PARALLEL DATA PROCESSING IN MAPREDUCE – CS 6240
Prof. Mirek Riedewald
VIJET BADIGANNAVAR

Describe the steps taken by Spark to execute your source code. In particular, for each method invocation of your Scala Spark program, give a brief high-level description of how Spark applies it to the data. (10 points)

```
val DANGLING_NODES_KEY = "DANGLING_NODES"
//Initialize the SparkContext and set the configuration Parameters. local[*] specifies the
//context to use maximum number of cores available on the machine that is running the program
var conf = new SparkConf().setAppName("PageRankAlgorithm").setMaster("local[*]")
val sc = new SparkContext(conf)

/**
* Preprocessing Task. The map task fetches each line from the input file and sends it to
* the Parser to parse it. The parser is going to return the line in the format
* (link|outlink~outlink) in case of non-dangling-node and (link|) in case of dangling
* node and the result is stored as an RDD[String]. On the result of this map RDD, filter is applied to remove those
* nodes which caused the exception during parsing or invalid nodes in the input. Thus the resulting links is an
* RDD[String]
*/
val links = sc.textFile(args(0)).map(rec => com.vijet.parser.Parser.process(rec))
                                .filter(rec => (rec.trim.length >= 1))

/**
* Build the graph network: For each of the valid links obtained after pre-processing, split the string using map and
* check if the link is dangling. If the link is not a dangling node, then emit outlinks list as empty list otherwise split
* on it and return the outlinks as list. Thus, the output is an PairRDD I,e allLinks[String,List[String]]
*/
val allLinks = links.map { x => {
        val parts = x.split ( "\\|")
                if(parts.size == 1){
                        (parts (0), List[String]())
                }else{
                        (parts (0), parts (1).split("~").toList)
                }
        }
}
// This map function takes a link and returns a List(List(link,outlink)...)) for each of the outlinks associated with it
// along with the original link. Thus the input is an RDD[(List[List[(String,List[String]]])]
.map(x => {List(List((x._1,x._2)),x._2.map { y => (y,List[String]())})})
// The below two flatMap functions flatten the list that is generated in upper step and return a List of(link, List of
//outlinks). This is final links list that contains all the nodes given in the corpus. The first FlatMap produces an
//RDD[List[(String,List[String])])] & the second flatMap produces an PairRDD[(String,List[String])]
.flatMap { z => z }
 .flatMap { p => p }
// This reduceByKey combines all the outlinks associated with a particular link into a single list.The output will be
// a PairRDD[(String,List[String])]
 .reduceByKey((a,b) => (a++b)).persist()

//Total Number of links
val TOTAL_LINKS_COUNT = allLinks.count()
```

```
//Default Pagerank value
val DEFAULT_PR_SHARE = 1.0/TOTAL_LINKS_COUNT

/**
 * Default PageRank: Initialize all the pageranks of all the links to the default value I,e 1/Total_#_of_links.The
* output is a PairRDD[String,Double]
 */
var pageRanks = allLinks.keys.map{ link => (link,DEFAULT_PR_SHARE)}
/**
 * Distributed graph RDD that contains link->([outlinks],pagerank). This RDD will be during each iteration to
*  share their PR values with all the outlinks and also during computation of the PageRank associated with the
* dangling nodes. The output nodes is an PairRDD[String,(List[String],Double)]
*/
var nodes = allLinks.join(pageRanks);

// Iterate 10 times
for (iteration <- 1 to 10){

/**
* Iterate over the nodes and filter out the dangling nodes and compute the sum of pageranks of all the dangling
* nodes. The filter function will filter on the values of all the nodes whose outlinks list is empty.
* The corresponding reduce function will accumulate the scores associated with all the dangling nodes into a
* single record and value of that record is the total sum. The filter produces an Rdd(String,Double) and reduce
* produces an PairRDD(List[String],Double).The resulting value is a Double value danglingNodesPR.
*/

        val danglingNodesPR =   nodes.values.filter(x=>x._1.isEmpty)
                                        .reduce((p,q) => (List[String](),(p._2+q._2)))._2

/**
* Iterate over all the nodes and each of the page rank share to all the outlinks associated with
* that link. The FlatMap will take each of the flatten the outlinks list and sends the pagerank share to all the
* individual links. If the node is a dangling node then just emit empty list. The nodes that don't have any
* inlinks get lost during this process. The contributions is a PairRDD[(String,Double)]
*/
        val contributions = nodes.values.flatMap{
                case (outlinks,score) => {
                        if(outlinks.isEmpty){
                                List()
                        }else{
                                val PR_SHARE = score/outlinks.size
                                outlinks.map { outlink => (outlink, PR_SHARE)}
                        }
                }
        }

/**
* Compute the dangling score that should be added to each of the links given in the network.
*/

val danglingCorrection =  danglingNodesPR/TOTAL_LINKS_COUNT
/**
*  Accumulate the pagerank scores for a particular link and map it over to compute the new
*  pagerank including the danglingCorrection corresponding to the link. The reduceByKey function accumulates
* the pagerank associated with a particular key and mapValues function on that key to compute the new pagerank
* The result is an PairRDD[(String,Double)].
*/
        pageRanks = contributions.reduceByKey (_ + _).mapValues[Double] (p =>
                                        0.15 * DEFAULT_PR_SHARE + 0.85 * (danglingCorrection + p))

/**
* Distribute the pagerank share to those nodes that don't have any inlinks. This can be achieved by performing the
* leftOuterJoin on allLinks and updating the pagerank scores for those tuples whose pagerank is not defined.
* For the tuples whose pagerank is defined is already addressed in earlier step. The output will be an
* PairRDD[String,(List[String],Double)]
*/
        nodes = allLinks.leftOuterJoin(pageRanks).map(f => {
                    if(f._2._2 == None){
                              (f._1, (f._2._1,0.15 * DEFAULT_PR_SHARE + 0.85 * danglingCorrection))
                    }else{
                             (f._1, (f._2._1,f._2._2.get))
                    }})
        }

/**
*  Sort the ranks RDD by descending order of the values and take the top 100 records. The Map tasks inverts the
* key and values and top function takes the top 100 records with repartition factor as 1. The combined output is
* written to the file.The map functions returns a PairRDD[(Double,String)] and final output is an
* PairRDD[(Double,String)]
*/
        val top100Records = sc.parallelize(pageRanks.map(x => (x._2,x._1)).top(100),1).saveAsTextFile(args(1))
        // stop the spring context
        sc.stop()
```

Compare the Hadoop MapReduce and Spark implementations of PageRank.

For each line of your Scala Spark program, describe where and how the respective functionality is implemented in your Hadoop jobs.

<u>Preprocessing:</u>
In case of the Scala program, the preprocessing is performed as described in the above question.
In case of the Map reduce, this task was performed as a separate Mapper only task, which was reading the data from the input and parsing it. The validated line was written into the HDFS with the link|outlinks~~pagerankvalue.

<u>PageRank calculation:</u>
In case of the Scala program, the network is built and pagerank is calculated as described in the above question.
In Map reduce, initially a separate map only task is invoked that assigns the default page rank to all the links in the network. Then there is a corresponding Map-reduce tasks per iteration where in the mapper task, the dangling links page rank is share is distributed across all its outlinks along with its pagerank share. In the reduce task, these pagerank values for a particular link are accumulated. If the node is a dangling node then the page rank is added to the accumulator otherwise the new page rank is calculated for each of the link and is written back to the file on HDFS. The dangling nodes pagerank values are accumulated and sent back to the driver program. The driver takes this file and accumulator as input for next iteration and performs the next iteration.

Top-k calculation:

In case of scala program, how the top-100 records are selected are described above.
In case of Map reduce program, this was done using a separate map reduce task. In the mapper, individual mapper was emitting top 100 records with a dummy key. The reducer is iterating on a single key and is responsible for accumulating the top 100 records and these records were flushed to the HDFS.

Discuss the advantages and shortcomings of the different approaches. This could include, but is not limited to, expressiveness and flexibility of API, applicability to PageRank, available optimizations, memory and disk data footprint, and source code verbosity.

- I feel Map Reduce is something like a low level programming and Spark provides a high level language and has rich API like reduce, map, etc which makes the programming easy. Spark can also accommodate different programming languages like Python, Scala etc which makes it more user friendly than Map reduce which was only Java.
- Spark is more expressive and flexible than Map reduce. It has rich interfaces which makes the task of programming easier when compared to MapReduce
- Spark is more efficient than MapReduce. As spark performs lazy loading and it builds a DAG before any action is encountered and performs tasks efficiently.
- In Map reduce there is no option to store double value as an accumulator directly. There needs to be some tweaks that required to achieve this functionality. But, in spark this functionality is on the go.
- In case of iteration algorithms (like the graph with links -> [outlinks]), the intermediate values cannot be stored on the nodes in map reduce. So, this data has to be read and written from the HDFS during each of the iteration. But, in spark there is an option to persist the data onto the nodes and re use them during each iteration there by reducing the number of io. Spark stores the data in memory whereas Hadoop persists the data back to the disk
- As we can also see, the time taken to perform the pagerank in spark was less when compared to that of Mapreduce, Spark is more efficient.
- Mapreduce code can be more verbose. Spark code is succinct and has many rich API which makes it more programming friendly. (Though I felt scala is a bit challenging, may be because I had very little time learning the language)
- Although Spark stores the data in memory, when we perform an action the data is brought back to the driver and is executed on the driver so there is a potential to blow off the driver machine memory. So during the execution of this experiment, it's very important to know where to perform an action so that the driver is not blown off.

Performance Comparison

Machines	**Processing Time(Hadoop)**			**ProcessingTime(Spark)**		
	Start Time	**EndTime**	**Time(sec)**	**StartTime**	**EndTime**	**Time(sec)**
6 Machines	17:30:01,419	18:10:00,846	239000	06:27:44	07:52:11	503700
11 Machines	18:45:54,264	19:09:36,061	142000	23:26:07	00:07:17	247100

Discuss which system is faster and briefly explain what could be the main reason for this performance difference. (4 points)

The spark approach is faster than the map reduce one. Some of the potential reasons could be:

In the map reduce approach, there was much IO involved in reading the file and writing the file to the HDFS at each iteration. In spark, the there is an option to persist the RDD locally on each of the nodes, thus time involved in reading and writing the files to the HDFS is avoided thus increasing the efficiency.

In case of Hadoop each of the Map and reduce tasks should be executed in a separate JVM so there is time involved in launching and shutting the JVM. In case of Spark, all the JVMs associated with the nodes are started only once and is kept running, but different tasks are allocated on the go.

As Spark has a built in intelligence of DAG it may have optimized the ordering of map and reduce calls that may take less time to complete the entire job.

Report the top-100 Wikipedia pages with the highest PageRanks, along with their PageRank values, sorted from highest to lowest, for both the simple and full datasets, from both the Spark and MapReduce execution. Are the results the same? If not, try to find possible explanations. (4 points)

The results are attached with the submission.

There is slight difference between the files that are obtained in the Mapreduce and Spark, although they are same in general. The slight difference is mainly because of rounding off the pageranks in case of mapreduce job. As I had rounded off the total pagerank values of the dangling nodes into long so some of the precision was lost during each iteration. But, in Spark as we are using a filter to get hold of the page rank of dangling nodes this is more accurate.