dSpark: Deadline-based Resource Allocation for Big Data Applications in Apache Spark

Muhammed Tawfiqul Islam, Shanika Karunasekera and Rajkumar Buyya

Cloud Computing and Distributed Systems (CLOUDS) Laboratory
School of Computing and Information Systems
The University of Melbourne, Australia
Email: tawfiq.shobhon@gmail.com, {karus, rbuyya}@unimelb.edu.au

***Abstract*—Large-scale data processing framework like Apache Spark is becoming more popular to process large amounts of data either in a local or a cloud deployed cluster. When an application is deployed in a Spark cluster, all the resources are allocated to it unless users manually set a limit on the available resources. In addition, it is not possible to impose any user-specific constraints and minimize the cost of running applications. In this paper, we present dSpark, a lightweight, pluggable resource allocation framework for Apache Spark. In dSpark, we have modelled the application completion time with respect to the number of executors and application input/iteration. This model is further used in our proposed resource allocation model where a deadline-based, cost-efficient resource allocation scheme can be selected for any application. As opposed to the existing frameworks that focus more on modelling the number of VMs to use for an application, we have modelled both the application cost and completion time with respect to executors, hence providing a fine-grained resource allocation scheme. In addition, users do not need to specify any application types in dSpark. We have evaluated our proposed framework through extensive experimentation, which shows significant performance benefits. The application completion time prediction model has a mean relative error (RE) less than 7% for different types of applications. Furthermore, we have shown that our proposed resource allocation model minimizes the cost of running applications and selects effective resource allocation schemes under varying user-specific deadlines.**

***Index Terms*—Big Data, Cloud Computing, Apache Spark, Resource Allocation, Deadline, Cost Minimization**

I. Introduction

Now-a-days, huge amount of data is generated from social media, mobile devices, IoT and many other emerging applications. Therefore, data processing and analytics have become really important in all the major domains, such as research, business and industry. Apache Spark [1] is one of the most prominent big data processing platforms. It is an open source, general purpose, large-scale data processing framework. It mainly focuses on high speed cluster computing and provides extensible and interactive analysis through high level APIs. Spark can perform batch or stream data analytics, machine learning and graph processing. It can also access diverse data sources like HDFS [2], HBase [3], Cassandra [4] etc. and use Resilient Distributed Dataset (RDD) [5] for data abstraction. Spark runs programs faster than Hadoop-MapReduce [6] by performing most of the computations in-memory. In addition, it caches intermediate results in memory for faster re-processing of data. Spark can run locally in a single desktop, in a local cluster and on the cloud. It runs on top of Hadoop Yarn [7], Apache Mesos [8] and the default standalone cluster manager.

In a Spark cluster, there are one or more worker nodes with the available resources (CPU cores, memory and disk). In addition, there is a master node which is responsible for allocating these resources to the applications. Each application uses the allocated resources to create executor processes where it can run tasks in parallel. Resource allocation in a Spark cluster can be done through the following three mechanisms: (1) *Default Resource Allocation*. It is used when the applications are submitted in a Spark cluster without specifying any resource allocation details. In this approach, all the applications will run in a FIFO style and each application consumes all the worker nodes. Hence, applications run one after another and when an application is running, it will use up all the worker nodes to create executors. (2) *Static Resource Allocation*. When an application is submitted, the user specifies how many executors, cores, memory etc. an application can have. Therefore, resources can be shared among multiple applications from one or more users. (3) *Dynamic Resource Allocation*. If this mode is turned on, applications may release idle executors to give back some resources to the cluster which can also be taken back in future if needed.

However, there are three major problems in these resource allocation mechanisms. First, when a single application is running in the cluster with the default resource allocation mechanism, it will consume all the resources. As a consequence, resource sharing among applications will be prevented. Second, in static resource allocation, the user has to manually set the amount of resources each application is going to use. Even with dynamic resource allocation, the user still has to set the initial amount of resources. As a result, improper allocation of resources might lead to severe performance issues. Lastly, if a production cluster has user-specific deadlines, default resource allocation mechanism may not work since any application with a strict deadline might have to wait in the FIFO queue. Furthermore, inappropriate resource allocation in both static and dynamic resource allocation techniques might affect the deadlines.

In this paper, we propose a resource allocation framework for distributed batch-based applications in Apache Spark. In addition, we propose an application completion time prediction model which can be built from the application profiles. This model is further used in the resource allocation model to select a deadline-based, cost-effective resource allocation scheme.

The main **contributions** of this work are as follows:

- We design an automatic, light-weight, pluggable **dSPark** *resource allocation framework* for Apache Spark that works from the master node along with the underlying cluster manager.
- We propose a *resource allocation model* where a cost-effective, deadline-based Resource Allocation Scheme (RAS) can be found for an application.
- We propose a *model* that predicts the completion time of an application based on the number of executors and properties of the application.
- We develop a *Spark Profiler* to profile any application with respect to varying input workloads, iterations, resource allocation schemes etc.
- We propose a simple *algorithm to generate Resource Allocation Schemes (RAS)* which can be used to deploy applications in an Apache Spark cluster.
- We implement the framework using the proposed models and algorithms. In addition, we run comprehensive experiments to show the accuracy and performance benefits of our proposed models.

The rest of the paper is organized as follows. In section II, we discuss the background of Apache Spark. In section III, we describe the existing works related to this paper. In section IV, we formulate the resource allocation and application completion time prediction models. In section V, we illustrate the architecture of the proposed dSpark framework. In section VI, we explain the methods we have used to implement the proposed framework. In section VII, we evaluate the performance of our proposed models. Section VIII concludes the paper.

II. Background

As compared to the disk-based MapReduce tasks of a typical Hadoop system, Apache Spark allows most of the computations to be performed in memory and provides better performance for some applications such as iterative algorithms. The intermediate results are written to the disk only when it cannot be fitted into the memory.

Fig. 1 shows a typical Apache Spark cluster. Applications are submitted through a cluster manager to run in the cluster. Spark supports *Apache Mesos* or *Hadoop Yarn* as cluster managers to allocate resources among applications. In addition, its own default *Standalone* cluster manager is also sufficient to handle a production cluster. All these cluster managers support both static and dynamic allocation of resources. In static resource allocation, each application is deployed with a fixed amount of resources which cannot be changed during the life-cycle of that application. However, in dynamic resource allocation, idle resources can be released to the cluster and any other application can use them. These resources can also be taken back from the cluster in future if needed.

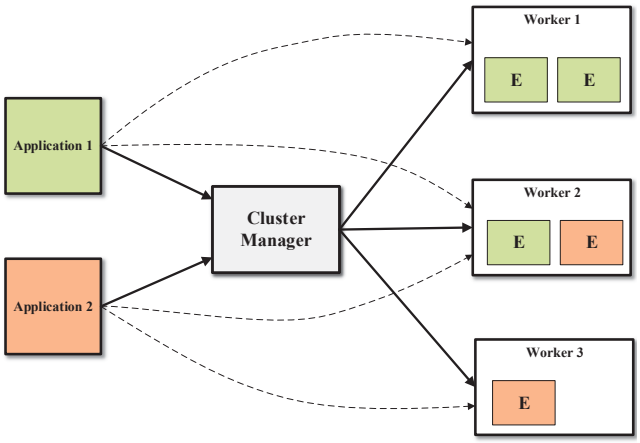


Fig. 1. An Apache Spark Cluster

Workers are the physical/compute nodes of an Apache Spark cluster where one or more application processes can be created depending on the resource capacity. In cloud deployments, one or more worker nodes can be created inside each allocated Virtual Machines (VM). A Spark cluster can have one or more worker nodes but there is only a single *Master* node that is responsible for managing the worker nodes.

Each application in Spark has a *SparkContext* object in its main program (also called the *Driver Program*) which creates and maintains *Executor* processes on worker nodes. An application uses its own set of executors to run tasks in parallel, in multiple threads and to keep data in memory and storage. In addition, these executors live for the whole duration of that application. All the executors of the same application must be identical in size. Hence, they will have same amount of resources (CPU cores, memory, disk). There are two benefits of isolating applications from each other. First, a driver program can independently schedule its own tasks in the acquired executors. Second, each worker can have multiple executors from different applications running in their own JVM processes.

Spark uses *Resilient Distributed Datasets (RDD)* to hold data in a fault tolerant way. Each job/application is divided into multiple sets of tasks called stages which are inter-dependant. All these stages form a directed acyclic graph (DAG) and each stage is executed one after another.

III. Related Work

A vast amount of research has been done in application performance modelling, resource provisioning and scheduling in cloud-based systems. Here, we only focus on discussing the related research works done for big data processing platforms. Most of the research were done for MapReduce-based big data frameworks as it was the most popular big data processing paradigm in the last decade. ARIA [9] was designed for MapReduce based environments where job profiles of map and reduce tasks of an application are collected to build job

profiles. A MapReduce performance model is built from the job profiles which is used to estimate the required resources for a job completion given its Service Level Objective (deadline). In [10], performance modelling of MapReduce jobs was done in heterogeneous cloud environments. [11] proposed a resource provisioning framework for MapReduce workloads. [12] showed a deadline-based workload management for MapReduce workloads. [13] predicts the expected performance of a big data workload from historical performance results using Support Vector Machine (SVM). However, these approaches are not straight forward to apply in Apache Spark as it is a DAG-based in-memory analytics platform.

[14] evaluated the performance of Apache Spark in MareNostrum supercomputer to explore its efficiency and applicability in HPC setup. In addition, they have also developed a framework called Spark4MN to automate the use of a Spark cluster in HPC environment. Furthermore, they have explored the impacts on performance by differernt parameters like worker size, tasks per core etc. Similar to this work, [15] also investigated different configuration parameter tuning of Spark applications. They have identified a set of important paramemters and provided a trial-and-error based methodology to tune these paramemters for performance speed-ups. A machine learning based configuration parameter tuning approach is proposed in [16]. Their method is composed of binary classification and multi-classification. As Spark has a huge parameter space, they have taken a random sample from the parameter space and generated a parameter list of 500 records for each type of workload. Then real execution data on these parameter lists is taken to train their models. Their experimental results show that a Decision Tree (C5.0) provides good accuracy and performance in diverse workloads.

TABLE I. Related Work

Parameter	Related Work						dSpark
	[15]	[16]	[17]	[18]	[19]	[20]	
Framework	✗	✗	✗	✗	✗	✗	✓
Performance Modelling	✓	✓	✓	✓	✓	✓	✓
Executor Cost Modelling	✗	✗	✗	✗	✗	✗	✓
Deadline	✗	✗	✗	✗	✗	✓	✓
Cost Saving	✗	✗	✗	✗	✗	✓	✓
Resource Saving	✗	✗	✗	✗	✗	✗	✓

[17] investigated the problem of resource waste that occurs while a Spark application runs in all the nodes in a cluster. To address this problem, they have proposed dynamic partitioning based solutions that tune the degree of parallelism of Spark application during execution to reduce resource consumption. To achive this, they have to trade small amount of running time. [18] built multiple polynomial regression models on the application profile data and applied k-fold cross validation to choose the best model to predict application execution time with unknown input data set or cluster configuration. [19] tried to model application performance in DAG-based in-memory analytics platforms and they have used Apache Spark to validate their methods. In this work, the execution times from different stage of an application is collected and then used to predict the execution time of the application.

TABLE II. Definition of Symbols

Symbol	Definition
A	a Spark application
E	total number of executors
P_{vm}	price (per second) of a VM
M_e	memory (GB) in each executor
C_e	number of cores assigned to each executor
P_e	price of one executor
N_w	total number of workers in the cluster
C_w	total number of cores in each worker
M_w	total memory (GB) in each worker
E_{max}	maximum possible executors in the cluster
T	completion time of an application
D	deadline of an application
I	input size or iteration of an application
RAS	resource allocation scheme
$RASL$	list of resource allocation schemes

However, these works did not consider cost minimization and user-specific deadlines. In optEx [20], a deadline oriented cost optimization model was proposed. However, in optEx, the user needs to specify the type of the application before deployment.

In all these works related to Spark performance modelling, they have considered VMs as the unit of resource of an application and tried to estimate application completion time with different number of VMs. However, in dSpark, we have considered executor processes as a unit of resource for the application. For any size of machine either in local or cloud deployed cluster, our model is capable of finding more fine-grained resource allocation schemes. Therefore, multiple applications will be able to run executor processes in the same worker node depending on the worker and executor size. Furthermore, dSpark also utilizes a flexible cost model that can be customized to integrate user's own pricing policies. Lastly, dSpark can provide efficient resource allocation schemes under varying SLO deadlines. The summary of the comparison between our work and other closely related works is given in Table I.

IV. Problem Formulation

A. Cost-efficient Resource Allocation Model

An Apache Spark cluster comprising of master and worker nodes can be deployed on cloud Virtual Machines (VM). For simplicity of our proposed model, we assume that all the VMs used as worker nodes are homogeneous. Therefore, each of the VM will have same amount of CPU cores, memory and storage disk. To deploy an application in the cluster, a Resource Allocation Scheme (RAS) needs to be defined. In each RAS, the total number of executors, CPU cores in each executor and memory in each executor should be specified. Our goal is to choose a cost-efficient RAS which ensures that an application will be completed before the user-specified deadline.

Suppose, we have an Apache Spark cluster with N_w total number of worker nodes and one (1) master node. In addition, all these nodes are created in distinct VMs. As all the workers are homogeneous, each worker has C_w CPU cores and M_w total memory. Furthermore, the price of running each VM is P_{vm} ($) per second. For a particular application (A), the user specifies a deadline (D) before which this application needs to complete. In addition, the user also defines the cores per executor (C_e) value. If all the memory of a worker (M_w) is evenly associated among all the cores (C_w), then (M_w/C_w) amount of memory will be associated with each core. Therefore, memory in each executor (M_e) will be $C_e * (M_w/C_w)$. In Apache Spark, for a particular application, all the executors need to be identical. Therefore, our problem is now to find the number of executors (E) to use with an application that meets the user deadline (D) and also minimizes the total cost.

We model this problem as a constrained non-linear optimization problem as follows:

$$\text{Minimize: } Cost = P_e * E * T \tag{1}$$

subject to:

$$1 \leq E \leq E_{\text{max}} \tag{2}$$

$$T \leq D \tag{3}$$

where:

$$P_e = C_e * (P_{\text{vm}}/C_w) \tag{4}$$

$$M_e = C_e * (M_w/C_w) \tag{5}$$

$$E_{\text{max}} = N_w * (C_w/C_e) \tag{6}$$

$$T = f(E, I) \tag{7}$$

$$E, C_e, M_e \in \mathbb{Z} \tag{8}$$

Cost Minimization: Eqn. 1 shows the objective function where $Cost$ is the dependent variable and executors (E) and application completion time (T) are the decision variables. In addition, P_e is a constant value which represents the cost of running one (1) executor.

Executor Capacity Constraint: As shown in Eqn. 2, we have a lower bound and an upper bound on the number of executors of an application. The lower bound should be one (1) as each application needs at least 1 executor to process data and the upper bound (E_{max}) depends on the available cluster resources as shown in Eqn. 6.

Application Deadline Constraint: As shown in Eqn. 3, application completion time (T) of a selected configuration should meet the user specified deadline (D). In a case where the model finds multiple resource configurations that satisfy the deadline constraint, it will only select the one which has the lowest cost.

Executor Price Estimation: As we associate equal amount of memory with all the CPU cores in a VM, the number of used CPU cores represents the price of a VM. Therefore, we can find the price (per second) of a single CPU core from the actual VM price by dividing the price for running each VM (P_{vm} ($)) with total number of available cores in a VM (C_w). Eqn. 4 shows the price estimation function of an executor process. Pricing policy of this model can be easily converted to a different scenario and allow users to use their own VM pricing model.

Memory Capacity Constraint: The amount of memory for an executor (M_e) depends on the number of cores (C_e) in that executor. In addition, it is also capped by the total memory of a worker as shown in Eqn. 5.

Application Completion Time Prediction: As shown in Eqn. 7, the proposed optimization model finds the value of completion time (T) as a function of executor (E) and the total application input (I). We propose an application completion time prediction model to be used as this function. This model will be discussed in detail in the following subsection.

Integer Constraints: The number of executors (E), cores in each executor (C_e) and memory in each executor (M_e) must be integers as shown in Eqn. 8.

B. Application Completion Time Prediction Model

An Apache Spark application uses it's allocated executors to process multiple chunks/splits of the whole input in parallel. The partitioning or splitting of the input imposes a little overhead on the actual running time. In addition, after all the processing is finished, the result needs to be serialized which also adds up to the total execution time. If the number of input chunks is more than the number of executors, these input chunks are processed like a batch in each executor. However, adding too many executors to achieve more parallelism can cause overheads due to serialization, de-serialization and intensive shuffle operations in the network. Therefore, when an application is given more and more executors, performance boost can be significant at the start. However, after some point, adding more executors does not give any performance benefit rather resources are wasted. Therefore, for a fixed input (I) of an application, we can assume that the relationship between executors (E) and completion time (T) can be modelled like a power function as:

$$T(E) = \alpha * E^{\beta} + \gamma \tag{9}$$

where α, β and γ are the power model coefficients.

However, in reality, the application input is not a fixed parameter. Therefore, we further assume that the coefficients in Eqn. 9 are determined by the application input (I) and can be modelled like a power function as:

$$\alpha(I) = u_\alpha * I^{v_\alpha} + w_\alpha \tag{10}$$

$$\beta(I) = u_\beta * I^{v_\beta} + w_\beta \tag{11}$$

$$\gamma(I) = u_\gamma * I^{v_\gamma} + w_\gamma \tag{12}$$

where, $\{u_\alpha, v_\alpha, w_\alpha\}$, $\{u_\beta, v_\beta, w_\beta\}$ and $\{u_\gamma, v_\gamma, w_\gamma\}$ are the power model coefficients in Eqn. 10 Eqn. 11 and Eqn. 12, respectively. If we substitute α, β and γ of Eqn. 9, we find:

$$T(E, I) = (u_\alpha * I^{v_\alpha} + w_\alpha) * E^{(u_\beta * I^{v_\beta} + w_\beta)} + u_\gamma * I^{v_\gamma} + w_\gamma \tag{13}$$

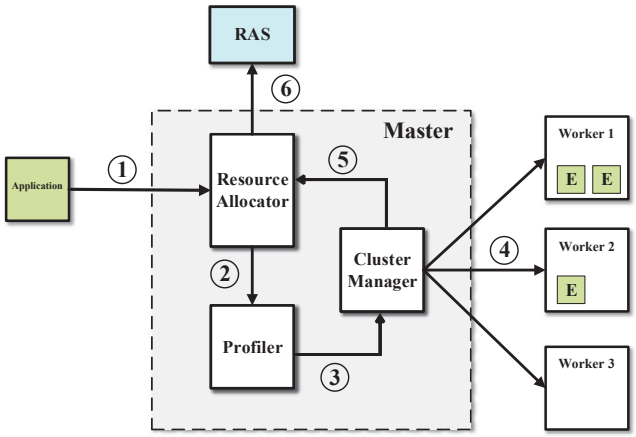


Fig. 2. dSpark Architecture

Eqn. 13 establishes the relationship of application completion time (T) with respect to both executor (E) and application input or iteration (I). Hence, it can be used in the resource allocation model (Eqn. 7) to determine the completion time of an application. In addition, this model can predict application completion time with any size of input/ iteration and any number of possible executors.

To determine the coefficients in Eqn. 13 for a particular application, we make n observations of the application with different inputs $\{I_1 \, to \, I_n\}$. For each application input, we measure the T values with respect to different E values and fit them to establish a relationship as shown in Eqn. 9. Therefore, we will get three (3) set of coefficients: $\{\alpha_1 \, to \, \alpha_n\}$, $\{\beta_1 \, to \, \beta_n\}$, $\{\gamma_1 \, to \, \gamma_n\}$ for n observations. Now, if we fit $\{\alpha_1 \, to \, \alpha_n\}$ vs $\{I_1 \, to \, I_n\}$ values as Eqn. 10, we will find $\{u_\alpha, v_\alpha, w_\alpha\}$ coefficient values. Similarly, the values of coefficient sets $\{u_\beta, v_\beta, w_\beta\}$ and $\{u_\gamma, v_\gamma, w_\gamma\}$ can be found.

V. dSpark Framework Overview

A production cluster can be built with multiple computing nodes connected in a local area network (LAN). However, we can avoid the hassle of maintaining local machines by using cloud services as it offers more affordable and flexible computing resources to deploy a cluster. dSpark framework can be used both locally or in a cloud deployed cluster.

Fig. 2 shows the proposed architecture of the *dSpark Framework*. We have two (2) main modules: *Profiler* and *Resource Allocator*. These modules work collaboratively on top of the cluster manager to generate a cost-effective, deadline-aware RAS for an application. This RAS can be used for real deployment of this application in the cluster.

1) Resource Allocator: It is the main component of our system. Algorithm 1 shows the steps performed by this module. As an input to this algorithm, the application program (A), input/iteration (I), user-specific deadline (D) and VM price (P_{vm}) is given. At first, the Profiler module is invoked to generate application profiles (line 3). Then the application completion time prediction model is built (line 4) as discussed in section III.B. While building the model, the algorithm finds all the coefficient values of Eqn. 13. The $Find - RAS()$ procedure called in line 5 implements the resource allocation model to select the optimal RAS.

Algorithm 1 Resource Allocator Algorithm

1: **Input:** $A, I, D \, and \, P_{\text{vm}}$
2: **Output:** Resource Allocation Scheme (RAS)
3: $ApplicationProfiles \leftarrow$ PROFILER(A, I)
4: TIME-ESTIMATE-MODEL$(ApplicationProfiles)$
5: $RAS \leftarrow$ FIND-RAS$(D, I, P_{\text{vm}}, \alpha, \beta, \gamma)$
6: **return** RAS

2) Spark-Profiler: This module is controlled by the Resource Allocator module to generate application profiles for an application. The profiler module runs the application with different RAS, varying inputs or iterations (in case of iterative applications like PageRank) in the cluster. After that, it uses a sub-module called *LogParser* to get the completion times of an application from the logs in the master node. Finally, it generates the application profiles and sends to the Resource Allocator module. Spark-Profiler is configurable by the user to set the portion of input that needs to be profiled before the actual deployment of an application. By default it uses 10% of the input workload for profiling. As the application completion time prediction model uses multiple increasing input to build the model, we used the initial chunk repeatedly to increase it to the desired size.

Algorithm 2 Resource Allocation Scheme (RAS) Generation

1: **Input:** $N_w, M_w, C_w \, and \, C_e$
2: **Output:** Resource Allocation Scheme List ($RASL$)
3: CALCULATE(M_e) $(Eqn. 5)$
4: CALCULATE(E_{max}) $(Eqn. 6)$
5: $E \leftarrow 1$
6: **while** $E \leq E_{\text{max}}$ **do**
7: $\quad RAS \leftarrow \{C_e, M_e, E\}$
8: $\quad RASL \leftarrow RASL + RAS$
9: $\quad E \leftarrow E + 1$
10: **end while**
11: **return** $RASL$

To submit an application to a Spark cluster, a RAS need to be specified as a limit on the possible cores per executor (C_e), memory per executor (M_e) and total executors (E) per application. In order to generate the application profiles, we need to run the application with different RAS and input/iteration. Algorithm 2 shows a simple Resource Allocation Scheme (RAS) generation technique which is used by the Profiler module. To generate the possible resource allocation schemes, knowledge on the total amount of cluster resources is needed. As previously noted, we assume that all the worker nodes (VM from cloud perspective) are homogeneous in a Spark cluster. Therefore, as an input to our algorithm, the total number of worker nodes and only the configuration of a single worker node is given. At first the algorithm finds M_e and E_{max} values. In the next part of the algorithm (line 5 to line 8), the

total number of executors per application is varied to generate different RAS. All the generated RAS are added to a list called Resource Allocation Scheme List (RASL).

dSpark can be installed as a small plug-in to the master node of an Apache Spark cluster. First, the user needs to specify any required configurations in dSpark. Then, the user should submit the applications directly to dSpark instead of the cluster. After selecting the RAS for an application, dSpark automatically submits the application to the production cluster with the selected RAS.

VI. Performance Evaluation

A. Implementation

We have used Java programming language to develop the proposed framework. We have implemented the *Spark-Profiler* module to profile any spark application with a given input size and a RAS. This module uses SparkLauncher Java API [21] to submit applications to the cluster. After an application finishes its execution, a sub-module called LogParser is used to parse the logs in the master node to retrieve the completion time of that application. We have implemented *Resource Allocator* as a separate module and it controls the *Spark-Profiler* module. At first this module reads the configuration files to get the information about the cluster resources. As discussed in Algorithm 1, this module implements both *Application Completion Time Prediction Model* and the *Resource Allocation Model* as two different procedures. To build up the application completion time prediction model, we have applied curve-fitting tools from Apache Common Maths Library [22]. For solving the constrained minimization problem in our resource allocation model, we haved used JOptimizer Library [23].

B. Experimental Setup

1) Cluster Configuration: We have deployed an experimental Apache Spark cluster on Microsoft Azure Virtual Machines (VM). For the master node, we have chosen "standard D4" size VM instance which has 8 cores and 28 GB memory. We have made two (2) worker nodes with "standard D5v2" size VM instance each having 16 cores and 56 GB memory. For storage, we have created an Azure Storage Account to deploy a shared storage device mounted in all the VMs. The replication option chosen for this storage was "Locally redundant storage (LRS)". In LRS, data is replicated three times within a single data center which is located in a single region. All the volumes and VMs were created in the "Australia South-East" region. We have installed Ubuntu Server Version 16.04 LTS in all the nodes and installed Apache Spark Version 2.0.1 on top of it. In addition, we have utilized the standalone cluster manager that comes by default with Apache Spark. We have kept 15 CPU cores and 45 GB of memory of a VM for each worker node. For OS specific daemons and other application programs, we have left the rest of the CPU cores and memory. In our experiments, we have defined the C_e value to be five (5) which is recommended by the Spark developers because using large number of cores in a single executor results bad I/O throughput and having more executors each with fewer cores results in high garbage collection (GC) and scheduling overhead. However, this value can be configured in dSpark by the user if required. The price (P_{vm}) of each "standard D5v2" instance was \$0.0795 AUD at the time of the experiments.

2) Benchmarking Applications: We have used BigDataBench [24], a big data benchmarking suite to evaluate the performance of our proposed models. We have chosen three different types of applications. These are: (1) *WordCount:* compute intensive application, (2) *Sort:* memory and compute intensive application and (3) *PageRank:* iteration based shuffle intensive application.

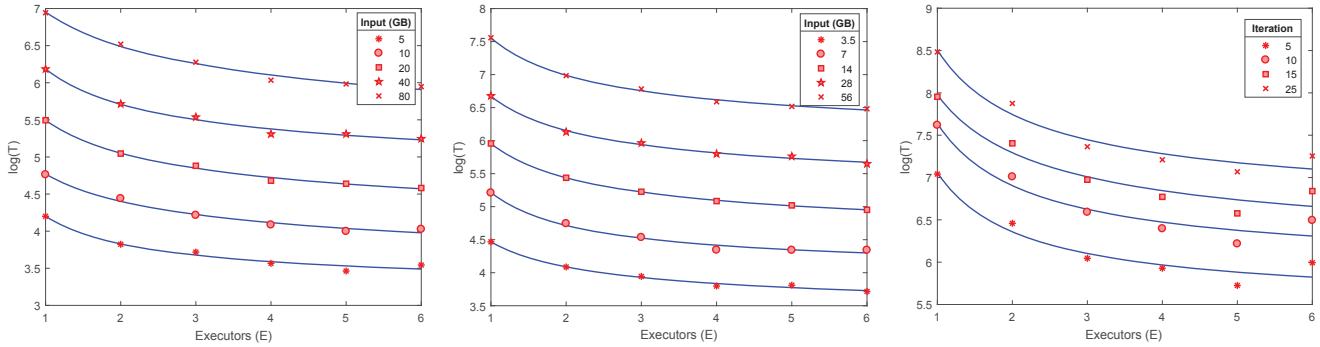
3) Application Profiles: We have used the Spark-Profiler module to collect application profiles for all the benchmarking applications. For WordCount application, we have collected application profiles for 5 GB, 10 GB, 20 GB, 40 GB and 80 GB of input workloads. For Sort application we have collected application profiles for 3.5 GB, 7 GB, 14 GB, 28 GB and 56 GB of input workloads. Lastly, for PageRank application, we have collected application profiles for 5, 10, 15, 20 and 25 iterations for the same 4 GB input graph. We have built the application completion time prediction model as discussed in section III.B and calculated all the coefficients of Eqn. 13.

C. Analysis of Results

1) Accuracy of Application Completion Time Prediction: Fig. 3 shows E vs T curves for three (3) different applications: WordCount (3a), Sort (3b) and PageRank (3c). For WordCount and Sort, we have used different size of application inputs. For PageRank application, we have considered different iterations on the same input graph. From these graphs, it can be observed that there is a decrease in execution time when the number of executors is increased. However, the decrease in execution time is steeper upto 3 or 4 executors. After this point, the execution time does not decrease significantly even if more executors are used for an application. Due to some performance limiting factors like: data serialization/de-serialization, network I/O and shuffle operations, this behaviour was seen from the applications. As all the curves shown in Fig. 3 follows a steady power model, it validates our assumption of using power models to establish the relationship between executor (E) and application completion time (T). While we built up the application completion time prediction model, we have found steady power models for the input vs coefficient graphs.

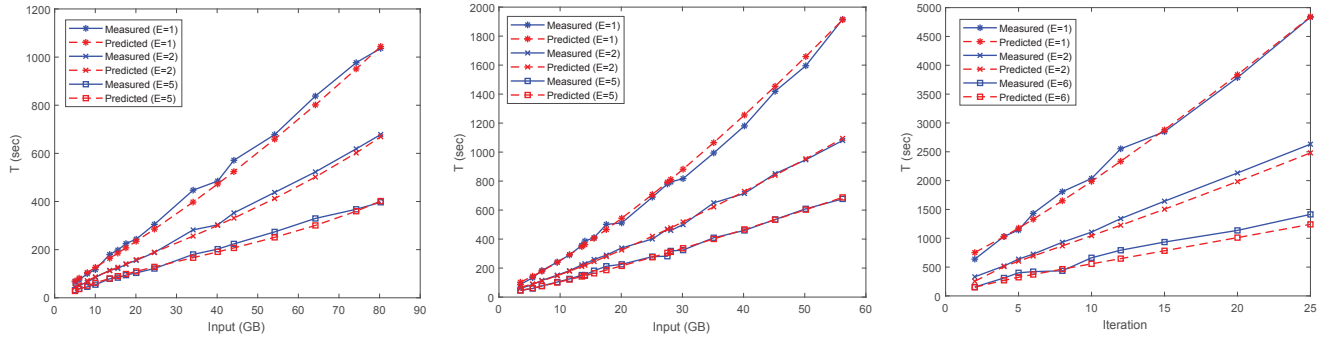
Fig. 4 illustrates the difference between the predicted completion times and the measured completion times of three (3) different applications. From all these graphs, it can be clearly seen that the predicted completion time curves fall closely to the measured completion time curves. We have computed the relative error $RE = (T_{predicted} - T_{measured})/T_{measured}$. We got a mean RE of 5%, 3% and 8% for WordCount, Sort and PageRank applications respectively. As our proposed model has a lower mean RE values for all the experimented applications, it can be used with the resource allocation model.

2) Cost Analysis: Fig. 4 compares the cost of running applications between the proposed resource allocation model and the default Spark resource allocation. We have measured the



(a) WordCount (b) Sort (c) PageRank

Fig. 3. Application completion time modelling

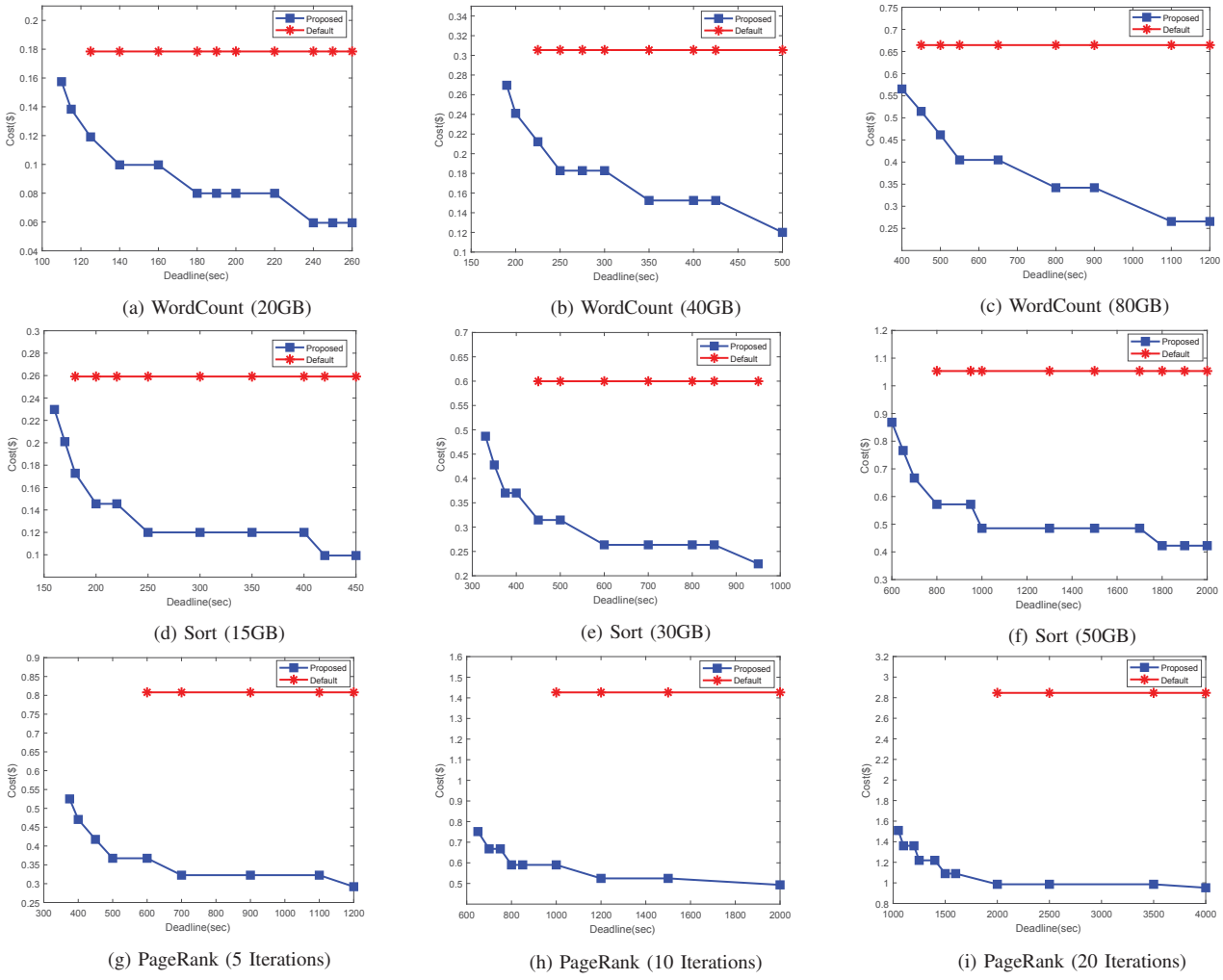


(a) WordCount (b) Sort (c) PageRank

Fig. 4. Accuracy of application completion time prediction for different applications

cost for both approaches with various user-specific deadlines. Fig. (5a-5c) illustrates cost comparison of the WordCount application with 20 GB, 40 GB and 80 GB inputs respectively. Fig. (5d-5f) illustrate cost comparison of the Sort application with 15 GB, 30 GB and 50 GB inputs respectively. Lastly, Fig. (5g-5i) illustrate cost comparison of the PageRank application with 5, 10 and 20 iterations respectively. As seen from all these graphs, cost for running application in the default approach shows a horizontal line in the all cases. In the default approach, each application uses all the resources in the whole cluster. Therefore, even for various user-specified deadlines, it gives the cost of using all the resources. However, our proposed model tends to use more resources only when an application has a strict deadline. In this case, the model utilizes more resources to meet the deadline thus costs higher. When the deadline starts to become more flexible, our model uses a small set of resources to meet the deadline and reduces the cost significantly.

3) Resource Usage Analysis: Fig. 6 compares the resource usage between the proposed approach and the default approach. In our models, we have considered Executors (E) as a chunk of resource as the actual VM resources (CPU cores, memory) are distributed among the executors. However, the size of the executors used by both of these approaches are not the same. In default resource allocation technique, only one (1) executor is launched in each worker node. Therefore, in our experimented cluster, the default approach makes two (2) executor each having fifteen (15) CPU cores. However, in our proposed approach, the cores per executor (C_e) value is flexible and can be tuned according to the application needs. As mentioned before, in our experimental setup, a developer recommended value is used for (C_e). Therefore, we compare the default and proposed approach in terms of CPU cores usage per application. Memory consumption is not shown because we evenly associated all the memory in a VM with the CPU cores. Therefore, higher number of CPU cores usage reflects high amount memory usage. Fig. (6a-6c) compares CPU cores usage of WordCount application for different size of input workloads. In addition, Fig. (6d-6f) compares CPU cores usage of Sort application for different size of input workload. Lastly, Fig. (6g-6i) compares CPU cores usage of PageRank application for different iterations of the same input graph. It can be observed from these graphs that, in all cases, default approach uses all the CPU cores available in the whole cluster to run an application. As we have total 30 CPU cores in the whole cluster, in default approach, all the applications have used 30 CPU cores. Variations in the user-specific deadline does not change resource usages for default resource allocation. However, in the proposed approach, our resource allocation model tries to meet user-specific deadline for an application. If it is possible to use less resources to meet the user-specific deadline, to minimize cost



(a) WordCount (20GB) (b) WordCount (40GB) (c) WordCount (80GB)

(d) Sort (15GB) (e) Sort (30GB) (f) Sort (50GB)

(g) PageRank (5 Iterations) (h) PageRank (10 Iterations) (i) PageRank (20 Iterations)

Fig. 5. Cost of resource usages by the proposed and the default approach for different applications

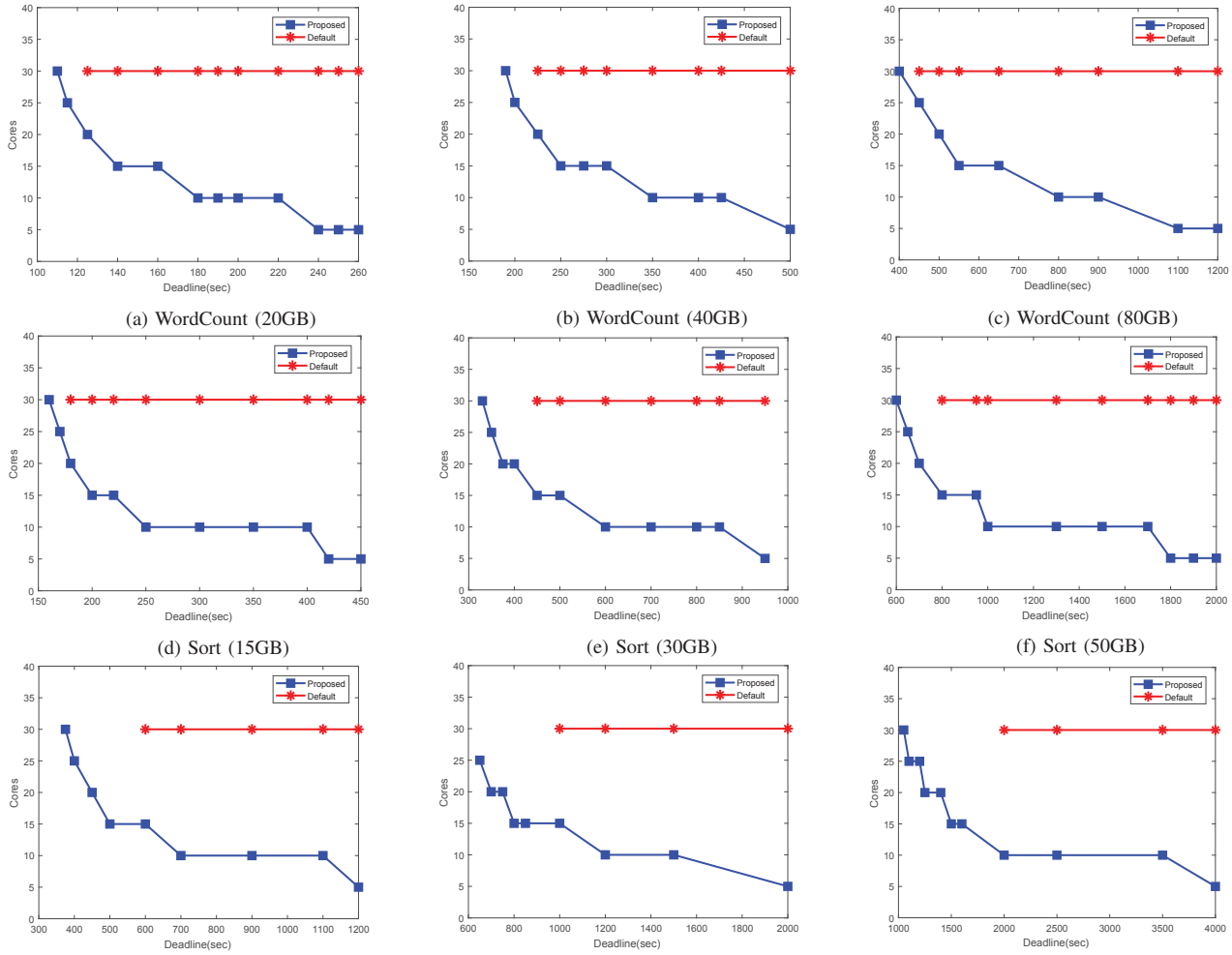
and resource usages, our model selects that resource allocation scheme. Therefore, for the applications with strict deadlines, we observe high resource usages and for the applications flexible deadlines, we see less resource usages.

From both Fig. 5 and Fig. 6 it can be observed that our model handles both strict and flexible deadlines better than the default approach. As we discussed before, using large executors poses performance overheads on the applications. Therefore, the default approach shows poor performance and more deadline violation occurs with strict deadlines. In both of our analysis, we did not include the initial profiling cost as it needs to be done only once for each application.

VII. Conclusions and Future Work

Distributed, large-scale processing of big data has a significant impact on both research and industry. Apache Spark is becoming more popular as a cluster computing engine due to its high-speed data processing capability, extensive applicability in various domains and wide-range of high level APIs. To support user-specific SLA requirements and to maximize an Apache Spark cluster utilization, our research focuses on proposing a cost-effective resource allocation model. The aim is to allow the user a way of automatic and efficient deployments of applications in a local or cloud cluster. We have developed a profiler for Spark which can be used to profile an application in the real cluster in terms of different resource allocation schemes and input workloads. Moreover, we have developed a light-weight resource allocation framework called dSpark that can be plugged into the master node of an Apache Spark cluster. Applications can be submitted to dSpark instead of directly submitting to the cluster. Based on the application profiles received from the profiler, dSpark uses the proposed resource allocation model to select a deadline-based cost-effective resource allocation scheme to deploy an application to the cluster.

We have conducted experiments to evaluate the efficiency of our proposed models. In addition, we have shown the accuracy of the application completion time prediction model for three (3) different applications. The mean relative error in the prediction model was less than 7% for different types of



(a) WordCount (20GB) (b) WordCount (40GB) (c) WordCount (80GB)

(d) Sort (15GB) (e) Sort (30GB) (f) Sort (50GB)

(g) PageRank (5 Iterations) (h) PageRank (10 Iterations) (i) PageRank (20 Iterations)

Fig. 6. Comparison of resource usages between the proposed and the default approach

applications. Furthermore, we have evaluated the effectiveness of the resource allocation model in terms of cost and resource usage and compared the results with the default resource allocation approach in Spark. We have showed that our model selects cost-effective resource allocation schemes that effectively handles various user-specific deadlines. In addition, unlike some existing works, dSpark does not require the users to specify application types as it would be difficult for an end-user to have proper understanding of the application to determine it's type.

As our application completion time prediction model is built by using knowledge from the application profiles, the accuracy of this model depends on the intensity of application profiling. The accuracy of this model increases with a higher number of application profiles. Therefore, there is a clear trade-off between model accuracy and the level of profiling. However, application profiles can be made from past application runs to reduce profiling overhead.

We have assumed that all the worker nodes of the cluster are homogeneous. To accommodate heterogeneous worker nodes in the cluster, the methods for finding the maximum possible number of executors need to be changed and we plan to do this in our upcoming work. Furthermore, we plan to develop an application-level scheduler for Apache Spark. Determining effective resource allocation schemes of a big data application is the first step towards SLA-oriented scheduling of multiple big data applications. dSpark can be used to build the knowledge of resource demands of an application under varying SLA. Therefore, knowledge acquired from dSpark can be used with the application-level scheduler.

ACKNOWLEDGEMENTS

We are indebted to Xunyun Liu for his numerous insightful feedbacks that helped shape this work. We are also grateful to Minxian Xu and Dr. Adel Nadjaran Toosi for their useful suggestions.

REFERENCES

[1] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi,

J. Gonzalez, S. Shenker, and I. Stoica, "Apache spark: A unified engine for big data processing," *Communications of the ACM*, 2016.

[2] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, Washington, DC, USA, 2010.

[3] L. George, *HBase: the definitive guide: random access to your planet-size data*. " O'Reilly Media, Inc.", 2011.

[4] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, 2010.

[5] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, San Jose, CA, 2012.

[6] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, 2008.

[7] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th ACM Annual Symposium on Cloud Computing*, Santa Clara, California, 2013.

[8] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, Boston, MA, USA, 2011.

[9] A. Verma, L. Cherkasova, and R. H. Campbell, "Aria: automatic resource inference and allocation for mapreduce environments," in *Proceedings of the 8th ACM International Conference on Autonomic Computing*, Karlsruhe, Germany, 2011.

[10] Z. Zhang, L. Cherkasova, and B. T. Loo, "Performance modeling of mapreduce jobs in heterogeneous cloud environments," in *Proceedings of the 6th IEEE International Conference on Cloud Computing (CLOUD)*, Santa Clara, CA, USA, 2013.

[11] A. Verma, L. Cherkasova, and R. H. Campbell, "Resource provisioning framework for mapreduce jobs with performance goals," in *Proceedings of the 12th International Middleware Conference*, Lisbon, Portugal, 2011.

[12] A. Verma, L. Cherkasova, V. S. Kumar, and R. H. Campbell, "Deadline-based workload management for mapreduce environments: Pieces of the performance puzzle," in *IEEE Network Operations and Management Symposium*, 2012.

[13] A. Gupta, W. Xu, N. Ruiz-Juri, and K. Perrine, "A workload aware model of computational resource selection for big data applications," in *Proceedings of the IEEE International Conference on Big Data*, Washington, DC, USA, 2016.

[14] R. Tous, A. Gounaris, C. Tripiana, J. Torres, S. Girona, E. Ayguadé, J. Labarta, Y. Becerra, D. Carrera, and M. Valero, "Spark deployment and performance evaluation on the marenostrum supercomputer," in *Proceedings of the IEEE International Conference on Big Data*, Santa Clara, CA, USA, 2015.

[15] P. Petridis, A. Gounaris, and J. Torres, "Spark parameter tuning via trial-and-error," *CoRR*, 2016.

[16] G. Wang, J. Xu, and B. He, "A novel method for tuning configuration parameters of spark based on machine learning," in *Proceedings of the 18th IEEE International Conference on High Performance Computing and Communications (HPCC)*, Sydney, Australia, 2016.

[17] A. Gounaris, G. Kougka, R. Tous, C. Tripiana, and J. Torres, "Dynamic configuration of partitioning in spark applications," *IEEE Transactions on Parallel and Distributed Systems*, 2017.

[18] G. P. Gibilisco, M. Li, L. Zhang, and D. Ardagna, "Stage aware performance modeling of dag based in memory analytic platforms," in *Proceedings of the 9th IEEE International Conference on Cloud Computing (CLOUD)*, San Francisco, USA.

[19] K. Wang and M. M. H. Khan, "Performance prediction for apache spark platform," in *Proceedings of the 17th IEEE International Conference on High Performance Computing and Communications*, New York, USA, 2015.

[20] S. Sidhanta, W. Golab, and S. Mukhopadhyay, "Optex: A deadline-aware cost optimization model for spark," in *Proceedings of the 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, Cartagena, Colombia, 2016.

[21] "Sparklauncher java api," https://spark.apache.org/docs/latest/api/java/index.html?org/apache/spark/launcher/package-summary.html, accessed: 2017-06-18.

[22] "The apache commons mathematics library," http://commons.apache.org/proper/commons-math/, accessed: 2017-06-18.

[23] "Joptimizer," http://www.joptimizer.com/l, accessed: 2017-06-18.

[24] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang *et al.*, "Bigdatabench: A big data benchmark suite from internet services," in *Proceedings of the 20th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Barcelona, Spain, 2014.