K-Means clustering on MapReduce

Contents

Introduction

This handout covers the following topics:

- Instructions for the projects
- A comparison between MPI and MapReduce K-Means clustering implementation
- Quick overview of the MapReduce Skeleton using the WordCount example
- K-Means MapReduce Implementation (Mapper, Reducer, Combiner and configurations)
- Command line instructions

Instructions

- You will use Java for this assignment
- Refer to P3 handout for how K-Means clustering works
- You will be using your hadoop account on your clusters:
  hadoop@<andrew-id>.n01.qatar.cmu.edu
  if you have not changed your password already, please change it with the command: $passwd

MPI vs. MapReduce K-Means implementation

	MPI	**MapReduce**
Data partitioning	In MPI we partitioned the points from the master (rank == 0) and sent them to the slaves for processing.	Hadoop DFS is responsible for chunking the input files as we have seen in wordcount.
Centroids calculation	The centroids are initially picked by the master code and sent across all participating machines. This is because MPI is not a shared-memory model.	MapReduce is a shared-memory model, the centroids can be shared among iterations. To share the centroids, a file can be created on HDFS to include the initial K centroids (in iteration 0) and the updated centroids in each iteration. You can create a `FileSystem` in your program's `Configuration()`

MapReduce Skeleton

In this section, we will further analyze the wordcount example from the recitation and we will introduce key concepts that we will utilize in the K-Means implementation:

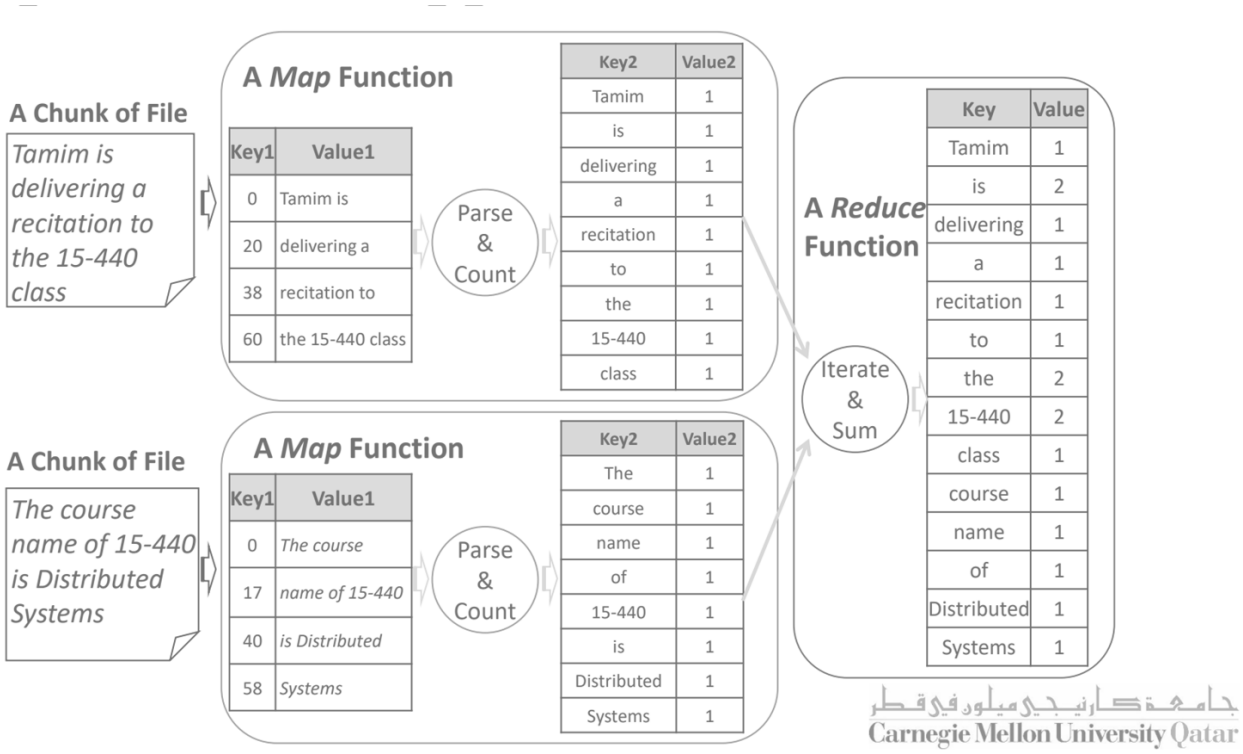

Figure 1MapReduce WordCount

Mapper class (i.e., WordCount mapper)

- Extends `MapReduceBase`
- Implements `Mapper <LongWritable, Text, Text, IntWritable>`
- Should implement the `map` function:
  `public void map (LongWritable key, Text value, OutputCollector<Text, IntWritable> output,Reporter report)`
  The map function takes a key, value, output collector and a reporter:
  - Key: represents the offset in the file (in Figure 1, key1)
  - Value: The value of the offset (In Figure 1, "Tamim is" for offset 0)
  - Output Collector: collects the output from the map function and feeds it into the Reduce phase. The type of the output collector depends on the <key,value> pair type of the map output (in Figure 1, the type is `<Text, IntWritable>`)
  - Reporter: reports any failure on the mapper

Reducer class (i.e., WordCoutReducer)

- Extands `MapReduceBase`
- Implements Reducer `Reducer <Text, IntWritable, Text, LongWritable>`
- Implements the `reduce` function with the following parameters:
  - Key: input key where the data is combined together (in case of word count, the key is the word.
  - Iterator for the values: iterates over the values assigned for a given key.
  - Output Collector: we output a word and its count, with types of output <key, value> pair as `<Text, LongWritable>`
  - Reporter: reports any failure on the reducer.

Main configurations

There is a set of configurations that should be considered in the main function, before running the job:

1. Defining a new job configuration: `new JobConf(class instance)`
2. Set the `mapper` and the `reducer` classes
3. Define the types of the map and reduce <key, value> output types:

```
conf.setMapOutputKeyClass(Text.class);
conf.setMapOutputValueClass(IntWritable.class);
conf.setOutputKeyClass(Text.class);
conf.setOutputValueClass(LongWritable.class);
```

4. Run the job

```
JobClient.runJob(conf);
```

KMeans on MapReduce

Map Phase

1) The mapper reads the data input file and gets the centroids from last iteration (or initial iteration)
2) The file is chunked and fed into the `map` function.
3) You should have predefined the mapper <key, value> pairs with the key being the offset and the value is a point read from the file.
4) For each find the nearest centroid and assign the point to it (Figure 2 illustrates the process)

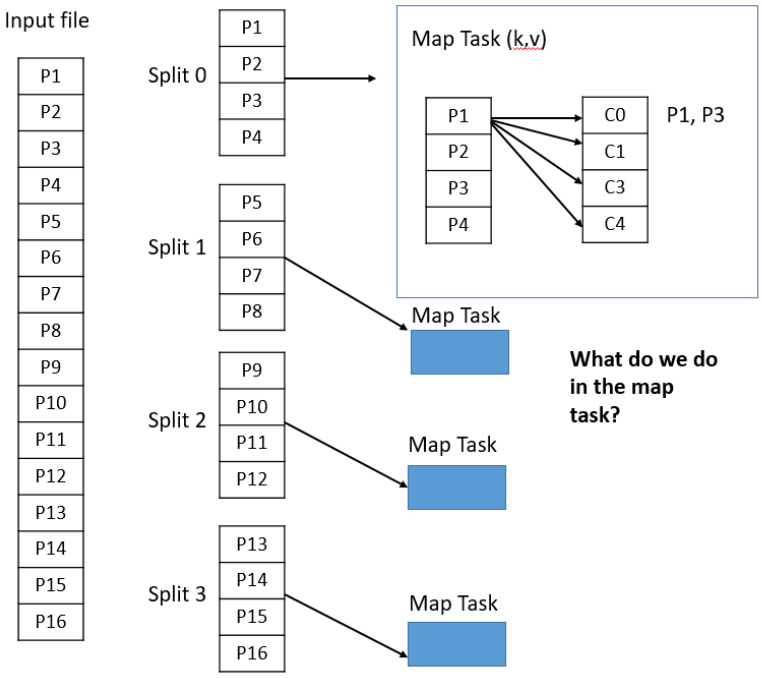

Figure 2 Kmeans Map Task

Combiner

The output of the map phase is huge (= total number of points) and we shall need to use a combiner to minimize the size of the data before sending it to the reducer. Let's see an example of a combiner in WordCount. Figure 3 shows a WordCount example without a combiner as opposed to Figure 3 with a combiner. The number of keys processed by the Reduce task is reduced from 9 to 4. This will be helpful in our KMeans implementation so that we minimize the number of points to be processed by the Reduce phase. The combiner calculates the average of the data instances for each cluster id, along with the number of the instances. It outputs (cluster id, (intermediate cluster centroid, number of instances)).

To define a combiner, you set it in your configuration as:

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

where IntSumReducer is a Reducer class.

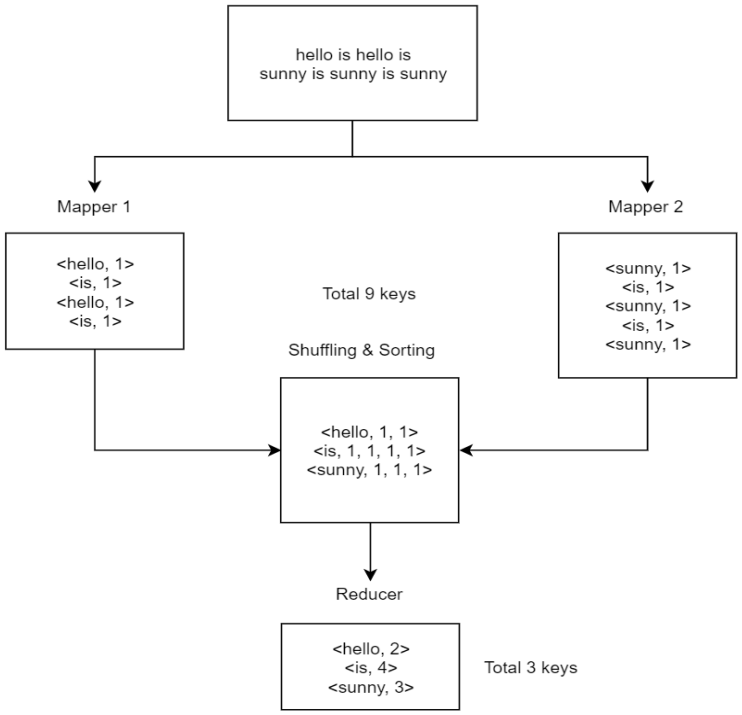

Figure 3 WordCount without a combiner

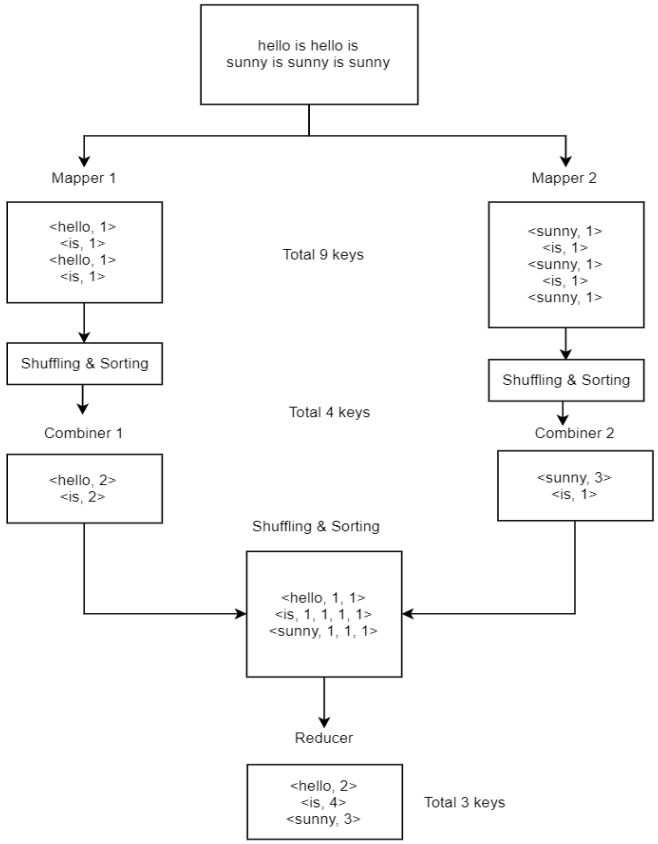

Figure 4 WordCount with a combiner

Reduce Phase

In the reduce phase, and based on the output of the combiner, you need to recalculate the centroids by iterating over the values and output the intermediate centroids. Since we are sharing the centroids among iterations, the centroid values have to be updated using the configuration file as stated in the previous section.

Running the Job

The main function involves two parts:

1. Configurations
2. Running multiple iteration jobs using the above Mapper + Combiner + Reducer. You can use the sample skeleton for the implementation (but feel free to use your own):

```
int iteration = 0;
// counter from the previous running import job
long counter = job.getCounters().findCounter(Reducer.Counter.CONVERGED).getValue();
iteration++;
while (counter > 0) {
        conf = new Configuration();
        conf.set("recursion.iter", iteration + "");
        job = new Job(conf);
        job.setJobName("KMeans " + iteration);
        // ...job.set Mapper, Combiner, Reducer... //
        // always take the output from last iteration as the input
        in = new Path("files/kmeans/iter_" + (iteration - 1) + "/");
        out = new Path("files/kmeans/iter_" + iteration);
        //... job.set Input, Output... //
        // wait for completion and update the counter
        job.waitForCompletion(true);
        iteration++;
        counter = job.getCounters().findCounter(Reducer.Counter.CONVERGED).getValue();
}
```

You can define the counter in the reducer class and update it as necessary:

```
        public enum Counter{
                CONVERGED //name of the counter
        }
        context.getCounter(Counter.CONVERGED).increment(1);
```

Command Line Instructions

How to run your code?

1. Create a folder for the .class files of your application:
   ```
   $ mkdir KMeans_Classes
   ```

2. Compile your KMeans program (where Kmeans.java is where you have your implementation)
   ```
   $ javac -classpath $(hadoop classpath) -d KMeans_Classes Kmeans.java
   ```

3. Create the jar file required by Hadoop to run your application using the following command:
   ```
   $ jar -cvf Kmeans.jar -C KMeans_Classes/ .
   ```

4. Create an input directory in HDFS using the following command:
   ```
   hadoop dfs –mkdir ./KmeansInput (for new versions)
   ```

5. Copy your points file to HDFS input directory using the following command:
   ```
   hadoop dfs - copyFromLocal points.txt ./KmeansInput
   ```

6. You can check if the points file was copied correctly by:
   ```
   hadoop dfs –ls ./ KmeansInput
   ```

7. Run your Kmeans application (where KMeans is the classname, if KmeansOutput is not created, please create it as you created the KmeansInput)
   ```
   hadoop jar Kmeans.jar KMeans ./KmeansInput ./KmeansOutput 100 3 5
   ```

8. You can check your output file:
   ```
   hadoop dfs -ls ./KmeansOutput
   ```

Troubleshooting

If you face any issues, try to restart using the below commands and then restart your cluster

```
1) #stop-all.sh
2) #start-all.sh
```