Int J Parallel Prog (2013) 41:495–525
DOI 10.1007/s10766-012-0227-4

Analytical Performance Models for MapReduce Workloads

Emanuel Vianna · Giovanni Comarela · Tatiana Pontes · Jussara Almeida · Virgílio Almeida · Kevin Wilkinson · Harumi Kuno · Umeshwar Dayal

Received: 10 December 2011 / Accepted: 9 October 2012 / Published online: 27 November 2012

Abstract MapReduce is a currently popular programming model to support parallel computations on large datasets. Among the several existing MapReduce implementations, Hadoop has attracted a lot of attention from both industry and research. In a Hadoop job, map and reduce tasks coordinate to produce a solution to the input problem, exhibiting precedence constraints and synchronization delays that are characteristic of a *pipeline* communication between maps (producers) and reduces (consumers). We here address the challenge of designing analytical models to estimate the performance of MapReduce workloads, notably Hadoop workloads, focusing particularly on the intra-job pipeline parallelism between map and reduce tasks belonging to the

E. Vianna (✉) · G. Comarela · T. Pontes · J. Almeida · V. Almeida
Federal University of Minas Gerais (UFMG), Belo Horizonte, Brazil
e-mail: vianna@dcc.ufmg.br

G. Comarela
e-mail: giovannicomarela@dcc.ufmg.br

T. Pontes
e-mail: tpontes@dcc.ufmg.br

J. Almeida
e-mail: jussara@dcc.ufmg.br

V. Almeida
e-mail: virgilio@dcc.ufmg.br

K. Wilkinson · H. Kuno · U. Dayal
Information Analytics Lab, HP Laboratories (HP-Labs), Palo Alto, CA, USA
e-mail: kevin.wilkinson@hp.com

H. Kuno
e-mail: harumi.kuno@hp.com

U. Dayal
e-mail: umeshwar.dayal@hp.com

same job. We propose a hierarchical model that combines a precedence graph model and a queuing network model to capture the intra-job synchronization constraints. We first show how to build a precedence graph that represents the dependencies among multiple tasks of the same job. We then apply it jointly with an approximate Mean Value Analysis (aMVA) solution to predict mean job response time, throughput and resource utilization. We validate our solution against a queuing network simulator and a real setup in various scenarios, finding very close agreement in both cases. In particular, our model produces estimates of average job response time that deviate from measurements of a real setup by less than 15 %.

Keywords Performance · Hadoop · Pipeline · Queuing Network · Task graph

1 Introduction

MapReduce [6,7] is a simple and currently very popular programming model to support parallel computations on large datasets. Indeed, several applications in different areas such as data mining, machine learning and graph processing are suitable to its simple interface [9]. Several implementations of the MapReduce model have been developed. Examples of implementations that have attracted a lot of attention from both research and industry are Hadoop [2,34] and the Hadoop Online Prototype (HOP) [5]. Hadoop, in particular, an open-source software, has been adopted by many different institutions for both educational and production uses [1].

MapReduce workloads, and Hadoop workloads in particular, are composed of jobs, each one containing a collection of map and reduce tasks. Map tasks are responsible for breaking the input into smaller sub-problems and producing solutions for each smaller sub-problem, in parallel. Reduce tasks are responsible for processing the results produced by the map tasks, combining them into a solution to the original problem. Each reduce task must retrieve and process the results produced by all maps. Thus, the communication between map and reduce tasks exhibit precedence constraints and synchronization delays that are inherent to a *pipeline* between producers (maps) and consumers (reduces).

Given the increasing use of MapReduce as platform for developing parallel data intensive applications, the design of methods that allow one to understand and predict the performance of such applications is appealing [8,32]. Indeed, a clear understanding of system performance under different circumstances is key to critical decision-making in workload management and capacity planning, providing answers to questions such as: how many and what type of machines are required in a commodity cluster to support a pre-defined Service Level Agreement (SLA)? How should system parameters be fine-tuned?

Analytical performance models are particularly attractive tools that serve that purpose as they might provide reasonably accurate job performance estimates such as job response time and throughput, and ultimately help one to answer the aforementioned questions, at significantly lower cost than simulation and experimental evaluation of real setups [11]. However, to the best of our knowledge, there is no previous effort to develop analytical models that capture the intra-job pipeline parallelism inherent to MapReduce workloads.

The challenge to develop such models is that they must capture, with reasonable accuracy, the various sources of delays that a job experiences. In particular, besides the execution time, tasks belonging to a job may experience two types of delays: (1) queuing delays due to contention at shared resources, and (2) synchronization delays due to precedence constraints among tasks that cooperate in the same job. Several modeling techniques to predict the performance of workloads that do not exhibit synchronization delays are available in the literature [11,22]. One such technique is Mean Value Analysis (MVA) [22], which has been applied to predict the average performance of several applications in various scenarios [13,29]. However, MVA cannot be directly applied to workloads that have precedence constraints, such as the synchronization among map and reduce tasks belonging to the same MapReduce job. Alternative exact solutions, which jointly exploit Markov Chains, to represent the possible states of the system, and queuing network models, to compute the transition rates between states, are also available [16,27]. However, such approaches do not scale well since the state space grows exponentially with the number of tasks.

There have also been some efforts to characterize the main components of the MapReduce platform [18], as well as analyze the performance of MapReduce workloads, aiming at helping parameter tuning [10,15,36], evaluating the impact of task scheduling on system performance [37], or even proposing new scheduling mechanisms for intercloud environments [14] as well as heterogeneous environments [3]. However, most of these efforts do not attempt to model the intra-job synchronization delays introduced by the pipeline between map and reduce tasks. Other authors have aimed at profiling and modeling network processor applications, which also exhibit pipeline parallelism [33]. However, they have focused on the worst-case performance, providing analytical expressions for performance estimates based on the critical (i.e., slowest) stage of the pipeline only. Thus, they have also not explicitly modeled the synchronization delays introduced by the pipeline.

In this context, this article proposes an efficient and reasonably accurate analytical performance model to predict performance metrics, notably average job response time, of MapReduce workloads, focusing specifically on the Hadoop system, which is currently very popular. Our solution is built from a previous model [19], referred to as *reference model*, which was designed to predict the performance of parallel computations. Given a tree specifying the precedence constraints among tasks of a parallel job as input, the reference model applies an iterative approximate Mean Value Analysis (aMVA) algorithm to predict performance metrics (e.g., average job response time, resource utilization and throughput). The reference model allows different types of precedence constraints among tasks of a job, specified by simple task operators, as will be discussed in Sect. 2.3. However, none of these operators capture the precedence constraints of a pipeline communication. Thus, the reference model cannot be directly applied to Hadoop workloads.

Our model, which extends the reference model, represents the intra-job pipeline as a precedence tree, built using the simple task operators introduced in [19]. Our contributions over [19] are twofold: (1) we explicitly address the synchronization delays introduced by the pipeline parallelism, and show how to use the primitive task operators introduced in the reference model to build a precedence tree for it; (2) we propose an alternative strategy to estimate the average response time of subsets of the tasks belonging to a Hadoop job, which leads to more accurate estimates of a job's average response time.

We validate our analytical model against an event-driven queuing network simulator as well as a real Hadoop setup, finding reasonably close agreements in both cases. In particular, our analytical model provides estimates of average job response time that are within 15 % of the corresponding values measured in a real system. Moreover, we also show that our model produces conservative results, which are a particularly nice property for supporting system management and planning tasks. This article builds on our previous work [30] by greatly increasing the experimental evaluation of our model. Unlike in our previous work, we here validate our model against measurements of a real Hadoop system. Moreover, we include new experiments that allow us to evaluate model accuracy under a richer set of scenarios.

The remaining of this article is organized as follows. Related work is presented in Sect. 2. The Hadoop architecture and workloads are described in Sect. 3. Section 4 presents our analytical model, whereas results from our experimental evaluation are presented in Sect. 5. Section 6 offers the conclusion and future work.

2 Related Work

MapReduce is a programming model to support parallel computations on large datasets. The main idea of the MapReduce model is to hide the details of the parallel execution from users, so that they can focus only on data processing strategies. It consists of two main phases, namely, map and reduce, each one implemented by multiple tasks running on multiple nodes. In general terms, during the map phase, the input is divided into smaller sub-problems and solutions for each smaller sub-problem are produced, in parallel. During the reduce phase, the answers to all sub-problems are combined, in some way, to produce a final output. Thus, a MapReduce job is composed of a number of map and reduce tasks, which run in parallel but exhibit precedence constraints similar to those of a pipeline communication between maps and reduces.

There are several implementations of the MapReduce model. The open-source Hadoop [2,34] and its recent extension Hadoop Online Prototype (HOP) [5] are two such implementations. Hadoop, in particular, is currently very popular [1]. For that reason, we here focus on developing an analytical model to predict the performance of Hadoop workloads. A detailed description of the Hadoop system and workload is given in Sect. 3.

We start this section by briefly reviewing, in Sect. 2.1, several analytical performance modeling techniques. Next, Sect. 2.2 presents previous efforts to analyze the performance of parallel applications, particularly of MapReduce systems. Finally, Sect. 2.3 introduces the strategy upon which our solution model was built, here referred to as the *reference model*.

2.1 Analytical Performance Modeling Techniques

The literature contains a variety of different techniques to model the performance of computer systems. In particular, analytical techniques based on queuing models have been widely applied to analyze and predict the performance of various types of systems. In a queuing model, a service center, containing a server and an associated queue, represents any hardware or software resource (e.g., a CPU, a disk, a semaphore) where there might be contention among tasks that are executing concurrently in the system. When arriving at a center, if a task finds the local server occupied with another task, it waits in the queue for processing. Thus, one can represent the main points of contention, and thus, the main sources of delays in the system, by a queuing network, composed of multiple interconnected service centers.

One approach to solve queuing network models is by means of Markov models, which have been widely used in various scenarios, including to model parallel applications, as will be discussed in the next section. Markov models are based on a representation of the system by a state diagram which captures all possible states[1] that the modeled system may find itself, as well as the possible transitions between such states and the rates at which such transitions occur. Thus, Markov models capture the dynamics of the system. Once parameterized, they can be solved to produce several performance-related measures. In particular, one can estimate the probabilities of the system being in each state and, derived from such probabilities, the probability distributions of several variables such as response time, throughput and queue length. However, Markov models have one key limitation: the complexity, defined by the size of the state space, grows exponentially with the number of service centers and task classes, which makes it hard to be applied in large and complex systems.

A different approach is Mean Value Analysis (MVA), which focuses on computing the *average* value of several performance metrics such as response time, throughput, queue length and resource utilization. An exact algorithm to solve an MVA model for a closed system[2] was proposed by Reiser and Lavenberg in 1980 [25]. We briefly revise this solution here as it is the basis of the reference model, on top of which our solution is built.

Suppose a system with C task classes and K service centers. The exact MVA algorithm estimates the average residence time of task i ($i = 1 \ldots C$) in center k ($k = 1 \ldots K$) as:

$$R_{ik}\left(\overrightarrow{N}\right) = D_{ik}\left(1 + A_{ik}\left(\overrightarrow{N}\right)\right) \tag{1}$$

where D_{ik} is the average demand[3] of a class i task on service center k $A_{ik}(\overrightarrow{N})$ is the average queue length as seen by an arriving task of class i, and $\overrightarrow{N}$ is a vector defining the number of tasks of each class in the system.

[1] The system state can be defined, for example, by the number of tasks in each modeled service center.

[2] In a closed system, the task population is fixed and predefined. Thus, new task arrivals are triggered only by task completions.

[3] The average demand of a class i task on service center k is the average amount of time that the task spends using the corresponding resource during its execution in the system.

According to Lavenberg's Theorem [17], it is possible compute $A_{ik}(\overrightarrow{N})$ as:

$$A_{ik}\left(\overrightarrow{N}\right) = \sum_{j=1}^{C} Q_{jk}\left(\overrightarrow{N-1_i}\right), \tag{2}$$

where $Q_{jk}(\overrightarrow{N-1_i})$ is the average queue length of service center k when the system has one less task of class i.

Using Little's Law [20], it is possible to write $Q_{jk}(\overrightarrow{N})$ as a function of the throughput $X_{jk}(\overrightarrow{N})$ and the average response time, $R_{jk}(\overrightarrow{N})$, of class j tasks in service center k, that is:

$$Q_{jk}\left(\overrightarrow{N}\right) = X_{jk}\left(\overrightarrow{N}\right) R_{jk}\left(\overrightarrow{N}\right) = \frac{N_j}{\sum_{k=1}^{K} R_{jk}\left(\overrightarrow{N}\right)} R_{jk}\left(\overrightarrow{N}\right), \tag{3}$$

where N_j is the number of class j tasks in the system. Using Eqs. (1–3) iteratively, starting with the cases where there is a single class in the system, it is possible to obtain $R_{ik}(\overrightarrow{N})$ by increasing the number of tasks of each class one at a time, and using the average queue lengths computed in a previous iteration ($Q_{jk}(\overrightarrow{N-1_i})$) to compute the average residence times of the next iteration ($R_{ik}(\overrightarrow{N})$). At the end, we can estimate the average system response time for a class i task as:

$$R_i\left(\overrightarrow{N}\right) = \sum_{k=1}^{K} R_{ik}\left(\overrightarrow{N}\right) \tag{4}$$

Average resource and system throughput as well as average resource utilizations can also be computed using Little's Law.

Note that this algorithm grows exponentially with the number of tasks and classes that compose the workload (as well as with the number of centers). In order to reduce the solution complexity, several approximate solutions, which replace the aforementioned iteration by an approximation, have been proposed. One such approximate Mean Value Analysis (aMVA) solution, originally proposed by Zahorjan [38] for the case of single task class and extrapolated for the case of multiple classes in [19], estimates the average residence time of a class j task in center k when the task population is given by $\overrightarrow{N-1_i}$ as:

$$R_{jk}\left(\overrightarrow{N-1_i}\right) \approx R_{jk}\left(\overrightarrow{N}\right) - \frac{D_{jk} R_{ik}\left(\overrightarrow{N}\right)}{\sum_{k=1}^{K} R_{ik}\left(\overrightarrow{N}\right)}. \tag{5}$$

Several other modeling techniques are available in the literature. In particular, hierarchical models combining different approaches, such as Markov models in a higher layer and MVA or aMVA in a lower layer, have also been proposed [22]. We discuss some of these models, in the context of parallel applications, in the next section.

2.2 Performance of Parallel Applications

A number of previous studies have addressed the performance analysis of parallel applications, and MapReduce in particular. For instance, Lee et al. [18] performed a survey on MapReduce, characterizing its main components as well as its inherent pros and cons. In [32], the authors presented a MapReduce simulator, MRPerf, for facilitating exploration of MapReduce design space. MRPerf captures several aspects of a MapReduce environment, and uses this information to predict application performance. In [31], the same authors used MRPerf to study the impact of different network topologies, data locality as well as software and hardware failures on system performance. In comparison with an analytical performance model, the simulator captures many more details of the real system, but this comes at the cost of a significantly longer execution times.

Ganapathi [8] applied machine learning techniques to predict the performance of MapReduce workloads. The modeling approach consists in correlating the pre-execution characteristics of the workload with measured post-execution performance metrics (e.g., average job response time, resource utilization). Complementarily, Morton et al. [23] proposed a method to estimate the remaining time of a MapReduce job that is based on identifying and tracking only those map and reduce tasks that belong to the *critical path*. However, in these two previous efforts, the proposed solutions do not explicitly capture the precedence constraints and synchronization delays between map and reduce tasks, which is our goal.

Pavlo [24] compared the performance of MapReduce systems with parallel database systems, finding that the former are significantly slower when performing a variety of analytic tasks. This finding motivated Jiang et al. [12] to address the question: must a system sacrifice performance to achieve flexibility and scalability? Their results show that, with proper implementation, the performance of MapReduce can be improved by a factor of 2.5–3.5, outperforming parallel databases. These previous efforts did not focus on predicting but rather on characterizing the performance of MapReduce, being thus complementary to our work.

There have also been efforts towards deriving analytical models to describe the execution of a Hadoop job, with the goal of helping parameter tuning [10,15,36]. In a different direction, several previous studies analyzed the impact of task scheduling on the performance of Hadoop [37], proposing alternative scheduling mechanisms for intercloud environments, characterized by the presence of private and public clusters [14], as well as for heterogeneous environments [3]. Chen et al. [4] showed that existing benchmarks fail to capture the workload characteristics observed in production MapReduce traces, and proposed a framework to build and execute representative workloads. These efforts are complementary to our present work. To the best of our knowledge, no previous work used queuing network models to predict the performance of Hadoop systems.

Considering other parallel environments, Weng et al. [33] used analytical modeling and profiling to evaluate the performance of network processor applications in a heterogeneous multiprocessor system. Although such systems do exhibit pipeline parallelism, the authors developed analytical performance models based only the critical (i.e., slowest) stage of the pipeline, as they focused on the worst case throughput, which is limited by the critical stage. Thus, unlike the analytical model we develop here, their models do not explicitly capture the synchronization delays introduced by the pipeline.

Other previous studies used stochastic hierarchical models [13,19,21] to predict the performance of parallel applications. In such models, the higher level captures the synchronizations among tasks, expressed by a task graph, whereas the lower level captures the contention at shared resources, represented by a queuing network model. This combination gives a *trade-off* between effectiveness and efficiency for analytical models. Note that, if taken separately, these two modeling strategies loose expressivity [13] since a queuing network model by itself is not able to capture synchronization delays among tasks whereas task graphs alone do not address delays due to resource contention.

One such solution was proposed by Thomasian and Bay [27]. The precedence among tasks was modeled by a Markov Chain whereas a closed queuing network was used to compute the state transition rate. Although this method captures the synchronization of tasks in a pipeline, the state space grows exponentially with the increase of the number of tasks, making it impractical to be applied to model jobs with many tasks, as is commonly the case of Hadoop jobs. Mak and Lundstron [21] proposed an alternative solution with polynomial time that modeled task precedence by a serial-parallel task graph. Liang and Tripathi [19] extended this work to allow task graphs containing other types of operators, although none of them explicitly address pipeline parallelism. We further discuss this solution, here referred to as reference model, next.

2.3 Reference Model

Liang and Tripathi [19] proposed an analytical model to predict the performance of parallel workloads. Their model combines a precedence tree, which establishes the order of execution of tasks belonging to the same job and defines the synchronization constraints for job execution, and a closed queuing network to address resource contention.

In the precedence tree, each leaf node represents a task, and each internal node is an operator describing the constraints in the execution of the tasks in their children nodes. The authors consider precedence trees built from four types of primitive operators, namely, *serial* (S), *parallel-and* (P_A), *parallel-or* (P_O) and *probabilistic fork* (P_F). An S operator is used to connect tasks that must run in sequence, whereas tasks connected by a P_A run in parallel and must all finish before job execution proceeds. A P_O operator also connects tasks that run in parallel. However, unlike P_A, the group of tasks connected by a P_O finishes execution as soon as one of the tasks finishes. The operator P_F implies that only one of the tasks connected by it is executed with probability p. One example precedence tree is shown in Fig. 1.

An approximate Mean Value Analysis (aMVA) technique is used to solve the queuing network. To do so, they introduce the effect of the precedence rules by inflating the mean queue length as seen by an arriving task (which usually captures only resource contention) by factors representing the overlap in the execution times of tasks belonging to the same job (intra-job overlap) and tasks belonging to different jobs (inter-job

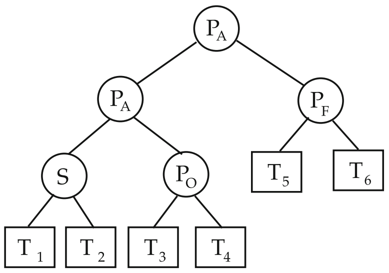

Fig. 1 Example of a precedence tree

overlap). In other words, they define two types of overlap factors, namely, α_{ij} and β_{ij}. The former captures the period of time during which two tasks i and j, belonging to the same parallel job, ran in parallel, whereas the latter captures the overlap in the execution times of two tasks i and j belonging to *different* jobs. They then replace Eq. 2 by the following approximation:

$$A_{ik}\left(\overrightarrow{N}\right) \approx \frac{1}{N}\sum_{j=1, i\neq j}^{C} \alpha_{ij} Q_{jk}\left(\overrightarrow{N-1_i}\right) + \frac{N-1}{N}\sum_{j=1}^{C}\beta_{ij} Q_{jk}\left(\overrightarrow{N-1_i}\right) \tag{6}$$

where N is the total number of jobs executing in the system. Both overlap factors are estimated based on the precedence tree, given as input to the model.

The main steps of the algorithm proposed by Liang and Tripathi are shown in Fig. 2a. The input parameters are the number K of service centers (physical resources), the service demand matrix D_{ik}, which specifies the average demand of task i in center k, and the multiprogramming level (MPL) N, which defines the average number of concurrent jobs in the system.

The algorithm starts by initializing the average residence time of each type of task at each center and the average response time of each task in the system (step 1). Next, the algorithm estimates the job response time distribution jointly with the intra-job overlap (step 2), followed by the inter-job overlap (step 3). These overlap factors are introduced in the aMVA solution, using Eq. 6, to produce new estimates of task average response time (step 4). Finally a convergence test is performed on the new estimates of average response times. If the new estimates are close enough (i.e., smaller than a pre-defined value ϵ) to the previous ones (i.e., they converge), a final average job response time as well as other performance metrics (throughput, resource utilizations) are produced. If the convergence test fails, the algorithm returns to step 2.

The job response time is estimated given the precedence tree and (estimates of) the distributions of the response time of each task. Since response time distribution in a closed queuing network is difficult to derive in general, the model relies on the assumption that tasks cycle back through the network many times before exiting (i.e., coarse grained) and, thus, their response times can be assumed to be exponentially distributed [26].

The job response time distribution is estimated by traversing the precedence tree in depth-first order, visiting nodes, recursively, from leaves (tasks) up to the root (job). The response time distribution of each (internal) node, J_i, is estimated combining

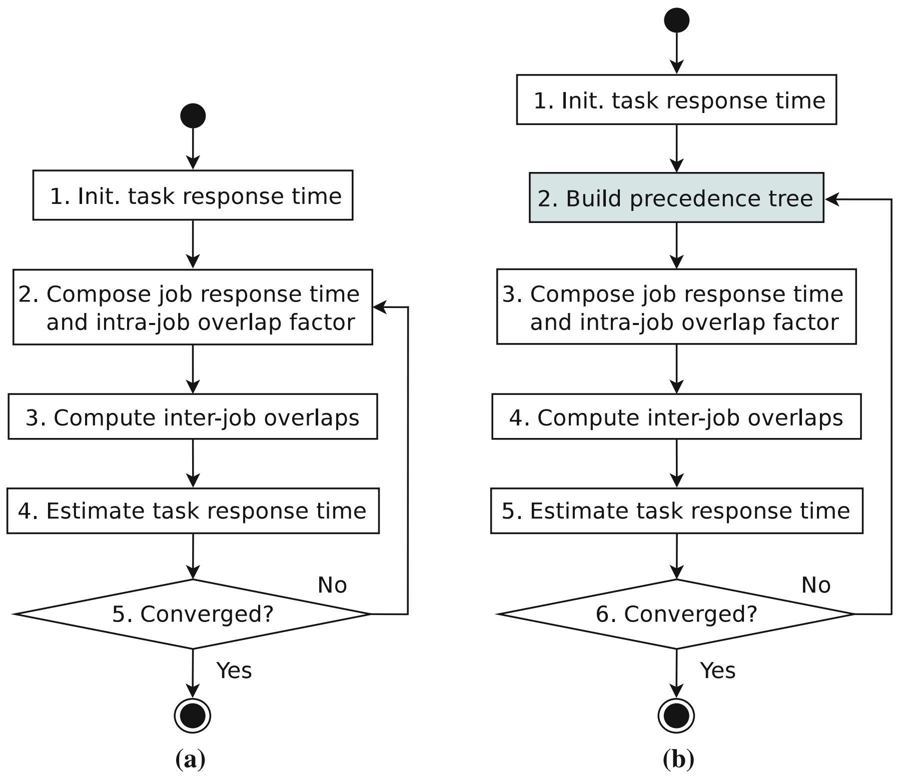
1. Init. task response time
2. Compose job response time and intra-job overlap factor
3. Compute inter-job overlaps
4. Estimate task response time
5. Converged?
No
Yes
(a)
1. Init. task response time
2. Build precedence tree
3. Compose job response time and intra-job overlap factor
4. Compute inter-job overlaps
5. Estimate task response time
6. Converged?
No
Yes
(b)

Fig. 2 Main steps of analytical model. **a** Reference model [19], **b** our solution

the (estimates) of response time distributions of the left and right sub-trees, J_1 and J_2, according to the type of the operator associated with the node. If it is serial, the composition of distributions J_1 and J_2 is given by the sum of their random variables, whereas for parallel-and, it is given by their maximum. Since, in either case, the exact solution for J_i is computationally intensive, the authors assume J_i to follow an Erlang distribution, if its coefficient of variation (CV) is less than or equal to 1, or an hyperexponential distribution otherwise [28]. Thus, at each internal node, the algorithm estimates the mean and variance (and CV) of response time, using the response time distributions (i.e., their means and variances) of the left and right children. For the sake of brevity we here choose to omit several details of the model solution, referring to [19] for its complete description.

Liang and Tripathi focused mainly on how to introduce precedence constraints in the iterative aMVA algorithm. Thus, they assumed that the precedence tree is given as input to their model. Moreover, they assumed input trees built from basic operators, none of which explicitly captures the pipeline parallelism of Hadoop workloads. We here show how to build this precedence tree and explain how to use the operators proposed by Liang and Tripathi to capture the synchronization delays inherent to pipeline parallelism.

3 Hadoop Architecture and Workload

This section presents a description of the main hardware (architecture) and software (workload) aspects that should be taken into account in order to model the performance of Hadoop [34].

In general, the infra-structure of Hadoop clusters consists of n hosts (nodes), each one containing a number of CPUs and disks, as illustrated in Fig. 3. There is a single master node and several worker nodes. The master node executes the JobTracker process and is responsible for accepting new jobs and assigning them to the worker nodes. Each worker node executes a TaskTracker process and manages the execution of the tasks assigned to its node. Each worker node has one or more attached disks with one local file system and one shared, called Hadoop Distributed File System (HDFS) [6,7].

A Hadoop job reads one or more input files and produces one output file. Each file is a sequence of registers containing two fields, namely, a key and a value. Moreover, each file is logically partitioned, defining the key range to be processed by each worker node. Each partition is further divided into sub-partitions of equal size, called splits.

Each Hadoop job has two main phases: map and reduce. Each map and reduce phase is further divided into m and r tasks, respectively. Moreover, the numbers of threads available in each node to execute (in parallel) map and reduce tasks are also fixed, given by parameters pm and pr, respectively. The master node uses a heartbeat protocol to check the status of the worker nodes. When the master node identifies that a worker node has an idle map/reduce thread (i.e., it finished processing its previously assigned map/reduce task), it assigns a new task of the same type to that thread.

Each map task reads registers from one split of the input file. For each split register, it executes the function MAP() and produces one or more output registers. The output registers are then sorted and written into a temporary file in the local file system. At this point, the map task terminates, and the corresponding thread becomes idle.

Each reduce task is responsible for processing a range of keys produced by the map tasks. One reduce task has three sub-phases, namely, *shuffle*, *sort* and *reduce*. A shuffle sub-phase transfers the data from the temporary file produced by the maps that matches the key range of the reduce to which the shuffle belongs. Note that the registers in the temporary file produced by each map are already partitioned according

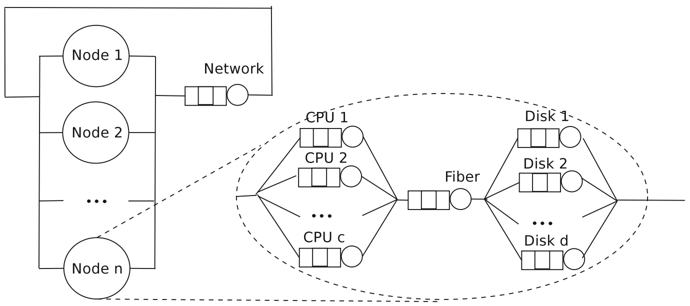

Fig. 3 Main architectural components of Hadoop

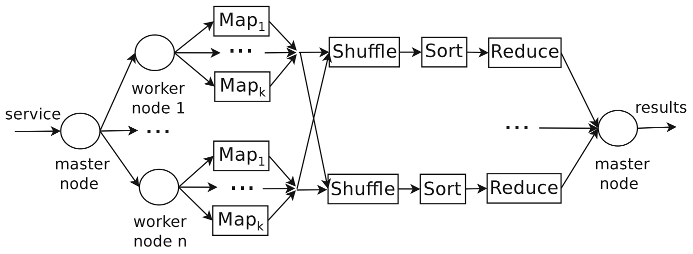

Fig. 4 Execution flow of a Hadoop job

to the key range of each reduce. Besides the task level parallelism, Hadoop also allows that each reduce executes a number ps of shuffle instances in parallel. A shuffle phase is thus divided into m sub-tasks, each one responsible for processing the results of a different map task, and run ps threads in parallel. After each shuffle, a partial sort is performed, and the registers are written into an output file in the local file system of the reduce node. When all partial sorts finish, a final sort, which merges the temporary files produced by the shuffles, is then performed. Finally, a reduce sub-phase reads this merged file, applies the function REDUCE() and writes the results in the shared file system. Figure 4 illustrates the execution flow of a typical Hadoop job.

Note that the results produced by each map task should be processed by all reduce tasks. That is,as soon as a map task, running on node n_i, finishes its execution, each reduce task, running on node n_j, sends a shuffle sub-task to n_i to transfer the data from n_i to n_j. Up to ps shuffles may be executed in parallel by each reduce. Moreover, after the map running on node n_i finishes, a new map task (if there is still one to be executed) is dynamically allocated to n_i. Note also that a reduce task may block if, when all shuffle sub-tasks finish to process the results of a sub-set of map tasks, there is no other finished map. Thus, one can see the communication between the collection of map tasks and the set of shuffle sub-tasks of each reduce as a pipeline, where each shuffle (consumer) needs to process the results processed by all maps (producers).

A recent extension of Hadoop, called Hadoop Online Prototype (HOP), implements pipeline between individual map and reduce tasks, by allowing intermediate data produced by a map task to be pushed to each reduce task as soon as they are available. In other words, each map task stores its results into a local buffer, and each reduce's shuffle sub-task starts as soon as the buffer grows to a given threshold size. HOP also allows pipeline between jobs, by letting the reduce tasks of one job to pipeline their output directly to the map tasks of another job. We here focus on modeling Hadoop jobs, because of its great popularity. As future work, we intend to extend our model to capture the more complex precedence constraints that arise in HOP workloads.

4 Our Analytical Performance Model

This section presents the analytical model developed to predict the performance of Hadoop jobs. Section 4.1 describes how we extend the reference model [19] to capture the synchronization delays inherent to them. The identification of the precedence rules in the execution flow of a Hadoop job, required to build the precedence tree, is presented in Sect. 4.2. We also propose, in Sect. 4.3, a tighter strategy to estimate the response time of a subset of tasks.

4.1 Modeling Strategy

We propose an analytical model to estimate the average performance of Hadoop jobs. Our discussion is focused on average job response time, but we note that other performance metrics, such as throughput and resource utilization, are also computed using Little's Law [22].

Our modeling approach is based on a hierarchical model. The lower level model represents an architecture with n nodes, each with c CPUs and d disks, where disks are connected to the CPUs by a fiber channel and the nodes are inter-connected by a high-speed network. We model this architecture with a closed queuing network with service centers representing each CPU, each disk, the fiber channel and the network. Memory constraints are not explicitly modeled. The higher level model is given by a precedence tree that captures the synchronization and precedence constraints among tasks.

The workload is composed by a number N of jobs executing concurrently in the system. Each job has m map tasks and r reduce tasks. The numbers of map and reduce tasks that each worker node can execute in parallel are limited and given by the parameters pm and pr, respectively (in general, these parameters are set as the number of CPU cores of each node). Recall that, a partial sort is performed after each shuffle. Since these two phases are performed in sequence, we choose, for modeling purposes, to group each pair of shuffle and sort into a single sub-task named *shuffle-sort*. Similarly, after all partial sorts finish, a final sort, followed by the final phase that applies the reduce function are sequentially executed. We also group these two phases into a single sub-task named *merge*. Thus, in our model, each reduce is composed by m *shuffle-sort* sub-tasks and one *merge* sub-task. Moreover, up to ps shuffle-sort sub-tasks may be executed in parallel. We consider individual sub-tasks as the main workload unit. For that purpose, a map task can be also seen as sub-task. For the sake of clarity, through the rest of this paper, we refer to maps, shuffle-sorts and merges as simply *tasks* belonging to a Hadoop job.

Table 1 presents the input parameters of the analytical model, which can be divided into two groups: (1) architecture parameters, and (2) workload parameters. The architecture parameters are the number of nodes n, as well as the number of CPU's c and the number of exclusive disks d per node. The workload parameters are the number of tasks of each type (m and r), number of threads (per node) to process tasks of each type (pm pr), number of threads (within each reduce task) to process shuffle tasks (ps), and the service demand matrix (D_{ik}), with the demand of each task i in each center k.

We note that all map, shuffle-sort and merge tasks usually have non-zero service demands on the CPUs, disks and on the fiber-channel, as all these tasks have to retrieve/store data from/to the local disks as well as do some processing on it. Each shuffle-sort task, in turn, may also impose some demand on the high-speed network

Table 1 Analytical model input parameters

Notation	Input parameter
n	Node number
c	Number of CPUs per node
d	Number of disks per node
$D_{i,k}$	Mean service demand of task class i in center k
m	Number of map tasks
r	Number of reduce tasks
pm	Number of threads (per node) to process map tasks
pr	Number of threads (per node) to process reduce tasks
ps	Number of threads (per reduce task) to process shuffles

connecting the nodes, as it has to recover the results produced by a finished map task to process them. Such results, stored at the local disks of the node where the map executed, must then be transferred to the node where the corresponding reduce is executing. Note, however, that no transference over the network is required if the shuffle-sort is running on the same node of the map task. In this case, the shuffle-sort can retrieve the map results directly from the local disks. We capture that, in our model, by setting the demand on the network imposed by this shuffle-sort to zero. Note also that the transference of the results over the network is distributed along the execution of the job, as the shuffle-sorts are executed, thus reducing the chance of contention at the network.

The main challenge to develop our model is how to capture the synchronization delays introduced by the pipeline parallelism that occurs among maps and shuffle-sorts. Our solution is built from the reference model, described in Sect. 2.3. Recall that the reference model considers that the precedence tree is given as an input parameter. However, the task precedences of Hadoop jobs cannot be defined beforehand, preventing us to give the tree as input. For instance, a shuffle-sort task of each reduce initiates execution as soon as the first map task finishes. The moment when this occurs depends on the resource contention experienced by each map. That is, a map may take longer to finish if it executes concurrently with many other tasks, all of them sharing the same physical resources. Moreover, a shuffle-sort may block or not, depending on the dynamics of the execution of the other maps. Shuffle-sorts, executing on the same node, may also experience contention at the shared resources, and thus extra delays. The response time of the shuffle-sorts, in turn, determines the time when a merge task initiates. In other words, there are complex dependencies. The precedence tree depends on the response times of the individual tasks, which depend on the contention experienced by them. Resource contention depends on which tasks execute concurrently, which, in turn, depends on the precedence tree.

We address this issue by proposing an algorithm to dynamically build the precedence tree for a job, and add it as an extra step to the algorithm proposed in [19]. Our solution is shown in Fig. 2b. Note that we rebuild the precedence tree at each iteration of the algorithm, using the average response times of individual tasks computed in

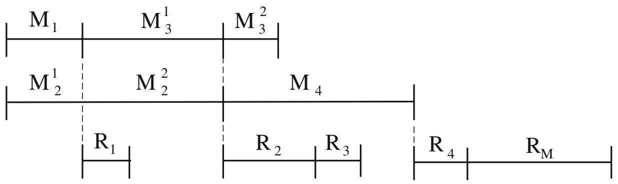

Fig. 5 Execution time line of an example HOP job

the previous iteration to build a new, possibly more accurate, precedence tree in the current iteration.[4]

Recall that our main goal with the precedence tree is to capture the execution flow of the job, expliciting the parallel/serial execution of individual tasks as well as their inter-dependencies, so that we can estimate the average response time of individual tasks and, by composition, of the whole job. Thus, there are two key challenges to dynamically build this precedence tree: (1) how to estimate when each task starts and finishes, and (2) how to estimate the average response time of the internal nodes of the tree. If we are able to estimate when each task starts and finishes, then we are able to define which tasks run concurrently, which tasks experience blocking due to synchronization, etc. Moreover, if we are able to estimate the average response time of groups of tasks running concurrently/sequentially, which, in the precedence tree, are captured by the internal nodes, then we can estimate the average job response time, our ultimate goal. These two key challenges will be discussed further in the next sections.

4.2 Modeling Pipeline Parallelism

Given that our model provides average estimates of the performance of a Hadoop job, the precedence tree should capture the synchronization delays and the parallelism experienced, on average, by the job tasks. The strategy adopted to build such tree is to start with a timeline of the execution of individual tasks, which is built based on estimates of each task's average response time and a set of rules defining the precedence constraints among different tasks. We first discuss, in high level, how the precedence tree can be built given this timeline in Sect. 4.2.1, and defer a description of how this timeline can be built to Sect. 4.2.2.

4.2.1 Building a Precedence Tree from a Job Execution Timeline

Figure 5 shows a simple example of a timeline representing the execution of a Hadoop job composed of $m = 4$ maps (M_1–M_4) and $r = 1$ reduce, running on $n = 1$ node. This node has $pm = 2$ threads to process the maps and $pr = 1$ thread to execute the reduce, and there is no shuffle parallelism (i.e., $ps = 1$). The reduce comprises four shuffle-sort tasks (R_1–R_4) and one merge task (R_M). The order at which the shuffle-sort tasks execute is defined by the order at which the map tasks finish. That is, R_i

[4] In the first iteration, we estimate the average response time of each task by the sum of its demands on all modeled resources.

should start as soon as M_i finishes and the reduce thread is idle. The merge task, in turn, can start only after all shuffle-sort tasks finish. For the sake of simplicity, we use this simple timeline to illustrate our strategy to build the precedence tree corresponding to it. We note however that the strategy can be applied to much more complex jobs running on larger infra-structures. In particular, even though the example shows the execution of shuffle-sort tasks at a specific order, we emphasize that there are no constraints on the order at which results of map tasks are consumed by shuffle-sort tasks. For instance, if M_3 finishes before M_2, then R_3 can start immediately, provided that R_1 has already finished (and the reduce thread is idle).

As shown in Fig. 5, the two map threads start execution immediately at the beginning of the job execution, whereas the reduce thread remains blocked. As soon as the first map finishes, the first shuffle-sort task (R_1) can initiate its execution, and another map task M_3 is assigned to the thread that was executing M_1. This point in time, shown in Fig. 5 by a dotted vertical line, marks a synchronization point when the set of tasks executing in parallel changes. From this point in time forward, M_3, M_2 and R_1 are executing. Note that, since M_3 starts executing only after M_1 finishes, there is a serial precedence between them. The same holds for R_1 and M_1. After R_1 finishes, the reduce thread blocks, waiting for the next map to finish. Once M_2 finishes, there is another synchronization point marked by the beginning of R_2's execution. The next synchronization point occurs only when M_4 finishes and R_4 starts executing. Note that R_3 can begin execution immediately after R_2 finishes, since M_3 has already finished by this time.

Thus, the execution of a Hadoop job can be seen as a series of synchronization points, each of which delimits the parallel execution of different sets of tasks. During each such “parallel phase”, a set of map and reduce tasks execute in parallel. Since the same map task may execute during multiple parallel phases (e.g., M_2 in Fig. 5), we use superscripts to distinguish between multiple phases of the execution of the same task (e.g., M_2^1 and M_2^2) and treat each such phase as an independent (sub-)task. Clearly, these (sub-)tasks must execute sequentially.

Given a timeline, as the one shown in Fig. 5, a corresponding precedence tree is derived as follows: each leaf node represents a map, shuffle-sort or merge (sub-)task (M_i and R_i). Internal nodes are either S or P_A operators, and the root represents the job. (Sub-)tasks representing phases of the execution of the same task, as well as tasks executed by the same thread, must run in sequence (S), whereas tasks executed by different threads during the same phase run in parallel (P_A).

The precedence tree is built by creating multiple sub-trees rooted by P_A operators, one for each parallel phase, and connecting them using S operators. Thus, each P_A-rooted sub-tree corresponds to a parallel phase. S nodes are added to represent the end of a phase, the serialization of maps dynamically allocated to the same thread, and the serial execution of the last shuffle-sort and the merge tasks of a given reduce. Figure 6 shows the construction of the precedence tree corresponding to the timeline shown in Fig. 5 after the nodes corresponding to each parallel phase have been added.

Generally speaking, the tree is built by mapping each synchronization point into an S-rooted sub-tree with two children: the left child is a sub-tree representing the previous phase, and the right child is either an S-operator for linking the sub-trees of the next phases, or the next phase sub-tree (for the final phase). Each P_A-rooted sub-tree

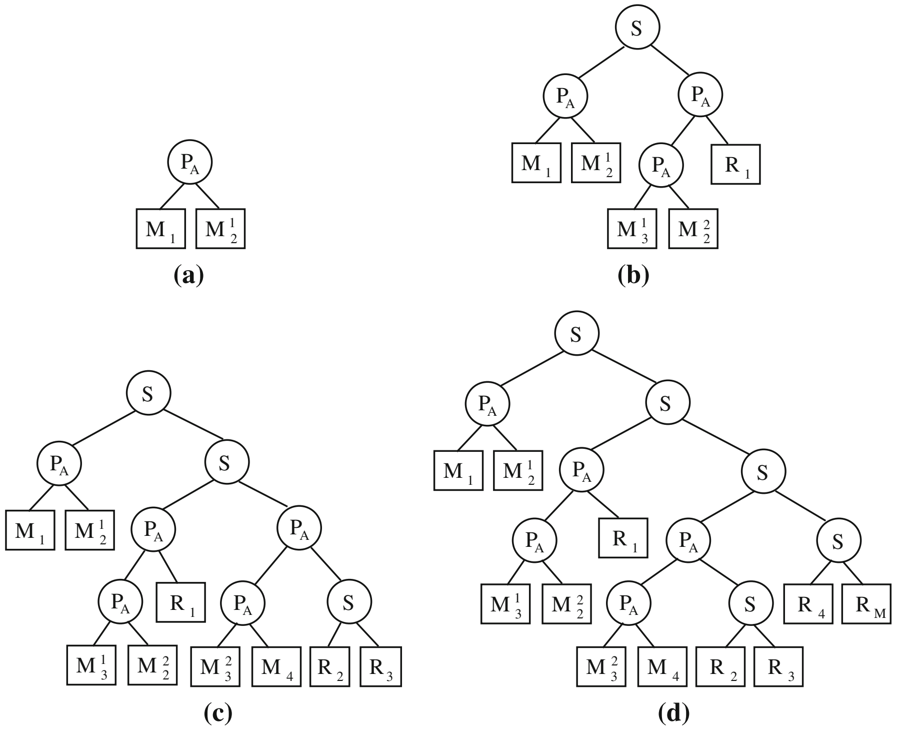

Fig. 6 The construction of a precedence tree for the execution timeline in Fig. 5. **a** Parallel phase 1, **b** parallel phases 1–2, **c** parallel phases 1–3, **d** complete tree

of a phase has two branches (i.e., children): one for the parallel map threads, at the left side, and one for the parallel reduce threads, at the right side. Each such branch may thus be a P_A-rooted sub-tree itself representing the parallel execution of multiple threads of a given type. S nodes may be added to represent the serial execution of multiple tasks of the same type (map, shuffle-sort or shuffle-sort and merge) by the same thread during the same parallel phase.

4.2.2 Building a Job's Execution Timeline

To define the execution timeline of a Hadoop job, we must identify events of termination of (sub-)tasks. Such events may trigger synchronization points and the beginning of the execution of other (sub-)tasks. That is, at each such event, a set of precedence rules, defined based on the pipeline among maps and shuffle-sorts and on the serial relationships among (sub-)tasks (e.g., maps dynamically allocated to the same thread), are applied to determine whether new tasks can start execution and whether there is blocking.

To capture the task synchronization, we assume that each reduce thread, which executes all shuffle-sorts and the merge of a single reduce, has a logical queue associated to it. Since each map is processed by all reduce threads, each map, after finishing execution, puts a “token” (with its identification) in the logical queue of each reduce to indicate that its results are ready to be processed by them. A shuffle-sort only starts executing if there is at least one token to be consumed at the logical queue of its corresponding reduce thread. Indeed, we control the execution of shuffle-sorts by letting all reduce threads start blocked. Whenever the first map task finishes and a new token is added to each logical queue, the reduce threads are unblocked and the first shuffle sort of each reduce can start execution. If the shuffle sorts finish before the maps that are currently running, the reduce threads block again, and remain blocked until the next map finishes and a new token is added to the logical queues. Whenever a shuffle-sort finishes, if there are available tokens in the logical queue, the reduce thread continues execution, running the next shuffle-sort(s). Note that, since tokens have the identification of the corresponding map tasks, we are able to determine the order at which the shuffle-sorts are executed by each reduce thread, which is required to build the job's execution timeline.

More specifically, the following events are considered:

1. **Map M_i is the next to finish**: this event occurs if the cumulative average response time of the thread executing M_i, defined as the sum of the average response times of all map tasks already executed by M_i's thread, is the smallest among the cumulative average response times of all executing map and reduce threads. If this event occurs, we insert a token corresponding to this map in the logical queues of all reduce threads.We must then check whether any reduce thread is blocked. If so, they must be unblocked, and new shuffle-sorts start being executed by the previously blocked reduce threads. The token is removed from the logical queue when the shuffle-sort starts. If ps is greater than 1, a number of shuffle-sorts equal to the number of tokens in the logical queue (but no greater than ps) are executed in parallel by each reduce. Moreover, if there are still maps waiting to execute, one of them is dynamically allocated to M_i's thread, which starts executing it. If there are blocked reduce threads, this event triggers the end of a parallel phase (i.e., synchronization point).
2. **Shuffle-sort S_i is the next to finish**: if S_i is the last shuffle-sort to be executed by its reduce thread[5], then the merge task starts execution. Otherwise, we check the logical queue. If it is not empty, then the next shuffle-sort starts execution and removes one token from its logical queue. Otherwise, if the logical queue is empty, the reduce thread blocks waiting for a map to finish.
3. **Merge R_{Mi} is the next to finish**: in this case the corresponding reduce thread terminates its execution.

The basic idea of the algorithm is to dynamically progress in the execution of the job by comparing the cumulative average response times of all map and reduce threads and defining events of termination of tasks. At each such event, we apply the rules described above to move forward in the timeline. At the same time, we add new nodes and sub-trees to the precedence tree corresponding to new (sub-)tasks and phases added to the timeline, as described in Sect. 4.2.1.

[5] We can determine that by keeping track of the number of tokens already processed by each reduce thread. When it reaches the number of map tasks, an input parameter, the current shuffle-sort is the final one.

Our algorithm takes as inputs the average response time of each task (M_i R_i and R_{M_i}), the number of threads per type of task (pm pr and ps), as well as the numbers of map tasks (m) and reduce tasks (r). The algorithm builds the tree from top to bottom as it progresses in the timeline.

Note that the precedence tree is recomputed at each iteration of the algorithm, so as to take the average response times of individual tasks computed in the previous iteration into account when building the timeline. The time complexity to build the precedence tree at each iteration depends on how, on average, all the map and reduce tasks are scheduled by the JobTracker to the WorkerNodes. The procedure to build the precedence tree is based on the (average) execution timeline of a Hadoop job (as described in Sect. 4.2.1). The processing cost to traverse this timeline is given by the time required to repeatedly search, in all running threads, for the next task to finish, until the termination of all tasks. In each of the n nodes, there are pm threads to execute map tasks and pr threads to run reduce tasks. Moreover, each reduce thread is further divided into ps threads to execute parallel shuffles. Let C be the total number of tasks in the timeline and T be the total number of threads in execution. C is equal to the sum of all map tasks plus all shuffle-sort tasks plus all merge tasks. Considering that each reduce task is divided into m shuffle-sort tasks and 1 merge task, C is given by $m + r \times (m + 1)$. The total number of threads T is given by $n \times (pm + pr \times ps)$. Thus, in the worst case, the time complexity to build the precedence tree at each iteration is given by the search for $m + r \times (m + 1)$ tasks in $n \times (pm + pr \times ps)$ threads, that is, $O(C \times T) = O([m + r \times (m + 1)] \times [n \times (pm + pr \times ps)])$. Note however that the computational cost of the whole solution is dominated by the approximate Mean Value Analysis algorithm, which has complexity $O(C^2 \times K) = O([m + r \times (m + 1)]^2 \times K)$, where K is the number of service centers. Recall that K increases linearly with the number of nodes. Thus, the time to solve the aMVA dominates the time complexity of our analytical solution.

4.3 Average Response Time Estimates

Once the precedence tree is built (step 2 of Fig. 2b), the next step is to estimate the average job response time. As described in Sect. 2.3, the average job response time is derived by performing a depth-first traversal in the precedence tree, from leaves (tasks) to the root (job), estimating the average response time of the internal nodes according to its precedence. The main challenge is how to estimate the synchronization time among a set of parallel tasks, i.e., the average response time associated with internal P_A-nodes in the precedence tree. Note that this corresponds to estimating the average of the distribution of the maximum of k random variables, which is not the same as taking the maximum of the averages of the variables. Indeed, we did verify, in preliminary experiments, that using the latter leads to large approximation errors. Thus, we consider two alternative strategies to estimate the average job response time:

1. **Tripathi-based**: this strategy corresponds to the approach adopted by the authors of the reference model [19] (see Sect. 2.3 for more details). To estimate the response time of a P_A-rooted sub-tree, we approximate the distribution of response time of each of its children by either an Erlang or a hiperexponential distribution,

depending on the coefficient of variation (CV) of the response times associated with each child node. We then compute the average and variance (and thus CV) of the maximum of the selected distributions (e.g., average and variance of the maximum of two Erlang distributions). Computed averages and variances (and thus CVs) are used in the computation of the average response time of other internal nodes that are located higher in the tree.

2. **Fork/join-based**: we consider the execution of a parallel-phase as a fork-join block, and use previously adopted estimates of the average response time of fork/joins [29]. One such estimate is the product of the k^{th} harmonic number ($H_k = \sum_{i=1}^{k} \frac{1}{i}$) by the maximum average response time of k tasks. For instance, given the sub-tree depicted in Fig. 6a, first we estimate the synchronization time among these two tasks ($k = 2$) by the 2nd harmonic number (H_2). Then we compute the maximum of the average response time of the P_A's leaves (M_1 and M_2^1). Finally, we multiply H_2 by $\max(M_1, M_2^1)$ to compose the response time of the P_A-rooted sub-tree.

For both approximations, we assume that task response times are exponentially distributed, as in [19]. As a final note, recall that in a real system, the first shuffle-sort starts when the first map finishes. Thus, the smallest response time of all running maps defines when the shuffle-sort starts. Since we build the timeline and the precedence tree based on average response times, in order to estimate the starting time of a shuffle-sort, we must take the average of the minimum of the response times of all running maps. This corresponds to taking the average of the minimum of different exponential distributions. Once again, this is not the same as taking the minimum of the averages.

5 Experimental Analysis

This section presents representative results of a set of experiments we performed with the proposed analytical performance model. Section 5.1 describes the experimental setup, whereas Sect. 5.2 presents validation results from a comparison of our model against event-driven simulation and measurements of a real setup. Finally, in Sect. 5.3, we evaluate model accuracy and ultimately system performance as key Hadoop parameters, notably the number of nodes, the number of map threads per node, and the number of shuffle threads per reduce task, are varied over a wide range of possible values.

5.1 Experimental Setup

In this section, we describe the experimental setup used to validate our analytical model. We performed experiments with both a real Hadoop setup and a queuing network simulator. In both cases, we consider a platform with n nodes, each with c CPUs and d disks. Moreover, we assume a multiprogramming level N equal to 1, considering a single job in the system at a time. We choose to focus, in our experiments, on the scenario with a single job so that we can better assess how accurately our proposed model is able to capture the synchronization delays between map and reduce threads, our main goal. Moreover, in general Hadoop clusters are used to run jobs with massive parallelism. Thus, there is often not enough resources to run more than one job concurrently. We note however, that despite our choice of $N = 1$, our proposed solution can be directly used with larger values of N, because, like the reference model, our model does capture the contention for resources among threads belonging to different jobs in the inter-job overlap factor. Thus there is no assumption that constraints this parameter in our model.

We consider a Hadoop job with m map tasks and r reduce tasks. The maps are dynamically allocated among the nodes, whereas each reduce runs on a separate node, which implies that the number of parallel reduce threads (pr) is set to 1. We varied the numbers of parallel map (pm) threads and parallel shuffle (ps) threads running on each reduce task.

Our real Hadoop setup runs on a platform containing n equal to 3 nodes, each one with c equal to 4 CPU and d equal to 1 dedicated disk. The disk on each node is connected to the CPU by a 1 Gbps fiber channel, and nodes are inter-connected by a **1** Gbps network. Each node runs the Hadoop open source framework [2], and the disks are divided into 2 partitions: one for the local file system and the other for the shared HDFS. Since our model considers that memory has no constraints, we configured the Java virtual machines running on the Hadoop nodes to use the maximum memory available.

The selected application to run on the Hadoop system was *sorting*, since it is widely applied to evaluate the scalability of MapReduce platforms [6,7,35]. We consider a Hadoop job that sorts a file, uniformly partitioned across the nodes, containing 100,000,000 registers, each register with 100 bytes. The job is composed of m equal to 150 map tasks and r equal to 3 reduce tasks. Its execution flow is as follows: each map task extracts the key from each input register, and writes an output register with the sorted key and the remaining data into a temporary file. Each reduce task is responsible for writing all values of its range of keys into the output file.

For each configuration consisting of the selected values for pm and ps, we ran experiments with a single sorting job running on the Hadoop platform, monitoring the execution of the several tasks running on each node. During each experiment, we measured for each (sub-)task: (1) the effective CPU execution time, in nanoseconds; (2) the elapsed time, in nanoseconds; (3) the total amount of bytes read and written in the local file system; (4) the total number of bytes read and written in the HDFS; (5) the total number of bytes transferred over the fiber channel; and (6) the total number of bytes transferred over the network. We used these measured values and the reported speed of each device (CPU, disk, network and fiber channel) to compute the average service demand of each individual (sub-)task on each device.

These average demands, in seconds, are shown in Table 2. Note that a large fraction of the job's average response time is spent in the merge task (mainly on the disk, which is the bottleneck device), which sorts the registers of a key range (REDUCE function). Recall that the network is used only by the shuffle-sort tasks to transfer the data produced by a map from its original node to the node where the corresponding reduce thread is running. The average network demands shown in Table 2 were computed for cases when map and shuffle-sort tasks run on different nodes. If both tasks run on the same node, the average network demand is set to zero, as previously mentioned.

Table 2 Average service demands (in seconds) of each (Sub-)task on each device

pm	*ps*	Task	CPU	Fiber	Disk	Network
1	1	Map	5.0764	0.0677	3.1844	0.0000
		Shuffle-sort	0.5217	0.0087	0.4082	0.1850
		Merge	37.4983	1.2583	59.2157	0.0000
1	4	Map	5.1360	0.0677	3.1847	0.0000
		Shuffle-sort	2.9440	0.0425	2.0000	0.9067
		Merge	20.4000	1.2580	59.2160	0.0000
4	1	Map	5.1057	0.0677	3.1844	0.0000
		Shuffle-sort	0.6608	0.0087	0.4082	0.1850
		Merge	47.2580	1.2580	59.2157	0.0000
4	4	Map	5.0764	0.0677	3.1844	0.0000
		Shuffle-sort	2.5563	0.0425	2.0000	0.9067
		Merge	38.4020	1.2580	59.2160	0.0000

Moreover, recall that our model is iterative, with a convergence test ruled by parameter ϵ. In our experiments, we set ϵ equal to 10^{-4}.

We also developed a simulator of the Hadoop platform which allowed us to validate our model for a larger variety of scenarios and test its assumptions (e.g., exponentially distributed task response times). We built an event-driven queuing network simulator that reproduces the dynamics of the execution of a job in terms of contention for resources and synchronization constraints. We do not simulate the details of each device. Instead, we choose to represent each device as a queuing center, and focus rather on the workload dynamics. In particular, unlike the analytical model, which is based on average estimates, the simulator reproduces the execution of individual tasks, thus capturing the dynamic aspects of the workload (e.g., individual blocking events, burst arrivals, etc), which are only approximated by our analytical model.

We used the average service demands measured in the real setup, shown in Table 2, as input to both the analytical model and the simulator. In particular, for the simulator, we assumed that task residence times at each center are exponentially distributed with averages given by the measured service demands.

The parameters used in the simulation and real setup are shown in Table 3. The main differences are in the numbers of nodes and map tasks. In the simulation experiments, we set n equal to 4 nodes, varying the number of map tasks as multiples of n. We note however that, although we vary the number of map tasks, we do keep the amount of work per task (i.e., the task demands) fixed. Thus, a larger number of map tasks corresponds to a larger input file to be sorted. By doing so, we are generating a larger number of tasks, thus allowing for more parallelism among map and shuffle-sort tasks.

5.2 Model Validation

We now discuss the validation of our analytical model by comparing its results against those obtained with simulation (Sect. 5.2.1) and measurements of the real setup

Table 3 Parameters of the simulator and Hadoop setup

Notation	Parameters	Real setup	Simulation
n	Node number	3	4
c	Number of CPUs/node	4	4
d	Number of disks/node	1	1
m	Number of maps	150	4, 8, 16, 32, 64, 128, 256
r	Number of reduces	3	4
pm	# Maps in parallel/node	1, 4	1, 4
pr	# Reduces in parallel/node	1	1
ps	# Shuffle in parallel/reduce	1, 5	1, 4

(Sect. 5.2.2). In each case, we parameterized our analytical model with system parameter values as defined in Table 3, and average service demands as shown in Table 2. In other words, in each case, we parameterized our model with the same values as those of the system (simulator or real setup) used as basis for comparison.

5.2.1 Validation Against Simulation

We start by presenting our validation against simulation results. In each considered scenario, simulation results are averages of 5,000 executions. Confidence intervals, which are omitted to improve graph readability, indicate that results deviated from the reported averages by at most 1.04 %, with 95 % confidence.

Figure 7 presents the average job response times estimated by our models and by the simulator as we vary the number of map tasks in the following four scenarios: (a) with neither map nor shuffle intra-node parallelism ($pm = ps = 1$), (b) without map intra-node parallelism, but with shuffle intra-node parallelism ($pm = 1, ps = 4$), (c) with map intra-node parallelism, but without shuffle intra-node parallelism ($pm = 4, ps = 1$), and (d) with both map and shuffle intra-node parallelisms ($pm = ps = 4$). Results for the two analytical approaches to estimate the average response time of a parallel phase are shown. Recall that, as we increase the number of map tasks, we are indirectly increasing the level of parallelism among maps and shuffles, since the total amount of work per map task is kept fixed.

Similarly, we also show in Fig. 8 average CPU and disk utilizations estimated by our model and measured during simulation for the same four scenarios. We show average results computed across all devices of the same type (CPU or disk). We omit the fiber channel and network utilization results, as these are much lower than the CPU and disk utilizations. Thus, we focus on the resources with larger utilizations, where contention may significantly impact task and job response times. We also focus only on the analytical results produced by the fork/join approximation, omitting the results produced by the approximation proposed by Tripathi et al. [19], to improve graph readability.

We first discuss the results of average job response time produced in the first scenario, with no intra-node parallelism of map and shuffle tasks, depicted in Fig. 7a. We

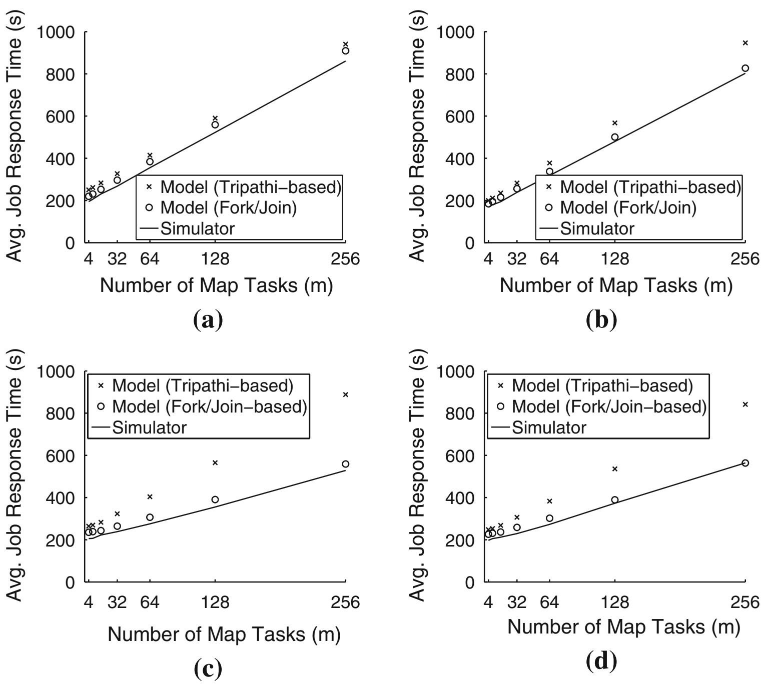

Fig. 7 Validation against simulation: average job response times. **a** $pm = 1$, $ps = 1$, **b** $pm = 1$, $ps = 4$, **c** $pm = 4$, $ps = 1$, **d** $pm = 4$, $ps = 4$

find that, in this scenario, the maximum relative difference between analytical results and simulation results is 29 %, when the average response time of parallel phases is estimated using the Erlang approximation proposed by Tripathi et al. [19]. However, the use of the fork/join approximation, proposed by us, leads to much better results: the maximum relative difference drops to below 13 %. For both approaches, the maximum relative difference occurs for a number of map tasks m equal to 1. Moreover, we note that the relative differences tend to decrease, implying a better agreement between analytical models and simulation, as the number of map tasks increases.

The fork/join-based model also produces very good agreement with the simulator in terms of disk and CPU utilizations, as shown in Fig. 8a. The maximum relative difference is only 12 %. In contrast, the Tripathi-based model produces disk and CPU utilization estimates that diverge from simulation results by as much as 22 %. For both models, the relative differences are larger for disk utilization, as the disks are the bottleneck.

We now turn to the second scenario, in which the number of parallel maps running on each node is fixed at 1, but we increase the number of parallel shuffles to 4. This implies that each reduce task may process the results of up to 4 map tasks in parallel, using 4 threads.Once again, Figs. 7a and 8b show very good agreement, in terms of average job response time and resource utilizations, between both analytical models and simulation, although the fork/join approximation clearly leads to more accurate estimates. For instance, the maximum difference between the average job

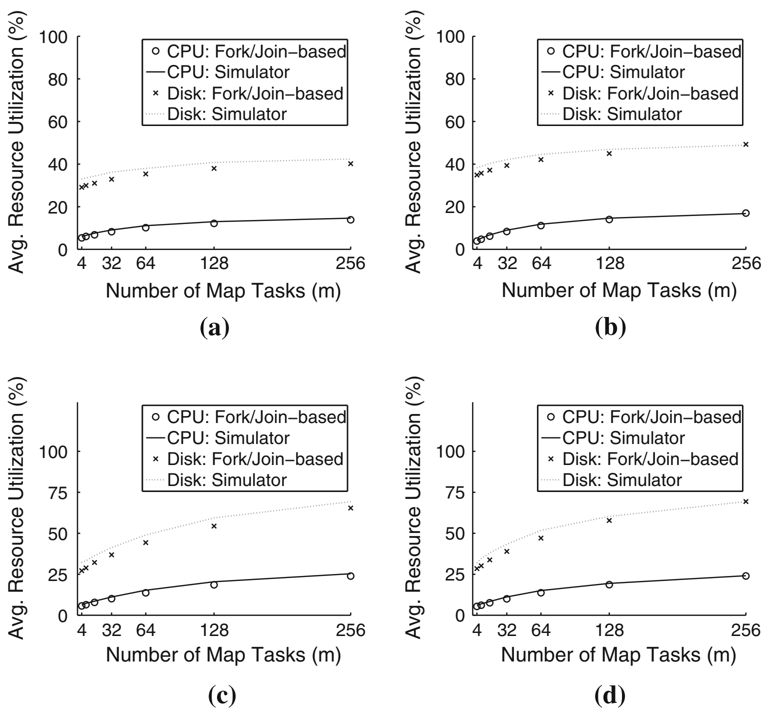

Fig. 8 Validation against simulation: average CPU and disk utilizations. **a** $pm = 1$, $ps = 1$, **b** $pm = 1$, $ps = 4$, **c** $pm = 4$, $ps = 1$, **d** $pm = 4$, $ps = 4$

response times estimated by the analytical model and measured during simulation is around 12 % for the Tripathi-based model, but only 4.6 % for the fork/join-based model. Also, the maximum difference in the disk utilization estimates drops from 15.8 % (Tripathi-based model) to 9.1 % (fork/join-based model). It is also worth mentioning that the increase in the shuffle parallelism contributes to a reduction of the average job response time, as expected. Indeed, by setting ps to only 4, we observe a reduction in average job response time of as much as 8.2 %.

In scenario 3, we let 4 maps run in parallel on each node ($pm = 4$), but allow no intra-node shuffle parallelism ($ps = 1$). Results are shown in Figs. 7c and 8c. Note that, in this scenario, map tasks face more contention on shared resources (particularly disks) which imply in longer queuing delays. We note that the fork/join-based model produces average job response time estimates that are in excellent agreement with simulation results, with a maximum relative difference under 15.3 %. In contrast, the Tripathi-based approach greatly over-estimates these measures, with a maximum relative difference reaching 68.1 %. Similarly, the fork/join-based model also produces reasonably accurate CPU and disk utilization estimates: they deviate from the simulation results by as much as 15.6 % (for $m = 4$).

Similarly, if we allow both shuffle and map intra-node parallelisms ($pm = 4$, $ps = 4$), once again we see that the fork/join-based model produces much more accurate estimates than the Tripathi-based model. Whereas the latter overestimates the average

Table 4 Validation against measurements of a real setup: average job response time

pm	*ps*	Measured	Tripathi-based model	Fork-join-based model
1	1	722.23	798.56	732.56
1	5	605.40	701.75	659.13
4	1	321.77	620.01	365.51
4	5	325.58	597.81	356.20

job response time (and thus resource utilizations) by as much as 49 %, the fork/join-based model produces results that are very accurate (relative difference under 12 %). Thus, we find that the fork-join approximation provides an improvement, quite significant in some scenarios, over the approach proposed by Tripathi and Liang and explored in the reference model. We also note that, in contrast with the scenario when $pm = 1$, allowing intra-node shuffle parallelism does not have a significant impact on average job response time (only 2.61 %) when there is already parallelism among map tasks, because the increase in shuffle parallelism also leads to an increase in resource contention (mainly in disk, the bottleneck).

As a final note, we also found that the average response times of individual tasks are also reasonable well estimated, with a maximum relative difference across all 4 scenarios below 13 and 7 %, for maps and merges, respectively, considering the fork/join approximation. The average response times of shuffle-sort tasks, on the other hand, were underestimated by as much as 43 %, due to underestimated fiber and network utilizations. However, given that in the measured real setup, shuffle-sort tasks demand much fewer resources than the other two types of tasks, the impact of such underestimate on the average job response time was only marginal. Nevertheless, we intend to further investigate how to improve our model to more accurately estimate the shuffle-sort average response time in the future.

5.2.2 Validation Against Measurements of a Real Setup

We now discuss validation results against measurements of the real Hadoop setup described in Sect. 5.1. Table 4 shows results for each combination of *pm* and *ps* values considered, presenting the average job response time estimated by both analytical approaches—the Tripathi-based and the fork/join-based models—and measured in the real system.

When there is neither map nor shuffle intra-node parallelism ($pm = ps = 1$), the Tripathi-based and fork/join-based analytical models produce estimates of average job response time that are only 11 and 1 %, respectively, larger than the average job response time measured in the real HOP setup. If we set *ps* equal to 5, the estimates produced by both models are also reasonably accurate, although the relative differences are somewhat larger (16 and 9 % for the Tripathi-based and fork/join models, respectively). Thus, consistently with our findings in the comparison against simulation, both models produce reasonably accurate estimates in these scenarios, although our fork/join model does produce some accuracy improvements over the original

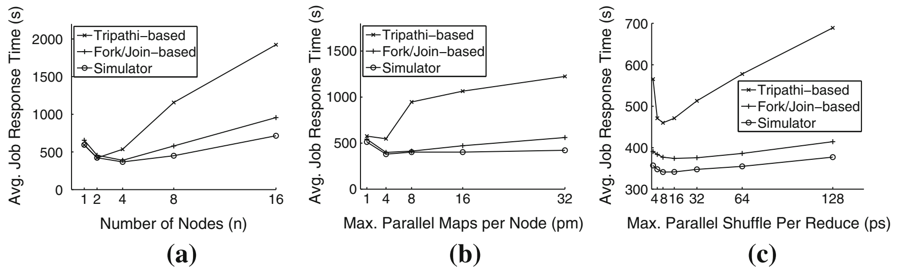

Fig. 9 Impact of key Hadoop parameters on system performance. **a** Number of nodes ($m = 128$, $pm = 4$, $ps = 4$), **b** # parallel map threads ($n = 4$, $m = 128$, $ps = 4$), **c** # parallel shuffle threads ($n = 4$, $m = 128$, $pm = 4$)

Tripathi-based model. Also consistently with our simulation results, we do observe a reduction in the average job response time (by 16.2 %) when the number of shuffle threads increases, provided that no intra-node map parallelism is allowed.

However, when there is intra-node parallelism among map tasks ($pm = 4$), the improvements of our fork/join approach over the original Tripathi-based model are really impressive. For instance, when we set $pm = 4$ and $ps = 1$ the Tripathi-based model over-estimates the average job response time by 93 %. In contrast, our fork/join-based model produces estimates that are only 14 % off the measured values. If we further increase ps to 5, the estimates produced by the Tripathi-based model are 84 % larger than the measured values, whereas the fork/join-based model produces results that are only 9 % larger. These results indicate that, consistently with our discussion in Sect. 5.2.1, the fork/join-based model produces much more accurate estimates, particularly when we allow for parallelism among maps running on the same node. Also consistently with our simulation results, we do observe a great reduction (by as much as 55 %) in the average job response time if intra-node map parallelism is allowed.

It is worth-mentioning that, just like in our comparison against simulation, both analytical models produce average response time estimates that are larger than the measured values in all considered scenarios. In other words, both models produce conservative estimates in comparison with both simulation and measurement results in these scenarios. Conservative estimates, provided that they are within reasonable accuracy, like the ones produced by our fork/join-based model, are a nice property to support effective SLA-driven capacity management and planning decisions [22].

5.3 Impact of Key Parameters on Model Accuracy and System Performance

In this section, we extend our validation experiments by evaluating the impact of the number of nodes (n), the number of map threads (pm) and the number of shuffle threads (ps) running in parallel on each node on the average job response time. Results are shown in Fig. 9. To further support our validation effort, we compare the analytical results produced by the Tripathi-based and the fork/join-based models against simulation results.

We start by noting that, as shown in Fig. 9a–c, the fork/join-based model produces estimates of average job response time that are, in most cases, much more accurate than the Tripathi-based model. Indeed, the estimates produced by the fork/join-based model deviate from simulation results by typically at most 25 %, and by no more than 34 % (maximum difference observed for number of nodes $n = 16$). In contrast, the Tripathi-based model greatly over-estimates the average job response time, exceeding simulation results by as much as 180 % (for $pm = 32$). In only a few cases, notably for small number of nodes (below 4), both analytical approaches produce comparable estimates. Moreover, we also note that the fork/join-based model better captures the impact of each parameter—n pm and ps—on the average job response time, since the curves match the simulation curves much more closely than the Tripathi-based curves. Furthermore, once again, we note that the analytical models produce conservative estimates across all considered scenarios.

We now discuss the impact of each analyzed parameter on system performance. We start by focusing on the results for varying number of nodes n (Fig. 9a). For these experiments, we fixed the number of map tasks $m = 128$, the number parallel map threads $pm = 4$, and the number of parallel shuffle threads $ps = 4$. Note that, as we increase the number of nodes, the average job response time decreases somewhat as more parallelism among the threads is allowed. This was expected. For number of nodes up to 4 ($n \leq 4$), since each node has four CPUs (which led us to set $pm = 4$), the processing rate of map tasks is not able to fully utilize the four CPUs of each node. Thus, the tasks will not queue up to use a CPU. However, as the number of nodes increases beyond 4, the average job response time starts increasing. This is because an increase in the number of nodes leads to an increase in the number of reduces, since we assume one reduce runs on each node. Recall that, reduce tasks should process the results of all map tasks. Since reduce tasks, particularly the merge sub-tasks, have very large resource demands (see Table 2), as the number of reduces increases, so does the total amount of work to be performed, and thus the average job response time.

Figure 9b shows results for increasing values of pm, while keeping $n = 4$, $ps = 4$ and $m = 128$. Note that, by increasing pm, we are increasing the level of parallelism among map tasks. Indeed, we used the demands shown in Table 2 as inputs to our model for scenarios with pm equal to 1 and 4. For larger values of pm, we estimated resource demands by extrapolating those values. That is, we estimate each resource demand for $pm = 8$ by increasing the demands measured for $pm = 4$ by the same proportion observed when comparing the resource demands for $pm = 1$ and $pm = 4$. We did the same for the other values of pm. Since each node has four CPUs, when $pm = 1$ the CPUs are underutilized. Thus, as pm increases up to 4, the level of parallelism increases without incurring in contention in the shared resources. Thus, average job response time decreases. However, as pm increases beyond 4, the average job response time starts increasing very slightly, as more parallel threads generate more resource contention, which leads to longer queuing delays. However, the increase is only marginal in the range of pm values analyzed.

Finally, Fig. 9c shows results of the impact of the number of parallel shuffle-sorts (ps) on the average job response time, when we set $n = 4$, $m = 128$ and $pm = 4$. Like we did for evaluating the impact of pm, we used the average demands taken from Table 2 to estimate average job response time for ps equal to 1 and 4. For larger values of ps, we estimated the resource demands of each task by extrapolating the measured values (keeping the same increase ratio for each resource).For values of ps smaller than the number of CPUs per node (4), an increase in the number of shuffle-sorts running in parallel, and thus in the pipeline parallelism between maps and shuffle-sorts, improves system performance, reducing the average job response time.For values of ps greater than the number of CPUs per node, the parallelism among maps and shuffle-sorts starts causing queuing delays, which in turn, increases the average job response time.

6 Conclusions and Future Work

In this paper, we addressed the challenge of modeling Hadoop workloads, which exhibit intra-job precedence constraints and synchronization delays. The modeling approach extends the solution proposed in [19] which is based on a hierarchical model where the execution flow of a parallel application is represented by a precedence tree and the contention at the physical resources are captured by a closed queuing network. Our main contributions are a method to automatically build a precedence tree for Hadoop jobs that captures the synchronization delays introduced by the communication between map and reduce tasks, and a tighter and more accurate approach to estimate the average response time of parallel phases of the job execution. We validated our model against both an event-driven queuing network simulator and measurements of a real Hadoop system. In particular, the proposed fork/join approach to estimate the average response time of a parallel phase led to results that are within 15 % of the values measured in the real system.

As future work, we plan to extend our model to address other types of resources (e.g., main memory) as well as other workload models, such as open workloads, which are characterized by a job arrival rate instead of the number of concurrent jobs in the system. We also intend to extend our model to address the more complex precedence constraints introduced by the pipeline between individual map and reduce tasks as well as between different jobs, implemented in the Hadoop Online Prototype [5].

Acknowledgments This work was partially supported by the INCT-Web (MCT/CNPq grant 57.3871/2008-6), and by the authors scholarships from CNPq, FAPEMIG and CAPES. We are also grateful for the fruitful collaboration with HP Laboratories.

References

1. Apache Software Foundation, Powered by Hadoop. URL http://wiki.apache.org/hadoop/PoweredBy. Access date: 1 July 2012 (2012)
2. Apache Software Foundation, Official Apache Hadoop Website. URL http://hadoop.apache.org/. Accessed date: 1 July 2012 (2012)
3. Berlińska, J., Drozdowski, M.: Scheduling divisible MapReduce computations. J. Parallel Distrib. Comput. **71**(3), 450–459 (2011)
4. Chen, Y., Ganapathi, A., Griffith R., Katz, R.: The case for evaluating MapReduce performance using workload suites. In: Proceedings of the 2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), Singapore, pp. 390–399 (2011)

5. Condie, T., Conway, N., Alvaro, P., Hellerstein, J.M., Elmeleegy, K., Sears R.: MapReduce Online Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI), San Jose, California, pp. 21–35 (2010)
6. Dean, J., Ghemawat, S., MapReduce : Simplified data processing on large clusters. In: Proceedings of Operating Systems Design and Implementation (OSDI), San Francisco, California, pp. 137–150 (2004)
7. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
8. Ganapathi, A.: Predicting and Optimizing System Utilization and Performance via Statistical Machine Learning. Technical Report UCB/EECS-2009-181. EECS Department, University of California, Berkeley (2009)
9. Ganapathi, A., Kuno, H., Dayal, U., Wiener, J., Fox, A., Jordan, M., Patterson, D. : Predicting multiple metrics for queries: better decisions enabled by machine learning. In: Proceedings of the 2009 IEEE International Conference on Data Engineering (ICDE), Shanghai, China, pp. 592–603 (2009)
10. Herodotou, H.: Hadoop Performance Models. Technical Report CS-2011-05. Computer Science Department, Duke University. URL http://arxiv.org/abs/1106.0940 (2011)
11. Jain, R.: The Art of Computer Systems Performance Analysis—Techniques for Experimental Design, Measurement, Simulation, and Modeling. Wiley, London (1991)
12. Jiang, D.R., Ooi, B.C., Shi, L., Wu, S.: The performance of MapReduce: an in-depth study. Proc. VLDB Endow **3**(1–2), 472–483 (2010)
13. Jonkers, H.: Queueing Models of Parallel Applications: The Glamis Methodology, Computer Performance Evaluation: Modeling Techniques & Tools (LNCS 794). pp. 123–138. Springer, Berlin (1994)
14. Kim, S., Won, J., Han, H., Eom, H., Yeom, H.Y.: Improving hadoop performance in intercloud environments ACM SIGMETRICS. Perform. Eval. Rev. **39**(3), 107–109 (2011)
15. Krevat, E., Shiran, T., Anderson, E., Tucek, J., Wylie, J.J. , Ganger, G.R.: Applying Performance Models to Understand Data-intensive Computing Efficiency. Technical Report CMU-PDL-10-108. Carnegie Mellon University, Pittsburgh (2010)
16. Kruskal, C.P., Weiss, A.: Allocating independent subtasks on parallel processors. IEEE Trans. Softw. Eng. **11**(10), 1001–1016 (1985)
17. Lavenberg, S., Reiser, M.: Stationary state probabilities at arrival instants for closed queueing networks with multiple types of customers. J. Appl. Probab. **17**(4), 1048–1061 (1980)
18. Lee, K.H., Lee, Y.J., Choi, H., Chung, Y.D., Moon, B.: Parallel data processing with MapReduce: a survey. ACM SIGMOD Record J. **40**(4), 11–20 (2011)
19. Liang, D.R., Tripathi, S. K.: On performance prediction of parallel computations with precedent constraints. IEEE Trans. Parallel Distrib. Syst. **11**(5), 491–508 (2000)
20. Little, J.: A proof for the queuing formula: $L = \lambda W$. Oper. Res. **9**(3), 383–387 (1961)
21. Mak, V.W., Lundstrom, S.F.: Predicting performance of parallel computations. IEEE Trans. Parallel Distrib. Syst. **1**(3), 257–260 (1990)
22. Menasce, D., Dowdy, L., Almeida, V.: Performance by Design: Computer Capacity Planning By Example. Prentice Hall PTR (2004)
23. Morton, K., Balazinska, M., Grossman, D.: ParaTimer: a progress indicator for MapReduce DAGs. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD), Indianapolis, Indiana, pp. 507–518 (2010)
24. Pavlo, A., Paulson, E., Rasin, A., Abadi, D., DeWitt, D., Madden, S., Stonebraker, M.: A Comparison of approaches to large-scale data analysis. In: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD), Providence, Rhode Island, pp. 165–178 (2009)
25. Reiser, M., Lavenberg, S.S.: Mean-value analysis of closed multichain queuing networks. J. ACM **27**(2), 313–322 (1980)
26. Salza, S., Lavenberg, S.S.: Approximating response time distributions in closed queueing network models of computer performance. In: Proceedings Performance, North Holland, Amsterdam, pp. 133–145 (1981)
27. Thomasian, A., Bay, P.F.: Analytic queueing network models for parallel processing of task systems. IEEE Trans. Comput. **35**(12), 1045–1054 (1986)
28. Trivedi, K.S.: Probability and Statistics with Reliability, Queuing and Computer Science Applications. Prentice Hall PTR, Upper Saddle River (1882)

29. Varki, E.: Mean value technique for closed fork-join networks. In: Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, Atlanta, Georgia, pp. 103–112 (1999)
30. Vianna, E.,Comarela, G., Pontes, T., Almeida, J., Almeida, V., Wilkinson, K., Kuno, H., Dayal, U.: Modeling the performance of the Hadoop online prototype. In: Proceedings of the 23rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), Vitória, Brazil, pp. 152–159 (2011)
31. Wang, G., Butt, A.R., Pandey, P., Gupta, K.: A simulation approach to evaluating design decisions in MapReduce setups. In: Proceedings of the IEEE International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), Imperial College London, UK, pp. 1–11 (2009)
32. Wang, G., Butt, A.R., Pandey, P., Gupta, K.: Using realistic simulation for performance analysis of MapReduce setups. In: Proceedings of the 1st ACM Workshop on Large-Scale System and Application Performance (LSAP), Munich, Germany, pp. 19–26 (2009)
33. Weng, N., Wolf, T.: Analytic modeling of network processors for parallel workload mapping. ACM Trans. Embed. Comput. Syst. **8**(3), 18:1–18:29 (2009)
34. White, T.: Hadoop—The Definitive Guide: Storage and Analysis at Internet Scale. 2nd edn. O'Reilly Media, Sebastopol (2011)
35. Yang, H.C., Dasdan, A., Hsiao, R.L., Parker, D.S.: Map-Reduce-Merge: simplified relational data processing on LargeClusters. In: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD), Beijing, China, pp. 1029–1040 (2007)
36. Yang, X., Sun, J.: An Analytical performance model of MapReduce. In: Proceedings of the 2011 IEEE International Conference on Cloud Computing and Intelligence Systems (CCIS), Beijing, China, pp. 306–310 (2011)
37. Zaharia, M., Konwinski, A., Joseph, A.D., Katz, R., Stoica, I.: Improving MapReduce performance in heterogeneous environments. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI), San Diego, California, pp. 29–42 (2008)
38. Zahorjan, J.: The Approximate Solution of Large Queueing Network Models, PhD. Thesis, University of Toronto, Canada (1980)