A Scheduling Algorithm for Hadoop MapReduce Workflows with Budget Constraints in the Heterogeneous Cloud

by

Andrew Wylie

A thesis submitted to the
Faculty of Graduate and Post Doctoral
Affairs in partial fulfillment of
the requirements for the degree of
Master of Computer Science
in

Computer Science
Carleton University
Ottawa, Ontario, Canada
August 21, 2015

Abstract

In recent years cloud services have gained much attention as a result of their availability, scalability, and low cost. One use of these services has been for the execution of scientific workflows as part of Big Data Analytics, which are employed in a diverse range of fields including astronomy, physics, seismology, and bioinformatics. There has been much research on heuristic scheduling algorithms for these workflows due to the problem's inherent complexity, however existing work has mainly considered execution in a utility grid environment using a generic distributed framework. For our research, we consider the ever-increasingly popular Apache Hadoop framework for scheduling workflows onto resources rented from cloud service providers. Contrary to other distributed frameworks, the Hadoop MapReduce model imposes a functional style onto application definition, and as such presents an interesting and unapproached challenge for workflow scheduling. Investigated in our work is budget-constrained workflow scheduling on the Hadoop MapReduce platform, wherein we devise both an optimal and a heuristic approach to minimize workflow makespan while satisfying a given budget constraint.

Acknowledgments

Firstly I would like to thank my parents for their ongoing support and encouragement throughout my academic career. As well, I would like to thank several of my close friends who helped to keep me motivated throughout my master's degree.

I would also like to thank Wei Shi and Jean-Pierre Corriveau for their guidance and advice while working on my thesis. Their experience and accessibility was invaluable throughout all stages of the dissertation.

Table of Contents

Abstract ii

Acknowledgments iii

Table of Contents iv

List of Tables vii

List of Figures viii

List of Acronyms x

1 Introduction 1

1.1 Introduction 1

1.2 Motivation 2

1.3 Contributions 7

1.4 Thesis Organization 8

2 Background and Related Work 10

2.1 Introduction 10

2.2 Workflows as Directed Acyclic Graphs 10

2.3 Distributed Environments 15

2.4 Distributed Frameworks 18

2.4.1 Condor 18
2.4.2 Pegasus 21
2.4.3 Apache Hadoop 23
2.5 Scheduling Algorithms 30
2.5.1 Deadline-Based 32
2.5.2 Deadline-Constrained 35
2.5.3 Deadline & Budget Optimization 38
2.5.4 Budget-Constrained 40

3 Problem Formulation **47**
3.1 Assumptions 47
3.2 Formulation 49
3.2.1 Stage Optimization 50
3.2.2 Workflow Optimization 51

4 Budget-Driven Algorithms **57**
4.1 Optimal Scheduling 58
4.2 Greedy Scheduling 65

5 Implementation **71**
5.1 Introduction 71
5.2 Job Submission Execution Flow 74
5.3 Workflow Submission Execution Flow 75
5.4 Workflow Scheduling Plan 83
5.4.1 Scheduling Plan Interface 85
5.4.2 Optimal Scheduling Plan 86
5.4.3 Greedy Scheduling Plan 89
5.4.4 Progress-Based Scheduling Plan 90

6 Empirical Studies **94**

6.1 Introduction 94

6.2 Test Setup 95

6.2.1 Cluster Configuration 95

6.2.2 Workflow Configuration & Job Definition 96

6.3 Data Collection 99

6.4 Workflow Scheduling Experiments 105

7 Conclusions **109**

7.1 Conclusions 109

7.2 Future Work 111

References **113**

List of Tables

1 A comparison of distributed environments. 17

2 The input and output values of various user-supplied MapReduce functions. 27

3 The definition of the time-price table. 50

4 An overview of the Amazon EC2 machine types used during experimentation. 96

List of Figures

1 A simplified LIGO workflow. 12
2 A simplified Montage workflow. 13
3 A simplified SIPHT workflow. 14
4 An enumeration of workflow substructure types. 15
5 An overview of the Condor architecture. 19
6 The matchmaking process in Condor. 20
7 An example of resource acquisition in Condor. 20
8 Level-based partitioning in Pegasus. 22
9 An example of a simple workflow, in terms of both its jobs and the tasks that the jobs comprise. 25
10 An overview of MapReduce job execution flow. 26
11 An overview of the YARN architecture. 28
12 An overview of the WordCount MapReduce job. 29
13 Workflow partitioning as shown [74]. 36
14 The method of workflow representation as proposed in [71]. 42
15 An example of task pairing using time-price tables during workflow scheduling. 59
16 An example of time-price tables used in greedy workflow scheduling. . 60
17 An example of time-price tables used in workflow scheduling. 61
18 A visualization of utility with respect to task execution times. 66

19 An overview of the Apache Hadoop architecture (version 1.x.) 72
20 Apache Hadoop job & workflow execution flow. 78
21 A sequence diagram displaying the execution flow of workflow scheduling. 82
22 Task execution times resulting from SIPHT execution on the *m3.medium* machine type. 100
23 Task execution times resulting from SIPHT execution on the *m3.large* machine type. 101
24 Task execution times resulting from SIPHT execution on the *m3.xlarge* machine type. 103
25 Task execution times resulting from SIPHT execution on the *m3.2xlarge* machine type. 104
26 The actual and computed execution time for the SIPHT workflow, according to multiple budget amounts. 106
27 The actual and computed cost for the SIPHT workflow, according to multiple budget amounts. 108

List of Acronyms

API Application Programming Interface

AWS Amazon Web Services

CSP Cloud Service Provider

DAG Directed Acyclic Graph

GA Genetic Algorithm

GPU Graphics Processing Unit

HDFS Hadoop Distributed FileSystem

HPC High Performance Computing

IaaS Infrastructure as a Service

LP Linear Programming

OSS Open-source Software

QoS Quality of Service

SLA Service Level Agreement

VPS Virtual Private Server

Chapter 1

Introduction

1.1 Introduction

Since gaining the ability to record written information, mankind has been able to increase its intelligence throughout successive generations. For as a result of these written records, we are not limited to the information gleaned from just our close relatives and social groups. Indeed, we can look for assistance from our ancestors in written works and from others in distant locations through written correspondence. Additionally, for centuries there has been a vast wealth of information made publicly available, which is also continually increasing. Since the advent of the Internet and the adoption of computers for common household use, we have seen a tremendous increase in our ability to both generate and store information. However, these abilities also bring with them complications. For as the constraints on data storage have been all but lifted due to repeated advancements in computer technology, it seems that the limitation placed on our ability to learn and better understand our environment has become comprehension. As a solution we have seen computers used: first as a method to perform simple arithmetic for data analysis, and then later for general data processing. More recently computers are again being utilized in an attempt to provide meaning, though datasets have become much larger within the past decade.

Consequently, groups of computers - called *clusters* - are now being organized to execute together in order to process these exponentially larger datasets.

With the move to clusters for data processing seen in recent years, new methods which take advantage of this infrastructure are favourable as they help to ease the burden of data processing. Formation of these methods is mainly accomplished by reviewing current techniques used to process large datasets, as well as considering any emerging trends in distributed processing. Armed with the knowledge from these studies, researchers can then respond by developing novel methods to optimize data processing operations.

1.2 Motivation

In recent years, a great deal of data processing has moved to distributed processing platforms. This migration has been caused by several factors. One such factor is dataset size, which has been constantly growing as companies are able to gather more data relevant to their interests. One example would be user engagement and usage patterns gathered by a game studio wanting to keep its subscribers. Other situations are not as novel, such as the tasks of website indexing or log processing. However, these tasks are still necessary, and therefore require innovation when older methods are unable to scale with dataset size. Another factor in the movement to distributed processing has been the low cost of commodity hardware brought on by the ubiquity of personal computers. Lastly, the emergence and adoption of distributed computing frameworks allow lower entrance barriers to distributed execution of programs.

Many distributed computing frameworks have been developed in the last decade, though few have reached widespread adoption. The frameworks themselves generally allow the management and administration of executable programs that process data, often known as *applications* or *jobs*. Several frameworks also allow the execution

of sequences of jobs known as *workflows*. Some commonly used frameworks include systems such as Apache Hadoop, CloudBATCH, DAGMAN/Condor/Pegasus, and VGE [6, 15, 17, 23, 78]. Of these, the most popular is Apache Hadoop, as it has seen deployment onto clusters owned by companies such as Amazon, Google, Microsoft, and Yahoo! [3, 4, 8, 9]. Many complex data processing jobs have recently been implemented on distributed computing frameworks. For instance, Apache Hadoop is used for tasks such as social network mining, log processing, video analysis, image analysis, search indexing, recommendation systems, web indexing, and large-scale scientific applications [59, 76]. As a result of the popularity of the Apache Hadoop framework, many scheduling algorithms have been proposed to optimize different aspects of job execution. The first of these algorithms are the Capacity and Fair schedulers developed by Yahoo! and Facebook, respectively [54]. These were then followed by research efforts dealing with issues such as data locality and node heterogeneity [44, 59, 68, 76]. In addition to its popularity, many schedulers have also been developed for Apache Hadoop due to the fact that they are 'pluggable'. As such, the scheduling method used by the framework is easily selected through a configuration file.

Regardless of the framework, efficient scheduling is an important requirement. Schedulers themselves vary according to many properties. For instance, they can work to schedule individual jobs, sets of individual jobs (Batch scheduling), or sets of jobs connected by dependencies. Additionally, schedulers can take into consideration constraints specified by the job's executor. These vary, though typically pertain to deadline or budget constraints. Deadline constraints specified to a scheduler instruct it to attempt to complete job execution within the specified time constraint, whereas budget constraints instruct it to complete job execution while satisfying a monetary constraint. Budget constraints are relatively new, and have been adopted as more users of distributed computing frameworks have decided to rent resources as opposed to purchasing them.

The combination of cluster ownership cost with data availability leaves some prospective users of distributed frameworks in a dubious position. Consider the choices one would have to make if requiring the use of a distributed framework. Shared resources for distributed computation (grids) do exist, however their performance is not guaranteed. Assuming the need for performance guarantees, one would have to purchase the hardware, configure, and maintain it. One would then also need to learn how to deploy, manage, and administrate the framework. Obviously, the costs of such a project would outweigh the benefits for most users. As another use case, many companies involved in DNA sequencing and protein folding require computing resources for only several weeks at a time, after which the resources are not needed [39]. In either case, the additional resources are only required for a short time, after which the user would not want to maintain ownership due to a combination of high ownership costs and low use. Luckily, distributed frameworks such as Apache Hadoop have recently become available through providers such as Amazon [1, 3]. Amazon's offerings include compute resources (EC2), as well as preconfigured Hadoop clusters (EMR). These products, often denoted as Infrastructure as a Service (IaaS), allow users to rent a number of different resources for a specific time period, each priced proportionally to their processing power [47]. Thus, it is the requirement for temporary resources that has driven the emergence of these IaaS providers, which has in turn created a need for budget-constrained scheduling.

The arrival of IaaS products again decreases entrance barriers to distributed computation, just as the introduction of distributed frameworks has. Additionally, selection of a particular distributed framework - Apache Hadoop - as the de facto platform for distributed computation has allowed the average consumer to process large datasets effortlessly. We believe that due to the widespread adoption of these products the requirement for efficient budget-constrained scheduling will only grow. Thus, this dissertation focuses on the creation of a scheduler for the Apache Hadoop

framework that allows for the specification and use of budget constraints. As many types of resources are available from IaaS providers, our scheduler is also written to handle execution on a set of heterogeneous resources.

We denote a cluster of resources rented from an IaaS provider as a *cloud*. Shared resources provided by a community are also available for distributed computation. These clusters are generally referred to as *grid*s, and in their general form are offered at no fee and with no Quality of Service (QoS) guarantees. This is contrary to clouds, which do provide QoS guarantees, as well as private access to the resources [25]. However, grids can also be provided with QoS guarantees for a fee. These types of grids are called *utility grids*, and are often used by scientific communities for execution of jobs relating to the fields of high-energy physics, gravitational-wave physics, geophysics, astronomy, and bioinformatics [34]. Similar to execution on IaaS clouds, these scientific experiments require optimization decisions that consider both execution time and cost [34]. As a result of this and other benefits, large-scale scientific applications have moved to execution on cloud resources, and Apache Hadoop in particular [19,76]. Efficient budget-constrained scheduling is not the only requirement for these applications, for many of the scientific applications are workflows: groups of jobs connected through dependency constraints. For example, the scientific workflow LIGO (Figure 1) consists of several discrete stages of computation realized by multiple executions of different jobs. In this workflow, several terabytes of input data are initially processed to perform consistency checks, and then passed on to detect the coalescence of binary star systems. In this case, the jobs inspecting for coalescence of star systems have as dependency constraints the jobs performing data consistency checks.

As workflow scheduling is not implemented in the Apache Hadoop framework, several workflow engines exist to provide this functionality; the three main ones being Oozie, Azkaban, and Luigi [18]. These schedulers do have several shortcomings

however, especially when considering the need for budget-constrained scheduling seen in recent years. Most importantly, none of these schedulers allow for constraints to be defined, causing them to be unrelated to the main goal of our work. Secondly, the workflow engines all handle the executed workflow themselves, while passing individual jobs to Hadoop for execution. As a result, any possible optimizations available through scheduling the jobs as a single unit are lost. Furthermore, workflow engines do not determine the method of scheduling used by Hadoop. As such, the scheduler used by Hadoop could unknowingly decrease the efficiency of workflow execution.

Along with the lack of feature-rich workflow engines for Hadoop, no budget-constrained workflow schedulers for Hadoop have been proposed in the literature. This is perhaps a result of the complexity of workflow scheduling, as optimal scheduling is an NP-complete problem, and is additionally non-approximable [19, 21, 37, 47, 52, 62, 70]. However, considering that many scientific applications require workflow scheduling which considers both execution time and cost, a concrete implementation of budget-constrained scheduling would prove extremely useful [26]. Such an implementation would also be practical for both users of the distributed framework, and for IaaS providers. For instance, users of cloud services would be given peace of mind through assurance of a particular cost for the work they require. Along with this, they would also be able to maintain control over the total cost for workflow execution. Providers of IaaS systems would also benefit through more efficient resource use, as they would be able to serve more customers, and thus outcompete competitors through superior economies of scale. It is for these reasons that we propose in this work both modifications to the Apache Hadoop framework to allow integrated generic workflow scheduling, and introduce a novel budget-constrained workflow scheduling algorithm for the Apache Hadoop framework in an IaaS cloud environment.

1.3 Contributions

As mentioned in Section 1.2, the use of distributed processing and its related frameworks for data processing has grown quickly over the last decade. Of the available distributed frameworks, Apache Hadoop stands out as the most mature, well-supported, and often used framework. Applications requiring distributed processing for efficient computation have also changed recently, moving from single jobs to workflows of jobs interconnected with dependency constraints. As well, the user base desiring execution of these applications lack the resources to support their distributed processing operations, and have thus turned to IaaS providers for access to computational resources. These demands all motivate implementation of workflow scheduling in Hadoop, a feature that until now has been ill-supported at best.

To remedy these issues, we have implemented modifications to the Apache Hadoop framework to allow fully integrated workflow scheduling. The modifications span the addition of 40 classes and around 8.5 thousand lines of code to the project, which as of this year contains over 1.8 million lines of code. The addition of workflow scheduling makes no assumptions about scheduler functionality, and similar to job scheduling allows for pluggable schedulers defined via framework configuration properties. As a result, workflow schedulers implemented for the Hadoop framework have full control over scheduling of the workflow and its composite jobs, and if required can consider scheduling of multiple workflows concurrently. In addition to the ability of workflow scheduling, we also provide several example workflows and jobs that use the added features.

Furthermore we have designed, completed a theoretical analysis, and provided an implementation for two different budget-constrained workflow schedulers. A third

deadline-constrained workflow scheduler has also been implemented based on the related work in [45]. The budget-constrained schedulers include an 'optimal'[1] scheduler and a greedy scheduler. The deadline-constrained scheduler is a progress-based scheduler adapted from [45] which is able to prioritize workflow jobs using several different methods. Testing of the greedy budget-constrained scheduler was also accomplished on a production-size cluster on Amazon EC2, the most popular IaaS cloud provider. The testing used execution on two scientific workflows to verify accurate scheduler execution, where one workflow was used for detailed analysis and another to corroborate the results.

As simply implementing the modifications does not allow widespread use of the added features, we have also open-sourced the modifications, and made them available to the general public. Documentation for the changes have also been included throughout the code, with all required steps for workflow execution contained in this dissertation.

Overall, our main contribution in this work are modifications to the Apache Hadoop framework that allow workflow scheduling. These modifications are novel and have led to the completion of the first generic workflow scheduler fully integrated with the Apache Hadoop framework. This integration has been extensively tested via the execution of a greedy scheduler on multiple workflow applications, which together demonstrates the flexibility of our implementation.

1.4 Thesis Organization

This thesis is organized to cover all work done, from problem conception to the proposed solution. We begin in Chapter 2 with a review of all related work, including both related scheduling algorithms and background on several different distributed

[1] As the algorithm offers the best result, we label is as an *optimal* scheduler. However, note that this is not a reference to mathematical optimality.

environments and distributed computation frameworks. We demonstrate the popularity of the Apache Hadoop framework, and provide justification for our decision to implement workflow scheduling in Hadoop. An overview of Apache Hadoop's method of operation is also given, including an explanation of its MapReduce programming model and the effects on our proposed work. Lastly, we review related algorithms that appear in the distributed environments, including deadline-based, deadline-constrained, deadline-optimized, budget-optimized, and budget-constrained algorithms.

Following the related work is the problem formulation in Chapter 3. This chapter contains the assumptions made to allow workflow scheduling and our budget-constrained algorithms. The chapter also contains the main problem formulation, where we introduce the mathematical models used by our proposed algorithms along with their method of execution. Chapter 4 then presents our optimal and greedy budget-constrained algorithms. In addition to the derivation of algorithm pseudocode we compute the running times of both proposed algorithms and present proofs of their correctness where necessary.

Chapter 5 contains an overview of the scheduling algorithm implementations. As well, it also explains changes made to the Apache Hadoop framework to allow generic integrated workflow scheduling. The entire program execution flow is also detailed for both job and workflow submissions, with an emphasis on control flow during scheduling. After explaining our implementation, we then present in Chapter 6 our methods of experimentation. Specifically, we examine the method used for testing of both the modifications to the Apache Hadoop framework, and for the implemented greedy scheduling algorithm.

Lastly, we conclude the thesis in Chapter 7 with a review of our motivations, contributions, and test results.

Chapter 2

Background and Related Work

2.1 Introduction

Budget-constrained scheduling problems arise in several different distributed environment types and situations. As a result, workflow scheduling algorithms not only present themselves in Hadoop-related articles, but also in works for scheduling on both utility grids and the IaaS cloud. In this chapter we first explain workflows in the context of scientific applications, along with introducing their Directed Acyclic Graph (DAG) representation. The DAG representation is then be used in Chapter 3 and 4 as we proceed through our problem formulation and on to presentation of our solution to the budget-constrained workflow scheduling problem. Following the section on workflows, we introduce the distributed environments and frameworks which the related algorithms execute in. The last section of this chapter then reviews the related algorithms themselves.

2.2 Workflows as Directed Acyclic Graphs

One aspect of the surveyed scheduling algorithms is that they consider a set of interdependent jobs, often called a *workflow*. Since the jobs represent the execution

of generic applications, there can be no circular dependencies formed by precedence constraints between jobs. This allows the set of jobs to be represented as a Directed Acyclic Graph (DAG).

For our problem, we first give a simple definition of a DAG, leaving the mathematical model to appear later in Chapter 3. We begin by defining a DAG as a graph containing a set of nodes connected by edges. Each node is connected to at least one other node in the graph, and the connections are directed. The direction allows traversal only in one direction, which is specified as part of the graph definition. DAGs have one more important property, which is acyclicity. This property specifies that by following a path of directed graph edges, one is never able to reach a node that has been previously visited. For our use, the nodes of a DAG represent the jobs appearing in a workflow, with the edges as dependencies placed on execution. For example, given two nodes v and u with a directed connection from v to u, we can say that v is dependent upon u. Alternately, we can also say that the job represented by u must execute before the job represented by v.

Often, nodes in a workflow DAG with no predecessors are called *entry* nodes, whereas nodes without successors are *exit* nodes. In the reviewed literature, many of the models either only consider DAGs with a single *entry* and *exit* node, or they apply a simple transformation to create such a DAG without affecting the schedule [19, 21, 22, 30, 31, 52, 55, 56, 62, 73, 77]. This is done by connecting all *entry* nodes to a new zero-cost pseudo *entry* node, and all *exit* nodes to a new single *exit* node. After applying this transformation the total schedule length (*makespan*) of the DAG can be defined as the maximum actual finish time of the *exit* node.

Workflow DAGs are often used in scientific applications, which cover a diverse range of fields including astronomy, bioinformatics, genetics, and seismology. For instance, the scientific applications CyberShake, LIGO, Montage, and SIPHT are all research projects that can be characterized by workflows [24, 28, 35, 48]. Figures 1, 2,

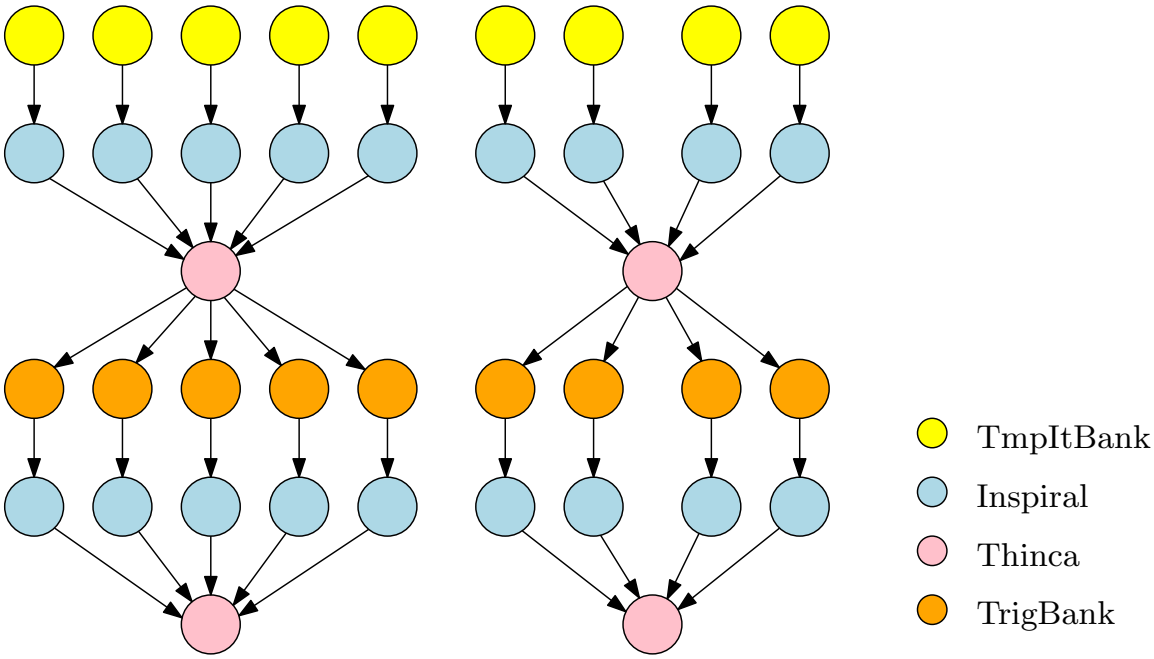


Figure 1: A simplified LIGO workflow. The job type is represented by node colour in the workflow DAG.

and 3 show a simplified characterization of the LIGO, Montage, and SIPHT workflows as reported in [26].

The Laser Interferometer Gravitational wave Observatory (LIGO) is an endeavour to detect gravitational waves produced by events in the universe, such as the collision of large celestial bodies. The project uses three interferometers to collect approximately 1TB of data daily, which is then analyzed through the use of the grid computing software Condor DAGMan [6, 61]. The analysis consists of the execution of a workflow DAG (similar to Figure 1) containing a large number of independent jobs which perform consistency checks, inspiral detection for coalescence of binary systems, and computation of filtered output [26, 28].

Montage is a scientific application created by NASA and the IPAC Infrared Science Archive that allows users to combine images of the sky into custom science-grade mosaics [24,26,36]. To achieve this end, several geometric transformations are applied on the input images to re-project them, correct & normalize background emissions,

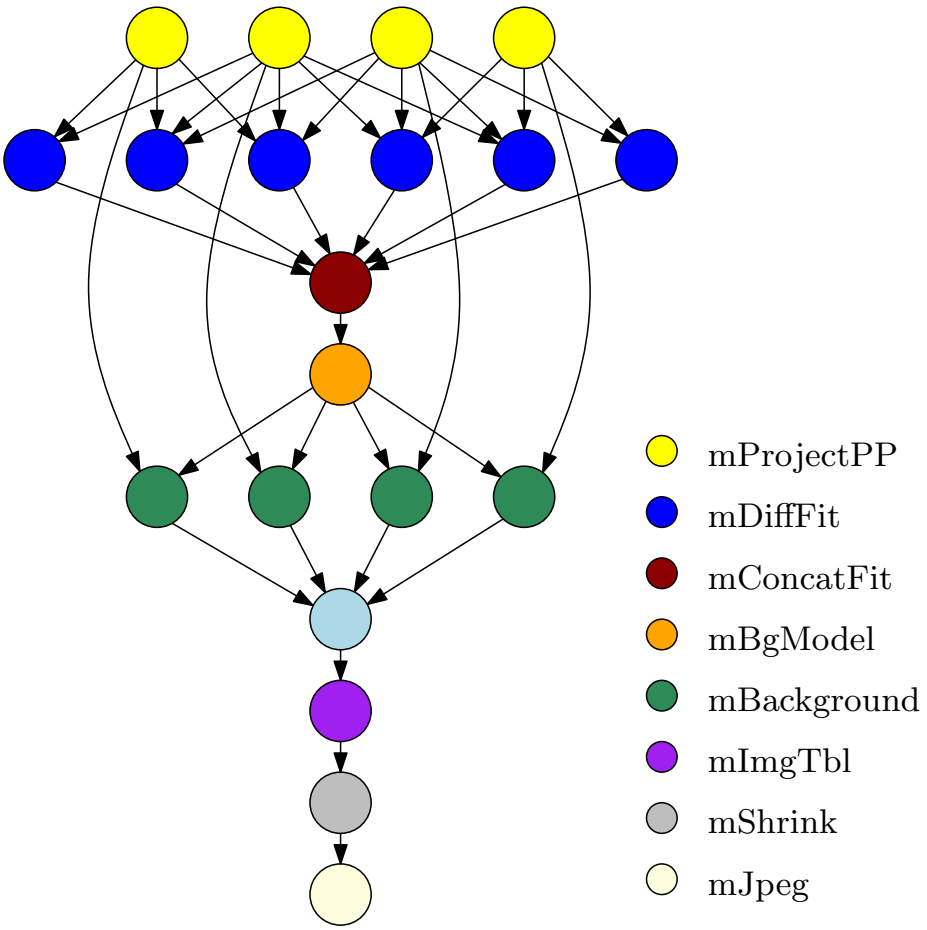


Figure 2: A simplified Montage workflow. The job type is represented by node colour in the workflow DAG.

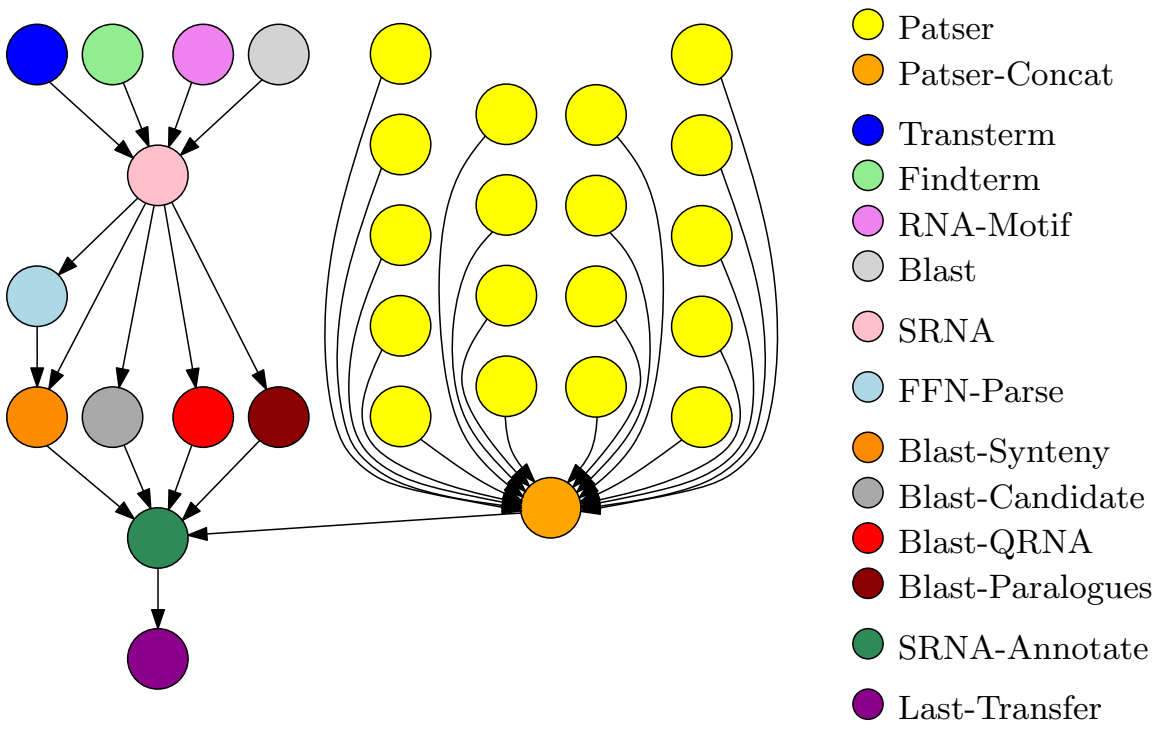


Figure 3: A simplified SIPHT workflow. The job type is represented by node colour in the workflow DAG.

and then to combine the images. The authors of [24] present a grid-enabled version of Montage, in which workflows (as in Figure 2) are submitted to Pegasus for scheduling, and DAGMan for execution [6, 37].

SIPHT, which stands for sRNA Identification Protocol using High-throughput Technologies, is a bioinformatics project at Harvard whose main objective is to search for and identify untranslated sRNAs. Similar to LIGO and Montage, this scientific workflow application also utilizes a distributed computing environment to automate its many independent jobs. SIPHT uses Condor DAGMan for job distribution and workflow execution, with kingdom-wide analysis consuming over 16,000 computing hours [6, 26, 48]. An example of a simplified SIPHT workflow is shown in Figure 3.

Clearly, the need to analyze and process large amounts of data is very real. Composing applications as a workflow permits parallelization of constituent jobs, which decreases the turnaround time when run in a distributed environment. As a result, a

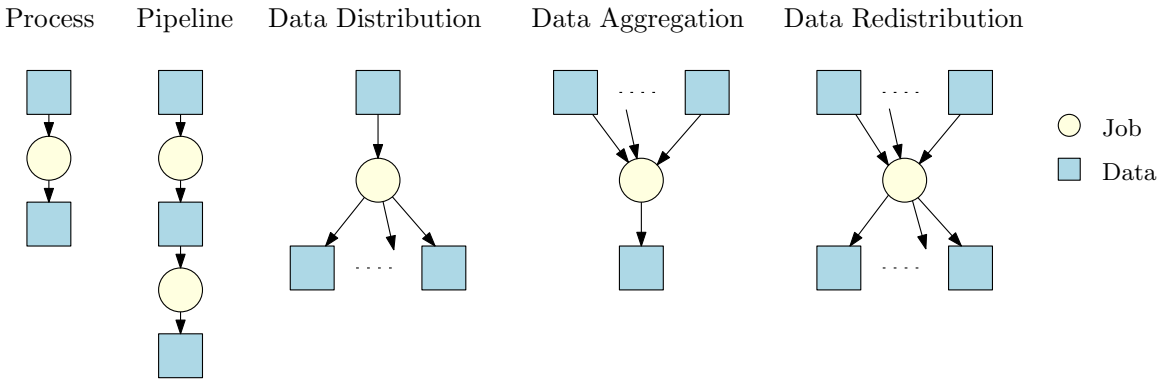


Figure 4: The different workflow substructures identified in [26]. Jobs appear as circles, while input and output data are denoted by rectangles.

large amount of research has been done relating to scheduling of workflows.

The basic substructures of a workflow are characterized in [26] through analysis of the mentioned scientific applications. The main substructures are identified as: a single process, a pipeline, data distribution (*fork*), data aggregation (*join*), and data redistribution [20, 66]. These workflow substructures are shown in Figure 4.

As our proposed method builds on the works [64–66], it is important to note that the authors assume a fork & join type workflow DAG in their research. This type of a workflow represents a class of problems which consist of a single pipeline of jobs, as demonstrated by the process and pipeline workflow substructures in Figure 4. Unfortunately, we see that the characterization of all of Montage, LIGO, and SIPHT are more complex than the supported workflow structure type considered in these works.

2.3 Distributed Environments

Workflow scheduling algorithms are proposed for execution in several different environments: mainly in either a grid, cluster, or cloud. Broadly speaking, a grid is

defined as a collection of resources which are owned by multiple agents, whereas clusters and clouds are generally owned by a single entity. As a result, clusters and clouds tend to be geographically co-located while grids are more geographically dispersed. In addition to these differences, there also exists subcategories which define specific attributes of the environment (eg. community vs. utility grids), as well as properties which can be applied to any environment (eg. resource homogeneity). Community grids contain resources which are contributed by many different users, and are therefore often spread out both geographically and over organizational boundaries [61]. Utility grids however are more similar to cloud environments, as resources are provided by a single entity and accompanied by a usage rate with QoS guarantees. In addition to cloud resources including a cost, the resources themselves are often assigned to and provided by virtual machines. This aspect allows the cloud service providers (CSPs) to provide on-demand provisioning and root access of the machines to potential users. For users of the provided services - called Infrastructure as a Service (IaaS) - this means easier deployment of operating system environments, reproducibility, and lower cost as adjustments to the amount of resources in use can be made dynamically [19, 43].

The differences between environment types have many effects on distributed frameworks, with most of the adverse effects occurring on grids. Solutions to problems caused by resource ownership are mainly reflected in the overall framework design, as well as methods for handling faults, and to a lesser extent scheduling. Despite this factor however, grid scheduling algorithms still exist which consider a large number of job and environment attributes (eg. data storage and location, budget and deadline constraints).

The effects of scheduling on grids mostly appear as an added *planning* stage that occurs before *scheduling* itself [61]. In the planning stage, resources are acquired for use during scheduling. This need for planning is also seen with admission control

Trait	Community Grids	Utility Grids	IaaS Cloud
Availability	Best effort	Reservation	Reservation/On-demand
QoS	Best effort	Contract/SLA	Contract/SLA
Pricing	Free, Usage/QoS-based	Usage/QoS-based	Usage/QoS-based

Table 1: A comparison between different types of distributed environments.

algorithms, which decide only whether enough resources exist for the given job to be properly executed [81, 82]. Admission control algorithms are used when additional constraints for the workflow can be specified (such as budget constraints), and usually also consider both short and long term planning with regards to system resource utilization. When grid frameworks operate in an environment where budget constraints are considered, we call the environment a *utility grid*. Utility grids can be contrasted with regular *community grids* by the approach they take towards availability, QoS, and pricing [71]. In community grids pricing is non-existent, while no guarantees are given on either availability or QoS. Utility grids however allow for advanced reservation of resources held through a negotiated contract which specifies the provided resources and their cost; a Service-Level Agreement (SLA). A comparison between these environments is shown visually in Table 1.

During the planning stage, workflows submitted to a grid framework only consist of information about the contained jobs and any file dependencies. This type of a workflow is called an *abstract* workflow [34,37]. After submission, the actual workflow planning is performed to map jobs to specific resources [61]. It is during this step that resources are acquired to determine the possibility of job execution, which is then followed by *concrete* workflow creation where scheduling occurs. The scheduling of a workflow adds information that maps jobs to specific resources, along with information about job order [34, 37, 61].

When running these workflows on a grid, we can identify the schedulers as either

using *task-based* algorithms, or *workflow-based* algorithms [27]. Task-based algorithms only know of the current jobs which need to be run, and as a result most often perform a greedy allocation of jobs to resources. Alternatively, workflow-based algorithms have access to the entire workflow DAG, and are therefore able to generate a more efficient allocation of jobs to resources. In [27], the authors find that both approaches have similar efficiencies for compute-intensive cases. In data-intensive cases however, workflow-based approaches perform better even though their execution time increases quicker with respect to workflow size.

2.4 Distributed Frameworks

In this section we review several distributed frameworks that are employed in different environments. Even though many frameworks exist [23, 34, 39, 40, 70, 71], we only review Condor, Pegasus, and Apache Hadoop, as they are the most used by related scheduling algorithms. Even though only the Condor and Pegasus frameworks allow for the submission of workflow applications, all reviewed frameworks are composed of subsystems which deal with resource discovery and management, planning, scheduling, user security, and fault tolerance.

2.4.1 Condor

Condor [61] is a distributed computing framework whose development began in 1984 at the University of Wisconsin. Its main development goal is to enable resource sharing across organizational boundaries; the main goal of grid computing. To meet this end, the software is split into two products. The first is HTCondor, a high-throughput computing system. It is realized as a distributed batch computing system that provides facilities for job management, scheduling policies, resource monitoring, and resource management. The second is Condor-G, an agent for grid computing.

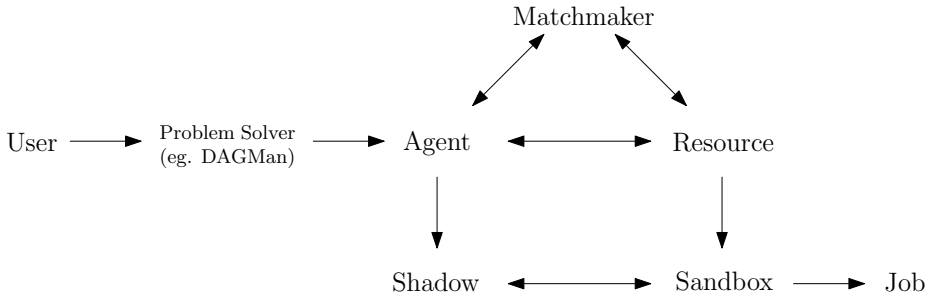


Figure 5: An overview of the Condor architecture. Jobs in a workflow submitted by users are run by agents on resources, paired by a matchmaker. Both agent and resource create a virtual environment for the execution of jobs.

Condor-G is built upon the HTCondor system and the Globus toolkit [41] to allow secure inter-domain communications and access to several remote batch systems.

When running the Condor system, users can decide whether they wish to only use any provided resources, or also provide some of their resources back to the grid. It is this maintained control over their resources that increases the importance of fault-tolerance in this grid framework, as users may arbitrarily decide to retract their resources from the resource pool.

Condor(-G) is organized as shown in Figure 5. A user of the system can submit jobs, which are handled for execution by a problem solver matching the required job format. For example, if the input job is a workflow, then the Directed Acyclic Graph Manager (DAGMan) problem solver can be used for execution, which in this case consists of resource acquisition for individual jobs and fault-tolerance. The figure also shows another aspect of the execution process; the execution of jobs after matchmaking occurs in a sandboxed environment. This is to protect against errors caused by incompatible system configuration.

Resource acquisition is handled by matchmaking (see Figure 6), wherein both *agents* and *resources* in the grid advertise their execution requirements and characteristics to a matchmaker, respectively. When a match is found based upon their requirements, the matchmaker alerts both the agent and resource of the match and

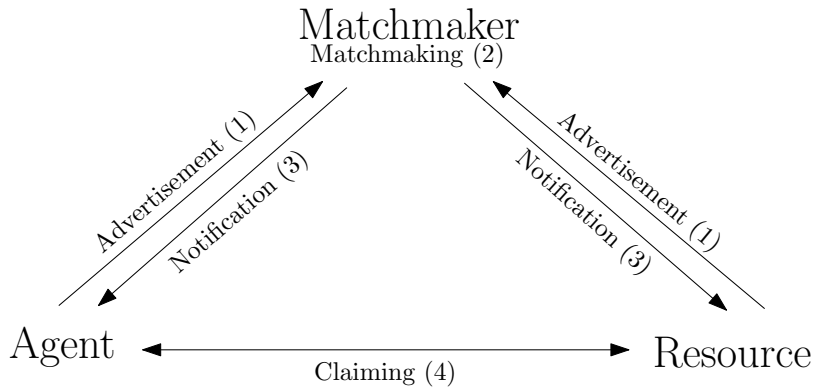


Figure 6: The matchmaking process in Condor. Agents and resources notify a matchmaker of their presence, which then pairs them. Agents and resources can then negotiate to agree upon provided services.

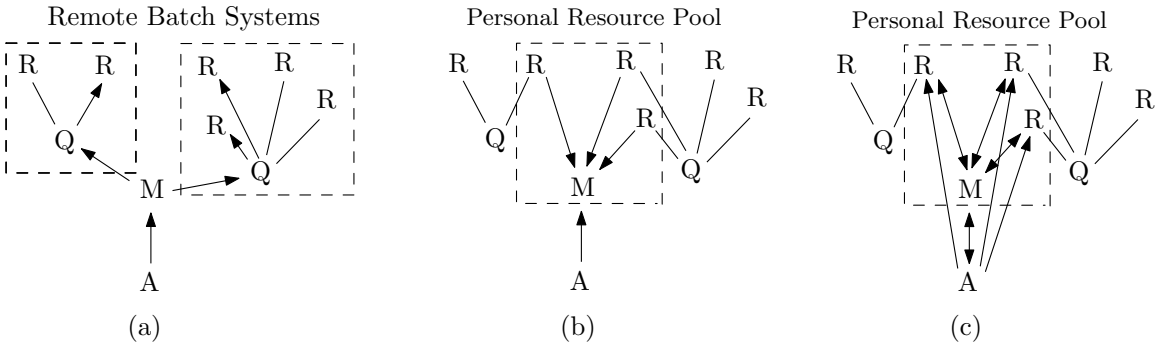


Figure 7: An example of 'gliding in' in Condor. (a) An agent informs a matchmaker of computational needs. (b) The submitted Condor servers are run on assigned resources, which contact the matchmaker. (c) The user now submits a workflow to be run on the claimed resources.

passes contact information to the agents. At this point agents then contact any resources directly to negotiate claims.

Note that in the environment seen by Condor-G, planning and scheduling in the grid is much different than in our assumed cloud platform due to ownership and configuration of resources. When sending jobs for execution, it is possible that they are received at a submission node of a remote batch system. In this case, Condor-G does not know the availability of resources controlled by the remote batch system, and therefore does not have any information on the quality of its decision.

To solve this situation, Condor-G uses a process called 'gliding in' which allows the obtained resources in the remote batch system to act as a single cluster of machines under Condor's direct control. This is accomplished by the agent submitting as jobs the Condor server application itself. The Condor server is thus run on the node in the remote system, where it contacts a predetermined matchmaker created by the same user. It is at this point that the user submits their regular job to the agent, to be executed on the claimed resources using a single centralized scheduling algorithm set by the user. This process is shown in Figure 7.

2.4.2 Pegasus

Pegasus [34, 37] is a framework that deals with the mapping and execution of complex scientific workflows onto distributed resources. It is structured as a set of tools that work on top of, and in unison with the Condor scheduler and the DAGMan workflow engine to facilitate workflow-based applications. It takes as input an abstract workflow and transforms it into a concrete (executable) workflow through a series of refinements, where the mapping of individual jobs to resources is performed by a pluggable scheduler. Along with this mapping, applications and the data they require are also transferred to the correct resources. After creation, Pegasus runs the concrete workflow using DAGMan for execution and Condor(-G) for task execution management.

As an available refinement within the abstract workflow mapping process Pegasus provides options for workflow clustering. Clustering generally partitions the workflow into sub-workflows while maintaining the internal dependencies, as Figure 8 shows via a level-based partitioning. The partitioning has the effect of altering the workflow DAG, and therefore also changes the individual job granularity with regards to execution on a resource. For instance, in [37] the authors apply a level-based partitioning to reduce a scientific workflow (Montage) from 1500 jobs to 35. When run,

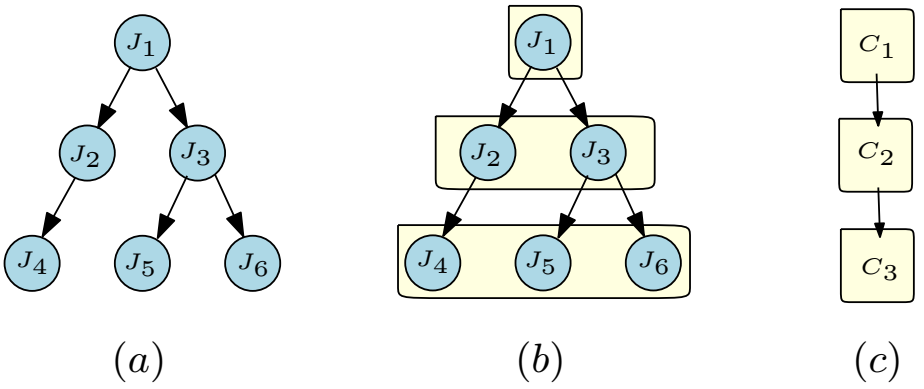


Figure 8: Level-based partitioning (workflow clustering) in Pegasus. Jobs j_i in (a) are assigned a level (b), where (c) each level then corresponds to a node c_i in the generated clustered workflow.

the execution speed was reduced from more than 6000 seconds to 2400 seconds. The clustering is also done to allow for dynamic planning; if only a single partitioned level is refined and executed before continuing then the set of available resources can be reconsidered for each partition. Clearly, this ability is quite advantageous in an environment with highly dynamic resources.

Considering the capabilities provided by grid frameworks such as Pegasus, it is not difficult to see that the differences between a grid, cluster, and cloud environment are minimized with respect to application development and execution environments. With these management systems also providing the ability for pluggable algorithms, schedulers require little modification to be run on disparate frameworks.

In our proposed method, we consider scheduling using the Apache Hadoop MapReduce framework. Unlike Pegasus and related distributed frameworks however, it uses a unique programming model which alters how scheduling can be approached. We review the MapReduce programming model in section 2.4.3, as well as investigating related scheduling algorithms in section 2.5.

2.4.3 Apache Hadoop

The MapReduce programming model was first described in [33] as a method employed at Google for processing large datasets. Since then, several versions of the model have been implemented, the most popular an open source version led by The Apache Software Foundation[1]; Hadoop MapReduce [17]. Hadoop adoption itself is widespread, with hosted services and full stack distributions provided by companies such as Microsoft [14], Google [9], Greenplum [7], and Amazon [3]. In fact, Microsoft discontinued active development of its own general purpose framework (Dryad) in favour of hosting Hadoop on its Windows Server and Windows Azure offerings [4]. Use of the framework also occurs at many organizations: Amazon, Adobe, eBay, Facebook, Google, IBM, LinkedIn, Twitter, Yahoo!, and several universities, for example [11].

The popularity of the model is a result of several factors. Mainly, the open-source nature of the project allows for ease of contributions and low cost of use. Another factor is that the programming model itself allows simpler application code, as the user-written functions comply with a predetermined interface and are run in a standardized fashion by the framework. This allows the framework to handle all of parallelization, fault-tolerance, data distribution, and load balancing itself, including automatic parallelization of input data for concurrent execution on a large number of commodity resources [33]. Execution itself is controlled by a scheduler, with several default schedulers provided by the Hadoop distribution. These include a Capacity Scheduler contributed by Yahoo!, and a Fair Scheduler contributed by Facebook [16]. In addition to these default schedulers, additional pluggable scheduling algorithms can also be used to control the assignment of computational tasks to resources in the distributed environment, allowing improved fault tolerance and optimized execution for a variety of factors [16, 17, 69]. This ability for pluggable scheduling algorithms

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

has attracted much research into the development of alternate schedulers for Hadoop, especially since the default schedulers seem to be optimized for environments with homogeneous resources [42, 69, 76].

Similar to the reviewed Pegasus and Condor frameworks, Hadoop MapReduce is also a framework for job execution on distributed resources. However, MapReduce differs much from these frameworks, if only for the fact that it exploits a functional programming model to allow submitted applications to be parallelized over distributed resources automatically. More specifically, after providing the framework with input data and a set of functions, the framework concurrently executes the functions over multiple nodes using partitioned input data. As a result, we call programs *jobs* before they are split, and label them as *tasks* after the split. This difference is visualized in Figure 9 by its effect on a workflow.

The MapReduce programming model provides an interface for the user-supplied functions to conform to, mainly as the *Map* and *Reduce* functions. Additionally, users can also supply a *Combine* function to the framework. Submitting these functions to the framework along with configuration parameters is all that is required to execute a MapReduce job, which then proceeds as shown in Figure 10. All functions in the framework are written to execute on data organized as $key, value$ pairs, and operate by mainly applying transformations to this data. The input and output formats of user-supplied functions are shown in Table 2, where each format is denoted by a pair k_i, v_j.

Initially, the framework partitions the input data into several smaller chunks, as determined through configuration parameters. Following this, each data partition is processed by the Map function. This initial processing occurs in the Map stage, and is optionally followed by execution of a supplied Combine function to merge local data. After the Map stage, the framework itself modifies the intermediate data, shuffling and sorting it so that the number of partitions matches the number of reduce tasks,

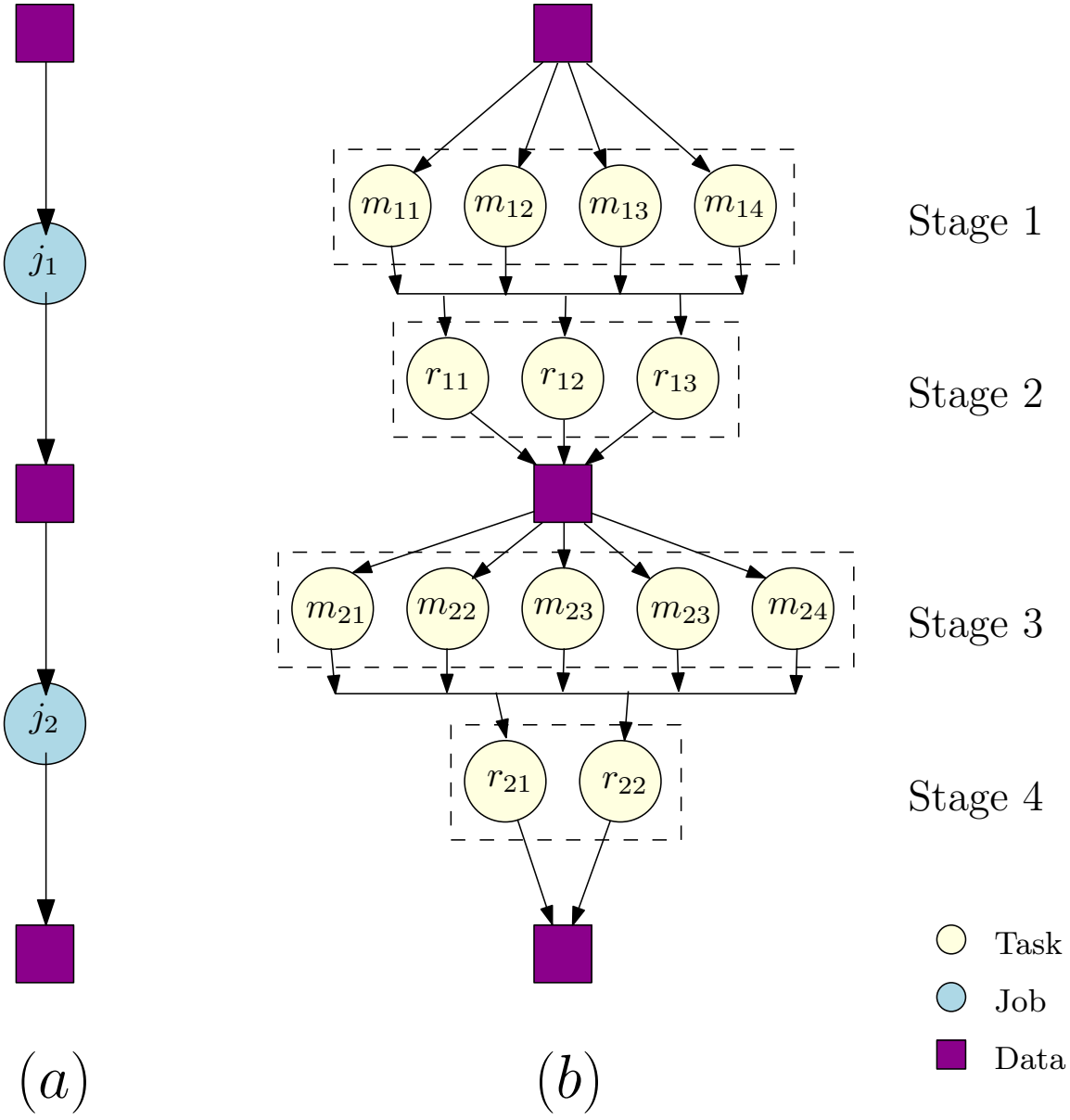


Figure 9: A logical comparison of data and control flow of a simple workflow, both as a sequence of jobs and the tasks they comprise. Each job in part (a) corresponds to several stages of function execution in part (b), where each stage comprises multiple tasks. Specifically, a job j_i has map tasks denoted m_{ix} and reduce tasks denoted r_{iy}, where in this case x and y represent a unique identifier for each task in a single map or reduce stage, respectively.

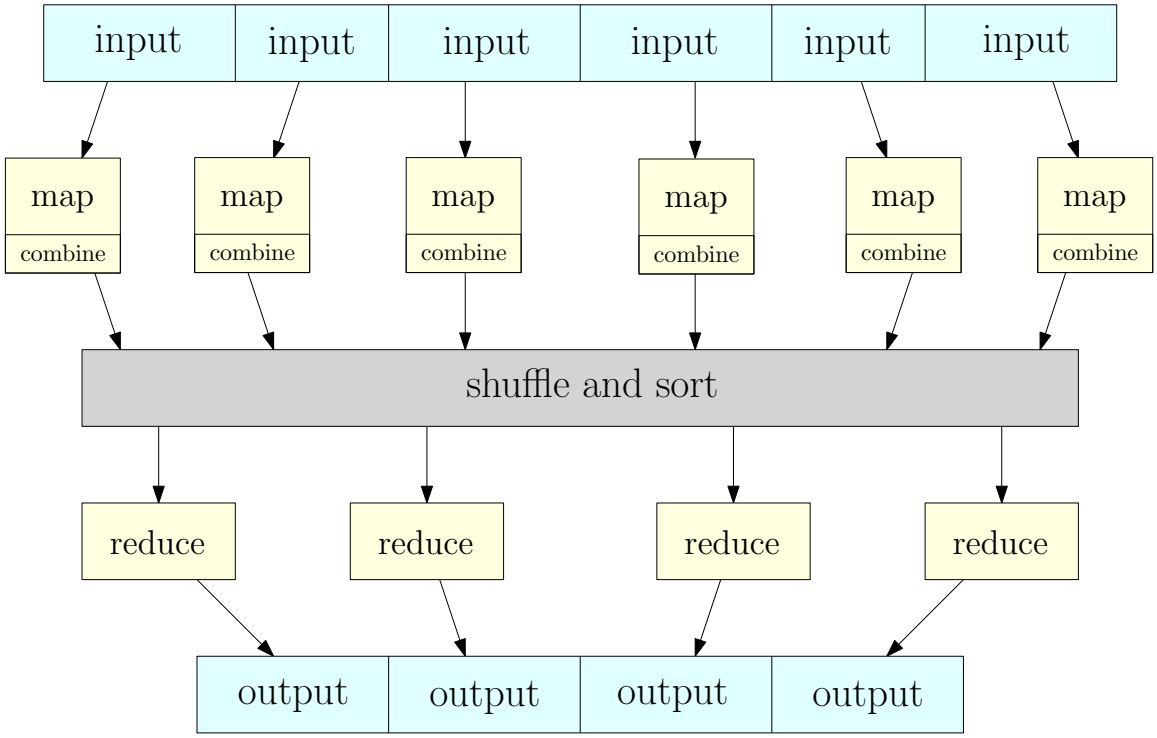


Figure 10: An overview of the flow of execution for a MapReduce job. User-supplied functions are coloured in yellow, with data shown in grey. Operations executed by the framework are the shuffling and sorting of intermediate data, as shown in grey.

and so that all values associated with a single key are processed during the execution of a single reduce function. Following this intermediate data shuffling stage is the Reduce stage. In this stage, the user-supplied Reduce function is executed to process intermediate data into the final output data [12, 33].

Scheduling of the Map and Reduce tasks onto resources is determined by the selected algorithm, which is set through configuration parameters. In older versions of Hadoop, scheduling would occur on a single master *JobTracker* node, with tasks assigned to *TaskTracker* nodes. Resources executing as TaskTrackers would contain multiple *slots* representing task capacity that could be occupied by Map and Reduce tasks [13]. As of version 0.23 / 2.0.0, MapReduce 2.0 (MRv2) or Yet Another Resource Negotiator (YARN) was introduced as a restructuring of the JobTracker and

Function	Input	Output
Map	k_1, v_1	k_2, v_2
Combine	k_2, v_2	k_2, v_2
Reduce	k_2, v_2	k_3, v_3

Table 2: The input and output values of various user-supplied MapReduce functions are shown. In each case these $key, value$ pairs are denoted by a key type k_i and value type v_j.

TaskTracker components [5, 10]. It should be noted however that MRv2 maintains API compatibility with the previous major release versions (1.x series), so applications written using the JobTracker/TaskTracker system still execute correctly after recompilation.

The changes which enable YARN complicate scheduling, but more importantly offer an improved allocation of responsibilities among components. Mainly, the responsibilities of resource management and job scheduling have been moved into separate processes. The YARN architecture consists of a single global *ResourceManager* that contains components to handle scheduling and application management. Replacing the TaskTracker is a *NodeManager* component along with separate *Container* instances. Individual jobs are represented by *ApplicationMaster* instances, which communicate with the ResourceManager and NodeManager(s) to obtain resources on a node. The allocated resources are then assigned to a Container belonging to the ApplicationMaster, wherein a task is run, and monitored by the corresponding NodeManager [5]. An overview of this process is shown in Figure 11.

In section 2.5, we review several scheduling algorithms for the MapReduce framework. Though many scheduling algorithms for MapReduce exist in the literature, few explicitly mention the targeted version of Hadoop for their implementation. However, all reviewed works describe the architecture as containing either a JobTracker or TaskTracker(s), narrowing their implementation to those before the introduction

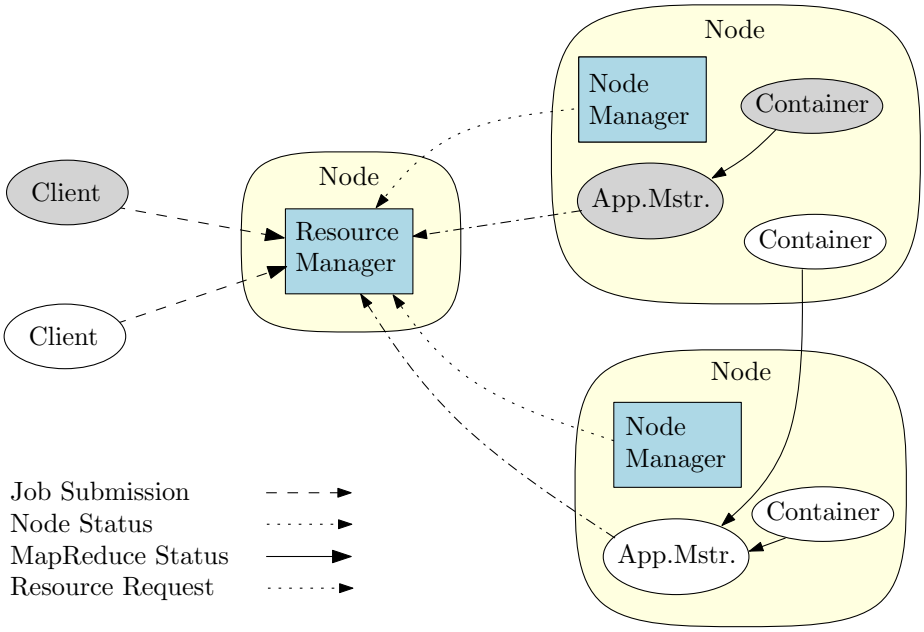


Figure 11: An overview of the YARN architecture, as introduced in Hadoop version 0.23 / 2.0.0.

of YARN. For instance, authors which did mention the version of Hadoop used either $0.20.x$ or $1.x.x$ [44,57,67,79][2]. Due to these reasons, we use the current stable release from the $1.x.x$ line for our implementation; version $1.2.x$.

Using an example from [12], we explain a simple program that uses all mentioned functionality of the MapReduce framework. An overview of the process is shown in Figure 12. Assume we wish to write a distributed word count program to process several plain text files, and have created Map, Combine, and Reduce functions. The input files are first partitioned by the framework according to input split settings. After this, a Map function is executed for each input split, each returning a set of $word, count$ pairs. Assuming a basic Map function, each output pair would only indicate a single word occurrence. Using the provided Combine function, these same-keyed pairs would be merged locally, producing a single pair indicating the total count for a word in the input split. Following execution of the Map stage, shuffling and

[2]The reported version in [67] reads as 20.0, though comparing with releases available at the date of publication we believe this was a typographical error, most likely for version 0.20.0.

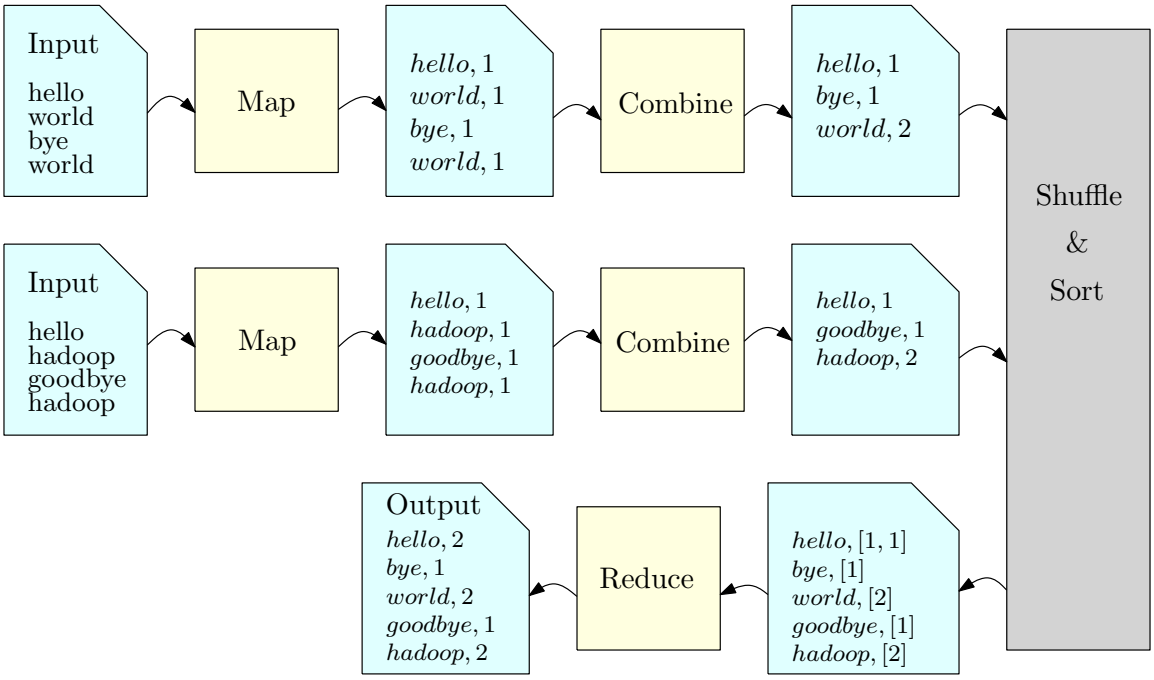


Figure 12: An overview of the WordCount MapReduce job.

sorting would occur, merging the pairs returned from separate map functions by key into $word, list(count)$ pairs. Reduce functions would then be called, each executing on one of these pairs to compute the final data; a total count for each individual word.

The MapReduce framework maintains periodic communication with resources in order to detect failures. When a response from a particular resource does not arrive within a certain amount of time, the resource is marked as failed, task progress is reset, and the task is eventually relaunched on a different resource. Relaunching tasks does not only occur when failures are detected though. Speculative execution, or 'backup tasks' are tasks which are relaunched on different resources when the framework detects that a task's execution has slowed by an unusual amount [33]. The reasoning behind the method is that the relaunched task has a chance to complete before the original slower task, which would increase the throughput of the currently running job [76].

Speculative tasks have the lowest priority, being considered for execution only after

both failed tasks and tasks which are not running. Note that failed tasks are given the highest priority for speculative execution to allow detection of repeated failures. For selection of the task to be speculatively executed, the progress of running tasks is monitored as a value between zero and one. Map tasks are given a value proportional to how much input data has been read, whereas Reduce tasks are given a value based on their progress through several phases. There are three phases in total, which comprise the copy, sort, and reduce phases. Each of the three phases account for one third of the value. Average values are computed for each type of tasks (Map or Reduce), and used for creation of a threshold value for speculative execution, where all tasks beyond the threshold are considered candidates. Several scheduling algorithms consider changes to speculative execution for improved task selection [42, 69, 76, 81].

2.5 Scheduling Algorithms

In the following sections, we review various algorithms relating to scheduling in distributed environments. Each section is organized to contain related work for a particular type of scheduling: either deadline-based, deadline-constrained, deadline & budget optimization, or budget-constrained. We define deadline-based scheduling as those algorithms that only aim to minimize the makespan of a job or workflow. Deadline & budget optimization algorithms attempt to minimize both cost and makespan of the schedule, usually using the trade-off (comparative advantage) between resources to jointly optimize the computed schedule. Deadline-constrained scheduling includes algorithms that attempt to minimize cost while satisfying a deadline constraint, and budget-constrained algorithms attempt to minimize makespan while satisfying a budget constraint. We also briefly review several less related scheduling algorithms in this section to give a more complete picture of the existing research on scheduling for Apache Hadoop and in the grid. Note that in addition to our summaries, many of

these algorithms have also been reviewed in [53].

In terms of relation to our work, the algorithms presented in Section 2.5.4 are most similar in their overall goal: budget-constrained workflow scheduling in a heterogeneous distributed environment. However, they may be developed with the grid in mind rather than the IaaS cloud, or may not assume a specific framework. In fact, many actually assume use of a framework other than Apache Hadoop, especially for those that consider workflow scheduling.

Overall, limited research has been done regarding scheduling of workflows using Apache Hadoop. Most algorithms for the architecture instead consider scheduling for multiple independent jobs. Along with this, most focus on improvement of scheduling with respect to makespan through changes to resource utilization. In particular, optimization of job makespan depends upon the relationship between the selected resource for a task and the method of data placement. As a result, several methods for arranging tasks and their required data are suggested in the literature. As an illustration, the authors of [68] propose a scheme to adaptively balance the amount of data stored at nodes to ensure an equal processing load. Alternatively, the placement of map and reduce tasks are coupled in [59] as an attempt to jointly optimize their execution. Lastly, the authors of [44] realize that task assignments by the default Hadoop schedulers do not consider the type of work executed by tasks. To deal with this deficiency, it is proposed that jobs and resources be profiled to categorize workload. Determining whether a task is I/O, CPU, or network bound, the scheduler can then appropriately assign tasks to resources as to balance workload type, thereby increasing overall resource utilization. Several works also consider the direct reduction of resource use through changes to criteria for speculative execution [42, 69, 76].

There has also been research into improvements to fairness for multi-user environments as suggested in [51, 57, 67, 78]. As one example, a dynamic proportional share scheduling algorithm is proposed in [57], where user spending determines resource

allocation. In the work, tasks are assigned to resources on a per-slot basis, based on bids made by users. In this way, cost is incorporated into the algorithm as a determinant of the provided Quality of Service (QoS); users who value a higher QoS will necessarily pay more than others to receive it. Scheduling is executed in stages called *allocation intervals*, wherein resources are reassigned based on bids placed during the last interval. Specifically, a user receives an amount of resources corresponding to the portion of their bid relative to the total amount bid. Similar to the other works mentioned in this section, this research has little relation to ours: workflows are not considered, no specifics are given on budget constraints or deadline minimization, and the method of charging for resources is quite different (users bid for time amongst themselves instead of renting resources from a cloud service provider).

Lastly, several approaches in the following sections use meta-heuristics such as Genetic Algorithms [32, 71, 72]. Genetic algorithms are based on the principle of evolution, and work to combine previously found high quality solutions with exploration of new regions in the solution space. A genetic algorithm works by repeatedly generating solutions based on previous iterations. They keep high quality solutions, generate new solutions from the retained solutions, and discard the lowest quality solutions such that a constant number of solutions are maintained. Solutions are ranked by a *fitness function*, with operations called *crossover* and *mutation* defined to generate new solutions. The solutions are encoded as strings, and are often called *chromosomes* as an allusion to the origins of the technique. Crossover creates new solutions from multiple solutions by combining and permuting sections of existing solutions, whereas mutation permutes the contents a single solution.

2.5.1 Deadline-Based

Several workflow scheduling algorithms we review are based upon the Heterogeneous Earliest Finish Time (HEFT) algorithm proposed in [62] for scheduling on the grid.

HEFT itself does not consider a budget-constrained model, so the methods which employ it either augment the algorithm to consider the constraint, or use it for subproblem solution [21, 22, 32, 56, 73, 81]. HEFT is a list-scheduling algorithm, and as such operates by assigning priorities to jobs based on their upward rank followed by assigning jobs to resources based on the job's earliest finish time. Specifically, upward rank is first defined for a job as the length of its critical path to an exit job, where a higher upward rank corresponds to a higher priority. Jobs are then assigned in accordance with their priorities to the resource which minimizes the job's earliest finish time.

Around the same time that the capacity and fair schedulers were implemented in the Hadoop source code, the authors of [76] proposed the Longest Approximate Time to End (LATE) algorithm. Arguing that the platform's performance is closely tied to its scheduler, the authors realize that the current assumptions made by speculative execution lead to suboptimal performance in a heterogeneous environment. To remedy this situation, they devise an algorithm which always speculatively executes the task that is thought to finish the furthest into the future. The reasoning is that this task provides the greatest opportunity to reduce the original task's response time. Specifically, the algorithm is a greedy heuristic which selects tasks based on their estimated time to completion, calculated as the ratio of the task's remaining progress to its overall rate of progress. In addition to this selection rule, the algorithm is augmented by several other criteria. Firstly, speculative tasks are not executed on slow nodes (as determined by their overall progress). Secondly, a limit on the total number of speculative tasks is created. Lastly, a new threshold is introduced to determine if the overall execution progress is slow enough to warrant speculative execution. These criteria support the fact that speculative tasks occupy resources that would otherwise be used to compute unexecuted tasks in the workflow.

Building on the LATE scheduler, the authors of [69] propose changes to improve

fairness and allow dynamic loading. These issues are demonstrated by LATE only operating on the first job in a queue of jobs, and by the use of a constant progress rate in the calculation of estimated finish time (other running processes could alter system load). To improve fairness, task assignment to empty slots is altered to be based on the availability of tasks from all jobs in the queue. To correct the progress rate estimation, system-level information is gathered (eg. system load, CPU frequency) and used during a task's execution instead of information kept by the framework. In the case that a speculative task is to be executed, estimation of task response time is calculated using the new method, so that scheduling only occurs if it would truly shorten the job's response time.

The authors of [42] also consider changes to speculative execution. In their work, Benefit Aware Speculative Execution is suggested to predict the benefit of running new speculative tasks. Similar to [76] and [69], the proposed method only allows speculative tasks to be launched if they are expected to improve execution time. Unlike the mentioned works however, it uses historical information about completed tasks from the same job to determine if a task is executing slowly. Also suggested is resource stealing, which addresses the issue of inefficient resource utilization. Resource stealing occurs when there are more task slots than available tasks, and allows tasks to use resources provided by the idle task slots until they are required.

Resource-aware scheduling is investigated in [51], which considers scheduling of multiple independent jobs, each with individual deadline constraints. As a goal, the proposed algorithm aims to determine the best possible placement of tasks for maximization of resource utilization. Interestingly, the authors use job profiling information to dynamically adjust the number of task slots on each resource, and for proper workload placement during allocation. Following their problem formulation, the resultant optimization objective is shown to be a variant of the class constrained multiple-knapsack problem, so a heuristic is adopted. The heuristic places reduce

tasks first to allow an even distribution across the nodes. Afterwards, map tasks are placed as per the objective function and the provided constraints.

2.5.2 Deadline-Constrained

In [74], the authors present an algorithm for deadline-constrained scheduling of scientific workflow applications on utility grids. They propose a divide and conquer technique which first divides a workflow graph into partitions before individually scheduling each partition using objective functions to rank cost. Partitioning, as illustrated in Figure 13, is done by categorization of individual jobs in the workflow as either simple, or synchronization. A simple job is defined as a job which has only a single parent and child, whereas a synchronization job has more than a single parent or child. Paths of simple jobs are assigned as a single partition, as are individual synchronization jobs. After partitioning, the deadline is distributed over the partitions with respect to several policies, and then assigned to the jobs using critical path, breadth-first, and depth-first analysis methods. The defined policies enforce that: deadline is assigned proportional to partition processing time, minimum sub-deadline length is greater than minimum processing time, the cumulative sub-deadline length from the entry node to the exit node will have deadline of at most the input deadline constraint, and that the cumulative sub-deadline of any independent path between two synchronization jobs is equivalent. Planning then occurs individually for each partition, where jobs are allocated to resources which meet the deadline at the lowest cost.

In [19], the authors consider workflow scheduling on IaaS clouds. Similar to cloud service providers such as Amazon [1], they assume that resource use is charged based on the number of time intervals used, for both computation and storage. Resource cost itself is defined by QoS parameters, such that a higher QoS (computation speed, memory, storage, etc.) corresponds to a higher cost. In their work, the authors adapt

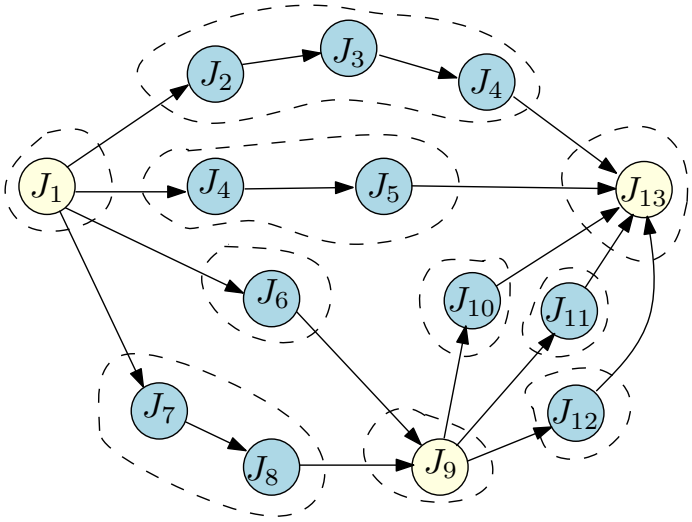


Figure 13: A workflow DAG partitioned as in [74]. Synchronization jobs are coloured yellow while simple jobs are coloured blue. Dashed circles enclosing job nodes indicate partitions.

an algorithm from their previous work on utility grids to the IaaS Cloud environment, with the objective to minimize workflow cost while satisfying deadline constraints.

Similar to other reviewed scheduling algorithms for utility grids, their Partial Critical Paths (PCP) algorithm comprises two main phases: deadline distribution and planning. In the deadline distribution phase, jobs along the critical path are found. Then the overall deadline is distributed to these jobs using one of several proposed policies. The assignment of deadline to all remaining tasks is then carried out recursively, as they are now considered a part of the critical path due to their successor's assigned latest start time. During the planning phase jobs are then assigned to the least expensive resource which meets their deadline.

The PCP algorithm is modified to produce a single-phase algorithm (IC-PCP), and another two-phase algorithm (IC-PCPD2). IC-PCP is single-phase as it schedules workflow jobs during deadline distribution instead of assigning a sub-deadline to jobs. When run, the Earliest Start Time (EST), Earliest Finish Time (EFT), and Latest Finish Time (LFT) of all jobs are first computed (assuming assignment to the fastest

resource). Following this, a recursive function is called on the exit job to accomplish scheduling. The function iteratively schedules all unassigned parents of the input job, beginning with the set of jobs which exist along the critical path. The path is followed back to the entry job, where all jobs on the path are then scheduled on a single least expensive resource which can complete them before their latest finish times. The EST, EFT, and LFT values of the jobs along this path are then updated before the recursive call on the assigned jobs. IC-PCPD2 is similar to IC-PCP, with the main difference that sub-deadlines are assigned to jobs instead of scheduling jobs onto a resource during the recursive function. Additionally, a planning function is added to execute after the recursive function. The planning function schedules each job on the least expensive instance that can complete execution of the job before its sub-deadline (IC-PCPD2 schedules singular jobs instead of a set of sequential jobs).

Similar to the model proposed in [19], the authors of [49] also consider workflow scheduling on IaaS clouds. In their work, they propose a method based on the formulation of a mixed integer programming problem. As such, mathematical constraints are created along with an objective function to be minimized. The main novelty of their work includes the use of multiple IaaS clouds, and the structuring of workflows as levels of independent identical tasks. This seems relevant to our work, as in the Hadoop MapReduce model jobs are split into multiple identical tasks which are able to execute concurrently. Along with this, the use of multiple IaaS clouds makes little difference as it can be viewed as the addition of more resource types. However, one clear difference between our work and the proposed algorithm is that we consider budget-constrained scheduling as opposed to deadline-constrained scheduling.

The work presented in [46] offers a deadline-constrained scheduling algorithm for energy reduction of MapReduce jobs that considers constraints through user SLAs. The environment model assumes a heterogeneous cloud, where resource costs are able to be influenced through dynamic scaling of processor voltages and disk speeds. The

proposed algorithm schedules map tasks first, prioritizing them using their calculated average execution time. Resources for the tasks are then selected as those which result in the minimum completion time. After scheduling of the map tasks, the reduce tasks are scheduled. They are prioritized in descending order of intermediate key data size, and assigned to the resource which produces the minimum earliest finish time. Assuming that the schedule meets the deadline constraints, slack time is then calculated and reduced using the resources' dynamic scaling for the purpose of further cost minimization.

2.5.3 Deadline & Budget Optimization

The main contribution in [32] is the proposal of a method to integrate partitioning and provisioning of large-scale scientific workflows using a Genetic Algorithm (GA). The GA is designed to enable concurrent consideration of job selection and placement by designing chromosomes that contain both job information and resource characteristics. The fitness function is defined as the composition of several functions, each based on individual characteristics such as the minimization of makespan, minimization of cost, validity of cost and makespan with regards to the constraints, and the trade-off of makespan and cost. As such, these functions are all considered together for optimization, including both budget and deadline constraints. Note that when required for use in the fitness functions, makespan values for sub-workflows are computed using HEFT [62].

Optimization of both budget and deadline constraints during scheduling has also been approached in domain-specific areas. As one example, [38] considers workflow scheduling for audio and video editing in the media processing and distribution industry. In this domain however, the problem is presented as scheduling for multiple

workflows, each containing individual cost and deadline constraints. Additionally, resources may have multiple different service types available (eg. transcoding, CGI rendering). Similar to several reviewed works, the proposed method uses a list-scheduling technique. Using this technique, jobs in a workflow are given a priority which determines their execution order, with the entry node initially assigned as the highest priority job in a workflow. In this case, the job priorities are recalculated after each assignment of a job to a resource. For the assignment of job priorities a random assignment is considered, along with an assignment which gives higher priority to jobs which are currently violating constraints, and assignments which give a higher priority to jobs depending upon resource availability. Several strategies for assignment of jobs to resources are also presented, including a random assignment, an assignment that executes the job close to its required data, and assignments based on the relative values of completion time, lowest cost, or a weighted average.

Another list-based scheduling method for execution on the cloud is proposed in [77]. It is a greedy heuristic which assigns priority and resources based upon several objective functions that characterize the comparative advantage of alternate assignments. The initial assignment phase considers the differences in cost and finish time of the new assignment with the current best assignment, which is followed by reassignment that considers the current total makespan and cost. After assignment, a slack removal step occurs which attempts to reduce the amount of resource idle time by moving jobs to available idle slots. Similar to the other methods in this section, the authors do not consider a strict budget or deadline constraint in their research, and instead attempt to minimize both constraints through use of their comparative advantages.

2.5.4 Budget-Constrained

The LOSS and GAIN algorithms are proposed in [56], and work by modifying an initial job assignment 'optimal' for one of the attributes (budget or deadline) until the workflow makespan has been minimized with respect to the budget constraint. LOSS begins by using the HEFT algorithm to provide an initial assignment optimal for makespan. After this initial assignment, jobs are reassigned to resources such that the exchange results in a minimum amount of increased execution time for the maximum cost savings. Job priorities for this reassignment are given by weightings computed using the function

$$LossWeight() = \frac{T_{new} - T_{old}}{C_{old} - C_{new}},$$

where T represents execution time and C represents cost. For prioritization itself, the algorithm considers jobs that correspond to the smallest calculated values. Note that this swapping function is the same as used in [29]. Inversely, the GAIN algorithm begins with an assignment corresponding to the least expensive budget, and exchanges jobs using a function defined as

$$GainWeight() = \frac{T_{old} - T_{new}}{C_{new} - C_{old}},$$

prioritizing jobs that correspond to the largest values. To reduce execution time, both algorithms only try to reassign any pair of jobs once. The authors also consider two other variants of the weight functions: the first modifies the functions to consider overall makespan improvement instead of individual makespan improvement, while the second increases the frequency of weight recomputation to occur after each reassignment. Results show that while both approaches produce correct schedules with respect to the constraints, the LOSS variants generally produce a better makespan

than the GAIN variants. With regards to relation to our work, the LOSS and GAIN algorithms are designed to execute in a utility grid setting, and as such do not consider the use of any particular framework (Apache Hadoop or any other).

In addition to scheduling algorithms seen thus far, service providers and users also make agreements based on contractual guarantees of program execution time and provided resource quality. In these cases, scheduling is used to determine in advance if a user's request for a particular QoS is feasible. Algorithms used for these situations are called *admission control* algorithms, and have generally been developed for budget-constrained scheduling in utility grid environments [81, 82]. In [81], an admission control algorithm is proposed in which both the order of job selection and job to resource mappings are decided based on priorities. In the algorithm, job priorities are computed using the upward ranking method defined by HEFT, with resources selected based on several rules. For resource selection, the set of viable resources is first filtered based upon available budget. If the set is not empty at this point then the resource with the earliest finish time is selected. Otherwise if the set is empty then the amount of remaining budget is considered: if there is still some budget that remains, then the resource with the earliest finish time is selected; otherwise the least expensive resource is selected. This algorithm is only tangentially related to budget-constrained scheduling, as computation of a valid schedule only determines if the submitted workflow is able to run within the user's supplied QoS constraints; it is not considered how to minimize the execution time or cost of the scheduled workflow.

In [71], the authors approach the budget-constrained scheduling problem using a genetic algorithm. They begin by first defining the conditions that a valid solution must satisfy. As such, they propose that a job can only be started after all its predecessors have been completed, and that all jobs must be run exactly once on a resource. A string encoding is then created to represent the schedule, which captures both the resource that the job is to run on, and the ordering of that job on the

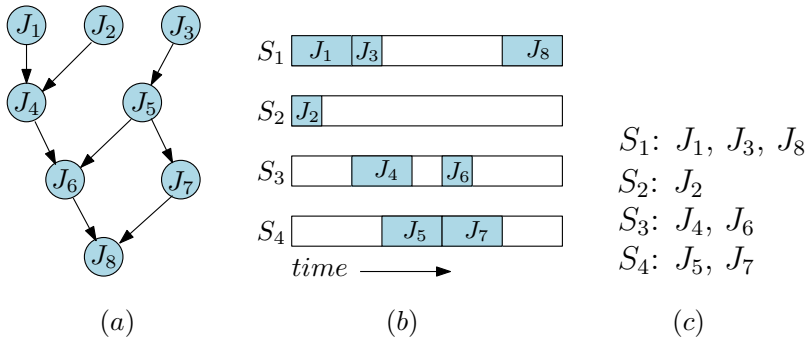


Figure 14: (*a*) The DAG representation of a schedule as proposed in [71]. A schedule (*b*) is show for the workflow in (*c*); its two-dimensional string representation (*d*) is then transformed to a one-dimensional encoding for application of genetic manipulation functions.

resource. The representation itself is a two-dimensional string, though is converted to a one-dimensional string for manipulations by the genetic algorithm (see Figure 14). After definition of the schedule encoding method, a fitness function is defined as the composition of functions which individually consider budget and deadline. The authors explain their implementation of genetic operators next. The crossover function first selects two schedules (chromosomes) at random, and from one selects all jobs between two random points. Each job is then exchanged with respect to the resource on which it runs on the opposing schedule. The defined mutation function takes as input only a single schedule, and selects two jobs from a resource at random whose execution order are then exchanged. Two additional functions are added to the process: a time slot assignment method and a schedule refinement technique. The time slot assignment method simply corrects any invalid schedules after either of the crossover or mutation operations have been run. The schedule refinement technique however is only run intermittently, and uses the partitioning method from [74] to reschedule jobs on less expensive resources when there is available slack time.

Four scheduling algorithms are presented in [29] that are formulated using objective functions. Two work to maximize throughput under a budget constraint (B-RATE, B-SWAP), while the remaining two work to minimize execution cost given a deadline constraint (TP-RATE, TP-SWAP). As we are concerned with makespan minimization with respect to a budget constraint, we only review the B-RATE and B-SWAP algorithms. Similar to several of the other reviewed algorithms, both B-RATE and B-SWAP are developed under the assumption of an arbitrary grid environment, as opposed to use with a specific framework (eg. Apache Hadoop).

The B-RATE algorithm works by first separating workflow jobs into ordered layers based on their dependencies (similar to Figure 8). A cost constraint is then calculated for each layer, followed by scheduling for each individual layer. For a job in the current layer, valid resources are first selected considering the current mapping of jobs in previous scheduled layers. All pairs of assignments of jobs to candidate resources are then considered through cost calculation. Resources are filtered by selecting those which satisfy the cost constraint, with final selection determined by changes to makespan. If multiple resources exist which have minimal addition to makespan, the the resource with minimal cost is selected.

The B-SWAP algorithm works by first scheduling all jobs to the most expensive resource(s) to produce a schedule with maximal throughput. Schedule cost is then calculated and compared to the budget constraint, and if the cost is larger than the constraint then jobs are remapped to a different resources. Remapping is based on a weight function which considers the difference in throughput and cost, attempting to swap jobs to resources which decrease the throughput the least for the most change in cost. The weight function is defined as

$$LossWeight() = \frac{Throughput_{Cur} - Throughput_{New}}{Cost_{Cur} - Cost_{New}},$$

where the smallest values are considered.

More related to our work is the research presented in [47]. Here, the authors also approach budget-constrained scheduling for scientific workflows in the cloud. However, they focus on scheduling in the generic IaaS cloud (using Amazon EC2 prices), as opposed to consideration of a specific framework. Also different from our approach is that workflow jobs are clustered based on dependencies before scheduling, wherein each cluster is then scheduled on a separate virtual machine in the cloud. During problem formulation, data transfer cost across the network is incorporated, though this attribute is then ignored during algorithm design as the authors decide to focus only on compute-intensive workflows. Their argument is that the time for data transfers for these types of workflows constitutes so little of the overall execution time that it can be assumed negligible. In addition to proposal of their greedy heuristic, the authors also prove that the formulated scheduling problem is NP-complete and non-approximable.

The author's proposed greedy heuristic is called *Critical-Greedy* (CG), and similar to [66] and our algorithm, it begins with an assignment of resources that produces the least-cost schedule. After this step, job cluster rescheduling is carried out according to time and cost differences between different intermediate results. Specifically, job clusters on the critical path are compared to select the cluster with the largest execution time reduction whose cost difference is still within the remaining budget. This selected cluster is then rescheduled onto a different virtual machine. Since after each reschedule the critical path is able to change, the current critical path is updated if necessary. These steps are then repeated until no rescheduling is feasible with the remaining budget. Performance evaluation is executed through both a simulation and experimentation, for both pipeline and random DAG structures. In both cases, the CG algorithm was tested against a variant of the GAIN algorithm and found to give superior performance.

Finally, the authors of [64–66] propose the model which our research is based upon. They consider workflow scheduling in a heterogeneous IaaS cloud environment, and mainly approach budget-constrained scheduling. The input workflow is considered for scheduling after the jobs are split into their respective map and reduce tasks, allowing more fine-grained control over scheduling. The workflow is assumed to match a fork & join model however, which both simplifies the problem and allows the workflow to be viewed as multiple dependent sets of independent jobs, labelled by the authors as a k-stage workflow. Similar to scheduling seen in previous algorithms, the tasks are assigned resources from a set of machines, each possibly possessing a different cost. To record these costs in a useful way, a table relating execution time to the price of paired tasks and resources is created, and sorted by times in increasing order and prices in decreasing order (cost and execution time are implicitly assumed to be inversely proportional). Two main algorithms are proposed in [66], which we review forthwith. Note that our work differs in several ways. First, we assume arbitrary DAGs in our method, instead of a simple fork & join workflow. Secondly, our implementation is written for the Hadoop framework and tested against other real schedulers as opposed to a simulation.

The first method proposed is a globally optimal algorithm that uses a greedy method for individual stage computation, followed by dynamic programming to distribute budget over the stages. For each single stage, the goal is to distribute the budget among constituent tasks such that the finish time for the stage is minimized. This is done by selecting a task in the stage which has the longest execution time and allocating additional budget to it, allowing for reassignment that results in a quicker execution time. This local solution is then extended using dynamic programming to consider all stages. An equation is created to calculate the minimum time for computation given a range of stages and available budget, which is then used by a dynamic programming recursion to build up a complete solution.

The second algorithm is a greedy heuristic extended from the previous local greedy method. Named *Global Greedy Budget* (GGB), the algorithm works by iteratively selecting a single task to reschedule from the set of slowest tasks, as based on a computed utility value. To construct this set, the algorithm first determines the slowest and second slowest tasks from each stage. Each pair of tasks are then compared to generate the utility value, which provides an indication of the relative changes in execution time and cost caused by a rescheduling of the slowest task. Each slowest task from a stage is added to this set, weighted by its computed utility value. At each iteration of task selection, the selected task is rescheduled on a quicker resource (assuming sufficient budget), after which the set of tasks is then recomputed. If the budget is insufficient for a particular selected task, the task is simply skipped. In this manner, the algorithm continues rescheduling until there are no longer any slowest tasks eligible for rescheduling. With respect to computational complexity, the authors provide a bound of $O(B(n + k \log k))$, where n is the total number of tasks in the workflow, and B is the given budget.

Chapter 3

Problem Formulation

3.1 Assumptions

Our assumed execution environment consists of a number of virtual machines rented from an IaaS provider. As we rent the resources, we can assume that the cluster is not shared with other users, and that there are no other programs running that would monopolize resources. Different types of virtual machines can be rented based on our needs, allowing for creation of a heterogeneous set of resources for computation. Of course, the provider charges different service rates for these provisioned machines, which are proportional to their attributes (such as processing power and amount of memory) [21]. We also note that even though some providers dynamically alter service rates based on market demands, we assume a static rate during scheduling.

We consider an environment wherein the Apache Hadoop framework is run on the rented machines, and focus on the use of MapReduce for workflow scheduling. As mentioned previously, workflow scheduling on Hadoop is not much studied, with only a few approaches considered [45, 50, 60, 66]. Concerning these studies, only [66] considers scheduling in a budget-constrained environment. In their research however, the authors also constrict the scheduled workflows to a simple fork & join structure. Following from the research in [66], we also consider budget-constrained workflow

scheduling, though assume no artificial limitations on input workflows.

As we consider general workflows which comprise a set of interdependent jobs, we can represent workflows as DAGs. For our problem, we define a DAG as a directed graph comprising a single connected component that contains no cycles. DAGs consist of a set of vertices (nodes) V along with a set of edges E, and can be formally defined as a tuple $G = (V, E)$. When a DAG represents a workflow, each vertex $v_i \in V$, $1 \leq i \leq |V|$ represents a job in the workflow. Edges are denoted by $e(i, j)$, and correspond to the path from v_i to v_j. This path represents either a control or data dependency constraint which requires v_j finish execution before v_i begins execution. Using this notation v_i is called a *parent* of v_j, and v_j a *child* of v_i. When edges and vertices in the DAG are given weightings, they represent the communication cost and processing cost, respectively. Costs are usually representative of time, though they may also be considered as monetary cost if the environment takes into account budget constraints during scheduling.

Mainly due to the fact that Hadoop abstracts the task of data organization throughout the cluster nodes, we do not consider the cost or time of data transmission in our model. However, as the resources rented from an IaaS provider are supplied with root access, we can assume complete control over the resources. This assumption also allows us to assume control over the configuration of the Apache Hadoop framework, and as such we can configure the number of map and reduce slots provided by different resources. Additionally, control over the Hadoop configuration allows the number of map tasks created for a job to be selected. The number of virtual machines available to rent from the IaaS provider is also assumed to be configurable, and only limited by the given budget constraints. Therefore, we can assume that machines (slots) are never competed for by more than a single task.

Our algorithm assumes scheduling of a single workflow at a time, each of which has access to all resources provided by the Hadoop cluster. For our tests we mainly

execute the SIPHT workflow, as shown in Figure 3. To produce a schedule for the input workflow, we assume that dependency information is provided at the beginning of execution along with all required data files and individual jobs. Individual task processing times are also assumed to be known (and can be calculated by various methods), where tasks split from the same job are generally assumed to be homogeneous in both execution time and resource utilization. Task processing times are modelled along with their cost for each machine type, and the information stored in a table (as in [66]).

3.2 Formulation

Similar to the algorithms proposed in [66], we consider makespan minimization through reduction of task execution time. In particular, tasks that lie on the critical path(s) of the workflow DAG have their execution time reduced through rescheduling on faster, more expensive machines. To accomplish this, we consider the workflow as consisting of k *stages*, where a single stage S_s is defined as the set of all map (or reduce) tasks in a single job: $S_s = \{\tau_{s1}, \tau_{s2}, \ldots, \tau_{sn_s}\}$, where $0 < s \leq k$, and n_s tasks exist in stage S_s. We also denote the total number of tasks in the workflow as n_τ. Decomposition of the workflow in this manner is permitted by data-flow constraints caused by the framework. Specifically, since all map tasks of a job J_j must complete before any reduce tasks of J_j can begin, and all reduce tasks of a job J_j must complete before the map tasks of any successor of J_j can begin, we can group these tasks together with regards to execution time and priority.

As the cost and execution time of tasks in a workflow are determined by the machine they are scheduled on, we need to account for this in our model. For each task, we store this information in a *time-price* table, as shown in Table 3. The table contains the time (t) and price (p) information with regards to all $M_u \mid 0 < u \leq n_m$

available machine/resource types for a specific stage S_s and task $\tau_{s\tau}$. For ease of notation, the table has task execution times sorted in increasing order and task cost in decreasing order.

Time-Price Table

$t^1_{s\tau}$	$t^2_{s\tau}$	$\cdots$	$t^{n_m}_{s\tau}$
$p^1_{s\tau}$	$p^2_{s\tau}$	$\cdots$	$p^{n_m}_{s\tau}$

Table 3: The time-price table for a single task $\tau_{s\tau}$ contains time and price information for each M_u machine type, $0 < u \leq n_m$.

3.2.1 Stage Optimization

Since our algorithm is driven by the provided budget constraint, we can use its value to select valid machine types for a particular task. For instance, if given a budget $B_{s\tau}$ for a task $\tau_{s\tau}$, valid machines $u \in M_u$ are those in the time-price table whose cost is less than the budget: $p^u_{s\tau} < B_{s\tau}$. Using the same table, we can compute the shortest time to finish a task when given a budget by selecting the most expensive machine that costs less than the budget. We can compute this time as

$$T_{s\tau}(B_{s\tau}) = t^u_{s\tau} \qquad | \; p^{u-1}_{s\tau} > B_{s\tau} > p^{u+1}_{s\tau}. \tag{1}$$

Using this information, we can determine how to compute the execution time of a single stage S_s with respect to a given budget B_s. As all tasks in a stage must finish for the stage to complete, and since all tasks are independent within that stage, we can define the stage makespan as the maximum execution time of all tasks in the stage:

$$T_s(B_s) = \max_{0<\tau\leq n_s} \{T_{s\tau}(B_{s\tau})\}. \tag{2}$$

Consequently, to decrease the stage execution time we simply reschedule the task

with maximum execution time.

3.2.2 Workflow Optimization

To minimize the workflow makespan we need to determine a method to calculate the makespan value with respect to the budget constraint B. Given that the workflow makespan is equivalent to the execution time of the longest path, we can find which stages to optimize by determining the DAG's critical path. Since our workflow is arbitrary however, we must also consider that there may exist multiple critical paths. To find a single critical path we first modify the DAG to contain a single entry and exit node, after which a single-source longest-path algorithm can be run between these nodes to find the critical path. Overall, all steps in the process have at most linear computational complexity with respect to the input workflow DAG size. In both the following steps and throughout Chapter 4, we consider our workflow $G = (V, E)$ to consist of $|V| = k$ stages (nodes) and $|E|$ dependencies (edges).

We begin by modifying the arbitrary workflow DAG G to contain only a single entry and exit node. To do this we first find all entry and exit nodes in the workflow DAG. After locating these nodes, all existing entry (exit, respectively) nodes are then connected to a single new zero-cost entry (exit, respectively) node in the DAG. As stated in [19, 21, 22, 62, 73, 77], adding these new nodes does not affect schedule length. Finding all entry and exit nodes takes $|V|$ time, as all nodes must be visited to determine existence of dependencies or successors. Adding the new nodes and edges takes $O(1) + O(|E|)$ time in the worst case (assuming adjacency list storage).

Generally, graph edges are required to be weighted for execution of a shortest or longest path-finding algorithm. However in our case, for each edge $e(u, v)$, $u, v \in V$ we can simply use the weight of v as the edge weight $e_w(u, v) = T_v(B_v)$. As an edge stores the incoming (from) and outgoing (to) vertices, this retrieval takes $O(1)$ time for the lookup of the vertex along with its attribute.

Theorem 1 *Let $G = (V, E)$ be a node-weighted DAG, and let the weight for any node $u \in V$ be defined as u_w. Also assume that G contains a single entry node s and a single exit node t, where $s_w = 0$ and $t_w = 0$. The results of a deterministic shortest-path algorithm SP on G using node weights is equivalent to the results of the same algorithm run on G using edge weights after setting each edge weight $e(u, v),\ u, v \in V$ to v_w.*

Proof. Consider an execution of SP on $G = (V, E)$ from s to t using node weights. As SP is deterministic, the same shortest path is returned for any run of SP on G, even if there exist multiple shortest paths in G. We denote the returned shortest path as P, and its weight as P_w. Since the shortest path is from s to t, both weights $s_w = 0$ and $t_w = 0$ are included in P_w. Therefore, $P_w = \sum_{u \in P} u_w = \sum_{u \in P \setminus \{s,t\}} u_w$.

Consider now an execution of SP on G from s to t using edge weights, where the weight of $e(u, v) = v_w \ \forall\ u, v \in V$. We denote the returned shortest path as P', and its weight by P'_w. Since there does not exist an edge $e(u, s) \mid u \in V$, s_w is not included in P'_w. This not an issue, as s_w also does not contribute any weight to P_w. In P', there must be an edge $e(u, t) \mid u \in V$, as the shortest path runs from s to t. However, as the weight of $e(u, t) = 0$, its inclusion does not contribute to P'_w. For any other edge $e(u, v) \mid u, v \in V$ where $e(u, v) \in P'$, the algorithm must have traversed the path from $u \rightsquigarrow v$, and as such visits both $e(u, v)$ and v. Note that as the graph is a DAG, no other edge $e(x, v) \mid x, v \in V$ can be selected for inclusion in P' after the selection of $e(u, v)$. This ensures that v_w is not considered more than once via the weights of any edges $e(x, v)$.

Therefore, as selection of an edge $e(u, v)$ represents the traversal from $u \rightsquigarrow v$, consideration of either the assigned edge weight of $e(u, v)$ or the node weight v_w is equivalent. ■

At this point we can consider G equivalent to an edge-weighted DAG. The next step taken is to run a topological sort on G, which orders nodes such that each node's

dependencies occur before the node itself appears in the ordering. A topological ordering can be found in $O(|V| + |E|)$ time using a modified DFS, and as such has complexity linear in the size of the DAG [58]. Pseudocode for a topological sort is shown in Algorithm 1.

Algorithm 1 A topological sort.

```
 1: procedure TOPOLOGICAL(graph, vertex)
 2:
 3:     order = []
 4:     marked = [False] * len(graph.vertices)        ▷ Initialize vertices as unmarked.
 5:
 6:     procedure DFS(graph, vertex)                              ▷ Post-traversal DFS.
 7:         marked[vertex] = True
 8:         for edge in graph.outgoing(vertex) do
 9:             w = edge.to
10:             if not marked[w] then
11:                 DFS(graph, w)
12:             end if
13:         end for
14:         order.prepend(vertex);
15:     end procedure
16:
17:     DFS(graph, vertex)
18:
19:     return order
20: end procedure
```

To compute the path weight information a single-source longest-path algorithm is employed, which uses the DAG's topological ordering to run in linear time. The pseudocode for this algorithm is shown in Algorithm 2. The algorithm works by iteratively updating longest path information as it visits new nodes, with the topological ordering allowing all dependencies of a node to have path weight information computed before the node itself. Due to this, each node must only be visited once. Path weight information is updated for a node using the *relax* function, which throughout the algorithm is executed $|V|$ times, overall visiting $|E|$ edges. Therefore the total

time taken to relax all edges is $O(|V| + |E|)$, while initialization takes linear time for the topological sort, and $|V|$ time for variable initialization.

Algorithm 2 Critical path calculation.

```
 1: procedure CALCULATE_CRITICAL(graph)
 2:
 3:     distance_to = []                          ▷ Distance from source to vertices.
 4:     ordering = TOPOLOGICAL(graph)                                  ▷ O(|V| + |E|)
 5:     source = ordering[0]
 6:
 7:     for vertex in graph.vertices do                                    ▷ O(|V|)
 8:         distance_to[vertex] = −∞
 9:     end for
10:     distance_to[source] = 0
11:
12:     procedure RELAX(graph, v)                                ▷ O(|graph.outgoing(v)|)
13:         for edge in graph.outgoing(v) do
14:             w = edge.to
15:             if distance_to[w] < distance_to[v] + edge.weight then
16:                 distance_to[w] = distance_to[v] + edge.weight
17:             end if
18:         end for
19:     end procedure
20:
21:     for vertex in ordering do                                      ▷ O(|V| + |E|)
22:         RELAX(graph, vertex)
23:     end for
24:
25:     return distance_to
26: end procedure
```

To prove a linear running time for the algorithm, we consider an argument by contradiction regarding the number of calls to *relax*. Assume that there exists an execution of our algorithm in which there are more than $|V|$ calls to the *relax* function. As a result, there must exist a node $v \in V$ that is relaxed more than once. Since nodes are relaxed to update their weight according to maximum path distance among their dependencies, this must mean that v's distance from the entry node s must have been updated more than once due to one of its dependencies having its weighting updated.

Otherwise, v would already have a correct weight, and would not have needed to be updated again after the first relaxation. This implies that v was relaxed before one of its dependencies. However, as the topological sort returns nodes in an ordering where all dependencies occur before their successors, it is impossible for any dependency of v to be relaxed after v. As such, v must have a correct weighting after its first relaxation, and therefore is only relaxed once.

Algorithm 3 Find stages on the critical path.

```
1:  procedure GET_CRITICAL_STAGES(graph, sink, distance_to)
2:      vertices = [sink]
3:      stages = {}
4:
5:      while len(vertices) > 0 do                          ▷ O(|V| + |E|)
6:          vertex = vertices.pop();
7:          next_vertices = []
8:          max = 0;
9:
10:         for edge in graph.incoming(vertex) do
11:             w = edge.from
12:
13:             if max < distance_to[w] then
14:                 max = distance_to[w]
15:                 next_vertices.clear()
16:             end if
17:
18:             if max ≤ distance_to[w] and not next_vertices.contains(w) then
19:                 next_vertices.append(w)
20:             end if
21:         end for
22:
23:         stages.append(next_vertices)
24:         vertices.append(next_vertices)
25:     end while
26:
27:     return stages
28: end procedure
```

After path weight information is computed, we can now determine which stages

lie on the critical path(s). This process is displayed in Algorithm 3. Beginning from the exit node, or *sink*, we traverse back along the critical paths using a modified BFS. More specifically, edges traversed are selected by considering only the node(s) of maximum value among all predecessors. These traversed nodes (stages) are added as they are found to a set of critical stages, which are returned when the function completes. The algorithm runs in $O(|V| + |E|)$ time in the worst case; when all nodes lie on the critical path(s), this forces all vertices to be visited by traversal along all edges. For instance, consider the nodes which can be added to the *vertices* set. These are selected as the predecessors of the current nodes in *vertices*. Since the graph is acyclic, no node can be added to *vertices* more than once. Similarly, since only incoming edges to a node are traversed, no edge can be traversed more than once. Therefore, Algorithm 3 has a linear time complexity of $O(|V| + |E|)$.

Chapter 4

Budget-Driven Algorithms

We propose two main algorithms based on the work in [66] that have been adapted to run on arbitrary workflow DAGs. The first is a globally optimal algorithm used as a benchmark to compare schedule creation times and efficacy during testing. The second is a heuristic that iteratively reschedules tasks from stages on the workflow's critical path(s). For the heuristic, selection of tasks for rescheduling is based on a utility value that accounts for changes in both the workflow cost and makespan caused by the reschedule. The principal goal for our algorithms is to efficiently distribute budget over the individual stages (and tasks), such that the makespan of the workflow is minimized while the total cost is within the given budget constraint.

These algorithms are presented in the following sections, and exploit previously introduced algorithms and data structures that have been modified. For the input workflow DAG G, task execution information is recorded as attributes on tasks themselves. As such, we denote the list of tasks in the workflow as $G.\tau$. Indexing into this list allows the price, time, and machine attributes to be set and retrieved. For instance, price information for a task Γ (or an index i) can be set by the statement $G.\tau_{\Gamma}.p = price$ (or $G.\tau_i.p = price$). Additionally, we also assume that the $G.\tau_{exit}$ attribute contains the workflow's exit stage. As we add the exit stage to the workflow

ourselves, this attribute is trivial to define. Also used in the algorithms is the time-price table, denoted by TP. The table is accessed to select either a p or t function for execution, which return the price or time, respectively. Similar to the original time-price table introduced in Section 3.2, the p and t functions require as input a task τ and a machine M_u.

4.1 Optimal Scheduling

For the globally optimal scheduling algorithm proposed in [66], the authors use dynamic programming to compute the set of task-resource pairings. Their algorithm uses the equation

$$T(s,r) = \begin{cases} \min_{0<q\leq r}\{T_s(n_s,q) + T(s+1, r-q)\}, & \text{if } s < k. \\ T_s(n_s, r), & \text{if } s = k. \end{cases} \tag{3}$$

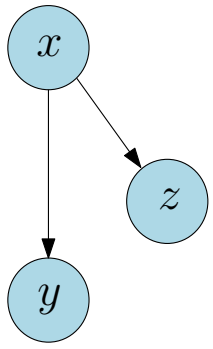
as a guide, where r is the given budget constraint, and s represents a stage $0 < s \leq k$. For this equation, $T_s(n_s, r)$ is the method used to optimize makespan of a single stage, similar to the process we have outlined in Section 3.2.1. The equation considers all stages equally for minimization of execution time, and as such is incorrect when considering arbitrary workflow DAGs as not all stages may influence the workflow makespan.

Consider, for example, the workflow represented in Figure 15. It introduces a three-stage workflow in (b), wherein stages are represented by a single task whose identifier is denoted on the node. Displayed in (a) are time-price tables for the tasks in the workflow: x, y, and z. If we consider executing the dynamic programming method on this workflow using a budget constraint of 11, there are only three pairings which are valid (the shaded rows of the table in (c)). As these pairings are inspected by

Time-Price Tables

Task	Attribute	Resource m_1	Resource m_2
x	time	8	2
	price	4	9
y	time	8	7
	price	3	5
z	time	6	4
	price	2	3

(a)



(b)

Task-Resource Pairings

Task Pairing x	Task Pairing y	Task Pairing z	Attribute time	Attribute price
m_1	m_1	m_1	22	9
m_2	m_1	m_1	16	14
m_1	m_2	m_1	21	11
m_2	m_2	m_1	15	16
m_1	m_1	m_2	20	10
m_2	m_1	m_2	14	15
m_1	m_2	m_2	19	12
m_2	m_2	m_2	13	17

(c)

Figure 15: In (a), the time-price tables are shown for the workflow tasks in (b). Part (c) shows the possible pairings of tasks to resources along with the resultant workflow time and price. Shaded rows correspond to valid schedules using a budget of 11.

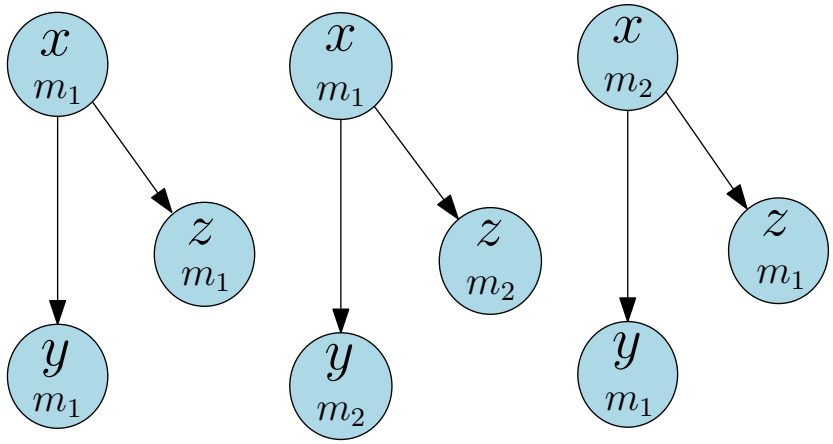
the algorithm, it selects from them the permutation with minimal time that satisfies the price constraint. The selected pairing is thus $\{x : m_1, y : m_1, z : m_2\}$, where the non-critical task z is rescheduled to decrease its makespan. This pairing however leaves the total workflow makespan unchanged, unlike the $\{x : m_1, y : m_2, z : m_1\}$ pairing that reduces the workflow makespan.

From these results, it may seem that a correct approach would be to reschedule tasks through selection of a stage on the critical path. This approach would indeed allocate budget to tasks affecting the makespan, and would therefore also reduce the workflow makespan at every iteration. Using this method, we consider another example visualized in Figure 16, where we assume a budget of 12 units. After an initial scheduling of all tasks on the least expensive resource type, shown in (b), the most cost-efficient stage on the critical path $x \rightarrow y$ is selected for rescheduling. Using

Time-Price Tables

Task	Attribute	Resource	
		m_1	m_2
x	time	4	1
	price	2	7
y	time	7	5
	price	2	4
z	time	6	3
	price	2	6

(a)



(b) **(c)** **(d)**

Figure 16: Given the time-price tables in (a) for the initial workflow in (b), a greedy critical-path based scheduling would pair y and z to resource type m_2, using a budget of 12 to reduce makespan from 17 to 9 in part (c). However, in (d), pairing x with m_2 instead uses a budget of 11 and reduces makespan to 8 units.

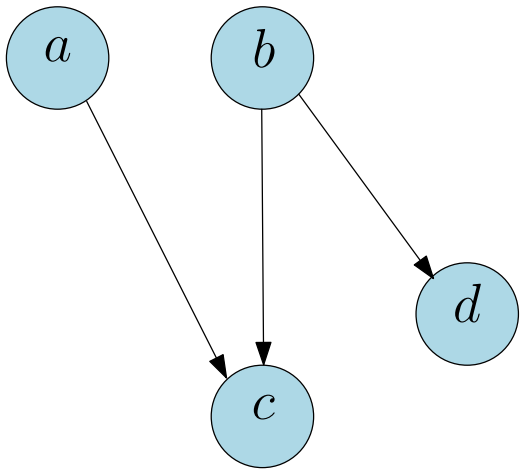
this common greedy strategy, y is selected during the first iteration as its unit-cost to decrease makespan is 1, which is better than both z's $1.\overline{33}$, and x's $1.\overline{66}$. During the second iteration the critical path changes to $x \rightarrow z$, and as a result the algorithm selects z for rescheduling. After these first two iterations both y and z are rescheduled from m_1 to m_2, as shown in (c). At this point, a total budget of 12 units has been used to achieve a makespan of 9. Consider now an alternate scheduling, shown in part (d), where x was rescheduled from m_1 to m_2 instead of y and z. In this case the schedule requires a budget of 11 units, and results in a makespan of 8 units. Clearly, the latter schedule is measurably better than the former, as both the budget and makespan have been reduced. As such, we see that the described greedy method is not optimal.

Investigating a different method for rescheduling tasks based on critical paths, we consider the outcome of prioritizing stages with the most successors. Intuitively, this method would be beneficial as a stage with more successors would have a higher

Time-Price Tables

Task	Attribute	Resource m_1	Resource m_2
a	time	2	1
	price	4	5
b	time	2	1
	price	4	5
c	time	5	3
	price	2	3
d	time	4	3
	price	1	2

(a)



(b)

Figure 17: A workflow DAG is shown in (b), with the time-price tables for its tasks presented in (a).

likelihood of occurring on multiple critical paths, either during the current iteration of the algorithm, or in the future. We apply this method to the example shown in Figure 17, assuming a budget of 12 units. After an initial scheduling on the least expensive resource type, there is one unit of budget left, which could be applied to any one stage in the workflow. As both $a \rightarrow c$ and $b \rightarrow c$ are critical paths, one of these stages are selected. If the stage with the most successors is prioritized, b is selected, which is suboptimal as the selection of c would result in a lower makespan.

With no simple method to determine the correct tasks for optimization, we instead resort to the use of a brute-force algorithm to check all permutations of task-resource mappings. Algorithm 4 details this approach, and uses the method introduced in Section 3.2.2 to compute workflow makespan in linear time for each possible permutation of task-resource mappings.

After receiving as input the workflow DAG G, the list of machines M, the time-price table TP, and budget B, the algorithm begins by declaring the local variable

Algorithm 4 A globally optimal, brute-force scheduling algorithm.

```
1: procedure SCHEDULE_OPTIMAL(G, M, TP, B)
2:
3:     μ = ∞                                              ▷ Set initial makespan.
4:
5:     ▷ Get all sets of repeated permutations of length n_τ over all resource types.
6:     for π ∈ REPEATED_PERMUTATIONS(M, G.n_τ) do                    ▷ O(n_m^{n_τ})
7:         cost = 0
8:
9:         ▷ Update the price-time mappings.
10:        for i ∈ range(π) do                                        ▷ O(n_τ)
11:            G.τ_i.t = TP.t(G.τ_i, π_i)
12:            G.τ_i.p = TP.p(G.τ_i, π_i)
13:            cost += G.τ_i.t
14:        end for
15:
16:        if cost > B then          ▷ Skip current permutation if cost is too high.
17:            continue
18:        end if
19:
20:        UPDATE_STAGE_TIMES(G)                          ▷ O(|V| + |E| + n_τ)
21:        distances = CALCULATE_CRITICAL(G)                  ▷ O(|V| + |E|)
22:
23:        if distances[G.τ_exit] < μ then
24:            μ = distances[G.τ_exit]
25:        end if
26:    end for
27:
28:    if μ = ∞ then
29:        return None
30:    else
31:        return G
32:    end if
33: end procedure
```

μ to keep track of the best schedule seen so far. After this initialization, a function is called to generate all permutations of task-resource pairings, with replacement. In the outer for loop the cost and makespan of each task-resource mapping is computed, and then compared to the best schedule seen so far. The inner for loop begins this process by updating the time and price of each task in the workflow according to the current permutation being considered. As well, the total cost of the current task-resource mapping is tracked in order to determine if the budget constraints satisfied. This condition is checked explicitly after the inner for loop. After setting the tasks to run on their respective resources, the stage times are updated. The function called uses a DFS - similar to the nested procedure in Algorithm 1 - to visit each stage in the workflow. For each stage, the function iterates through the contained tasks to determine the task with the longest execution time. This time is then set as the weighting for the stage, and will be used later during critical path calculation. The algorithm then calls Algorithm 2 to obtain the workflow's critical path information. Now that the critical path information for the current task-resource mapping has been found, it is compared to the best schedule seen so far. After comparing all task-resource pairs, the schedule information is then returned via the set attributes on G.

Theorem 2 *The running time of Algorithm 4 is* $O((|V| + |E| + n_\tau) \times n_m^{n_\tau})$.

Proof. The algorithm contains an outer for loop that iterates through a set of machine-task permutations, computing the scheduling cost for each permutation. We define a machine-task permutation as a mapping of machine types onto workflow tasks. As we know that there exist a total of n_m machine types, we can derive that each task yields n_m possible schedulings; one for each machine type. Since there are a total of n_τ tasks, with each task allowing n_m possibilities, there exist a total of $n_m^{n_\tau}$ permutations. By defining an ordering on the machine types we can simply 'count'

up through the permutations, as each number corresponds to a unique permutation, and each digit a machine type. Thus, the generation of machine-task permutations takes $O(n_m^{n_\tau})$ time.

The rest of the computation is done within this outer loop, and works to update the time-price mappings of tasks, update stage times, and then to finally calculate a critical path.

To update the task times and prices, each of the n_τ tasks must be visited. For each task both of these values can be computed in $O(1)$ time by use of the time-price table. The task identifier along with its current machine type are simply provided to the table, which maps these values to their time and price values. The total cost is also incremented while visiting each task, which takes $O(1)$ time. Therefore, updating task times and prices for a single machine-task permutation takes $O(n_\tau)$ time.

The *UPDATE_STAGE_TIMES* function is simply a modified DFS algorithm, and as such initially runs in time proportional to the workflow DAG's size. As stated in Section 3.2.2, we consider our workflow $G = (V, E)$ to consist of $|V| = k$ stages (nodes) and $|E|$ dependencies (edges). Thus, the DFS algorithm initially takes $O(|V| + |E|)$ time to execute. As modifications, for each stage the algorithm uses task time information to compute the maximum execution time for the stage. In total, the execution time of all tasks must be retrieved, which takes $O(n_\tau)$ time. Therefore, $O(|V| + |E| + n_\tau)$ time is used to update stage execution times.

Lastly, Algorithm 2 is used to compute the critical path for the current machine-task permutation. As shown in Section 3.2.2, this takes $O(|V| + |E|)$ time.

Overall, the algorithm takes $O((|V|+|E|+n_\tau) \times n_m^{n_\tau})$ time, as time-price mapping, stage time updating, and critical path calculation must occur for each of the $n_m^{n_\tau}$ machine-task permutations. ■

Though the algorithm has a very large runtime, it does explicitly check every single possible task-resource mapping. As a result it is guaranteed to determine the

schedule of minimum makespan that also satisfies the imposed budget constraints, and is therefore correct.

4.2 Greedy Scheduling

In Section 4.1, we began by considering the applicability of the globally optimal algorithm proposed in [66] to our modified problem. Following this, we also considered several other possible methods for computation of an optimal scheduling. Due to the problem complexity however, our method resorted to the use of a brute-force algorithm to generate an optimal workflow schedule. In this section, we consider the application of a greedy scheduling heuristic. Similar methods have been approached in the literature, though none use arbitrary workflows with the Apache Hadoop framework [19, 47, 66, 75].

Recall the initial greedy method mentioned in Section 4.1. Scheduling begins with an initial assignment of all tasks to the least expensive resource type and is followed by rescheduling of select stages on the critical path, based on prospective changes to makespan and cost. The initial assignment simply ensures that the given budget is valid for the input workflow, along with establishing a base configuration to build from. After this step the algorithm proceeds in an iterative manner, selecting a stage on the critical path to have its slowest task rescheduled. This rescheduling has the potential to alter the workflow's critical paths, and as such recomputation of the critical path is necessary. Stage selection is based on the relative improvement of schedule makespan with respect to cost increase, compared against other eligible stages (those on the critical path). As mentioned the rescheduling is done iteratively, and executes until there is either insufficient budget left for rescheduling, or until there are no tasks left that can be rescheduled. The pseudocode for this algorithm is shown in Algorithm 5, with an explanation forthwith.

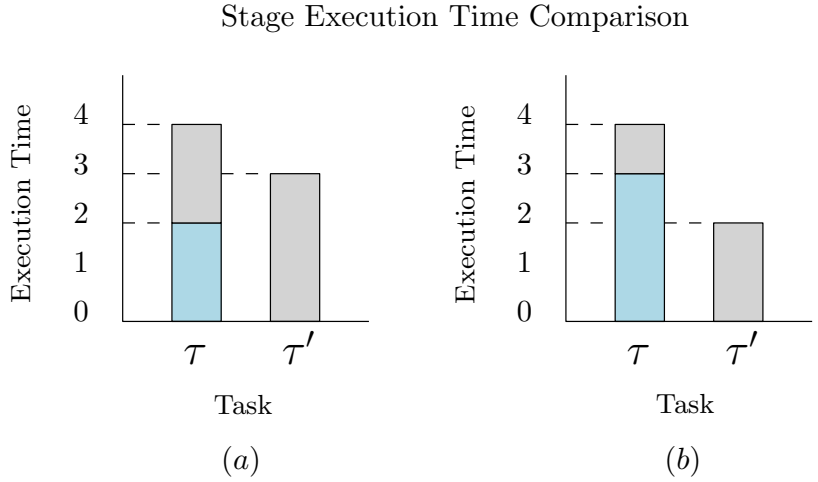


Figure 18: Task execution times before rescheduling are shown in grey, with shorter, rescheduled execution times shown in blue. In (a), rescheduling τ allows τ' to become the slowest task. However it is possible, as in (b), for τ to continue to be the slowest task in a stage.

To direct stage selection, we denote the *utility* of a task τ in stage s as $v_{s\tau}$, calculated by

$$v_{s\tau} = \frac{\min\left\{(t^u_{s\tau} - t^{u-1}_{s\tau}), (t^u_{s\tau} - t^{u'}_{s\tau'})\right\}}{p^{u-1}_{s\tau} - p^u_{s\tau}} \tag{4}$$

if there exists more than one task in a stage. Otherwise, *utility* is given by

$$v_{s\tau} = \frac{t^u_{s\tau} - t^{u-1}_{s\tau}}{p^{u-1}_{s\tau} - p^u_{s\tau}}. \tag{5}$$

In Equation 4, the slowest task τ is assumed to be currently assigned to the machine u, whereas the second-slowest task τ' is assigned to u'. Also recall that the time-price table is sorted, with times in increasing order and cost in decreasing order. The meaning of this equation is visualized in Figure 18, which shows a bar chart representing the makespan of tasks in a single stage. For example, consider the difference between part (a) and (b). In (a), rescheduling the slowest task causes the bottleneck to change to the second-slowest task. However in (b), rescheduling the slowest task does not change which task is limiting the stage makespan. It is because

of these two separate cases that the minimum execution time decrease between the slowest and second-slowest tasks is used by the equation, as this allows realization of the actual time speedup caused by rescheduling.

Similar to Algorithm 4, Algorithm 5 accepts as arguments the workflow DAG G, set of machines M, time-price table TP, and a budget B. It begins by performing an initial scheduling in the for loop on Line 3. This initialization assigns the least expensive machine type to each task in the workflow, along with updating the time and price information for the task. As this task-resource mapping is being performed, the schedule cost is also recorded so that it can be reconciled against the input budget B later in the algorithm, on Line 10.

The main while loop begins on Line 13, where it repeatedly reschedules tasks until no budget remains. Within the main execution loop several functions are first called to configure the graph weightings, and then to calculate the critical stages. The first of these functions updates stage weights to be the maximum weight of their composite tasks (as described in Algorithm 4). In addition to updating stage weight information, the *UPDATE_STAGE_TIMES* function is also modified in this case to record for each stage the slowest task and second-slowest task. The following two functions serve to calculate the critical stages, first by determining makespan information, and then using the computed information to ascertain which stages lie on the critical path(s). Following computation of the critical stages, utility values are computed in the for loop beginning on Line 18. In this for loop the slowest and second-slowest tasks are retrieved for each stage, and then are used in conjunction with the time-price table to calculate each stage's utility.

At this point the while loop on Line 22 is entered, where the critical stages' utility values are iterated through, as orderd by descending utility values. For each utility value the corresponding task Γ is retrieved and analyzed. Since the utility value only ensures that the most cost-efficient task is selected, it is still possible that the

Algorithm 5 A greedy scheduling algorithm.

1: **procedure** SCHEDULE_HEURISTIC(G, M, TP, B)
2: $cost = 0$
3: **for** $i \in G.n_\tau$ **do** ▷ $O(n_\tau)$
4: $G.\tau_i.m = M_{n_m - 1}$
5: $G.\tau_i.p = TP.p(G.\tau_i, G.\tau_i.m)$
6: $G.\tau_i.t = TP.t(G.\tau_i, G.\tau_i.m)$
7: $cost \mathrel{+}= G.\tau_i.t$
8: **end for**
9: $B \mathrel{-}= cost$
10: **if** $B < 0$ **then**
11: **return** $None$
12: **end if**
13: **while** $B \geq 0$ **do** ▷ $O(n_\tau \times n_m)$
14: UPDATE_STAGE_TIMES(G) ▷ $O(|V| + |E| + n_\tau)$
15: $distances$ = CALCULATE_CRITICAL(G) ▷ $O(|V| + |E|)$
16: $S_{critical}$ = GET_CRITICAL_STAGES($G, G.\tau_{exit}, distances$) ▷ $O(|V| + |E|)$
17: $v = \varnothing$
18: **for** $S_s \in S_{critical}$ **do** ▷ $O(|V|)$
19: $(\Gamma, \gamma) = (S_s.slowest, S_s.second\_slowest)$
20: $v[\Gamma] = \frac{\min\{(TP.t(\Gamma, G.\tau_\Gamma.m) - TP.t(\Gamma, G.\tau_\Gamma.m-1)), (TP.t(\Gamma, G.\tau_\Gamma.m) - TP.t(\gamma, G.\tau_\gamma.m))\}}{TP.p(\Gamma, G.\tau_\Gamma.m-1) - TP.p(\Gamma, G.\tau_\Gamma.m)}$
21: **end for**
22: **while** $v \neq \varnothing$ **do** ▷ $O(|V| \log |V|)$
23: $\Gamma = \max_{\tau \in v.keys()}\{v[\tau]\}$
24: $new\_price = TP.p(\Gamma, G.\tau_\Gamma.m - 1)$
25: $old\_price = TP.p(\Gamma, G.\tau_\Gamma.m)$
26: **if** $B < (new\_price - old\_price)$ **then**
27: $v.remove(\Gamma)$
28: **else**
29: $G.\tau_\Gamma.m \mathrel{-}= 1$
30: $G.\tau_\Gamma.p = TP.p(\Gamma, G.\tau_\Gamma.m)$
31: $G.\tau_\Gamma.t = TP.t(\Gamma, G.\tau_\Gamma.m)$
32: $B \mathrel{-}= (new\_price - old\_price)$
33: **break**
34: **end if**
35: **end while**
36: **if** $v = \varnothing$ **then**
37: **return** G
38: **end if**
39: **end while**
40: **end procedure**

magnitude of the cost is greater than allowed. Thus, the change in price caused by rescheduling is computed and compared against the available budget. If the budget is insufficient then the task is discarded while the algorithm continues on to the next task. Otherwise, we reschedule the task onto a more powerful machine, update the related attributes, and break out of the inner loop to allow critical path information to be recomputed. The algorithm exits based upon the condition on Line 36; if no critical stages can be rescheduled then the workflow makespan cannot be decreased any further, and so the algorithm exits.

Theorem 3 *The running time of Algorithm 5 is* $O(n_\tau + (n_\tau \times n_m) \times (|V| \log |V| + |V| + |E| + n_\tau))$.

Proof. The algorithm begins with an initialization loop on Line 3. The loop runs once for each of the n_τ tasks in the workflow DAG $G = (V, E)$. In each iteration it does a constant amount of work to initialize the machine type, price, and time of each task. As well, an initial cost is also calculated. As a constant amount of work is done in each of the n_τ loop iterations, the initialization stage takes $O(n_\tau)$ time.

Next, we consider the execution time of the main while loop on Line 13. As a bound on the running time, we know that the maximum number of times the loop can run is related to the number of times rescheduling can occur, as this is also the maximum number of times that the budget can be updated. Specifically, since each of the n_τ tasks can be rescheduled $n_m - 1$ times, the loop must execute less than $n_\tau \times (n_m - 1)$ times. Therefore, the outer loop executes at most $O(n_\tau \times n_m)$ times.

The first computation within the outer while loop updates the stage times. This is accomplished by the *UPDATE_STAGE_TIMES* function, whose run time was derived as $O(|V| + |E| + n_\tau)$ in the proof of Theorem 2. In this instance however, it has been modified to record both the slowest and second slowest tasks of each stage. To find the these tasks for a particular stage, the algorithm needs to iterate over all tasks

belonging to the stage. However, since all tasks are already visited for computation of the stage's execution time, this addition only adds a constant amount of work for each task visited, leaving the total execution time unmodified. As such, the run time of the algorithm is $O(|V| + |E| + n_\tau)$.

After the stage times are updated the critical path information is computed and then retrieved. These operations are achieved using Algorithm 2 and Algorithm 3, respectively. As shown in Section 3.2.2, both algorithms take $O(|V| + |E|)$ time.

To place an upper bound on the inner for loop, we realize that as the workflow DAG is arbitrary there is no guarantee on the number of stages which comprise the critical path(s). As a result, all stages can in fact lie on the critical path, causing the loop to execute $|V|$ times in the worst case. For each iteration of this loop, stage utilities are computed in constant time. Therefore, the inner for loop takes at most $O(|V|)$ time to execute in the worst case.

The inner while loop can execute at most $|V|$ times, assuming that for each stage the cost for rescheduling is found to be larger than the available budget. Within the loop at each iteration, we first retrieve the task with maximum utility, and then compare its current and new execution prices using information from the time-price table. The lookup of information from the time-price table along with all subsequent operations take $O(1)$ time. To obtain tasks ordered by their utilities, we use a priority queue structure to allow retrieval in $O(1)$ time, albeit with an overhead of $\log |V|$ time for initialization in the worst case. However, this is much more efficient than a linear search, which would take $O(|V|)$ time per iteration. Therefore, the execution time of the inner while loop is at most $O(|V| \log |V|)$.

Overall, the algorithm's time is therefore $O(n_\tau + (n_\tau \times n_m) \times (|V| \log |V| + |V| + |E| + n_\tau))$. Simplifying this, we can both remove the non-dominant terms and assume that the number of machines n_m is bounded by a small integer constant (4 in our experiments) to give a resultant time of $O(n_\tau \times (|V| \log |V| + |E| + n_\tau))$. ■

Chapter 5

Implementation

5.1 Introduction

Implementation of the proposed algorithm was carried out in Hadoop version 1.2.1. It includes the addition of several packages, with around 8500 lines of code added in total. This included creation of the *org.apache.hadoop.mapred.workflow*, *org.apache.hadoop.mapred.workflow.schedulers*, and *org.apache.hadoop.mapred.workflow.scheduling* packages. Several existing classes were also modified to allow workflow execution. In this case, the main modifications were made to the *org.apache.hadoop.mapred.core.util.RunJar*, *org.apache.hadoop.mapred.JobTracker*, and *org.apache.hadoop.mapred.JobClient* classes. For testing of the implementation, classes were also added to the new packages *org.apache.hadoop.workflow.examples* and *org.apache.hadoop.workflow.examples.jobs*. The changes made along with the method of execution are reviewed in detail in this section.

Recall that the architecture in the 1.x series of the Hadoop MapReduce framework is split into two main parts (for our purposes we can ignore the components which comprise the Hadoop Distributed FileSystem (HDFS)): the JobTracker, and the TaskTrackers. An overview of the architecture is shown in Figure 19. In this

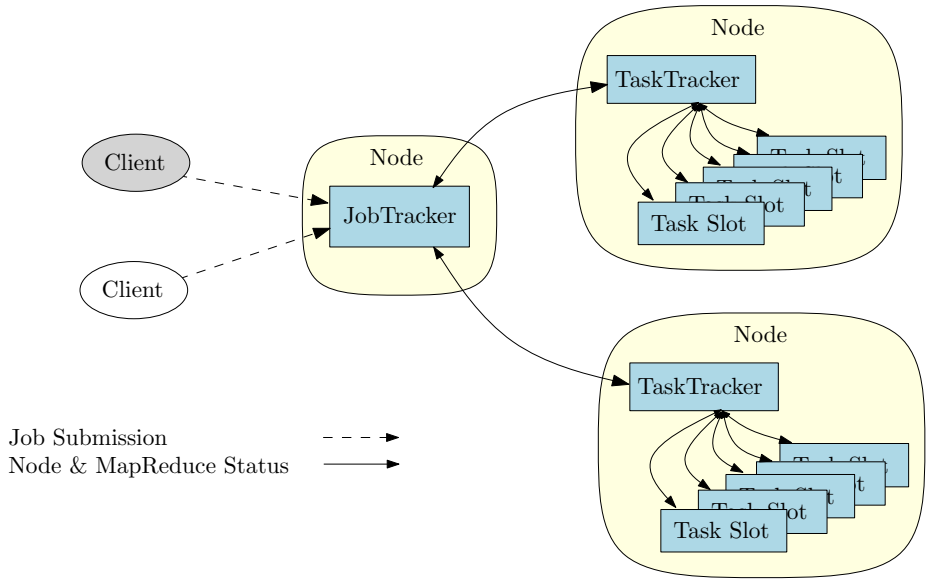


Figure 19: An overview of the Apache Hadoop MapReduce architecture, for Hadoop versions 1.x. Each client node (*TaskTracker*) contains a number of task slots that can run either map or reduce tasks.

layout, jobs are submitted from a local client machine, which then relay the job submission request to the main server node - the *JobTracker*. The *JobTracker* handles initialization of the MapReduce framework itself, creating the appropriate scheduler and data structures that store all job-related information. This includes the main code structures for job submission and lifecycle management. The architecture itself follows a client-server pattern, with the *JobTracker* acting as the server and *TaskTrackers* as clients.

As a brief overview of the execution flow during job scheduling, we consider the result of a single heartbeat message sent from a *TaskTracker* node to the *JobTracker*. These heartbeat messages are sent repeatedly after framework initialization, and allow regular communication with the *JobTracker* in order to synchronize status information, and for task assignment. While handling a message, the *JobTracker* delegates task assignment to the current task scheduler. The scheduler itself operates alongside

the *JobTracker* on the server node, and is constructed via reflection from a configuration property during startup. When initialized the task scheduler creates listeners, which it then adds to the *JobTracker*. It is these listeners that the *JobTracker* and task scheduler use to control scheduling, as they notified when jobs are added, changed, or removed. As a result, when the *JobTracker* calls the scheduler to assign tasks to the *TaskTracker* the currently submitted jobs can be retrieved from a listener and used to determine which and how many tasks should be executed.

The implementation of workflow scheduling attempts to take advantage of the current job scheduling features as much as possible. As a result, it follows the same basic pattern as job scheduling. Modifications include a customized listener which listens for both job events and workflow events (addition, modification, removal), and the scheduler itself which launches executable workflow jobs as well as assigning tasks to querying *TaskTracker*s. Due to the complexity of workflow scheduling, our assumptions of known data, and the constraint requirements, much of the scheduling decisions are delegated to an additional *SchedulingPlan* class. The class is instantiated on the client machine during workflow submission, where it generates the plan based on cluster properties. The plan is then passed to the scheduler (via the *JobTracker*) during workflow submission, and used to decide how to run both workflow jobs and their composite tasks.

Throughout the rest of this chapter we review both job and workflow execution in more detail. Additionally, common methods used among the implemented scheduling plans (algorithms) are examined. Specifically, Section 5.2 covers the execution flow of job submission, Section 5.3 the execution flow of workflow submission, and Section 5.4 the scheduling plan implementations.

5.2 Job Submission Execution Flow

Typical job submission begins with execution of the Hadoop script on the job's jar file, for example `hadoop jar hadoop-examples.jar org.apache.hadoop.examples.WordCount /input /output` to execute the *WordCount* job in `hadoop-examples.jar`, with the arguments `/input` and `/output` specifying the input and output paths to the job in HDFS respectively. The `jar` argument of this command determines the entry point for execution, selecting the *RunJar* class to begin execution of the provided job jar file. It unpacks the jar file, creates a classpath from the contents, and then invokes the main class via reflection. Arguments passed to the job are then parsed in the job's main method, which is followed by definition and configuration of a job configuration object (*JobConf*). After this configuration, the job is submitted to the *JobClient* using the constructed *JobConf*. The *JobClient* then prepares the job for submission to the *JobTracker*. It retrieves an identifier for the job, sets up its staging area in HDFS, copies any relevant files into HDFS so that they will be replicated and available across the *TaskTrackers*, and then submits the job to the *JobTracker*. When the *JobTracker* receives the job submission it creates several structures to track the status information of the job, and then adds the job through notification of any listeners (*JobInProgressListener*). The listeners keep a queue of jobs that are currently in progress, enabling access to the *TaskScheduler* object for scheduling. Throughout this computation, all *TaskTracker*s repeatedly send messages to the *JobTracker* via a heartbeat method. The method notifies the *JobTracker* that the given *TaskTracker* wishes to have its status information updated. Additionally, *TaskTracker*s can request to be assigned new tasks. If a *TaskTracker* is requesting task assignment, the *JobTracker* delegates this operation to the *TaskScheduler*, which iterates through the jobs currently in progress to locate those that have tasks waiting to be assigned. These jobs are

located based on the *TaskScheduler*'s concrete class, as loaded through reflection when the framework is started. After finding a job with tasks to run, the tasks are created, and passed back to the *JobTracker*, and then to the *TaskTracker*. As the process of task execution continues, the *JobTracker* keeps all data structures up to date with their current status. Status information is then gathered from the data structures and used by the *JobClient* through an adapter class (*NetworkedJob*), which relays the information to the user who submitted the job.

5.3 Workflow Submission Execution Flow

The execution flow for a workflow follows the same overall process as that of job execution. The main differences in execution occur during scheduling, and during workflow job submission due to the addition of identification information. An overview of the workflow execution lifecycle is shown in Figure 20, and is explained throughout this section.

When scheduling a workflow with the implemented algorithms, several additional requirements exist. To begin with, there exists a *WorkflowSchedulingPlan* class to be implemented in addition to a *TaskScheduler*. Both are specified in Hadoop's `mapred-site.xml` configuration file: the task scheduler under the normal *mapred.jobtracker.taskScheduler* property, and the scheduling plan under a *mapred.workflow.schedulingPlan* property. Regarding implementation, the task scheduler implements a *WorkflowScheduler* interface in addition to extension of the *TaskScheduler* class, while the scheduling plan extends a *WorkflowSchedulingPlan* class. For our implementations, the schedulers all included a workflow listener, which both implements a *WorkflowInProgressListener* interface and extends the *JobInProgressListener* class. In this way, the implemented schedulers can both decide how to schedule tasks, and how to execute workflow jobs in relation to each other.

In addition to this configuration data, two XML files are required for the proposed algorithms. The first contains a list which identifies the types of machines available in the cluster. It specifies for each machine a unique name, its attributes (hard disk space, memory, number of CPU's and their frequency), and the hourly cost to run the machine. The second file contains information on job execution times. Specifically, an entry exists for each job - identified by its unique name - which contains the execution time for a single map and reduce task on each machine type. We can assume that map and reduce inputs are split more or less evenly, and therefore it is valid to consider that all tasks of a specific type have the same execution time. These two files are used together by the *WorkflowClient* to create the time-price table.

Similar to job submission, workflow submission begins with execution of the Hadoop script on the workflow's jar file. This is accomplished using the same format as for jobs: `hadoop jar workflow.jar org.apache.hadoop.workflow.examples.Sipht /input /output`. As with jobs, the flow of execution travels through the *RunJar* class and back into the main method of the jar file. As opposed to configuration of a *JobConf* object however, a *WorkflowConf* is created and configured. The *WorkflowConf* provides methods for budget or deadline constraints to be set, jobs to be added (through specification of a unique name, jar file, main class, optional command-line arguments, number of map & reduce tasks), and for dependencies to be created between them. Entry jobs are also able to have an alternate input directory set which overrides the input path supplied to the workflow. After the workflow is configured, it is submitted to a *WorkflowClient* using the constructed *WorkflowConf*. Similar to the *JobClient*, the *WorkflowClient* prepares the workflow for submission to the *JobTracker*. It retrieves an identifier for the workflow, sets up its staging area on HDFS, retrieves cluster status information, loads machine type and job execution time information to create a time-price table, updates the information of jobs contained in the workflow, and

then runs the workflow's scheduling plan before submission to the *JobTracker*.

To update the workflow jobs, the *WorkflowClient* sets their workflow (*WorkflowID*) and job (*JobID*) identifiers, as well as setting their input and output directories according to dependency information. For example, entry jobs have their input set to the workflow's input directory, and exit jobs have their output directory set to the workflow's output directory. Otherwise, all jobs have their own output directory while input is set to the list of all predecessor's output directories. Following these updates the scheduling plan is provided with cluster and machine information and run to generate a valid schedule. Note that during scheduling the plan is kept synchronized with workflow progress, and as such the number of map and reduce tasks set in the *WorkflowConf* for each workflow job must be correct. Lastly, submission to the *JobTracker* occurs.

During submission to the *JobTracker*, server-side data structures are created to track workflow execution. Additionally, the workflow is added to any existing *WorkflowListener*s to actually begin workflow execution. It is at this point that workflow scheduling occurs. As shown in Figure 21, workflow scheduling is mainly carried out through the cooperation of a *WorkflowTaskScheduler*, *WorkflowListener* (a concrete implementation of a *WorkflowInProgressListener*), and a *WorkflowSchedulingPlan*. It should also be noted that the provided *WorkflowTaskScheduler* need not be used; both it and a custom *WorkflowSchedulingPlan* can be provided together to the Hadoop framework for customized workflow scheduling. Similar to job execution, the implemented *WorklfowListener* keeps a queue of workflows that are currently in progress, and enables access to the *WorkflowTaskScheduler* object to carry out scheduling. In this case, the listener keeps a queue of both job (*JobInProgress*) and workflow (*WorkflowInProgress*) objects. The *WorkflowTaskScheduler* then extends both the *TaskScheduler* and *WorkflowScheduler* protocols. This enables the *WorkflowTaskScheduler* to handle scheduling of the whole workflow, both at the task level,

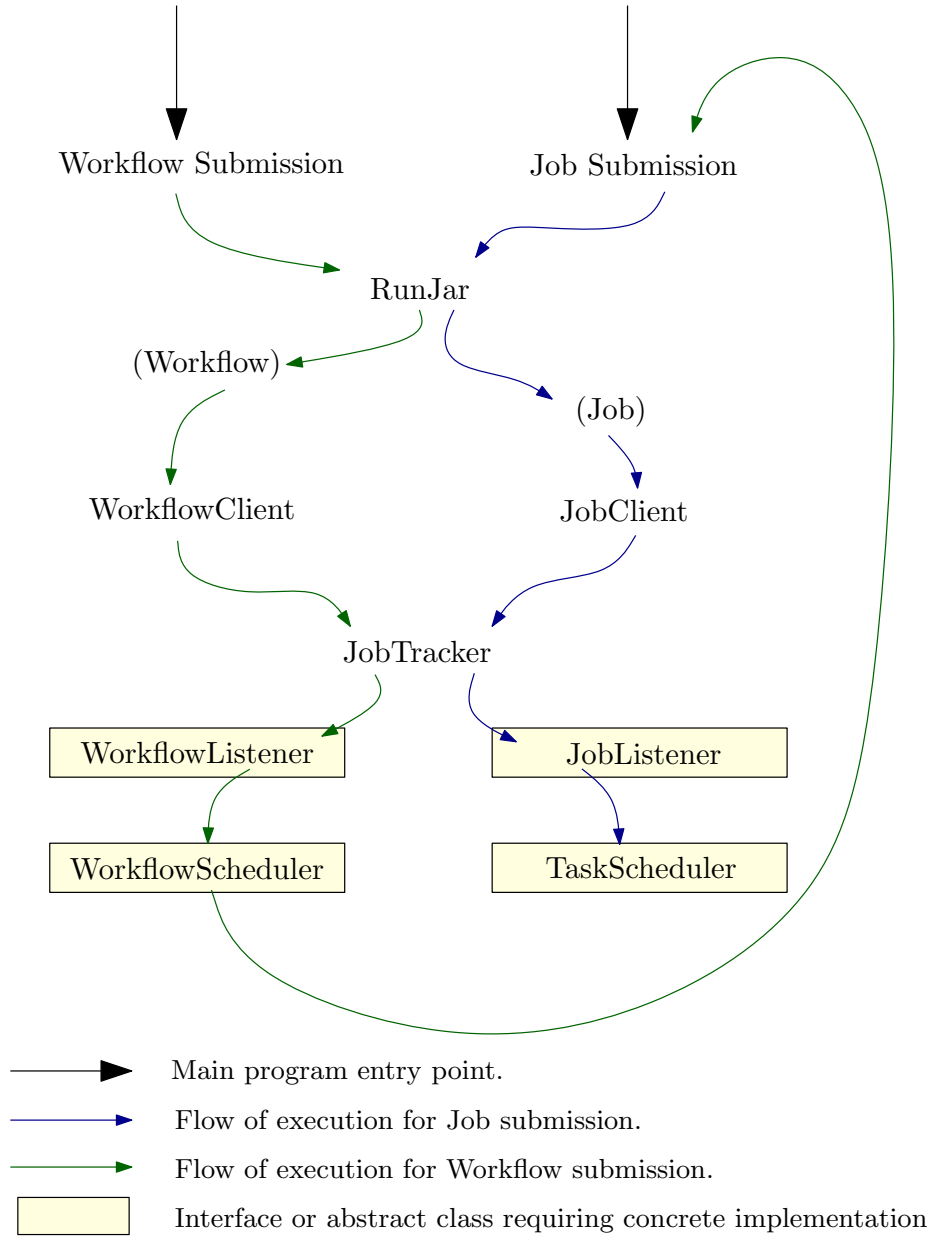


Figure 20: Hadoop job & workflow execution flow. Job execution flow is shown using blue arrows, while workflow execution is shown with green arrows. User submission of either a job or workflow is indicated by black arrows. In our work, both listener interfaces (and separately, scheduler interfaces) were realized in one concrete implementation.

and at the job level. Due to the increased complexity of scheduling workflows, the implemented *WorkflowTaskScheduler* delegates all scheduling decisions to the chosen *WorkflowSchedulingPlan*. This allows all scheduling-specific logic to be maintained within a single point, as the implemented *SchedulingPlan* both determines the ordering and matching of tasks to machines, and also ensures that the schedule is enforced during execution.

Again, throughout the execution of a workflow, all *TaskTracker*s repeatedly send heartbeat messages to the *JobTracker*, with the *JobTracker* fulfilling their request for new tasks through use of the *WorkflowTaskScheduler*'s $assignTasks$ method. When called, the $assignTasks$ method retrieves the queue of job and workflow objects from the *WorkflowListener*. It then iterates through all of the objects, determining for both jobs and workflows which actions are currently valid. When a workflow object (*WorkflowInProgress*) is detected, its status information is first queried to retrieve a list of its currently completed jobs. These jobs are then passed as a parameter to the *WorkflowSchedulingPlan*'s $getExecutableJobs$ method, which passes back a list of the currently executable jobs, ordered by priority. Jobs which have already been started are ignored, while new jobs are executed. When a job object (*JobInProgress*) is detected, its status information is also queried. However in this case the status information is used to determine which workflow the job belongs to. This information is then used to obtain the proper scheduling plan for the job. At this point, cluster information is gathered to determine if there are available slots to run tasks on. Additionally, other relevant information is gathered including the querying *TaskTracker*'s machine type, and the job's unique name. These values are then used together with the *WorkflowSchedulingPlan*'s $matchMap$ and $matchReduce$ functions to determine if either a map or reduce task can be run on the tracker, respectively. If so, a task from the job is obtained for execution using Hadoop's internal API. The execution is kept track of by the *WorkflowSchedulingPlan*'s $runMap$ or $runReduce$ functions,

respectively.

The execution of individual workflow jobs is accomplished by leveraging existing code in the Hadoop framework. This is done by running the job's jar file with the *RunJar* class's main method through execution in a new thread. As a result, the jobs are automatically set up and executed by the framework. Execution in this manner allows jobs to be run 'correctly', in that any required command-line arguments can be properly supplied to the jobs, which would not be possible through submission of their *JobConf* only. However, the method does introduce some issues which had to be dealt with. The main problem was that since submission of the jar file created a new *JobConf* object for the job, there was no direct link to existing job objects stored in the *WorkflowConf*. To remedy this issue, several pieces of additional identifying information are written to the job's jar file manifest before submission. This information includes the job's job identifier (*JobID*), workflow identifier (*WorkflowID*), main class, command-line arguments, job name, input directories, and output directories. Some of this information is then read in the *RunJar* class, allowing the main class and command-line arguments to be specified programatically. The remaining information is then read in the *JobClient* class during job submission, where the input directory, output directory, workflow identifier, job identifier, and job name are applied. Note that the input and output directory information is loaded and used in the *JobClient* as opposed to being used in the *RunJar* class as command-line arguments. This is because jobs with multiple dependencies are not run correctly when passed the multiple input paths - only the first path is recognized as an input directory, while each additional path takes the place of an additional argument. As a result, *RunJar* is modified to pass in a fake input and output directory to workflow jobs, while the actual directories are set in the *JobClient* before job submission. One last consequence of executing workflow jobs in this manner is that since the jobs are being run programmatically, the general ordering of arguments must be kept consistent from one

job to the next. As such, we have defined that the parameters for all workflow jobs are ordered as: `input-directory output-directory [job-arguments ...]`.

The actual data created during workflow execution is handled by storage within the Hadoop Distributed FileSystem, as explained forthwith. When the workflow is submitted, all required resources are assumed to also be located in the same directory as the workflow's jar file. This includes workflow job jar files, which during workflow submission are copied to the workflow's staging directory. The staging directory is located in HDFS, and as such allows replication of the job's jar files throughout the cluster so that any *TaskTracker* can access the files. It also allows the modifications made to the job jar file manifests to occur on the copied job, rather than the original. For the user, this means that multiple jobs can be packaged in a single jar without any issues, and thus that multiple jobs can be submitted using the same jar file (even if only a separate instance of the same job jar). When an individual workflow job is submitted however, the jar cannot be run from within HDFS, nor can its temporary data be stored in the staging directory. This is solved by the workflow job's jar file being copied from HDFS to the local filesystem, where submission is then run from. During execution of the workflow job a directory is then created in HDFS for the job's output, labelled by a combination of the workflow and job names. Note that after workflow completion both the local job jar files and the temporary data files are removed.

Similar to job execution, workflow status information is also periodically retrieved during execution and supplied to the *WorkflowClient* where execution began. The information presented to the user currently includes an overview of the status information for all workflow jobs. However, future work would also include detailed status information of any currently executing jobs, similar to the information presented when running a single job on the Hadoop framework.

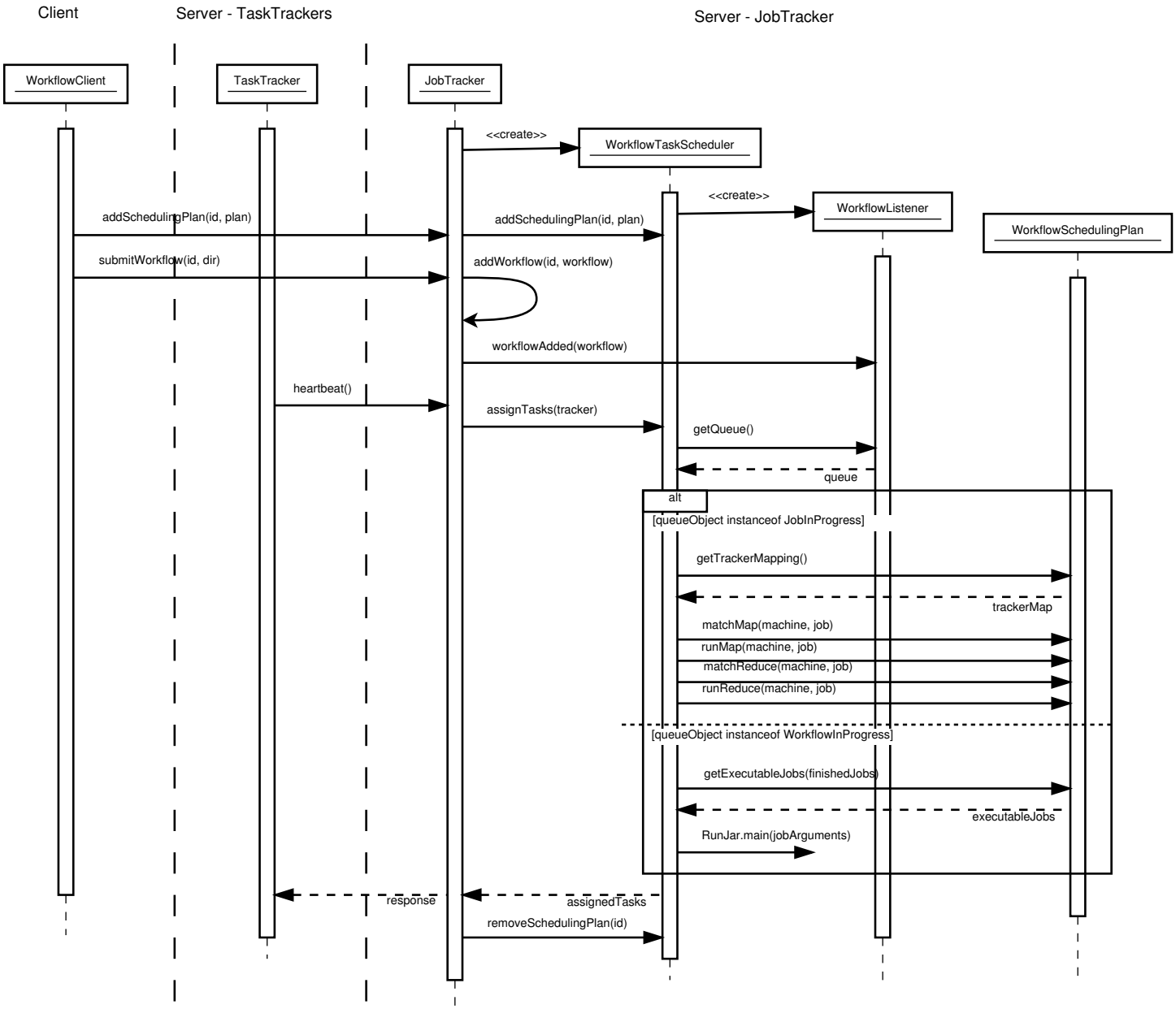


Figure 21: A sequence diagram showing execution flow of workflow scheduling. Actions taken for both task scheduling and job scheduling are shown within an alternative combination fragment. Tasks are scheduled for job (*JobInProgress*) objects, and jobs for workflow (*WorkflowInProgress*) objects.

5.4 Workflow Scheduling Plan

The information generated by the specified *WorkflowSchedulingPlan* defines the manner in which to pair tasks to available machines. Along with this, prioritization information can also be given to individual tasks and jobs, and used when communicating scheduling information to the *WorkflowTaskScheduler*. This scheduling information is computed during client-side submission. By computing the schedule before submission to the *JobTracker*, existing jobs or workflows currently in progress are not affected by any extreme computational load. This is especially helpful if the submitted workflow proves to be unschedulable, as effort will also not have been expended to replicate workflow resources across HDFS.

As mentioned, during client-side workflow submission the concrete implementation of the *WorkflowSchedulingPlan* generates the information required for job and task scheduling. This occurs immediately after workflow job information is updated, and is followed by addition of the computed scheduling plan to the *WorkflowTaskSheduler* via the *JobTracker*. From the *WorkflowClient*, a method of the *JobTracker*'s is called which adds the scheduling plan to a collection of scheduling plan objects owned by the *WorkflowTaskScheduler*. The *JobTracker* acts as a proxy, forwarding the scheduling plan to the current *WorkflowTaskScheduler*. During scheduling, the correct scheduling plan is then retrieved by using the scheduled object's (job or workflow) workflow identifier (*WorkflowID*). This allows jobs and tasks to be scheduled according to the plan they belong to, and more importantly enables multiple workflows to run concurrently. So even though we only consider scheduling of a single workflow in our algorithm and during testing, the implementation has been written to allow for multiple workflows to be executed concurrently.

Broadly speaking, this collection of scheduling plan objects owned by the *WorkflowTaskScheduler* is managed by the *JobTracker*, as it both proxies calls to add

scheduling plans, and determines when to remove completed scheduling plans. As mentioned, the *JobTracker* keeps a handful of data structures that track both submitted workflows and jobs. Throughout its lifetime, an additional thread (*RetireJobs*) is run alongside the main *JobTracker* thread that periodically queries for the current jobs and workflows in progress. If an instance of either is found to be completed then it is removed from the data structures. When a workflow is found to be completed, the *JobTracker* additionally calls for scheduling plan removal on the *WorkflowTaskScheduler*.

Using the information presented here and in Section 5.3, it is easy to see how any scheduling plan can fit into the provided framework. As all scheduling decisions are delegated from the *WorkflowTaskScheduler* to the chosen scheduling plan, the scheduling decisions can be made on a wide array of information. For our purposes, we implemented the Optimal and Greedy Scheduling algorithms. Also implemented was a modified Progress-Based algorithm, as proposed in [45]. We say slightly, as the proposed algorithm was mainly designed to make scheduling decisions between multiple workflows, whereas we assume execution of a single workflow which has access to the full resources offered by the Hadoop cluster. This criteria eliminates most of the need for inter-workflow scheduling decisions, which was an important focal point in the related work. However, the authors still proposed several methods to approach intra-workflow scheduling, all of which are unique when compared to our proposed methods.

Each of the implemented scheduling plans parse the workflow configuration (*WorkflowConf*) using a method provided by the *WorkflowDAG* class to create a DAG of workflow jobs (*WorkflowNode*s). A *WorkflowDAG* instance contains predecessor and successor information for all jobs, along with the jobs themselves. The jobs each contain a set of both map and reduce tasks (*WorkflowTask*s). The *WorkflowDAG* class provides methods to query the workflow for information, such as retrieval of entry

and exit nodes, successors and predecessors of a node, and more involved processes such as computation of the current critical path, total workflow execution time, and total workflow cost. The scheduling plans manipulate the DAG by setting machine types for the DAG's tasks. Following assignment of this information, the computations use the set machine types along with the constructed time-price table to allow computation of the queried information.

5.4.1 Scheduling Plan Interface

Each implemented scheduling plan enforces its scheduling method through several functions defined in the *WorkflowSchedulingPlan* interface. These are the $getTrackerMapping$, $matchMap$, $runMap$, $matchReduce$, $runReduce$, $getExecutableJobs$, and $generatePlan$ methods.

The $getTrackerMapping$ function is perhaps the least involved of the listed functions. It returns a mapping from actual nodes in the cluster to their machine type (as given in the loaded machine types XML file). Currently, this function matches potential resource types to existing resources through a weighted distance function that considers machine attributes (eg. RAM, number of CPUs, CPU frequency). After distance computation, pairs between the two sets with lowest distance are considered to be matched.

The $matchMap$ and $matchReduce$ functions are called from the *WorkflowTaskScheduler* during the scheduling of jobs. They are passed a machine type along with a job name, and return whether a task from the input job can be run on the given machine type. The $matchMap$ and $matchRed$ functions are assumed to be executed before the $runMap$ or $runReduce$ functions, and are used only to verify a given task-machine pairing. After this test, the $runMap$ and $runReduce$ functions are then used to update the scheduling plan by considering the obtained task to have successfully run. Note that as all tasks are considered to be nearly homogeneous, the specific task

to run does not matter. This issue is also a limitation of the Hadoop framework, in that the *JobInProgress.obtainNewMapTask* (and *obtainNewReduceTask*) functions do not allow selection of a specific task. Nevertheless, tasks do end up to be mostly homogeneous due to the method used for input split creation. Namely, for the default *FileInputFormat* class, the split size is computed by dividing the total number of bytes for all files by the requested number of splits. As a result, a job with n tasks has at least $n - 1$ tasks of the same size.

As previously mentioned, the *getExecutableJobs* method takes as a parameter the currently finished jobs, and returns the jobs that are able to be executed at this point in workflow execution according to their priority.

The last function specified in the interface is *generatePlan*, which is the main function executed during scheduling plan creation in the *WorkflowClient*. This function is where each individual *WorkflowSchedulingPlan* generates its plan, determining how to respond to the other required functions called during workflow execution. The function takes as parameters a set of available machine types, the actual machines which are available for use in the cluster, the time-price table, and the workflow configuration. Note that the workflow configuration (*WorkflowConf*) also contains constraint information. After execution, the function returns a boolean indicating whether the given constraints can be satisfied with the set of machines available in the cluster. It is currently assumed that workflow execution does not proceed if the workflow cannot be scheduled to meet the supplied constraints.

In the following subsections 5.4.2, 5.4.3, and 5.4.4, an overview of the implementation for each tested scheduling plan is given.

5.4.2 Optimal Scheduling Plan

The *OptimalSchedulingPlan* class - similar to the *GreedySchedulingPlan* and *ProgressBasedSchedulingPlan* - extends the *WorkflowSchedulingPlan* class, implementing the

methods reviewed in section 5.4.1. Like the *GreedySchedulingPlan*, its implementation follows closely to the pseudocode presented in Chapter 4. The scheduling method presented in the pseudocode is executed throughout the *generatePlan* function, while the remaining methods are satisfied through use of the generated *WorkflowDAG* and a related *taskMapping* variable.

Recall that the scheduling plan pairs tasks together with machines (types), while also possibly prioritizing these tasks and the overall job ordering within the workflow. The method by which this is done occurs in the *generatePlan* method. First, a *WorkflowDAG* instance is generated, after which its tasks have their machine types set throughout execution of *generatePlan*. These pairings are then used in the *matchMap*, *runMap*, *matchReduce*, *runReduce*, and *getExecutableJobs* methods. The *taskMapping* variable simply maps workflow job names to the actual job objects stored in the *WorkflowDAG*, as jobs are passed to and from the *matchMap*, *runMap*, *matchReduce*, *runReduce*, and *getExecutableJobs* methods using their unique names.

The *generatePlan* function begins by reading constraint information and using a utility class to generate a *trackerMapping*, which maps a *TaskTracker* node in the cluster to a defined machine type. It is this *trackerMapping* variable that is returned by the *getTrackerMapping* function. A basic check for schedulability is then done by setting all tasks to run on the least expensive resource type. Workflow cost is calculated, and if greater than the budget constraint the scheduling returns. Otherwise, the provided workflow is schedulable, and the optimal scheduling algorithm presented in Algorithm 4 is run. All tasks are gathered from the *WorkflowDAG* so that their machine types can be easily set while iterating through possible permutations of pairings. The permutations themselves are then generated, yielding a list of lists. Each inner list contains a single permutation of pairings, as defined by an ordering of machine types. For each permutation the stored machine types are paired

with the retrieved tasks, which is then followed by cost calculation. Both the cost calculation (*getCost*) and makespan computation (*getTime*) are defined as methods of the *WorkflowDAG* class. For cost calculation, the sum of costs for all tasks in the *WorkflowDAG* is returned. For makespan, the *WorkflowDAG*'s critical path is first calculated using the *getCriticalPath* method (following Algorithm 2). After this the times of all stages on the critical path are summed to produce the workflow makespan, where the time of a map or reduce stage refers to the longest executing task in the map and reduce stages of a job on the path, respectively. After the permutation's cost is calculated, it is compared with the budget constraint to check validity. Next, the workflow running time is compared to the best computed schedule so far, where the scheduling with the minimum time always kept. After considering all permutations, the optimal scheduling plan is guaranteed to be found, and is stored via task-machine type pairs within the *WorkflowDAG*.

The *matchMap*, *runMap*, *matchReduce*, and *runReduce* functions are all factored out into a single *runTask* function. This *runTask* function takes as input parameters the machine type and job name values from the original function, along with variables that indicate the type of task to be run and whether the task should actually be considered as run. This allows testing of validity that the *match∗* functions require, as well as the plan synchronization required by the *run∗* functions. Upon entry into the *runTask* function, a collection of tasks is retrieved from the specified job. These are either the job's map tasks or reduce tasks, depending upon the input task type. Then each task is iterated through as the function attempts to find a task assigned to the specified machine type. If a match is found the function returns true, as well as removing the task considered to be run from the job's list of tasks. Otherwise, the function returns false to indicate that a task from the input job can not be run on the input machine type.

Lastly, the *getExecutableJobs* function determines the currently available jobs

when given a collection of completed jobs. If the input is an empty job list, the function simply returns the *WorkflowDAG*'s entry nodes. Otherwise, each new finished job's successors is checked for eligibility, and added to the return list if found eligible. The test is done for each successor by checking for its predecessors in the finished jobs list. If all of the job's predecessors are completed, then it can be run. After checking all newly finished job's successors for eligibility, the function returns the computed list of executable jobs.

5.4.3 Greedy Scheduling Plan

As the *GreedySchedulingPlan* extends the *WorkflowSchedulingPlan* class, it also implements the $getTrackerMapping$, $matchMap$, $runMap$, $matchReduce$, $runReduce$, $getExecutableJobs$, and $generatePlan$ methods. As both the *GreedySchedulingPlan* and *OptimalSchedulingPlan* operate at the task-level by producing a task to machine type pairing, the only functions not shared between the two is the method of generating the pairing. Thus, for the two classes the $getTrackerMapping$, $matchMap$, $runMap$, $matchReduce$, $runReduce$, and $getExecutableJobs$ functions are the same. This leaves the $generatePlan$ function to be explained.

The $generatePlan$ function follows closely to the method outlined in Algorithm 5. It begins by obtaining the workflow's budget constraint, and then constructs the $trackerMapping$ variable. After this, the input machine types are sorted according to their cost, and the *WorkflowDAG* is constructed. Tasks have their machine type set to the least expensive machine to begin with, which also allows basic validation of schedulability in addition to providing a starting point for optimization. After these initial startup actions the main loop is entered, which executes while there is remaining budget available. In this main loop the critical path of the *WorkflowDAG* is first calculated. After this, the slowest pair of tasks in each stage in the critical path are computed using the time-price table. This is carried out by iterating through

all workflow jobs in the critical path, and for each stage in each job computing their slowest pair of tasks. For each of these pairs a utility value is computed and stored in a *TreeSet*. The next section in the main loop begins iteration through the computed utility values in order to reschedule tasks onto quicker machines. For each utility value considered, the slowest task is retrieved and its machine type updated. From this point the cost difference is computed to determine if the remaining budget is sufficient to allow the rescheduling. If the remaining budget is not sufficient the current utility value is removed and iteration continues through the remaining utility values. Otherwise the budget is updated, and the loop is broken out of. This is done as a rescheduling modifies the critical path, and thus a recomputation of the critical path and the utility values is required. The last case that can occur is if no utility values yield a reschedulable task. In this case the inner loop exits with an empty *TreeSet* of utilities, causing the outer loop to be broken out of.

In our implementation utility values are actually instances of a *Utility* class, which contains a (slowest) task and the actual utility value. Instances of this class are *Comparable* to each other, allowing sorting by storage in the mentioned *TreeSet*. Additionally, we also define a *WorkflowTaskPair* which is used to contain the slowest pair of tasks used for the computation of utility.

5.4.4 Progress-Based Scheduling Plan

The *ProgressBasedSchedulingPlan* follows the main algorithm presented in [45]. For this scheduling plan, the idea is to actually simulate scheduling of the workflow execution through use of scheduling events (*SchedulingEvent*), free slot events (*FreeEvent*), and their time of occurrence. The *SchedulingEvent*s represent the submission of a number of tasks from a given job at a certain time, whereas the *FreeEvent*s represent completion through release of a number of task slots at a given time. Prioritization of task scheduling is performed by a *WorkflowPrioritizer*, which orders sets of workflow

jobs into several priority queues where they await execution. The authors of [45] considered several prioritizers in their work, though in our implementation of their algorithm we selected a highest level first prioritizer (*HighestLevelFirstPrioritizer*). The prioritizer assigns levels to the jobs in a workflow based on dependency information, and assigns a higher priority to jobs at a higher level. It should be noted that in their work, the authors considered only deadline-constrained scheduling. Additionally, they made no mention of any rationale for selection of machines for task scheduling. As such, we assume that in a budget and deadline-constrained environment, their proposed algorithm would be modified to retain an emphasis on makespan minimization. To that end, all tasks are assigned to the quickest machine type, as this would provide the greatest makespan minimization.

Similar to the *GreedySchedulingPlan* and *OptimalSchedulingPlan*, the $generatePlan$ function begins by using a utility class to generate the $trackerMapping$ variable returned by the $getTrackerMapping$ function. After this the quickest machine is found, and the *WorkflowDAG* is constructed. The *HighestLevelFirstPrioritizer* is then constructed and priority queues for workflow jobs are created. Variables used to track the simulation are also initialized at this point. The total number of map and reduce tasks for each workflow job are retrieved, and the total number of map and reduce slots are recorded. A *PriorityQueue* of *FreeEvent*s is created for both map tasks and reduce tasks ($mapQueue$ and $redQueue$, respectively), along with a priority queue of workflow jobs (*WorkflowNode*) for each stage ($addMapQueue$, $addRedQueue$, and $addSuccQueue$). The last queue holds jobs whose successors can now be checked for scheduling eligibility. The prioritizer is then used to add the initial set of workflow jobs to the $addMapQueue$, and slot free events are added according to the total number of map and reduce slots in the cluster.

After this initialization the main loop begins, which executes until all workflow jobs have had their tasks scheduled. There are four major sections within the main

loop. They consist of updating the number of free slots, scheduling map tasks, scheduling reduce tasks, and the addition of new jobs to the map task execution queue (*addMapQueue*). In addition to these sections, the current time within simulated execution is kept track of using task execution times. It is this value, as stored in the *SchedulingEvent*s that is used in the *matchMap*, *runMap*, *matchReduce*, *runReduce*, and *getExecutableJobs* functions.

The main loop begins by iterating through all entries in both free slot queues with a time less than or equal to the current simulation time, updating the number of currently free map slots and free reduces slots. Next, map tasks are scheduled until either there are no free map slots remaining, or all workflow jobs in the *addMapQueue* have been considered. In this loop a workflow job is first retrieved from the priority queue. The number of map tasks that are left to schedule are then retrieved, and the actual number of maps to schedule is computed as the minimum of the free map slots and the job's remaining map tasks. The number of slots and tasks are then updated, and a *SchedulingEvent* created based on the computed values. Along with this, a new *FreeEvent* is created based on the values with its time set as the sum of current time and task execution time. If a job is in the *addMapQueue* but has no map tasks remaining for scheduling, then it is moved to the *addRedQueue*. After the section for scheduling map tasks, the reduce task scheduling section begins. This section uses the same logic and execution flow as the map task scheduling section, except that jobs without any reduce tasks are added to the *addSuccQueue*. At this point in the implementation the *addSuccQueue* is then checked for jobs. If it is non-empty, the newly finished jobs are passed to the prioritizer, which passes back any newly eligible jobs. The newly eligible jobs (if any) are then added to the *addMapQueue*. Finally, the current time is updated by assignment of the minimum time value of free slot events.

Again, similar to the *GreedySchedulingPlan* and *OptimalSchedulingPlan*, the

matchMap, *runMap*, *matchReduce*, and *runReduce* functions are all factored out into a single *runTask* function. The *runTask* function first retrieves the next *SchedulingEvent* from the queue that has a time less than or equal to the current time. If no event exists then no task can be run. Otherwise, the event is validated against the input job name, and is also checked to ensure there are still tasks of the correct type for scheduling. If so, all tasks of the input task type are retrieved according to the input job name, and then iterated through in an attempt to locate a task that can be run on the input machine. If a match is found, the event's number of tasks is reduced (in accordance with the given task type), and the task removed from the workflow job's list of tasks. In addition, if the event has neither map or reduce tasks remaining then it is removed from the queue, and the current time updated to coincide with the task completion time.

Lastly, we review the *getExecutableJobs* function. This function, like the *runTask* function, uses the queue of *ScheduleEvent*s to compute the jobs that are currently eligible for execution. To begin with, all events with a time less than or equal to the current time are retrieved from the queue. These events are then iterated through, and are added to the list of eligible jobs if all of their predecessors have completed execution.

Chapter 6

Empirical Studies

6.1 Introduction

To verify and evaluate the proposed algorithm and study its performance behaviours in reality, our additions to the Hadoop framework have been tested in an 81-node cluster. For our experiments we utilized the Amazon EC2 platform, wherein the modified Hadoop framework was deployed to the rented Virtual Private Server (VPS) machines. Testing was executed in several steps. We began by determining the influence of data transfer speed on execution times, which yielded insight into the design for jobs used during testing. Following this phase, we constructed a workflow to test our modifications against, and then gathered the necessary input data for proper algorithm execution. Lastly, our greedy budget-constrained workflow scheduling algorithm was tested on the Hadoop framework.

In this chapter we review all steps taken to validate our modifications. We begin in Section 6.2 with an overview of the test setup, which is then followed in Section 6.3 by a description of the data collection method for the time-price table required by our greedy algorithm. Concluding the chapter in Section 6.4 is an explanation of the main results produced by testing of the greedy scheduling algorithm.

6.2 Test Setup

6.2.1 Cluster Configuration

In the related literature there exist many different test configurations proposed for validation of modifications to the Hadoop framework. The authors of [76] utilize a cluster of 800 virtual machines on Amazon EC2 (spread over 160 dedicated physical machines). At the other end of the spectrum, [79] employs a 30-node cluster on Amazon EC2, with nodes split evenly between three different machine types. Lastly, the authors of [80] propose the most convincing configuration. In their work, three different configurations are tested on Amazon EC2. Each configuration varies in both the total number of machines (68, 97, and 99) and the composition (between two different machine types). Comparing their configuration to a 2010 survey outlining the average Hadoop cluster size as 66 nodes, this seems to be an excellent cluster size [63].

Using these figures, we decided on a cluster size of 81 nodes, all located in one Amazon EC2 region for the purpose of minimizing data transfer times. The size was selected due to the figures proposed in [63], along with the cluster configuration used in [80]. Another factor in the decision of overall cluster size is the fact that many large companies are continuing to grow their Hadoop clusters, and as a result the average as of 2010 has likely also increased.

With regards to machine types, Amazon EC2 offers many machines suitable for different computational situations [2]. As an overview, they provide instances optimized for any of compute, memory, Graphics Processing Unit (GPU), or storage scenarios. They also offer instances suitable for general-purpose computation, which are the types that we employ in our tests. Altogether, we utilize four different machine types within the EC2 *m3* family: *m3.medium*, *m3.large*, *m3.xlarge*, and *m3.2xlarge*. The properties for these machines are shown in Table 4.

Instance Type	CPUs	Memory (GiB)	Storage (GB)	Network Performance	Clock Speed
m3.medium	1	3.75	1 x 4 SSD	Moderate	2.5
m3.large	2	7.5	1 x 32 SSD	Moderate	2.5
m3.xlarge	4	15	2 x 40 SSD	High	2.5
m3.2xlarge	8	30	2 x 80 SSD	High	2.5

Table 4: An overview of the Amazon EC2 machine types used during experimentation.

We base our selected machine distribution on the insight that composition in a production cluster is often not balanced; as machines become obsolete or otherwise are decommissioned, they are replaced by newer machines which have more computational power. With this in mind, we propose a test configuration comprising 30 *m3.medium* nodes, 25 *m3.large* nodes, 21 *m3.xlarge* nodes, and 5 *m3.2xlarge* nodes. A single node of type *m3.xlarge* is used as the master (*JobTracker*), while the remaining nodes are retained as slaves to take on the role of *TaskTracker*s.

6.2.2 Workflow Configuration & Job Definition

Apart from configuration of the cluster environment, the other main testing decision is which workflow to test with. Originally we selected both the SIPHT and LIGO workflows, as shown in Figures 3 and 1, respectively. The aforementioned workflows were mainly selected for two reasons. First, they both contain all workflow substructures as explained in Figure 4. Second, both workflows are sufficiently large at 31 jobs for SIPHT and 40 jobs for LIGO. In addition to these properties, SIPHT was constructed to use two separate input directories, and the LIGO workflow is actually defined as two DAGs contained in a single graph. These properties together cover all edge cases to be tested with the workflow scheduling modifications made to the Hadoop framework. However, testing of our modifications to the Hadoop framework and the proposed scheduler on the final 81-node cluster were only accomplished with

the SIPHT workflow. This is mainly a result of confirmation of LIGO's proper execution during the collection of task execution times, as well as the scope of the thesis.

The workflows are constructed such that each job executes the same Java program. This program is written in such a way that execution by any machine type produces behaviour indistinguishable from an actual submitted workflow job. We accomplish this by providing both a computational load and performing data read & write operations during the job's execution. The computational load is provided by the calculation of a Pi approximation via the Leibniz iterative method, until the point when a certain precision is obtained. This precision value, which we call the *margin of error*, is a configurable parameter passed to the job that alters the computational load, and thus the execution time. Similar to the generation of a computational load, the data operations are defined in both the map and reduce functions of the job. The data processing performed includes reading the input data, appending a unique task identifier, and finally writing out the modified information.

The reason for use of a synthetic job in the workflows is two-fold. First, due to time constraints the burden of execution for jobs used in production systems is too high. Generally, jobs run in production systems take hours to execute, whereas we require an execution time of minutes to allow both generation of historical data and final testing of the workflow scheduler. To achieve this, we require a method to tune the task execution time which also captures the relative differences between execution times on different machine types. This requirement is accomplished by the aforementioned Pi approximation because a real computational load is placed on the machine, which completes in time proportional to the machine's computational power. For instance, assume that a lower execution time is desired. To obtain this lower execution time, we realize that a higher margin of error allows less iterations of the approximation algorithm, and thus places less computational load on any machine

executing the job. This lower computational load then results in a lower execution time.

Selection of a specific value for the margin of error is based on several considerations. First, larger execution times result in a larger cost range and higher overall cost magnitude. As such, larger execution times are advantageous to make scheduling decisions more pronounced in the output data. Similarly, modifications to the budget are easier to make if there is a larger range of valid budgets available. Second, as mentioned previously, the scope of the thesis does not allow for hundreds of runs with day-long workflows. Therefore, we mainly desire a lower overall execution time.

Lastly, we must consider that the greedy scheduler only considers task execution times when making scheduling decisions. This means that any data transfers between workflow jobs or their contained tasks are not included in scheduling decisions. To minimize the effect of data transfer times on scheduling, we then desire the computation time of a workflow job to exceed its data transfer times. As a result, differences in execution time between budgets are more likely to be a consequence of scheduling decisions rather than variations in data transfer times caused by network instabilities.

For example, to determine the effect of data transfer times on total execution time we observed the difference in workflow execution times between two smaller clusters of 5 nodes when executing a workflow with no computational load. One cluster comprised *m3.medium* machines, whereas the other consisted of *m3.2xlarge* machines. On each cluster 5 runs of the LIGO workflow were executed with the greedy scheduling algorithm, producing workflow execution times averaging 284 seconds on the *m3.medium* cluster and 102 seconds on the *m3.2xlarge* cluster. Task execution times on the *m3.medium* cluster averaged 10 seconds for the *patser* entry job map tasks. Using the mentioned criteria, we decided upon a margin of error of 5E-8 for the tests in Section 6.4 as it increased the average task execution time to 30 seconds.

The second reason for use of the synthetic job is that it allows us to ensure correct

functioning of the scheduler with regards to both task and job ordering through documentation of workflow execution. For example, to validate a new workflow scheduler we first run the scheduler on a workflow of synthetic jobs. After workflow completion, the output of the scheduler is compared with the *WorkflowConf* specification. The output from scheduling contains a single line for each path in the executed workflow DAG, tracing the execution flow from the first map task to the last reduce task. These paths are then compared against dependencies specified in the *WorkflowConf* to ensure that no paths exist which disregard the submitted configuration.

6.3 Data Collection

As explained in [27, 37, 70], performance estimation for the use of scheduling can be achieved through several different techniques. These include analytical & performance modeling, consideration of historical data, simulation, on-line learning, and hybrid approaches. In our case, task execution times are required for each machine type used, so that the greedy scheduling algorithm is able to efficiently prioritize between any tasks requiring execution. In a production system whose administrators have recently decided to switch to the proposed algorithm, this performance estimation would likely be done using historical data. This would be accomplished by the administrators adding methods for data collection during the execution of jobs. The gathered data would then be used when scheduling using any algorithm requiring time-price table information. For situations where new jobs or workflows are executed on the cluster, a scheduler not requiring this information could be used (such as a simple FIFO scheduler). This is possible as task execution times are independent of job execution order. Additionally, on-line techniques could be applied wherein the time-price table information is continuously refined as workflows continue to be run, enabling the scheduler to operate more efficiently with each execution.

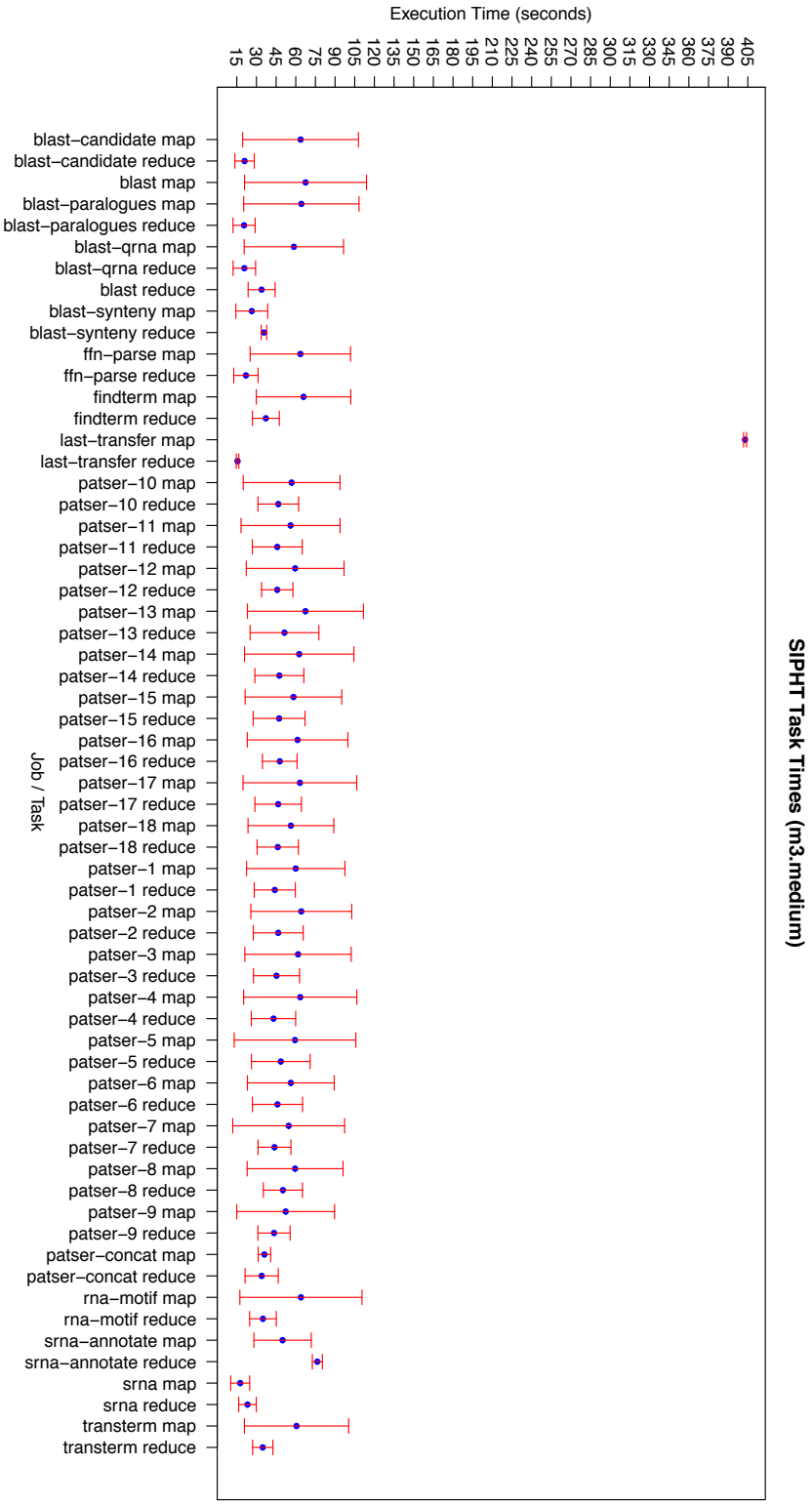


Figure 22: Displayed in this graph are the execution times of tasks belonging to the SIPHT workflow, as captured from workflow executions on the *m3.medium* machine type.

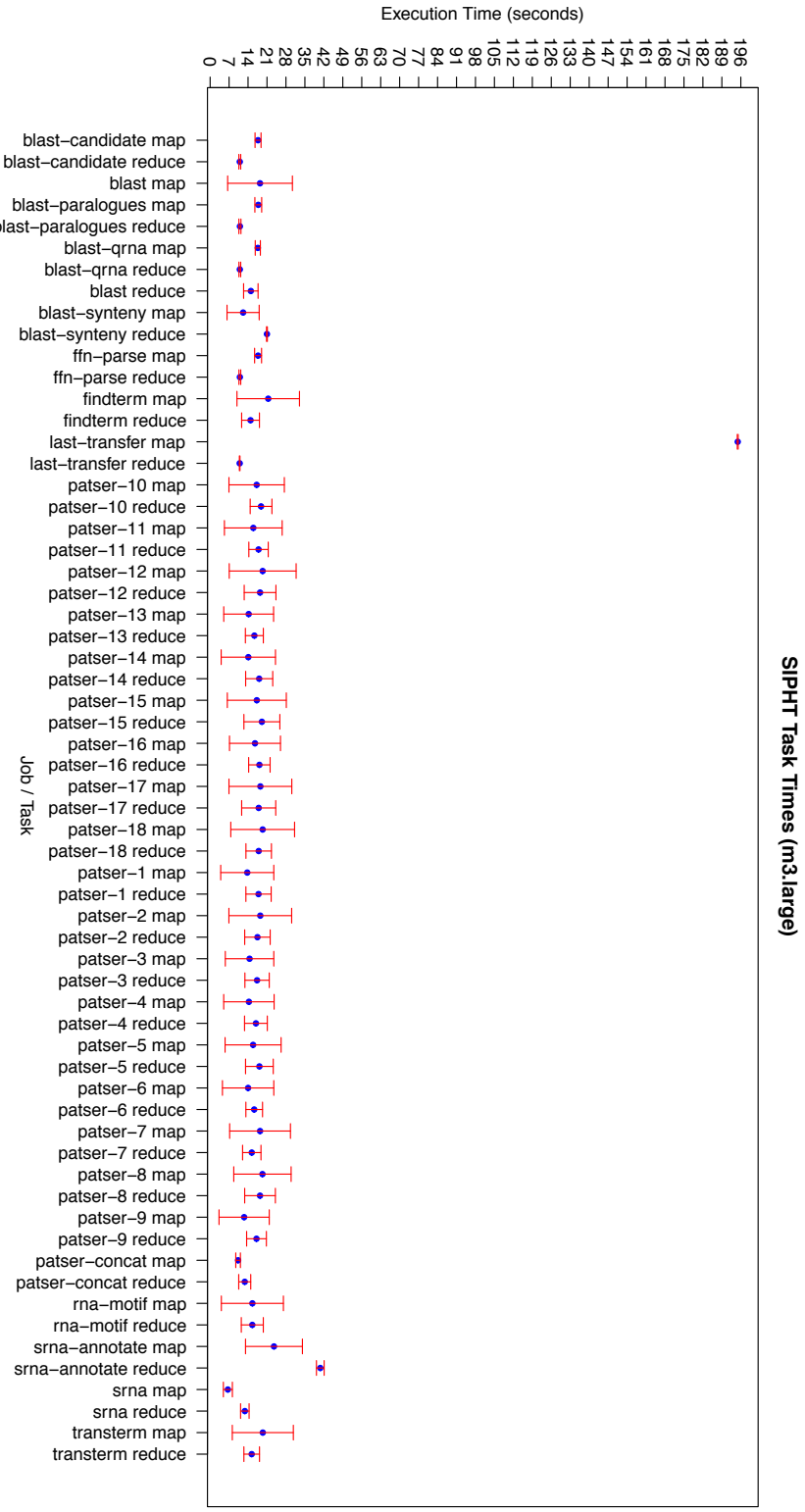


Figure 23: Displayed in this graph are the execution times of tasks belonging to the SIPHT workflow, as captured from workflow executions on the *m3.large* machine type.

Since the most likely method of performance estimation is the consideration of historical data, we employ this method during our data collection. To collect the task times for each job/machine type pair we first create a smaller homogeneous cluster of each machine type used in the final 81-node test cluster. The clusters vary in size with respect to their machine's processing power to allow parallel computation of the task times; more powerful machines obviously allow a smaller cluster size to complete the workflow in the same total time as a larger cluster of less powerful machines. After creation of the clusters, the greedy scheduler is executed on the SIPHT and LIGO workflows between 32 and 36 times on each cluster, with logging used to capture metric information. Again, recall that the scheduler used does not influence task times. Another important consideration with respect to the task execution time information is that inaccurate execution times does not halt execution of the proposed greedy scheduler. Instead, the incorrect task times force the algorithm to assign incorrect priorities, producing a schedule with sub-optimal makespan.

The final average task times along with their standard deviation for the SIPHT workflow are shown below in Figure 22, 23, 24, and 25. Each graph shows the task times for a single machine type, via the average task time and its standard deviation. Task times are given along the y-axis in seconds, and job name with task type along the x-axis. Several interesting facts about the data are shown through the graphs. The first is perhaps that total task execution time decreases as the machine type becomes more powerful, except for the transition from *m3.xlarge* to *m3.2xlarge*. As shown in Table 4, this is most likely due to the fact that the main difference between the machines was an increase in the number of cores, memory, and storage space. As an explanation, our synthetic workflow job both does not require much memory, nor is it easily parallelized. These are most likely the reasons for an absence of execution time reduction, and will be addressed in future work. Also regarding execution time reduction, we see that the main difference between execution times of

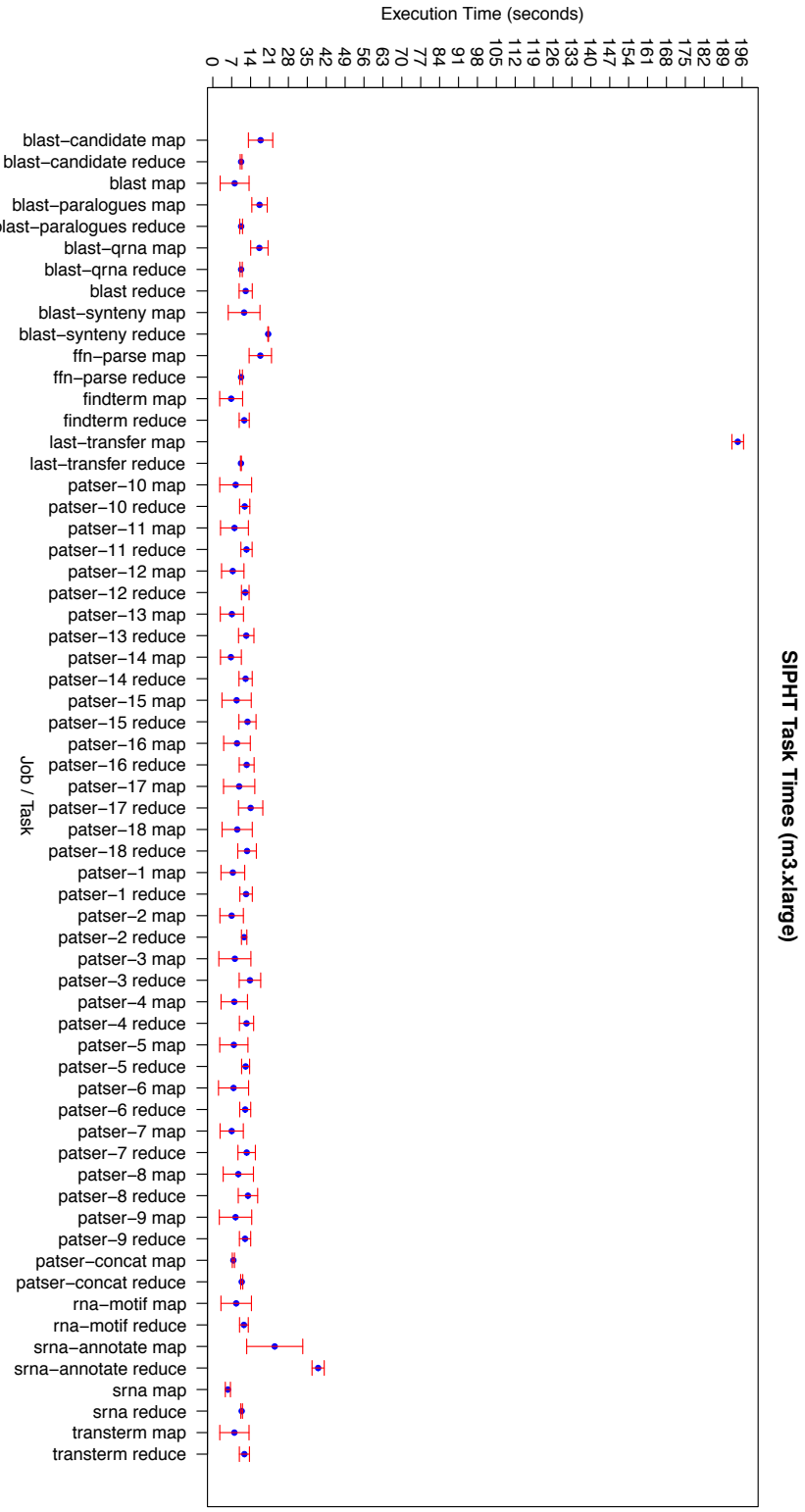


Figure 24: Displayed in this graph are the execution times of tasks belonging to the SIPHT workflow, as captured from workflow executions on the *m3.xlarge* machine type.

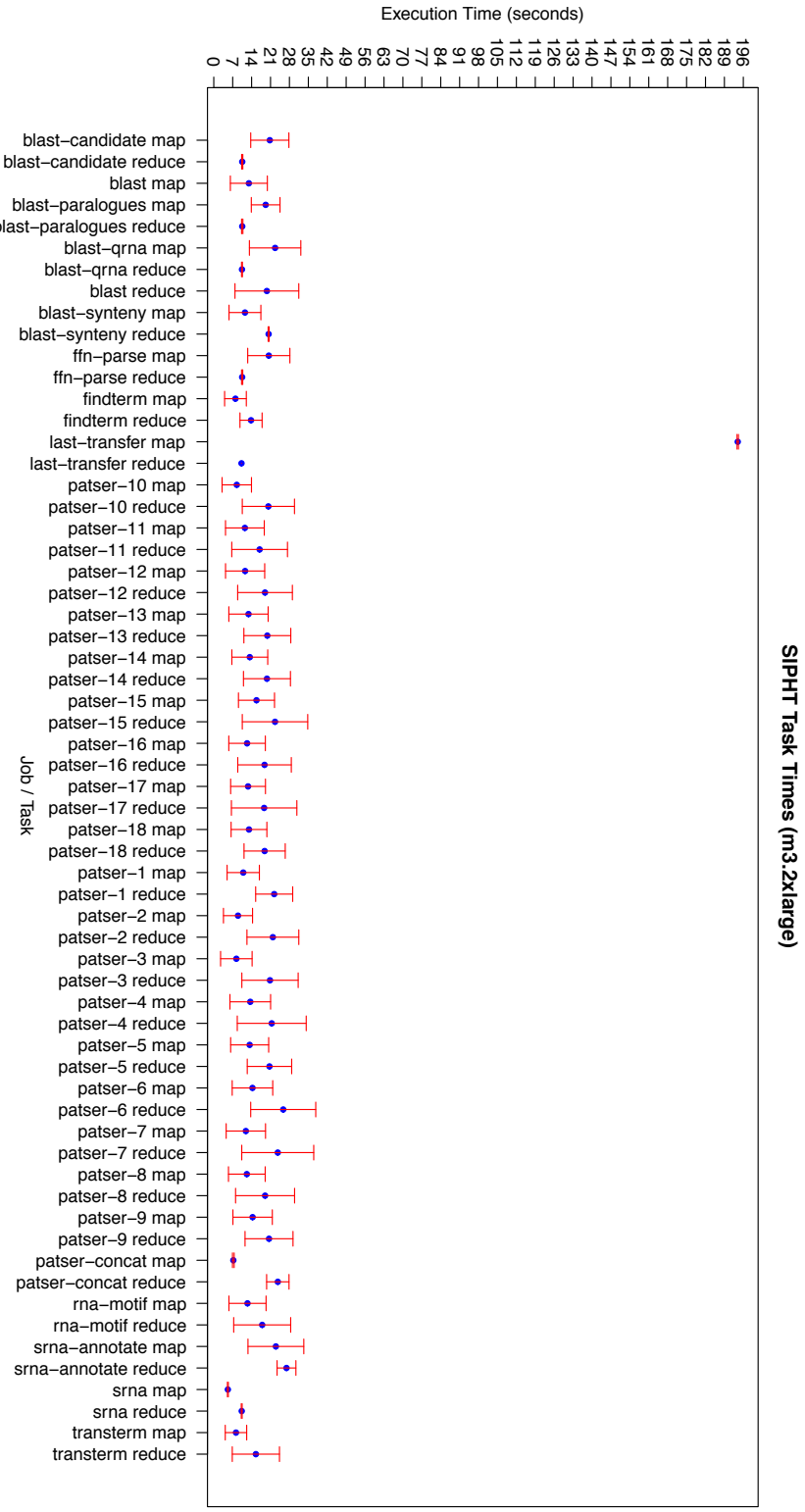


Figure 25: Displayed in this graph are the execution times of tasks belonging to the SIPHT workflow, as captured from workflow executions on the *m3.2xlarge* machine type.

tasks on *m3.large* and *m3.xlarge* is not the overall execution time, but the variance of execution times. Along with the increase in machine processor count, this difference could also be caused by the increase in network performance offered with the *m3.xlarge* machine. All graphs however show the correlation between individual task execution times. Recall that for SIPHT, the $srna - annotate$ and $last - transfer$ jobs perform the main data aggregation. As a result, we see that correctly the map and reduce tasks for these jobs have a much higher execution time. Along with this, we can also compare the *patser* input jobs to correctly see that they all are identical with respect to execution time.

6.4 Workflow Scheduling Experiments

After collection of task times for all machines utilized in the 81-node test cluster, we executed the greedy budget-constrained workflow scheduler on the SIPHT workflow using our modified Hadoop framework. This execution was run 5 times for 8 budget values within the range \$0.129 to \$0.16. The budgets were selected such that the range covered from an infeasible amount (budget is less than workflow cost when using only the least expensive machine type) up to an amount larger than the highest cost selected by the scheduler (all tasks assigned to the most expensive machine type). Within these boundaries, additional values were selected at even intervals between the boundary values. For each run the computed execution cost and time were recorded, along with the actual time and machine type mapping. The metric logging code used during task time collection was also used during testing, and along with the machine type mapping allowed us to compute the actual cost of workflow execution. The values for these runs were then averaged to a single value for each budget value, the results of which are displayed in Figure 26 and Figure 27.

Figure 26 shows both the execution time computed by the greedy scheduler and the

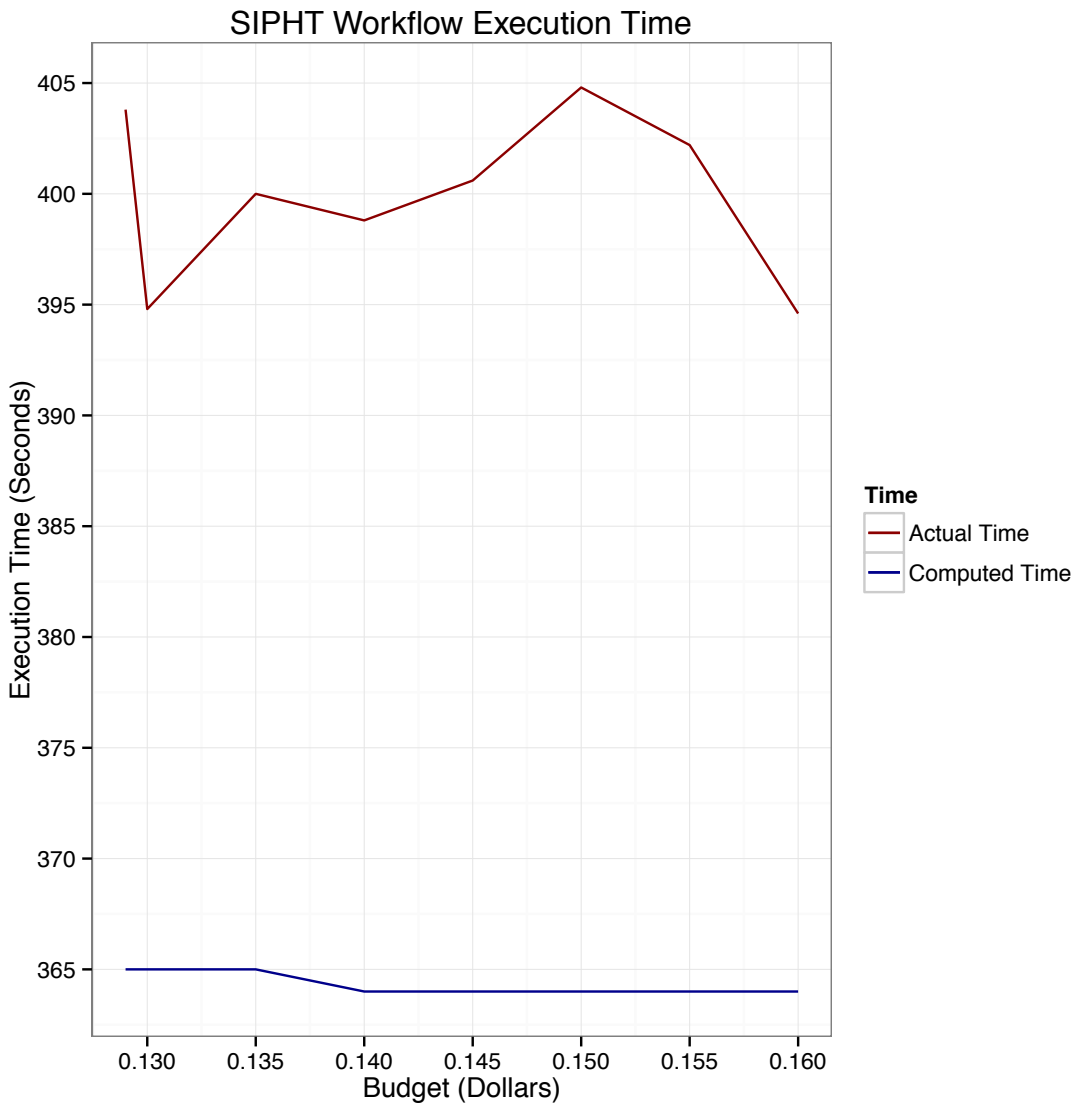


Figure 26: The actual and computed time for execution of the SIPHT workflow are shown according to different budget amounts. The proposed greedy scheduler was used for execution on our modified Hadoop framework.

actual execution time as recorded during testing. From these two figures, we observe that the actual execution time was generally 35 seconds above the computed execution time. As mentioned previously, and as shown through earlier testing in Section 6.2.2, the data transfer times experienced during execution are not insignificant. Since they are also not considered by the greedy scheduler, it is easy to realize this as the main source of the disparity.

Figure 27 similarly shows both the execution time computed by the greedy scheduler and the actual cost as computed from testing. As expected, both cost values increase as the budget increases, while still remaining below the budget amount. However, the actual cost is generally \$0.03 below the computed cost. The most likely reason is rounding errors seen with float values at the higher precision required for these computations. In reality, workflows are generally larger in size and comprise jobs with longer execution times. As such, less precision is required, and the difference between computed cost and actual computed cost seen in these experiments will not manifest.

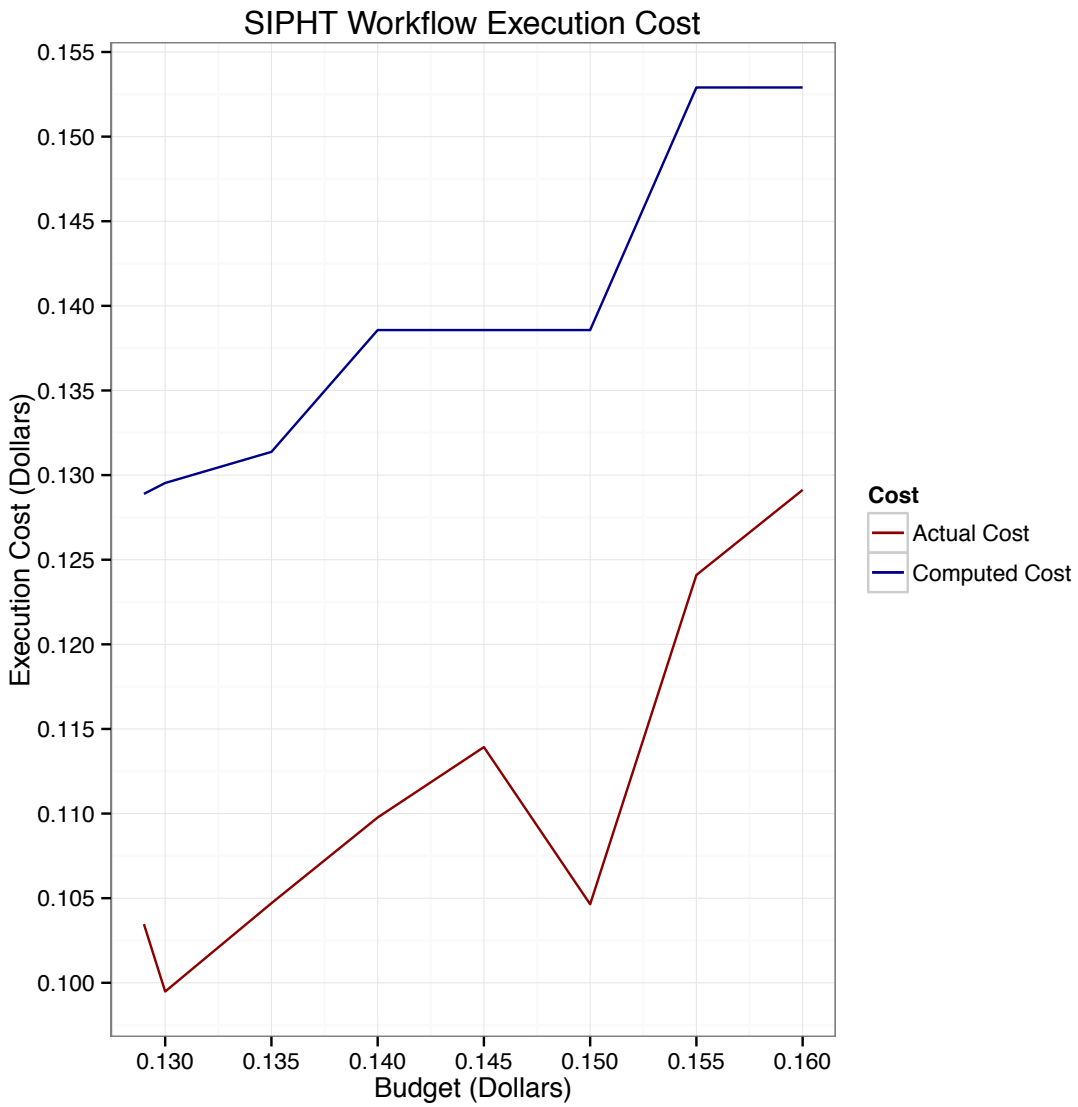


Figure 27: The actual and computed cost for execution of the SIPHT workflow are shown according to different budget amounts. The proposed greedy scheduler was used for execution on our modified Hadoop framework.

Chapter 7

Conclusions

7.1 Conclusions

In recent years cloud services - such as those supplied by IaaS providers - have been gaining traction, mainly due to their availability, scalability, and low cost. The ability to rent resources allows use cases for distributed processing which in the past would have been impossible. As one example, many companies involved in DNA sequencing and protein folding require computing resources for only several weeks at a time, after which the resources are not needed [39]. Along with applications in DNA sequencing and protein folding, these services are also used for the execution of scientific workflows in other fields including astronomy, physics, and seismology. This increase in the use of cloud services, combined with the proliferation of distributed computing frameworks and widespread adoption of Apache Hadoop in particular has allowed more users to take advantage of the benefits of distributed computing.

However, the combination of Apache Hadoop with execution of applications on rented infrastructure reveals several important, unimplemented features. One such feature is that of constraint-based scheduling. Considering that many scientific applications have moved to execution on IaaS clouds and Apache Hadoop as a result of several benefits including cost and exclusive use of resources, it is important that

scheduling algorithms can also make optimization decisions that consider both execution time and cost [19, 34, 76]. Another such feature is the execution of workflows in an efficient and well-supported manner, a requirement of many scientific applications. While several workflow engines exist to provide workflow scheduling on Apache Hadoop, none currently allow for constraints to be defined. Additionally, workflow engines for Hadoop handle the executed workflow themselves while passing individual jobs to the framework for execution. As a result, any possible optimizations available through scheduling the jobs as a single unit are lost. Furthermore, workflow engines do not determine the method of scheduling used by Hadoop. As such, the selected scheduler could unknowingly decrease the efficiency of workflow execution.

To address these issues, we have implemented modifications to the Apache Hadoop framework to allow fully integrated workflow scheduling. Additionally, we have developed and tested a greedy budget-constrained scheduling algorithm against several workflows, as well as developing both a progress-based and an optimal brute-force algorithm. These modifications are novel and have led to the completion of the first generic workflow scheduler fully integrated with the Apache Hadoop framework. Moreover, our greedy budget-constrained algorithm is the first scheduler for Apache Hadoop that both deals with budget constraints and executes workflows. Both the framework modifications and the greedy scheduler implementation have been extensively tested via execution on multiple workflow applications, which demonstrates the ability of our implementation to handle all possible workflow substructures. Results from our empirical studies establish these facts, given that the greedy scheduler both executes correctly, and produces an expected makespan and cost according to various budget constraint values.

7.2 Future Work

As future work, we would mainly want to accomplish additional testing of the implemented progress-based deadline-constrained algorithm. As well as additional testing for the progress-based scheduler, tests with larger workflows consisting of real jobs would complement the current testing performed to validate our contributions. We would expect to see more improvements with larger workflows, as small workflows bias against the optimizations available to finer-grained scheduling methods due to their simple topologies [45]. Along with this, larger workflows would allow a wider range of budgets to be tested, where changes in the workflow cost would also occur on a larger scale.

Also a focal point of future work would be incorporation of data transfer times and costs into scheduling plan creation, as the greedy scheduler does not yet consider data transfer times that occur between job or task executions within a workflow. This would also allow more optimized scheduling, as historical data could be used to provide more knowledge when making scheduling decisions.

Additionally, one of our assumptions includes the fact that the number of virtual machines available to rent from the IaaS provider is taken to be configurable, and only limited by the given budget constraints. As a result, we can assume that machines (slots) are never competed for by more than a single task. With the current testing this assumption was easy to satisfy due to large cluster size and relatively lower (when compared to thousands of jobs) workflow sizes. However, when employed in production systems the assumption breaks down. As a solution, we plan to either create a workflow scheduler that considers the available cluster size when creating a scheduling plan, or to determine a method that efficiently maps scheduling plans from what we denote as an 'unconstrained resource' problem to a 'constrained resource' problem.

Lastly, an important aspect of budget-constrained workflow scheduling on Amazon EC2 is that machines are rented on an hourly time scale. During our experiments however, we computed a fractional cost with as much precision as required. Generally, this meant that the cost for task execution corresponded to their actual execution times, whether it was several seconds or minutes. Implementation of a separate scheduler, or configurable parameters to change the time/cost granularity would be useful in cases when IaaS cloud providers rent machines on different time scales.

References

[1] Amazon EC2. http://aws.amazon.com/ec2/. [Accessed 2015-01-21].

[2] Amazon EC2 Instances. http://aws.amazon.com/ec2/instance-types/. [Accessed 2015-10-17].

[3] Amazon Elastic MapReduce. http://aws.amazon.com/elasticmapreduce/. [Accessed 2014-12-15].

[4] Announcing the Windows Azure HPC Scheduler and HPC Pack 2008 R2 Service Pack 3 releases! http://blogs.technet.com/b/windowshpc/archive/2011/11/11/hpc-pack-2008-r2-sp3-and-windows-azure-hpc-scheduler-released.aspx. [Accessed 2014-12-15].

[5] Apache Hadoop NextGen MapReduce (YARN). http://hadoop.apache.org/docs/r2.6.0/hadoop-yarn/hadoop-yarn-site/YARN.html. [Accessed 2014-12-17].

[6] Dagman. http://research.cs.wisc.edu/htcondor/dagman/dagman.html. [Accessed 2014-11-10].

[7] Greenplum HD. http://www.ndm.net/datawarehouse/Greenplum/greenplum-hd. [Accessed 2014-12-15].

[8] Hadoop at Yahoo! http://yahoohadoop.tumblr.com/. [Accessed 2015-08-18].

[9] Hadoop on Google Cloud Platform. https://cloud.google.com/hadoop/. [Accessed 2014-12-15].

[10] Hadoop Releases. http://hadoop.apache.org/releases.html. [Accessed 2014-12-17].

[11] Hadoop Wiki; PoweredBy. http://wiki.apache.org/hadoop/PoweredBy. [Accessed 2014-12-15].

[12] MapReduce Tutorial. http://hadoop.apache.org/docs/r2.6.0/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html. [Accessed 2014-12-16].

[13] MapReduce Tutorial. http://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html. [Accessed 2014-12-17].

[14] Microsoft Expands Data Platform to Help Customers Manage the 'New Currency of the Cloud'. http://blogs.microsoft.com/blog/2011/10/12/microsoft-expands-data-platform-to-help-customers-manage-the-new-currency-of-the-cloud/. [Accessed 2014-12-15].

[15] Pegasus. http://pegasus.isi.edu/. [Accessed 2014-11-10].

[16] Scheduling in Hadoop. http://www.ibm.com/developerworks/library/os-hadoop-scheduling/. [Accessed 2014-12-15].

[17] Welcome to Apache Hadoop! http://hadoop.apache.org/. [Accessed 2014-12-15].

[18] Workflow Engines for Hadoop. http://www.slideshare.net/jcrobak/data-engineermeetup-201309. [Accessed 2015-08-19].

[19] S. Abrishami, M. Naghibzadeh, and D. Epema. Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds. *Future Gener. Comput. Syst.*, 29(1):158–169, January 2013.

[20] G. Apostolos and Y. Tao. On the granularity and clustering of directed acyclic task graphs. *Parallel and Distributed Systems, IEEE Transactions on*, 4(6):686–701, Jun 1993.

[21] H. Arabnejad and J. Barbosa. A budget constrained scheduling algorithm for workflow applications. *Journal of Grid Computing*, pages 1–15, 2014.

[22] H. Arabnejad and J. Barbosa. Budget constrained scheduling strategies for online workflow applications. In *Computational Science and Its Applications – ICCSA 2014*, volume 8584 of *Lecture Notes in Computer Science*, pages 532–545. Springer International Publishing, 2014.

[23] S. Benkner, I. Brandic, G. Engelbrecht, and R. Schmidt. Vge - a service-oriented grid environment for on-demand supercomputing. In *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, pages 11–18, November 2004.

[24] B. Berriman, E. Deelman, J. Good, J. Jacob, D. Katz, C. Kesselman, A. Laity, T. Prince, G. Singh, and M. Su. Montage: A grid enabled engine for delivering custom science-grade mosaics on demand, 2004.

[25] B. Berriman, E. Deelman, G. Juve, M. Rynge, and J. Vöckler. The application of cloud computing to scientific workflows: a study of cost and performance. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 371(1983), 2012.

[26] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M. Su, and K. Vahi. Characterization of scientific workflows. In *Workflows in Support of Large-Scale Science, 2008. WORKS 2008. Third Workshop on*, pages 1–10, Nov. 2008.

[27] J. Blythe, S. Jain, E. Deelman, Y. Gil, K. Vahi, A. Mandal, and K. Kennedy. Task scheduling strategies for workflow-based applications in grids. In *Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 02*, CCGRID '05, pages 759–767, Washington, DC, USA, 2005. IEEE Computer Society.

[28] D. Brown, P. Brady, A. Dietz, J. Cao, B. Johnson, and J. McNabb. A case study on the use of workflow technologies for scientific analysis: Gravitational wave data analysis. In *Workflows for e-Science*, pages 39–59. Springer London, 2007.

[29] F. Cao, M. Zhu, and D. Ding. Distributed workflow scheduling under throughput and budget constraints in grid environments. In *Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science, pages 62–80. Springer Berlin Heidelberg, 2014.

[30] H. Cao, H. Jin, X. Wu, S. Wu, and X. Shi. Dagmap: Efficient scheduling for dag grid workflow job. In *Grid Computing, 2008 9th IEEE/ACM International Conference on*, pages 17–24, Sept 2008.

[31] E. Caron, F. Desprez, A. Muresan, and F. Suter. Budget constrained resource allocation for non-deterministic workflows on an iaas cloud. In *Proceedings of the 12th international conference on Algorithms and Architectures for Parallel Processing - Volume Part I*, ICA3PP'12, pages 186–201, 2012.

[32] W. Chen and E. Deelman. Integration of workflow partitioning and resource provisioning. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, CCGRID '12, pages 764–768, 2012.

[33] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[34] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, K. Blackburn, A. Lazzarini, A. Arbree, R. Cavanaugh, and S. Koranda. Mapping abstract complex workflows onto grid environments. *Journal of Grid Computing*, 1(1):25–39, 2003.

[35] E. Deelman, S. Callaghan, E. Field, H. Francoeur, R. Graves, V. Gupta, T. Jordan, C. Kesselman, P. Maechling, G. Mehta, D. Okaya, K. Vahi, and L. Zhao. Managing large-scale workflow execution from resource provisioning to provenance tracking: The cybershake example. In *In Proceedings of the Second IEEE international Conference on E-Science and Grid Computing*, pages 4–6, 2006.

[36] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good. The cost of doing science on the cloud: the montage example. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 50:1–50:12, Piscataway, NJ, USA, 2008.

[37] E. Deelman, G. Singh, M. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, B. Berriman, J. Good, A. Laity, J. Jacob, and D. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Sci. Program.*, 13:219–237, 2005.

[38] S. Desmet, B. Volckaert, and F. De Turck. Autonomous resource-aware scheduling of large-scale media workflows. In *Mechanisms for Autonomous Management of Networks and Services, 4th International Conference on Autonomous Infrastructure, Management and Security, AIMS 2010, Zurich, Switzerland, June 23-25, 2010. Proceedings*, volume 6155 of *Lecture Notes in Computer Science*, pages 50–64. Springer, 2010.

[39] T. Eilam, K. Appleby, J. Breh, G. Breiter, H. Daur, S. Fakhouri, G. Hunt, T. Lu, S. Miller, L. Mummert, J. Pershing, and H. Wagner. Using a utility computing framework to develop utility systems. *IBM Systems Journal*, 43(1):97–120, 2004.

[40] T. Fahringer, R. Prodan, R. Duan, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H. Truong, A. Villazon, and M. Wieczorek. Askalon: a grid application development and computing environment. In *Grid Computing, 2005. The 6th IEEE/ACM International Workshop on*, pages 10 pp.–, November 2005.

[41] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11:115–128, 1996.

[42] Z. Guo and G. Fox. Improving mapreduce performance in heterogeneous network environments and resource utilization. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 714–716, May 2012.

[43] G. Juve, E. Deelman, B. Berriman, B. Berman, and P. Maechling. An evaluation of the cost and performance of scientific workflows on amazon ec2. *J. Grid Comput.*, 10(1):5–21, Mar 2012.

[44] B. Kapil and S. Kamath. Resource aware scheduling in hadoop for heterogeneous workloads based on load estimation. In *Computing, Communications and Networking Technologies (ICCCNT),2013 Fourth International Conference on*, pages 1–5, July 2013.

[45] S. Li, S. Hu, S. Wang, L. Su, T. Abdelzaher, I. Gupta, and R. Pace. Woha: Deadline-aware map-reduce workflow scheduling framework over hadoop clusters. In *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, pages 93–103, June 2014.

[46] Y. Li, H. Zhang, and K. Kim. A power-aware scheduling of mapreduce applications in the cloud. In *Dependable, Autonomic and Secure Computing (DASC), 2011 IEEE Ninth International Conference on*, pages 613–620, 2011.

[47] X. Lin and C. Wu. On scientific workflow scheduling in clouds under budget constraint. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 90–99, Oct 2013.

[48] J. Livny, H. Teonadi, M. Livny, and M. Waldor. High-throughput, kingdom-wide prediction and annotation of bacterial non-coding rnas, 2008.

[49] M. Malawski, K. Figiela, M. Bubak, E. Deelman, and J. Nabrzyski. Cost optimization of execution of multi-level deadline-constrained scientific workflows on clouds. In *Parallel Processing and Applied Mathematics*, Lecture Notes in Computer Science, pages 251–260. Springer Berlin Heidelberg, 2014.

[50] Y. Mao, W. Wu, H. Zhang, and L. Luo. Greenpipe: A hadoop based workflow system on energy-efficient clouds. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 2211–2219, May 2012.

[51] J. Polo, C. Castillo, D. Carrera, Y. Becerra, I. Whalley, M. Steinder, J. Torres, and E. Ayguande. Resource-aware adaptive scheduling for mapreduce clusters. In *Middleware 2011*, volume 7049 of *Lecture Notes in Computer Science*, pages 187–207. Springer Berlin Heidelberg, 2011.

[52] H. Prajapati and V. Shah. Advance reservation based dag application scheduling simulator for grid environment. *International Journal of Computer Applications*, 61(7):45–51, January 2013. Published by Foundation of Computer Science, New York, USA.

[53] B. Rao and L. Reddy. Survey on improved scheduling in hadoop mapreduce in cloud environments. *International Journal of Computer Applications*, 34(9):29–33, Nov 2011.

[54] B. Rao and L. Reddy. Survey on improved scheduling in hadoop mapreduce in cloud environments. *CoRR*, abs/1207.0780, 2012.

[55] R. Sakellariou and H. Zhao. A hybrid heuristic for dag scheduling on heterogeneous systems. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 111–, April 2004.

[56] R. Sakellariou, H. Zhao, E. Tsiakkouri, and M. Dikaiakos. Scheduling workflows with budget constraints. In *Integrated Research in GRID Computing*, pages 189–202. Springer US, 2007.

[57] T. Sandholm and K. Lai. Dynamic proportional share scheduling in hadoop. In *Job Scheduling Strategies for Parallel Processing*, volume 6253 of *Lecture Notes in Computer Science*, pages 110–131. Springer Berlin Heidelberg, 2010.

[58] R. Sedgewick and K. Wayne. *Algorithms*. Pearson Education, 4 edition, 2011.

[59] J. Tan, X. Meng, and L. Zhang. Coupling task progress for mapreduce resource-aware scheduling. In *INFOCOM, 2013 Proceedings IEEE*, pages 1618–1626, April 2013.

[60] Z. Tang, M. Liu, K. Li, and Y. Xu. A mapreduce-enabled scientific workflow framework with optimization scheduling algorithm. In *Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2012 13th International Conference on*, pages 599–604, Dec 2012.

[61] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: The condor experience. *Concurr. Comput. : Pract. Exper.*, 17(2-4):323–356, February 2005.

[62] H. Topcuoglu, S. Hariri, and M. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems, IEEE Transactions on*, 13(3):260–274, Mar 2002.

[63] A. Verma, L. Cherkasova, and R. Campbell. Aria: Automatic resource inference and allocation for mapreduce environments. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ICAC '11, pages 235–244, New York, NY, USA, 2011. ACM.

[64] Y. Wang and W. Shi. On optimal parallel scheduling algorithms with budget&deadline constraints for fork&join workflows in clouds. In *ICON*, pages 245–252, Dec 2013.

[65] Y. Wang and W. Shi. On scheduling algorithms for mapreduce jobs in heterogeneous clouds with budget constraints. In *Principles of Distributed Systems*, volume 8304 of *Lecture Notes in Computer Science*, pages 251–265. Springer International Publishing, 2013.

[66] Y. Wang and W. Shi. Budget-driven scheduling algorithms for batches of mapreduce jobs in heterogeneous clouds. *Cloud Computing, IEEE Transactions on*, 2(3):306–319, July 2014.

[67] J. Wolf, D. Rajan, K. Hildrum, R. Khandekar, V. Kumar, S. Parekh, K. Wu, and A. Balmin. Flex: A slot allocation scheduling optimizer for mapreduce workloads. In *Middleware 2010*, volume 6452 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin Heidelberg, 2010.

[68] J. Xie, S. Yin, X. Ruan, Z. Ding, Y. Tian, J. Majors, A. Manzanares, and X. Qin. Improving mapreduce performance through data placement in heterogeneous hadoop clusters. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–9, April 2010.

[69] H. You, C. Yang, and J. Huang. A load-aware scheduler for mapreduce framework in heterogeneous cloud environments. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 127–132, 2011.

[70] J. Yu and R. Buyya. A taxonomy of workflow management systems for grid computing. *Journal of Grid Computing*, 3(3-4):171–200, 2005.

[71] J. Yu and R. Buyya. A budget constrained scheduling of workflow applications on utility grids using genetic algorithms. In *Workflows in Support of Large-Scale Science, 2006. WORKS '06. Workshop on*, pages 1–10, June 2006.

[72] J. Yu and R. Buyya. Scheduling scientific workflow applications with deadline and budget constraints using genetic algorithms. *Sci. Program.*, 14(3,4):217–230, Dec 2006.

[73] J. Yu, R. Buyya, and K. Ramamohanarao. Workflow scheduling algorithms for grid computing. In *Metaheuristics for Scheduling in Distributed Computing Environments*, volume 146 of *Studies in Computational Intelligence*, pages 173–214. Springer Berlin Heidelberg, 2008.

[74] J. Yu, R. Buyya, and C. Tham. Cost-based scheduling of scientific workflow applications on utility grids. In *e-Science and Grid Computing, 2005. First International Conference on*, pages 8 pp.–147, July 2005.

[75] Y. Yuan, X. Li, Q. Wang, and X. Zhu. Deadline division-based heuristic for cost optimization in workflow scheduling. *Inf. Sci.*, 179(15):2562–2575, July 2009.

[76] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.

[77] L. Zeng, B. Veeravalli, and X. Li. Scalestar: Budget conscious scheduling precedence-constrained many-task workflow applications in cloud. In *Proceedings of the 2012 IEEE 26th International Conference on Advanced Information Networking and Applications*, AINA '12, pages 534–541, 2012.

[78] C. Zhang and H. De Sterck. Cloudbatch: A batch job queuing system on clouds with hadoop and hbase. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 368–375, November 2010.

[79] Z. Zhang, L. Cherkasova, and B. Loo. Performance modeling of mapreduce jobs in heterogeneous cloud environments. In *Proceedings of the 2013 IEEE Sixth International Conference on Cloud Computing*, CLOUD '13, pages 839–846, Washington, DC, USA, 2013. IEEE Computer Society.

[80] Z. Zhang, L. Cherkasova, and B. Loo. Exploiting cloud heterogeneity for optimized cost/performance mapreduce processing. In *Proceedings of the Fourth International Workshop on Cloud Data and Platforms*, CloudDP '14, pages 1:1–1:6, New York, NY, USA, 2014. ACM.

[81] W. Zheng and R. Sakellariou. Budget-deadline constrained workflow planning for admission control in market-oriented environments. In *Proceedings of the 8th international conference on Economics of Grids, Clouds, Systems, and Services*, GECON'11, pages 105–119, Berlin, Heidelberg, 2012. Springer-Verlag.

[82] W. Zheng and R. Sakellariou. Budget-deadline constrained workflow planning for admission control. *Journal of Grid Computing*, 11(4):633–651, 2013.