A Review On Suitable Execution Environment For Executing Parallel Workloads With Mapreduce And Spark

Suvam Jain, L. Shyamala

Abstract: The high volume of networked computers, workstations, LANs has prompted users to move from a simple end user computing to a complex distributed computing environment. This transition is not just networking the computers, but also involves improvement of performance when used to solve various computational problems. Sequential and parallel execution has its significance on enhancing the performance of a system. Hence when a computational intensive problem solved by a different approach and configuration, it may lead to either performance enhancement or degradation. Therefore, to improve the efficiency, an appropriate method needs to be adopted. This paper discusses to find a suitable method for a problem by examining several approaches and configurations for its execution. Hadoop MapReduce and Apache Spark open source frameworks are used to conduct our experiments and results are analysed.

Index Terms: Apache Spark Clusters, Big Data, Hadoop Map Reduce, HPC Metrics, K-Means Clustering, Linear Regression, RDD

1. INTRODUCTION

A new model of cluster computing has become widely popular, in which data-parallel computations are executed on clusters of unreliable machines by systems that automatically provide locality-aware scheduling, fault tolerance [7], and load balancing. MapReduce [3] has pioneered this model. These systems achieve their scalability and fault tolerance by providing a programming model where the user creates acyclic data flow graphs to pass input data through a set of operators. This allows the system to manage scheduling and to react to faults without user intervention. In this paper, one such class of an application is considered which reuses a set of data across multiple parallel operations. This includes two use cases, one where Hadoop users report that MapReduce is deficient [12] and another where spark doesn't perform well [8]. The paper presents a new cluster computing framework called Spark [10], which supports applications with working sets while providing similar scalability and fault tolerance [7] properties to MapReduce.

1.1 Iterative jobs

Many common machine learning algorithms apply a function repeatedly to the same dataset to optimize a parameter (e.g., through gradient descent). While each iteration can be expressed as a MapReduce [3] job, each job must reload the data from disk, incurring a significant performance penalty. Thus, Hadoop's Map Reduce technique makes the application much sophisticated and doesn't perform well whereas, Spark uses resilient distributed dataset (RDD) [6] technique to handle with such issues. In this paper, k-Means clustering algorithm is used to analyze the performance of both the frameworks i.e. MapReduce [3] and RDD [6] while executing iterative jobs. Chu et al. [5] have shown that iterated MapReduce can also be used to implement other common learning algorithms.

- *Suvam Jain, UG Student, Department of Computer Science and Engineering, Vellore Institute of Technology, Chennai, PH-+917358691791. E-mail: jainsuvam1@gmail.com*
- *L. Shyamala, Associate Professor, Department of Computer Science and Engineering, Vellore Institute of Technology, Chennai, PH-+919884655300. E-mail: shyamalal@vit.ac.in*

1.2 Regression analytics

Hadoop [2] is often used to train a Linear Regression model on large datasets for the prediction of values given some input data-points in future. The fitting of a regression line can't work independently on each node. Instead they require sharing of previous experience gained by other nodes while working parallely in cluster, to fit a regression model. Hence, MapReduce technique outperformed RDD [6] here, as in RDD the latency incurred due to sharing of knowledge gained by each node to all other nodes after each step is too high and can't be used in situations when results are required immediately after feeding the input dataset to the cluster. The main abstraction in Spark is that of a resilient distributed dataset (RDD) [6], which represents a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Users can explicitly cache [9] an RDD in memory across machines and reuse it in multiple MapReduce-like parallel operations.Spark is implemented in Scala [1], a statistically typed high-level programming language for the Java VM. In addition, Spark can be used interactively from a modified version of the Scala interpreter, which allows the user to define RDDs, functions, variables and classes and use them in parallel operations on a cluster. Spark is the first system to allow an efficient, general-purpose programming language to be used interactively to process large datasets on a cluster and also has the advantage of processing an online streaming dataset unlike Hadoop [2] MapReduce framework where only a batch dataset can be processed. This paper is organized as follows. Section 2 describes Methodology used to implement spark cluster and advantage of RDDs in Spark over MapReduce. Section 3 include a short description about the dataset used for training over distributed cluster. Section 4 shows some example jobs executed for contrasting the performance of two frameworks considering various constraints. Section 5 describes our implementation model used over the two-major machine learning algorithms as discussed previously, including our integration into Scala and its interpreter. Section 6 discusses the analysis and results obtained and Section 7 includes the conclusion.

2 METHODOLOGY

In order to use spark for clustering and distributed computing, one needs to write a driver program that implements the high-

level control flow of the applications executed and launches its various jobs/operations in parallel over the nodes. Spark has two main abstractions to support the parallel programming: resilient distributed datasets and parallel operations on these datasets (invoked by passing a function to apply on a dataset). Additionally, Spark supports two restricted types of shared variables that can be used in functions running on the cluster: Broadcast and Accumulator variables. Programmers can invoke operations like map, filter and reduce by passing closures to Spark.

2.1 RDDs

A resilient distributed dataset (RDD) [6] is nothing but a read-only collection of objects in spark which are partitioned across a set of machines that can be rebuilt if a partition is lost. Its elements need not exist in physical storage, instead, a simple handle to an RDD contains enough information to compute the RDD starting from data in reliable storage which signifies that the RDD [6] nodes can always be reconstructed in case of any failure. In Spark, each RDD is represented by a Scala [1] object. Spark lets programmers construct RDDs in four different ways:

1. From a file either passed as argument to spark context object or in a shared file system, such as the Hadoop Distributed File System (HDFS) [2].
2. Through transformation of an existing RDD object expressed using flatMap and filter.
3. By parallelizing a code snippet that should be executed in parallel for better utilization of resources. This can be done by slicing and broadcasting it to multiple nodes present in the cluster. Spark object provides a function called parallelize to obtain this.
4. Persistence of an existing RDD can be changed by using cache module of spark object.

2.2 Parallel Operations

Spark supports various mechanisms to execute operations in parallel fashion on the cluster using RDD such as reduce, collect and foreach. Also, it should be noted that currently Spark doesn't support grouped reduce operations as in MapReduce, instead reduce results are only collected at one process i.e. the driver process. This effect diminishes the efficiency gained in small datasets due to overhead but in larger datasets (size in TBs) the effect is almost negligible and hence efficiency is much better compared to MapReduce technique.

3 DATASET

For this paper, "New York City Taxi Fare Prediction" dataset is used. The dataset comprised of 6 features and 1 target variable namely:

1. pickup_datetime - timestamp value indicating when the taxi ride started.
2. pickup_longitude - float for longitude coordinate of where the taxi ride started.
3. pickup_latitude - float for latitude coordinate of where the taxi ride started.
4. dropoff_longitude - float for longitude coordinate of where the taxi ride ended.
5. dropoff_latitude - float for latitude coordinate of where the taxi ride ended.
6. passenger_count - integer indicating the number of passengers in the taxi ride.

A key given in each row is a unique string to identify each row in both the training and test sets Also, a target variable fare_amount is provided in the training dataset. It is a float type, amount of the cost of taxi ride (in dollar). This is what predicted in the test set. This dataset is suitable for our analysis as it contains approximately 60M rows in total hence, can be considered as an example of Big Data and provide us the opportunity to analyze the performance in a better way for both the MapReduce and Spark frameworks.

4 EXAMPLES

Different operations have been implemented on this dataset to measure the efficiency using two ML algorithms provided by the inbuilt Spark MLlib in order to contrast the major differences in data processing by the two frameworks discussed here. Both of the algorithms are executed on MapReduce and Spark separately and their performance is analyzed based on various HPC metrics that are discussed in section 6.

4.1 Reading a Dataset

This operation is used to find the time taken to read the dataset by different configurations. A dataset file has been hosted using local file system for spark and using hdfs on Hadoop framework. The time required to read the dataset from the csv file and store it into the main memory is noted. The results obtained can give an idea about which framework is good for input handling while processing a large dataset.

4.2 Datatype Conversion

Since, the dataset contains a date-time of travel in a GMT format. So, in order to calculate the distance and fit the model requires to parse the relevant information such as date and time of travel separately which needs string formatting. This operation provides us with the opportunity to analyze the time required for operations like datatype conversion in both the frameworks.

4.3 Haversine Distance Calculation

Also, the distance travelled by each passenger is not given directly into the dataset. Instead, the latitudes and longitudes of starting and dropping location is provided as features. So, before training calculate the distance in standard units like meters or kilo-meters. Equation (1) shows that the Haversine formula can be used to obtain the distance from latitudes and longitudes. This gives the scope to analyze the calculation intensive operation and the time required to do so in each of the two frameworks.

$$d = 2r\arcsin\left(\sqrt{sin^2\left(\frac{\varphi_2-\varphi_1}{2}\right) + \cos(\varphi_1)\cos(\varphi_2)sin^2\left(\frac{\lambda_2-\lambda_1}{2}\right)}\right) \quad (1)$$

where

d	distance between the two points;
r	radius of the sphere;
φ1, φ2	latitude of point 1 and latitude of point 2 (in radians);
λ1, λ2	longitude of point 1 and longitude of point 2 (in radians).

4.4 Linear Regression

First, the taxi fare dataset is trained with Linear Regression model. It fits the model on the training set by using the

experience gained by all neurons and finally fit a regression line. The coefficients and intercepts of line can be accessed by the param object of LR model.

In Hadoop's MapReduce technique, sklearn library of python is used to implement this Algorithm.

```
clf = LinearRegression(c=0.001, n_iter=10)
clf.fit(X_train)
y_pred = clf.predict(X_test)
accuracy_score(y_test, y_pred)
```

The same Algorithm can be implemented in Spark using Java Spark object as given below.

```
SparkSession spark = SparkSession.builder()
        .appName("Spark").getOrCreate();
LinearRegression lr = new LinearRegression()
      .setLabelCol("fare_amount");
LinearRegressionModel lrm = lr.fit(data);
coeff = lrm.coefficients();
incept = lrm.intercept();
```

4.5 k-Means Clustering

The following program implements k-means clustering, an unsupervised iterative classification algorithm that attempts to group the data points into a cluster based on some similar characteristics. The algorithm first plots the data points into higher dimensional space and then calculates the distance of each point from all other points and finally, based on the threshold value it groups the points into a cluster. All these steps require huge amount of calculations.

In MapReduce technique using sklearn's library of python this Algorithm is implemented as follows:

```
km = kMeans(n_clusters=2, random_state=0)
km.fit(X_train)
y_pred = km.predict(X_test)
accuracy_score(y_test, y_pred)
labels = km.labels_
centers = km.cluster_centers_
```

The same Algorithm can be implemented in Spark using Java Spark object as given below.

```
SparkSession spark = SparkSession.builder()
      .appName("Spark").getOrCreate();
KMeans kmeans = new KMeans().setK(10)
      .setSeed(1L);
KMeansModel model = kmeans.fit(data);
Dataset<Row> predictions =
      model.transform(data);
ClusteringEvaluator evaluator =
      new ClusteringEvaluator();
double silhouette = evaluator.
                        evaluate(predictions);
```

In k-Means model silhouette score and elbow plot is analyzed to find the optimal number of clusters for the given dataset to reduce overfitting.

Finally, after prediction the total execution time is printed on the console for performance analysis.

5 IMPLEMENTATION

The factors like Time, Memory, Scalability and Robustness are considered as the primary measures in this paper. It's known that Spark is built on top of Mesos, a "cluster operating system" that lets multiple parallel applications share a cluster in a fine-grained manner and provides an API for applications to launch tasks on a cluster. This allows Spark to run alongside existing cluster computing frameworks, such as Mesos ports of Hadoop and MPI, and share data with them which also reduces the programming effort that had to go into Spark. The core of Spark is the implementation of RDDs [6] as discussed in Section 2.1. For implementation, bootstrap the dataset using k-fold techniques into various train and test sets for better optimization of the model. The results obtained from different sets of data of various sizes viz. 110 MB (normal data size), 55MB (half data size) and 220 MB (Double data size) are analysed. This gives the trend of how the latency occurred due to communication overhead can affect the execution of models for various sizes of big data. Another approach that followed in this paper is to execute the same application in the cluster over various numbers of cores ranging from 1 to 7 to see how well the resources are distributed and utilized under tremendous work load. All these executions gave an analysis report containing different high-performance metrics like Garbage Collection Time (GC), Deserialization Time, Scheduler Delay, Result Serialization Time etc. as shown in the next Section using which the performance can be interpreted.

6 ANALYSIS AND RESULTS

After the execution of all the 5 major examples as discussed in Section 4, the results obtained are shown in Table 1. The table contains various parameters like number of cores used to execute the application in spark standalone and cluster nodes along with the Hadoop [2] MapReduce implementation. For better understanding of the overhead generated during distributed operation, the same application has also been executed on 3 cores once on cluster mode and then on client/master mode. The results obtained can clearly contrast the time difference between two deploy modes which is due to the fact that when executed in client mode there is no communication overhead as the application is executed on multiple cores of a single machine hence no need of broadcasting the result. Whereas, when executed over distributed nodes the results obtained should be collected at the driver process which has started the daemon. The same results can also be verified from the results which shows that scheduler delay in case of cluster mode is 28ms whereas in the client mode is only 3ms. Other jobs like reading of Input file is very fast in MapReduce as it reads the whole dataset at once and host it in a single hdfs file system whereas the spark implementation took minimum of 13s which is 23% slower than the python. For a job like Datatype Conversion, python MapReduce implementation again outperformed the Spark as in python all rows of a column are treated as a series object than can be directly converted to other format without any requirement of fetching rows one by one parsing the date and time of travel which is the case in Spark. But while calculating the distance for each row, Spark's JAVA implementation performed excellently as it is a computation intensive operation and require efficient distributed computing to obtain results in a lesser time which can easily be implemented using the RDD's approach [6]. Similarly, for k-means clustering it is observed that Spark is handling the calculation very fast in a well distributed manner as seen in Fig. 1. This is due to the fact that, spark nodes first calculate the distances obtained from calculations and then after grouping it finally maps all the data to the driver process whereas in MapReduce [3], after

each calculation done all the nodes in cluster needs to be alerted by broadcasting which generates huge overhead due to iterative communication. Finally, for the Linear Regression model it is noticed that, Spark doesn't perform well as it belongs to a class of algorithm that need not to be parallelized. Hence, when executed in cluster deploy mode delay in response has been noticed. The DAG of the linear Regression shows the above discussed points in Fig. 2. For the various jobs executed, spark also provides the HPC metrics in a tabular format for better analysis of tasks.

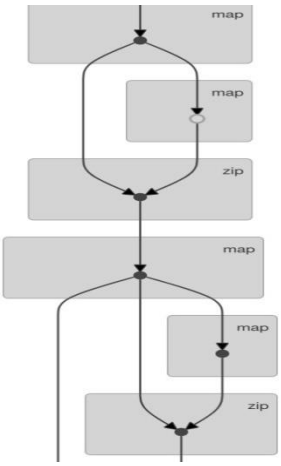

Fig. 1. *k-Means Clustering DAG*

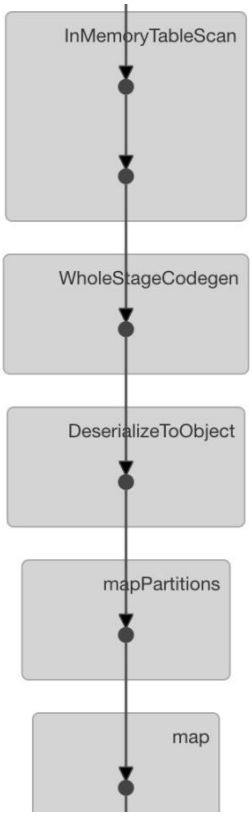

Fig. 2*. Linear Regression DAG*

TABLE 1
ANALYTICAL RESULTS OBTAINED FOR VARIOUS JOBS BASED ON DIFFERENT CONSTARINTS

Jobs	Python Implementation	Spark (JAVA) Implementation	1-Core Cluster	3-Core Cluster	3-Core at master	5-Core Cluster	7-Core Cluster
Reading File Input of 850000 data points	3.45s	29.46s	24.54s	23.85s	13.41s	27.63s	25.31s
Data Type Conversion Parsing date from string	0.73s	7.87s	4.35s	3.63s	2.76s	2.68s	2.46s
Distance Calculation Calculating Haversine distance for each row	810.91s	6.61s	3.74s	3.2s	2.51s	2.35s	1.82s
Linear Regression Applying Linear Processing over data with the default parameters	0.43s	12.18s	16.07s	11.90s	10.74s	11.87s	11.15s
K-Means Clustering Fitting K-	300.45s	18.58s	19.7s	10.86s	11.21s	11.35s	19.76s

Means Model over the data							
Total	1115.96s	74.70s	68.4s	53.44s	40.63s	55.88s	60.5s

7 CONCLUSIONS

From the overall analysis, it can be said that Spark outperformed Hadoop by 10x in iterative machine learning workloads where clustering is done based on huge calculations and on the other hand, MapReduce technique is the best choice to obtain immediate results for regression analysis cases. The difference in core architecture of both the frameworks is a major reason behind this high-speed computing. Although, the three simple abstractions of Spark viz. resilient distributed datasets [6] (RDDs), and two restricted types of shared variables: broadcast variables and accumulators are limited to a certain extent, it is found that they are powerful enough to express several applications that deals with challenges for existing cluster computing frameworks, including iterative and interactive computations. Furthermore, Spark works well for fast data processing, and provides near real time processing with the capability of performing tasks on batch as well as streaming data. Spark's MLlib makes it very easy for the programmers to implement the ML algorithms at ease. It also achieves fault tolerance through checkpointing [7] and enhances system performance by caching [9] data in memory for further iteration which Hadoop lacks, hence MapReduce works at lower processing speed. Due to its speed and low-latency Spark also outperforms MapReduce in joining of datasets. All these features make it a perfect data analytics engine for a Data scientist. Whereas, the MapReduce framework is a complex and high latency computing framework with certain drawbacks [12] like slower processing speed due to frequent reads/writes on disk, absence of job scheduler, no support for streaming data, and requirement of external ML tools which makes it a basic data processing engine. Thus, it can be used for use cases such as linear processing of huge datasets where parallelization is not required, to obtain economical results if no immediate results are expected by the firm i.e. speed of processing is not a critical issue. Hadoop has also designed an extended version of its framework called HaLoop [4] to improve the performance of iterative MapReduce programs which can integrate to RDDs. Applications expressed using HaLoop's programming model uses the output of the reduce phases of earlier iterations as the input to the map phase of the next iteration.

ACKNOWLEDGMENT

We would like to acknowledge the anonymous reviewers and all other contributors to the open source SparkML project. Authors would also like to thank Department of Computer science and Engineering, Vellore Institute of Technology, Chennai, for their support, guidance and encouragement for writing this paper.

REFERENCES

[1] Scala programming language, available at http://www.scala-lang.org.
[2] Apache Hadoop, available at http://hadoop.apache.org.
[3] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," Commun. ACM, 51(1):107–113, 2008.
[4] Y. Bu, B. Howe, M. Balazinska and M.D. Ernst, "HaLoop: Efficient Iterative Data Processing on Large Clusters," Proc. VLDB Endow., 3:285–296, September 2010.
[5] C.T. Chu, S.K. Kim, Y.A. Lin, Y. Yu, G.R. Bradski, A.Y. Ng and K. Olukotun, "Map-reduce for machine learning on multicore," in NIPS '06, pages 281–288. MIT Press, 2006.
[6] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker and I. Stoica, "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," in EECS Berkeley, 2011.
[7] A.M. Kermarrec, G. Cabillic, A. Gefflaut, C. Morin and I. Puaut, "A Recoverable Distributed Shared Memory Integrating Coherence and Recoverability," in FTCS '95, 1995.
[8] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazie`res, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman, "The case for RAMClouds: Scalable High-performance Storage Entirely in DRAM," SIGOPS Oper. Syst. Rev., 43:92–105, Jan 2010.
[9] P. K. Gunda, L. Ravindranath, C.A. Thekkath, Y. Yu, and L. Zhuang, "Nectar: Automatic Management of Data and Computation in Datacenters," in OSDI '10, 2010.
[10] How to Create Spark Java Application and Submit it to Spark Cluster, available at http://www.techburps.com/misc/create-spark-java-application-and-submit-it-to-spark-cluster/.
[11] Spark Research Tutorials, https://spark.apache.org/research.html.
[12] J.F. Weets, M.K. Kakhani and A. Kumar, "Limitations and Challenges of HDFS and MapReduce," in IEEE-ICGCIoT'16, 2016.