TABLE OF CONTENTS

On Studying the Performance of Hadoop Map Reduce vs. MPI for Aggregation Operations: A Big Data Challenge

- Devang Swami

I. INTRODUCTION

IT Revolution has been chiefly responsible for generating a tremendous amount of data, often referred to as Big Data [5]. The challenge to analyze big data begins from the very first set of analytics stages wherein the analyst performs aggregation tasks to summarise the data with an objective of understanding the data. Since big data is quite massive in size, it becomes essential to reduce the processing time it takes on average to compute the aggregation operation so that a single iteration of analytics cycle is completed in the shortest possible time [4]. This challenge of computing aggregation in the shortest possible time gives an opportunity to utilize parallel and distributed processing systems that can scale horizontally.

Of various distributed & parallel computing systems like vector processing systems, coprocessors (e.g., GPU's), and others, Map Reduce, and Message Passing Interface (MPI) have found increased attention in recent times because of their capacity to scale horizontally very well. MPI has been a de facto standard of parallel programming for decades [18], while Map Reduce is of comparatively recent development. A significant difference between both the platforms is that prominent Map Reduce implementation (i.e., Apache Hadoop) uses HTTP/RPC protocols for communication which are slower than the MPI's communication (which provides both point-to-point and collective communication) [1][37]. On the flip side, programming in MPI is more difficult and requires lots of skills (in programming, networking, domain knowledge, and others) in comparison with Map Reduce. Both the solutions are used extensively in the development environment.

However, when it comes to performing aggregation on big data, the challenge is not only to scale the workload horizontally but also to select optimal strategies for executing the workload on relevant parallel and distributed solutions. For the proposed study, the concentration is on optimizing the MPI and the Hadoop Map Reduce for aggregation workloads and then compare them. Both the solutions provide different ways of tweaking the workload execution. For example, Hadoop Map Reduce there are three parameters (block size, input split size and compression) that have been found to have made a direct impact on the computation time [1] while MPI puts a limit on the improvement based on the operation to be computed [41]. For instance, with aggregation operations, one could read the data file in parallel [31].

The proposed study aims to optimize the time it takes to compute an aggregation operation on Hadoop Map Reduce & MPI and then compare the wall clock computation time it takes for these solutions to execute an aggregate query. The dataset used for this study is New York TLC taxi cab dataset for yellow taxi trip records starting from Jan 2009 to Dec 2017 [28]. The dataset has information about vendor id, total fare with breakups, co-ordinates for pick and drop-off, and pick up and drop off time [28]. The data is available as files storing monthly data that begins from January 2009 and runs till December 2017 [28]. For this dataset, it would be interesting to know which records have a minimum value of the taxi ride fare for each month and year. Since a minimum value of taxi ride fare less than $2.5 could necessarily mean that the data point/s is/are potentially stale, and further research may be required to uncover bugs in the data collection and data preprocessing system. These experiments would be performed on Single Node Hadoop cluster on a local system for Map Reduce and MPI based tests would be performed on Cedar clusters which are a part of the Compute Canada.

II. LITERATURE REVIEW

i. Background on Big Data Techniques

It is a well-established fact that Map Reduce finds application in the business community while MPI has been used more by the research community [17]. There many available implementations of both the technologies to choose from. For instance, available MPI implementations are Open MPI, Boost MPI, MPICH, and others [14], while available Map Reduce implementations are Apache Hadoop, Apache Spark, MongoDB, Clusterpoint XML, Apache Cassandra, Hive, HBase, Big Table, Oracle Big Data SQL, MarkLogic, and others [1][9].

Among many available implementations of both solutions, it becomes clear that communities and businesses around the world have extensively utilized these technologies. To narrow down the solution list, many parameters such as availability of support, online documentation, ease of use, and other parameters are taken into consideration.

For MPI, it was found that almost all the implementations had both proper documentation and online community support. MPI based tests were performed on the Cedar clusters on the SHARCNET consortium. An investigation of the Cedar Clusters, it was found that Cedar clusters used Intel E5-2683 v4 "Broadwell" processors at 2.1Ghz [4]. Also, the Intel MPI library and the Intel XE Parallel Studio both have been optimized for usage on Intel processors [14], and it seems a fair choice to use Intel MPI library for the proposed study. The Intel MPI library used for this study was Intel MPI v2016. These tests would be performed on Cedar clusters of the SHARCNET consortium.

For Map Reduce, the challenge is that every software claims optimized map-reduce operations for specific use cases that these tools were designed to solve. Clusterpoint XML was one of the earlier supporters of Map Reduce, but the support for Clusterpoint in no longer available [10]. Among other implementations, Apache Cassandra, Big Table & HBase are more optimized for messaging services-oriented workloads, Oracle Big Data SQL and Hive have been designed as data warehouses [10]. This leaves out only MongoDB and Apache Hadoop implementations of Map Reduce. The disadvantage with MongoDB is that it would always have the data ingested into a type of JSON format called BSON and this may not be an optimal strategy for relational data [11]. This leaves us with Apache Hadoop as the only possible candidate. Also, [11] showed that Hadoop Map Reduce performs better than MongoDB build-in Map Reduce. Apache Spark is also an excellent solution and performs better than Apache Hadoop because it distributes workload using main memory instead of disks. However, the performance of Apache Spark is limited by the amount of main memory (or RAM) available [30]. Also, Apache Spark tends to perform worse on Commodity hardware especially due to the limited amount of main memory available on commodity hardware [30]. Thus, it was decided that Apache Hadoop would be used as a solution to test the Map-Reduce framework performance. For the proposed work, a single node implementation of Hadoop would be utilized. The purpose of this study is not to test scalability but parameters apart from scalability that may have a significant impact on the performance of the system.

Hence it becomes essential to review the architecture of these systems:

1) Hadoop Map Reduce Architecture: Apache Hadoop is a cluster-based solution that offers a distributed filesystem and MapReduce programming framework to store and analyse massive amount of data [1]. It consists of many entities that coordinate together to provide services. For instance, DFS or HDFS software provides the distributed file system for storing data, YARN (Yet Another Resource Negotiator) provides a management framework to schedule jobs on the Hadoop cluster, a history server keeps logs of previously completed Map-Reduce jobs [1].

   The HDFS filesystem consists of three processes, a namenode, a secondary namenode, and a datanode [1]. The purpose of Namenode and secondary namenode is to maintain the metadata about the different files loaded in the filesystem. Namenode acts as the primary interface between the user and the HDFS file system [1]. If the namenode is down, then secondary namenode takes its charge [1]. The purpose of datanode is to manage the data in its system [1]. There can be only one namenode in the HDFS but multiple datanodes across the Hadoop cluster. Purpose of datanode is limited to storing the data.

   On the flip side, YARN consists of a resource manager, an application manger and one or more node manager [1]. A resource manager is the software that manages all available resources on the Hadoop cluster, while the node manager is responsible for managing resources on individual nodes in the cluster[1]. Once the job is submitted by a user, it is received by application manager [24]. An application manager is responsible for negotiating resources for the submitted application with Resource Manager. Once the resources are available for the application, it starts execution. This execution in controlled by two types of processes Job Tracker and Task Tracker. Where a Job Tracker is concerned with overall completion of the job, it is the duty of Task Tracker to ensure successful completion of each task[24].

The process of analyzing data in Apache Hadoop begins with uploading the data to the distributed file system (i.e. HDFS file system). Then, the client may submit a job to process the uploaded data on the HDFS filesystem. A client submits the job to Application Manager which negotiates the resources with resource manager and then finally the execution of the job is managed by Job tracker, an entity that is responsible for breaking up the job into tasks that achieve some work to complete the job [24]. It is the job of Job Tracker to keep the work as close to the data as possible. Each task assigned by the Job Tracker is managed by a Task Tracker. A Task Tracker ensures the successful completion of each task. Should a task fail, the Task Tracker informs the Job Tracker about the same, and then the Job Tracker may spin a new Task Tracker for the failed tasks. Most of these tasks may be classified into one of the following seven types [1][24]:

1) InputSplit: This stage splits the data into logical partitions. It reads a chunk of data from the HDFS filesystem. In layman's term, "InputSplit says Start at offset x, and continue for y bytes" [24].
2) Record Reader: This stage takes the input split as an input and reads records from the same. These records may be split based on various heuristics. The most common heuristic is to use the newline character as the determinant of the end of record. This heuristic is implemented as TextInputFileFormat class in the Map-Reduce Java API.
3) Mapper: Also known as a Map task, this task takes the data as input and produces them as key value pair. This tasks in commonly employed with preprocessing the data.
4) Reducer: This task combines the data from multiple tasks and performs compute on them by grouping the records together by value of the key.

5) Shuffle: This stage of Map-Reduce is employed to shuffle the records so that records with same value of key are grouped together.
6) Sort: This stage sorts the input key-value pair based on the value of the key.
7) Merge: This stage merges records from multiple systems and brings them to a common system for processing. It is usually carried out after the Map stage to bring all the records to the system where reducer would be spun.

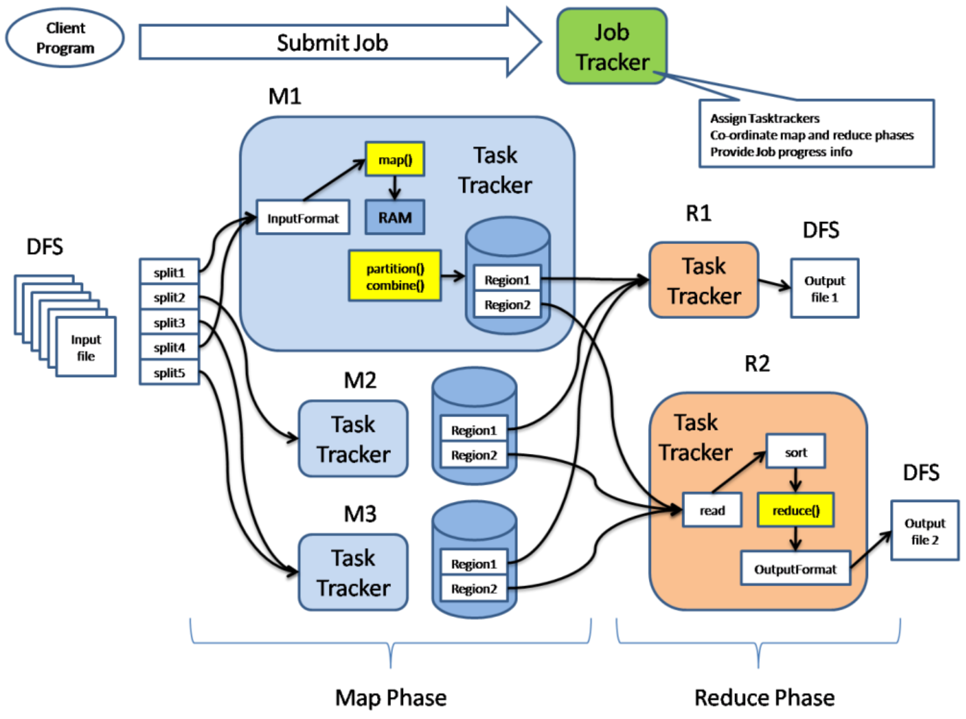


Figure 1 Map reduce process in Apache Hadoop [24]

The complete process of Map-Reduce is displayed in Figure 1. The process begins with user submitting the job, which is broken down into different tasks. The number of tasks created by the Job Tracker depends on the size of the data and the input split size which determines the amount of data that is processed by each task. This splitting of data is rather logical than physical. In most scenarios, it is found that overhead of starting a task is relatively higher and hence a higher value of input split size is recommended. However, an input split size larger than the block size of the HDFS could result in invoking read from a system on the network and hence the same is not recommended. Once, the tasks are decomposed they are computed usually as a pipeline of tasks in the order, Mapper, Shuffle/Merge, and Reduce [24].

It should be noted that between two stages there may be data interchange and hence reducing the amount of data exchange also becomes important. This is sometimes termed as “small data aside big data”. Use of compression to minimize network data exchange is one of the simplest solutions for this scenario. Other scope of improvements in the Map-Reduce frameworks include decreasing the task failure rate, replication of data (increases fault tolerance), and others [1] [15][17][24] [36].

2) MPI Architecture: “MPI defines an API that is used for a specific type of portable, high-performance inter-process communication (IPC): message passing. Specifically, the MPI document describes the reliable transfer of discrete, typed messages between MPI processes” [35]. MPI finds application in developing distributed and parallel application for High Performance Computing (HPC). The hierarchy for MPI is dependant on its implementation. Figure 2 displays the MPI architecture implemented by the Open-MPI framework. The architecture for other implementations like Intel MPI, MPICH, and others is also almost similar.

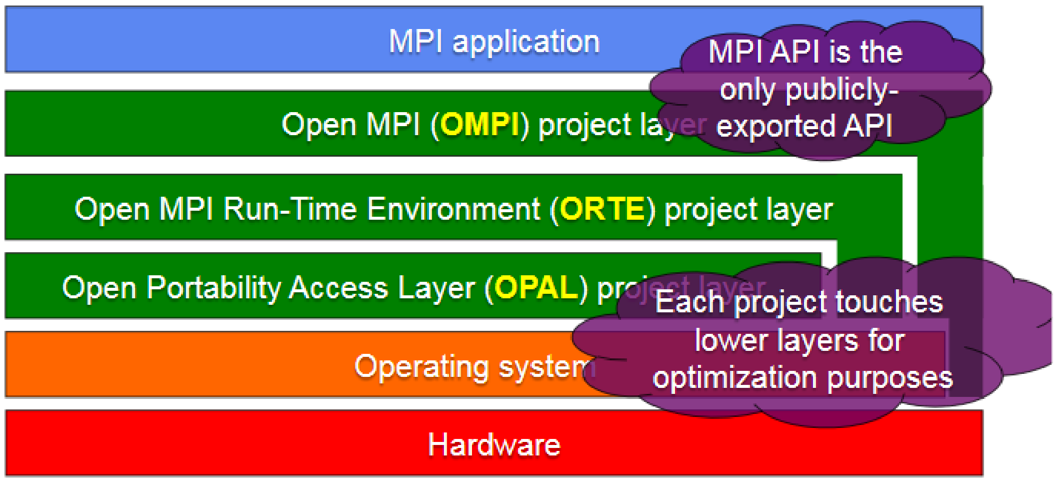

MPI application
MPI API is the only publicly-exported API
Open MPI (OMPI) project layer
Open MPI Run-Time Environment (ORTE) project layer
Open Portability Access Layer (OPAL) project layer
Each project touches lower layers for optimization purposes
Operating system
Hardware

Figure 2 MPI Layered Architecture [17]

The MPI provides routines that allows communication between the processors over a network that allows them to distribute the processing task. MPI has been a de facto standard for implementing distributed and parallel applications because it supports both peer-to-peer as well as group communication [17]. The MPI architecture consists of a layered hierarchy of libraries to provide various message passing routines.

In general, MPI applications developed on Open-MPI library execute on top of the Open MPI project layer, that abstracts the Open MPI runtime environment and Open Portability Access Layer [34]. The purpose of the Open MPI project layer is to provide a framework that allows few routines to communicate directly with the hardware by bypassing the operating system calls [34]. This increases the efficiency of the MPI application. Open MPI runtime environment and Open Portability Access layer provide additional functionalities to MPI applications to run atop the operating system.

Important aspect of the MPI applications are its data transfer routines. Most commonly used data transfer routines are as follows [17][34][37][38]:

1) MPI_Send/MPI_Receive: These routines are used to send and receive data. Here one process sends the data and the other receives the same. These have been found to be not efficient owing to chances of having an unsafe data transmission if proper care is not taken during programming.
2) MPI_Bcast: This method uses MPI_Send and MPI_Receive to send the data. The limitation of this method is that it sends data to all the processors and hence is not optimal
3) MPI_Scatter/MPI_Gather: To overcome the limitations of the MPI_Bcast, Scatter and gather functions of MPI send only a portion of the data to each node thus reducing the amount of data transmission over the network. These functions are useful only when the number of data points are a multiple of number of processing nodes.
4) MPI_Scatterv/MPI_Gatherv: These routines work like MPI_Scatter/MPI_Gather functions except that they allow sending data of unequal sizes to different processors. Usage of these functions requires a special algorithm that calculates the displacement of the data that has to be send to each process.
5) MPI_IO: These are a group of routine that provide functionalities to read data from a file over the network. This method is more suitable for applications that

must process very large amount of data. Use of MPI_IO function also requires the data file to be viewed as a shared resource and this poses more challenges in using these routines for the MPI applications.

MPI routines finds application based on the amount of data exchange that is to be carried out by the processor to complete the task.

From the architectures of Hadoop and MPI, it is clear that they are two different animals and MPI is more sophisticated and allows maximizing the amount of parallelism that can be explored. Above all, MPI applications have to communicate with a far smaller number of processes in comparison with Hadoop Map-Reduce which has a uniquely strict hierarchy to be followed for executing a job. MPI processes interact with themselves, while with Map-Reduce applications have to interact with application manager, resource manager, namenode, datanode, node manager, job tracker, and task tracker. However, these interaction helps Map-reduce recover from task failures and enables fault tolerance, while ensuring the same in MPI is very challenging and requires implementing a fault tolerance from scratch, which may be counter productive given the time it takes to write a fault tolerant application.

ii. Related Works

An entrance into the era of Big Data, which is the next frontier for innovation, competition, and productivity, a new wave of scientific revolution is about to begin [23]. This revolution has brought an opportunity to test various approaches like multi-core systems, GPU's, clusters, and other technologies to process a massive amount of data in parallel. Of these technologies, Apache Hadoop and MPI have found a unique place as discussed above. While Apache Hadoop is limited by the execution strategy of using a Map-Reduce framework, MPI is an open platform. MPI is a language-independent communications protocol for parallel computing where point-to-point and collective communication both are supported [39]. "MPI is currently the dominant model used in high-performance computing and is a de facto communication standard that provides portability among parallel programs running on distributed memory systems" [30]. Hence it is evident that Map Reduce is slower than MPI when it comes to performance.

Many works have presented a comparative study of Map Reduce vs. MPI. It has also been clear that Hadoop Map Reduce is still very far from achieving state-of-the-art performance like that of MPI [30]. However, it must be noted that the time required to implement the HPC using MPI is relatively very high. This is also the very reason that businesses have discouraged the use of MPI. One motivation that comes from previous works is in the form of testing scalability using different options. For instance, [30] [39] focus on testing scalability and performance of Map-Reduce based systems with that of MPI. However, essential tasks like that of optimizing the work on both the platforms have not been detailed in any of these works. An exception to these works would be [13] which focuses on data layout and indexing for job optimization on Hadoop. However, fundamental factors such as compression, block size, input split size, and other parameters have not been considered for optimization.

Moreover, most works in the literature review also do not focus more on the row-based layout of the data. This is a vital piece of information because it puts a limit on operations that can be performed on the data. For example, the challenge with row-oriented data is in designing an algorithm that logically splits based on the record. In Hadoop Map Reduce, a special TextInputFileFormat allows logical splitting of records based on newline character, which is helpful if the data is in CSV/TSC format. However, this becomes challenging on MPI since this is not available as a function in the standard MPI library implementation [31]. This feature in Map Reduce would reduce the development time by a considerable amount in comparison with MPI. Non-functional requirements like the performance of MPI and Map Reduce under failures has also been a subject of concern for [18]. The limitations of [18] were the fact that the results of MPI and Map Reduce under failures by changing parameters were not studied. The authors of [18] proposed to expand the work in the future by introducing models that consider the block size, number of replicas, and others. To tie a bow, most of the past works which had compared Map Reduce & MPI, have mostly not introduced on how to optimize both solutions. It would be interesting to notice the behavior of these systems when limits are stretched for more possible optimizations.

It is clear from the past works that less emphasis has been paid on optimizing the program execution on Map-Reduce and MPI before testing their performance. This may be unfair because it does not stretch both the systems to their limits before comparing them.

III. PROBLEM STATEMENT

This study aims to minimize the wall clock time it takes to compute aggregation operation on Map Reduce and MPI and then compare the results of both the systems. For the same, optimal parameters for Map Reduce & optimal strategies for MPI needs to be benchmarked. The aggregation query to that for benchmarking these systems under review for the current study calculates minimum taxi ride fare by month and year for NYC Taxi trip records. The parameters for tuning the performance of Hadoop Map Reduce and MPI are as follows:

1) Hadoop Map Reduce:

Map Reduce usually works more optimally when computation can be carried out using one-pass algorithms (An algorithm that computes by passing through data only once). For the given problem, we are finding the minimum value grouped by year and month, which is possible to be implemented using a one-pass algorithm. Since Map Reduce already is optimized for such works, no programming optimization is required. However, there are multiple factors such as the amount of data chuck read for each task (a.k.a. input split size), size of the data block, and the choice of intermediate using compression; can impact the processing time of Map-Reduce tasks. Thus, we have the following three parameters for the Apache Hadoop Map Reduce implementation to optimize:

1) Optimal size of the block:

   A block (a.k.a. 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 a higher value but only in multiples of 2. However, since each task is carried out on a block of data, a single failure may result in 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 the block size is 64MB and 128MB.

2) Optimal input-split size:

   While processing the data in blocks, each block is broken down into logical blocks manageable sizes called input-split. The size of the input-split is a critical factor, and it determines the total amount of main memory that may be utilized by the Map-Reduce function. Thus, input-split presents the data as a logical block. An algorithm computes the decision for the input-split size in Hadoop Map-Reduce using a heuristic that determines a reasonable size based on the block size of the HDFS file system and other parameters. It can also be set conventionally by the programmer/developer per job by using relevant methods provided by Map-Reduce API. These parameters may be controlled easily by using relevant routines from the Map Reduce API.

3) Optimal compression format:

   Data transmission using intermediate stages like between Mapper and Reducer tasks occur by writing the 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. Also, we need to compress data such that data file splits can be decompressed independently of the other splits from the same file. It becomes essential to compress using only splittable compressing techniques like LZO, bzip2, or snappy [1]. For the given work snappy compression was tested to decrease the processing time in comparison with when not using compression. The parameters for comparison for the above technologies are wall clock time for the workload completion.

Thus, for Map Reduce parameters like block size, input split size, and compression format have been considered for analysis, and the same would be used to optimize the performance of Map-Reduce program.

2) Message Passing Interface (MPI):

Aggregation task for the given problem which is to calculate the minimum taxi ride fare by month and year may be carried out using multiple strategies, and these strategies differ based on the amount of parallelism explored. One of the conventional approaches for MPI based solutions begins by reading the data from a file in the root processor after which it is scattered for computing by multiple processors, and finally, the results from multiple processors are merged. However, there are many opportunities to optimize this strategy. For instance, the said strategy may be improvised by reading data concurrently into different processors or optimize the methodology that reduces the overhead in combining the results from multiple processors (a.k.a. Reduce operation). For the proposed study, the following two approaches are being reviewed:

1) Sequential Read: The Master node reads the complete data and distributes it to the slave nodes for processing. The slave than returns the processed data which are eventually combined to find the final solution. The merit of the technique is its simplicity. The following technique is more favorable in the presence of a system that has a substantial amount of main memory available.
2) Concurrent Read: Each processor determines the logical split of the data chunk it is supposed to process and then reads the same. This approach works well if data is being read from a single large file and if the MPI infrastructure supports the relevant function, theoretically. The approach saves much time by minimizing the time to transfer data.

The difference between the two approaches under review is in the amount of parallelism that is they are exploring. Figure 3 demonstrates the flow chart of the two approaches under review. Now based on Amdahl's law, since Approach 2 explores more parallelism than Approach 1, theatrically Approach two should perform better. However, it would be interesting to check if Approach 2 works as good as Amdahl's law predict and review the scope of any further improvement in these strategies. Thus, for MPI two strategies would be tested for performance to identify the bottlenecks.

Finally, the optimal results of MPI and Hadoop Map Reduce would be compared using wall clock time for task completion as a critical parameter apart from code development time.

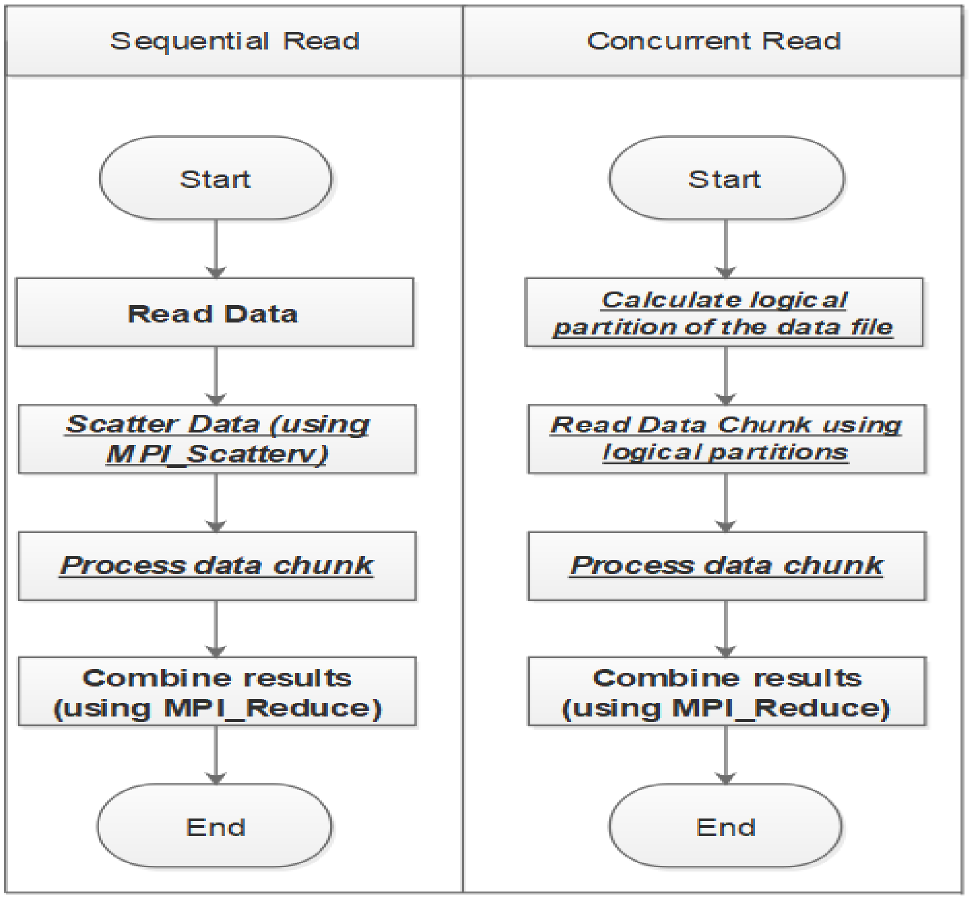

Sequential Read
Concurrent Read
Start
Read Data
Scatter Data (using MPI_Scatterv)
Process data chunk
Combine results (using MPI_Reduce)
End
Start
Calculate logical partition of the data file
Read Data Chunk using logical partitions
Process data chunk
Combine results (using MPI_Reduce)
End

Figure 3 Sequential Read vs. Concurrent Read MPI program flowchart. The processes that are italics and underlined are performed in parallel.

IV. PROJECT PROCESS & DISCUSSIONS

The scope of this study was comparatively large in comparison with the time which gives an opportunity to use Agile software development strategy to organize the study. "Agile software development is an approach to software development under which requirements and solutions evolve through the collaborative effort of self-organizing and cross-functional teams and their customer(s)/end user(s)." [8]. Of different frameworks available in Agile for the organization the most popular is SCRUM. Scrum is commonly used for managing knowledge works and hence was selected to manage and organize this study [35][41]. SCRUM begins by breaking down the complete work into timeboxed actions called sprints. Most projects allocate between 2 weeks to a month for one sprint. However, due to the nature of this study, the length of each sprint was kept 35 hours, and there were in all 12 sprints. The sprints were further granularized to tasks for ease of management. In all 93 tasks were distributed among these 12 sprints. Use of SCRUM allowed the progress to be checked and managed using weekly sprint updates using burndown chart.

For ease of organizing the study using the SCRUM framework, the 12 sprints were grouped into two parts as below:

Sprints for Hadoop Map Reduce tests:

1) Apache Hadoop Review, Build & Setup:

   "The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage." [9]. The Hadoop Map-Reduce performance comparison experiments were carried out on a single node Hadoop cluster. The properties of the cluster designed are as below:

   i. A system with 8 GB main memory and Intel Core i5 - 2.3 GHz processor was used for deployment. The system had 4 logical cores.

   ii. Linux mint 19 cinnamon was used as the operating system since it is very lightweight. Linux Mint consumes only 1 GB memory while other OS such as Ubuntu, Fedora, Windows 10, OpenSUSE, Debian, and others require a minimum of 1.6 GB of main memory.

iii. Source code for Apache Hadoop v3.1.1 was used to build the binaries for the Hadoop cluster with the support of Lzo4 and snappy libraries [1]. Few build errors were encountered because Java 9 no longer support activation package. Initially, for first few builds the package was supplied as an optional package by using JAVA_OPTS. Finally, the java version was degraded to Java 8 to avoid any discrepancies that may arise.

iv. The primary objective of building Hadoop was to ensure that the binary supports Intel ISA-L library which is used for erasure coding. Although, there is no network component involved, having support for erasure coding would ensure error correction when inserting a massive amount of data. Also, this library would optimize the binary for execution on Intel systems [1].

v. Single node Hadoop cluster was set up with support for snappy and Lzo4 compression libraries.

vi. There were 4 logical processors on the system and hence only 4 processes were running at a time. These were Operating System, HDFS processes, job history server, and finally the Mapper/Reducer process. This design comes from the recommendations of the Cloudera system provisioning excel sheet.

vii. Size of the disk space available allocated to the HDFS is 500 GB.

viii. The block size for the HDFS filesystem was kept 128 MB or 64 MB as deemed necessary for the tests.

ix. The minimum amount of main memory allocated for Mapper processes was 128 MB and was increased in multiples of 128 MB moreover if the mapper process requested more memory up to a maximum of 1024 MB. This was done by setting the java-opts for mapred child processes to -Xmx1024M.

x. The maximum number of virtual cores per mapper was one.

xi. Only one mapper process and one reducer process were invoked for the given set of experiments.

xii. Uberized mode was disabled since implications of running mapper process inside of Application Manager were not known. However, enabling uberized mode has been found to be more useful when running interactive queries on Hive when using Hadoop as a sink [1].

2) Data Ingestion into Apache Hadoop: The challenge here is to ingest more than 200 GB data which is stored in the form of 108 CSV files. This was carried out using Hadoop command-line tools, which was found to be more optimal than other tools. Finally, the consistency of ingestion was ensured by comparing the md5 digest hash for each of the 108 data files.
3) MapReduce program development & testing: In this sprint, a word count program of Map Reduce was modified to be able to change the task to aggregation operation as required by the study. The MapReduce WordCount program was obtained from official Hadoop documentation. Use of a pre-developed code help reduces the task completion time. On testing the program, more modifications were introduced in the form of data quality checks so that bad records may be removed. For instance, for few data points, a "-" was used to denote a missing value. Such values caused the MapReduce to fail giving error "Map Reduce failed due to NA." To identify these bad records more than 5 checks using if-else blocks were used. It became necessary to reduce the number of if blocks for better performance. Initially, the SkipBadRecords class was considered as a possible solution. However, the new Map Reduce API does not support this operation any longer. Hence, to reduce the number of if-else blocks, regular-expressions were used. Usage of regular expression decreased the no of if-else block to only one statement. Above all, there were changes in the schema of the data from 2015 onwards and the same also had to be accommodated in the Map-Reduce program [28].
4) Map Reduce Test for optimal compression choice: This sprint demanded the compiling of a snappy library to move forward with the tests. Map Reduce tests were first performed without using compression and then using snappy compression. At the end of this sprint, it was established that the processing of data by Map Reduce could take anywhere between 6 to 18 hours and this information helped in rescheduling the later tests. This proved very helpful because later all Map Reduce tests were carried out overnight to have more time to spend on other works. The primary challenge in this sprint was that the Map-Reduce execution should be uninterrupted thus the system should always be
5) Map Reduce Test for optimal block size: The tests for optimal block size were carried out in this sprint. The primary challenge here was in the organization of the tasks. Since

after changing the block size, HDFS needs to be formatted, and then the complete dataset has to be ingested again. The data ingestion process takes on an average of 5-7 hours. Finally, it was decided to ingest the data over the afternoon and then performing the Map-reduce tests over the night to be able to achieve the task completion on time. The decision comes from the fact that data ingestion requires attention because an md5 hash comparison needs to be carried out for each data file to ensure the data consistency after ingestion.

6) Map Reduce Test for optimal input split size: This sprint was the most challenging with respect to time because it was estimated that the longest running Map Reduce task would take more than 20 hours. Above all, there were three Map Reduce tests in total to be performed in this sprint. The strategy used for this sprint was to start the Map-Reduce process for the longest job at 6 p.m. so that the results are available by 3 PM on a succeeding day. The same strategy was used for the rest of the tests as well. The test for input split size was also more complicated because it required rebuilding the Map Reduce bytecode for execution as the input split size is provided inside the Map-Reduce implementation.
7) Analyze & compile the results: This sprint was quite straightforward with challenges only lying on finding a possible explanation of the results obtained from various tests.

Sprints for MPI test:

8) Hands-on Cedar and uploading data: This sprint comprised of getting hands-on Cedar cluster of SHARCNET with an objective of understanding how the cluster works and how jobs may be submitted. The first task in this sprint was to enable the Intel MPI library for compilation and execution of the MPI workload. Then, basic 'Hello World' MPI programs were executed to get familiarized with the system. Finally, the data file was uploaded to the file servers of Cedar cluster that stored data file using a block size of 16 MB [4].
9) Basic MPI program development & tests: A simple "Hello World" MPI program was used as a sample and modifications were introduced to carry out the aggregation query as detailed in Section 3.2. Basic tests were performed to uncover most of the bugs and memory leaks. These tests were performed on a local system using Microsoft Visual

Studio & Intel Parallel Studio XE to estimate the size of main memory required and the processing time. It was found helpful to use '- Wall' parameter while compiling the MPI programs because this parameter enables displaying all the warnings that the compiler has encountered. Apart from the same, there were issues in using C++ 2011 libraries on Cedar cluster essentially because of the settings in the GCC compiler. The remedy for the same was found to be enabling C++ v11 by using the flag '-lstd++' and '-std=c++11'. The MPI programs were executed on 4 nodes of Cedar cluster and used only one core per node. The amount of main memory requested from each of these nodes in 1GB for sequential read version and 512 MB for concurrent read version. These nodes on Cedar cluster were using Intel E5-2683 v4 "Broadwell" at 2.1Ghz processing [4]. The maximum completion time for these programs was set to 20 minutes since it was available through analysis that the program would terminate within 18 minutes when the debugging mode was enabled. Enabling debugging mode would printing all the records for both the programs and displays the values for relevant variables to help debug the program.

10) MPI Test for sequential read version: This sprint demanded significant modifications to the basic MPI hello world program so that more parallelism could be explored. Many significant improvements had to be introduced to be able to optimize this program. One of the most significant improvement was in using the MPI_Scatter function over MPI_Bcast Apart from the same use of MPI_Reduce over MPI_Gather also decreased processing time significantly. However, while executing the workload over complete dataset, it was found that not all the data points are being processed. The reason for the same was the fact that the number of data points was not a multiple of the number of processors. Hence, the MPI_Scatter function was replaced by MPI_Scatterv. Use of MPI_Scatterv brought its challenge on calculating the displacements and the number of data elements that would be sent to each processor. This was handled swiftly by using previous works as a reference. One of the more challenging tasks in this sprint was that the execution of the program could take an unpredictable amount of time. Since, Cedar uses a queuing system to decide when to execute a job, based on past jobs submitted by the user, resources requested by the job, and many other parameters. The best bet for this challenge was to submit the job as soon as possible and then check

back for its completion at regular intervals. It was found that using Termux application and OpenSSH package on an Android device, terminal access to Cedar cluster was available and used extensively throughout the progression of the study

11) MPI Test for concurrent read version: This was the most challenging sprint owing to the problems it presented. The first problem was to develop an algorithm that could establish a logical split of the data file for each processor. Then it was essential to make sure that the logical splits are such that partial records do not go to a processor. For the same, an overlap was used that allowed duplication of few records between two processors. The proposed solution was primarily influenced by [20]. It was later found that since the study focused on minimum operation, redundancy of records does not change the results.

    Moreover, after a thorough analysis, it was found that an overlap of 250 characters would serve the purpose. Finally, the problem of executing the job on Cedar was tackled by using the same strategy as with the previous sprint by submitting the tasks as soon as possible and checking at regular intervals for their completion. Use of Termux android application to connect and check task completion on Cedar cluster was beneficial for this sprint because more than eight variants of the program had been executed before a working solution was available for benchmarking. Most of the program failures were because of the bugs in the decision regarding the size of the overlap and errands in parsing parameters to the MPI routines.

12) Analyze & compile the results: "Amdahl's theoretical constructs about the performance limits of parallelization are an important foundation to our understanding of how future software will manifest the power of future hardware." [3]. Amdahl's law brought more challenges in the explanation of the improvements in parallel concepts majorly due to the unexplained overhead that had reduced the performance [3]. For instance, it was approximated that concurrent read version of MPI would be completed under 7.5 s according to Amdahl's law. However, the concurrent read version of MPI took 9.3 s. A direct explanation of the same is not available through either the Amdahl's law [14] or the Gustafson's law [19]. More challenging was the task of accounting for the overheads in those parallel processes. Finally, the same was explained using analytical methods than a theoretical explanation.

To sum up the project process, it took 11 weeks to complete all the 12 sprints that were required to be completed as a part of this study. These 12 sprints required a total of 93 tasks. The progress of the study was tracked using the number of tasks as a parameter and using the burndown chart. In SCRUM framework, the burndown chart demonstrates the number of tasks completed and the number of tasks planned for completion during each period [33]. Use of burndown chart enabled for the review of current progress and making necessary amendments to achieve the completion of work on time [7]. Figure 4 shows the burndown chart after the completion of the work. From this chart, it is clear that on an average there were delays in completing 4-6 tasks for most of the sprints and the same was recovered as soon as possible. It is also clear that the work was too tight on schedules, but since there were many independent tasks, it was also simple to move things around and reorganize the project deliverables to deliver results of the study on time.

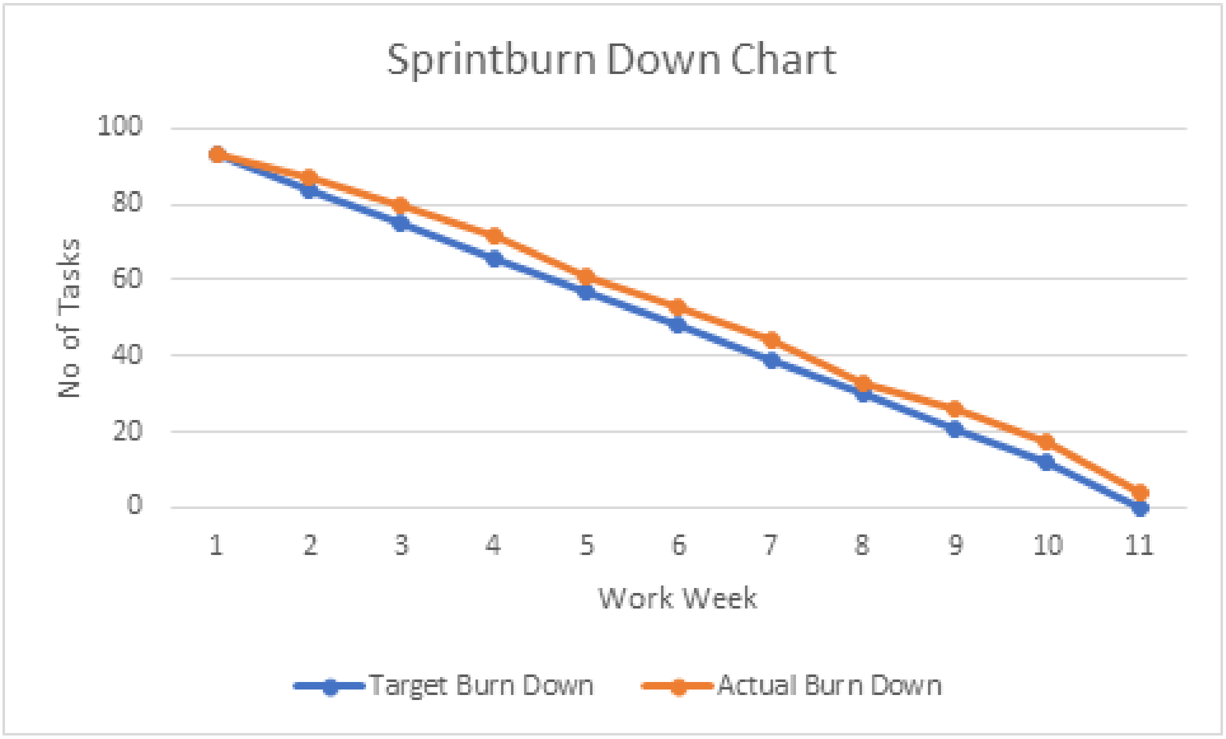


Figure 4 Sprint Burndown Chart for the study

V. MAJOR FINDINGS

The study analyzed two solutions, i.e., Hadoop Map Reduce and MPI for an aggregate query which calculates minimum taxi ride fare by month and year. The results of the analysis are as follows:

1) Hadoop Map Reduce results:

Analysis of performance for Hadoop Map Reduce was done using the NYC taxi trip records from Jan 2009 to Dec 2017. In total, the analytic experiments were performed using more than 200 GB which was disbursed among 108 files and each file contained taxi trip records for one month of data.

The performance of Map-Reduce on parameters under study using the cluster with the above properties are as follows:

1) Block size: It has been found that processing time decreases significantly when increasing the block size of data while maintaining input split size at a constant and using snappy compression. As shown in Figure 5, a block size of 128 MB decreases the processing time by 12% in comparison with a block size of 64 MB. Increase in availability of index by 50% times more for 128 MB block size in comparison with 64 MB block size may be the cause of such an improvement. However, this does not explain all the performance improvement observed. Apparently, a larger block size is always more favorable in practice.

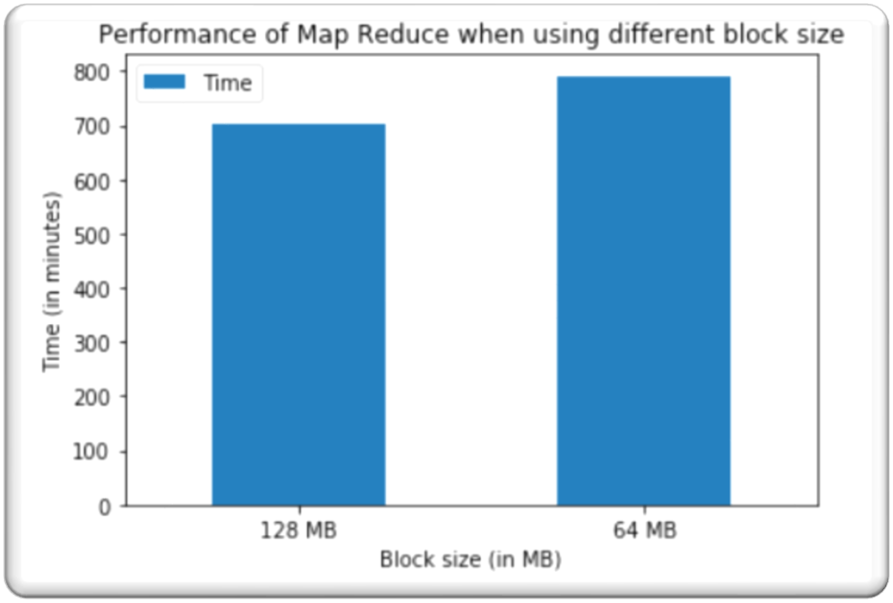


Figure 5 Performance of Map Reduce when using different block size

2) Input Split Size: When keeping block size constant and using snappy compression, it was discovered that a larger input split size performs better as shown in the Figure 6. Decreased processing time for larger input split size may be attributed to the fact that a smaller input split size has a significantly higher number of partitions in comparison with a larger input split size. For instance, a 32 MB split size had around 6000+ partitions in comparison with 2000+ partitions for a 128 MB split. Having many partitions is always disadvantageous because of the overhead associated with starting a mapper task is quite high. For instance, an overhead of 0.8 s was observed per task on average when using 128 MB block size, 128 MB input split size and snappy compression format.

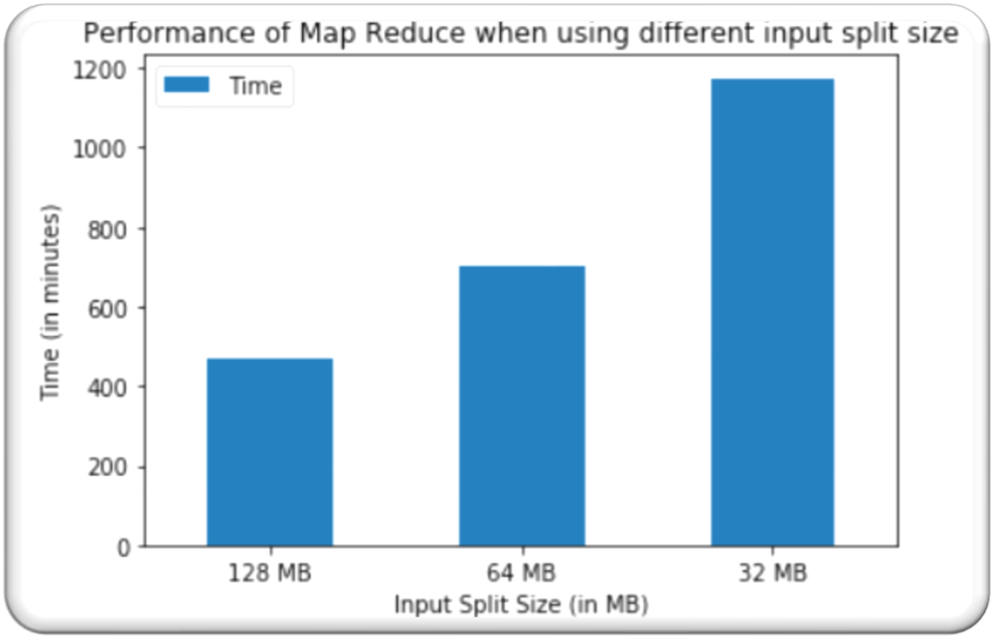


Figure 6 Performance of Map Reduce when using different input split size

3) Compression: Use of compression helps in intermediate stages of Map Reduce to decrease the time it takes in exchanging and processing data. Thus, reducing the overall process completion time. As shown in Figure 7, the use of snappy compression decreases the processing time significantly. On close observation, of each Map Reduce stage after mapper, we find significant improvement in every stage as demonstrated in the Figure 8. The decrease in processing time of intermediate Map-Reduce stages may be because compression decrease the size of data and decompression is relatively a faster operation.

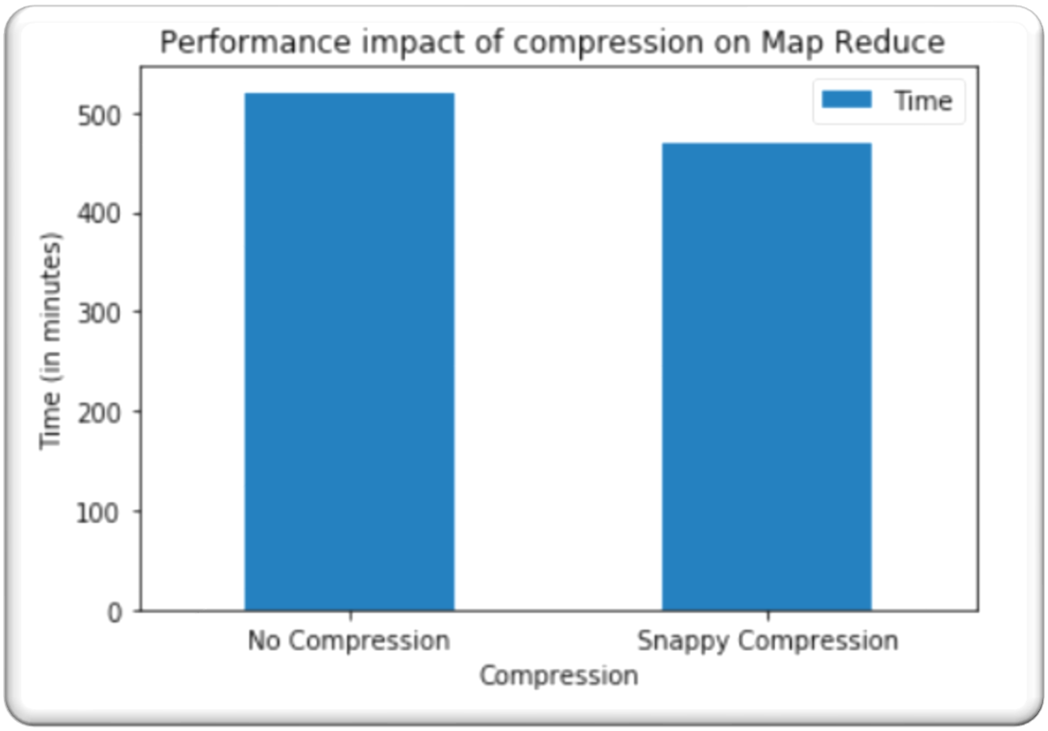


Figure 7 Performance impact of compression on the Map Reduce Application

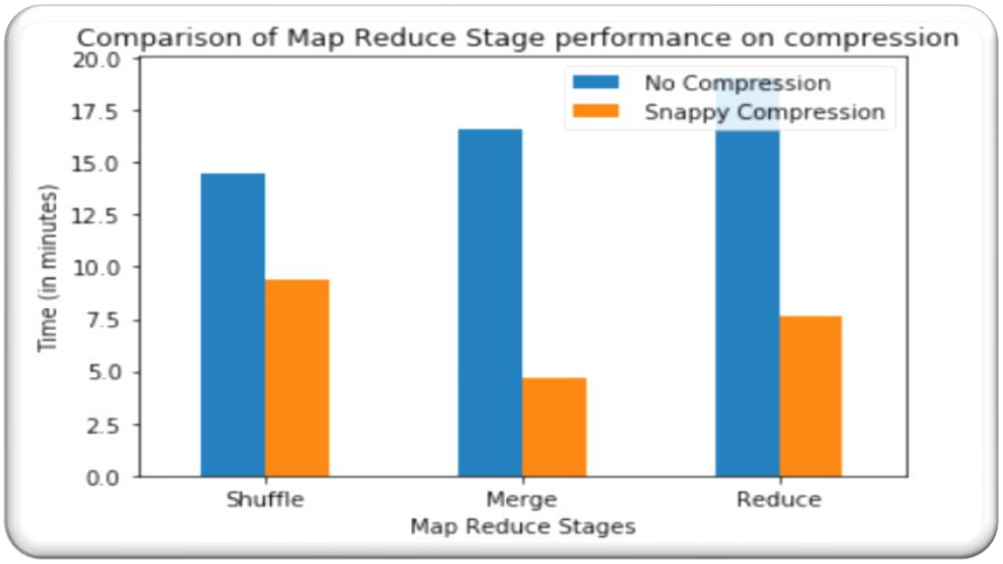


Figure 8 Performance comparison of compression on different stages of the Map Reduce Application

From, these experiments it may be concluded that Hadoop Map Reduce performs better on higher block size, larger input split size and using compression in intermediate stages of map reduce. Although, an upper bound on the block size may be established by using failure rates and performing a break-even analysis and an input split size up to a maximum of lower value of block size of Hadoop and amount of main memory available should be used. A decrease in performance when using an input split size more extensive than the block size of the data is in fact due to invoking a read from a system on the network when two or more consecutive data blocks are on a different system (i.e., loses data localization).

2) MPI: The analysis of MPI was carried out on Cedar cluster of the Shared Hierarchical Research Computing Network (SHARCNET). SHARCNET is a consortium of Canadian academic institutions who share a network of high-performance computers as part of Compute Canada. The Cedar cluster of SHARCNET provides a set of supercomputers to execute MPI workloads. Of various MPU libraries available for building the solution, the Intel MPI libraries were used for the execution of the workload on Cedar clusters. The first step in before executing the workload was to establish that the MPI clusters would allow reading the data file concurrently using multiple processors. This was necessary because the relevant MPI routines required for concurrent read might not be available. For the experiments, four nodes with one process per node were utilized to evaluate the performance of MPI using sequential vs. concurrent read.

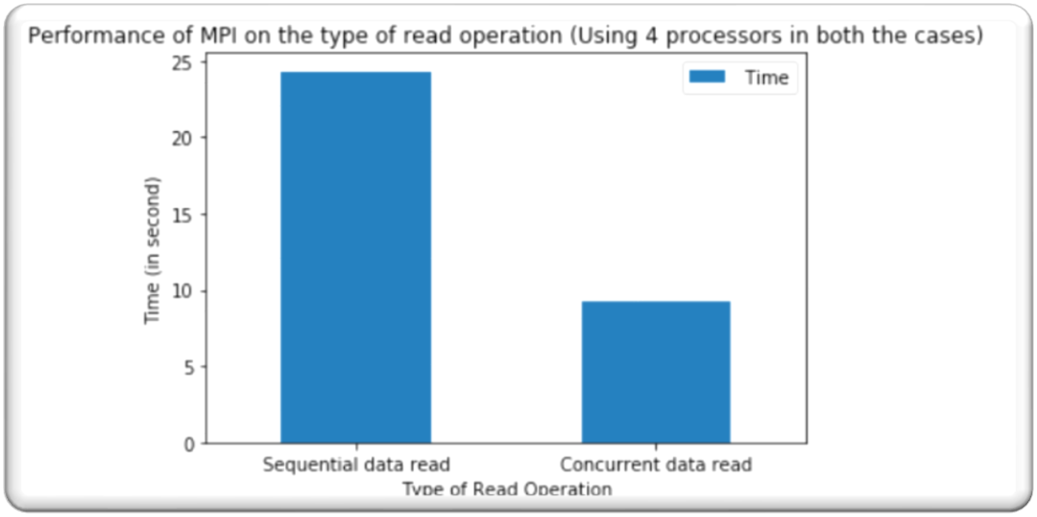


Figure 9 Performance of MPI on type of read operation (using 4 processors)

From the experiments, it is established that the MPI implementation with concurrent file read operation performs better than the sequential read operation as shown in Figure 9. The reason for the same is the Amdahl's law which states that in parallelization, if P is the proportion of a system or program that can be made parallel, and 1-P is the proportion that remains serial, then the maximum speedup that can be achieved using N number of processors is 1/((1-P)+(P/N). Here, as observed in Figure 3, the parallel region in the concurrent read operation is higher than that in sequential read version. Now, it takes on average around 24.2 seconds to read the data from the file and around 500 ms to process the data in sequential read version. Hence, in concurrent read version, it should take about 6.5 (i.e., time in serial/ no of processors) seconds to read the data when using four processors and then a processing time of 500 ms on average. Thus, theatrically the concurrent read should have been completed under 7 s. However, the concurrent read MPI version took on average 9.3 s for completion. A probable cause for the same may be the new overhead associated with the operation of reading the data file in parallel.

Furthermore, the Amdahl's law denotes an incomplete relationship between the speedup from parallelizing any computing problem and the presence of serial (non-parallelizable) portions [3]. To add to it, Amdahl's law focuses only on the theoretical performance scaling in an ideal case, rather than on the confounding factors which limit the scalability [3]. Alternatively, Gustafson's law may be used, but it also faces the same limitations as Amdahl's law since it does not consider other factors that can have implications on scalability. It is out of the scope of this study to perform a close analysis of the individual overheads associated with the MPI concurrent data reading processes. However, it can be fairly stated that the use of blocking synchronization by the data access routines used in the MPI implementation may be the most responsible candidate for the unexplained increase in processing time for the concurrent file read version of MPI. On further investigation into the MPI_File_Read_At_All collective, which was the data access routine utilized in the implementation, it was found that the given routine was using blocking I/O routines [9].

To tie a bow, Hadoop Map Reduce performs better when using higher block size, larger split size, and compression format, while MPI performs better with concurrent data read operation. These recommendations should be used apart from horizontal scaling to reduce the computation time for aggregate operations, in practice. Also, it should be noted that Hadoop Map Reduce provides fault tolerance as well as recovery, but MPI does not. In principle, a break-even analysis becomes essential using Map-Reduce task failure rate when making a decision regarding block size for Hadoop Map Reduce. The task failure rate is dependent on both the type and the size of hardware used for processing, storing data and networking. In general, commodity hardware's have more failure rate and static switches are more prone to flooding and hence their usage in implementing clusters for businesses should be discouraged.

VI. CHALLENGES & SKILLS LEARNED

A) Challenges:

Use of distributed & parallel programming platforms have been known to have required a considerable effort in getting the right libraries, debugging network problems, and implementation of data distribution logic. For the proposed work, we have two sets of solutions wherein the said efforts would be required. Both solutions need a different set of skills 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++ and debugging skills.

For the proposed work, the following challenges have been encountered and solved:

A. Challenges for Map Reduce Tests:

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. These issues were 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. Use of Linux worked well and passed the initial debugging tests for the snappy library and has not given any warnings or errors related to the Snappy library for four Map-Reduce job tests. These tests were carried out using Map Reduce standard word count program.
3) Memory-based 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. The same was handled by suppressing the test of virtual memory calculation by setting the "yarn.nodemanager.vmem-check-enabled" flag as false. The reason being the fact that enough virtual memory was not allocated as per the requirements (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 the future if a related situation arises in the future.

4) Storage size estimation: The amount of data to be analyzed is more than 200 GB for Map Reduce. However, an estimation of the size of the data during intermediate stages must be carried out to make sure that there is enough space for the intermediate data. This was computed by performing the Map Reduce on a smaller scale of data, and the results were extrapolated for the complete dataset.
5) Data ingestion problem: Since we have data in 108 files to be loaded into HDFS before being able to perform analysis, it becomes essential to have software to ingest the data. From the literature, we found multiple ingestion software's like Apache Gobblin, Apache NIFI, Apache Kafka, and others [26]. From a review of these solutions, it was found that most of them use Web HDFS to ingest data and hence it was decided that a layer of abstraction by using another software is not an optimal strategy to ingest the data [26]. Hence, data would be ingested by using the traditional approach using the HDFS command-line tools.
6) Challenges for storing logs: Job History service was used to store all the logs of Map Reduce jobs. The Job History server uses HDFS filesystem to a sink to store the Map Reduce logs [1]. The job history server enables to review logs for the job as well as all the tasks completed as a part of the job. Job history server helped in debugging the errands in implementations by enabling a quick review of all the necessary information from logs without going through all the log files. Also, the use of Job history server enabled for the bad records to be identified and stored separately for analysis. These bad records were then sent to the standard output and

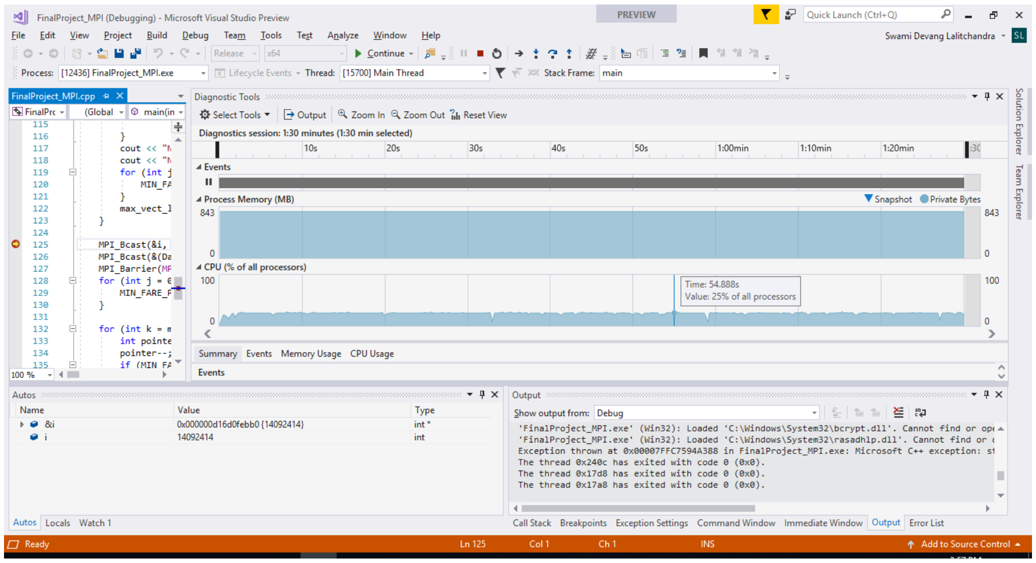
were finally visualized using the web interface of the Job history server was used to visualize these data points.

7) Selection of Log Store for Resource Manager: Resource manager is the entity inside the Hadoop software system that helps guarantee high availability (HA) and is also responsible for managing and maintaining the health of the cluster member. It uses the Hadoop FS Log store as a sink by default and may be changed to other solutions for logging like the Apache ZooKeeper and Apache Log4j log store [1]. However, Apache ZooKeeper & Log4j are recommended to run on a separate system [1]. Since this was not available, the default Hadoop HDFS FS store was used to guarantee HA.
8) Data Enrichment Issues: Data enrichment is defined as the task of converting data into the right format as a part of preprocessing. For example, fetching month and year from a timestamp. There are two types of enrichment, simple and complex. For the proposed study, simple enrichment was used to fetch year and month from the timestamp and to convert taxi ride fare from string to float. These values were used for processing only if they were consistent with the regular expressions. Data points that do not meet the requirements of regular expressions are bad records.

B. Challenges for MPI test:

Debugging challenges especially for unsafe codes.

1) Memory issues: The compute nodes in Cedar cluster required as input the amount of main memory to be reserved for the program. Thus, an accurate estimate of main memory usage was required before carrying out analysis. For the same, Microsoft Visual Studio was configured to use Intel MKL MPI library using Intel Cluster XE edition of libraries, and the behavior of the program was monitored to be able to establish a benchmark for the primary memory consumption. It was found that the program uses a little more than 800 MB of space for a four-node processing job as demonstrates in Figure 10. This process also helped establish a benchmark for computation time, which was around 80 s. These experiments were carried out on the local system with a clock rate of 2.1 GHz, 8 GB main memory, and Windows 10 operating system.


FinalProject_MPI (Debugging) - Microsoft Visual Studio Preview
PREVIEW
Quick Launch (Ctrl+Q)
File Edit View Project Build Debug Team Tools Test Analyze Window Help
Swami Devang Lalitchandra
Continue
Process: [12436] FinalProject_MPI.exe
Lifecycle Events
Thread: [15700] Main Thread
Stack Frame: main
FinalProject_MPI.cpp
Diagnostic Tools
Select Tools
Output
Zoom In
Zoom Out
Reset View
Diagnostics session: 1:30 minutes (1:30 min selected)
Events
Process Memory (MB)
Snapshot
Private Bytes
CPU (% of all processors)
Time: 54.888s
Value: 25% of all processors
Summary Events Memory Usage CPU Usage
Solution Explorer
Team Explorer
Ready
Ln 125
Col 1
Ch 1
INS
Add to Source Control

```
        }
        cout << "N
        cout << "N
        for (int j
            MIN_FA
        }
        max_vect_l
    }

    MPI_Bcast(&i,
    MPI_Bcast(&(Da
    MPI_Barrier(MF
    for (int j = 0
        MIN_FARE_F
    }

    for (int k = m
        int pointe
        pointer--;
        if (MIN_FA
```

Autos

Name	Value	Type
&i	0x000000d16d0febb0 {14092414}	int *
i	14092414	int

Output

Show output from: Debug

```
'FinalProject_MPI.exe' (Win32): Loaded 'C:\Windows\System32\bcrypt.dll'. Cannot find or ope
'FinalProject_MPI.exe' (Win32): Loaded 'C:\Windows\System32\rasadhlp.dll'. Cannot find or 
Exception thrown at 0x00007FFC7594A388 in FinalProject_MPI.exe: Microsoft C++ exception: s
The thread 0x240c has exited with code 0 (0x0).
The thread 0x17d8 has exited with code 0 (0x0).
The thread 0x17a8 has exited with code 0 (0x0).
```

Figure 10 MPI Debugging & Memory test using Microsoft Vision

2) Algorithm for chunking data files: The conventional approach to breaking a CSV file for analysis is by logically splitting the file based on the number of lines in the dataset. However, calculating the number of lines from the data would require going through the data file once, and hence would degrade the performance that the study thrives to achieve. The other option is to split based on an equal size of data when running through the same sequentially. However, with this tactic, a single row of data may be split across two processors which is demonstrated in Figure 11. To overcome this problem, an overlap was used. An overlap allows repetition of a portion of data being analyzed between two processes. This allowed redundancy of few records among two processors to ensure that all the data points are analyzed. After, multiple tests it was found that an overlap of 250 characters works as expected.

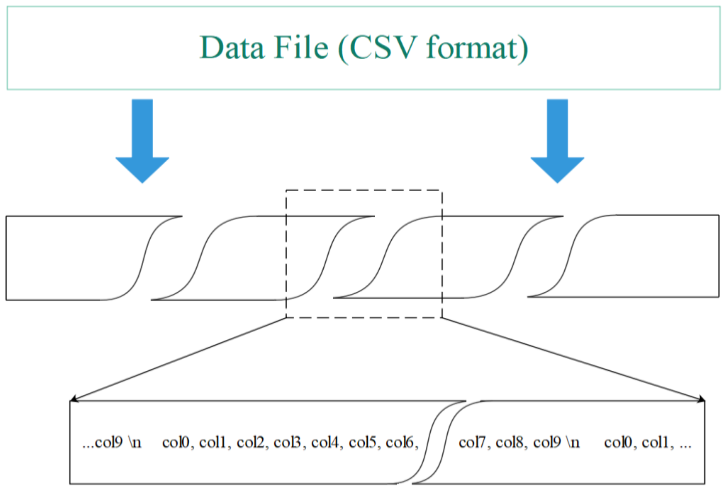
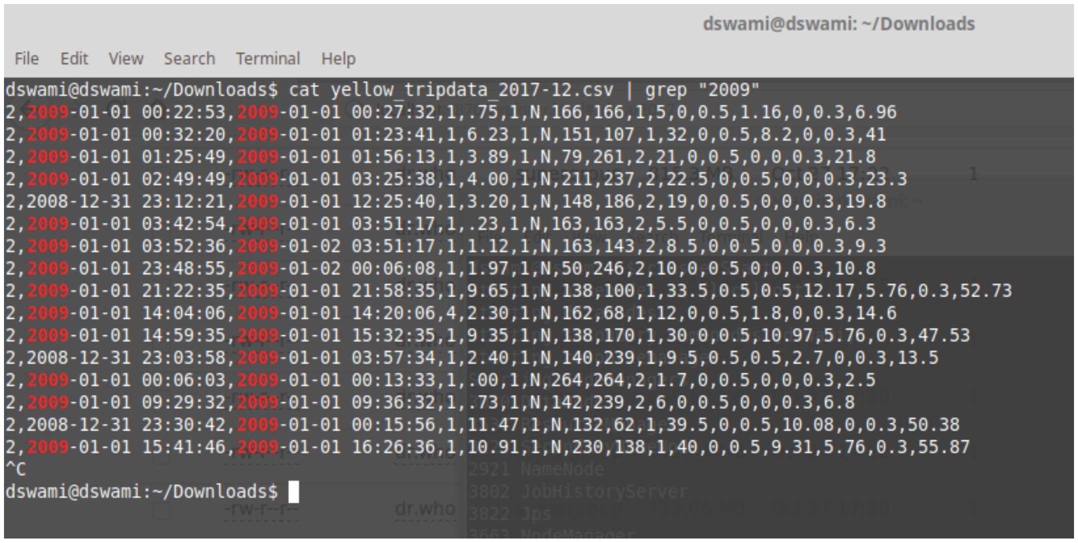


Figure 11 Problem of splitting data file based on index rather than number of records

3) Due to limitations of the penalty imposed by SHARCNET on long-running jobs [5], the dataset was cut down to a little less than 1 GB for processing. If time permits, analysis with even larger dataset size may be carried out. The number of data points for the MPI program was not a multiple of the number of processors hence changes were made in MPI routines to adjust to the new challenge. The MPI routine MPI_Scatter was changed to MPI_Scatterv to accommodate these new challenges. MPI routines MPI_Bcast and MPI_Scatter were tested for performance to be able to select them for optimization. It was found that MPI_Scatter performs better than MPI_Bcast and hence the same was used.
4) Data Enrichment Issues: MPI implementation also had to perform simple enrichment to fetch year and month from the timestamp and to convert taxi ride fare from string to float like Map Reduce implementation. However, to check the quality of the data instead of regular expressions, try-catch blocks were used with flags to mark the bad records. Use of try-catch blocks was encouraged over regular expressions because the new string to float conversion libraries were more efficient and could handle the necessary errands. With Java implementation, this was tricky because unlike C++ routines, java methods throw a parse exception when values cannot be converted into float and then it breaks execution of the program throwing a parse exception.

C. Data Quality Challenges:

On analysis of the data, it was found that the file containing records for December 2017, had records from 2003, 2008, 2009, and 2018. These bad data points for 2009 in December 2017 record file are shown in Figure 12. This showed that the data obtained from the NYC Taxi Cab Service dataset had errands running through them. It denotes a probable bug in the data preprocessing techniques employed by NYC TLC. The organization was informed about the same, and the issue had been forwarded to the data quality team. It was recommended by them to remove those records as erroneous and proceed further with the analysis.



```
dswami@dswami:~/Downloads$ cat yellow_tripdata_2017-12.csv | grep "2009"
2,2009-01-01 00:22:53,2009-01-01 00:27:32,1,.75,1,N,166,166,1,5,0,0.5,1.16,0,0.3,6.96
2,2009-01-01 00:32:20,2009-01-01 01:23:41,1,6.23,1,N,151,107,1,32,0,0.5,8.2,0,0.3,41
2,2009-01-01 01:25:49,2009-01-01 01:56:13,1,3.89,1,N,79,261,2,21,0,0.5,0,0,0.3,21.8
2,2009-01-01 02:49:49,2009-01-01 03:25:38,1,4.00,1,N,211,237,2,22.5,0,0.5,0,0,0.3,23.3
2,2008-12-31 23:12:21,2009-01-01 12:25:40,1,3.20,1,N,148,186,2,19,0,0.5,0,0,0.3,19.8
2,2009-01-01 03:42:54,2009-01-01 03:51:17,1,.23,1,N,163,163,2,5.5,0,0.5,0,0,0.3,6.3
2,2009-01-01 03:52:36,2009-01-02 03:51:17,1,1.12,1,N,163,143,2,8.5,0,0.5,0,0,0.3,9.3
2,2009-01-01 23:48:55,2009-01-02 00:06:08,1,1.97,1,N,50,246,2,10,0,0.5,0,0,0.3,10.8
2,2009-01-01 21:22:35,2009-01-01 21:58:35,1,9.65,1,N,138,100,1,33.5,0.5,0.5,12.17,5.76,0.3,52.73
2,2009-01-01 14:04:06,2009-01-01 14:20:06,4,2.30,1,N,162,68,1,12,0,0.5,1.8,0,0.3,14.6
2,2009-01-01 14:59:35,2009-01-01 15:32:35,1,9.35,1,N,138,170,1,30,0,0.5,10.97,5.76,0.3,47.53
2,2008-12-31 23:03:58,2009-01-01 03:57:34,1,2.40,1,N,140,239,1,9.5,0.5,0.5,2.7,0,0.3,13.5
2,2009-01-01 00:06:03,2009-01-01 00:13:33,1,.00,1,N,264,264,2,1.7,0,0.5,0,0,0.3,2.5
2,2009-01-01 09:29:32,2009-01-01 09:36:32,1,.73,1,N,142,239,2,6,0,0.5,0,0,0.3,6.8
2,2008-12-31 23:30:42,2009-01-01 00:15:56,1,11.47,1,N,132,62,1,39.5,0,0.5,10.08,0,0.3,50.38
2,2009-01-01 15:41:46,2009-01-01 16:26:36,1,10.91,1,N,230,138,1,40,0,0.5,9.31,5.76,0.3,55.87
^C
dswami@dswami:~/Downloads$ 
```

Figure 12 Bad Records from 2009 in the data file for Jan 2018.

B) SKILLS LEARNED:

Conducting this study provided technical expertise for overcoming the challenges when performing computation over large-scale data using distributed and parallel systems. It has also tested our capabilities to think out-of-the-box and debugging the solutions developed. Regular weekly meetings helped in the cross-fertilization of thoughts to improvise. To add to it, skills in Software Engineering process was developed to overcome challenges to design, coding, and, testing.

Moreover, skills were obtained in using Agile software development framework SCRUM. Use of GitHub repository throughout project gave an opportunity to get hands-on using version control system. GitHub also provided an opportunity to use its project management framework to organize tasks. This was quite similar but is not a replacement to the SCRUM framework of Agile software development methodology.

VII. CONCLUSION & FUTURE WORKS

Aggregation is an operation that summarises a set of data points and is a vital operation that allows for a quick review in a short span of time. However, with the advent of cyber-physical systems at the beginning of 21st-century massive amount of data was generated. This data was often referred to as big data since it required distributed computing systems for processing. Since the amount of data was too large, aggregation processes took a long time and were responsible for the delay in the decision-making. In this study, a focus on reducing the processing time for aggregation operations on Hadoop Map Reduce and MPI has been presented. It has been found that using a large block size, a considerable input split size, and using compression helps to decrease the processing time for aggregation operations when using Hadoop Map Reduce, while with MPI reading the data file concurrently reduces the processing time. To add to it, MPI performs significantly better compared to Hadoop Map Reduce but the time to develop a software solution in MPI is more challenging and takes more time compared to Hadoop Map Reduce.

Moreover, MPI also does not provide fault tolerance like Hadoop Map Reduce. In the future, the study may be expanded by testing comparing various compression codecs or perform more experiments to uncover why a larger block size works better when keeping input split size constant for Hadoop Map Reduce. For MPI, a comparative performance study on MPI_Reduce vs. MPI_Allgather may be performed to increase the scope of this study. Also, use of non-blocking I/O routines may be tested to visualize the performance improvement when using concurrent file read routines.

VIII. BIBLIOGRAPHY

1) Apache Hadoop, Hadoop Documentation, https://hadoop.apache.org/docs
2) Apache Log4j, Log4j Documentation, https://logging.apache.org/log4j/1.2/manual.html
3) Amdahl's Law, Gustafson's Trend, and the Performance Limits of Parallel Applications. (2015, January 01). Retrieved from https://software.intel.com/en-us/articles/amdahls-law-gustafsons-trend-and-the-performance-limits-of-parallel-applications
4) Bonnet, L., Laurent, A., Sala, M., Laurent, B., & Sicard, N. (2011, August). Reduce, you say: What nosql can do for data aggregation and bi in large repositories. In Database and Expert Systems Applications (DEXA), 2011 22nd International Workshop on (pp. 483-488). IEEE.
5) Cedar - Compute Canada Doc. (2018, November 16). Retrieved from https://docs.computecanada.ca/wiki/Cedar#Choosing_a_node_type
6) Chen, C. P., & Zhang, C. Y. (2014). Data-intensive applications, challenges, techniques and technologies: A survey on Big Data. Information Sciences, 275, 314-347.
7) Cervone, H. Frank. "Understanding agile project management methods using Scrum." OCLC Systems & Services: International digital library perspectives 27.1 (2011): 18-22.
8) Collier, Ken W. (2011). Agile Analytics: A Value-Driven Approach to Business Intelligence and Data Warehousing. Pearson Education. pp. 121 ff. ISBN 9780321669544. "What is a self-organizing team?"
9) Data Access Routines. (2018, November 29). Retrieved from https://www.mpi-forum.org/docs/mpi-2.2/mpi22-report/node276.htm
10) Dbengines. Clusterpoint System Properties. https://db-engines.com/en/system/Clusterpoint
11) Dede, E., Govindaraju, M., Gunter, D., Canon, R. S., & Ramakrishnan, L. (2013, June). Performance evaluation of a mongodb and hadoop platform for scientific data analysis. In Proceedings of the 4th ACM workshop on Scientific cloud computing (pp. 13-20). ACM.

12) Dittrich, J., & Quiané-Ruiz, J. A. (2012). Efficient big data processing in Hadoop MapReduce. Proceedings of the VLDB Endowment, 5(12), 2014-2015.
13) 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.
14) G.M. Amdahl, Validity of the single-processor approach to achieving large scale computing capabilities. In AFIPS Conference Proceedings vol. 30 (Atlantic City, N.J., Apr. 18-20). AFIPS Press, Reston, Va., 1967, pp. 483-485.
15) Intel, Tutorial: Using MPI Tuner for Intel® MPI Library on Linux OS. https://software.intel.com/en-us/download/mpi-tuner-tutorial-lin-pdf
16) J. Dean , S. Ghemawat, MapReduce: simplified data processing on large clusters. in: OSDI04: sixth Symposium on Operating System Design and Implementation, 2004.
17) Jeff Squyres. Open MPI: Overview / Architecture. https://www.open-mpi.org/video/internals/Cisco_JeffSquyres-1up.pdf
18) 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.
19) John Gustafson, Reevaluating Amdahl’s Law, Communications of the ACM 31(5), 1988; reposted at http://www.scl.ameslab.gov/Publications/Gus/AmdahlsLaw/Amdahls.html
20) Jonathan Dursi (Stackoverflow User), MPI io reading file by processes equally by line (not by chunk size). https://stackoverflow.com/questions/13327127/mpi-io-reading-file-by-processes-equally-by-line-not-by-chunk-size
21) 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.
22) Kshemkalyani, A. D., & Singhal, M. (2011). Distributed computing: principles, algorithms, and systems. Cambridge University Press.
23) 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.

24) Lessons Learned from Cloudera's Hadoop Developer Training Course - Cloudera Engineering Blog. (2011, January 26). Retrieved from https://blog.cloudera.com/blog/2011/01/lessons-learned-from-clouderas-hadoop-developer-training-course

25) 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.

26) PAT Research. Top 18 Data ingestion tools. https://www.predictiveanalyticstoday.com/data-ingestion-tools/

27) MPICH.org, Man pages for MPICH. https://www.mpich.org/static/docs/v3.1/

28) MPI Forum, Examples using MPI_SCATTER, MPI_SCATTERV. https://www.mpi-forum.org/docs/mpi-1.1/mpi-11-html/node72.html

29) NYC Taxi & Limousine Commission, New York Taxi Trip Records, http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml

30) Reyes-Ortiz, Jorge L., Luca Oneto, and Davide Anguita. "Big data analytics in the cloud: Spark on hadoop vs mpi/openmp on beowulf." Procedia Computer Science 53 (2015): 121-130.

31) Sandia Labs, Map reduce, http://mapreduce.sandia.gov/

32) Sharcnet HPC. (Sept, 18, 2018). Concurrent File I/O by Multiple Processes. https://www.youtube.com/watch?v=3s-dBEopfwQ

33) Sutherland, J. (2001). Inventing and Reinventing SCRUM in five Companies. Sur le site officiel de l'alliance agile.

34) The Architecture of Open Source Applications (Volume 2): Open MPI. (2016, June 29). Retrieved from https://www.aosabook.org/en/openmpi.html

35) Verheyen, Gunther. "Scrum: Framework, not methodology". Retrieved February 24, 2016.

36) Wang, L., Tao, J., Ranjan, R., Marten, H., Streit, A., Chen, J., & Chen, D. (2013). G-Hadoop: MapReduce across distributed data centers for data-intensive computing. Future Generation Computer Systems, 29(3), 739–750. doi: 10.1016/j.future.2012.09.001

37) Wes Kendall, Launching an Amazon EC2 MPI Cluster. http://mpitutorial.com/tutorials/launching-an-amazon-ec2-mpi-cluster/

38) Wes Kendall, A Comprehensive MPI Tutorial Resource. http://mpitutorial.com/

39) W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. Parallel computing, 22(6):789–828, 1996.

40) Steven J.PlimptonKaren D.Devine, MapReduce in MPI for Large-scale graph algorithms. https://www.sciencedirect.com/science/article/pii/S0167819111000172

41) Schwaber, Ken (February 1, 2004). Agile Project Management with Scrum. Microsoft Press. ISBN 978-0-7356-1993-7.

42) Torsten Hoefler, Andrew Lumsdaine, Jack Dongarra, Towards Efficient MapReduce Using MPI. https://link.springer.com/chapter/10.1007/978-3-642-03770-2_30