On studying the performance of Hadoop Map Reduce vs MPI for Aggregation Operations: A Big Data Challenge

I. Introduction

IT Revolution has been chiefly responsible for generating tremendous amount of data, often referred to as Big Data. Parallel and distributed computing systems have largely been utilized for solving problem involving big data. Of various distributed & parallel computing systems like vector processing systems, co-processors (e.g. GPU's), and others; Map Reduce and MPI (Message Passing Interface) have found increased attention in recent times. MPI has been a de facto standard of parallel programming for decades [11], while Map Reduce is of comparatively recent development. The major difference between both the platforms is that Map Reduce uses HTTP/RPC protocols for communication which are clearly slower than the MPI's communication (which provides both point-to-point and collective communication). On the flip side, programming MPI is clearly more difficult and requires lots of skills (in programming, networking, domain knowledge, and others) than Map Reduce, which much simpler because programmer does not have to handle networking problems. Both of the paradigms of programming have been extensively utilized.

For the proposed work, minimum and maximum fare values by month & year would be calculated using Map Reduce and minimum fare by month & year would be calculated using MPI for taxi trip records data collected by New York's Taxi and Limousine commission. The choice of maximum and minimum values by month & year was based on the fact that aggregation tasks are required by both the scientific and business community since they provide summaries of the data before proceeding with any type of analysis or visualization. The challenge here is to optimize the performance of MPI and Map Reduce for aggregation tasks.

In the following section a review of past works has been presented. Subsequently, the problem definition has been restated. Section 4 introduces the dataset used for the project. Finally, a discussion and conclusion of the proposed work has been presented.

II. Literature Review

It is a well-established fact that Map Reduce finds application in the business community while MPI has been used more by research community. There many available implementations of both the technologies to choose from. For instance, available MPI implementations are Open MPI, Boost MPI, MPICH, and others, while available Map Reduce implementations are Apache Hadoop, MongoDB (the database has built-in map reduce framework.), Clusterpoint XML (another database having built-in map reduce support), and others.

The choice of implementation for MPI was MPICH since it was available on SHARCNET consortium and does not require any setup. For Map Reduce, MongoDB itself recommends using Apache Hadoop Map Reduce if Apache Hadoop is used as a sink or to use its new aggregation framework otherwise and website for Clusterpoint is not operational any longer. Thus, leaving us with only one solution for Map Reduce, i.e. Apache Hadoop.

III. Problem Definition

The proposed work focuses on optimizing the aggregate queries for Map Reduce Framework and MPI. Both of these models solve problems in different approaches. However, it might be noted that MPI is a paradigm and may be used to create framework like Map reduce like the Map Reduce framework by Sandia Labs [1]. There are two technologies for which optimization needs to be carried out, as follows:

1) Message Passing Interface: Aggregation task for the given problem may be carried out in one of the following two ways both of which may be expressed as a famous Drinking Philosophers problem [12]:

   Approach 1. The Master node reads a data chunk and distributes it to the slave nodes for processing. The slave than returns the processed data. The merit of the technique is that it uses less total memory than a sequential process where master reads all the data before distributing. This technique was demonstrated in SHARCNET [13]. The following technique is more suitable in presence of a very large size of a file. Also, this program can be worked around to take advantage of replication so that processors access data file from more than one disk and decrease the I/O waiting time.

   Approach 2. The master node sends a list of file names for slaves to process. Each slave processes the files provided to the same and returns the results. This approach works only if data is to be read from multiple files and if the MPI infrastructure supports the operation of reading multiple files at the same time. The approach saves a lot of time by minimizing the time to transfer data. The only difference in regards with Approach 1, is that instead of a chunk of data we share different files. This is more suitable also in files are on separate disks. Either, it can take advantage of sharding explicitly.

Now as discussed below in Section 4, our dataset has multiple files and size of the files are almost same. Thus, for the given problem it is possible to use Approach 2. However, it has yet to be established if the MPI implementation at Graham or ORCA systems at SHARCNET provides the facility to access multiple files at the same time. If it is not possible to access the same file multiple times, than we would shift to Approach 1.

2) Map Reduce: Map Reduce usually works the best when you wish to implement one-pass algorithms. For the given problem, we have to find minimum and maximum value which satisfy this property. Hence, no programming optimization is required for the Map Reduce code. However, there are several factors such as amount of data chuck read for each task (AKA input split), size of the data block, and the choice of intermediate compression format; that can impact the processing time of Map Reduce tasks. Thus, we have following three parameters for the Apache Hadoop Map Reduce implementation to optimize:
    1) Optimal size of the block: A block (AKA physical record) is a sequence of bytes or bits, usually containing some whole number of records. In Apache Hadoop, the recommended minimum block size is 64MB and may be increased to higher size. However, since each task is carried out on a block of data, a single failure may result into redoing the work for the data of the size of one block. Hence, keeping a large block size may not be ideal. For the given problem we wish to compare the time consumption for aggregation task when block size is 64MB and 128MB.
    2) Optimal input-split size: While processing the data in blocks, each block is split into manageable sizes called input-split. The size of input-split is a critical factor that determines the total main memory that may be utilized by the Map Reduce function. The decision regarding input-split size is based on the minimum values set in the configuration files and a heuristic that determines a good size based on block size and other parameters or as set by the programmer/developer. These parameters may be controlled easily by using relevant functions in java Map Reduce API.
    3) Optimal compression format: Data transmission using intermediate stages like between Mapper and Reducer tasks occur by re-writing output of the Mapper onto disk. Since,

disk operations are 300 times slower than the main memory operations, compressing the data during intermediate stages may save both space and time. Since, data needs to be retrievable even after selecting only a portion of the compressed data. It becomes essential to compress using only splittable compressing techniques like LZO, bzip2, or snappy. Hence, for the given work we wish to test the optimal compression format for no compressing vs snappy compression technique.

The parameters for comparison for the above technologies are wall clock time for completion and amount of data written to disk during intermediate stages (only for Map Reduce).

IV. Dataset

To serve the purpose of the given work, taxi trip record dataset from New York's Taxi and Limousine Commission (TLC) for yellow taxi has been used [3]. The dataset has information about vendor id, total fare with breakups, co-ordinates for pick and drop-off, and pick up and drop of time. The data is available as files storing monthly data that begins from January, 2009 and runs till June, 2018. The total size of dataset in 215 GB. For the given problem, we would calculate the minimum and maximum fare by month and year for the Map Reduce and maximum value for the MPI problem.

V. Discussion:

Use of distributed & parallel programming platforms have been known to require a considerable efforts in getting the right libraries, debugging network problems, safety against race conditions, and implementation of data distribution logic. For the proposed work, we have two sets of techniques wherein the said efforts would be required. Both of these techniques need different form of skillsets and expertise. While Map Reduce requires competency in programming language java (for better control over the jobs which is available through only Java Map Reduce API), handling library issues (majorly library version problems or corrupted library builds) and YARN based scheduler policy issues (especially in allocating virtual memory and main memory); MPI requires expertise in programming language C/C++, handling race conditions, and debugging networking skills.

For the proposed work, so far following issues have been encountered and few which have been solved:

For Map Reduce:

1) Oracle Java version problem: Apache Hadoop build process displayed error because Java 9 removed the activation package as a part of its main release. Initially, the problem was solved by using JAVA_OPTS as -D.javax.activation. However, later due to consideration of issues of the same package for running the software, Java 9 was degraded to Java 8.
2) Compression Library Issues: There were issues encountered during the build process of Snappy library support with Apache Hadoop build on Windows 10. This was solved by custom building the snappy library using Microsoft Visio. However, the build errors with snappy still prevailed. Finally, the system was shifted to Linux Mint for hosting the Apache Hadoop solution. This worked well and passed the initial debugging tests for the snappy library and has not given any warnings or errors related to Snappy library for four Map Reduce job tests. These tests were carried out using Map Reduce standard word count program.
3) Memory issues: Virtual memory limit issues had been encountered while performing Map Reduce on relatively small data of 700 MB. The culprit was identified to be a YARN policy. YARN does not allow virtual memory to exceed the size of main memory reserved. This was handled by suppressing the test of virtual memory calculation by setting the "yarn.nodemanager.vmem-check-enabled" flag as false. This is because it was found that enough virtual memory is available (The system has a swap space of 8GB of which less than 3GB was utilized as found using the "top" command and confirmed using "VmStat"). Another work around the same was found to be using the right value for "yarn.nodemanager.vmem-pmem-ratio". It was identified that 2.1 usually works as the best estimate for "yarn.nodemanager.vmem-pmem-ratio" and the same may be considered in future if a related situation arises in the future.
4) Storage size estimation: The amount of data to be analyzed is 215 GB for Map Reduce. However, an estimation of size of the data during intermediate stages must be carried out to make sure that there is enough space for the intermediate data.

5) Data ingestion problem. Since we have data in 102 files to be loaded into HDFS before being able to perform analysis, it becomes essential to have a software to ingest the data. From literature we found multiple ingestion software's like Apache Gobblin, Apache NIFI, Apache Kafka, and others. From a review of these solutions it was found that most of them uses Web HDFS to ingest data and hence it was decided that a layer of abstraction by using another software is not a good way to ingest the data. Hence, data would be ingested by using traditional approach using the hdfs command-line tools.

For MPI:

1) Debugging challenges for network programming part.
2)

The following work would be able to identify the parameters that works best for aggregate tasks for MPI and Map Reduce. Due to limitations of penalty imposed by SHRCNET on long running jobs, I wish to cut down the dataset for processing of MPI to 5 GB. If time permits, analysis with larger dataset may be carried out.

A lot of tasks must be accomplished in the given project and the same would be carried out using plan as shown in Figure 1.

Tasks\Month		October			November			
		2nd Week	3rd Week	4th Week	1st Week	2nd Week	3rd Week	4th Week
MPI SHARCNET Tasks	Optimal no of Processors		Y					
Map Reduce	Optimal block size	Y						
	Optimal input-split size		Y					
	Optimal Compression format			Y				
Compile Results					Y			
Report Writing					Y			

Figure 1. Plan for the project

VI. Conclusion

To tie a bow, an aggregate query to find minimum and maximum value of fare by month and year for Map Reduce & minimum fare value for MPI has been proposed to study the performance & identify the best parameters for these systems. For the same after review MPICH over SHARCNET and Apache Hadoop for Map Reduce have been suggested for measuring the performance. The parameters for comparison suggested are wall clock time for completion and amount of data written to disk during intermediate stages (only for Map Reduce). For performing the proposed analysis, NYC TLC taxi trip records dataset has been suggested, which is 215 GB in size.

Bibliography

1. Map reduce, Sandia Labs, http://mapreduce.sandia.gov/
2. Apache Hadoop, Hadoop Documentation, https://hadoop.apache.org/docs
3. NYC Taxi & Limousine Commission, New York http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml
4. Lee, K. H., Lee, Y. J., Choi, H., Chung, Y. D., & Moon, B. (2012). Parallel data processing with MapReduce: A Survey. ACM SIGMOD Record, 40(4), 11-20.
5. Dittrich, J., & Quiané-Ruiz, J. A. (2012). Efficient big data processing in Hadoop MapReduce. Proceedings of the VLDB Endowment, 5(12), 2014-2015.
6. Doulkeridis, C., & NØrvåg, K. (2014). A survey of large-scale analytical query processing in MapReduce. The VLDB Journal—The International Journal on Very Large Data Bases, 23(3), 355-380.
7. Chen, C. P., & Zhang, C. Y. (2014). Data-intensive applications, challenges, techniques and technologies: A survey on Big Data. Information Sciences, 275, 314-347.
8. Wes Kendall, Launching an Amazon EC2 MPI Cluster. http://mpitutorial.com/tutorials/launching-an-amazon-ec2-mpi-cluster/
9. Lu, X., Liang, F., Wang, B., Zha, L., & Xu, Z. (2014, May). DataMPI: extending MPI to hadoop-like big data computing. In Parallel and Distributed Processing Symposium, 2014 IEEE 28th International (pp. 829-838). IEEE.
10. Kang, Sol Ji, Sang Yeon Lee, and Keon Myung Lee. "Performance comparison of OpenMP, MPI, and MapReduce in practical problems." Advances in Multimedia 2015 (2015): 7.

11. Jin, Hui, and Xian-He Sun. "Performance comparison under failures of MPI and MapReduce: An analytical approach." Future Generation Computer Systems 29.7 (2013): 1808-1815.
12. Kshemkalyani, A. D., & Singhal, M. (2011). Distributed computing: principles, algorithms, and systems. Cambridge University Press.
13. Sharcnet HPC. (Sept, 18, 2018). Concurrent File I/O by Multiple Processes. https://www.youtube.com/watch?v=3s-dBEopfwQ