K-means clustering algorithm with MapReduce paradigm

Stefano Costanzo, 2022.

Stefano.costanzo.97@gmail.com.

Progetto B1 SDCC. Laurea magistrale Engineering Informatic.

***Abstract*-- The aim of the project is to realise a distributed application in the Go programming language that implements the k-means clustering algorithm in a distributed version according to the MapReduce computing paradigm. The k-means algorithm is a clustering algorithm that makes it possible to divide a set of points into k groups or clusters on the basis of Euclidean distance.**

I. INTRODUCTION

MapReduce is a style of computing that has been implemented in several systems, including *Google's internal implementation* (symply called MapReduce) and a popular open-source implementation *Hadoop*[5] which can be obtained, along with the *HDFS* file system from the Apache Foundation. All we need is to define two functions, called **Map** and **Reduce**.

MapReduce systems splits the data into **smaller chunks**, processes them in parallel (Mapper phase) and aggregates all the data from multiple servers (Reducer phase) to return a consolidated output back to the application.

Our goal is to implement the K-means algorithm that allows a set of points to be divided into **k groups** or clusters on the basis of a Euclidean Distance. The implentation is realized with the MapReduce paradigm to manage many large-scale, **parallel**, computations fast and in a way that is tolerant to **hardwars faults**.

A master-worker architecture is implemented, in which the master distributes the workload among the node-workers, who implement the mappers and reducers.

Docker containers are used to provide the relevant image files, and to deploy the application on an **EC2** instance. Many important tools were used such as Docker compose, Ansible, and Aws grant.

II. DESIGN CHOICES

A. Infrastructure

- Master: Distributes the workload among the node-workers, execute the reorganization of mapper results (shuffle-and-sort) and manage fails.

- Mapper: Get chuncks of points and turn them in a sequence of key-value pairs and send results to master

- Reducer: Get organizes Key-value from master, combine the values and send the merged output

B. One Reducer

In the k means implantation, the workload of the reducer is small compared to the other workers, so the choice was made to implement it as a single instance. In addition this helps an easier management (single output, no division by key)

C. Combiners

However, a single reducer obviously highlights the bottleneck of the application. Therefore, the use of combiners was opted in order to lighten the workload. Part of the reduce work is moved to the mappers and executed in parallel. the output key-value of the mapping phase is aggregated.

I still chose to leave the old version without combiners. Available in the attached directory (*MasterMR_oldVersion*).

Mapper output:

Old Version: *[key:value] = [center index: Observation Coordinates]*

New Version: *[key:value] = [center index: Sums points and number aggregated points] [IMAGE1]*

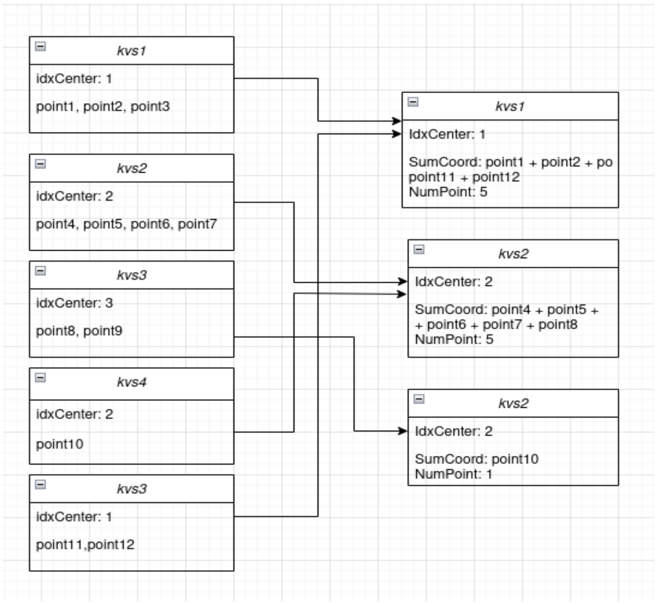


Figure 1) Combine mapper results

D. Communication

Nodes communicate with each other through the **Remote Procedure Call** mechanism. For the implementation of RPC, gRPC was not used, but the package package "net/rpc" was used since the application is totally written in Go and therefore versatility with regard to the programming language was not needed.

Figure2 shows all the overall described infrustrure

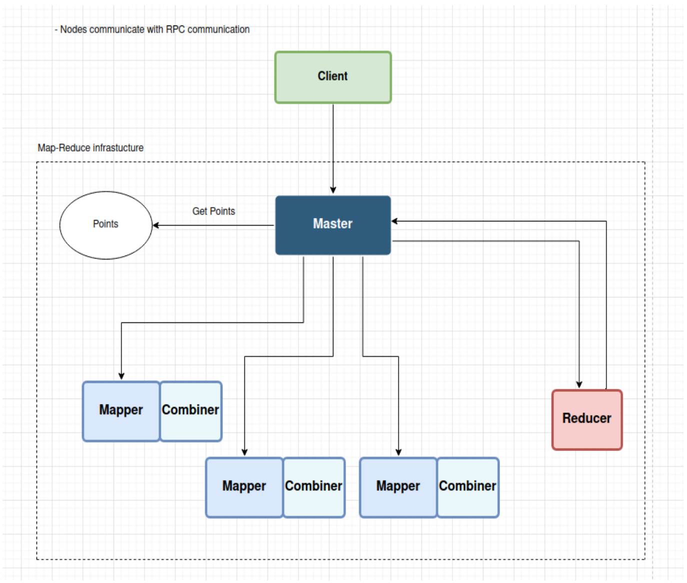


Figure 2) Map-Reduce Infrastructure

III. Data structures

A. **Cluster structures** [2]

- *Centre coordinates*

- *Array of coordinates.* Stores all point belonging to the cluster

B. **K-means structure**

- *Port master*: Main port accessed for communication (default: 5000)

- *Mappers* : Index of available mappers: If some mapper fail is removed from the list

- *Deltathreshold* (0.0 to 1.0 in percentage): Aborts communication is general distance satisfies the criteria.

- *IterationThreshold*: Maximum number of iterations

C. Points Folder

The points folder has within it several sets of observations that can be used to execute the algorithm and an executable in go for generating new point files (*genPoints.go*). The results of the execution are then stored in the charts folder - these include random point files in the 0-1 range (*rand1000.txt* and *rand10000.txt*), quadratic, linear and square root functions

D. **Input/Output**

The information in I/O is restricted to a minimum. For example, the output of the reducer is only the new position of the clusters and not the entire structure with the assignment of points to them.

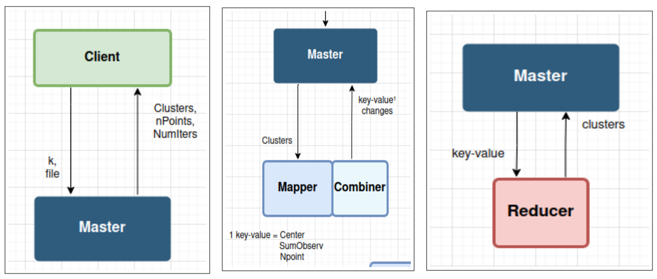


Figure 3) Input/Output communication

IV. Nodes

A. **Master**

The task of the master is to dialogue with the client and manage communication and the proper functioning of the system. The operations performed are in order:

1) **Initialising** main data structures: Threads, channels, dataset and main configurations passed as input by the client

2) Send clusters **coordinates** to mappers and wait results of rpc call. If some **mapper fail** remove from the list of available workers, reassigning chunks of points to mappers and restart.

3) **Shuffle and sort** key-value mapped data

4) Send Key-value to reducer and get clusters result

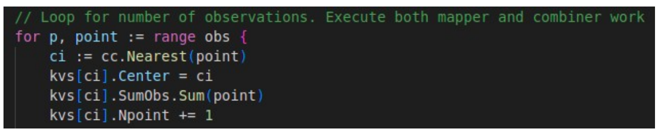
5) **Verify condition** to Loop: If execution has not yet finished, continue from point 2

6) **Return clusters coordinates**

B. **Mapper**

Running many mappers in parallel does the bulk of the work. As soon as they are initialised, they obtain the chunk of points to be analysed. Calculate the **minimum Euclidean** distance and assign the point to the corresponding cluster.

Performs a part of the reducer work by **combining** the points of each cluster (calculates the partial sum and the number of points assigned to each cluster) and sends the result to the master



```
// Loop for number of observations. Execute both mapper and combiner work
for p, point := range obs {
    ci := cc.Nearest(point)
    kvs[ci].Center = ci
    kvs[ci].SumObs.Sum(point)
    kvs[ci].Npoint += 1
```

Figure 4) Combining code

C. **Reducer**

Obtains the key values from the master combines them and recalculates the centre of the clusters. It then sends the result to the master.

Prevents clusters from running out of points by reassigning points appropriately: The **empty cluster** takes a point from another cluster with at least two elements.

V. Containers and AWS

To manage the instantiation of containers, we used **Docker** and the base image selected is golang:1.16-alpine as it has a smaller footprint than, for instance, Ubuntu and contains everything needed to support the application.

Docker Compose manages the orchestration of the containers: Is assumed port 5000 to communicate with the server. Uses Environment variables to have configurable parameters: The number of mappers, the maximum amount of iterations and the threshold of confidency to set the convergence limit, using the minimization of the sum of squares within the cluster as a criterion.

Ansible installs and configurates Docker software and copied the necessary files to the machine.
hosts.ini marks the **EC2** instance to be used.

VI. Testing

Two executables are provided to test the correct functioning of the application:

- ***client.go*** to execute a specific point file (present in the folder folder) by setting the number of desired clusters; The test

- ***test.go*** to execute Map-Reduce several times with different values: File of points to be analysed, number of mappers, number of clusters, convergence limits (delta threshold) and fixed limits (maximum iterations).

Entering test.go you can configure the testing values differently.

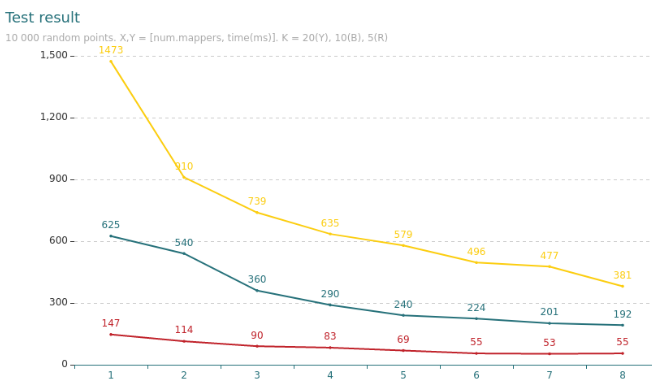


Figure 5) Test chart: [x, y] = [num.mappers, time]

The figure shown presents the algorithm running on **10 000 random points**. The 3 curves have different number of clusters: k = 20 (yellow), k = 10 (blue), k = 5 (red) the execution times are shown in ms according to the number of mappers (from 1 to 8) As expected, there is a substantial increase in performance as the number of mappings increases.

- From 1 to 3 mappers the performance increases by about **200%**.

- From 1 to 8 mappers the performance increases by about **390%**.

Moreover, as it was easy to expect, there is a degradation of performance as the number of clusters increases. Doubling the number of clusters would also seem to **double the execution times.**

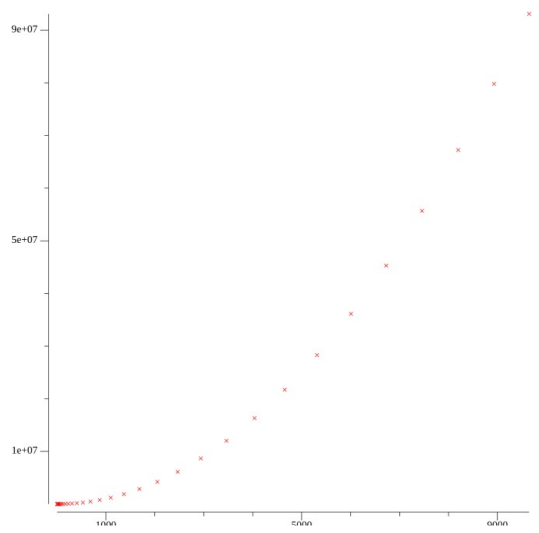
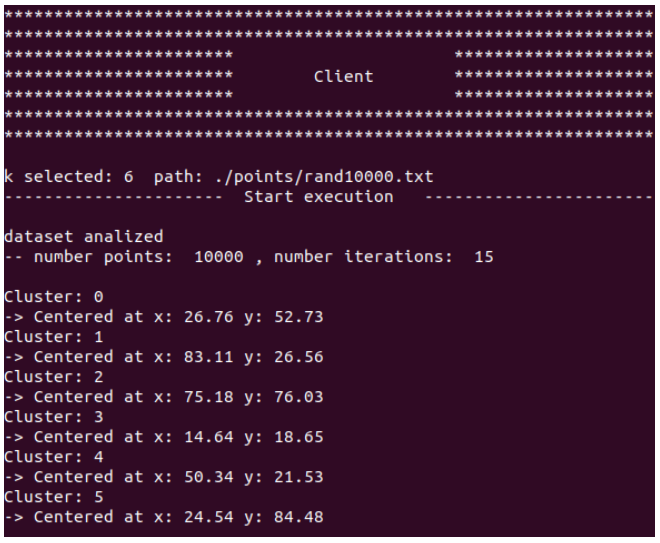


Figure 6) chart of an quadratic function: The red crosses are the cluster coordinates

VII. Console

Client:



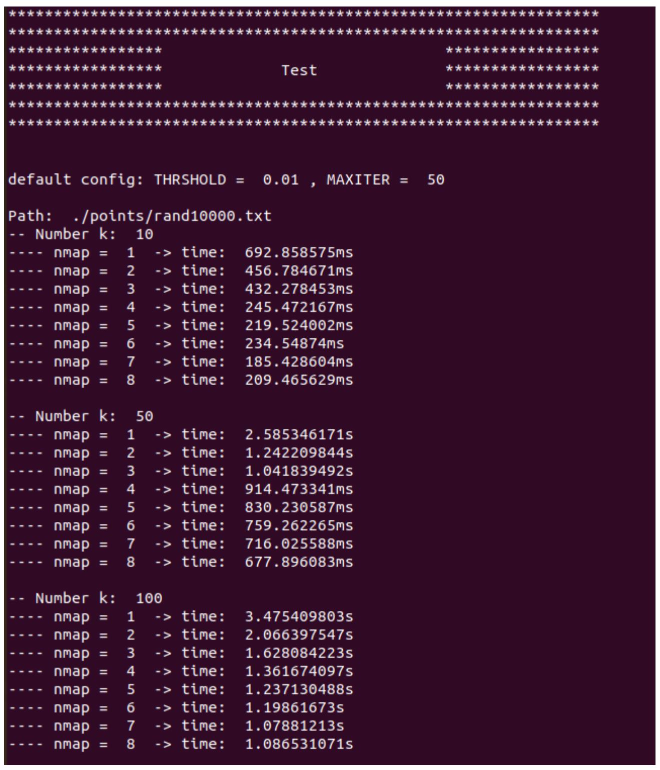
```
*******************************************************************
*******************************************************************
**********************                      ********************
**********************         Client       ********************
**********************                      ********************
*******************************************************************
*******************************************************************

k selected: 6  path: ./points/rand10000.txt
----------------------  Start execution   -----------------------

dataset analized
-- number points:  10000 , number iterations:  15

Cluster: 0
-> Centered at x: 26.76 y: 52.73
Cluster: 1
-> Centered at x: 83.11 y: 26.56
Cluster: 2
-> Centered at x: 75.18 y: 76.03
Cluster: 3
-> Centered at x: 14.64 y: 18.65
Cluster: 4
-> Centered at x: 50.34 y: 21.53
Cluster: 5
-> Centered at x: 24.54 y: 84.48
```

Test:



```
*****************************************************************
*****************************************************************
*****************                               *****************
*****************              Test             *****************
*****************                               *****************
*****************************************************************
*****************************************************************


default config: THRSHOLD =  0.01 , MAXITER =  50

Path:  ./points/rand10000.txt
-- Number k:  10
---- nmap =  1  -> time:  692.858575ms
---- nmap =  2  -> time:  456.784671ms
---- nmap =  3  -> time:  432.278453ms
---- nmap =  4  -> time:  245.472167ms
---- nmap =  5  -> time:  219.524002ms
---- nmap =  6  -> time:  234.54874ms
---- nmap =  7  -> time:  185.428604ms
---- nmap =  8  -> time:  209.465629ms

-- Number k:  50
---- nmap =  1  -> time:  2.585346171s
---- nmap =  2  -> time:  1.242209844s
---- nmap =  3  -> time:  1.041839492s
---- nmap =  4  -> time:  914.473341ms
---- nmap =  5  -> time:  830.230587ms
---- nmap =  6  -> time:  759.262265ms
---- nmap =  7  -> time:  716.025588ms
---- nmap =  8  -> time:  677.896083ms

-- Number k:  100
---- nmap =  1  -> time:  3.475409803s
---- nmap =  2  -> time:  2.066397547s
---- nmap =  3  -> time:  1.628084223s
---- nmap =  4  -> time:  1.361674097s
---- nmap =  5  -> time:  1.237130488s
---- nmap =  6  -> time:  1.19861673s
---- nmap =  7  -> time:  1.07881213s
---- nmap =  8  -> time:  1.086531071s
```

VIII. Conclusion

The K-means algorithm for large set of data fits well with the Map-Reduce paradigm where the workload can be separated into chucks and run in parallel. Achieving good levels of efficiency is certainly the cornerstone of the system

Our system gets pretty good running times and is robust to crashes from the Mappers. The design choice to have only Reducer is a possible double-edged sword being fast for a small number of Workers but acting as a possible bottleneck on the workload. Luckily the case study allowed an efficient use of the combiners thus decreasing the workload on it.

References

[1] V. Cardellini. MapReduce and Hadoop, 2022.http://www.ce.uniroma2.it/courses/sabd2122/slides/MapReduce%26Hadoop.pdf

[2] Christian Muehlhaeuser. Clusters. https://github.com/muesli/clusters

[3] Max Bodoia https://stanford.edu/~rezab/classes/cme323/S16/projects_reports/bodoia.pdf

[4] Hadoop open-source Apache framework. https://hadoop.apache.org/