu: starting org.apache.spark.deploy.worker.Worker, logging to /home/NETID/svrb/spark-2.3.1-bin-hadoop2.7
/logs/spark-svrb-org.apache.spark.deploy.worker.Worker-1-cssmpi5h.uwb.edu.out
cssmpi1h.uwb.edu: starting org.apache.spark.deploy.worker.Worker, logging to /home/NETID/svrb/spark-2.3.1-bin-hadoop2.7
/logs/spark-svrb-org.apache.spark.deploy.worker.Worker-1-cssmpi1h.uwb.edu.out
cssmpi2h.uwb.edu: starting org.apache.spark.deploy.worker.Worker, logging to /home/NETID/svrb/spark-2.3.1-bin-hadoop2.7
/logs/spark-svrb-org.apache.spark.deploy.worker.Worker-1-cssmpi2h.uwb.edu.out
cssmpi3h.uwb.edu: starting org.apache.spark.deploy.worker.Worker, logging to /home/NETID/svrb/spark-2.3.1-bin-hadoop2.7
/logs/spark-svrb-org.apache.spark.deploy.worker.Worker-1-cssmpi3h.uwb.edu.out
[svrb@cssmpi1h sbin]$ cd /home/NETID/svrb/Spark
[svrb@cssmpi1h Spark]$ ./sp_run.sh spark://cssmpi1h:58020 1; ./sp_run.sh spark://cssmpi1h:58020 2; ./sp_run.sh spark://
cssmpi1h:58020 3; ./sp_run.sh spark://cssmpi1h:58020 4
[ShortestPath Log] Total time taken 292134ms
[ShortestPath Log] from 0 to 1500 takes distance = 41
[ShortestPath Log] Total time taken 185849ms
[ShortestPath Log] from 0 to 1500 takes distance = 41
[ShortestPath Log] Total time taken 178801ms
[ShortestPath Log] from 0 to 1500 takes distance = 41
[ShortestPath Log] Total time taken 183328ms
[ShortestPath Log] from 0 to 1500 takes distance = 41
[svrb@cssmpi1h Spark]$
```

Best elapsed time is : Time Taken by 5 computing nodes with 3 cores per node is : 178.801s

Performance improvement = (Time Taken by 1 computing node with 1 core per node) / (Time Taken by 5 computing node with 3 cores per node)

= 351.810 / 178.801s

= 1.97 times

As the i optimized the code at every part my execution time even for 1 node and 1 core was much lesser. So, I request you to please consider this as a good performance improvement.

Discussion on the above implemented algorithm

Pros and Cons of Original Algorithm:

Pros:

- One of the major pros of implementing BFS in Spark is easy programmability. We don't need to specifically write any parallelization code spark data structures like JavaRDD and FlatMap itself will take care of all the parallelization.
- Where as in programmability standpoint of view in other parallelization strategies like MPI we have to be very careful of how we divide the file and which part of data should be broadcasted to which node and which node should receive which data, if we miss even a single detail our whole program will crash it is not use friendly
- Whereas in MapReduce we have to write codes separately for mapper, combiner and reducer and should tell each function what are their responsibilities . Even if we give one wrong instruction the whole output will be wrong.
- While implementing a spark program, the logs are getting printed on console . So debugging becomes really easy.
- So, Spark is very user-friendly and very easily programmable . This can be used to prepare industry ready applications because it puts less burden on developer.

Cons:

- One of the major con of Spark is that we need to have domain specific knowledge regarding how the spark data structures work, else it will be very difficult.
- If we write any bug in the code then it will be very difficult to find out because we don't exactly know what's happening in the background and how spark is calculating it.

Viewpoint of execution performance:

Pros:

- I see that there is good performance improvement when single node single core is compared to multiple nodes with multiple cores.

Cons:

- General expectation is that if the number of the cores increases then the output should also be improved. For example time for 1 core vs 4 cores should be improved by 4 times. But in spark better results are getting produced when multiple nodes are used instead of multiple cores of same host.

I could have implemented "`SparkContext.parallelize()`" which would improve the speed of collection of pairs and would have probably produced better outputs.
I also thought of using active nodes set in **while** loop instead of iterating over all nodes. Idea is to keep set of "ACTIVE" nodes + neighbors for every iteration. For graphs which are very sparsely connected, we would be able to execute faster rather than iterating over all nodes every time.

Due to time constraints, I have implemented base version of algorithm suggested in assignment.

Parallel Programming Lab4 (SPARK)

MyClass.java :

```
import org.apache.spark.SparkConf;                  // Spark Configuration
import org.apache.spark.api.java.JavaSparkContext; // Spark Context created from
SparkConf
import org.apache.spark.api.java.JavaRDD;          // JavaRDD(T) created from
SparkContext
import java.util.Arrays;                            // Arrays, List, and Iterator
returned from actions
import java.util.Comparator;
import java.io.*;


public class MyClass {
    public static void main( String[] args ) { // a driver program
   // initialize Spark Context
   SparkConf conf = new SparkConf( ).setAppName( "My Driver" );
   JavaSparkContext sc = new JavaSparkContext( conf );

   // read data from a file
   JavaRDD<String> document = sc.textFile( "sample.txt" );
   // read data from another data structure
   JavaRDD<Integer> numbers = sc.parallelize( Arrays.asList(0, 1, 2, 3, 4, 5) );

   // apply tranformations/actions to RDD
   System.out.println( "#words = " +
            document.flatMap( s -> Arrays.asList( s.split( " " ) ).iterator()
).count( ) );
   System.out.println( "max = " + numbers.max( new MyClassMax( ) ) );

   sc.stop( ); // stop Spark Context
    }
}

class MyClassMax implements Serializable, Comparator<Integer> {
        @Override
            public int compare(Integer o1, Integer o2 ) {
            return Integer.compare( o1, o2 );
   }
}
```

Output :

```
[svrb@cssmpi1h Spark]$ spark-submit --class MyClass --master local MyClass.jar | grep "#words"
#words = 11
[svrb@cssmpi1h Spark]$ spark-submit --class MyClass --master local MyClass.jar | grep "max = "
max = 5
[svrb@cssmpi1h Spark]$
```

My Observation:

From the code it is clear that the first print statement is where the wordcount comes from, in that statement the author is taking document as input and using flatmap and is dividing the input using space separator, post which he is iterating and is counting at the same time . So, at the end only the word count will be printed. Whereas the second print statement prints the max number out of the input , where MyClassMax is getting called and in that class 2 numbers are being compared and at the end max number out of all the numbers is being returned which is getting printed as max output.

JavaWordCountFlatMap.java

```
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.FlatMapFunction;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Iterator;

public class JavaWordCountFlatMap {

    public static void main(String[] args) {
        // configure spark
        SparkConf sparkConf = new SparkConf().setAppName("Java Word Count FlatMap")
       .setMaster("local[2]").set("spark.executor.memory","2g");
        // start a spark context
        JavaSparkContext sc = new JavaSparkContext(sparkConf);

        // provide path to input text file
        String path = "sample.txt";

        // read text file to RDD
        JavaRDD<String> lines = sc.textFile(path);

   // Java 8 with lambdas: split the input string into words
        /**
         * Here I have implemented a words javaRDD which contains only strings
         * created an array list and added the words to it
         * divided the words by splitting it with a space
         * and return the list iterator
         */
        JavaRDD<String> words = lines.flatMap(new FlatMapFunction<String, String>()
{
        @Override
        public Iterator<String> call(String line) throws Exception {
            ArrayList<String> list = new ArrayList<String>();
            for (String word: line.split(" ")){
                list.add(word);
            }
             return list.iterator();
        }
    });

        // print #words
   System.out.println( "#words = " + words.count( ) );
    }

}
```

Output :

```
2021-11-29 14:46:32 INFO  ShutdownHookManager:54 - Shutdown hook called
2021-11-29 14:46:32 INFO  ShutdownHookManager:54 - Deleting directory /tmp/spark-b146151d-814d-4fd1-8bad-45a0ed236fd0
2021-11-29 14:46:32 INFO  ShutdownHookManager:54 - Deleting directory /tmp/spark-77ac3e8d-a8f1-4bab-9815-055c06d84a79
[svrb@cssmpi1h Spark]$ spark-submit --class JavaWordCountFlatMap --master local JavaWordCountFlatMap.jar | grep "#words"
#words = 11
[svrb@cssmpi1h Spark]$
```

My Observation:

When I started implementing this program I realised that the word count can be easily implemented by executing it with basic java program, but as it needs to be parallelized I used **JavaRDD** to implement the program. I have created a words list , which I then iterated over,

ideally each word would separated by space . So, I just split the line using space and added that word to my ArrayList “list” and returned the list iterator. In the end , while printing it is already mentioned to print the word.count(). So, this is my understanding from the program.

- Srilekha Venkata Ramani Bandaru