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

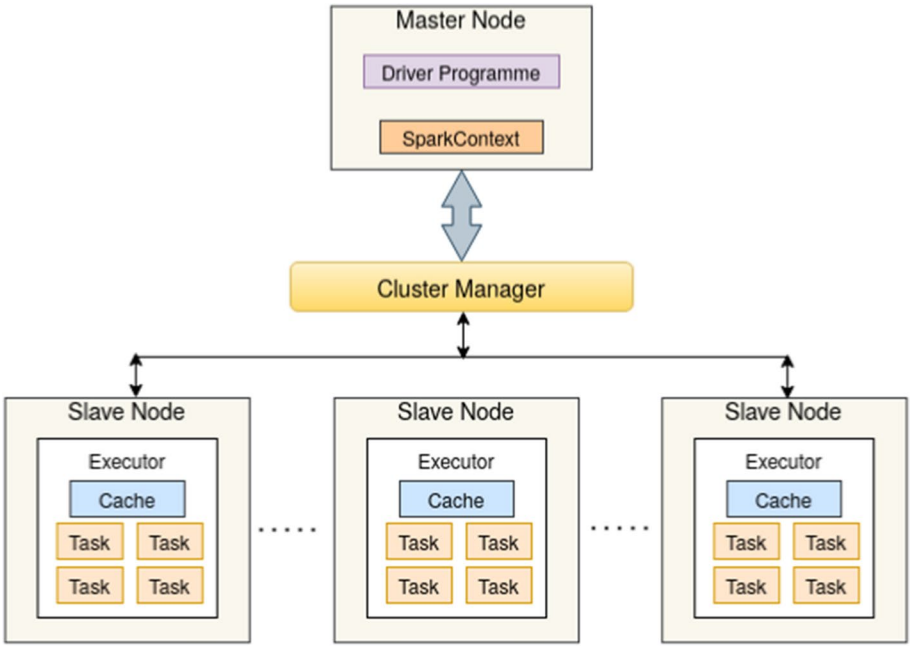

Fig. 1 A typical Spark cluster architecture

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:

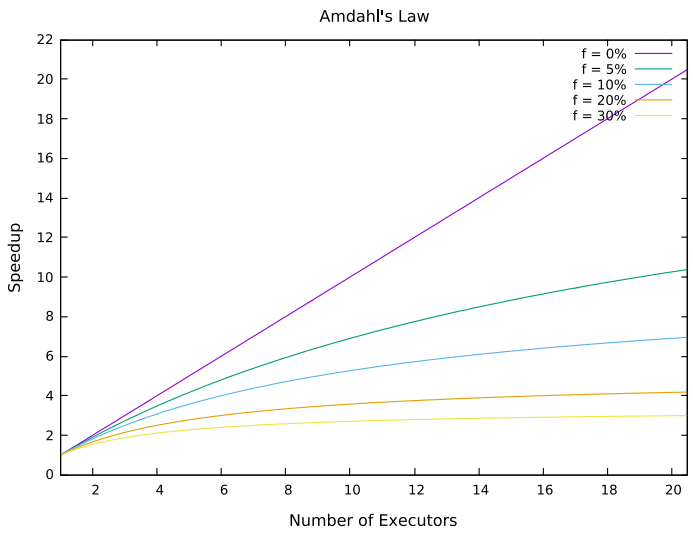

Fig. 2 Amdahl's law with various percentages of serial work

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

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.

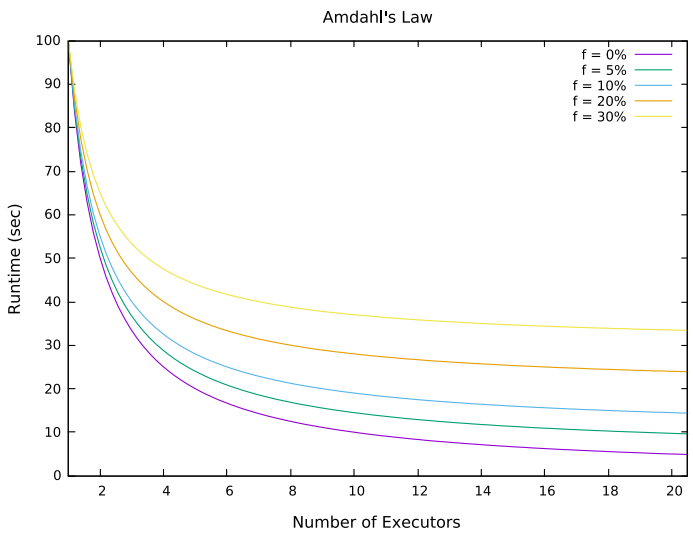

Fig. 3 Amdahl's law for various percentages of serial work

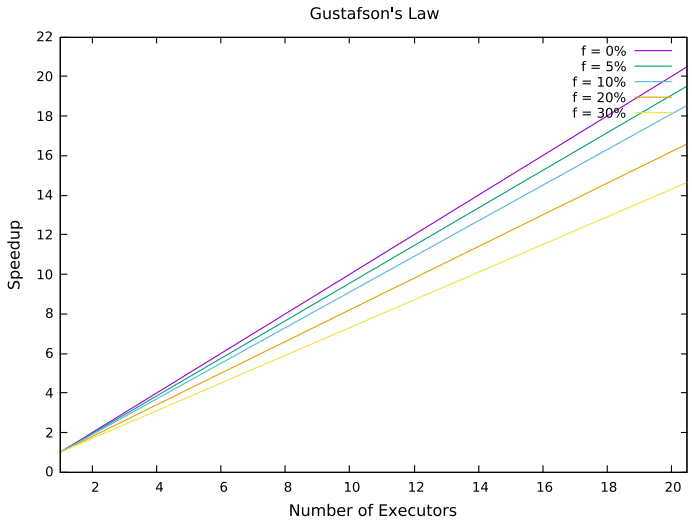

Fig. 4 Gustafson's law for various percentages of serial work

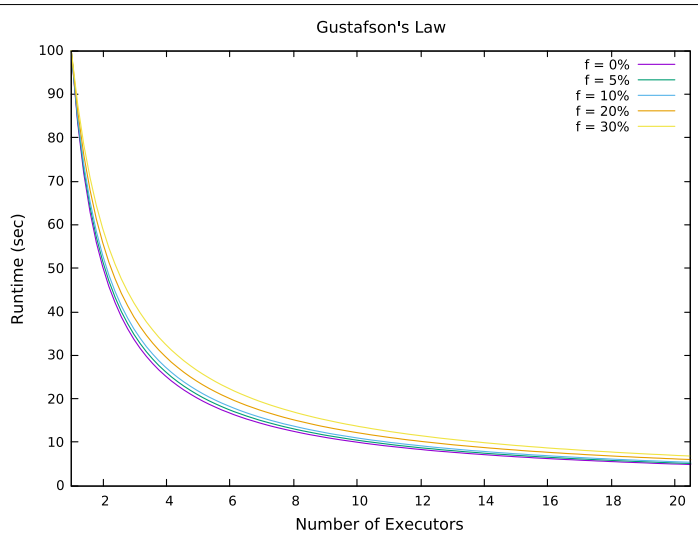

Fig. 5 Gustafson's law for various percentages of serial work

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:

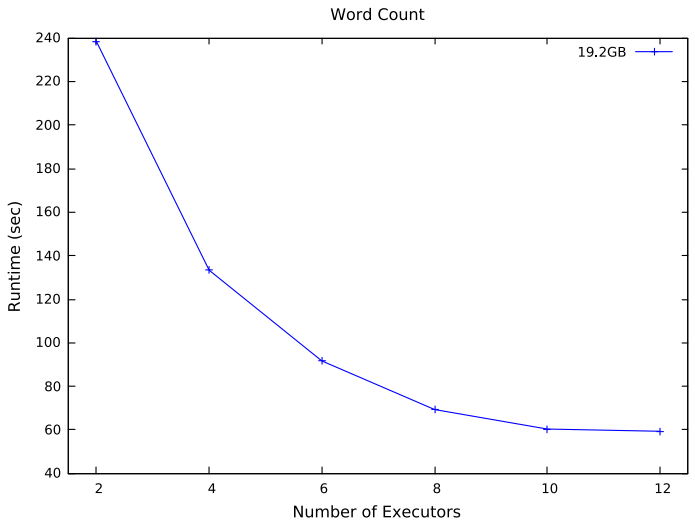

Fig. 6 A WordCount workload running on different number of executors

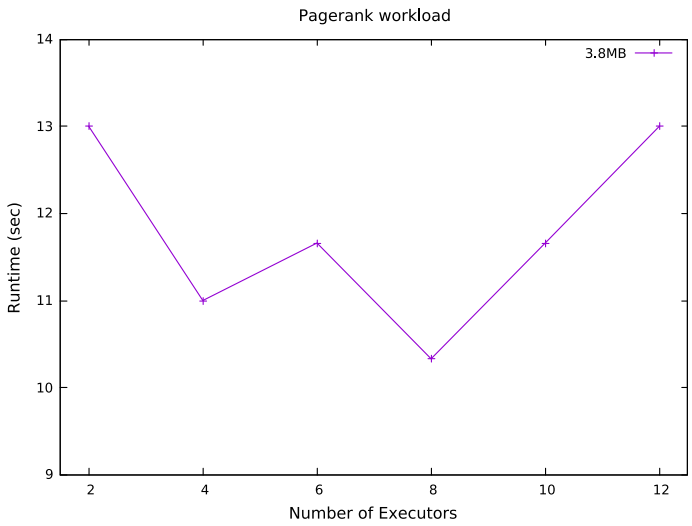

Fig. 7 Page rank workload running on different number of executors

- 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

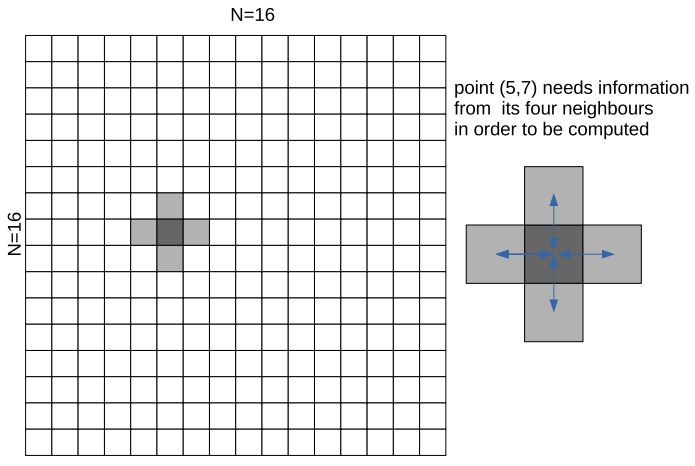

Fig. 8 A 2D plate model running on a single executor

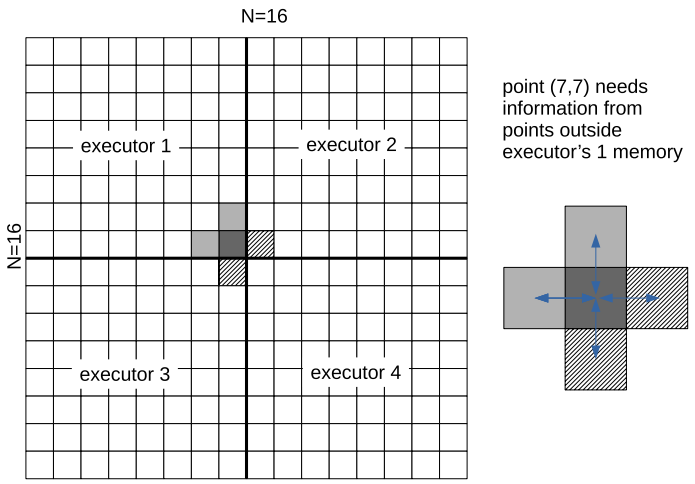

Fig. 9 A 2D plate model running on 4 executors

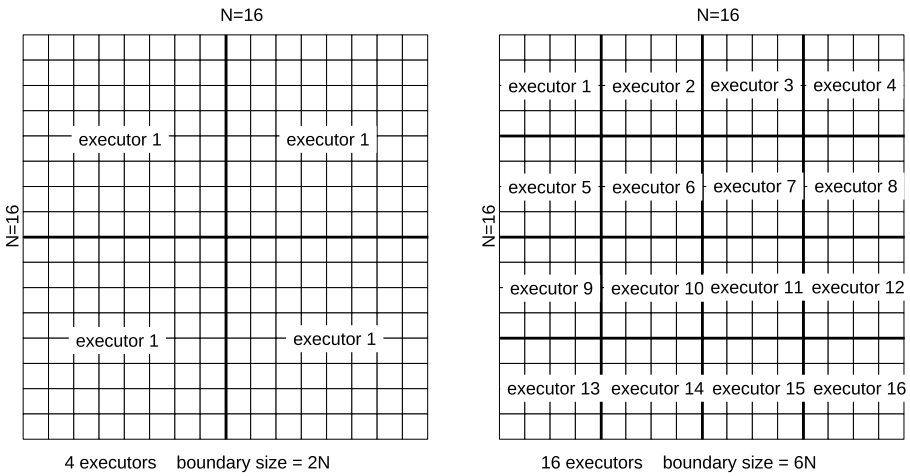

Fig. 10 Boundaries for different number of executors

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

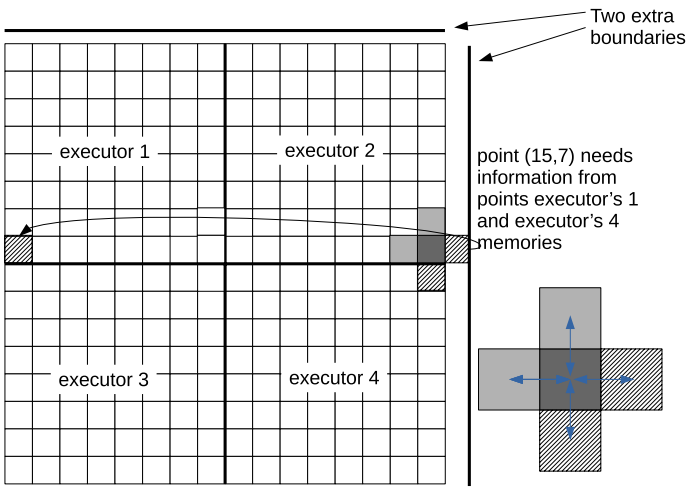

Fig. 11 Two (2) extra boundaries per plate for homogeneous communication amongst nodes

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 N^2, 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

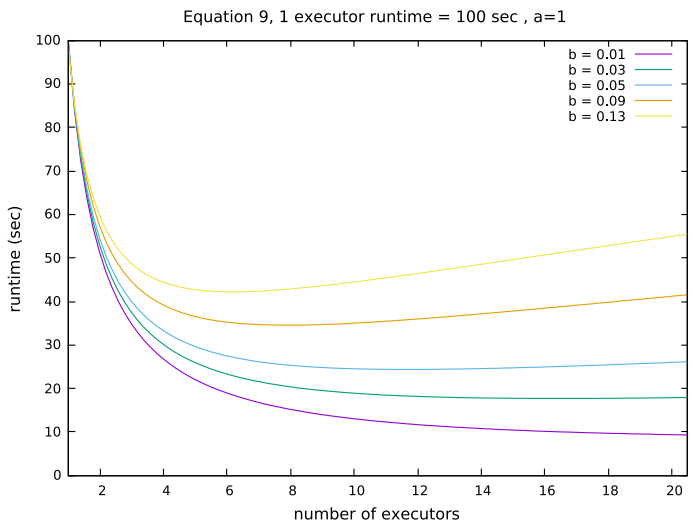

Fig. 12 Equation 9 for various b values

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

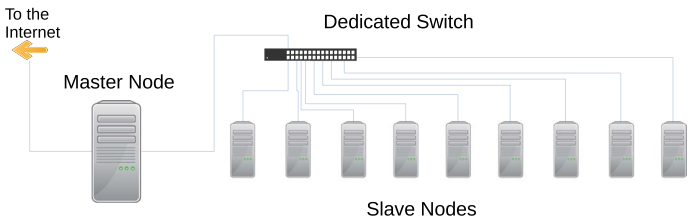

Fig. 13 The Hadoop cluster used in the experiments

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.

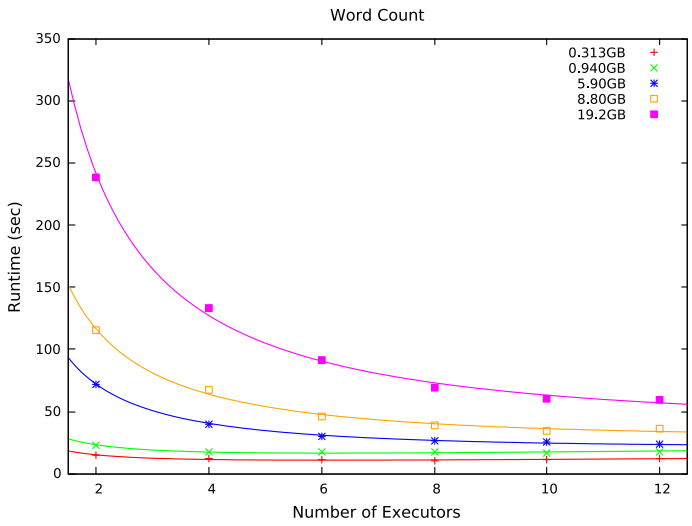

Fig. 14 Fitting the 2D plate model to Wordcount

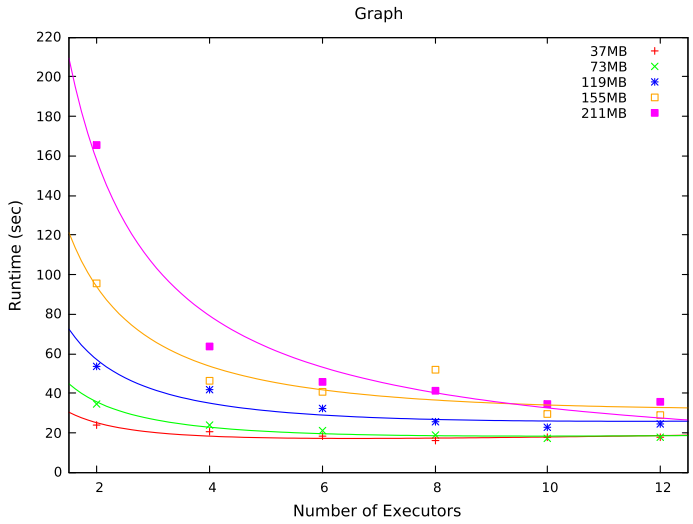

Fig. 15 Fitting the 2D plate model to NWeight (Graph)

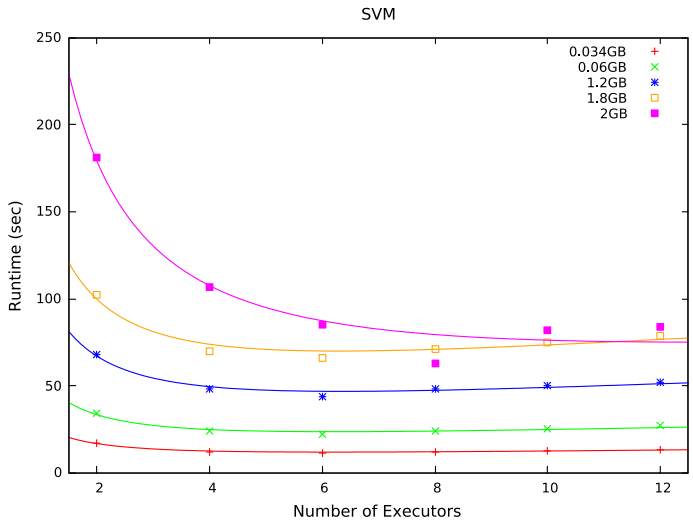

Fig. 16 Fitting the 2D plate model to SVM

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.

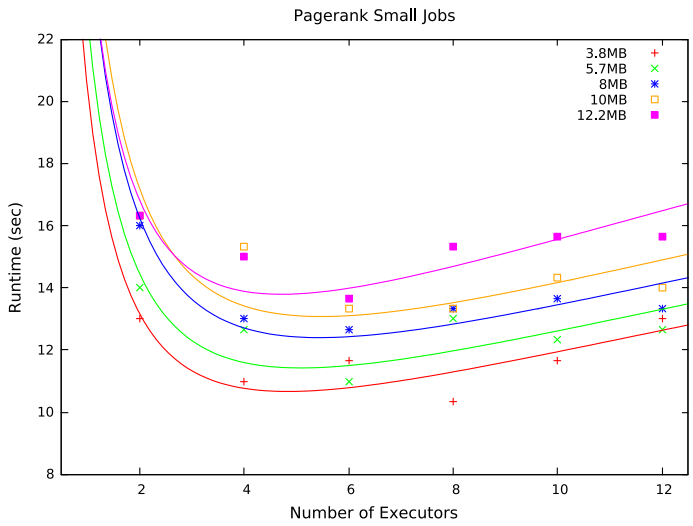

Fig. 17 Fitting the 2D plate model to Pagerank

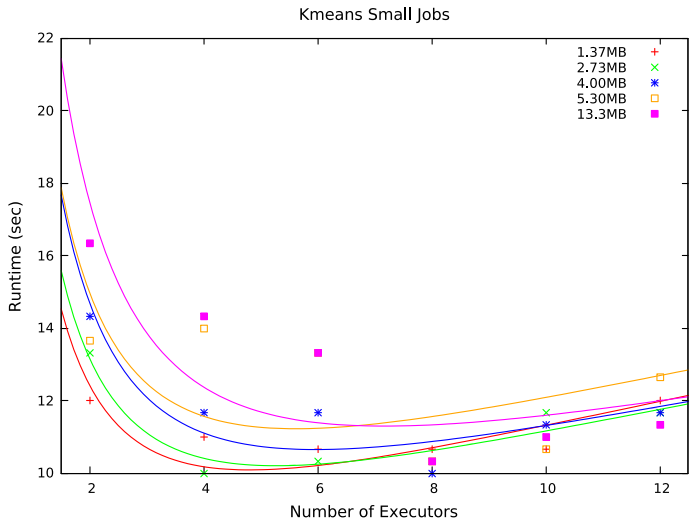

Fig. 18 Fitting the 2D plate model to Kmeans

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).

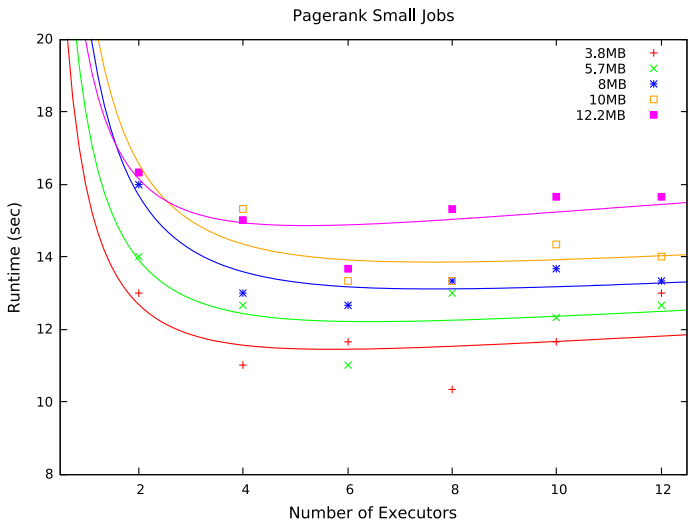

Fig. 19 Fitting Eq. 11 to Pagerank jobs, with c=0.14

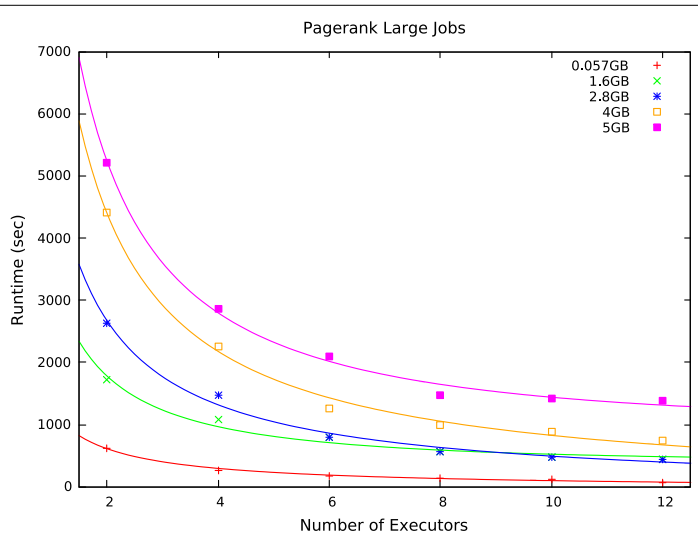

Fig. 20 Fitting the 2D plate model to Pagerank with larger problem sizes

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

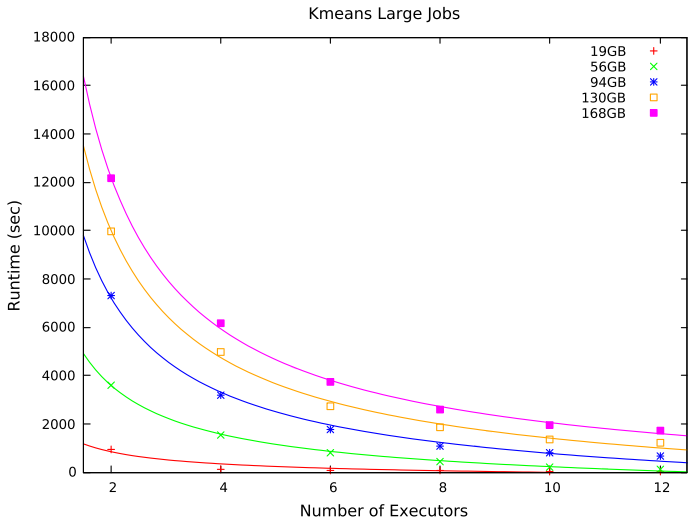

Fig. 21 Fitting the 2D plate model to Kmeans with larger problem sizes

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

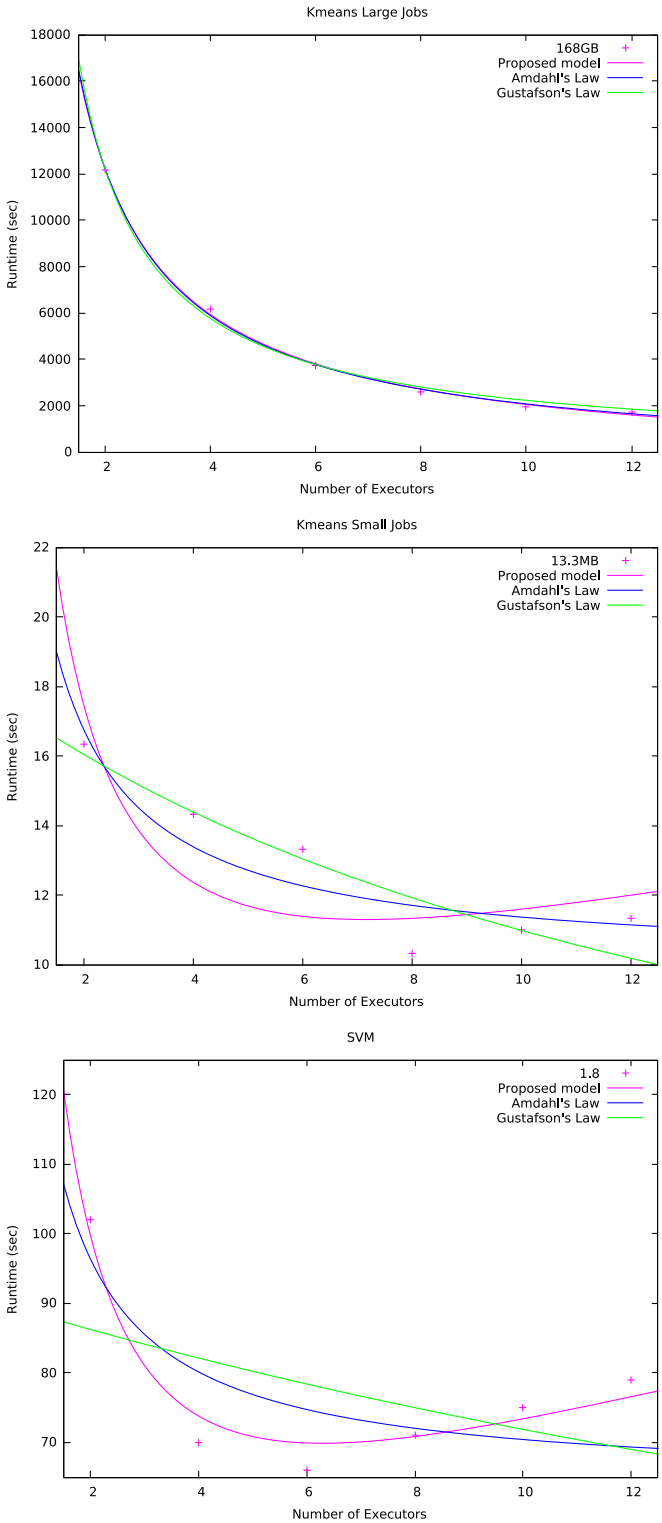

Fig. 22 Comparison for the fitting accuracy using the proposed model, Amdahl's law and Gustafson's law

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