Performance Prediction of Cloud-Based Big Data Applications

Danilo Ardagna,
Enrico Barbierato,
Athanasia Evangelinou,
Eugenio Gianniti,
Marco Gribaudo
Dipartimento di Elettronica e Informazione
Politecnico di Milano
Milano, Italy
name.lastname@polimi.it

Túlio B. M. Pinto,
Anna Guimarães,
Ana Paula Couto da Silva,
Jussara M. Almeida
Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
Belo Horizonte, Brazil
tuliobraga@dcc.ufmg.br,anna@dcc.ufmg.br,
ana.coutosilva@dcc.ufmg.br,jussara@dcc.ufmg.br

ABSTRACT

Big data analytics have become widespread as a means to extract knowledge from large datasets. Yet, the heterogeneity and irregularity usually associated with big data applications often overwhelm the existing software and hardware infrastructures. In such context, the flexibility and elasticity provided by the cloud computing paradigm offer a natural approach to cost-effectively adapting the allocated resources to the application's current needs. However, these same characteristics impose extra challenges to predicting the performance of cloud-based big data applications, a key step to proper management and planning. This paper explores three modeling approaches for performance prediction of cloud-based big data applications. We evaluate two queuing-based analytical models and a novel fast ad hoc simulator in various scenarios based on different applications and infrastructure setups. The three approaches are compared in terms of prediction accuracy, finding that our best approaches can predict average application execution times with 26% relative error in the very worst case and about 7% on average.

1 INTRODUCTION

Nowadays, the big data adoption has moved from experimental projects to mission-critical, enterprise-wide deployments providing new insights, competitive advantage, and business innovation [11]. IDC estimates that the big data market grew from $3.2 billion in 2010 to $16.9 billion in 2015 with a compound annual growth rate of 39.4%, about seven times the one of the overall ICT market [2].

Key properties characterizing big data applications are high volumes of data and increasing heterogeneity and irregularity in data access patterns. Such properties impose challenges to the hardware and software infrastructure. On the other hand, cloud computing infrastructures have become a versatile computing platform as cloud resources fit application requirements, as they are needed, leveraging the elastic nature of the cloud. Thus, big data applications find in cloud-based infrastructures a natural hosting platform to cost-effectively provisioning the necessary resources to their execution. For example, 61% of Spark adopters ran their applications on Cloud last year with a 20% increase with respect to 2015 [1].

However, though flexible, the shared infrastructure that powers the Cloud together with the natural irregularity of big data applications may impact the predictability of cloud-based big data jobs. As applications based on big data become an integral part of our daily lives, predictability, robustness, and dependability become primary requirements. Accurate performance prediction of an application is a key step to both planning and managing: it is a key component to drive the automatic system (re-)configuration so as to properly meet the applications' dynamic needs and avoid Service Level Agreement (SLA) violations.

To address the challenges of predicting the performance of big data applications, we here explore three different modeling approaches to estimate, given the available resources, the *average execution time* of a target application. In other words, given a target application, specified by a directed acyclic graph (DAG) representing the individual tasks and their parallelism and dependencies, the purpose is to predict how long it will take for the application to run (on average) given a target resource allocation (e.g., numbers of cores, nodes). The focus concerns applications running on Spark (http://spark.apache.org/), which is a fast and general engine for large-scale data processing whose adoption has steadily increased and which probably will be the reference big data engine for the next 5–10 years [10].

Firstly, we investigate the use of analytic queuing network (QN) models for predicting the performance of Spark applications. QN models offer good abstractions to capture the main points of resource contention, which in turn contribute to longer delays and waiting times in the various components (e.g., nodes) of the system. However, capturing the delays due to synchronization between individual tasks of a parallel application in a QN model is a non-trivial task as model complexity greatly increases, hurting efficiency.

Two QN modeling approaches are explored here. One is based on a simple upper-bound on the average execution time for Fork-Join queuing networks, proposed by Nelson and Tantawi [19], which depends only on the number of parallel tasks and on the average execution time of a single task. We refer to this model as *Fork-Join*. Since such approach demonstrated to provide limited accuracy, a more sophisticated QN model proposed in [16], which extends an approximated Mean Value Analysis (AMVA) technique by modeling the precedence relationships and parallelism between individual tasks of the same job, is considered. This model here referred to as

ICPE 2018, April 2018, Berlin, Germany
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Task Precedence model, explicitly captures the overlap in execution times of different tasks of the same job, thus potentially being more general, though also more complex.

In addition to the aforementioned QN approaches, complex systems can be modeled by exploiting other formalisms or modeling languages such as Stochastic Petri Nets (SPNs). Specifically, SPNs provide a set of primitives to easily model the synchronization of stages belonging to a DAG. Inspired by this formalism, *dagSim* was implemented as an ad-hoc and fast discrete event simulator to model the execution of complex DAGs. The novelty introduced by dagSim is twofold: on one side, the simulation process achieved great accuracy within a short timescale with respect to other formalisms (such as SPNs) or specific tools (such as JMT [6] or GreatSPN [8]), whose execution time resulted less satisfactory when using the same models considered by dagSim. Furthermore, the flexibility and modularity of the tool allow the implementation of future enhancements such as the computation of percentiles, which cannot be provided by the *Task Precedence model*.

We evaluate the three modeling approaches in seven scenarios consisting of different virtual machine environments and applications, considering both SQL like workloads and machine learning benchmarks. For each scenario, we use our models to estimate the application average execution time for different resource configurations.

The results indicate that the Fork-Join model is not suitable for predicting Spark application response time. This is evidenced by the high prediction errors found during experimentation. Vice versa, we found a good overall accuracy for average system response time prediction through the Task Precedence model and dagSim simulator (below 17% relative error in every considered scenario). Both models presented similar performance, whereas dagSim performed better for interactive queries while the Task Precedence model performed better for iterative ML algorithms.

The rest of this paper is organized as follows. Section 2 presents related work, discussing the efforts against predicting Spark application response time as well as detailing both queuing network models and simulation approaches related to the present research. Section 3 describes Spark characteristics and model assumptions relevant to understand our work. It also details the Fork-Join and Task Precedence models, and the dagSim Simulator. Section 4 introduces the experimental scenarios we explored, further showing and discussing the results obtained for the three prediction approaches. Section 5 comes up with our research discussions and conclusion.

2 RELATED WORK

The QoS study of Big Data applications in the cloud can be addressed from different perspectives.

Recently, sophisticated projects have emerged in the study of Spark applications performance, such as PREDIcT [21] and RISE-2016 [12]. PREDIct is a tool including a set of prediction techniques for different areas of data analytics, while RISE2016 is a collection of scalable performance prediction techniques for big data processing in distributed multi-core systems. More specifically, the aim is to study distributed computer clusters and their parallel applications, spanning performance hybrid models (which include both machine learning and traditional analytical models like QN) and optimization solutions related to hardware configurations. In [30], the authors provide hierarchical models, which leverage the multi-stage execution structure of Apache Spark jobs, and they are able to obtain good accuracy generalizing the measurements performed on a fraction of the real application data set.

This article focuses on the use of modeling techniques to guide the initial configurations or to enable the analysis of the viability of jobs. The most relevant related work is subdivided into two parts, specifically i) analytical queuing network methods and ii) simulation approaches.

2.1 Queuing Network Models

Applications running in parallel systems have to share physical resources (processors, memory, bus, etc.). Competition for computational resources can occur among different applications (inter-application concurrency) or among tasks of the same application (intra-application concurrency). Given system resource limitation, performance analysis techniques are important for studying fundamental performance measures, such as mean response time, system throughput, and resource utilization. In this context, queuing networks have been successfully used for studying the impacts of the resource contention and the queuing for service in the applications running on top of parallel systems [16, 24, 25, 27].

The parallel execution of multiple tasks within higher level jobs is usually modeled in the QN literature with the concept of fork/join: jobs are spawned at a fork node in multiple tasks, which are then submitted to queuing stations modeling the available servers. After all the tasks have been served, they synchronize at a join node. Unfortunately, there is no known closed-form solution for fork-join networks with more than two queues, unless a special structure exists [15].

The authors in [19] present a model for predicting the response time of homogeneous fork/join queuing systems. The observed system is made up of a cluster of *homogeneous* index servers, each holding portions of querible data, and the query requests to the index servers go in a FCFS (First-Come First-Served) scheduling queuing discipline. In order to represent system parallelism, the index server subsystem is modeled as a fork-join network. In this model, an incoming task is split (forked) into identical subtasks, which are then sent to individual servers and executed in parallel, independently from one another. Once all subtasks have finished executing, they are joined and the task execution is completed. The average response time is determined by the slowest server.

Following the fork-join model paradigm, the authors in [27] present an analysis of closed, balanced fork-join queuing networks, in which a fixed number of identical jobs circulate. They introduce an inexpensive bounding technique referred to as balanced job bounds for fork-join systems (BJB-FJ), which is analogous to balanced job bounds developed for product form networks. Servers have a FCFS queuing discipline and exponentially distributed service times. Based on Markov models theory, authors provide accurate approximation results for the job response time.

In the same direction, [24] models a multiprocessing computer system as K homogeneous servers, each with an infinite capacity queue. Jobs arriving to the system are split into K independent tasks, each of which is assigned to a server. The authors provide a

computationally efficient algorithm for obtaining upper and lower bounds on the expected response time of this system. Moreover, the algorithm guarantees an error bound and, if one desires, tighter error bounds can be obtained at the cost of more computation.

The work in [16] also considers the issue of estimating performance metrics in parallel applications. The proposed method is computationally efficient and accurate for predicting performance of a class of parallel computations, which can be modeled as task systems with deterministic precedence relationships represented as series-parallel DAGs. Tasks are represented as nodes and edges mark precedence relationships between pairs of nodes. A task can be executed once its parent tasks have finished executing. Furthermore, whenever nodes are independent, their executions may overlap fully or partially, according to resource availability. This overlap can be determined from the start and ending times of task executions. The amount of overlap between tasks can then be used to reduce the initial task DAG and successively estimate task response times, ultimately leading to an estimate of the full application response time. While the models proposed in [19, 24, 27] assume a fork-join abstraction to represent parallel behavior, here the authors focus on the precedence relationships resulting from tasks that must run sequentially, combined with those that may run in parallel. An extension of this model, capturing not only intra-job, but also inter-job overlap to evaluate application response times, is presented in [25].

In our work, we apply the models proposed by the authors in [19] and [16], given that the parameters of both models are easily obtained (for instance, service demands and task structure) and results are obtained with low complexity cost. More details on [19] and [16] models are presented in Sections 3.2.1 and 3.2.2.

2.2 Simulation approaches

Classical simulation methods consist of discrete event, continuous, and Monte Carlo techniques.

A simulation based on discrete events (e.g., a number of customers waiting for a service) takes advantage of a mathematical model of a system changing its state in a (simulated) time. The system state is measured only at certain fixed regular time instants. On the other hand, in classical continuous simulation, models of physical systems are described through differential equations and simulated in a continuous way: the representation of the relationships occurring between states and time are not necessarily explicit and the system state varies continuously in time. Finally, Monte Carlo methods model degrees of uncertainty where representing time is not requested. The idea is to approximate an integral by taking a well-known geometrical shape and circumscribing a specific region with it.

Literature presents a wide range of simulation software tools based on stochastic formalisms varying in complexity and capabilities, including the valuable possibility to share data and communicate with other applications.

Petri nets (PN, presented by Carl Adam Petri in his Ph.D. dissertation, see [22]) represent an intuitive and powerful analysis technique, being applicable to a wide range of scenarios. Augmented PN models considering a temporal specification, i.e., including transitions firing with a probabilistic delay, identify an extended PN class denoted by the term Stochastic Petri Net (SPNs). The simulation tool proposed in this paper is inspired to SPNs, since they provide a set of primitives to easily model the synchronization of stages belonging to a DAG.

Several tools to study the behavior of SPNs have been implemented. Examples include the Stochastic Petri Net Package (SPNP, presented in [20]) and GreatSPN [8]. GreatSPN supports the analysis of Generalized Stochastic Petri Nets (GSPNs) including both immediate (the fire event occurs immediately) and timed (the fire event occurs within a stochastic time) transitions and of Stochastic Well-Formed Nets (SWNs, i.e., Petri nets where the tokens can be distinguished). SMART (Symbolic Model checking Analyzer for Reliability and Timing, [9]) includes both stochastic models and logical analysis. Currently, SMART offers a high level formalism (i.e., PNs) and two low level formalisms (discrete-time and continuous-time Markov chains, i.e., DTMC and CTMC) to the modeler who is free to calculate a set of measures for each model and exchange parameters between models. SHARPE (Symbolic Hierarchical Automated Reliability and Performance Evaluator) is a tool to analyze stochastic models [26], the most notable being fault trees, product form queuing networks, Markov chains, and Generalized Stochastic Petri nets. Finally, JMT [6] is a suite of applications offering a framework for performance evaluation, system modeling, and capacity planning.

The complexity introduced by applications based on big data has stimulated the scientific community to develop tools and frameworks based on more sophisticated architectures, in order to predict the behavior of these scenarios in a timely and efficient manner.

The problem of studying the performance prediction of individual jobs is explored in [23] through a framework consisting of a Hadoop job analyzer, while the prediction component exploits locally weighted regression methods. A similar issue is studied in [28] by using instead a hierarchical model including a precedence graph model and a queuing network model to simulate the intra-job synchronization constraints. In [7], the authors consider the problem of minimizing the cost involved in the search of the optimal resource provisioning, proposing a cost function that takes into account: i) the time cost, ii) the amount of input data, iii) the available system resources (Map and Reduce slots), and iv) the complexity of the Reduce function for the target MapReduce job. The usage of a simulator to better understand the performance of MapReduce setups is described in [29] with particular attention to i) the effect of several component inter-connect topologies, ii) data locality, and iii) software and hardware failures.

Finally, the authors in [4] describe multiple queuing network models (simulated with JMT) and stochastic well formed nets (simulated with GreatSPN) to model MapReduce applications, highlighting the tradeoffs and additional complexity required to capture system behavior to improve prediction accuracy. As a result, general purpose simulators such as GreatSPN and JMT are not suitable to study efficiently massively parallel applications introducing tens (or even hundreds) of stages and thousands of parallel tasks for each stage.

3 PERFORMANCE PREDICTION MODELS

This section presents the three modeling approaches analyzed in this paper to predict the performance of cloud-based big data applications, namely Fork-Join, Task Precedence, and dagSim. Since our main focus is on applications running on Spark, we start by first presenting some key components of this framework, highlighting some assumptions behind its parallel execution model that may affect the performance models (Section 3.1). The considered queuing network models (Section 3.2) are then discussed as well as the proposed dagSim discrete event simulator built to analyze the performance of Spark applications (Section 3.3).

3.1 Spark Overview and Model Assumptions

Spark is a fault-tolerant cluster computing framework that provides a set of abstractions for parallel computations across distributed nodes with multiple cores. It is a fast and general purpose engine for large-scale data processing and it was first proposed as an alternative to Hadoop MapReduce [31]. Spark is the state-of-art for fault-tolerant parallel processing and it recently became popular for big data processing on the cloud [1].

The general unit of computation in Spark is an application. It may be composed by a single job, multiple jobs, or a continuous processing. A job is composed by a set of data transformations and terminates with an action requesting a value from the transformed data. Each transformation represents a specific piece of code that launches data-parallel tasks on read-only data divided into blocks of almost equal size, called partitions. This set of same class tasks is called stage. Within a stage, a single task is launched for each data partition, thus the number of tasks inside the stage is equal to the number of partitions. During the stage runtime, each core (also called CPU slot) can run only a single task at a time. Since cores are a limited resource, the tasks are assigned to CPU slots until all resources become busy. Thus, the remaining tasks are enqueued and scheduled to be executed as soon as the cores become available.

The Spark execution model is represented by a DAG. Considering a logical plan of transformations that is fired by an action, the Spark *DAGScheduler* constructs a DAG of stages and their precedence relations. The stages are submitted for execution as a set of tasks that follows FCFS policy. The *TaskScheduler* does not know the dependencies between stages. Each stage is a fully-independent sequence of tasks that can run right away based on the data that is already on the cluster [13]. Thus, only stages have precedence relationships and these are represented by the DAG.

The key idea of this work is to apply a set of performance prediction techniques to estimate the execution time of Spark applications and evaluate their effectiveness. The issue of performance prediction in parallel systems has been approached in several ways, with varying degrees of detail, cost, and accuracy. Focusing on such data-parallel framework based on a DAG execution model, one of the main concerns is to model the synchronization step that happens when a stage terminates. Then, the models for calculating performance measures have to take into account how the executions of stages overlap among themselves.

In this work, we made the following assumptions for all the three (analytical and simulation) analyzed models: i) the concurrent system is modeled as a closed queuing model, with a single application that splits into one or more Spark jobs, ii) jobs are sequentially scheduled and comprehend one or more stages, iii) multiple stages may run in parallel or may have some precedence relationships, iv) a stage is composed by tasks of the same class with no precedence relationship among themselves (i.e., they may run in parallel), v) an individual application obtains dedicated resources for its execution (i.e., VMs that are executed on a cloud cluster), vi) resources (such as memory, CPU, disk) are homogeneous (as frequently happens in cloud deployments, see, e.g., [18]). Moreover, queuing network models are based also on the assumption that the task demand times on each resource (i.e., the time required to process a single task) are exponentially distributed.

3.2 Queuing Network Models

This section describes the two queuing network models explored in this paper. We first present the Fork-Join model, which relies on a known approximation of response time for parallel applications, and then briefly describe the Task Precedence model, which takes as input a DAG representation of the parallel application.

3.2.1 Fork-Join Model [19]. This model provides a very simple upper-bound on the average execution time for queuing networks with fork-join synchronization. The main idea is as follows: the cluster architecture is represented as a two-level graph in which the leaves represent the execution nodes, modeled as Spark Worker cores, and the root represents the unit that controls the execution flow, modeled as the Spark Master node. Stages are forked into same class tasks and scheduled to execute in parallel across the available cores. After execution at all worker cores, the results from each unit are joined.

The model expects as input the number of execution nodes in the graph, that is, the total number of cores available through the worker nodes, and the average execution time at an individual core, which can be estimated based on historic data (i.e., logs of previous executions of the same Spark application). It outputs two values, namely a lower and an upper bound. The lower bound is the execution time at a single core, here referred to as R_{core}, and reflects the application execution time when there are no synchronization delays caused by discrepancies across individual cores. In the present case, this lower bound is not of interest, for prediction purposes, as it is one of the model inputs. However, R_{core} could be estimated based on some finer grained modeling strategy (e.g., Mean Value Analysis). The upper bound, on the other hand, is our main interest: it provides an approximation that aims to capture the effects of a slower core, which would cause delays and affect the overall execution times.

According to Nelson and Tantawi [19], an upper bound for the execution time of an application running on K cores is given by the product of the average execution time at a single core (R_{core}) and the H_K harmonic number (more details in [19]). Thus, the bounds for the application response time ($R_{\text{application}}$) are:

$$R_{\text{core}} \leq R_{\text{application}} \leq H_K R_{\text{core}},$$

with $H_K = 1 + 1/2 + 1/3 + \cdots + 1/K$.

We note that the fork-join structure assumed by this model may be a very coarse approximation for the execution of Spark applications. In particular it does not explicitly capture any precedence

relationship between stages. Rather it assumes such relationships are indirectly captured in the input (R_{core}). In other words, by looking at historic data, we compute the average total execution time at each individual core, considering all stages and individual tasks of the given application, thus capturing the precedence of individual tasks in a single core. The model, in turn, aims at capturing all synchronization delays across different cores by the inflation factor H_K. Though a coarse approximation, the simplicity of this approach motivated us to assess to which extent it can provide accurate performance predictions for Spark applications.

3.2.2 Task Precedence Model [16] . In this prediction method, the performance of a parallel application is modeled by explicitly capturing the precedence relationships between different blocks of computation. We start by presenting the main ideas behind the model, as proposed in [16]. We refer to the original paper for a detailed derivation of the model. We then discuss how we applied this model to Spark applications at the end of the section.

In the original paper [16], each block of computation was called a task, and the goal was to estimate the average execution time of an application composed by multiple parallel/sequential tasks. The precedence relationships between different tasks are expressed as a series-parallel directed acyclic graph (DAG), where each node is a task. Available resources (e.g., cores) are modeled as service centers in a queuing network model. By exploiting both the queuing network and the DAG, the authors modified a traditional iterative Mean Value Analysis (MVA) approach to account for delays caused by synchronization and resource constraints originated from task precedence and parallelism.

The solution uses a traditional MVA model to estimate the average execution time of each task. In order to explicitly capture the synchronization delays between parallel tasks, the model estimates an *overlap* probability between each pair of tasks based on the input task precedence DAG. This probability captures the chance that the executions of the two tasks overlap in time, and is used as an inflation factor to estimate a new set of task average execution times, according to the MVA equations. The model will continue to iterate over these values until they converge below a given error threshold. As a final step of each iteration, the precedence graph is reduced to determine the average execution time of the whole application. For example, execution times of sequential tasks will be added; execution times of parallel applications will be aggregated according to a probabilistic approach that takes into account the overlap probabilities between them.

Since jobs in Spark are sequentially executed by default, we here apply the model by considering each node in the input DAG as a stage of the Spark application, thus explicitly capturing the dependencies among stages (unlike the Fork-Join model) that exist in Spark applications. Each stage is fully described by its average execution time, which is estimated based on historic data (Spark logs of previous executions of the same application). Thus, the model takes as input the application's DAG and the average execution time of each individual stage and produces as output the average execution time of each job. To estimate the average execution time of the application, the execution times of all jobs are simply added together.

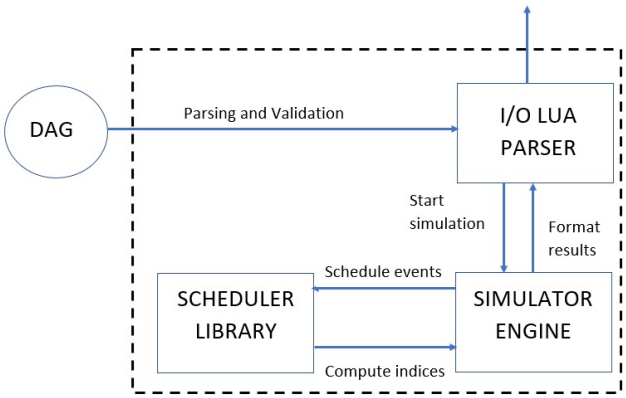

Figure 1: dagSim and its components

3.3 dagSim Simulator

dagSim (see Figure 1) is a high speed discrete event simulator built to analyze DAGs corresponding to MapReduce and Spark jobs[1] and consists of three main components: i) a *parser*, ii) a *simulation component*, and iii) an *output module*.

Models are described with a data driven approach defining the DAG stages and the workload they have to handle. Specifically, a DAG model is defined as a tuple:

$$DAG = (S, N_{\text{Nodes}}, N_{\text{Users}}, \mathcal{Z}) \tag{1}$$

Where $N_{\text{Nodes}} \in \mathbb{N}, N_{\text{Nodes}} \geq 1$ represents the number of computational nodes and $N_{\text{Users}} \in \mathbb{N}, N_{\text{Users}} \geq 1$ the number of users concurrently submitting jobs to the system, and $\mathcal{Z}$ is the "think time distribution": the time a user will wait before submitting a new job. Set $S = \{s_1, \ldots, s_{N_{\text{Stages}}}\}$ is the set of stages that define the DAG. Furthermore, each stage $s_i \in S$ is a tuple:

$$s_i = (\texttt{id}, N_{\text{Tasks}}, Pre, Post, \mathcal{T}) \tag{2}$$

where `id` is a symbolic constant assigning a name to the stage, $N_{\text{Tasks}} \in \mathbb{N}, N_{\text{Tasks}} \geq 1$ accounts for the tasks composing the stage, $Pre \in S$ and $Post \in S$ define respectively the stages that must have been completed for s_i to be executable, and the set of stages that will be able to run after the completion of s_i. The probability distribution $\mathcal{T}$ defines the duration of each task of the stage and is obtained from Spark logs.

As a file interchange format to encode the previously formalized model, LUA, a multi-purpose procedural programming language [3], was selected. There are several advantages in using LUA with respect to other alternatives. First, it has a very compact syntax for defining complex structured constants, which allows to initialize complex objects with a short textual overhead. Other languages, such as XML, require a complex markup structure, which in the end makes input files very long and difficult to generate. In our approach, an input model is an instantiation of a set of predefined global variables: thanks to the names used to identify such variables, files are easily readable by a human, and simple to automatically generate by a software component. Next, even if only the instances of the global variables are used, model files are effectively LUA

[1]The tool is available at https://github.com/eubr-bigsea/dagSim

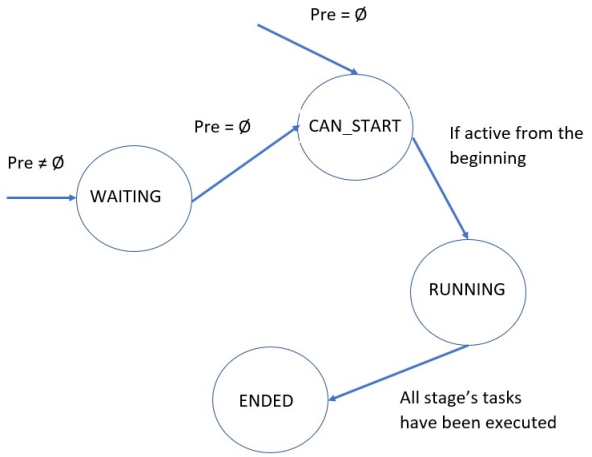

Figure 2: Job finite state machine

programs. This means that they can exploit a full set of instructions and systems commands to algorithmically compose a model by loading pieces from external datasets, computing rates with algebraic expressions (e.g., 1/10000 instead of 0.0001), and use loop constructs to repeat the same assignment several times.

The performance indices computed during the simulation are fed back into LUA variables. The simulator then executes some predefined LUA code that outputs the results stored in such variables. Currently performance indices are displayed as plain text. However, the ability of the user to replace the legacy LUA output code with custom procedures allows to easily integrate the tool within larger frameworks. Further enhancements may include library procedures to support different formats, such as CSV, XML, or HTML.

The simulation engine has been written in the C language. It is based on a classic discrete event simulation algorithm and has been designed for high performance. Though dagSim is a lightweight tool compared to other commercial programs, it targets specifically DAG models. Simulation can run efficiently thanks to a proprietary scheduler library offering data structures that perform well when a high volume of events is generated. The tool is highly portable, since it can be easily recompiled without the requirement of external tools or libraries not supplied with the source code.

When addressing to the DAG's components, the engine refers to a few terms summarized in the following:

- A *job* is a process submitted by a user and is composed of a set of *stages* $\{s_1, s_2, ..., s_n\}$;
- Each *stage* consists in turn of one or more *tasks* $\{t_1, t_2, ..., t_m\}$;
- Each *task* corresponds to an event and is characterized by a *timestamp*, denoting a start time;
- Given a job $\mathcal{J}_i$, Pre_i denotes the set of jobs connected to $\mathcal{J}_i$.
- According to the finite state machine depicted in Figure 2, a *job* can be in one of the following four states:
  - $CAN\_START : Pre_i = \emptyset$;
  - $WAITING : Pre_i \neq \emptyset$;
  - RUNNING: this is the case of a job J_i that was eligible to be executed (its previous state was CAN_START) and is not depending on any other job;
  - ENDED: all the stages in job J_i have been executed.

The core idea of the simulation engine is that each time a stage $s_k \in \mathcal{J}_i$ has been executed, a counter is decremented of one unit. When the counter reaches zero, the engine determines that a stage is completed and which ones are now eligible to start. By using a doubly-linked list storing the relevant information about the jobs in CAN_START state, it is possible to determine which one can be executed without performing a full search on the set of jobs. In this sense, the approach provided by dagSim's engine is original and more efficient with respect to other scheduling mechanisms implemented in JMT [6] or GreatSPN [8].

Specifically, the engine implements Algorithm 3.1 (the most important functions and data structures are summarized, respectively, in Table 2 and 1), which is articulated in different steps:

- Initially (lines 4–6), for each user accessing the system, a doubly-linked list called *UJD* is populated with a set of information, notably i) the number of stages ready to be started, ii) the remaining tasks for each stage that need to be completed, iii) the state of each stage, iv) the start and end time of each stage, and v) a pointer to a list of jobs ready to be started. A Node data structure is initialized at line 9;
- In a similar manner, the following phase (lines 12–16) adds a new event (whose timestamp corresponds to the *think time*) on a *CalendarEvent* structure. Furthermore, a doubly-linked list *JobData* is populated by inserting i) a user identifier, ii) a job and a stage status, and a iii) task identifier. Finally, *JobData* is linked to the event created earlier;
- The most important part of the algorithm consists of a cycle performed for all the available jobs (lines 20–54);
- At line 21, an event is extracted (*pop* operation) from the *CalendarEvent* structure (if the latter is empty, the cycle terminates);
- If the user requests to launch a new job (line 26), both *initUserJobData* and *createReadyList* functions are invoked (lines 27 and 28(: the former initializes all the job's stages to STAGE_ST_CAN_START or STAGE_ST_WAITING depending on whether the stage has incoming arcs or not, while the latter reviews the status for each stage: if the stage is ready to start, its execution begins and its status is updated to STAGE_ST_RUNNING. For each task of the stage, an event is created and stored in an doubly-linked list AUX, representing the events waiting to be executed;
- If there are available computational nodes and no lock has been set (line 29), the *scheduleReadyTasksOnAvailableNodes* function is invoked (line 30) to i) set a lock if the considered node was not engaged and ii) schedule the waiting jobs on available nodes. In the other case, the waiting jobs are temporarily inserted into an auxiliary list (line 32);
- In case that a task of a running job ended (line 36), the corresponding counter of the remaining tasks in a stage is decremented of one unit (line 37). If there are no tasks left (line 38), the stage's state is updated to STAGE_ST_ENDED and the *checkNewStageStart* function is invoked to verify

Table 1: Notable data structures

Data structure	Relevant parameters
UserJobData	- User identifier - Job identifier - Start and End time - Stages that still needs to be started - Jobs that still needs to be completed per stage - State of the stages - Start / End time for the stage - Lists of jobs that can be started
NodeData	- Number of free nodes - User locker - Jobs that still needs to be executed
Job	- User identifier - Job identifier - Stage identifier - Task identifier
sList	- The description of an Event
sAuxlists	Auxiliary List to store Events that are in waiting state

if another stage can start: in case of positive response, the *createReadyList* function is called;

- Line 42 verifies if there are no tasks ready to be executed and no stages to start: in this case, the stage end time and the ended stage counter are updated;
- At the end of the main loop, finally the function *releaseNode* is invoked (line 51), in order to remove the lock taken earlier on the node if the the following conditions are met: i) no more tasks need to be executed, ii) no other user has locked the node, and iii) there are no other stages to start. If all the conditions are met, the lock put by the current user on the node can be released. Furthermore, in case there are still some jobs in waiting state, the first job is removed from the list and *scheduleReadyTasksOnAvailableNodes* is executed on that job.

4 EXPERIMENTAL RESULTS

In this section we present the results of a set of experiments we performed to explore and validate Fork-Join and Task Precedence predictors, and dagSim simulator on the TPC-DS industry benchmark as well as on some reference Machine Learning (ML) benchmarks, namely Support Vector Machines (SVMs), Logistic Regression, and K-Means. Such experiments were performed on Microsoft Azure Cloud. Overall our tests include sequential workloads (obtained from the TPC-DS SQL queries execution plan) and iterative workloads, which characterize ML algorithms and are becoming more and more popular in the Spark community [1].

Algorithm 3.1 Simulation engine algorithm

```
1:  function SOLVE(Model M, Users U, CalendarEvent ce)
2:      /* Initialize user job data */
3:      UserJobData **UJD;
4:      for i = 0; i < size(U); i++) do
5:          UJD[i] = createUserJobData(M);
6:      end for
7:      /* Initialize the Nodes data */
8:      NodeData *ND = initNodeData(M);
9:      /* Initialize Job structure */
10:     for user_i ∈ Users do
11:         nEv = AddEvent( ce, ThinkTime);
12:         JobData *jobData = populateJobData();
13:         nEv->data = jobData;
14:     end for
15:     /* Main simulation loop */
16:     int TotalJobEnded = 0;
17:     while TotalJobEnded < maxJobs(M) do
18:         event = pop(CE);
19:         if event is NULL then
20:             break;
21:         end if
22:         Job *jd = ev->data;
23:         if jd->stageId == USER_WANTS_TO_START then
24:             initUserJobData(jd->userId, M);
25:             createReadyList(jd->userId, M);
26:             if (ND->NfreeNodes > 0) AND (!getLock(ND)) then
27:                 scheduleReadyTasksOnAvailableNodes( ce, ND, UJD[jd->userId]);
28:             else
29:                 addToAux(event, WAITLIST);
30:             end if
31:         else
32:             /*A task of a running job ended */
33:             UJD[jd->userId]->remainingTasksXStage[jd->stageId]- -;
34:             if UJD[jd->userId]->remainingTasksXStage[jd->stageId] ≤ 0 then
35:                 setstatus(s_k), STAGE_ST_ENDED);
36:                 if createReadyList(UJD, M) then
37:                     checkNewStageStart(J_i, M);
38:                     if (UJD[jd->userId]->readyList == ∅ AND UJD[jd->userId]->stagesToStart == 0) then
39:                         UJD[jd->jobId->userId]->endTime = currTime;
40:                         nEv = addEvent(ce, T);
41:                         JobData *jobData = populateJobData();
42:                         nEv->data = jobData;
43:                         TotalJobEnded++;
44:                     end if
45:                 end if
46:             end if
47:             releaseNode(currTime, ce, ND, UJD);
48:         end if
49:     end while
50: end function
```

Table 2: Notable functions

Function	Description
InitNodeData	Populates a NodeData structure
AddEvent	Adds a new event to Calendar Event data structure
populateJobData	Initializes a JobData structure
maxJobs	Returns the maximum jobs number as per the LUA input file
pop	Performs a *pop* operation on a CalendarEvent data structure. The output is an Event (sList data structure)
initUserJobData	Populates JobData structure for a specific user
createReadyList	Populates ReadyList structure including the jobs to be executed and update the job' status.
scheduleReadyTasks OnAvailableNodes	Schedules waiting jobs on available nodes
addToAux	Adds an Event to an Auxiliary data structure
isEmpty	Returns *true* of *false* depending on the list passed as argument is empty or not
getLock	Returns *true* or *false* depending on the job passed as argument has been locked by a user or not
setLock	Locks a job

4.1 Scenarios

The experimental scenarios cover the most widely used applications on Spark [1]. The ML benchmarks, namely K-means, Logistic Regression, and SVM, are core activities in machine learning applications and represent important steps on such data processing pipelines. They are iterative algorithms. The Q26 and Q52 come up as an example of interactive queries that are currently popular on Spark. Indeed nowadays big data applications are moving from the early days' batch processing to more interactive workloads.

We conduct our experiments on three types of virtual machine environments on the Microsoft Azure HDInsight PaaS [17], all of them running Spark. The goal is to explore different deployments of what the provider has to offer, including general purpose, CPU, and memory optimized instances. Considering that fault-tolerant parallel systems such as Spark are built to run on commodity clusters, it is important to guarantee the stability of the methods across different resource configurations. Two different Spark versions have also been considered.

For what concerns the A3 and D12v2 VMs, the Spark 1.6.2 release and Ubuntu 14.04 were considered. The D4v2 VM featured Ubuntu 16.04 and Spark 2.1.0. All the scenarios had two dedicated master nodes over D12v2 VMs. Table 3 details the configurations.

In the A3 case, the workers configuration consisted of 6 up to 48 cores, while in the case of D12v2 the number of cores has varied between 12 and 52. The D4v2 deployments consisted of 24 cores and 48 cores, on three and six nodes respectively. Table 4 describes the set of scenarios we analyze. Each TPC-DS query and machine learning benchmark were run 10 times for each considered configuration.

Table 3: Experimental Deployment Configurations

VM	Cores	Cores per Exec.	Exec. RAM	Driver RAM	VM RAM	Persistent Disk
D12v2	4	2	2GB	4GB	28GB	200GB local SSD
A3	4	2	2GB	4GB	7GB	250GB local disk
D4v2	8	4	10GB	8GB	28GB	400GB local SDD

Table 4: Scenarios Description

#	Application	VM	Configuration (nodes; cores; data)
1	TPCDS Q26	D12v2	3-13; 4 cores per node; 500GB
2	TPCDS Q52	D12v2	3-13; 4 cores per node; 500GB
3	TPCDS Q26	A3	3-13; up to 4 cores per node; 500GB
4	TPCDS Q52	A3	3-13; up to 4 cores per node; 500GB
5	K-Means	D4v2	3 and 6; 8 per node; 8GB,48GB,96GB
6	Log. Regression	D4v2	3 and 6; 8 per node; 8GB,48GB,96GB
7	SVM	D4v2	3 and 6; 8 per node; 8GB,48GB,96GB

The accuracy of the performance prediction models is measured by the relative error metric, evaluated using the average real time measured on the system and prediction time for each application:

$$\varepsilon_{\mathrm{r}} = \frac{T_{\mathrm{real}} - T_{\mathrm{predict}}}{T_{\mathrm{real}}}. \tag{3}$$

4.2 D12v2 Environment (Scenarios 1 & 2)

This section presents the results obtained by the Fork-Join prediction model [19], the Task Precedence prediction model [16] and dagSim simulator over Spark 1.6.1 experiments executed on Azure HDInsight D12v12 VMs. As previouly described, the Fork-Join model outputs two values: a lower bound and an upper bound. The first does not provide any useful information since it is does not take into consideration the task synchronization delays while the latter is the value we are interested in, since it is usually considered for capacity planning. Its value represents an application execution time threshold for that cluster's hardware and configuration.

Considering scenario 1, the Fork-Join model's upper bound error ranges from −256.18% to −162.27%. The maximum error was obtained for the largest configuration. We attribute these high errors to the model's overestimation of delays caused by task synchronization in our setup: as a single fork-join task may be broken into successive steps, due to its resource demands and server capacity, extra synchronization overheads are introduced to the system and add to the overall response time of a single task. These response times are then overinflated by the model's use of a constant to approximate server synchronization delays, which is applied to response times that already include delays caused by task synchronization.

Both the Task Precedence model and dagSim simulator showed better estimates, with errors ranging from 4.4% to 20.7% and −0.1% to 16.2%, respectively. These models consider the parallel execution DAG to better capture the dependencies and interactions between

tasks, resulting in more accurate estimates of synchronization delays. For scenario 2, we found similar results when comparing the three models. Table 5 details these results.

Table 5: Scenarios 1 & 2: Real and predicted execution times (seconds). Results in *bold* for the maximum error and *shaded cells* for the minimum error.

Cores	Real	Fork-Join (error %) Lower	Fork-Join (error %) Upper	Task Prec. (error %)	DagSim (error %)
Scenario 1: TPC-DS Q26 over D12v2 VMs					
12	722.2	626.9 (13.2)	1945.3 (-169.4)	690.2 (4.4)	682.3 (5.5)
16	582.9	465.5 (20.1)	1573.8 (-170.0)	543.9 (6.7)	526.5 (9.7)
20	515.9	391.6 (24.1)	1408.9 (-173.1)	469.0 (9.1)	455.3 (11.8)
24	447.6	335.4 (25.1)	1266.4 (-182.9)	398.3 (11.0)	394.3 (11.9)
28	415.7	**290.0 (30.2)**	1138.7 (-173.9)	367.2 (11.7)	**348.4 (16.2)**
32	366.1	**255.6 (30.2)**	1037.4 (-183.4)	316.5 (13.5)	312.4 (14.7)
36	306.1	228.5 (25.4)	953.8 (-211.6)	256.1 (16.3)	290.3 (5.2)
40	287.5	211.3 (26.5)	903.9 (-214.4)	236.8 (17.6)	270.3 (6.0)
44	259.7	195.5 (24.7)	855.1 (-229.3)	209.6 (19.3)	250.6 (3.5)
48	248.6	194.2 (21.9)	865.7 (-248.2)	**197.2 (20.7)**	249.0 (-0.1)
52	220.2	172.9 (21.5)	**784.5 (-256.3)**	181.4 (17.6)	221.0 (-0.4)
Scenario 2: TPC-DS Q52 over D12v2 VMs					
12	719.9	657.9 (8.6)	2041.5 (-183.6)	660.8 (8.2)	716.0 (0.6)
16	562.7	521.5 (7.3)	1763.2 (-213.3)	517.3 (8.1)	559.6 (0.6)
20	471.8	404.2 (14.3)	1454.2 (-208.2)	412.7 (12.5)	468.3 (0.8)
24	417.7	353.7 (15.3)	1335.4 (-219.7)	358.3 (14.2)	415.3 (0.6)
28	364.1	298.6 (18.0)	1172.8 (-222.1)	304.7 (16.3)	**360.7 (0.9)**
32	324.7	263.2 (18.9)	1068.4 (-229)	265.0 (18.4)	322.3 (0.7)
36	306.8	240.5 (21.6)	1004.2 (-227.3)	247.0 (19.5)	**304.2 (0.9)**
40	275.2	215 (21.9)	920.0 (-234.3)	215.2 (21.8)	273.1 (0.8)
44	258.8	202.2 (21.9)	884.2 (-241.7)	200.2 (22.7)	257.0 (0.7)
48	250	**192.4 (23)**	857.7 (-243.1)	**190.7 (23.7)**	248.3 (0.7)
52	226.1	177.5 (21.5)	**805.6 (-256.3)**	179.3 (20.7)	224.2 (0.8)

Overall, taking absolute values, on average the errors were 201.14% for the Fork-Join model upper bound, 13.45% for the Task Precedence model, and 7.73% for dagSim in scenario 1. For scenario 2, the Fork-Join model obtained 225.33%, the Task Precedence model obtained 16.92%, and dagSim obtained 0.74%.

As described in Section 3, Fork-Join model is a very simple approach, which depends only on the number of parallel tasks and on the average execution time of a single task for performance prediction. Unfortunately, its simplicity is not suitable for predicting perfomance, with reasonable accuracy, in the scenarios we are interested in. Then, in the next sections, we will consider only Task Precedence model and the dagSim Simulator approaches. As these models explicitly capture the precedence relation between stages, they may be able to better estimate the synchronization delays and provide more accurate predictions.

4.3 A3 Environment (Scenarios 3 & 4)

Although the scenarios 3 and 4 consider the same cluster sizes, data set size and queries as the scenarios 1 and 2, they differ from the latter on the VM type and on how the experiments were planned. Note that for these scenarios there is no regular amount of cores per nodes as in scenarios 1 and 2. Here, we ran experiments simulating an environment with memory pressure. Since A3 VMs have less available RAM memory than the D12v2 ones, a lower number of executors were allocated. Note that this is due to memory contention among the executors and the underlying operating system processes and the behavior was not deterministic.

Regarding the results, the prediction error of Task Precedence and dagSim varies from 0.8% and 10.0% and 0.01% to −11.7%, respectively. Similar results were found for scenario 4. In both scenarios, the models presented lower errors for the smaller configuration and higher errors for the larger configurations. We interpret this as an accumulation in the synchronization delay prediction error. Once we have more cores to execute the tasks, we also have more synchronization delay to predict and more errors to sum up. The results for scenarios 3 and 4 are detailed in Table 6.

Table 6: Scenarios 3 & 4: Real and predicted execution times (seconds). Results in *bold* for the maximum error and *shaded cells* for the minimum error.

Nodes (Cores)	Scenario 3 (error %) Real	Task Prec.	DagSim	Scenario 4 (error %) Real	Task Prec.	DagSim
3(6)	2532.3	2512.8(0.8)	2538.5(-0.3)	2158.9	2107.6(2.4)	2153.0(0.3)
3(8)	2071.2	2052(0.9)	2086.1(-0.7)	1709.2	1656.8(3.1)	1702.5(0.4)
4(10)	1778.8	1763.6(0.9)	1778.6(0.01)	1327.4	1276.4(3.8)	1316.3(0.9)
4(12)	1690.6	1674.5(1.0)	1704.5(-0.8)	1124.5	1072.5(4.6)	1117.4(0.6)
5(14)	1439.3	1414(1.8)	1452.9(-0.9)	976.4	924.0(5.4)	970.5(0.6)
5(16)	1271.6	1243.3(2.2)	1281.0(-0.7)	884.9	832.6(5.9)	880.4(0.5)
6(18)	1127.2	1099.8(2.4)	1152.9(-5.4)	816.5	764.8(6.3)	809.8(0.8)
6(20)	1093.6	1064.2(2.7)	**1095.2(-11.7)**	738.8	687.3(7.0)	733.3(0.7)
7(22)	996.7	963.2(3.4)	1050.4(5.4)	667.9	613.6(8.1)	663.3(0.7)
7(24)	911.2	874.8(4.0)	933.9(-2.5)	620.1	566.2(8.7)	615.8(0.7)
8(26)	720.8	682.2(5.4)	735.6(-2.1)	572.1	512.7(10.4)	568.4(0.6)
9(30)	658.8	617.6(6.3)	691.6(5.0)	525.9	572.9(7.6)	**465.0(11.6)**
9(32)	629.8	589(6.5)	643.7(-2.2)	492.3	432.7(12.1)	487.9(0.9)
10(34)	625.8	584.3(6.6)	647.6(-3.5)	462.2	402.5(12.9)	457.4(1.0)
10(36)	577.4	532.4(7.8)	602.9(-4.4)	442.1	382.6(13.5)	439.3(0.6)
11(38)	546.6	503.1(8.0)	572.1(-4.7)	438.7	380.2(13.3)	436.8(0.4)
11(40)	530.6	487.3(8.2)	555.2(-4.6)	418.4	359.1(14.2)	415.6(0.7)
12(42)	488.4	447.3(8.4)	510.6(-4.6)	392.1	334.2(14.8)	390.2(0.5)
12(44)	470.7	427.6(9.2)	493.8(-4.9)	383.7	325.9(15.1)	379.5(1.1)
13(46)	446.4	405.6(9.1)	455.2(-2.0)	378.4	318.8(15.8)	380.1(-0.5)
13(48)	430.5	**387.5(10.0)**	460.9(-7.1)	362.8	**305.0(15.9)**	359.3(1.0)

The results found for the scenarios 3 & 4 support those found for scenarios 1 & 2, signifying that the models are stable for both VMs and queries tested. Overall, taking absolute values, on average the errors were 5.03% for the Task Precedence model and 3.50% for dagSim in scenario 3. For scenario 4, the Task Precedence model obtained 9.8%, and dagSim obtained 1.17%. Furthermore, both models performed well under a deployment subject to memory pressure. The dagSim simulator proved stable, specially on scenario 4 with low errors except for the 9 nodes and 30 cores experiment. The Task Precedence errors increased as the number of cores increased, indicating that the prediction can be affected by the cluster size.

We assume that this is due to the accumulation of synchronization delay estimation errors present in the model.

4.4 D4v2 Environment (Scenarios 5, 6 & 7)

For scenarios 5, 6, and 7 we executed the Task Precedence prediction model and dagSim Simulator considering Spark 2.1.0 logs for a set of machine learning algorithms, namely K-Means, Logistic Regression, and SVM. The ML workloads are iterative algorithms and usually characterized by a larger number of stages than the scenarios 1 to 4. For these applications, data partitions are cached and accessed multiple times during the iterations. As noticed, these workloads present a higher variability since each iteration consists of data processing and RDD partitions re-computation in case of RDD cache eviction.

As detailed by Table 7, for every algorithm, the Task Precedence model prediction error is inversely proportional to the size of data sets, i.e., the larger the data sets, the lower the prediction error. Since processing larger data sets requires more tasks to be executed, the experiments yield a lower variance on the application response times. Analogously, a smaller number of tasks would result in higher variance across multiple runs. Regarding the different cluster sizes, results for all three algorithms were similar to those found previously. For larger cluster sizes, the models produce higher errors: as previously discussed, this is attributed to the accumulation of synchronization delays over a large number of distributed tasks running in multiple cores.

Differently, dagSim did not show any error pattern and its worst case error (−25.6%) is achieved for K-Means.

We further looked into the response times measured for individual runs of each algorithm on each configuration and observed that the setup with the largest errors for all three benchmarks for Task Precedence (8 GB on 48 cores) coincides with the scenario with the highest variance across multiple runs. The large number of cores used on a relatively small dataset, which might occasionally cause resource underutilization, may explain the slightly worse performance of the model in this setup.

With regard to absolute errors, both Task Precedence and dagSim provide very good prediction accuracy across the considered set of experiments, covering different platforms and configurations. While Task Precedence obtained a 5.48% average percentage error across scenarios 5, 6, and 7, dagSim achieved 8.34%.

Concerning the scenarios, Task Precedence obtained 9.03%, 1.62%, and 5.80% errors for scenarios 5, 6, and 7, respectively. Similarly, dagSim performed at an average 16.45%, 2.42%, and 7.43%.

4.5 Summary of Results

To summarize all our results, we observe that the Task Precedence model achieved errors that vary from 0.8% to 20.7%, being on average only 7.38% (average computed across all errors taken in absolute values). The errors achieved by dagSim, on the other hand, vary from 0.7% up to −25.6%, but with an average of only 5.65%. It is important to observe that in the performance evaluation literature, 30% errors (consistent across cluster size) in execution time predictions can be usually expected, especially from analytical models (see [14]). Thus, both approaches are suitable for predicting the performance of Big Data applications. Moreover, we note that dagSim

Table 7: Scenarios 5, 6 & 7: Real and predicted execution times (seconds). Results in *bold* for the maximum error and *shaded cells* for the minimum error.

Nodes (cores)	Data set size (GB)	Real	Task Prec. (error %)	dagSim (error %)
Scenario 5: K-Means on Azure D4v2				
3 (24)	8	99.0	81.9 (17.3)	75.6 (23.6)
3 (24)	48	342.2	325.1 (5.0)	364.6 (-6.5)
3 (24)	96	862.1	845.9 (1.9)	788.4 (8.5)
6 (48)	8	90.3	**74 (18.1)**	70.3 (22.1)
6 (48)	48	195.0	178.8 (8.3)	219.2 (-12.4)
6 (48)	96	594.3	572.9 (3.6)	**746.2 (-25.6)**
Scenario 6: Logistic Regression on Azure D4v2				
3 (24)	8	164.6	159.5 (3.1)	156.1 (5.1)
3 (24)	48	669.4	664.4 (0.7)	671.7 (-0.3)
3 (24)	96	1418.8	1414.1 (0.3)	1404.9 (0.9)
6 (48)	8	166.5	**161.0 (3.3)**	**156.5 (6.0)**
6 (48)	48	368.2	362.5 (1.5)	362.9 (1.4)
6 (48)	96	1200.7	1192.6 (0.6)	1193.9 (0.5)
Scenario 7: SVM on Azure D4v2				
3 (24)	8	190.9	171.7 (10.1)	167.7 (12.2)
3 (24)	48	356.7	339.2 (4.9)	358.1 (0.4)
3 (24)	96	1,367.0	1349.6 (1.3)	1323.9 (3.2)
6 (48)	8	189.7	**170.2 (10.3)**	**164 (13.5)**
6 (48)	48	372.5	353.2 (5.2)	352.2 (5.4)
6 (48)	96	650.5	631.1 (3.0)	635.4 (2.3)

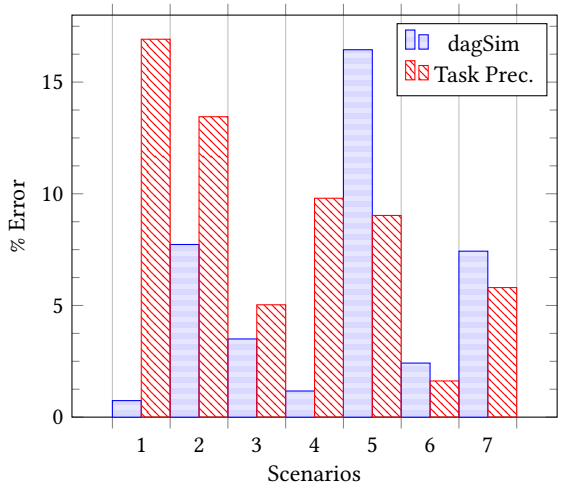

Figure 3: Average prediction errors across all analyzed scenarios (averages computed across errors taken in absolute values)

outperforms the Task Precedence model in all scenarios with interactive queries, whereas the latter was the best approach for the iterative ML algorithms. A summary of these results is shown in Figure 3. Our results of the Fork-Join model, on the other hand, indicate that the simpler model provides only a very coarse approximation, which is too conservative, especially compared to the other approaches.

Moreover, for what concerns the execution time, dagSim usually runs in 10 seconds on average, but this time can increase up to nearly 10 minutes for complex DAGs. When compared with the literature, JMT or GreatSPN for the same models can take up to one hour without obtaining greater accuracy (see [5] for additional details). On the other hand, on equivalent scenarios, the Task Precedence model performed quite well in terms of model solving time (always around one second). However, the Task Precedence model is limited to assess average execution time, whereas dagSim can be extended to provide also percentiles of application performance, thus enabling much finer grained analyses. In light of all these considerations, we find that there is no clear winner between the two approaches.

5 CONCLUSIONS

In this paper we compared two analytical models and proposed an ad hoc simulator for the performance prediction of Spark applications running on Cloud clusters.

Multiple Cloud configurations and workloads (including SQL and iterative machine learning benchmarks) have been considered. From the results we achieved, the Fork-Join model proposed in [19] is too inaccurate to be considered in practice. On the other side, results show that both the Task Precedence model proposed by Mak & Lundstrom and the dagSim simulator perform very well for predicting the average system response time and are effective in capturing the dynamic resource assignment implemented in Spark, achieving 7.38% and 5.63% average percentage error across all the experiments, respectively.

In our future work we plan to extend our models to cope with scenarios where multiple applications run concurrently competing to access the resources in the same clusters. Finally, we will embed the models into a runtime optimization tool for managing dynamically Cloud resources with the aim of providing application execution within an a priori fixed deadline while minimizing Cloud operational costs.

ACKNOWLEDGEMENT

The authors work has been partially funded by the EUBra-BIGSEA project by the European Commission under the Cooperation Programme (MCTI/RNP 3rd Coordinated Call), Horizon 2020 grant agreement 690116. This research was also be partially funded by CNPq and FAPEMIG, Brazil.

REFERENCES

[1] [n. d.]. Apache Spark Survey 2016 Results Now Available. ([n. d.]). https://databricks.com/blog/2016/09/27/spark-survey-2016-released.html
[2] [n. d.]. The Digital Universe in 2020. ([n. d.]). http://idcdocserv.com/1414
[3] [n. d.]. The Programming Language LUA. ([n. d.]). https://www.lua.org/home.html
[4] Danilo Ardagna, Simona Bernardi, Eugenio Gianniti, Soroush Karimian Aliabadi, Diego Perez-Palacin, and José Ignacio Requeno. 2016. Modeling Performance of Hadoop Applications: A Journey from Queueing Networks to Stochastic Well Formed Nets. In *Algorithms and Architectures for Parallel Processing - 16th International Conference, ICA3PP 2016, Granada, Spain, December 14-16, 2016, Proceedings*. 599–613.
[5] Danilo Ardagna, Simona Bernardi, Eugenio Gianniti, Soroush Karimian Aliabadi, Diego Perez-Palacin, and José Ignacio Requeno. 2016. Modeling Performance of Hadoop Applications: A Journey from Queueing Networks to Stochastic Well Formed Nets. In *Algorithms and Architectures for Parallel Processing - 16th International Conference, ICA3PP 2016, Granada, Spain, December 14-16, 2016, Proceedings*. 599–613. https://doi.org/10.1007/978-3-319-49583-5_47
[6] Marco Bertoli, Giuliano Casale, and Giuseppe Serazzi. 2009. JMT: performance engineering tools for system modeling. *SIGMETRICS Performance Evaluation Review* 36, 4 (2009), 10–15. https://doi.org/10.1145/1530873.1530877
[7] Keke Chen, James Powers, Shumin Guo, and Fengguang Tian. 2014. CRESP: Towards Optimal Resource Provisioning for MapReduce Computing in Public Clouds. *IEEE Transactions on Parallel and Distributed Systems* 25, 6 (2014), 1403–1412. https://doi.org/10.1109/TPDS.2013.297
[8] Giovanni Chiola. 1985. A Software Package for the Analysis of Generalized Stochastic Petri Net Models. In *International Workshop on Timed Petri Nets, Torino, Italy, July 1-3, 1985*. 136–143.
[9] G. Ciardo, R. L. Jones, III, A. S. Miner, and R. I. Siminiceanu. 2006. Logic and stochastic modeling with SMART. *Perform. Eval.* 63 (June 2006), 578–608. Issue 6. https://doi.org/10.1016/j.peva.2005.06.001
[10] Derrick. 2015. Survey shows huge popularity spike for Apache Spark. (2015). http://fortune.com/2015/09/25/apache-spark-survey
[11] H. V. Jagadish, Johannes Gehrke, Alexandros Labrinidis, Yannis Papakonstantinou, Jignesh M. Patel, Raghu Ramakrishnan, and Cyrus Shahabi. 2014. Big Data and Its Technical Challenges. *Commun. ACM* 57, 7 (July 2014), 86–94.
[12] Miriam Leeser Janki Bhimani, Ningfang Mi. [n. d.]. Scalable Performance Prediction Techniques for Big Data Processing in Distributed Multi-Core Systems. ([n. d.]). http://staging-rise.s3.amazonaws.com/1082/3/1001/bhimani_janki_niljll.pdf?AWSAccessKeyId=AKIAIZD5HUIXRXZ4FWDA&Expires=1775950506&Signature=87LWnIaPnB%2BS%2BNPgavF6VM9ypQo%3D
[13] Jacek Laskowski. 2016. Mastering Apache Spark. https://www.gitbook.com/book/jaceklaskowski/mastering-apache-spark. (2016). [Online; accessed 28-June-2016].
[14] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. 1984. *Quantitative System Performance*. Prentice-Hall. http://homes.cs.washington.edu/~lazowska/qsp/
[15] Deron Liang and Satish K. Tripathi. 2000. On Performance Prediction of Parallel Computations with Precedent Constraints. *IEEE Trans. Parallel Distrib. Syst.* 11, 5 (2000), 491–508. https://doi.org/10.1109/71.852402
[16] V.W. Mak and S.F. Lundstrom. 1990. Predicting Performance of Parallel Computations. *IEEE Transactions on Parallel & Distributed Systems* 1, undefined (1990), 257–270. https://doi.org/doi.ieeecomputersociety.org/10.1109/71.80155
[17] Microsoft. [n. d.]. Sizes for Windows virtual machines in Azure. https://docs.microsoft.com/en-us/azure/virtual-machines/windows/sizes. ([n. d.]). [Online; accessed 15-January-2017].
[18] Microsoft. 2016. What is PaaS? https://azure.microsoft.com/en-us/overview/what-is-paas/. (2016). [Online; accessed 30-August-2016].
[19] Randolf D. Nelson and Asser N. Tantawi. 1988. Approximate Analysis of Fork/Join Synchronization in Parallel Queues. *IEEE Trans. Computers* 37, 6 (1988), 739–743. http://dblp.uni-trier.de/db/journals/tc/tc37.html#NelsonT88
[20] D. M. Nicol and A. S. Miner. 1995. The Fluid Stochastic Petri Net Simulator. In *Proceedings of the Sixth International Workshop on Petri Nets and Performance Models (PNPM '95)*. IEEE Computer Society, Washington, DC, USA, 214–. http://dl.acm.org/citation.cfm?id=826033.826758
[21] Adrian Daniel Popescu. 2015. *Runtime Prediction for Scale-Out Data Analytics*. Ph.D. Dissertation. IC, Lausanne. https://doi.org/10.5075/epfl-thesis-6629
[22] Wolfgang Reisig, Grzegorz Rozenberg, and P. S. Thiagarajan. 2013. *In Memoriam: Carl Adam Petri*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–5. https://doi.org/10.1007/978-3-642-38143-0_1
[23] Ge Song, Zide Meng, Fabrice Huet, Frederic Magoules, Lei Yu, and et al. 2013. A Hadoop MapReduce Performance Prediction Method. In *HPCC 2013*. 820–825.
[24] Don Towsley, John C.S. Lui, and Richard R. Muntz. 1998. Computing Performance Bounds of Fork-Join Parallel Programs Under a Multiprocessing Environment. *IEEE Transactions on Parallel & Distributed Systems* 9, 3 (1998), 295–311. https://doi.org/doi.ieeecomputersociety.org/10.1109/71.674321
[25] Satish K. Tripathi and De-Ron Liang. 2000. On Performance Prediction of Parallel Computations with Precedent Constraints. *IEEE Transactions on Parallel & Distributed Systems* 11, undefined (2000), 491–508. https://doi.org/doi.ieeecomputersociety.org/10.1109/71.852402
[26] Kishor S. Trivedi. 2002. SHARPE 2002: Symbolic Hierarchical Automated Reliability and Performance Evaluator. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*. IEEE Computer Society, Washington, DC, USA, 544.
[27] Elizabeth Varki and Lawrence W. Dowdy. 1996. Analysis of Balanced Fork-join Queueing Networks. *SIGMETRICS Perform. Eval. Rev.* 24, 1 (May 1996), 232–241.

https://doi.org/10.1145/233008.233048

[28] Emanuel Vianna, Giovanni Comarela, Tatiana Pontes, Jussara Almeida, Virgílio Almeida, Kevin Wilkinson, Harumi Kuno, and Umeshwar Dayal. 2013. Analytical Performance Models for MapReduce Workloads. *International Journal of Parallel Programming* 41, 4 (2013), 495–525. https://doi.org/10.1007/s10766-012-0227-4

[29] Guanying Wang, Ali Raza Butt, Prashant Pandey, and Karan Gupta. 2009. A simulation approach to evaluating design decisions in MapReduce setups.. In *MASCOTS*. IEEE Computer Society, 1–11. http://dblp.uni-trier.de/db/conf/mascots/mascots2009.html#WangBPG09

[30] Kewen Wang and Mohammad Maifi Hasan Khan. 2015. Performance Prediction for Apache Spark Platform.. In *HPCC/CSS/ICESS*. IEEE, 166–173. http://dblp.uni-trier.de/db/conf/hpcc/hpcc2015.html#WangK15

[31] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.