# MapReduce Performance Models for Hadoop 2.x

Daria Glushkova
Universitat Politècnica de Catalunya
Barcelona, Spain
dglushkova@essi.upc.edu

Petar Jovanovic
Universitat Politècnica de Catalunya
Barcelona, Spain
petar@essi.upc.edu

Alberto Abelló
Universitat Politècnica de Catalunya
Barcelona, Spain
aabello@essi.upc.edu

## ABSTRACT

MapReduce is a popular programming model for distributed processing of large data sets. Apache Hadoop is one of the most common open-source implementations of such paradigm. Performance analysis of concurrent job executions has been recognized as a challenging problem, at the same time, that it may provide reasonably accurate job response time at significantly lower cost than experimental evaluation of real setups.

In this paper, we tackle the challenge of defining MapReduce performance models for Hadoop 2.x. While there are several efficient approaches for modeling the performance of MapReduce workloads in Hadoop 1.x, the fundamental architectural changes of Hadoop 2.x require that the cost models are also reconsidered. The proposed solution is based on an existing performance model for Hadoop 1.x, but it takes into consideration the architectural changes of Hadoop 2.x and captures the execution flow of a MapReduce job by using queuing network model. This way the cost model adheres to the intra-job synchronization constraints that occur due the contention at shared resources.

The accuracy of our solution is validated via comparison of our model estimates against measurements in a real Hadoop 2.x setup. According to our evaluation results, the proposed model produces estimates of average job response time with error within the range of 11% - 13.5%.

## CCS Concepts

**•Information systems → MapReduce-based systems; •Theory of computation → Parallel computing models;**

## Keywords

MapReduce Performance Models; Hadoop 2.x; Queuing Theory; Mean Value Analysis

## 1. INTRODUCTION

MapReduce-based systems are increasingly being used for large-scale data analysis applications. Minimizing the execution time is vital for MapReduce as well as for all data processing applications, and accurate estimation of the execution time is essential for optimizing. Therefore, we need to build performance models that follow the programming model of such data processing applications. Furthermore, a clear understanding of system performance under different circumstances is key to critical decision making in workload management and resource capacity planning. Analytical performance models are particularly attractive tools as they might provide reasonably accurate job response time at significantly lower cost than simulation and experimental evaluation of real setups.

Programming in MapReduce requires adapting an algorithm to two-stage processing model, i.e., Map and Reduce. Programs written in this functional style are automatically parallelized and executed on computing clusters. Apache Hadoop is one of the most popular open-source implementations of MapReduce paradigm. In the first version of Hadoop, the programming paradigm of MapReduce and the resource management were tightly coupled. In order to improve the overall performance as well as the usefulness and compatibility with other distributed data processing applications, some requirements were added, such as high cluster utilization, high level of reliability and availability, support for programming model diversity, backward compatibility, and flexible resource model [12]. Thus, the architecture of the second version of Hadoop has undergone significant improvements, introducing YARN (Yet Another Resource Negotiator), a separate resource management module that noticeably changes the Hadoop architecture. It decouples the programming model from the resource management infrastructure and delegates many scheduling functions to per-application components. The cluster resources are now being considered as continuous, hence there is no static partitioning of resources per map and reduce tasks (i.e., a division between map and reduce slots). Clearly, it is impossible to apply the cost models relaying on such a static resource allocation as in the first version of Hadoop, and hence it is necessary to find other approaches.

In this paper, we address the challenges of defining accurate performance models for estimating the execution time of MapReduce workloads in Hadoop 2.x. We analyzed the approaches for Hadoop 1.x and also the architecture of Hadoop 2.x and we decided to base our model on performance model proposed for the first version of Hadoop in [12]. This model combines a precedence graph model, which allows to capture dependencies between different tasks within a one job, and

queueing network model to capture the intra-job synchronization constraints. Due to changes in the Hadoop architecture, we adapted that model for Hadoop 2.x.

**Contributions**. The main contributions of this paper can be summarized as follows:

- By analyzing the architecture of Hadoop 2.x, we identify cost factors that can potentially affect the cost of the MapReduce job execution.
- We theoretically define a MapReduce cost model for Hadoop 2.x that captures the precedence of different tasks of MapReduce jobs as well as the synchronization delays due to shared resources.
- We evaluate the accuracy of our cost model by implementing the cost estimation prototype and comparing the obtained estimates with real MapReduce executions.

# 2. RELATED WORK

We observe two groups of approaches for analyzing the performance of MapReduce job for the first version of Hadoop. All performance models described in Subsection 2.1 are static, they do not take into account the queuing delays due to contention at shared resources and the synchronization delays between different tasks. In Subsection 2.2, we introduce two most common approaches for constructing dynamic performance models for parallel applications and describe a performance model proposed for Hadoop 1.x that takes into consideration the queuing delays.

## 2.1 Static MapReduce Performance Models

There are significant efforts and important results towards modeling the task phases in order to estimate the execution of a MapReduce job in Hadoop 1.x. Herodotou proposed performance cost models for describing the execution of a MapReduce job in Hadoop 1.x [3]. In his paper, performance models describe dataflow and cost information at the finer granularity of phases within the map and reduce tasks. This model captures the following phases of Map task: read, map, collect, spill and merge. For the reduce task there are independent formulas for shuffle phase, merge phase and reduce and write phases. In terms of Herodotou's model, the overall job execution time is simply the sum of the costs from all map and reduce phases. As we can see in these cost formulas, there is a fix amount of slots per Map and Reduce tasks. Since in the first version of Hadoop, the number of resources for Map and Reduce jobs is determined in advance and does not change. YARN completely departs from the static partitioning of resources for maps and reduces, and there is no slot configuration. Thus, we cannot apply Herodotou's cost formulas directly and it is necessary to find other approaches.

There has also been an effort of defining the lower and upper bounds for job completion time and resource allocation to a job so that it finishes within the required deadline. In [11], the authors proposed a framework called ARIA (Automatic Resource Inference and Allocation for MapReduce Envinronments) that for a given job completion deadline could allocate the appropriate amount of resources required for meeting the deadline. This framework consists of three inter-related components. The first component is a Job Profile that contains the performance characteristics of application during map and reduce stages. The second component constructs a MapReduce performance model, that for a given job and its soft deadline estimates the amount of resources required for job completion within a deadline. Provided performance model captures the following stages of MapReduce job: map, shuffle/sort and reduce stages. The last component is the scheduler itself that determines the job ordering and the amount of resources required for job completion within the deadline.

For estimating the job completion time authors applied the Makespan Theorem for greedy task assignment, which allows to identify the upper $T_J^{Up}$ and lower bounds $T_J^{Low}$ for the task completion time as a function of the input dataset size and allocated resources. According to the research $T_J^{Avg} = \frac{T_J^{Up}+T_J^{Low}}{2}$ is the closest estimation of job completion time T. It was observed that the relative error between the predicted average time $T_J^{Avg}$ and the measured job completion time is less than 15%, and hence, the predictions based on $T_J^{Avg}$ are well suited for ensuring the job completion within the deadline. Nevertheless, this model has significant limitations that do not allow us to apply it to the second version of Hadoop. As in Herodotou's cost models, the proposed model uses a fixed amount of slots per map and reduce tasks within one node.

There has also been an attempt of evaluating the impact of task scheduling on system performance. However, current schedulers neither pack tasks nor consider all their relevant resource demands. This results in fragmentation and overallocation of resources and, as a consequence, it decreases noticeably the overall performance. Robert Grandl et al. present in [2] Tetris, a multi-resource cluster scheduler, that packs tasks to nodes based on their requirements of all resource types, which allows to avoid the main limitations of existing schedulers. The objective in packing is to maximize the task throughput and speed up job completion. Thus, Tetris combines both heuristics - best packing and shortest remaining job time - to reduce average job completion time. Authors proved that achieving desired amounts of fairness can coexist with improving cluster performance. This scheduler was implemented in YARN and showed gains of over 30% in makespan and job completion time. Based on new scheduler authors proposed a performance model that has a number of shortcomings. First of all, fast solvers are only known for a few special cases with non-linear constraints, meanwhile several of the constraints are non-linear: resource malleability, task placement and how task duration relates to the resources allocated at multiple machines. Finding the optimal allocation is computationally very expensive. Scheduling theory shows that even with elimination the placement considerations, the problem of packing multidimensional balls to minimal number of bins is APX-Hard [13]. Secondly, ignoring dependencies between tasks is unacceptable in case of MapReduce jobs, where the shuffle/sort phase starts as the first map task is completed.

## 2.2 Dynamic MR Performance Models

The main challenge in developing the cost models for MapReduce jobs is that they must capture, with reasonable accuracy, the various sources of delays that job experiences. In particular, tasks belonging to a job may experience two types of delays: queuing delays due to contention at shared resources, and synchronization delays due to precedence constraints among tasks that cooperate in the same job - map and reduce phases. There are two main techniques to esti-

mate the performance of workloads of parallel applications that do not take into account the synchronization delays. One such technique is Mean Value Analysis (MVA)[14,15]. MVA technique takes into consideration only task queueing delays due to sharing of common resources. Thus, 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 classical solution is to jointly exploit Markov Chains for representing the possible states of the system, and queuing network models, to compute the transition rates between states [16,17]. However, such approaches do not scale well since the state space grows exponentially with the number of tasks, making it impossible to be applied to model jobs with many tasks, which is commonly the case of MapReduce jobs.

Vianna et al. in their work [12] proposed a performance model for MapReduce workloads, which is based on reference model [4]. Given a tree specifying the precedence constraints (i.e., precedence tree) among tasks of a parallel job as input, the reference model applies an iterative approximate Mean Value Analysis (MVA) 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, such as parallel or sequential execution. However, this model cannot be directly applied to MapReduce workload due to the fact that in a MapReduce job the beginning of a shuffle phase in a reduce task depends on the end of the first map task. The model proposed in [12] enhances the reference model as follows:

- It explicitly addresses the synchronization delays due to precedence constraints among tasks from the same job;
- It takes into account queuing delays due to contention at shared resources;
- It proposes an alternative strategy to estimate the average response time of subsets of the tasks belonging to a MapReduce job, which leads to more accurate estimates of a job's average response time.

According to the model validation results, the proposed model produces estimates of average job response time that deviate from measurements of a real execution by less than 15%.

Although this model does not capture the dynamic resource allocation and it assumes a fixed amount of threads to process map and reduce tasks per node as one of the input parameters, it has important advantages in comparison with previous models. First of all, unlike Herodotous's model that does not capture resource contention between tasks, this model is taking into account the queuing delays due to the contention at shared resources. Secondly, it is able to capture the synchronization delays introduced by the communication between map and reduce tasks (ARIA and Tetris are not considering this property of MapReduce job execution).

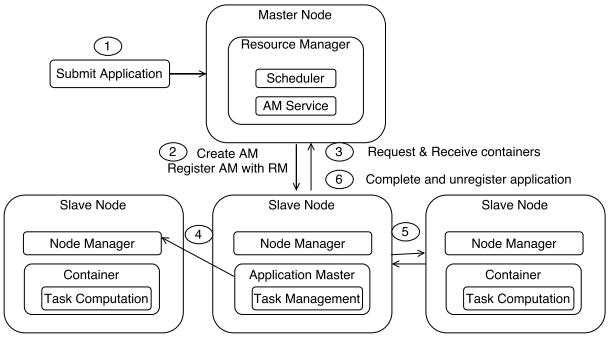



**Figure 1: Job execution process in YARN [6]**

# 3. ARCHITECTURE ANALYSIS

In this section, we analyze the architecture and components of Hadoop 2.x in order to identify the architectural changes that affect the cost of executing MapReduce jobs.

## 3.1 Running Example

To illustrate our approach and facilitate the explanations throughout the paper, we introduce a running example. Assume that we have $n = 3; m = 4; r = 1$, where $n$ - total number of nodes, $m$ - number of containers required for map tasks, $r$ - number of containers required for reduce tasks. All nodes have the same capacity. Using this data, we will illustrate the main steps of our approach.

## 3.2 Main components of YARN module

In the second version of Hadoop, the YARN module appeared and changed the architecture significantly. It is responsible for managing cluster resources and job scheduling. In the previous versions of Hadoop, this functionality was integrated with the MapReduce module where it was realized by the JobTracker component. The fundamental idea of YARN is to split the two major functionalities of the JobTracker, resource management and task scheduling/monitoring in order to have a global ResourceManager, and application-specific ApplicationMaster. By separating resource management functions from the programming model, YARN delegates many scheduling-related tasks to per-job components and completely departs from the static partitioning of resources for maps and reduces, considering the cluster resources as a continuum, which brings significant improvements to cluster utilization. The YARN module consists of three main components:

- Global Resource Manager (RM) per cluster
- Node Manager (NM) per node
- Application Master (AM) per job

RM runs as a daemon on a dedicated machine and arbitrates all the available resources among various competing applications. We will not go in detail of all components of RM [1] and will focus on the most important ones:

- *Scheduler*, which is responsible for allocating resources to the various applications that are running.
- *Application Manager Service* that negotiates the first container (logical bundle of resources bound to a particular node) for the Application Master. AMs are

| Number of containers | Priority | Size | Locality constraints | Task type |
|---|---|---|---|---|
| 2 | 20 | x | n1 | map |
| 2 | 20 | x | n2 | map |
| 1 | 10 | x | * | reduce |

**Table 1: ResourceRequest Object**

responsible for negotiating resources with the RM and for working with the NMs to start, monitor, and stop the containers.

Based on the core functionalities of YARN components, the general schema of job execution process is presented in Figure 1. The process starts when an application submits a request to the ResourceManager. The AM registers with the RM through AM Service and is started in the container that AM Service dedicated for it. Then, the AM requests containers from the RM to perform actual work. Once the AM obtains containers, it can proceed to launch of them by communicating to a NM. Computation takes place in the containers, which keep in contact with the AM. When the application is complete, AM should unregister from the RM.

## 3.3 Resource management in Hadoop 2.x

For performance model construction it is necessary to understand in detail the resource request process. AM needs to figure out its own resource requirements, which can be:

- *Static.* If the resource requirements are decided at the time of application submission, and when the AM starts running, there is no change to the resource requirement that specification. In case of Hadoop MapReduce, the number of map tasks is based on the input splits (i.e., HDFS chunks), and the number of reducers on user-defined parameter. Thus, the total number of mappers and reducers is fixed before the application submission.
- *Dynamic.* When dynamic resource requirements are applied, the AM may choose how many resources to request at run time based on criteria such as user hints, availability of cluster resources, and business logic.

Once a set of resource requirements is clearly defined, the AM can begin sending the requests in a heartbeat message to the RM. Based on the task requirements, AM calculates how many containers it needs and requests them from the RM via a list of ResourceRequest objects. ResourceRequest object for running example from Subsection 3.1 is presented in the Table 1. In the ResourceRequest object, containers can have different priorities. Higher-priority requests of an application are served first by the RM. There is no cross-application implication of priorities. According to the source code of MapReduce AM (package org.apache.hadoop.mapreduce.v2.app.rm; RMContainerAllocator class), MapReduce AM assigns a higher priority to containers needed for the Map tasks and a lower priority for the Reduce tasks' containers, with default priorities values equal to 20 and 10 respectively.

One thing to note is that containers may not be immediately allocated to the AM. This does not imply that the AM should keep on asking the pending count of required containers. Once an allocated request has been sent, the AM will eventually allocate the containers based on cluster capacity, priorities and the scheduling policy. The AM should request for containers again if and only if its original estimate changed and it needs additional containers.


**Figure 2: Lifecycle of map task**

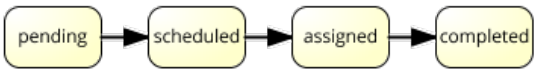



**Figure 3: Lifecycle of reduce task**

## 3.4 Job scheduling in Hadoop 2.x

There is another differentiating characteristic in terms of how the scheduling of those resources happens:

- Resource usage follows a static all-or-nothing model, when all containers are required to run together. For example, if AM asks for $n$ containers, the job will start when AM receives exactly $n$ containers.
- Alternatively, resource usage may change elastically, depending on the availability of resources. In this case, the job starts even if AM receives less than required number of containers.

For cost model construction, it is necessary to understand the way to distribute containers for tasks within different nodes. By analyzing the source code of MapReduce (package org.apache.hadoop.mapreduce.v2.app.rm; RMContainerAllocator.java class), we observed that map and reduce tasks have different lifecycles that are presented in Figure 2 and Figure 3.

*Vocabulary Used:*
pending → requests which are not yet sent to RM
scheduled →requests which are sent to RM but not yet assigned
assigned → requests which are assigned to a container
completed → requests for which the container has completed the execution

Furthermore, AM can do a second level of scheduling and assign its containers to whichever task that is part of its execution plan. Thus, resource allocation in YARN is late binding. The AM is obligated only to use resources as provided by the container, it does not have to apply them to the logical task for which it originally requested the resources. Thus, the MapReduce AM takes advantage of the dynamic two-level scheduling. When the AM receives a container, it matches that container against the set of pending tasks, selecting a task with input data closest to the container, first trying data local tasks, and then falling back to rack locality.

# 4. PROPOSED SOLUTION

As a basis of our MapReduce performance model for Hadoop 2.x, we decided to take the performance model for MapReduce workloads proposed for Hadoop 1.x [12]. The main challenges of adapting the existing performance model to the architectural changes of Hadoop 2.x were: (1) the construction of the precedence tree, taking into consideration the dynamic resource allocation as opposed to the predefined slot configuration per map and reduce tasks in the

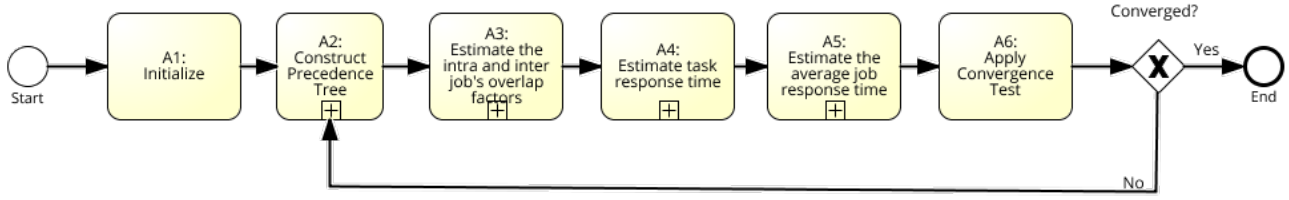



**Figure 4: The main steps of Modified MVA algorithm [4]**

| Notation | Input Parameter |
|---|---|
| **Configuration parameters** | |
| $numNodes$ | Number of Nodes |
| $cpuPerNode$ | Number of CPU per node |
| $diskPerNode$ | Number of disks per node |
| **Workload Parameters** | |
| $S_{i,k}$ | Residence time for task of class $i$ in the service center $k$ |
| $AvgResponseTime_i$ | Response time for task of class $i$ |
| $m$ | Number of map tasks |
| $r$ | Number of reduce tasks |
| $MaxMapPerNode$ | Maximum number of containers per node for map tasks |
| $MaxReducePerNode$ | Maximum number of containers per node for reduce tasks |

**Table 2: Input parameters**

Hadoop 1.x, and (2) how to capture the synchronization delays introduced by the pipeline that occur among maps and shuffle phase of the reduce tasks.

## 4.1 Input Parameters

For the sake of simplicity, we consider a distributed cluster with a set of computing nodes equal to $numNodes$, all of them having the same technical characteristics. The workload is composed by $N$ MapReduce jobs executing concurrently in the system. Each job has $m_i$ map tasks and $r_i$ reduce tasks. We are not dividing the map task into phases. As a partial sort is performed after each shuffle, we group each pair of shuffle and sort in a single subtask called shuffle-sort. After all partial sorts are finished, a final sort is executed, followed by the final phase of reduce tasks that applies the reduce function. We group the final sort and the reduce function into one merge subtask. Thus, according to our terminology, the reduce task is divided into two subtasks: shuffle-sort and merge. The input parameters for our model are presented in Table 2. We consider 2 types of service centers (resources): CPU&Memory and Network. The overall number of task classes $C$ is 3 (i.e., map, shuffle-sort, and merge). We would like to emphasize the difference between the residence and response time for a task. The average response time is the total time that task spends in the cluster. Meanwhile, the residence time of task class $i$ on service center $k$ is the average amount of time that task spends using the corresponding resource $k$ during its execution.

## 4.2 Modified Mean Value Analysis (MVA) Algorithm

To solve the queueing network model, we use the modified Mean Value Analysis. An algorithm to solve the MVA for a closed network system initially was proposed by Reiser and Lavenberg [7] on top of which, we build our performance model. Bellow we describe the main steps of the algorithm and the assumptions we consider in our approach.

Suppose a system with $C$ task classes and $K$ service centers. Let $\vec{N}$ be a vector, $i$-th component of which indicates the number of tasks of class $i$ in the system; $S_{jk}$ is the average demand of class $j \in C$ task on service center $k \in K$ (i.e., the average amount of time).

The main steps of the algorithm are presented in Figure 4, which consists of 6 main activities: A1-A6. We start by initializing the average residence time of each type of task at each service center and the average response time of each task in the system. Then based on the average response time of each individual task, the precedence tree is constructed. The next step is to take into account the effects of the queuing delays 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 overlap). These overlap factors produce the new estimates of task average response time. The final step is to apply the convergence test on the new estimates of average response time. In case that the convergence test fails, we return to the construction of precedence tree step trying to build a new, and more accurate precedence tree based on estimates of task response time obtained during the previous iteration. In case that current estimates are close enough to the previous ones, the algorithm finishes, and as a result, a final job average response time is produced.

In the following subsections, we explain the activities of the modified MVA algorithm. In particular, we extensively explain our modification of precedence tree construction procedure in Subsection 4.2.2.

### *4.2.1 Initialization of task response time*

Initialization process consists of two sub processes that can run in parallel: initializing the average residence time of each type of task at each service center and the average response time of each task in the system. For initializing the residence time, we take the average of residence time from the history of corresponding real Hadoop job executions. To initialize the tasks response time, we can apply the following approaches:

- Using sample techniques - taking the average of task response time from job profile.
- Obtaining from the existing static cost models, for ex-

ample, from Herodotou's cost models [3] (we can assume that first all map tasks will be executed then reduce tasks). Thus, we will give all available resources to the map tasks and then to the reduce tasks.

The second approach leads to faster algorithm convergence due to more accurate response time initialization and, as consequence, less number of iterations of the algorithm. In out model we use the second approach.

### 4.2.2 Building precedence tree

In a precedence tree, each leaf represents a task and each internal node is an operator describing the constraints in the execution of the tasks. We will consider a precedence binary tree built from 2 types of primitive operators: serial ($S$) and parallel-and ($P$). The $S$ operator is used to connect tasks that run sequentially, whereas the $P$ operator connects tasks that run in parallel. An example of the precedence tree is presented on Figure 5.

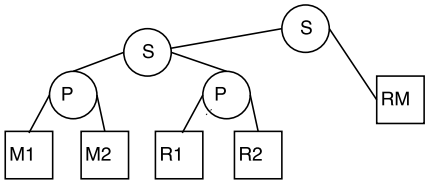



**Figure 5: Precedence Tree**

The main goal of building the precedence tree is to capture the execution flow of the job, identifying the parallel or serial order of execution of individual tasks and their interdependencies. Based on new estimates for task response time, we rebuild the precedence tree at each iteration of the algorithm (the complexity analysis of building precedence tree procedure can be found in Subsection 4.3).

The precedence tree depends on the response time of individual tasks and is built using a task response timeline. Based on the obtained timeline, the precedence tree can be constructed uniquely up to graph isomorphism. To be able to distinguish the parallel and sequential task executions, we have to identify the beginning of a new phase in a timeline. By the phase we mean the maximum period of time, during which all tasks are executed simultaneously. Thus, tasks within the same phase are executed in parallel, while tasks from different phases are executed sequentially.

Based on the architectural analysis (see Section 3), the core assumptions and factors that influence the timeline construction process can be divided into two subgroups: (1) related to the job scheduling, and (2) related to the resource management system.

The first subgroup, related to the job scheduling, consists of the following factors:

1. We assume that RM uses the Capacity scheduler which is the default scheduler of the Hadoop YARN distribution. The fundamental unit of the Capacity scheduler is a queue. We assume that we do not have any hierarchical queues and we have only one root queue. Thus, resource allocation among applications will be in the FIFO order, i.e., the priority will be given to the first application requesting the resources.
2. Due to architectural changes, some responsibilities of job scheduling are delegated to the AM. We have to determine the way to distribute containers for tasks within different nodes. According to findings in Subsection 3.4, map and reduce tasks have different lifecycles that we need to take into account during the timeline construction procedure.
3. We are assuming that AM will use requested containers for the same type of tasks as originally requested, thus we ignore the late binding functionality of AM.

**Algorithm 1** Timeline Construction

**Input:** M,R,N
**Output:** TL
{*st*->startTime; *et*->endTime; *d*->duration; *sd*->shuffleDuration; *an*->assingedNode; }
1: **for** $i := 1$ to $|N|$ **do**
2: $TL[i] := \emptyset$;
3: **end for**
4: **for** $m \in M$ **do**
5: $i := min(TL)$;
$m.an := i$;
$m.st := min(TL[i])$;
$m.et := m.st + m.d$;
$TL[i] := TL[i] \cup \{m\}$;
6: **end for**
7: **if** ($slow\_start$) **then**
8: $border := TL[min(TL)].et$;
9: **else**
10: $border := TL[max(TL)].et$;
11: **end if**
12: **for** $r \in R$ **do**
13: $i := min(TL)$;
$r.an = i$;
$r.st := max(TL[i].et, border)$;
14: **for** $m \in M$ **do**
15: **if** $(m.an <> i)$ **then**
16: $r.d := r.d + \frac{m.sd}{|R|}$;
17: **end if**
18: **end for**
19: $r.et := r.st + r.d$;
20: $TL[i] := TL[i] \cup \{r\}$;
21: **end for**
22: Return $TL$;

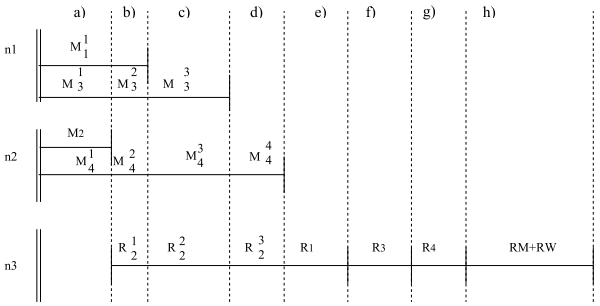



**Figure 6: Timeline example**

The second subgroup, related to resource management, is composed of the following factors and assumptions:

1. Considering the finding in Subsection 3.2 related to

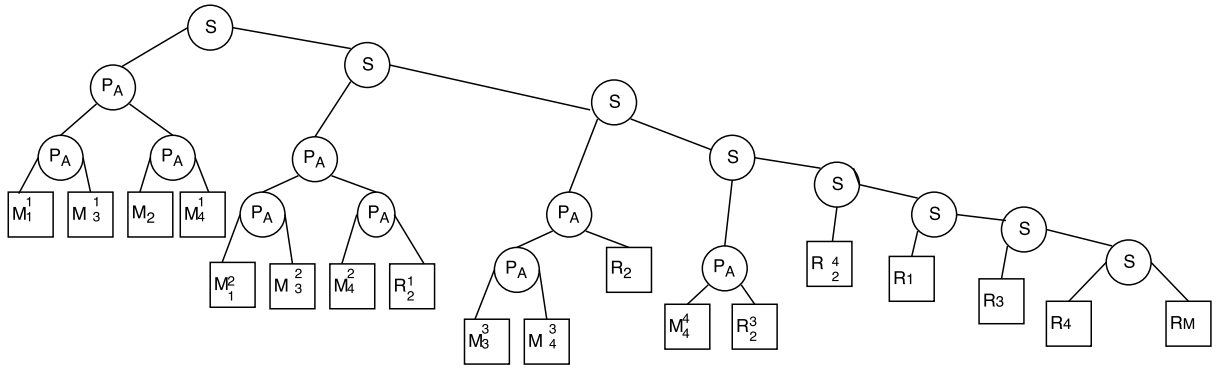



**Figure 7: Precedence Tree Example**

different priorities for map and reduce tasks, we provide a container first to map task and after to reduce task.

2. Assigning containers for map tasks mainly depends on whether we consider or not locality constraints (configuration parameter).

   In our model, we consider a node locality constraints for map task and ignore locality constraints for reduce tasks. In case of ignoring the locality constraints, we distribute containers for tasks uniformly among nodes with the highest remaining capacity. Assuming that all nodes have the same capacity, we will take into consideration the occupancy rate and assign containers to the nodes with the lowest value.

   Container allocation process for reduce tasks conform to the following steps:

   - Check for slow start: by default, schedulers wait until 5% of the map tasks in a job have completed before scheduling reduce tasks for the same job.
   - Check if all maps are assigned: if not, schedule reduce tasks based on the percentage of completed map tasks (conf. parameter). Otherwise, schedule all reduce tasks (map output locality is not taken into consideration, request asks for a container on any host/rack).

The last rule that we have to consider is how to divide the timeline into phases. All tasks within the same phase are executed in parallel, and tasks that belong to different phases are executed sequentially. It means that each start or end of a task indicates the start of a new phase.

As a summary, we present below our algorithm for the timeline construction, considering that map tasks have higher priority than reduce tasks. We start in lines 1-6 distributing containers for map tasks.In case the slow start is set, the beginning of the shuffle-phase of reduce task will coincide with the end of first map task on the node that has the lowest occupancy rate. Thus, shuffling starts as earlier as possible. In the opposite case, when we do not have a slow start, the shuffle-phase of reduce task starts as late as possible (lines 7-11). Further, in lines 12-21 we distribute containers for reduce tasks.

Then based on the timeline we build a binary precedence tree. In order to reduce the maximal depth of precedence tree, we apply a balancing procedure for each P-subtree of it.

**Example**. Applying the above timeline construction algorithm to example from Section 3, we obtain the timeline, that is presented in Figure 6. Based on this timeline we are able to construct the precedence tree (Figure 7)

#### 4.2.3 *Estimation of the Intra- and Inter- job overlap factors*

For a system with multiple classes of tasks the queueing delay of task class $i$ due to task class $j$ is directly proportional to their overlaps [5]. We are going to consider two types of overlap factors: the intra-job overlap factor $\alpha_{ij} \forall i, j$ - taskID's from the same job, and inter-job overlap factor $\beta_{kr} \forall k, r$ - taskID's from different jobs. In Figure 8, we provide an example for intra- and inter-job overlap factors.

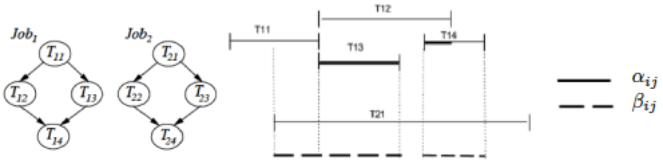



**Figure 8: Intra- and inter-job overlap factors**

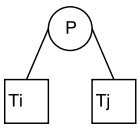
The algorithm for estimating the overlap factors can be found in [4].

#### 4.2.4 *Average Job Response Time Estimation*

There are 2 alternative approaches to estimate the job response time:

1. Tripathi-based [4]: To estimate the response time of P-rooted ans S-rooted sub-trees, we approximate the distribution of response time of each of its children by either an Erlang or a Hyperexponential distribution depending on the coefficient of variation ($CV = \frac{\mu}{\sigma}$) of the response times associated with each child node [4], [9]. We assume that the distribution of $X$ of Erlang type if its $CV <= 1$, and Hyperexponential distribution if $CV >= 1$. Knowing the distribution of leafs, we can determine the distribution type (Erlang or Hyperexponential) and mean value of response time for $P$ and $S$ [4].
2. Fork/join-based [12]: 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. One such estimate is the product of the $k-th$ harmonic function by the maximum average response time of $k$ tasks [10].




$R_{ik} = H_k \cdot max(T_i, T_j)$,
where $H_k = \sum_{i=1}^{s} \frac{1}{i}$,
$s$ - is the number of child nodes

The precedence tree is a binary tree. Thus, $H_k = \frac{3}{2}, \forall k$ The intuition behind this formula is the response time for a parent node equal to the biggest child response time plus possible delay (multiplication by $\frac{3}{2}$).

#### 4.2.5 Estimation of task response time

To solve the queuing network models we apply Mean Value Analysis (MVA) [7]. MVA is based on the relation between the mean waiting time and the mean queue size of a system with one job less. The algorithm for estimating the task response time consists of 5 main steps that are presented in Figure 9, whose detailed explanation can be found in [4].

#### 4.2.6 Applying convergence test

During the convergence test, we are comparing the Total Response Time from the previous iteration with the Total Response Time received in the current iteration. In case they are close enough (i.e., $|R^{curr} - R^{prev}| \leq \epsilon$), the algorithm finishes. Otherwise, we return to the precedence tree construction process and repeat activities A2-A6. We use $\epsilon = 10^{-7}$, which is the recommended value for MVA [4]. Theoretically, this value provides a good trade-off between the level of accuracy and the complexity of the algorithm (number of iterations). Moreover, we performed empirical tests and confirmed that $\epsilon = 10^{-7}$ gives a good trade-off (with lower values of $\epsilon$ the job response time almost does not change, meanwhile the number of iterations continues to grow).

### 4.3 Complexity Analysis

We can find the complexity of the proposed performance model by analyzing the complexity of the MVA algorithm and the complexity of the precedence tree construction.

According to [4], the MVA algorithm is computationally efficient, having the complexity – $O(C^2N^2K)$, where $C$ is the number of task classes in the job, $N$ is the number of jobs, $K$ is the number of service centers.

The time complexity to build the precedence tree is equal to the complexity of timeline construction. The cost to construct this timeline can be identified by the time required to repeatedly search for the next task to finish until the termination of all the tasks.

Let $C$ be the total number of tasks in the timeline and $T$ be the total number of containers in execution.

$C = allMapTasks + allShuffleSortTasks + allMergeTasks$;

$T = n \times max(pMaxMapsPerNode, pMaxReducePerNode)$,
where $n$ - the number of nodes; $pMaxMapsPerNode$,
and $pMaxReducePerNode$ - the maximum number of containers for map and reduce tasks correspondingly,

$$pMaxMapsPerNode = \lfloor \frac{TotalNodeCapacity}{SizeOfContainerForMapTask} \rfloor$$

$$pMaxReducePerNode = \lfloor \frac{TotalNodeCapacity}{SizeOfContainerForReduceTask} \rfloor$$

Thus, in the worst case, the time complexity to build a precedence tree at each iteration is given by the search for $m + r(m + 1)$ tasks in $T$ containers, that is $O(C \times T) = O((m + r(m + 1)) \times (n \times max(pMaxMapsPerNode, pMaxReducePerNode)))$, where $m, r$ -is the number of map and reduce tasks in the job correspondingly. The computational cost of the whole solution: $O(C^2N^2K) + O(((m+r(m+1)) \times (n \times max(pMaxMapsPerNode, pMaxReducePerNode))) \times numberOfIterations)$. As we can notice, the computational cost of the whole solution is dominated by the MVA algorithm that has quadratic complexity equal to $O(C^2N^2K)$.

## 5. EVALUATION

This section presents the results of a set of experiments we performed with the proposed performance model. We provide the validation results from a comparison of our model (two approaches: Tripathi-based and fork/join-based) against measurements of a Hadoop 2.x setup. For evaluation we decided to use map-and-reduce-input heavy jobs (i.e., wordcount[1]) that process large amounts of input data and also generate large intermediate data [8].

### 5.1 Experiments Setup

We performed a set of experiments analyzing the job response time in terms of the following parameters:

- the number of nodes: 4,6,8;
- the size of input data: 1GB, 5GB;
- the number of jobs that are executed simultaneously in the cluster: 1,2,3,4.

Each node in the cluster has the same technical characteristics:

- 2x Intel Xeon E5-2630L v2 a 2.40 GHz
- 128 GB Memory RAM
- 1 hard disk TB SATA-3
- 4 Network Intel Gigabit Ethernet

For each experiment we analyze the job response time fixing two out of three parameters. Each experiment we repeated 5 times and then took the median of response time.

### 5.2 Results

First, we present the response time for different number of jobs (1 and 4) that are executed simultaneously in the cluster on different number of nodes (4,6,8) with a fixed size of input data. In all graphs we use blue continuous line to show results for the real Hadoop setup, red dashed - for Fork/join based approach, green 2 dots 1 dash line - for Tripathi based approach. Results for the input size equal to 1 GB and 5GB are presented in Figures 10 and 11, and Figures 12 and 13, respectively. Figure 14 shows the response time depending on the number of jobs (from 1 to 4) that are executed simultaneously in the cluster with a fixed size (i.e., 4 nodes).

We can notice that the Fork/join based approach provides more accurate estimation of job response time with error between 11% and 13,5%, meanwhile the Tripathi-based approach shows an error between 19% and 23%. For 5GB input size, we obtain the bigger value of an error: 13.5%

[1]WordCount Example from the Hadoop distribution: https://wiki.apache.org/hadoop/WordCount

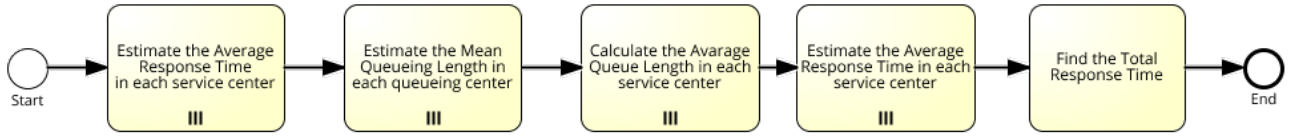



**Figure 9: The main steps for task response time estimation [4]**

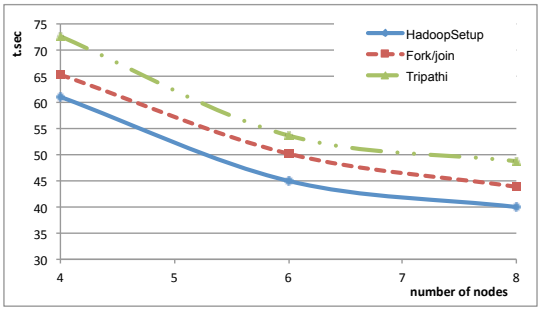



**Figure 10: Input: 1GB; #jobs: 1**

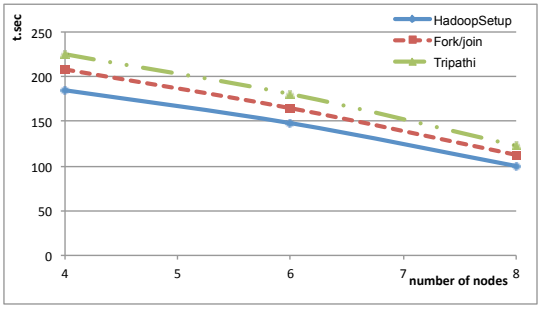



**Figure 11: Input: 1GB; #jobs: 4**

and 23%, respectively. We observe that the accuracy of our algorithm depends on the number of map tasks and not necessarily on the size of input data. The bigger value of error is connected with the complexity (the maximal depth) of the precedence tree, which is increasing with the higher number of map tasks. In order to prove this hypothesis, we increase the number of map tasks without increasing the input data size. Thus, we reduced the default block size for the map task from $128MB$ to $64MB$ and repeated the experiments. The results for the input data size equal to 5GB and number of jobs equal to 1 are presented in the Figure 15. As showed by these results, experiments confirm our supposition, as we obtained the biggest values of errors: 17% and 25% for fork/join and Tripathi-based approaches, respectively. For reducing the maximal depth of the precedence tree and, as consequence, for decreasing the error, we balance it.

The Fork/join approach in our model produces accuracy improvements over the original model for Hadoop 1 [12], on which we based our solution. For one job in the cluster we received the error within 13.5% against 15% in [12].

In conclusion, we can notice that the Fork/join based approach provides more accurate results than Tripathi-based, but with both approaches we overestimate the execution time. The cost model can be further fine tuned for improving the accuracy of the estimations by changing the calculation the overlap factors.

# 6. CONCLUSIONS AND FUTURE WORK

In this work, we tackled the challenge of creating a MapReduce performance model for Hadoop 2.x, which takes into consideration queuing delays due to contention at shared resources, and synchronization delays due to precedence constraints among tasks that cooperate in the same job (map and reduce phases). The modeling approach extends the solution proposed for Hadoop 1.x in [12], where the execution flow of a job was presented by a precedence tree and the contention at the physical resources were captured by a closed queuing network. Our main contributions are the deep analysis of the Hadoop 2.x internals, identifying the main architectural changes in Hadoop, and the creation of the MapReduce performance model for Hadoop 2.x. In particular, considering the identified changes in the architecture of Hadoop 2.x and taking into account the dynamic resource allocation, we created the method for timeline construction, based on which the precedence tree is built.

We validated our model against the measurements obtained from a real Hadoop setup for different number of jobs that were executed simultaneously. Our experiments showed the effectiveness of our approach: the average error of job response time estimation for standard block size is in the range of 11% and 13.5%. Our model can be used for theoretically estimating of the jobs response time at a significantly lower cost than simulation and experimental evaluation of real setups. It can also be useful for critical decision making in workload management and resource capacity planning.

Our future plans focus on the tuning of provided performance model in order to decrease the error of job response time estimation. Furthermore, we are planning to extend our model to be able to estimate the amount of consumed resources for each task and the whole job.

# 7. ACKNOWLEDGMENTS

This research has been funded by the European Commission through the Erasmus Mundus Joint Doctorate "Information Technologies for Business Intelligence - Doctoral College" (IT4BI-DC).

The research has been partially supported by the Spanish *Ministerio de Economía, Industria y competitividad*, grant TIN2016-79269-R.

# 8. REFERENCES

[1] http://hortonworks.com/blog/apache-hadoop-yarn-resourcemanager/. Accessed: 2016-11-16.

[2] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. In *ACM SIGCOMM Computer*

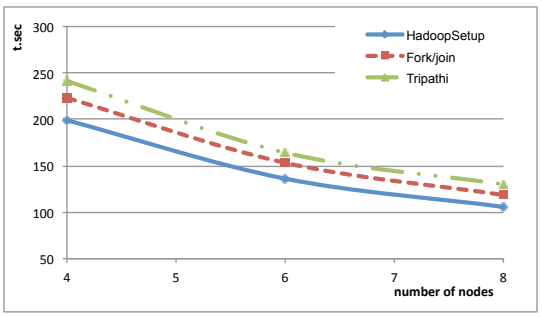



**Figure 12: Input: 5GB; #jobs: 1**

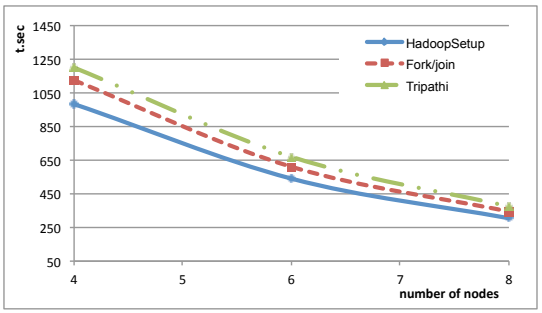



**Figure 13: Input: 5GB; #jobs: 4**

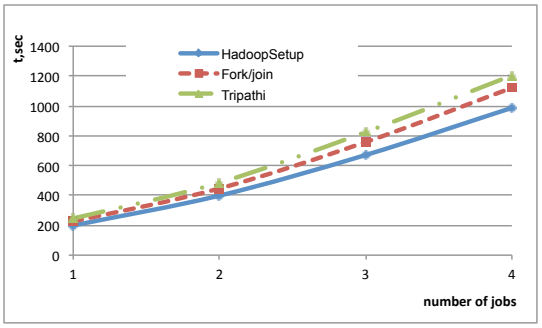



**Figure 14: #Nodes: 4; Input: 5GB**

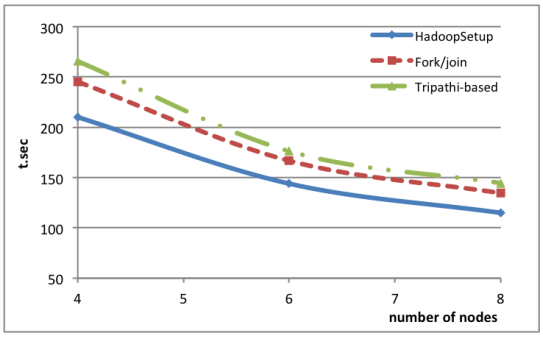



**Figure 15: Block: 64MB; Input:5GB; #jobs: 1**

*Communication Review*, volume 44, pages 455–466. ACM, 2014.

[3] H. Herodotou. Hadoop performance models. *arXiv preprint arXiv:1106.0940*, 2011.

[4] D.-R. Liang and S. K. Tripathi. On performance prediction of parallel computations with precedent constraints. *IEEE Transactions on Parallel and Distributed Systems*, 11(5):491–508, 2000.

[5] V. W. Mak and S. F. Lundstrom. Predicting performance of parallel computations. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):257–270, 1990.

[6] A. C. Murthy, V. K. Vavilapalli, D. Eadline, J. Niemiec, and J. Markham. *Apache Hadoop YARN: Moving Beyond MapReduce and Batch Processing with Apache Hadoop 2*. Pearson Education, 2013.

[7] M. Reiser and S. S. Lavenberg. Mean-value analysis of closed multichain queuing networks. *Journal of the ACM (JACM)*, 27(2):313–322, 1980.

[8] J. Shi, Y. Qiu, U. F. Minhas, L. Jiao, C. Wang, B. Reinwald, and F. Özcan. Clash of the titans: Mapreduce vs. spark for large scale data analytics. *Proceedings of the VLDB Endowment*, 8(13):2110–2121, 2015.

[9] K. S. Trivedi. *Probability & statistics with reliability, queuing and computer science applications*. John Wiley & Sons, 2008.

[10] E. Varki. Mean value technique for closed fork-join networks. In *ACM SIGMETRICS Performance Evaluation Review*, volume 27, pages 103–112. ACM, 1999.

[11] A. Verma, L. Cherkasova, and R. H. Campbell. ARIA: automatic resource inference and allocation for mapreduce environments. In *Proceedings of the 8th ACM international conference on Autonomic computing*, pages 235–244. ACM, 2011.

[12] E. Vianna, G. Comarela, T. Pontes, J. Almeida, V. Almeida, K. Wilkinson, H. Kuno, and U. Dayal. Analytical performance models for MapReduce workloads. *International Journal of Parallel Programming*, 41(4):495–525, 2013.

[13] G. J. Woeginger. There is no asymptotic PTAS for two-dimensional vector packing. *Information Processing Letters*, 64(6):293–297, 1997.