Large Scale Parallel Data Processing – Spring 2019
Name :Vaibhav Shekhar Dave
Email : dave.v@husky.neu.edu
GitHub repo: https://github.ccs.neu.edu/vaibhavdave5/parallelDataProcessing
Submission for : HW1

Hadoop Map-Reduce Implementation
Source code : https://github.ccs.neu.edu/vaibhavdave5/parallelDataProcessing/tree/master/MR-Demo

Psuedo Code:
Step 1:
Take input file and output file dir as input from the user:

```
FileInputFormat.addInputPath(job, new Path(args[0]));
```

Step 2: Mapper phase

The mapper implementation via map method, processes one line at a time, as provide in the file It then splits the line into two tokens separated by "," and selects the first word(User ID) and emits a key-value pair as <User-ID,1>

```
public void map(final Object key, final Text value, final Context context)     throws IOException,
InterruptedException {
            final StringTokenizer itr = new StringTokenizer(value.toString());
            while (itr.hasMoreTokens()) {
                // Split each line of the csv by ","  and take the first word as the input
                word.set(itr.nextToken().split(",")[0]);
                context.write(word, one);
            }
        }
```

Step 3: Combiner phase:
A combiner is also specified. Hence, the output of each map is passed through the local combiner for local aggregation, after being sorted on the keys.

```
job.setCombinerClass(IntSumReducer.class);
```

Step 4: Reducer phase:
The reducer implementation, via the reduce method just sums up the values, which are the occurrence counts for each key.

```
@Override
        public void reduce(final Text key, final Iterable<IntWritable> values, final Context context) throws
IOException, InterruptedException {
            int sum = 0;
            for (final IntWritable val : values) {
                sum += val.get();
            }
            result.set(sum);
            context.write(key, result);}
```

Step 5: Output:

Once jobs are completed the result is saved as output in the directory specified by the user.

```
FileOutputFormat.setOutputPath(job, new Path(args[1]));
```

General Idea:

- My program assumes the its output is a csv file in the format of userid, follow's userid.
- It takes input as a csv file.
- The TokenizerMapper helps to read the input line by line.
- Each line is then split by "," using the set function and the first word is selected to be the key
- the map function spits out a <userid, id> after reading each line.
- The output of the local mapper then goes to the local combiner for sorting.
- The local combiner sorts the value of the map based on the key.
- The reduce method is use to sum up the occurrence of each key emitted by all the map.
- This way we can find all the occurences of the userids in edges.csv and we can find how many followers each user has.

Spark Scala Implementation

Source: https://github.ccs.neu.edu/vaibhavdave5/parallelDataProcessing/tree/master/Spark-Demo

Note: I have used RDD APIs for the implementation.

Psuedo Code:

1) Reads the input from the text file and tokenises

```
val textFile = sc.textFile(args(0))
```

2) Each token(Each line) is split by "," and we select the first word(userID).

```
val counts = textFile.map(line => line.split(",")(0))
```

3) The mapping phase just takes the userID as input and spits out <userID, 1> as a key value pair

```
.map(word => (word, 1))
```

4) In the reduce phase the number of occurrences of userid key is stored as <userid, number of occurences>

```
        .reduceByKey(_ + _)
```

5) We save the sparks RDD's logical exectution plan by the following snippet of code

```
val file = new File("log.txt")
val bw = new BufferedWriter(new FileWriter(file))
bw.write(counts.toDebugString)
bw.close()
```

The output is as follows:

```
(40) ShuffledRDD[4] at reduceByKey at WordCount.scala:30 []
+-(40) MapPartitionsRDD[3] at map at WordCount.scala:29 []
   |  MapPartitionsRDD[2] at map at WordCount.scala:28 []
   |  input/edges.csv MapPartitionsRDD[1] at textFile at WordCount.scala:27 []
   |  input/edges.csv HadoopRDD[0] at textFile at WordCount.scala:27 []
```

6 counts.saveAsTextFile(args(1)) // Action save the output

Running Time Measurements

Question

Copy to your Github the syslog (MapReduce) and stderr (Spark) log files of the runs you are reporting the measurements for. Check that the log is not truncated—there might be multiple pieces for large log files! Include a link to each log file/directory (4 links total) in the report. Similarly, copy the output produced (all parts of it) to your Github and include the links to the output directories (4 links total) in the report. (2 points)

Logs for map reduce run 1

https://github.ccs.neu.edu/vaibhavdave5/parallelDataProcessing/blob/master/Important%20logs/hadoop%201/syslog

Logs for map reduce run 2

https://github.ccs.neu.edu/vaibhavdave5/parallelDataProcessing/blob/master/Important%20logs/hadoop2/syslog

Output :
https://github.ccs.neu.edu/vaibhavdave5/parallelDataProcessing/tree/master/MR-Demo/output

---

Logs for spark scala run 1

https://github.ccs.neu.edu/vaibhavdave5/parallelDataProcessing/tree/master/Important%20logs/spark%201

Logs for spark scala run 2

https://github.ccs.neu.edu/vaibhavdave5/parallelDataProcessing/tree/master/Important%20logs/spark%202

Output :
https://github.ccs.neu.edu/vaibhavdave5/parallelDataProcessing/tree/master/Spark-Demo/output

Question

Measure the running time of each program. Repeat the time measurements one more time, starting each program from scratch as a new job. Report all 2 programs * 2 independent runs = 4 running times you measured

1) Runtime for run 1 – Map-Reduce - 1.6 min

```
Total time spent by all maps in occupied slots (ms)=55838448
Total time spent by all reduces in occupied slots (ms)=13295904
Total time spent by all map tasks (ms)=1163301
Total time spent by all reduce tasks (ms)=138499
Total vcore-milliseconds taken by all map tasks=1163301
Total vcore-milliseconds taken by all reduce tasks=138499
Total megabyte-milliseconds taken by all map tasks=1786830336
Total megabyte-milliseconds taken by all reduce tasks=425468928
```

2) Runtime for run 2 – Map-Reduce – 1.7 min

```
Total time spent by all maps in occupied slots (ms)=54659760
Total time spent by all reduces in occupied slots (ms)=12879744
Total time spent by all map tasks (ms)=1138745
Total time spent by all reduce tasks (ms)=134164
Total vcore-milliseconds taken by all map tasks=1138745
Total vcore-milliseconds taken by all reduce tasks=134164
Total megabyte-milliseconds taken by all map tasks=1749112320
Total megabyte-milliseconds taken by all reduce tasks=412151808
```

1) Runtime for run 1 – Spark-scala – 1.1 min

2) Runtime for run 2 – Spark-scala – 1.1 min

Question:

Report the amount of data transferred to the Mappers, from Mappers to Reducers, and from Reducers to output. There should be 3 numbers. (3 points)

For map reduce run -1
Data transferred to mappers -1319507620 bytes

Mappers to reducers – 955121298 bytes

Reducers to Output – 87376129 bytes

For map reduce run - 2
Data transferred to mappers -1319487591 bytes

Mappers to reducers – 955121298 bytes

Reducers to Output – 87376129 bytes

Question
Argue briefly, why or why not your MapReduce program is expected to have good speedup. Make sure you discuss (i) how many tasks were executed in each stage and (ii) if there is a part of your program that is inherently sequential (see discussion of Amdahl's Law in the module.)

Conceptually:
Speedup = sequentialTime / parallelTime

1) Number of Map task - 20
2) Number of reduce task - 9

No there is no part in the program that is inherently sequential.

In a sequential way we would have read the whole file edges.csv and we would have had to add each token to a hashmap to maintain the sum of each user id.

However in map-reduce the mapper tokenizes the user-id (disrributed into 20 jobs) and the reducers takes the input from the maps and maintains a sum of the partial data and then reduces it further until all the keys are taken care of.

Hence the mapreduce programs is expected to have a good speed up.