

# MapReduce Performance Models for Hadoop 2.x

IT4BI MSc Thesis

Student: Daria Glushkova  
Advisors: Petar Jovanovic, Alberto Abello

Master on Information Technologies for Business Intelligence  
Universitat Politècnica de Catalunya

Barcelona  
08/09/2016



A thesis presented by Daria Glushkova  
in partial fulfillment of the requirements for the MSc degree on  
*Information Technologies for Business Intelligence*

# 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 MapReduce paradigm. Performance analysis of concurrent job executions has been recognized as a challenging problem. Analytical performance models may provide reasonably accurate job response time at significantly lower cost than experimental evaluation of real setups.

In this thesis, we tackle the challenge of theoretically defining and implementing MapReduce performance models for Hadoop 2.x. We review the existing MapReduce performance models for the first version of Hadoop and conclude, that due to architectural changes and dynamic resource allocation, existing models could not be applied for Hadoop 2.x. The proposed solution is based on performance model for Hadoop 1.x that combines a precedence graph model, that allows to capture the execution flow of the job, and a queueing network model to capture the intra-job synchronization constraints due the contention at shared resources. We adopted this model to Hadoop 2.x by modifying the key step in the model construction. The accuracy of our solution is validated via comparison of our model against measurements of a real Hadoop 2.x setup. According to our evaluation results, the proposed model produces enough accurate estimates of average job response time, and allows further fine tuning of the model.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Background</b>	<b>2</b>
2.1	Hadoop Architecture . . . . .	2
2.2	Main components of YARN module . . . . .	4
2.3	Resource management in Hadoop 2.x . . . . .	6
2.4	Job scheduling in Hadoop 2.x . . . . .	7
<b>3</b>	<b>Related work</b>	<b>8</b>
3.1	Static MapReduce Performance Models . . . . .	8
3.2	Dynamic MapReduce Performance Models . . . . .	11
<b>4</b>	<b>Proposed Solution</b>	<b>13</b>
4.1	Input Cost Parameters . . . . .	13
4.2	The Modified Mean Value Analysis (MVA) Algorithm . . . . .	13
4.2.1	Initialization of task response time . . . . .	15
4.2.2	Building precedence tree . . . . .	15
4.2.3	Estimation of the Intra- and Inter- job overlaps factors . . . . .	18
4.2.4	Average Job Response Time Estimation . . . . .	19
4.2.5	Estimation of task response time . . . . .	19
4.2.6	Applying convergence test . . . . .	19
4.3	Complexity Analysis . . . . .	20
4.4	Implementation details . . . . .	21
<b>5</b>	<b>Evaluation</b>	<b>22</b>
<b>6</b>	<b>Conclusions and Future Work</b>	<b>26</b>
<b>7</b>	<b>Appendix</b>	<b>27</b>
7.1	Appendix A . . . . .	27
7.2	Appendix B . . . . .	28
7.2.1	Example of building a precedence tree . . . . .	28
7.2.2	Finding the optimal value for $\epsilon$ . . . . .	30
7.3	Appendix C . . . . .	31
7.3.1	Modified MVA using iterative approximation . . . . .	31
7.3.2	Response Time Estimation . . . . .	32
7.3.3	Estimation of Overlap Factors . . . . .	33
7.4	Appendix D . . . . .	34
7.4.1	UML class diagrams . . . . .	34
7.4.2	Comparison of results of modified AMVA, BardSchweitzerAMVA and exact MVA solution . . . . .	35
7.4.3	Evaluation results . . . . .	37
	References	45

# 1. Introduction

Distributed data processing systems have emerged as a necessity for processing large-scale data volumes in reasonable time. MapReduce is a programming paradigm for distributed processing of large data sets. The main idea of the MapReduce model is to hide the details of the parallel execution from users, so that they can focus only on data processing strategies. MapReduce operates in two main stages: Map stage and Reduce stage. Map stage consists of a set of Map tasks, each task is processing a block of input data. Reduce stage consists of 2 parts: Shuffle, that transfers the outputs of Map tasks to the Reduce tasks, and a set of Reduce tasks that further process groups of transferred data and output the final result to HDFS. Each Map and Reduce task consists of several phases, which may access and require different groups of resources. Thus, a MapReduce job is composed of a number of Map and Reduce tasks, which run in parallel but exhibit precedence constraints between map and shuffle tasks and synchronization delays due to sharing resources.

Programming in MapReduce is just a matter of adapting an algorithm to this peculiar two-phase processing model. Programs written in this functional style are automatically parallelized and executed on the computing clusters. Apache Hadoop is one of the most popular open-source implementation of MapReduce paradigm. All the modules in Hadoop are designed with a fundamental assumption that hardware failures are common and thus should be automatically handled by the framework. It provides strong support to fault tolerance, reliability, and scalability for distributed data processing scenarios. In the first version of Hadoop, the programming paradigm of the MapReduce and the resource management were tightly coupled. In order to improve the overall performance of Hadoop, some requirements were added, such as high cluster utilization, high level of reliability/availability, support for programming model diversity, backward compatibility, and flexible resource model [2]. The architecture of the second version of Hadoop has undergone significant changes: it decouples the programming model from the resource management infrastructure and delegates many scheduling functions to per-application components.

MapReduce-based systems are increasingly being used for large-scale data analysis applications. To minimize the execution time is vital for MapReduce application as well as for all data processing applications, especially in per-per-use cloud environments. One of the main requirements for optimizing the execution time is to estimate the execution as accurately as possible. For accurate estimation of the execution time, we need to build performance models that follow the programming model of data processing applications. Furthermore, a clear understanding of system performance under different circumstances is a 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.

There exist efforts for developing performance models for MapReduce taking into account Hadoop 1.x settings [3][10][19]. The existing cost models for Hadoop 1.x have been implemented in Starfish - an open source self-tuning system for big data analysis [6].

The architectural changes in version 2.x introduces the dynamic resource allocation to Hadoop. The cluster resources are now being considered as continuous, hence there is no static partitioning of resources per map and reduce tasks (i.e., 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. This thesis is dedicated to defining and evaluating the cost models for Hadoop 2.x. As a base of our model we took the analytical

performance model proposed for the first version of Hadoop in [19]. 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. and proposed a method for a timeline construction, based on which the precedence tree is built. The defined performance model for Hadoop 2.x must be finally evaluated for their accuracy, and if necessary further tuned for providing better estimations. We validated the accuracy via comparison of our model against measurements of a Hadoop 2.x setup.

In particular, our main contributions are as follows:

- Considering the architecture of Hadoop 2.x, we identify the main differences from the first version of Hadoop, focusing on those that can potentially affect the cost of the MapReduce job execution.
- Theoretically defining the MapReduce cost models for Hadoop 2.x. As a base for our performance cost model we took the mathematical model from [19] and adopt it for Hadoop 2.x.
- Implementation, tuning and accuracy evaluation of the MapReduce performance models for Hadoop 2.x.

#### **Outline**

The paper is organized as follows. In Section 2, we focus on the architecture of Hadoop 1.x and Hadoop 2.x, outlining the most significant differences and focusing on the resource management and job scheduling in Hadoop 2.x. In Section 3, we provide a review of the related work, describing existing approaches for constructing performance cost models for Hadoop 1.x. Theoretical definition of the analytical MapReduce performance models for Hadoop 2.x is presented in Section 4. Section 5 is dedicated to evaluation of the created MapReduce performance model for Hadoop 2.x. We summarize our achievements and present ideas for future work in Section 6. Finally, the Appendix contains the description of the main algorithms and intermediate results that were mentioned in previous sections.

## **2. Background**

In this section we will identify the most significant differences in the architecture of Hadoop 1.x and Hadoop 2.x, considering in details the job execution and resource requirement processes.

### **2.1 Hadoop Architecture**

The initial design of Apache Hadoop was mostly focused on processing and generating an immense amount of data through running MapReduce jobs. In the first version of Apache Hadoop architecture there were two significant drawbacks. The first shortcoming was a tight coupling of a specific programming model with the resource management infrastructure. All applications had to fit the MapReduce programming model. The second important drawback was centralized

handling of job's control flow that caused the problem of scalability for the scheduler. In order to improve the overall performance of Hadoop, some requirements were added, such as high cluster utilization, high level of reliability/availability, support for programming model diversity, backward compatibility, and flexible resource model [2]. These new requirements and the main limitations of the first version of Hadoop have caused significant changes in the architecture of Hadoop 2.x. The architecture of Hadoop 2.x decouples the programming model from the resource management infrastructure and delegates many scheduling functions to per-application components.

Figure 2.1 represents the transition from Hadoop 1.x to Hadoop 2.x.

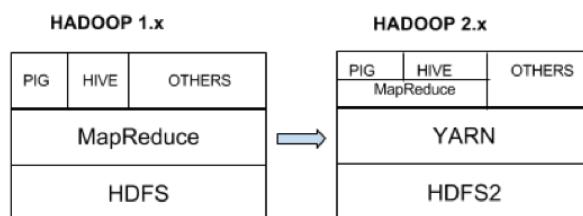


Figure 2.1: Comparison of architecture of Hadoop 1.x and Hadoop 2.x

The main components of Hadoop 1.x were:

- Hadoop Distributed File System (HDFS): A distributed file system that provides high-throughput access to application data [5].
- Hadoop MapReduce: Distributed programming model and associated implementation for processing and generating large datasets [1]. In the Hadoop's architecture there were two main components: Single master JobTracker (JT) and one slave TaskTracker (TT) per cluster node. Users submitted MapReduce jobs to the JobTracker, which coordinated its execution across the TaskTrackers. JobTracker was responsible for scheduling, 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. TaskTracker was configured with a fixed number of map slots and reduce slots. It means, that there was the fixed maximal number of map and reduce tasks that can run in parallel in one cluster node. TTs periodically heartbeated to the JT to report the status of running tasks on that node and to affirm its liveness.

The base Apache Hadoop 2.x framework is composed of the following modules:

- Hadoop Distributed File System (HDFS)
- Hadoop YARN (Yet Another Resource Negotiator): A module for job scheduling and cluster resource management [2]
- Hadoop MapReduce

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 job 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. In the new version of Hadoop, MapReduce is only one of the applications layered on top of YARN. YARN completely departs from the static partitioning of resources for maps and reduces, considering the cluster resources as a continuum, which brought significant improvements to cluster utilization. Thanks to decoupling of resource management and programming framework, YARN provides greater scalability, higher efficiency, and enables a large number of different frameworks to efficiently share a cluster. Programming frameworks running on YARN coordinate intra-application communication, execution flow, and dynamic optimizations, unlocking noticeable performance improvements.

## 2.2 Main components of YARN module

The YARN module consists of three main components:

- Global ResourceManager (RM) per cluster
- NodeManager (NM) per each node
- Application Master (AM) per each application

*The ResourceManager* provides scheduling of applications. Each application is managed by an ApplicationMaster that requests per-task computation resources in the form of containers. Containers are scheduled by the ResourceManager and locally managed by the per node NodeManager. A detailed description of the responsibilities and components of the ResourceManager, NodeManager, and ApplicationMaster are presented below.

The ResourceManager (RM) runs as a daemon on a dedicated machine one per cluster and arbitrates all the available cluster resources among various competing applications in the cluster. We will not go in detail of all components of RM [23] and will focus on the most important ones.

RM consists of two main components:

- 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. It is also responsible for termination and unregister-requests from any finishing AMs, obtaining container-allocation and deallocation requests from all running AMs and forward them over to the YarnScheduler. It also restarts AM on nodes in case of failure.

RM works together with the following components:

- The per node NodeManagers, which take instructions from the ResourceManager, manage resources available on a single node, and accept container requests from Application-Masters. NodeManagers are also reporting the resource status of their nodes back to the ResourceManager.



- The per application ApplicationMasters, which are responsible for negotiating resources with the ResourceManager and for working with the NodeManagers to start, monitor, and stop the containers.

The NodeManager (NM) is a special worker system daemon running on each node. It is responsible for managing resources available on a single node and accepting container requests from Application Masters. NM's main responsibilities can be found in [24].

The ApplicationMaster (AM) itself runs in the cluster just like any other container. The AM is managing all lifecycle aspects of application, including dynamically increasing and decreasing resources consumption, managing the flow of execution, and handling faults. The main responsibilities of Application Master can be defined as follows:

- Initializing the process of reporting liveness to the ResourceManager
- Computing the resource requirements of the application
- Translating the requirements into ResourceRequests that are understood by the YARN scheduler
- Negotiating those resource requests with the scheduler
- Based on the containers it receives from the RM, the AM may update its execution plan to accommodate perceived abundance or scarcity
- Launch of containers by communicating to NodeManagers
- Tracking the status of running containers and monitoring their progress
- Reacting to container or node failures by requesting alternative resources from the scheduler, if needed

Based on the core functionalities of YARN components, the general schema of job execution process in YARN can be determined as described in a Figure 2.2.

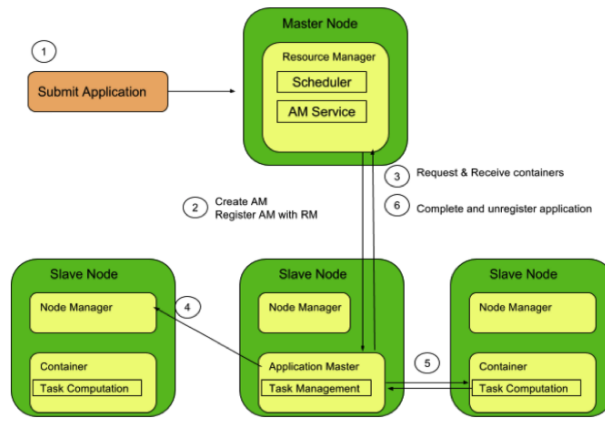


Figure 2.2: Job execution process in YARN

The main responsibilities of YARN:

1. The process starts when an application submits a request to the ResourceManager.
2. Next, the ApplicationMaster registers with the ResourceManager through AM Service and is started in the container that AM Service dedicated for it.
3. The ApplicationMaster then requests containers from the ResourceManager to perform actual work.
4. Once the ApplicationMaster obtains containers, it can proceed to launch of containers by communicating to a NodeManager.
5. Computation takes place in the containers, which keep in contact with the ApplicationMaster. Monitoring the progress is done inside the AM's container and it is strictly the AM's responsibility.
6. When the application is complete, ApplicationMaster should unregister from the ResourceManager.

## 2.3 Resource management in Hadoop 2.x

Let us consider in detail the resource requirement process.

Application Master needs to figure out its own resource requirements. Resource requirements can be:

- *Static*  
Resource requirements are decided at the time of application submission and when the ApplicationMaster starts running, there is no change in that specification. In case of Hadoop MapReduce, the number of map tasks is based on the input splits for MapReduce applications and the number of reducers on user input. Thus the total number of mappers and reducers is static and defined before the application submission.
- *Dynamic*  
When dynamic resource requirements are applied, the ApplicationMaster 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 ApplicationMaster can begin sending the requests in a heartbeat message, via the allocate API, to the ResourceManager. Based on the task requirements, AM calculates how many containers it needs and request those many containers. One thing to note is that containers will not be immediately allocated to the AM. This does not imply that the ApplicationMaster should keep on asking the pending count of required containers. Once an allocate request has been sent, the ApplicationMaster will eventually be allocated the containers based on cluster capacity, priorities and the scheduling policy. The ApplicationMaster should only request for containers again if and only if its original estimate changed and it needs additional containers.

The ApplicationMaster asks for specific resources via a list of ResourceRequests objects, and a list of containers ToBeReleased. The containers ToBeReleased are any containers that were

allocated by the scheduler in earlier cycles, but are no longer needed. The ResourceRequest object consists of the following elements:

- Priority of the request. When asking for sets of containers, an AM may define different priorities to each set. For example, the Map-Reduce AM may assign a higher priority to containers needed for the Map tasks and a lower priority to containers needed for the Reduce tasks. Higher-priority requests of an application are served first by the ResourceManager before the lower priority requests of the same application are handled. Potentially, resources of different capabilities can be requested at the same priority, in which case the ResourceManager may order them arbitrarily. There is no cross-application implication of priorities.
- The name of the resource location on which the allocation is desired. It currently accepts a machine or a rack name.
- Resource capability, which is the amount or size of each container required for that request.
- Number of containers, with respect to the specifications of priority and resource location that are required by the application.
- A Boolean relaxLocality flag (defaults to true), which tells the ResourceManager if the application wants locality to be loose or strict.

The response contains a list of newly allocated containers, the statuses of application-specific containers that completed since the previous interaction between the ApplicationMaster and the ResourceManager, and an indicator to the application about available headroom for cluster resources. The ApplicationMaster can use the container statuses to collect information about completed containers and, for example, react to failure. The headroom can be used by the ApplicationMaster to tune its future requests for resources. For example, the MapReduce ApplicationMaster can use this information to schedule map and reduce tasks appropriately so as to avoid deadlocks (e.g., to prevent using up all its headroom for reduce tasks).

## 2.4 Job scheduling in Hadoop 2.x

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

- All of the allocated containers may be required to run together a kind of scheduling where resource usage follows a static all-or-nothing model.
- Alternatively, resource usage may change elastically, such that containers can proceed with their work independently of the availability of resources for the remaining containers.

ApplicationMaster 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 ApplicationMaster 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. The MapReduce ApplicationMaster takes advantage of the dynamic two-level scheduling. When the MapReduce ApplicationMaster receives a container, it matches that container against the set of pending map

tasks, selecting a task with input data closest to the container, first trying data local tasks, and then falling back to rack locality.

According to all the above-described we can conclude, that the fundamental idea of YARN is to split the two major responsibilities of the Job Tracker that is, resource management and job scheduling/monitoring into separate daemons: a global ResourceManager and a per-application ApplicationMaster (AM). Specifically, a per-cluster ResourceManager tracks usage of resources, monitors the health of various nodes in the cluster, enforces resource-allocation invariants, and arbitrates conflicts among users. By separating these multiple duties that were previously shouldered by a single daemon, the JobTracker, in Hadoop 1.x, the ResourceManager can simply allocate resources centrally based on a specification of an application's requirements, but ignore how the application makes use of those resources. That responsibility is delegated to an ApplicationMaster, which coordinates the logical execution of a single application by requesting resources from the ResourceManager, generating a physical plan of its work, making use of the resources it receives, and coordinating the execution of such a physical plan.

## 3. Related work

We start this section by briefly reviewing in Subsection 3.1 the previous efforts to analyze the performance of MapReduce applications for the first version of Hadoop. All performance models described in Subsection 3.1 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 3.2 we introduced two most common approaches for modeling parallel applications and described the analytical performance model proposed for Hadoop 1.x that takes into consideration the queuing delays.

### 3.1 Static MapReduce Performance Models

In the MapReduce programming model there are two main stages: Map stage and Reduce stage. Map stage consists of a set of Map tasks, each task is processing a block of input data. Reduce stage consists of 2 parts: Shuffle, that transfers the outputs of Map tasks to the Reduce tasks, and a set of Reduce tasks that further process groups of transferred data and output the final result to HDFS. Each Map and Reduce task consists of several phases, which may access and require different groups of resources.

There are significant efforts and important results towards modeling the task phases in order to model accurately the execution of a MapReduce job in Hadoop 1.x.

Herodotou proposed performance cost models for describing the execution of MapReduce job on Hadoop 1.x in [3]. In his paper, performance models describe dataflow and cost information at the final granularity of phases within the map and reduce tasks. Models capture the following phases of Map task: read, map, collect, spill and merge. For the reduce task there are performance models for shuffle phase, merge phase and reduce and write phases. In terms of

Herodotou model the overall job execution time is simply the sum of the costs from all map and reduce phases and can be estimated using the following formulas:

$$totalJobTime = \begin{cases} totalMapsTime, & \text{if } pNumReducers = 0; \\ totalMapsTime + totalReducesTime, & \text{if } pNumReducers > 0; \end{cases} \quad (3.1)$$

where  $totalMapsTime$  and  $totalReduceTime$  can be obtained as following:

$$totalMapsTime = \frac{pNumMappers \times totalMapTime}{pNumNodes \times pMaxMapsPerNode}, \quad (3.2)$$

$$totalReducesTime = \frac{pNumReducers \times totalReduceTime}{pNumNodes \times pMaxRedPerNode}, \quad (3.3)$$

where  $pNumMappers$  - number of Map tasks;  $totalMapTime$  - the cost on one Map task;  $pNumNodes$  - the total number of nodes;  $pMaxMapsPerNode$  - number of slots per Map tasks for one node;  $pNumReducers$  - number of Reduce tasks;  $totalReduceTime$  - the cost on one Reduce task;  $pMaxRedPerNode$  - number of slots per Reduce tasks for one node.

As we can see in these cost formulas there is a fix amount of slots per Map and Reduce tasks -  $pMaxMapsPerNode$  and  $pMaxRedPerNode$  respectively. 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, there is no slot configuration in YARN allowing it to be more flexible. Thus we cannot apply Herodotou's cost formulas directly and it is necessary to find another approaches.

There has also been an effort of defining the low and upper bounds for job completion time and resource allocation to the job so that it finishes before required deadline. In [10], the authors proposed the framework called ARIA (Automatic Resource Inference and Allocation for MapReduce Environments) 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 a Makespan Theorem for greedy task assignment, which allows to identify the upper and lower bounds for the task completion time. Then, by Makespan Theorem, the job completion time lies between the following lower and upper bounds:

$$T_J^{Low} = T_M^{Low} + Sh_{avg}^1 + T_{Sh}^{Low} + T_R^{Low} \quad (3.4)$$

$$T_J^{Up} = T_M^{Up} + Sh_{Max}^1 + T_{Sh}^{Up} + T_R^{Up} \quad (3.5)$$

$$T_J^{Avg} = \frac{T_J^{Up} + T_J^{Low}}{2}, \quad (3.6)$$

where  $T_M^{Low}$  and  $T_M^{Up}$  - the lower and upper bounds for the duration of the entire map stage respectively;  $T_R^{Low}$  and  $T_R^{Up}$  - the lower and upper bounds of completion time for reduce phase;  $Sh_{avg}^1$ ,  $Sh_{max}^1$  - the average and maximum of task duration during the shuffle phases of the first reduce wave;  $T_{Sh}^{Low}$ ,  $T_{Sh}^{Up}$  - the lower and upper bounds on the duration of typical shuffle phase.

According to the research  $T_J^{Avg}$  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 10%, and hence, the predictions based on  $T_J^{Avg}$  are well suited for ensuring the job completion within the deadline. Authors also tackled the problem of finding the optimal number of map and reduce slots that need to be allocated to the job in order to guarantee job termination within time  $T_J^{Avg}$ . Thus, provided model can be used for defining the possible upper and low bounds for the job completion time as a function of the input dataset size and allocated resources. Nevertheless, this model has significant limitations that do not allow us to apply it to the second version of Hadoop. As in Herodotou performance cost models the proposed model uses the fixed amount of slots per map and reduce tasks within one node. Moreover, to be able to improve the overall performance it is necessary to change the Hadoop infrastructure and replace the standard scheduler by proposed deadline scheduling.

There has also been an attempt of evaluating the impact of task scheduling on system performance. 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 in [9] Tetris, a multi-resource cluster scheduler, that packs tasks to nodes based on their requirements of all resource types. This approach allows to avoid the main limitations of existing schedulers. The objective in packing is to maximize the task throughput and speed up job completion. Multi-resource packing of tasks is analogous to multidimensional bin packing. Given balls and bins with sizes in  $R_d$ , where  $d$  is the number of resources to be allocated, multidimensional bin packing assigns the balls to the fewest number of bins. Achieving good packing efficiency improves makespan but does not necessarily speed up individual jobs. Preferentially offering resources to the job with the smallest remaining time. 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 gain of over 30% in makespan and job completion time. The more detailed model description can be found in Appendix A.

It should be noticed that this model has a number of shortcomings:

- Fast solvers are only known for a few special cases with non-linear constraints, meanwhile several of the constraints are non-linear: resource malleability (1), task placement (2) and how task duration relates to the resources allocated at multiple machines (3). Finding the optimal allocation is computationally very expensive. Scheduling theory shows that even with elimination the placement considerations, the problem of packing multi-dimensional balls to minimal number of bins is APX-Hard [18].

- Ignoring dependencies between tasks. It is unacceptable in case of MapReduce jobs, where the shuffle/sort phase starts as the first map task is completed.
- New job arrival requires resolving the problem.

### 3.2 Dynamic MapReduce Performance Models

The main challenge in developing the analytical 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 estimate 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, are also available [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, as is commonly the case of MapReduce jobs.

Vianna et al. in their work [19] proposed analytical performance model for MapReduce workloads. Proposed model is based on reference model [12]. Given a tree specifying the precedence constraints among tasks of a parallel job as input, the reference model applies an iterative approximate Mean Value Analysis (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, the reference model cannot be directly applied to MapReduce workload due to the fact that in MapReduce job the beginning of shuffle phase of reduce task depends on the end of first map task.

Proposed in [19] model enhances the reference model. Contributions over classical reference model are the following:

- explicitly address the synchronization delays due to precedence constraints among tasks that cooperate in the same job, and show how to use the primitive task operators introduced in the reference model to build a precedence tree for it;
- taking into account queuing delays due to contention at shared resources;
- propose 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 model the distributed architecture with a closed queuing network with service centers representing each CPU, each disk, the fiber channel that connects the CPUs and disks and the

network. Memory constraints were not modeled. The workload is composed by a number  $N$  of jobs executing concurrently in the system. Each job has  $m$  map tasks and  $r$  reduce tasks. The numbers of map and reduce tasks that each worker node can execute in parallel are limited and given by the parameters  $pm$  and  $pr$  - the number of threads to process the map and reduce tasks respectively. The reduce task is composed by  $m$  shuffle-sort sub-tasks and one merge sub-task. Map tasks are not divided into subtasks.

The input parameters for the model can be divided into two categories:

- architecture parameters: the number of nodes  $n$ , the number of CPU's  $c$  and the number of exclusive disks  $d$  per node;
- workload parameters: the number of tasks of each type ( $m$  and  $r$ ), number of threads (per node) to process tasks of each type ( $pm$ ,  $pr$ ), number of threads (within each reduce task) to process shuffle tasks ( $ps$ ), and the service demand matrix ( $D_{ik}$ ), with the demand of each task  $i$  in each center  $k$ .

The task precedencies of MapReduce jobs cannot be defined beforehand, as a result, the precedence tree cannot be an input parameter as in the reference model. Authors solve this issue by proposing an algorithm to dynamically build the precedence tree for a job, and add it as an extra step to the algorithm proposed by Liang and Tripathi in [12]. The precedence tree is rebuilding at each iteration of the algorithm, using the average response times of individual tasks computed in the previous iteration. This approach allows to build a new more accurate precedence tree in the current iteration.

The main aim of the dynamic construction of precedence tree is to capture the execution flow of the job by taking into account the parallel/serial execution of individual tasks as well as their inter-dependencies. It gives a possibility to estimate the average response time of individual tasks and, by composition, of the whole job. In their work authors also proposed solutions of how to estimate when each task starts and finishes and the average response time of the internal nodes of the tree.

Once the precedence tree is built, the next step is to estimate the average job response time. Authors consider two alternative strategies to estimate the average job response time: Tripathi-based [12]: this strategy corresponds to the approach adopted by the authors of the reference model and the Fork/Join-based [13]. The evaluation results show that the second approach provides more accurate results.

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%. In the paper authors concentrated on the average job response time, but they mentioned that other performance metrics, such as throughput and resource utilization, can also be computed using the same approach.

Although this model does not capture the dynamic resource allocation and it has 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 Herodotus's models where there is no 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. Furthermore, it tackled the problem of estimation the average response time of parallel phases of job execution.

## 4. Proposed Solution

In this section, we describe the proposed analytical MapReduce performance cost model for Hadoop 2.x taking into consideration significant changes in the architecture with appearance of YARN and the dynamic resource allocation. The objective is to develop an efficient algorithm to approximately estimate two measures of interest: the mean response time of individual tasks and the mean response time for a job. In addition to mathematical representation, we also provide the logical representation of algorithm in terms of BPMN (Business Process Model and Notation).

As a basis of our MapReduce performance model for Hadoop 2.x, we decided to take the analytical performance model for MapReduce workloads proposed for Hadoop 1.x. in [19]. For constructing the performance model they proposed to use the reference model with dynamic precedence tree construction. According to our research on existing cost models for Hadoop 1.x., unlike others, this model is able to capture the queueing delays due the contention at shared resources and takes into account the pipeline parallelism of map and reduce tasks. Our main challenges were how to construct the precedence tree taking into consideration the dynamic resource allocation, as there is no predefined slot configuration per map and reduce tasks in the Hadoop 2.x and how to capture the synchronization delays introduced by the pipeline that occurs among maps and shuffle-sorts.

### 4.1 Input Cost Parameters

We have a distributed network with the amount of computers equal to  $numNodes$ , all of them have the same technical characteristics. The workload is composed by  $N$  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, followed by the final phase that applies the reduce function, is sequentially executed. We group these two phases into one merge subtask. Thus, according to our terminology, the reduce task is divided into following subtasks: shuffle-sort and merge.

The input parameters for our model are presented in the Table 4.1.

### 4.2 The 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 in [11] and it underlies in the reference model [12] on top of each, we build our analytical performance cost model. Bellow we describe the main steps of the algorithm and our assumptions.

Table 4.1: Input parameters for Performance Cost Model

Notation	Input Parameter
<b>Configuration parameters</b>	
$numNodes$	Number of Nodes
$cpuPerNode$	Number of CPU per node
$discPerNode$	Number of disks per node
<b>Workload parameters</b>	
$D_{i,k}$	Mean service demand of task class $i$ in center $k$
$m$	Number of map tasks
$r$	Number of reduce tasks
$MaxMapPerNode$	The maximum number of containers per node for map tasks
$MaxReducePerNode$	The maximum number of containers per node for reduce tasks
+ all from Herodotous's Model [3]	To initialize the task response time

Suppose a system with  $C$  task classes and  $K$  service centers. Let  $\vec{N}$  be a vector defining the number of tasks of each class in the system (workload),  $S_{jk}$  is the average demand of class  $j$  task on service center  $k$  (the average amount of time).

The main steps of the algorithm are presented in the figure below.

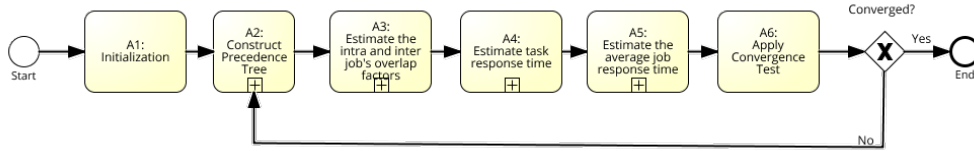


Figure 4.1: The main steps of Modified MVA algorithm

The algorithm 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 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 overlaps 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 of convergence test fails we return to the construction of precedence tree step trying to build a new more accurate precedence tree based on estimates of task response time obtained during the previous iteration. In case of current estimates are close enough to the previous ones, it means the end of algorithm, 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. We will consider 2 types of service centers: CPU&Memory and Network. For initializing the residence time, we take the average of residence time from the history of 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.
- Obtain from the 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. Based on this assumption we can apply Herodotou's formulas for map and reduce response time estimation. In Herodotou's cost models, map task execution was divided into five phases: Read, Map, Collect, Spill, Merge. The reduce task was divided into four phases: Shuffle, Merge, Reduce and Write. In our model each reduce is composed by  $m$  shuffle-sort subtasks and one merge subtask. Thus, we can initialize the map and reduce task response time applying Herodotou's cost formulas for a single map and reduce task respectively. The overall cost for a single reduce task according to [3] can be calculated as follows:

$$totalMapTime = \begin{cases} cReadPhaseTime + cMapPhaseTime + \\ cWritePhaseTime, & \text{if } pNumReducers = 0; \\ cReadPhaseTime + cMapPhaseTime + cCollectPhaseTime + \\ cSpillPhaseTime + cMergePhaseTime, & \text{if } pNumReducers > 0; \end{cases} \quad (4.1)$$

According to our terminology, the reduce task is divided into following subtasks: shuffle-sort and merge. Then the overall cost for a single reduce task according to [3] can be estimated as:

$$shuffleSortTask = cShufflePhaseTime \quad (4.2)$$

$$merge = cMergePhaseTime + cReducePhaseTime \quad (4.3)$$

This approach should guarantee the less number of iterations of algorithm due to more accurate response time initialization and, as consequence, the faster algorithm convergence.

### 4.2.2 Building precedence tree

In Appendix B we provide an example of timeline construction and precedence tree building procedure.

In the 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 ( $Pa$ ).  $S$  operator is used to connect tasks that run sequentially, whereas  $Pa$  operator connects tasks that run in parallel. Our main goal with the precedence tree is to capture the execution flow of the job, identifying the parallel or serial order of the execution of individual tasks and their inter-dependencies. To be able to obtain as accurate estimates of task response time as possible, 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. Then, tasks within the same phase are executed in parallel, meanwhile tasks from different phases are executed sequentially.

The algorithm for timeline construction will be presented below. For better understanding the key steps of the algorithm we need to consider the main factors that could effect the timeline construction process. The core assumptions and factors that influence on the timeline construction process can be divided into two subgroups - related with the job scheduling and resource management system.

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

1. We assume that RM has a Capacity scheduler as it is the default scheduler that comes with the Hadoop YARN distribution. The fundamental unit of Capacity scheduler is a queue. A queue is either a logical collection of applications submitted by various users or a composition of more queues. For simplicity, we assume that we do not have any hierarchical queues and we have only one root queue. Thus, resource allocation within applications will be in FIFO order, i.e., the priority will be given to the first application in the queue.
2. Due to architectural changes, some responsibilities of job scheduling are dedicated to the AM. We have to determine the way to distribute containers for tasks within different nodes. Looking through the source code of MapReduce Application Master (package `org.apache.hadoop.mapreduce.v2.app.rm`; `RMContainerAllocator.java` class), we found that map and reduce tasks have different lifecycles that are presented in the figures below.

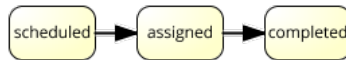


Figure 4.2: Lifecycle of map task



Figure 4.3: Lifecycle of reduce task

*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 → request corresponding to which container has completed

3. Ignore late binding. We are assuming that MapReduce AM will use requested containers for the same type of tasks as originally requested.

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

1. In the resource request object containers can have different priorities. Higher-priority requests of an application are served first by the ResourceManager. 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 correspondingly. This finding allows us to provide a container first to map task and after to reduce task (depending on slow start configuration parameter and the amount of finished map tasks.)
2. Assigning containers for map tasks mainly depends on whether we consider or not locality constraints (configuration parameter). If we are taking into account locality constraints then we have to obey three rules:
  - try to assign to all nodes first to match node local
  - try to match all rack local
  - assign remaining

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 occupancy rate value.

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

- Check for slow start. If there are enough completed map tasks (by default `mapreduce.job.reduce.slowstart.completedmaps = 5%`) go to the second step.
- Check if all maps are assigned:
  - no  $\rightarrow$  schedule reducers based on the percentage of completed map tasks (conf parameter)
  - yes  $\rightarrow$  schedule all reduce tasks (map output locality is not taking into consideration, request ask for a containers 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 the task indicates the start of new phase.

As a summary, we present below an algorithm for the timeline construction. Considering that map tasks have higher priority than reduce tasks. We start in lines 1-10 to distribute containers for map tasks, taking into account the node locality constraints. In case of slow start is set and there are enough completed map tasks, we start to distribute containers for reduce tasks. Further, in lines 11-24 we distribute the rest of required containers for reduce tasks.

- 1: **for** all requested containers for map tasks **do**
- 2:   **if** (*slow\_start* is set) and  
       (the percentage of completed map tasks is greater 5%) and  
       (there are requested containers for reduce tasks)  
       **then**
- 3:     Distribute container for reduce task among nodes with the highest capacity rate;

```

4:   Reduce the amount of requested containers for reduce tasks by 1;
5: end if
6:   Distribute container among nodes considering the node locality constraints;
7:   Fix the start and end time of map task;
8:   Add map task to the set of completed map tasks;
9: end for
10: if slow_start is set then
11:   for all distributed containers for reduce tasks do
12:     for all completed map tasks do
13:       Fix the start and end time of shuffle-sort phase for map task;
14:     end for
15:     Fix the start and the end of merge task;
16:   end for
17: end if
18: for all requested containers for reduce tasks do
19:   Distribute container among nodes with the highest capacity rate;
20:   for all completed map tasks do
21:     Fix the start and end time of shuffle-sort phase for map task;
22:   end for
23:   Fix the start and the end of merge task;
24: end for

```

### 4.2.3 Estimation of the Intra- and Inter- job overlaps factors

For a system with multiple classes of tasks the queueing delay of task  $i$  class due to class  $j$  task is directly proportional to their overlaps [22]. We are going to consider two types of overlaps factors: the intrajob overlap factor  $\alpha_{ij} \forall i, j$  - taskID's from the same job and interjob overlap factor  $\beta_{kr} \forall k, r$  - taskID's from different jobs. The intrajob overlap factor measures the overlaps between tasks from the same job meanwhile the interjob overlap factors reflect overlaps between tasks from different jobs. In the Figure 4.4 we provide an example for intra- and inter- job overlap factors.

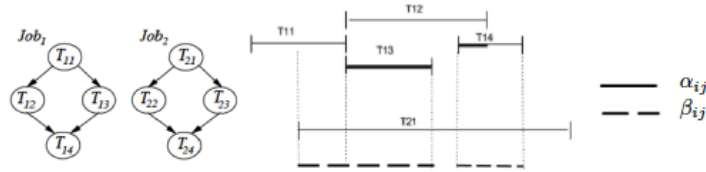


Figure 4.4: Intra- and inter- job overlap factors

The algorithm for estimation the overlap factors can be found in Appendix C, Subsection 7.3.3.

#### 4.2.4 Average Job Response Time Estimation

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

1. Tripathi-based[12]:

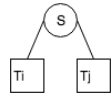
To estimate the response time of a PA-rooted sub-tree, we approximate the distribution of response time of each of its children by either an Erlang or a Hyperexponential distribution depending on the coefficient of variation ( $CV$ ) of the response times associated with each child node. We assume that the distribution of  $X$  of Erlang type if its  $CV \leq 1$ , and Hyperexponential distribution if  $CV \geq 1$ .

The Precedence tree for MapReduce job will have only 2 types of nodes:  $P_A$  and  $S$ . Knowing the distribution of leafs we can determine the distribution type (Erlang or Hyperexponential) for  $P_A$  and  $S$  [12].

2. Fork/join based

We consider the execution of a parallel-phase as a fork-join block, and use previously adopted estimates of the average response time of fork/joins. One such estimate is the product of the  $k$  - th harmonic function by the maximum average response time of  $k$  tasks [13].

$$R_{ik} = H_k \cdot \max(T_i, T_j),$$

$$\text{where } H_k = \sum_{i=1}^k \frac{1}{i}, \text{ where } k \text{ - is the number of children nodes}$$


#### 4.2.5 Estimation of task response time

To solve the queuing network models we apply Mean Value Analysis (MVA)[11], which focuses on computing the average value of response time for each task. MVA is based on a 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 the Figure 4.5

The detailed explanation of each step can be found in Appendix C Algorithm 2.

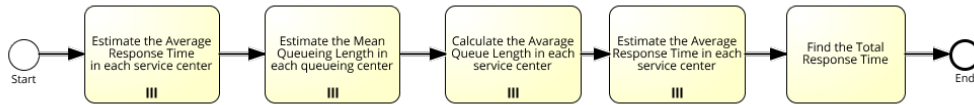


Figure 4.5: The main steps for task response time estimation

#### 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, it means the end of algorithm, otherwise we return to the precedence tree construction process and repeat activities A2-A6. The algorithm is presented bellow.

```

1: if  $|R_i^{curr} - R_i^{prev}| \leq \epsilon, \forall i$  then
2:   Calculate the Performance Metrics of the System
3:   Stop
4: else
5:   Set  $R_i^{prev} = R_i^{curr} \forall i$ 
6:   Go to Precedence tree construction process
7: end if

```

We assume that  $\epsilon = 10^{-7}$ , which is the recommended value for MVA [12]. 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 some 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. The test results can be found in Appendix B, Subsection 7.2.3.

### 4.3 Complexity Analysis

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

According to [12], the MVA algorithm is computationally efficient, it has 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.

Precedence tree is recomputed at each iteration of the algorithm. 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.

$C = allMapTasks + allShuffle + sortTasks + allMergeTasks$ . The total number of containers  $T = n \times \max(pMaxMapsPerNode, pMaxReducePerNode)$ , 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 polynomial complexity equal to  $O(C^2N^2K)$ .



## 4.4 Implementation details

Our model is implemented in Java programming language. The high level diagram of our solution is presented in the Figure 4.6, while more detailed UML class diagrams can be found in Appendix D.

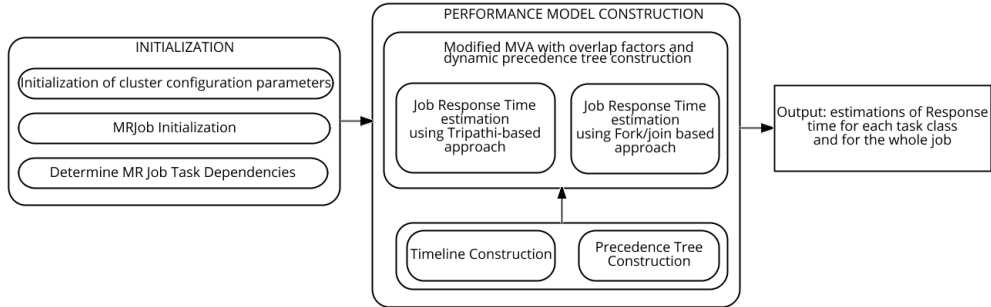


Figure 4.6: The high level diagram of implementation solution

The first step is initialization. We have to specify the cluster configuration parameters: the number of nodes, CPU and memory for each node. We need to determine the total workload in the cluster - the number of MapReduce jobs that are executed simultaneously. We also need to initialize each MapReduce job: the number of map and reduce tasks and to determine the demand matrix (the average time that each task spent in each service center) and the average response time for each task. For determining the demand matrix we were using information from the history of real job executions. For initializing the average response time for map and reduce tasks we were using the what-if analysis component of Starfish [6].

The next step is to construct the performance model. The performance model construction module can be divided into three main parts:

- Modified MVA algorithm with estimation of intra- and inter- overlap factors and dynamic precedence tree construction
- Timeline construction
- Precedence tree construction

For the implementation of the modified MVA, we used the algorithm described in Section 4.2. The implementation of this algorithm was done by extending Java Modeling Library (JML) [25]. As can be seen from UML class diagrams (Figure 7.4.1), the approximate MVA with dynamic precedence tree construction is extended from SolverMultiClosedAMVA class, i.e., the implementation of standard approximation of MVA algorithm that was included in JML. We implemented the modification of this class and compared results with the exact MVA, which is included in JML by default. The test results are presented in Appendix D, Subsection 7.4.2. As we can see, proposed in [12] approximate MVA algorithm provides more accurate results comparing with BardSchweitzerAMVA - approximate MVA algorithm that was included in JML. Furthermore, based on this approximation of MVA, we implemented approximate MVA with dynamic precedence tree construction that is described in Section 4.2. We implemented two approaches: Tripathi-based and fork/join-based for response time estimation. The explanation of these approaches for the response time estimation can be found in Subsection 4.2.3.

Timeline and precedence tree construction procedures were explained in Section 4.2.2. The representation of precedence tree was implemented extending the Stanford CS Education Library for binary trees [26].

## 5. Evaluation

This section presents the results of a set of experiments we performed with the proposed analytical performance model. We provide the validation results from a comparison of our model against measurements of a Hadoop 2.x setup. For evaluation we decided to use map-and-reduce-input heavy jobs that process large amounts of input data and also generate large intermediate data. The selected application to run on a Hadoop system was *wordcount*.

We performed a set of 36 experiments analyzing the job response time in terms of following parameters:

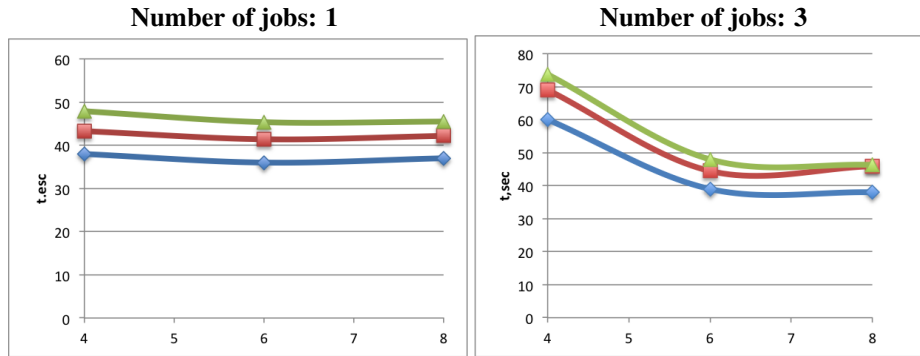
- the number of nodes: 4,6,8;
- the size of input data: 0.5GB, 1GB, 10GB;
- the number of jobs that are executed simultaneously in the cluster: 1,2,3,4.

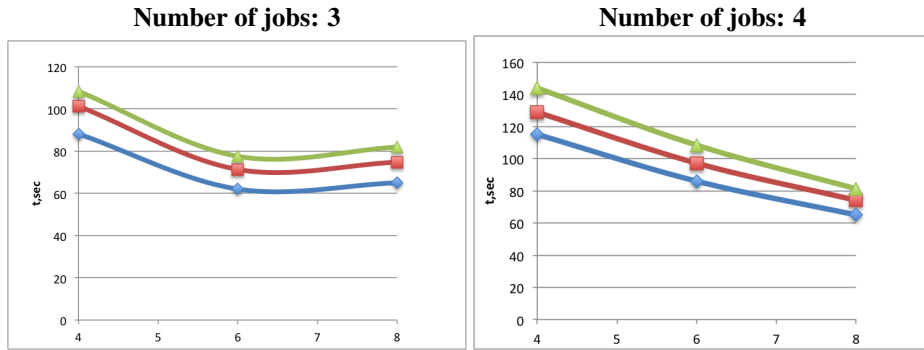
Each node in the cluster has the same technical characteristics:

- 2x Intel Xeon E5-2630L v2 a 2.40 GHz
- 128 GB de memria RAM
- 1 disc dur d'1 TB SATA-3
- 4 targes de xarxa Intel Gigabit Ethernet

For each experiment we analyze the job response time fixing two parameters and one is changing.

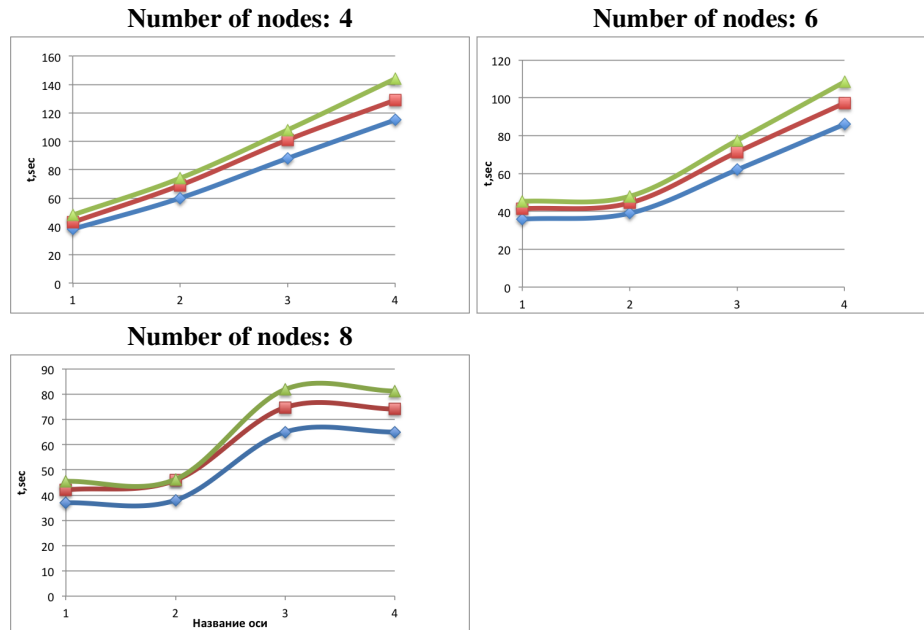
First, we present the response time for different number of jobs that are executed simultaneously in the cluster (from 1 to 4) on different number of nodes (4,6,8) and the fixed size of input data equal to 0.5GB. Results are shown below.





As we can see from the graphs, the Fork/join based approach provides more accurate results, with average error  $\approx 15\%$ , meanwhile the Tripathi-based gives us less accurate estimation of job response time with error  $\approx 25\%$ .

The second series of graphs that we would like to present show the response time depending on the number of jobs that are executed simultaneously in the cluster on different number of nodes. We performed experiments for the number of nodes equal to 4, 6, and 8. Results are presented below.



Analyzing graphs we can conclude that Fork-join based approach gives more accurate results.

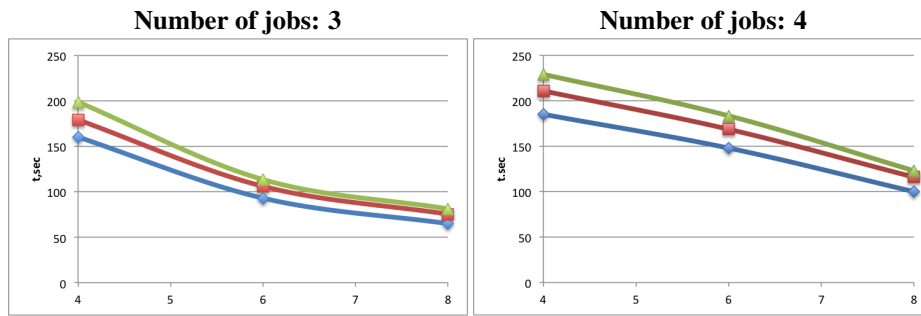
We repeated the same experiments for input equal to 1GB and 10 GB. Results are in the tables below. As for the input 0.5, Fork/join based approach provides more precise estimations with the average error  $\approx 15\%$  for the input size = 1GB and error  $\approx 18\%$  for the input size equal to 10GB. The error using the Tripathi-based approach is noticeably higher -  $\approx 24\%$  and  $40\%$  for

1GB and 10GB correspondingly.

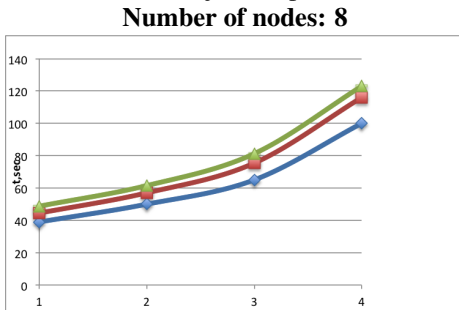
The complexity ( the maximal depth) of precedence tree is directly proportional to the size of input data, the number of map and reduce tasks will increase with the increasing size of input data. In both approaches, we calculate the job response time applying rules, that were described in Subsection 4.2.4, going bottom-up, accumulating errors. Thus, we obtained the higher error values for the bigger input data size.

### Results for 1GB

The number of nodes/response time(sec); fixed number of jobs



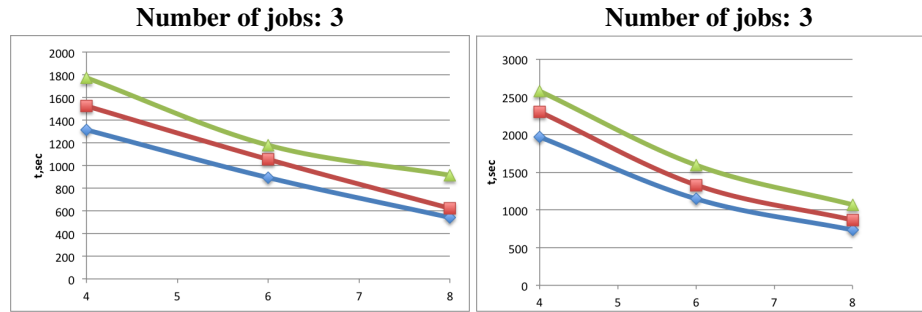
The number of jobs/response time(sec); fixed number of nodes



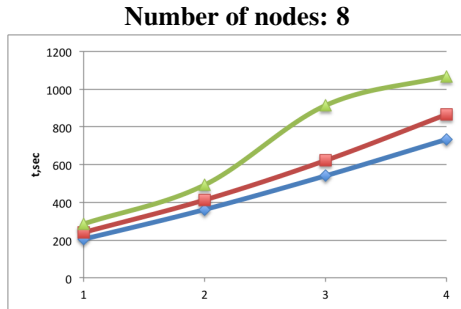
More graphs for 1GB input size can be found in Appendix D, Subsection 7.4.3.

### Results for 10GB

The number of nodes/response time(sec); fixed number of jobs



The number of nodes/response time(sec); fixed number of jobs



More graphs for 10GB input size can be found in Appendix D, Subsection 7.4.3.

In conclusion, we can notice that the Fork/join based approach provides enough accurate estimation of job response time with error between 12% and 17% depending on the input size, meanwhile the Tripathi-based approach shows worse results with an error of more than 40% for large volumes of data. Our future plan will be to tune our model for providing better results for bigger input sizes.

## 6. Conclusions and Future Work

In this thesis, we tackled the challenge of creating the 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 [19], which was based on reference model where the execution flow of a job was presented by precedence tree and the contention at the physical resources were captured by a closed queuing network. Our main contributions is the adaptation of existing solution to Hadoop 2.x.. Considering changes in the architecture of the second version of Hadoop 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 measurements of 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 is around 16%. Our model can be used for theoretically estimation of the jobs response time at a significantly lower cost than simulation and experimental evaluation of real setups. It can also be helpful in 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 per each task and the whole job.

## 7. Appendix

### 7.1 Appendix A

The constructed analytical model is based on task profiles. Task demands of four types of resources are considered: CPU cores, Memory, Disk and Network bandwidth. For every resource  $r$  are determined: the capacity of that resource on machine  $i$  as  $c_i^r$ , the demand of task  $j$  on resource  $r$  as  $d_j^r$ .

The model provides the estimation for job duration:

$$duration_j = \max \left( \begin{array}{l} \frac{f_{ij}^{diskR}}{\sum_t X_{i,j}^{cpu,t}}, \quad \frac{f_j^{diskW}}{\sum_t X_{i,j}^{diskW,t}}, \\ \forall i \frac{f_j^{diskR}}{\sum_t X_{i,j}^{diskR,t}}, \\ \frac{\sum_{i \neq i_j^*} f_{ij}^{diskR}}{\sum_t X_{i,j}^{netIn,t}}, \\ \forall i \neq i_j^* \frac{f_{ij}^{diskR}}{\sum_t X_{i,j}^{netOut,t}} \end{array} \right), \quad (7.1)$$

where  $X_{ij}^{r,t}$  - task  $j$  is allocated units of resource type  $r$  on machine  $i$ , at time  $t$ . From top to bottom terms correspond to:

- cpu cycles, writing output to local-disk;
- reading from disks;
- network bandwidth into the machine that runs the task;
- network bandwidth out of other machines that have task input.

It is also necessary to take into account the following constraints:

- The cumulative resource usage on a machine  $i$  at any given time cannot exceed its capacity:

$$\sum_j X_{i,j}^{r,t} \leq C_i^r \forall i, t, r \quad (7.2)$$

- Tasks need no more than their peak requirements and no resources are allocated when tasks are inactive.

$$0 \leq X_{i,j}^{r,t} \leq d_i^r \forall i, t, r \notin \quad (7.3)$$

$$\forall i, t, r X_{i,j}^{r,t} = 0 \text{ if } t \notin [start_j, start_j + duration_j] \quad (7.4)$$

- The cumulative resource usage on a machine  $i$  at any given time cannot exceed its capacity:

$$\sum_{t=start_j}^{start_j+duration_j} Y_{i,j}^t = \begin{cases} duration_j, & \text{machine } i = i_j^* \\ 0 & \end{cases} \quad (7.5)$$

Then the problem of finding the optimal job schedule is equivalent to the optimization problem of minimizing the total makespan:

$$Makespan = \max_{job_j} \max_{task_{i \in J}} \max_{time_t} (Y_{ij}^t < 0) \quad (7.6)$$

where  $Y_{ij}^t$  - the indicator variable that is 1 if task  $j$  is allocated to machine  $i$  at time  $t$ ,  $i_j^*$  - the machine that task  $j$  is scheduled at.

## 7.2 Appendix B

### 7.2.1 Example of building a precedence tree

Assume that we have input data on  $n_1$  and  $n_2$  nodes and replicas on  $n_3$  and  $n_4$  correspondingly. All nodes have the same capacity.  $n = 4; m = 4; r = 1$ ; where  $n$  - total number of nodes,  $m$  - number of containers for map tasks,  $r$  - number of containers for reduce tasks.

ResourceRequest Object:

Number of containers	priority	size	locality constraints	
2	20	x	$n_1$	For map tasks
2	20	x	$n_2$	For map tasks
1	10	x	*	For reduce tasks(shuffle-sort + merge)

Timeline construction (considering host locality constraints) is presented in the Figure 7.1. The Precedence tree is constructed based on timeline and presented in the Figure bellow.

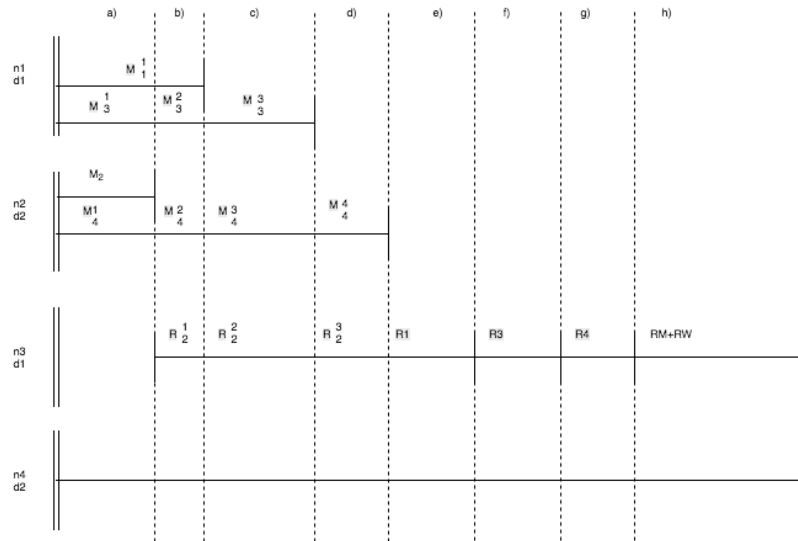


Figure 7.1: Example of timeline construction



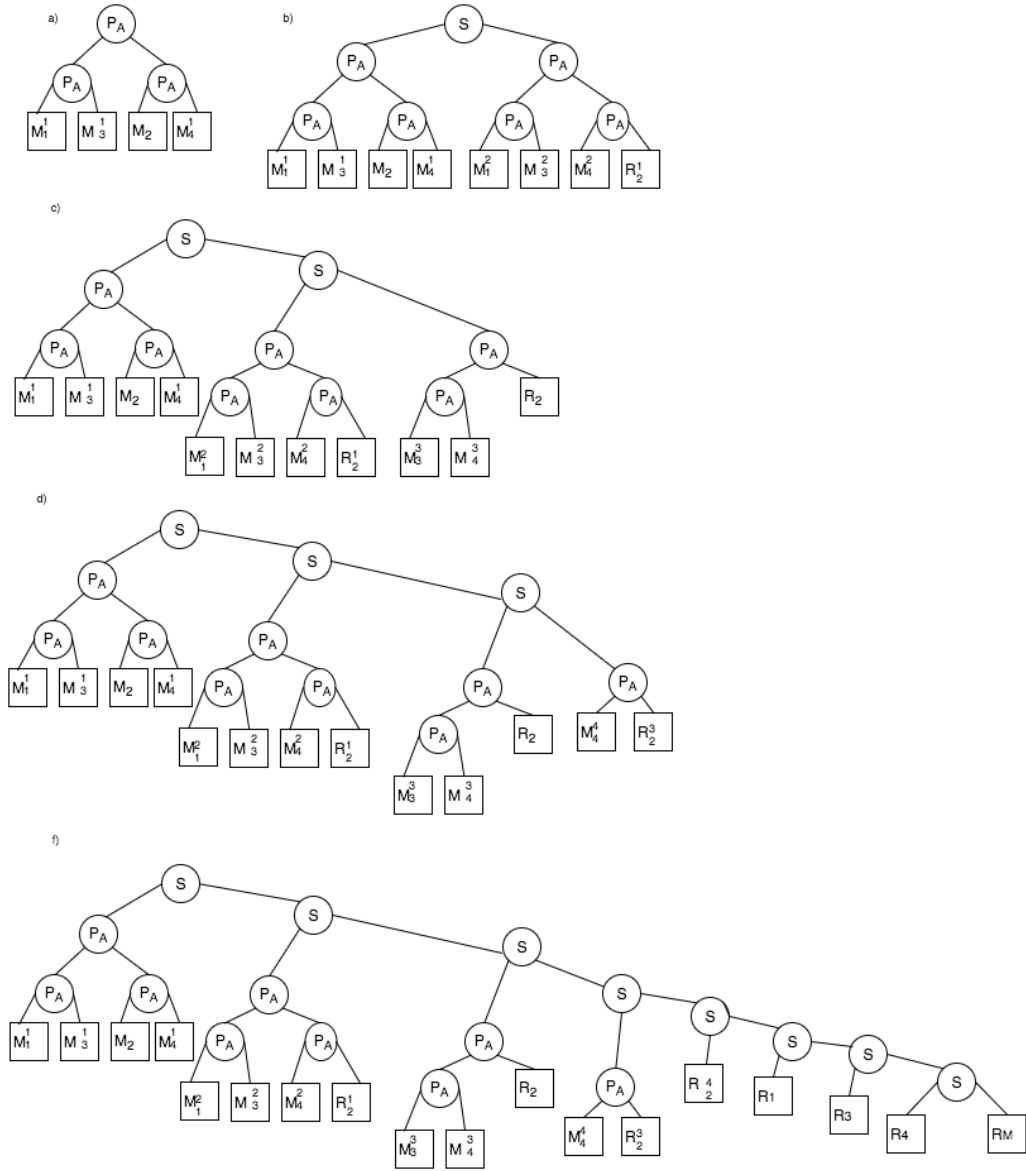


Figure 7.2: Example of Precedence tree construction

As we can see from this example, it is not important on which node we receive container per map or reduce task, it does not influence the precedence tree construction.

## 7.2.2 Finding the optimal value for $\epsilon$

We performed an experiment for finding the optimal value for epsilon, by running the Hadoop example for Pi value estimation.

bin/hadoop jar share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.2.jar pi 2 10000000.

The results are presented below.

$\epsilon$	job execution time(sec)	difference	iterations
$10^{-3}$	27.5180319668948	27.5180319668948	3
$10^{-4}$	27.5196451587285	0.00161319183370	5
$10^{-5}$	27.51966550911	0.0000203503814972	6
$10^{-6}$	27.5196671285146	0.0000016194046033	7
$10^{-7}$	27.5196672782758	0.0000001497611990	9
$10^{-8}$	27.5196672628305	0.0000000154453019	10
$10^{-9}$	27.5196672677622	0.0000000049316995	12
$10^{-10}$	27.5196672680584	0.0000000002962004	13
$10^{-11}$	27.5196672680132	0.0000000000451976	15
$10^{-12}$	27.5196672680109	0.0000000000023022	17
$10^{-13}$	27.5196672680107	0.0000000000001990	18
$10^{-14}$	27.5196672680107	0.0000000000000000	20
$10^{-15}$	27.5196672680107	0.0000000000000000	21

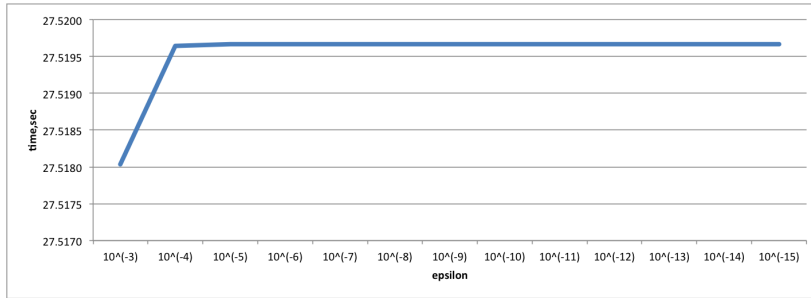


Figure 7.3: Dependence between  $\epsilon$  and Job response time

## 7.3 Appendix C

### 7.3.1 Modified MVA using iterative approximation

---

**Algorithm 1** Modified MVA using iterative approximation

---

**[S1] Initialization**

Initialize  $R_{jk}(\vec{N}) \forall j = 1..C; k = 1..K$  - the residence response time of task class  $i$  in the center  $k$ .

Set the tolerance constant  $\epsilon$ .

**[S2] Construct Precedence Tree**

**[S3] Estimate the intra  $\alpha_{ij}$  and inter  $\beta_{ij}$  overlap factors using the Algorithm3 “Estimation of Overlap Factors”.**

**[S4] Estimate Task Response Time using the Algorithm2 “Response Time Estimation”**

**[S5] Convergence Test**

**if**  $|R_i^{curr} - R_i^{prev}| \leq \epsilon, \forall i$  **then**

    Calculate the Performance Metrics of the System

    Stop

**else**

    Set  $R_i^{prev} = R_i^{curr} \forall i$

    Go to step [S2]

**end if**

---

### 7.3.2 Response Time Estimation

---

**Algorithm 2** Response Time Estimation

---

/\* Estimates the mean response time of each task class, assuming that the overlap factors  $\alpha_{ij}, \beta_{ij}$  are given,  $\forall i, j$  \*/

**[S1] Estimate the Average Response Time of class  $j$  task in center  $k$  when the task population is given by  $\vec{N} - \vec{1}_i$**

Initialize  $R_{jk}(\vec{N}) \forall j = 1..C; k = 1..K$  - the residence response time of task class  $i$  in the center  $k$ .

$$R_{jk}(\vec{N} - \vec{1}_i) \approx \begin{cases} R_{jk}(\vec{N}) - (\frac{1}{N}\alpha_{ji} + \frac{N-1}{N}\beta_{ji}) \cdot \frac{S_{jk} \cdot R_{ik}(\vec{N})}{\sum_{k=1}^K R_{i,k}(\vec{N})}, & \text{if } j \neq i; \\ R_{jk}(\vec{N}) - \beta_{ji} \cdot \frac{S_{jk} \cdot R_{ik}(\vec{N})}{\sum_{k=1}^K R_{i,k}(\vec{N})}, & \text{if } i = j; \end{cases}$$

**[S2] Estimate the Mean Queue Length at each queueing center**

$$Q_{jk}(\vec{N} - \vec{1}_i) \approx \begin{cases} \frac{N_j \times R_{jk}(\vec{N} - \vec{1}_i)}{\sum_{k=1}^K R_{j,k}(\vec{N} - \vec{1}_i)}, & \text{if } i \neq j; \\ \frac{(N_j - 1) \times R_{jk}(\vec{N} - \vec{1}_i)}{\sum_{k=1}^K R_{j,k}(\vec{N} - \vec{1}_i)}, & \text{if } i = j; \end{cases}$$

**[S3] Estimate the Average Queue Length as seen by arriving task  $i$**

$$A_{ik}(\vec{N}) = \frac{1}{N} \sum_{j=1, j \neq i}^C \alpha_{ij} Q_{jk}(\vec{N} - \vec{1}_i) + \frac{N-1}{N} \sum_{j=1, j \neq i}^C \beta_{ij} Q_{jk}(\vec{N} - \vec{1}_i)$$

**[S4] Estimate the Mean Response Time at each center**

$$R_{ik}(\vec{N}) = S_{ik}(1 + A_{ik}(\vec{N}))$$

**[S5] Estimate the Total Response Time**

$$R_i(\vec{N}) = \sum_{k=1}^K R_{ik}(\vec{N})$$


---

### 7.3.3 Estimation of Overlap Factors

---

**Algorithm 3** Estimation of Overlap Factors

---

*/\* Input:  $R_i$  for  $\forall i$ ; Output:  $\alpha_{ij}, \beta_{ij}$  for  $\forall i, j$  \*/*

**[S1] For each internal node  $J_i$  of the composition tree from bottom to the top**

**[S1.1] Compute the mean and variance of the response time of a subjob  $J_i, \mathfrak{R}_i$ , combined from the left and right subtrees by Equations 21-39 from [12]**

**[S1.2] Compute the intra overlap time  $L_X(T_i, T_j)$  where  $T_i$  and  $T_j$  are the tasks belong to the left and the right subtree respectively.**

Given  $J = J_1 \diamond J_2, \diamond \in \{+, \vee, \wedge, \setminus_f\}$

We wish to find out  $L_x(T_i, T_j), \forall T_i \in J_1 \text{ and } T_j \in J_2$

If  $\diamond = +$  or  $\setminus_f, L_x(T_i, T_j) = 0$

We need to consider  $J = J_1 \wedge J_2$  or  $J = J_1 \vee J_2$ ,

where  $J_1 = J_{11} + J_{12} + \dots + J_{1n}, J_2 = J_{21} + J_{22} + \dots + J_{2m}$ ,

$J_i, j$  are of types  $P_A(\text{and}), P_O(\text{or})$  or  $P_f(\text{probabilistic fork})$

$L_x(T_i, T_j) \approx P_r(T_i|J_{1i}) \cdot P_r(T_j|J_{2j}) \cdot L_x(J_{1i}, J_{2j}) = \frac{R_i}{\mathfrak{R}_{1i}} \cdot \frac{R_j}{\mathfrak{R}_{2j}} \cdot L_x(J_{1i}, J_{2j})$

**[S1.3] Compute  $\alpha_{ij}$  for task  $T_i$  and  $T_j$  from  $\mathfrak{R}_i$  and  $L_X(T_i, T_j)$  by equation:**

$$\alpha_{ij} = \frac{L_x(T_i, T_j)}{R_i(\bar{N})}$$

**[S2] Compute the inter overlap time  $L_I T_i, T_j$**

$L_I(T_i, T_j) = P(T_j|J_0) \cdot L_I(T_i, J_0)$ ,

where  $P(T_j|J_0)$  - is a conditional probability that  $T_i$  "sees"  $T_j$  in the system while  $J_0$  is in the system,  $T_j$  belonged to  $J_0$ ;  $L_I(T_i, J_0)$  - is the interjob overlap time between  $T_i$  and  $J_0$

$$P(T_j|J_0) \approx \frac{R_j}{\mathfrak{R}_0}$$

$$L_I(T_i, J_0) = R_i,$$

where  $\mathfrak{R}_0$  is the mean response time distribution of  $J_i, T_i$  arrives at any point in the response time of  $J_0$  with equal probability.

Finally, we obtain:  $L_I(T_i, T_j) = \frac{R_j}{\mathfrak{R}_0} \cdot R_i$

**[S3] Compute the inter overlap factor  $\beta_{ij}$**

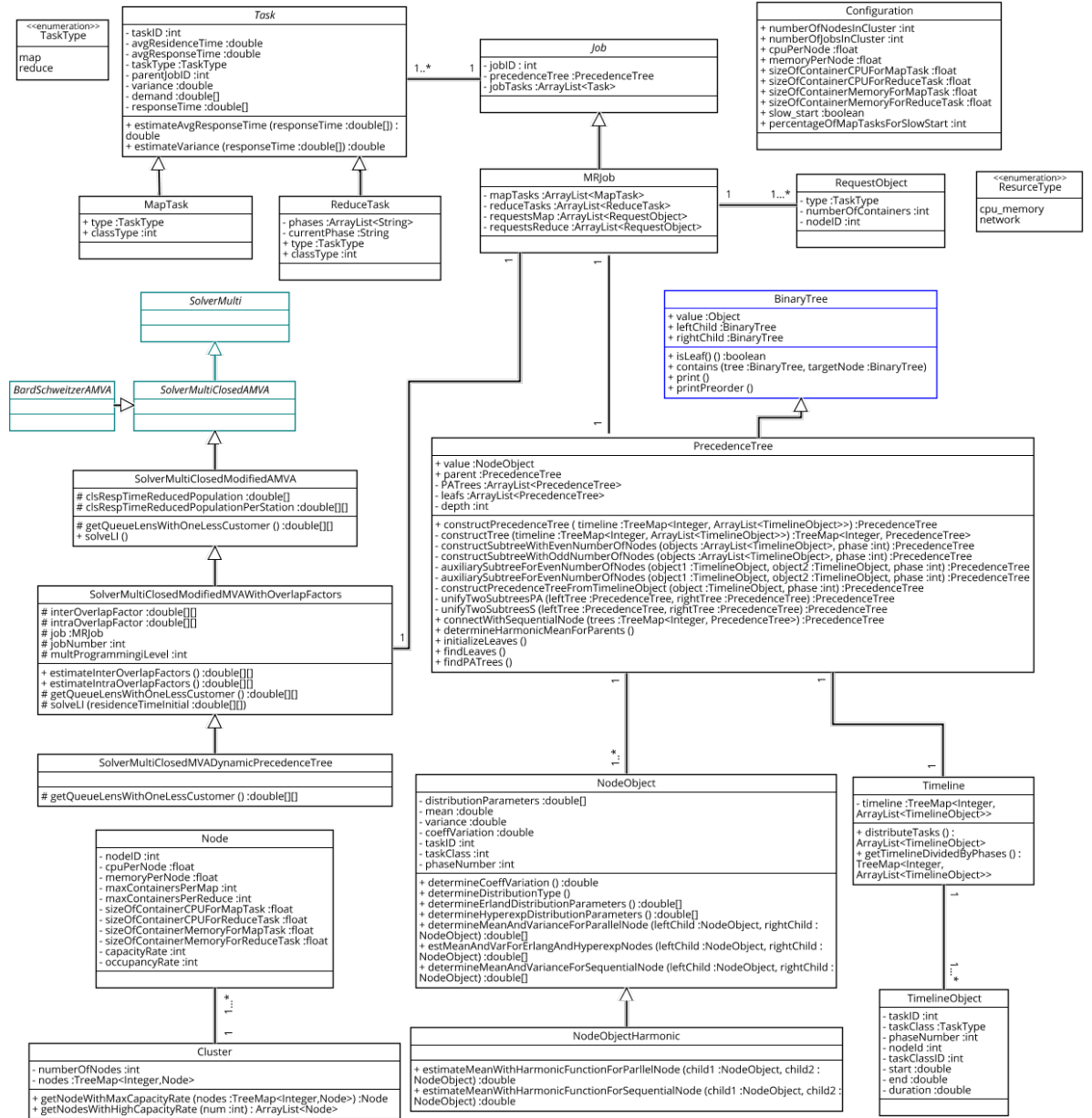
$$\beta_{ij} = \frac{L_I(T_i, T_j)}{R_i(\bar{N})}$$


---

## 7.4 Appendix D

### 7.4.1 UML class diagrams

External libraries, \_\_\_\_\_ Implemented classes



## 7.4.2 Comparison of results of modified AMVA, BardSchweitzerAMVA and exact MVA solution

$N$  - is the number of job that are executed simultaneously.

Table 7.1: Comparison of modified AMVA, BardSchweitzerAMVA and exact MVA solution

$N$	exactMVA (sec)	BardSchweitzerAMVA	modifiedAMVA	Abs.Difference for BardSchweitzer-AMVA	Abs.Difference for modifiedAMVA
1	261	261	261	0	0
2	345.7547892720306	350.0807247235834	344.3687936473084	4.325935451552766	1.3859956247222271
3	437.2496841825314	447.00707268311027	437.2763141446204	9.757388500578884	0.026629962089032233
4	534.8833901198851	550.1792659206093	538.1189989673728	15.295875800724161	3.235608847487697
5	637.9181859050525	657.9215087842349	644.8977437118436	20.003322879182406	6.979557806791149
6	745.496078952993	768.8651176953447	755.7485594451655	23.369038742351677	10.25248049217248
7	856.7116398307605	882.0290742318098	869.2848734602655	25.31743440104924	12.573233629505012
8	970.7005749496686	996.7480913970354	984.5880746396457	26.047516447366775	13.887499689977062
9	1086.7091081136948	1112.5769875539704	1101.0744966297261	25.8678794402756	14.365388516031317
10	1204.1288555865292	1229.215632826835	1218.3740207639073	25.086777240305764	14.245165177378112
11	1322.499684261118	1346.458166533383	1336.2479678559735	23.958482272264973	13.748283594855366
12	1441.4918447986586	1464.1605031030206	1454.5386130020652	22.668658304361998	13.0467682034066
13	1560.8792893925	1582.219631432593	1573.139024495622	21.34034204009299	12.259735103122011
14	1680.512613571095	1700.5603634378738	1691.9748073804262	20.04774986677876	11.462193809331211
15	1800.2959921597972	1819.1268078490707	1810.9929611715727	18.83081568927355	10.696969011775536
16	1920.169471526971	1937.8766616671485	1930.1548942873922	17.707190140177545	9.985422760421216
17	2040.0963073580567	2056.7773965606007	2049.431945064978	16.681089202543944	9.335637706921489
18	2160.0543657131157	2175.803634396504	2168.8024425758704	15.749268683388436	8.748076862754715
19	2280.030506872071	2294.935312244442	2288.2497347980047	14.904805372370902	8.219227925933865
20	2400.0170269795617	2414.1563772783884	2407.7608381217665	14.139350298826685	7.743811142204777
21	2520.009457339568	2533.453843633635	2527.325494492912	13.444386294067044	7.316037153344041
22	2640.0052298075907	2652.817100652394	2646.9355013908303	12.811870844803252	6.930271583239573
23	2760.0028804248714	2772.237398484391	2766.58422785947	12.234518059519814	6.581347434598683
24	2880.001580626117	2891.7074711469236	2886.2662596339974	11.705890520806406	6.264679007880204
25	3000.000864437056	3011.221199692406	3005.9771352977345	11.220335255350165	5.976270860678596
26	3120.000471285417	3130.7734464932946	3125.7131475945394	10.772975207877607	5.712676309122344
27	3240.000256202269	3250.359837332084	3245.471192027678	10.35958112981507	5.470935825409015
28	3360.0001389060662	3369.9766323370714	3365.2486502215174	9.97649343100511	5.248511315451196
29	3480.0000751242533	3489.6206156458825	3485.043299146979	9.62054052162921	5.043224022725553
30	3600.00004053527	3609.2890072037226	3604.8532398054226	9.288966668452758	4.8531992701528
31	3720.0000218245855	3728.9793917096817	3724.6768407049244	8.979369885096276	4.676818880338942
32	3840.000011726781	3848.6896609619894	3844.5126926914513	8.689649235208435	4.5126809646703805
33	3960.0000062890604	3968.4179667572125	3964.359572575815	8.41796046815216	4.359566286754671
34	4080.0000033667948	4088.1626821654922	4084.2164136322367	8.16267879869747	4.216410265441937

35	4200.000001799353	4207.922369500777	4204.0822815083975	7.9223677014233544	4.082279709044087
36	4320.000000960122	4327.695753678406	4323.956354429184	7.695752718283984	3.9563534690623783
37	4440.000000511547	4447.481699935323	4443.837918099399	7.481699423776263	3.8379175878517344
38	4560.000000272163	4567.279195104331	4563.726307136005	7.279194832168287	3.726306863842183
39	4680.000000144606	4687.087331800097	4683.620960978221	7.087331655490743	3.620960833614845
40	4800.000000076734	4806.905295003596	4803.521369589227	6.905294926861643	3.521369512493038
41	4920.000000040669	4926.732350632195	4923.427076554636	6.732350591526483	3.427076513967222
42	5040.00000002153	5046.567835761682	5043.337672247662	6.567835740152077	3.3376722261327814
43	5160.000000011386	5166.411150228908	5163.252788009515	6.411150217521936	3.2527879981289516
44	5280.000000006014	5286.261749393421	5283.1720911739385	6.26174938740769	3.1720911679249184
45	5400.000000003173	5406.119137876043	5403.095280796971	6.119137872869942	3.0952807937974285
46	5520.0000000016735	5525.982864124312	5523.022083978352	5.982864122638603	3.022083976678914
47	5640.000000000088	5645.852515680363	5642.952252681459	5.852515679482167	2.9522526805785674
48	5760.000000000464	5765.727715047817	5762.88556097489	5.727715047352831	2.885560974425971
49	5880.000000000244	5885.608116071158	5882.821802632136	5.608116070914548	2.821802631891842
50	6000.000000000128	6005.493400755124	6002.760789036386	5.493400754995491	2.7607890362578473
51	6120.0000000000655	6125.383276463077	6122.702347346272	5.383276463011498	2.702347346206807
52	6240.000000000036	6245.277473442756	6242.646318885557	5.2774734427193835	2.646318885520486
53	6360.000000000019	6365.175742635733	6362.592557725535	5.175742635713505	2.5925577255156895
54	6480.00000000001	6485.077853733374	6482.54092943389	5.077853733363554	2.5409294338796826
55	6600.0000000000055	6604.98360380209	6602.491309967656	4.983603802084872	2.4913099676505226
56	6720.000000000003	6724.892774470602	6722.443584691371	4.892774470598852	2.4435846913684145
57	6840.000000000002	6844.805192238884	6842.397647504212	4.805192238882228	2.397647504209999
58	6960.0	6964.720686268202	6962.353400062311	4.720686268202371	2.353400062311266
59	7080.000000000001	7084.639097482529	7082.310751084406	4.639097482528086	2.3107510844047283
60	7199.999999999999	7204.560277574555	7202.269615730545	4.560277574555585	2.2696157305463203
61	7320.0	7324.484088110687	7322.229915045146	4.484088110686571	2.2299150451462992
62	7440.0	7444.410399723747	7442.191575456692	4.410399723747105	2.1915754566916803
63	7560.0	7564.339091383526	7562.154528327538	4.339091383525556	2.1545283275381735
64	7680.0	7684.270049736563	7682.1187095480345	4.270049736563124	2.1187095480345306
65	7800.0	7804.203168507625	7802.084059169999	4.203168507625378	2.084059169998909
66	7920.0	7924.1383479562655	7922.050521075149	4.138347956265534	2.050521075148936
67	8039.999999999999	8044.075494382647	8042.01804267468	4.075494382647776	2.0180426746810554
68	8160.000000000001	8164.014519677494	8161.986574636627	4.014519677492899	1.9865746366258463
69	8280.000000000002	8283.955340911665	8281.956070638076	3.95534091166337	1.9560706380743795
70	8400.0	8403.89787996131	8401.926487139617	3.8978799613105366	1.9264871396171657
71	8520.0	8523.842063165093	8521.897783179738	3.84206316509335	1.897783179738326
72	8640.0	8643.78782101033	8641.869920187175	3.7878210103299352	1.869920187175012
73	8760.0	8763.735087845187	8761.842861809348	3.7350878451870813	1.8428618093475961
74	8880.0	8883.683801614565	8881.816573755388	3.6838016145648	1.8165737553881627
75	9000.0	9003.633903617263	9001.791023652257	3.6339036172630585	1.7910236522566265
76	9120.0	9123.585338282606	9121.766180912728	3.5853382826062443	1.7661809127275774
77	9240.0	9243.53805296467	9241.742016614124	3.538052964669987	1.7420166141237132
78	9360.0	9363.491997752477	9361.718503386783	3.491997752476891	1.718503386782686
79	9480.0	9483.44712529481	9481.69561531136	3.4471252948096662	1.6956153113605978



80	9600.0	9603.403390638301	9601.673327824159	3.4033906383010617	1.6733278241590597
81	9720.0	9723.360751077644	9721.65161762977	3.3607510776437266	1.651617629770044
82	9840.0	9843.319166016894	9841.630462620375	3.319166016894087	1.630462620374601
83	9960.0	9963.278596840906	9961.609841801132	3.278596840906175	1.6098418011315516
84	10080.0	10083.239006796046	10081.589735221103	3.239006796045942	1.589735221103183
85	10200.0	10203.200360879428	10201.570123909238	3.200360879427535	1.5701239092377364
86	10320.0	10323.162625735958	10321.550989815045	3.1626257359584997	1.5509898150448862
87	10440.0	10443.125769562552	10441.532315753495	3.1257695625517954	1.5323157534949132
88	10560.0	10563.089762018955	10561.514085353801	3.089762018955298	1.5140853538014198
89	10680.0	10683.054574144639	10681.496283011855	3.0545741446385364	1.4962830118547572
90	10800.0	10803.020178281298	10801.4788938459	3.0201782812982856	1.4788938459005294
91	10920.0	10922.986548000508	10921.461903655254	2.9865480005082645	1.461903655253991
92	11040.0	11042.95365803614	11041.445298881847	2.953658036140041	1.445298881846611
93	11160.0	11162.921484221219	11161.42906657431	2.921484221218634	1.4290665743101272
94	11280.0	11282.890003428789	11281.413194354456	2.8900034287889866	1.4131943544562091
95	11400.0	11402.85919351662	11401.39767038595	2.8591935166205076	1.397670385949823
96	11520.0	11522.82903327537	11521.382483345016	2.829033275369511	1.382483345016226
97	11640.0	11642.799502379956	11641.36762239301	2.799502379955811	1.3676223930106062
98	11760.0	11762.770581344023	11761.353077150758	2.770581344022503	1.3530771507575992
99	11880.0	11882.742251477097	11881.33883767447	2.742251477096943	1.338837674469687
100	12000.0	12002.714494844395	12001.324894433184	2.7144948443947214	1.324894433184454

As we can see, modified AMVA algorithm provides more accurate results than BardSchweitzer AMVA comparing with exact MVA algorithm.

### 7.4.3 Evaluation results

Blue - real setup; red - Fork/Join based approach, green - Tripathi based approach;

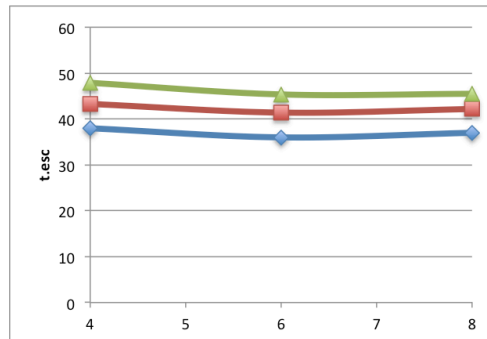
$e_1$  - error of Fork/Join based approach;

$e_2$  - error of Tripathi based approach

**WordCount: 0.5GB**

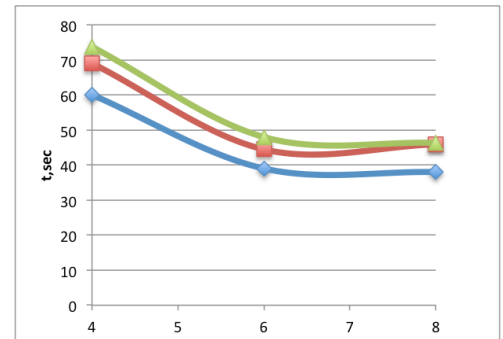
**Number of nodes/response time(sec); fixed number of jobs**

**Number of jobs: 1**



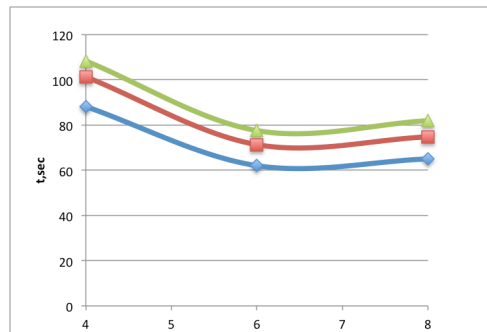
Number of nodes	e1	e2
4	0.14	0.26
6	0.15	0.26
8	0.14	0.23

**Number of jobs: 2**



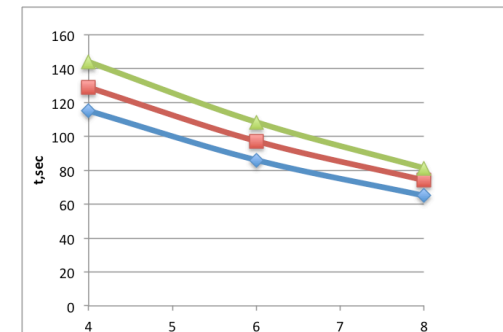
Number of nodes	e1	e2
4	0.2	0.2
6	0.1	0.2
8	0.2	0.2

**Number of jobs: 3**



Number of nodes	e1	e2
4	0.15	0.23
6	0.15	0.25
8	0.15	0.26

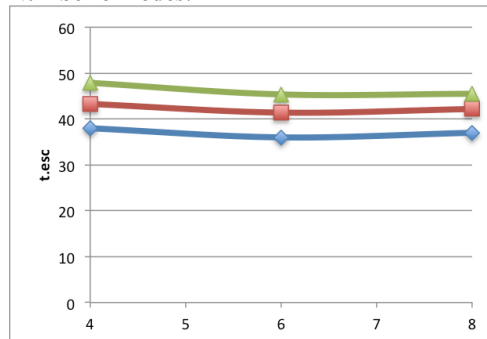
**Number of jobs: 4**



Number of nodes	e1	e2
4	0.12	0.25
6	0.13	0.26
8	0.14	0.25

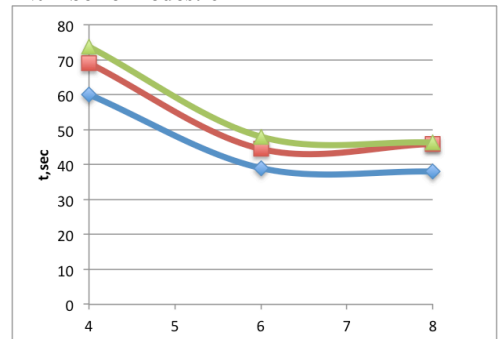
Number of jobs/response time(sec); fixed number of nodes

Number of nodes: 4



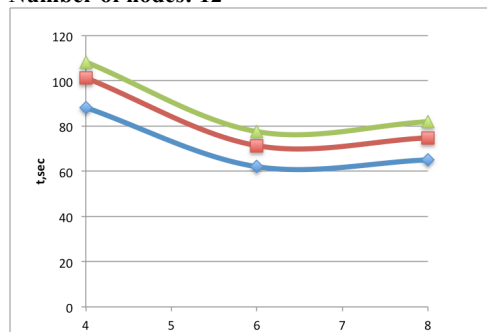
Number of jobs	e1	e2
1	0.13	0.26
2	0.15	0.23
3	0.15	0.23
4	0.12	0.25

Number of nodes: 6



Number of jobs	e1	e2
1	0.15	0.26
2	0.14	0.23
3	0.15	0.25
4	0.13	0.26

Number of nodes: 12

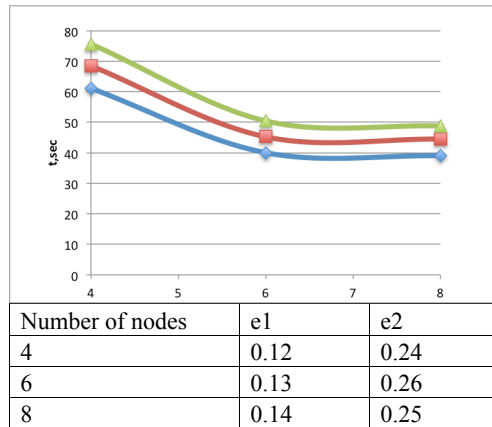


Number of jobs	e1	e2
1	0.14	0.23
2	0.21	0.22
3	0.15	0.26
4	0.14	0.25

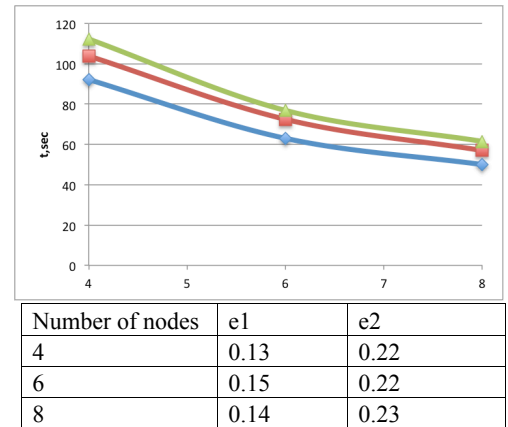
**WordCount: 1GB**

**Number of nodes/response time(sec); fixed number of jobs**

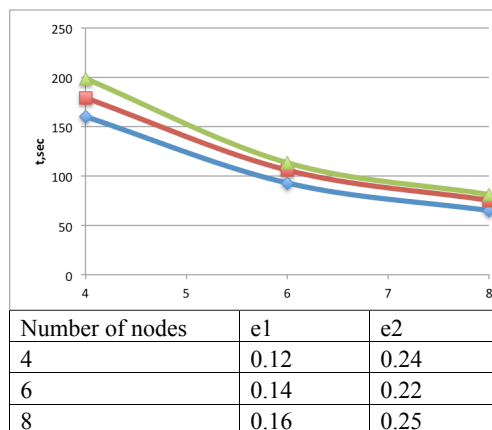
**Number of jobs: 1**



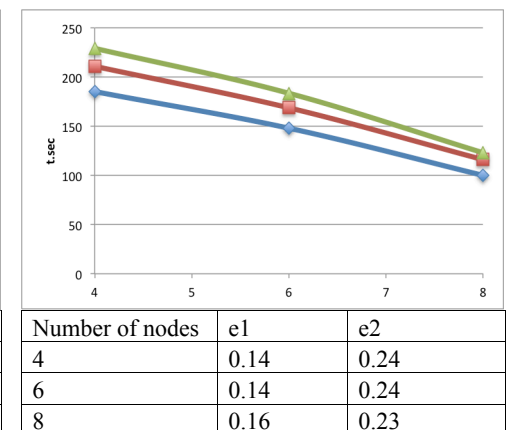
**Number of jobs: 2**



**Number of jobs: 3**

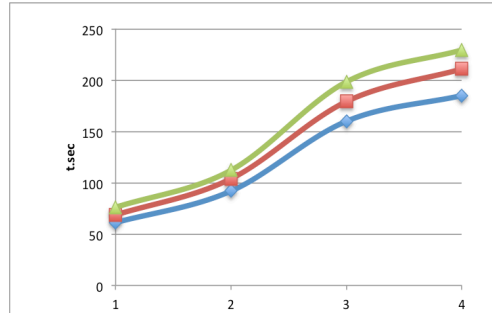


**Number of jobs: 4**



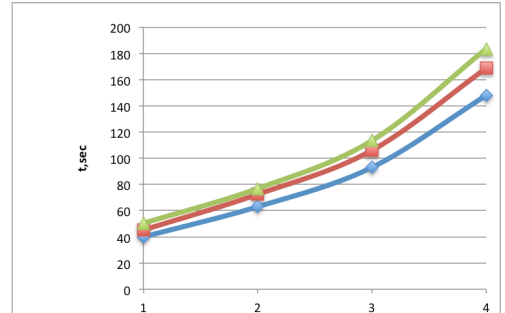
### Number of jobs/response time(sec); fixed number of nodes

Number of nodes: 4



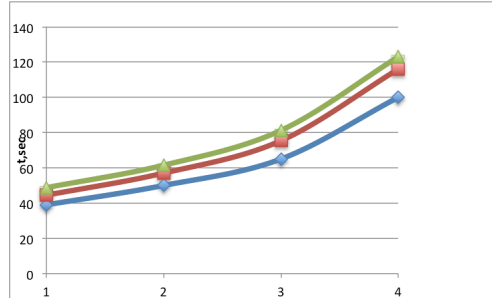
Number of jobs	e1	e2
1	0.12	0.25
2	0.13	0.22
3	0.12	0.24
4	0.14	0.24

Number of nodes: 6



Number of jobs	e1	e2
1	0.13	0.26
2	0.15	0.22
3	0.14	0.22
4	0.14	0.24

Number of nodes: 12

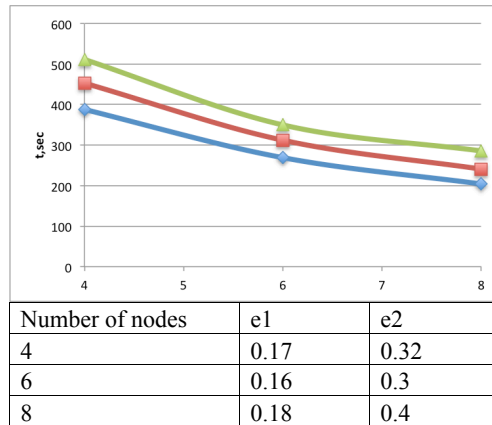


Number of jobs	e1	e2
1	0.14	0.25
2	0.14	0.23
3	0.16	0.25
4	0.16	0.23

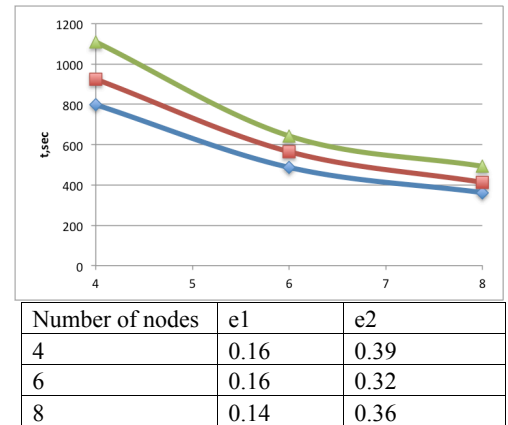
**WordCount: 10GB**

**Number of nodes/response time(sec); fixed number of jobs**

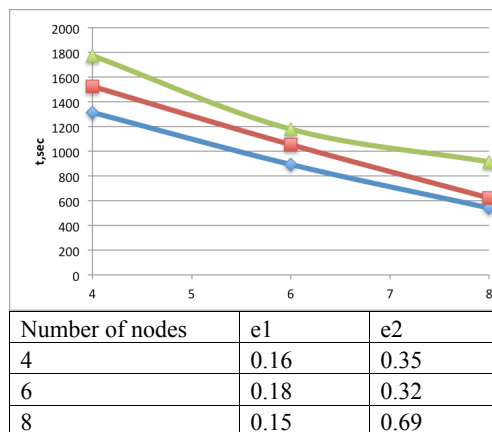
**Number of jobs: 1**



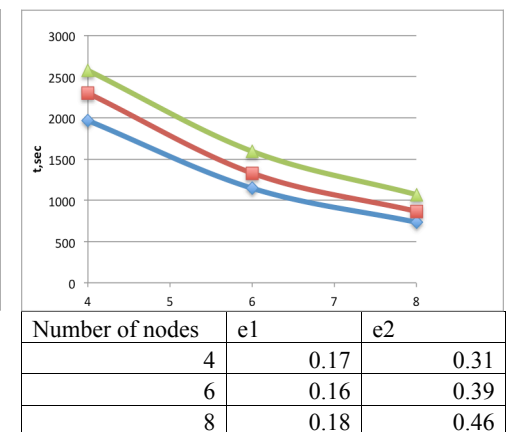
**Number of jobs: 2**



**Number of jobs: 3**

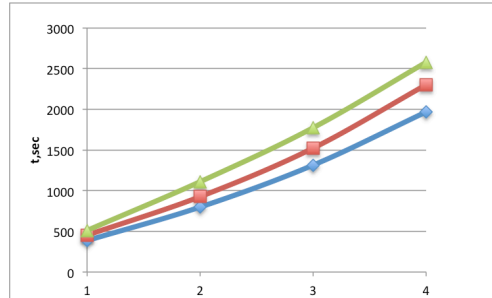


**Number of jobs: 4**



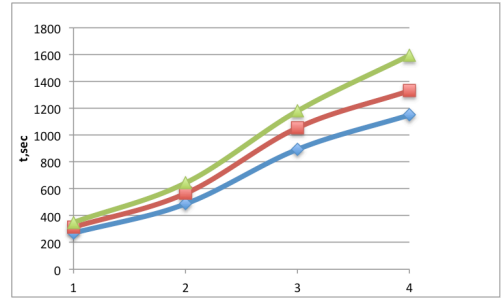
### Number of jobs/response time(sec); fixed number of nodes

**Number of nodes: 4**



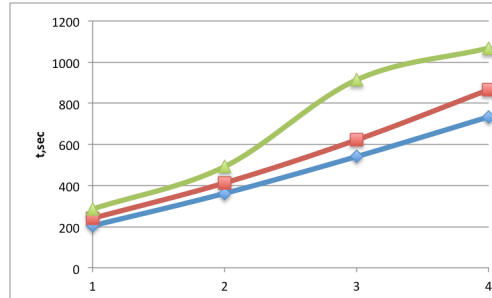
Number of jobs	e1	e2
1	0.17	0.32
2	0.16	0.39
3	0.16	0.35
4	0.17	0.31

**Number of nodes: 6**



Number of jobs	e1	e2
1	0.16	0.30
2	0.16	0.32
3	0.18	0.32
4	0.16	0.39

**Number of nodes: 12**



Number of jobs	e1	e2
1	0.18	0.28
2	0.14	0.36
3	0.15	0.69
4	0.18	0.46

## List of Figures

2.1	Comparison of architecture of Hadoop 1.x and Hadoop 2.x . . . . .	3
2.2	Job execution process in YARN . . . . .	5
4.1	The main steps of Modified MVA algorithm . . . . .	14
4.2	Lifecycle of map task . . . . .	16
4.3	Lifecycle of reduce task . . . . .	16
4.4	Intra- and inter- job overlap factors . . . . .	18
4.5	The main steps for task response time estimation . . . . .	19
4.6	The high level diagram of implementation solution . . . . .	21
7.1	Example of timeline construction . . . . .	28
7.2	Example of Precedence tree construction . . . . .	29
7.3	Dependence between $\epsilon$ and Job response time . . . . .	30

## List of Tables

4.1	Input parameters for Performance Cost Model . . . . .	14
7.1	Comparison of modified AMVA, BardSchweitzerAMVA and exact MVA solution	35



# References

- [1] Dean, J. and Ghemawat, S. “*MapReduce: Simplified data processing on large clusters*”, Communications of the ACM 51.1 (2008): 107-113.
- [2] Vavilapalli V. K. et al. “*Apache hadoop yarn: Yet another resource negotiator*”, Proceedings of the 4th annual Symposium on Cloud Computing. – ACM, 2013. – C. 5.
- [3] Herodotou, H. “*Hadoop Performance Models*”, Technical Report, CS-2011-05 Computer Science Department Duke University, p. 19.
- [4] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, Shivrath Babu “*Starfish: A Selftuning System for Big Data Analytics*”, CIDR. Vol. 11. 2011.
- [5] Shvachko, Konstantin, et al. “*The hadoop distributed file system.*”, Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on. IEEE, 2010.
- [6] Starfish <https://www.cs.duke.edu/starfish/index.html>
- [7] Apache Hadoop <http://hadoop.apache.org>
- [8] Job Profiler and Job Analyzer Starfish code <https://github.com/wangyu0air/Starfish-for-hadoop-2.x>
- [9] Grandl, Robert, et al “*Multi-resource packing for cluster schedulers*”, ACM SIGCOMM Computer Communication Review. Vol. 44. No. 4. ACM, 2014.
- [10] Verma, Abhishek, Ludmila Cherkasova, and Roy H. Campbell “*ARIA: automatic resource inference and allocation for mapreduce environments*”, Proceedings of the 8th ACM international conference on Autonomic computing. ACM, 2011.
- [11] Reiser, M. and Lavenberg, S. “*Mean-Value Analysis of Closed Multichain Queuing Networks*”, Journal of the ACM (JACM) 27.2 (1980): 313-322.
- [12] Liang, De-Ron, and Satish K. Tripathi. “*On performance prediction of parallel computations with precedent constraints*”, Parallel and Distributed Systems, IEEE Transactions on 11.5 (2000): 491-508.
- [13] Varki, Elizabeth. “*Mean value technique for closed fork-join networks*”, ACM SIGMETRICS Performance Evaluation Review. Vol. 27. No. 1. ACM, 1999.
- [14] Bukh, Per Nikolaj D., and Raj Jain. “*The art of computer systems performance analysis, techniques for experimental design, measurement, simulation and modeling*”, (1992): 113-115.
- [15] Menasce, Daniel A., et al “*Performance by design: computer capacity planning by example*”, Prentice Hall Professional, 2004.
- [16] Kruskal, Clyde P., and Alan Weiss. “*Allocating independent subtasks on parallel processors.*”, Software Engineering, IEEE Transactions on 10 (1985): 1001-1016.
- [17] Thomasian, Alexander, and Paul F. Bay. “*Analytic queueing network models for parallel processing of task systems.*”, SComputers, IEEE Transactions on 100.12 (1986): 1045-1054.
- [18] Woeginger, Gerhard J. “*There is no asymptotic PTAS for two-dimensional vector packing.*”, Information Processing Letters 64.6 (1997): 293-297.
- [19] Vianna, Emanuel, et al. “*Analytical performance models for MapReduce workloads.*”, International Journal of Parallel Programming 41.4 (2013): 495-525.
- [20] Murthy, Arun C., et al. “*Apache Hadoop YARN: Moving Beyond MapReduce and Batch Processing with Apache Hadoop 2.*”, Pearson Education, 2014.
- [21] <http://ercoppa.github.io/HadoopInternals/ApplicationMaster.html>
- [22] Mak, Victor W., and Stephen F. Lundstrom. “*Predicting performance of parallel computations.*”, IEEE Transactions on Parallel and Distributed Systems 1.3 (1990): 257-270.

- [23] <http://hortonworks.com/blog/apache-hadoop-yarn-resourcemanager/>
- [24] <http://hortonworks.com/blog/apache-hadoop-yarn-nodemanager/>
- [25] <http://jmt.sourceforge.net/>
- [26] <http://cslibrary.stanford.edu/110/BinaryTrees.html>