MapReduce Performance Model for Hadoop 2.x

Daria Glushkova, Petar Jovanovic, Alberto Abelló

Universitat Politècnica de Catalunya, BarcelonaTech, Barcelona, Spain

dglushkova@essi.upc.edu, petar@essi.upc.edu, 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 may provide reasonably accurate job response time estimation at significantly lower cost than experimental evaluation of real setups.

In this paper, we tackle the challenge of defining MapReduce performance model for Hadoop 2.x. While there are several efficient approaches for modeling the performance of MapReduce workloads in Hadoop 1.x, they could not be applied to Hadoop 2.x due to fundamental architectural changes and dynamic resource allocation in Hadoop 2.x. Thus, the proposed solution is based on an existing performance model for Hadoop 1.x, but taking into consideration architectural changes and capturing the execution flow of a MapReduce job by using queuing network model. This way, the cost model reflects 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.

Keywords: Hadoop 2.x; MapReduce Performance Model

1. Introduction

MapReduce-based systems are increasingly being used for large-scale data analysis applications. Minimizing the execution time is vital for MapReduce as for any data processing application, 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 the 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 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[1], 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, and flexible resource model [1]. 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, which 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 (i.e., a division between map and reduce slots). Therefore, map and reduce tasks compete now for the same resources. Clearly, it is impossible to apply the cost models relaying on such a static resource allocation as in the first version of Hadoop, and it is necessary to find other approaches.

In this paper, we address the challenges of defining an accurate performance model for estimating the execution time of MapReduce workloads in Hadoop 2.x. We analyze the approaches for Hadoop 1.x and the architecture of Hadoop 2.x to propose the performance model. Our solution is based on the model proposed for the first version of Hadoop in [1]. This model combines a precedence graph model, which allows to capture dependencies between different tasks within a single 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 potentially affect the cost of the MapReduce job execution.
- We theoretically define and implement a MapReduce performance 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 performance model by implementing a cost estimation prototype and comparing the obtained estimates with real MapReduce executions.

[1]https://hadoop.apache.org/docs/r1.2.1/; Accessed 15/01/2017

2. Related work

We observe two groups of approaches for analyzing the performance of MapReduce job for the first version of Hadoop. Performance models described in Subsection 2.1 are static, as 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 important 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 [2]. In his paper, performance models describe the dataflow and cost information at the finer granularity of phases within the map and reduce tasks. It captures the following phases of a Map task: read, map, collect, spill, and merge. For a Reduce task, there are independent formulas for shuffle phase, merge phase and reduce and write phases. In terms of the 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 static slot configuration. Thus, we cannot apply directly Herodotou's cost formulas, and it is necessary to find other approaches.

There has also been an effort for defining the lower and upper bounds of the job completion time and provide the resource allocation to a job so that it finishes within the required deadline. In [3], 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, shuffle/sort 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 its completion within the deadline. The last component is the scheduler itself that determines the job ordering and the amount of resources required for job completion. 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 over-allocation of resources, and as a consequence, it decreases noticeably the overall performance. Robert Grandl et al. present Tetris [4], 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 the new scheduler, the authors proposed a performance model that has several 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 eliminating the placement considerations, the multidimensional bin packing problem is APX-Hard [5]. Secondly, ignoring dependencies between tasks is unacceptable in case of MapReduce jobs, where the shuffle/sort phase can start as the first map task is completed.

2.2. Dynamic MapReduce Performance Models

The main challenge in developing cost models for MapReduce jobs is that they must capture, with reasonable accuracy, the various sources of delays that a job may experience. In particular, tasks belonging to a job may experience two types of delays: *queuing delays* due to the contention at shared resources, and *synchronization delays* due to the precedence constraints among tasks that cooperate in the same job - map and reduce phases. There are two main techniques to estimate the performance of workloads of parallel applications that natively do not take into account the synchronization delays. One such technique is Mean Value Analysis (MVA) [6],[7], which takes into consideration only task queueing delays due to sharing of common resources. Thus, it 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 [8], [9]. 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 [1] proposed a performance model for MapReduce workloads, which is based on a reference model [10]. 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 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 [1] enhances the reference model as follows:

1. It explicitly addresses the synchronization delays due to precedence constraints among tasks from the same job;
2. It takes into account queuing delays due to contention at shared resources;
3. 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. Authors use the Herodotou's static cost model for initialization the task durations.

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).

3. Architecture Analysis

In this section, we analyze the architecture and components of Hadoop 2.x in order to identify the factors 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. Let us simply assume that we have $n = 3; m = 4; r = 1$, where n - total number of nodes, m - number of containers required for map tasks, and r - number of containers required for reduce tasks, and all nodes have the same capacity. Using such a scenario, we will illustrate the main steps of our approach.

3.2. Main components of the 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 in the MapReduce module, where it was realized by the JobTracker[2] component. The JobTracker was responsible for scheduling, resource management, monitoring and re-execution of failed tasks, reporting job status to users, recording audit logs, aggregation of statistics, user authentication, and many others functions. The great amount of responsibilities caused limitation of scalability. 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.

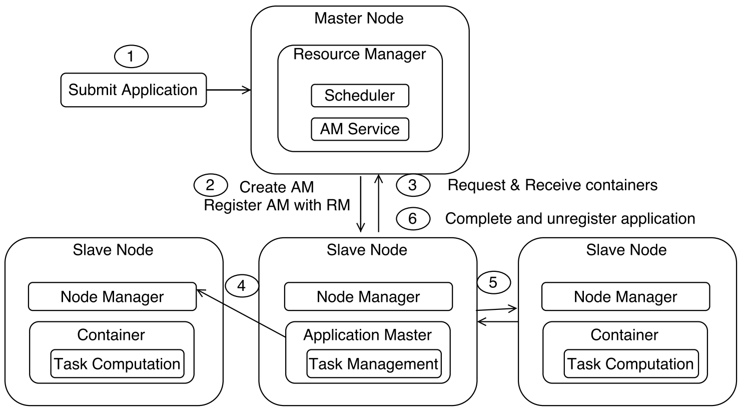


Figure 1: Job execution process in YARN [11]

YARN consists of three main components:

- Global Resource Manager (RM) per cluster, executing on the master node.
- Application Master (AM) per job
- Node Manager (NM) per slave node

[2]https://hadoop.apache.org/docs/r1.2.1/api/org/apache/hadoop/mapred/JobTracker.html

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 [3] 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 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. It starts when a client submits a request to the RM for executing an application (1). The AM registers with the RM through AM Service and is started in the container that AM Service dedicated for it (2). Then, the AM requests containers from the RM to perform the actual work (3). Once the AM obtains containers, it can proceed to launch them by communicating to a NM (4). Computation takes place in the containers, which keep in contact with the AM (5). When the application is complete,the AM should unregister from the RM (6).

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:

(a) *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 a user-defined parameter. Thus, the total number of mappers and reducers is fixed before the application submission.

(b) *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, the AM calculates how many containers it needs and requests them from the RM via a list of ResourceRequest objects. The ResourceRequest object for the running example from Subsection 3.1 is presented in Table 1. In this ResourceRequest object, containers can have different priorities, in which they will be served by the RM. There is no cross-application implication of

[3]`http://hortonworks.com/blog/apache-hadoop-yarn-resourcemanager/`

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

priorities. According to the source code of MapReduce AM [4], it assigns a higher priority to containers needed for the Map tasks and a lower priority for the Reduce tasks' containers, with default priority values equal to 20 and 10 respectively.

One thing to note is that containers may not be immediately allocated to the AM, which 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 changes and needs additional containers.

3.4. Job scheduling in Hadoop 2.x

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

(a) 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 only when AM receives exactly n containers.

(b) Resource usage changes elastically, depending on the availability of resources. In this case, the job starts even if AM receives less than the required number of containers.

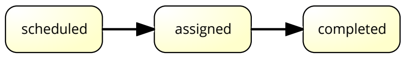


Figure 2: Lifecycle of a map task

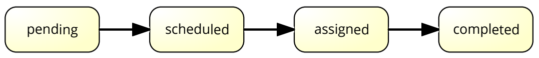


Figure 3: Lifecycle of a reduce task

Thus, for the 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[5], we observed that map and reduce tasks have different lifecycles presented in Figure 2 and Figure 3, respectively.

Vocabulary Used:

- pending → requests which are not yet sent to RM.
- scheduled →requests which are sent to RM, but not yet assigned.

[4]Package org.apache.hadoop.mapreduce.v2.app.rm; RMContainerAllocator class

[5]Package org.apache.hadoop.mapreduce.v2.app.rm; RMContainerAllocator.java class

- assigned → requests which are assigned to a container.
- completed → requests for which the container has completed the execution.

Furthermore, the 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, but 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 solution, we decide to take the performance model for MapReduce workloads proposed for Hadoop 1.x [1]. The main challenges of adapting the existing performance model to the architectural changes of Hadoop 2.x are: (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 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.

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 tasks of class i in the service center k
$AvgResponseTime_i$	Response time for tasks of class i
$\|M\|$	Number of map tasks
$\|R\|$	Number of reduce tasks
$MaxMapPerNode$	Maximum number of containers per node for map tasks (from AM configuration)
$MaxReducePerNode$	Maximum number of containers per node for reduce tasks (from AM configuration)

Table 2: Input parameters

4.1. Input Parameters

The input parameters for our model are presented in Table 2. We consider 2 types of resources (service centers): CPU&Memory and Network. The overall number of task classes C is 3 (i.e., map, shuffle-sort, and merge). We would like also to emphasize the difference between the *response* time and *residence* time of a task. The average response time is the total time that a task spends in the cluster, while the residence time of task class i on service center k is the average amount of time that a task of class i spends using the corresponding resource k during its execution (i.e., it does not include queuing delays).

4.2. Modified Reference Model

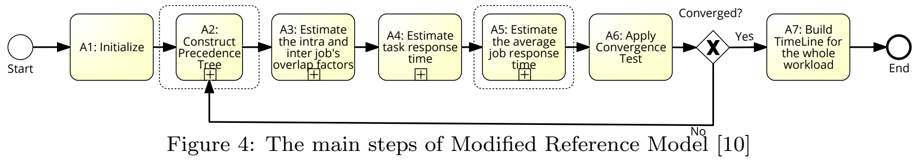

Start
A1: Initialize
A2: Construct Precedence Tree
A3: Estimate the intra and inter job's overlap factors
A4: Estimate task response time
A5: Estimate the average job response time
A6: Apply Convergence Test
Converged?
Yes
No
A7: Build TimeLine for the whole workload
End

Figure 4: The main steps of Modified Reference Model [10]

We build our performance model on top of reference model proposed by Vianna et al. [1]. 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, whose i-th component 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 residence time).

The main steps of the algorithm are presented in Figure 4, which consists of 7 main activities: A1-A7. 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 timeline for one job is constructed applying Algorithm 1. Using the obtained timeline, the precedence tree is constructed, capturing the synchronization delays introduced by map tasks and shuffle-sort phase of reduce tasks. The next step is to take into account the effects of the queuing delays by factors representing the overlap in the residence 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 applies 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 to build a new, and more accurate precedence tree based on the estimates of task response time obtained during the previous iteration. In case that current estimates are close enough to the previous ones, the algorithm converges, and as a result, a final job average response time and tasks response time are obtained. Finally, in step A7, using the obtained estimates, we construct the final timeline for the whole workload to estimate the total execution time. 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 (A2) and estimation of average job response time (A5).

A1: Initialization of task response time

Initialization activity consists of two sub activities 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:

(a) Using sampling techniques - taking the average of task response time from job profile.

(b) Obtaining them from the existing static cost models, for example, from Herodotou's cost models [2] (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.

According to our experiments, the second approach leads to faster algorithm convergence due to more accurate response time initialization and, as a consequence, less number of iterations of the algorithm. To confirm this, we further experimented with different errors added to the initial task duration obtained from the Herodotou's cost model. The results are presented in Figure 5, where $delta(t)$ is a percentage value we added to the initial task duration obtained from Herodotou's cost model. Indeed, the results show that the smaller the error is in the initial data, the faster will the algorithm converge. We would like to stress the robustness of this approach, as the potential error in the initial values does not affect the final obtained value. Thus, in our model, we use the second approach, while it still remains robust to other initialization methods.

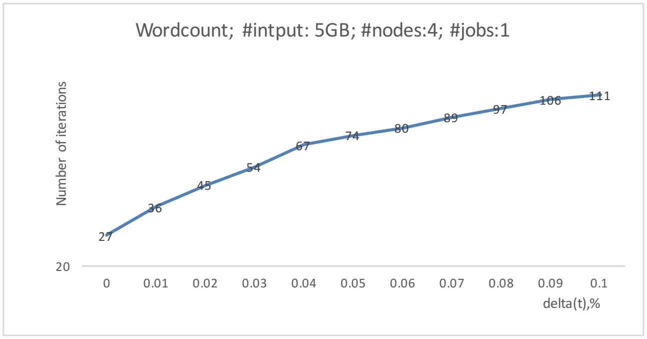


Figure 5

A2: 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. To represent

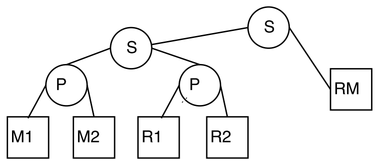


Figure 6: Precedence Tree

the execution flow of a MapReduce job, we will consider a binary precedence tree built from 2 types of primitive operators: serial (S) and parallel-and (P). The serial operator is used to connect tasks that run sequentially, whereas the parallel operator connects tasks that run in parallel. An example of a precedence tree is presented in Figure 6.

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 inter-dependencies. Based on new estimates for task response time that we obtain in step A4, we rebuild the precedence tree at each iteration of the algorithm. The complexity analysis of building a precedence tree can be found in Subsection 4.3.

The precedence tree depends on the response time of individual tasks that belong to one job and is built based on the timeline representing job's tasks execution inside the cluster. The timeline construction procedure is presented below.

Timeline Construction

For the sake of simplicity, we consider in the explanations a distributed cluster with a set of computing nodes equal to $numNodes$, all of them having the same technical characteristics (this can be easily generalized to heterogeneous clusters). The workload is composed of N MapReduce jobs executing concurrently in the system. Each job J_i has $|M_i|$ map tasks and $|R_i|$ reduce tasks. We are working at the granularity of complete map tasks without distinguishing different phases (like it was done in Herodotou's cost model [2]). As a partial sort is performed before each shuffle, we group each pair of sort and shuffle 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 kinds of subtasks: shuffle-sort and merge.

Based on the architectural analysis (see Section 3), the core factors that influence the timeline construction process are related to the job scheduling and to the resource management system, and can be defined as following:

1. **RM Scheduler.** 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. Without loss of generality, we assume a single, root queue, thus, resource allocation among jobs will be

in the FIFO order (i.e., the priority will be given to the first job requesting the resources).

2. **AM Scheduler.** Due to architectural changes, some responsibilities of job scheduling are delegated to the AM, thus we have to determine the way the AM distributes containers for tasks among 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 (see Figures 2 and 3).
3. **Binding of resources.** 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.
4. **Task priorities.** Considering the findings in Subsection 3.2 related to different priorities for map and reduce tasks, AM provides a container first to map task and after to reduce task.
5. **Locality constraints.** Assigning containers for map tasks mainly depends on whether we consider or not locality constraints (configuration parameter). In our model, we consider the node locality constraints for map task, but ignore the 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.
6. **Task dependencies.** Container allocation process for reduce tasks depends on the assignment of map tasks. First, it is necessary to check if all map tasks have been assigned. If yes, we schedule all reduce tasks (map output locality is not taken into consideration, request asks for a container on any host/rack). Otherwise, scheduling of reduce tasks is based on the *slow start* configuration parameter. By default, schedulers wait until some percentage of map tasks have completed before scheduling reduce tasks for the same job.

The last consideration is how to divide the timeline into phases. Tasks within the same phase can be executed one after another or in parallel, but tasks that belong to different phases are always executed sequentially, due to synchronization barriers. Tasks belong to different phases, if they are dependent and have precedence constraints. For example, shuffle-phase of reduce tasks cannot start before certain percentage of map tasks finish.

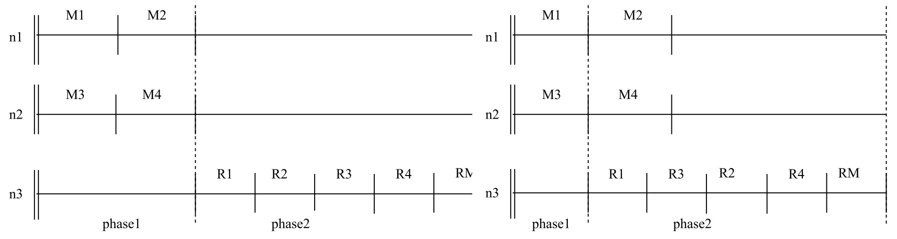
As a summary, we explain below our algorithm for the timeline construction of one job. We start in lines 1-7 distributing containers for map tasks. In case the slow start is set, the beginning of the shuffle-phase of reduce task will coincide with the end of map task on the node that has the lowest occupancy rate. Thus, shuffling starts as early as possible. In the opposite case, the shuffle-phase of reduce task starts as late as possible (lines 8-12). Further, in lines 13-24 we distribute containers for reduce tasks.

Algorithm 1 Timeline Construction for one job

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

Based on the obtained timeline, the precedence tree can be constructed uniquely up to graph isomorphism. In order to reduce the depth of precedence tree, we balance it.

Example. Applying the timeline construction algorithm to the running example from Section 3, we obtain the timelines, that are presented in Figure 7. In both cases, we first assign map tasks as they have higher priorities than reduce tasks. Assigning shuffle-phases of reduce tasks depends on the slow start configuration parameter. Without slow start (a), we are waiting while all map tasks finish, while with slow start (b), we can start assigning shuffle-phases of the reduce tasks, if a certain amount of map tasks was finished (e.g., 25%). Thus, based on the timeline assuming that we do not have slow start (a), we are able to construct the precedence tree depicted in Figure 8.



(a) without slow start (b) with slow start (25%)

Figure 7: Example of timeline construction

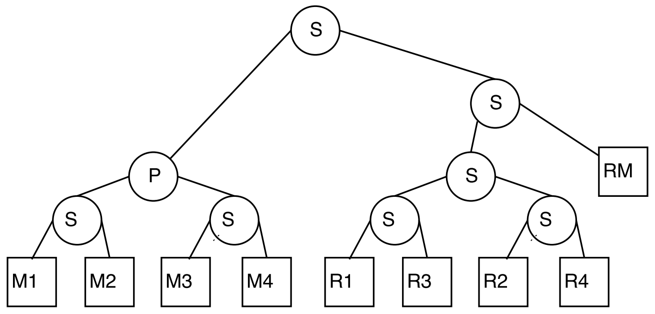


Figure 8: Precedence Tree Example (without slow start)

A3: 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 [12]. We consider

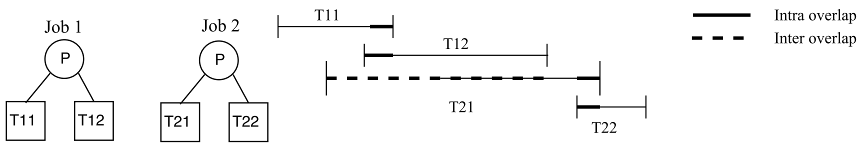


Figure 9: Intra- and inter-job overlaps

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 9, we provide an example for intra- and inter-job overlaps. We use continuous line to present overlaps in the time execution between tasks from the same job, and dash line to depict overlaps between tasks that belong to different jobs.

To estimate the overlap factors, we use the following formulas [10]:
$\alpha_{ij} = \frac{L_X(T_i,T_j)}{RT_i}$, $\beta_{kr} = \frac{L_I(T_k,T_r)}{RT_k}$,
where $L_X(T_i, T_j)$ and $L_I(T_k, T_r)$ are the overlap time of (T_i, T_j) and (T_k, T_r); RT_i and RT_k - average response time of task that belongs to the task class i and k, respectively. The obtained overlap factors are used in the next step to estimate a new value for task response time.

A4: Estimation of task response time

To predict the task response time, we apply modified approximate Mean Value Analysis (aMVA) [10]. MVA is an efficient technique that allows us to solve the queueing network models and obtain the mean values for tasks response time. It is based on the relation between the mean waiting time and the mean queue size of a system at the moment of arrival of the new job (i.e., having one job less in the system). The algorithm for estimating the task response time consists of 5 main steps that are presented in Figure 10. The detailed explanation of aMVA can be found in [10].

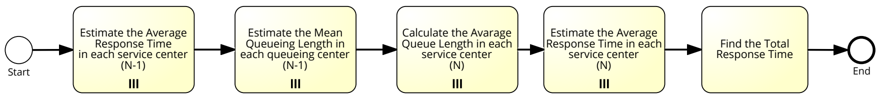

Start
Estimate the Average Response Time in each service center (N-1)
Estimate the Mean Queueing Length in each queueing center (N-1)
Calculate the Avarage Queue Length in each service center (N)
Estimate the Average Response Time in each service center (N)
Find the Total Response Time
End

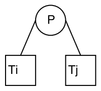
Figure 10: The main steps for task response time estimation [10]

A5: Average Job Response Time Estimation

For estimating the job response time, we need to know the dependencies between tasks (the precedence tree reflects them) and the estimations for the task durations. Then, going from bottom of the precedence tree to the top, we are estimating the durations for internal sequential and parallel nodes. In the end, the estimation of duration for the root node will be the duration for the whole job. As we always going from bottom to the top, the depth of precedence tree influences on the error of job response time estimation. In order to reduce the error, we balance the precedence tree.

There are 3 alternative approaches to estimate the job response time:

1. Fork/join-based [1]: 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 [13].

   $RT = H_k \cdot max(T_i, T_j)$ - response time,
   where $H_k = \sum_{i=1}^{k} \frac{1}{i}$ - harmonic function,
   k - is the number of child nodes

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

2. Tripathi-based [10]: 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 [10], [14]. We assume that the distribution of X is Erlang type if its $CV <= 1$, and Hyperexponential if $CV >= 1$. Knowing the distribution of leafs, we can determine the distribution type (Erlang or Hyperexponential) and then the mean value of response time for P and S [10].

3. The third approach is based on the assumption that task durations are independent variables and have Gaussian distribution. Indeed, we empirically showed using Pearson's Criterion, that durations of map and reduce tasks have Gaussian distribution, at 95% significance level. Thus, given the mean and variance of each leaf of the precedence tree, going from bottom to top, we can find the mean and variance for internal, P and S nodes, as mean and variance for functions of $F = max(T_1, T_2)$ and $F = T_1 + T_2$, respectively [10]. It is well known that $F = T_1 + T_2$ will also have the Gaussian distribution with mean and variance equal to $\mu = \mu_1 + \mu_2$, $\sigma^2 = \sigma_1^2 + \sigma_2^2$, respectively. According to [15], the distribution of $F = max(T_1, T_2)$ can be approximated by the Gaussian distribution with mean and variance equal to: $\mu(F) = \mu_1\Phi(\frac{\mu_1-\mu_2}{\theta}) + \mu_2\Phi(\frac{\mu_2-\mu_1}{\theta}) + \theta\phi(\frac{\mu_1-\mu_2}{\theta})$, and $\sigma^2(F) = \mu(F^2) - (\mu(F))^2$, where $\theta = \sqrt{\sigma_1^2 + \sigma_2^2 - 2\rho\sigma_1\sigma_2}$;
μ_1, μ_2 and σ_1^2, σ_2^2 - means and variances for T_1 and T_2;
Φ, ϕ - distribution and density functions for standard normal distribution;
$\mu(F^2) = (\sigma_1^2 + \mu_1^2)\Phi(\frac{\mu_1-\mu_2}{\theta}) + (\sigma_2^2 + \mu_2^2)\Phi(\frac{\mu_2-\mu_1}{\theta}) + (\mu_1 + \mu_2)\theta\phi(\frac{\mu_1-\mu_2}{\theta})$;
This approximation is reasonable only when the difference between two standard deviations is relatively small, thus, taking this into consideration, we search for pairs with the closest variance value in order to approximate the distributions of parallel nodes more accurately.

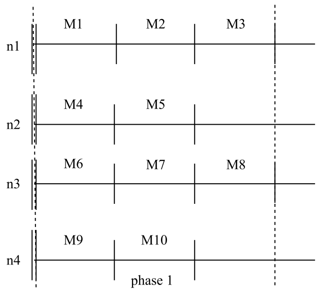
Example

This example illustrates the process during one phase. The timeline and the corresponding precedence tree are presented in Figure 11.

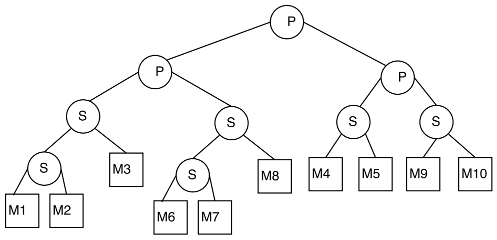
First, we define the distributions of variables that correspond to the sum of tasks per container:

- $F_1 = M_1 + M_2 + M_3$
- $F_2 = M_4 + M_5$
- $F_3 = M_6 + M_7 + M_8$
- $F_4 = M_9 + M_{10}$

The second step is to find, among obtained variables, pairs that have the closest values of variance, and to approximate the *max* by Gaussian distribution. In this example, those will be (F_1, F_3) and (F_2, F_4). Thus, we



(a) Timeline



(b) Precedence tree

Figure 11: Example of Timeline and corresponding Precedence tree

approximate the distribution of $max(F_1, F_3)$ and $max(F_2, F_4)$ by Gaussian distributions with mean and variance calculated as discussed above.

A6: 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 one. In case that the difference is below a certain value (i.e., $|RT^{curr} - RT^{prev}| \leq \epsilon$), 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 [10]. Theoretically, this value provides a good trade-off between the level of accuracy and the complexity of the algorithm, which we empirically confirmed (with lower values of ϵ the job response time almost does not change, meanwhile the number of iterations continues growing). The results are presented in Figure 12.

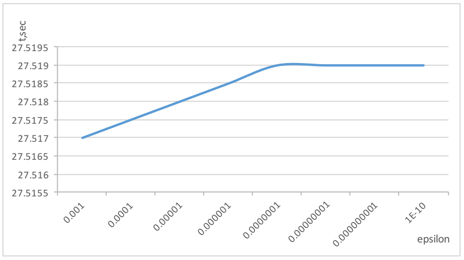


Figure 12: Dependency between ϵ value and job response time

A7: Building the final timeline

In order to build a timeline for the complete workload consisting of multiple jobs, we iteratively apply Algorithm 1 for each job taking in into consideration the initial assumption, that RM uses the Capacity scheduler with one root queue. Thus, resource allocation among jobs will be in FIFO order (Algorithm 2).

Algorithm 2 Timeline Construction for multiple jobs

```
Input: J – queue of jobs; N - set of cluster nodes
Output: TL – final timeline
1: for $i := 1$ to $|N|$ do
2:     $TL[i] := \emptyset$;
3: end for
4: for $j \in J$ do
5:     $TL := Algorithm1(TL, J.M, J.R)$
6: end for
7: Return $TL$;
```

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 [10], the MVA algorithm is computationally efficient, having a complexity of $O(T^2N^2K)$, where T is the number of tasks 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 $N_{containers}$ be the total number of containers in execution.

$T = allMapTasks + allShuffleSortTasks + allMergeTasks$;

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

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

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

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 $N_{containers}$ containers, that is $O(T \times N_{containers}) = O((m + r(m + 1)) \times (n \times max(pMaxMapsPerNode, pMaxReducePerNode)))$, where m, r -is the number of map and reduce tasks in the job. The computational cost of the whole solution: $O(T^2N^2K) + O(((m + r(m+1)) \times (n \times max(pMaxMapsPerNode, pMaxReducePerNode))) \times iterationNum)$. As we can notice, the computational cost of the whole solution is dominated by the MVA algorithm.

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 the comparison of our model against the measurements of the Hadoop 2.x real setup. To evaluate the accuracy of our performance model regarding the typical MapReduce jobs covering all parts of a MapReduce flow, we decided to use map-and-reduce-input heavy jobs (WordCount[6], Sort[7]) that process large amounts of input data and also generate large intermediate data [16].

5.1. Experimental Setup

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 1 TB SATA-3
- 4 Network Intel Gigabit Ethernet

We performed a set of experiments analyzing the workload response time estimated with three different approaches 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 concurrently in the cluster: 1,2,3,4.

For each experiment, we analyze the job response time fixing two out of three parameters. Each experiment is repeated 5 times and the medians of response time are plotted in the charts.

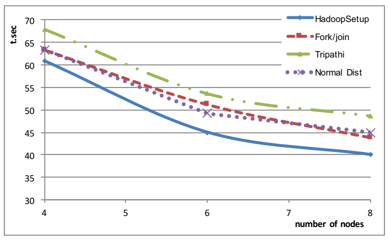
5.2. Results

All experiments have been done for 2 types of jobs: (a) *Wordcount* and (b) *Sort*. We are depicting the response time values taking into account three approaches for estimating the job response time (Section 4). First, we evaluate the accuracy of our performance model in terms of increasing number of nodes in the cluster and a fixed workload, i.e., for each experiment, we fix the number of concurrent jobs and the size of input data (see Figure 13 and Figure 14).

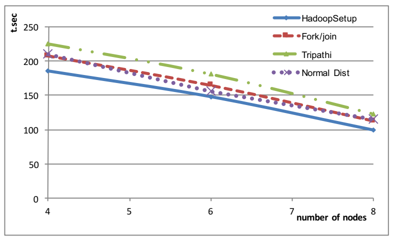
We can notice that the Fork/join based approach and the approach based on normal distribution of tasks response time provide more accurate estimation of job response time with error between 11% and 14,5%, while the Tripathi-based approach shows an error between 18% and 22%. For 5GB input size, we obtain the bigger value of an error: 14.5% and 22%, respectively. We observe that the accuracy of our algorithm depends on the number of map tasks and not directly on the size of input data. The observed differences in estimation errors are thus related to 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

[6]WordCount example from the Hadoop distribution: `https://wiki.apache.org/hadoop/WordCount`

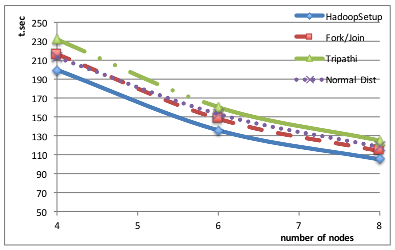
[7]Sort example from the Hadoop distribution: `https://wiki.apache.org/hadoop/Sort`



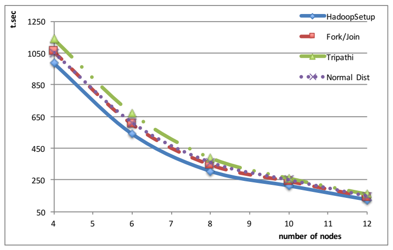
(a) Input: 1GB; #jobs: 1



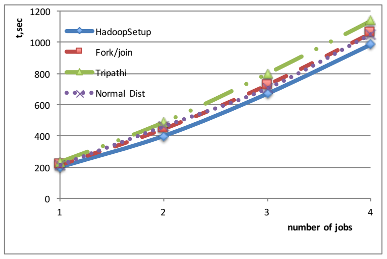
(b) Input: 1GB; #jobs: 4



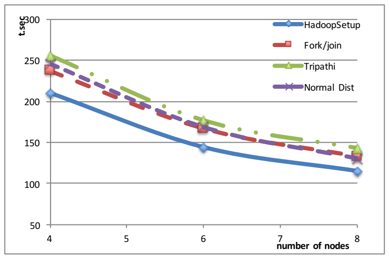
(c) Input: 5GB; #jobs: 1



(d) Input: 5GB; #jobs: 4



(e) #Nodes: 4; Input: 5GB



(f) Block: 64MB; Input:5GB;#jobs: 1

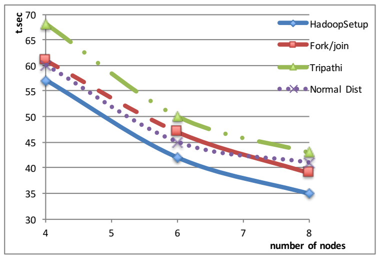
Figure 13: Results for *WordCount*

input data size. Thus, we reduced the default block size for the map task from $128MB$ to $64MB$ and repeated the experiments. These results for *WordCount* and *Sort* for the input data size equal to 5GB and number of jobs equal to 1 are presented in Figure 13 (f) and Figure 14 (f), respectively.

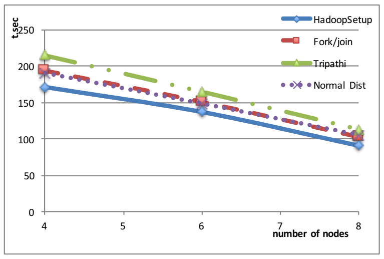
As showed by these results, experiments confirm our hypothesis, as we obtained the biggest values of errors: 15%, 17% and 23% for Fork/join, Normal Dist and Tripathi-based approaches, respectively. Although the accuracy does not directly depend on the input size, the block size should be reasonably chosen based on the size of input data and the cluster characteristics. As explained in Section 4.2, A5, for reducing the maximal depth of the precedence tree and, as a consequence, decreasing the error, we balance the constructed precedence tree.

The Fork/join approach and approach based on normal distribution of task durations in our model produces accuracy improvements over the original model for Hadoop 1 [1], on which we based our solution. For one job in the cluster we obtain competitive error rate results (13.5% accuracy error against 15% in [1]).

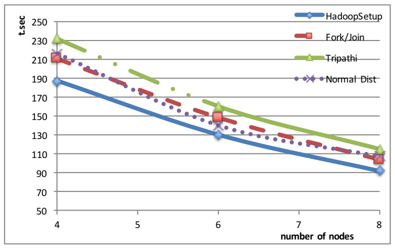
In conclusion, we can notice that the Fork/join based approach and the approach based on normal distribution of task durations provide more accurate results than the Tripathi-based one, but with all three approaches we overestimate the execution time.



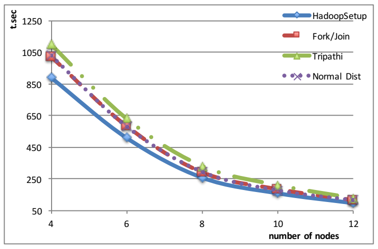
(a) Input: 1GB; #jobs: 1



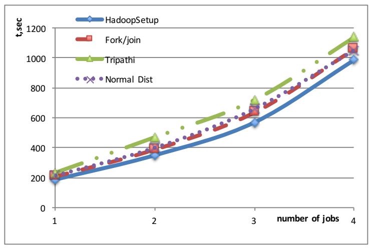
(b) Input: 1GB; #jobs: 4



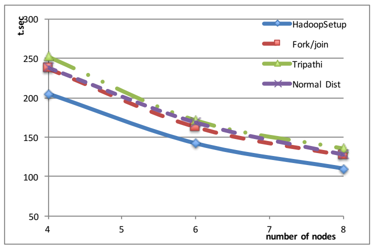
(c) Input: 5GB; #jobs: 1



(d) Input: 5GB; #jobs: 4



(e) #Nodes: 4; Input: 5GB



(f) Block: 64MB; Input:5GB; #jobs: 1

Figure 14: Results for *Sort*

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 [1], 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. Moreover, we checked the type of distribution for durations of map and reduce tasks and conclude that, unlike the assumptions in [10], task durations have Gaussian distribution. Based on this conclusion, we propose the third method for estimating the job response time, which gives us better results than Tripathi-based approach.

We validated our model against the measurements obtained from the real Hadoop setup for different number of jobs that were executed. 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 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. We are also planning to adapt our model to Hadoop 3.x [8]. The most important change appeared in the third version of Hadoop that can affect our model is the support for opportunistic containers[9]. The main goal of opportunistic containers is to improve the resource utilization and increase task throughput by dispatching the containers to NMs before they can even start their execution.

Furthermore, we are planning to generalize our solution, building the framework for cost estimation in distributed data processing engines, and instantiating it to other systems like Apache Spark[10] as well.

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.

References

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

[8] http://hadoop.apache.org/docs/r3.0.0-alpha4/

[9] http://hadoop.apache.org/docs/r3.0.0-alpha2/hadoop-yarn/hadoop-yarn-site/OpportunisticContainers.html

[10] http://spark.apache.org/; Accessed: 2017-01-17

[2] H. Herodotou, Hadoop performance models, arXiv preprint arXiv:1106.0940.

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

[4] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, A. Akella, Multi-resource packing for cluster schedulers, in: ACM SIGCOMM Computer Communication Review, Vol. 44, ACM, 2014, pp. 455–466.

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

[6] P. N. D. Bukh, The art of computer systems performance analysis, techniques for experimental design, measurement, simulation and modeling (1992).

[7] D. A. Menasce, V. A. Almeida, L. W. Dowdy, L. Dowdy, Performance by design: computer capacity planning by example, Prentice Hall Professional, 2004.

[8] C. P. Kruskal, A. Weiss, Allocating independent subtasks on parallel processors, IEEE Transactions on Software engineering (10) (1985) 1001–1016.

[9] A. Thomasian, P. F. Bay, Analytic queueing network models for parallel processing of task systems, IEEE Transactions on Computers (12) (1986) 1045–1054.

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

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

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

[13] E. Varki, Mean value technique for closed fork-join networks, in: ACM SIGMETRICS Performance Evaluation Review, Vol. 27, ACM, 1999, pp. 103–112.

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

[15] S. Nadarajah, S. Kotz, Exact distribution of the max/min of two gaussian random variables, IEEE Transactions on very large scale integration (VLSI) systems 16 (2) (2008) 210–212.

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