Performance Evaluation of Apache Spark Vs MPI: A Practical Case Study on Twitter Sentiment Analysis

1,2 Deepa S Kumar and 3,4 M Abdul Rahman

1 Research Scholar, Karpagam Academy of Higher Education, Karpagam University, Tamilnadu (Dist), Coimbatore, India
2 Department of Computer Science and Engineering, College of Engineering Munnar, Kerala, India
3 Pro-vice Chancellor, APJ Abdul Kalam Technological University, Kerala, India
4 Research guide, Karpagam Academy of Higher Education, Karpagam University, Tamilnadu(Dist), Coimbatore, India

Abstract: The advent of various processing frameworks which happens under big data technologies is due to tremendous dataset size and its complexity. The speed of execution was much higher with High Performance computing frameworks rather than big data processing frameworks. As majority of the jobs under big data are mostly data intensive rather than computation intensive, the High Performance Computing paradigms were not been used in big data processing. This paper reviews two distributed and parallel computing frameworks: Apache Spark and MPI. Sentiment analysis on twitter data is chosen as a test case application for benchmarking and implemented on Scala programming for spark processing and in C++ for MPI. Experiments were conducted on Google cloud virtual machines for three data set sizes, 100 GB, 500 GB and 1 TB to compare the execution times. Results shown that MPI outperforms Apache Spark in parallel and distributed cluster computing environments and hence the higher performance of MPI can be exploited in big data applications for improving speedups.

Keywords: Big Data, High Performance Computing, Apache Spark, MPI, Sentiment Analysis, Scala Programming, Cluster Computing

Introduction

Processing of huge volume of data in a variety of forms is one of the major challenges in the last two decades. Several parallel computing architectures were involved in shared memory, multi processor, multi-core processing, distributed processing, cluster computing, massively parallel processing, grid computing and specialized parallel computers like FPGA implementation etc based on the level of hardware support (Sliwinski and Kang, 2017). The parallel computers are designed for High Performance Computing (HPC). In contrast, another type of parallelism can be observed in High Throughput Computing (HTC). HPC provides computational resources for working with large datasets and mainly focus on how fast the computations can be done in parallel by exhibiting high computing power in a very short span of time. Whereas HTC is looking for how many tasks can be completed over a long period of time.

Over the last few decades, processing of tremendous volume of data and its analytics have become one of the big challenges in the field of computing which leads to the concept of Big data and the associated processing. In data limited problems, data transfer time is more significant than processing time and in computationally limited problems, processing is faster than data transfer time (Sliwinski and Kang, 2017). Hadoop integrates processing and data and introduces application-level scheduling to facilitate heterogeneous application workloads and high cluster utilization (Jha et al., 2014). Hadoop implementation of Big data is dealing with the data-intensive task execution, with the data storage and job scheduling. Hadoop Map reduce, Apache Spark, Storm etc are the big data processing frameworks. Among the frameworks, Spark is faster compared to Map reduce for batch processing.

Paper compared and evaluated execution times of existing big data processing framework, Apache Spark and the High Performance Computing Library, MPI.
paper explored a detailed evaluation on the parallel execution of Sentiment Analysis along with the additional delay metrics and the workflow using Directed Acyclic Graph (DAG) during spark processing in both frameworks.

Comparison on different big data processing frameworks and High Performance Computing frameworks had been presented in the literature.

Related Work

Zaharia et al. (2010) pointed out that Map reduce and its variants had been highly successful in implementing large-scale data-intensive applications on commodity clusters with the added features such as scalability and fault tolerance. But it is not suitable for iterative and real time applications. They focused on one such class of applications: Those that reuse a working set of data across multiple parallel operations. This includes many iterative machine learning algorithms, as well as interactive data analysis tools. They proposed a new framework called Spark that supports these applications while retaining the scalability and fault tolerance of Map reduce. To achieve these goals, Spark introduces an abstraction called Resilient Distributed Datasets (RDDs). An RDD is a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. They have concluded that Spark outperforms Hadoop by 10 x in iterative machine learning jobs and can be used to interactively query datasets in sub-second response time.

Jha et al. (2014) discussed various problems associated with the processing large amounts of data that require managing large-scale data distribution, co-placement and scheduling of data with compute prominent paradigms for data-intensive applications in the high-performance computing and the Apache-Hadoop paradigm. Their micro-benchmark shown that MPI outperforms the Hadoop-based implementation and also pointed out that the Hadoop framework, Spark has improved performance significantly by adopting techniques in HPC and maintains a very high and accessible level of abstraction and a coupling to resources managed in HDFS or RDD chunk size. In HPC applications, communication operations and application-specific files lack a common runtime system for efficiently processing of data objects, but speedup performance with the usage of communication primitives in HPC.

Reyes-Ortiz et al. (2015) made the first comparative study between Apache Spark and MPI/OpenMP by exploring the two distributed computing frameworks, implemented on commodity cluster architectures and evaluated two supervised machine learning iterative algorithms: KNN and SVM. Investigations shown with a conclusion that MPI/OpenMP outperforms Spark by more than one order of magnitude in terms of processing speed and provides more consistent performance. Unlike MPI, Spark is integrated with data management infrastructure known as HDFS and hence it automatically handles node failure and data replication management.

Kang et al. (2015) analysed two benchmark problems, all-pair-shortest-job and Join problem and the parallel programs had been implemented using OpenMP, MPI and Map reduce, respectively. They arrived at a conclusion that MPI could be the best choice if the data size is moderate and the problem is computation-intensive and if the data size is large, then Map reduce will performs better, provided the tasks are non-iterative. Map reduce programs took considerable time for the problems requiring much iteration, like all-pairs-shortest-path problem. MPI was a high performance computing paradigm which allows rapid message passing in a distributed system and much more efficient than Map reduce and hence MPI is a good choice when a program is needed to be executed in parallel and distributed manner, but with complicated coordination among processes.

Gittens et al. (2016) compared the performance of Spark and C+MPI frameworks by exploring relational algebra with scientific dataset of several TBs and have detailed observations on various delays and overheads introduced while performing in both the frameworks. They have pointed out that Spark has more delays when comparing with HPC implementation on C+MPI. Spark execution time delays include task start delay, task scheduling delay, Lazy Evaluation nature, RDD partitioning, persist method to store the intermediate RDDs and synchronization between tasks among RDDs, straggler effect, i.e., the idle time when the a finished task has to wait for other tasks to finish and more over I/O delay etc. As far as MPI is concerned, synchronization among tasks is not required for non blocking operations, but mostly suffers from I/O access delay when dataset size is very large and scalability is another major issue in data intensive applications. Hence they have suggested improving the I/O management and scalability of MPI framework in case of terabyte scale data parallel processing and incorporating into Apache Spark framework to have the ideal performance.

Kanavos et al. (2017) implemented the algorithm Sentiment Analysis on Twitter Data in Map reduce and in Apache Spark framework and proposed a classification method of sentiment types in a parallel and distributed manner. They have pointed out that the Map reduce implementation with the Bloom filters for their analysis had minimized the storage space required by a fraction between 15-20% and with Spark implementation, the
Bloom filters manage to marginally minimize the storage space required for the feature vectors up to 9% only.

Anderson et al. (2017) aimed to bridge the performance gap between MPI and Apache Spark. In many analytics operations, the implementations in MPI outperforms an order of magnitude to Apache Spark, but the benefits of the Spark ecosystem such as availability, productivity and fault tolerance are to be retained. The authors proposed a system for integrating MPI with Spark and analyse the costs and benefits in machine learning applications. They have shown that offloading computation to an MPI environment from within Spark provides 3.1-17.7\times speedups, including all of the overheads. This opens up a new approach to reuse existing MPI libraries in Spark with little effort.

Several implementations on different applications such as SVM and KNN, all-pair-shortest-job and joining problem, large-scale matrix multiplication etc have been reviewed in the literature in the Hadoop processing frameworks -Map reduce and Apache Spark. In batch processing kind of applications, Map reduce performs better due to the integrated storage architecture for Hadoop-HDFS (Plimpton and Devine, 2011). But it’s processing results in degradation of performance in situations like iterative jobs and in interactive applications due to the lack of In-Memory Computing capability. Hence a new processing framework was evolved known as Apache Spark, which relies on the core data structure RDD and it speed up 100\times faster than Map reduce in In-Memory Computing and 10\times faster when the data is stored on disk (https://www.edureka.co/blog/apache-spark-vs-hadoop-mapreduce). Most attractive feature of Spark is that it supports real-time streaming and distributed processing. Hence Spark was the choice for faster computing. Spark provides libraries for graph analytics and machine learning libraries to support for the respective applications (Satish et al., 2014). When the data size is very high, recent studies have shown that load time is high in a number of Spark applications (Ousterhout et al., 2015). Raveendran et al. (2011) suggested adding more computing performance by utilizing the HPC framework, MPI after evaluating the MPI-based communication mechanism. Spark speedups well compared to Hadoop, but its performance is not comparable with the High Performance Computing (HPC) frameworks, such as MPI (Forum, 2015). The Spark implementation has significant overhead to scheduling tasks and managing the resources and start delay etc. when comparing with MPI parallel task implementation (Gittens et al., 2016). Thrill (Axmann et al., 2016) is a project which aims at making a kind of data processing system in C++ and using MPI for communication. The evaluation results clearly indicates that MPI based implementations outperforms several order of magnitude than with Spark implementation like SVM and KNN (Reyes-Ortiz et al., 2015) and k-means clustering (Jha et al., 2014). The major difference with Spark framework and MPI frameworks is that the big data Hadoop frameworks tightly integrates the storage framework and the processing framework along with the coordination of jobs and resource allocation. Whereas MPI primitives are being used explicitly to perform all the associated tasks like storage, coordination, distribution and compute and there is no integrated storage in MPI. Hence even though the MPI compute outperforms Spark, MPI is not well suited to data intensive applications. Michael Anderson et al. (2017) proposed a system for integrating MPI with Spark and tested on machine learning applications and proved that the integrated Spark+MPI shows excellent performance. They have open up a new approach of using MPI in data intensive applications too. Fenix (Gamell et al., 2014), FTMP (Fagg and Dongarra, 2000), H2O.ai (2016), Deeplearning4j (2016), SWAT (Grossman and Sarkar, 2016) were the several implementations from 2000 onwards, mainly aimed to improve the performance of Spark using MPI primitives or building a Spark like data processing system or bridging the gap between Spark processing and MPI processing.

Parallel and Distributed Cluster Computing Frameworks

In this section we present relevant details of big data processing framework, Apache Spark and the High Performance Computing paradigm, known as Message Passing Interface in the context of big data processing.

Apache Spark

Spark is a faster cluster computing framework that executes the applications much faster than Hadoop. This is achieved by keeping data in memory. Spark enables sharing of data across multiple Map reduce steps in interactive applications (Khanam and Agarwal, 2015). Spark provides a new storage primitive called Resilient Distributed Datasets (RDD). RDDs can read and written up to 40\times times faster than the distributed file system (Zaharia et al., 2012). Resilient Distributed Dataset (RDD) is a distributed, inflexible and fault tolerant memory abstraction overcomes the problem of sharing data across multiple Map reduce steps that is required in multi pass and interactive applications. Memory is logically partitioned into RDDs with a set of elements and a permissible set of operations applied. The operations on RDDs are either transformations or actions, in which transformations are RDDs itself. The actions follow a Lazy Evaluation strategy (LE strategy) due to which the computations are
performed only during actions and the computations are not cached in RDDs. Hence there should be separate persist methods to store the computed actions. Spark goals are fault tolerant, data replication and provides a better storage infrastructure.

**Message Passing Interface (MPI)**

MPI is a message passing library specification which defines a message passing model for parallel and distributed programming (Gropp et al., 1998). MPI is a standard created to facilitate the transport of data between the distributed processing elements (Jha et al., 2014). MPI extents from a serial program executed by a single process within a single machine to multiple processes distributed across many cluster of nodes. MPI utilize the resources of all of those nodes at once by facilitating the communication between them across the interconnecting network. MPI methods include optimized communication primitives such as send, receive, scatter, gather etc. MPI implementations provide extremely high performance in terms of extremely low latency and network injection rate for short messages, maximum bandwidth and maintain a balance of low resource utilization. MPI is an HPC vendor independent implementation of the parallel programming environment and several implementations have been made such as OpenMPI, MPICH and GridMPI (Diaz et al., 2012). MPI main goals are high performance, scalability and portability.

From the literature review, it is clear that the Spark is slower compared to native implementations written with high performance computing in MPI. MPI supports a wide variety of high performance computing platforms and environments. But there is no specific storage architecture relies on MPI as in Spark. Figure 1 presents the traditional HPC processing architecture, in which data and compute nodes are separated and compute nodes are tightly interconnected for low latency during computation, but data is to be accessed from separate storage nodes. MPI takes the data from Network File System (NFS), Cassandra or any other file system including HDFS. MPI is at the peak edge of high performance computing. Case study shows that MPI is 4.6-10.2× faster than Spark on large matrix factorizations on an HPC cluster with 100 compute nodes (Gittens et al., 2016).

Experiments on sentiment analysis on twitter data for different sizes are being conducted to ascertain the performance parameters like execution time, CPU and memory utilization in the existing big data framework, Apache Spark and the High Performance Computing Library, MPICH2. Theoretical studies shown that, unlike Spark, MPI lacks integrated storage architecture. If MPI overcomes the limitation, it can be one of the next generation big data processing technologies.

![Fig. 1: Storage nodes and Compute nodes separated in traditional HPC](image_url)
Sentiment Analysis on Spark Vs MPI

In order to evaluate the performance of the distributed and cluster computing frameworks-spark and MPI, Sentiment analysis on Twitter data is being chosen as an application. Sentiment Analysis is the process of deriving the opinion by computations from the data. It is otherwise called as opinion mining which determines whether the content is positive, negative or neutral.

Sentiment Analysis

Sentiment analysis is being done to understand opinion in a set of documents. The analysis on our application undergoes the following steps, shown as pseudo code. The application accepts a blocks of lines of tweets from the storage to tasks and processed in parallel, then read next block of data until the whole data is being processed. The intermediate results in each iteration are an array of five numbers [very_bad_tweets, bad_tweet, normal, good_tweet and very_good_tweet] and outputs among the set {highly negative, negative, neutral, positive and highly positive}. Then count the number of positive words to get the positive score, count the negative words to get the negative score of the tweets and this score is clamped into a range of five integers to map towards the category and finally outputs on each category.

For spark implementations, programs were written in Scala. MPI program was written in C++. For the analysis of execution performance and the different delays incurred during processing, both programs were executed on 7 GB, 100 GB, 500 GB and 1TB dataset sizes for various attempts. Hadoop HDFS and YARN were installed over the cluster of nodes.

Pseudo Codes for the Implementation of Spark and MPI

The algorithms for the implementation of sentiment analysis in spark processing are shown in algorithm 1 and the analyzer function in algorithm 2.

Algorithm 1: Sentiment Analysis on Twitter Data on Spark Processing

```
INPUT:    input filename inputFile
          Positive words file positiveWord
          Negative words file negativeWord
OUTPUT:   RDD textFile (PositiveList, NegativeList)
1 Open file positiveWord
2 Load contents of positiveWord to PositiveList
3 close file positiveWord
4 open file negativeWord
5 Load contents of negativeWord to NegativeList
6 close file negativeWord
7 Create sparkcontext sc
8 Open inputFile using sc and load the data to RDD textfile
9 Call filter function to retain only sentences starting with “W” and store it in RDD textfile.
10 Call map function analyzer and store result in RDD textfile with arguments PositiveList, NegativeList
11 Call reducebykey on RDD textfile
12 return
```

Algorithm 2: Analyzer (Spark Processing)

```
Input: Vector PositiveList, Vector NegativeList
Function Map (lines):
for all line ε lines do
    Score = 0
    for all word in line do
        if word ε PositiveList then
            score = score +1
        end if
        if word ε NegativeList then
            Score = Score-1
        end if
    end for
    if Score >2 then
        Score = 2
    end if
    if Score<2 then
        Score = -2
    end if
    if score = -2 then
        Output (“verybad”:1)
    else if score = -1 then
        output (“bad”:1)
    else if Score = 0 then
        output (“neutral”:1)
    else if Score = 1 then
        output (“good”:1)
    else if Score = 2 then
        output (“verygood”:1)
    end if
Function Reduce (scoreTuples)
    Result = {}
    For each (key, value) ε scoreTuples
        Result [key] = Result [key] + value
    Return Result
```

Algorithm 3: Sentiment Analysis on Twitter Data in MPI

```
Input filename: inputFilename
Block size: blockSize
Nodes number: rank
```
Total number of nodes: size

MPI init()
1. Open file inputFilename as fp
2. Set scores = {0, 0, 0, 0, 0}
3. execution_flag = 0
4. If (rank == 0)
   For i in 1 to (size - 1)
      MPI_Send (1, 1, MPI_INT, i, 1, MPI_COMM_WORLD)
   Else
      MPI_Recv (&execution_flag, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
6. If (execution_flag == 1 or rank == 0)
   While fp not null
      Read a block of blockSize from fp
      Scores = scores + Call analyze with argument block
   End while
7. Initialize received_scores to an empty array of integers whose size is (number_of_nodes * 5)
8. MPI_Gather (scores, 5, MPI_INT, received_scores, 5, MPI_INT, 0, MPI_COMM_WORLD)
9. If (rank == 0)
   For i = 0 to (number of nodes * 5)
      final_scores [i mod 5] = received_scores[i]
   End for
   Output final_scores
10. Return
   MPI Finalize().

Algorithm 4. Analyzer (MPI)

Input String array: block
   Negative words list negativelist
   Positive words list positivelist

1. Set positive 0
2. Set negative 0
3. Set neutral 0
4. Set highlypositive 0
5. Set highlynegative 0
6. Set score=0
7. For each word in positivelist
   If word in line
      Increment score by 1
   End if
End for
8. For each word in negativelist
   If word in line
      Decrement score by 1
   End if
End for
9. Clamp the value of score between -2 and 2
10. If score = -2:
    Increment highlynegative
Else if score = -1
    Increment negative
Else if score = 0
    Increment neutral
Else if score = 1
    Increment positive
Else if score = 2
    Increment highlypositive
End If
11. Return (highlynegative, negative, neutral, positive, highlypositive)

Algorithm 3 and Algorithm 4 outline the algorithm for sentiment analysis in MPI and its analyser, respectively.

Cluster Setup

The experiments were tested in two different cluster setups and the two configuration setups are mentioned below.

Cluster Setup-Configuration I

Processor Intel Haswell
Number of cores 4
RAM/core 3.6 GB
Hard Disk/core 500 GB
Total cores 40

Worker nodes(10 nodes)-configuration

Processor Intel Haswell
Number of cores 4
RAM 3.6 GB
Hard Disk 2000 GB

Results and Discussions

In this section, we exhibited the performance evaluation results and the overhead time associated with the computing time on spark. The execution time was evaluated for 100 GB, 500 GB and 1 TB. Table 1 includes the results for dataset size = 100 GB, on the cluster setup.

Table 2 includes the results for dataset size = 100 GB, on the cluster setup on one master machine and ten slave machines (S1-S10). Table 1 and 2 evaluated the execution time for the sentiment application along with the CPU utilization in each worker core. The average CPU utilization per core as 64.83 percent approximately for the application in apache spark in the mentioned configuration as shown in Table 1. Table 2 has shown the average CPU utilization per core when the application implemented in MPI was approximately 89.98%. Hence the CPU utilization was high in MPI processing framework. The execution times were compared in Table 3 and 4.
### Table 1: Sentiment analysis on apache spark (Scala) (100 GB)

| Exec no/ITN no. | 1   | 2   | 3   | 4   | 5   | 6   | 7   | 8   | 9   | 10  |
|----------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| Exe.Time(min)  | 3.6 | 3.5 | 3.6 | 3.5 | 3.6 | 3.6 | 3.5 | 3.6 | 3.6 | 3.7 |
| CPU Utilization % (W1) | 58.8 | 52.0 | 60.0 | 46.3 | 62.0 | 60.0 | 88.8 | 61.8 | 58.8 | 52.0 |
| CPU Utilization % (W2) | 58.3 | 84.0 | 62.0 | 64.8 | 63.0 | 56.5 | 66.8 | 72.0 | 58.3 | 84.0 |
| CPU Utilization % (W3) | 49.3 | 50.8 | 67.5 | 84.3 | 58.0 | 86.0 | 85.8 | 83.8 | 49.3 | 50.8 |
| CPU Utilization % (W4) | 60.0 | 66.3 | 76.0 | 67.0 | 68.0 | 59.0 | 78.8 | 53.5 | 60.0 | 66.3 |
| CPU Utilization % (W5) | 46.5 | 51.3 | 74.5 | 73.3 | 53.3 | 69.5 | 82.5 | 74.0 | 46.5 | 51.3 |
| CPU Utilization % (W6) | 63.0 | 71.3 | 54.0 | 68.8 | 57.8 | 82.0 | 67.8 | 73.8 | 63.0 | 71.3 |
| CPU Utilization % (W7) | 74.5 | 62.5 | 50.0 | 59.5 | 76.3 | 49.0 | 68.5 | 54.8 | 74.5 | 62.5 |
| CPU Utilization % (W8) | 77.5 | 80.0 | 58.5 | 61.8 | 70.8 | 50.3 | 65.3 | 54.5 | 77.5 | 80.0 |
| CPU Utilization % (W9) | 57.8 | 62.0 | 59.3 | 50.3 | 78.8 | 60.5 | 62.5 | 81.5 | 57.8 | 62.0 |
| CPU Utilization % (W10)| 63.0 | 66.5 | 55.3 | 69.5 | 72.3 | 74.0 | 49.8 | 76.3 | 63.0 | 66.5 |

### Table 2: Sentiment analysis on MPI (C/C++) (100 GB)

| Iteration | Master | S1 | S2 | S3 | S4 | S5 | S6 | S7 | S8 | S9 | S10 |
|-----------|--------|----|----|----|----|----|----|----|----|----|-----|
| Iteration 1 | CPU Util % | 88.8 | 89.8 | 90.3 | 89.5 | 89.3 | 89.0 | 89.0 | 89.5 | 89.5 | 89.5 |
| Iteration 1 | Exec. Time(min) | 1.56 |
| Iteration 2 | CPU Utilization % | 89.8 | 90.5 | 90.5 | 89.8 | 90.0 | 89.8 | 90.0 | 90.3 | 90.3 | 90.0 |
| Iteration 2 | Exec. Time(min) | 1.57 |
| Iteration 3 | CPU Utilization % | 89.5 | 89.8 | 90.8 | 89.8 | 89.8 | 89.5 | 89.8 | 90.0 | 90.3 | 90.0 |
| Iteration 3 | Exec. Time(min) | 1.57 |
| Iteration 4 | CPU Utilization % | 89.8 | 90.0 | 90.5 | 89.3 | 89.5 | 89.8 | 90.3 | 89.3 | 89.3 | 89.8 |
| Iteration 4 | Exec. Time(min) | 1.56 |
| Iteration 5 | CPU Utilization % | 90.0 | 90.0 | 90.3 | 89.3 | 89.8 | 90.5 | 89.5 | 89.5 | 89.8 | 90.0 |
| Iteration 5 | Exec. Time(min) | 1.57 |
| Iteration 6 | CPU Utilization % | 89.5 | 90.0 | 90.8 | 90.0 | 90.0 | 89.8 | 89.3 | 90.0 | 90.3 | 90.0 |
| Iteration 6 | Exec. Time(min) | 1.57 |
| Iteration 7 | CPU Utilization % | 89.5 | 89.8 | 90.5 | 90.3 | 90.3 | 89.8 | 89.5 | 89.8 | 90.3 | 90.3 |
| Iteration 7 | Exec. Time(min) | 1.56 |
| Iteration 8 | CPU Utilization % | 89.3 | 90.3 | 90.5 | 90.0 | 89.3 | 90.0 | 89.3 | 90.0 | 90.0 | 89.8 |
| Iteration 8 | Exec. Time(min) | 1.57 |
| Iteration 9 | CPU Utilization % | 89.3 | 90.0 | 90.3 | 90.3 | 89.8 | 90.0 | 89.3 | 90.5 | 90.0 | 90.3 |
| Iteration 9 | Exec. Time(min) | 1.57 |

### Table 3: Execution times on 100 GB/500 GB/1 TB datasets in spark processing

| Data size | ITN1 | ITN2 | ITN3 | Avg. Exe. Time (Min) |
|-----------|------|------|------|----------------------|
| 100 GB    | 3.6  | 3.5  | 3.6  | 3.58                 |
| 500 GB    | 17.0 | 16.0 | 17.0 | 16.67                |
| 1 TB      | 33.0 | 32.0 | 32.0 | 32.33                |

### Table 4: Average execution times on 100GB/500GB/1TB datasets in MPI

| Data size | ITN1 | ITN2 | ITN3 | Avg. Exe. Time (Min) |
|-----------|------|------|------|----------------------|
| 100 GB    | 1.56 | 1.57 | 1.57 | 1.56                 |
| 500 GB    | 10.04 | 10.07 | 10.06 | 10.06               |
| 1 TB      | 22.18 | 22.36 | 22.17 | 22.23               |
Fig. 2: Execution time (minutes) Spark Vs MPI (100 GB)

Fig. 3: Execution time (minutes) Spark Vs MPI comparison

Fig. 4: Summary on all overhead times for Sentiment analysis using Spark on 100 GB dataset (From UI application, Spark 2.0.2)
Figure 2 and 3 shows that MPI was approximately 2 x times faster than Spark processing over a cluster of 10 machines with 40 cores with the comparable results obtained in (Jha et al., 2014), (Reyes-Ortiz et al., 2015) and (Gittens et al., 2016). Spark processing incurs some delay metrics during execution as summarized in Fig. 4. Figure S1 and S2 supplements the overhead summary for 500 GB and 1 TB datasets. Spark computes Directed Acyclic Graph (DAG) from the RDDs as shown in Fig. 5. The DAG generally includes stages with stage0 will have a series of pipelined transformations and the subsequent stages are based on the computed actions. All the transformations are grouped in stage0 and depending on the number of actions; subsequent stages will be included in the DAG. Figure 5 includes the filter RDD and map RDDs were pipelined to stage0 and the reduce operation for final result evaluation in stage1. For the implementation of actions on an RDD, Spark submits the DAG to the DAG Scheduler. It schedules the operations in each stage of the DAG and combines the operations which can be performed on a single stage. Then these stages were passed to the task tracker over the cluster of workers through the YARN/Mesos cluster managers (https://spark.apache.org/docs/1.2.1/api/java/org/apache/spark/scheduler/DAGScheduler.html). Task trackers launch the tasks and executes in parallel without knowing the dependency among the operations in stages and hence spark handles an effective task parallelism.

The stage-wise evaluation of execution time was shown in Figure 6 for 100 GB dataset. Stage0 was computed the entire filter and a series of map operations in 3.5 min and the final collect operation at stage1 in 0.1 sec. For 500 GB and 1 TB, the two-stage summary of execution time was provided as supplementary material Figures S3 and S4.

Factors Included in Performance Measures of Parallel Computing

Execute Time

The measure of performance will vary with respect to the data volume, the analytics used in the corresponding application, whether the algorithm is iterative or non-iterative and the application is interactive or batch processing etc. With respect to the chosen application, the execute time will vary in various frameworks. The time taken to execute the job, once the metadata is captured and the corresponding data made available to the application is called the execute time. An average Execution time for sentiment analysis on 100 GB/500 GB/1 TB data for 3 different attempts (ITN1-ITN3) is displayed on Table 3.
Table 3 is based on ten different iterations on 100 GB and three iterations on 500 GB and 1 TB, with an average execute time was 3.58, 16.67 and 32.33 min respectively on the three data set sizes on Spark processing on 40 cores.

Table 4 shows the average execute times, 1.56 minutes, 10.06 and 22.23 min, on 100 GB, 500 GB and 1 TB datasets respectively.

**Overhead**

The additional time elapsed due to various latencies in accessing the data, metadata, network bandwidth, delays in the processing frameworks and some undetectable factors that affects the extension of total execution time is called as overheads.

Spark overheads other than accessing data, metadata, network bandwidth were shown in Figure 4 for 100 GB data set size and the latencies was evaluated as 26.4 seconds which includes the Task initiation delay (Duration-(14.0 s)), Scheduler delay (0.3 s), task deserialization time (1.0 s), Garbage Collection time (GC Time -0.6 s), Result serialization time (5 ms) and Getting Result Time (0 ms). Kryo serializer was installed for task serialization/deserialization. Since the cluster was setup on Google cloud, the network latencies were not evaluated. The data were directly stored on each nodes in the cluster and hence delay in reading the data from HDFS and metadata read latency were not included in this experiment. In MPI, the experiment was conducted after distributing data over the cluster of nodes and hence latency in reading data from the disk was not evaluated. MPI overheads will be affected during I/O scalability and due to data parallel operations in extremely large scale data processing (Anderson et al., 2017).

**Total Time**

The actual time incurred for the execution of jobs to avail the results is termed as total time, which includes Execute time and Overhead.

The total time for sentiment analysis on spark was evaluated as 3.58, 16.67 and 32.33 min and in MPI, the total time was 1.56, 10.06 and 22.23 min, on 100 GB, 500 GB and 1 TB datasets, respectively. Hence practically verified that MPI outperforms Spark by two orders of magnitude for the application of lexicon based sentiment analysis.

**Conclusion**

Detailed study on spark processing was conducted in Spark2.0.2 on Hadoop2.7.3 and YARN on a Google cloud cluster setup. Experimental results were analyzed based on peak memory utilization, peak CPU utilization and execution time. Sentiment analysis application was chosen as the application and it was implemented in Scala for spark processing and programmed in C++ for the implementation in MPI. In the experiment, the execution time on spark and MPI shows that the execution speed of MPI was approximately 2* faster than spark processing with the configuration II at par with the results in (Reyes-Ortiz et al., 2015).

The results shows better performance on MPI environment than Spark was due to two reasons. Firstly, In MPI, the freedom of the programmer to choose the memory requirements, in terms of the number of lines of tweets to be executed to avail better performance. Whereas in Spark, the memory allocation and task scheduling are purely under the control of Spark processing Framework and programmer intervention is not possible. Secondly, C++ is more flexible programming language than Scala.

The CPU utilization on the cores and the average utilization were proven to be higher on MPI than Apache Spark. The memory utilization factor could be adjusted in MPI programming for better performance,
whereas the memory utilization factor in Spark programming was controlled by the Spark by the Spark job controller. But MPI lacks a common runtime for big data processing environment (Jha et al., 2014). Bridging the gap between big data processing and HPC by incorporating MPI with the integrated I/O management and fault tolerance is one of the motivating scientific studies as proposed in (Anderson et al., 2017). Further research is required on I/O management and fault tolerance integration to handle the big data in MPI processing environment.

Acknowledgment
I sincerely express my gratitude to my guide Dr (Prof) M Abdul Rahman for his constant encouragement and support.

Author’s Contributions
Deepa S Kumar: Implemented and tested the algorithms.
M Abdul Rahman: Project supervisor.

Ethics
The authors declare no conflict of interest.

References
Anderson, M., S. Smith, N. Sundaram, M. Capotă and Z. Zhao et al., 2017. Bridging the gap between HPC and big data frameworks. Proc. VLDB Endowment, 10: 901-912. DOI: 10.14778/3090163.3090168
Axtmann, M., T. Bingmann, E. Jobstl, S. Lamm and H.C. Nguyen et al., 2016. Thrill-distributed big data batch processing framework in C++.
Deeplearning4j, 2016. Deep learning for java. Open-Source, Distributed, Deep Learning Library for the JVM.
Diaz, J., C. Munoz-Carlo and A. Ni no, 2012. A survey of parallel programming models and tools in the multi and many-core era. IEEE Trans. Parallel Distr. Syst., 23: 1369-1386. DOI: 10.1109/TPDS.2011.308
Fagg, G.E. and J.J. Dongarra, 2000. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. Proceedings of the 7th European PVM/MPI Users’ Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, (MPI’00), Springer Nature, London, pp: 346-353.
Forum, M.P.I., 2015. Mpi: A message-passing interface standard version 3.1. Technical Report.
Gamell, M., D.S. Katz, H. Kolla, J. Chen and S. Klaskey et al., 2014. Exploring automatic, online failure recovery for scientific applications at extreme scales. Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Nov. 16-21, IEEE Xplore Press, New Orleans, LA, USA, pp: 895-906. DOI: 10.1109/SC.2014.78
Gittens, A., A. Devarakonda, E. Racah, M. Ringenburg and L. Gerhardt et al., 2016. Matrix factorizations at scale: A comparison of scientific data analytics in spark and C+MPI using three case studies. Proceedings of the IEEE International Conference on Big Data, Dec. 5-8, IEEE Xplore Press, Washington, DC, USA, pp: 204-213. DOI: 10.1109/BigData.2016.7840606
Gropp, W., S. Huss-Lederman and A. Lumsdaine, 1998. MPI: The complete reference, the MPI-2 Extensions. The MIT Press.
Grossman, M. and V. Sarkar, 2016. SWAT: A programmable, in-memory, distributed, high-performance computing platform. Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing, May 31-Jun 04, ACM, Kyoto, Japan, pp: 81-92. DOI: 10.1145/2907294.2907307
H2O.ai. 2016. Sparkling Water. https://github.com/h2oai/sparkling-water
https://spark.apache.org/docs/1.2.1/api/java/org/apache/spark/scheduler/DAGScheduler.html
https://www.edureka.co/blog/apache-spark-vs-hadoop-mrmapreduce.
Jha, S., J. Qu, A. Luckow, P. Mantha and G.C. Fox, 2014. A tale of two data-intensive paradigms: Applications, abstractions and architectures. Proceedings of the IEEE International Congress on Big Data, Jun. 27-Jul. 2, IEEE Xplore Press, Anchorage, AK, USA, pp: 645-652. DOI: 10.1109/BigData.Congress.2014.137
Kanavos, A., N. Nodarakis, S. Sioutas, A. Tsakalidis and D. Tsolis et al., 2017. Large scale implementations for twitter sentiment classification, Algorithms, 10: 33-33. DOI: 10.3390/a10010033
Kang, S.J., S.Y. Lee and K.M. Lee, 2015. Performance comparison of OpenMP, MPI and MapReduce in practical problems. Adv. Multimedia, 2015: 575687-575687. DOI: 10.1155/2015/575687
Khanam, Z. and S. Agarwal, 2015. Map-reduce implementations: Survey and performance comparison. Int. J. Comput. Sci. Inform. Technol., 10: 119-126. DOI: 10.5121/ijcsit.2015.7410
Ousterhout, K., R. Rasti, S. Ratnasamy, S. Shenker and B.G. Chun, 2015. Making sense of performance in data analytics frameworks. Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation, May 04-06, USENIX Association, Berkeley, CA, USA, pp: 293-307.
Plimpton, S.J. and K.D. Devine, 2011. MapReduce in MPI for large-scale graph algorithms. Parallel Comput. J., 37: 610-632. DOI: 10.1016/j.parco.2011.02.004

Raveendran, A., T. Bicer and G. Agrawal, 2011. A framework for elastic execution of existing MPI programs. Proceedings of the IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum, May 16-20, IEEE Xplore Press, pp: 940-947. DOI: 10.1109/IPDPS.2011.240

Reyes-Ortiz, J.L., L. Oneto and D. Anguita, 2015. Big data analytics in the cloud: Spark on hadoop Vs MPI/OpenMP on Beowulf. Proc. Comput. Sci., 53: 121-130. DOI: 10.1016/j.procs.2015.07.286

Satish, N., N. Sundaram, M.M.A. Patwary, J. Seo and J. Park et al., 2014. Navigating the maze of graph analytics frameworks using massive graph datasets.

Supplementary Material

Figure S1 and S2 summarizes the overhead times for the implementation on the algorithm using spark on 500GB and 1TB. The stage-wise execution summary of Spark processing on 500GB and 1TB were shown as Fig. S3 and S4. Spark Processing framework evaluations for 500GB and 1TB results appended in Table S1 and S2 along with the values calculated in Table 3. MPI processing framework evaluations for three different attempts for 500GB and 1TB data set sizes were exhibited in Table S3 and S4 along with the values calculated in Table 4.

Table S1: Sentiment analysis on apache Spark (Scala) 500 GB, configuration II)

| Execution number | Avg Exe Avg CPU |
|------------------|-----------------|
|                  | Exe time(min)   | Utilization% |
| 1                | 16              | 42.8          | 48.0 |
| 2                | 16              | 44.0          | 61.5 |
| 3                | 17              | 43.3          | 76.3 |
|                  | 16.67           | 65.5          | 42.5 |
|                  | 56.3            | 43.5          | 43.8 |
|                  | 59              | 44            | 44.8 |
|                  | 72.0            | 44.0          | 47   |
|                  | 67.5            | 43.3          | 47   |
|                  | 56.3            | 46            | 56.3 |
|                  | 70.3            | 65.5          | 56.3 |
|                  | 59              | 44            | 71.5 |

| Execution number | Avg exec. Avg CPU |
|------------------|-------------------|
|                  | Exe time(MIN)     | Utilization% |
| 1                | 32               | 42.5          | 42.5 |
| 2                | 32               | 44.0          | 47   |
| 3                | 32               | 44.8          | 47   |
|                  | 32.33            | 43.5          | 44.8 |
|                  | 54               | 43.8          | 43.5 |
|                  | 57.3             | 44            | 56.3 |
|                  | 75               | 43.5          | 44.8 |
|                  | 72.3             | 44.0          | 50   |
|                  | 72.3             | 43.3          | 47   |
|                  | 64.5             | 46            | 60   |

Table S2: Sentiment analysis on apache spark (Scala) (1 TB, Configuration II)

| Execution number | Avg exec. Avg CPU |
|------------------|-------------------|
|                  | Exe time(MIN)     | Utilization% |
| 1                | 32               | 42.5          | 42.5 |
| 2                | 32               | 44.0          | 47   |
| 3                | 32               | 44.8          | 47   |
|                  | 32.33            | 43.5          | 44.8 |
|                  | 54               | 43.8          | 43.5 |
|                  | 57.3             | 44            | 50   |
|                  | 75               | 43.5          | 47   |
|                  | 72.3             | 44.0          | 55.3 |
|                  | 64.5             | 46            | 60   |
Table S3: Sentiment analysis on MPI (C/C++) (500 GB, configuration II)

| Execution time (minutes) | Avg Exe time | Avg CPU Utilisation% |
|--------------------------|--------------|----------------------|
| 1                        | 10.04        | 10.07                |
| 2                        | 10.06        | 88.2                 |
| 3                        | 10.06        | 88.33                |
| Average                  | 10.06        | 88.2                 |

CPU utilization:
- Master: 90.25%
- S1: 90%
- S2: 88.73%
- S3: 88.5%
- S4: 88.75%
- S5: 88.5%
- S6: 88.25%
- S7: 88.5%
- S8: 87.75%
- S9: 88%
- S10: 88.25%

Table S4: Sentiment analysis on MPI (C/C++) (1 TB, configuration II)

| Execution time (minutes:seconds:ms) | Avg exec. Time(MIN) | Avg CPU Utilisation% |
|-------------------------------------|---------------------|----------------------|
| 1                                   | 22.18               | 90.05                |
| 2                                   | 22.36               | 90.05                |
| 3                                   | 22.17               | 90.05                |
| Average                             | 22.23               | 90.05                |

CPU utilization:
- Master: 89.25%
- S1: 90.25%
- S2: 90%
- S3: 90.25%
- S4: 90%
- S5: 90.5%
- S6: 90.25%
- S7: 90.5%
- S8: 90.25%
- S9: 90.25%
- S10: 88.5%

Fig. S1: Summary on all overhead times for the implementation on the algorithm using spark on 500 GB

Fig. S2: Summary on all overhead times for the implementation on the algorithm using spark on 500 GB
Fig. S3: Summary on Job execution time on stages (500 GB dataset)

Fig. S4: Summary on Job execution time on stages (1 TB dataset)