Project 4-
Experiments and Analysis
K-Means Algorithm

-Kaustubh Iyer

Initial Notes:

This document is a detail on experiences of having applied various methods to perform K-means clustering, along with an analysis on their efficiency of their implementation in Hadoop's MapReduce, varying several parameters. We start by comparing the 3 used methods: an ordinary sequential implementation, a parallelized implementation over 4 machines using MPI- Message Passing Interface, and finally, one using Hadoop's MapReduce over iterations.

The sequential implementation of K-means was fairly easy to code, mainly owing to the K-means algorithm being fairly simple in the first place. In terms of performance, however, we see a problem in that for larger datasets, it is unable to compete with the other parallel versions we have implemented, and is unable to even work for datasets larger than the machine's RAM. This can be understood, and hence it is a simple approach that can be made use of for small experiments and in understanding the algorithm's working better.

The MPI implementation was possibly the most difficult to implement of the three. Forcing the communication between the 4 machines used in the implementation, and splitting each stage of the clustering evenly among them, was rather difficult to do with a primitive interface like MPI, which did little to support the specific task as well. The MPI version did perform well for medium sized data sets, but for larger ones, the amount that had to be communicated over the medium posed the largest threat, degrading relative performance. The same could be said for smaller sets, making the sequential almost as fast as or faster for small enough sets where the tradeoff of communication for parallelism did not work in its favor.

MapReduce was very simple because all that had to be worried about was the map and the reduce function- the communication required no concern from our side. Indeed, there was only a single slight issue that had to be worked around- since MapReduce doesn't work well for iterative algorithms, we had to force the job to be run repeatedly, which needed some clever IO tricks to work around the way the job processing works. MapReduce performed well as data set size increased, especially on the larger data sets, since its overhead on communication time only grows linearly with the size of the input data set. This made the tradeoff for parallelism more to its favor even in the larger data sets.

Of the three, it could be pretty easily concluded that MapReduce is the most scalable of the three algorithms, as adding more machines or larger jobs for the MPI algorithm is difficult in implementation terms, and the sequential algorithm requires too much memory and computing power to be able to handle larger data sets.

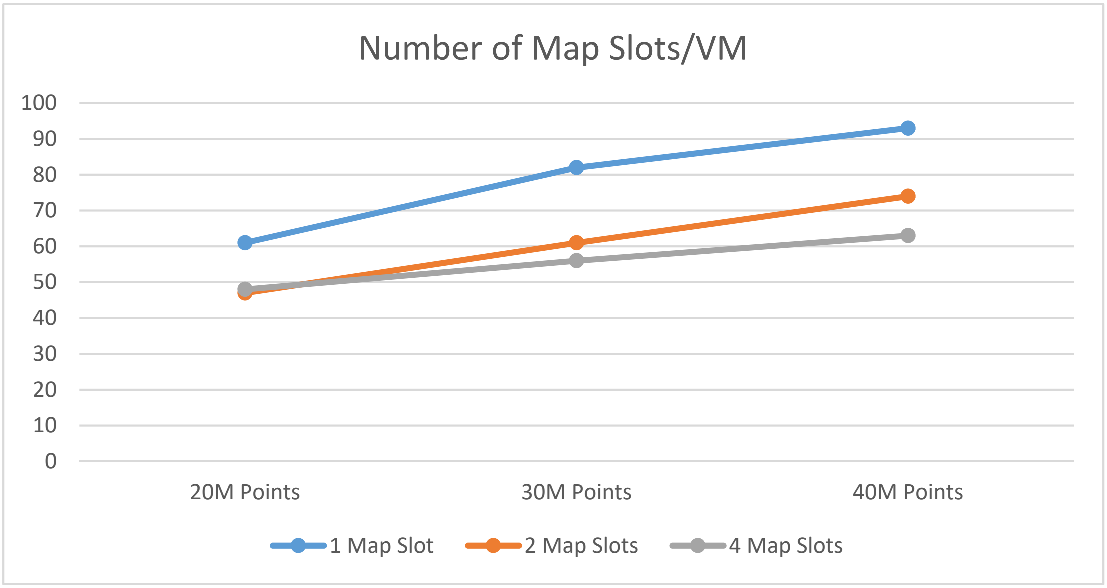
Scalability Analysis: MapReduce for K-Means:

We mainly looked at three factors and how their performance scaled with increased number of data points: The number of Map slots per machine, the size of each HDFS block, and the number of VMs actually used. In each case, we kept the other variables constant. Now we go through each of these factors, theorizing what effects on their change would be, and we see from our study the actual effect in runtime over 3 standard data sets that grow linearly off each other: one with 20 million points and 10 clusters, one with 30 million points over 15 clusters, and one with 40 million points over 20 clusters. We keep the number of iterations to 3 as a standard, and calculate the mean time per iteration for each of these cases.

I. Number of Map Slots per VM:

As we increase the number of Map slots per VM, we are increasing shuffle time while increasing parallelism as well. This comes back to the tradeoff between increased communication and more parallelism, and so for less data points, the tradeoff is less in favor for the 4 slots per VM compared to the 2 slots. But as the data set grows larger, the communication matters less as the amount of computation increases at a larger pace.

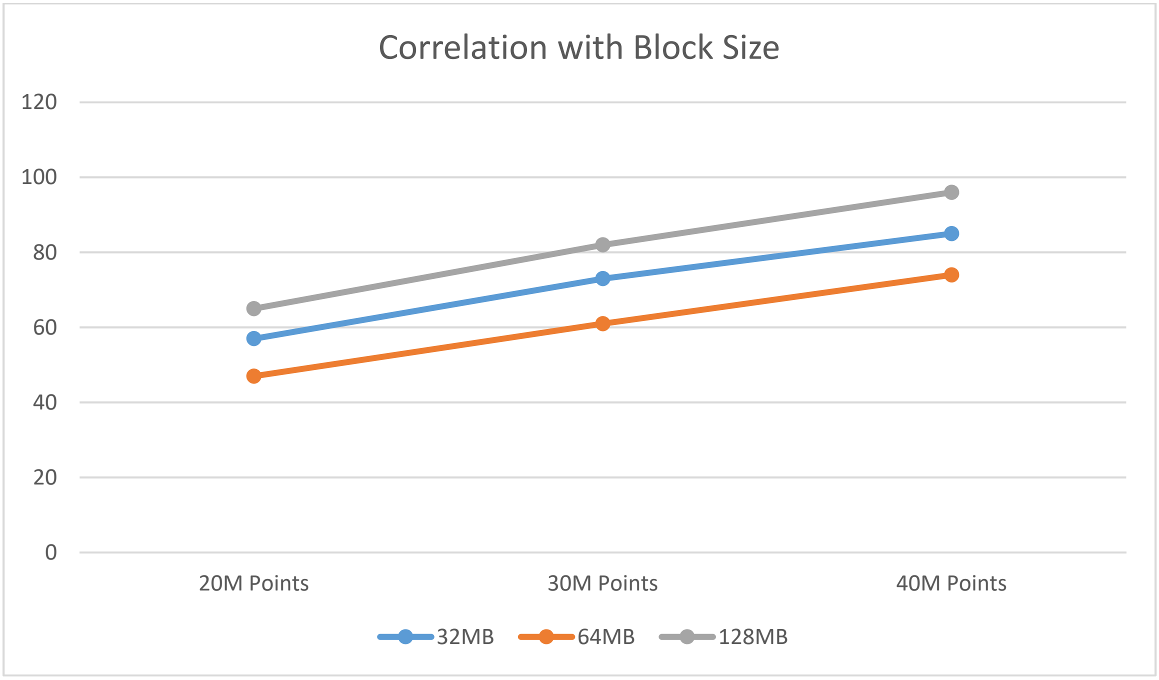
So we can conclude from this analysis that fewer map slots per VM are better for smaller data sets and the number has to be increased as the data sets used increases, to improve parallelism.



#Map slots per VM	Time taken for the average iteration (s):		
	20 million points over 10 clusters	30 million points over 15 clusters	40 million points over 20 clusters
1	61	82	94
2	47	61	74
4	48	56	62

II. HDFS Block Size:

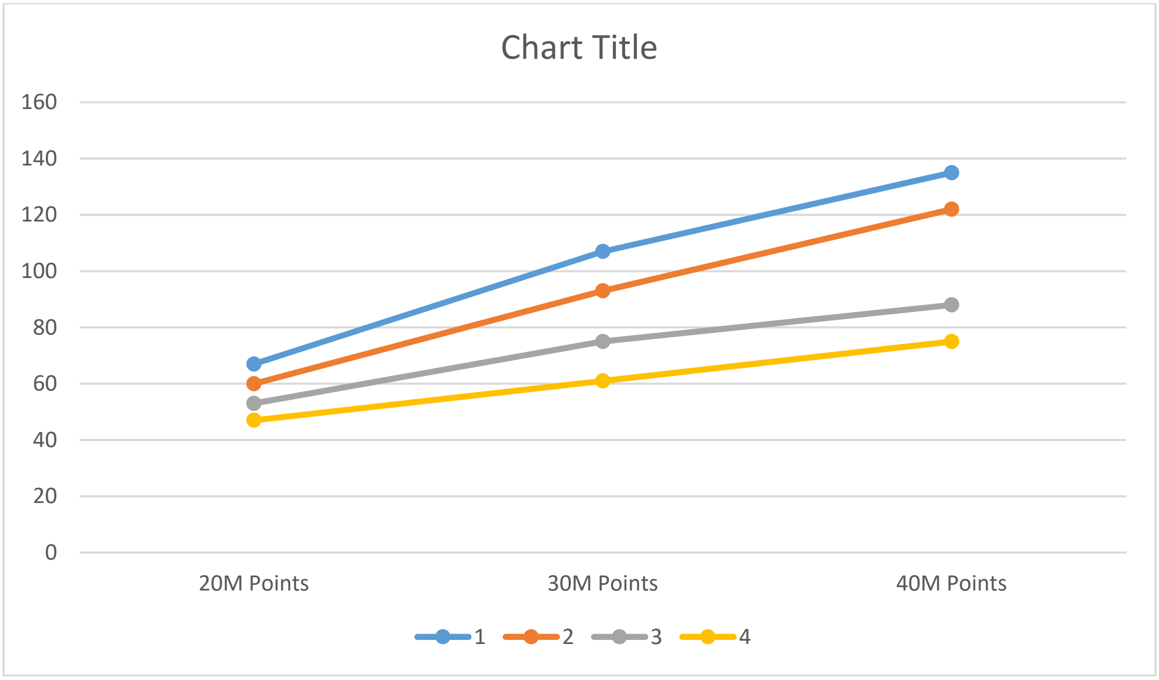
There are also tradeoffs to consider in the case of block size, with greater block sizes resulting in less tasks to be assigned in terms of "waves" of computation, and smaller block sizes resulting in more waves. It is ideal to keep the number of waves around 1 in this case, and we infer from this data that 64MB is the optimum of the 3 chunk sizes examined.



Block Size	Time taken for the average iteration (s):		
	20 million points over 10 clusters	30 million points over 15 clusters	40 million points over 20 clusters
32MB	57	73	85
64MB	47	61	74
128MB	65	82	96

III. Number of VMs

The number of VMs is a direct indicator of the degree of parallelism, and naturally, decreasing the number of VMs would increase the workload of each VM, decreasing the runtime. Having 1 VM is not very useful, as the sequential variant we wrote also works on 1 machine, and so isn't much different. The results from testing these were pretty plain to see: a direct linear correlation with increase in number of VMs.



No. of VMs used	Time taken for the average iteration (s):		
	20 million points over 10 clusters	30 million points over 15 clusters	40 million points over 20 clusters
1	67	107	135
2	60	93	122
3	53	75	88
4	47	61	75