Hi-WAY: Execution of Scientific Workflows on Hadoop YARN

Marc Bux
Humboldt-Universität zu Berlin
Berlin, Germany
bux@informatik.hu-berlin.de

Jörgen Brandt
Humboldt-Universität zu Berlin
Berlin, Germany
joergen.brandt@hu-berlin.de

Carl Witt
Humboldt-Universität zu Berlin
Berlin, Germany
wittcarx@informatik.hu-berlin.de

Jim Dowling
Swedish Institute of Computer Science (SICS)
Stockholm, Sweden
jdowling@sics.se

Ulf Leser
Humboldt-Universität zu Berlin
Berlin, Germany
leser@informatik.hu-berlin.de

ABSTRACT

Scientific workflows provide a means to model, execute, and exchange the increasingly complex analysis pipelines necessary for today's data-driven science. However, existing scientific workflow management systems (SWfMSs) are often limited to a single workflow language and lack adequate support for large-scale data analysis. On the other hand, current distributed dataflow systems are based on a semi-structured data model, which makes integration of arbitrary tools cumbersome or forces re-implementation. We present the scientific workflow execution engine Hi-WAY, which implements a strict black-box view on tools to be integrated and data to be processed. Its generic yet powerful execution model allows Hi-WAY to execute workflows specified in a multitude of different languages. Hi-WAY compiles workflows into schedules for Hadoop YARN, harnessing its proven scalability. It allows for iterative and recursive workflow structures and optimizes performance through adaptive and data-aware scheduling. Reproducibility of workflow executions is achieved through automated setup of infrastructures and re-executable provenance traces. In this application paper we discuss limitations of current SWfMSs regarding scalable data analysis, describe the architecture of Hi-WAY, highlight its most important features, and report on several large-scale experiments from different scientific domains.

1. INTRODUCTION

Recent years have brought an unprecedented influx of data across many fields of science. In genomics, for instance, the latest generation of genomic sequencing machines can handle up to 18,000 human genomes per year [41], generating about 50 terabytes of sequence data per week. Similarly, astronomical research facilities and social networks are also generating terabytes of data per week [34]. To synthesize succinct results from these readouts, scientists assemble complex graph-structured analysis pipelines, which chain a multitude of different tools for transforming, filtering, and aggregating the data [18]. The tools used within these pipelines are implemented by thousands of researchers around the world, rely on domain-specific data exchange formats, and are updated frequently (e.g., [27, 31]). Consequently, easy ways of assembling and altering analysis pipelines are of utmost importance [11]. Moreover, to ensure reproducibility of scientific experiments, analysis pipelines should be easily sharable and execution traces must be accessible [12].

Systems fulfilling these requirements are generally called scientific workflow management systems (SWfMSs). From an abstract perspective, scientific workflows are compositions of sequential and concurrent data processing tasks, whose order is determined by data interdependencies [36]. Tasks are treated as black boxes and can therefore range from a simple shell script over a local command-line tool to an external service call. Also, the data exchanged by tasks is typically not parsed by the SWfMS but only forwarded according to the workflow structure. While these black-box data and operator models prohibit the automated detection of potentials for data-parallel execution, their strengths lie in their flexibility and the simplicity of integrating external tools.

To deal with the ever-increasing amounts of data prevalent in today's science, SWfMSs have to provide support for parallel and distributed storage and computation [26]. However, while extensible distributed computing frameworks like Hadoop YARN [42] or MESOS [19] keep developing rapidly, established SWfMSs, such as Taverna [47] or Pegasus [13] are not able to keep pace. A particular problem is that most SWfMSs tightly couple their own custom workflow language to a specific execution engine, which can be difficult to configure and maintain alongside other execution engines that are already present on the cluster. In addition, many of these execution engines fail to keep up with the latest developments in distributed computing, e.g., by storing data in a central location, or by neglecting data locality and heterogeneity of distributed resources during workflow

©2017, Copyright is with the authors. Published in Proc. 20th International Conference on Extending Database Technology (EDBT), March 21-24, 2017 - Venice, Italy: ISBN 978-3-89318-073-8, on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0

scheduling [10]. Furthermore, despite reproducibility being advocated as a major strength of scientific workflows, most systems focus only on sharing workflows, disregarding the provisioning of input data and setup of the execution environment [15, 33]. Finally, many systems severely limit the expressiveness of their workflow language, e.g., by disallowing conditional or recursive structures. While the scientific workflow community is becoming increasingly aware of these issues (e.g., [8, 33, 50]), to date only isolated, often domain-specific solutions addressing only subsets of these problems have been proposed (e.g., [6, 14, 38]).

At the same time, support for many of these features has been implemented in several recently developed distributed dataflow systems, such as Spark [49] or Flink [5]. However, such systems employ a semi-structured white-box (e.g., key-value-based) data model to automatically partition and parallelize dataflows. Unfortunately, a structured data model impedes the flexibility in workflow design when integrating external tools that read and write file-based data. To circumvent this problem, additional glue code for transforming to and from the structured data model has to be provided. This introduces unnecessary overhead in terms of time required for implementing the glue code as well as for the necessary data transformations at runtime [48].

In this application paper, we present the **Hi**-WAY **W**orkflow **A**pplication master for **Y**ARN. Technically, Hi-WAY is yet another application master for YARN. Conceptually, it is a (surprisingly thin) layer between scientific workflow specifications expressed in different languages and Hadoop YARN. It emphasizes data center compatibility by being able to run on YARN installations of any size and type of underlying infrastructure. Compared to other SWfMSs, Hi-WAY brings the following specific features.

1. *Multi-language* support. Hi-WAY employs a generic yet powerful execution model. It has no own specification language, but instead comes with an extensible language interface and built-in support for multiple workflow languages, such as Cuneiform [8], Pegasus DAX [13], and Galaxy [17] (see Section 3.2).

2. *Iterative* workflows. Hi-WAY's execution model is expressive enough to support data-dependent control-flow decisions. This allows for the design of conditional, iterative, and recursive structures, which are increasingly common in distributed dataflows (e.g., [28]), yet are just beginning to emerge in scientific workflows (see Section 3.3).

3. *Performance* gains through *adaptive* scheduling. Hi-WAY supports various workflow scheduling algorithms. It utilizes statistics of earlier workflow executions to estimate the resource requirements of tasks awaiting execution and exploit heterogeneity in the computational infrastructures during scheduling. Also, Hi-WAY supports adaption of schedules to both data locality and resource availability (see Section 3.4).

4. *Reproducible* experiments. Hi-WAY generates comprehensive provenance traces, which can be directly re-executed as workflows (see Section 3.5). Also, Hi-WAY uses Chef [2] and Karamel [1] for specifying automated setups of a workflow's software requirements and input data, including (if necessary) the installation of Hi-WAY and Hadoop (see Section 3.6).

5. *Scalable* execution. By employing Hadoop YARN as its underlying execution engine, Hi-WAY harnesses its scalable resource management, fault tolerance, and distributed file management (see Section 3.1).

While some of these features have been briefly outlined in the context of a demonstration paper [9], this is the first comprehensive description of Hi-WAY.

The remainder of this paper is structured as follows: Section 2 gives an overview of related work. Section 3 presents the architecture of Hi-WAY and gives detailed descriptions of the aforementioned core features, which are highlighted in italic font throughout the rest of the document. Section 4 describes several experiments showcasing these feature in real-life workflows on both local clusters and cloud computing infrastructure. Section 5 concludes the paper.

2. RELATED WORK

Projects with goals similar to Hi-WAY can be separated into two groups. The first group of systems comprises traditional SWfMSs, which, like Hi-WAY, employ black-box data and operator models. The second group encompasses distributed dataflow systems developed to process mostly structured or semi-structured (white-box) data. For a comprehensive overview of data-intensive scientific workflow management, readers are referred to [10] and [26].

2.1 Scientific Workflow Management

The SWfMS Pegasus [13] emphasizes *scalability*, utilizing HTCondor as its underlying execution engine. It enforces the usage of its own XML-based workflow language called DAX. Pegasus supports a number of scheduling policies, all of which are static, yet some of which can be considered *adaptive* (such as HEFT [39]). Finally, Pegasus does not allow for *iterative* workflow structures, since every task invocation has to be explicitly described in the DAX file. In contrast to Hi-WAY, Pegasus does not provide any means of reproducing scientific experiments across datacenters. Hi-WAY complements Pegasus by enabling Pegasus workflows to be run on top of Hadoop YARN, as outlined in Section 3.2.

Taverna [47] is an established SWfMS that focuses on usability, providing a graphical user interface for workflow design and monitoring as well as a comprehensive collection of pre-defined tools and remote services. Taverna emphasizes *reproducibility* of experiments and workflow sharing by integrating the public myExperiment workflow repository [16], in which over a thousand Taverna workflows have been made available. However, Taverna is mostly used to integrate web services and short-running tasks and thus does not support *scalable* distribution of workload across several worker nodes or any *adaptive* scheduling policies.

Galaxy [17] is a SWfMS that provides a web-based graphical user interface, an array of built-in libraries with a focus on computational biology, and a repository for sharing workflows and data. CloudMan [3] extends Galaxy with limited *scalability* by enabling Galaxy clusters of up to 20 nodes to be set up on Amazon's EC2 through an easy-to-use web interface. Unfortunately, Galaxy neither supports *adaptive* scheduling nor *iterative* workflow structures. Similar to Pegasus and as described in Section 3.2, Hi-WAY complements Galaxy by allowing exported Galaxy workflows to be run on Hadoop YARN. For a comparative evaluation of Hi-WAY and Galaxy CloudMan, refer to Section 4.2.

Text-based parallel scripting languages like Makeflow [4], Snakemake [23], or Swift [45] are more light-weight alternatives to full-fledged SWfMSs. Swift [45] provides a functional scripting language that facilitates the design of inherently data-parallel workflows. Conversely, Snakemake [23] and Makeflow [4] are inspired by the build automation tool GNU make, enabling a goal-driven assembly of workflow scripts. All of these system have in common that they support the *scalable* execution of implemented workflows on distributed infrastructures, yet disregard other features typically present in SWfMSs, such as *adaptive* scheduling mechanisms or support for *reproducibility*.

Nextflow [38] is a recently proposed SWfMS [14], which brings its own domain-specific language. In Nextflow, software dependencies can be provided in the form of Docker or Shifter containers, which facilitates the design of *reproducible* workflows. Nextflow enables *scalable* execution by supporting several general-purpose batch schedulers. Compared to Hi-WAY, execution traces are less detailed and not re-executable. Furthermore, Nextflow does not exploit data-aware and *adaptive* scheduling potentials.

Toil [43] is a *multi-language* SWfMS that supports *scalable* workflow execution by interfacing with several distributed resource management systems. Its supported languages include the Common Workflow Language (CWL) [6], a YAML-based workflow language that unifies concepts of various other languages, and a custom Python-based DSL that supports the design of *iterative* workflows. Similar to Nextflow, Toil enables sharable and *reproducible* workflow runs by allowing tasks to be wrapped in re-usable Docker containers. In contrast to Hi-WAY, Toil does not gather comprehensive provenance and statistics data and, consequently, does not support any means of *adaptive* workflow scheduling.

2.2 Distributed Dataflows Systems

Distributed dataflow systems like Spark [49] or Flink [5] have recently achieved strong momentum both in academia and in industry. These systems operate on semi-structured data and support different programming models, such as SQL-like expression languages or real-time stream processing. Departing from the black-box data model along with natively supporting concepts like data streaming and in-memory computing allows these systems to in many cases execute even sequential processing steps in parallel and circumvent the materialization of intermediate data on the hard disk. It also enables the automatic detection and exploitation of potentials for data parallelism. However, the resulting gains in *performance* come at the cost of reduced flexibility for workflow designers. This is especially problematic for scientists from domains other than the computational sciences. Since integrating external tools processing unstructured, file-based data is often tedious and undermines the benefits provided by dataflow systems, a substantial amount of researchers continue to rely on traditional scripting and programming languages to tackle their data-intensive analysis tasks (e.g., [27, 31]).

Tez [32] is an application master for YARN that enables the execution of DAGs comprising map, reduce, and custom tasks. Being a low-level library intended to be interfaced by higher-level applications, external tools consuming and producing file-based data need to be wrapped in order to be used in Tez. For a comparative evaluation between Hi-WAY and Tez, see Section 4.1.

While Tez runs DAGs comprising mostly map and reduce tasks, Hadoop workflow schedulers like Oozie [20] or Azkaban [35] have been developed to schedule DAGs consisting mostly of Hadoop jobs (e.g., MapReduce, Pig, Hive) on a Hadoop installation. In Oozie, tasks composing a workflow are transformed into a number of MapReduce jobs at runtime. When used to run arbitrary scientific workflows, systems like Oozie or Azkaban either introduce unnecessary overhead by wrapping the command-line tasks into degenerate MapReduce jobs or do not dispatch such tasks to Hadoop, but run them locally instead.

Chiron [30] is a *scalable* workflow management system in which data is represented as relations and workflow tasks implement one out of six higher-order functions (e.g., map, reduce, and filter). This departure from the black-box view on data inherent to most SWfMSs enables Chiron to apply concepts of database query optimization to optimize *performance* through structural workflow reordering [29]. In contrast to Hi-WAY, Chiron is limited to a single, custom, XML-based workflow language, which does not support *iterative* workflow structures. Furthermore, while Chiron, like Hi-WAY, is one of few systems in which a workflow's (incomplete) provenance data can be queried during execution of that same workflow, Chiron does not employ this data to perform any *adaptive* scheduling.

3. ARCHITECTURE

Hi-WAY utilizes Hadoop as its underlying system for the management of both distributed computational resources and storage (see Section 3.1). It comprises three main components, as shown in Figure 1. First, the Workflow Driver parses a scientific workflow specified in any of the supported workflow languages and reports any discovered tasks to the Workflow Scheduler (see Sections 3.2 and 3.3). Secondly, the Workflow Scheduler assigns tasks to compute resources provided by Hadoop YARN according to a selected scheduling policy (see Section 3.4). Finally, the Provenance Manager gathers comprehensive provenance and statistics information obtained during task and workflow execution, handling their long-term storage and providing the Workflow Scheduler with up-to-date statistics on previous task executions (see Section 3.5). Automated installation routines for the setup of Hadoop, Hi-WAY, and selected workflows are described in Section 3.6.

3.1 Interface with Hadoop YARN

Hadoop version 2.0 introduced the resource management component YARN along with the concept of job-specific application masters (AMs), increasing scalability beyond 4,000 computational nodes and enabling native support for non-MapReduce AMs. Hi-WAY seizes this concept by providing its own AM that interfaces with YARN.

To submit workflows for execution, Hi-WAY provides a light-weight client program. Each workflow that is launched from a client results in a separate instance of a Hi-WAY AM being spawned in its own container. Containers are YARN's basic unit of computation, encapsulating a fixed amount of virtual processor cores and memory which can be specified in Hi-WAY's configuration. Having one dedicated AM per workflow results in a distribution of the workload associated with workflow execution management and is therefore required to fully unlock the *scalability* potential provided by Hadoop.

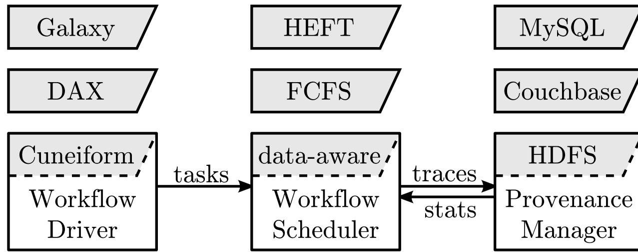

Figure 1: The architecture of the Hi-WAY application master: The Workflow Driver, described in Sections 3.2 and 3.3, parses a textual workflow file, monitors workflow execution, and notifies the Workflow Scheduler whenever it discovers new tasks. For tasks that are ready to be executed, the Workflow Scheduler, presented in Section 3.4, assembles a schedule. Provenance and statistics data obtained during workflow execution are handled by the Provenance Manager (see Section 3.5) and can be stored in a local file as well as a MySQL or Couchbase database.

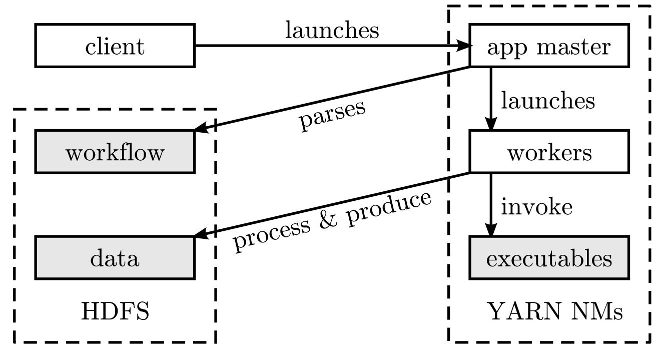

Figure 2: Functional interaction between the components of Hi-WAY and Hadoop (white boxes; see Section 3.1) as well as further requirements for running workflows (gray boxes; see Section 3.6). A workflow is launched from a client application, resulting in a new instance of a Hi-WAY AM within a container provided by one of YARN's NodeManagers (NMs). This AM parses the workflow file residing in HDFS and prompts YARN to spawn additional worker containers for tasks that are ready to run. During task execution, these worker containers obtain input data from HDFS, invoke locally available executables, and generate output data, which is placed in HDFS for use by other worker containers.

For any of a workflow's tasks that await execution, the Hi-WAY AM responsible for running this particular workflow then requests an additional worker container from YARN. Once allocated, the lifecycle of these worker containers involves (i) obtaining the task's input data from HDFS, (ii) invoking the commands associated with the task, and (iii) storing any generated output data in HDFS for consumption by other containers executing tasks in the future and possibly running on other compute nodes. Figure 2 illustrates this interaction between Hi-WAY's client application, AM and worker containers, as well as Hadoop's HDFS and YARN components.

Besides having dedicated AM instances per workflow, another prerequisite for *scalable* workflow execution is the ability to recover from failures. To this end, Hi-WAY is able to re-try failed tasks, requesting YARN to allocate the additional containers on different compute nodes. Also, data processed and produced by Hi-WAY persists through the crash of a storage node, since Hi-WAY exploits the redundant file storage of HDFS for any input, output, and intermediate data associated with a workflow.

3.2 Workflow Language Interface

Hi-WAY sunders the tight coupling of scientific workflow languages and execution engines prevalent in established SWfMSs. For this purpose, its Workflow Driver (see Section 3.3) provides an extensible, *multilingual* language interface, which is able to interpret scientific workflows written in a number of established workflow languages. Currently, four scientific workflow languages are supported: (i) the textual workflow language Cuneiform [8], (ii) DAX, which is the XML-based workflow language of the SWfMS Pegasus [13], (iii) workflows exported from the SWfMS Galaxy [17], and (iv) Hi-WAY provenance traces, which can also be interpreted as scientific workflows (see Section 3.5).

Cuneiform [8] is a minimal workflow language that supports direct integration of code written in a large range of external programming languages (e.g., Bash, Python, R, Perl, Java). It supports *iterative* workflows and treats tasks as black boxes, allowing the integration of various tools and libraries independent of their programming API. Cuneiform facilitates the assembly of highly parallel data processing pipelines by providing a range of second-order functions extending beyond map and reduce operations.

DAX [13] is Pegasus' built-in workflow description language, in which workflows are specified in an XML file. Contrary to Cuneiform, DAX workflows are static, explicitly specifying every task to be invoked and every file to be processed or produced by these tasks during workflow execution. Consequently, DAX workflows can become quite large and are not intended to be read or written by workflow developers directly. Instead, APIs enabling the generation of DAX workflows are provided for Java, Python, and Perl.

Workflows in the web-based SWfMS Galaxy [17] can be created using a graphical user interface, in which the tasks comprising the workflow can be selected from a large range of software libraries that are part of any Galaxy installation. This process of workflow assembly results in a static workflow graph that can be exported to a JSON file, which can then be interpreted by Hi-WAY. In workflows exported from Galaxy, the workflow's input files are not explicitly designated. Instead, input ports serve as placeholders for the input files, which are resolved interactively when the workflow is committed to Hi-WAY for execution.

In addition to these workflow languages, Hi-WAY can easily be extended to parse and execute other non-interactive workflow languages. For non-*iterative* languages, one only needs to extend the Workflow Driver class and implement the method that parses a textual workflow file to determine the tasks and data dependencies composing the workflow.

3.3 Iterative Workflow Driver

On execution onset, the Workflow Driver parses the workflow file to determine inferable tasks along with the files they

process and produce. Any discovered tasks are passed to the Workflow Scheduler, which then assembles a schedule and creates container requests whenever a task's data dependencies are met. Subsequently, the Workflow Driver supervises workflow execution, waiting for container requests to be fulfilled or for tasks to terminate. In the former case, the Workflow Driver requests the Workflow Scheduler to choose a task to be launched in that container. In the latter case, the Workflow Driver registers any newly produced data, which may induce new tasks becoming ready for execution and thus new container requests to be issued.

One of Hi-WAY's core strengths is its ability to interpret *iterative* workflows, which may contain unbounded loops, conditionals, and recursive tasks. In such *iterative* workflows, the termination of a task may entail the discovery of entirely new tasks. For this reason, the Workflow Driver dynamically evaluates the results of completed tasks, forwarding newly discovered tasks to the Workflow Scheduler, similar to during workflow parsing. See Figure 3 for a visualization of the Workflow Driver's execution model.

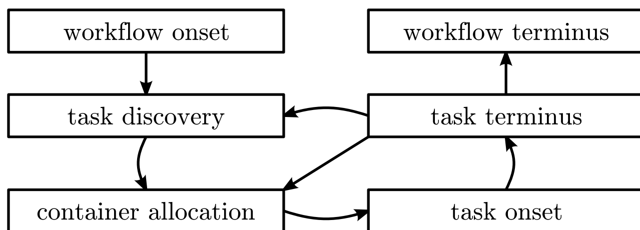

Figure 3: The *iterative* Workflow Driver's execution model. A workflow is parsed, entailing the discovery of tasks as well as the request for and eventual allocation of containers for ready tasks. Upon termination of a task executed in an allocated container, previously discovered tasks might become ready (resulting in new container requests), new tasks might be discovered, or the workflow might terminate.

As an example for an *iterative* workflow, consider an implementation of the k-means clustering algorithm commonly encountered in machine learning applications. k-means provides a heuristic for partitioning a number of data points into k clusters. To this end, over a sequence of parallelizable steps, an initial random clustering is iteratively refined until convergence is reached. Only by means of conditional task execution and unbounded iteration can this algorithm be implemented as a workflow, which underlines the importance of such *iterative* control structures in scientific workflows. The implementation of the k-means algorithm as a Cuneiform workflow has been published in [9].

3.4 Workflow Scheduler

Determining a suitable assignment of tasks to compute nodes is called workflow scheduling. To this end, the Workflow Scheduler receives tasks discovered by the Workflow Driver, from which it builds a schedule and creates container requests. Based on this schedule, the Workflow Scheduler selects a task for execution whenever a container has been allocated. This higher-level scheduler is different to YARN's internal schedulers, which, at a lower level, determine how to distribute resources between multiple users and applications. Hi-WAY provides a selection of workflow scheduling policies that optimize *performance* for different workflow structures and computational architectures.

Most established SWfMSs employ a first-come-first-served (FCFS) scheduling policy in which tasks are placed at the tail of a queue, from whose head they are removed and dispatched for execution whenever new resources become available. While Hi-WAY supports FCFS scheduling as well, its default scheduling policy is a data-aware scheduler intended for I/O-intensive workflows. The data-aware scheduler minimizes data transfer by assigning tasks to compute nodes based on the amount of input data that is already present locally. To this end, whenever a new container is allocated, the data-aware scheduler skims through all tasks pending execution, from which it selects the task with the highest fraction of input data available locally (in HDFS) on the compute node hosting the newly allocated container.

In contrast to data-aware and FCFS scheduling, static scheduling policies employ a pre-built schedule, which dictates how the tasks composing a workflow are to be assigned to available compute nodes. When configured to employ a static scheduling policy, Hi-WAY's Workflow Scheduler assembles this schedule at the beginning of workflow execution and enforces containers to be placed on specific compute nodes according to this schedule. A basic static scheduling policy supported by Hi-WAY is a round-robin scheduler that assigns tasks in turn, and thus in equal numbers, to the available compute nodes.

In addition to these scheduling policies, Hi-WAY is also able to employ *adaptive* scheduling in which the assignment of tasks to compute nodes is based on continually updated runtime estimates and is therefore adapted to the computational infrastructure. To determine such runtime estimates, the Provenance Manager, which is responsible for gathering, storing, and providing provenance and statistics data (see Section 3.5), supplies the Workflow Scheduler with exhaustive statistics. For instance, when deciding whether to assign a task to a newly allocated container on a certain compute node, the Workflow Scheduler can query the Provenance Manager for (i) the observed runtimes of earlier tasks of the same signature (i.e., invoking the same tools) running on either the same or other compute nodes, (ii) the names and sizes of the files being processed in these tasks, and (iii) the data transfer times for obtaining this input data.

If available, based on this information the Workflow Scheduler is able to determine runtime estimates for running any task on any machine. In order to quickly adapt to performance changes in the computational infrastructure, the current strategy for computing these runtime estimates is to always use the latest observed runtime. If no runtimes have been observed yet for a particular task-machine-assignment, a default runtime of zero is assumed to encourage trying out new assignments and thus obtain a more complete picture of which task performs well on which machine.

To make use of these runtime estimates, Hi-WAY supports heterogeneous earliest finish time (HEFT) [39] scheduling. HEFT exploits heterogeneity in both the tasks to be executed as well as the underlying computational infrastructure. To this end, it uses runtime estimates to rank tasks by the expected time required from task onset to workflow terminus. By decreasing rank, tasks are assigned to compute nodes with a favorable runtime estimate, i.e., critical tasks with a longer time to finish are placed on the best-performing nodes first.

Since static schedulers like round-robin and HEFT require the complete invocation graph of a workflow to be deductible at the onset of computation, static scheduling can not be used in conjunction with workflow languages that allow *iterative* workflows. Hence, the latter two (static) scheduling policies are not compatible with Cuneiform workflows (see Section 3.3).

Additional (non-static) adaptive scheduling policies are in the process of being integrated and will be described and evaluated in a separate manuscript. However, note that due to the black-box operator model, scheduling policies may not conduct structural alterations to the workflow automatically, as commonly found in database query optimization.

3.5 Provenance Manager

The Provenance Manager surveys workflow execution and registers events at different levels of granularity. First, it traces events at the workflow level, including the name of the workflow and its total execution time. Secondly, it logs events for each task, e.g., the commands invoked to spawn the task, its makespan, its standard output and error channels and the compute node on which it ran. Thirdly, it stores events for each file consumed and produced by a task. This includes its size and the time it took to move the file between HDFS and the local file system. All of this provenance data is supplemented with timestamps as well as unique identifiers and stored as JSON objects in a trace file in HDFS, from where it can be accessed by other instances of Hi-WAY.

Since this trace file holds information about all of a workflow's tasks and data dependencies, it can be interpreted as a workflow itself. Hi-WAY promotes *reproducibility* of experiments by being able to parse and execute such workflow traces directly through its Workflow Driver, albeit not necessarily on the same compute nodes. Hence, workflow trace files generated by Hi-WAY constitute a fourth supported workflow language.

Evidently, the amount of workflow traces can become difficult to handle for heavily-used installations of Hi-WAY with thousands of trace files or more. To cope with such high volumes of data, Hi-WAY provides prototypical implementations for storing and accessing this provenance data in a MySQL or Couchbase database as an alternative to storing trace files in HDFS. The usage of a database for storing this provenance data brings the added benefit of facilitating manual queries and aggregation.

3.6 Reproducible Installation

The properties of the scientific workflow programming model with its black-box data and operator models, as well as the usage of Hadoop for resource management and data distribution, both dictate requirements for workflow designers (for an illustration of some of these requirements, refer to Figure 2). First, all of a workflow's software dependencies (executables, software libraries, etc.) have to be available on each of the compute nodes managed by YARN, since any of the tasks composing a workflow could be assigned to any compute node. Secondly, any input data required to run the workflow has to be placed in HDFS or made locally available on all nodes.

To set up an installation of Hi-WAY and Hadoop, configuration routines are available online in the form of Chef recipes. Chef is a configuration management software for the automated setup of computational infrastructures [2]. These Chef installation routines, called recipes, allow for the setup of standalone or distributed Hi-WAY installations, either on local machines or in public compute clouds such as Amazon's EC2. In addition, recipes are available for setting up a large variety of execution-ready workflows. This includes obtaining their input data, placing it in HDFS, and installing any software dependencies required to run the workflow. Besides providing a broad array of use cases, these recipes enable *reproducibility* of all the experiments outlined in Section 4. The procedure of running these Chef recipes via the orchestration engine Karamel [1] to set up a distributed Hi-WAY execution environment along with a selection of workflows is described in [9] and on http://saasfee.io.

Note that this means of providing *reproducibility* exists in addition to the executable provenance traces described in Section 3.5. However, while the Chef recipes are well-suited for reproducing experiments across different research groups and compute clusters, the executable trace files are intended for use on the same cluster, since running a trace file requires input data to be located and software requirements to be available just like during the workflow run from which the trace file was derived.

4. EVALUATION

We conducted a number of experiments in which we evaluated Hi-WAY's core features of *scalability*, *performant* execution, and *adaptive* workflow scheduling. The remaining properties (support for *multilingualism*, *reproducible* experiments, and *iterative* workflows) are achieved by design. The workflows outlined in this section are written in three different languages and can be automatically set up (including input data) and run on Hi-WAY with only a few clicks following the procedure described in Section 3.6.

Across the experiments described here, we executed relevant workflows from different areas of research on both virtual clusters of Amazon's EC2 and local computational infrastructure. Section 4.1 outlines two experiments in which we analyze the *scalability* and *performance* behavior of Hi-WAY when increasing the number of available computational nodes to very large numbers. In Section 4.2, we then describe an experiment that contrasts the *performance* of running a computationally intensive Galaxy workflow on both Hi-WAY and Galaxy CloudMan. Finally, in Section 4.3 we report on an experiment in which the effect of provenance data on *adaptive* scheduling is evaluated. Table 1 gives an overview of all experiments described in this section.

4.1 Scalability / Genomics

For evaluating the *scalability* of Hi-WAY, we employed a single nucleotide variant calling workflow [31], which determines and characterizes genomic variants in a number of genomes. The input of this workflow are genomic reads emitted from a next-generation sequencing machine, which are aligned against a reference genome in the first step of the workflow using Bowtie 2 [24]. In the second step of the workflow, alignments are sorted using SAMtools [25] and genomic variants are determined using VarScan [22]. Finally, detected variants are annotated using the ANNOVAR [44] toolkit. Input data, in the form of genomic reads, was obtained from the 1000 Genomes Project [37].

In a first experiment we implemented this workflow in both Cuneiform and Tez. We ran both Hi-WAY and Tez on a Hadoop installation set up on a local cluster comprising

Table 1: Overview of conducted experiments, their evaluation goals and the section in which they are outlined.

workflow	domain	language	scheduler	infrastructure	runs	evaluation	section
SNV Calling	genomics	Cuneiform	data-aware	24 Xeon E5-2620	3	*performance*, *scalability*	4.1
SNV Calling	genomics	Cuneiform	FCFS	128 EC2 m3.large	3	*scalability*	4.1
RNA-seq	bioinformatics	Galaxy	data-aware	6 EC2 c3.2xlarge	5	*performance*	4.2
Montage	astronomy	DAX	HEFT	8 EC2 m3.large	80	*adaptive* scheduling	4.3

24 compute nodes connected via a one gigabit switch. Each compute node provided 24 gigabyte of memory as well as two Intel Xeon E5-2620 processors with 24 virtual cores. This resulted in a maximum of 576 concurrently running containers, of which each one was provided with its own virtual processor core and one gigabyte of memory.

The results of this experiment are illustrated in Figure 4. *Scalability* beyond 96 containers was limited by network bandwidth. The results indicate that Hi-WAY *performs* comparably to Tez while network resources are sufficient, yet *scales* favorably in light of limited network resources due to its data-aware scheduling policy, which reduced data transfer by preferring to assign the data-intensive reference alignment tasks to containers on compute nodes with a locally available replicate of the input data. However, probably the most important finding of this experiment was that the implementation of the workflow in Cuneiform resulted in very little code and was finished in a few days, whereas it took several weeks and a lot of code in Tez.

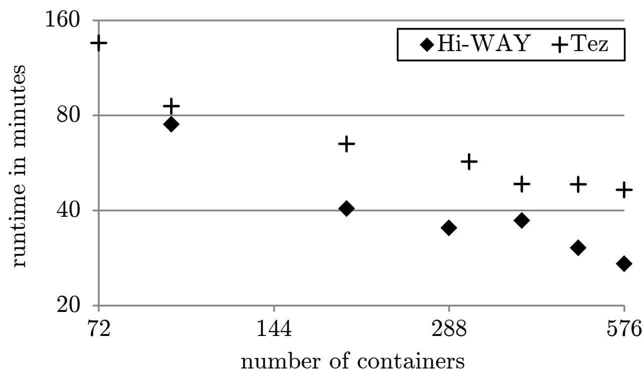

Figure 4: Mean runtimes of the variant calling workflow with increasing number of containers. Note that both axes are in logarithmic scale.

In a second experiment, we increased the volume of input data while at the same time reducing network load by (i) using additional genomic read files from the 1000 Genomes Project, (ii) compressing intermediate alignment data using CRAM referential compression [25], and (iii) obtaining input read data during workflow execution from the Amazon S3 bucket of the 1000 Genomes Project instead of storing them on the cluster in HDFS.

In the process of this second experiment, the workflow was first run using a single worker node, processing a single genomic sample comprising eight files, each about one gigabyte in size, thus amounting to eight gigabytes of input data in total. In subsequent runs, we then repeatedly doubled the number of worker nodes and volume of input data. In the last run (after seven duplications), the computational infrastructure consisted of 128 worker nodes, whereas the workflow's input data comprised 128 samples of eight roughly gigabyte-sized files each, amounting to a total volume of more than a terabyte of data.

The experiment was run three times on virtual clusters of Amazon's EC2. To investigate potential effects of datacenter locality on workflow runtime (which we did not observe during the experiment), these clusters were set up in different EC2 regions – once in the EU West (Ireland) and twice in the US East (North Virginia) region. Since we intended to analyze the *scalability* of Hi-WAY, we isolated the Hi-WAY AM from the worker threads and Hadoop's master threads. To this end, dedicated compute nodes were provided for (i) the Hi-WAY AM, running in its own YARN container, and (ii) the two Hadoop master threads (HDFS's NameNode and YARN's ResourceManager). All compute nodes – the two master nodes and all of the up to 128 worker nodes – were configured to be of type m3.large, each providing two virtual processing cores, 7.5 gigabytes of main memory, and 32 gigabytes of local SSD storage.

All of the experiment runs were set up automatically using Karamel [1]. Over the course of the experiment we determined the runtime of the workflow. Furthermore, the CPU, I/O, and network performance of the master and worker nodes was monitored during workflow execution using the Linux tools *uptime*, *ifstat*, and *iostat*. Since the workflow's tasks required the whole memory provided by a single compute node, we configured Hi-WAY to only allow a single container per worker node at the same time, enabling multithreading for tasks running within that container whenever possible. Hi-WAY was configured to utilize the basic FCFS queue scheduler (see Section 3.4). Other than that, both Hi-WAY and Hadoop were set up with default parameters.

The average of measured runtimes with steadily increasing amounts of both compute nodes and input data is displayed in Table 2 and Figure 5. The regression curve indicates near-linear *scalability*: The doubling of input data and the associated doubling of workload is almost fully offset by a doubling of worker nodes. This is even true for the maximum investigated cluster size of 128 nodes, in which a terabyte of genomic reads was aligned and analyzed against the whole human genome. Note that extrapolating the average runtime for processing eight gigabytes of data on a single machine reveals that aligning a whole terabyte of genomic read data against the whole human genome along with further downstream processing would easily take a month on a single machine.

We identified and evaluated several potential bottlenecks when scaling out a Hi-WAY installation beyond 128 nodes. For instance, Hadoop's master processes, YARN's ResourceManager and HDFS's NameNode, could prove to limit *scalability*. Similarly, the Hi-WAY AM process that handles the scheduling of tasks, the assembly of results, and the tracing of provenance, could collapse when further increasing the workload and the number of available compute nodes. To

Table 2: **Summary of the scalability experiment described in Section 4.1. The number of provisioned VMs is displayed alongside the volume of processed data, average runtime (over three runs), and the incurred cost.**

number of worker VMs number of master VMs	1 2	2 2	4 2	8 2	16 2	32 2	64 2	128 2
data volume	8.06 GB	16.97 GB	33.10 GB	69.47 GB	136.14 GB	270.98 GB	546.76 GB	1096.83 GB
avg. runtime in min.	340.12	350.36	351.62	344.82	375.57	372.09	380.24	353.39
runtime std. dev.	1.96	0.14	2.15	1.88	14.84	22.10	22.34	6.01
avg. cost[1] per run	$2.48	$3.41	$5.13	$8.39	$16.45	$30.78	$61.07	$111.79
avg. cost[1] per GB	$0.31	$0.20	$0.16	$0.12	$0.12	$0.11	$0.11	$0.10

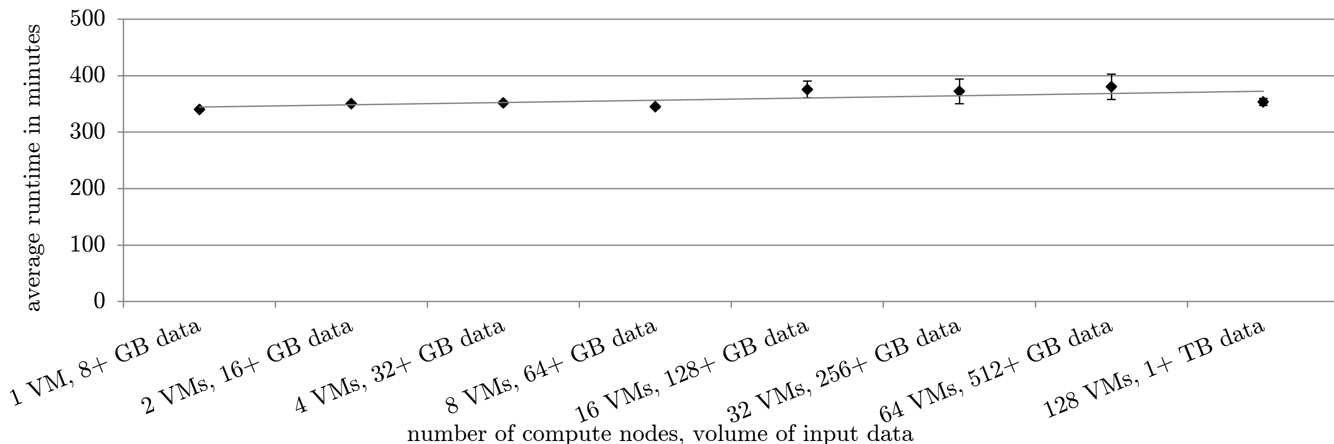

Figure 5: Mean runtimes for three runs of the variant calling workflow described in Section 4.1 when repeatedly doubling the number of compute nodes available to Hi-WAY along with the input data to be processed. The error bars represent the standard deviation, whereas the line represents the (linear) regression curve[2].

this end, we were interested in the resource utilization of these potential bottlenecks, which is displayed in Figure 6.

We observe a steady increase in load across all resources for the Hadoop and Hi-WAY master nodes when repeatedly doubling the workload and number of worker nodes. However, resource load stays well below maximum utilization at all cluster sizes. In fact, all resources are still utilized less than 5 % even when processing one terabyte of data across 128 worker nodes. Furthermore, we observe that resource utilization for Hi-WAY's master process is of the same order of magnitude as for Hadoop's master processes, which have been developed to scale to 10,000 compute nodes and beyond [42].

While resource utilization on the master nodes increases when growing the workload and computational infrastructure, we observe that CPU utilization stays close to the maximum of 2.0 on the worker nodes, whereas the other resources stay under-utilized. This finding is unsurprising, since both the alignment step and the variant calling step of the workflow support multithreading and are known to be CPU-bound. Hence, this finding confirms that the cluster is nearly fully utilized for processing the workflow, whereas the master processes appear to be able to cope with a considerable amount of additional load.

4.2 Performance / Bioinformatics

RNA sequencing (RNA-seq) methodology makes use of next-generation sequencing technology to enable researchers to determine and quantify the transcription of genes in a given tissue sample. Trapnell et al. [40] have developed a workflow that has has been established as the *de facto* standard for processing and comparing RNA-seq data.

In the first step of this workflow, genomic reads are aligned against a reference genome using the two alignment tools Bowtie 2 [24] and TopHat 2 [21]. The alignment serves the purpose of identifying the reads' genomic positions, which have been lost during the sequencing process. This first step is comparable to the first step in the variant calling workflow described in Section 4.1. However, in this workflow, reads are obtained by sequencing only the transcriptome, i.e., the set of transcribed genes, as opposed to sequencing the whole genome. In the second step of the workflow, the Cufflinks [40] package is utilized to assemble and quantify transcripts of genes from these aligned reads and, finally, to compare quantified transcripts for different input samples, for instance between diseased and healthy samples. See Figure 7 for a visualization of the RNA-seq workflow.

[1]Here, we assume a price of $0.146 per minute, as listed for m3.large instances in EC2's EU West region at the time of writing. We also assume billing per minute and disregard time required to set up the experiment.

[2]The standard deviation is higher for cluster sizes of 16, 32, and 64 nodes, which is due to the observed runtime of the CPU-bound variant calling step being notably higher in one run of the experiment. Since these three measurements were temporally co-located and we did not observe similar distortions at any other point in time, this observation can most likely be attributed to external factors.

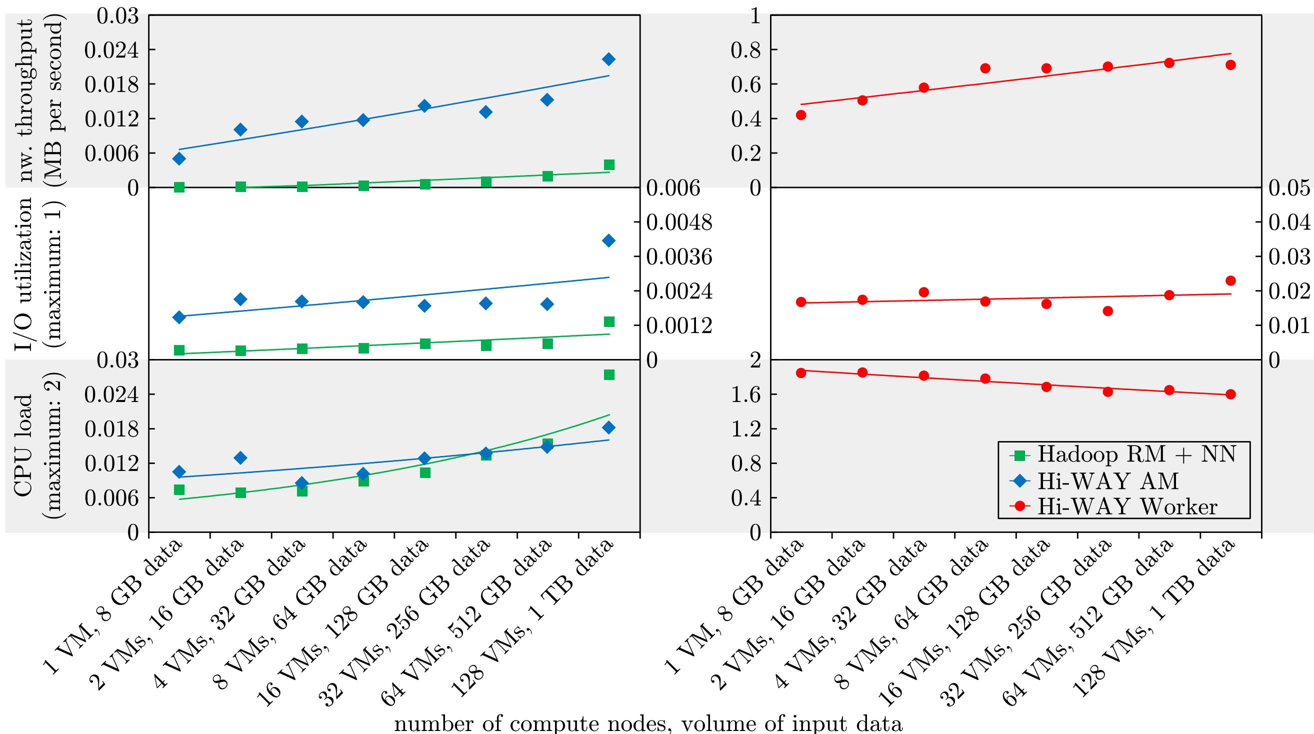

Figure 6: Resource utilization (CPU load, I/O utilization, and network throughput) of virtual machines hosting the Hadoop master processes, the Hi-WAY AM and a Hi-WAY worker process. Average values over the time of workflow execution and across experiment runs are shown along with their exponential regression curve. We observed the following peak values for worker nodes: 2.0 for CPU load (due to two virtual processing cores being available per machine), 1.0 for I/O utilization (since 1.0 corresponds to device saturation, i.e., 100 % of CPU time spent for I/O requests) and 109.35 MB per second for network throughput. Note the different scales for the master nodes on the left and the worker nodes on the right.

Wolfien et al. [46] implemented an extended version of this workflow in Galaxy, making it available through Galaxy's public workflow repository. Their implementation of the workflow, called TRAPLINE, compares two genomic samples. Since each of these two samples is expected to be available in triplicates and the majority of data processing tasks composing the workflow are arranged in sequential order, the workflow, without any manual alterations, has a degree of parallelism of six across most of its parts.

We executed the TRAPLINE workflow on virtual clusters of Amazon's EC2 consisting of compute nodes of type c3.2xlarge. Each of these nodes provides eight virtual processing cores, 15 gigabytes of main memory and 160 gigabytes of local SSD storage. Due to the workflow's degree of parallelism of six, we ran the workflow on clusters of sizes one up to six. For each cluster size, we executed this Galaxy workflow five times on Hi-WAY, comparing the average runtime against an execution on Galaxy CloudMan. Each run was launched in its own cluster, set up in Amazon's US East (North Virginia) region.

As workflow input data, we used RNA-seq data of young versus aged mice, obtained from the Gene Expression Omnibus (GEO) repository[3], amounting to more than ten gigabytes in total. We set up Hi-WAY using Karamel [1] and CloudMan using its Cloud Launch web application. Default parameters were left unchanged. However, since several tasks in TRAPLINE require large amounts of memory, we configured both Hi-WAY as well as CloudMan's default underlying distributed resource manager, Slurm, to only allow execution of a single task per worker node at any time. Omitting this configuration would lead either of the two systems to run out of memory at some point during workflow execution. The results of executing the TRAPLINE workflow on both Hi-WAY and Galaxy CloudMan are displayed in Figure 8. Across all of the tested cluster sizes, we observed that Hi-WAY *outperformed* Galaxy CloudMan by at least 25 %. These differences were found to be significant by means of a one-sample t-test (p-values of 0.000127 and lower).

The observed difference in *performance* is most notable in the computationally costly TopHat2 step, which makes heavy use of multithreading and generates large amounts of intermediate files. Therefore, this finding can be attributed to Hi-WAY utilizing the worker node's transient local SSD storage, since both HDFS as well as the storage of YARN containers reside on the local file system. Conversely, Galaxy CloudMan stores all of its data on an Amazon Elastic Block Store (EBS) volume, a persistent drive that is accessed over the network and shared among all compute nodes[4].

[3]series GSE62762, samples GSM15330[14|15|16|45|46|47]

[4]While EBS continues to be CloudMan's default storage option, a recent update has introduced support for using transient storage instead.

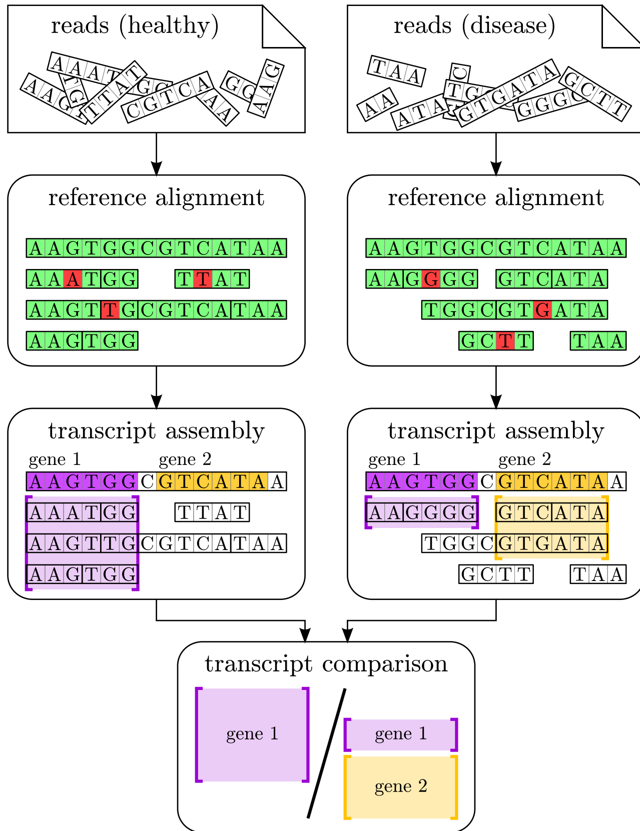

Figure 7: The RNA sequencing workflow described in Section 4.2. Genomic reads, the output of whole transcriptome sequencing, are aligned against a reference genome. Transcribed genes are then determined and quantified based on these alignments. Finally, transcription is compared between samples.

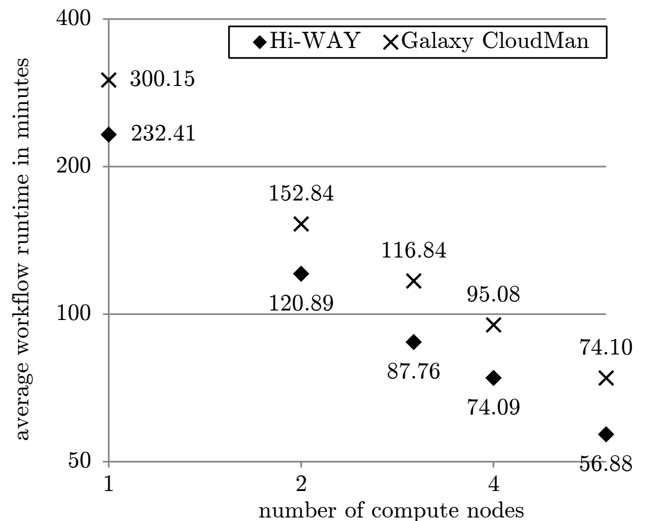

Figure 8: Average runtime of executing the RNA-seq workflow described in Section 4.2 on Hi-WAY and Galaxy CloudMan. The number of EC2 compute nodes of type c3.2xlarge was increased from one up to six. Note that both axes are in logarithmic scale.

Apart from the observed gap in *performance*, it is important to point out that Galaxy CloudMan only supports the automated setup of virtual clusters of up to 20 nodes. Compared to Hi-WAY, it therefore only provides very limited *scalability*. We conclude that Hi-WAY leverages the strengths of Galaxy, which lie in its intuitive means of workflow design and vast number of supported tools, by providing a more *performant*, flexible, and *scalable* alternative to Galaxy CloudMan for executing data-intensive Galaxy workflows with a high degree of parallelism.

4.3 Adaptive Scheduling / Astronomy

To underline the benefits of *adaptive* scheduling on heterogeneous computational infrastructures, an additional experiment was performed in which we generated a Pegasus DAX workflow using the Montage toolkit [7]. The resulting workflow assembles a 0.25 degree mosaic image of the Omega Nebula. It comprises a number of steps in which images obtained from telescopic readings are projected onto a common plane, analyzed for overlapping regions, cleaned from background radiation noise and finally merged into a mosaic.

The Montage toolkit can be used to generate workflows with very large number of tasks by increasing the degree value. However, the degree of 0.25 used in this experiment resulted in a comparably small workflow with a maximum degree of parallelism of eleven during the image projection and background radiation correction phases of the workflow. In the experiment, this workflow was repeatedly executed on a Hi-WAY installation set up on a virtual cluster in the EU West (Ireland) region of Amazon's EC2. The cluster comprised a single master node as well as eleven worker nodes to match the workflow's degree of parallelism. Similar to the *scalability* experiment in Section 4.1, all of the provisioned virtual machines were of type m3.large.

To simulate a heterogeneous and potentially shared computational infrastructure, synthetic load was introduced on these machines by means of the Linux tool *stress*. To this end, only one worker machine was left unperturbed, whereas five worker machines were taxed with increasingly many CPU-bound processes and five other machines were impaired by launching increasingly many (in both cases 1, 4, 16, 64, and 256) processes writing data to the local disk.

A single run of the experiment, of which 80 were conducted in total, encompassed (i) running the Montage workflow once using a FCFS scheduling policy, which served as a baseline to compare against, and (ii) running the workflow 20 times consecutively using the HEFT scheduler. In the process of these consecutive runs, larger and larger amounts of provenance data became available over time as a consequence of prior workflow executions. Hence, workflow executions using the HEFT scheduler were provided with increasingly comprehensive runtime estimates. Between iterations however, all provenance data was removed.

The results of this experiment are illustrated in Figure 9. Evidently, the performance of HEFT scheduling improves with more and more provenance data becoming available. Employing HEFT scheduling in the absence of any available provenance data results in subpar performance compared to FCFS scheduling. This is due to HEFT being a static scheduling policy, which entails that task assignments are fixed, even if one worker node still has many tasks to run

while another, possibly more performant worker node is idle.

However, with a single prior workflow run, HEFT already outperforms FCFS scheduling significantly (two-sample t-test, p-value of 0.033). The next significant performance gain can then be observed between ten and eleven prior workflow execution (two-sample t-test, p-value of $6.22 \cdot 10^{-7}$). At this point, any task composing the workflow, even the ones that are only executed once per workflow run, have been executed on all eleven worker nodes at least once. Hence, runtime estimates are complete and scheduling is no longer driven by the need to test additional task-machine-assignments. Note that this also leads to more stable workflow runtimes, which is reflected in a major reduction of the standard deviation of runtime. We argue that the observed performance gains of HEFT over baseline FCFS scheduling emphasize the importance of and potential for *adaptive* scheduling in distributed scientific workflow execution.

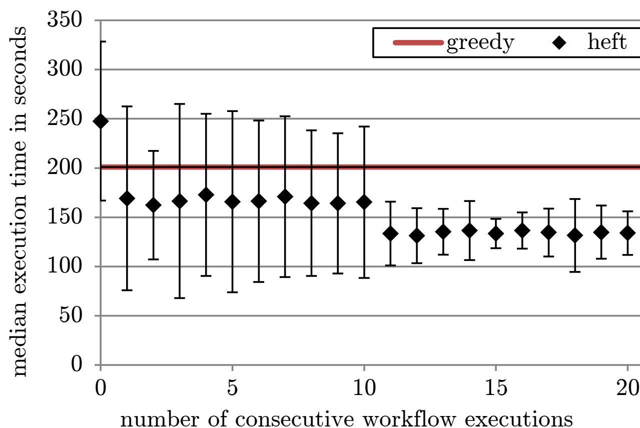

Figure 9: Median runtime of executing Montage on a heterogeneous infrastructure when using HEFT scheduling and increasing the number of previous workflow runs and thus the amount of available provenance data. The error bars represent the standard deviation.

5. CONCLUSION AND FUTURE WORK

In this paper, we presented Hi-WAY, an application master for executing arbitrary scientific workflows on top of Hadoop YARN. Hi-WAY's core features are a *multilingual* workflow language interface, support for *iterative* workflow structures, *adaptive* scheduling policies optimizing *performance*, tools to provide *reproducibility* of experiments, and, by employing Hadoop for resource management and storage, *scalability*. We described Hi-WAY's interface with YARN as well as its architecture, which is built around the aforementioned concepts. We then outlined four experiments, in which real-life workflows from different domains were executed on different computational infrastructures comprising up to 128 worker machines.

As future work, we intend to further harness the statistics on resource utilization provided by Hi-WAY's Provenance Manager. Currently, the containers requested by Hi-WAY and provided by YARN all share an identical configuration, i.e., they all have the same amounts of virtual processing cores and memory. This can lead to under-utilization of resources, since some tasks might not be able to put all of the provided resources to use. To this end, we intend to extend Hi-WAY with a mode of operation, in which containers are custom-tailored to the tasks that are to be executed.

Funding

Marc Bux and Jörgen Brandt were funded by the EU project BiobankCloud. We also acknowledge funding by the DFG graduate school SOAMED and support by an AWS in Education Grant award.

References

[1] Karamel. http://www.karamel.io.

[2] Opscode Chef. https://www.chef.io.

[3] E. Afgan et al. Galaxy CloudMan: Delivering Cloud Compute Clusters. *BMC Bioinformatics*, 11(Suppl 12):S4, 2010.

[4] M. Albrecht, P. Donnelly, P. Bui, and D. Thain. Makeflow: A Portable Abstraction for Data Intensive Computing on Clusters, Clouds, and Grids. *SWEET Workshop*, 2012.

[5] A. Alexandrov et al. The Stratosphere Platform for Big Data Analytics. *VLDBJ*, 23(6):939–964, 2014.

[6] P. Amstutz et al. Common Workflow Language, v1.0. *Figshare*, 2016.

[7] G. B. Berriman et al. Montage: A Grid-Enabled Engine for Delivering Custom Science-Grade Mosaics on Demand. *SPIE Astronomical Telescopes + Instrumentation*, 5493:221–232, 2004.

[8] J. Brandt, M. Bux, and U. Leser. Cuneiform: A Functional Language for Large Scale Scientific Data Analysis. *Workshops of the EDBT/ICDT*, 1330:17–26, 2015.

[9] M. Bux et al. Saasfee: Scalable Scientific Workflow Execution Engine. *PVLDB*, 8(12):1892–1895, 2015.

[10] M. Bux and U. Leser. Parallelization in Scientific Workflow Management Systems. *arXiv preprint, arXiv:1303.7195*, 2013.

[11] S. Cohen-Boulakia and U. Leser. Search, Adapt, and Reuse: The Future of Scientific Workflows. *SIGMOD Record*, 40(2):6–16, 2011.

[12] S. B. Davidson and J. Freire. Provenance and Scientific Workflows: Challenges and Opportunities. *SIGMOD Conference*, 2008.

[13] E. Deelman et al. Pegasus, a Workflow Management System for Large-Scale Science. *Future Generation Computer Systems*, 46:17–35, 2015.

[14] P. Di Tommaso et al. The Impact of Docker Containers on the Performance of Genomic Pipelines. *PeerJ*, 3:e1273, 2015.

[15] Y. Gil et al. Examining the Challenges of Scientific Workflows. *Computer*, 40(12):26–34, 2007.

[16] C. Goble and D. de Roure. myExperiment: Social Networking for Workflow-Using e-Scientists. *WORKS Workshop*, 2007.

[17] J. Goecks, A. Nekrutenko, and J. Taylor. Galaxy: a Comprehensive Approach for Supporting Accessible, Reproducible, and Transparent Computational Research in the Life Sciences. *Genome Biology*, 11(8):R86, 2010.

[18] T. Hey, S. Tansley, and K. Tolle. *The Fourth Paradigm*. Microsoft Research, 2nd edition, 2009.

[19] B. Hindman et al. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. *NSDI Conference*, 2011.

[20] M. Islam et al. Oozie: Towards a Scalable Workflow Management System for Hadoop. *SWEET Workshop*, 2012.

[21] D. Kim et al. TopHat2: Accurate Alignment of Transcriptomes in the Presence of Insertions, Deletions and Gene Fusions. *Genome Biology*, 14:R36, 2013.

[22] D. C. Koboldt et al. VarScan: Variant Detection in Massively Parallel Sequencing of Individual and Pooled Samples. *Bioinformatics*, 25(17):2283–2285, 2009.

[23] J. Köster and S. Rahmann. Snakemake – a Scalable Bioinformatics Workflow Engine. *Bioinformatics*, 28(19):2520–2522, 2012.

[24] B. Langmead and S. Salzberg. Fast Gapped-Read Alignment with Bowtie 2. *Nature Methods*, 9:357–359, 2012.

[25] H. Li et al. The Sequence Alignment/Map Format and SAMtools. *Bioinformatics*, 25(16):2078–2079, 2009.

[26] J. Liu, E. Pacitti, P. Valduriez, and M. Mattoso. A Survey of Data-Intensive Scientific Workflow Management. *Journal of Grid Computing*, 13(4):457–493, 2015.

[27] I. Momcheva and E. Tollerud. Software Use in Astronomy: An Informal Survey. *arXiv preprint, arXiv:1507.03989*, 2015.

[28] D. G. Murray et al. CIEL: A Universal Execution Engine for Distributed Data-Flow Computing. *NSDI Conference*, 2011.

[29] E. Ogasawara et al. An Algebraic Approach for Data-Centric Scientific Workflows. In *PVLDB*, volume 4, pages 1328–1339, 2011.

[30] E. Ogasawara et al. Chiron: A Parallel Engine for Algebraic Scientific Workflows. *Concurrency and Computation: Practice and Experience*, 25(16):2327–2341, 2013.

[31] S. Pabinger et al. A Survey of Tools for Variant Analysis of Next-Generation Genome Sequencing Data. *Briefings in Bioinformatics*, 15(2):256–278, 2014.

[32] B. Saha et al. Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications. *SIGMOD Conference*, 2015.

[33] I. Santana-Perez et al. Leveraging Semantics to Improve Reproducibility in Scientific Workflows. *Reproducibility at XSEDE Workshop*, 2014.

[34] Z. D. Stephens et al. Big Data: Astronomical or Genomical? *PLoS Biology*, 13(7):e1002195, 2015.

[35] R. Sumbaly, J. Kreps, and S. Shah. The Big Data Ecosystem at LinkedIn. *SIGMOD Conference*, 2013.

[36] I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields. *Workflows for e-Science: Scientific Workflows for Grids*. Springer, 2007.

[37] The 1000 Genomes Project Consortium. A Global Reference for Human Genetic Variation. *Nature*, 526(7571):68–74, 2015.

[38] P. D. Tommaso, M. Chatzou, P. P. Baraja, and C. Notredame. A Novel Tool for Highly Scalable Computational Pipelines. *Figshare*, 2014.

[39] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. *Transactions on Parallel and Distributed Systems*, 13(3):260–274, 2002.

[40] C. Trapnell et al. Differential Gene and Transcript Expression Analysis of RNA-seq Experiments with TopHat and Cufflinks. *Nature Protocols*, 7(3):562–578, 2012.

[41] E. L. Van Dijk, H. Auger, Y. Jaszczyszyn, and C. Thermes. Ten Years of Next-Generation Sequencing Technology. *Trends in Genetics*, 30(9):418–426, 2014.

[42] V. K. Vavilapalli et al. Apache Hadoop YARN: Yet Another Resource Negotiator. *ACM SOCC*, 2013.

[43] J. Vivian et al. Rapid and Efficient Analysis of 20,000 RNA-seq Samples with Toil. *bioRxiv*, 062497, 2016.

[44] K. Wang, M. Li, and H. Hakonarson. ANNOVAR: Functional Annotation of Genetic Variants from High-Throughput Sequencing Data. *Nucleic Acids Research*, 38(16):e164, 2010.

[45] M. Wilde et al. Swift: A Language for Distributed Parallel Scripting. *Parallel Computing*, 37(9):633–652, 2011.

[46] M. Wolfien et al. TRAPLINE: A Standardized and Automated Pipeline for RNA Sequencing Data Analysis, Evaluation and Annotation. *BMC Bioinformatics*, 17:21, 2016.

[47] K. Wolstencroft et al. The Taverna Workflow Suite: Designing and Executing Workflows of Web Services on the Desktop, Web or in the Cloud. *Nucleic Acids Research*, 41:557–561, 2013.

[48] D. Wu et al. Building Pipelines for Heterogeneous Execution Environments for Big Data Processing. *Software*, 33(2):60–67, 2016.

[49] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. *HotCloud Conference*, 2010.

[50] Y. Zhao et al. Enabling scalable scientific workflow management in the Cloud. *Future Generation Computer Systems*, 46:3–16, 2015.