Project 7: Harp K-Means
Cloud Computing
Spring 2017

Professor Judy Qiu

Goal

The goal of this project is to familiarize yourself with the concept of map-collective applications. Harp is similar to MapReduce in programming with the exception that it provides collective communication support across map tasks.

Deliverables

Zip your source code and output as username_harp-kmeans.zip. Please submit this file to the Canvas Assignments page.

Evaluation

The point total for this project is 20, where the distribution is as follows:

- Completeness of your code (16 points)
- Correct output (4 points)

Prerequisites

- Before working on Harp K-Means, make sure you can successfully run Harp WordCount following the instructions of Lab 10 (the presentation is in Canvas).
- Download **simplekmeans** folder from Canvas assignments under **B649_Project7** folder.
- Copy that folder to **harp2-project-master/harp2-app/src/edu/iu**
- Open **harp2-project-master/harp2-app/build.xml** and add the following line next to where you find other **include** tags. Note: this is within the **javac** tag of the **build.xml**
  <include name="edu/iu/simplekmeans/" />**

K-Means Clustering Algorithm

K-Means is a clustering algorithm to partition n observations $(x_1, x_2, ..., x_n)$ into k ($\leq n$) sets $S = \{S_1, S_2, ..., S_k\}$ clusters in which each observation belongs to the cluster with the nearest mean of Euclidean distance. The objection function is to minimize the sum of distance functions for each point to centroids, where μ_i is the mean of points in S_i.

$$\arg\min_{S} \sum_{i=1}^{k} \sum_{x \in S_i} ||x - \mu_i||^2$$

The input is data points (data) and the model is cluster centers (centroids). The problem is computationally difficult (NP-hard); however, there are efficient algorithms via an iterative refinement approach.

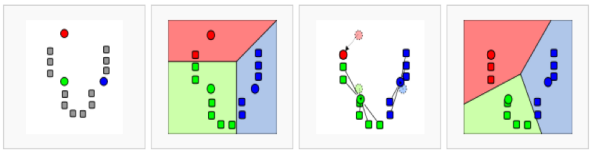

Figure 1: Image source: https://en.wikipedia.org/wiki/K-means_clustering

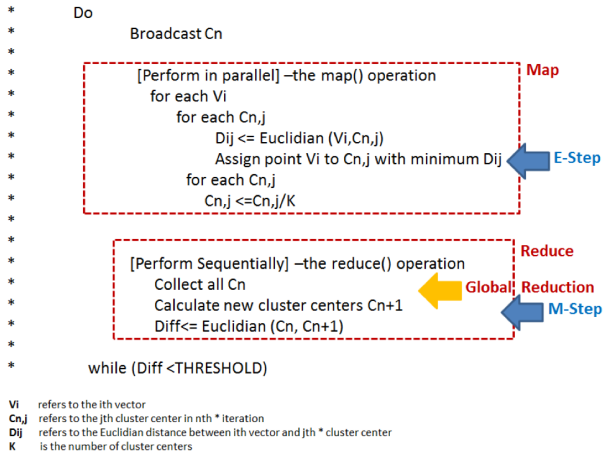

Figure 2: K-Means Clustering for MapReduce

Harp Data Structure

The main data structure used for this assignment is ArrTable<DoubleArray>. You can think of this as a collection of double array objects. Each double array represents a center. For example, if the centers for your program are 3D, then each array will have $3+1$ (4) elements. The first 3 will be the x, y, z coordinates. The last element is used to store the number of points assigned to this center.

To retrieve all the centers you can invoke the **getPartitions()** method on ArrTable object, which will return a collection of ArrPartition objects. To retrieve the underlying double array from these ArrPartition objects, you can invoke the **getArray()** method on ArrPartition object. To figure out the index (ID) of this center you can invoke **getPartitionID()** method on the same ArrPartition object.

Harp Implementation

Most of the code is completed for you; your task will be to perform the nearest center finding computation and updating new centers. The code to implement this is in the **simplekmeans/KmeansMapper.java**

The two functions are listed below.

Finding Nearest Center

```
1 private void findNearestCenter(ArrTable<DoubleArray> cenTable, ArrTable<DoubleArray>
    previousCenTable, ArrayList<DoubleArray> dataPoints) {
2   double err = 0;
3   for (DoubleArray aPoint : dataPoints) {
4     //for each data point, find the nearest centroid
5     double minDist = -1;
```

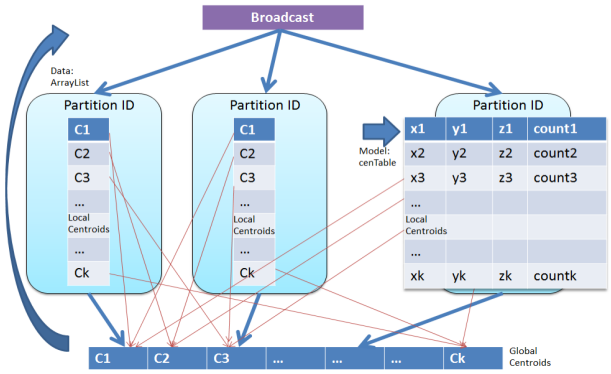

Figure 3: Parallelization of K-means Clustering

```
    double tempDist = 0;
    int nearestPartitionID = -1; // Keeps track of the nearest center ID
    for (ArrPartition ap : previousCenTable.getPartitions()) {
      DoubleArray aCentroid = (DoubleArray) ap.getArray();
      /* TODO     Write code here */
      tempDist = // this is the Euclidean distance between aPoint and aCentroid. Use Utils.calcEucDistSquare() for this

      // Next check if tempDist is smaller than current minDist (unless it s -1)
      // If so, set the minDist to tempDist and mark the current center ID as the nearest center found so far. To get the center ID use the ap.getPartitionID() method.
    }
    err += minDist;

    //for the certain data point, found the nearest centroid.
    // add the data to a new cenTable.
    double[] partial = new double[vectorSize + 1];
    for (int j = 0;j < vectorSize;j++) {
          partial[j] = aPoint.getArray()[j];
        }
        partial[vectorSize] = 1;

        if (cenTable.getPartition(nearestPartitionID) == null) {
          ArrPartition<DoubleArray> tmpAp = new ArrPartition<DoubleArray>(nearestPartitionID, new DoubleArray(partial, 0, vectorSize + 1));
          cenTable.addPartition(tmpAp);
        }
        else {
            ArrPartition<DoubleArray> apInCenTable = cenTable.getPartition(nearestPartitionID);
            for (int i = 0;i < vectorSize + 1;i++) {
              apInCenTable.getArray().getArray()[i] += partial[i];
            }
        }
    }
  System.out.println("Errors: " + err);
}
```

Updating Centers

```
private void updateCenters(ArrTable<DoubleArray> cenTable) {
  for (ArrPartition<DoubleArray> partialCenTable : cenTable.getPartitions()) {
    double[] doubles = partialCenTable.getArray().getArray();
    /* TODO     Write code here */
    // Go through the components of the vector (i.e. a for loop) and divide it by the number of points assigned to that center.
    // The number of points can be found at doubles[vectorSize]
    // Vector components (x,y,z, etc.) can be found through 0 to (vectorSize     1) indices
```

```
    in the doubles array

    doubles[vectorSize] = 0;
  }
  System.out.println("after calculate new centroids");
  Utils.printArrTable(cenTable);
}
```

Compilation and Running

- To compile the code, simply go into **harp2-project-master/harp2-app** and type **ant**
- Then copy the **harp2-project-master/harp2-app/build/harp2-app-hadoop-2.6.0.jar** to **$HADOOP_HOME**
- To run the file, use the following command within **$HADOOP_HOME**. This will produce 100 data points and cluster them into 12 clusters. We use 3D points. The program will run 2 parallel map tasks for 20 iterations. Note: you may want to give unique directory names for the last two parameters each time that you test. Otherwise, you may run into issues because of existing directories. Alternatively, you can delete old directories and reuse the same names.

  The folder **harpkmeans** will be in HDFS. The **/tmp/simplekmeansdata** will be in your local disk.

```
$ hadoop jar harp2-app-hadoop-2.6.0.jar edu.iu.simplekmeans.KmeansMapCollective 100 12
    3 2 20 harpkmeans /tmp/simplekmeansdata
```

- To get the output do the following and look at the part- files as usual.

```
$ hdfs dfs -get harpkmeans/out .
```