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

Data heterogeneity and irregularity are key characteristics of big data applications that 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. Yet, the same characteristics impose extra challenges to predicting the performance of cloud-based big data applications, a central step in proper management and planning. This paper explores two modeling approaches for performance prediction of cloud-based big data applications. We evaluate a queuing-based analytical model and a novel fast ad-hoc simulator in various scenarios based on different applications and infrastructure setups. Our results show that our approaches can predict average application execution times with 26% relative error in the very worst case and about 12% on average. Moreover, our simulator provides performance estimates 70 times faster than state of the art simulation tools.

KEYWORDS

Performance modeling; Big data; Spark; Approximate methods; Simulation

ACM Reference Format:
Danilo Ardagna, Enrico Barbierato, Athanasia Evangelinou, Eugenio Gianniti, Marco Gribaudo and Túlio B. M. Pinto, Anna Guimarães, Ana Paula Couto da Silva, Jussara M. Almeida . 2018. Performance Prediction of Cloud-Based Big Data Applications. In *ICPE '18: ACM/SPEC International Conference on Performance Engineering, April 9–13, 2018, Berlin, Germany.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3184407.3184420

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 [13]. 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, the elastic nature of cloud computing systems provide a natural hosting platform to cost-effectively provision the dynamic resource requirements of big data applications. Indeed, 61% of Spark adopters ran their applications on the cloud in 2016 [1].

Yet, 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. 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 meet the applications' dynamic needs, avoiding Service Level Agreement (SLA) violations.

A plethora of different modeling techniques, varying from analytical approaches to simulation tools, have been proposed and applied in the past to study system performance [5, 7, 16, 19, 23, 24, 26]. Nevertheless, their efficiency to model massively parallel applications introducing thousands of parallel tasks has been shown to be an issue [3]. Thus, we here take the challenge of predicting the performance of big data applications by exploring two very different techniques, an analytical model and a simulation tool, which, as will be discussed, have complementary pros and cons in terms of prediction accuracy and efficiency. Our goal is to efficiently estimate (in a few seconds), the *average execution time* of a target application, given the available resources, in a way we can support run-time reconfiguration decisions. That is, 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) on a given resource deployment (described in terms, e.g., of numbers of cores or nodes). We focus on applications running on Spark[1], 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 [9].

Firstly, we investigate the use of an analytical queuing network (QN) model for predicting the performance of Spark applications.

[1]http://spark.apache.org/

The model, originally proposed in [16] for performance prediction of parallel application, extends an Approximated Mean Value Analysis (AMVA) technique by modeling the precedence relationships and parallelism between individual tasks of the same job. This model, here referred to as *Task Precedence model*, explicitly captures the overlap in execution times of different tasks of the same job to estimate the average application execution time.

We also propose and evaluate *dagSim*, a novel ad-hoc and fast discrete event simulator to model the execution of complex DAGs. The advantages of dagSim with respect to state of the art simulators and AMVA techniques are twofold. On one side, the simulation process achieved great accuracy within a shorter timescale with respect to other formalisms (e.g., Stochastic Petri Nets) or specific tools (e.g., JMT [5] or GreatSPN [7]). Furthermore, the tool provides percentile estimate, which cannot be obtained via the analytical *Task Precedence model*.

We evaluate the modeling approaches in four scenarios consisting of different virtual machine environments and applications, as well as different resource configurations. Our results indicate a good overall accuracy for both Task Precedence model and the dagSim simulator (with 26% relative error in the very worst case and about 12% on average). Both models presented similar performance, specifically dagSim performed better for interactive queries while the Task Precedence model performed better for iterative machine learning (ML) algorithms. dagSim demonstrated to be on average 70 times faster than JMT while providing the same accuracy.

The rest of this paper is organized as follows. Section 2 presents related work, while Section 3 introduces our two prediction models. Section 4 describes the experimental scenarios we explored and discusses our main results. Conclusions are offered in Section 5.

2 RELATED WORK

This paper focuses on the use of modeling techniques to enable the analysis of the viability of big data jobs. Recently, sophisticated projects have emerged in the study of Spark applications performance, such as PREDIcT [20] and RISE2016 [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 multicore systems. From a more general perspective, the most relevant related work has been subdivided into two parts, specifically i) analytical queuing network methods and ii) simulation approaches.

Analytical Queuing Network Methods: 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 limitations, 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 applications running on top of parallel systems [16, 19, 23, 24, 26].

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.

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 queriable data. The index server subsystem is modeled as a fork-join network. In this model, an incoming task is split into identical subtasks, which are 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 [26] 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, which is analogous to balanced job bounds developed for product form networks. In the same direction, [23] models a multiprocessing computer system as K homogeneous servers, each with an infinite capacity queue. The authors provide a computationally efficient algorithm for obtaining upper and lower bounds on the system expected response time.

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. While the models proposed in [19, 23, 26] 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 [24].

In our work, we apply the model proposed by the authors in [16], given that the model parameters are easily obtained (for instance, service demands and task structure) and results are obtained with low complexity cost. More model details are presented in Section 3.2.

Simulation Approaches: Several simulation tools, which are tailored to study the behavior of parallel applications through stochastic formalisms such as SPNs (Stochastic Petri Nets see [21]) have been implemented. GreatSPN supports the analysis of Generalized Stochastic Petri Nets (GSPNs) including both immediate and timed (the fire event occurs either immediately or within a stochastic time) transitions and of Stochastic Well-Formed Nets (SWNs, i.e., Petri nets where the tokens can be distinguished) [7]. SMART (Symbolic Model checking Analyzer for Reliability and Timing, [8]) includes both stochastic models and logical analysis. SHARPE (Symbolic Hierarchical Automated Reliability and Performance Evaluator) is a tool to analyze stochastic models [25], the most notable being fault trees, product form queuing networks, Markov chains, and GSPNs. It is also able to mix submodels of fork-joins and queues. JMT [5] is a suite of applications offering a framework for performance evaluation, system modeling, and capacity planning.

The problem of studying the performance prediction of individual jobs is explored in [22] through a framework consisting of a Hadoop job analyzer, while the prediction component exploits locally weighted regression methods. A similar issue is studied in [27] by using instead a hierarchical model including a precedence graph model and a queuing network model to simulate the intra-job synchronization constraints. In [6], 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 [28] with particular attention to i) the effect of several component inter-connect topologies, ii) data locality, and iii) software and hardware failures.

Our previous work [3] describes multiple queuing network models (simulated with JMT) and stochastic well formed nets (simulated with GreatSPN) to model MapReduce applications, highlighting the trade-offs 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. A comparison between dagSim and JMT is reported in Section 4.2.

Finally, parallel and distributed processing have been investigated also by means of Process Algebra (PA, [10]). A PA is a mathematical framework describing how a system evolves by using algebraic components and providing a set of methods for their manipulation. Among the different implementations, Performance Evaluation Process Algebra (PEPA, [11] is a formal language for distributed systems, whose models correspond to continuous time Markov chains (CTMC).

3 PERFORMANCE PREDICTION MODELS

This section presents the two modeling approaches analyzed in this paper to predict the performance of cloud-based big data applications. 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). We then discuss the Task Precedence queuing network model (Section 3.2), and introduce the dagSim discrete event simulator (Section 3.3).

3.1 Spark Overview and Model Assumptions

Spark is a fault-tolerant cluster computing framework that provides abstractions for parallel computation across distributed nodes with multiple cores. It is a fast and general purpose engine for large-scale data processing, which was first proposed as an alternative to Hadoop MapReduce [29]. Spark is the state of the 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 can be composed of a single job, multiple jobs, or a continuous processing. A job is composed of 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 a 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 run time, 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 Directed Acyclic Graph (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 sequence of fully-independent tasks that can run right away based on the data that is already on the cluster [14]. Only stages have precedence relationships, which are represented by the DAG.

Our present goal is to evaluate the effectiveness of two performance prediction techniques to estimate the execution time of Spark applications. The performance prediction in parallel systems has been approached in several ways, with varying degrees of detail, cost, and accuracy. Focusing on a 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. That is, the models for calculating performance measures have to take into account how the executions of stages overlap among themselves.

To that end, we made the following assumptions for both (analytical and simulation) 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 of tasks of the same class with no precedence relationship among themselves (i.e., they may run in parallel), v) the number of tasks within a stage is constant and known a priori, vi) an individual application obtains dedicated resources for its execution (i.e., VMs that are executed on a cloud cluster), vii) resources (such as memory, CPU, disk) are homogeneous (as often happens in cloud deployments, see, e.g., [18]).

3.2 Task Precedence Model

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], and then discuss how it was applied to Spark applications. The reader is referred to the original paper for a detailed derivation of the model.

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 of multiple parallel/sequential tasks. The precedence relationships between different tasks are expressed as a series-parallel 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 iterates over these computations until they converge below a given error threshold. At each iteration, the precedence graph is reduced by aggregating multiple tasks and estimating the average execution time of aggregated node. In particular, execution times of sequential tasks are added, and execution times of parallel tasks are 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 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. 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 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 summed up.

As a final note, the original model assumes that the times required to process the execution times of each block of computation represented by a node in the input DAG are exponentially distributed [16]. Since we here consider each node in the DAG as a stage, the assumption is that the execution times of Spark stages are exponentially distributed. Having said that, we emphasize that this assumption may not hold in practice, possibly depending on characteristics of the application. In other words, it is a potential source of approximation error of the model. Yet, the low prediction errors we obtain in all considered scenarios, as will be shown in Section 4, indicate that the model is quite robust to such assumption.

3.3 dagSim Simulator

dagSim is a high speed discrete event simulator built to analyze DAGs corresponding to MapReduce and Spark jobs[2].

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:

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

where $N_{\text{Nodes}} \in \mathbb{N}, N_{\text{Nodes}} \geq 1$ represents the number of computational nodes $N_{\text{Users}} \in \mathbb{N}, N_{\text{Users}} \geq 1$ the number of users concurrently submitting jobs to the system, and $\mathcal{Z}$ is the "think time distribution", i.e., the time a user will wait before submitting a new job. Set $S = \left\{s_1, \ldots, s_{N_{\text{Stages}}}\right\}$ 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.

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 ([4]) 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.

In order to perform an efficient simulation of jobs, stages are characterized by a set of possible states:

- CAN_START: represents stages that can be executed, since all the previous stages have completed, but that have not started yet because the scheduler is still waiting for resources to be available.
- $WAITING$: identifies stages that cannot be executed since some of the previous stages have not been completed.
- $RUNNING$: Tasks belonging to the stage in this state are being executed (i.e., a stage that was in the CAN_START state has found the necessary resources).
- $ENDED$: all the tasks of the considered stage have been completed.

Initially, only the stages s_i that have no dependencies (i.e., such that $Pre(s_i) = \emptyset$) are in the CAN_START state, and all the other are in the $WAITING$ state. Each stage in the $RUNNING$ state exploits a variable to count the number of tasks that still need to be completed. The core idea of the simulation engine is that each time a task of stage s_k has been executed, this counter is decremented of one unit. When the counter reaches zero, the engine can determine i) that a stage has been completed and ii) which stages are now eligible to start, changing their state from $WAITING$ to CAN_START.

By using a doubly-linked list storing the relevant information about the tasks belonging to stages in the CAN_START state, it is possible to determine which one can be executed without performing a full search on the complete set of tasks in the DAG. In this sense, the approach provided by dagSim's engine is original and more efficient with respect to other scheduling mechanisms implemented in general purpose tools such as JMT [5] or GreatSPN [7].

Algorithm 3.1 summarizes the procedure to simulate the execution of one job according to the given DAG. Initially (lines 2–5), for each of the N_{Users} users 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 that need to be completed for each stage, 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 start. The data structure modeling the execution nodes is

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

initialized at line 6: it is mainly used to determine whether a node is free or working on a task.

The algorithm continues by scheduling the time at which each user submits her first job (lines 7–9) by adding a new event whose timestamp corresponds to the *think time*. Events are collected in a *CalendarEvent* data structure. Each job is characterized by a doubly-linked list *JobData*, populated by i) a user identifier, ii) a job and a stage status, and a iii) task identifier.

The most important part of the algorithm consists of the cycle repeating the simulation for all the considered jobs (lines 11–37). At line 12, the next simulation event is extracted (*pop* operation) from the *CalendarEvent* structure.

If the event represents a user requesting the launch of a new job (line 14), the function *initUserJobData* is invoked (line 15) to initialize all the job's stages to *CAN_START* or *WAITING* state depending on whether the stage has dependencies or not.

The simulator assumes that nodes are locked for a job and that they can be used by the next one only when they are no longer needed by a user. This is implemented by exploiting a lock that is set when a new job starts and reset when all its stages have been started. If there are available computational nodes and no lock has been set (line 16), the *scheduleReadyTasksOnAvailNodes* function is invoked (line 17) to i) set a lock if a new job is started and ii) schedule the waiting jobs on available nodes. If instead the job cannot be started, it is inserted into an auxiliary list (line 19).

If the event identifies the end of a task (line 21), the corresponding counter of the remaining tasks in the stage is decremented by one unit (line 22). Function *releaseNode* is invoked (line 23) in order to free the computational resources; this also removes the lock on the nodes if 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.

The stage is considered to be over if there are no tasks left (line 24): in this case the stage state is updated to *ENDED* (line 25) and the *UpdateStageStatus* function is invoked to see if the completion of this stage allows other stages to change their status from *WAITING* to *CAN_START*. If another stage can start (line 27), the new tasks are scheduled (line 28); otherwise the job is considered to be completed. The job ending time (line 30) is set at the current time and the next job from the same user is submitted after another think time (lines 31-32). To allow the simulation to stop when the total number of considered jobs has been executed, the number of completed jobs is increased (line 33).

4 EXPERIMENTAL RESULTS

In this section, we present the results of a set of experiments we performed to explore and validate the Task Precedence analytical model as well as the dagSim simulator. Our evaluation considers scenarios with different types of applications: the TPC-DS industry benchmark and some reference machine learning (ML) benchmarks, namely K-Means and Logistic Regression. In other words, our tests include SQL 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]. All experiments were performed on the Microsoft Azure cloud platform.

Algorithm 3.1 Simulation engine algorithm

```
 1: function SOLVE(Model M, Users U, CalendarEvent ce)
 2:     UserJobData **UJD;
 3:     for user_i ∈ Users do
 4:         UJD[i] = createUserJobData(M);
 5:     end for
 6:     NodeData *ND = initNodeData(M);
 7:     for user_i ∈ Users do
 8:         nEv = AddEvent( ce, ThinkTime);
 9:     end for
10:     int TotalJobEnded = 0;
11:     while TotalJobEnded < maxJobs(M) do
12:         event = pop(CE);
13:         Job *jd = event->data;
14:         if isNewJobStarting(event) then
15:             initUserJobData(jd->userId, M);
16:             if (ND->freeNodes > 0) AND (!lock(ND)) then
17:                 scheduleReadyTasksOnAvailNodes(ce, ND);
18:             else
19:                 addToAux(event, WAITLIST);
20:             end if
21:         else
22:             remainingTasksXStage[jd->stageId]−;
23:             releaseNode(currTime, ce, ND, UJD);
24:             if remainingTasksXStage[jd->stageId] ≤ 0 then
25:                 setstatus(s_k, ENDED);
26:                 UpdateStageStatus(UJD, M)
27:                 if NewStageCanStart(J_i, M) then;
28:                     scheduleReadyTasksOnAvailNodes(ce, ND);
29:                 else
30:                     SetJobEndTime(currTime);
31:                     nEv = addEvent(ce, T);
32:                     nEv->data = populateJobData();
33:                     TotalJobEnded++;
34:                 end if
35:             end if
36:         end if
37:     end while
38: end function
```

4.1 Scenarios

Our experimental scenarios cover the most widely used applications on Spark [1]. The ML benchmarks, namely K-means and Logistic Regression, are core activities in machine learning applications and represent important steps on such data processing pipelines. They are iterative algorithms. We also selected the TPC-DS Q26 and Q52 queries as examples of interactive SQL queries that are currently popular on Spark. Indeed nowadays big data applications are moving from the early days' batch processing to more interactive workloads. Note that while TPC-DS DAGs are rather simple, including up to 7 stages, ML DAGs are very complex and introduce a high level of parallelism, up to 114 stages for Logistic Regression.

We conduct our experiments on two types of virtual machine environments on the Microsoft Azure HDInsight PaaS [17], namely D12v2 and D4v2. 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 taken in account. For what concerns the 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. In the D12v2 case, the workers' configuration

Table 1: 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	K-Means	D4v2	3 and 6; 8 per node; 8GB,48GB,96GB
4	Log. Regression	D4v2	3 and 6; 8 per node; 8GB,48GB,96GB

consisted of 12 up to 52 cores. The D4v2 deployments consisted of 24 cores and 48 cores, on three and six nodes respectively.

Table 1 describes the set of scenarios we analyze. Each TPC-DS query and ML benchmark was run 10 times for each considered configuration.

We evaluate the Task Precedence analytical model and the dagSim simulation with respect to prediction accuracy and average execution time. Prediction accuracy is estimated by the relative error ε_r, computed using the average real execution time (T_{real}), measured on the real system, and the execution time predicted by the model (T_{predict}), for each application:

$$\varepsilon_r = \frac{T_{\text{real}} - T_{\text{predict}}}{T_{\text{real}}}. \quad (3)$$

Note that negative values of ε_r imply overestimates, while positive values correspond to underestimates.

Execution times of the analytical model and simulator have been gathered on a Ubuntu 16.04 Virtualbox VM with eight cores running on an Intel Nehalem dual socket quad-core system with 32 GB of RAM. The virtual machine has eight physical cores dedicated with guaranteed performance and 4 GB of memory reserved. Unless otherwise stated, we report the average of 10 runs.

Before presenting our results, we first compare the execution time of dagSim against the one of the JMT tool.

4.2 Comparison with JMT

In this section, we compare the average execution time of dagSim with that of the event based QN simulator available within the JMT 1.0.2 tool suite. JMT is very popular among researchers and practitioners and since 2006 has been downloaded more than 58,000 times. The comparison focuses on the average execution time at 95% confidence level. JMT accuracy analyses are reported in our previous work [3], where we obtained an average percentage error up to 33% while the mean of its absolute value was around 14.13%. The ratio between the average simulation times of JMT and dagSim for two considered scenarios are reported in Figure 1. dagSim is clearly much faster than JMT (about 70 times on average and up to 115 times in the very worst case for the Q26 DAG, which includes a larger number of stages), also with slightly better accuracy than JMT (as will be discussed extensively in the following sections).

4.3 Results on the D12v2 VMs (Scenarios 1 & 2)

This section presents the results obtained by the Task Precedence model and the dagSim simulator in scenarios 1 and 2, over Spark 1.6.2 executed on Azure HDInsight D12v12 VMs. Real and predicted application execution times for each scenario and various configurations (i.e., numbers of nodes and cores) are shown in Table 2. In this

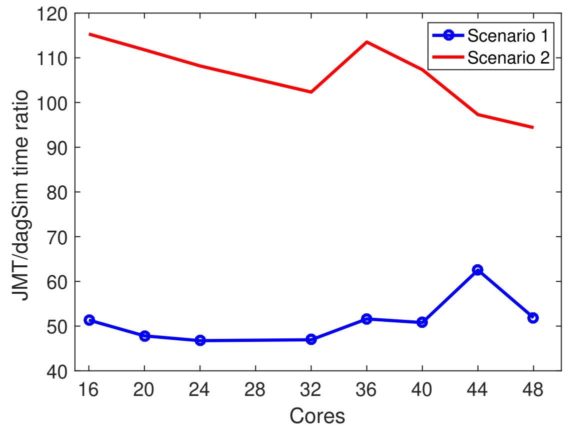

Figure 1: JMT and dagSim execution time ratio

table (as in the following ones), relative errors of each tool in each scenario/configuration are presented in parentheses, and maximum and minimum errors are shown in bold and shaded, respectively.

For scenario 1, both the Task Precedence model and the dagSim simulator showed very good estimates, with errors ranging from 4.4% to 20.7% and −0.1% to 16.2%, respectively. Similar results were also obtained in scenario 2: the errors of the Task Precedence model varied between 8.1% and 23.7%, whereas dagSim showed excellent accuracy, with errors below 1%.

Table 2: Scenarios 1 & 2: Real and predicted execution times (seconds).

Nodes	Scenario 1 (error %)			Scenario 2 (error %)		
(cores)	Real	Task Prec.	DagSim	Real	Task Prec.	DagSim
3(12)	722.2	690.2 (4.4)	682.3 (5.5)	719.9	660.8 (8.2)	716.0 (0.6)
4(16)	582.9	543.9 (6.7)	526.5 (9.7)	562.7	517.3 (8.1)	559.6 (0.6)
5(20)	515.9	469.0 (9.1)	455.3 (11.8)	471.8	412.7 (12.5)	468.3 (0.8)
6(24)	447.6	398.3 (11.0)	394.3 (11.9)	417.7	358.3 (14.2)	415.3 (0.6)
7(28)	415.7	367.2 (11.7)	**348.4 (16.2)**	364.1	304.7 (16.3)	**360.7 (0.9)**
8(32)	366.1	316.5 (13.5)	312.4 (14.7)	324.7	265.0 (18.4)	322.3 (0.7)
9(36)	306.1	256.1 (16.3)	290.3 (5.2)	306.8	247.0 (19.5)	**304.2 (0.9)**
10(40)	287.5	236.8 (17.6)	270.3 (6.0)	275.2	215.2 (21.8)	273.1 (0.8)
11(44)	259.7	209.6 (19.3)	250.6 (3.5)	258.8	200.2 (22.7)	257.0 (0.7)
12(48)	248.6	**197.2 (20.7)**	249.0 (-0.1)	250.0	**190.7 (23.7)**	248.3 (0.7)
13(52)	220.2	181.4 (17.6)	221.0 (-0.4)	226.1	179.3 (20.7)	224.2 (0.8)

Overall, taking absolute values, average errors were 13.45% and 16.92% for the Task Precedence model, and 7.73% and 0.74% for dagSim, in scenarios 1 and 2, respectively. These are very good estimates, given the complexity of the environment and workloads, especially for practical purposes of planning and managing the resource requirements. The greater errors of the analytical model are probably due to the several sources of approximations embedded in this solution (see Section 3.2 and [16]).

Table 3 reports, as an example, the quartiles of Q26 and Q52 at 16 cores. The table displays both the simulated quartiles and the ones

Table 3: Real and predicted execution time quartiles

Query	Quartile	dagSim [s]	Real [s]	ε_r [%]
Q26	Q_1	492.496	515.449	4.66
Q26	Q_2	495.077	537.436	8.56
Q26	Q_3	497.800	597.302	19.99
Q52	Q_1	509.974	509.810	0.03
Q52	Q_2	511.676	515.547	0.76
Q52	Q_3	513.454	520.582	1.39

derived from 20 sample runs on the real system. The estimated quartiles are quite accurate, with a worst case relative error of 19.99%, but at an average as low as 5.90%. Note that percentile distributions can be obtained only through simulation based approaches and cannot be provided by the Task Precedence method.

4.4 Results on the D4v2 VMs (Scenarios 3 & 4)

In scenarios 3 and 4 we executed the Task Precedence model and dagSim simulator considering Spark 2.1.0 logs for two machine learning algorithms, namely Logistic Regression and K-Means. The ML workloads are iterative algorithms and characterized by a larger number of stages than the scenarios 1 and 2. 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 4, for both algorithms, the Task Precedence model prediction error is inversely correlated 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. We also found that the model produces somewhat higher errors for larger cluster sizes. This is attributed to the accumulation of synchronization delays over a larger number of distributed tasks running in multiple cores.

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 the two 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.

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

With regards to errors taken in absolute value, once again we find that both Task Precedence and dagSim provide very good prediction accuracy across the considered set of experiments, covering different platforms and configurations. Average errors for the analytical model are 9.03% and 1.62% for scenarios 3 and 4, respectively. Average errors for dagSim were somewhat higher — 16.45% and 2.42%, respectively — though still very low for practical purposes.

Table 4: Scenarios 3 & 4: Real and predicted execution times (seconds).

Nodes (cores)	Data set size (GB)	Real	Task Prec. (error %)	dagSim (error %)
Scenario 3: K-Means				
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 4: Logistic Regression				
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)

4.5 Summary of Results

In sum, we observe that the Task Precedence model achieved errors that vary from 0.8% to 20.7%, being on average 11.70% (average computed across all errors taken in absolute values). The errors achieved by dagSim, on the other hand, vary from −0.1% up to −25.6%, but with an average of only 6.06%. It is important to observe that in the performance evaluation literature, 30% errors (consistent across cluster sizes) in execution time predictions can be usually expected, especially from analytical models (see [15]). Thus, both approaches are suitable for predicting the performance of big data applications. Moreover, we notice that dagSim outperforms the Task Precedence model in the scenarios with interactive queries, whereas the latter was the best approach for the iterative ML algorithms. Figure 2 summarizes our results.

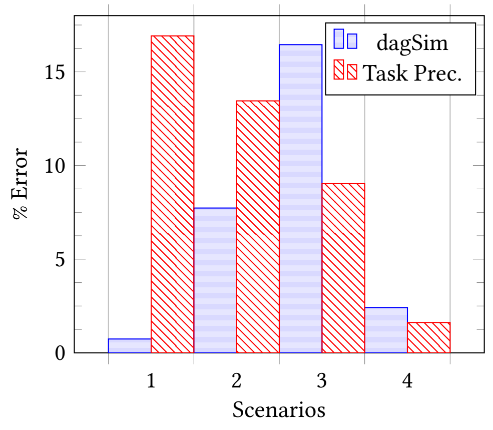

Figure 2: Prediction errors across analyzed scenarios (averages computed across errors taken in absolute values)

Moreover, both tools ran very quickly and are suitable for online predictions. The average execution times of dagSim were 3.09 seconds for scenario 1 and only 0.76 seconds for scenario 2, with very low variability across multiple runs (coefficient of variation[3] (CV) of 0.06 in both cases). Vice versa, JMT took on average 156 and 83 seconds, respectively.

For scenarios 3 and 4, despite the higher variability (CVs of 0.9 and 0.8, respectively), the average execution times were still short, 1.2 and 2.4 seconds, respectively. Note that in this latter scenario the higher variability was due to the different size of the underlying dataset (which has an impact on the number of tasks within stages and the number of simulated events).

The execution times of the analytical Task Precedence model was very short, varying from only 4.18 milliseconds (for scenario 2) to up to 40 milliseconds (for scenario 4). They were also mostly stable (i.e., low CVs) across all scenarios. The average execution times are 5.35 ms, 4.59 ms, 9.42 ms and 28.38 ms for scenarios 1 to 4, respectively, whereas the corresponding CVs are 0.12, 0.05, 0.32, and 0.29. Thus, comparing both tools, dagSim's execution times exceed those of our analytical model by some orders of magnitude: their ratio varies from around 10 to over 680. However, the Task Precedence model is limited to assess average execution time, whereas dagSim can provide also percentiles of application performance, thus enabling much finer-grained analyses.

5 CONCLUSIONS

In this paper, we analized an analytical models and proposed an adhoc 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, 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 11.07% and 6.06% 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 run-time optimization tool for dynamically managing 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] D. Ardagna, S. Bernardi, E. Gianniti, S. Karimian Aliabadi, D. Perez-Palacin, and J. I. Requeno. 2016. Modeling Performance of Hadoop Applications: A Journey from Queueing Networks to Stochastic Well Formed Nets. In *ICA3PP*. 599–613. https://doi.org/10.1007/978-3-319-49583-5_47
[4] E. Barbierato. 2016. *dagSim Documentation.* Technical Report. Politecnico di Milano. https://github.com/eubr-bigsea/dagSim/blob/master/simlib/Documentation/scheduler/manual/1.63/manual.html
[5] M. Bertoli, G. Casale, and G. Serazzi. 2009. JMT: Performance Engineering Tools for System Modeling. *ACM SIGMETRICS Performance Evaluation Review* 36, 4 (2009), 10–15.
[6] K. Chen, J. Powers, S.Guo, and F. Tian. 2014. CRESP: Towards Optimal Resource Provisioning for MapReduce Computing in Public Clouds. *IEEE TPDS* 25, 6 (2014), 1403–1412. https://doi.org/10.1109/TPDS.2013.297
[7] G. Chiola. 1985. A Software Package for the Analysis of Generalized Stochastic Petri Net Models. In *International Workshop on Timed Petri Nets*. 136–143.
[8] 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
[9] H. Derrick. 2015. Survey Shows Huge Popularity Spike for Apache Spark. (2015). http://fortune.com/2015/09/25/apache-spark-survey
[10] W.J. Fokkinkk. 2000. *Introduction to Process Algebra.* Springer.
[11] J. Hillston. 1996. *A Compositional Approach to Performance Modelling.* Cambridge University Press, New York, NY, USA.
[12] M. Leeser J. Bhimani, N. Mi. [n. d.]. Scalable Performance Prediction Techniques for Big Data Processing in Distributed Multi-Core Systems. ([n. d.]). http://hdl.handle.net/2047/D20215315
[13] 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.
[14] J. Laskowski. 2016. Mastering Apache Spark. (2016). https://www.gitbook.com/book/jaceklaskowski/mastering-apache-spark
[15] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. 1984. *Quantitative System Performance.* Prentice-Hall. http://homes.cs.washington.edu/~lazowska/qsp/
[16] V. W. Mak and S. F. Lundstrom. 1990. Predicting Performance of Parallel Computations. *IEEE Trans. Parallel Distrib. Syst.* 1, 3 (July 1990), 257–270. https://doi.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? (2016). https://azure.microsoft.com/en-us/overview/what-is-paas/
[19] R. D. Nelson and A. N. Tantawi. 1988. Approximate Analysis of Fork/Join Synchronization in Parallel Queues. *IEEE Trans. Computers* 37, 6 (1988), 739–743.
[20] A. D. Popescu. 2015. *Runtime Prediction for Scale-Out Data Analytics.* Ph.D. Dissertation. IC, Lausanne. https://doi.org/10.5075/epfl-thesis-6629
[21] W. Reisig, G. 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
[22] G. Song, Z. Meng, F. Huet, F. Magoules, L. Yu, and et al. 2013. A Hadoop MapReduce Performance Prediction Method. In *HPCC*. 820–825.
[23] D. Towsley, J. C.S. Lui, and R. 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/10.1109/71.674321
[24] S. K. Tripathi and D. Liang. 2000. On Performance Prediction of Parallel Computations with Precedent Constraints. *IEEE Transactions on Parallel & Distributed Systems* 11 (2000), 491–508. https://doi.org/10.1109/71.852402
[25] K. S. Trivedi. 2002. SHARPE 2002: Symbolic Hierarchical Automated Reliability and Performance Evaluator. In *DSN*. IEEE Computer Society, Washington, DC, USA, 544.
[26] E. Varki and L. 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
[27] E. Vianna, G. Comarela, T. Pontes, J. Almeida, V. Almeida, K. Wilkinson, H. Kuno, and U. 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
[28] Gu. Wang, A. R. Butt, P. Pandey, and K. Gupta. 2009. A Simulation Approach to Evaluating Design Decisions in MapReduce Setups. In *MASCOTS*. IEEE Computer Society, 1–11.
[29] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. 2010. Spark: Cluster Computing with Working Sets. In *HotCloud*. USENIX Association, Berkeley, CA, USA, 10–10. http://dl.acm.org/citation.cfm?id=1863103.1863113

[3] Ratio of standard deviation to mean value.