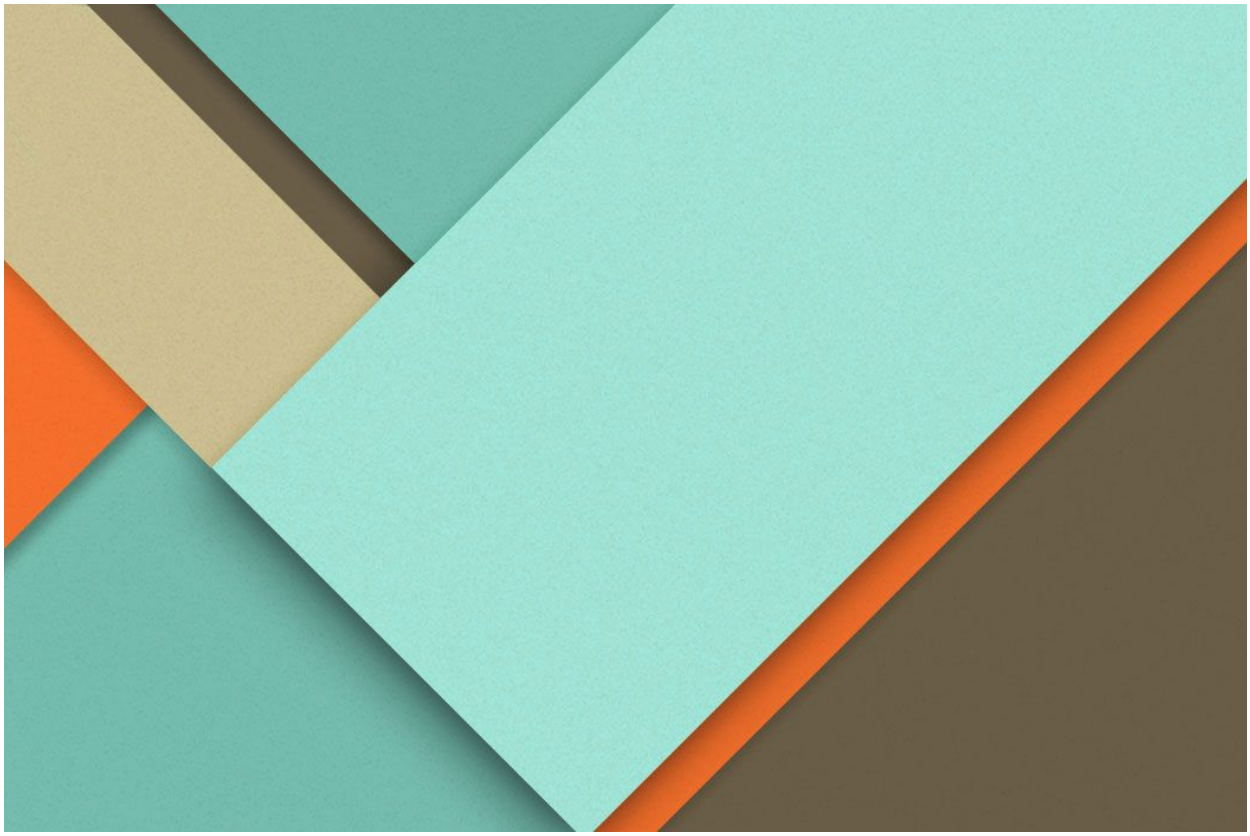


Parallel K-Means using Hadoop

07-03-2019

Team Members

Abdelrhman Yasser - 37

Amr Mohamed Fathy - 46

Mohamed Ibrahim Shapan - 58

Overview

A very common task in data analysis is grouping a set of unlabeled data such that all elements within a group are more similar among them than they are to the others. This falls under the field of unsupervised learning. Unsupervised learning techniques are widely used in several practical applications, e.g. analyzing the GPS data reported by a user to identify her most frequent visited locations. For any set of unlabeled observations, clustering algorithms tries to find the hidden structures in the data.

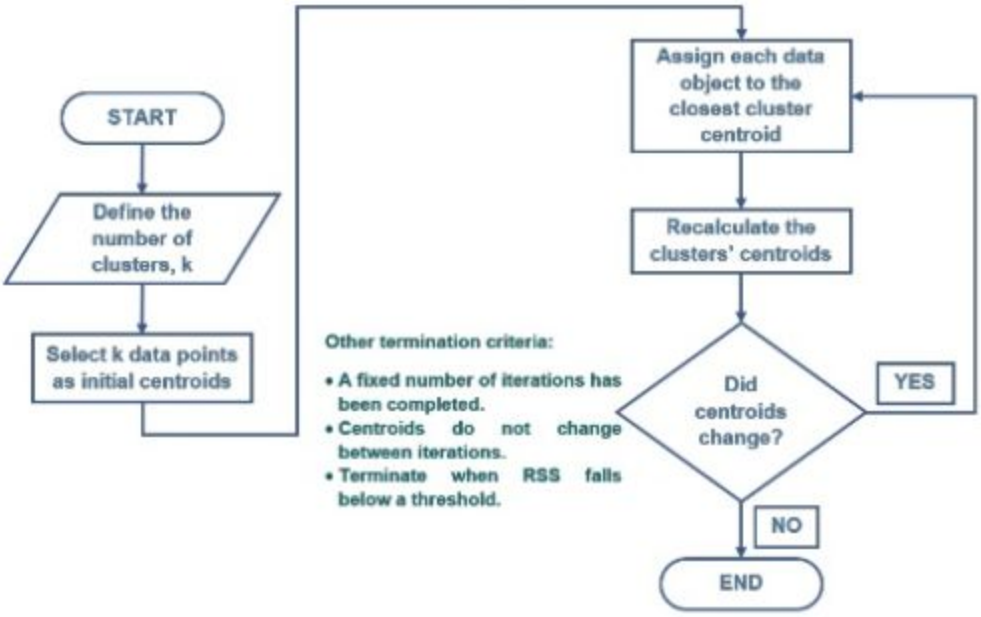
With the development of information technology, data volumes processed by many applications will routinely cross the peta-scale threshold, which would in turn increase the computational requirements. Efficient parallel clustering algorithms and implementation techniques are the key to meeting the scalability and performance requirements entailed in such scientific data analyses.

The Hadoop and the MapReduce programming model represents an easy framework to process large amounts of data where you can just implement the map and reduce functions and the underlying system will automatically parallelize the computations across large-scale clusters, handling machine failures, inter-machine communications, etc. In this assignment you are asked to make a parallel version of the well-known and commonly used K-Means clustering algorithm using the map-reduce framework.

Goals

1. Implementing a parallel version of the K-Means algorithm using the MapReduce framework.
2. Evaluating the parallel version and compare it with the unparalleled version of K-Means using the IRIS dataset here in terms of run time and clustering accuracy.

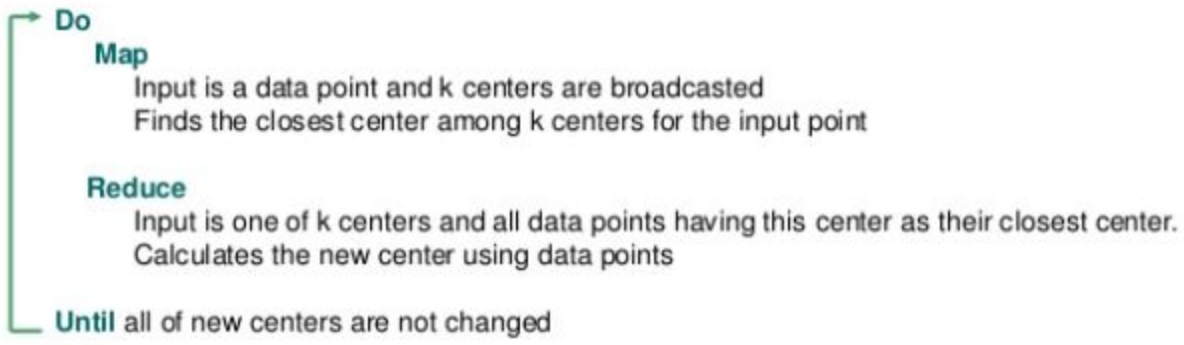
Unparalleled K-Means



Parallel K-Means Idea

We will make each iteration of the K-Means as a map-reduce phase.

We will follow the following map and reduce procedure.

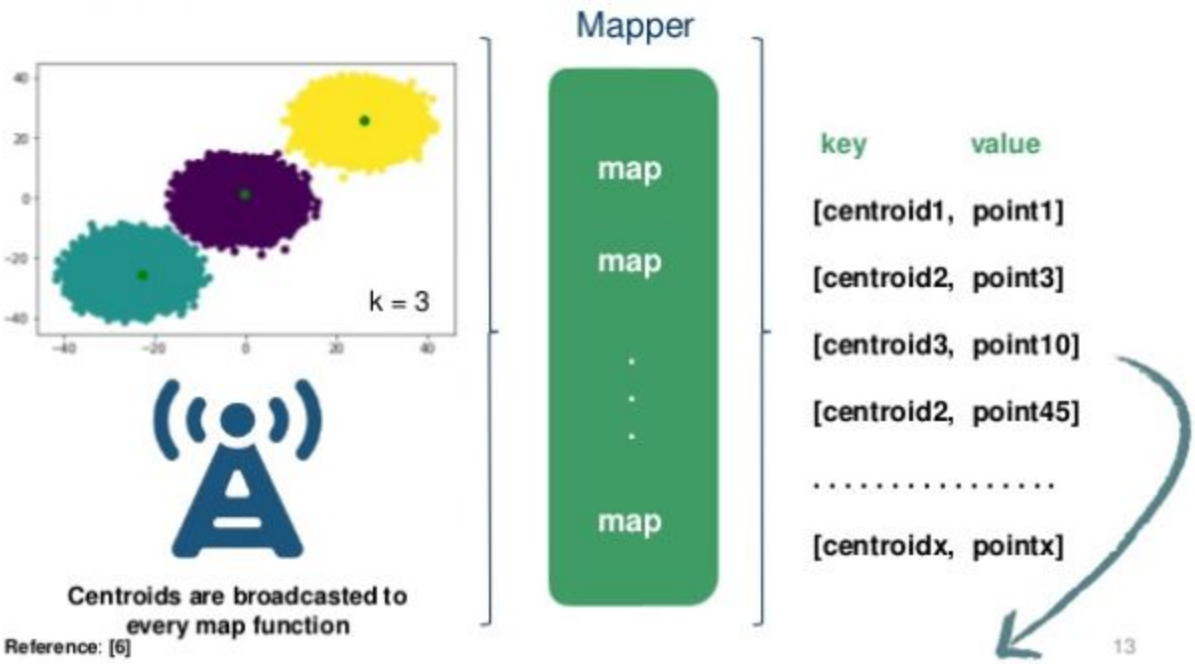


We will see each phase in details.

We made an extra initialization map-reduce phase at first to build our centroids at beginning, we selected the first k points to be the initial centroids.

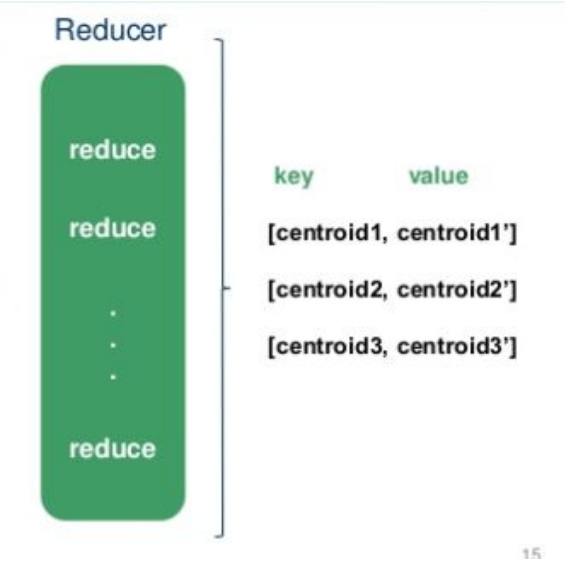
Map Phase

The Mapper has input of a single point then it calculate the closet cluster centroid to be assigned to it.



Reduce Phase

The Reducer will take the cluster index as a key, and the list of points which are assigned to that cluster. Then it will update the cluster centroid by finding the mean of the assigned points.



Running UnParalleled K-Means

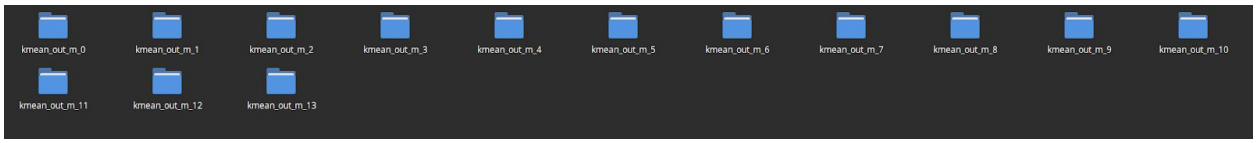
```
****************************************
6.462499999999997 2.9612500000000006 5.173750000000001 1.8012500000000002
5.574999999999999 2.56875 3.975 1.2125
5.005999999999999 3.4180000000000006 1.464 0.2439999999999999
5.0 2.3 3.275 1.025
****************************************
6.570149253731341 2.988059701492538 5.338805970149254 1.885074626865671
5.724137931034482 2.6827586206896554 4.131034482758621 1.282758620689655
5.005999999999999 3.4180000000000006 1.464 0.2439999999999999
5.0 2.3 3.275 1.025
****************************************
6.632203389830504 2.9983050847457617 5.4305084745762695 1.9372881355932199
5.808108108108106 2.732432432432432 4.245945945945945 1.3297297297297297
5.005999999999999 3.4180000000000006 1.464 0.2439999999999999
5.0 2.3 3.275 1.025
****************************************
6.692156862745095 3.0117647058823525 5.541176470588233 1.9882352941176467
5.8999999999999995 2.776744186046512 4.365116279069768 1.397674418604651
5.005999999999999 3.4180000000000006 1.464 0.2439999999999999
5.2 2.3666666666666667 3.3833333333333333 1.0166666666666666
****************************************
6.802325581395348 3.0441860465116273 5.648837209302324 2.0302325581395344
5.970212765957446 2.8 4.514893617021277 1.4744680851063827
5.005999999999999 3.4180000000000006 1.464 0.2439999999999999
5.3100000000000005 2.4699999999999998 3.55 1.1
****************************************
6.8500000000000005 3.073684210526315 5.742105263157893 2.0710526315789473
6.041666666666667 2.8333333333333326 4.604166666666666 1.5229166666666665
5.005999999999999 3.4180000000000006 1.464 0.2439999999999999
5.421428571428572 2.457142857142857 3.6714285714285713 1.1285714285714286
****************************************
6.8742857142857146 3.088571428571428 5.79142857142857 2.117142857142857
6.128260869565217 2.8565217391304345 4.6869565217391305 1.5478260869565215
5.005999999999999 3.4180000000000006 1.464 0.2439999999999999
5.457894736842105 2.5105263157894733 3.805263157894737 1.1736842105263159
****************************************
6.9 3.0969696969696963 5.827272727272726 2.127272727272727
6.188636363636363 2.859090909090909 4.752272727272728 1.593181818181818
5.005999999999999 3.4180000000000006 1.464 0.2439999999999999
5.48695652173913 2.5739130434782607 3.8782608695652177 1.1869565217391305
****************************************
6.9125000000000005 3.099999999999999 5.846874999999999 2.1312499999999996
6.216279069767441 2.86046511627907 4.786046511627907 1.6116279069767434
5.005999999999999 3.4180000000000006 1.464 0.2439999999999999
5.507999999999999 2.6 3.908 1.204
****************************************
6.9125000000000005 3.099999999999999 5.846874999999999 2.1312499999999996
6.2285714285714295 2.861904761904762 4.792857142857143 1.6190476190476184
5.005999999999999 3.4180000000000006 1.464 0.2439999999999999
5.515384615384614 2.6076923076923078 3.9307692307692315 1.2076923076923078
****************************************
6.9125000000000005 3.099999999999999 5.846874999999999 2.1312499999999996
6.23658536585366 2.8585365853658535 4.807317073170731 1.6219512195121943
5.005999999999999 3.4180000000000006 1.464 0.2439999999999999
5.529629629629629 2.6222222222222222 3.940740740740741 1.2185185185185188
Time token by un-parallel is : 56ms
```

Running Parallel K-Means

```
2019-03-07 12:12:32,383 INFO  [pool-43-thread-1] reduce.MergeManagerImpl (MergeManagerImpl.java:finalMerge(806)) - Merging 0 segments, 0 bytes from memory into reduce
2019-03-07 12:12:32,383 INFO  [pool-43-thread-1] mapred.Merger (Merger.java:merge(606)) - Merging 1 sorted segments
2019-03-07 12:12:32,383 INFO  [pool-43-thread-1] mapred.Merger (Merger.java:merge(705)) - Down to the last merge-pass, with 1 segments left of total size: 5446 bytes
2019-03-07 12:12:32,383 INFO  [pool-43-thread-1] mapred.LocalJobRunner (LocalJobRunner.java:statusUpdate(620)) - 1 / 1 copied.
2019-03-07 12:12:32,394 INFO  [pool-43-thread-1] mapred.Task (Task.java:done(1105)) - Task:attempt_local1445619809_0014_r_000000_0 is done. And is in the process of committing
2019-03-07 12:12:32,394 INFO  [pool-43-thread-1] mapred.LocalJobRunner (LocalJobRunner.java:statusUpdate(620)) - 1 / 1 copied.
2019-03-07 12:12:32,394 INFO  [pool-43-thread-1] mapred.Task (Task.java:commit(1272)) - Task attempt_local1445619809_0014_r_000000_0 is allowed to commit now
2019-03-07 12:12:32,395 INFO  [pool-43-thread-1] output.FileOutputCommitter (FileOutputCommitter.java:commitTask(582)) - Saved output of task 'attempt_local1445619809_0014_r_000000_0' to file
2019-03-07 12:12:32,396 INFO  [pool-43-thread-1] mapred.LocalJobRunner (LocalJobRunner.java:statusUpdate(620)) - reduce > reduce
2019-03-07 12:12:32,396 INFO  [pool-43-thread-1] mapred.Task (Task.java:sendDone(1231)) - Task 'attempt_local1445619809_0014_r_000000_0' done.
2019-03-07 12:12:32,396 INFO  [pool-43-thread-1] mapred.LocalJobRunner (LocalJobRunner.java:run(352)) - Finishing task: attempt_local1445619809_0014_r_000000_0
2019-03-07 12:12:32,398 INFO  [Thread-371] mapred.LocalJobRunner (LocalJobRunner.java:runTasks(485)) - reduce task executor complete.
2019-03-07 12:12:33,321 INFO  [main] mapreduce.Job (Job.java:monitorAndPrintJob(1640)) - Job job_local1445619809_0014 running in uber mode : false
2019-03-07 12:12:33,321 INFO  [main] mapreduce.Job (Job.java:monitorAndPrintJob(1647)) -  map 100% reduce 100%
2019-03-07 12:12:33,322 INFO  [main] mapreduce.Job (Job.java:monitorAndPrintJob(1658)) - Job job_local1445619809_0014 completed successfully
2019-03-07 12:12:33,325 INFO  [main] mapreduce.Job (Job.java:monitorAndPrintJob(1665)) - Counters: 30
        File System Counters
                FILE: Number of bytes read=412152
                FILE: Number of bytes written=9643756
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
        Map-Reduce Framework
                Map input records=150
                Map output records=150
                Map output bytes=5150
                Map output materialized bytes=5456
                Input split bytes=108
                Combine input records=0
                Combine output records=0
                Reduce input groups=4
                Reduce shuffle bytes=5456
                Reduce input records=150
                Reduce output records=4
                Spilled Records=300
                Shuffled Maps =1
                Failed Shuffles=0
                Merged Map outputs=1
                GC time elapsed (ms)=9
                Total committed heap usage (bytes)=2710568960
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=4549
        File Output Format Counters
                Bytes Written=314
```

After finishing each map-reduce phase, we will find an output directory as a result containing the reducer output file.

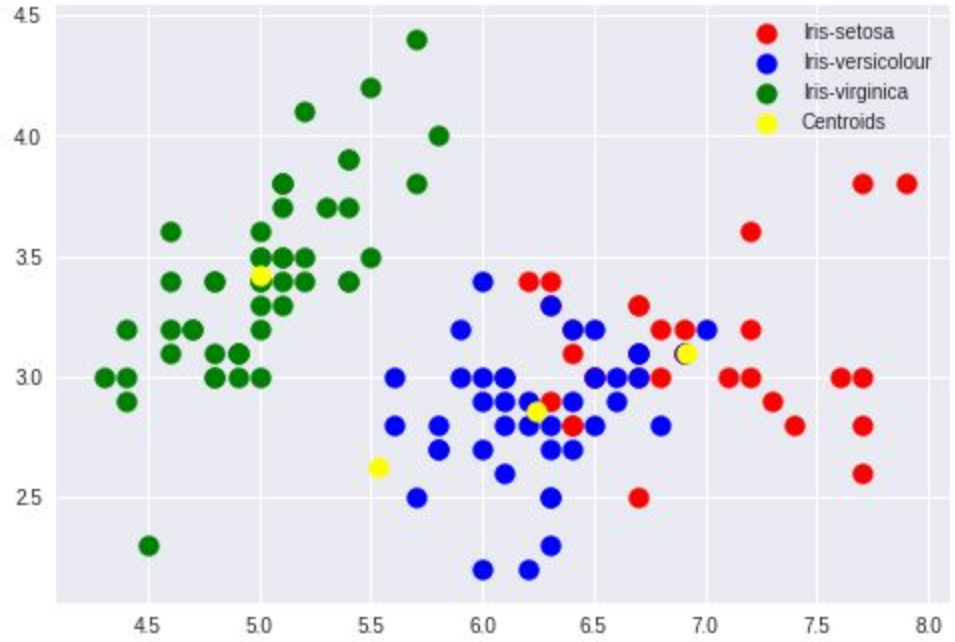
K-Means will finish after 14 iterations, so we will find 14 output directory.



Results

Both the parallel K-Means and unparallel K-Means versions will have the same final cluster centroids as following:

6.9125	3.1	5.846875	2.13125
6.23658537	2.85853659	4.80731707	1.62195122
5.006	3.418	1.464	0.244
5.52962963	2.62222222	3.94074074	1.21851852



K-Means Version	Time Token (ms)
Parallel	15462
Unparalleled	56

Notice that the same results obtained using the built-in K-Means in Sklearn.clusters package in python.

You can see the implementation and visualization from here.

References

- https://www.slideshare.net/StratosGounidellis/kmeans-algorithm-implementation-on-hadoop-79249681
- https://www.linkedin.com/pulse/back-first-principle-clustering-algorithm-java-using-bowen-gong/
- https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html
- https://www.kaggle.com/ayanmaity/kmeans-clustering-on-iris-data