Parallelism Comparison - K-Means Clustering in MPI vs in Spark MLlib

Binwei Xu and **Brandon Liang**
{bixu,brliang}@davidson.edu
Davidson College
Davidson, NC 28035
U.S.A.

Abstract

This project focused on the fundamental properties and differences in applications of parallelism. We applied K-Means Unsupervised clustering algorithm to position distribution breakdown data for NBA teams derived from play-by-play game data from National Basketball Association (NBA) on Apache Spark and on MPI to compare the respective run times. The result showed that MPI ran faster with relatively small sizes of clusters and relatively small numbers of iterations while Spark provided a stable run time of execution when the parameters become large.

1 Introduction

Machine Learning is one of the most popular big data analytic approaches that has become a driving force in the current technology industries. Most machine learning algorithms iteratively learn from data to extract useful insights, namely training process. Training on big data usually requires abundant memory and takes enormous amount of computation. One solution is to run machine learning algorithms in parallel on multiple machines. The benefit of parallelizing algorithms is to better allocate the memory resources and thus to raise efficiency. Considering the complexity of nowadays data structure, the volume of data available and the increasing velocity of data production, machine learning algorithms have to directly face the challenge of parallelism and try to improve on the speed of complex computations and the efficiency of inter-machine communication.

Inspired by the research of Reyes-Ortiz *et al* (Reyes-Ortiz, Oneto, and Anguita 2015) to investigate the performances of two parallelization frameworks on two supervised learning algorithms, we decided to implement an unsupervised learning algorithm, K-Means clustering, on the play-by-play game data from National Basketball Association (nba 2016) with Apache Spark MLlib (Spark 2016b) and with Open MPI. This research aims to compare the performance of the same algorithm on two different parallelization frameworks in terms of run time.

The paper is organized as follows: in Section 2, we introduce the data source, data processing and final data's structure. In Section 3, we introduce the background knowledge of K-Means clustering algorithm, Apache Spark and Open MPI. We will also explain in detail how we implement K-Means clustering on MPI. After we talk about the experiment set-up and the method to compare the two frameworks in Section 4, we justify the correctness of the clustering result in Section 5. Finally, in Section 6 and 7, we talk about the research results and our conclusion.

2 Data

In this section, we introduce the subject of our problem, followed by the source of the data and processing of the data.

Context of Problem

NBA, National Basketball Association, is the best professional basketball league around the world. Not only does it attract millions of basketball fans around the world, more and more masterminds are applying data analytics to the basketball world to understand the game from another perspective. Common analysis involve statistical analysis, team ranking, etc. In this project, we intended to cluster NBA teams by their position distribution breakdown within different margin range. Even though our primary focus of the project is to compare run time between MPI and Spark applications, the actual results of the clustering could still be insightful as it can offer another way of analyzing the game of basketball.

Raw Data and Data Processing

NBA provides a detailed play-by-play data for each regular-season game on its official website, where each play is a data point for that game. The data for each game are in the form of JSON and here is a sample link(nba 2016):

```
http://stats.nba.com/stats/playbyplayv2?
EndPeriod=10&EndRange=55800&GameID=
0021600003&RangeType=2&Season=2015-16&
SeasonType=Regular+Season&StartPeriod=1\
&StartRange=0
```

Each data point has a field "Score Margin", which is the point differential between the home and opponent team for each play of a game. For two consecutive plays, the field could generate the change of score margin on this very play.

For example, if a team leads by 3 points on the previous play and leads by 1 point on the current play, its score margin change on the current play is then -2.

After data scraping and processing, we were able to obtain the information of a single team's score margin and score margin change on each play for each game of a season. Combining such information, we were able to finalize each team's score margin and score margin change distribution breakdown for the entire season. The score margin distribution breakdown is the proportion of total plays when the team is within a specific range of score margins. For example, "-30 < x ⩽ -25" shows the percentage of plays when the team is trailing by less than 30 points and trailing by more than 25 points. Moreover, the score margin change distribution breakdown is the proportion of total plays when the team is within a specific range of score margins just before the current play and changed the score margin on the current play. For example, "-30 < x ⩽ -25 +" shows the percentage of plays when the team is trailing by less than 30 points and trailing by more than 25 points coming to this play and is able to cut down the deficit on this play.

Final Data

The score margin distribution breakdown contains 18 ranges, from "x ⩽ -40", "-40 < x ⩽ -35" to "35 < x ⩽ 40" and "x > 40". The score margin change distribution breakdown contains twice as many ranges, as each score margin range has both "+" and "-" directions. We also included two more features: proportion of total plays when the team scores and proportion of total plays when the opponent scores. This yields a total of 56 features for each team. Note that every feature is a percentage, thus carrying the same weight.

Moreover, there are 30 NBA teams in each season and we were able to scrape play-by-play data from the past 12 seasons, therefore yielding 360 team-season pairs. Note that a same team in different seasons should be treated as a different data entity.

Therefore, in our final dataset, we had 360 team-season pairs, each having 56 features of position breakdown. Figure 1 shows two sample data points (two teams), the Philadelphia 76ers and the Charlotte Bobcats in the 2004-2005 season. Throughout the entire 2004-2005 season, the Charlotte Bobcats had 7.394% of the plays when they led by more than 5 points and less than 10 points, 0.776% of the plays where they managed to score when leading by more than 5 points and less than 10 points; there were 11.578% of the plays when they managed to score and 12.398% of the plays when they let opposing teams score.

Team-Season Pair	4-76ers	4-Bobcats
5 <x <= 10	14.122	7.394
...	...	...
5 <x <= 10 +	1.585	0.776
...	...	...
Total_+_%	12.277	11.578
Total_-_%	12.322	12.398

Figure 1: For display reason, the two sample data point table is transposed.

3 Background

In this section, we introduce the background of the machine learning algorithm we implemented as well as the parallel systems we utilized.

K-Means Clustering

K-Means clustering is a prototype unsupervised learning algorithm that partitions data points into k clusters in which each point belongs to the cluster with the closest center. In this case, each point and each center are a vector with 56 features that represents the position distribution breakdown of a team in a season.

K-Means custering takes in two parameter: k, number of clusters and I, number of iterations. The algorithm starts with k randomly chosen points as centers for k clusters respectively. In order to cluster the data points, K-Means clustering calculates the Euclidean distances between each point and each center to find the closest center and assign this point to that center's cluster. This process needs to cover the entire dataset to compare all k centers in order to find the closest in each iteration. Thus, if there are n data points, there will be $O(n * k)$ many calculations. Since the centers are randomly chosen, the resulted clusters are not optimized. The algorithm calculates the mean value of each feature for all points in their corresponding clusters and uses mean values of all features to form a new center for each cluster. After all k centers are updated, the algorithm runs another iteration of the clustering over the entire dataset. Therefore, theoretically, after sufficient iterations, K-Means clustering is able to partition data points into their most suitable clusters.

Apache Spark and Spark MLlib

Apache Spark is a fast and general engine for large-scale data processing (Spark 2016a). It is an upgrade on MapReduce that provides in-memory computing on clusters with its Resilient Distributed Dataset. Spark has been implemented in common programming languages such as Python, Java and Scala, and thus provides an easier environment for development. Moreover, Spark is fault-tolerant.

MLlib is Spark's Machine Learning library. It provides an API for Machine Learning algorithms in Spark, such as regression, classification and clustering (Spark 2016b). In this project, we are interested in applying K-Means clustering, which is already implemented in Spark MLlib.

Message Passing Interface

Message passing interface(MPI) is a standardized and flexible message-passing system developed by a group of researchers since 1992. It provides a thin layer and thus allows a wide variety of parallel computing programs. The basic idea of MPI is to distribute memory resources to compute in parallel on multiple machines and communicate via collective operations. Since its nature of high performance

orientation and flexibility, MPI usually generates desirable performances within a considerably short amount of time. However, one of its disadvantages is also its flexibility because, unlike Spark, researchers need to construct specific implementations of MPI for specific projects. Furthermore, MPI is not fault tolerant as Spark.

Collective Operation

In our implementation of K-Means clustering on MPI, the entire data is divided based on the number of machines where the algorithm runs on each partition of the data in parallel. Communication happens when the algorithm finishes one iteration of clustering and all machines are ready to update the centers. The program then finds a new center for each cluster by calculating the mean of each features within this cluster across all machines. The update process is carried out only in root machine. Hence, all machines need to pass in the sums of each features of points in each cluster and the sizes of each cluster, both of which are computed locally. In the root machine, we used `gatherv` to collect the sums and sizes from all machines. Afterwards, we used `bcast` to broadcast the new global centers and proceed to next iteration of clustering using the updated centers. The key idea here it to minimize the collective operation calls in MPI implementation since shuffling is the most expensive operation in MPI and any distributed computing schemes.

4 Experiments

In the experiment, we imported data as a comma-separated values(csv) file and applied both K-Means clustering from Spark MLlib and the algorithm constructed by our own on MPI using Open MPI library. The following parameters are controllable and crucial to K-Means clustering in both systems and allow us to compare the run times: number of iterations and number of clusters. Furthermore, MPI takes in an additional parameter: number of machines. As a result, these are the three parameters we vary to compare the run times. To see how to embed parameter settings for each program in command line, please read README.txt.

Assumption

The Spark clustering program was executed on our personal computer. That computer has a 2.6-GHz Intel Core i5 processor and a 8-GB memory. The MPI clustering program was executed on a school computer in the computer lab (B110). The school computer has a 2.70-GHz Intel(R) Core(TM) i5 processor and a 16-GB memory. Due to the nature of concurrent and parallel programming, hardware setting may play a big role in resulting performance. Thus, for this project, we assumed that the computation power is consistent and equivalent between our personal computer and the school computer.

5 Correctness

Although our primary focus is the performance of run time, the correctness of the algorithm cannot be ignored. However, it is hard to evaluate the correctness of both programs due to the nature of unsupervised learning. Nevertheless, since K-Means clustering algorithm is already implemented in Spark MLlib, we only needed to worry about our own implementation of K-Means clustering in MPI. Moreover, given the subject matter in this case, we could subjectively check the correctness of our algorithm in MPI by comparing its clustering results to that of Spark. The following shows a cluster of teams from the results of both the MPI program (with 10 threads, 30 clusters and 100 iterations) and the Spark program (with 30 clusters and 100 iterations).

MPI: [13-Suns, 13-Trailblazers, 11-Lakers, 11-Rockets, 11-Thunder, 12-Heat, 9-Cavaliers, 9-Celtics, 9-Spurs, 9-Thunder, 6-Nuggets, 4-Grizzlies, 4-Heat, 10-Bulls, 10-Celtics, 10-Thunder, 13-Mavericks, 8-Rockets]

Spark: [4-Grizzlies, 6-Nuggets, 7-Hornets, 8-Spurs, 9-Celtics, 9-Lakers, 9-Trail Blazers, 10-Bulls, 11-Hawks, 11-Lakers, 11-Rockets, 12-Pacers, 13-Suns, 13-Trailblazers])

There are 8 overlaps between these two clusters resulted from MPI and Spark, which means that both MPI and Spark recommend that these 8 teams should belong to the same group of team due to their similarities in features. In an expected cluster length of 12 teams per cluster (360 team-season pairs into 30 clusters), having 8 teams for one cluster agreed by both MPI and Spark shows that the K-Means clustering algorithm worked quite well in MPI.

6 Results

After checking correctness, we now turn to our main focus. We tested K-Means clustering on Spark and on MPI with various parameters for run times and compared the results.

MPI with Different Numbers of Machines

Since MPI runs on multi-machine processes, it has one more parameter than the Spark program: number of machines. In order to show the relationship between total execution time and number of machines, we first set the number of clusters to be 30 and number of iterations to be 100 for the MPI clustering program and tested it with different numbers of machines. The following table shows the resulting execution time for each number of machines given.

MPI: with 30 Clusters in 100 Iterations	
2	0.128
3	0.141
4	0.172
5	0.243
6	0.292
7	0.327
8	0.365
9	0.475
10	0.504

Figure 2: Total Execution Time (s) of the MPI program with 30 Clusters in 100 Iterations with Different Number of Machines.

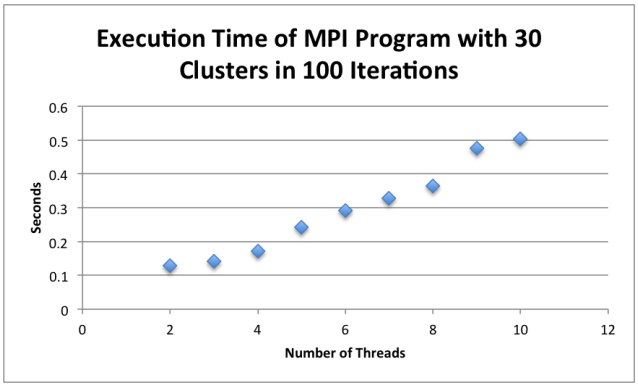

Figure 3: Execution Time of MPI Program with 30 Clusters in 100 Iterations with Different Number of Machines.

As we can see from the table 2 and figure 3, The total execution time of the MPI program with the same settings increased as the number of machines increased; moreover, it also indicates a roughly linear growth rate of total execution time on number of machines. This means that as the program got more machines to distribute the work to, the more time it took the program to finish the job. This result was opposite to what we expected, as we expected the program to run faster with more machines. Nonetheless, this doesn't mean that all MPI programs take longer to run with more machines. What it implies is that while more machines partition data into a rising number of decreasing-sized local blocks, they also raises the cost for collective operations to communicate between machines, as such operations are the most expensive ones. Keep in mind that MPI's thin layer provides large flexibility for implementations and various implementations may vary in how they invoke collective operations. For this project, our implementation of K-Means clustering appeared to be a very expensive one in terms of collective operations, in that more machines cost more time. This could be one thing to improve as a future work.

Moreover, on the other hand, this result may have also caused by the relatively small ratio of data size and feature size (360 / 56). In other words, the data may not have been massive enough for the advantage of MPI's distribution to stand out.

Number of Clusters and Iterations - MPI vs Spark

After examining the effect of machines, we then fixed the number of machines to be 10 for the MPI program and ran both the MPI and the Spark clustering programs with numbers of clusters varying from 15, 30, 60, 120 and numbers of iterations varying from 100, 500, 1000, 2000. That yields a total of 16 sets of parameter combination (# of Clusters, # of Iterations) for each program. For each combination of parameters, the code returned a total execution time in seconds and we did 5 trials runs for each to get an average run time. Then, we were able to compare the performances between MPI and Spark with the same control variables and also compare how each program's performance differs with different sets of parameters.

MPI

Table 4 shows the total execution times for the MPI program in 16 different parameter settings.

MPI: Total Execution Time (Seconds)				
#Clusters / #Iterations	100	500	1000	2000
15	0.34	0.659	0.915	1.42
30	0.457	0.763	1.13	1.83
60	0.429	0.858	1.37	2.4
120	0.497	1.19	2.04	3.72

Figure 4: Total Execution Time (s) of the MPI program with different combinations of Number of Clusters and Number of Iterations.

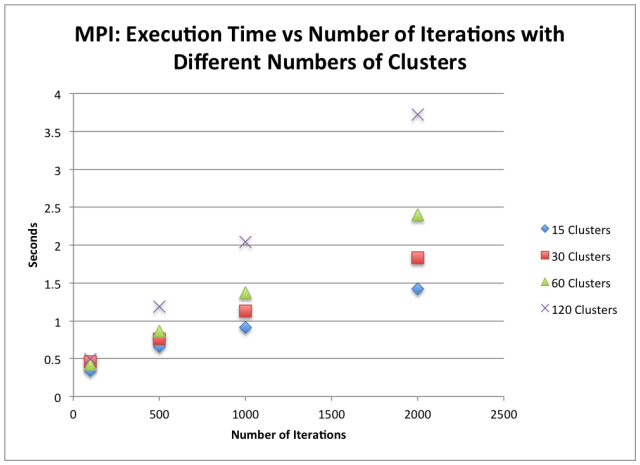

Figure 5: Execution Time of MPI Program vs Number of Iterations with Different Numbers of Clusters.

From table 4, figure 5 compares total execution time versus number of iterations for different numbers of clusters and figure 6 compares total execution time versus number of clusters for different numbers of iterations. We can see that, with the same number of clusters given, total execution time increased with more iterations; with the same number of iterations given, total execution time increased with more clusters. Moreover, the time complexity for the MPI program is linear over number of iterations and over number of clusters. It is expected that the program is O(#Iterations) since the iterative process in MPI could only be carried out sequentially; however, the fact that the program is also O(#Clusters) implies more. It doesn't automatically degrade the performance of MPI but instead shows how flexible MPI is. The reason that our MPI program is O(#Clusters) is at the step of updating centers after each iteration. As mentioned earlier, the algorithm requires all machines to pass in the sums of each feature in each cluster; each passage of information here is carried out by MPI's collective operation gatherv. Although one `gatherv` call can allow gathering from all machines to the root machine, the program still needed to call it #Cluster times for all required clusters. Additionally, since time complexity for the MPI program is both O(#Iterations)

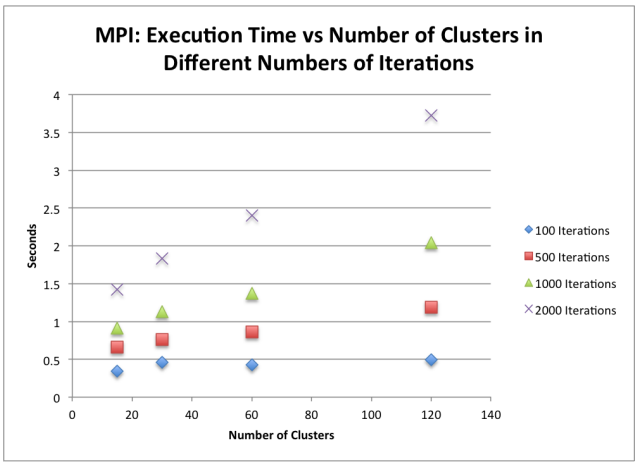

Figure 6: Execution Time of MPI Program vs Number of Clusters in Different Numbers of Iterations.

and O(#Clusters), it's fair to argue that the MPI program has an overall time complexity of O(#Iterations $*$ #Cluster) given a fixed number of machines. In other words, the MPI clustering program works extremely well in terms of time for relatively small numbers of clusters in relatively small numbers of iterations; its performance in time degrades proportionally as the number of clusters and the number of iterations increase.

Spark

Table 7 shows the total execution times for the Spark program in 16 different parameter settings.

Spark: Total Execution Time (Seconds)				
#Clusters / #Iterations	100	500	1000	2000
15	3.495	3.846	3.426	3.484
30	3.383	3.105	3.408	3.338
60	2.667	2.758	2.781	2.675
120	2.92	2.817	2.818	2.804

Figure 7: Total Execution Time (s) of the Spark program with different combinations of Number of Clusters and Number of Iterations.

From the table 7, figure 8 compares total execution time versus number of iterations for different numbers of clusters. Figure 9 compares total execution time versus number of clusters for different numbers of iterations. We can see that, with the same number of clusters, the run times were quite stable, independent from number of iterations; this means that the K-Means clustering implementation in Spark MLlib probably has its own optimization setting to parallelize the iterative computations.

On the other hand, with the same number of iterations, the run times decreased as the number of clusters increased from 15 to 30 and from 30 to 60, reached minimums when the program was given 60 as the number of cluster and then

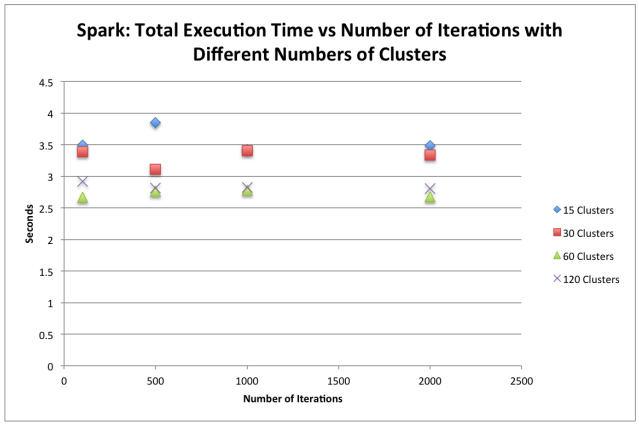

Figure 8: Execution Time of Spark Program vs Number of Iterations with Different Numbers of Clusters.

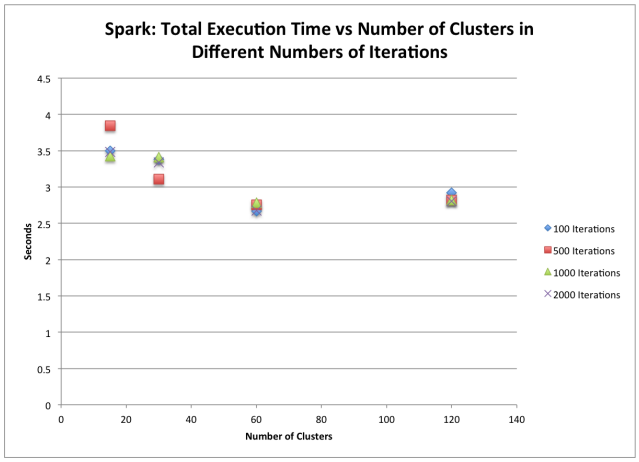

Figure 9: Execution Time of Spark Program vs Number of Clusters in Different Numbers of Iterations.

increased as the number of clusters went up to 120. This means that for this set of data, the Spark clustering program probably reaches an optimal performance when the number of cluster is between 30 and 90, preferably close to 60.

Also note that while the MPI program generally ran faster than the Spark program most of the time according to our data, that doesn't tell the big picture. Refer to the bottom-right corner of table 4 and table 7. It took the MPI program 2.4 seconds for 60 clusters in 2000 iterations and 2.04 seconds for 120 clusters in 1000 iterations; however, the time suddenly jumped to 3.72 for 120 clusters in 2000 iterations. On the other hand, the run time is stable for the Spark program in these 3 settings, at about 2.6 to 2.8 seconds. Thus, there must exist a threshold of settings (number of clusters, number of iterations) where the performances of the MPI program and the Spark program intersect. From the existing experiment data, it seems that with relatively small parameter values, MPI is faster; with relatively large parameter values, Spark stays stable and thus outperforms MPI.

(Clusters, Iterations)	MPI	Spark
(60,2000)	2.4	2.675
(120,1000)	2.04	2.818
(120,2000)	3.72	2.804

Figure 10: Crossing points of MPI and Spark programs run time

7 Conclusions

In conclusion, for K-Means clustering, the MPI program runs slower with more machines because it requires more collective operation calls per computation and thus slow down the execution. In addition, MPI works better with small number of clusters and small number of iterations as parameters. Spark run time is stable regardless of the number of iteration and Spark outperforms MPI with large number of clusters. Thus, MPI is great for relatively small cluster size or small number of iterations while Spark is suitable for computation-heavy parameter settings.

As mentioned earlier, one idea for further work is to find out an algorithm to update sums of features for each cluster after each iteration with a better time complexity than that of our current implementation, which is O(#Clusters). Another idea is to explore the implementation of K-Means clustering in Spark MLlib to find out why its performance can be independent from number of iterations.

8 Acknowledgements

We would like to thank Dr. Hammurabi Mendes for his great support and guidance during the process of preparing data and developing algorithm on MPI, as well as his sample codes from class. We would also like to thank Reyes-Ortiz *et al* (Reyes-Ortiz, Oneto, and Anguita 2015) for the inspiration of this project idea.

References

2016. Nba stats. `http://stats.nba.com`. Retrieved on Nov. 5, 2016.

Reyes-Ortiz, J. L.; Oneto, L.; and Anguita, D. 2015. *Big Data Analytics in the Cloud: Spark on Hadoop vs MPI/OpenMP on Beowulf*. Elsevier B.V.

Spark. 2016a. Apache spark. `http://spark.apache.org`. Retrieved on Dec. 8, 2016.

Spark. 2016b. Spark mllib clustering. `http://spark.apache.org/docs/latest/ml-clustering.html#example`. Retrieved on Nov. 20, 2016.