PERFORMANCE MODELLING, ANALYSIS AND PREDICTION OF SPARK JOBS IN HADOOP CLUSTER

A Thesis by Publications Presented in Partial Fulfilment of the Requirements for the Degree of

DOCTOR OF PHILOSOPHY
IN
COMPUTER SCIENCE

School of Mathematical & Computational Sciences
Massey University, Auckland, New Zealand

Nasim Ahmed

July 1, 2022

Abstract

Big Data frameworks have received tremendous attention from the industry and from academic research over the past decade. The advent of distributed computing frameworks such as Hadoop MapReduce and Spark are powerful frameworks that offer an efficient solution for analysing large-scale datasets running under the Hadoop cluster. Spark has been established as one of the most popular large-scale data processing engines because of its speed, low latency in-memory computation, and advanced analytics.

Spark computational performance heavily depends on the selection of suitable parameters, and the configuration of these parameters is a challenging task. Although Spark has default parameters and can deploy applications without much effort, a significant drawback of default parameter selection is that it is not always the best for cluster performance. A major limitation for Spark performance prediction using existing models is that it requires either large input data or system configuration that is time-consuming. Therefore, an analytical model could be a better solution for performance prediction and for establishing appropriate job configurations.

This thesis proposes two distinct parallelisation models for performance prediction: the *2D-Plate* model and the *Fully-Connected Node* model. Both models were constructed based on serial boundaries for a certain arrangement of executors and size of the data. In order to evaluate the cluster performance, various HiBench workloads were used, and workload's empirical data were fitted with the models for performance prediction analysis. The developed models were benchmarked with the existing models such as Amdahl's, Gustafson, ERNEST, and machine learning. Our experimental results show that the two proposed models can quickly and accurately predict performance in terms of runtime, and they can outperform the accuracy of machine learning models when extrapolating predictions.

Authors Declaration

This thesis was produced according to Massey University's "PhD thesis by publication" guidelines. This thesis is based on research that has been published in Journals and IEEE conference. In accordance with Journal and IEEE copyright policy, this thesis is contains all published manuscript. Consequently, there may be stylistic differences between the enclosed work or the published versions. Furthermore, the work contained within this thesis was published in several Big Data core Journals. Each manuscript has addressed different problems and proposed efficient models to solve those problems. However, this may have some repetition in literature review.

I, Nasim Ahmed, declare that this thesis is a fulfilment of the requirements for the fulfilment of the degree of Doctor of Philosophy (Ph.D.), from School of Mathematical & Computational Sciences, Massey University, is wholly my own work unless otherwise referenced acknowledged. This document has not been submitted for qualifications at any other academic institution.

Nasim Ahmed

Acknowledgements

All praises to Allah Almighty, the Merciful, the Beneficent.

First and foremost, I would like to express my appreciation and sincere gratitude to my supervisor Dr. Andre Barczak for giving me the opportunity to be part of this research project. I am grateful to him for his invaluable encouragement, constant support, and guidance throughout the progress of my research project. I'd like to give thanks to my co-supervisors, Dr. Mohammed Abdur Rashid whose supportive and insightful advice was invaluable in the pursuit of my research as well as Dr. Teo Susnjak for sharing his valuable knowledge, academic curiosity and enthusiasm.

I sincerely acknowledge the generosity of Massey University for supporting my PhD studies with a Doctoral Scholarship and the School of Natural and Computational Sciences for providing additional support.

Last but not the least, I would like to acknowledge the support and sacrifice of my wife, Dr. Noorzahan Begum and lovely daughter, Najifah Tasnim Saba.

Finally, I'd like to thank my parents for the love they give me and supporting me spiritually throughout my life.

Funding

I gratefully acknowledge the fundings received from:

- Massey Doctoral Scholarship provided by the Massey University.
- School of Natural and Computational Sciences (SNCS) for open access and conference registration fees.
- Massey University for providing me casual funding support through the Massey University Research Fund (MURF).

Contents

List of Tables

List of Figures

Abbreviations and Terms

SVM	Support Vector Machines
API	Application Programming Interface
SQL	Structured Query Language
HDFS	Hadoop Distributed File System
RDD	Resilient Distributed Datasets
MLlib	Machine Learning Library
CPU	Central Processing Unit
I/O	Input/Output
UC	University of California
AMP	Algorithms, Machines and People
DAG	Directed Acyclic Graph
YARN	Yet Another Resource Negotiator
PERIDOT	Performance PredIction MoDel fOr Spark ApplicaTions
NEXEC	Number of Executor
SQRT	Square Root
2D	Two Dimensional
GHz	Gigahertz
TB	Terabyte
RAM	Random Access Memory

DDR	Double Data Rate
GB	Gigabyte
MB	Megabyte
WC	WordCount
Exec	Executor
MOP	Multi-Object Optimization
EC2	Elastic Computing
JVM	Java Virtual Machine
IaaS	Infrastructures as a Service
MLR	Multi Linear Regression
LR	Linear Regression
DT	Decision Tree
RF	Random Forest
RLR	Regularized Linear Regression
UI	User Interface
ML	Machine Learning
AMPLAB	Algorithms, Machines and People Lab
KNN	K-nearest Neighbour
SVR	Support Vector Regression
MF	Matrix Factorization

GBR	Gradient Boost Regression
LR	Logistic Regression
NB	NaiveBase
GBN	Gradient Boost Machine
NN	Neural Networks
MLRQ	MLR-Quadratic
TSt	Two-Stage tree
LS-SVM	Least Square Support Vector Machine

Chapter 1

The content of this chapter presents the thesis introduction, and chapter overview in association with objectives, thesis questions, research scope, list of publications based on contributions.

1.1 Introduction

In the 21st century, data science has attracted tremendous attention in the industry and academic areas [1]. It has been an attractive multidisciplinary research area associated with every domain such as Computer Science, Mathematics, Engineering, Biology, Statistics, Medicine, Transport, Manufacturing etc [2]. With the rapid growth of digital technology, massive amounts of data are generated from every sector: social media platform, smartphone, online shopping, industrial processes, scientific experiments, health care records, sensory, business transactions and many more. Usually, the collection of this data is extensive in volume. In principle, this large volume of data is called big data. It is called big data, but there is no proper definition. There are several big data definitions and terminology that have been proposed by the literature. Among these, a popular terminology is "big data problem". Many researchers, data scientists, and experts have defined four main characteristics that are referred to as 4V's or dimensions: Volume, Variety, Veracity, and Velocity [3].

Due to the explosion of big data, it has been a challenging task to store the data in a single machine and analyse them efficiently. Therefore, big

data requires a system with large storage, fast processing, quick decision making, better analytic, and optimisation ability. Distributing computing is a promising paradigm that is capable of solving these problems. Big data frameworks are developed based on distributed computing concepts, which plays a vital role in large-scale data processing. Apache Hadoop [4] and Apache Spark [5] are popular platforms to handle large data sets. Over the years, Apache Hadoop has become a popular data processing framework for all data sizes. Apache Hadoop consists of two core components: Hadoop Distributed File System (HDFS) and the MapReduce programming model. HDFS is used for data storage and MapReduce [6] to process the Map and Reduce function.

Recently, Apache Spark was launched, which offers numerous advantages for developers to build big data applications that support fast and quick data processing. The Resilient Distributed Datasets (RDD) and Directed Acyclic Graph (DAG) techniques work jointly and accelerate Spark much more faster than Hadoop (MapReduce) under certain circumstances. As a unified engine of big data analytics, Apache Spark [5] provides high scalability and fault tolerance with its unique memory engine. In Apache Spark, there are more than 150 configurable parameters. Default values are assigned to each parameter. The default values, however, are not always suitable settings for individual workloads. Spark is a relatively new data processing framework, hence, essential research gaps remain. In particular, the system runtime performance improvement is based on various HiBench workloads [7]. A typical Hadoop eco-system works with Spark and HDFS as shown in Figure 1.

Researchers have proposed various Spark performance prediction tech-

niques namely; trial-and-error [8], black-box [9], cost-based [10], and machine learning models [11] [12]. All these techniques have identical limitations; for example, they are time consuming or require large amounts of training data. This study found that a number of existing limitations can be overcome by developing a simple and generic analytical models utilizing the existing parallel computing methods. An efficient analytical model can significantly reduce the prediction time and required data. Investigation into the literature suggests that new models based on communication patterns that are similar to Amdahl's, Gustafson and others could be created for significant improvements.

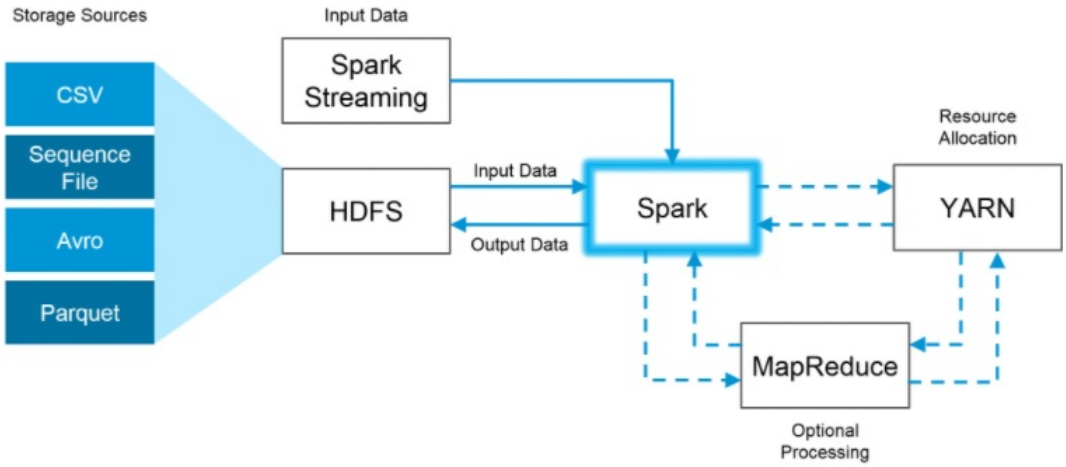


Figure 1: A typical Hadoop eco-system works with Spark and HDFS [13].

1.2 Objectives

The main goal of this research is to develop a generic tuning model to predict Spark runtime performance based on HiBench workloads. The objectives of this research are as follows:

1. To identify and explore the most important parameters of Hadoop and Spark frameworks by configuring jobs with different parameter settings.
2. To investigate the Trial-and-Error tuning technique for better tuning efficiency of Hadoop and Spark benchmarks.
3. To develop and design generic models to predict Spark runtime performance quickly and accurately.
4. To compare the feasibility and the accuracy of the new models with the existing analytical and machine learning regression models.

1.3 Research Questions

Prediction of cluster performance is a complex and tedious work. Spark framework runtime performance was analysed in detail. From the study, it was observed that performance predictions are either time-consuming or require large amounts of training data. In this context the following questions were answered:

1. How does a large number of parameters affect the system performance and what are the ideal parameters for individual workloads?

2. How quickly system administrators, architects, and data engineers can tune the possible system parameters, considering the number of executors and data sizes, for any Spark job running on Hadoop cluster?

3. What parallelisation model for a Hadoop cluster can be found and implemented quickly and efficiently to improve the performance prediction of a job?

4. How can the practitioners predict system performance can be achieved by running a few jobs, short runtimes, limited empirical data, and be able to predict the runtime of longer untested data set sizes?

1.4 Scope of the Research

The main scope of this research is to develop a general tuning model to help the administrators, users, operators, and researchers with the Spark framework to predict the overall system performance. This research is based on HiBench workloads, and thus the implementation of this work is suitable not only for Hadoop based clusters but also other clusters. The scope of this research has been limited to:

- The cluster size is composed of nine slave nodes and one master node.
- Constrained to only using HiBench workloads and considered parameters from resource utilization, input splits, and shuffling categories.
- The experimental input data is generated randomly and the maximum data size is limited to 600GB.

- Employing the number of executors and the data size parameters for the function of the models and compared with analytical and machine learning models.
- The regression machine learning models were considered for the comparison with analytical models due to the small sample size (limited number of experiments for one workload).

1.5 List of Publications based on Contributions

This thesis makes 5 contributions based on publications in peer reviewed journals and conference:

- N. Ahmed, Andre L.C. Barczak, Mohammed A. Rashid and Teo Susnjak, "Runtime Prediction of Big Data Jobs: Performance Comparison of Machine Learning Algorithms and Analytical Models", Journal of Big Data 9, 67, SpringerLink, Q1 , SJR. I.F.: 1.025, 2022.
- N. Ahmed, Andre L.C. Barczak, Mohammed A. Rashid and Teo Susnjak, "An Enhanced Parallelization Model for Performance Prediction of Apache Spark on a Multinode Hadoop Cluster", 5(4), pp. 1-25, Journal of Big Data and Cognitive Computing, MDPI, Q1, SJR. I.F.: 0.551, 2021.
- N. Ahmed, Andre L.C. Barczak, Mohammed A. Rashid and Teo Susnjak, "A Parallelization Model for Performance Characterization of

Spark Big Data Jobs on Hadoop Clusters", vol 8, pp. 1-28, Journal of Big Data, SpringerLink, Q1, SJR. I.F.: 1.025, 2021.

- N. Ahmed, Andre L.C. Barczak, Teo Susnjak, and Mohammed A. Rashid, "A Comprehensive Performance Analysis of Apache Hadoop and Apache Spark for Large Scale Data Sets Using HiBench" Journal of Big Data, SpringerLink (DOI: 10.21203/rs.3.rs 43526/v1, SJR. I.F.: 1.025, Q1, 2020.

- N. Ahmed, Andre L.C. Barczak, Sibghat Ullah Bazai, Teo Susnjak, and Mohammed A. Rashid, "Performance Analysis of Multi Node Hadoop Cluster based on Large Data Sets" The 7th IEEE CSDE 2020, The Asia Pacific Conference on Computer Science and Data Engineering (CSDE, IEEE), DOI: 10.1109/CSDE50874.2020.9411587, December 2020, Gold Coast, Australia.

1.6 Thesis Overview

Since the current state of the literature shows the significant research gap relating to the Spark performance prediction, this thesis will explore how quickly and efficiently Spark performance prediction can be achieved. In this study, the distinct parallelisation models were developed for performance prediction of Spark jobs on the Hadoop cluster. Each model is based on a different communication pattern between the nodes of a Hadoop cluster. Indeed, the proposed models significantly help researchers, cluster users, operators, and system administrators in their research and cluster infrastructures

development. This research contains some interesting, comprehensive comparative performance studies between Apache Hadoop and Apache Spark, which is a significant gap in the literature. An effective model was developed that can explain various HiBench jobs' performance patterns as a function of the number of executors. This model was implemented as a black-box approach, i.e., without knowing the internal workings of communication between the executors or the I/O involved in running the jobs (via HDFS). Models were further developed based on two main parameters, the number of executors and the amount of data for each job. The models are based on a different communication pattern between the nodes of a Hadoop cluster. The performance accuracy of the model's were compared with several analytical and machine learning models and analysed their efficiency.

Chapter 1 presents the introductory topics of the thesis covering objectives, thesis questions, research scope, list of publications based on contributions. Chapter 2 demonstrates and accomplishes a comprehensive empirical performance analysis between Apache Hadoop and Apache Spark frameworks by correlating system resources. Both system performances were explored by utilising large scale datasets (600 GB). A novel model based on the black-box approach was proposed and presented the HiBench jobs' performance in Chapter 3. Chapter 4 proposed two distinct models based on different communication patterns between the nodes and the Hadoop cluster, which extend the first model in Chapter 3. Chapter 5 investigates the proposed model performance with various analytical and machine learning models. Appendix 1 reports the extension work of comprehensive empirical performance analysis between Apache Hadoop and Apache Spark frameworks. In this thesis, there

is no traditional literature review. Every chapter and the appendix presents a peer reviewed research publication, which has self-contained literature review. Next, a brief summary of each chapter is presented.

1.7 Chapter Overview and Contributions

Chapter 1 of this thesis presents thesis introduction, objectives, research questions, research scope, and a list of publications based on contributions.

Chapter 2 presents comprehensive performance analysis between Apache Hadoop and Apache Spark for large scale datasets (600 GB). The experiment was performed on a 9 node Hadoop cluster. Both system performance is evaluated based on HiBench workloads such as WordCount and Terasort. In this investigation, the well-known approach called trial-and-error is taken into account for parameter configuration. In response to the parameters' consideration, this study considered various combinations and their correlation which includes resource utilisation, split size, and shuffle behaviour. Experiment confirmed that both system performance heavily depends on input data size and selection of right parameters. Trial-and-error approach found that Apache Spark process data faster than Apache Hadoop when input data size is smaller. In special cases, Apache Hadoop can processes large data sets faster than Apache Spark. Insight from this study indicates that Spark is more stable and faster than Hadoop because of Spark data processing ability in memory instead of stored in disk. Primarily, this extensive study guides us to choose the best data processing framework for further investigation. The findings of this study were published as a peer reviewed journal (Journal

of Big Data, SpringerLink) article included in Chapter 1, and a conference paper included in Appendix 1.

Chapter 3 reports the development of a new parallelisation model (2D-Plate) for quick prediction of Spark jobs. The model was constructed to find a pattern for the parallelisable and non-parallelisable portions of a generic job. The proposed model mitigate various issues that have been addressed in the literature. This model offers excellent accuracy for various types of Hi-Bench workloads. The proposed 2D-Plate model acts as a black-box approach without knowing any specific communication pattern between the executors and input/output involvement running through HDFS. It has demonstrated the limitations of the 2D-Plate model concerning executors and data size. The limitations were solved in Chapter 3. This work was published in a peer-reviewed journal (Journal of Big Data, SpringerLink).

In Chapter 4 two distinct models (2D-plate model and Fully-Connected Node Model) were proposed; where each models are based on a different communication pattern between cluster nodes. In this development, the previous model is extended and introduced new parallelisation models that simultaneously consider the executors and input data size. These model do not require large experimental data like machine learning. These models significantly reduce overall cluster configuration time. These models are more accurate than existing analytical and machine learning models and they solve the limitation of performance prediction of Spark job on the Hadoop cluster. This work was published in Journal of Big Data and Cognitive Computing, MDPI.

Chapter 5 investigates the prediction accuracy between the proposed models with other well-known analytical and machine learning models. Machine learning models usually require either large training data or have poor data fitting accuracy while the proposed models successfully overcome those limitations especially for extrapolated data. To the best of the author's knowledge this is the first reported performance comparison work with minimum experimental input data where models are effective and accurately predict the system performance. In this analysis, various methods such as interpolation, extrapolation, and k-fold cross-validation were used for the model's accuracy validation. This work was published in a peer-reviewed journal (Journal of Big Data, SpringerLink).

References

[1] Matthew A Waller and Stanley E Fawcett. *Data science, predictive analytics, and big data: a revolution that will transform supply chain design and management*. 2013.

[2] Brad Brown, Michael Chui, and James Manyika. "Are you ready for the era of big data". In: *McKinsey Quarterly* 4.1 (2011), pp. 24–35.

[3] Wissem Inoubli et al. "An experimental survey on big data frameworks". In: *Future Generation Computer Systems* 86 (2018), pp. 546–564.

[4] *Apache Hadoop Documentation 2014*. URL: `http://hadoop.apache.org/`.

[5] Matei Zaharia et al. “Apache spark: a unified engine for big data processing”. In: *Communications of the ACM* 59.11 (2016), pp. 56–65.

[6] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: simplified data processing on large clusters”. In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[7] Intel-bigdata. *HiBench Benchmark Suit*. URL: `https://github.com/Intel-bigdata/HiBench`.

[8] Panagiotis Petridis, Anastasios Gounaris, and Jordi Torres. “Spark parameter tuning via trial-and-error”. In: *INNS Conference on Big Data*. Springer. 2016, pp. 226–237.

[9] Alexandre Maros et al. “Machine learning for performance prediction of spark cloud applications”. In: *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE. 2019, pp. 99–106.

[10] Herodotos Herodotou et al. “Starfish: A Self-tuning System for Big Data Analytics”. In: *Cidr*. Vol. 11. 2011. 2011, pp. 261–272.

[11] Sara Mustafa, Iman Elghandour, and Mohamed A Ismail. “A machine learning approach for predicting execution time of spark jobs”. In: *Alexandria engineering journal* 57.4 (2018), pp. 3767–3778.

[12] Guoli Cheng et al. “Efficient Performance Prediction for Apache Spark”. In: *Journal of Parallel and Distributed Computing* 149 (2021), pp. 40–51.

[13] Ashok R. Dinasarapu. *Building a real-time big data pipeline (2: Spark Core, Hadoop, Scala)*. May 2020. URL: `https://adinasarapu.github.io/big-data/2020/02/blog-post-spark/`.

Chapter 2

The contents of this chapter are from the following article. In accordance with the Springerlink open access policy, any full text that has been included, is the unmodified accepted article.

Ahmed *et al. J Big Data* (2020) 7:110
https://doi.org/10.1186/s40537-020-00388-5

Journal of Big Data

RESEARCH Open Access

A comprehensive performance analysis of Apache Hadoop and Apache Spark for large scale data sets using HiBench

N. Ahmed[1*], Andre L. C. Barczak[1], Teo Susnjak[1] and Mohammed A. Rashid[2]

*Correspondence:
nasim751@yahoo.com
[1] School of Natural and Computational Sciences, Massey University, Albany, Auckland 0745, New Zealand
Full list of author information is available at the end of the article

Abstract

Big Data analytics for storing, processing, and analyzing large-scale datasets has become an essential tool for the industry. The advent of distributed computing frameworks such as Hadoop and Spark offers efficient solutions to analyze vast amounts of data. Due to the application programming interface (API) availability and its performance, Spark becomes very popular, even more popular than the MapReduce framework. Both these frameworks have more than 150 parameters, and the combination of these parameters has a massive impact on cluster performance. The default system parameters help the system administrator deploy their system applications without much effort, and they can measure their specific cluster performance with factory-set parameters. However, an open question remains: can new parameter selection improve cluster performance for large datasets? In this regard, this study investigates the most impacting parameters, under resource utilization, input splits, and shuffle, to compare the performance between Hadoop and Spark, using an implemented cluster in our laboratory. We used a trial-and-error approach for tuning these parameters based on a large number of experiments. In order to evaluate the frameworks of comparative analysis, we select two workloads: WordCount and TeraSort. The performance metrics are carried out based on three criteria: execution time, throughput, and speedup. Our experimental results revealed that both system performances heavily depends on input data size and correct parameter selection. The analysis of the results shows that Spark has better performance as compared to Hadoop when data sets are small, achieving up to two times speedup in WordCount workloads and up to 14 times in TeraSort workloads when default parameter values are reconfigured.

Keywords: HiBench, BigData, Hadoop, MapReduce, Benchmark, Spark

Introduction

Hadoop [1] has become a very popular platform in the IT industry and academia for its ability to handle large amounts of data, along with extensive processing and analysis facilities. Different users produce these large datasets, and most of data are unstructured, increasing the requirements for memory and I/O. Besides, the advent of many new applications and technologies brought much larger volumes of complex data, including social media, e.g., Facebook, Twitter, YouTube, online shopping,

machine data, system data, and browsing history [2]. This massive amount of digital data becomes a challenging task for the management to store, process, and analyze.

The conventional database management tools are unable to handle this type of data [3]. Big data technologies, tools, and procedures allowed organizations to capture, process speedily, and analyze large quantities of data and extract appropriate information at a reasonable cost.

Several solutions are available to handle this problems [4]. Distributed computing is one possible solution considered as the most efficient and fault-tolerant method for companies to store and process massive amounts of data. Among this new group of tools, MapReduce and Spark are the most commonly used cluster computing tools. They provide users with various functions using simple application programming interfaces (API). MapReduce is a framework used for distributed computing used for parallel processing and designed purposely to write, read, and process bulky amounts of data [1, 5, 6]. This data processing framework is comprised of three stages: Map phase, Shuffle phase and Reduce phase. In this technique, the large files are divided into several small blocks of equal sizes and distributed across the cluster for storage. MapReduce and Hadoop distributed file systems (HDFS) are core parts of the Hadoop system, so computing and storage work together across all nodes that compose a cluster of computers [7].

Apache Spark is an open-source cluster-computing framework [8]. It is designed based on the Hadoop and its purpose is to build a programing model that "fits a wider class of applications than MapReduce while maintaining the automatic fault tolerance" [9]. It is not only an alternative to the Hadoop framework but it also provides various functions to process real streaming data. Apart from the map and reduce functions, Spark also supports MLib1, GraphX, and Spark streaming for big data analysis. Hadoop MapReduce processing speed is slow because it requires accessing disks for reads and writes. On the other hand, Spark uses memory to store data reducing the read/write cycle [1]. In this paper, we have addressed the above mentioned critical challenges. According to our knowledge, none of the previous works have addressed those challenges. Our proposed work will help the system administrators and researchers to understand the system behavior when processing large scale data sets. The main contributions of this paper are as follows:

- We introduced a comprehensive empirical performance analysis between MapReduce and Spark frameworks by correlating resource utilization, splits size, and shuffle behavior parameters. As per our knowledge, few previous studies have presented information regarding that. Considering this point, the authors have focused on a comprehensive study about various parameters impact with large data set instead of a large number of workloads.
- We accomplished comprehensive comparison work between Hadoop and Spark where large scale datasets (600 GB) are used for the first time. The experiments present the various aspects of cluster performance overhead. We applied two Hibenchmark workloads to test the efficiency of the system under MapReduce and Spark, where the data sets are repeatedly changing.

- We selected several parameters covering different aspects of system behavior. Multiple parameters are used to tune job performance. The results of the analysis will facilitate job performance tuning and enhance the freedom to modify the ideal parameters to enhance job efficiency.
- We measured the scalability of the experiment by repeating the experiment three times, getting the average execution time for each job. Besides, we investigate the system execution time, maximum sustainable throughput and speedup.
- We used a real cluster capable of handling large scale data set (600 GB) with benchmarking tools for a comprehensive evaluation of MapReduce and Spark.

The remainder of the paper is organized as follows: "Related work" section presents a critical review of related research works, and then describes Hadoop and Spark systems. The difference between Hadoop and Spark is explained in "Difference between Hadoop and Spark" section. The experimental setup is presented in "Experimental setup" section. In "The parameters of interest and tuning approach" section, we explain the chosen parameters and tuning approach. "Results and discussion" section presents the performance analysis of the results and finally, we conclude in "Conclusion" section.

Related work

Shi et al. [10] proposed two profiling tools to quantify the performance of the MapReduce and Spark framework based on a micro-benchmark experiment. The comparative study between these frameworks are conducted with batch and iterative jobs. In their work, the authors consider three components: shuffle, executive model, and caching. The workloads, Wordcount, k-means, Sort, Linear Regression, and PageRank, are chosen to evaluate the system behavior based on CPU bound, disk-bound, and network bound [11]. They disabled map and reduce function for all workloads apart of a Sort. For the Sort, the reduce task is configured up to 60 map tasks, and the reduce task conFigured to 120. The map output buffer is allocated to 550 MB to avoid additional spills for sorting the map output. Spark intermediate data are stored in 8 disks where each worker is configured with four threads. The authors claim that Spark is faster than MapReduce when WordCount runs with different data sets (1 GB, 40 GB, and 200 GB). The TeraSort is used by sort-by-key() function. They have found that Spark is faster than MapReduce when the data set is smaller (1 GB), but Mapreduce is nearly two times faster than Spark when the data set is of bigger sizes (40 GB or 100 GB). Besides, Spark is one and a half times faster than MapReduce with machine learning workloads such as K-means and Linear Regression. It is claimed that in a subsequent iteration, Spark is five times faster than MapReduce due to the RDD caching and Spark-GraphX is four times faster than MapReduce.

Li et al. [12] proposed a spark benchmarking suite [13], which significantly enhances the optimization of workload configuration. This work has identified the distinct features of each benchmark application regarding resource consumption, the data flow, and the communication pattern that can impact the job execution time. The applications are characterized based on extensive experiments using synthetic data sets. There are ten different workloads such as Logistic Regression, Support Vector Machine, Matrix Factorization, Page Rank, Tringle Count, SVD++, Hive, RDD Relation, Twitter, and

PageView used with different input data sizes. An eleven nodes virtual cluster is used to analyze the performance of the workloads. The workload analysis is carried out concerning CPU utilization, memory, disk, and network input/output consumption at the time of job execution. They have found that most of the workloads spend more than 50% execution time for MapShuffle-Tasks except logistic regression. They concluded that the job execution time could be reduced while increasing task parallelism to leverage the CPU utilization fully.

Thiruvathukal et al. [14] have considered the importance and implication of the language such as Python and Scala built on the Java Virtual Machine (JVM) to investigate how the individual language affects the systems' overall performance. This work proposed a comprehensive benchmarking test for Massage Passing Interface (MPI) and cloud-based application considering typical parallel analysis. The proposed benchmark techniques are designed to emulate a typical image analysis. Therefore, they presented one mid-size (Argonne Leadership Computing Facility) cluster with 126 nodes, which run on COOLEY [14] and a large scale supercomputer (Cray XC40 supercomputer) cluster with a single node which runs on THETA [14]. Significantly, they have increased some important Spark parameters (Spark driver memory, and executor memory) values as per the machine resource. They have recommended that COOLEY and THETA frameworks are be beneficial for immediate research work and high-performance computing (HPC) environments.

Marcue et al. [15] present the comparative analysis between Spark and Flink frameworks for large scale data analysis. This work proposed a new methodology for iterative workloads (K-Means, and Page Rank) and batch processing workloads (WordCount, Grep, and TeraSort) benchmarking. They considered four most important parameters that impact scalability, resource consumption, and execution time. Grid 5000 [16] has used upto 100 nodes cluster deploying Spark and Flink. They have recommended that Spark parameter (i.e., parallelism and partitions) configuration is sensitive and depends on data sets, while the Flink is highly extensive memory oriented.

Samadi et al. [7] has investigated the criteria of the performance comparison between Hadoop and Spark framework. In his work, for an impartial comparison, the input data size and configuration remained the same. Their experiment used eight benchmarks of the HiBench suite [13]. The input data was generated automatically for every case and size, and the computation was performed several times to find out the execution time and throughput. When they deployed microbenchmark (Short and TeraSort) on both systems, Spark showed higher involvement of processor in I/Os while Hadoop mostly processed user tasks. On the other hand, Spark's performance was excellent when dealing with small input sizes, such as micro and web search (Page Rank). Finally, they concluded that Spark is faster and very strong for processing data in-memory while Hadoop MapReduce performs maps and reduces function in the disk.

In another paper, Samadi et al. [9] proposed a virtual machine based on Hadoop and Spark to get the benefit of virtualization. This virtual machine's main advantage is that it can perform all operations even if the hardware fails. In this deployment, they have used Centos operating system built a Hadoop cluster based on a pseudo-distribution mode with various workloads. In their experiments, they have deployed the Hadoop machine on a single workstation and all other demos on its JVM. To justify the big data

framework, they have presented the results of Hadoop deployment on Amazon Elastic Computing (EC2). They have concluded that Hadoop is a better choice because Spark requires more memory resources than Hadoop. Finally, they have suggested that the cluster configuration is essential to reduce job execution time, and the cluster parameter configuration must align with Mappers and Reducers.

The computational frameworks, namely Apache Hadoop and Apache Spark, were investigated by [17]. In this investigation, the Apache webserver log file was taken into consideration to evaluate the two frameworks' comparative performance. In these experiments, they have used Okeanos's virtualized computing resources based on infrastructures as a Service (IaaS) developed by the Greek Research and Technology Network [17]. They proposed a number of applications and conducted several experiments to determine each application's execution time. They have used various input files and the slave nodes to find out the execution time. They have found that the execution time is proportional to the input data size. They have concluded that the performance of Spark is much better in most cases as compared to Hadoop.

Satish and Rohan [18] have shown a comparative performance study between Hadoop MapReduce and Spark-based on the K-means algorithm. In this study, they have used a specific data set that supports this algorithm and considered both single and double nodes when gathering each experiment's execution time. They have concluded that the Spark speed reaches up to three times higher than the MapReduce, though Spark performance heavily depends on sufficient memory size [19].

Lin et al. [20] have proposed a unified cloud platform, including batch processing ability over standalone log analysis tools. This investigation has considered four different frameworks: Hadoop, Spark, and warehouse data analysis tools Hive and Shark. They implemented two machine learning algorithms (K-means and PageRank) based on this framework with six nodes to validate the cloud platform. They have used different data sizes as inputs. In the case of K-means, as the data size increased and exceed memory size, the latency schedule and overall Spark performance degraded. However, the overall performance was still six times higher than Hadoop on average. On the other hand, Shark shows significant performance improvement while using queries directly from disk.

Petridis et al. [21] have investigated the most important Spark parameters shown in Table 4 and given a guideline to the developers and system administrators to select the correct parameter values by replacing the default parameter values based on trial-and-error methodology. Three types of case studies with different categories such as Shuffle Behavior, Compression and Serialization, and Memory Management parameters were performed in this study. They have highlighted the impact of memory allocation and serialization when the number of cores and default parallelism values change. Therefore, there are 12 parameters chosen with three benchmarking applications: sort-by-key, shuffling, and k-means. The sort-by-key experiments used both 1 million and 1 billion key-values of lengths 10 and 90 bytes and the optimal degree of partition is set to 640. The Hash performance is increased to 127 s, which is 30 s faster than the default parameter, and shuffle.file.buffer is increased by 140 s. The rest of the parameters do not play any important role in improving the performance. For another Shuffling experiment, they used a 400 GB dataset. The Hash shuffle performance is degraded by 200 s, and

Tungsten-Sort speed is increased by 90 s. By decreasing the buffer size from 32 to 15 KB, the system performance was degraded by about 135s, which is more than 10% from the primary selection. For K-means, they used two sizes of data input (100 MB and 200 MB). They have not found significant k-means performance improvement by changing the parameters. Therefore, they have concluded that based on their methodology, the speedup achievement is tenfold. However, the main challenges of tuning Hadoop and Spark configuration parameters are due to the complicated behavior of distributed large scale systems while the parameter selection is not always trivial for the system administrators. Inappropriate combination of parameter values can affect the overall system performance. Inappropriate combination of parameter values can affect the overall system performance.

The published literature in Table 1 presents some empirical studies. None of these studies have considered larger data sizes (600 GB), more parameters, and real clusters. In our study, we chose a conventional trial-and-error approach [21], larger data set, and 18 important parameters (listed in Tables 3 and 4) from resource utilization, input splits, and shuffle category.

Difference between Hadoop and Spark

Hadoop [22] is a very popular and useful open-source software framework that enables distributed storage, including the capability of storing a large amount of big datasets across clusters. It is designed in such a way that it can scale up from a single server to thousands of nodes. Hadoop processes large data concurrently and produces fast

Table 1 Published related work

Author's	Date	Workloads	Data size	Parameters	Hardware
Lin et al. [20]	2013	K-means PageRank	10,000 to 20 mil points 1 mil to 10 mil points	Log analysis	Nodes—6, 2 CPU cores 4 GB memory per node Nodes—4, 16 CPU cores 48 GB memory per node
Satish and Rohan [18]	2015	K-means	62–1240 MB	Default	Virtual machine Nodes—2, 4 GB RAM and 500 GB (HD)
Yasir Samadi et al. [7]	2016	Micro Benchmarks Web Search SQL Machine Learning	18–328 MB 5000 to 12 * 10e4 pages	3	Virtual machine Disk (SDD)—40 GB
Petridis et al. [21]	2017	K-means shuffling and sort-by-Key	400 GB	12	Barcelona supercomputing center
Mavridis et al. [17]	2017	Spark SQL and Spark Hive	1.1 GB, 1.5 GB and 11 GB	Log analysis	Virtual machine—6 Memory—8 GB Master node—8 cores Salve node—4 cores
Yasir Samadi et al. [9]	2018	Micro Benchmarks Web Search SQL Machine Learning	1 GB, 5 GB and 8 GB	3	Virtual machine Disk(SDD)—40 GB
Proposed experiments	2020	WordCount and TeraSort	50–600 GB	18	SNCC, Production Cluster CPU cores—80 Total Storage—60 TB Master node—1 Slaves nodes—9

results. With Hadoop, the core parts are Hadoop Distributed File System (HDFS) and MapReduce.

HDFS [23] splits the files into small pieces into blocks and saves them into different nodes. There are two kinds of nodes on HDFS: data-nodes (worker) and name-nodes (master nodes) [24, 25]. All the operations, including delete, read, and write, are based on these two types of nodes. The workflow of HDFS is like the following flow: firstly, the name-node asks for access permission. If accepted, it will turn the file name into a list of HDFS block IDs, including the files and the data-nodes that saved the blocks related to that file. The ID list will then be sent back to the client, and the users can do further operations based on that.

MapReduce [26] is a computing framework that includes two operations: Mappers and Reducers. The mappers will process files based on the map function and transfer them into the new key-value pairs [27]. Next, the new key-value pairs are assigned to different partitions and sorted based on their keys. The combiner is optional and can be recognized as a local reduces operation which allows counting the values with the same key in advance to reduce the I/O pressure. Finally, partitions will divide the intermediate key-value pairs into different pieces and transfer them to a reducer. MapReduce needs to implement one operation: shuffle. Shuffle means transferring the mapper output data to the proper reducer. After the shuffle process is finished, the reducer starts some copy threads (Fetcher) and obtains the output files of the map task through HTTP [28]. The next step is merging the output into different final files, which are then recognized as reducer input data. After that, the reducer processes the data based on the reduced function and writes the output back to the HDFS. Figure 1 depicts a Hadoop MapReduce architecture.

Spark became an open-source project from 2010. Zahari has developed this project at UC Berkely's AMPLab in 2009 [4, 29]. Spark offers numerous advantages for developers to build big data applications. Spark proposed two important terms: Resilient Distributed Datasets (RDD) and Directed Acyclic Graph (DAG). These two techniques work together perfectly and accelerate Spark up to tens of times faster than Hadoop under certain circumstances, even though it usually only achieves a performance two to three times more quickly than MapReduce. It supports multiple sources that have a fault tolerance mechanism that can be cached and supports parallel operations. Besides, it can represent a single dataset with multiple partitions. When Spark runs on the Hadoop cluster, RDDs will be created on the HDFS in many formats supported by Hadoop, likewise text and sequence files. The DAG scheduler [30] system expresses the dependencies of RDDs. Each spark job will create a DAG and the scheduler will drive the graph into the different stages of tasks then the tasks will be launched to the cluster. The DAG will be created in both maps and reduce stages to express the dependencies fully. Figure 2 illustrates the iterative operation on RDD. Theoretically, limited Spark memory causes the performance to slow down.

Experimental setup

Cluster architecture

In the last couple of years, many proposals came from different research groups about the suitability of Hadoop and Spark frameworks when various types of data

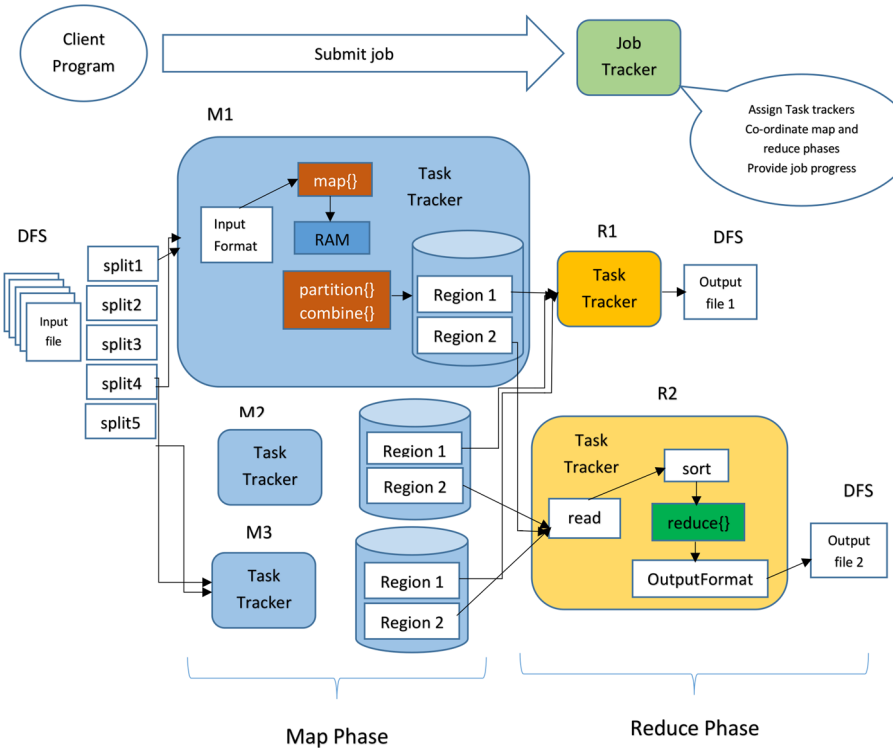


Fig. 1 Hadoop MapReduce architecture [1]

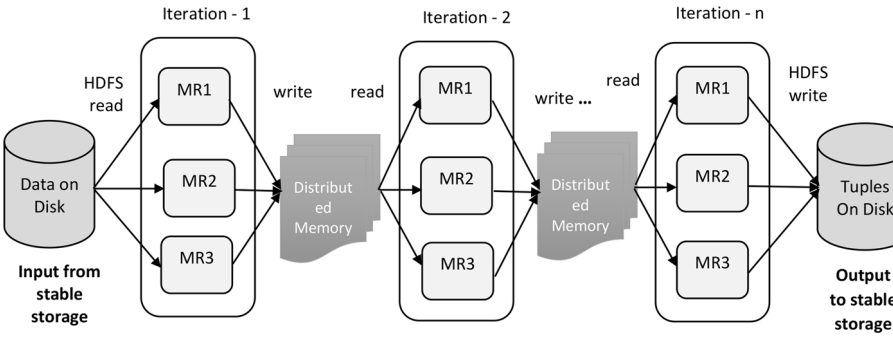


Fig. 2 Spark workflow [31]

of different sizes are used as input in different clusters. Therefore, it becomes necessary to study the performance of the frameworks and understand the influence of various parameters. For the experiments, we will present our cluster performance based on MapReduce and Spark using the HiBench suite [23, 23]. In particular, we have selected two Hibench workloads out of thirteen standard workloads to represent the two types of jobs, namely WordCount (aggregation job) [32], and TeraSort (shuffle job) [33] with large datasets. We selected both the workloads because of their complex characteristics to study how efficiently both the workloads analyze the cluster performance by correlating MapReduce and Spark function with a combination of groups of parameters.

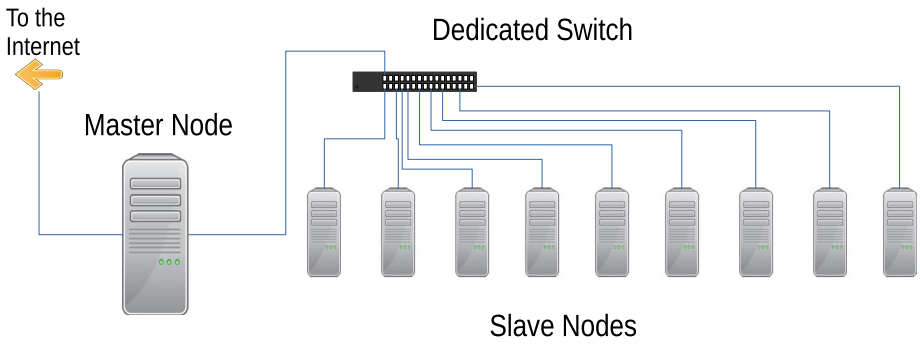


Fig. 3 Hadoop cluster nodes

Table 2 Experimental Hadoop cluster

Server configuration	Processor	2.9 GHz
	Main Memory	64 GB
	Local Storage	10 TB
Node configuration	CPU	Intel(R) Xeon(R) CPU E3-1231 v3 @ 3.40GHz
	Main Memory	32 GB
	Number of Nodes	10
	Local Storage	6 TB each, 60 TB total
	CPU cores	8 each, 80 total
Software	Operating System	Ubuntu 16.04.2 (GNU/Linux 4.13.0-37-generic×86 64)
	JDK	1.7.0
	Hadoop	2.4.0
	Spark	2.1.0
Workload	Micro Benchmarks	WordCount, and TeraSort

Hardware and software specification

The experiments were deployed in our own cluster. The cluster is configured with 1 master and 9 slaves nodes which is presented in Fig. 3. The cluster has 80 CPU cores and 60 TB local storage. The implemented hardware is suitable for handling various difficult situations in Spark and MapReduce.

The detailed Hadoop cluster and software specifications are presented in Table 2. All our jobs run in Spark and MapReduce. We have selected Yarn as a resource manager, which can help us monitor each working node's situation and track the details of each job with its history. We have used *Apache Ambari* to monitor and profile the selective workloads running on Spark and MapReduce. It supports most of the Hadoop components, including HDFS, MapReduce, Hive, Pig, Hbase, Zookeeper, Sqoop, and Hcatalog" [34]. Besides, Ambari supports the user to control the Hadoop cluster on three aspects, namely provision, management, and monitoring.

Table 3 Hadoop configuration parameters

Configuration parameters category	Hadoop	Tuned values
Resource utilization	mapreduce.reduce.memory	8 GB
	mapred.reduce.task	16,384 MB, 25,600 MB
	mapreduce.reduce.cpu.vcores	4
Input split	mapred.min.split.size, mapred.max.split.size	128 MB (default), 256 MB, 512 MB, 1024 MB
Shuffle	i/o.sort.mb	25, 50, 75, 100
	i/o.sort.factor	512, 1024, 1536, 2047
	mapreduce.reduce.shuffle.parallelcopies	50, 100, 150, 200
	mapreduce.task.io.sort.factor	15, 30, 45, 60

Table 4 Spark configuration parameters

Configuration parameters category	Spark	Tuned values
Resource utilization	num-executors	50
	executor-cores	4
	executor-memory	8 GB
Input split	spark.hadoop.MapReduce.input .fileinputformat.split.minsize	128 MB (default), 256MB, 512MB, 1024MB
Shuffle	spark.shuffle.file.buffer	16 k, 32 k (default), 48 k, 64 k
	spark.reducer.maxSizeInFlight	32 M, 48 M (default), 64 M, 96 M
	spark.hadoop.dfs.replication	1
	spark.default.parallelism	80, 100, 200, 300

Workloads

As stated above, in this study we chose two workloads for the experiments [32, 33]:

WordCount: The wordCount workload is map-dependent, and it counts the number of occurrences of separate words from text or sequence file. The input data is produced by *RandomTextWriter*. It splits into each word by using the map function and generates intermediate data for the reduce function as a key-value [35]. The intermediate results are added up, generating the final word count by the reduce function.

TeraSort: The TeraSort package was released by Hadoop in 2008 [36] to measure the capabilities of cluster performance. The input data is generated by the *TeraGen* function which is implemented in Java. The TeraSort function does the sorting using the MapReduce, and the TeraValidate function is used to validate the output of the sorted data. For both workloads, we used up to 600 GB of synthetic input data generated using a string concatenation technique.

The parameters of interest and tuning approach

Tuning parameters in Apache Hadoop and Apache Spark is a challenging task. We want to find out which parameters have important impacts on system performance. The configuration of the parameters needs to be investigated according to work-load,

data size, and cluster architecture. We have conducted a number of experiments using Apache Hadoop and Apache Spark with different parameter settings. For this experiment, we have chosen the core MapReduce and Spark parameter setting from resource utilization, input splits and shuffle groups. The selected tuned parameters with their respective tuned values on the map-reduce and Spark category are shown in Tables 3 and 4.

Results and discussion

In this section, the results obtained after running the jobs are evaluated. We have used synthetic input data and used the same parameter configuration for a realistic comparison. Each test was repeated three times, and the average runtime was plotted in each graph. For both frameworks, we show the execution time, throughput, and speedup to compare the two frameworks and visualize the effects of changing the default parameters.

Execution time

The execution time is affected by the input data sizes, the number of active nodes, and the application types. We have fixed the same parameters for the fair comparative analysis, such as the number of executors to 50, executor memory to 8 GB, executor cores to 4.

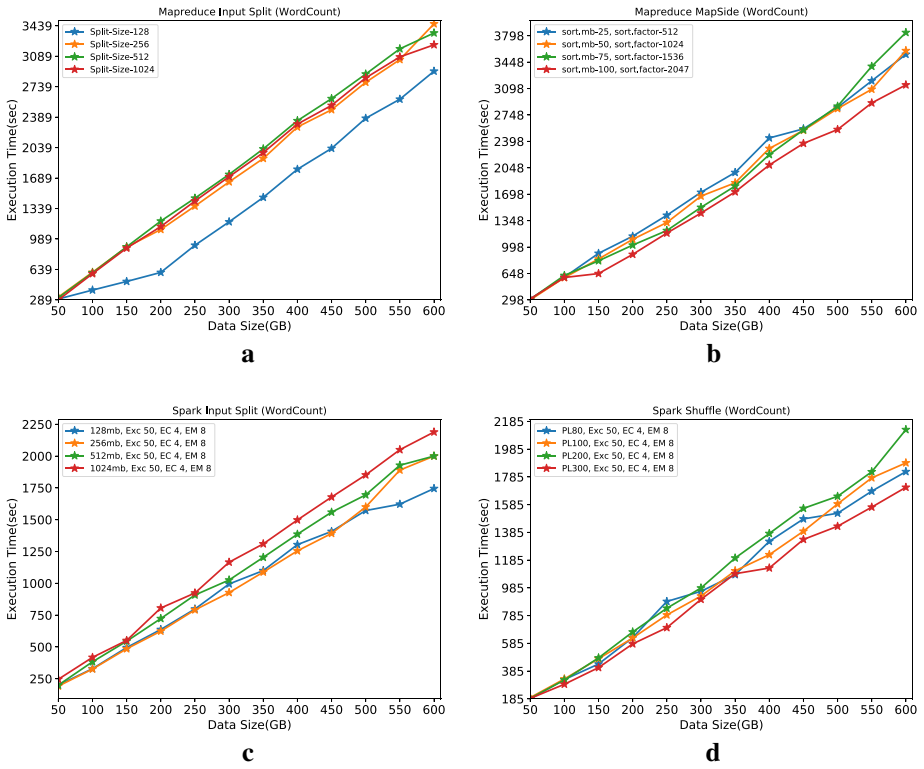


Fig. 4 The performance of the WordCount application with a varied number of input splits and shuffle tasks

Figure 4a, b show how MapReduce and Spark execution time depend on the datasets' size and the different input splits and shuffle parameters. The execution time of MapReduce WordCount workload with the default input split size (128 MB) and shuffle parameter (*sort.mb* 100, *sort.factor* 2047) obtained better execution time for entire data sizes compared to other parameters. Hadoop Map and Reduce function behave better because of their faster execution time and overlooked container initialization overhead for specific workload types. This result suggests that the default parameter is more suitable for our cluster when using data sizes from 50 to 600 GB.

In Fig. 4c the default input splits of Spark is 128 MB. Previously, we have mentioned that the number of executors, executor memory, and executor cores are fixed. From the above Fig. 4c, we see that the execution time of input split size 256 MB outperforms the default set up until 450 GB data sizes. In fact, the default splits size (128 MB) is more efficient when the data size is larger than the 450 GB. Notably, we can see that the default parameter shows better execution performance when the data set reaches 500 GB or above. The new parameter values can improve the processing efficiency by 2.2% higher than the default value (128 MB). Table 5 presents the experimental data of WordCount workload between MapReduce and Spark while the default parameters are changing.

For the Spark shuffle parameter, we have chosen the default serializer, the *(JavaSerializer)* because of the simplicity and easy control of the performance of the serialization [37]. In this category, the serializer is PL100 object [37]. We can see from Fig. 4d that the improvement rate is significantly increased when we set the PL value to 300. It is evident that the best performance is achieved for sizes larger than 400 GB. Also, it shows that when tuning the PL value to 300, the system can achieve a 3% higher improvement for the rest of the data sizes. Consequently, we can conclude that input splits can be considered an important factor in enhancing Spark WordCount jobs' efficiency when executing small datasets.

Figure 5a is comparing MapReduce TeraSort workloads based on input splits that include default parameters. In this analysis, we have set (*Red_Task* and *InSp*) value fixed with default split size 128 MB. We have changed the parameter values and tested whether the splits' size can keep the impact on the runtime. So, for this reason, we have selected three different sizes: 256 MB, 512 MB, and 1024 MB. We have observed that with a split size of 256MB, the execution performance is increased by around 2% in datasets with up to 300 GB. On the contrary, when the data sizes are larger than 300 GB, the default size outperforms split size equals 512 MB. Moreover, we have noticed that the improvement rates are similar when the data sizes are smaller than 200 GB.

Table 5 The best execution time of MapReduce and Spark with WordCount workload

	Split sizes (MB)	Execution time (s)
MapReduce input splits (WordCount)	128	2376
Spark input splits (WordCount)	256	1392
MapReduce shuffle (WordCount)	100	2371
Spark shuffle (WordCount)	300	1334

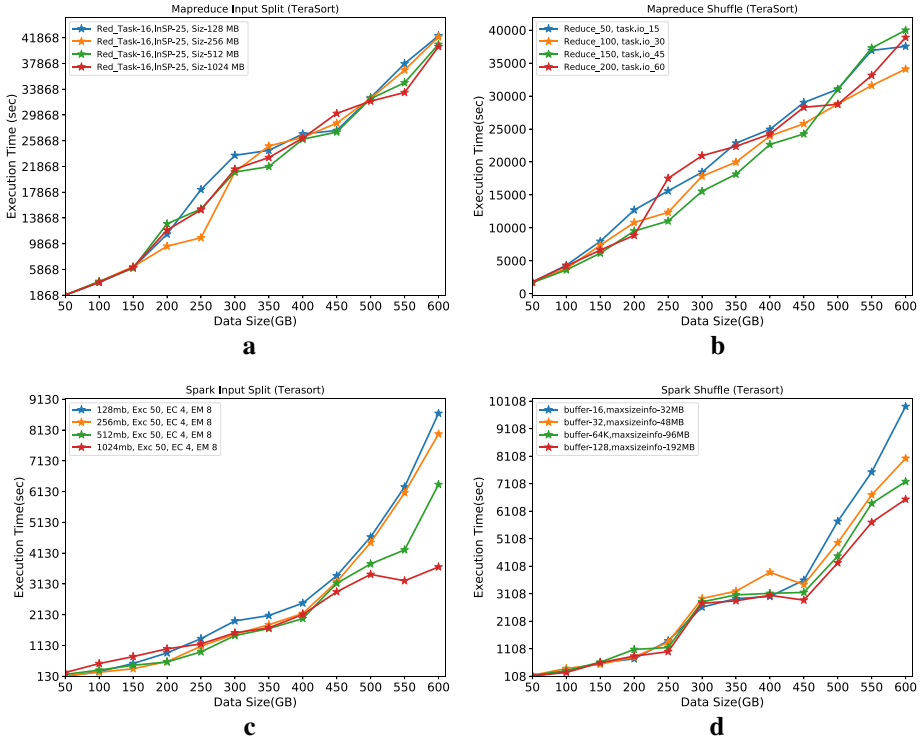


Fig. 5 The performance of the TeraSort application with a varied number of input splits and shuffle tasks

Figure 5b illustrates the execution performance with the MapReduce shuffle parameter for the TeraSort workload. We have seen that the average execution time behaves linearly for sizes up to 450 GB when the parameter change to (*Reduce_150* and *task.io_45*) as compared to the default configuration (*Reduce_100* and *task.io_30*). Besides, We have also noticed that the default configuration is outperforming all other settings when the data sizes are larger than 450 GB. So, we can conclude that by changing the shuffled value, the system execution performance improves by 1%. In general, this is very unlikely that the default size has optimum performance for larger data sizes.

Figure 5c illustrates the Spark input split parameter execution performance analysis for the TeraSort workload. The Spark executor memory, number of executors, and executor memory are fixed while changing the block size to measure the execution performance. Apart from the default block size (128 MB), there are 3 pairs (256 MB, 512 MB, and 1024 MB) of block size is taken into this consideration. Our results revealed that the block size 512 MB and 1024 MB present better runtime for sizes up to 500 GB data size. We have also observed a significant performance improvement achieved by the 1024 block size, which is 4% when the data size is larger than 500 GB. Thus, we can conclude that by adding the input splits block size for large scale data size, Spark performance can be increased.

Figure 5d shows Spark shuffle behaviour performance for TeraSort workloads. We have taken two important default parameters (*buffer* = 32, *spark.reducer.maxSizeIn Flight* = 48 MB) into our analysis. We have found that when the buffer and *maxSizeIn-Flight* are increased by 128 and 192, the execution performance increased proportionally

Table 6 The best execution time of MapReduce and Spark with Terasort workload

	Split sizes (MB)	Execution time (s)
MapReduce input splits (TeraSort)	256	21,014
Spark input splits (TeraSort)	512 & 1024	3780 & 3439
MapReduce shuffle (TeraSort)	150 & 45	24,250
Spark shuffle (TeraSort)	128 & 192	6540

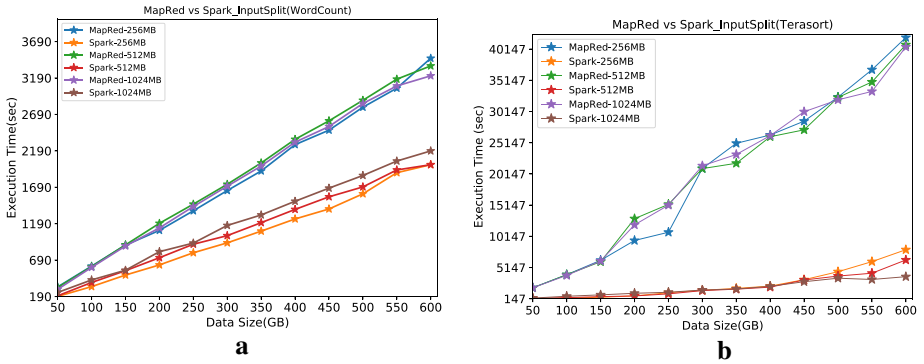


Fig. 6 The comparison of Hadoop and Spark with WordCount and TeraSort workload with varied input splits and shuffle tasks

up to 600 GB data sizes. Our results show that the default execution is equal, with a tested value of up to 200 GB data sizes. The possible reason for this performance improvement is the larger number of splits size for different executors. Table 6 presents the experimental data of the TeraSort workload between MapReduce and Spark, while the default parameters are changing.

Figure 6a illustrates the comparison between Spark and MapReduce for WordCount and TeraSort workloads after applying the different input splits. We have observed that Spark with WordCount workloads shows higher execution performance by more than 2 times when data sizes are larger than 300 GB for WordCount workloads. For the smaller data sizes, the performance improvement gap is around ten times. Figure 6 shows a TeraSort workload for MapReduce and Spark. We can see that Spark execution performance is linear and proportionally larger as the data size increase. Also, we noticed that the runtime for MapReduce jobs are not as linear in relation to the data size as Spark jobs. The possible reason could be unavoidable job action on the clusters and as a result that the dataset is larger than the available RAM. So, we conclude that MapReduce has slower data sharing capabilities and a longer time to the read-write operation than Spark [4].

Throughput

The throughput metrics are all in MB per second. For this analysis, we only considered the best results from each category. We have observed that MapReduce throughput performance for the TeraSort workload is decreasing slightly as the data size crosses beyond 200 GB. Besides, for the WordCount workload, the MapReduce throughput is almost linear. For the Spark TeraSort workload, it can be observed that

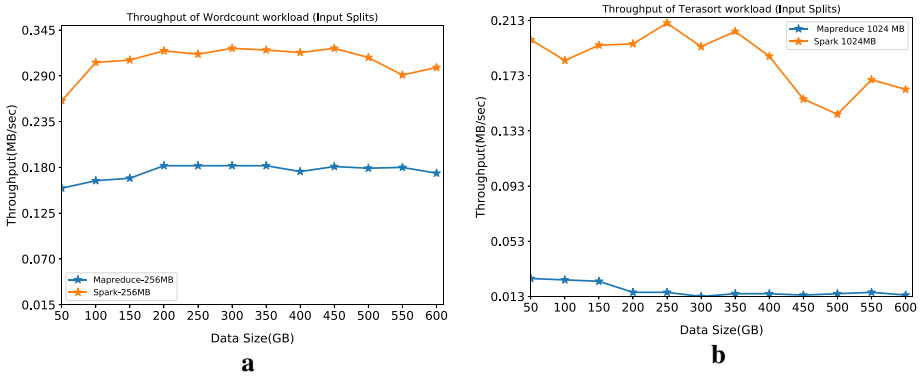


Fig. 7 Throughput of WordCount and TeraSort workload

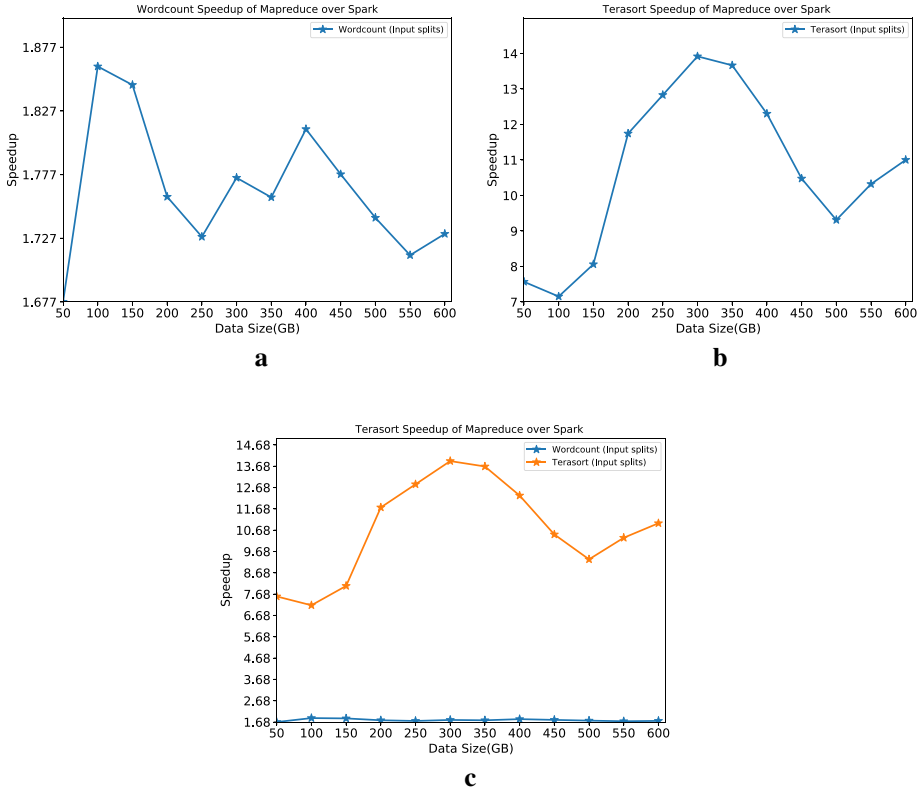


Fig. 8 Spark over MapReduce speedup on input splits and shuffle

the throughput is not constant, but for the WordCount workload, the throughput is almost constant. In this analysis, the main focus was to present the throughput difference between WordCount and TeraSort workload for MapReduce and Spark. We found that WordCount workload remains almost stable for most of the data sizes, and concerning the TeraSort workload, MapReduce remain stable than Spark (see Fig. 7).

Speedup

Figure 8a–c show the Spark's speed up compared to MapReduce. Figure 8a, b depicts individual workload speedup. The best results are taken into this consideration from each category in order to get a speedup. From the above figures, we can see that as the data size increases, WordCount workload speedup decreases with some non-linearity. Besides, we can see that the TeraSort speedup decreases when data reaches sizes larger than 300 GB. Notably, as the data size increases to more than 500GB for both workloads, the speedup starts to increase. Figure 8c illustrates the speedup comparison between the workloads. It can be seen that the TeraSort workload outperforms WordCount workload and achieves an all-time maximum speedup of around 14 times. The literature presents that Spark is up to ten times faster than Hadoop under certain circumstances and in normal conditions, and it only achieves a performance two to three times faster than MapReduce [38]. However, this study found that Spark performance is degraded when the input data size is big.

Conclusion

This article presented the empirical performance analysis between Hadoop and Spark based on a large scale dataset. We have executed WordCount and Terasort workloads and 18 different parameter values by replacing them with default set-up. To investigate the execution performance, we have used trial-and-error approach for tuning these parameters performing number of experiments on nine node cluster with a capacity of 600 GB dataset. Our experimental results confirm that both Hadoop and Spark systems performance heavily depends on input data size and right parameter selection and tuning. We have found that Spark has better performance as compared to Hadoop by two times with WordCount work load and 14 times with Tera-Sort workloads respectively when default parameters are tuned with new values. Further more, the throughput and speedup results show that Spark is more stable and faster than Hadoop because of Spark data processing ability in memory instead of store in disk for the map and reduced function. We have also found that Spark performance degraded when input data was larger.

As future work, we plan to add and investigate 15 HiBench workloads, consider more parameters under resource utilization, parallelization, and other aspects, including practical data sets. The main focus would be to analyze the job performance based on auto-tuning techniques for MapReduce and Spark when several parameter configurations replace the default values.

Acknowledgements
The authors acknowledge Sibgat Bazai for his valuable suggestions.

Authors' contributions
NA was the main contributor of this work. He has done an initial literature review, data collection, experiments, prepare results, and drafted the manuscript. ALCB and TS deployed and configured the physical Hadoop cluster. ALCB also worked closely with NA to review, analyze, and manuscript preparation. TS and MAR helped to improve the final paper. All authors read and approved the final manuscript.

Funding
This work was not funded.

Availability of data and materials
The data that support the findings of this study are available from the corresponding author upon reasonable request.

Ethics approval and consent to participate
Not applicable.

Consent for publication
Not applicable.

Competing interests
The authors declare that they have no competing interests.

Author details
[1] School of Natural and Computational Sciences, Massey University, Albany, Auckland 0745, New Zealand. [2] Department of Mechanical and Electrical Engineering, Massey University, Auckland 0745, New Zealand.

Received: 30 July 2020 Accepted: 26 November 2020
Published online: 14 December 2020

References

1. Apache Hadoop Documentation 2014. http://hadoop.apache.org/. Accessed 15 July 2020.
2. Verma A, Mansuri AH, Jain N. Big data management processing with hadoop mapreduce and spark technology: A comparison. In: 2016 symposium on colossal data analysis and networking (CDAN). New York: IEEE; 2016. p. 1–4.
3. Management Association IR. Big Data: concepts, methodologies, tools, and applications. Hershey: IGI Global; 2016.
4. Zaharia M, Chowdhury M, Das T, Dave A, Ma J, Mccauley M, Franklin M, Shenker S, Stoica I. Fast and interactive analytics over hadoop data with spark. Usenix Login. 2012;37:45–51.
5. Dean J, Ghemawat S. Mapreduce: simplified data processing on large clusters. Commun ACM. 2008;51(1):107–13.
6. Wang G, Butt AR, Pandey P, Gupta K. Using realistic simulation for performance analysis of mapreduce setups. In: Proceedings of the 1st ACM workshop on large-scale system and application performance; 2009. p. 19–26.
7. Samadi Y, Zbakh M, Tadonki C. Comparative study between hadoop and spark based on hibench benchmarks. In: 2016 2nd international conference on cloud computing technologies and applications (CloudTech). New York: IEEE; 2016. p. 267–75.
8. Ahmadvand H, Goudarzi M, Foroutan F. Gapprox: using gallup approach for approximation in big data processing. J Big Data. 2019;6(1):20.
9. Samadi Y, Zbakh M, Tadonki C. Performance comparison between hadoop and spark frameworks using hibench benchmarks. Concurr Comput Pract Exp. 2018;30(12):4367.
10. Shi J, Qiu Y, Minhas UF, Jiao L, Wang C, Reinwald B, Özcan F. Clash of the titans: mapreduce vs. spark for large scale data analytics. Proc VLDB Endow. 2015;8(13):2110–211.
11. Veiga J, Expósito RR, Pardo XC, Taboada GL, Tourifio J. Performance evaluation of big data frameworks for large-scale data analytics. In: 2016 ieee international conference on Big Data (Big Data). New York: IEEE; 2016. p. 424–31.
12. Li M, Tan J, Wang Y, Zhang L, Salapura V. Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform spark. In: Proceedings of the 12th ACM international conference on computing frontiers; 2015. p. 1–8.
13. Wang L, Zhan J, Luo C, Zhu Y, Yang Q, He Y, Gao W, Jia Z, Shi Y, Zhang S. Bigdatabench: a big data benchmark suite from internet services. In: 2014 IEEE 20th international symposium on high performance computer architecture (HPCA). New York: IEEE; 2014. p. 488–99.
14. Thiruvathukal GK, Christensen C, Jin X, Tessier F, Vishwanath V. A benchmarking study to evaluate apache spark on large-scale supercomputers. 2019; arXiv preprint arXiv:1904.11812.
15. Marcu O-C, Costan A, Antoniu G, Pérez-Hernández MS. Spark versus flink: Understanding performance in big data analytics frameworks. In: 2016 IEEE international conference on cluster computing (CLUSTER). New York: IEEE; 2016. p. 433–42.
16. Bolze R, Cappello F, Caron E, Daydé M, Desprez F, Jeannot E, Jégou Y, Lanteri S, Leduc J, Melab N, et al. Grid'5000: a large scale and highly reconfigurable experimental grid testbed. Int J High Perform Comput Appl. 2006;20(4):481–94.
17. Mavridis I, Karatza E. Log file analysis in cloud with apache hadoop and apache spark 2015.
18. Gopalani S, Arora R. Comparing apache spark and map reduce with performance analysis using k-means. Int J Comput Appl. 2015;113(1):8–11.
19. Gu L, Li H. Memory or time: Performance evaluation for iterative operation on hadoop and spark. In: 2013 IEEE 10th international conference on high performance computing and communications & 2013 IEEE international conference on embedded and ubiquitous computing. New York: IEEE; 2013. p. 721–7.
20. Lin X, Wang P, Wu B. Log analysis in cloud computing environment with hadoop and spark. In: 2013 5th IEEE international conference on broadband network & multimedia technology. New York: IEEE; 2013. p. 273–6.
21. Petridis P, Gounaris A, Torres J. Spark parameter tuning via trial-and-error. In: INNS conference on big data. Berlin: Springer; 2016. p. 226–37.
22. Landset S, Khoshgoftaar TM, Richter AN, Hasanin T. A survey of open source tools for machine learning with big data in the hadoop ecosystem. J Big Data. 2015;2(1):24.
23. HiBench Benchmark Suite. https://github.com/intel-hadoop/HiBench. Accessed 15 July 2020.
24. Shvachko K, Kuang H, Radia S, Chansler R. The hadoop distributed file system. In: 2010 IEEE 26th symposium on mass storage systems and technologies (MSST). New York: IEEE; 2010. p. 1–10.
25. Luo M, Yokota H. Comparing hadoop and fat-btree based access method for small file i/o applications. In: International conference on web-age information management. Berlin: Springer; 2010. p. 182–93.
26. Taylor RC. An overview of the hadoop/mapreduce/hbase framework and its current applications in bioinformatics. BMC Bioinform. 2010;11:1.

27. Vohra D. Practical Hadoop ecosystem: a definitive guide to hadoop-related frameworks and tools. California: Apress; 2016.
28. Lee K-H, Lee Y-J, Choi H, Chung YD, Moon B. Parallel data processing with mapreduce: a survey. AcM sIGMoD record. 2012;40(4):11–20.
29. Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I. Spark: cluster computing with working sets. HotCloud. 2010;10:95.
30. Kannan P. Beyond hadoop mapreduce apache tez and apache spark. San Jose State University); 2015. http://www.sjsu.edu/people/robert.chun/courses/CS259Fall2013/s3/F.pdf. Accessed 15 July 2020.
31. Spark Core Programming. https://www.tutorialspoint.com/apache_spark/apache_spark_rdd.htm. Accessed 15 July 2020.
32. Huang S, Huang J, Dai J, Xie T, Huang B. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In: 2010 IEEE 26th international conference on data engineering workshops (ICDEW 2010). New York: IEEE; 2010. p. 41–51.
33. Chen C-O, Zhuo Y-Q, Yeh C-C, Lin C-M, Liao S-W. Machine learning-based configuration parameter tuning on hadoop system. In: 2015 IEEE international congress on big data. New York: IEEE; 2015. p. 386–92.
34. Ambari. https://ambari.apache.org/. Accessed 15 July 2020.
35. Xiang L-H, Miao L, Zhang D-F, Chen F-P. Benefit of compression in hadoop: A case study of improving io performance on hadoop. In: Proceedings of the 6th international asia conference on industrial engineering and management innovation. Berlin: Springer; 2016. p. 879–90.
36. O'Malley O. Terabyte sort on apache hadoop. Report, Yahoo!; 2008. http://sortbenchmark.org/YahooHadoop.pdf. Accessed 15 July 2020.
37. Apache Tuning Spark 1.1.1. https://spark.apache.org/docs/1.1.1/tuning.html. Accessed 15 July 2020.
38. Rathore MM, Son H, Ahmad A, Paul A, Jeon G. Real-time big data stream processing using gpu with spark over hadoop ecosystem. Int J Parallel Progr. 2018;46(3):630–46.

Publisher's Note

Chapter 3

The contents of this chapter are from the following article. In accordance with the Springerlink open access policy, any full text that has been included, is the unmodified accepted article.

Ahmed *et al. J Big Data* (2021) 8:107
https://doi.org/10.1186/s40537-021-00499-7

Journal of Big Data

RESEARCH Open Access

A parallelization model for performance characterization of Spark Big Data jobs on Hadoop clusters

N. Ahmed[1*], Andre L. C. Barczak[1], Mohammad A. Rashid[2] and Teo Susnjak[1]

*Correspondence:
nasim751@yahoo.com
[1] School of Natural and Computational Sciences, Massey University, Albany, Auckland 0745, New Zealand
Full list of author information is available at the end of the article

Abstract

This article proposes a new parallel performance model for different workloads of Spark Big Data applications running on Hadoop clusters. The proposed model can predict the runtime for generic workloads as a function of the number of executors, without necessarily knowing how the algorithms were implemented. For a certain problem size, it is shown that a model based on serial boundaries for a 2D arrangement of executors can fit the empirical data for various workloads. The empirical data was obtained from a real Hadoop cluster, using Spark and HiBench. The workloads used in this work were included WordCount, SVM, Kmeans, PageRank and Graph (Nweight). A particular runtime pattern emerged when adding more executors to run a job. For some workloads, the runtime was longer with more executors added. This phenomenon is predicted with the new model of parallelisation. The resulting equation from the model explains certain performance patterns that do not fit Amdahl's law predictions, nor Gustafson's equation. The results show that the proposed model achieved the best fit with all workloads and most of the data sizes, using the R-squared metric for the accuracy of the fitting of empirical data. The proposed model has advantages over machine learning models due to its simplicity, requiring a smaller number of experiments to fit the data. This is very useful to practitioners in the area of Big Data because they can predict runtime of specific applications by analysing the logs. In this work, the model is limited to changes in the number of executors for a fixed problem size.

Keywords: Big Data, Performance prediction, System configuration, HiBench, Spark

Introduction

Apache Spark [1] is an alternative open-source distributed computing platform of MapReduce [2] for large-scale data processing. Spark introduces Resilient Distributed Data set (RDD) [3] with high fault-tolerance, fast processing speed, and scalability to improve real-time performance. Moreover, Spark offers various data analysis tools and modules such as Spark SQL, MLlib, and Graphs [4]. The execution time of Spark application is a significant factor in measuring real-time processing. Users need to allocate multiple resources, efficient memory allocation, adequate data partition, and an optimized cluster configuration based on the desired execution time. Cluster users

and administrators can benefit from accurate models, which provide a quick prediction for runtime of a certain job.

In recent years, researchers have published works on the prediction of the performance of big data processing platforms such as Spark [5–12]. Virtually all the publications make use of machine learning models to predict runtime and other performance characteristics. However, machine learning models require large sampling sets to work accurately. Moreover, these models are not very good at interpolating performance data if the samples are not dense enough. Also, even though machine learning models can be very effective, they do not necessarily explain why the performance shows a certain pattern [13].

In order to mitigate these issues, we propose a new parallelisation model based on finding a pattern for the parallelisable and the non-parallelisable portions of a generic job. Any algorithm can be parallelised, but not all algorithms can run efficiently in parallel machines such as a cluster. The parallel performance depends mostly on how the algorithm operates.

For example, some algorithms are embarassingly parallel (a term coined in the 90s) [14], meaning that no extra work is needed when the job is parallelised. In this case, the speedup is proportional to the number of processors available. In other cases, the speedup can be superlinear, as in the case of searching algorithms running in parallel. Unfortunately, there are also groups of algorithms that do not present this optimistic speedup.

The main reason for a degraded performance is the fact that the nature of the algorithm requires extra communication and I/O operations that are inherently serial in nature. This was understood by Amdahl in the 60s, when he published his findings with an equation that became known as Amdahl's law [15]. Later, in the 80s, Gustafson observed that Amdahl's law was a special case of performance because Amdahl's assumption was that any job needs a fixed portion of serialised work that cannot be parallelised [16]. Gustafson came up with an alternative assumption that could explain why some of the jobs he was running were performing better (better speedup) than what Amdahl's equation was predicting.

Both Amdahl and Gustafson did not generalised their models to predict the performance for any job, but only for jobs for which their assumptions are true. In this paper, our main target is to understand the relationship between execution time (runtime) and the number of executors used in Spark jobs. To the best of our knowledge, none of the previous studies have come up with a simple model that can fit the data for different workloads. Indeed, the proposed technique will significantly help researchers, cluster users, operators, and system administrators. Moreover, the proposed model can be implemented in any large scale Hadoop physical cluster, either in industries or academic research. This would be helpful for system administrators, system architects, and data engineers to predict the possible system parameters, specifically the number of executors, for any Spark job on Hadoop physical cluster. In particular, the model can help to find insights about the pattern for the parallelisable and non-parallelisable portions of a generic jobs. The model will present a precise generic equation for a cluster relying on a very limited number of experiments. The key contributions of this paper are as follows:

- A very effective model is introduced that can explain various HiBench jobs' performance patterns as a function of the number of executors. The model achieves a good accuracy for different workloads, treating the implementation as a black box, i.e., without any knowledge of the internal workings of communication between the executors or the I/O involved in running the jobs (via HDFS).
- Accomplished extensive experimental work of Spark application on the physical cluster environment. The experiments present the various aspects of cluster performance overheads. We considered five HiBenchmark workloads for testing the system's efficiency, where the fixed data sets are changed with different executors.
- Using the proposed model, we consider the problem and determine the experiment's scalability by repeating the experiment three times, getting the average execution time for each job.

The paper is organised as follows. "Apache Spark environment" section describes the Apache Spark environment. In "Related work" section we review a number of works that are related to the performance prediction of Spark running on a Hadoop cluster. In "Modelling of a 2D plate parallel application" sectiom we propose our model based on a 2D configuration of executors, and discuss the motivation for this model. In "Experimental setup" section the experimental setup is discussed, detailing how we obtained the empirical data. "Findings from the analytical model" section presents several workloads and show how the main equation for the model fits the data. Finally, in "Conclusion" section we present our conclusions with a discussion on the future developments for the model.

Apache Spark environment

Spark offers numerous advantages for developers to build big data applications. Apache Spark proposed two important concepts: Resilient Distributed Datasets (RDD) and Directed Acyclic Graph (DAG) [3]. A new abstraction method called Resilient Distributed Datasets (RDD) is used to increase the data uses efficiently for a wide range of applications. The RDD is designed in such a way that it can provide efficient fault tolerance. For fault tolerance, RDD is used as an interface based on coarse-grained transformations (i.e., map, filter, and join) for various data items. The DAG scheduler [17] system expresses the dependencies of RDDs. Each spark job will create a DAG, and the scheduler will drive the graph into the different stages of tasks, then the tasks will be launched to the cluster. The DAG will be created in both maps and reduce stages to express the dependencies fully. These two techniques work together perfectly and accelerate Spark up to twenty times with iterative application and ten times faster than Hadoop under certain circumstances. In normal conditions, it only achieves a performance two to three times faster than MapReduce. It supports multiple sources that have a fault tolerance mechanism that can be cached and supports parallel operations. Besides, it can represent a single data set with multiple partitions. Spark consists of master and worker nodes where it can hold either single or multiple interactive jobs. When Spark runs on the Hadoop cluster, RDDs will be created on the HDFS in many formats supported by Hadoop, as well as text and sequence files. In Spark, a job is executed into one or multiple physical units, and the jobs are divided into a smaller set of tasks that are on the

stage. A single spark job can trigger a number of jobs that are dependent on the parent stage. So, the submitted job can be executed in parallel. Spark executes submitted jobs in two stages: ShuffleMapStage and ResultStages. The ShuffleMapStage is an intermediate stage where the output data is stored for the input data for the following stages in the DAG. The ResultStages is the final stage of this process that assigns a function on one or multiple partitions of the target RDD. In Spark, executors run on a worker node in the cluster. The executors start their processes once the system receives the input file and continue until the job is completed. In this case, the executors keep themselves active for the entire workload time and use multiple CPU threads for the task parallelly. For any given work, the executor size, numbers, and threads play a vital role in the performance [18]. The block manager acts as a cache storage for a user's program when executors allocate memory storage for the RDDs. Spark runs on Hadoop cluster with Apache YARN (Yet Another Resource Negotiator) [19] as a framework for resource management and job scheduling or monitoring into separate demons and Apache Ambari, an open source tool which manage, monitor and profile the individual workloads running Hadoop cluster. Figure 1 shows a typical Spark cluster architecture.

Related work

In this section, we discuss relevant published works in the area of performance prediction for Hadoop clusters running Spark. A simulation-based prediction model is proposed by Kewen Wang [20]. The model simulates the execution of the main job by using only a fraction of input data and collects execution traces to predict job performance for each execution stage separately. They have proposed a standalone cluster mode on top of the Hadoop Distributed File System (HDFS) with default 64 MB block settings. They

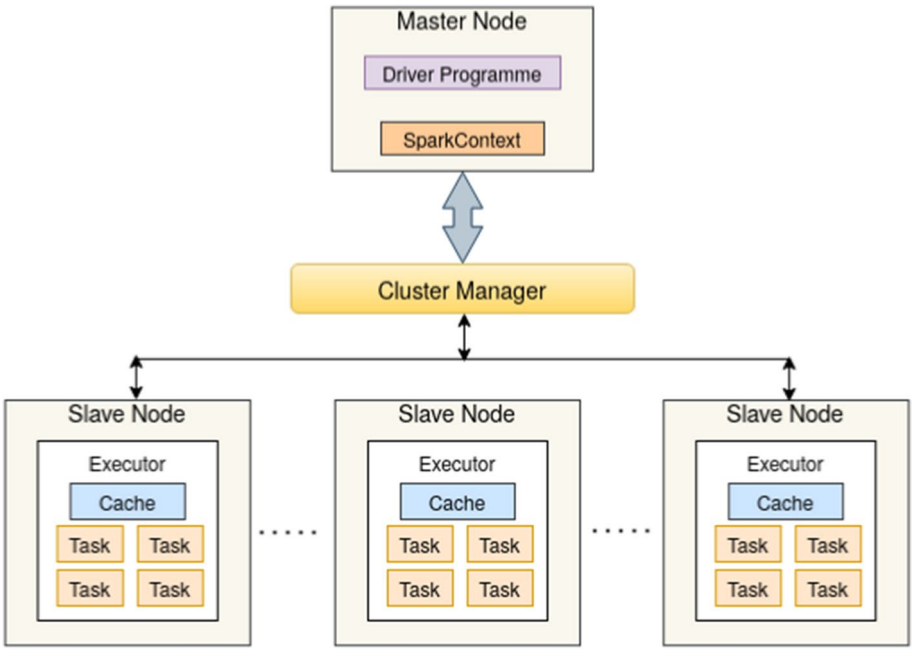


Fig. 1 A typical Spark cluster architecture

have evaluated this framework using four real-world applications and claimed that this model is capable of predicting execution time for an individual stage with high accuracy.

Singhal and Singh [21] addressed the Spark platform's challenges to process huge data sizes. They found that as the data size increases, the Spark performance reduces significantly. To overcome this challenge, they ensured that the system would perform on a higher scale. They proposed two techniques, namely, black box and analytical approaches. In the black-box technique, the Multi Linear Regression (MLR- Quadratic) and Support Vector Machine (SVM) are used to determine the accuracy of the prediction model and the analytical approach to predict an application execution time. They found that Spark parameter selection is very complex to identify the suitable parameters which impact an application execution time for varying data and cluster sizes. Therefore, they carefully selected parameters that could be changed during an application execution time and analyze the performance sensitivity for several parameters, which are very important for the feature selection. In the integrated performance prediction model with an optimization algorithm, the system performance improvement showed 94%. Finally, they summarized that machine learning algorithm requires more resources and data collection time.

Maros [22] conducted a cost-benefit analysis of a supervised machine learning model for Spark performance prediction and compared their results with Ernest [23]. In this investigation, they considered the black box and gray box techniques. For the black box technique, they considered four ML algorithms such as Linear Regression (LR), Decision Tree (DT), Random Forest (RF), and L1-Regularized Linear Regression (RLR). The gray box technique is used to capture the features of the execution time. In this approach, not a single machine learning algorithm outperforms others. To chose the best model, different techniques are required to evaluate the individual scenario.

Hani et al. [24] proposed a methodology based on gray box model for Spark runtime prediction. This model works with white box and black box models, and the models focus not only on impact data size but also on platform configuration settings, I/O overhead, network bandwidth, and allocated resources. This model methodology can predict the runtime by taking the consideration both the previous factors and application parameters. They achieved a high matching accuracy of about 83–94% between average and actual runtime applications. Based on this model methodology, the Spark runtime would be predicted accurately.

Cheng [25] proposed a performance model based on Adaboost at stage-level for Spark runtime prediction. They considered a classic projective sampling and data mining technique such as projective sampling and advanced sampling to reduce the model's overhead. They claimed that projective sampling would offer optimum sample size without any prior assumption between configuration parameters, thus enhancing the entire prediction process's utility.

Gulino [26] proposed a data-driven workflow approach based on DAGs in which the execution time is predicted of Spark operation. In this approach, they combined analytical and machine learning models and trained on small DAGs. They found that prediction accuracy of the proposed approach is better than the black box and gray box technique. Nevertheless, they did not present how this approach will work for iterative and machine learning workloads. This approach only considers SQL type queries.

Gounaris et al. [6] proposed a trial-and-error methodology in their previous work, but in this paper [27], they considered shuffling and serialization and investigated the impact of Spark parameters. They addressed that the number of cores of Spark executor has the most impact on maximising performance improvement, and the level of parallelism, for example, the number of partitions per participating core, plays a crucial role. They focused on 12 parameters related to shuffling, compression, and serialization. It is an iterative technique; the lower parts' configurations can be tested only after the upper parts' completion. Three real-world case studies are considered to investigate the methodology efficiency. Due to no-iterative methodology, the run time decreased between 21.4% and 28.89%. They also found that the significant speed-up achievement yields at least 20% lower running times. They concluded that the methodology is robust concerning the changes of its configurable parameters.

Amannejad et al. [28] proposed an approach for Spark execution time prediction with less prior executions of the applications based on Amdahl's law [15]. This approach is capable of predicting the execution time within a short period. This approach requires two reference files at the same data size and different resource settings to predict the execution time. They considered relatively small data sets and a limited application setup which do not have complex dependencies and parallel stages. They found that the proposed technique shows good accuracy. The average prediction error of the workloads is about 4.8%, except Linear Regression (LR) which is 10%. One of the limitations of this work is that they validated this approach only with a single node cluster, not on a real cluster environment. Amannejad and Shah extended their previous work [28] and proposed an alternative model called PERIDOT [29] for quick execution time prediction with limited cluster resource settings and a small subset of input data. They analysed the logs from both of the executions and checked the internal dependencies between the internal stages. Based on their observations, the data partitions, the number of executors' impact, and data size play a critical role. Therefore, they used eight different workloads with a small data set and claimed that apart from naive prediction techniques, the models show significant improvement by overall mean prediction error by 6.6% for all the workloads.

Amdahl's law and Gustafson's law

It is important to determine the benefits of adding processors to run a certain job. In this section, we will use the words *processor* and *executor* as synonymously, although there is a distinction when considering a certain context. In Spark for example, the word *executor* is used to indicate that CPU resources are allocated via a certain physical node. Generally, a single executor is launched in the physical nodes and stays with the physical node. Each of the CPU cores are aligned with the physical nodes [30].

The executor can use one or several cores, which would be analogous to say that several processors are being used per executor. However, as the executors only use cores within a physical node, we consider the number of executors as the variable for our model. In section 5, the experiments were carried out with each executor using three cores. Changing the number of cores obviously changes the parameters of the equation, but the family of equations remain valid for the model. This simplification of the terms is valid because the executors within a node share memory, and any communication

between them would be much faster than any communication between executors running in different physical nodes.

If no communication between the various executors is needed to run a job, the job is called "embarassingly parallel" [14]. The implication of having no need to communicate between different executors is that the speed up is proportional to the number of executors, i.e., if one executors takes time t, then n executors will take time $\frac{t}{n}$. However, any small portion of the job that is not parallelisable can bring major consequences for parallel performance.In this case, the linear speedup achieved by adding more executors (in the form of CPUs or cores, or separate node) declines sharply.

Amdahl came up with a generic equation to predict the speedup factor of a parallel application as a function of the number of processors [15]. The equation considers that parts of the application (or job, or workload) would be inherently serial in nature and would not be parallelisable. He arrived at the following equation for the speedup factor $S()$:

$$S(nexec) = \frac{nexec}{1 + (nexec - 1)f} \tag{1}$$

where $nexec$ is the number of processors (or executors) and f is the percentage of the job that cannot be parallelised (because of its serial characteristic). Figure 2 shows that the speedup gets worse with an increasing factor f.

In practise, an increasing number of executors has to make economical sense, and an ideal number of executors can be found given a target improvement in the speedup. The factor f (the serial percentage of the job) depends entirely on the algorithm and on the platform it is running under. If the serial portion is representing I/O or networking, it may have different influences in percentage f. Perfect linear speedups only happen when $f = 0$.

From Eq. 1, and considering that a single processor takes time t to run a certain workload, the predicted runtime running on multiple processors would be:

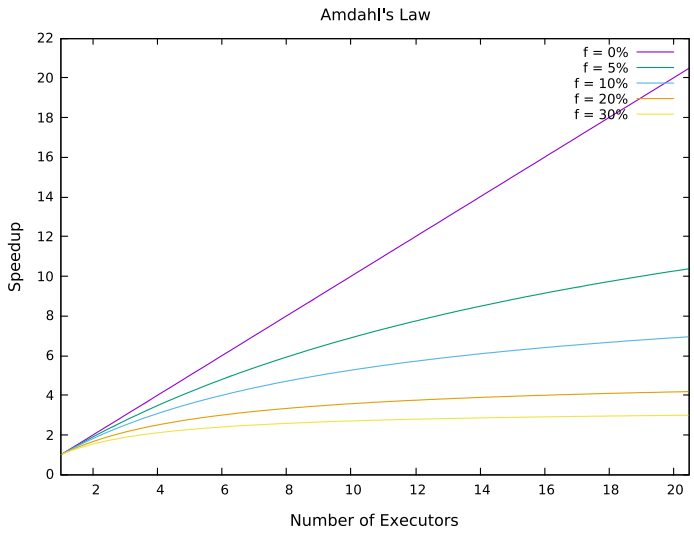


Fig. 2 Amdahl's law with various percentages of serial work

$$runtime = \frac{(1-f)\,t}{nexec} + f\,t \tag{2}$$

where t is a hypothetical runtime needed to run a job in a single executor. As an example, if the job takes 100 s to run on a single executor, then Fig. 3 shows how the runtime is going to decrease with the additional executors depending on how much of the job is serial.

Initially the runtime decreases sharply with the increase of executors, until the runtime converges at some point with infinite executors. It is clear from Figs. 2 and 3 that this is a very pessimistic view of the potential that parallel systems offer.

A few years after Amdahl's publication, Gustafson argued that the percentage of the serial part of a job is rarely fixed for different problem sizes [16]. In Amdahl's even a small percentage of serial work can be detrimental to the potential speedup after adding more executors. Gustafson noticed that for many practical problems the serial portion would not grow with an increase problem size. For example, the serial portion of the job could be a simple communication to establish the initial parameters for a simulation, or it could be I/O to read some data that is independent of the problem size of the algorithm.

He came up with a scaled version of Amdahl's speedup equation. Gustafson's speedup equation is:

$$S(nexec) = nexec + (1 - nexec)\,f \tag{3}$$

$$runtime = \frac{t}{nexec + (1 - nexec)\,f} \tag{4}$$

The speedup for different serial portions f using Gustafson's law are shown in Fig. 4. In Fig. 5 several curves were plotted to show the runtime trends considering that for a single executor the time would be 100 s.

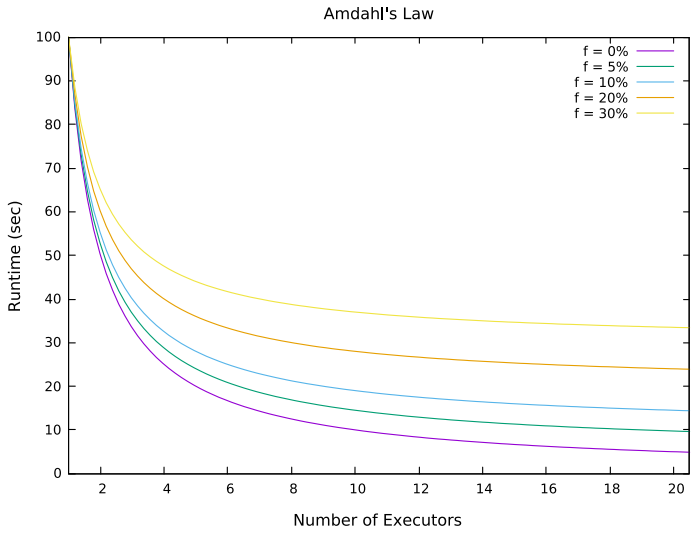


Fig. 3 Amdahl's law for various percentages of serial work

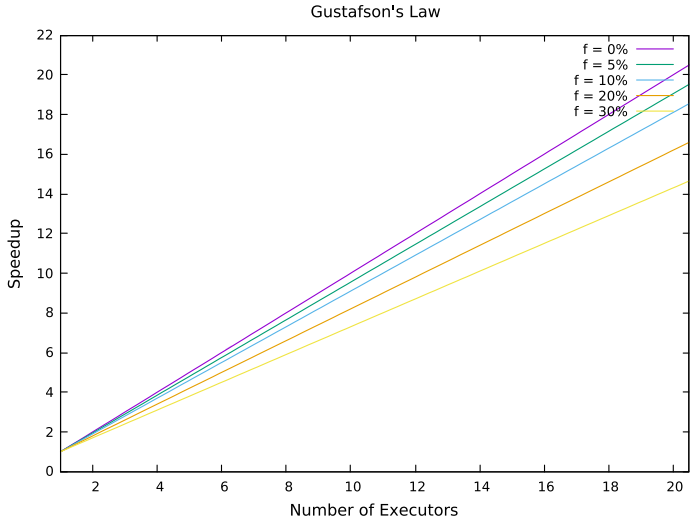


Fig. 4 Gustafson's law for various percentages of serial work

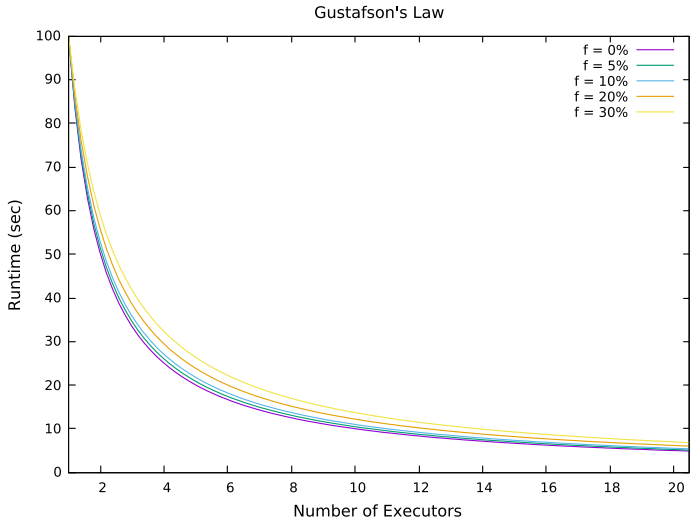


Fig. 5 Gustafson's law for various percentages of serial work

Gustafson's law is much more optimistic than Amdahl's law. Indeed, Gustafson showed that the speedup for certain algorithms could be achieved with the results based on Eq. 3. However, for many other applications and algorithm implementations, the true picture can be even more pessimistic than Amdahl's. That does not mean that we should not attempt to parallelise these algorithms, but one needs to be aware of the performance consequences of adding more executors. In the next section we will show that some of the HiBench workloads [31] fall into this category.

Modelling of a 2D plate parallel application

In this section we discuss the modelling of parallel applications where the serial portion of the job grows faster than expected. As discussed in the literature review in "Related work" section, the performance of every parallel application is dependent on the number of executors, be that in the form of CPUs or cores, and its communication pattern.

For many workloads, the behaviour of the runtime can be predicted by Amdahl's Law or Gustafson's Law. For example, WordCount gains performance by adding executors, until adding more executors makes little difference and brings no new gains in performance. This can be clearly appreciated in Fig. 6.

However, many other workloads behave in a very strange way. Initially, adding more executors results in a better performance. But after a certain number of executors, the performance degrades to such an extent that the runtime is longer than that using very few executors. For example, running jobs on the Pagerank (HiBench [31]) for a certain problem size shows performance as depicted in Fig. 7. After analysing Fig. 7 we realised that a new modelling for the runtime is needed for these applications. This pattern of getting worse performance by adding executors is not unknown, and happens when the communication between each parallel portion of a job grows faster than the benefit of having additional executors.

Finding a model as a function of the number of executors

The serial portion of a job is responsible for the drop in an otherwise perfect speed up. Among the causes for unparallelizable portions of a job, we can consider the two most important ones:

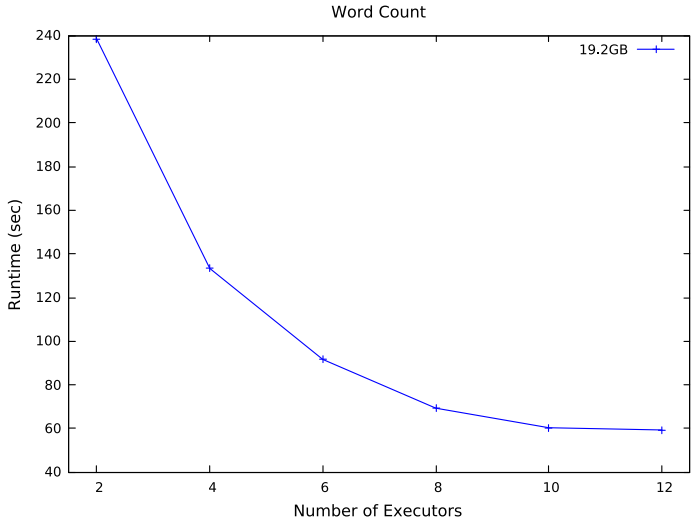


Fig. 6 A WordCount workload running on different number of executors

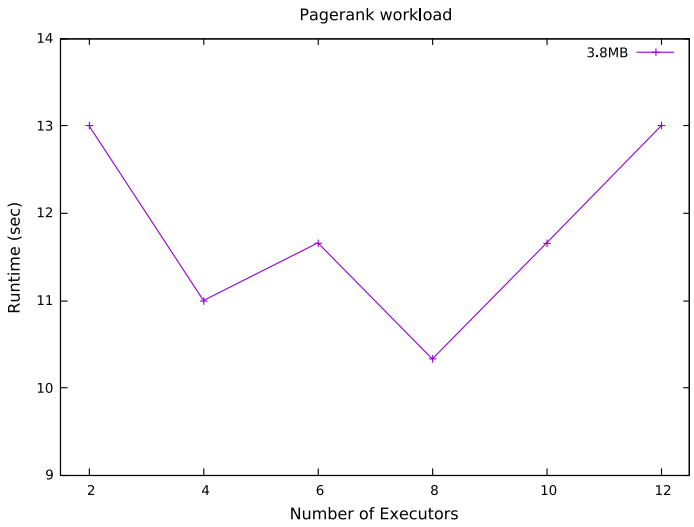


Fig. 7 Page rank workload running on different number of executors

- I/O: in a Hadoop cluster, the data is scattered among the nodes, and sometimes a node will need to read data only available on other nodes. HDFS is responsible for this process in a Hadoop cluster.
- Communication: even if there is no additional need for I/Os, the application may require that data computed on another node updates its own computations. The communication performance is driven by the networking infrastructure available to the cluster. Typically communication between nodes in a parallel computer can be: one to one, one to all (aka broadcasting), all to all and all to one (aka reduction) [14].

The distinction between Hadoop cluster I/O related to HDFS and the Communication is important. Every workload will uses the first one to access data, as the location of the data can be anywhere in some of the nodes. The cluster used in these experiments use a replication factor of 3 (the default). The communication factor in this scope refers specifically to the application communications, i.e., where the data computed by one node is needed to complete the computation on another node.

For example, an embarassingly parallel application would have no communication between nodes for its own purpose, but still would need to use the same network infrastructure to access data via HDFS if portion of data happen to be located on other nodes. On the other hand, an application that would compute heat flow using the 2D plate model would require extra communication between certain executors that is independent of the HDFS access to data. Moreover, in that case a delayed executor can hold the computation on other executors, as these would be waiting for new boundary data to be available. We start the building up of the model with the concept of a 2D plate. This concept can be used for simulating heat distribution simulations in parallel machines, as discussed by [14]. In their simulation, each point of a 2D plate has its temperature computed as a function of its four neighbours. In order to parallelise any job, one needs to consider the serial and the parallel parts of the runtime:

$$runtime = \frac{t}{nexec} + t_{serial} \tag{5}$$

where t is the time to run the application in a single executor, *nexec* is the number of executors and t_{serial} is the extra time needed to make the communication between the executors and additional I/O. If t_{serial} is zero, i.e., no extra communication or I/O is needed, then the runtime is inversely proportional to the number of executors.

The crucial aspect of Eq. 5 is the t_{serial}. Without any knowledge about the internal implementation of the algorithms of the application, it is difficult to model it correctly. Assuming that the serial part grows as a function of the number of executors, we can start by approximating the function to that of parallelising a 2D plate algorithm. We can make some assumptions about the communication and I/O boundaries.

Figure 8 shows a 2D plate that has 256 points ($N = 16$). For this 2D plate, each point has to be computed iteratively. Each point's computation is interdependent with its four neighbours, as it needs the current state from each neighbour.

No communication is needed when the entire set of points is computed by a single executor. As soon as more executors are used, then some communication activity needs to be carried out between the boundaries. Figure 9 shows the idea of the boundaries when using 4 executors. Now there is a communication boundary that adds extra runtime due to networking communication between two different nodes. In this case, the boundary size is proportional to $2N$.

In Fig. 10 two cases are shown, one with 4 executors, and another with 16 executors. The sum of the boundary in the 4 executors job is $2N$, and in the 16 executors it is $6N$. We could try to generalise it for any number of executors. However, to get a smooth growth we should only use square divisions of the 2D plate contained NxN points. Therefore, *nexec* is restricted to the sequence 1, 4, 9, 16, 25... Moreover, we assume that N is sufficiently large to offset the differences between the executors data when N is not exactly divisible by *nexec*.

One problem with this simplistic model is that the communication is not necessarily homogeneous among the executors. For example, an executor on the left top corner of the 2D plate may be carrying out half of the communication that an executor in the

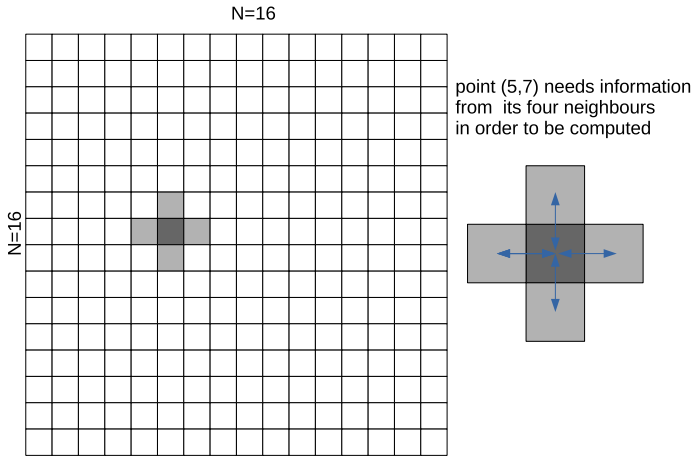


Fig. 8 A 2D plate model running on a single executor

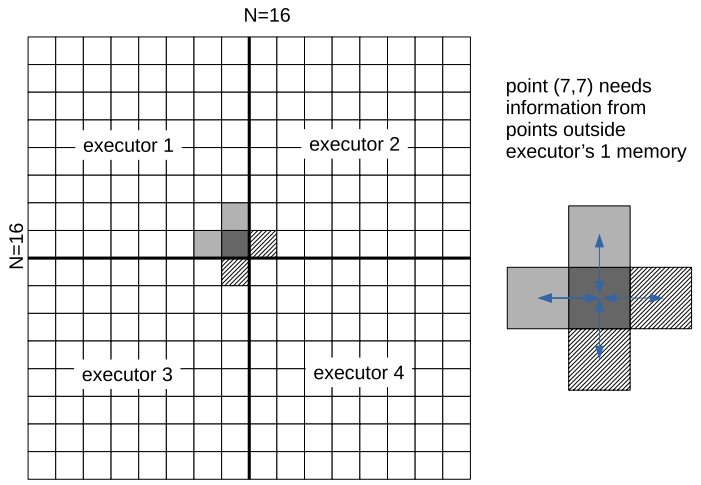


Fig. 9 A 2D plate model running on 4 executors

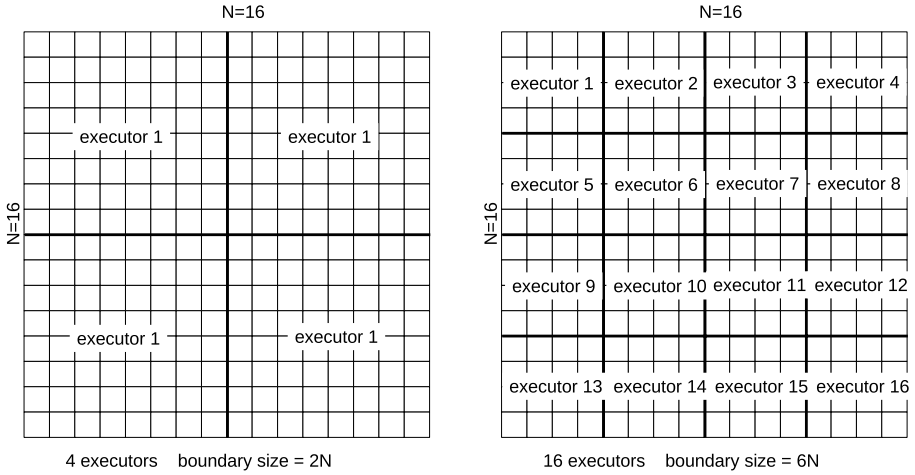


Fig. 10 Boundaries for different number of executors

middle of the plate needs to carry out. One way to consider that possibility is to assume that every executor has 4 neighbouring executors. It is the equivalent of having the 2D plate folded like a cylinder over each dimension simultaneously (Fig. 11).

Looking at Table 1, it is apparent that the boundary size grows at a rate of $(2\ (\sqrt{nexec} - 1))N$. If we consider that all executors have the same boundary, then the boundary grows as $(2\ (\sqrt{nexec} - 1) + 2)N$.

We do not know if the serial time is going to be exactly that amount, as the communication pattern inside the Hadoop cluster can be very complex. For example, executors may have to communicate between them, but also get data via HDFS from other nodes. Also, there is some parallelism implied in the communication, as pairs of nodes would be able to communicate with each other without interfering much with the communication between other pairs. This could cause the parallel and serial portions of the job in each executor to be misaligned, causing executors to temporarily stop computing because they are waiting

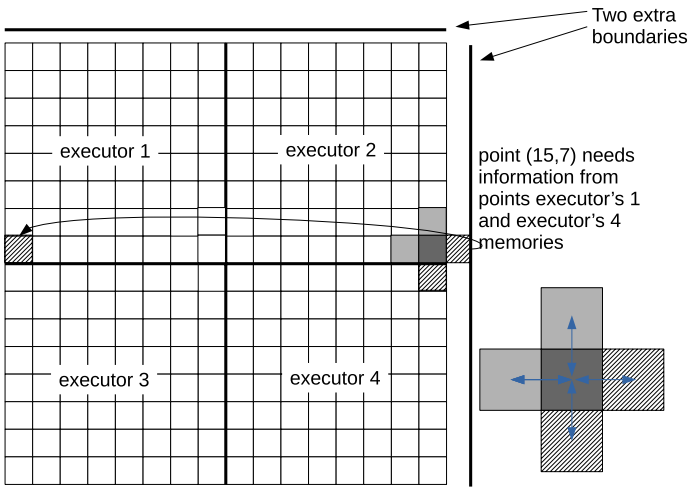


Fig. 11 Two (2) extra boundaries per plate for homogeneous communication amongst nodes

Table 1 Boundary versus *nexec*, showing the size as a function of *N*

Number of executors *(nexec)*	**Boundary size for simple 2D**	**Boundary size for homogeneous communication**
1	0	0
4	2N	4N
9	4N	6N
16	6N	8N
25	8N	10N
36	10N	12N
49	12N	14N

for data from the neighbours or from Hadoop Distributed File System (HDFS). Making the assumption that the growth of the boundary is proportional to the communication time and that the serial portion is also proportional to the problem size width N as per Table 1, Eq. 5 becomes:

$$runtime = \frac{t}{nexec} + n\,N\,(2\,(\sqrt{nexec} - 1) + 2) \tag{6}$$

where n is a constant.

Assuming that the time t is proportional to number of points N^2 of the entire plate, we can simplify Eq. 6 to:

$$runtime = \frac{m\,N^2}{nexec} + 2\,n\,N\,(\sqrt{nexec} - 1 + 1) \tag{7}$$

simplified to:

$$runtime = \frac{m\,N^2}{nexec} + 2\,n\,N\,(\sqrt{nexec}) \tag{8}$$

For a certain (fixed) problem size $_N2$, we can replace $m\ N^2$ by a constant a and replace $2\ n\ N$ by a constant b:

$$runtime = \frac{a}{nexec} + b\ \sqrt{nexec} \tag{9}$$

Now we arrived at a model that can explain the strange behaviour of having a peak performance at a certain number of executors, and have a degraded runtime with more executors being added. One can visualise the effects of the growth of the serial portion of the jobs by examining Fig. 12. Depending on the constants a, b, some curves resemble Amdahl's, while other curves have the serial portion growing faster, to the point where adding more executors make the runtime longer than with a single executor. When the influence of the boundary is smaller (with a corresponding low value for b), then the curves are more similar to Amdahl's law or Gustafson's law. For larger values of b, the speedup falls rapidly with the addition of more executors.

There is another aspect to the modelling regarding the problem size. The assumption for Eq. 9 is that the runtime is proportional to N^2, but this would not be the case for many algorithms, where the complexity would be different than linear in relation to the total number of points (or quadratic if one considers width or height as the problem size). In fact, the final runtime would depend completely on two functions $f(N)$ and $g(N)$ that would only be known if one has more information about the internal implementation of the algorithm running the job. The first function, $f(N)$ would rule the growth of the runtime t for one executor, analogous to its time complexity for the algorithm, considering a large N. The second function, $g(N)$, would rule the growth of the communication needs once more than one executor is used for the job.

Consequently, Eq. 9 can only predict runtime if the constants a and b are known for a certain problem size. A separate model has to be found for the growth of the runtime and the communication boundary as a function of the problem size. Nonetheless, such a simple model can still be of great value for runtime prediction by running a few jobs and

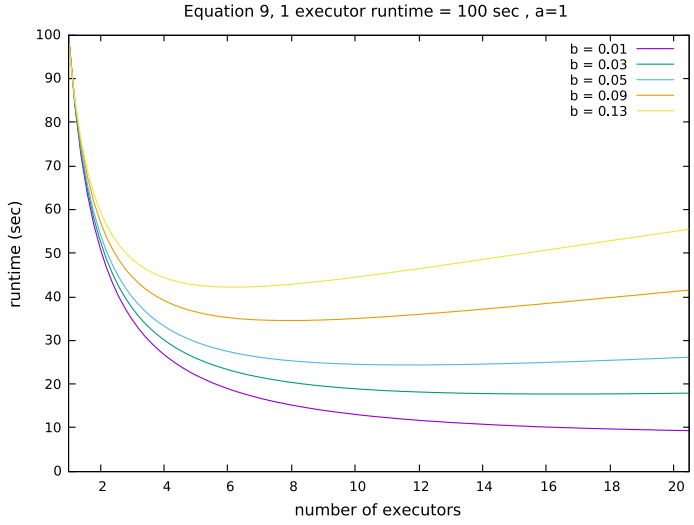


Fig. 12 Equation 9 for various b values

Table 2 Experimental Hadoop cluster

Server configuration	Processor	2.9 GHz
	Main memory	64 GB
	Local storage	10 TB
Node configuration	CPU Specification	Intel(R) Xeon(R) CPU E3-1231 v3 @ 3.40 GHz
	Main memory	32 GB
	Number of nodes	10
	Local storage	6 TB each, 60 TB total
	CPU cores	8 each, 80 total
Software	Operating System	Ubuntu 16.04.2 (GNU/Linux 4.13.0-37-generic x86 64)
	JDK	1.7.0
	Hadoop	2.4.0
	Spark	2.1.0

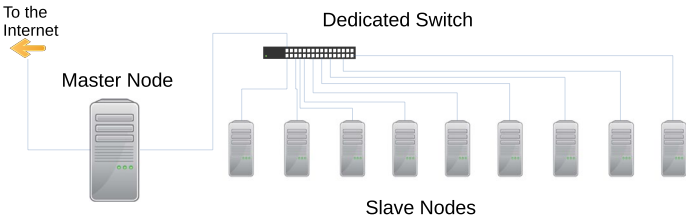


Fig. 13 The Hadoop cluster used in the experiments

forecasting the ideal number of processors for that kind of job. In the next section, we experiment with various workloads to see whether this model can fit some of the empirical data.

Experimental setup

The experimental cluster has its dedicated networking infrastructure, with dedicated switches. The cluster was designed and deployed by a group of experienced academics who previously built Beowulf clusters with optimised performance [32]. This infrastructure is isolated from any other machine to reduce unwanted competition for network resources. The cluster is configured with 1 master node and nine 9-slave nodes. The cluster hardware configuration is presented in Table 2 and a simple schematic is shown in Fig. 13.

Performance evaluation applications

HiBench Benchmark suite [31] comes from the Hadoop testing program to evaluate the cluster's performance. In the following section, the benchmark workloads that are used in this experiment for the Spark performances are shown in Table 3. There are five benchmark workloads from four different categories: Micro Benchmark, Web Search, Graph, and Machine Learning.

Table 3 Spark HiBenchmark workload considered in this study

Benchmark categories	Application	Input data size	Input samples
Micro Benchmark	WordCount	313 MB, 940 MB, 5.9 GB, 8.8 GB, and 19.2 GB	-
Machine learning	K-means (small job)	1.3 MB, 2.7 MB, 4 MB, 5.3 MB, and 13.3 MB	3000, 5000, 7000 (sample), 1 and 3 (million samples)
	K-means (large job)	19 GB, 56 GB, 94 GB, 130 GB, and 168 GB	10, 30, 50, 70, and 90 (million samples)
	SVM	34 MB, 60 MB, 1.2 GB, 1.8 GB, and 2 GB	2100, 2600, 3600, 4100, and 5100 (samples)
Web search	PageRank (small job)	3.8 MB, 5.7 MB, 8 MB, 10 MB, and 12.2 MB	1, 15, 20, 25, and 30 (thousand of samples)
	PageRank (large job)	507 MB, 1.6 GB, 2.8 GB, 4 GB, and 5 GB	1, 3, 5, 7, and 9 (million of pages)
Graph	Nweight	37 MB, 70 MB, 129 MB, 155 MB, and 211 MB	1, 2, 4, 5, and 7 (million of edges)

The WordCount workload is a map-dependent, and in the data set, it counts the number of occurrences of separate words from text or sequence file. The function of Sort takes the input file as a text by key. Each word in the input data, which is generated using RandomTextWriter.

NWeight is an iterative graph-parallel algorithm implemented by Spark GraphX and pregel. The algorithm computes associations between two vertices that are n-hop away. The input data consist of more than 1 million edges.

PageRank is a search page ranking algorithm where every single page comes with a numerical value, and each page is ranked as par vote. It counts a vote when one page is linked with the other page. Normally, a page linked with many other pages considered as the higher PageRank. The data source is generated from Web data whose hyperlinks follow the Zipfian distribution. We used different sets of input data which consist of more than a million samples.

K-means is a very popular and well-known algorithm which is used to group data points into clusters. The input data set is generated by GenKMeansDataset based on Uniform Distribution and Gaussian Distribution. We used different sets of input data, and each set of data contain more than 5 million samples.

Support Vector Machine (SVM) is a standard method for large-scale classification tasks. This workload is implemented in spark.mllib, and the input data set is generated by SVM DataGenerator, which consists of more than 1 million samples.

Cluster parameters configuration

Spark parameter selection and tuning is a challenging task. Every single parameter has an impact on the system performance of the cluster. Hence, the configuration of these parameters needs to be investigated according to the applications, data size, and cluster architecture. To validate our cluster, we try to select the most impactful parameters that have a crucial factor in the system's performance. Generally, Spark configuration parameters can be categorized into 16 classes [33]:

1. Application properties

2. Runtime environment
3. Shuffle behavior
4. Spark user interface (UI)
5. Compression & serialization
6. Memory management
7. Execution behavior
8. Execution metrics
9. Networking
10. Scheduling
11. Barrier execution mode
12. Dynamic allocation
13. Streaming
14. SparkSQL
15. SparkR
16. Thread configuration.

The selected parameters in Table 4 are closely related to the Spark system performance. The default and range column presents the system default values and tuned values used in this experiment. The listed configuration parameters are chosen for two reasons; firstly, these parameters have a greater impact on the Spark runtime performance, such as runtime environment, shuffle behavior, compression and serialization, memory management, execution behavior [31], and the performance of these key aspects ultimately determine the performance of the Spark application.

Generally, the selection extensive parameters and their configurations are based on memory distribution, I/O optimization, task parallelism, and data compression [34]. A noteworthy phenomenon is that the input RDD partition and the allocated memory affect the rate of data spill to disk where the core of assigned executors run concurrently and share their resources. So, the prediction model would be significantly affected without sufficient memory and partitions [24].

Secondly, the impact of these parameters can occupy all available resources, such as CPU, disk read and write, and memory. The selected Spark HiBench application characteristics are presented in Table 5. The applications consist of a number of jobs, number of stages, Directed Acyclic Graph (DAG) architectures and the operations that are used. Most of the selected applications have covered the pattern communication in Spark such as Collect, Shuffle, Serialization, Deserialization and Tree Aggregation.

Findings from the analytical model

In this section, we present the results obtained from the experiments that were carried out with five different workloads, different sizes and number of executors. For accuracy and reproducibility of results, each experiment was repeated three times and considered the average runtime to produce each graph. In this case, we have collected log files from the Spark history server and execute a script to get the execution time.

Table 4 Selected Spark configuration parameters

Parameters	Description	Default	Range
Spark.executor.memory	Amount of memory to use per executor process, in GB	1	12
Spark.executor.cores	The number of cores to use on each executor	#	2–12
Spark.driver.memory	Amount of memory to use for the driver process, in GB	1	4
Spark.driver.cores	The Number of cores to use for the driver process.	1	3
Spark.shuffle.file.buffer	Size of the in-memory buffer for each shuffle file output stream, in KB	32	48
Spark.reducer.maxSizeInFlight	Maximum size of map outputs to fetch simultaneously from each reduce task, in MB	48	96
Spark.default.parallelism	The default number of partitions in RDDs returned by transformations like join, reduceByKey, and parallelize when not set by the user	#	8–100
Spark.python.worker.memory	Amount of memory to use per python worker process during aggregation, in MB	512	512–1024
Spark.python.worker.reuse	Reuse Python worker or not	True	True
Spark.rdd.compress	Whether to compress serialized RDD partitions	False	True/False
Spark.serializer	Class to use for serializing objects that will be sent over the network or need to be cached in serialized form	Java	Java
Spark.memory.fraction	Fraction of heap space used for execution and storage	0.6	0.1–0.4
Spark.memory.storageFraction	Amount of storage memory immune to eviction expressed as a fraction of the size of the region	0.5	0.1–0.4
Spark.task.maxFailures	Number of failures of any particular task before giving up on the job	4	5
Spark.speculation	If set to "true", performs speculative execution of tasks	False	True/False
Spark.rpc.message.maxSize	Maximum message size to allow in "control plane" communication, in MB	128	256
Spark.io.compression.codec	Compress map output files	snappy	lz4/lzf/snappy
Spark.io.compression.snappy.blockSize	Block size in Snappy compression, in KB	32	32–128

Table 5 Workload application characteristics

Workloads	Stages	Parallel Stages	Collect	Serialization	Deserialization	Shuffle	Aggregate
WC	2	Yes	Yes	–	–	Yes	–
SVM	209	Yes	Yes	No	Yes	Yes	Yes
Nweight	9	Yes	–	No	Yes	Yes	–
K-means	20	Yes	Yes	Yes	Yes	Yes	–
Pagerank	5	Yes	–	No	Yes	Yes	–

Fitting and metrics

For every set of data acquired by running the workloads in the Hadoop cluster, we found which are the best parameters a and b in Eq. 9. In order to find the parameters for the equation, we used Gnuplot's fitting function [35] to fit empirical data to the equation.

Once the parameters a and b are computed for each size series, it is possible to compute a fitting metric. One can compute what the runtime for the fitted equation is and compare to the empirical data. We adopted the R-squared values, which is also known as coefficient of determination. R-squared is computed using the following equation [36]:

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}} \tag{10}$$

where SS_{res} is the sum of the squares of the residuals and SS_{tot} is the sum of the squares relative to the mean of the data. For a perfect fitting, $SS_{res} = 0$ and $R^2 = 1$. Generally, the closer R^2 is to one, the better the fitting.

The results

Firstly, we present how a and b in Eq. 9 are different for each curve with fixed problem sizes.

In Figs. 14, 15 and 16 the model fits the empirical data reasonably well. For both the Wordcount and Graphs,the curves are smoothing out the runtime as the number of executors grows. In the SVM case (Fig. 16), the model fits nicely and it shows that the performance reaches a peak for a certain number of executors. This is exactly the case that the model explains. It seems that for these three workloads the serial part growth follows Eq. 9 very closely.

For workloads Pagerank and Kmeans, the model does not fit very well (Figs. 17 and 18). This is the case when the sizes are too small, and the runtime is relatively short. For these workloads, the overheads related to the Hadoop cluster overshadows the model.

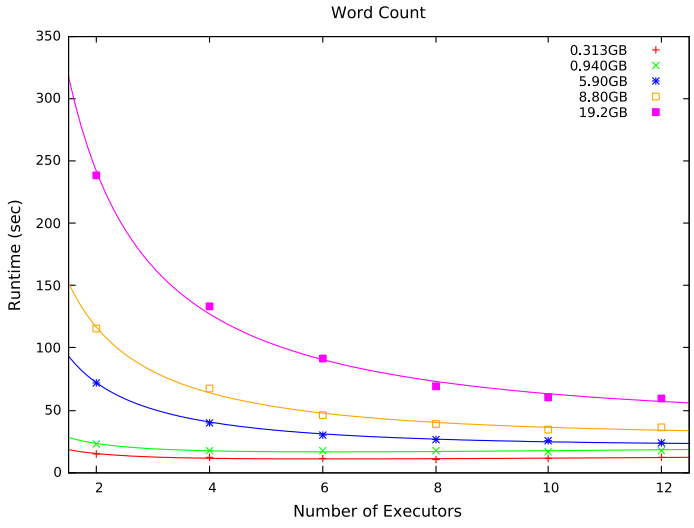


Fig. 14 Fitting the 2D plate model to Wordcount

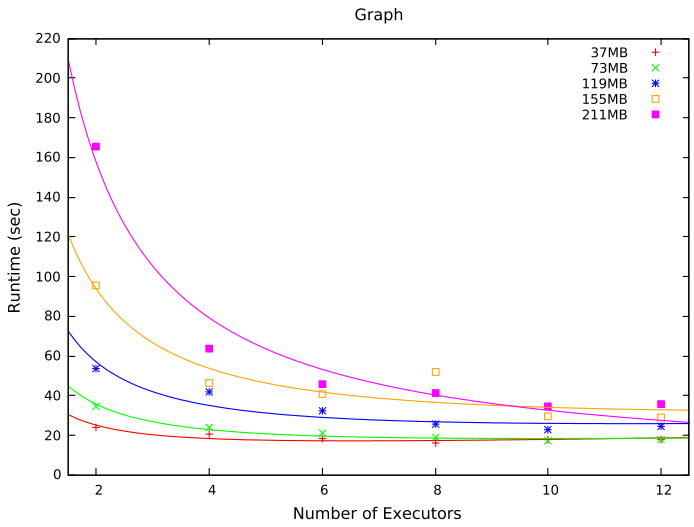


Fig. 15 Fitting the 2D plate model to NWeight (Graph)

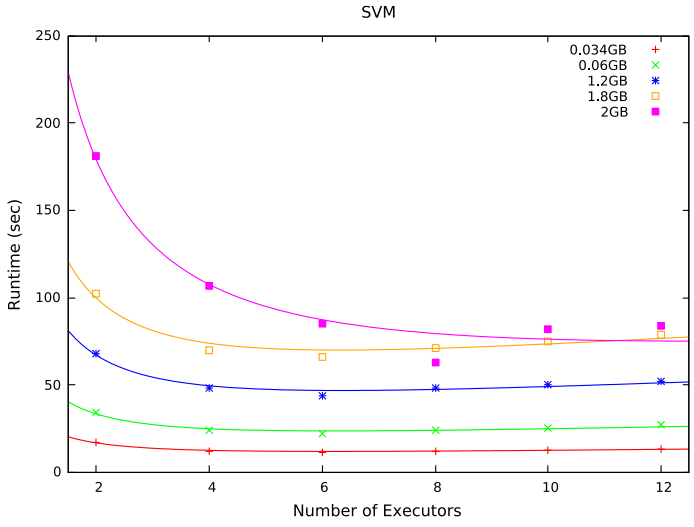


Fig. 16 Fitting the 2D plate model to SVM

For these two workloads, we have experimented with a different equation. We have seen that in Eq. 9, the boundary grows at a rate proportional to the square root of *nexec*. We then adjusted this function to a different exponent, making it:

$$runtime = \frac{a}{nexec} + b\ nexec^{c} \tag{11}$$

It is important to note that Eq. 9 is a special case of Eq. 11, where $c = 0.5$. Interestingly, after fitting Eq. 11 via Gnuplot [35], we found that for a value of $c = 0.14$, the data used in Fig. 17 fitted much more accurately, as shown in Fig. 19. In this figure, the R-squared valued achieved a maximum of 0.870 for size 8 MB and a minimum of 0.497 for size 3.8 MB. For the other three sizes 5.7 MB, 10 MB and 12.2 MB, the R-squared values were 0.560, 0.744 and 0.619 respectively.

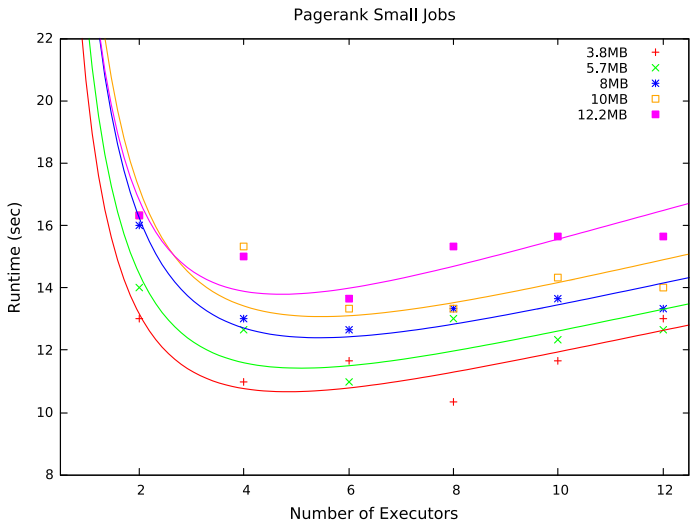


Fig. 17 Fitting the 2D plate model to Pagerank

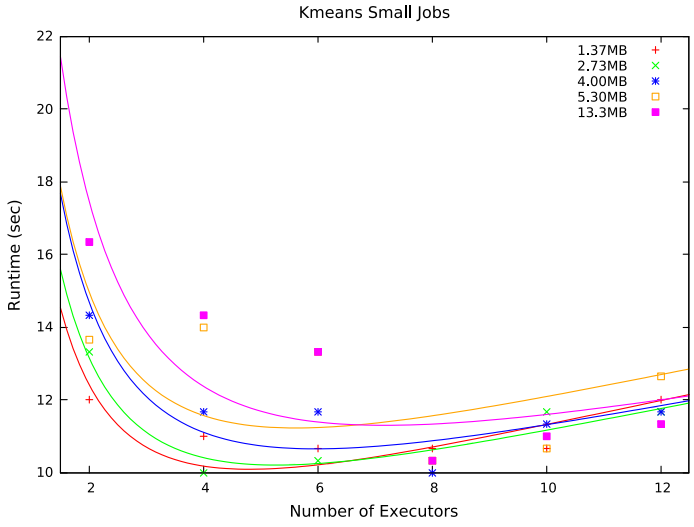


Fig. 18 Fitting the 2D plate model to Kmeans

This shows that an exponential function explains the same behaviour that we targeted in this work, i.e., the runtime reaches a peak performance for a certain number of executors, and then the runtime keeps growing, degrading the performance even when more executors are added to run the job.

For Pagerank and Kmeans, we repeated the experiments with larger problem sizes. For larger sizes, Pagerank fits the original Eq. 9 (Fig. 20). Kmeans also shows a better fit to Eq. 9 (Fig. 21).

This shows that the relationship between the serial part and the problem size can also vary. It seems that the constant c works well for Wordcount, SVM and NWeight for $c = 0.5$ (which is the c value in the original Eq. 9).

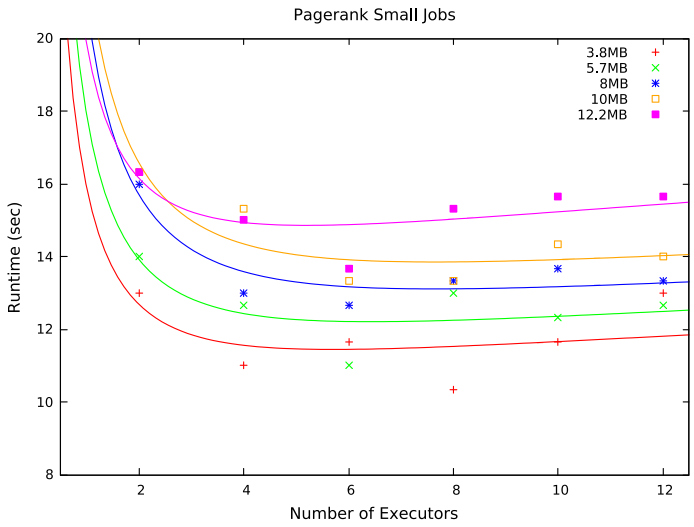


Fig. 19 Fitting Eq. 11 to Pagerank jobs, with c=0.14

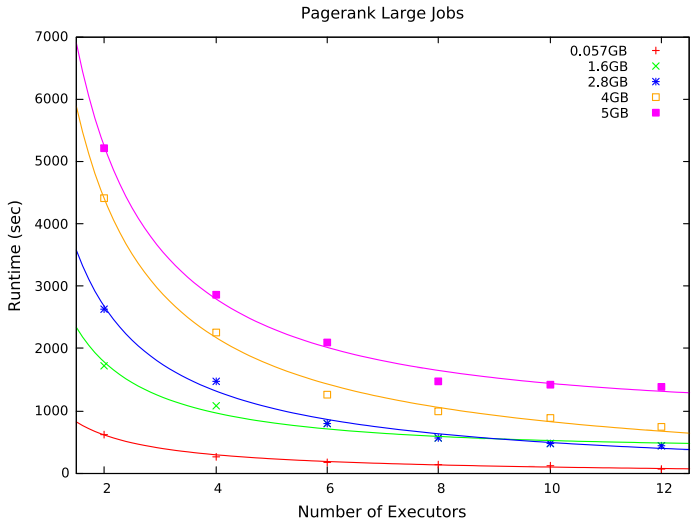


Fig. 20 Fitting the 2D plate model to Pagerank with larger problem sizes

For Pagerank and Kmeans it shows that the constant c can vary with the problem size. The explanation is that the unpredictable overheads may overshadow the pattern of the runtime when the sizes are too small, and the jobs run in just a few seconds. Longer jobs are more stable, and the pattern of the growth of the boundaries (serial part) can be found more easily. More work needs to be done for the other workloads.

Fitting errors and comparison with Amdahl's and Gustafson's laws

The figures in "Findings from the analytical model" section showed the fitting results for the proposed model. Although we have compared each one of the curves with Amdahl's and Gustafson's Laws, in this section we only show three examples. In the

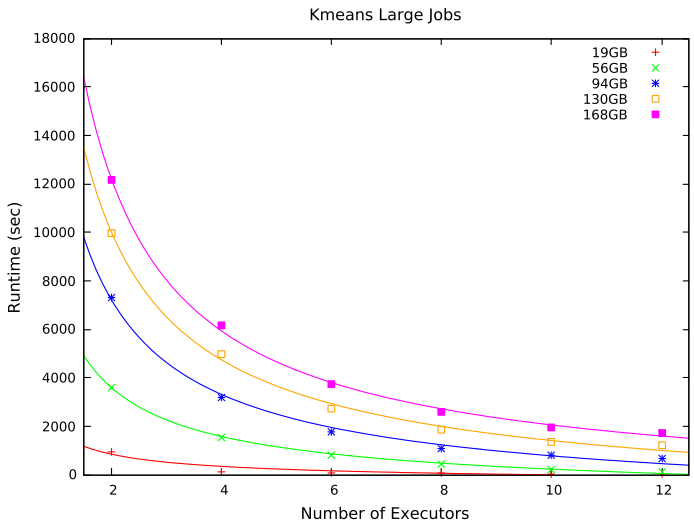


Fig. 21 Fitting the 2D plate model to Kmeans with larger problem sizes

majority of the curves, the proposed model fits the empirical data more accurately. However, in a few cases Amdahl's or Gustafson's fit better. Figure 22 shows three graphs.

The first graph shows that the empirical data fits accurately for all three models. The R-squared value for the proposed model is 0.999, for Amdahl's is 0.998, and for Gustafson's it is very close, 0.997.

The second graph in Fig. 22 shows that Gustafson's Law has the best fit, but with a low R-squared of 0.849. The R-squared values for the proposed model and Amdahl's are 0.649 and 0.840 respectively.

Finally, the third graph in Fig. 22 shows that the proposed model achieved the best fit. The R-squared values were 0.962 for the proposed model, 0.611 for Amdahl's model and 0.198 for Gustafson's model. We can state that in applications where the runtime goes down and up again with increasing executors, our model will work better than Amdahl's or Gustafson's. For the cases where the runtime keeps going down until it converges to a fixed value, all three models may work.

The R-squared values for all the curves fitted from Figs. 14 to 21 are shown in Table 6. These results show that generally our model fits the data better than Amhdal's or Gustafson's equations. Among the 35 rows in Table 6, 25 indicate that our model worked better, while 4 rows worked better for Amdahl's equation and 6 worked better for Gustafson's equation.

Conclusion

This paper has proposed a new parallelisation model for different workloads of Spark Big Data applications running on Hadoop clusters. The proposed model can predict the runtime for generic workloads as a function of the number of executors without necessarily knowing how the algorithms were implemented, with a relatively small number of experiments to determine the parameters for the model's equation. The main focus is to provide a quick insight into the system's parameters and reduce the runtime to help

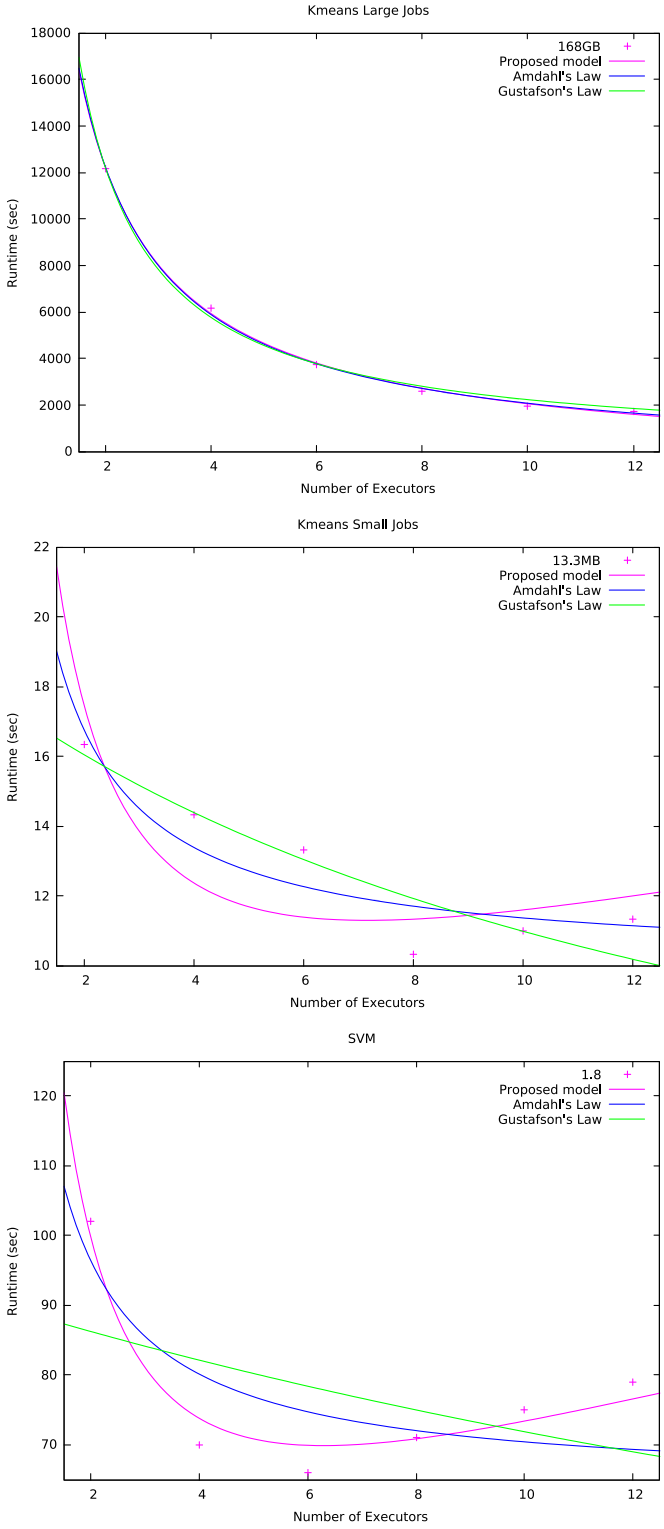


Fig. 22 Comparison for the fitting accuracy using the proposed model, Amdahl's law and Gustafson's law

Table 6 R-squared estimates for all the workloads

Workloads	Size (MB/GB)	R-squared proposed model	R-squared Amdahl	R-squared Gustafson
WordCount (Fig. 14)	313.6 MB	**0.945**	0.743	0.344
	940 MB	0.874	**0.937**	0.641
	5.9 GB	**0.999**	0.992	0.956
	8.8 GB	**0.996**	0.995	0.981
	19.2 GB	**0.997**	0.997	0.995
SVM (Fig. 16)	34 MB	**0.958**	0.586	0.175
	60 MB	**0.962**	0.596	0.184
	1.2 GB	**0.971**	0.662	0.249
	1.8 GB	**0.962**	0.611	0.198
	2 GB	**0.956**	0.925	0.827
NWeight (Fig. 15)	37 MB	0.823	**0.912**	0.706
	73 MB	0.973	**0.997**	0.917
	119 MB	0.886	0.936	**0.970**
	155 MB	**0.893**	0.890	0.843
	211 MB	**0.967**	0.966	0.934
K-means (large job) (Fig. 21)	19 GB	0.943	0.912	**0.974**
	56 GB	**0.999**	0.998	0.979
	94 GB	**0.999**	0.999	0.986
	130 GB	**0.997**	0.997	0.971
	168 GB	**0.999**	0.998	0.997
K-means (small job) (Fig. 18)	1.3 MB	**0.670**	0.233	0.007
	2.73 MB	**0.941**	0.338	0.024
	4 MB	**0.803**	0.750	0.425
	5.30 MB	0.087	0.346	**0.400**
	13.3 MB	0.649	0.840	**0.849**
Pagerank (large job) (Fig. 20)	507 MB	0.992	0.994	**0.997**
	1.6 GB	0.974	0.983	**0.997**
	2.8 GB	**0.991**	0.990	0.990
	4 GB	**0.995**	0.995	0.995
	5 GB	**0.996**	0.996	0.990
Pagerank (small job) (Fig. 17)	3.8 MB	**0.664**	0.137	0.001
	5.7 MB	**0.535**	0.372	0.105
	8 MB	**0.897**	0.670	0.253
	10 MB	0.541	**0.730**	0.474
	12.2 MB	**0.668**	0.144	0.000

users, operators, and administrators to optimise the application performance. We have used a physical cluster and various HiBench workloads of Spark applications on the proposed performance model.

The results show that a particular runtime pattern emerged when adding more executors to run a certain job. This pattern is driven by a growth of the serial portion of jobs, found to be proportional to the square root of the number of executors.

For some workloads, the runtime reached a low point, growing again despite the fact that more executors were added. This phenomenon is predicted by the proposed model of parallelisation. We have found that for three workloads, WordCount, SVM and Nweight, the runtime versus executors fit the model's equation very well. However,

for the workloads Pagerank and Kmeans the model only works well for large data jobs. Finally, we can conclude that the results are satisfactory, considering the job sizes and parameters we chose. The proposed model can give precise recommendations for the number of executors for a certain problem size, so it is beneficial in terms of performance tuning.

For future work, the model will be tested for most of the HiBench workloads to determine which one works well with the model, or to find an alternative equation that can fit the data. For each workload, larger problem sizes should be used, with a wider range of sizes as well. This would allow for a more accurate prediction of the runtime for a certain physical cluster, with a minimum number of experiments to determine the two most important parameters for runtime, number of executors and problem sizes.

Acknowledgements
This work was supported in part by the Massey University Doctoral Scholarship.

Authorws' contributions
NA and ALCB were the main contributors of this work. NA has done an initial literature review and data collection, run experiments, prepare results, and drafted the manuscript. NA and ALCB has done the literature review on Amhdal's and Gustafson's laws and wrote the 2D plate model section. NA and ALCB fitted the data into the models and analysed the results. ALCB and TS deployed and configured the physical Hadoop cluster. TS and MAR helped to improve the final paper. All authors read and approved the final manuscript.

Funding
This work was not funded.

Availability of data and materials
The data that support the findings of this study are available from the corresponding author upon reasonable request.

Declarations

Ethics approval and consent to participate
Not applicable.

Competing interests
The authors declare that they have no competing interests.

Consent for publication
Not applicable.

Author details
[1]School of Natural and Computational Sciences, Massey University, Albany, Auckland 0745, New Zealand. [2]Department of Mechanical and Electrical Engineering, Massey University, Auckland 0745, New Zealand.

Received: 5 June 2021 Accepted: 5 August 2021
Published online: 14 August 2021

References

1. Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I, et al. Spark: cluster computing with working sets. HotCloud. 2010;10(10–10):95.
2. Dean J, Ghemawat S. Mapreduce: simplified data processing on large clusters. Commun ACM. 2008;51(1):107–13.
3. Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauly M, Franklin MJ, Shenker S, Stoica I Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12), 2012; 15–28
4. Armbrust M, Xin RS, Lian C, Huai Y, Liu D, Bradley JK, Meng X, Kaftan T, Franklin MJ, Ghodsi A, et al. Spark sql: Relational data processing in spark. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data; 2015, p. 1383–1394.
5. Kroß J, Krcmar H. Pertract: model extraction and specification of big data systems for performance prediction by the example of apache spark and hadoop. Big Data Cognit Comput. 2019;3(3):47.
6. Petridis P, Gounaris A, Torres J. Spark parameter tuning via trial-and-error. In: INNS Conference on Big Data. Springer; 2016, p. 226–237.
7. Ardagna D, Barbierato E, Evangelinou A, Gianniti E, Gribaudo M, Pinto TB, Guimarães A, Couto da Silva AP, Almeida JM. Performance prediction of cloud-based big data applications. In: Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering; 2018, p. 192–199.

8. Nguyen N, Khan MMH, Wang K. Towards automatic tuning of apache spark configuration. In: 2018 IEEE 11th International Conference on Cloud Computing (CLOUD). 2018, p. 417–425. IEEE.
9. Ahmed N, Barczak AL, Susnjak T, Rashid MA. A comprehensive performance analysis of apache Hadoop and apache spark for large scale data sets using Hibench. J Big Data. 2020;7(1):1–18.
10. Wang G, Xu J, He B. A novel method for tuning configuration parameters of spark based on machine learning. In: 2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), pp. 586–593 (2016). IEEE
11. Costa RLC, Moreira J, Pintor P, dos Santos V, Lifschitz S. A survey on data-driven performance tuning for big data analytics platforms. Big Data Res. 2021;25:100206.
12. Aziz K, Zaidouni D, Bellafkih M. Leveraging resource management for efficient performance of apache spark. J Big Data. 2019;6(1):1–23.
13. Tong W, Li L, Zhou X, Franklin J. Efficient spatiotemporal interpolation with spark machine learning. Earth Sci Inf. 2019;12(1):87–96.
14. Wilkinson B, Allen M. Parallel Programm. New Jersey: Prentice Hall; 1999.
15. Amdahl GM Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the April 18-20, 1967, Spring Joint Computer Conference, 1967; 483–485
16. Gustafson JL. Reevaluating Amdahl's law. Commun ACM. 1988. https://doi.org/10.1145/42411.42415.
17. Kannan P Beyond hadoop mapreduce apache tez and apache spark. San Jose State University. http://www.sjsu.edu/people/robert.chun/courses/CS259Fall2013/s3/F.pdf (02.08.2016) 2015.
18. Chen Y, Goetsch P, Hoque MA, Lu J, Tarkoma S: d-simplexed: Adaptive delaunay triangulation for performance modeling and prediction on big data analytics. IEEE Trans. Big Data. 2019.
19. Vavilapalli VK, Murthy AC, Douglas C, Agarwal S, Konar M, Evans R, Graves T, Lowe J, Shah H, Seth S, et al. Apache hadoop yarn: Yet another resource negotiator. In: Proceedings of the 4th Annual Symposium on Cloud Computing. 2013, p. 1–16.
20. Wang K, Khan MMH. Performance prediction for apache spark platform. In: 2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems, pp. 166–173 (2015). IEEE
21. Singhal R, Singh P. Performance assurance model for applications on spark platform. In: Technology Conference on Performance Evaluation and Benchmarking, pp. 131–146 (2017). Springer
22. Maros A, Murai F, da Silva APC, Almeida JM, Lattuada M, Gianniti E, Hosseini M, Ardagna D. Machine learning for performance prediction of spark cloud applications. In: 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), pp. 99–106 (2019). IEEE
23. Venkataraman S, Yang Z, Franklin M, Recht B, Stoica I Ernest: Efficient performance prediction for large-scale advanced analytics. In: 13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16), 2016; 363–378
24. Al-Sayeh H, Hagedorn S, Sattler K-U. A gray-box modeling methodology for runtime prediction of apache spark jobs. Distrib Parallel Databases. 2020;38:1–21.
25. Cheng G, Ying S, Wang B, Li Y. Efficient performance prediction for apache spark. J Parallel Distrib Comput. 2021;149:40–51.
26. Gulino A, Canakoglu A, Ceri S, Ardagna D. Performance prediction for data-driven workflows on apache spark. In: 2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), 2020, p. 1–8. IEEE.
27. Gounaris A, Torres J. A methodology for spark parameter tuning. Big Data Res. 2018;11:22–32.
28. Amannejad Y, Shah S, Krishnamurthy D, Wang M. Fast and lightweight execution time predictions for spark applications. In: 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), pp. 493–495 (2019). IEEE
29. Shah S, Amannejad Y, Krishnamurthy D, Wang M Quick execution time predictions for spark applications. In: 2019 15th International Conference on Network and Service Management (CNSM), pp. 1–9 (2019). IEEE
30. Chao Z, Shi S, Gao H, Luo J, Wang H. A gray-box performance model for apache spark. Future Gener Comput Syst. 2018;89:58–67.
31. Intel-bigdata: Intel-bigdata/HiBench. https://github.com/Intel-bigdata/HiBench
32. Barczak ALC, Messom CH, Johnson MJ Performance characteristics of a cost-effective medium-sized Beowulf cluster supercomputer. In: LNCS 2660. 2003; p. 1050–1059. SpringerLink
33. Spark Configuration. https://spark.apache.org/docs/latest/configuration.html.
34. Lucas Filho ER, de Almeida EC, Scherzinger S, Herodotou H. Investigating automatic parameter tuning for sql-on-hadoop systems. Big Data Research. 2021;25:100204100204.
35. Williams T, Kelley C, many others: Gnuplot 5.4: an interactive plotting program. 2020. http://gnuplot.sourceforge.net/.
36. James G, Witten D, Hatie T, Tibshirani R. An introduction to statistical learning. 2nd ed. Cham: Springer; 2021.

Publisher's Note

Chapter 4

The contents of this chapter are from the following article. In accordance with the Multidisciplinary Digital Publishing Institute (MDPI) open access policy, any full text that has been included, is the unmodified accepted article.

big data and cognitive computing

MDPI

Article

An Enhanced Parallelisation Model for Performance Prediction of Apache Spark on a Multinode Hadoop Cluster

Nasim Ahmed [1,*], Andre L. C. Barczak [1], Mohammad A. Rashid [2] and Teo Susnjak [1]

[1] School of Natural and Computational Sciences, Massey University, Auckland 0745, New Zealand; a.l.barczak@massey.ac.nz (A.L.C.B.); T.Susnjak@massey.ac.nz (T.S.)
[2] Department of Mechanical and Electrical Engineering, Massey University, Auckland 0745, New Zealand; m.a.rashid@massey.ac.nz
* Correspondence: nasim751@yahoo.com

Abstract: Big data frameworks play a vital role in storing, processing, and analysing large datasets. Apache Spark has been established as one of the most popular big data engines for its efficiency and reliability. However, one of the significant problems of the Spark system is performance prediction. Spark has more than 150 configurable parameters, and configuration of so many parameters is challenging task when determining the suitable parameters for the system. In this paper, we proposed two distinct parallelisation models for performance prediction. Our insight is that each node in a Hadoop cluster can communicate with identical nodes, and a certain function of the non-parallelisable runtime can be estimated accordingly. Both models use simple equations that allows us to predict the runtime when the size of the job and the number of executables are known. The proposed models were evaluated based on five HiBench workloads, Kmeans, PageRank, Graph (NWeight), SVM, and WordCount. The workload's empirical data were fitted with one of the two models meeting the accuracy requirements. Finally, the experimental findings show that the model can be a handy and helpful tool for scheduling and planning system deployment.

Keywords: big data processing; Apache Spark; execution time prediction; performance prediction; modelling

Citation: Ahmed, N.; Barczak, A.L.C.; Rashid, M.A.; Susnjak, T. An Enhanced Parallelisation Model for Performance Prediction of Apache Spark on a Multinode Hadoop Cluster. *Big Data Cogn. Comput.* **2021**, *5*, 65. https://doi.org/10.3390/bdcc5040065

Academic Editor: Carson K. Leung

Received: 22 September 2021
Accepted: 1 November 2021
Published: 5 November 2021

1. Introduction

An increasing amount of data is coming from many applications, and it has become a challenging task to store and process big data efficiently [1]. In the last decade, researchers proposed and developed efficient distributed parallel file systems, such as MapReduce [2] and Spark [3], which provide various functions, including fault-tolerant, high scalability, open access [4], and simple application programming interfaces (APIs). Spark got prompt attention from professionals and researchers because of those features and fast data processing [5]. Spark can support of a wide range of data processing libraries, such as SQL spark for structured data processing; MLlib; and GraphX for machine learning, image processing, and streaming [6]. Besides, it can also store batch and streaming data and process this data using the applications and store the results in HDFS.

Spark introduced a new data abstraction technique called resilient distributed dataset (RDDs) [3] that improves multiple applications' performances. Its application execution time is an essential factor in measuring real-time processing where the optimum execution time can be obtained based on accurate resource allocation. Spark's performance expansively depends on the suitable selection of parameters, as this system has more than 150 parameters, and the selection and configuration of these parameters are challenging. The users need to adjust the configuration parameters as per the cluster resources; else, the cluster's performance degrades significantly. Indeed, it is essential to select and configure the parameters that play an important role in system's performance [7]. In the recent past, researchers proposed number of techniques, such as trial-and-error [8], cost-based (analytical) [9], and machine

learning modelling [10,11]. However, all these techniques are either time-consuming or require large amounts of training and test data [12]. There are many issues practitioners may encounter when trying to model the performance of a cluster. One standard option to create a model is the use of machine learning algorithms, but this requires that enough sample runs are acquired, and this can take time. If not enough samples are acquired, capturing diverse data points, the accuracy of the model may suffer. Furthermore, machine learning can be a black box for the practitioner, and finding a simple model would be very useful because that would minimise the need to run workloads repeatedly too many times. Therefore, the following research question arises: *"What parallelisation model for a Hadoop cluster can be found and implemented quickly and efficiently in order to improve the performance prediction of a job?"*

Any algorithm can be parallelised, but not all algorithms can run efficiently in parallel machines such as a Hadoop cluster. It is a common phenomenon that the parallel performance depends mostly on how the algorithm operates and how nodes communicate to one another. In any parallel system, two of the most important parameters that will determine the runtime are the size of the job and the number of available executors (here executors can be interpreted as CPUs, or nodes of a cluster). Other parameters can drag the performance down, but they will not necessarily increase the performance. For example, if not enough memory is available to a job, this will increase runtime. If the minimum amount of memory is available, more memory will not make the job run faster. The number of available executors is very important, especially when the algorithm being parallelised requires communication that is not needed when the same algorithm is implemented and run on a single executor. For example, some algorithms are embarrassingly parallel (a term coined in the 90s), meaning that no extra work is needed when the job is parallelised. In this case, the speed-up is proportional to the number of processors available. In other cases, the speed-up can be super-linear, as in the case of searching algorithms running in parallel. Unfortunately, there are also groups of algorithms that do not present this optimistic speed-up [13]. One important factor that causes the degradation of performance is the fact that the algorithm may require extra communication and I/O operations that are inherently serial in nature.

The motivation for this paper was to extend our previous work, where we proposed a simple model to predict runtime as a function of number of executors [14]. The novelty of that work was the consideration of the importance of the amount of data in such performance prediction models. Accordingly, we extend the previous model and propose new parallelisation models that consider the number of executors and the the amount of data simultaneously. To the best of the authors' knowledge, such models have not been published in the literature before. These new runtime performance prediction models rely on simple equations. They can potentially be as fast and as accurate as models created using machine learning. The authors have experimentally confirmed that the proposed ideas can be very useful for the runtime performance prediction of Spark jobs on the Hadoop cluster because they require minimum training data to achieve good predictions in less time.

The key contributions of this paper are as follows:

- We introduced two distinct parallelisation models for performance prediction of Spark jobs on Hadoop cluster. Each model is based on a different communication pattern between the nodes of a Hadoop cluster.
- We accomplished extensive experimental work. The authors analysed and verified the performance pattern based on two main parameters, the number of executors and the amount of data for each job. The data reliability was verified by running each workload at least three times.
- We evaluated our models on five HiBench workloads in order to test the data fitting accuracy. Our results show that the experimental data fitted one of the models accurately, and the fitness was compared with Amdhal's law, Gustafson's law, and Ernest's model. The data fitness was compared based on two criteria, Rsquared and RRSE.

The remainder of this paper is organised as follows: Section 2 provides a brief overview of Apache Spark. Section 3 presents some interesting Spark performance prediction based on a recently published Hadoop cluster-related study. In Section 4, we discuss existing models for runtime prediction for a Hadoop cluster. In Section 5, we describe a parallel model based on a fully connected network, and discuss the motivation for this model. In Section 6, we explain the experimental setup and present the workload execution and show the DAG of stages. Section 7 presents the results and analysis; in particular, it shows how the different equations fit the data. Finally, in Section 8 we present our conclusions with a discussion on the future developments for the model.

2. Apache Spark Platform

Matei Zahari developed Apache Spark at UC Berkely's AMPLab in 2009 [3]. In 2010, Spark became an open-source project. Spark has since been very popular and serves as an alternative to the MapReduce model for open access, high-performance [4], and real-time data processing [15]. Spark presents a new way to process data faster, and its uses are in data analytics, big data processing, and machine learning. The major advantage of Apache Spark for machine learning is its end-to-end capabilities. As per the Datanyze market research [16], Apache Spark's market share is almost 6.40% with more than 2770 companies globally. As per enlyft data [17], 59% of customers of Apache Spark are in the United State, 6% are in the United Kingdom, and 6% are in India.

Many programming languages, such as Python, Scala, Java, and SQL APIs, are embedded within this technology for use and development purposes. Compared to Hadoop, Spark offers a hundred times faster memory and ten times faster performance on disk. Due to its memory, Spark increases the performance of the application. Spark is an ecosystem which consists of various components, such as Spark SQL, Spark Streaming, Mllib, GraphX, and core API components. These components are designed to work closely to the core, and an application can be developed based on their libraries. Apache Spark offers well-defined architecture. In this architecture, the two main abstractions are Resilient Distributed Datasets (RDDs) and Directed Acyclic Graph (DAG). Generally, in the Spark cluster, an RDD collects the data and splits the data into partitions; then, this partitioned data are stored in the memory on worker nodes and parallel operations are performed. Spark's RDD supports two types of operations, transformations and actions. A transformation uses the existing data to create a new dataset, and actions perform computations on the dataset and return their values to the driver program [18]. In Apache Spark, DAG consists of sequences of vertices and edges. Any job submitted in Spark creates a DAG and forwards the job into the stage level, where every stage is comprised of tasks based on input data and the RDD partition.

Apache Spark architecture uses master–slave systems with driver programs. The driver program runs as a master node, and the executors run as slave nodes. The executors start their processes once they receive the input file and continue until the job is completed. In this case, the executors keep themselves active the entire time and use multiple CPU threads for the task in parallel. The driver program creates the SparkContext and stores all the components. Spark driver and SparkContext look after the job execution in the cluster. In Spark, a job is executed in one or multiple physical units, and the jobs are divided into smaller sets of tasks at this stage. A single spark job can trigger many jobs that are dependent on the parent stage. Thus, the submitted job can be executed in parallel. Spark runs submitted jobs in two stages: ShuffleMapStage and ResultStages. ShuffleMapStage is an intermediate stage where the output data are stored for the following stages in the DAG. The ResultStages is the final stage of this process that applies a function to one or multiple partitions of the target RDD.

For any given work, the number of executors, the amount of data, and the number of threads play vital roles in the performance [19]. The block manager acts as a cache storage for a user's program when executors allocate memory storage for the RDDs. Spark runs on a Hadoop cluster with Apache YARN (Yet Another Resource Negotiator) [20]

as a framework for resource management and job scheduling or monitoring, in separate domains; and Apache Ambari manages, monitors, and profiles the individual workloads running the Hadoop cluster. Figure 1 shows a typical Spark cluster architecture.

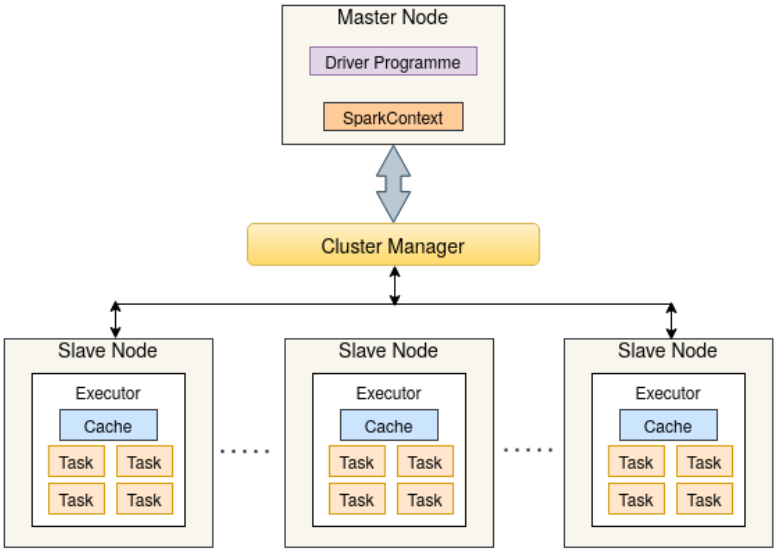


Figure 1. A typical Spark cluster architecture.

3. Related Work

The current state-of-art in Spark performance predictions of big data has received widespread attention from researchers. Researchers have proposed several exciting works based on trial-and-error [8], grey-box modelling [21], black-box modelling [12], and machine learning approaches [22,23]. In this section, we present a review of the literature published in the recent past.

Petridis et al. [8] presented a trial-and-error methodology to predict the execution time of a Spark job. This work highlighted how the number of cores and parallelism play significant roles in the performance. There were twelve parameters considered with three benchmark applications—sort-by-key, shuffling, and Kmeans. They obtained significant performance improvements by using KryoSerializer rather than using the default Java Serializer, and the speed-up achievement was 10-fold. In their second work [24], they proposed an alternative systematic methodology for parameter tuning which can be applied to any computing infrastructure. They identified that the number of cores of the Spark executor has most impact; and the level of parallelism—for example, the number of partitions per participating core—plays a significant role in maximising the performance improvement.

Muhammad Usama Javaid et al. [25] proposed a robust performance model based on a machine learning (ML) algorithm. In order to train the ML algorithm, they used various amounts of input data, sets of spark parameters, and features.

A complex data-driven workflow application was proposed by Gulino et al. [26] where they combined ML and an analytical model to predict the execution times of arbitrary complex workflow applications.

Cheng et al. [27] proposed a machine-learning-based, efficient, performance prediction model for Apache Spark. This technique was capable of predicting the execution times accurately for the given application and configurations. At the stage level, Adaboost was used to build the model. They used projective sampling and data mining techniques to mitigate the modelling overhead. They claimed that the proposed model offers three advantages: no prior assumption of configuration parameters, stand-out robustness and stability, and less overall cost for the modelling process. They found that the average prediction error of the model was only 9% as compared to the other techniques.

In their second work, Cheng [11] proposed combined multi-object optimization (MOP) and an Adaboost algorithm to find the optimal configuration of parameters and predict the model's performance. They evaluated the system with six Spark benchmarks. Five different datasets were used to analyse the performance. They claimed that the model can find the appropriate configuration setup and minimise the time and computational cost. The average improvement in computational cost was about 35% to 40%.

Aziz et al. [28] presented the resource management and data processing, the system processing time and speed-up, and the impact of persistence of resilient distributed datasets (RDDs) in Spark based on machine learning algorithms. In this analysis, the appropriate storage level of execution time was presented for Spark using a machine learning algorithm in RDD. They found that the speed-up does not improve by adding additional nodes, and the performance is degraded; and the total processing time increased significantly. There were many factors behind this degradation: among them, the most significant reason was the 100% allocation of cores to executors.

Boden [29] proposed a representative set of machine learning algorithms (supervised and unsupervised) to investigate large-scale datasets. The mathematical variation and appropriate system parameters were tuned for the amount of data and dimensionality of the data. The author reported that machine learning algorithm problems exhibit very high dimensionality due to data scaling and model size scaling. Therefore, they focused on the aspects likely affecting scaling the data and scaling the model's dimensionality. Their study found that as the the amount of data was increased, the system exhibited linearly increments in time consumed.

A cost–benefit Spark performance prediction model based on a machine learning algorithm was proposed by Maros [30]. They have proposed both black-box and grey-box models based on four machine learning algorithms. They considered three different aspects: the amount of training data, platform configurations, and workloads. They compared their model with Ernest [31]. They found that the performance estimation error was better than Ernest when the dataset extrapolation was required. Mustafa [10] proposed a new platform to predict the execution time for SQL queries and machine learning applications. This technique is very similar to the grey-box model. They applied three different approaches which used existing methods to predict the execution times of the queries. The authors claimed that the SQL query workload produced less than 10% error, whereas the machine learning workload produced less than 25%.

An exciting system was proposed by Amannejad et al. [32], which can predict the execution time in a short time. In this method, minimum resource settings are considered, which do not have complex dependencies and parallel stages. An application is used to analyse the work log files. This method requires two reference files, and the files are relatively small. This method had excellent accuracy regarding execution time, where the average prediction error of the workloads was about 4.8%. Unlike this work, they considered only a single node cluster, not a real cluster environment.

In a related but alternative model, PERIDOT was presented by Amannejad et al. in their second paper [33]. A small subset of input data and fixed limited cluster resources settings were considered to get quick execution time. They analysed the logs from both the executions and checked the internal dependencies between the internal stages. There were eight HiBench workloads used this experiment. They reported that the data partitions and the number of executors had significant impacts on execution time. This method had an overall mean prediction error of 6.6%, except for naive prediction techniques.

We summarise the different approaches in Table 1.

Unlike our approach, other approaches described in the literature may require time to modify several default parameters, which are very complex and tedious to work with. Apart from this, machine learning models usually require a large number of experiments in order to generate enough data for model training. Our proposed models need very few experiments, fitting the data obtained into simple equations. The equations can also

give some insights into the pattern of communication between the nodes when running Spark jobs.

Table 1. The recent approaches to Spark performance prediction.

References	Approach/Method	System/Environments
Cheng et al. [27]	Machine Learning	Efficient performance prediction for Apache Spark.
Ahmed et al. [34]	Comprehensive Trial-and-Error	Apache Hadoop and Apache Spark for large scale datasets.
Al-Sayeh et al. [21]	Gray-box modelling	Runtime prediction of Spark jobs.
Shah et al. [33]	PERIDOT	Quick execution time predictions for Spark applications.
Aziz et al. [28]	Machine Learning	Resource management for efficient performance of Apache Spark.
Gounaris et al. [24]	Alternative Systematic	Spark parameter tuning.
Mustafa et el. [10]	Machine Learning	Predicting execution time of Spark jobs.
Chao et al. [35]	Gray-box modelling (Machine Learning)	Spark performance model for accuracy improvements.
Petridis et al. [8]	Trial-and-Error	Spark parameter tuning.

4. Parallelisation Models

4.1. Amdahl's Law and Gustafson's Law

If no communication between the various executors is needed to run a job, the job is called "embarrassingly parallel" [13]. The implication of having no need to communicate between different executors is that the speed-up is proportional to the number of executors; i.e., if one executor takes time t, then n executors will take time $\frac{t}{n}$. However, any small portion of the job that is not parallelisable can bring major consequences for parallel performance. In this case, the linear speed-up achieved by adding more executors (in the form of CPUs or cores, or separate nodes) may decline sharply.

Amdahl came up with a generic equation to predict the speed-up factor of a parallel application as a function of the number of processors [36]. The equation considers that parts of the application (or job, or workload) are inherently serial in nature and would not be parallelisable.

$$S(nexec) = \frac{nexec}{1 + (nexec - 1)\, f_{np}} \tag{1}$$

where $S(nexec)$ is a function that represents the speed-up as a function of the number of executors, $nexec$ is the number of executors (often interpreted and nodes or CPUs available in the infrastructure), and f_{np} is the factor of non-parallelisable portions of a job. $f_{np} = 0$ represents a perfectly parallelisable job that will yield full speed-up (e.g., if there are 10 executors available, the job will run 10 times faster, or $S(10) = 10$).

From Equation (1), and considering that a single processor takes time t to run a certain workload, the predicted runtime running on multiple processors would be:

$$runtime = \frac{(1 - f_{np})\, t}{nexec} + f_{np}\ t \tag{2}$$

where t is a hypothetical runtime needed to run a job in a single executor.

If we consider the size of the job, we can modify Equation (2) to:

$$runtime = a\, f(Size) \left(\frac{(1 - f_{np})}{nexec} + f_{np} \right) \tag{3}$$

where a is a constant coefficient, and $f(Size)$ is a function that reflects the growth of the runtime with increasing sizes (an approximation of the algorithm complexity). As most of the workloads implemented in HiBench are either linear or quadratic, $f(Size)$ can be replaced by either $Size$ or $Size^2$.

An example of what Amdahl's law means for different serial factors and numbers of executors is shown in Figure 2. It shows the influence of both parameters on the runtime of a simulated job with a fixed dataset size.

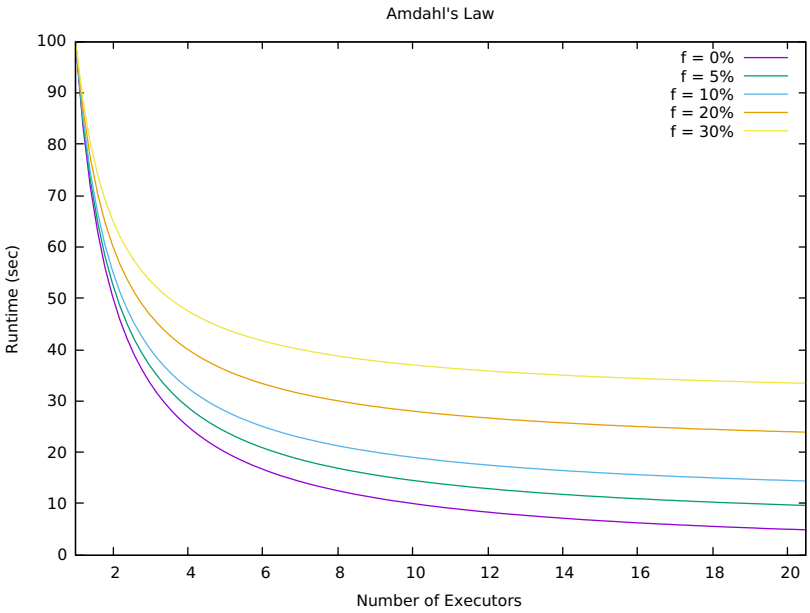


Figure 2. Amdahl's law for various serial factors and numbers of executors.

A few years after Amdahl's publication, Gustafson argued that the percentage of the serial part of a job is rarely fixed for different problem sizes [37]. In Amdahl's law even a small percentage of serial work can be detrimental to the potential speed-up after adding more executors. Gustafson noticed that for many practical problems the serial portion would not grow with an increase in problem size. Gustafson's speed-up equation is:

$$S(nexec) = nexec + (1 - nexec)\, f_{np} \tag{4}$$

Additionally, the runtime equation as a function of $Size$ and $nexec$ can be written as:

$$runtime = \frac{a\, f(Size)}{nexec + (1 - nexec)\, f_{np}} \tag{5}$$

Both Amdahl's and Gustafson's equations show that runtime will always go down as the number of executors increases. However, often in practice the communication can impose an overhead, so runtime might increase after a certain limit on the number of executors. We compare Equations (3) and (5) to our own model of parallelisation, as discussed in the next section.

An example of what Gustafson's equation (5) means for different serial factors and numbers of executors is shown in Figure 3. It shows the influences of both parameters on the runtime of a simulated job with a fixed dataset size.

4.2. A Model Using a 2D Plate Communication Pattern

In our previous work [14] a model using a 2D plate communication pattern based on a description by [13] was proposed and tested. In that work, we used the following equation, which is a function of *nexec* only (fixed problem sizes):

$$runtime = \frac{a}{nexec} + b\sqrt{nexec} \tag{6}$$

where a and b are coefficients, and *nexec* is the number of executors.

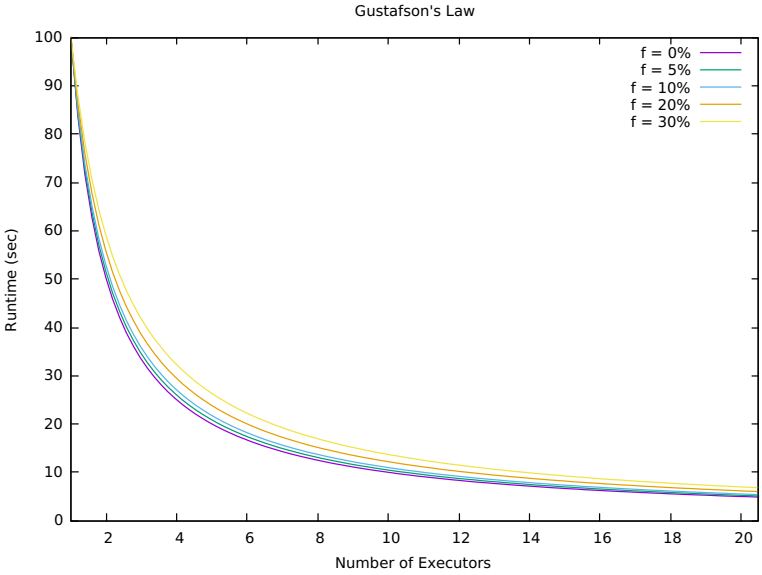


Figure 3. Gustafson's law for various percentages of serial work.

The equations were based on a communication model where each node has to exchange information with certain neighbours, but not all. Figure 4 shows an example of the boundaries of communication between nodes for this model.

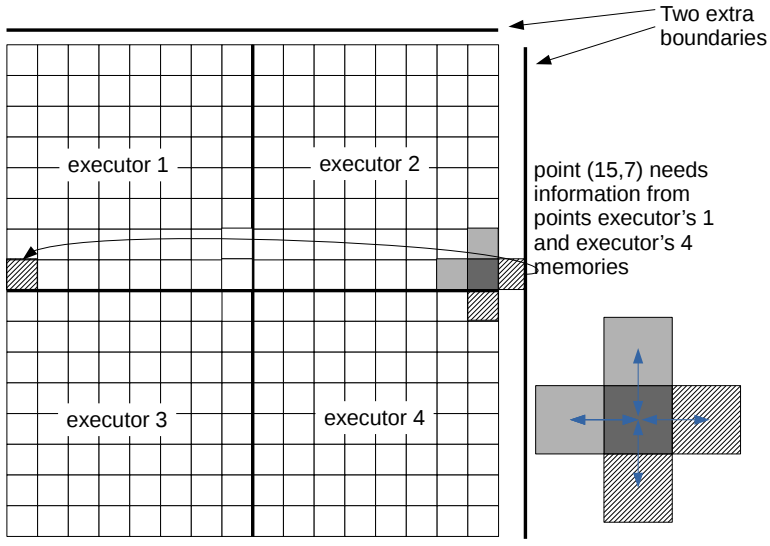


Figure 4. A 2D plate's homogeneous node communication.

The second part of Equation (6) is a function of $sqrt(nexec)$. We have also experimented with a different function, $nexec^c$, where $c <= 1$, and found that for some data it worked even better. Furthermore, based on [31], we added a constant term d to the equation to improve the fitting. Therefore, Equation (6) can be rewritten as:

$$runtime = \frac{a\, f(Size)}{nexec} + b\; g(Size)\; nexec^c + d \tag{7}$$

where $f(Size)$ is a function representing the time complexity of the workload when it runs on one executor, and $g(Size)$ is a function that indicates the growth of the communication and overhead when parallelising the job. After preliminary experiments, we found that $g(Size) = Size$ works well.

For linear algorithms, $f(Size) = Size$, so we can rewrite Equation (7) as:

$$runtime = \frac{a\; Size}{nexec} + b\; Size\; nexec^c + d \tag{8}$$

If the workload is quadratic, $f(Size) = Size^2$, and Equation (7) can be rewritten as:

$$runtime = \frac{a\; Size^2}{nexec} + b\; Size\; nexec^c + d \tag{9}$$

5. An Enhanced Model for Runtime Prediction

The model described in Section 4 has its limitations, as it assumes that each node would only communicate with a small number of neighbour nodes. We observed that although the model fits the data well for some workloads, it still may not reflect the communication that may be required when using HDFS, where copies of the data may be anywhere in the cluster. It is a known issue that for different algorithms, different communication patterns emerge [13], as the algorithm itself may require data located elsewhere or computations carried out by other nodes. In the proposed models, we took communication as a single factor, as this simplifies the model.

In order to expand the model to include both the number of executors and job sizes, we decided to reformulate the model. Thus, we considered that a node (where the executors have CPU resources) can communicate with any other node in the cluster. Although the communication pattern is not known for a black box implementation, we can infer what is happening through the empirical data acquired by running the same workload with many sizes and numbers of executors.

The communication pattern is assumed to be of the shape of a fully connected graph (see Figure 5). The assumption is that a function (unknown) of the size and number of executors drives the extra runtime needed to complete the communication between nodes. The extra runtime is, of course, compensated by the extra nodes involved in the job. Therefore, two components of the equation drive the runtime in opposite directions: the extra nodes will divide the processing to run the job, but communication between them requires extra time. The basic parallel equations start as:

$$runtime = \frac{t}{nexec} + t_{serial} \tag{10}$$

where t is the runtime for a job of a certain size to run in a single executor (no communications involved), $nexec$ is the number of executors, and t_{serial} is the serial portion of the job that cannot be parallelised, here considered to be communication overheads and any other overheads required to run the job in parallel.

If the size is added to the model, we need to know the algorithmic complexity of the implemented code.

$$runtime = \frac{f(Size)}{nexec} + t_{serial} \tag{11}$$

For simplicity, we assume that each node will communicate with every other node, and that the communication (be it HDFS or partial computations being exchanged between nodes) is symmetric and homogeneous. This makes the growth of t_{serial} a function of both the size and number of executors. We hypothesise that a good approximation for t_{serial} depends on the number of links between the nodes. This is the same as the number of edges in a fully connected graph:

$$nlinks = \frac{nexec(nexec-1)}{2} \tag{12}$$

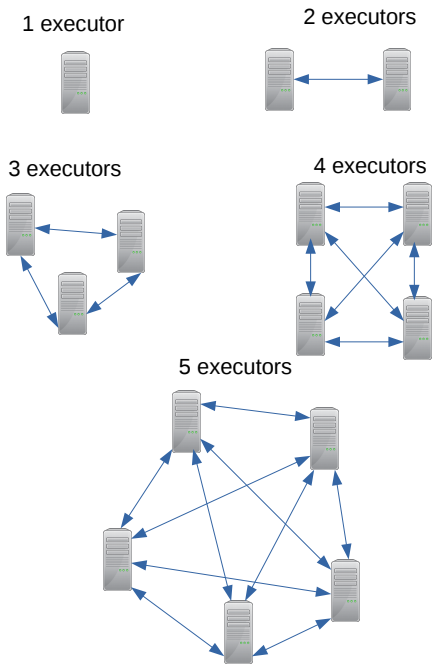


Figure 5. Communication model based on fully connected graphs.

Furthermore, the serial portion becomes:

$$t_{serial} = g(Size)\left(\frac{nexec(nexec-1)}{2}\right) \tag{13}$$

Equation (11) can be rewritten as:

$$runtime = \frac{f(Size)}{nexec} + g(Size)\left(\frac{nexec(nexec-1)}{2}\right) \tag{14}$$

In the parallelisable part of the runtime, the function $f(Size)$ can be simplified to the complexity of the algorithm implemented for the workload. However, for the serial part, the $g(Size)$ function is unknown. The data can fit well considering that $g(Size) = b\,Size^c$, where c is a constant exponent less than 1. For better fitting, we added another coefficient, d, representing a constant term for a given dataset (similarly to [31]).

For linear algorithms, $f(Size) = a\,Size$, so we can rewrite Equation (14) as:

$$runtime = \frac{a\,Size}{nexec} + b\;Size^c\left(\frac{nexec(nexec-1)}{2}\right) + d \tag{15}$$

If the workload is quadratic, $f(Size) = a\,Size^2$, and Equation (14) can be rewritten as:

$$runtime = \frac{a\ Size^2}{nexec} + b\ Size^c \left(\frac{nexec(nexec-1)}{2} \right) + d \tag{16}$$

6. Experiments

6.1. Experimental Setup

The experimental big data cluster used in this work was designed and developed by a group of academics at Massey University, Auckland campus [38]. The hardware for this experimental big data cluster is similar to a Beowful cluster. The cluster runs on dedicated network infrastructure with dedicated switches. All other network machines are kept away from this infrastructure to reduce the network latency and unwanted network resource utilisation. The cluster was designed and developed with one master node and nine slave/worker nodes. The Hadoop cluster server and node configuration is presented in Table 2. The schematic diagram of the cluster is presented in Figure 6.

Table 2. Experimental configuration of the Hadoop cluster.

Server Configuration	
Processor	2.9 GHz
Main memory	64 GB
Storage	10 TB
Node Configuration	
CPU	Intel (R) Xeon (R) CPU E3-1231 v3@3.40 GHz
Main memory	32 GB
Number of Nodes	9
Storage	6 TB each, 54 TB total
CPU cores	8 each, 72 total
Software	
Operating System	Ubuntu 16.04.2 (GNU/Linux 4.13.0-37-generic x86 64)
Hadoop	2.4.0
Spark	2.1.0
JDK	1.7.0

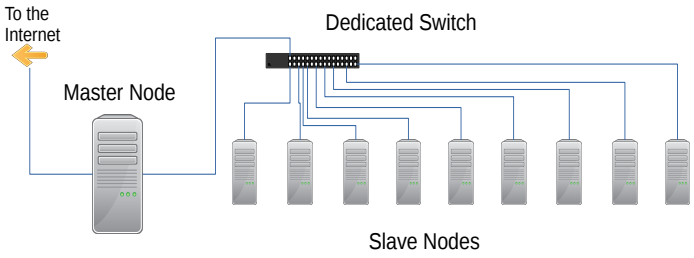


Figure 6. Schematic diagram of the Hadoop cluster used in the experiment.

6.2. Experiment Performance Evaluation

HiBench [39] is a popular big data benchmark suite that helps researchers and professionals to evaluate big data frameworks' performances. HiBench offers various characteristics and evaluates cluster deployment through comprehensive benchmarking [40]. It consists of various Hadoop programs, namely, synthetic micro-benchmarking and real-world applications. This experiment used five workloads from four different benchmark categories: Micro-Benchmark (WordCount), Machine Learning (Kmeans and SVM), Web Search (PageRank), and Graph (NWeight). The statistics of the experimental workloads

are presented in Table 3 and the workload of Spark HiBench's characteristics are presented in Table 4. Our target was to predict the Spark execution time considering the above workloads, and show how the execution time will be fitted with the proposed models. The individual workload DAG of stages and their execution are presented in the following section.

WordCount (WC): The WC workload performs the operation based on the Map function, which transforms the data into various representations. In HiBench, the WC input data are produced based on *RandomTextWriter*, which is contained in the Hadoop distribution. It counts the occurrences of separate words from the text or sequence file. An example of a job execution plan and its DAG of stage is presented in Figure 7. As shown in the figure, WC performed the operation in two stages; five tasks were involved in this operation.

Table 3. Spark HiBenchmark workload considered for this study.

Benchmark Categories	Application	Input Data Size		Input Samples
		Multiple-Exec.	Single-Exec.	
Micro Benchmark	WordCount	313 MB, 940 MB, 5.9 GB, 8.8 GB, and 19.2 GB	3 GB, 5 GB, 7 GB, 10 GB, 12.8 GB, 14.4 GB, 16 GB, 18 GB, and 21.6 GB	-
Machine Learning	Kmeans	19 GB, 56 GB, 94 GB, 130 GB, and 168 GB	1 GB, 38 GB, 75 GB, 113 GB, 149 GB, and 187 GB	10, 30, 50, 70, and 90 (million samples)
	SVM	34 MB, 60 MB, 1.2 GB, 1.8 GB and 2 GB	200 MB, 400 MB, 600 MB, 800 MB, 1.35 GB, 2 GB, 2.3 GB, and 2.5 GB	2100, 2600, 3600, 4100, and 5100 (samples)
Web Search	PageRank	507 MB, 1.6 GB, 2.8 GB, 4 GB, and 5 GB	100 MB, 250 MB, 750 MB, 6 GB, 7 GB, 8 GB, 9 GB, and 10 GB	1, 3, 5, 7, and 9 (million of pages)
Graph	NWeight	37 MB, 70 MB, 129 MB, 155 MB, and 211 MB	20 MB, 55 MB, 99 MB, 141 MB, 175 MB, 214 MB, 247 MB, 262 MB, and 286 MB	1, 2, 4, 5, and 7 (million of edges)

Kmeans: K-means is a well-known clustering algorithm that is commonly used for knowledge discovery and data mining. The Kmeans input data are a group of samples, generated by *GenKMeansDataset*, which is based on uniform and Gaussian distribution. We used various amounts of input data, such as tiny, small, and large, with the dimensions of 3 and 20; "no of cluster", 5; "max-iteration", 5; centroid, 10; and converged, 0.5. The job was executed in 19 different stages, and a sample of job DAG stages is shown in Figure 8.

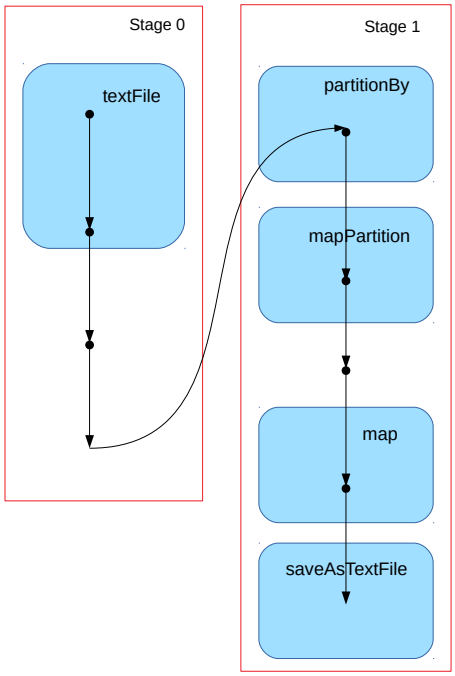


Figure 7. Spark stages DAG of WC.

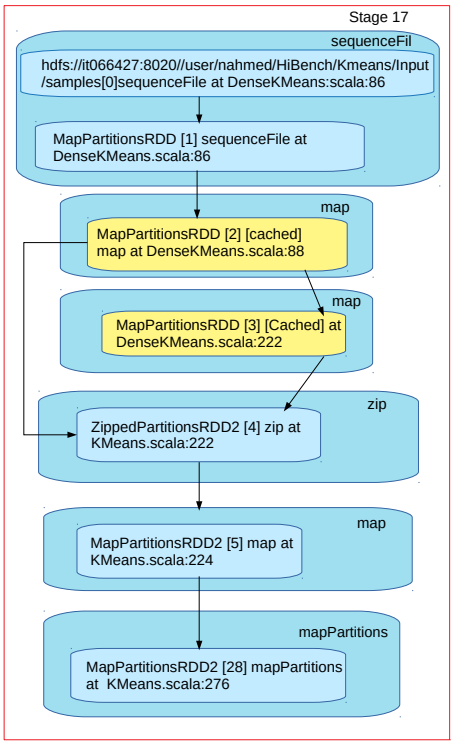


Figure 8. Spark stages DAG of Kmeans.

SVM: Support vector machines (SVMs) are used for large-scale data classification tasks. It is considered one of the standard methods of big data classification. In Spark, MLlib is used for SVM workload implementation. Its input data are generated by the SVM

DataGenerator. We selected the SVM parameters such as number of iterations, stepSize, and regParam and modified their values to 100, 1.0, and 0.01. The system required 213 stages to complete the task. Figure 9 shows a sample DAG of SVM.

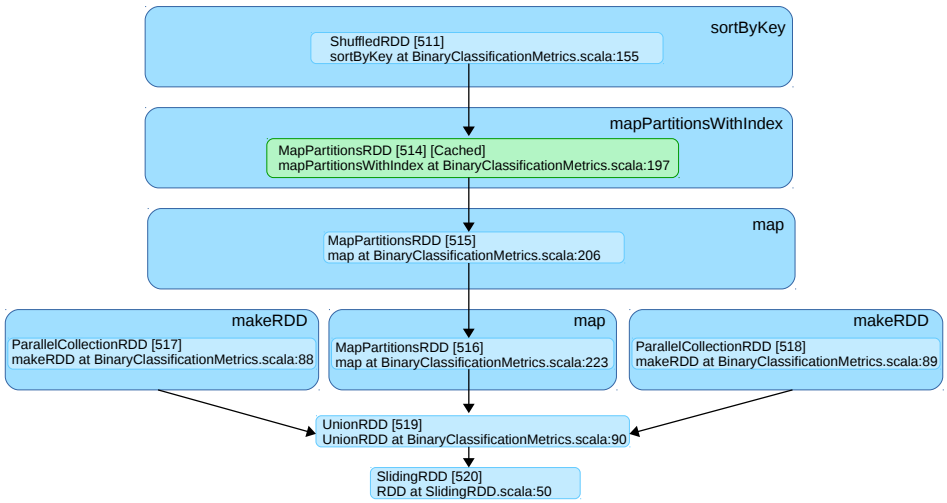


Figure 9. Spark stages DAG of SVM.

NWeight: NWeight is implemented by Spark GraphX library and pregel, and it works as an iterative parallel algorithm. It enhances the Spark RDD with a directed multigraph, which consists of properties enclosed with vertices and edges. The input files consist of millions of edges. It required eight different stages to complete the task. The workload DAG of stages is shown in Figure 10.

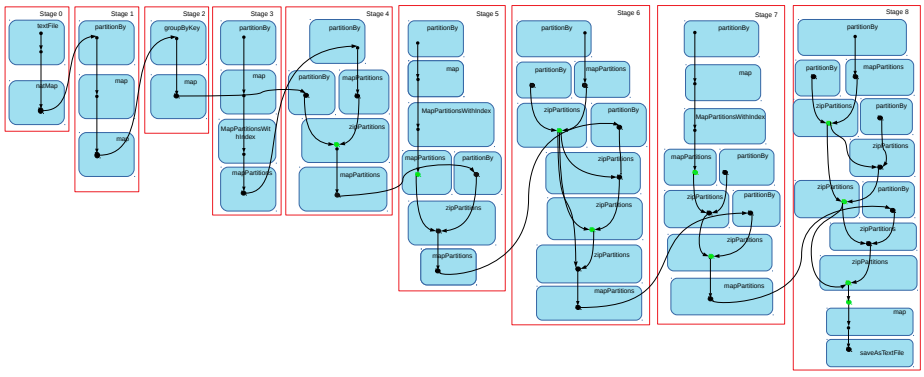


Figure 10. Spark stages DAG of NWeight.

PageRank: PageRank is a well-known page search algorithm where every page has a unique number, and an individual page is ranked as per the vote. The vote is counted when the pages are connected with the other pages. Generally, when a page is linked with several different pages, it is considered as a higher PageRank. In PageRank, the data source is generated from Web data. The hyperlinks of those data follow the Zipfian distribution. Various sets of input samples (from thousands to millions) were used in this experiment. The job was executed in four different stages. Figure 11 shows the workload DAG of stages.

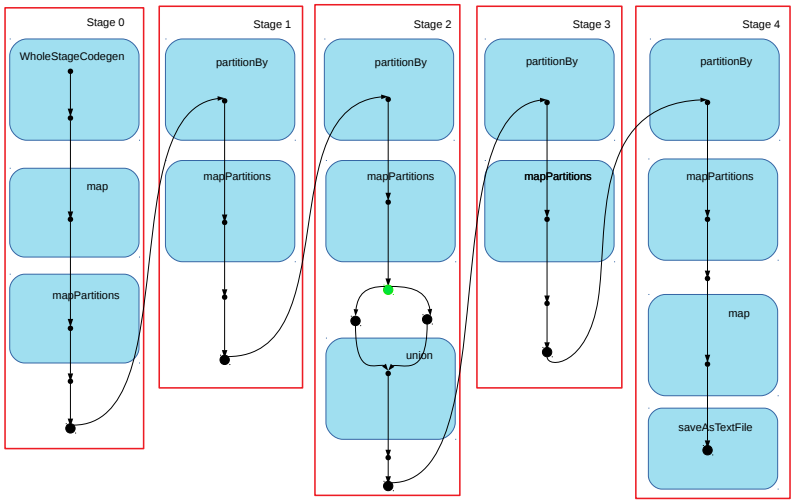


Figure 11. Spark stages DAG of PageRank.

Table 4. Workload application characteristics.

Workloads	Stages	Parallel Stages	Collect	Serialization	Deserialization	Shuffle	Aggregate
WC	2	no	yes	-	-	yes	-
SVM	209	no	yes	no	yes	yes	yes
NWeight	9	yes	-	no	yes	yes	-
Kmeans	20	no	yes	yes	yes	yes	-
PageRank	5	no	-	no	yes	yes	-

6.3. Configuration of Parameters

One of the challenging jobs of Spark cluster deployment is the parameter selection. There are more than 150 configurable parameters [8] in the Spark system, and each parameter plays a vital role in improving system performance. Spark's cluster performance relies on hardware infrastructure and accurate parameter selection. Performance improvements can be achieved by tuning the values of the parameters. The configuration of these parameters needs to be investigated according to the applications, amount of data, and cluster architecture. However, some influential parameters—*executors*, *executors core*, *executor memory*, *driver memory*, etc.—massively influence the system's performance. Besides, normally the number of parameters used is the default value. We have performed extensive experiments, selected the impactful parameters, and tuned their crucial factors to validate our cluster to get optimum performance. In the recent past, some studies [35,41] were carried out which illustrate the impacts and importance of the parameters. The chosen parameters for our study are listed in Table 5.

The default column in Table 5 presents the system's default configuration, the range column presents the tuned values used in this experiment, and the description column presents parameter information. There were two reasons to choose these parameters: firstly, Spark's runtime performance heavily depends on these parameters; secondly, these parameters control pivotal resources: CPU, disk read and write, and memory [42].

Table 5. Spark HiBenchmark parameters considered in this study.

Parameters	Default	Range	Description
Spark.executor.memory	1	12	Amount of memory to use per executor process, in GB.
Spark.executor.cores	1	2–14	The number of cores to use on each executor.
Spark.driver.memory	1	4	Amount of memory to use for the driver process, in GB.
Spark.driver.cores	1	3	The Number of cores to use for the driver process.
Spark.shuffle.file.buffer	32	48	Size of the in-memory buffer for each shuffle file output stream, in KB.
Spark.reducer.maxSizeInFlight	48	96	Maximum size of map outputs to fetch simultaneously from each reduce task, in MB.
Spark.memory.fraction	0.6	0.1–0.4	Fraction of heap space used for execution and storage.
Spark.memory.storageFraction	0.5	0.1-0.4	Amount of storage memory immune to eviction expressed as a fraction of the size of the region.
Spark.task.maxFailures	4	5	Number of failures of any particular the task before giving up on the job.
Spark.speculation	False	True/ False	If set to "true" performs speculative execution of tasks.
Spark.rpc.message.maxSize	128	256	Maximum message size to allow in "control plane" communication, in MB.
Spark.io.compression.codec	snappy	lz4/lzf/snappy	Compress map output files.
Spark.io.compression.snappy.blockSize	32	32–128	Block size in Snappy compression, in KB

7. Results and Analysis

In this part, we present the experimental findings. We have considered various amounts of data and systematically increased the number of executors to study the system's behaviour. For the results' reproducibility, each experiment was repeated at least three times, and the average execution time was taken into consideration in the final graph. We have collected the log files from the history server and calculated the job execution time using a Python script. We found that there is some fraction of the time difference between Ambari and our Python script. We have considered the most realistic time.

7.1. Procedure to Fit Equations

The procedure for the experiments for the HiBench workloads used for this work is summarised as follows. Firstly, for a certain workload, we ran a few jobs with different sizes using only one executor (Section 6.2). We then estimated the function $f(Size)$, which reflects the time complexity of the implemented algorithm (Section 7.2).

We ran more jobs with similar sizes as above, while varying the number of executors (Section 6.2). For each workload, we used up to 14 executors and five different sizes, chosen appropriately for each workload (Table 3).

Using the multi-parameter fitting function available in Gnuplot [43], each dataset was fitted to the following Equations: (8) (linear) or (9) (quadratic), (15) (linear) or (16) (quadratic), Amdahl (3), Gustafson (5), and Ernest [31] (Section 7.3).

The best fit for each equation above was chosen considering Rsquared and $\frac{RSE}{\mu}$ as a criterion, as discussed in Section 7.4.

7.2. Finding the Approximate Algorithm Complexity ($f(Size)$)

Regarding the nominal time complexity of the algorithms used in HiBench for these experiments, some are linear and some are quadratic. SVM is typically $O(N^2)$ or even $O(N^3)$ [44]. K-means is usually quadratic [45]. The PageRank algorithm can be $O(n * m)$, where n is the number of nodes and m is the number of arcs [46]. WordCount is usually linear $O(N)$ [47]. The Graph (NWeight) algorithm can be either linear or quadratic, depending on the graph representation. Using an edge list, it is quadratic $O(N^2)$ [48].

In order to find the function $f(Size)$ for Equation (11), we ran several jobs using a single executor. The results are shown in Figure 12.

Based on the residual standard error (given in Gnuplot [43] as the *rms* value) of the fitting to linear or quadratic trends, it was found that only SVM and NWeight had quadratic trends. The other methods produced linear trends. The appropriate equations were fitted to the complete data. Dataset size and number of executors were used as parameters for the model.

The possible complexity for the workloads and the actual data fittings for single executors are summarised in Table 6.

Table 6. Time complexity for the workloads.

Workload	**Theoretical Time Complexity**	**Single Executor** Best Fit $f(Size)$
WordCount	$O(N)$ [47]	linear
SVM	$O(N^2)$ [44]	quadratic
PageRank	$O(n * m)$ [46]	linear
Kmeans	$O(N^2)$ or $O(N)$ [45]	linear
NWeight	$O(N^2)$ or $O(N)$ [48]	quadratic

7.3. The Full Model Fitting

After running several jobs with different HiBench workloads, we collected runtime data for various sizes and numbers of executors. The data were fitted using equations for the fully connected model: Equation (14) (replaced by Equation (15) for linear complexity or (16) for quadratic complexity); and for the special case where $c = 1$, Amdahl's Equation (3), Gustafson's Equation (5), Ernest's equation [31], and the 2D plate model using Equation (7) (replaced by Equation (8) for linear complexity or (9) for quadratic complexity). The best fit is shown in the figures below, considering the fitting criteria described in Section 7.4. Table 7 presents the Rsquared results for each equation, and Table 8 presents the RRSE results for each equation.

Figure 13 shows the graph presentation of the WordCount workload for the amounts of data between 0.3 and 19 GB. The best fit used Equation (8) and there was a draw with Amdahl's equation, yielding an Rsquared of 0.997 and an RRSE of 0.074.

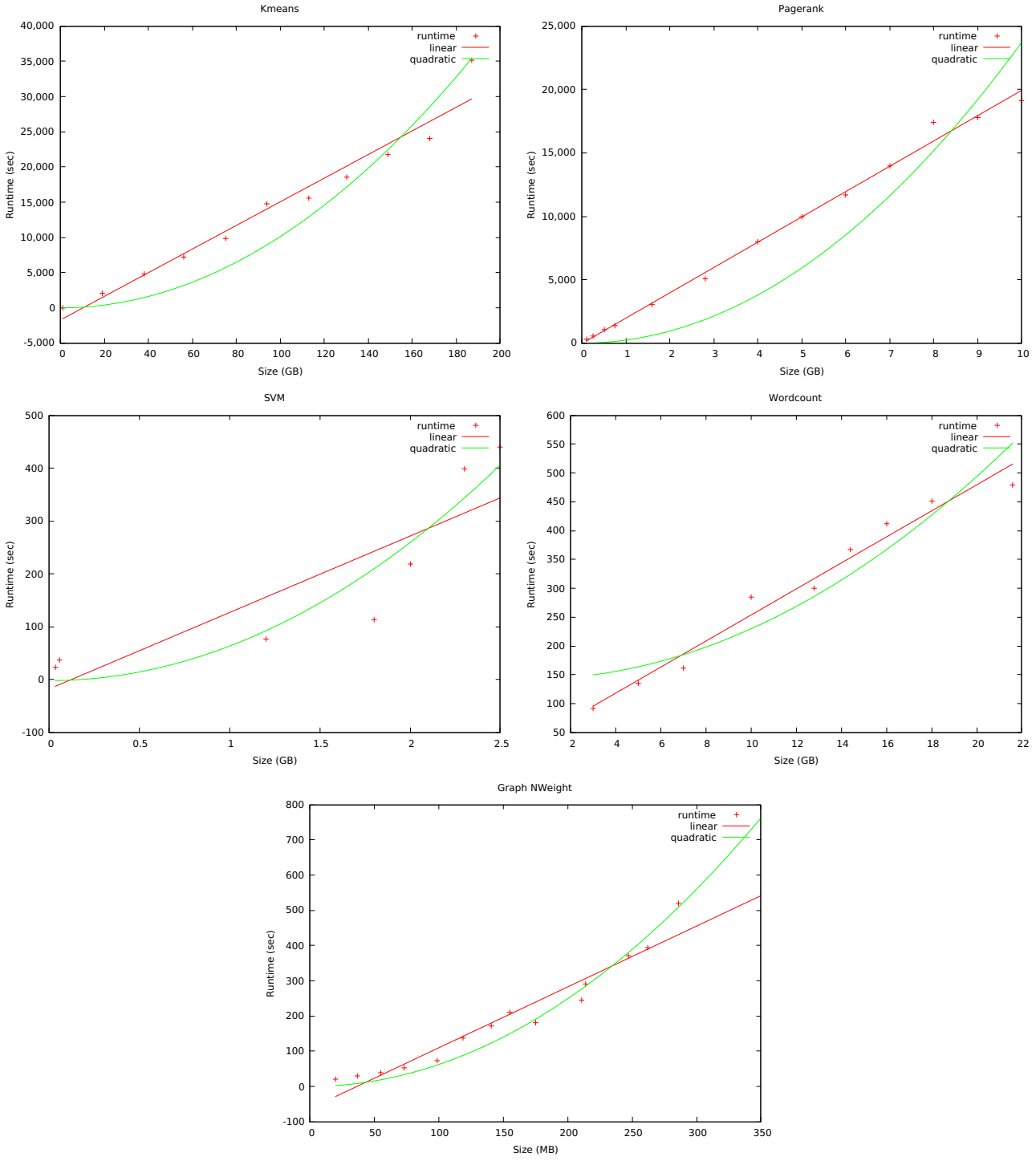


Figure 12. Single executor runtime complexity with different sizes.

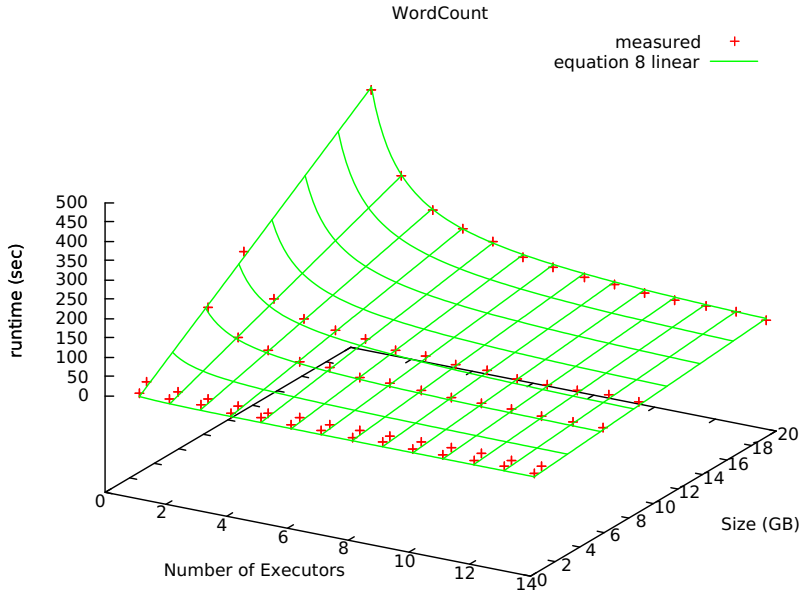


Figure 13. Fitting the model to WordCount workload for amount of data.

Figure 14 shows the SVM workload for sizes between 0.034 and 2 GB. Equations (9) and (16) were the best fit for the data, with an Rsquared of 0.917 and an RRSE of 0.271. The relatively high RRSE indicates that the data may be dependent on other factors, which may be investigated in future works.

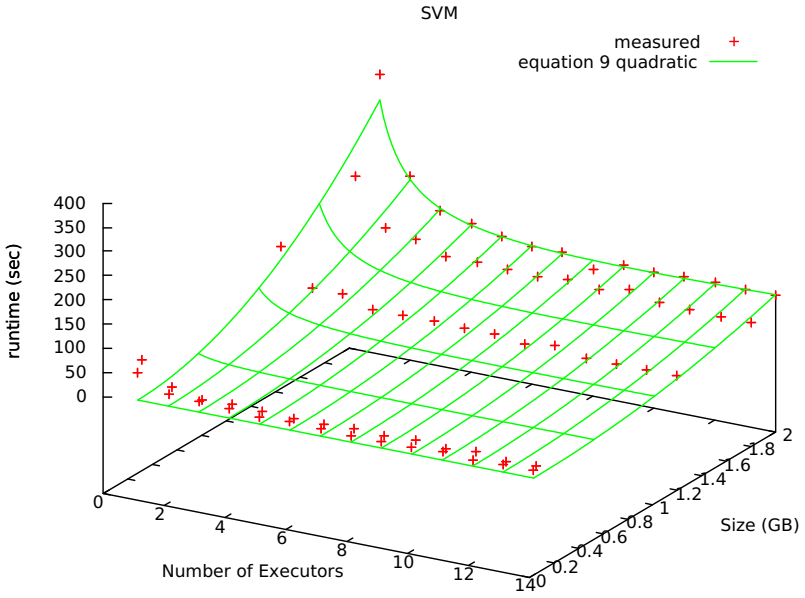


Figure 14. Fitting the model to SVM workload with different dataset sizes.

Figure 15 shows the PageRank workload for sizes between 0.057 and 5 GB. Equation (8) and Amdahl's were the best fit for the data, with an Rsquared of 0.990 and an RRSE of 0.113.

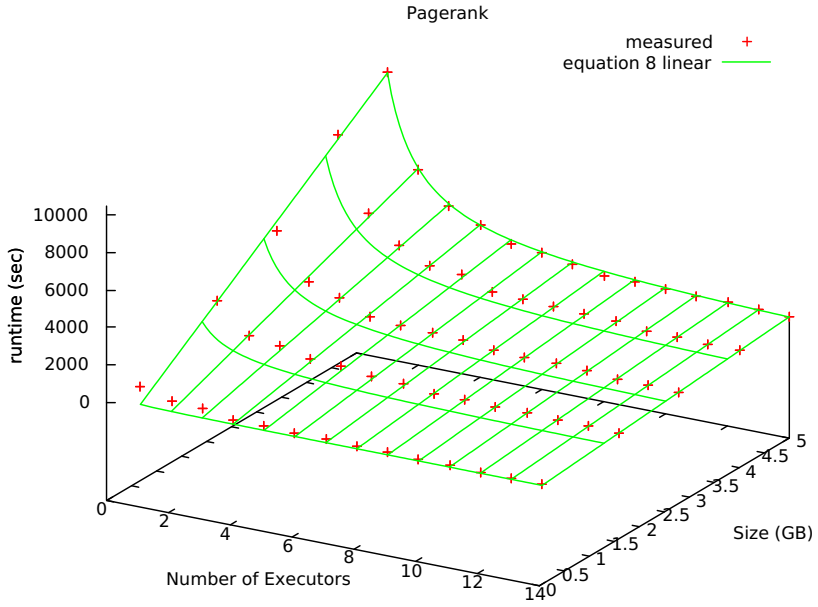


Figure 15. Fitting the model to PageRank workload for amount of data.

Figure 16 shows the Kmeans workload for sizes between 19 and 168 GB. Equation (8) was the best fit for the data, with an Rsquared of 0.993 and an RRSE of 0.130.

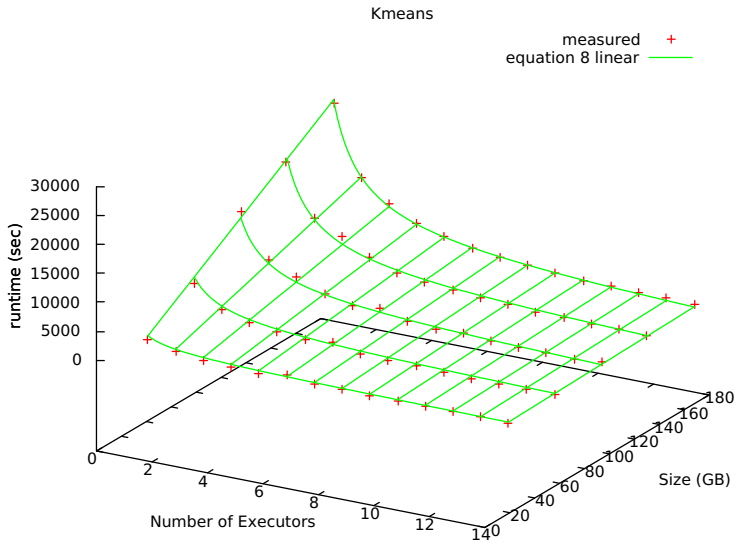


Figure 16. Fitting the model to Kmeans workloads with different sizes.

Figure 17 shows the Graph (NWeight) workload for sizes between 37 and 211 MB. Equation (9) was the best fit for the data with an Rsquared of 0.966 and an RRSE of 0.189.

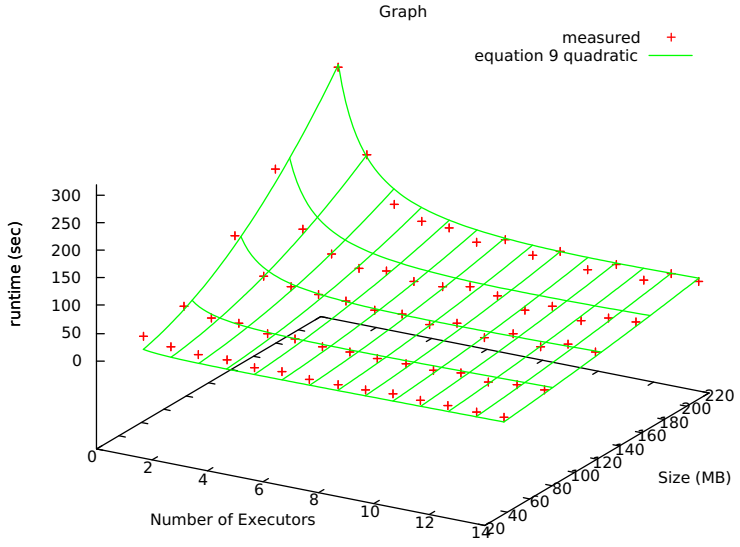


Figure 17. Fitting the model to Graph (NWeight) workloads of different sizes.

7.4. Evaluation of the Fitting Errors

The proposed models' fitting results are shown in Section 7. The nominal time complexities of different implemented algorithms in the HiBench workloads are also presented. This section illustrates the accuracy error of the proposed models, and shows the comparison results among the two proposed models and Amdahl's and Gustafson's laws. In addition, the proposed models offer an improvement over those of Ernest [31]. Our results revealed that accuracy and effectiveness of our proposed models are better than those of the Ernest models. We used both the Rsquared (R^2) and the relative residual standard error $\frac{RMS}{\mu}$ as metrics for the quality of the fitting. Rsquared values (also known as coefficient of determination) are calculated by the following equation:

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}} \tag{17}$$

where SS_{res} is the sum of the squares of the residuals and SS_{tot} is the sum of the squares relative to the mean of the data. For a perfect fitting, $SS_{res} = 0$ and $R^2 = 1$, so the closer R^2 is to one, the better the fitting.

The residual standard error [49] is:

$$RSE = \sqrt{\frac{\sum_{i=1}^{n}(y_i - \bar{y}_i)^2}{df}} \tag{18}$$

where $(y_i - \bar{y}_i)$ is the difference between the observed data and the predicted value using the model, and df is the degrees of freedom given by the number of samples minus the number of parameters being fitted.

The relative residual standard error is:

$$RRSE = \frac{RSE}{\mu} \tag{19}$$

The $RRSE$ gives a metric for the error or distance between the observed points and the ones generated by the model. The smaller the $RRSE$, the better the fit accuracy.

Table 7. Rsquared (Equation (17)) values for different models and workloads.

Workload	Best Fit	Equations (15) or (16)	Equations (15) or (16) (c=1)	Amdhal Equation (3)	Gustafson Equation (5)	Equations (8) or (9)	Ernest [31]
Wordcount	linear	0.996	0.996	**0.997**	0.996	**0.997**	0.995
SVM	quadrat.	**0.917**	0.912	0.906	0.887	**0.917**	0.847
PageRank	linear	**0.990**	0.989	**0.990**	0.989	**0.990**	0.988
Kmeans	linear	0.992	0.992	0.992	**0.993**	**0.993**	0.992
NWeight	quadrat.	0.964	0.964	0.956	0.965	**0.966**	0.950

Table 8. Relative residual standard error (RRSE Equation (19)) values for different models and workloads.

Workload	Best Fit	Equations (15) or (16)	Equations (15) or (16) (c=1)	Amdhal Equation (3)	Gustafson Equation (5)	Equations (8) or (9)	Ernest [31]
Wordcount	linear	0.083	0.083	**0.074**	0.082	**0.074**	0.091
SVM	quadrat.	**0.271**	0.276	0.285	0.313	**0.271**	0.367
PageRank	linear	0.116	0.118	**0.113**	0.121	**0.113**	0.127
Kmeans	linear	0.138	0.137	0.139	0.131	**0.130**	0.137
NWeight	quadrat.	0.193	0.193	0.212	0.190	**0.189**	0.226

7.5. Benefits of the Proposed Models

Our work considered three well-established equations, namely, those of Amdhal [36], Gustafson [37], and Ernest [31], as comparative models. We limited our analysis to these models while recognising the existence of alternative models in the published literature, which we deemed out of scope for the purposes of this study. As shown in Tables 7 and 8, for every workload, both proposed models (Equations (8) or (9), and (15) or (16)) had better fitting results than Ernest. Only in two cases did Amdahl's model Rsquared tie with our models, Wordcount and PageRank. Only in the case of Kmeans workload did Gustafson's model tie with one of the proposed models (Equation (8)). The results show that the two proposed models either tied with or performed better than the previously published models.

Considering the above results, the proposed models can be used as very effective tools for performance prediction, as they can offer several benefits for Spark job run time prediction using the Hadoop cluster. Several key benefits differentiate the proposed models from existing approaches. One of the crucial benefits is that using one of the proposed equations, it is possible to estimate the runtime. This can be achieved with a small number of experiments given the amount of data for the job and the chosen number of executors. The major advantage of the proposed models is that they do not require any trial-and-error approach, nor do they require large amounts of training or test data that are usually needed by machine learning models. Both models can capture the performance characteristics of a large number of complex workloads and are capable of predicting the runtime with good accuracy. Finally, the results also show that the models are highly effective, generic, and platform agnostic. Based on these models, it is possible for the managerial teams of big-data-driven organisations to minimise the time of their systems' configuring processes, plan and schedule large jobs by allocating critical resources for the clusters, and choose appropriate numbers of executors to maximise resource utilisation.

8. Conclusions

This paper proposed and investigated parallelisation models with enhanced capabilities for predicting the runtime performance of Apache Spark for several workloads running on Hadoop clusters. The configuration of Spark parameters is a complex and challenging task for the users. The system's performance mainly depends on the user's choice and targets.

To overcome this challenge, we proposed two models based on a function of the two most important parameters, the number of executors and the size of the job. A significant contribution of this work is the finding that with limited data points, one can fit the data into simple equations and understand the pattern of communication between the nodes when

running Spark jobs in a Hadoop cluster. We have found that the communication patterns can vary wildly between different workloads. This is expected, as different algorithms have different requirements from the Hadoop cluster. However, it can be noted that all the workloads used in the experiments could fit one of the two proposed models. The experimental results show that all the workloads could be fitted very accurately using the models proposed and completely outperformed the Ernest model. For two of the workloads (SVM and NWeight), the Rsquared values produced were lower than those the other three, alongside relatively high residual standard error. The two models fit the data better or at least as well as other alternative models (Amdahl, Gustafson, and Ernest).

However, the proposed models should be evaluated with other benchmark workloads, such as SQL and streaming. Due to time constraints, we considered only five workloads and selected a limited number of suitable Spark parameters. As future work, we have a plan to test the proposed model on the latest version of Apache Spark. Besides, we aim to add more suitable Spark parameters and workloads, and compare the proposed models with machine learning models. Furthermore, we intend to expand the experiments to other HiBench workloads to determine which equations are more suitable for which workloads.

Author Contributions: Conceptualization: N.A. and A.L.C.B.; methodology: N.A. and A.L.C.B.; resources: N.A.; validation: N.A. and A.L.C.B.; formal analysis: N.A. and A.L.C.B.; investigation: N.A. and A.L.C.B.; Data curation: N.A. and A.L.C.B.; writing—original draft preparation: N.A. and A.L.C.B.; writing—review and editing: N.A., A.L.C.B., M.A.R. and T.S.; visualization: N.A. and A.L.C.B.; supervision: A.L.C.B., M.A.R. and T.S. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the REaDI funding [Project code: 96670].

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Data are contained within the article. However, the correspondence author can be contacted for more details.

Acknowledgments: This work was supported in part by the Massey University Doctoral Scholarship.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

SVM	Support vector machines
API	Application programming interface
SQL	Structured query language
HDFS	Hadoop distributed file system
RDD	Resilient distributed datasets
MLlib	Machine learning library
CPU	Central processing unit
I/O	input/output
UC	University of california
AMP	Algorithms, machines and people
DAG	Directed acyclic graph
YARN	Yet another resource negotiator
PERIDOT	Performance predIction moDel fOr Spark applicaTions
NEXEC	Number of executor
SQRT	Square root
2D	Two dimensional
GHz	Gigahertz
TB	Terabyte
RAM	Random access memory

DDR	Double data rate
GB	Gigabyte
MB	Megabyte
WC	WordCount
Exec	Executor
MOP	Multi-object optimization

References

1. Katal, A.; Wazid, M.; Goudar, R.H. Big data: Issues, challenges, tools and good practices. In Proceedings of the 2013 Sixth international conference on contemporary computing (IC3), Nodia, India, 8–10 August 2013; pp. 404–409.
2. Dean, J.; Ghemawat, S. Mapreduce: Simplified data processing on large clusters. *Commun. ACM* **2008**, *1*, 107–113. [CrossRef]
3. Zaharia, M.; Chowdhury, M.; Das, T.; Dave, A.; Ma, J.; McCauly, M.; Franklin, M.J.; Shenker, S.; Stoica, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In Proceedings of the 9th Symposium on Networked Systems Design and Implementation(NSDI), San Jose, CA, USA, 25 April 2012; pp. 15–-28.
4. Mazhar Javed, A.; Rafia Asad, K.; Haitham, N.; Awais, Y.; Syed Muhammad, A.; Usman, N.; Vishwa Pratap, S. A Recommendation Engine for Predicting Movie Ratings Using a Big Data Approach. *Electronics* **2021**, *10*, 1215.
5. Zaharia, M.; Xin, R.S.; Wendell, P.; Das, T.; Armbrust, M.; Dave, A.; Meng, X.; Rosen, J.; Venkataraman, S.; Franklin, M.J.; et al. Apache spark: A unified engine for big data processing. *Commun. ACM* **2016**, *59*, 56–65. [CrossRef]
6. Meng, X.; Bradley, J.; Yavuz, B.; Sparks, E.; Venkataraman, S.; Liu, D.; Freeman, J.; Tsai, D.B.; Amde, M.; Owen, S.; et al. Mllib: Machine learning in apache spark. *J. Mach. Learn. Res.* **2016**, *17*, 1235–1241.
7. Kroß, J.; Krcmar, H. PerTract: Model Extraction and Specification of Big Data Systems for Performance Prediction by the Example of Apache Spark and Hadoop. *Big Data Cogn. Comput.* **2019**, *3*, 47. [CrossRef]
8. Petridis, P.; Gounaris, A.; Torres, J. Spark Parameter Tuning via Trial-and-Error. In *Proceedings of the INNS Conference on Big Data*; Springer: Cham, Switzerland, 2016; pp. 226–237.
9. Herodotou, H.; Lim, H.; Luo, G.; Borisov, N.; Dong, L.; Cetin, F.B.; Babu, S. Starfish: A self-tuning system for big data analytics. In Proceedings of the 5th Biennial Conference on Innovative Data Systems Research (CIDR), Asilomar, CA, USA, 9–12 January 2011; pp. 261–272.
10. Mustafa, S.; Elghandour, I.; Ismail, M.A. A machine learning approach for predicting execution time of spark jobs. *Alex. Eng. J.* **2018**, *57*, 3767–3778. [CrossRef]
11. Cheng, G.; Ying, S.; Wang, B. Tuning configuration of apache spark on public clouds by combining multi-objective optimization and performance prediction model. *J. Syst. Softw.* **2021**, *180*, 111028. [CrossRef]
12. Wang, G.; Xu, J.; He, B. A novel method for tuning configuration parameters of spark based on machine learning. In Proceedings of the 2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), Sydney, NSW, Australia, 12–14 December 2016; pp. 586–593.
13. Wilkinson, B.; Allen, M. *Parallel Programming*, 2nd ed.; Prentice Hall: Hoboken, NJ, USA, 1999; p. 268.
14. Ahmed, N.; Barczak, A.L.; Rashid, M.A.; Susnjak, T. A Parallelization Model for Performance Characterization of Spark Big Data Jobs on Hadoop Clusters. *J. Big Data* **2021**, *8*, 1–28. [CrossRef]
15. Mavridis, I.; Karatza, H. Performance evaluation of cloud-based log file analysis with apache hadoop and apache spark. *J. Syst. Softw.* **2017**, *125*, 133–151. [CrossRef]
16. Apache Spark Market Share. Available online: https://www.datanyze.com/market-share/big-data-processing--204/apache-spark-market-share (accessed on 9 October 2021).
17. Companies using Apache Spark. Available online: https://enlyft.com/tech/products/apache-spark (accessed on 9 October 2021).
18. Apache Spark Overview 2.4.4. RDD Programming Guide. Available online: https://spark.apache.org/docs/2.4.4/ (accessed on 7 August 2020).
19. Chen, Y.; Goetsch, P.; Hoque, M.A.; Lu, J.; Tarkoma, S. d-simplexed: Adaptive delaunay triangulation for performance modeling and prediction on big data analytics. *IEEE Trans. Big Data* **2019**, 1–12.10.1109/TBDATA.2019.2948338. [CrossRef]
20. Vavilapalli, V.K.; Murthy, A.C.; Douglas, C.; Agarwal, S.; Konar, M.; Evans, R.; Graves, T.; Lowe, J.; Shah, H.; Seth, S. Apache hadoop yarn: Yet another resource negotiator. In Proceedings of the 4th Annual Symposium on Cloud Computing, Santa Clara, CA, USA, 1–3 October 2013; pp. 1–16.
21. Al-Sayeh, H.; Hagedorn, S.; Sattler, K.U. A gray-box modeling methodology for runtime prediction of apache spark jobs. *Distrib. Parallel Databases* **2020**, *38*, 1–21. [CrossRef]
22. Assefi, M.; Behravesh, E.; Liu, G.; Tafti, A.P. Big data machine learning using apache spark mllib. In Proceedings of the 2017 IEEE International Conference on Big Data (Big Data), Boston, MA, USA, 11–14 December 2017; pp. 3492–3498.
23. Taneja, R.; Krishnamurthy, R.B.; Liu, G. Optimization of machine learning on apache spark. In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), Las Vegas, NV, USA, 25–28 July 2016; pp. 163–167.
24. Gounaris, A.; Torres, J. A methodology for spark parameter tuning. *Big Data Res.* **2018**, *11*, 22–32. [CrossRef]

25. Javaid, M.U.; Kanoun, A.A.; Demesmaeker, F.; Ghrab, A.; Skhiri, S. A Performance Prediction Model for Spark Applications. In Proceedings of the International Conference on Big Data, Honolulu, HI, USA, 18–20 September 2020; pp. 13–22.
26. Gulino, A.; Canakoglu, A.; Ceri, S.; Ardagna, D. Performance Prediction for Data-driven Workflows on Apache Spark. In Proceedings of the 2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOT.S.), Nice, France, 17–19 November 2020; pp. 1–8.
27. Cheng, G.; Ying, S.; Wang, B.; Li, Y. Efficient performance prediction for apache spark. *J. Parallel Distrib. Comput.* **2021**, *149*, 40–51. [CrossRef]
28. Aziz, K.; Zaidouni, D.; Bellafkih, M. Leveraging resource management for efficient performance of apache spark. *J. Big Data* **2019**, *6*, 1–23. [CrossRef]
29. Boden, C.; Spina, A.; Rabl, T.; Markl, V. Benchmarking data flow systems for scalable machine learning. In Proceedings of the 4th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond, Chicago, IL, USA, 19 May 2017; pp. 1–10.
30. Maros, A.; Murai, F.; da Silva, A.P.C.; Almeida, M.J.; Lattuada, M.; Gianniti, E.; Hosseini, M.; Ardagna, D. Machine learning for performance prediction of spark cloud applications. In Proceedings of the 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), Milan, Italy, 8–13 July 2019; pp. 99–106.
31. Venkataraman, S.; Yang, Z.; Franklin, M.; Recht, B.; Stoica, I. Ernest:Efficient performance prediction for large-scale advanced analytics. In Proceedings of the 13th Symposium on Networked Systems Design and Implementation (NSDI), Santa Clara, CA, USA, 16–18 March 2016; pp. 363–378.
32. Amannejad, Y.; Shah, S.; Krishnamurthy, D.; Wang, M. Fast and lightweight execution time predictions for spark applications. In Proceedings of the 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), Milan, Italy, 8–13 July 2019; pp. 493–495.
33. Shah, S.; Amannejad, Y.; Krishnamurthy, D.; Wang, M. Quick execution time predictions for spark applications. In Proceedings of the 2019 15th International Conference on Network and Service Management (CNSM), Halifax, NS, Canada, 21–25 October 2019.
34. Ahmed, N.; Barczak, A.L.; Susnjak, T.; Rashid, M.A. A comprehensive performance analysis of apache hadoop and apache spark for large scale data sets using HiBench. *J. Big Data* **2020**, *7*, 1–18. [CrossRef]
35. Chao, Z.; Shi, S.; Gao, H.; Luo, J.; Wang, H. A gray-box performance model for apache spark. *Future Gener. Comput. Syst.* **2018**, *89*, 58–67. [CrossRef]
36. Amdahl, G.M. Validity of the single processor approach to achieving large scale computing capabilities. In Proceedings of the AFIPS '67 (Spring): Spring Joint Computer Conference, Sunnyvale, CA, USA, 18–20 April 1967; pp. 483–485.
37. Gustafson, J.L. Reevaluating amdahl's law. *Commun. ACM* **1988**, *31*, 532–533. [CrossRef]
38. Barczak, A.L.; Messom, C.H.; Johnson, M.J. Performance characteristics of a cost-effective medium-sized beowulf cluster supercomputer. In Proceedings of the International Conference on Computational Science, Melbourne, Australia, 2–4 June 2003; pp. 1050–1059.
39. HiBench Suite. Available online: https://github.com/Intel-bigdata/HiBench (accessed on 4 June 2019).
40. Huang, S.; Huang, J.; Dai, J.; Xie, T.; Huang, B. The HiBench benchmark suite: Characterization of the mapreduce-based data analysis. In Proceedings of the 2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010), Long Beach, CA, USA, 1–6 March 2010; pp. 41–51.
41. Zhao, Y.; Hu, F.; Chen, H. An adaptive tuning strategy on spark based on in-memory computation characteristics. In Proceedings of the 2016 18th International Conference on Advanced Communication Technology (ICACT), PyeongChang, Korea, 31 January–3 February 2016; pp. 484–488.
42. Marcu, O.C.; Costan, A.; Antoniu, G.; Perez-Hernandez, M. Spark versus flink: Understanding performance in big data analytics frameworks. In Proceedings of the 2016 IEEE International Conference on Cluster Computing (CLUSTER), Taipei, Taiwan, 12-16 September 2016; pp. 433–442.
43. Williams, T.; Kelley, C. Gnuplot 5.4: An Interactive Plotting Program. 2020. Available online: http://gnuplot.sourceforge.net/ (accessed on 7 July 2021).
44. Bottou, L.; Lin, C.-J. Support vector machine solvers. *Large Scale Kernel Mach.* **2007**, *3*, 301–320.
45. Li, X.; Fang, Z. Parallel clustering algorithms. *Parallel Comput.* **1989**, *11*, 275–290. [CrossRef]
46. Chen, P.; Xie, H.; Maslov, S.; Redner, S. Finding scientific gems with Google's PageRank algorithm. *J. Inf.* **2007**, *1*, 8–15. [CrossRef]
47. Goel, A.; Munagala, K. Complexity measures for map-reduce, and comparison to parallel computing. *arXiv* **2012**, arXiv:1211.6526.
48. Tomita, E.; Tanaka, A.; Takahashi, H. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theor. Comput. Sci.* **2006**, *363*, 28–42. [CrossRef]
49. James, G.; Witten, D.; Hatie, T.; Tibshirani, R. *An Introduction to Statistical Learning*, 2nd ed.; Springer: New York, NY, USA, 2021.

Chapter 5

The contents of this chapter are from the following article. In accordance with the Springerlink open access policy, any full text that has been included, is the unmodified accepted article.

Ahmed *et al. Journal of Big Data* (2022) 9:67
https://doi.org/10.1186/s40537-022-00623-1

Journal of Big Data

RESEARCH Open Access

Runtime prediction of big data jobs: performance comparison of machine learning algorithms and analytical models

Nasim Ahmed[1*], Andre L. C. Barczak[1], Mohammad A. Rashid[2] and Teo Susnjak[1]

*Correspondence:
nasim751@yahoo.com

[1] School of Mathematical and Computational Sciences, Massey University, Auckland 0745, New Zealand
Full list of author information is available at the end of the article

Abstract

Due to the rapid growth of available data, various platforms offer parallel infrastructure that efficiently processes big data. One of the critical issues is how to use these platforms to optimise resources, and for this reason, performance prediction has been an important topic in the last few years. There are two main approaches to the problem of predicting performance. One is to fit data into an equation based on a analytical models. The other is to use machine learning (ML) in the form of regression algorithms. In this paper, we have investigated the difference in accuracy for these two approaches. While our experiments used an open-source platform called Apache Spark, the results obtained by this research are applicable to any parallel platform and are not constrained to this technology. We found that gradient boost, an ML regressor, is more accurate than any of the existing analytical models as long as the range of the prediction follows that of the training. We have investigated analytical and ML models based on interpolation and extrapolation methods with k-fold cross-validation techniques. Using the interpolation method, two analytical models, namely 2D-plate and fully-connected models, outperform older analytical models and kernel ridge regression algorithm but not the gradient boost regression algorithm. We found the average accuracy of 2D-plate and fully-connected models using interpolation are 0.962 and 0.961. However, when using the extrapolation method, the analytical models are much more accurate than the ML regressors, particularly two of the most recently proposed models (2D-plate and fully-connected). Both models are based on the communication patterns between the nodes. We found that using extrapolation, kernel ridge, gradient boost and two proposed analytical models average accuracy is 0.466, 0.677, 0.975, and 0.981, respectively. This study shows that practitioners can benefit from analytical models by being able to accurately predict the runtime outside of the range of the training data using only a few experimental operations.

Keywords: Big data, Performance prediction, Machine learning, System configuration, HiBench, Apache Spark, Extrapolation and interpolation

Introduction

Due to the massive amount of data generated by social media [1], public health [2], industry and natural language processing [3], data storing and processing becomes a challenging task for organisations [4]. The organisations require a fast processing and

intelligent system that can quickly process and present the insights of the data. Big data applications have become an ultimate choice in every organisation. There is a number of big data applications available, either in the form of physical clusters or cloud computing. In recent times cloud computing such as Amazon EC2, Google Cloud, Microsoft Azure has attracted tremendous attention. All these platforms allow the users to deploy their cluster virtually where they can choose and allocate resources according to their requirements. This virtualised platform also offers resources at very minimal prices. However, the enterprise needs to consider some data security concerns before selecting cloud computing services. On the other hand, the deployment of the physical Spark cluster is complex and expensive [5]. The physical cluster infrastructures offer numerous benefits and mitigate security concerns.

The deployment of these types of cluster infrastructures heavily depends on distributed parallel computing such as Apache Hadoop and Apache Spark. Due to the open-source, real-time data processing, and fault tolerance [6], Apache Spark has become an attractive framework after Hadoop. Spark supports various components, namely, MLliB for machine learning (ML), GraphX for image processing and Spark SQL [7] for structured data processing. More than 180 configurable Spark parameters play an essential role to support various types of jobs. Though the primary deployment of this cluster depends on the default parameters, however; Spark's performance heavily depends on its correct parameter selection and their configurations. The user must understand the relationship between the parameters and the cluster hardware availability and requirements because the parameter configuration and achieving optimum performance are always challenging and complex. The cluster parameter configuration is tedious work for the users because it requires a vast amount of time to configure and process data.

Due to this limitation, the performance prediction of this system is very challenging. In order to mitigate these challenges, several prediction models such as trial-end-error [8, 9], analytical [10], machine learning [11–14] were proposed by researchers but all these models have limitations; hence, in order to predict runtime for a certain job to run in a Hadoop cluster, one can use machine learning regression algorithms or equation fitting. Both methods need a certain amount of empirical data because there is no general analytic method that would cover different hardware and different configurations for a given cluster. In general, ML regression methods need more data to be accurate, specially if the predictions are made by extrapolation. On the other hand, equation fitting can be very accurate with very little data, but only if the equation reflects the true patterns of inter-node communication that emerges from the job execution. It is difficult to find a generic equation for a cluster because even specific algorithm implementations can influence the communication between nodes, and therefore a given forecasting equation can completely break down for a certain application.

In our previous works [15, 16], we have concluded that two parameters are crucial when determining the runtime: the *size* of the workload, and the *number of executors* available to run the job. We have tested two main models to generate equations that can fit empirical data. The first model assumed that limited communication happens between the nodes, working only with a certain number of neighbouring nodes. The second model assumed that a fully-connected graph between the nodes reflects the communication pattern. Also, in these models the complexity of the algorithm was taken

into consideration. Only two types of workloads were tested in terms of complexity of the algorithms, either they were linear or quadratic when considering the growth of the runtime as a function of the workload size.

The motivation and the key contribution of this paper are as follows:

- We accomplished extensive performance prediction accuracy comparison based on machine learning and existing analytical models. We achieved very good accuracy when only limited empirical data is available. Our underlying intention is that practitioners would run a few jobs, preferably with short runtimes, and be able to predict the runtime of longer untested dataset sizes.
- We investigated KRR regression parameter relationship between alpha and degree. Our analysis found that, for most of the workloads, the best R-squared can be achieved by selecting the small degree with alpha. Our analysis also found higher degrees can produce best R-squared but the data overfitting can be a major limitation. For the GBR regression, we kept all the parameters default.
- We extensively measured the analytical and ML regression models accuracy based on interpolation and extrapolation methods using k-fold cross validation. ML methods are not accurate when one tries to extrapolate predictions from small amounts of data. However, ML methods are much better at making adjustments to existing data, and can do interpolation very well [17]. The equations are derived to fit data well, and using the correct one can yield more accurate extrapolations of runtime forecasts than ML methods.

The remainder of the paper is organised as follows: "Apache Spark architecture" section provides a brief overview of the Apache Spark architecture. "Related work" section discusses some notable recent advances on Spark performance prediction using machine learning algorithms. "Prediction methods" section explains evaluation methods of both the analytical models and the ML regressors. "Experimental setup" section discusses the experimental setup while "Performance evaluations and analysis" section presents the performance analysis for the two approaches using interpolation method with cross-validation technique. "Performance analysis using extrapolation" section shows a detailed analysis of the extrapolation method, splitting the data into two categories, size and number of executors. "Discussion" section discusses the limitations of each approach, and the consequences of extrapolating data with a small number of experiments. Finally, "Conclusion" section concludes the paper with hints for extending the work in future.

Apache Spark architecture

Apache Spark is a parallel data processing framework that can rapidly process large amounts of data, often in real-time [18]. It can also perform data processing in the distributed cluster platform. Apache Spark has become an open access [6] project, and a popular data processing engine in many organisations. Its development has centred at the University of California, Berkeley's AMPLAB by the group of researchers that Matei Zahari led in 2009 [19]. The Spark codebase is donated to the Apache foundation as an open-source tool and has been maintained since then. In 2010, Spark proposed a Resilient Distributed Dataset (RDD) [20] that mitigates the limitation of the MapReduce

cluster computing paradigm. It works as an immutable collector of the objects. RDD splits the input data set into logical partitions and the partition data stored in the memory where worker nodes compute parallel operations. Spark RDD has two operations: transformation and actions. The transformation function uses the existing RDD as input and produces the new RDD from the existing one. Whenever the transformation function becomes active, it creates a new RDD. The action operation activates when it works on the actual dataset. A typical Apache Spark architecture representation is shown in Fig. 1.

The Spark application has three important components: the spark driver program, Spark executor, and the resource manager, where the action operation is forwarded from the executor towards the driver. The driver converts the user code in most tasks, and the executors run the code among the nodes. In this operation, the cluster manager is responsible for the resource allocation in the cluster. The cluster manager allocates the resources whenever the Spark driver program [21] requests and shares the information with the worker nodes. In Spark, the workflow is managed by a directed acyclic graph (DAG) [22]. The DAG consists of sequences of vertices and edges. The vertices represent the RDDs, and the edges represent the operation of the RDD. The DAG forwards the new job towards the stage level. The task consists of the initial input data and the RDD partition at each stage level. Spark creates two stages with the submitted job; firstly, ShuffleMapStage and secondly, ResultStages. At the ShuffleStage, the output data is stored for the following stages in the DAG. At the ResultStage, either single or multiple partitions functions are targeted the RDD. Spark can operate with many programming languages, such as Java, Scala, Python and R, and supports Spark SQL, ML, GraphX processing, and Spark Streaming. These programming language libraries offer comprehensive benefits for the user to develop applications. Spark allows the integration of various tools from the Hadoop technology ecosystem, where the resource management and job scheduling is maintained by Apache YARN (Yet Another Resource Negotiator) [23]. A cluster monitoring tool like Ambari assists with the monitoring the workloads running in the cluster.

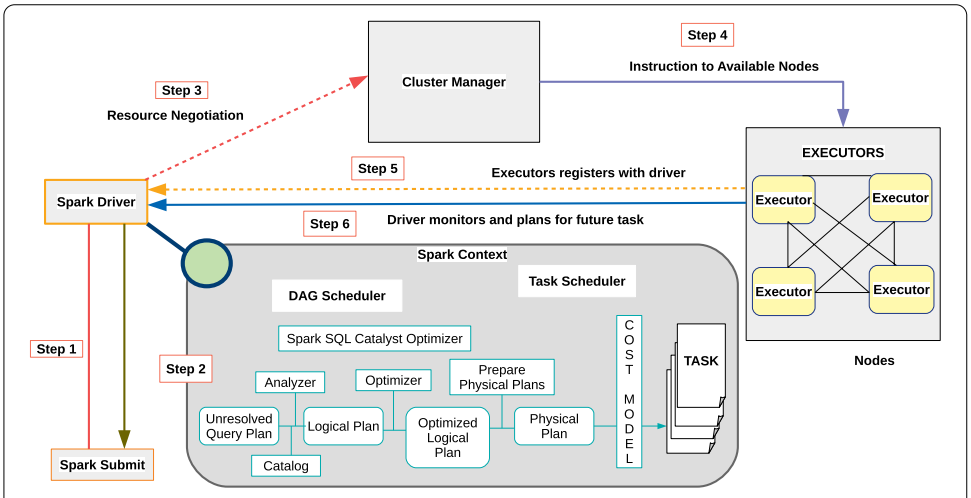


Fig. 1 A typical Apache Spark architecture modified from [21]

Related work

The runtime performance prediction of big data processing on a cluster is a challenging task. In the recent past, many prediction techniques [8–10, 24], Gray-Box techniques [25–27] and auto tuning techniques [12, 13, 28–30] have been proposed by researchers. However, the ML approach has become very popular and has received significant attention. In the following section, we will present recently published works based on ML techniques.

Prediction using machine learning

Douglas de Oliveira et al. [31] proposed an interpretable predictive ML model based on decision trees from which patterns are extracted. The decision tree model is used to classify the parameter performance by considering the training data. They used the extracted patterns and configured the system parameters for the workflow execution that significantly improved the system performance. Besides, they also considered two essential aspects: input data partitions and distributed data partitions through nodes. They found that the proposed predictive model can achieve 70% accuracy, and the accurate data partitioning knowledge can help choose the workflow function.

Christoph Boden et al. [32] presented an interesting work using ML algorithms for a large-scale distributed settings of Apache Spark and Flink performance. They implemented analytical models which are similar to ML algorithms and tuned the parameters to assess the scalability of the system concerning the data size and dimensionality of the data. They carried out a comprehensive investigation based on a single-node implementation with data size and data dimensionality. They found that several ML algorithm problems exhibit high dimensionality due to data scaling and model size scaling. So, they employed both supervised learning algorithms (batch gradient descent and TreeAggregate) for Flink and Spark, respectively. For the unsupervised learning algorithm, kmeans clustering was used. The proposed benchmark algorithm was placed on top of Apache Flink and Apache Spark and analysed the performance using non-representative workloads such as Wordcount, Grep, and Sort. They found that when the data size increased, the system behaviour exhibited a linear increment. They concluded that the system can perform robustly with the increasing data size of both Flink and Spark with 4.6 billion data points. Spark fails to train when the data size is beyond 6 million dimensions for scaling the model dimensionality. They concluded that current data flow systems could process an increased amount of data points but are incapable of coping with high dimensional data, which is a key requirement for large scale ML algorithms.

Christoph Boden et al. [33] proposed a novel data processing system based on a ML algorithm in their second work. This work categorized the proposed data processing system into three major groups: Clustering, Classification, and Recommender Systems. The raw data is transformed into extracted features for the data pre-processing, and the training data set is represented by a numerical data matrix. For this implementation, they have used kmeans, Batch Gradient Descent, and Matrix Factorization algorithms. As per their suggestion, logistic regression is a compelling choice for the prediction problem that can easily handle many data sets. They concluded that the latest data processing system requires more hardware resources to obtain a comparable prediction quality.

Ali Mostafaeipour et al. [34] presented an empirical analysis of the Hadoop and Spark frameworks that considers three criteria such as runtime, memory, and network usages. They implemented the K-nearest neighbour (KNN) algorithm on various datasets for both frameworks. This analysis demonstrated that with small data sets, Spark offers faster data processing than Hadoop. They also found that Spark is suitable for quick data processing because it processes the data in-memory. As for memory utilisation, Hadoop requires less memory than Spark, and Spark requires fewer network usages than Hadoop. Another empirical study of Apache Spark performance prediction based on ML algorithms is proposed by Mehdi Assefi et al. [35]. The authors have examined both qualitative and quantitative attributes of the framework. This study leverages the Apache ML library to handle big data analytics and evaluate the impact of multiple big data ML models such as classification and clustering on the different hardware and software configurations with big data analysis tasks. Some ML algorithms such as Support Vector Machine, Decision Tree, Naïve Bayes, Random Forest, kmeans are evaluated to analyse the ability of MLlib 2.0. They found that Apache Spark MLlib demonstrates better performance; in particular, this presented a noteworthy performance in terms of execution time.

Javaid [36] proposed a robust Spark performance prediction model based on ML algorithms. In this analysis, authors offered substantial experimental works and their applications with various data features. In order to build the performance model, they implemented four ML algorithms. They found that the gradient boost and Random Forest algorithm showed a better performance than the other algorithms on their datasets. In [37], the authors proposed a tool to predict the Spark application runtime before the deployment of the cluster. They claimed that the tool can be used for extensive Spark job profiling, determining the prior execution time and the system bottleneck. They claimed that the tool could predict a 20% error bound for the selected workloads. In [38], the authors proposed a ML-based auto-tune model for cluster parameter selection based on the Support Vector Regression (SVR) model and a practical end-to-end auto-tuning model by combining existing models with a smart search algorithm. They found that the overall performance of ML is much better than the traditional models. In particular, the SVR displayed the best performance for Sort. They concluded that the proposed model is robust and flexible, and adaptable to any changes.

Guoli Cheng [12] proposed a model based on the Adaboost ML algorithm. Adaboost is implemented at the stage level, and the classic projective sampling, including the data mining technique was applied to predict the Spark performance accurately. They used six benchmark workloads and five different data sizes. They concluded that the proposed model minimizes 9% runtime cost as compared to the previous model. In their recently published work [13], they stated that the performance trade-off heavily depends on the optimum configurations where the cost is an influential factor. So, they proposed a multi-object optimization algorithm model based on the Adaboost ML algorithm for Spark performance prediction. They applied six benchmark workloads and five different data sizes to evaluate the system performance. They claimed that the model can find the appropriate configuration setup and minimize the time and cost. They also concluded

that the proposed method can improve execution time performance by 30% and cost by 40%.

Table 1 summarises some notable studies by considering the models used and their performance based on selected workloads. It can be noted that most of the works used ML and very few works proposed analytical models, but the workloads and model performance metrics are not similar in these works, which make it difficult to compare the accuracy between them. To the best of the authors' knowledge, the literature has not presented any comparative performance analysis based on standard performance metrics because no standard performance metrics have been recommended.

Unlike the reviewed ML models described in the related work section, we compare analytical models [15] with ML (kernel ridge regression (KRR) [39] and Gradient Boost Regression (GBR) [40]), ERNEST [41], Amdahl [42] and Gustafson [43] models. The runtime prediction based on ML models shows satisfactory performance as per the published work, but all ML models require large input data. On the other hand, we have seen that our published models 2D-plate model (4) and fully-connected (5) model are very effective and can predict runtime accurately with limited data points.

Table 1 Various models on Spark performance prediction

Published work	Workloads and data sets	Models	Metrics (error and accuracy)
Cheng [12]	WordCount, Kmeans, TeraSort, PageRank, Bayes, and Nweight	Adaboost, ensemble learners, multiple learners, and projective sampling	Average accuracy error: Adaboost (30 cases): 9.02%, ensemble learners: 18.63%, multiple learners: 21.98%, projective sampling: 14.09%
de Oliveria [31]	Data sets: astronomy and bioinformatics Data partitions: 3	Decision tree (DT)	Prediction accuracy: best 3 scenario out of 7: SC1: 90.4%, 88.8%, and 86.5%
Boden [32]	WordCount, Grep, and Sort	Logistic regression (LR), and Kmeans	High data dimensionality
Boden [33]	Data set: CriteoClick Logs and Netflix Prize Kmeans, logistic regression (LG), matrix factorization (MF), and gradient boost regression (GBR)	Kmeans, logistic regression (LR), matrix factorization (MF), and gradient boost regression (GBR)	MF: required more time than single LibMF, LR: Spark MLlib required more hardware resources, GBR: better than LR
Assefi [35]	Data sets: HEPMASS, SUSY, HIGGS, LIGHT, HETROACT I and II	Support vector machine (SVM), decision tree (DT), Kmeans, NaiveBayes (NB), Weka, and random forest (RF)	t-test: $p < 0.01$
Javaid [36]	KMeans, PageRank, sorting, WordCount, binomial logistic regression, linear regression, groupby decision tree classifier, single source shortest path, and breadth first search	Linear regression (LR), random forest (RF) gradient boost machine (GBM), and neural networks (NN)	Average accuracy error: LR, GBM, RF, and NN 10% (approx)
Singhal [37]	Wordcount, Terasort, Kmeans and SQL	Multi linear regression (MLR), MLR-quadratic (MLRQ), support vector machine (SVM), and analytical model	Prediction accuracy error: MLR, SVM, and, MLRQ: MAPE 22%, analytical models: 80%
Cheng [13]	WordCount, Kmeans, TeraSort, PageRank, Bayes, and Nweight	AB-MOEA/D, random forest (RF), and two-stage tree (TSt)	Prediction error: AB-MOEA/D: 3.6%, RF: 8.97%, TSt:14.57%

Prediction methods

Machine learning algorithms

Many studies have explored supervised ML models for runtime performance prediction of large systems. These techniques are known as black box solutions because they can make predictions on previously collected data. In this supervised ML model, the training phase uses the experimental data that comes according to system configuration parameters. Indeed, the collection of these data is tedious and requires significant time resources. In this paper, we used two regression algorithms, kernel ridge regression (KRR) and Gradient Boost Regression (GBR), and implemented them based on Sklearn implementation. We choose gradient boost algorithm because it is a popular method for a large cluster setup [44], whereas the kernel ridge regression can perform cross-validation and predictive variance more efficiently on small and large data [45]. In this implementation, the Python programming language is used to evaluate the models' performance. We introduce the algorithms and their model operation principles in the following section.

1. Kernel ridge regression

   In 2000, Cristianini and Shawe-Taylor [39] proposed the kernel ridge regression (KRR) algorithm. KRR combines ridge regression with the kernel trick. For the linear kernel, this communicates with the linear function in the space induced by the respective kernel and data but for the non-linear kernel, this communicates with the non-linear function. The KRR is a simplified version of the Supervised Vector Regression (SVR), and it is also known as the least square support vector machine (LS-SVM). It uses different loss functions and twelve regularisations. Regularisation always uses positive floating point values, improves the problem complexity, and minimises the estimates' variance. In KRR, the kernel mapping works internally, and the parameters are passed through the pairwise kernel. A kernel function expressed as: $K : \mathcal{X}x\mathcal{X} \rightarrow \mathcal{R}$, is a function that is symmetric to $K(x_1, x_2) = K(x_2, x_1)$ and positive definite. In this study, we employed the Polynomial kernels from the Sklearn implementation [46]. In the Polynomial kernel [47], the assigned values and its distance calculate as per their assigned values where the parameter values must be positive. We can express the polynomial kernel expression as follows: $Parameters : \alpha, c, d$ and the kernel function: $K(X_1, X_2) = (\alpha_{X_1^T X_2} + c)^d$.
2. Gradient boost regression
   The Gradient Boost Regression (GBR) algorithm is a popular algorithm used for building predictive models and for large cluster setups [44]. In 2002, Friedman [40] proposed a modified version of the GBR algorithm based on a regression tree of fixed sizes. At the regression problem, the boosting approach works as a form of "functional gradient decent". The boosting approach is an optimisation technique that minimises the loss function of the training data. In this case, the loss function measures the difference between the predicted values and training data. GBR algorithm generates the learners iteratively by combining the weak learners into a single strong learner. The fixed size of multiple decision trees is used as a weak learner to build the GBR. In this study, GBR is used the default parameters within the sklearn [48]

implementation to evaluate the results. The GBR can be used in two ways, either as a regressor or classifier, with the former used in this study to predict the system runtime data.

Prediction models based on specific equations for parallel systems

For any parallel system, including Hadoop clusters running Spark, two parameters are the most influential in determining runtime: size and number of executors. In Spark, other parameters can deteriorate the performance. However, once these parameters relinquish enough resources for running a certain job, they do not have the ability to speed up the execution of that job. Therefore, while most parameters have a minimum threshold for the job to use the cluster's resources appropriately, one cannot improve the performance of a job beyond a certain point, limited by other factors such as size and number of executors available [9].

Also, in any parallel system the runtime has two components: parallelisable and non-parallelisable portions of time [49]. The parallelisable portion can be found as a function of the size of the job and the number of executors used. The non-parallelisable portion is more difficult as it depends on implementation and communication between nodes.

Since the early days of parallel systems, several models have been proposed for equations that can drive the runtime. Three important ones are Amdahl's law, Gustafson and Ernest. In our previous works, we have proposed two new models [15, 16] and have tested them against Amdahl [42], Gustafson [43] and ERNEST [41]. In order to compare the models, we adapted Amdahl's law and Gustafson's law as equations that determine runtime given the size and number of executors. These models use simple equations that can fit experimental data, and can be used to predict the runtime of jobs for different clusters, with specific hardware.

For completion, we summarise each model and the corresponding equations. For all the equations in this section, the following notations apply:

- S is the size of the workload (usually in GB),
- $f(S)$ is the function that expresses the runtime complexity of the algorithm,
- E is the number of executors, and
- a, b, c, d are the coefficients of the equations that need to be found via data fitting.

For $f(S)$, all the workloads used in this work were either linear or quadratic. Therefore, either $f(S) = S$ or $f(S) = S^2$. If the time complexity of the algorithm is known, its equation can replace $f(S)$.

Amdahl's law

In the early days of parallel systems, Amdahl proposed a performance model where the number of executors and the percentage of the non-parallelisable time drives the speedup of a job running with multiple executors when compared to the same job running on a single executor [42]. The equations can be modified to predict runtime given S and E:

$$runtime = af(S)\left(\frac{(1-b)}{E} + b\right) + c \tag{1}$$

Gustafson's law

Gustafson proposed an alternative model to that proposed by Amdahl [43]. The modified equation to predict runtime is:

$$runtime = \frac{af(S)}{E + b\,(1-E)} + c \tag{2}$$

ERNEST

More recently, Venkataraman et al. [41] proposed a model specifically for big data clusters called ERNEST. Their equation to predict runtime is:

$$runtime = \frac{a\,S}{E} + b\ log(E) + c\ E + d \tag{3}$$

2D-plate model

We proposed a 2D-plate model where the nodes communicate only with its direct neighbours [16] This model was based on insights by Wilkinson and Allen [49] that can be found in chapter 6, sections 6.3.2 and page 180. The details of how we derived equation (4) are in [16]. The equation is:

$$runtime = \frac{af(S)}{E} + b\ S\,E^c + d \tag{4}$$

Fully-connected node model

We also proposed an alternative model where the communication between nodes is assumed to work like a fully-connected graph. Both the 2D-plate and the fully-connected models were as accurate or more accurate than alternative models [15]. The equation for the fully-connected model is:

$$runtime = \frac{af(S)}{E} + b\,S^c\left(\frac{E(E-1)}{2}\right) + d \tag{5}$$

A special case of this equation was considered when the communication growth is linear in relation to the size S, i.e., $c = 1$:

$$runtime = \frac{af(S)}{E} + b\,S\left(\frac{E(E-1)}{2}\right) + d \tag{6}$$

Experimental setup

All our experiments have been conducted on a high-end Hadoop cluster. In 2016, the group of academicians and researchers designed and developed the cluster at Massey University, Auckland campus. This cluster is designed with a dedicated switch and

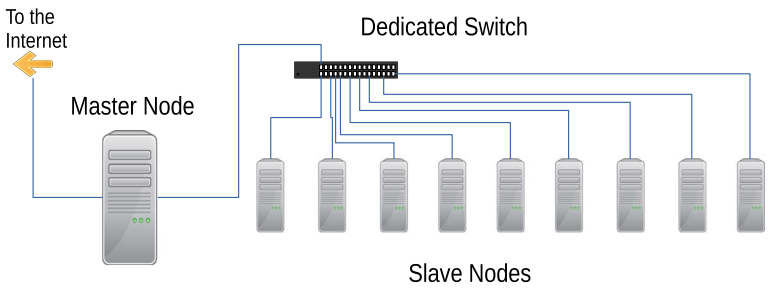


Fig. 2 A schematic diagram of the Hadoop cluster master and slave nodes used in the experiment

Table 2 Experimental configuration of the Hadoop cluster

Server configuration	
Processor	2.9 GHz
Main memory	64 GB
Storage	10 TB
Node configuration	
CPU	Intel (R) Xeon (R) CPU E3-1231 v3@3.40 GHz
Main memory	32 GB
Number of nodes	9
Storage	6 TB each, 54 TB total
CPU cores	8 each, 72 total
Software	
Operating system	Ubuntu 16.04.2 (GNU/Linux 4.13.0-37-generic x86 64)
Hadoop	2.4.0
Spark	2.1.0
JDK	1.7.0

different network infrastructures, similar to a Beowulf cluster [50]. In order to reduce the network latency and unwanted network resource utilization, all other network machines were isolated from this infrastructure.

A schematic diagram of the cluster is presented in Fig. 2, and the specifications for the servers and nodes are presented in Table 2.

HiBench workloads

It is a challenging task to evaluate the performance of a cluster. In the recent past, researchers presented CloudSuite [51] and CloudStone [52] benchmarks for cluster performance. Intel also proposed a Hibench suite under Apache Licence HiBench suite [53]. Since then, this has become a heavily used cluster performance testing tool, especially for Hadoop and Spark frameworks. The existing benchmark can be divided into three categories, such as Micro-Benchmarks, End-to-End benchmark, and Benchmark suite [54]. The Hibench suite has also been categorised into four categories: Micro-Benchmark, Web Search, SQL, and ML. In this experiment, there are five different workloads,

Table 3 Spark HiBenchmark workload considered for this study

Benchmark categories	Application	Input data size		Input samples
		Multiple-Exec.	Single-Exec.	
Micro benchmark	WordCount	313 MB, 940 MB, 5.9 GB, 8.8 GB, and 19.2 GB	3 GB, 5 GB, 7 GB, 10 GB, 12.8 GB, 14.4 GB, 16 GB, 18 GB, and 21.6 GB	–
Machine learning	kmeans	19 GB, 56 GB, 94 GB, 130 GB, and 168 GB	1 GB, 38 GB, 75 GB, 113 GB, 149 GB, and 187 GB	10, 30, 50, 70, and 90 (million samples)
	SVM	34 MB, 60 MB, 1.2 GB, 1.8 GB and 2 GB	200 MB, 400 MB, 600 MB, 800 MB, 1.35 GB, 2 GB, 2.3 GB, and 2.5 GB	2100, 2600, 3600, 4100, and 5100 (samples)
Web search	Pagerank	507 MB, 1.6 GB, 2.8 GB, 4 GB, and 5 GB	100 MB, 250 MB, 750 MB, 6 GB, 7 GB, 8 GB, 9 GB, and 10 GB	1, 3, 5, 7, and 9 (million of pages)
Graph	NWeight	37 MB, 70 MB, 129 MB, 155 MB, and 211 MB	20 MB, 55 MB, 99 MB, 141 MB, 175 MB, 214 MB, 247 MB, 262 MB, and 286 MB	1, 2, 4, 5, and 7 (million of edges)

Table 4 Workload application characteristics

Workloads	Stages	Parallel stages	Collect	Serialization	Deserialization	Shuffle	Aggregate
WC	2	No	Yes	–	–	Yes	–
SVM	209	No	Yes	No	Yes	Yes	Yes
Nweight	9	Yes	–	No	Yes	Yes	–
kmeans	20	No	Yes	Yes	Yes	Yes	–
Pagerank	5	No	–	No	Yes	Yes	–

comprising WordCount, kmeans, SVM, Pagerank, and NWeight. From four categories: Micro-Benchmark, ML, Web Search and Graph were taken into consideration. Table 3 presents the Spark Hibench workloads while Table 4 presents the workload application characteristics.

Cluster parameters configuration

In this work, a set of configuration parameters are considered to evaluate the performance of the system. Spark has more than 150 configurable parameters [8, 9] where the system performance heavily depends on the correct parameters selection. Therefore, we have judiciously selected only the parameters that are closely bound to system performance for evaluation purposes. However, cluster performance depends not only on the right parameters selection but also on tuning the parameters to achieve optimum system performance. We have seen the configuration of these parameters heavily depends on the cluster hardware, workload characteristics and size of the workloads. Out of numerous parameters, the most important parameters are the number of executors, executor memory, executor core size, and the driver memory. This experiment has therefore chosen a subset of only impactful parameters and tuned their values to achieve the best cluster performance.

Recently, several notable studies [26, 55] presented the importance and effectiveness of the outlined tunable parameters. Our study revealed that the right parameters selection is the primary requirement to get the best cluster performance. In our work, the chosen parameters are listed in Table 5. It can be seen from Table 5 that the default column presents the system default parameters, we tuned several parameters values including the default values that are listed in the range columns. Our investigation found that in most of the cases, the default values are not appropriate for our cluster performance. In some cases, for example *Spark.memory.fraction and Spark.memory.storageFraction*, there is no performance difference even if we use lower than the default values. On the other side, for example, *Spark.driver.memory, Spark.driver.cores, Spark.shuffle.file.buffer, Spark.reducer.maxSizeFlight*, the higher values showed better performance than the default values. Therefore, we have considered only those tuned values that are listed in the column of value used in the experiment column. The description of the parameters is presented in the description column. Our insight on the cluster performance, the selection of these parameters and their values are firstly based on the fact that the spark performance heavily depends on the available resources of the hardware. Secondly, these parameters and their values were chosen

Table 5 Description of selected Spark configuration parameters selected as the input of the proposed model

Parameters	Default	Range	Value used in the experiment	Description
Spark.executor.memory	1	1–12	12	Amount of memory to use per executor process, in GB
Spark.executor.cores	1	2–14	2–14	The number of cores to use on each executor
Spark.driver.memory	1	1–4	4	Amount of memory to use for the driver process, in GB
Spark.driver.cores	1	1–3	3	The number of cores to u for the driver process
Spark.shuffle.file.buffer	32	32–48	48	Size of the in-memory buffer for each shuffle file output stream, in KB
Spark.reducer. maxSizeInFlight	48	48–96	96	Maximum size of map outputs to fetch simultaneously from each reduce task, in MB
Spark.memory.fraction	0.6	0.1–0.4	0.4	Fraction of heap space used for execution and storage
Spark.memory. storageFraction	0.5	0.1–0.4	0.4	Amount of storage memory immune to eviction expressed as a fraction of the size of the region
Spark.task.maxFailures	4	4–5	5	Number of failures of any particular task before giving up on the job
Spark.speculation	False	True/false	–	If set to "true", performs speculative execution of tasks
Spark.rpc.message. maxSize	128	128–256	256	Maximum message size to allow in "control plane" communication, in MB
Spark.io.compression. codec	Snappy	lz4/lzf/snappy	Snappy	Compress map output files
Spark.io.compression. snappy. blockSize	32	32–128	32	Block size in snappy compression, in KB

since they control pivotal resources such as CPU, disk read and write, and memory. [56].

Performance evaluations and analysis

In this section, we present the comparative results between the analytical and ML models. To validate the system performance, we used five HiBench workloads with various data sizes. The system runtime characteristics are obtained by running jobs for five workloads using different number of executors and data sizes.

The proposed work is categorised into six stages: loading runtime data, data preprocessing, cross-validation and extrapolation methods, proposed models, ML models, and lastly, performance measurements (Fig. 3). To avoid overfitting and selection bias, a threefold cross validation process was used. The workload execution time is extracted from the Ambari history server log files, where a Python script is used to calculate the workload execution times. For the final graph presentation, each experiment was repeated at least three times, and the average time is considered as a final result.

We calculate the job execution time based on the job log files. We have collected all job log files from the Ambari history server and used a Python script to calculate the execution time. We found a fraction of time difference between the Python script and the Ambari server. One of the possible reasons for this time difference, Python scripts calculate log files independently while Ambari saves the execution time into the server, where the network latency can play an important role. In stage two of Fig. 3, data prepossessing is an essential step to achieve the best results from the models. Therefore, well-structured data is required to get the best performance from the models.

In stage three of Fig. 3 two types of data split were used. For the threefold cross-validation, a balanced split was used, where in both training and test data all sizes and number of executors are present in the data. For the extrapolation split, the training data receives all measured points in the middle of the range for either size or number of executors, also using the same proportion of data for the training set, with 66% of the data, and the test set, which uses the remaining 34% of the data.

In stage four, we applied the fitting to the equations of the analytical models (proposed and from the literature), and use two ML regression algorithms, namely KRR

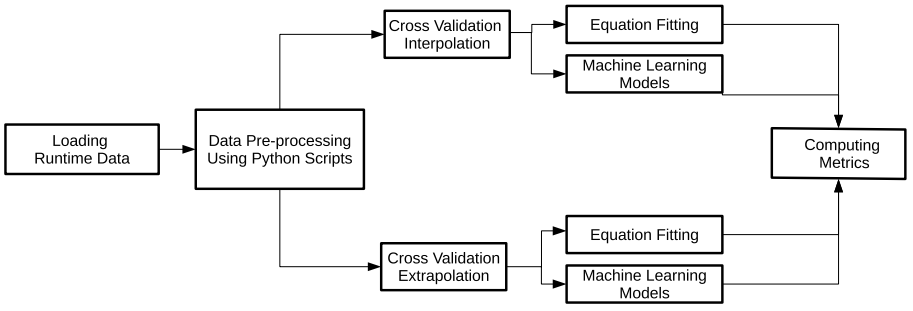


Fig. 3 The workflow of the performance analysis

and GBR. Finally, the performance of the models and ML regression is measured based on R-squared and Residual Relative Square Error (RRSE). Only the most accurate results have been used to present the graphs in Figs. 6, 7, 8, 9, 10.

Evaluation metrics

The choice of model evaluation metrics is an important factor for conducting comparative analysis. To verify performance in literature, researchers lean on a variety of metrics. In this study, the R-squared (R^2) and relative residual standard error (RRSE) are used. R-squared is used as a dominant index in regression algorithm to verify the predicted results accuracy, and RSE is used to determine the goodness-of-fit. The R^2 values (also known as Correlation Coefficient(R)) are presented as follows.

$$R^2 = 1 - \frac{SS_{Regression}}{SS_{Total}} \tag{7}$$

where SS_{rs} is the sum of the squares of the residuals and and SS_{tot} is the sum of the squares relative to the mean of the data. R-squared value is between 0 and 1. Higher values indicate a more optimal fit. The residual standard error is represented as follows:

$$RSE = \sqrt{\frac{\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2}{df}} \tag{8}$$

where $(y_i - \bar{y}_i)$ is the difference between the observed data and the predicted value using the model, and df is the degrees of freedom given by the number of sample size minus and the number of parameters being fitted. The relative standard error (RSE) is:

$$RRSE = \frac{RSE}{\mu} \tag{9}$$

The Residual Relative Standard Error ($RRSE$) metric allows us to distinguish the error between the observed points and the ones generated by the model. The smaller the $RRSE$, the better the fit accuracy.

Kernel ridge models

We used KRR from scikit-learn [46] implementation where only the polynomial kernel is considered, and others are ignored. For the model simplicity, we kept the coefficient and gamma parameter as a $default = 1$, and no other parameters but the different degrees and alpha values are examined to improve the model's accuracy. It can be noted that the proposed model produced the best R-squared results when $degree = 70$. Table 6 presents the best results of the individual workloads by measuring R^2, standard deviation and relative residual standard values (RRSE). Our study found that except for the Kmeans and Graph workloads, all three workloads produce the best results with degree 70 and alpha 1. In contrast, the Kmeans and Graph show the best results with $degree = 30$ and alpha value always achieves a better performance with $alpha = 1$. Our study also revealed that the small values of alpha improves the model performance and reduces the variant of the estimates for three workloads. We noticed that the model performance for the individual workloads is satisfactory. The R-squared comparison of KRR algorithm across different

degrees and alphas on the performance of selected HiBench workloads are shown in Fig. 4.

Despite showing a better R-squared value for higher degrees, polynomial regression has a known issue related to over fitting [57]. We showed the results of higher degrees to make the point that ML approaches using polynomials and easily overfit. One can see that the fitting follows the experimental data very well, and it can encompass the full range of the parameters. However, when showing an example of fitting, one can see that the higher degree polynomial can create instability in the model. In Fig. 5 four KRR fittings show values interpolated between the experimental data for different sizes. The interpolation points are in the middle of the measured data points. What can be clearly seen in the higher degree polynomials is that the interpolated data extends out of the value range of the plot, even though the training and test data are still well within the model prediction. This means that even with a high R-squared value, the prediction of new sizes can be very inaccurate. For this reason, we kept the degree to a maximum of 4 in the experiments in "Performance comparison of ML and analytical models" and "Performance analysis using extrapolation" sections.

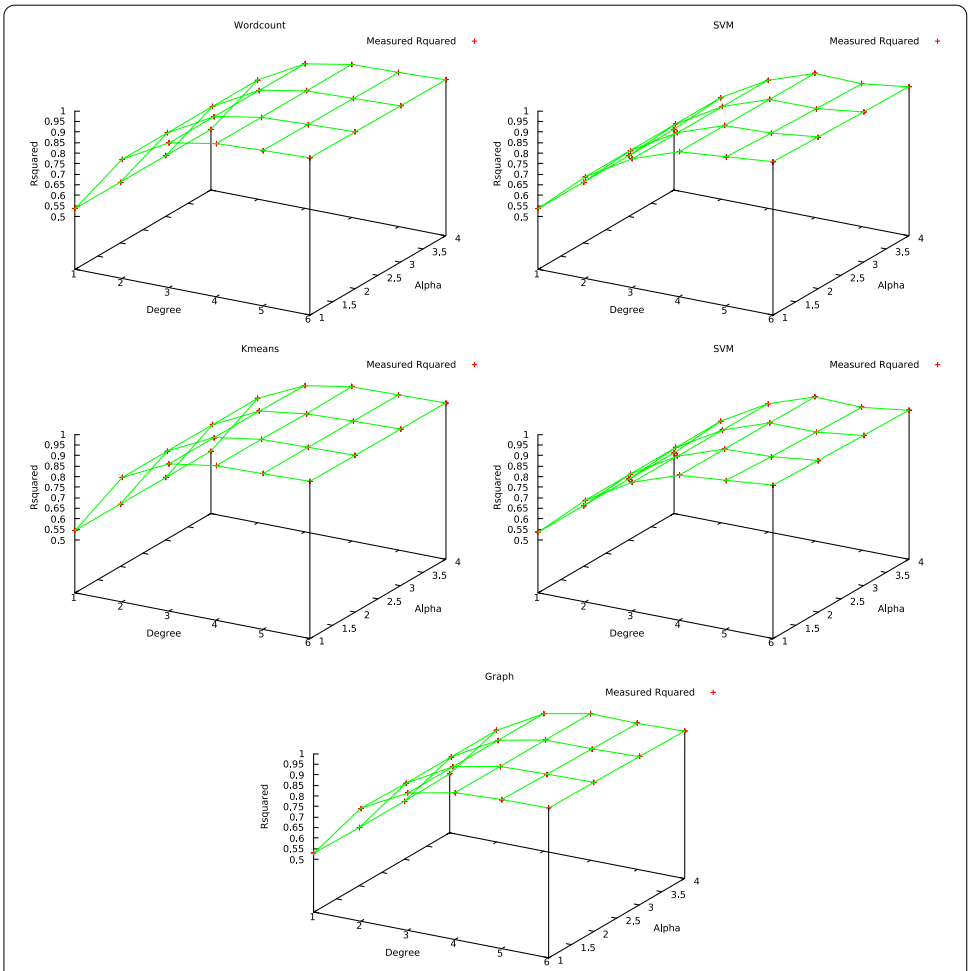


Fig. 4 Performance comparison of KRR algorithm across different degrees and alphas on the performance of selected HiBench workloads

Gradient boost models

GBR algorithm is a popular technique used for building prediction models. GBR uses the forward stage-wise fashion technique with the optimization of arbitrary differentiable loss functions. In this experiment, we used GBR from scikit-learn [48] with the default parameters. We obtained the best results with default random state 1 for all five workloads. However, we further investigated the model by increasing the random state, but the results were unsatisfactory and out of scope for inclusion in this study. In Table 7, the statistical results are shown based on R^2, standard deviation, and RRSE scores.

Performance comparison of ML and analytical models

This section illustrates the proposed model accuracy in terms of R^2, standard deviation and the RRSE values. It shows the performance comparison results between well-known parallelisation models (Amdahl's and Gustafson, ERNEST) and ML algorithms such as KRR and GBR where KRR algorithm parameters were optimised. The KRR optimised parameters assist the model to maximise the performance accuracy of the model. The k-fold cross-validation technique was applied with all the models to achieve highest prediction accuracy. We obtained all the table results and figures using cross-validation with $k = 3$. The individual workload's best results obtained by the proposed models against the ML models are described in the following section.

Wordcount

The wordcount workload comparative statistical results between ML and analytical models with cross validation, are presented in Table 8. From this analysis we can see that except ERNEST, all analytical models' accuracy is 0.995, which is better than the KRR algorithm of 0.974. The GBR algorithm shows better performance as compared to others, where the accuracy is 0.998. The best analytical results and GBR results are presented in Fig. 6. The RRSE results in Table 9 show very low accuracy of GBR algorithm presenting the best fit in the results.

SVM

The comparison between ML and analytical models with cross validation for SVM workloads runtime prediction results are shown in Table 8. In this workload, both ML algorithms show significantly better results than the analytical models where the GBR algorithm completely outperforms the KRR algorithm. It may be noted that the GBR accuracy and RRSE is 0.995 and 0.064 respectively with corresponding standard deviations of 0.001 and 0.010. The comparative best results are plotted in Fig. 7.

Pagerank

The Pagerank performance evaluated in terms of ML algorithms and analytical models with cross-validation approach in Tables 8 and 9. Our results revealed that the model (Eq. 4) either outperforms or equal to all analytical models and KRR algorithms, but the

GBR algorithm achieves the best results among models. In Fig. 8, the best results are plotted from GBR and Eq. (4).

Kmeans

The comparative performance measurement results of ML algorithm and analytical model for kmeans workload are presented in Table 8. The GBR algorithm is the most effective model that shows the best accuracy and produces low RRSE among all the models. It can be noted that analytical models outperform KRR algorithms where the accuracy is 0.981, but analytical models are better with a higher margin of accuracy of 0.992. Among the analytical models, their performance is either equal or slightly different. For example, the Gustafson accuracy is equal among analytical models, but the RRSE is slightly better, as shown in Table 9. The best-obtained results among models are shown in Fig. 9.

Graph

The ML algorithms (KRR and GBR) show excellent results using Graph workload. From the statistical results shown in the Tables 8 and 9, the GBR algorithm records the best results among analytical models and outperforms KRR. Equation (4) indicates a significant performance improvement among the analytical models where other equations previously proposed by us clearly defeat ERNEST, Amdahl and Gustafson models. The best results from the ML model and an analytical model is shown in Fig. 10.

In summary, the above results demonstrate model performances for the selected workloads. The GBR algorithm achieves an excellent performance in comparison to all models. On the other side, Eqs. (4) and (5) show excellent results among analytical models and both equations are better than KRR for WordCount, SVM, PageRank and Kmeans workload. For the graph workload, both ML algorithms demonstrated the best results. The above analysis shows the effectiveness of the analytical models over ML approaches.

Performance analysis using extrapolation

Data obtained from the results in "Performance comparison of ML and analytical models" section is used for the performance analysis using extrapolation in this section. However, rather than carrying out cross validation on the entire dataset, we reserved part of the data set to test how well each model can deal with extrapolation. This is a very important aspect of the prediction models, as extrapolation would allow practitioners to make accurate predictions with a very small number of experiments when using a specific cluster and workload. Our hypothesis was that despite the better results for the models using ML presented in "Performance comparison of ML and analytical models" section, equations that can represent the cluster behaviours could be more accurate for extrapolation. In other words, ML is an effective approach for predictions that fall within the range of values captured in the existing data, but can yield poor results if not enough data is available to describe runtimes beyond a given range. In these scenarios, ML methods may not be able to predict the actual pattern of communication that drives the runtime.

Due to limited data points, we employed the linear extrapolation approach to estimate the data values that are close to the existing data. Generally, it has been proven

that nonlinear model accuracy is higher than the linear models because it is more likely to overfit the training data set, which shows the poor performance of the models [58]. In contrast, the linear models fit the data more accurately; thus, better fitting can be achieved from the unseen data. We observe from the presented results in Table 10 that the performance of the linear workloads is better than the quadratic workloads. Two extrapolation scenarios were considered: extrapolation by size, and extrapolation by number of executors.

Extrapolation by size

Wordcount

Figure 11 presents the Wordcount workload results using the extrapolation approach. In this case, the extrapolation by size yielded the best results among analytical and ML models. The model accuracy is measured by finding the accuracy of the models. The comparative results are presented in Table 10, and it can be noted that Eq. (1) shows the best fit of the data compared to other models. By using the ML algorithm, the data fitting accuracy decreases. Our analysis concludes that the performance of the proposed equations are better than ML when the data are extrapolated.

SVM

For SVM workloads, the extrapolation by size found the best results using Eq. (5), where the model accuracy is 0.981, but the KRR and GBR accuracy is lower than that of all the analytical equations. We used 3D charts to illustrate the relationship between size, number of executors and runtime. We chose the 3D charts because it is easier to show their relationship more accurately. We select the best analytical and ML results that are plotted in Fig. 12. In this case, we found that KRR shows better performance as compared to the GBR algorithm.

Pagerank

In the case of Pagerank workload, analytical models completely outperform KRR and GBR. Equations (5) and (6) have low accuracy and prove the models' effectiveness over ML when the data is extrapolated by size. The maximum accuracy of the analytical models is 0.996, whereas ML is 0.875. The influence of the extrapolation is less effective for KRR and GBR. For the comparison, Fig. 13 shows the fitting for Eq. (5) and for GBR.

Kmeans

In the case of the Kmeans workload, when the data is extrapolated by size, Eq. (6) and ERNEST show an equally excellent performance, and other analytical models demonstrate better performance over ML models. As shown in Table 10, Eq. (6) achieved notable accuracy at 0.998, but KRR and GBR accuracy proved to be relatively poor at 0.836 and 0.875 respectively. We can conclude from these results that Eq. (6) is a very effective model on the unseen data points. For comparison, Fig. 14 shows the fitting for Eq. (6) and for GBR.

Graph

In the case of Graph workload when data is extrapolated by size, Eq. (4) is either at par or higher among the analytical models, and both ML algorithms appear to be less effective. We found that equation 4 and KRR performance is better than other models where the Eq. (4) and KRR accuracy is 0.940 and 0.904, respectively. For comparison, Fig. 15 shows the fitting for Eq. (4) and for KRR.

Extrapolation by number of executors

Wordcount

The results of extrapolation by a number of executors based on Wordcount workload are presented in Table 11. As expected, ML performance is considerably lower than that of the analytical model. The performances of the Eqs. (4), (6), and (1) (Amdahl) are equal or the same as the other models. The best accuracy achieved by both the equations is 0.997, whereas the accuracies for ML algorithms KRR and GBR are 0.786 and 0.535, respectively. We compared all the results and plotted the best results in Fig. 16 and (6). These results demonstrate that the extrapolation by a number of executors yielded the best results using equations while the results for ML algorithms were less accurate.

SVM

In the case of SVM, the extrapolation by the number of executors yielded the best results using Eq. (4). The KRR and GBR regression performance is poor as expected. The effectiveness of Eq. (4) is excellent, where the accuracy is 0.893. The best comparative performance in Fig. 17 shows the fitting for Eq. (4) and for KRR. These results demonstrate that the extrapolation by a number of executors yielded the best results using equations while the results were less accurate for the ML algorithms..

Pagerank

Equations (4), (5), (6) and Amdahl (1) show remarkable performance improvement for the Pagerank workload where the number of executors extrapolates the data. All four equations obtained the accuracy of 0.994 while KRR and GBR regression showed relatively demonstrated a poor accuracy which was 0.619 and 0.408 respectively. We examined (5) and compared the models' performance and plotted the best performance in Fig. 18.

Kmeans

In the case of the Kmeans workload, Eqs. (5) and (6) produce better fit than all the models and show a significant performance improvement when the data is extrapolated considering the number of executors. As shown in Table 11 both equations performed

Table 6 Kernel ridge regression algorithm statistical parameters using set of different workloads

Workload	Degree	Alpha	R-squared	RRSE
WordCount	70	1	0.999 ± 0.000	0.097 ± 0.000
	3 (default)		0.935 ± 0.002	2.039 ± 0.046
SVM	70	1	0.999 ± 0.000	0.083 ± 0.003
	3 (default)		0.860 ± 0.001	2.434 ± 0.010
Pagerank	70	1	0.999 ± 0.000	6.507 ± 0.025
	3 (default)		0.932 ± 0.001	53.885 ± 0.376
Kmeans	30	1	0.999 ± 0.000	7.868 ± 0.290
	3 (default)		0.947 ± 0.008	119.551 ± 8.875
Graph (NWeight)	30	1	0.996 ± 0.000	0.239 ± 0.010
	3 (default)		0.900 ± 0.015	1.242 ± 0.0411

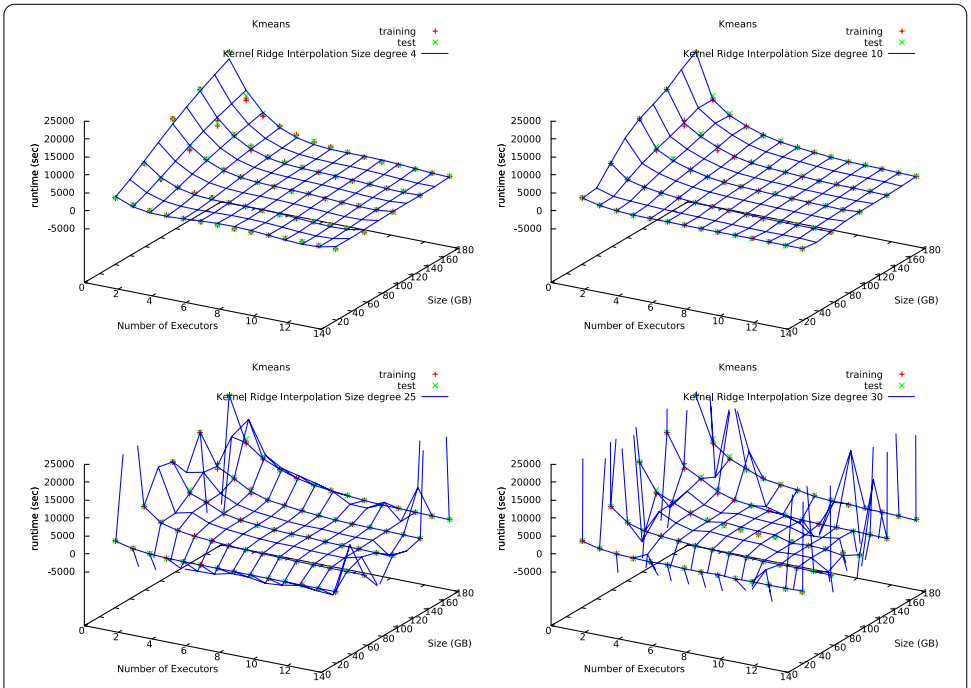


Fig. 5 Kernel ridge regression models for Kmeans with increasing degrees. Despite better R-squared, interpolation is very inaccurate for higher degrees

Table 7 GBR algorithm statistical parameters using set of different workloads

Workload	Random state	R-squared	RRSE
WordCount	1	0.998 ± 0.000	0.348 ± 0.018
SVM	1	0.994 ± 0.001	0.465 ± 0.075
Pagerank	1	0.999 ± 0.000	6.132 ± 0.550
Kmeans	1	0.997 ± 0.000	28.086 ± 3.760
Graph (NWeight)	1	0.986 ± 0.003	0.452 ± 0.015

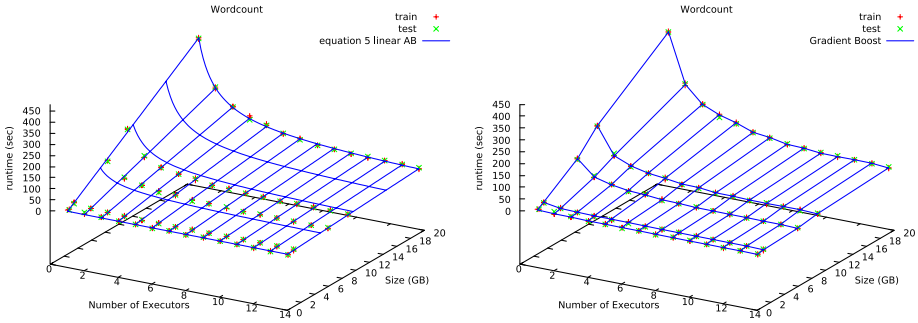


Fig. 6 Comparison of ML (GBR) and analytical models Eq. (5) showing best R-Squared for Wordcount

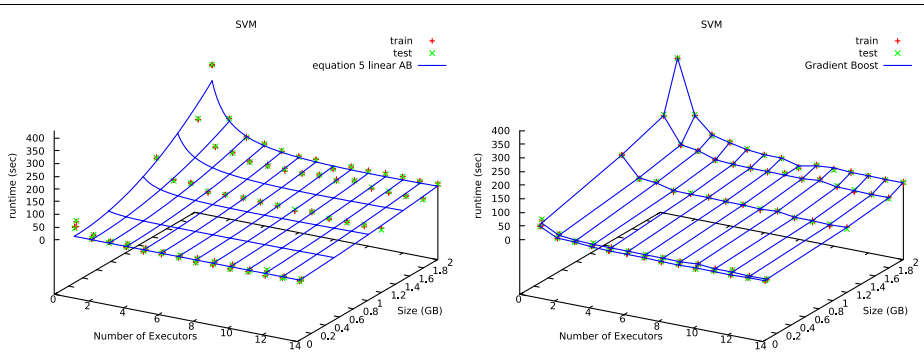


Fig. 7 Comparison of ML (GBR) and analytical models Eq. (5) showing best R-Squared for SVM

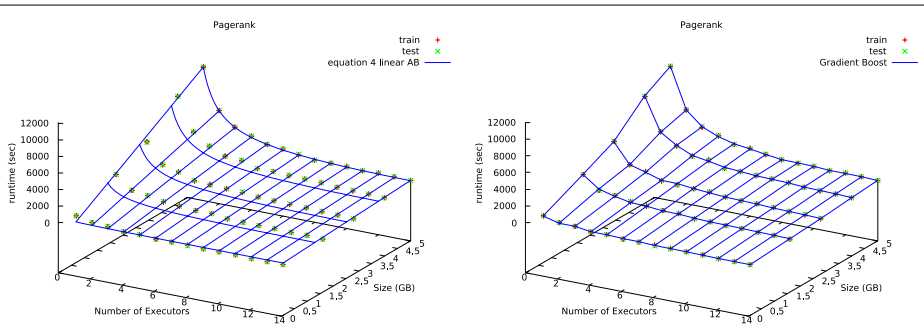


Fig. 8 Comparison of ML (GBR) and analytical models Eq. (4) showing best R-Squared for Pagerank

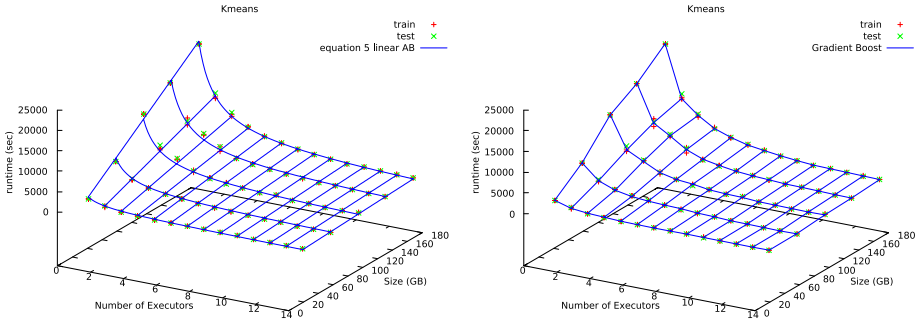


Fig. 9 Comparison of ML (GBR) and analytical models Eq. (5) showing best R-Squared for Kmeans

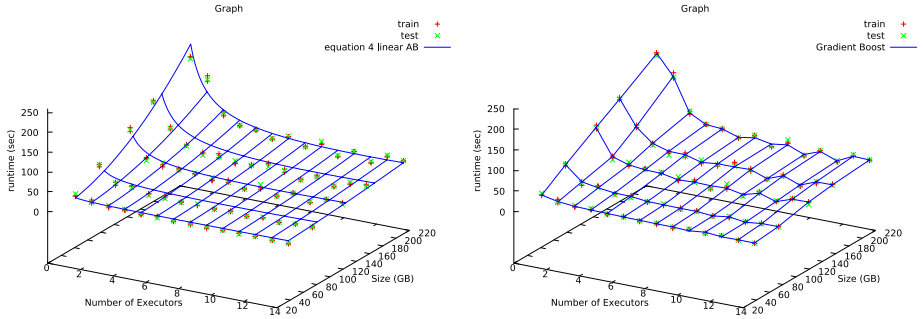


Fig. 10 Comparison of ML (GBR) and analytical models Eq. (4) showing best R-Squared for Graph

Table 8 R-squared values for a different set of workloads and models

Workload $f(S)$	Wordcount linear	SVM quadratic	Pagerank linear	Kmeans linear	Graph (NWeight) quadratic
Amdhal equation (1)	0.995 ± 0.000	0.908 ± 0.005	0.990 ± 0.000	0.992 ± 0.002	0.901 ± 0.012
Gustafson equation (2)	0.995 ± 0.000	0.888 ± 0.002	0.988 ± 0.000	0.992 ± 0.000	0.898 ± 0.013
ERNEST equation (3)	0.994 ± 0.000	0.848 ± 0.001	0.987 ± 0.000	0.992 ± 0.002	0.916 ± 0.003
2D plate equation (4)	0.995 ± 0.000	0.916 ± 0.005	0.990 ± 0.000	0.992 ± 0.002	0.918 ± 0.009
Connected graph equation (5)	0.995 ± 0.001	0.918 ± 0.005	0.989 ± 0.000	0.992 ± 0.002	0.911 ± 0.005
Con. graph $c = 1$ equation (6)	0.995 ± 0.001	0.914 ± 0.005	0.989 ± 0.000	0.992 ± 0.002	0.911 ± 0.005
Kernel ridge regression	0.974 ± 0.002	0.934 ± 0.001	0.977 ± 0.000	0.981 ± 0.005	0.945 ± 0.009
Gradient boost regression	$\mathbf{0.998 \pm 0.000}$	$\mathbf{0.995 \pm 0.001}$	$\mathbf{0.999 \pm 0.000}$	$\mathbf{0.997 \pm 0.001}$	$\mathbf{0.986 \pm 0.003}$

The bold data in each column indicates the largest R-squared value in the corresponding column

Table 9 Relative mean standard error (RRSE) values for a different set of workloads and models

Workload $f(S)$	Wordcount linear	SVM quadratic	Pagerank linear	Kmeans linear	Graph (NWeight) quadratic
Amdhal equation (1)	0.085 ± 0.005	0.282 ± 0.009	0.113 ± 0.000	0.140 ± 0.019	0.252 ± 0.009
Gustafson equation (2)	0.091 ± 0.004	0.311 ± 0.003	0.120 ± 0.000	0.134 ± 0.009	0.254 ± 0.008
ERNEST equation (3)	0.100 ± 0.006	0.366 ± 0.001	0.127 ± 0.000	0.142 ± 0.017	0.231 ± 0.009
2D plate equation (4)	0.086 ± 0.005	0.272 ± 0.009	0.113 ± 0.000	0.141 ± 0.019	0.226 ± 0.007
Connected graph equation (5)	0.091 ± 0.009	0.268 ± 0.009	0.116 ± 0.000	0.141 ± 0.018	0.244 ± 0.008
Con. graph $c = 1$ equation (6)	0.091 ± 0.009	0.273 ± 0.008	0.118 ± 0.000	0.140 ± 0.018	0.243 ± 0.008
Kernel ridge regression	0.203 ± 0.009	0.234 ± 0.002	0.173 ± 0.001	0.207 ± 0.024	0.178 ± 0.003
Gradient boost regression	$\mathbf{0.058 \pm 0.003}$	$\mathbf{0.064 \pm 0.010}$	$\mathbf{0.033 \pm 0.002}$	$\mathbf{0.062 \pm 0.038}$	$\mathbf{0.087 \pm 0.004}$

The bold data in each column indicates the smallest RRSE value in the corresponding column

Table 10 R-squared values for extrapolation on size

Workload *f*(*S*)	Wordcount linear	SVM quadratic	Pagerank linear	Kmeans linear	Graph (NWeight) quadratic
Amdhal equation (1)	**0.998 ± 0.000**	0.965 ± 0.001	0.994 ± 0.000	0.997 ± 0.001	0.937 ± 0.006
Gustafson equation (2)	0.996 ± 0.001	0.949 ± 0.004	0.994 ± 0.000	0.996 ± 0.001	0.913 ± 0.008
ERNEST equation (3)	0.996 ± 0.001	0.958 ± 0.002	0.990 ± 0.000	0.998 ± 0.001	0.921 ± 0.008
2D plate equation (4)	0.997 ± 0.001	0.951 ± 0.003	0.993 ± 0.000	0.997 ± 0.001	**0.940 ± 0.005**
Connected graph equation (5)	0.257 ± 0.061	**0.981 ± 0.001**	**0.996 ± 0.000**	0.996 ± 0.001	0.940 ± 0.006
Con. graph $c = 1$ equation (6)	0.997 ± 0.001	0.978 ± 0.001	**0.996 ± 0.000**	**0.998 ± 0.001**	0.940 ± 0.006
Kernel ridge regression	0.836 ± 0.011	0.745 ± 0.004	0.836 ± 0.011	0.836 ± 0.011	0.904 ± 0.043
Gradient boost regression	0.875 ± 0.005	0.690 ± 0.003	0.875 ± 0.005	0.875 ± 0.005	0.775 ± 0.009

The bold data in each column indicates the largest R-squared values in the corresponding column

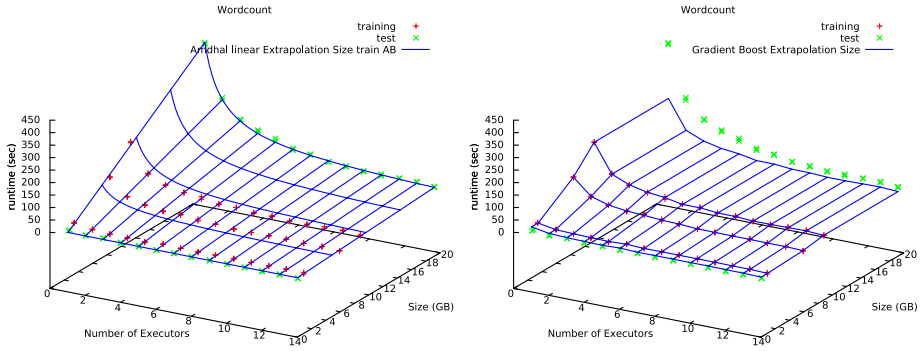


Fig. 11 Extrapolation relationship showing Wordcount workload by size using Eq. (1) and GBR

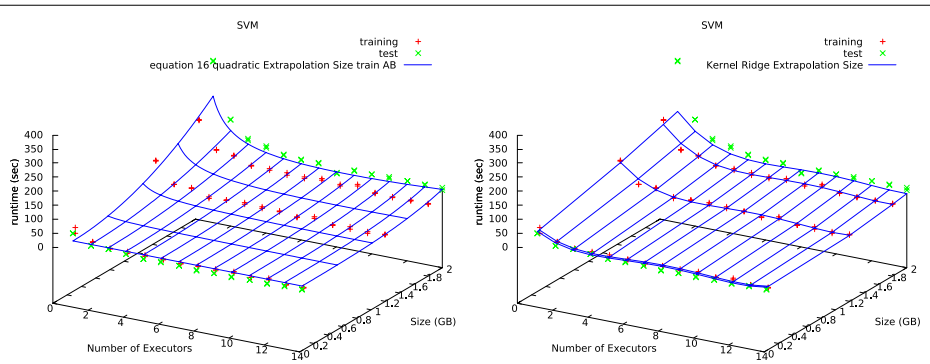


Fig. 12 Extrapolation relationship showing SVM workload by Size using Eq. (5) and KRR

well where the equation achieved the accuracy of 0.995. The KRR and GBR achieved poor accuracy of 0.917 and 0.510 respectively. These results indicate that the ML performance is less accurate when training on limited data and when having to extrapolate

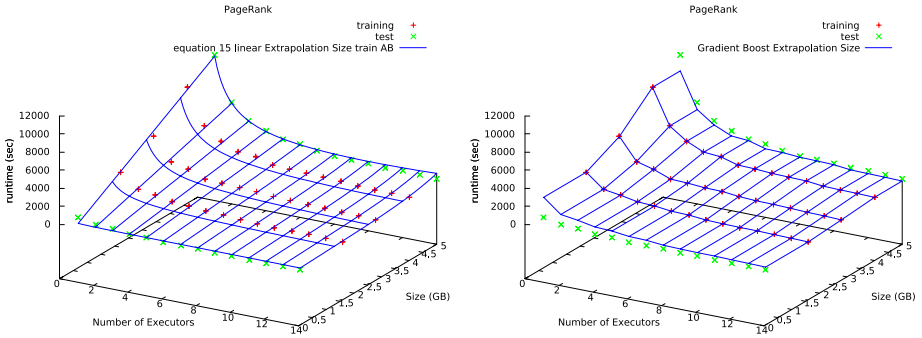


Fig. 13 Extrapolation relationship showing Pagerank workload by Size using Eq. (5) and GBR

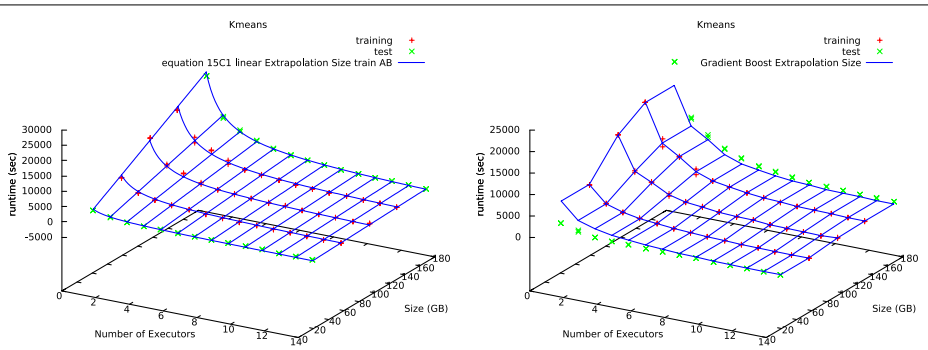


Fig. 14 Extrapolation relationship showing Kmeans workload by Size using Eq. (6) and GBR

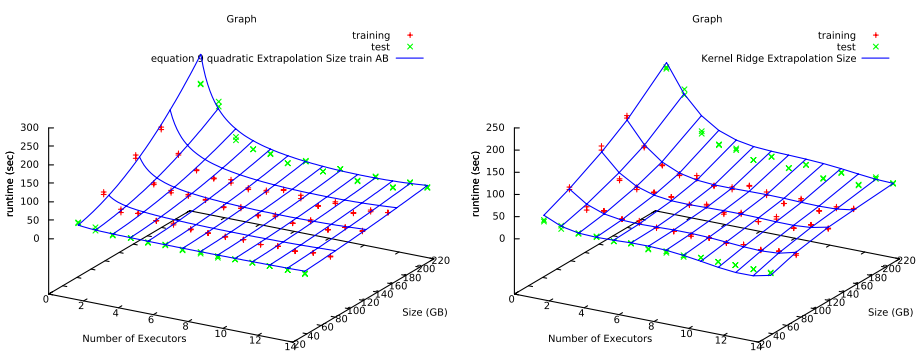


Fig. 15 Extrapolation relationship showing Graph workload by Size using Eq. (4) and Kernel Ridge

results beyond the values seen in the training data. We examined and compared the performance of the models and plotted the best performance in Fig. 19 and (6).

Graph

In the case of the Graph workload, the data were extrapolated by size. Equation (4) shows the best performance among all the models. In this case, the KRR and GBR performances were inferior. As shown in Table 11, the Eq. (4) accuracy is 0.945, while

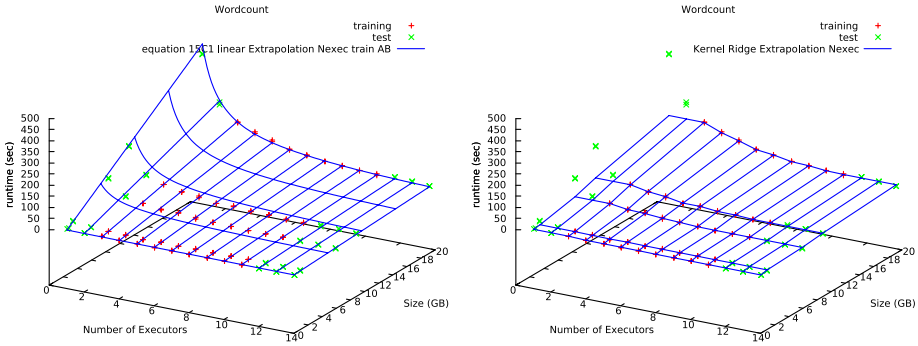


Fig. 16 Extrapolation for Wordcount by Nexec using Eq. (6) and KRR

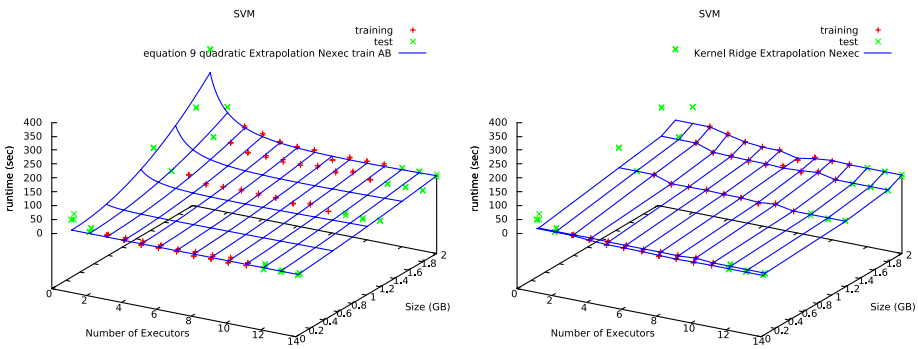


Fig. 17 Extrapolation for SVM by Nexec using Eq. (4) and KRR

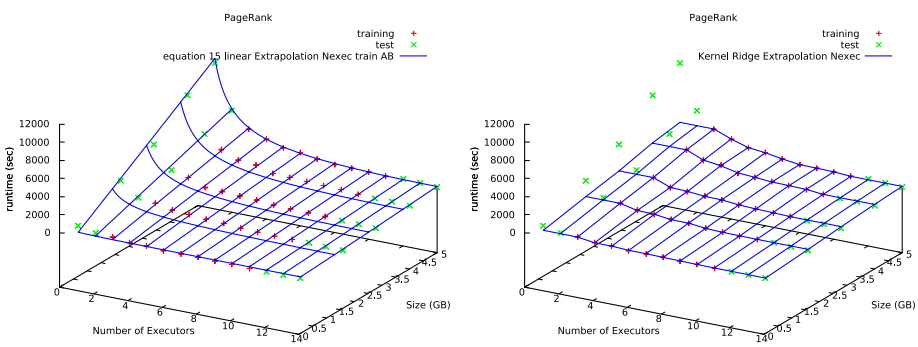


Fig. 18 Extrapolation for Pagerank by Nexec using Eq. (5) and KRR

the KRR and GBR accuracies are 0.150 and 0.371 respectively. These results indicate that the selected ML algorithms are not suitable when the extrapolation is required. Fig. 20 shows this performance comparison results for Eq. (4) and GBR.

Discussion

To evaluate the performance of the prediction models, this paper used the analytical and ML models and depicted the comparative analysis between them. To predict Spark runtime performance, several experiments have been performed to evaluate the

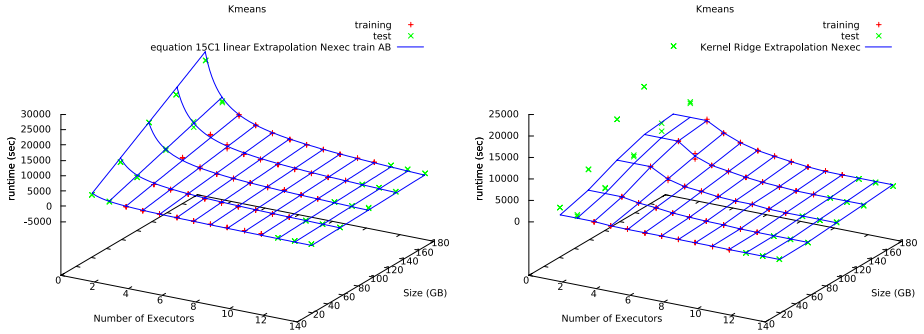


Fig. 19 Extrapolation for Kmeans by Nexec using Eq. (6) and KRR

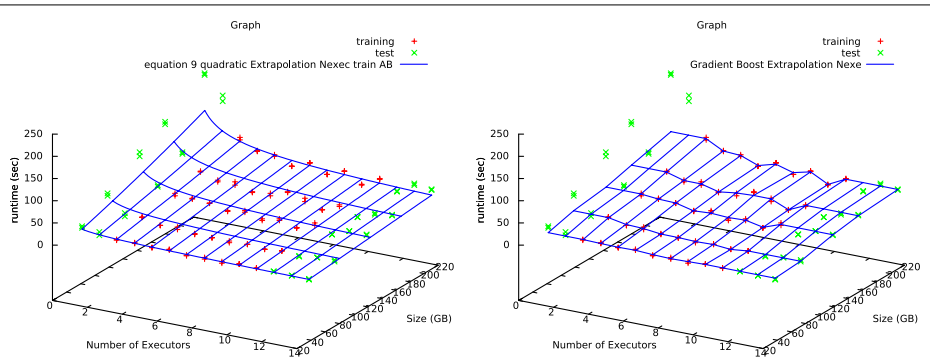


Fig. 20 Extrapolation for Graph by Nexec using Eq. (4) and GBR

Table 11 R-squared values for Extrapolation on Number of Executors

Workload *f*(*S*)	Wordcount linear	SVM quadratic	Pagerank linear	Kmeans linear	Graph (NWeight) quadratic
Amdhal equation (1)	**0.997 ± 0.000**	0.878 ± 0.002	**0.994 ± 0.000**	0.994 ± 0.001	0.932 ± 0.001
Gustafson equation (2)	0.995 ± 0.000	0.728 ± 0.000	0.988 ± 0.000	0.921 ± 0.001	0.922 ± 0.000
ERNEST equation (3)	0.996 ± 0.000	0.822 ± 0.002	0.991 ± 0.000	0.992 ± 0.001	0.930 ± 0.007
2D plate equation (4)	**0.997 ± 0.000**	**0.893 ± 0.004**	**0.994 ± 0.000**	0.992 ± 0.002	**0.945 ± 0.004**
Connected graph equation (5)	0.996 ± 0.000	0.853 ± 0.072	**0.994 ± 0.000**	**0.995 ± 0.001**	0.917 ± 0.002
Con. graph $c = 1$ equation (6)	**0.997 ± 0.000**	0.850 ± 0.072	**0.994 ± 0.000**	**0.995 ± 0.001**	0.916 ± 0.002
Kernel ridge regression	0.786 ± 0.012	0.619 ± 0.012	0.917 ± 0.014	0.917 ± 0.014	0.150 ± 0.060
Gradient boost regression	0.535 ± 0.001	0.408 ± 0.007	0.510 ± 0.011	0.510 ± 0.011	0.371 ± 0.015

The bold data in each column indicates the largest R-squared values in the corresponding column

performance. The comprehensive comparative study of the results presented in Tables 6, 7, 8, 9, 10, and 11 inspires us to the following three steps analysis.

Firstly, we present the KRR regression parameter relationship between alpha and degree. This study found, for most of the workloads, the best R-squared can be

achieved by selecting the small degree with alpha. Our analysis found higher degree can produce best R-squared but the data overfitting can be a major limitation. For the GBR, we kept all the parameters default. However, we have examined different random states, but there are no effects on the accuracy improvement. The detailed results are presented in "Performance evaluations and analysis" and "Performance analysis using extrapolation" sections.

Secondly, interpolation experiments were carried out. The data split for the cross-validation used data points for all the available sizes and number of executors for both test and training sets. The presented results showed that analytical models are better than KRR regression and produce similar accuracy as ERNEST, Amdahl or Gustafson. However, the GBR model was the most accurate when compared to all other models.

Finally, we used the extrapolation method with the cross-validation technique, and the analysis was carried out using size and executors. We noticed the performance of ML models are poor in both cases, but the analytical models are more accurate and effective. The presented results in Tables 10 and 11 showed that the linear workloads are more accurate among the ML models than the quadratic workloads; in fact, the accuracy is significantly poorer. The KRR, GBR, 2D-plate (Eq. 4) or fully-connected (Eq. 5) models average accuracies are 0.466, 0.677 and 0.950. These results indicate that both 2D-plate model and the fully-connected models are more effective and accurate when using extrapolation of data, either over size or number of executors.

Conclusion

This work aimed to compare five analytical models against ML regression algorithms for Spark performance prediction. We investigated two ML algorithms, namely KRR and GBR algorithms, and five analytical models, namely, 2D-plate, fully-connected, ERNEST, Amdahl, Gustafson models. The key challenges were how to use limited data points for generating models that fit the data accurately and generalise well when extrapolating.

To address these challenges, we used interpolation and extrapolation methods with k-fold cross-validation technique for both ML and analytical models. Using the interpolation method, 2D-plate and fully-connected models outperformed the KRR algorithm, ERNEST, Amdahl and Gustafson, but the GBR showed a better fitting accuracy (R^2) than all other models.

Due to the limited available input data when using the extrapolation method, ML algorithms proved not to be as accurate as 2D-plate, and fully-connected models. Our experimental findings confirm that both 2D-plate and fully-connected models reduce the percentage error significantly and can accurately fit the data for prediction purposes. For this reason, 2D-plate and fully-connected models stand out as very effective approaches in the presence of limited input data for predicting Spark performance as well as parallel system performance.

For future work, we plan to study different ML algorithms for comparative analysis as well as perform more robust experimentations with different HiBench workloads in order to yield further conclusive findings.

Abbreviations
SVM: Support vector machines; SVR: Support vector regression; GBR: Gradient boost regression; GBM: Gradient boost machine; DT: Decision tree; LR: Logistic regression; KRR: Kernel ridge regression; MF: Matrix factorization; NB: Naıve Bayes; NN: Neural networks; MLR: Multi linear regression; TSt: Two-stage tree; LS-SVM: Least square support vector machine; API: Application programming interface; SQL: Structured query language; HDFS: Hadoop distributed file system; RDD: Resilient distributed datasets; ML: Machine learning; MLlib: Machine learning library; AMPLab: Algorithms, machines and people lab; CPU: Central processing unit; I/O: Input/output; UC: University of California; AMP: Algorithms, machines and people; DAG: Directed acyclic graph; YARN: Yet another resource negotiator; NEXEC: Number of executor; SQRT: Square root; RRSE: Root relative squared error; 2D: Two dimensional; GHz: Gigahertz; TB: Terabyte; RAM: Random access memory; DDR: Double data rate; GB: Gigabyte; MB: Megabyte; WC: Wordcount; Exec: Executor; Amazon EC2: Amazon elastic computing.

Acknowledgements
This work was supported in part by the Massey University Doctoral Scholarship.

Author contributions
NA and ALCB were the main contributors of this work. NA has done an initial literature review and data collection, run experiments, prepare results, and drafted the manuscript. Both NA and ALCB fitted the data into the models and analysed the results. ALCB and TS deployed and configured the physical Hadoop cluster. TS and MAR helped to improve the final paper. All authors read and approved the final manuscript.

Funding
Not applicable. This research received no fund from any agency.

Availability of data and materials
Data are contained within the article. However, the correspondence author can be contacted for more details.

Declarations

Ethics approval and consent to participate
Not applicable.

Consent for publication
Not applicable.

Competing interests
The authors declare that they have no competing interests.

Author details
[1]School of Mathematical and Computational Sciences, Massey University, Auckland 0745, New Zealand. [2]Department of Mechanical and Electrical Engineering, Massey University, Auckland 0745, New Zealand.

Received: 22 December 2021 Accepted: 2 May 2022
Published online: 19 May 2022

References

1. Ghani NA, Hamid S, Hashem IAT, Ahmed E. Social media big data analytics: a survey. Comput Hum Behav. 2019;101:417–28.
2. Fang R, Pouyanfar S, Yang Y, Chen S-C, Iyengar S. Computational health informatics in the big data age: a survey. ACM Comput Surv. 2016;49(1):1–36.
3. Hirschberg J, Manning CD. Advances in natural language processing. Science. 2015;349(6245):261–6.
4. Maros A, Murai F, da Silva APC, Almeida JM, Lattuada M, Gianniti E, Hosseini M, Ardagna D. Machine learning for performance prediction of spark cloud applications. In: 2019 IEEE 12th international conference on cloud computing (CLOUD). New York: IEEE; 2019. p. 99–106.
5. Salloum S, Dautov R, Chen X, Peng PX, Huang JZ. Big data analytics on apache spark. Int J Data Sci Anal. 2016;1(3):145–64.
6. Awan MJ, Khan RA, Nobanee H, Yasin A, Anwar SM, Naseem U, Singh VP. A recommendation engine for predicting movie ratings using a big data approach. Electronics. 2021;10(10):1215.
7. Meng X, Bradley J, Yavuz B, Sparks E, Venkataraman S, Liu D, Freeman J, Tsai D, Amde M, Owen S, et al. Mllib: machine learning in apache spark. J Mach Learn Res. 2016;17(1):1235–41.
8. Petridis P, Gounaris A, Torres J. Spark parameter tuning via trial-and-error. In: INNS conference on big data. Berlin: Springer; 2016. p. 226–37.
9. Ahmed N, Barczak AL, Susnjak T, Rashid MA. A comprehensive performance analysis of Apache Hadoop and Apache Spark for large scale data sets using HiBench. J Big Data. 2020;7(1):1–18.
10. Herodotou H, Babu S. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. Proc VLDB Endow. 2011;4(11):1111–22.
11. Mustafa S, Elghandour I, Ismail MA. A machine learning approach for predicting execution time of spark jobs. Alex Eng J. 2018;57(4):3767–78.
12. Cheng G, Ying S, Wang B, Li Y. Efficient performance prediction for apache spark. J Parallel Distrib Comput. 2021;149:40–51.

13. Cheng G, Ying S, Wang B. Tuning configuration of apache spark on public clouds by combining multi-objective optimization and performance prediction model. J Syst Softw. 2021;180:111028.
14. Luo N, Yu Z, Bei Z, Xu C, Jiang C, Lin L. Performance modeling for spark using svm. In: 2016 7th international conference on cloud computing and big data (CCBD). New York: IEEE; 2016. p. 127–31.
15. Ahmed N, Barczak AL, Rashid MA, Susnjak T. An enhanced parallelisation model for performance prediction of apache spark on a multinode hadoop cluster. Big Data Cogn Comput. 2021;5(4):65.
16. Ahmed N, Barczak ALC, Susnjak T, Rashid MA. A parallelization model for performance characterization of spark big data jobs on Hadoop clusters. J Big Data. 2021;8(107):1–28. https://doi.org/10.1186/s40537-021-00499-7.
17. Dogan A, Birant D. Machine learning and data mining in manufacturing. Expert Syst Appl. 2021;166:114060.
18. Mavridis I, Karatza H. Performance evaluation of cloud-based log file analysis with apache hadoop and apache spark. J Syst Softw. 2017;125:133–51.
19. Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauly M, Franklin MJ, Shenker S, Stoica I. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: 9th {USENIX} symposium on networked systems design and implementation ({NSDI} 12), 2012. p. 15–28.
20. Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I, et al. Spark: cluster computing with working sets. HotCloud. 2010;10(10–10):95.
21. Shahul A. Spark architecture: Apache Spark tutorial. 2021. https://www.learntospark.com/2020/02/spark-architecture.html.
22. Chen J, Li K, Tang Z, Bilal K, Yu S, Weng C, Li K. A parallel random forest algorithm for big data in a spark cloud computing environment. IEEE Trans Parallel Distrib Syst. 2016;28(4):919–33.
23. Vavilapalli VK, Murthy AC, Douglas C, Agarwal S, Konar M, Evans R, Graves T, Lowe J, Shah H, Seth S et al. Apache hadoop yarn: yet another resource negotiator. In: Proceedings of the 4th annual symposium on cloud computing, 2013. p. 1–16.
24. Ousterhout K, Rasti R, Ratnasamy S, Shenker S, Chun B-G. Making sense of performance in data analytics frameworks. In: 12th {USENIX} symposium on networked systems design and implementation ({NSDI} 15), 2015. p. 293–307.
25. Al-Sayeh H, Hagedorn S, Sattler K-U. A gray-box modeling methodology for runtime prediction of apache spark jobs. Distrib Parallel Databases. 2020;38(4):819–39.
26. Chao Z, Shi S, Gao H, Luo J, Wang H. A gray-box performance model for apache spark. Future Gener Comput Syst. 2018;89:58–67.
27. Lattuada M, Gianniti E, Hosseini M, Ardagna D, Alexandre M, Fabricio M, COUTO da SILVA AP, Jussara MA. Gray-box models for performance assessment of spark applications. In: 9th international conference on cloud computing and services science, SciTePress; 2019. p. 609–18.
28. Prats DB, Portella FA, Costa CH, Berral JL. You only run once: spark auto-tuning from a single run. IEEE Trans Netw Serv Manag. 2020;17(4):2039–51.
29. Jia Z, Xue C, Chen G, Zhan J, Zhang L, Lin Y, Hofstee P. Auto-tuning spark big data workloads on power8: Prediction-based dynamic smt threading. In: 2016 international conference on parallel architecture and compilation techniques (pact), New York: IEEE; 2016. p. 387–400.
30. Nikitopoulou D, Masouros D, Xydis S, Soudris D. Performance analysis and auto-tuning for spark in-memory analytics. In: 2021 design, automation & test in Europe conference & exhibition (DATE). New York: IEEE; 2021. p. 76–81.
31. de Oliveira D, Porto F, Boeres C, de Oliveira D. Towards optimizing the execution of spark scientific workflows using machine learning-based parameter tuning. Concurr Comput Pract Exp. 2021;33(5):5972.
32. Boden C, Spina A, Rabl T, Markl V. Benchmarking data flow systems for scalable machine learning. In: Proceedings of the 4th ACM SIGMOD workshop on algorithms and systems for mapreduce and beyond. 2017. p. 1–10.
33. Boden C, Rabl T, Schelter S, Markl V. Benchmarking distributed data processing systems for machine learning workloads. In: Technology conference on performance evaluation and benchmarking. Berlin: Springer; 2018. p. 42–57.
34. Mostafaeipour A, Jahangard Rafsanjani A, Ahmadi M, Arockia Dhanraj J. Investigating the performance of Hadoop and Spark platforms on machine learning algorithms. J Supercomput. 2021;77(2):1273–300.
35. Assefi M, Behravesh E, Liu G, Tafti AP. Big data machine learning using apache spark MLlib. In: 2017 IEEE international conference on big data (big Data). New York: IEEE; 2017. p. 3492–8.
36. Javaid MU, Kanoun AA, Demesmaeker F, Ghrab A, Skhiri S. A performance prediction model for spark applications. In: International conference on big data. Berlin: Springer; 2020. p. 13–22.
37. Singhal R, Phalak C, Singh P. Spark job performance analysis and prediction tool. In: Companion of the 2018 ACM/SPEC international conference on performance engineering. 2018. p. 49–50.
38. Yigitbasi N, Willke TL, Liao G, Epema D. Towards machine learning-based auto-tuning of mapreduce. In: 2013 IEEE 21st international symposium on modelling, analysis and simulation of computer and telecommunication systems. New York: IEEE. 2013. p. 11–20.
39. Cristianini N, Shawe-Taylor J, et al. An introduction to support vector machines and other kernel-based learning methods. Cambridge: Cambridge University Press; 2000.
40. Friedman JH. Stochastic gradient boosting. Comput Stat Data Anal. 2002;38(4):367–78.
41. Venkataraman S, Yang Z, Franklin M, Recht B, Stoica I. Ernest: Efficient performance prediction for large-scale advanced analytics. In: 13th {USENIX} symposium on networked systems design and implementation ({NSDI} 16), 2016. p. 363–78.
42. Amdahl GM. Validity of the single processor approach to achieving large scale computing capabilities. In: AFIPS '67 (spring): spring joint computer conference. 1967. p. 483–5. https://doi.org/10.1145/1465482.1465560.
43. Gustafson JL. Reevaluating Amdahl's law. Commun ACM. 1988;31(5):532–3. https://doi.org/10.1145/42411.42415.
44. Boden C, Rabl T, Markl V. Distributed machine learning-but at what cost. In: Machine learning systems workshop at the 2017 conference on neural information processing systems. 2017.
45. Cawley GC, Talbot NL, Chapelle O. Estimating predictive variances with kernel ridge regression. In: Machine learning challenges workshop. Berlin: Springer; 2005. p. 56–77.
46. Kernel Ridge Regression. https://scikit-learn.org/stable/modules/generated/sklearn.kernel_ridge.K ernelRidge.html.

47. Ashcroft M. Advanced machine learning: basics and kernel regression. https://www.futurelearn.com/info/courses/advanced-machine-learning/0/st eps/49560.
48. sklearn.ensemble.GradientBoostingRegressor. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingRegressor.html.
49. Wilkinson B, Allen M. Parallel programming. New Jersey: Prentice Hall; 1999.
50. Barczak ALC, Messom CH, Johnson MJ. Performance characteristics of a cost-effective medium-sized Beowulf cluster supercomputer. In: LNCS 2660. (2003). SpringerLink; 2003. p. 1050–9.
51. Vazquez C, Krishnan R, John E. Cloud computing benchmarking: a survey. In: Proceedings of the international conference on grid, cloud, and cluster computing (GCC), 2014. p. 1.
52. Sobel W, Subramanyam S, Sucharitakul A, Nguyen J, Wong H, Klepchukov A, Patil S, Fox A, Patterson D. Cloudstone: multi-platform, multi-language benchmark and measurement tools for web 2.0. In: Proc. of CCA, Vol. 8. 2008. p. 228.
53. Intel-bigdata: HiBench benchmark suit. https://github.com/Intel-bigdata/HiBench.
54. Han R, John LK, Zhan J. Benchmarking big data systems: a review. IEEE Trans Serv Comput. 2017;11(3):580–97.
55. Zhao Y, Hu F, Chen H. An adaptive tuning strategy on spark based on in-memory computation characteristics. In: 2016 18th international conference on advanced communication technology (ICACT). New York: IEEE; 2016. p. 484–8.
56. Marcu O-C, Costan A, Antoniu G, Pérez-Hernández MS. Spark versus flink: understanding performance in big data analytics frameworks. In: 2016 IEEE international conference on cluster computing (CLUSTER). New York: IEEE; 2016. p. 433–42.
57. NIST/SEMATECH e-handbook of statistical methods. National Institute of Standards and Technology (NIST). 2018. https://www.itl.nist.gov/div898/handbook/pmd/section8/pmd811.htm
58. Ding Y, Pervaiz A, Carbin M, Hoffmann H. Generalizable and interpretable learning for configuration extrapolation. In: Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering. 2021. p. 728–40.

Publisher's Note

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This study has resulted in four (4) Journal articles (all articles have already been published in Q1 Journal) and one (1) peer-reviewed conference paper. Each chapter contained within this thesis clearly outlines the original contribution and the novelty of the work. A summary of each chapter contribution is listed as follows.

- Investigated the HiBenchmark performance of Apache Hadoop and Apache Spark system considering the large scale (600 GB) datasets.
- Identified the influential parameters for individual workloads and configured the workloads according to the settings.
- The comprehensive experimental results have been obtained based on various resource utilisation and HiBench workloads.
- A novel parallelisation model called *2D-Plate* was proposed. It can predict various Apache Spark workloads' performance patterns as a function of the executors on the Hadoop cluster. This is the first reported experimental work that has come up with a simple model that can fit the data accurately for multiple workloads.
- The 2D parallel model was extended to include the data size as a parameter. A novel parallelisation model called *Fully-Connected Node* model was proposed. It can also predict various Apache Spark workloads' performance patterns as a function of the executors and the data size. Both models are based on different communication patterns between the nodes of a Hadoop cluster.
- An extensive empirical analysis of the two proposed models showed that although machine learning has better accuracy when using interpolation of data, both proposed models are more accurate when extrapolating data.

This study has discussed several points and explored different data processing approaches of existing technology. Number of real-world workloads have been considered, conducted extensive experiments, and collected a large amount of system performance data. The key challenges were to develop efficient models that can predict the performance based on limited training data points and fit the data into the models accurately.

6.2 Future Work

This section discusses some some future work of the research. Some aspects might be considered to improve the system performance as follows:

Hadoop cluster benchmark performance heavily depends on the hardware infrastructure. Since our cluster is relatively small and has only nine salve nodes with a single master node, the cluster can process and aggregate the input data between 50 GB to 600 GB. Future work needs to be done to expand the existing hardware infrastructure, particularly additional slave nodes with large storage.

In this study, seven types of workloads were considered from HiBench. HiBench provides more than 20 workloads for cluster performance analysis. Thus, future work can be done to consider more workloads in the experiments.

Literature has shown that the right parameter selection is very important for system performance. Spark has more than 150 parameters, and selecting these parameters is very challenging. This study has tuned tuned executors, data size including 15 more parameters and reset kept as default. Thus, future work can be done by considering more parameters to observe their affect on the system's performance.

The current study has shown that models can simultaneously predict runtime as a function of the number of executors and data size. Future work also needs to optimise the models by considering other functions such as executor cores and memory. Adding extra function into the models may show more insights about the models.

The current study has considered several analytical models and machine learning

regression algorithms for comparative analysis. However, other machine learning algorithms and analytical models can be taken into account for future investigation.

Appendices

Appendix 1

The contents of this appendix are from the following article. In accordance with the IEEE republishing policy, any full text that has been included, is the unmodified accepted article.

Performance Analysis of Multi-Node Hadoop Cluster Based on Large Data Sets

N. Ahmed*
Big Data Lab, SNCS
Massey University
Auckland, New Zealand
0000-0001-5663-0042

Andre L.C. Barczak
Big Data Lab, SNCS
Massey University
Auckland, New Zealand
0000-0001-7648-285X

Sibghat Ullah Bazai
Cyber Security Lab, SNCS
Massey University
Auckland, New Zealand
0000-0003-3042-5977

Teo Susnjak
Big Data Lab, SNCS
Massey University
Auckland, New Zealand
0000000194161435

Mohammed A. Rashid
Mechanical and Electrical Engineering
Massey University
Auckland, New Zealand
0000-0002-0844-5819

2020 IEEE Asia-Pacific Conference on Computer Science and Data Engineering (CSDE) | 978-1-6654-1974-1/20/$31.00 ©2020 IEEE | DOI: 10.1109/CSDE50874.2020.9411587

***Abstract*—The purpose of this paper is to assess the performance of a Hadoop cluster in MapReduce and Spark using two different clusters, one with 5 slave nodes, and another with 9 slave nodes. For the experiment, the HiBenchmark workloads WordCount and TeraSort are used with varied data scale from 50GB to 600GB. We have chosen a few different parameters and replaced their default values with the tuned values, allowing us to analyze the effects of such changes in each job's runtime. The results show that for both WordCount and Terasort workloads, depending on the tuned parameters, MapReduce and Spark achieved 64% and around 60% performance improvement at each data point. Besides, we also got slightly interesting results of speed-up progress by 1% using extra slave nodes. These results show that cluster performance can be improved by changing default values of a few parameters and adding additional slave nodes.**

***Index Terms*—Hadoop, MapReduce, Spark, HiBench, and Resilient Distributed (RDD)**

I. Introduction

In the recent past, "big data" has gained unquestionable attention and success amoung a wide range of researchers performing scientific computing, biomedical research, finance, business informatics, the Internet of Things, the trade industry, and others. Multiple questions emerge as the research community continues facing questions such as how big data will keep an impact on services. In particular, how the complexity of big data will help us to create better tools for data analytics [1]. Study reports that more than 200.000 pictures are posted on Facebook, more than 100.000 tweets to Twitter, and 72 hours of videos to YouTube in every single minute [2]. Thus, this large volume of big data becomes a major challenge for the companies and organisations to store, process, analyze, and extract meaningful information. This massive amount of data becomes a challenging management task. Due to the number of limitations, conventional database management tools become less attractive [3]. The researchers proposed a number of solutions [4], and distributed computing system is one of the prominent solutions.

Hadoop Distributed File System (HDFS) has become the most efficient distributed computing platform which provides various functions like fault-tolerance, high throughput, and job execution efficiency [5]. HDFS can distribute multiple files across the servers, and it can scale up various number of nodes at a time. HDFS consists of two nodes, namely, data node, which is called as a worker node and master node as a name node. The file system metadata and application data store separately into HDFS where the metadata stores in NameNodes and application data stores in DataNodes [6]. The nodes are responsible for managing and processing the chunk of data that come from the filesystem [7]. Hadoop has two core components: Hadoop Distributed File System (HDFS) and the MapReduce programming model. HDFS is used for data storage and MapReduce to process the workload using Map and Reduce function [8]. The map function reads the input data from HDFS and breaks into the number of pieces as a (key and value) pair and prepares intermediate results in the form of key and value pairs for the reducer function. The intermediate values are grouped and associated with the same intermediate key and value for the reducer function [9]. Both tasks execute and process independently and store results again back to HDFS, which shows high parallelism.

Matei Zahari proposed Spark in 2010 [10]; since then, it has become an open-source project. Spark offers numerous advantages, such as parallel operations, high fault tolerance, and multiple partitions. Spark uses parallel operation, namely Resilient Distributed Datasets (RDD) and Directed Acyclic Graph (DAG), and both techniques work together and accelerate Spark performance significantly [4].

This paper has investigated a multinode Hadoop cluster performance using HiBench [11] tools and large scale data sets. This work will help the system administrators and researchers to understand the system behavior when processing large scale data sets. The main contributions of this paper are as follows:

- We design two (2) multinode Hadoop clusters, namely

the Teaching and Production cluster.

- We presented an overview of the Hadoop, and Spark. In particular, how YARN distributes and allocates resource utilization to Hadoop and Spark jobs, and how Ambari manages, monitors and profiles the individual workloads running into Hadoop cluster.
- We conducted extensive performance analysis between Teaching and Production clusters in a fully-distributed mode and showed a comparison between their nodes.
- We conducted Hadoop and Spark benchmarking in both clusters consisting of multiple machines using HiBench suite and large scale data sets (600GB). We measured the performance by correlating resource utilization, splits size, and shuffle behavior parameters.
- We measured the experiment execution time for each job at every data scale, we found that the experimental results are robust and reliable than those obtained from virtual machine and pseudo-distribution mode with minimal data size.

The paper is organized as follows: **Section 2** gives the background about related research and describes Hadoop and Spark benchmarking and performance analysis. **Section 3** presents details of the cluster hardware and nodes configuration about both clusters including tuned parameters information. **Section 4** presents the experimental performance analysis of the system and finally conclude the paper in **Section 5**.

II. Related Work

Several methods and techniques are used to analyze Big Data. Firstly, Google has published a number of papers that shows how Google handles the massive amount of data. Google filesystem paper was published in 2003 [12], followed by data processing in a cluster; MapReduce in 2004 [13] and BigTable structure data in 2006 [14]. The fundamental idea of Big Data processing came from these 3 papers. Apache Hadoop became a popular tool in the last decade, and Spark opens a new window for fast data processing. Matei Zahari [4] proposed Spark for big data processing, which is faster than Google's developments. In the following section, we will overview several previously published work on cluster performance and workload analysis.

Huang [11] has proposed a comprehensive HiBench benchmark suite for the Hadoop system in 2010. The open-source MapReduce model [15] is used to investigate the system characterization. The micro and synthetic benchmark programs were used to evaluate job run time, throughput, and aggregated HDFS bandwidth, including system resources, CPU, memory, I/O utilization, and data access patterns [16]. The outcome depicts that the system performance depends not only on the right parameter values but also on the cluster size.

Chen [17] proposed a method and strategies for testing a big data analytic system and highlighted challenges such as distributed system architectures, dataset complexity, testing methods with tools, and the professionals' testability for testifying big data analytics systems. In this contest, they have proposed two benchmark cases, for instance, automated solutions for Transwarp Inceptor by TPC-DS and the comparative performance study between Hive and Spark SQL by TPCx-BigBench testing for big data analytics system. The outcome depicts that Spark-SQL performance is 1-8 times faster than Hive while Spark and Hive performance is the same considering machine learning workloads.

The comparative performance study on Hadoop, Spark, and Flink using the big data workloads is presented by Veiga [18]. In this analysis, the system parameters were characterized by modifying number of the significant parameters such as input data size, HDFS block size, and thread connection by the correlating the number of CPU cores and memory size. For workload analysis, three different benchmark workloads category is chosen and mainly highlighted the impact of different HDFS block size (64Mb, 128Mb, 256Mb, and 512Mb) using a similar input data size. The results revealed that Hadoop best perform at 128Mb and Spark and Flink at 64Mb block size. They have concluded that Hadoop can be replaced with Spark and Flink for better performance by replacing the source code.

Chen [19] proposed a new method for users to classify and quickly optimize the Hadoop performance parameters for various applications. Their proposed approach has classified the parameters in three categories and set the threshold value to find out the best parameter combination. The outcome depicts that block size is the most crucial parameter to improve the application's performance.

Gounaris [20] extensively investigates the impact of the Spark parameters such as shuffling, compression, and serialization of the application performance. The experiment is deployed with standalone mode in Spark-enabled Marenostrum III (MN3) computing infrastructure to chose alternative systematic methodology for parameter tuning applied to any computing infrastructure. They have highlighted the number of cores of Spark executor that has the most impact on maximizing performance improvement and the level of parallelism. The systematic tuning methodology, the iterative technique is applied in this analysis. The results revealed that speed-up is increased and decreased run time is due to the no-iterative method.

Samadi et. al. [21] has proposed Hadoop and Spark framework benchmarking performance based on virtual machine. The Hadoop experiment was deployed on single work station include all demons on Java Virtual Machine (JVM). On the other hand, Spark was deployed on Spark in MapReduce application. They have selected number of workloads from different categories of HiBench suite [11] with small data sizes. The results shows that Spark performance is much better with small size of data as compared to big data size while Hadoop performance is significantly better with big data size. The obtained results are justified with Hadoop deployment on Amazon Elastic Computing (EC2). They have concluded that the cluster configuration is critical to reduce job execution time, and the cluster parameter configuration must align with Mappers and Reducers.

So far, most of the Hadoop and Spark benchmarking analysis investigated based on a virtual machine with small scale

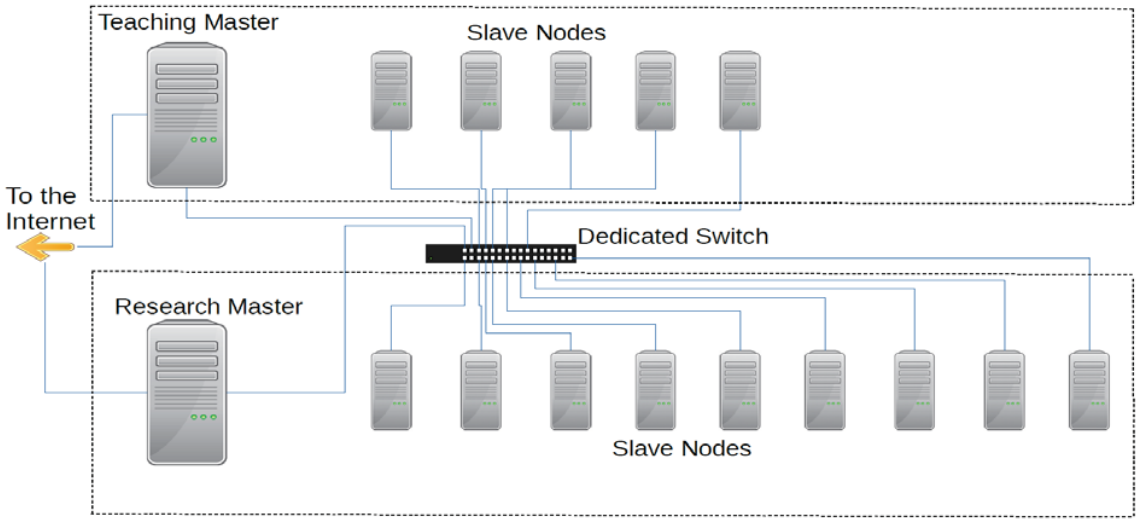


Fig. 1: Schematic of the two clusters.

TABLE I: Experimental Hadoop Cluster

Server Configuration	Processor	2.9 GHz
	Main Memory	64 GB
	Local Storage	10 TB
Node Configuration	CPU	Intel(R) Xeon(R) CPU E3-1231 v3 @ 3.40GHz
	Main Memory	32 GB
	Number of Nodes	10
	Local Storage	6 TB each, 60TB total
	CPU cores	8 each, 80 total
Software	Operating System	Ubuntu 16.04.2 (GNU/Linux 4.13.0-37-generic x86 64)
	JDK	1.7.0
	Hadoop	2.4.0
	Spark	2.1.0
Workload	Micro Benchmarks	WordCount, and Tera-Sort

data size. This paper presents real Hadoop cluster performance with large scale data size. Besides, this paper also presents the comparative performance analysis between the two clusters and shows system behaviors.

III. Hardware and Hadoop Cluster Benchmark

In our school our staff has previous experience with Beowulf clusters [22]. In 2017 the newly formed data science group built an experimental Hadoop cluster. The new cluster is composed of two master nodes and 14 slave nodes, and they can be deployed as separate clusters. In these experiments, two clusters are configured as Teaching (with 1 master and 5 nodes) and Production (with 1 master and 9 nodes) present in figure 1. The cluster has its own networking infrastructure, with dedicated switches that are not used by any other machine. This is an ideal for performance assessments because the jobs can be completed in isolation, without any interference from network traffic. We configured Yarn and Hadoop Distributed File System (HDFS) using Apache Ambari. HDFS distributes

TABLE II: Cluster Tuned Parameters Configuration for all Experiments

	Tuned Value (TV)	**Default Value (DV)**
Spark Configuration		
Executor Memory	50GB	2GB
No of Executor	8	1
Executor Cores	4	1
MapReduce Configuration		
Map Memory	4GB	512MB
Reduce Memory	8GB	512MB
Vcores	1	1
Input splits	256MB	128MB
Io.sort.mb Records	2GB	128MB
Io.sort.factor	150	100

data in a NameNode (worked as a master node), a secondary NameNode, and six DataNodes (worked as worker nodes). We allocated 3GB memory to Yarn NodeManager, and 1GB memory to ResourceManager, Driver, and Executor memories each. We used Spark version 2.1 [23] along with Yarn as a cluster manager.

The details of the cluster hardware setup and their configurations are illustrated in table I and table II presents tuned parameters' details with their default values.

IV. Experimental Results

This section presents the comparison results between the two clusters and evaluate both clusters performance using Hadoop and Spark frameworks and measure their execution time by considering different workloads and parameters.

In this analysis, we tuned MapReduce resource utilization parameters such as executors, executors core, and executors memory for 5 and 9 nodes clusters and compared them with their default parameters.

Figure 2 presents a comparison between 5 and 9 nodes cluster in MapReduce and Spark. In this analysis, the word

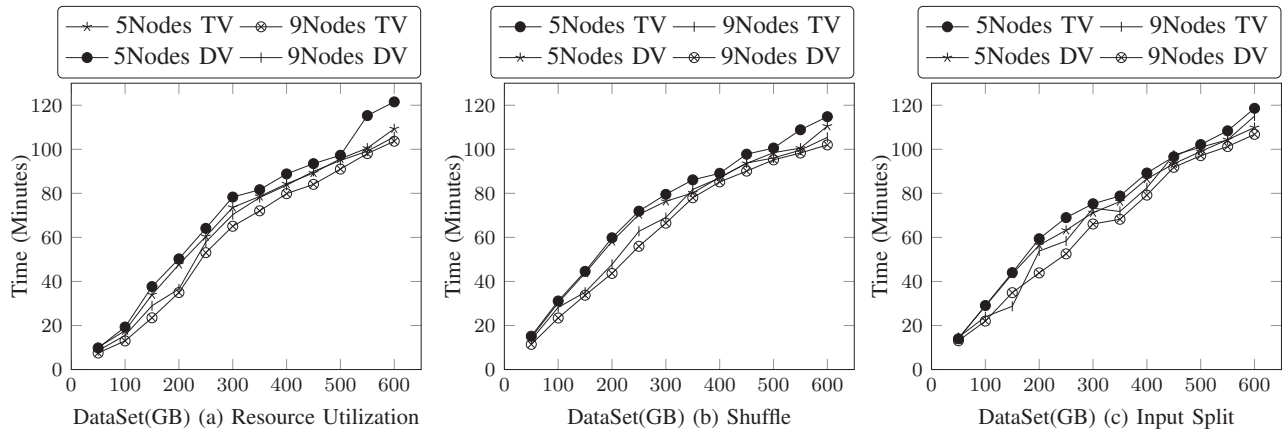


Fig. 2: Word Count) Teaching cluster vs Production cluster in MapReduce (a)Resource Utilization, (b)Shuffle, and (c)Input Split

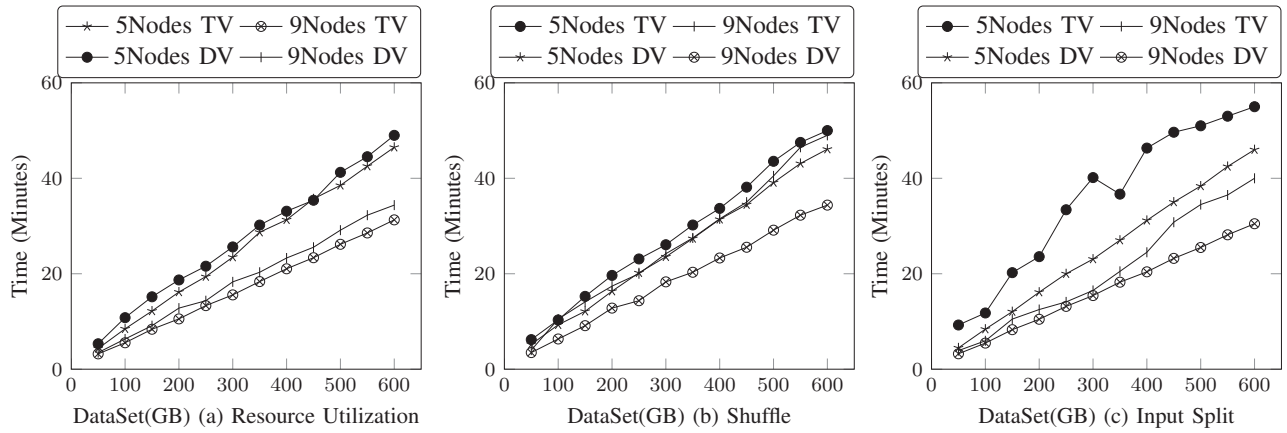


Fig. 3: Word Count) Teaching cluster vs Production cluster in Spark (a)Resource Utilization, (b)Shuffle, and (c)Input Split

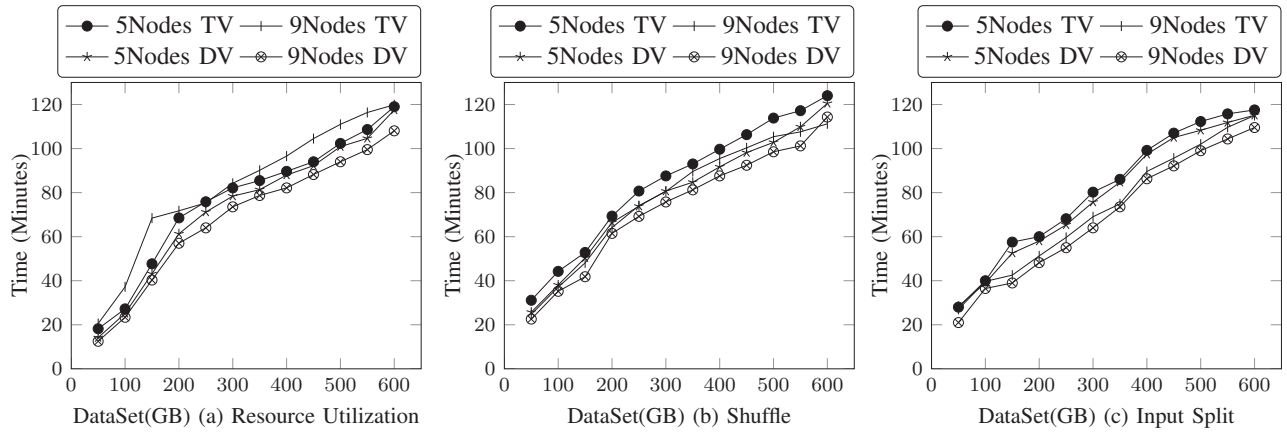


Fig. 4: Tera Sort) Teaching cluster vs Production cluster in MapReduce (a)Resource Utilization, (b)Shuffle, and (c)Input Split

count workload is executed in both clusters to find the cluster's execution time. The resource utilization, input split size, and shuffle parameters are tuned and replace the default values (DV) with tuned values (TV). The results show that each tuned parameters have a significant impact on system performance and 5 and 9 nodes cluster performance is improved by 1%.

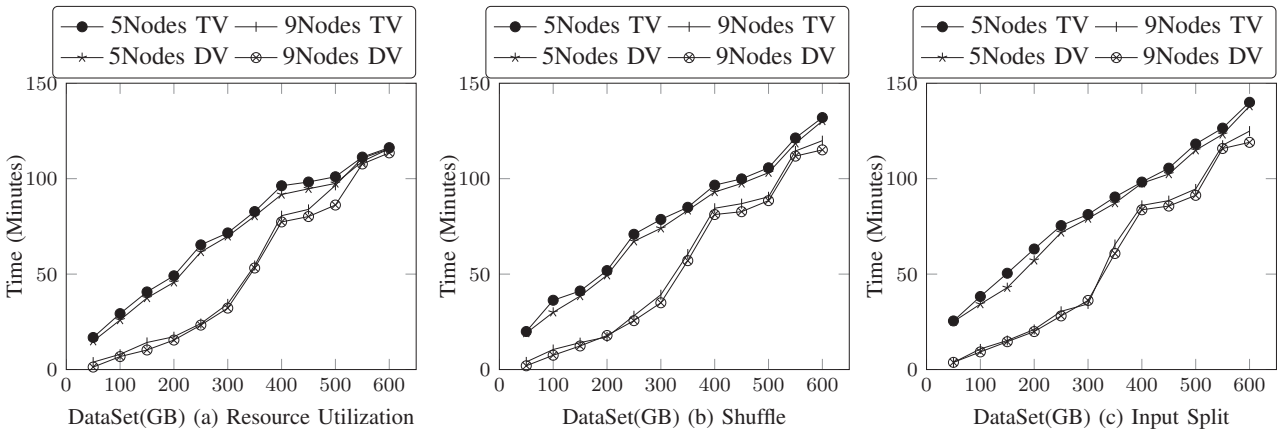


Fig. 5: Tera Sort) Teaching cluster vs Production cluster in Spark (a)Resource Utilization, (b)Shuffle, and (c)Input Split

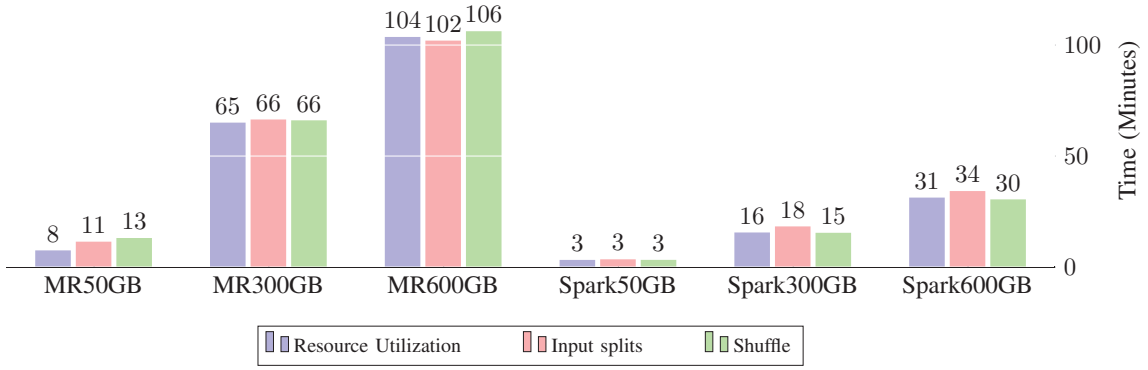


Fig. 6: **Comparison of the performance of MapReduce and Spark Word Count on various data sets and average between the two clusters**

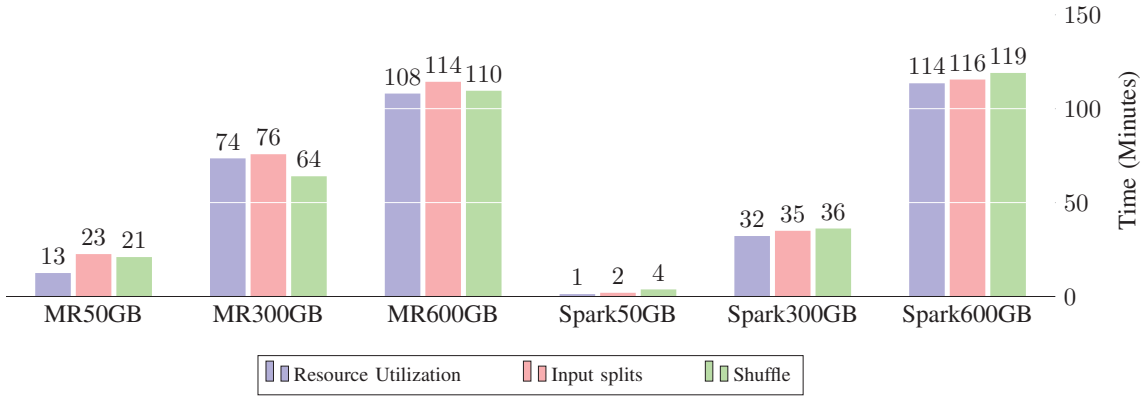


Fig. 7: **Comparison of the performance of MapReduce and Spark TeraSort on various data sets and average between the two clusters**

Spark produces slightly better performance when the word-count workload is executed. The similar parameters are chosen and tuned their values by replacing new values. It can be seen in figure 3 that 5 node cluster performance is improved by 1.053%, 1.084%, and 1.19%. Besides, the 9 node cluster performance is improved by 1.34%, 1.42%, and 1.31%. These results verify that Spark outperforms MapReduce even after changing the default parameters value and 9 node cluster

improve system performance as compared to 5 node cluster due to the extra nodes available.

TeraSort workload in Mapreduce execution time is presented in figure 4. Both clusters (5 nodes and 9 nodes) performance is evaluated by measuring the execution time. It is shown in figure 5 that by replacing the default value of resource utilization, input split size and shuffle parameter with the tuned value, the 5 node cluster performance is improved by 0.98%, 0.97%, and 0.99%. Besides, a 9 node cluster performance is improved by 1.109%, 0.97%, and 1.049%, respectively. From this results, it is evident that the tuned parameters improve both MapReduce and Spark performance.

In figure 5 the Spark performance is evaluated with the TeraSort workload in 5 nodes and 9 nodes cluster. It can be seen that the tuned resource utilization, input splits, and shuffle parameters enhance 5 node cluster performance by 1%, 1.013%, and 1.014%. Besides, 9 node cluster performance is improved by 1.019%, 1.041%, and 1.049% when the above parameters are tuned. From these results, we can conclude that the proposed tuned parameters increase cluster performance slightly.

The comparison between MapReduce and Spark, considering the resource utilization, input splits, and shuffle-tuned parameters, are presented in figure 6. The wordcount workload with various data sizes is used to evaluate system performance. From figure 6, we can see that in every data point, MapReduce took a longer time to process the workload as compared to Spark. It is evident that new parameter values enhance system performance by 64%. The reason MapReduce took extra time because of the file process in the disk and the number of blocks.

In figure 7, the comparative study between Spark and MapReduce is presented. In this analysis, the TeraSort workload is used to measure system performance. The results show that the tuned parameters of resource utilization and input split increase the system performance by an average of 50% to 86%. However, for shuffle behavior, MapReduce shows better performance compared to Spark by less than 5%. The shuffle parameter does not have any impact on system performance. In conclusion, this study revealed that the proposed parameter change has increased both our cluster performance and is strongly recommended that this parameters should be tuned.

V. Conclusion

This paper has presented the comparative performance analysis of the Hadoop cluster. The two different clusters, one with 5 slave nodes, and another with 9 slave nodes are investigated extensively. The investigation was carried out based on HiBenchmark workloads such as Wordcount and TeraSort with a large scale of input data sets. A small number of important parameters were considered and tuned their values to verify the system's execution time. This analysis showed that our cluster performance can be improved by 64% and 50% to 86% in MapReduce and Spark by replacing the default parameters. Overall, the clusters performance is increased by 1% by adding extra nodes.

References

[1] D. Boyd and K. Crawford, "Critical questions for big data: Provocations for a cultural, technological, and scholarly phenomenon," *Information, communication & society*, vol. 15, no. 5, pp. 662–679, 2012.

[2] W. Inoubli, S. Aridhi, H. Mezni, M. Maddouri, and E. M. Nguifo, "An experimental survey on big data frameworks," *Future Generation Computer Systems*, vol. 86, pp. 546–564, 2018.

[3] I. R. M. Association *et al.*, *Big data: Concepts, methodologies, tools, and applications*. IGI Global, 2016.

[4] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. Mccauley, M. Franklin, S. Shenker, and I. Stoica, "Fast and interactive analytics over hadoop data with spark," *Usenix Login*, vol. 37, no. 4, pp. 45–51, 2012.

[5] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl, "Benchmarking distributed stream data processing systems," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 2018, pp. 1507–1518.

[6] D. Borthakur, "The hadoop distributed file system: Architecture and design," 2008.

[7] A. B. Patel, M. Birla, and U. Nair, "Addressing big data problem using hadoop and map reduce," in *2012 Nirma University International Conference on Engineering (NUiCONE)*. IEEE, 2012, pp. 1–5.

[8] H. Ahmed, M. A. Ismail, and M. F. Hyder, "Performance optimization of hadoop cluster using linux services," in *17th IEEE International Multi Topic Conference 2014*. IEEE, 2014, pp. 167–172.

[9] D. Vohra, *Practical Hadoop Ecosystem: A Definitive Guide to Hadoop-Related Frameworks and Tools*. Apress, 2016.

[10] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica *et al.*, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[11] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*. IEEE, 2010, pp. 41–51.

[12] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003, pp. 29–43.

[13] S. Perera, *Hadoop MapReduce Cookbook*. Packt Publishing Ltd, 2013.

[14] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, pp. 1–26, 2008.

[15] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[16] H. Lee and G. Fox, "Big data benchmarks of high-performance storage systems on commercial bare metal clouds," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 2019, pp. 1–8.

[17] M. Chen, W. Chen, and L. Cai, "Testing of big data analytics systems by benchmark," in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2018, pp. 231–238.

[18] J. Veiga, R. R. Expósito, X. C. Pardo, G. L. Taboada, and J. Tourifio, "Performance evaluation of big data frameworks for large-scale data analytics," in *2016 IEEE International Conference on Big Data (Big Data)*. IEEE, 2016, pp. 424–431.

[19] X. Chen, Y. Liang, G.-R. Li, C. Chen, and S.-Y. Liu, "Optimizing performance of hadoop with parameter tuning," in *ITM Web of Conferences*, vol. 12. EDP Sciences, 2017, p. 03040.

[20] A. Gounaris and J. Torres, "A methodology for spark parameter tuning," *Big data research*, vol. 11, pp. 22–32, 2018.

[21] Y. Samadi, M. Zbakh, and C. Tadonki, "Performance comparison between hadoop and spark frameworks using hibench benchmarks," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 12, p. e4367, 2018.

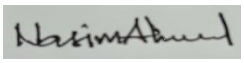
[22] A. L. C. Barczak, C. H. Messom, and M. J. Johnson, "Performance characteristics of a cost-effective medium-sized beowulf cluster supercomputer," in *LNCS 2660*. Springer Verlag, 2003, pp. 1050–1059.

[23] "Spark overview. in: Overview - spark 2.1.0 documentation." [Online]. Available: https://spark.apache.org/docs/2.1.0/index.html.

MASSEY UNIVERSITY
GRADUATE RESEARCH SCHOOL

STATEMENT OF CONTRIBUTION
DOCTORATE WITH PUBLICATIONS/MANUSCRIPTS

We, the candidate and the candidate's Primary Supervisor, certify that all co-authors have consented to their work being included in the thesis and they have accepted the candidate's contribution as indicated below in the *Statement of Originality*.

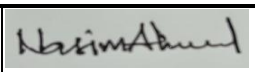
Name of candidate:	Nasim Ahmed
Name/title of Primary Supervisor:	Dr. Andre Barczak
Name of Research Output and full reference:	
N. Ahmed, A. Barczak, T. Susnjak, and M. A. Rashid," A Comprehensive Performance Analysis of Apache Hadoop and Apache Spark for Large Scale Data Sets Using HiBench " Journal of Big Data, DOI: 10.21203/rs.3.rs 43526/v1 , 2020.	
In which Chapter is the Manuscript /Published work:	Chapter 2
Please indicate:	
• The percentage of the manuscript/Published Work that was contributed by the candidate:	80
and	
• Describe the contribution that the candidate has made to the Manuscript/Published Work:	
The candidate was the main contributor of this work, and has done the literature review, experiments, and drafted the manuscript. The final draft based was completed with the suggestions from the co-authors.	
For manuscripts intended for publication please indicate target journal:	
Candidate's Signature:	
Date:	05/04/2022
Primary Supervisor's Signature:	
Date:	06/04/2022

(This form should appear at the end of each thesis chapter/section/appendix submitted as a manuscript/ publication or collected as an appendix at the end of the thesis)

MASSEY UNIVERSITY
GRADUATE RESEARCH SCHOOL

STATEMENT OF CONTRIBUTION
DOCTORATE WITH PUBLICATIONS/MANUSCRIPTS

We, the candidate and the candidate's Primary Supervisor, certify that all co-authors have consented to their work being included in the thesis and they have accepted the candidate's contribution as indicated below in the *Statement of Originality*.

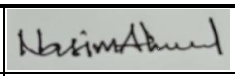
Name of candidate:	Nasim Ahmed
Name/title of Primary Supervisor:	Dr. Andre Barczak
Name of Research Output and full reference:	
N. Ahmed, A. Barczak, T. Susnjak, and M. A.Rashid,"A Parallelization Model for Performance Characterization of Spark Big Data Jobs on Hadoop Clusters ", vol 8, pp. 1-28, Journal of Big Data, 2021.	
In which Chapter is the Manuscript /Published work:	Chapter 3
Please indicate:	
• The percentage of the manuscript/Published Work that was contributed by the candidate:	80
and	
• Describe the contribution that the candidate has made to the Manuscript/Published Work:	
The candidate was the main contributor of this work, and has done the literature review, experiments, and drafted the manuscript. The final draft based was completed with the suggestions from the co-authors.	
For manuscripts intended for publication please indicate target journal:	
Candidate's Signature:	
Date:	05/04/2022
Primary Supervisor's Signature:	
Date:	06/04/2022

(This form should appear at the end of each thesis chapter/section/appendix submitted as a manuscript/ publication or collected as an appendix at the end of the thesis)

MASSEY UNIVERSITY
GRADUATE RESEARCH SCHOOL

STATEMENT OF CONTRIBUTION
DOCTORATE WITH PUBLICATIONS/MANUSCRIPTS

We, the candidate and the candidate's Primary Supervisor, certify that all co-authors have consented to their work being included in the thesis and they have accepted the candidate's contribution as indicated below in the *Statement of Originality*.

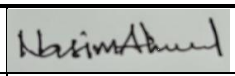
Name of candidate:	Nasim Ahmed
Name/title of Primary Supervisor:	Dr. Andre Barczak
Name of Research Output and full reference:	
N. Ahmed, A. Barczak, M.A. Rashid and T. Susnjak, "An Enhanced Parallelization Model for Performance Predictionof Apache Spark on a Multinode Hadoop Cluster", 5(4), pp. 1-25, Journal of Big Data and Cognitive Computing, 2021.	
In which Chapter is the Manuscript /Published work:	Chapter 4
Please indicate:	
• The percentage of the manuscript/Published Work that was contributed by the candidate:	80
and	
• Describe the contribution that the candidate has made to the Manuscript/Published Work:	
The candidate was the main contributor of this work, and has done the literature review, experiments, and drafted the manuscript. The final draft based was completed with the suggestions from the co-authors.	
For manuscripts intended for publication please indicate target journal:	
Candidate's Signature:	
Date:	05/04/2022
Primary Supervisor's Signature:	
Date:	06/04/2022

(This form should appear at the end of each thesis chapter/section/appendix submitted as a manuscript/ publication or collected as an appendix at the end of the thesis)

MASSEY UNIVERSITY
GRADUATE RESEARCH SCHOOL

STATEMENT OF CONTRIBUTION
DOCTORATE WITH PUBLICATIONS/MANUSCRIPTS

We, the candidate and the candidate's Primary Supervisor, certify that all co-authors have consented to their work being included in the thesis and they have accepted the candidate's contribution as indicated below in the *Statement of Originality*.

Name of candidate:	Nasim Ahmed
Name/title of Primary Supervisor:	Dr. Andre Barczak
Name of Research Output and full reference:	
N. Ahmed, A. Barczak, M. A. Rashid and T. Susnjak, Predicting Big Data Jobs Runtime: Performance Comparison between Machine Learning and Mathematical Models ", Journal of Big Data, (under review), 2021	
In which Chapter is the Manuscript /Published work:	Chapter 5
Please indicate:	
• The percentage of the manuscript/Published Work that was contributed by the candidate:	80
and	
• Describe the contribution that the candidate has made to the Manuscript/Published Work:	
The candidate was the main contributor of this work, and has done the literature review, experiments, and drafted the manuscript. The final draft based was completed with the suggestions from the co-authors.	
For manuscripts intended for publication please indicate target journal:	
with corrections and has been submitted for the 2nd round peer review on 24th of Ma	
Candidate's Signature:	
Date:	05/04/2022
Primary Supervisor's Signature:	
Date:	06/04/2022

(This form should appear at the end of each thesis chapter/section/appendix submitted as a manuscript/ publication or collected as an appendix at the end of the thesis)