Georg-August-Universität
Göttingen
Zentrum für Informatik

ISSN 1612-6793
Nummer ZAI-MSC-2015-02

Masterarbeit

im Studiengang "Angewandte Informatik"

Automated Deployment and Distributed Execution of Scientific Software in the Cloud using DevOps and Hadoop MapReduce

Michael Göttsche

am Institut für Informatik

Gruppe Softwaretechnik für Verteilte Systeme

Bachelor- und Masterarbeiten
des Zentrums für Informatik
an der Georg-August-Universität Göttingen

21. April 2015

Georg-August-Universität Göttingen
Zentrum für Informatik

Goldschmidtstraße 7
37077 Göttingen
Germany

Tel.	+49 (5 51) 39-17 20 00
Fax	+49 (5 51) 39-1 44 03
Email	office@informatik.uni-goettingen.de
WWW	www.informatik.uni-goettingen.de

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Göttingen, den 21. April 2015

Master's thesis

Automated Deployment and Distributed Execution of Scientific Software in the Cloud using DevOps and Hadoop MapReduce

Michael Göttsche

April 21, 2015

Supervised by
Prof. Dr. Jens Grabowski
Institute of Computer Science
Georg-August-University of Göttingen, Germany

Dr. Steffen Herbold
Institute of Computer Science
Georg-August-University of Göttingen, Germany

Abstract

Researchers of various disciplines that employ quantitative modelling techniques are confronted with an ever-growing need for computation resources to achieve acceptable execution times for their scientific software. This demand is often fulfilled by the use of distributed processing on mid- to large size research clusters or costly specialized hardware in the form of super computers. However, to many users these resources are either not available due to financial constraints or their usage requires in-depth experience with parallel programming models. The acquisition of respective knowledge and the provisioning of an autonomous computation infrastructure are both often too time-intensive tasks to be an option.

This thesis presents an approach that requires the researcher neither to have the financial resources to purchase hardware or concern herself with administrative tasks, nor to implement a complex parallel programming model into the software. Instead, we propose a Cloud-based system that transparently provisions computing resources from Infrastructure-as-a-Service providers with the goal of a reduced overall runtime. Besides others, utilizing such providers offers the benefit of a pay-per-use model. Our solution consists of two components based on DevOps tools: (1) A tool for the provisioning and deployment of a Hadoop cluster on the Cloud computing resources based, (2) A tool for the automated deployment and distributed execution of scientific software with minimal effort on the user's side.

We perform two case studies to compare the performance of our solution with the non-distributed execution of the software as well as with a native MapReduce implementation. The results show that our approach outperforms both alternatives with negligible setup effort by the user and thus is a viable choice for the scenario outlined above. While we focus on scientific software in this thesis, the suitability of our approach is by no means inherently limited to this category, but is much rather applicable to a wide variety of domains.

Contents

1. Introduction

Over the last decade a growing number of academic disciplines has either begun to employ quantitative models or intensified their use. For example, while for some research directions like meteorology or engineering modelling has long been an established tool of scientific work, other disciplines such as the humanities and even some natural sciences only later began adapting it at a broader scale. Not only has the application of mathematical models in science spread, the models have also become more and more complex.

Solving such complex models requires appropriate computing power and, as is well known, such has increased vastly in terms such as floating point operations per second and speed and capacity of storage. A well-known ranking of the world's fastest supercomputers even sometimes referred to in general audience news, the 'Top 500' list, shows that since its inception in 1994 the aggregated performance has grown exponentially from one Teraflop/second to more than half an Exaflop/second [43].

The architectures of these supercomputers, or more broadly large-sized hardware including compute clusters in general, however often requires vast knowledge about parallel computing on the user's side or is even designed for a very specific set of applications and thus not available to the average modeller that has extensive domain and average programming knowledge, but no sufficient knowledge about parallel computation models or system administration.

Having at one extreme such large and hardly accessible compute clusters and at the other extreme the researcher's lone desktop computer leaves a gap for the case of shortening computation times of small to average applications: **(1)** Scientists often lack adequate hardware for their problems due to financial constraints, **(2)** Writing parallel software or parallelizing existing software is a non-trivial task as is configuring and administrating the underlying software infrastructure.

The advent of public Cloud Computing with steadily falling prices has rectified the first problem, because it is now possible for everyone to rent an arbitrary amount of compute power for an arbitrarily long or short timeframe. This "per-pay-use" model allows cost savings compared to acquisition of physical resources for such use cases where the compute resources would be underused for a substantial part of the year [51]. Still, having access to a set of virtual servers provided by his cloud provider does not relief the researcher from the burden of configuring them and developing his/her software in a way that it utilizes the compute resources in a parallel fashion with an automated deployment process.

Addressing that second problem is the topic of this thesis. More specifically, it explores the issue of combining established technology into accessible tools that hide from the researcher the underlying complexity. The topic can broadly be separated into three questions:

- What software stack is well suited for building an easy-to-use Cloud infrastructure for automated deployment and parallel execution of existing software?
- How can such a tool be designed and implemented?
- How does it perform, i.e. does it fulfill the goal of shortening computation times for modelling problems and how well does it do so?

At the core of our proposed solution will be the Hadoop [17] framework developed by the Apache Software Foundation which – next to supporting tools – implements a distributed file system (called HDFS) that we will employ for sharing data between servers and the MapReduce paradigm which is a system for parallel processing of data sets. The Cloud infrastructure employed for our practical evaluation was provided by the Gesellschaft für wissenschaftliche Datenverarbeitung Göttingen (GWDG).

1.1. Goals and Contributions

The goals and contributions of the thesis can be summarized as follows:

- Explore an approach for transparent automated deployment and parallel execution of scientific software in the Cloud.
- Development of accessible tools that can be used by researchers for
  - provisioning and deploying clusters in the Cloud with a software stack appropriate for parallel execution of scientific software.
  - parallelizing execution of software with minimal effort on the researcher's side.
- Case studies evaluating the performance of our approach with the status quo, i.e. non-distributed execution.
- A comparison of our approach with parallelization using the MapReduce. paradigm.

1.2. Outline

The outline for this thesis is as follows. In the chapter following this introduction we will document the foundations for this work, more specifically Cloud Computing as a concept, the software-based management of Cloud infrastructures and configuration management for software deployment onto such infrastructures (Chapter 2). Moreover, it will introduce Hadoop as the core piece upon which our solution is built and its different modes of operation relevant for us, namely Hadoop Streaming and MapReduce. Chapter 3 covers the conceptual design of our approach from a high level perspective, i.e. the interactions of the system's different components and the user's interaction with the system. The actual implementation of our approach is afterwards detailed in Chapter 4. Chapter 5 presents two case studies to evaluate the performance of our approach as well as the effort needed to deploy and distributedly execute applications with it. To further compare it with a native MapReduce application, Chapter 6 will describe both the general steps for migrating a non-parallel application to the MapReduce paradigm as well as carry out these steps for one of our case study applications. Also, we will compare both approaches' performance with each other. Chapter 7 provides an overview of related work. Finally, Chapter 7 concludes the thesis and points out possible directions for future research.

2. Background

In this chapter we will describe the concepts and frameworks that form the foundation of the thesis. We will briefly recall the taxonomy of parallel computing in Section 2.1. Section 2.2 introduces the cloud computing model that has emerged to play an essential role in many organization's IT infrastructures and which we will employ for our infrastructure. Section 2.3.1 discusses the software-supported management of such virtualized infrastructures. Configuration management as the complementing part on the operating system level is discussed in Section 2.3.2. An overview of the Hadoop framework concludes this chapter (Section 2.4).

2.1. Parallel Computing

Different definitions and taxonomies for parallel computing exist, but at their core lies the idea that problems are solved concurrently by dividing them up into smaller problems [50]. One of the eldest taxonomies, published already in 1966 by Michael J. Flynn, that is still often cited, is known as Flynn's taxonomy. Though it classifies computer architectures, the classifications are also applicable to software-level parallelism. In total, there are four classifications which are depicted in Table 2.1.

	Single Instruction	**Multiple Instruction**
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

Table 2.1.: Flynn's Taxonomy

As shown, Flynn distinguishes alongside two dimensions, instructions and data:

- *Single Instruction, Single Data*: A single processor unit executes one single instruction at a time. This equals the von Neumann-model.
- *Single Instruction, Multiple Data*: Again there is only a single instruction stream, but multiple data streams. Examples include e.g. vector processors that operate on arrays.

- *Multiple Instruction, Single Data*: Multiple processors execute different operations on the same single data. This architecture is not practically relevant except for highly specialized use cases.
- *Multiple Instruction, Multiple Data*: Multiple processors execute different instructions on different data. This is the de-facto standard for parallel computing.

Another more recent way to distinguish parallelization on the software level is by the terms of *task parallelism* and *data parallelism*. While the former describes a form of parallelization that emphasizes the distribution of tasks, i.e. that different processors execute different threads which may execute the same or different code, *data parallelism* emphasizes the distribution of data. In an *SIMD* system this means that different processors on different pieces of data perform the same task [52]. The approach developed in the context of this thesis is built on the *data parallelism* paradigm.

2.2. Cloud Computing

2.2.1. Definition and Characteristics

Although there is a multitude of definitions for the term *Cloud Computing*, the definition most often cited is the definition by the National Institute of Standards and Technology (USA):

> "Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This Cloud model is composed of five essential characteristics, three service models, and four deployment models." [60]

The object of the definition, namely "a shared pool of configurable computing resources" which is broadly defined by not only including physical resources such as servers, but also non-physical resources like applications, is further defined by five characteristics. *On-demand self service* describes the fact that customers can unilaterally provision resources without human interaction with the provider, e.g. via Application Programming Interfaces (APIs). The capabilities need to be available "over the network and accessed through standard mechanisms" [60] for use by heterogeneous clients including e.g. smartphones to fulfill the *broad network access* requirement. The term *resource pooling* means that the service provider dynamically assigns and reassigns physical and virtual resources according to the customer's demand while the customer usually "has no control or knowledge over the exact location of the provided resources" [60]. According to the *rapid elasticity* requirement resources can be dynamically provisioned and released while the available capabilities should appear to be unlimited to the customer. Finally, *measured service* describes that the service usage can be "monitored, controlled and reported" [60] to establish transparency for both sides.

2.2.2. Service Models

There are three service models in Cloud Computing which can be differentiated based on the layer they operate on and their targeted audience [60]:

- *Infrastructure as a Service (IaaS)*: On the IaaS level customers provision "processing, storage, networks, and other fundamental computing resources" [60], e.g. virtual machines, and are able to run arbitrary software on it, including the choice of operating systems. The service provider manages the underlying infrastructure.
- *Platform as a Service (PaaS)*: PaaS providers operate on top of the IaaS layer and provide the customer with the possibility to deploy self-created or acquired applications on the Cloud infrastructure. These applications can be created using "programming languages, libraries, services, and tools supported by the provider" [60].
- *Software as a Service (SaaS)*: SaaS providers offer ready-to-use applications to customers which are available for different client devices via e.g. a web browser or application interface. In this model the customer does not control any part of the infrastructure, but can only change user-specific application settings.

2.2.3. Deployment Models

Based on who is eligible to access the Cloud's services they can further be differentiated by their deployment models, which are the following [60]:

- *Private Cloud*: The Cloud infrastructure is exclusively used by a single organization, e.g. a company or a university. It can, but is not necessarily owned and/or managed by this organization.
- *Public Cloud*: In contrast to a private Cloud a public Cloud is open for use by the general public and is usually owned and managed by a company, but could also be a government's or some different organization's service.
- *Hybrid Cloud*: Hybrid Clouds are composed of two or more distinct infrastructures which are "bound together by standardized or proprietary technology that enables data and application portability" [60]. An example use case would be Cloud bursting to account for load balancing.

- *Community Cloud*: Finally, community Clouds are provisioned for use by a community of consumers that belong to different organizations which have shared concerns and for this purpose establish a shared Cloud.

2.2.4. Virtualization

The use of the term ”virtualized resources” in the previous sections conveys that Cloud Computing users do not acquire direct access to physical resources from their provider. Instead, the service provider abstracts away from the user the underlying infrastructure by creating a virtual infrastructure on top of it. The user neither has nor needs knowledge about the physical hardware because performance characteristics of the services are expressed by means of alternative metrics, e.g. *Input/Output Operations Per Second (IOPS)* for virtual storage. This is an on-going development, which besides established technologies, as for example virtual storage, extends to other parts of the infrastructure for which *Network Functions Virtualization (NFV)* is a recent example [14].
Most important in the context of this thesis is however the virtualization of servers through the use of *Virtual Machines (VMs)* of which usually many share one physical machine called the host. This approach has a number of advantages for both the service provider and consumer including [59]:

- *Server Consolidation*: Workloads between multiple unter-utilized machines can be consolidated.
- *Multiple execution environments*: By creating multiple execution environments the Quality of Service can be increased through load balancing and removing *Single Point of Failures.*
- *Debugging and Sandboxing*: Users can define different execution environments making it possible to test software on different platforms and in a secure way because machines are isolated from one another.
- *Software Migration*: Applications can be migrated more easily, e.g. for scaling to higher demand.

The ability of Cloud providers to maximize hardware utilization is the bottom line of the economics of Cloud Computing: On average, independently run servers are not operating anywhere near their capacity limit, but Cloud Computing providers attempt to allocate VMs to them with optimal resource utilization in mind. This fact, together with other factors such as economies of scale, allows Cloud services to offer rates cheap enough to provide users cost savings compared with investments in physical hardware while still operating profitable. [51]

2.3. Deployment Management with DevOps

Setting up clusters with more than just a handful of nodes by hand, be it in a traditional environment or a Cloud Computing platform, is an unmanageable task which can roughly be separated into the management of the infrastructure and the management of the software configuration, but they are also interconnected.
In recent years, roughly at the same time of the emergence of the term *DevOps*, tools for these tasks have been developed and gained popularity. There is not yet an accepted academic definition of the term, but its name – being a portmanteau of the words "development" and "operations" – stresses the interdependence of these two fields [48]. Specifically, the increased use of automation tools is one of its characteristics. Therefore, since our approach is also developed with heavy use of such tools, we subsume them under the term DevOps.

2.3.1. Infrastructure Management

Since this thesis addresses only Cloud Computing-based architectures, we refer here to "infrastructure management" as such software which helps automating the *provisioning* of virtual machines based on user-defined rules. Provisioning here is defined as the instantiation of the virtual machine with the base operating system image, but without further software setup and configured, which is handled by the configuration management software illustrated in the next section.

The need for infrastructure management software arises mainly from the complexity of multi-machine architectures and the related need for extensively testing, i.e. also repeatedly rebuilding them. Without tool support, the administrator and/or developer has to either manually interact with the Cloud platform at hand every time or write her own set of scripts that repeat these actions and perform error checking.

Infrastructure management tools replace this approach by providing the user with the possibility of describing the desired infrastructure in a file-based fashion and handle its provisioning, suspension, unprovisioning and possibly other actions.

2.3.1.1. Vagrant

For the remainder of this section we will use `Vagrant` [45] for illustrating this concept which we also employ as part of our approach.

Listing 2.1: Vagrant Example

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|

  config.vm.define "web" do |web|
    web.vm.box = "apache"
  end

  config.vm.define "db" do |db|
    db.vm.box = "mysql"
  end

  config.vm.synced_folder "data/", "/vagrant_data"
end
```

With Vagrant, the infrastructure is defined in a so-called *Vagrantfile* (Listing 2.1) that is written in Ruby [40] syntax and supports embedding arbitrary Ruby code to enhance the infrastructure description with dynamic parts. In the example above, two virtual machines are defined, one under the name "web" and the other under the name "db". The *xx.vm.box* lines refer to the image that should be installed on the machines, in this example they refer to user-defined images for web and database servers, respectively. Vagrant's vendor HashiCorp [28] also provides a Cloud service where vendors and community users can upload general or special purpose operating system images which users can then directly make use of in their *Vagrantfile*s. Also, new image files can easily be created from running machines using the *package* command. The last line instructs Vagrant to set up a directory synchronization between the host machine's *data* directory and the virtual machines linked as *vagrant_data* on them.

There is no mentioning of where the virtual machines should be created in the example just given. By default, Vagrant works with the virtualization software `VirtualBox`, however one of its strengths is that it provides a plugin system to add additional so-called *providers*. For instance, there are providers for Amazon Web Services (AWS), `VMware` or the `OpenStack` IaaS platform which we are going to use later on. Since different providers may require different or additional user input, the example above cannot be applied to all of them without modification. In the case of OpenStack, the plugin requires at least login credentials and information about the Cloud provider besides the infrastructure rules already given. Depending on the provider in use, Vagrant also supports up to some point the specification of network settings for the described infrastructures, e.g. which IP to use for the machine or whether to assign just a local IP address or also a public ("floating") one.

Once the user has defined the infrastructure, the next step in the workflow is to provision it using the *vagrant up* command. Vagrant will communicate with the specified provider, e.g. using calls to the VirtualBox utilities or using API calls to a Cloud provider, to provision the requested virtual machines. After successful initialization, *vagrant ssh <machine_name>* establishes a SSH connection to a particular machine. As Vagrant covers the whole lifecycle of an infrastructure, it also provides commands for e.g. incorporating changes of the *Vagrant-*

file (*vagrant reload*), checking the infrastructure status (*vagrant status*) as well as shutting down (*vagrant halt*) or permanently removing the virtual machines (*vagrant destroy*).

2.3.2. Software Configuration Management

The infrastructure management described in the previous section covers only the provisioning of the infrastructure. The software-level configuration of the machines including installation, configuration and advanced system settings needs to be handled by other measures, for example using *Software Configuration Management (SCM)* software packages. A handful of popular SCM tools have been developed and gained popularity in recent years such as Chef `[7]`, `Puppet` [39] or – the tool our approach is relying on – `Ansible` [2].
All these software packages are built upon the maxim that a system's setup should be declared in the form of rules in a *Domain-Specific Language (DSL)* rather than in conglomerated bash scripts and/or by manual interaction. This approach is often referred to as "Infrastructure as Code" (where infrastructure refers to both, computing resources and software installed on it) in the DevOps community [32] and associated with a number of advantages compared to the manual approach:

- *Structured Approach*: The system's configuration can be decomposed into manageable pieces and hierarchically structured. This, as is usually the case with complex entities of any kind, allows for better manageability.
- *Reproducibility*: Infrastructures can be recreated in the same or replicated to a new environment in the exact same way. This is beneficiary under different circumstances including
  - Failure: In case of deterioration of the system it can systematically be rebuilt by rerunning the SCM tool.
  - Debugging Software: Often errors in software show up only in particular setups. When these can be replicated exactly, it helps developers debug such errors which otherwise could not be retraced by them.
- *Versioning*: Changes to the infrastructure can be managed using version control systems, thus making them transparent.

2.3.2.1. Ansible

Listing 2.2 shows a very basic Ansible example Playbook which is its language for defining configuration, deployment and orchestration of the infrastructure.

Listing 2.2: Ansible Example

```
---
- hosts: webservers
  vars:
    http_port: 80
    max_clients: 200
  remote_user: root
  tasks:
  - name: ensure apache is at the latest version
    yum: pkg=httpd state=latest
  - name: write the apache config file
    template: src=/srv/httpd.j2 dest=/etc/httpd.conf
    notify:
    - restart apache
  - name: ensure apache is running
    service: name=httpd state=started
  handlers:
    - name: restart apache
      service: name=httpd state=restarted
```

Ansible uses the simple `YAML` [49] format which ensures human-readable configuration files and a structure that is, to a large extent, self-explanatory. The given example defines that for the hosts in the *webservers* group the tasks listed in section *tasks* should be executed as user *root* and defines two variables. These can be used in the *tasks* section which defines an update procedure for the Apache web server in three steps. Each task consists of a descriptive name and an action. For the latter, Ansible provides a pool of modules for different purposes, e.g. *yum* for the corresponding packet management software, *template* for setting system files based on templates (which can include the variables defined earlier). Finally, the example defines a handler for restarting Apache using the *service* module. This handler is called by the second task to incorporate the changes to the configuration file.

2.4. Hadoop

Hadoop [17] is a framework for distributed processing of large data sets developed as a project of the Apache Software Foundation (ASF). As previously mentioned, large-scale computation is often performed on either expensive special-purpose hardware designed for *High Performance Computing (HPC)* or on high availability clusters appearing as a single machine to the user. Rather than relying on hardware to ensure high availability, Hadoop has been designed to operate fail-safe on commodity hardware for which occasional failure is expected. By splitting up large problems into smaller chunks and distributing them to the cluster it can utilize the capabilites of all nodes and account for hardware failures by simply reassigning jobs to other nodes.

Hadoop's development started as an open-source implementation of a new programming model called *MapReduce* soon after its inventor, GOOGLE, published its first paper[54] about it. In the remainder of this section we will describe this programming model as well as the other core parts of the Hadoop framework.

2.4.1. MapReduce

MapReduce emerged at Google out of the need for a programming model for processing large datasets in a variety of tasks that enables the developer to concentrate on the – sometimes very short – application-specific code without having to write boilerplate code for the common problems such as data partitioning, failure handling, load balancing et al. The description in this section is based on their often cited paper *MapReduce: Simplified Data Processing on Large Clusters* [54].

At the most basic level, a MapReduce program consists of a *map* and a *reduce* function, an abstraction inspired by primitives in `Lisp` and other functional programming languages. The *map* function takes as input key/value pairs and produces new intermediate key/value pairs as output. The library sorts these by key and passes those with same key to the *reduce* function which usually merges the values for that key together by some action to – most often – just zero or one values. An often cited example for the application of this paradigm is the problem of counting words in text documents (see [23] for a Hadoop-based implementation). Using MapReduce this can be expressed in pseudo-code as follows:

Listing 2.3: MapReduce WordCount in Pseudo-code [54]

```
map(String key, String value):
        // key: document name
        // value: document contents
        for each word w in value:
                EmitIntermediate(w, "1");

reduce(String key, Iterator values):
        // key: a word
        // values: a list of counts
        int result = 0;
        for each v in values:
                result += ParseInt(v);
                Emit(AsString(result));
```

In this example, the *map* function emits ”1” for every word found and the *reduce* function iterates over the values for each key – here just ”1” a single or multiple times – and sums them up to calculate the number of occurrences for each word. More formally we can express the model as:

$$\begin{array}{lll} \text{map:} \quad (\text{k1, v1}) & \rightarrow & \text{list(k2, v2)} \\ \text{reduce:} \ (\text{k2, list(v2)}) & \rightarrow & \text{list(v2)} \end{array}$$

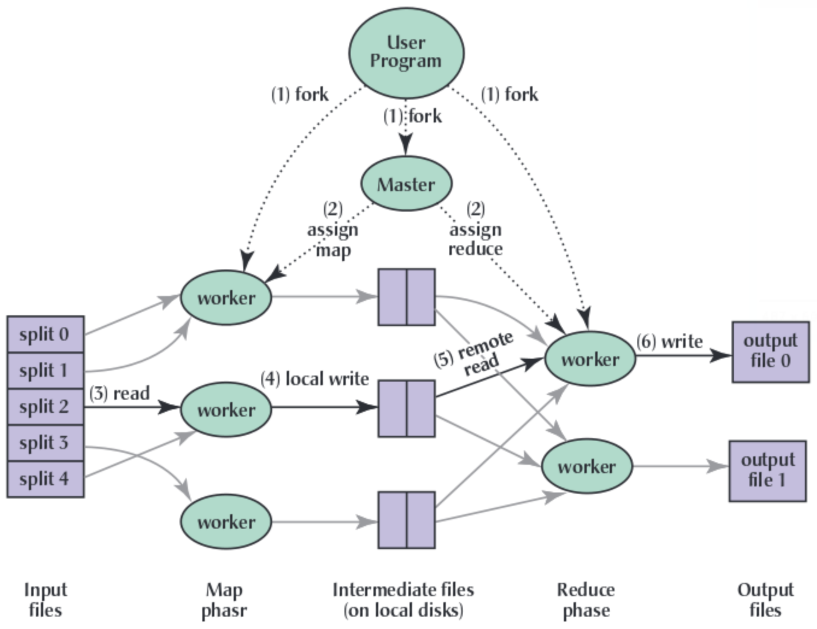


Figure 2.1.: MapReduce Execution Phases [54]

The execution steps depend on the actual implementation of the MapReduce paradigm and, as the authors point out, optimality depends on the underlying hardware. Google’s implementation was designed similar to Hadoop’s with average commodity PCs in mind with the following seven processing phases:

1. The MapReduce framework splits the input file(s) into M smaller pieces of a pre-configured size. Each map job will receive one of these pieces as input.

2. One node is assigned the *master* status and assigns idle nodes map tasks.

3. The worker nodes read their assigned input split, parse the key/value pairs and pass them to the user-defined *map* function. The key/value pairs generates as output of this function are held in memory.

4. Periodically these pairs are partitioned into regions and written to disk. The master node receives the locations of the buffered pairs and passes them to the reduce workers.

5. The reduce workers read the data assigned to them and sorts it by key to account for the fact that usually different keys are usually passed to one reducer.

6. For each key of the input data the reducer node executes the user-defined *reduce* function and appends its output to the final output file.

7. Upon completion of the map and reduce tasks the user program is notified and can now either process the output files of which there are as many as reducers.

The process just described is depicted in figure 2.1.

2.4.2. HDFS

Because all worker nodes of a Hadoop cluster need access to shared data sets, it requires a shared file system. This is due to the fact that, in addition to other reasons, a centralized storage system would induce a *Single Point of Failure* and be inconsistent with the approach of employing commodity hardware. Hadoop therefore instead relies on its own distributed file system called *Hadoop Distributed File System (HDFS)*. HDFS has been designed with the assumption that hardware failures are the norm so that fail-safeness has to be implemented on the software level rather than investing in hardware measures such as *RAID*. Conceptually, part of this is the quick detection of faults and automatic recovery, which in practice is achieved primarily by redundant data replication between nodes which we will elaborate on after introducing HDFS's architecture (figure 2.2).

HDFS uses a master/slave approach where one node is designated as the *Namenode*. The *Namenode* is responsible for the file system namespace (depicted as "Metadata" in the figure) and access control and is the entry point for input/output requests from users by mapping them to the corresponding *Datanodes*. *Datanodes* act as slaves in HDFS which store the data and serve read and write requests from clients. However, in contrast to other filesystems, in HDFS *Datanodes* do not store complete files, but blocks that result from splitting the files based on a user-defined block size which, at the time of this writing, defaults to 64MB.

These blocks are replicated to different nodes. The number of replicas is configurable down to the level of single files with a default count of 3. The placement of the replicas is chosen with respect to reliability and performance via HDFS's concept of *rack awareness* (i.e. nodes are aware in which rack they reside). With the default number of three one

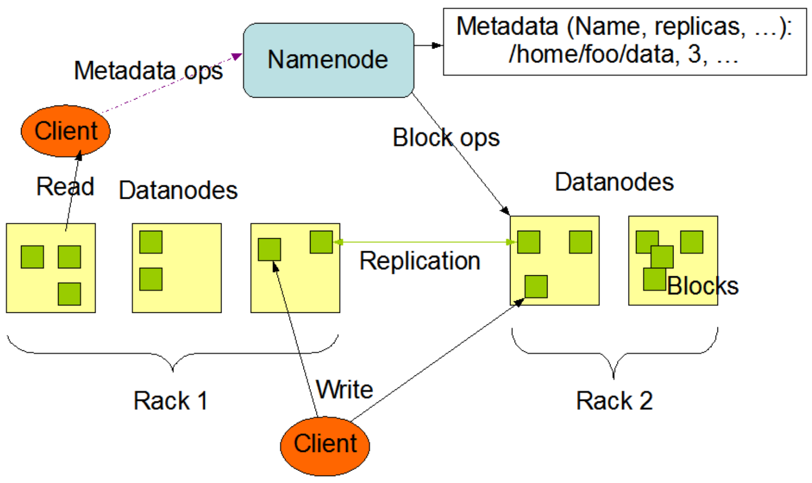


Figure 2.2.: HDFS Architecture [18]

replica is placed on one node, the second on another node in the same rack and the third on a node in a different rack, possibly even in a different data centre. While the split to different racks insures against rack failures, placing two replicas in one rack reduces inter-rack network traffic and thus improves performance. The concept of rack awareness also plays a role in Hadoop's MapReduce implementation as *map* and *reduce* jobs are distributed with the goal of maximizing data locality, i.e. assigning nodes those jobs whose input data is near.

Although this architecture impedes data loss conceptually very well (and the more the higher the replication factor), the reliance on a single *Namenode* renders the filesystem inaccessible in case of its outage. Hadoop supports a so-called Secondary *Namenode* that periodically checkpoints the first *Namenode* and thus can be used to restore filesystem metadata onto another node. It however cannot be contacted by the *Datanodes* or replace the primary *Namenode* in any way so if the *Namenode* fails, no access to the filesystem will be possible until it has recovered or been replaced. [25]

2.4.3. Streaming

Hadoop's MapReduce API has been written in and for Java and thus requires Java knowledge from the user in order to write MapReduce applications. To execute existing applications in a distributed fashion on a Hadoop cluster, users would have to write a wrapper application first if there was only the API, which is the reason why Hadoop introduced a utility called `Hadoop Streaming` that allows the creation of Map/Reduce jobs with arbitrary scripts or executables as a replacement for a *map* or *reduce* function.

Listing 2.4 shows an example call:

Listing 2.4: Hadoop Streaming WordCount Example

```
hadoop jar hadoop-streaming-2.6.0.jar \
    -input myInputDirs \
    -output myOutputDir \
    -mapper /bin/cat \
    -reducer /usr/bin/wc
```

In this example, the Unix utility *cat* – which simply prints the contents of the files passed to it to *Standard Output (STDOUT)*) – serves as the mapper and the utility *wc* – which, besides other measures, prints the number of words in a file – as the reducer. In other words, this Streaming call is equivalent to the MapReduce example above except that it also prints out other statistics.

Technically, Hadoop Streaming operates by passing the mapper its input split line by line via the *Standard Input (STDIN)*. The mapper processes the input and its output is interpreted line-wise as key/value pair where everything up to the first tab character is the key and the rest its value. The same applies to the reducer. This default input format can be changed via the *-inputformat* switch. Hadoop Streaming will be an essential part of our approach and we will come back to the topic of input formats later.

2.4.4. Yet Another Resource Negotiator (YARN)

The description of Hadoop's MapReduce architecture has been postponed to this section in order to put it into context with YARN. YARN, also referred to as MapReduce 2.0, has been introduced in Hadoop version 2.0 with the goal of allowing users to write more general purpose applications outside of the MapReduce schema but with HDFS as the data backend and Hadoop's resource management library for load distribution. In this model MapReduce is just one potential data processing paradigm and others can be added via user-defined applications which are not limited to running in batch mode. The architecture is depicted in Figure 2.3. [62]

Clients of the Hadoop cluster interact with its *ResourceManager* which consists of a *Scheduler* and an *ApplicationManager*. The latter is primarily responsible for handling job submissions and negotiating the first container, while the *Scheduler* is responsible for the allocation of resources to the different running applications under the constraints of the available resources. [27]

The counterpart on each node is the *NodeManager* that monitors the local resource usage of its applications and reports back to the *ResourceManager*. Both parts together form the data-computation framework. On the application side, each application has an *Application-Master* responsible for negotiating resources from the *Scheduler* in the form of *Container*s that are distributedly running on the various nodes and monitors their status. [27]

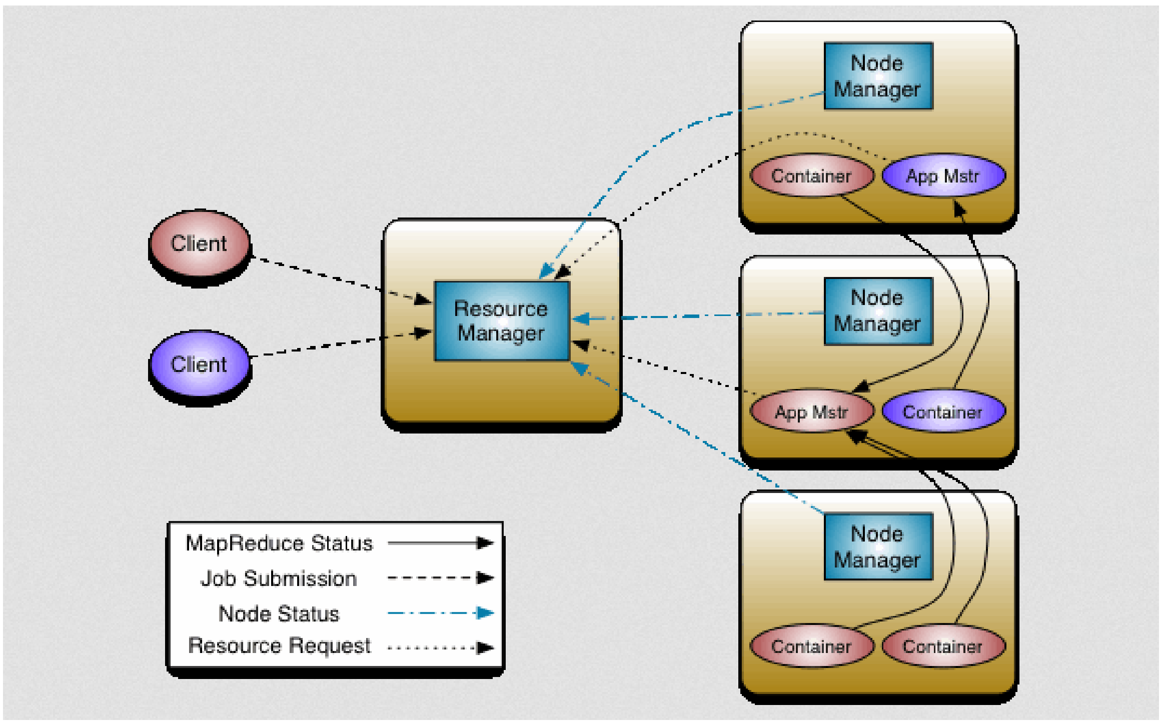


Figure 2.3.: YARN Architecture Overview [27]

3. Conceptual Design

In this chapter we will present the design of our approach from a high level perspective. Prior to that, we will describe the current situation that researchers in the need of higher computing power than their local resources provide are confronted with (Section 3.1). This will justify the need for an alternative approach which the remainder of the chapter will present split into two major topics: First, Section 3.2 illustrates the automated provisioning and configuration of computing resources with minimal user interaction. Second, Section 3.3 covers the job management component that allows users to move their computations into these acquired resources with no or only small adaptations to the original program.

3.1. Situation Analysis

3.1.1. Potential Use Cases

The forms of parallel computing are extremely diverse along different dimensions ranging from different hardware architectures over different parallel programming models up to varying implementations of these. We have already elaborated on the different forms of parallel computing in Section 2.1 and pointed out that we are employing the data-parallelism approach.

Furthermore, the applications of parallel programming are no less diverse than its forms of implementation. Nowadays, parallelization is utilized in all possible kinds of environments and use cases from desktop applications running on commodity desktop hardware for better user experience to large-scale scientific experiments both in research and industry running on specialized supercomputers. This diversity warrants for a definition of the use cases our approach is designed for from a high-level perspective which will later be specified more technically.

Examplary Use Case

Steve is a computer scientist working as research fellow at the University of Tardiness. In a software he developed for his research he employs machine learning techniques via the popular Weka [47] library. In the beginning, the execution of the software on

his work machine finishes in acceptable time frames of minutes to a few hours. As his experiments get more numerous over time and more complex in nature due to the application of more machine learning algorithms, the run times grow substantially and now range from a day to even weeks.
Steve knows about parallel programming and works on a multicore computer so he adapts his software in a way that it spawns as many threads as there are cores available handing one experiment (each defined in its own input file) to each of them. The speedup is satisfying, however the absolute run times are still unacceptably high. As Steve knows, the university's data centre runs a HPC cluster for message passing applications. However, he reckons rewriting his software from the ground up for MPI is too time-consuming and the queue for the cluster is long due to recently increased user demand. The purchase of an own cluster for himself and his research group is out of question due to financial constraints, so he instead considers the use of Cloud Computing resources. Though this would be a more economical choice, it still requires Steve to know about *nix-like system administration and even if the servers are setup the question of how to parallelize his computations across them remains open.

The example in the box above informally describes the targeted user base and use cases for our approach. More generally, the following characteristics define our area of application:

- *Hardware*: Users have limited local computing resources that are not sufficient to execute their research software in an acceptable time frame. Furthermore, it is assumed that either the financial resources are a constraint for the purchase of adequate hardware or the computations are too large to be finished on a single machine in acceptable time.
- *Software*: The software has not been designed upon a parallel programming model. At best, it launches threads for different inputs, thereby utilizing multiple CPU cores, but it has no support for cross-node parallelization models. All that can be assumed is that the computation the software performs is configurable via its input parameters and that the total computation can be split up into chunks by these means.
- *Users*: The users are technologically savvy, usually writing their own software which at the same time is the software that should be parallelized. Their experience with parallel programming is however either limited or time constraints limit the possibilities for a post hoc adaptation to a parallel programming model.

As a side note, our approach is built upon Ubuntu Linux and while it can be adapted to other Linux distributions easily, this is not the case for non-Unix-type operating systems. This implies that the user applications also need to be runnable on Linux. In the next sections we will briefly evaluate the status quo for such scenarios and its potential shortcomings.

3.1.2. Status Quo

With a scenario as exemplified in the previous section the user can make choices alongside the dimensions of hardware and software while the user side naturally is constant. The sheer multitude of potential combinations makes it impossible to describe them all, so only those that seem most practically relevant will be mentioned.
On the hardware side there are mainly two options:

1. Utilizing existing infrastructure for computationally intensive applications. Leaving aside specialized supercomputers, which are usually neither designed nor available to the average researcher, this infrastructure comes most often in the form of small clusters belonging to a single institution (e.g. a university data centre) which is made available to its researchers or as federated resources from multiple organizations.

2. Renting hardware resources in the form of Cloud Computing.

Often, option one will not be available, because not all research institutions maintain respective infrastructure. When both options are available, a cost comparison may – depending on the particular use case and pricing models – put existing infrastructure in favour of Cloud Computing, but from a technical perspective it may be inferior. Firstly, its flexibility may be limited due to organizational constraints. For example, there may be an application process researchers have to go through because of scarcity of computing resources and once they have been given access they may still be limited by a priority queue and an overall insufficient resource quota. Secondly, and this brings us to the other dimension, the software side of the infrastructure may be incompatible with the defined scenario.

Research clusters are often designed for explicitly parallel applications that are built upon the message-passing idiom – for example in the form of an *Message Passing Interface (MPI)* application – or a shared memory model, for example OpenMP applications. In other words, the application contains the parallelization logic and is started as one long-running task which exits after completion. Adapting a non-parallelized application for aforementioned programming models is generally a challenging task [61] and even intractable in the worst case due to hardly reversible prior design choices.
Underlying the execution of parallel programs in such clusters is usually a batch system that queues, launches and monitors user jobs (e.g. TORQUE [44]). Conceptually, these batch systems allow for task parallelism in that instead of launching one inherently parallel application the same non-parallel application can be launched concurrently multiple times with different input parameters. Given that, it is in principle a solution for the scenario we

defined. The general problems of the use of shared clusters however remain.

Our approach will hence focus on the use of on-demand Cloud resources and aims to provide an accessible user interface which hides from the user all aspects of parallelization so that the distributed job execution is almost no more complicated than local execution of the software.

3.2. Cluster Deployment

The first step in the workflow of our approach is the provisioning and configuration of resources at a IaaS Cloud provider where resources here means a set of Linux-based virtual machines. As described in the backgrounds chapter this process is twofold: First, virtual machines have to be provisioned, i.e. created on the service provider's hardware with a base operating system and then the virtual machines need to be configured with the necessary software. Both tasks are handled by our `cluster` tool. Figure 3.1 depicts the process from the point of view of the user.

3.2.1. User Interaction

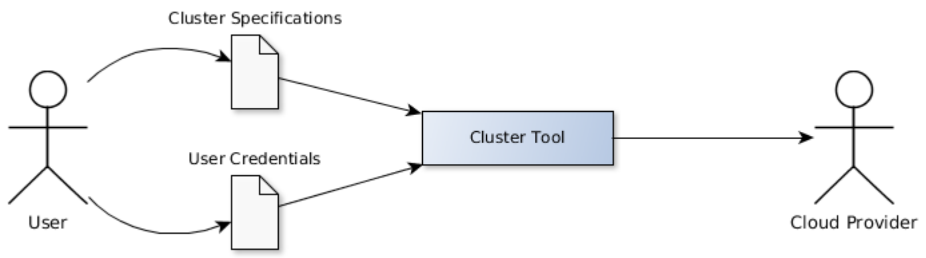


Figure 3.1.: Cluster Deployment Overview

The figure shows that the user's involvement in the deployment process is limited to providing two sets of inputs to the `cluster` tool. The first are the user credentials which the `cluster` tool needs to interact with the Cloud provider. As is usually the case with online services, Cloud providers have a database of user accounts for their customers against which they need to authorize themselves. Table 3.1 lists the credentials needed as input for our tool.

Parameter	*Purpose*
Username	Identifier for the user, e.g. email address
Password	Secret key only known to the user
Tenant	Project identifier

Table 3.1.: User Credentials

While username and password are well-known parameters for login procedures, tenants are specific to OpenStack – the IaaS software stack that the GWDG employs for its Cloud infrastructure – and represent a project within the Cloud, i.e. resources that conceptually belong to each other. The second set of inputs is the cluster specification which gives the *cluster* tool the necessary parameters for determining the size and computational capabilities of the cluster to create (Table 3.2).

Parameter	*Purpose*
Node Count	Number of virtual machines
Node Flavor	Computational capabilities of each node
Access key	Access key for remote control of the nodes

Table 3.2.: Cluster Specification

The size of the cluster is defined by the number of virtual machines it operates with and each machine's computational capabilities expressed by the so-called flavour of the machines which is an identifier for a particular combination of CPU cores, memory and disk space, besides potentially other specifications. Finally, the access key is needed to securely remote control the machines. We will further concretise the inputs in the context of the implementation (Chapter 4).

3.2.2. Deployment Process

3.2.2.1. Initialization

With the user credentials and the desired cluster specification the `cluster` tool can communicate with the Cloud provider on behalf of the user in order to instantiate the virtual machines.

The first step in the process is the creation request by the `cluster` tool of which there is one request for each virtual machine. The request is directed to the API of the Cloud provider. The Cloud provider processes the request by creating the virtual machines, given that the usual conditions – valid login credentials, eligibility to utilize further resources etc. – are met. Finally, if the request was successful, the information needed for accessing the virtual machines are returned to the `cluster` tool.

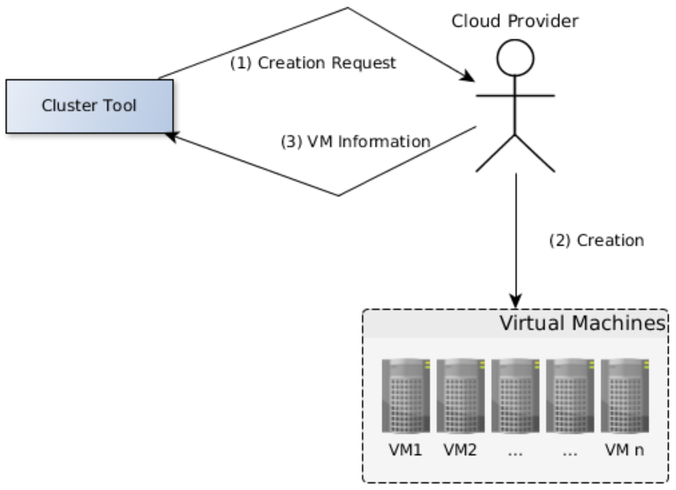


Figure 3.2.: Virtual Machine Instantiation

3.2.2.2. Configuration

The virtual machines with the base operating system are available for use after the initialization step. At this step, they have no knowledge about each other and therefore don't operate as a cluster. Thus, the next step in the workflow is the configuration of the nodes.

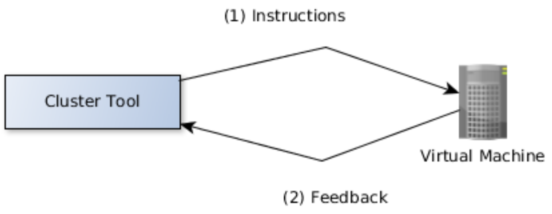


Figure 3.3.: Virtual Machine Configuration

The process is exemplified in Figure 3.3 for one node and it is repeated for each node. The `cluster` tool directly communicates with the machine to send instructions and receives feedback from the machine about success or failure of these instructions or additional information attached to it. Instructions here include all steps required to build a cluster with the virtual machines, i.e. changes to the system setup so the different nodes are known to each other, installation of required software as well as its configuration.
At the end of the configuration process the cluster is ready to receive jobs from the second utility in our toolbox, the job tool.

3.3. Job Management

Once the `cluster` tool has setup the infrastructure, the user can execute her applications on it. For this second part of the workflow, the job management, we introduce the `job` tool which requires (1) that the user's application adheres to a particular application structure and (2) that the user describes the desired execution scenarios of the application in job description files.
Figure 3.4 depicts the job management which will be described in the following section.

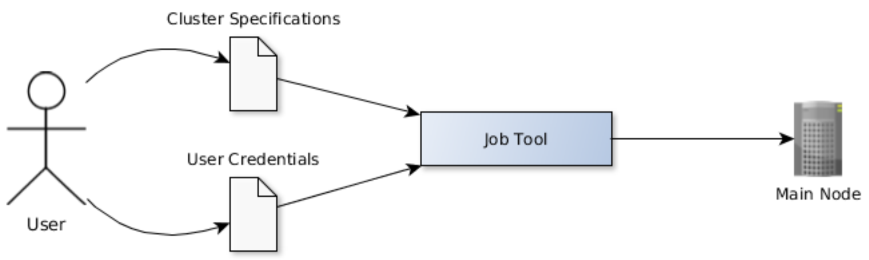


Figure 3.4.: Job Management Overview

3.3.1. Required Application Structure

Frameworks always to a certain degree impose a structure onto the applications that utilize them and our approach marks no exception. This is paramount, because regardless of which parallelization approach is chosen, the framework needs to make assumptions about the application to bring it to execution in the intended way. In the context of this work, satisfying the requirement that user applications should be parallelized without implementing a parallel programming model in the application itself, the prerequesites are limited to (1) the way the application receives input specification from the user, i.e. how the user tells the application which input data to process when starting it manually and (2) the directory structure of the application including the input data and run configurations.

There are only minimal requirements to the first point, i.e. that the application can be executed via the command line and takes as last input to its call a path for the input file it should process.
The applications need further be layed out in three directories:

- **fixed**: The *fixed* directory contains the application executable as well as all other constant data, for example third-party libraries or data sets which the application processes. The data in this directory remains fixed over the course of the life-cycle in the Cloud.

- **transient**: Data that may change between different executions belongs to the *transient* directory. This includes e.g. run configurations for which the user would like to retain the possibility of modifications during the lifecycle in the cluster.
- **out**: The *out* directory is initially empty and later on holds the output data the application writes.

It is noteworthy that the *transient* directory is optional from the point of view of our framework. Technically, the user could place all input files in the *fixed* folder if no later changes are intended, but the convention is to place them in the designated directory nonetheless.
Besides for technical reasons which will be described in Chapter 4 this directory structure was chosen to cover a wide variety of applications with few or no modifications given the simplicity of the layout. The case studies in Chapter 5 will also discuss the steps that were necessary to adapt them to our structure.

3.3.2. Job Description

Besides adhering to the required application structure, the user needs to define in so-called job configuration files which input files the application should process, where its output should be saved and which parameters the program needs for its execution. In our terminology, a *job* is a particular combination of program parameters and inputs that represents an analysis. As an example, a job could represent the program configuration for the analysis of the weblog data of a specific one-day period or the clustering of recently (e.g. during the last hour) crawled data.

Parameter	*Purpose*
Name	An arbitrary identifier that describes the job
Command	The executable's name
Parameters	Further parameters that the user wants to pass to the application
Output Type	Whether the application prints its output to screen or to file
Input/Output mapping	Relates input to output files

Table 3.3.: Job Description Input

The job description (Table 3.3) requires only a handful of parameters from the user. Besides an arbitrarily chosen name which helps the user identify the purpose for which he wrote the description it contains the name of the application's executable, parameters the application should get passed and specifications regarding the inputs and outputs of the program. The latter inform the `job` tool about how the application outputs its results, i.e. whether it prints to screen or directly writes to files – which requires different treatment by

the tool – and how the output files should be called.
With the completed description file the user can proceed to executing the job.

3.3.3. Job Execution

Prior to execution of the application, it needs to be deployed on the cluster. Because this is a cluster-specific operation and generally not related to just a particular job of an application, the `cluster` tool was chosen to carry out this task. The purpose of the import is to bring the application directory into a filesystem shared by the nodes so that all of them have access to the application's data once jobs are run. We pointed out earlier, that the *fixed* directory is assumed to stay constant. This assumption is made because data sets are often complex in size and should not be transferred from the user to the cluster every time a job executes in order to reduce network traffic. For technical reasons out of our reach that will be detailed in Chapter 4, it is not possible to transfer just the changes in *fixed* to the cluster, i.e. to perform an incremental update, so a complete new upload of the files is rendered necessary.

To execute the application on the Cloud cluster, the user passes the job description and its corresponding application directory – structured as described in Section 3.3.1 – to the `job` tool along with the requested operation, in this case "start". This will commence the following operations:

- Parsing of the job description file.
- Syncing of the application directory to the cluster.
- Scheduling of the job for distributed execution.
- Distributed execution and output aggregation.

Since this chapter covers the conceptual concepts of our approach we omit the technical details here which we instead will elaborate on in the next chapter. It is noteworthy however, that although for efficiency reasons the *fixed* directory may not be changed after import onto the cluster, the sync operation does commit changes to the *transient* directory.

3.3.4. Output Retrieval

The user application will typically produce one or more output files during its execution phase. Once the application run is finished, the `job` tool can be employed to download the output files from the cluster to the user's computer. Since all output files by convention reside in the *out* folder, the tool simply pulls this directory.

4. Implementation

The previous chapter covered the conceptual design of our approach from a higher-level perspective to give an overview of the system without going into the technical details. This chapter will make up leeway and present the implementation side. Section 4.1 will firstly give an architectural overview, putting the components introduced in the previous chapter into relation from a global perspective. Following, our two tools, the `cluster` and the `job` tool and their interactions with the other components will be described in Sections 4.2 and 4.3, covering not only the client side, but also the implementation on the server side.

We will not give a real-world application example in this chapter, because Chapter 5 will present two real-world examples in-depth.

4.1. Architectural Overview

This section will connect the foundations introduced in Chapter 2 and our tools introduced in the previous chapter. Figure 4.1 depicts our architecture.

Shown on the left are our two tools with their executables' names. Both of them are written in Python. The arrows in the diagram are labeled with the primary way of interaction between the different components. The fact that they are directional simply indicates who establishes the interaction, but does not imply that it is a one-sided communication. In fact, in multiple cases the communication is two-sided, e.g. between Vagrant and the Cloud API.
Our tools do not communicate with the Cloud provider or the virtual machines directly via SSH. Instead, the software configuration management tool Ansible is used as an abstraction layer. This applies to the `cluster` as well as the `job` tool and to all client/server communication. Furthermore, Vagrant, which is used by `cluster.py`, in turn also employs Ansible for the virtual machine configuration. This homogeneous approach has – besides the advantages of SCM already described in Section 2.3 – the benefit of having a common internal way of interacting with the cluster's nodes which unloads the source code from cluttering with administrative directives. The second component Vagrant interacts with is the Cloud API, in our case that of GWDG's Compute Cloud service which is based on OpenStack.
The Cloud API is only involved in the creation and destruction of the nodes as well as possibly rebooting or suspending them. All other management tasks are carried out using Ansible on behalf of the other tools. This is because Cloud APIs are generally only designed

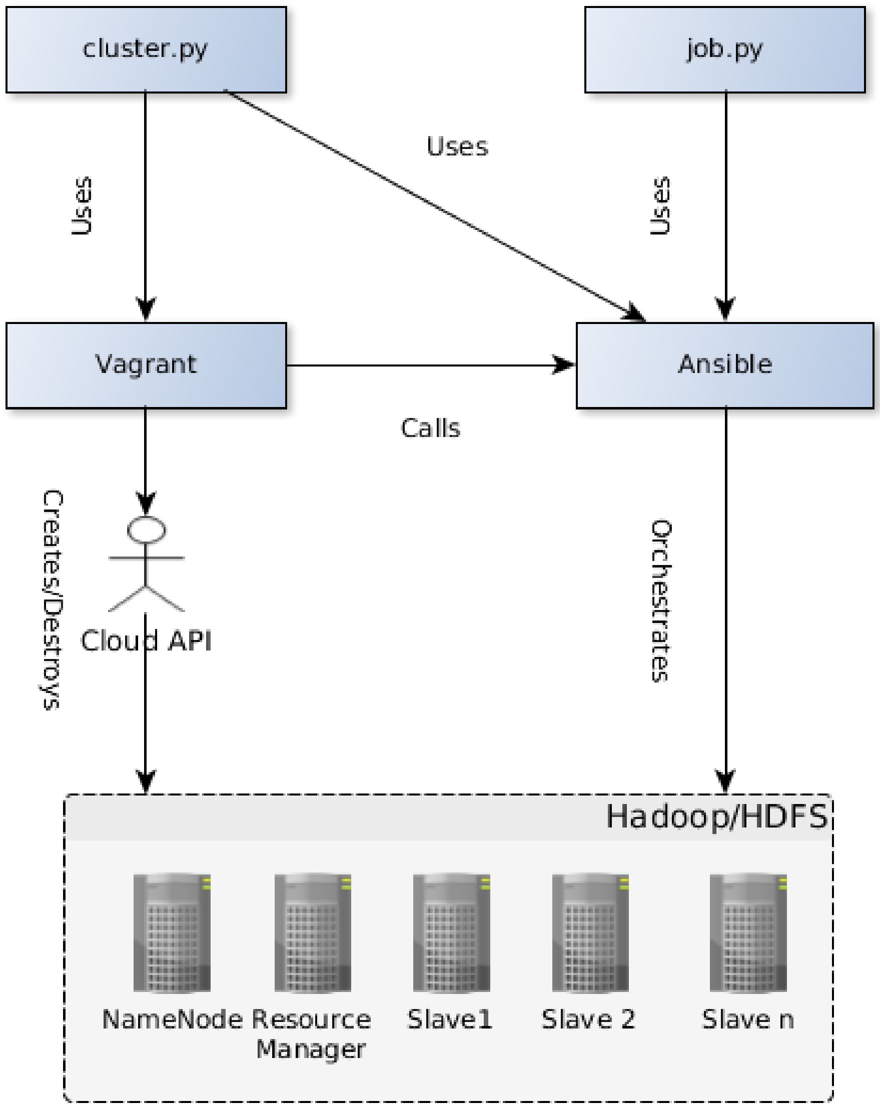


Figure 4.1.: Architectural Overview

to support infrastructure management operations, but are not involved in system management tasks.
Finally, the right side of the figure depicts the virtual machines with their respective roles. These are determined by Hadoop's client/service architecture which we elaborated on in Section 2.4. Therefore, there is always a designated *NameNode*, a *ResourceManager* and a number of slaves, where slave means those machines that run the *DataNode* and *NodeManager* services. While there are always only two dedicated nodes running the master services, the number of slaves is variable depending on the specifications by the user which in turn will depend on her quota. Though the nodes are independent in the sense that they don't appear as one logical machine to the user, they together form a cluster because of the running Hadoop services. This is illustrated by the ellipse around the nodes labelled "Hadoop/HDFS" to indicate that the objectives are to provide a Hadoop managed compute cluster with a shared file system (HDFS).

4.2. Cluster Tool

The `cluster` tool is responsible for the operations described in Section 5.1 plus a few minor ones. We will first give an overview of the corresponding functions, then describe the necessary configuration files and afterwards sketch the implementation of the different functions.

4.2.1. Function Overview

`cluster.py` has five different user-visible functions (i.e. not functions in the sense of Python functions) of which three belong to the cluster instantiation/destruction process and two to application management.

- **init**: Initializes the configuration for a new cluster.
- **deploy**: Deploys the specified cluster, i.e. initializes and configures the virtual machines.
- **destroy**: Irreversibly destroys the specified cluster, i.e. deletes its nodes and the local files associated with it.

The application management specific functions are:

- **project_init**: Initializes the project by syncing it to the specified cluster.
- **project_destroy**: Removes the project from the cluster. Counterpart to the previous operation.

4.2.2. Configuration Files

The program requires two configuration files. One is needed for the user to define the desired cluster size, the other in order for the tool to be able to communicate with the Cloud provider's API on behalf of the user. Both files are written in `JSON` (JavaScript Object Notation), which is a lightweight human-readable file format often used as an alternative to XML [31]. An example for the first file, *cluster_config.json* is given in Listing 4.1.

Listing 4.1: cluster_config.json Example

```
{
  "cluster": {
    "flavor": "m1.small",
    "key_name": "gwdgcc",
    "key_path": "~/.clusters/<clustername>/gwdgcc.pem",
    "number_of_nodes": "7"
  }
}
```

The file contains one single object named *cluster* with the configuration parameters as its members. The *flavor* defines the virtual hardware template that should be used for the virtual machines. The flavors are defined by the Cloud provider and vary in terms of number of cores, RAM size, disk space and others. The *number_of_nodes* parameter configures the number of nodes that should be instantiated. The number includes the *NameNode* and *ResourceManager* nodes, i.e. the number of slaves will be the total count minus two.
The remaining two parameters, *key_name* and *key_path* are related to authentication: In OpenStack and other IaaS stacks users upload the public part of their public/private key pair and tag it with a name. The locally stored private key part is identified by the *key_path* parameter, while the *key_name* corresponds to the identifier assigned to the key pair on the Cloud side. It is worth noting that the key upload to the provider is a manual task that the user needs to perform prior to deploying a cluster with the `cluster` tool.

Listing 4.2: user_config.json Example

```
{
  "user": {
    "username": "mgoettsche",
    "password": "secret123",
    "tenant_name": "12345"
  }
}
```

The *user_config.json* file is similarly structured. The *user* object containts a *username*, a *password* and a *tenant_name* parameter. The first two parameters are the login credentials for the user's account at the Cloud provider, the *tenant_name* identifies the OpenStack project in which the virtual machines should be created. We do not make further use of this concept, but only note that the tenant needs to be specified in order for Vagrant to communicate with the OpenStack API.

4.2.3. Functions

4.2.3.1. init

The *init* function creates a configuration skeleton for the cluster with a user-specified name. This step is very basic and consists of the following steps:

1. Creating a sub-directory in ~/.clusters with the specified name of the cluster.
2. Copying a *user_config.json* and a *cluster_config.json* template to this directory.
3. Copying a Vagrant template (*Vagrantfile.tpl*).
4. Generating a SSH key pair for the user to upload to the Cloud provider. The user is free to replace it with her own in case she already has a key pair.

In summary, *init* does neither instantiate nor configure the cluster, but prepares the local files needed for the next step.

4.2.3.2. deploy

The most complex operation implemented in the program is *deploy* which performs both the instantiation and the configuration of the virtual machines. We will therefore split up the discussion into two sections.

Instantiation As illustrated in Figure 4.1, Vagrant carries out the task of communicating with the Cloud provider via the OpenStack API. We described the foundations of the application in Section 2.3.1.1 and pointed out the need for a *Vagrantfile* that includes the infrastructure definition, i.e. which and how many virtual machines should be created alongside other settings.
deploy in this phase executes the following steps:

1. Parse the *user_config.json* and *cluster_config.json* files.
2. Modify the *Vagrantfile* template prepared in the previous step according to these configuration files.

3. Execute Vagrant.

In the default installation, Vagrant does not support interaction with OpenStack. To enable this support, we utilize a third-party plugin [46] that provides easy access to the most important API calls.
We will focus the discussion on the second step since the other steps are straightforward. Large parts of the *Vagrantfile* (full source code in Appendix B.1) consist of obligatory boilerplate code such as credential or network configuration (lines 11-26). Also, the instantiation of the *NameNode* and *ResourceManager* are statically defined, because there is always exactly one for each cluster. The aforementioned dynamic nature through the inclusion of arbitrary Ruby code however comes into play in the definition of the slave nodes due to the possibility of configuring the cluster size in *cluster_config.json*.

Listing 4.3: Vagrant Slave Instantiation

```
SLAVES_COUNT.times do |i|
    config.vm.define "slave#{i+1}" do |slave|
        slave.vm.hostname = "slave#{i+1}"
    end
end
```

The slave instantiation (Listing 4.3) is performed in a Ruby loop with the number of iterations equalling the number of slaves to be created. For each iteration, the loop body defines one virtual machine via the *config.vm.define* directive and assigns it a hostname.
As depicted in the architectural overview figure, Vagrant calls Ansible to actuate the configuration of the nodes. Listing 4.4 shows the necessary code for this setup.

Listing 4.4: Setup of Vagrant/Ansible Interaction

```
        resourcemanager.vm.provision :ansible do |ansible|
            ansible.verbose = "v"
            ansible.sudo = true
            ansible.playbook = "$MTT_HOME/cluster/deployment/site.
                yml"
            ansible.limit = "all"

            slaves = (1..SLAVES_COUNT).to_a.map {|id| "slave#{id
                }"}
            ansible.groups = {
                "NameNode" => ["namenode"],
                "ResourceManager" => ["resourcemanager"],
                "Slaves" => slaves
            }
```

```
13          end
```

The Ansible setup is placed in the last virtual machine definition of the file to prevent premature execution, i.e. before all nodes are ready. Ansible requires primarily two components, the *Playbook* containing the cluster configuration code and an *Inventory* mapping IPs to hostnames as well as grouping them. The main Playbook resides in *$MTT_HOME/cluster/deployment* where the environment variable *$MTT_HOME* stands for the path where our tools are installed. Lines 8-12 define the Inventory, creating a group for each of the different roles.
The result of this phase – `cluster.py` having executed *vagrant up --no-provision* – are a number of virtual machine instances running the base operating system (in our case Ubuntu Server 14.04). The next paragraph will give attention to their actual configuration.

Configuration In this step, all the `cluster` tool does is to execute *vagrant provision*, thereby indirectly launching Ansible with the Playbook configured. Ansible encourages a modularized configuration via the concept of *roles* which are analogous to include files, i.e. the logically split up configuration is aggregated upon execution.

Role	*Purpose*
common	Installation of required base software
hadoop_common	Hadoop download and system preparation
configuration	Hadoop configuration
format_hdfs	HDFS formatting
services	Hadoop services start/restart

Table 4.1.: Ansible Roles for Cluster Deployment

One *role* should encapsulate the automation of a specific task or service. We use five roles in total as described in Table 4.1. These are executed in the shown order because they depend on the outcome of prior execution of other roles. For example, the Hadoop configuration cannot be replaced with our production configuration before the base one has been installed and the respective services cannot be started without Java being installed.
Some of the role files are themselves lengthy or involve lengthy templates and do not have parts that stand out, so for brevity their code will be omitted here. Instead, we will shortly describe the state after execution of each role and which Ansible modules dominates:

- *common*: Required base software that is not part of the default Ubuntu Server deployment has been installed, e.g. Java and *Network File System (NFS)*. The different virtual machines have aliases set up in the network configuration so that they are known to each other via their name (e.g. *slave1*) instead of just their respective IP addresses. The installation procedures were performed via the `apt` package and the network configuration via `template`.

- *hadoop_common*: The Hadoop package has been downloaded from a mirror (module `get_url` and uncompressed to its target directory (module `shell`). The system user and system group have been created and the *hadoop* user been given *sudo* permissions. The modules `user`, `group`, `lineinfile` and `file` have been utilized for these tasks.
- *configuration*: The data directories for Hadoop and its NFS wrapper (making the HDFS available via a *POSIX* compatible file system) have been created via the `file` module as well as that its mount point has been set up. The multitude of required Hadoop configuration files for the setup of YARN, MapReduce et al. has been copied from previously written local templates into the target directories (module `template`).
- *format_hdfs*: The HDFS file system has been formatted using the `shell` module. Arguably, this small task could be part of the previous step as well.
- *services*: The various Hadoop services have been started via the `shell` module. For services that can be managed using standard init systems, the `service` module is utilized for this, but Hadoop services in its standard distribution are started differently. This step takes into account the different roles the nodes play, Table 4.2 shows which service(s) are started on which machine. Finally, the HDFS is mounted via the NFS wrapper on all nodes using the `mount` module.

Service	*Hosts*
NameNode	namenode
ResourceManager (YARN)	resourcemanager
JobHistoryServer	resourcemanager
Portmap (required for NFS wrapper)	namenode
NFS Wrapper	namenode
NodeManager	All slaves
DataNode	All nodes

Table 4.2.: Services running on different Node Types

Notably, all tasks in the configuration process are idempotent, meaning in this context that repeated runs of the process will not change the resulting system. This may seem unnecessary for a configuration step that is supposed to be performed only once, but can be helpful in at least two cases. The first would be a later upscaling of the cluster, i.e. the addition of nodes which need to be configured while ensuring that the other nodes are not affected. The second would be temporary network problems that cause some operations during the process to fail, thus leaving one or multiple nodes in a transitional state. Rerunning the deployment configuration then finishes it and brings them into the desired state.

4.2.3.3. destroy

The destroy operation is very straightforward and simply a shorthand to the equivalent Vagrant operation. Besides executing *vagrant --destroy* the tool removes the corresponding local cluster directory in ~/.clusters. The only motivation for implementing this operation is to provide the user with a thorough interface so that she does not have to work with Vagrant directly.
In effect, *destroy* shuts down and removes all virtual machines including the data stored in the cluster, thereby freeing up resources in the user's Cloud account. Output data contingently stored in the cluster that has not been downloaded yet must be retrieved prior to the destruction in order to avoid data loss.

4.2.3.4. project_init

We explained earlier that the motivation to make the initialization of the project a cluster operation rather than a job operation. To recall, the assumption is that a project consists of fixed data that remains constant and transient data such as experiment configurations which the user may want to edit e.g. as a result of new information. The initialization takes this into account by performing the initial upload of the project directory as a one-time operation, thereby putting the cluster into a state where corresponding jobs can be started. Job operations upon execution resync only the *transient* directory.
The rationale for this is the lack of random read/writes in HDFS. Under the assumption that files once created need not be changed, the Hadoop team decided in favour of a write-once-read-many access model [18]. This poses a problem for synchronization software that follows the principle of transferring only deltas for modified files instead of replacing the files altogether to save network traffic and reduce the amount of input/output operations. Since the assumption of a constant *fixed* folder appears justified by our two case studies, the user can simply destroy and reinitialize the project in the rare occasions of changes to *fixed*.
The *--project_init* function imports the data to the cluster in two steps:

1. Upload of data to the *NameNode* via Ansible's `synchronization` module.
2. Copying of the data to HDFS.

We use the *NameNode* as the entry point to the cluster to upload the data to it, but any other node could just as well be used, because naturally all of them have access to the

shared file system. Once copied to the node, the data can be moved from the node's local storage to HDFS, thereby making it available to all nodes. The Ansible code is listed in Appendix B.2.
Besides importing the data into the cluster, the `init` operation creates an invisible *.job_config* directory in the project folder which initially only contains a file that links the project to the cluster it has been imported to and will later on hold the job description files.

4.2.3.5. project_destroy

The counterpart to the project initialization is its removal via *--project_destroy*. This operation reverses the initialization step by removing the data from the HDFS file system and the *NameNode*. Again, these steps are supported by Ansible. The corresponding code is listed in Appendix B.3.

4.3. Job Tool

This section covers the operations of the `job` tool that implements the functionality described in Section 3.3. Again, we will first provide an overview of its functions, then its job description format and finally sketch the implementation of the functions.

4.3.1. Function Overview

`job.py` has the following four user-visible functions.

- **init**: Initializes the configuration for a new job.
- **start**: Starts the given job.
- **download**: Downloads the output of the job if it has finished already.
- **delete**: Deletes files corresponding to a job.

In our terminology, a *job* is the template for runs of an application with a particular combination of input files and input parameters. Therefore, a *job* does not describe a single execution of the application, but the configuration for (possibly repeated) executions.

4.3.2. Job Description Format

As in the case of the cluster configuration, we chose JSON as the file format for the job descriptions. Each job is defined in its own file with the parameters introduced earlier (Section 3.3.2). Listing 4.5 presents an example for a simple job description.

Listing 4.5: Job Description Example

```
{
  "job": {
    "name": "md5sum of input files",
    "command": "md5sum",
    "parameters": "",
    "output_type": "stdout",
    "input_output": {
        "directory": [
        {
          "input": "../transient/",
          "output": "../out/"
        }
      ]
    }
  }
}
```

The purpose of this job is to execute the md5sum command on all files in the *transient* directory. `md5sum` [12], as the name suggests, computes the *MD5* hash sum of the input and prints it to the standard output.
All parameters are members of the single object named *job* of which there is exactly one for each job description file. Since the parameters have already been described earlier, we will focus on the technical aspects of the output type parameter and the input/output-mapping here.
Some applications print their results to the system standard output, others write them directly to files. Our `job` tool cannot distinguish these two types without user support, but needs knowledge about the type to handle the applications differently: Since the applications are executed in the context of the servers, any output to the screen is lost if not redirected. This redirection and storing to files is performed by `job.py` if the *output_type* is specified as *"stdout"*. In this case, the tool redirects the output to files of the same name as the input files in the respective output directory so they can be retrieved later on. If *output_type* is set to *"file"*, the user application is responsible for writing the output to files directly. Since the file system is POSIX compatible, this does not require any particular precautions in the application. The only requirement is that it writes to the *"out"* folder.
The input/output-mapping is an array whose members are either *"directory"* or *"file"*. For directories, all files in that path are regarded as input files which is a shorthand for the user to listing all files individually via the *"file"* type. The latter option will be used if only a subset of the files should be processed. This is useful if the available pool of files should be divided up into different smaller experiments. The case studies in Chapter 5 will present

examples for this kind of job description.

4.3.3. Functions

4.3.3.1. init

As in the case of `cluster.py`, the *init* function here also serves to initialize a new object, in this case a *job*. This operation consists of just two steps:

1. Creating a sub-directory in the invisible *.job_config* with the name of the newly created job.

2. Placing a job description template in the *.job_config* directory.

The job template contains no runnable description, but is well structured and contains all parameters to ease adaptation by the user before starting the job.

4.3.3.2. start

The *start* operation is the most complex one of the `job` tool involving logic on both the client and server side which we will explain separately.

Client Side First, *start* parses the job description file and checks whether its input/output-mapping is valid. While this mapping may by design include both, directories and files sections, the server side – for reasons to be explained in the next paragraph – requires a file-based mapping. This conversion is performed upon job start to incorporate any changes that the user may have made to the job description. Technically, file entries are simply copied into an associative array and the directory entries are converted into file entries by scanning the directory's contents and creating an entry in the array for each file found. `job.py` then serializes this associative array for later transfer to the cluster.

Second, using the parameters from the job description, the function creates a Python script from a template that will later serve as the Hadoop Streaming mapper script. See below for a discussion of this script.

Lastly, the Ansible *job_start.yml* playbook (Appendix B.4) is launched. The first part of the playbook synchronizes the project files to the cluster, more specifically just the *transient* directory. The second part launches the Hadoop Streaming job which marks the transition to the server side.

Server Side We begin our description of the server side job execution with the corresponding Hadoop call:

Listing 4.6: Server-side Job Execution

```
1 /opt/hadoop/bin/hadoop jar /opt/hadoop/share/hadoop/tools/lib/hadoop-streaming-*.jar
2         -D mapreduce.map.cpu.vcores=2
3         -input /{{ project_name }}/.job_config/{{ job }}/input_list.txt
4         -output /mr_tmp
5         -mapper /mnt/hdfs/{{ project_name }}/.job_config/{{ job }}/mr_job.py
6         -inputformat org.apache.hadoop.mapred.lib.NLineInputFormat
7 && touch /mnt/hdfs/{{ project_name }}/.job_config/jobdone
```

Line 1 references the *JAR* file for Hadoop Streaming. Line 2 assigns each map task two cores. As described in Section 2.4.3, Hadoop expects an input and an output location. In this case (line 4), the output directory is a dummy, because we are not interested in Hadoop's diagnostic output. The input parameter refers to a file created on the client side, which simply lists line by line the files that should be processed and thus equals keys of the serialized associative array discussed above. This array is used in the mapper script specified in line 5 (more on that below), but cannot be used as the input parameter because it does not comply with any of the supported file types. Of these, we chose the *NLineInputFormat* instead of the default *TextInputFormat*, because it ensures that each map task receives just one input file and each a separate one which is what we need for optimal distribution of the workload. Finally, upon successful execution of the Hadoop job, the *jobdone* file is created as an indicator for `job.py` that the output can be downloaded. The variables in the call – identifiable by their encapsulation in double curly braces – are replaced by Ansible prior to execution.

From the start of the Hadoop job on it remains in charge of managing the different map processes, monitoring the overall progress, failure handling et cetera, but the actual execution of the user application is encapsulated in the mapper script which will be discussed now. A version shortened to the key parts is shown in Listing 4.7.

Listing 4.7: Mapper Script

```
... module imports ...
... variable declarations ...

io_mapping_file = open('/mnt/hdfs/$project/.job_config/$job/io_mapping.pickle', 'r')
```

```
5  io_mapping = pickle.load(io_mapping_file)
6
7  for task in sys.stdin:
8      (key, task) = task.rstrip().split('\t')
9
10     # Build task command
11     cmd = command + '␣' + parameters + '␣' + task
12     if output_type == 'stdout':
13         cmd += '␣>␣' + io_mapping[task]
14
15     # Execute task
16     status = os.system(cmd)
17
18 os.system('cp␣/tmp/local_job_out/*␣%s' % os.path.join(projectdir,
       'out'))
```

Line 4-5 open the serialized input/output-mapping and deserialize it. Importantly, the file string in line 4 is modified prior to the execution as *$project* and *$job* are variables. The replacement of these variables with the true values occurs on the client side prior to the job execution. The mapper script receives its input from Hadoop via the STDIN channel. The Streaming call above implies that each mapper process normally receives just one input file. However, for fail safety the script iterates over its input (line 7) to be able to deal with multiple input files loop-wise. Due to the choice of *NLineInputFormat* as the input format, the mapper receives not just the file name from the input list, but a key/value pair separated by a tabular character with the key corresponding to the character position of the input list (line 8). The remainder of the for loop prepares the call to the user application and executes it. Line 13 marks the different treatment depending on the output type of the application: If the user application prints to STDOUT, the command line call arranges for the output redirection into the file name specified by the user in the input/output-mapping (line 13). Finally, after the user application has exited (line 16), the script copies the locally cached output files to the HDFS so that all node's output files are available at a common location.

4.3.3.3. download

The *download* function allows the user to retrieve the output files generated by her job run. The download needs to be implemented as a pull operation (i.e., the user initiates the data transfer) separate from the job execution, because the job may finish at a time when the user's machine is not running and the data could not be transferred if that is the case.
The download operation uses an Ansible *play* shown in Appendix B.4.

Listing 4.8: Ansible File for Result Download

```
---
# job --download play

- hosts: namenode
  remote_user: root
  sudo: yes
  sudo_user: hadoop
  tasks:
  - stat: path=/mnt/hdfs/{{ project_name }}/.job_config/jobdone
    register: jobdone
  - fail: msg="Job has not finished yet"
    when: not jobdone.stat.exists
  - synchronize: mode=pull src=/mnt/hdfs/{{ project_name }}/out
     dest={{ project_path }}
    when: jobdone.stat.exists
```

Prior to the tasks, the Ansible file specifies that it should work with the *namenode* machine and be executed as the user *hadoop*, which is the user that also runs the Hadoop services. The actual *tasks* section employs three different Ansible modules. First, `stat` checks whether there exists a file called *jobdone* in the project directory and registers the result of the check in a variable of the same name. The *jobdone* file is created after the job run has finished and thus tells `job.py` if the download may be started. If the result of the check is negative, Ansible aborts with a message that the job has not finished yet. Otherwise, `synchronize` performs a pull operation, syncing the project's *out* folder on the server side to the client's machine.

4.3.3.4. delete

The *delete* operation solely removes local traces of a configured job, i.e. its directory and job description file in *.job_config*.

5. Case Studies

In this chapter we will examine the practicability of our approach with the help of two applications written by researchers of the group of Software Engineering for Distributed Systems. We chose these two applications because they represent real-world use cases for our approach and therefore they can provide realistic results for the judgement of the feasibility of our work. We will begin with the description of our test environment in Section 5.1. Following, in Section 5.2 we will discuss the metrics with which we evaluate the performance of our approach. The remainder of the chapter will then cover the two case studies and their results (Sections 5.3 and 5.4).

5.1. Environment

5.1.1. Cluster Specifications

For mainly two reasons, we chose the Compute Cloud service of the GWDG as the Cloud service for our evaluation. The first reason is that the GWDG is the main provider of IT services for the University of Göttingen. Therefore, should the approach turn out to be useful and also generalizable enough, it may be interesting for other local researchers for whom the Compute Cloud is possibly the most cost-effective choice. The second reason is that research grants of e.g. Amazon are no longer as easily available as in the past and there were no resources available for purchasing commercial Cloud resources for this thesis. The GWDG provided an account with an initial quota of 10 virtual machines of a custom flavour which was later on extended to 15 for testing the deployment of a second cluster next to an existing one.
In expectation of possibly large data sets, the storage-rich private flavour *hadoop* was created for us with the specifications listed in Table 5.1.

Virtual CPUs	2 Cores
Memory	4GB RAM
Storage	275GB HDD
Architecture	64 bit

Table 5.1.: Hadoop Flavour

Our evaluations were performed on a cluster consisting of 10 machines of this flavour. Naturally, using a different flavour with other characteristics would yield different results, which may be better or worse depending on the respective user application. However, the rationale of our evaluation was not to find the best suited infrastructure for our case studies, but to execute them in an average setup to explore the performance in non-optimized configurations.

5.1.2. Software Versions

We here list the versions of all software that we used and deem directly relevant. Noteworthy, while our approach may be more sensible to changes in some software than in other, it should not be affected by minor or even major version changes due to the backward compatibility in the underlying foundations. Table 5.2 lists the software.

Software	*Versions*
Hadoop	2.5.2
Ansible	1.9
Vagrant	1.6.5
Java	OpenJDK 1.7.0
Python	2.7.6
Linux	Ubuntu Server 14.04

Table 5.2.: Environment Software Versions

5.2. Metrics

The most widely used metric to express the relative performance gain from parallelization of a task is the *speedup* which is defined as[58]:

$$Speedup = \frac{\text{Execution time}_{\text{old}}}{\text{Execution time}_{\text{new}}}$$

where the *old* execution time is the wall-clock time for executing the entire task without parallelization and the *new* execution time the duration with parallelization. Ideally – ignoring cases of super-linear speedup irrelevant for our case – the speedup for p processors/nodes is itself p, meaning that for p nodes the execution time reduces to *1/p*.
We will employ the speedup as our main and only metric for the following reasons:

- Most importantly, the speedup is the metric motivating this thesis, i.e. we want to evaluate how much speedup can be gained with no effort on the user side. Again, we are interested in the wall-clock time, because it is the metric of interest for the user.

- Other metrics such as the average utilization of the infrastructure depend too much on the particular use case. As an example, given a fixed cluster size of n it is obvious that a job with m input files where $m < n$, the cluster cannot possibly be fully utilized, without this being a drawback of our approach. Even with $m > n$, the utilization depends primarily on the degree of input heterogeneity, as will become clear from the discussion of our case studies.
- Yet other low-level metrics such as the communication overhead which is of interest e.g. in message-passing parallelization are neither measurable nor of interest here because – being determined by Hadoop – they are out of scope of our implementation, thus not improvable and implicitly a determinant of the overall speedup.

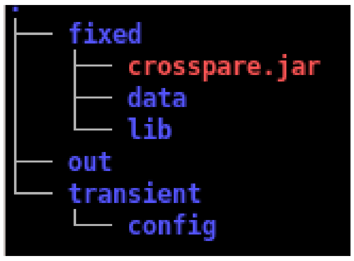
Our evaluation of the speedup, here being the quotient of the execution time on just one node (i.e. the time of non-parallelized execution) and the time of a parallelized job in the cluster, will investigate if it is beneficial to use our approach in terms of reduced execution times and quantify this for different scenarios.

5.3. CrossPare

5.3.1. Application Description

`CrossPare` is an application developed by one supervisor of this thesis, Steffen Herbold, that employs machine learning algorithms on software project data to identify potentially erroneous application components. More specifically, it takes as input data from the TERA-PROMISE REPOSITORY [42] that collects metrics from the different components of a pool of open-source projects on the source code level as well as the information whether each component contains bugs or not. Examples for such metrics are the lines of code (LOC) or the number of methods in a class. By feeding this data into a set of configurable machine learning algorithms, CrossPare attempts to identify the most predictive metrics. Depending on the type and number of selected algorithms – which are defined in so-called *experiments* – the run time can vary substantially. We will describe this in more detail in the next section. The application directory structure had to be slightly adjusted to fit our requirements detailed in Section 3.3.1. However, since the application has no hard-coded input/output-mapping defined in its code, but rather reads it from its input files, the restructuring merely involved some file moves and slight modifications to the experiments, but no changes on the source code level.

Figure 5.1 depicts the directory organization after the restructuring. *crosspare.jar* as the application's *JAR* file resides in *fixed* along with a directory containing libraries it depends on (*lib*) and the *data* directory that contains the input data. The experiments are stored in a sub-directory of *transient* because there may be changes to the pool of experiments. For example, an algorithm may be regarded as unnecessary after the first run or new experiments



```
.
├── fixed
│   ├── crosspare.jar
│   ├── data
│   └── lib
├── out
└── transient
    └── config
```

Figure 5.1.: CrossPare Directory Structure

may be added, while the application itself remains unchanged.
Finally, it is important to note that CrossPare uses thread-level parallelization, i.e. each process launches as many threads as there are CPU cores if the number of input files to that process is large enough. Thus, for a dual core machine with two or more input files, two threads will be started. This fact will be important in our evaluation.

5.3.2. Evaluation Setup

As mentioned above, experiments are defined in XML files. Listing 5.1 shows exemplarily one of the experiment configurations we examined. Important for our discussion are the lines 3 and 13 as well as 9-12. The prior define the input and output data locations and mark those parameters that needed to be adapted to match our directory structure. The latter lines define the *trainers*, i.e. the machine learning classification algorithms, that should be applied to the input data.

Listing 5.1: Experiment Configuration (SMALL1)

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <config xmlns="experimentconfig" xmlns:xsi="http://www.w3.org
     /2001/XMLSchema-instance" xsi:schemaLocation="experimentconfig␣
     experimentconfig.xsd">
3 <loader name="CSVFolderLoader" datalocation="../fixed/data"
     relative="false"/>
4 <versionfilter name="MinClassNumberFilter" param="5" />
5 <setwisepreprocessor name="Normalization" param=""/>
6 <setwisepreprocessor name="AverageStandardization" param="" />
7 <setwisepostprocessor name="Undersampling" param="" />
8 <postprocessor name="Undersampling" param="" />
9 <trainer name="WekaTraining" param="RandomForest␣weka.classifiers.
     trees.RandomForest␣-CVPARAM␣I␣5␣25␣5" />
```

```
<trainer name="WekaTraining" param="C4.5-DTree weka.classifiers.
    trees.J48 -CVPARAM C 0.1 0.3 5" />
<trainer name="WekaTraining" param="Logistic weka.classifiers.
    functions.Logistic" />
<trainer name="WekaTraining" param="NeuralNetwork weka.classifiers
    .functions.MultilayerPerceptron -CVPARAM M 1.5 2.5 3.0 L 0.2
    0.4 3.0" />
<resultspath path="../out/results"/>
<eval name="NormalWekaEvaluation" param="" />
</config>
```

The computational complexity of these trainers is extremely heterogeneous, ranging from less than a minute to hours. For example, the logistic regression algorithm finishes a lot faster than the neural network given the parameters from the experiment. For the purpose of our evaluation, we define three experiments: *SMALL*, *MEDIUM* and *LARGE*. The first, as already shown in 5.1 consists of four trainers with varying run times. *MEDIUM* doubles the trainers, *LARGE* quadruples them.
For the sake of comparing performance in different scenarios, we define the following cases:

1. 2x *SMALL*
2. 4x *SMALL*
3. 8x *SMALL*

The rationale for the first case is to measure the overhead induced by our approach. If the overhead was zero, the execution time on one machine with two times the *SMALL* experiment as input should be as fast as the parallelization to two nodes. This may be counterintuitive at first, because the execution on two nodes should ideally be twice as fast. However, as each map task receives just one input file, it cannot make use of the thread-level parallelism in CrossPare and thus each node takes approximately the same time for the processing of two input files as for one. We will discuss this shortcoming in the conclusion.

The rationale for the second and third case are to measure the speedup for parallelizing to four or eight nodes. Ideally, the execution time should have between 2x *SMALL* and 4x *SMALL* and again between 4x and 8x. The other two experiment files, *MEDIUM* and *LARGE* are not part of our evaluation in this section. Instead, they will play a role in the comparison with a native MapReduce implementation in Chapter 6.

5.3.3. Results

We execute each case parallelized via our approach and non-parallelized on just one machine. For consistent results, we chose one virtual machine of the cluster as the reference setup

for local execution to ensure that the computational capabilities are comparable in both scenarios.
Also, since we have no control over the Cloud system as a whole, we can not be sure if and how many other virtual machines of other customers are operating on the same physical servers at the same time. This can influence our results if the workload the other customers induce on the hardware differs between the runs. To account for this noise, we execute each scenario two times and take the average.

Scenario	*Non-Distributed Ø*	*Distributed Ø*	Speedup
2x SMALL	16:00h	12:01h	1.33
4x SMALL	24:10h	11:58h	2.01
8x SMALL	52:32h	23:44h	2.21

Table 5.3.: CrossPare Results

Contradicting the expectations from our discussion above, even the first case exhibits a speedup greater than one (1.33), obviously outweighing the overhead introduced by Hadoop. Since the individual runtimes are consistent with this average-based observation, the most likely explanation appears to be that the per-experiment processing time is noticeably lower for a single input than for two inputs despite the thread-level parallelism.
The second case, i.e. executing the *SMALL* experiment four times, leads to an speedup factor of 2.01 which is optimal given our expectations. Again, the optimal speedup in this context is just half the theoretically optimal speedup of 4.0 due to the fact that CrossPare is able to utilize two cores in the reference case.
The speedup in the last case is approximately equal to the former case and thus only a quarter of the optimal speedup. While in the reference case the eight input files are processed in chunks of two concurrently on two cores, our approach processes seven input files in parallel which takes roughly half the total execution time of the reference case. However, because one slave node runs the *ApplicationMaster*, it is not available to run the actual user application. Hence, the task for the eighth file can only then be started when another map task has finished which leads to a doubled runtime. In other words, had we chosen 7x *SMALL* instead, the speedup would have been substantially larger, because all input files could have been processed concurrently. This also demonstrates that the optimal cluster size depends on the respective use case.

5.4. TaskTreeCrossvalidation

5.4.1. Application Description

Our second case study, for the lack of an official name here referred to as `TaskTreeCrossvalidation`, is an application written by a researcher of the Software Engineering for Distributed Systems group in the context of his PhD thesis, Patrick Harms. Again, we will briefly introduce the background of the application before discussing the adaptations needed to be compatible with our approach.
Harms research is partially concerned with the usability analysis of websites. Part of this is the use of so-called task trees to formally describe user interactions in a tree structure where parent nodes define the order of actions and leaf nodes the actual actions performed by the user, called tasks (e.g. clicks, text input etc.). The rationale for the trees is to analyse them later on to detect potential usability issues [57] [56]. The recording of the user actions is performed using the `AutoQUEST` tool suite [6].
The application for our case study is an intermediate tool that verifies whether the trees the approach generates are similar trees for different recordings of the same website, because otherwise they would not be considered suitable for the usage analysis.
The changes needed to the application for the use with our approach were small, especially in terms of the application structure. Basically, there is only one folder structured as follows:

- *autoquest-distribution*: Contains the AutoQUEST tool suite with the verification code.
- *run.sh*: A wrapper script for executing AutoQUEST.
- *crossvalidation.data*: The recording of the user interactions.
- *Script files*: Configuration files that instruct *run.sh.* Equivalent to the experiment configuration files of CrossPare.

The structure could remain the same for execution with our framework, though moving the experiment configurations to the *transient* directory is recommended for reasons explained earlier. However, because no changes to the files were planned for the course of the case study and to demonstrate that our approach can be used with minimal overhead, we left the structure as is.
Before planning of the case study, *run.sh* relied heavily on command line switches for the run configurations. Since our approach assumes the same parameters for all runs, this poses a problem because all run-specific informations need to be passed via the *input* file. To account for this, TaskTreeCrossvalidation has been modified as to receive its configuration via one single file, the script file.

5.4.2. Evaluation Setup

The runtime of the application is determined by the complexity of the experiment defined in the input file. More specifically, depending on the experiment configuration, the application needs to perform more or less task tree comparisons on the data set. The size of the experiment is expressed as the percentage of the data set which is used to generate the task trees for comparison, e.g. *crossvalidation_10_1_script* (Listing 5.2) defines an experiment where task trees are generated and compared for 10% subsets of the data.

Listing 5.2: TaskTreeCrossvalidation Configuration Example

```
load ../crossvalidation.data
generateTaskTree sequences_subset_10_1 tasktree_subset_10_1
generateTaskTree sequences_subset_10_2 tasktree_subset_10_2
getTaskModelSimilarity tasktree_subset_10_1 tasktree_subset_10_2
save ../../out/crossvalidation_10_1-results.data
exit
```

The larger the subsets, the more comparisons are needed and the longer the execution time. Subset sizes in the distribution provided to us range from 1% to 20%. Furthermore, in addition to the *crossvalidation_** experiment configurations, there are *crossvalidation_fullrecall_** files that compares the subsets with the full dataset, leading to a substantially longer runtime. For our case study, we define two scenarios for evaluating our approach's performance:

1. The heterogeneous case: Consists of each two times 1%, 2.5%, 5% and 10% size experiment configurations, i.e. 8 input files in total.

2. The homogeneous case: Consists of 6 input files in total, but this time only 10% experiment configurations.

The rationale for the two cases is to evaluate whether there is a difference in the speedup for heterogeneous and homogeneous experiments. This contrasts our first case study where instead of evaluating differently structured inputs, we examined the use of a varying number of input files, i.e. a different number of parallel processes.

5.4.3. Results

We perform the evaluation in the same fashion as in our first case study. To measure the reference (i.e. non-parallelized) performance, we execute the application on one machine of our cluster and repeat both the reference and parallel case to mitigate the effects of volatile Cloud performance. Table 5.4 summarizes the results.

Scenario	*Non-Distributed Ø*	*Distributed Ø*	*Speedup*
Heterogeneous Input	4:24h	2:02h	2.16
Homogeneous Input	9:57h	2:10h	4.57

Table 5.4.: TaskTreeCrossvalidation Results

As can be seen from the table, our approach processes both scenarios faster than the non-distributed execution. However, the speedup is twice as large for the homogeneous case compared to the heterogeneous case. The explanation for this circumstance is simple: In both cases, six worker nodes are each assigned one of the input files. While in the heterogeneous case the smaller experiments finish after just some minutes and the respective nodes are then in an idle state, in the homogeneous case the nodes are all utilized for approximately the same time, leading to a higher average utilization and thus to a higher speedup.
The situation for the heterogeneous case would improve with an increased problem size. Assuming that each input file exists e.g. ten times, the average utilization of each worker node would also increase, because once the smaller input configurations have been processed, there were still larger ones in the queue.

6. Comparison with MapReduce

In the approach we presented in the previous chapters, we have utilized Hadoop Streaming as a wrapper to MapReduce for executing arbitrary applications. The results of our case studies were promising with regard to the possible runtime reductions involved in using our approach. In this chapter, we are going to compare these results with that of a native MapReduce application.
Section 6.1 will present general considerations needed for transforming an application into a MapReduce application. It is important to note that while we try to keep them as general and high-level as possible, they will not apply to any potential application. Section 6.2, following this guideline, shows the steps involved in porting the CrossPare application. Finally, in Section 6.3, we compare its performance with both the non-distributed execution and with the execution via our Streaming-based approach.

6.1. General Considerations

In this section we attempt to generalize the procedure of parallelizing an existing application using MapReduce in the form of a few general steps that should be executed in the respective order. Naturally, applications vary a lot in terms of many dimensions such as their intended purpose, program structure, employed libraries et cetera and not all applications will be compatible with the schema presented. To start with the most basic level, the software will need to be implemented in Java to use Hadoop's API.
Following, we will describe the 6 steps we identified on a high-level basis prior to showing their implementation in CrossPare, one of our case study applications.

1. Identify Parallelizable Tasks The perhaps most critical step is the identification of parallelizable tasks in the software. This is not only of concern for our context, but for application parallelization in general. Naturally, strictly sequential (atomic) operations cannot be distributed across different nodes because each of them depends on the result of the operations preceding it. Therefore, the developer needs to identify self-contained tasks whose execution process is independent of others.
Also, as a second criterion, the identified tasks should be complex enough to qualify for parallelization. Complex enough here means (1) that the runtime of the tasks accounts for a non-negligible share of the overall runtime of the application and (2) that each task's

runtime justifies the overhead involved in distributing them. The rationale for the first requirement is that it only when it is fulfilled, there is a possibility to substantially reduce the application's runtime. The second requirement arises from the fact that parallelization itself comes with an overhead (most importantly communication), that, if it is too large compared to the gains of distributed processing, leads to an increased overall runtime instead of a reduction. The Hadoop documentation recommends a minimum *map* task time of one minute [26].

2. Identify Needed Objects Technically, MapReduce tasks run in their own JVM and have no access to the object space of the user application. This has severe implications for the parallelization of tasks because the developer needs to think of the *map* tasks as independent programs that need to receive input from the calling application in a substantially different way than e.g. simply calling a Java method.
We call this this input ”objects”, because the Java objects the mapper class needs are the fundamental entity: While static input data to the application such as experiment data can be read by the mapper class directly from the shared file system (HDFS) and parameters of certain types (e.g. *String*s) can be passed via the Hadoop API, objects of other types of the user application that are needed for performing the tasks to be parallelized cannot be accessed directly, thus establishing the need to identify and transfer these explicitly.
Which parts of the program these are exactly can hardly be generalized in a satisfactory way. Our example in the next section will shed more light on this step.

3. Serialize Objects We noted in the previous step that the *map* tasks share access to Hadoop's filesystem. Consequently, it is also predestined to provide the storage for the previously identified objects to the corresponding map tasks. The de-facto standard to share objects between processes is to serialize them, i.e. converting their state to a byte stream adhering to a certain specification which can be transformed back into the same object [29]. By serializing the identified objects and storing them in files in the HDFS, the *map* tasks gain access to their required input.

4. Implement Mapper Class As explained in the overview on the MapReduce paradigm in Section 2.4.1, at the very least a *map* function is required for each MapReduce application. In Hadoop's API, this function is embedded in classes that implement the *Mapper* interface. In this context, the mapper class is responsible for the processing of the parallelized task and thus contains all code related to its execution.
Its schema can be described as (1) Deserialize (2) Process (3) Serialize: In step (1), the map function deserializes the previously serialized objects to use them in (2) for processing the task and (3) serializing the result for later use in the user application.

5. Prepare MapReduce task Once the mapper class is ready and the data it needs available, the actual MapReduce task can be launched. This is done using the Hadoop API and occurs at the point in the program flow where the parallelization replaces the original non-distributed processing.

6. Deserialize Processed Data After all tasks have finished, the control flow returns to the user application. Now, the objects serialized by the *map* tasks can be deserialized and used for further non-distributed processing in the user application.

6.2. CrossPare MapReduce port

Having introduced a guideline for introducing MapReduce parallelism into existing applications, we are now going to apply it to one of our case study applications, namely CrossPare, and describe the steps in the same order as above. For a better understanding, we will first very roughly sketch CrossPare's program flow:

1. For each experiment configuration passed via the command line, the *main()* function calls a *createConfig()* function that parses the file and creates an object of type *Experiment* from it.
    - As shown in Listing 5.1, an experiment configuration consists of different components, in the given example: *setwisepreprocessor*, *postprocessor*, *trainer*. Other experiment configurations also include *setwiseselector* or *setwisetrainer*. We don't want to concern us with the exact purpose these components serve. What is relevant for our context is that for each experiment configuration, each of the items belonging to the same category are executed together in a loop.
2. In the *run()* method of *Experiment*, the software metrics data is split up into a test and a training set as is usually done for model validation in statistical analysis.
3. Following, the different components are applied to the data. The components are represented by corresponding Java classes from the respective packages created using introspection (e.g. the *setwisepreprocessor* named *Normalization* is represented by *de.ugoe.cs.cpdp.dataprocessing.Normalization*). The outcome of this step are trained models, in our example e.g. a decision tree model.
4. Finally, an evaluator is applied to the models, generating the resulting output in the corresponding CSV file.

1. Identify Parallelizable Tasks Recalling the requirements for the identification of parallelizable tasks we laid out above, the first criterion is to find operations that can potentially be distributed. We find these in the *run()* operation of the *Experiment* class that consists of eight loops similar to the one in Listing 6.1.

Listing 6.1: Exemplary Loop from run() Method

```
for( ITrainingStrategy trainer : config.getTrainers() ) {
        Console.traceln(Level.FINE, String.format("[%s]␣[%02d/%02d
            ]␣%s:␣applying␣trainer␣%s",             config.
            getExperimentName(), versionCount, testVersionCount,
            testVersion.getVersion(), trainer.getName()));
        trainer.apply(traindata);
}
```

In principle, all these loops can be distributed by having each *map* task apply *1/n'th* – where n is the number of nodes – of the processors to the data.
We can however filter them by the second criterion, i.e. whether the tasks are complex enough to justify their distributed processing. We find that the runtimes vary substantially within and between the different components, but that on average only the *trainer*s have a runtime that appears to satisfy our criterion. Noteworthy, we say "appears", because despite the general recommendation that *map* tasks should last at least one minute, only a case-specific analysis can judge optimality. Our measures however show that the *trainer*s are clearly the most complex task, account for the majority of the overall runtime and thus should be parallelized first regardless of the other components. In the following, we will thus detail the parallelization of the *trainer*s, but the approach is representative.

2. Identify Needed Objects From the description of the program flow above we know that the trainers need the *traindata* as input. *traindata* is of Weka type *Instances* which is a collection class for storing objects of types that implement the *Instance* interface which represents an instance of the experiment data. Consequently, since we aim to parallelize the execution of the trainers, their input data needs to be made available.
Obviously, if the trainers should be distributed, they are themselves also objects that need to be serialized. This however poses a problem due to the different serialization interfaces of `Java` and `Hadoop`. We will cover this in the next paragraph.

3. Serialize Objects Specifically, in our case the concern is to convert the trainers into *Writable* objects to make them Hadoop-compliant (see box below).

Java vs. Hadoop Serialization

Java and Hadoop employ different serialization implementations which differ both in their underlying storage and in their user-visible interface. CrossPare contains multiple trainer classes for different purposes which all implement the *ITrainingStrategy* interface and extend the *WekaBaseTraining* class, but which are also differ in their suitability for being serialized as *Writable*s.
In Java, classes need to implement the *java.io.Serializable* [30] interface. Given this and that all of its members themselves are serializable or marked as being transient, this suffices to serialize objects of the class using an *ObjectOutputStream*. In the case of Hadoop, classes that should be serializable implement the *org.apache.hadoop.io.Writable* [21] interface. Noteworthy, all objects that appear as input or output in *map* or *reduce* operations need to be serializable. While in Java serialization no further methods need to be implemented, the Hadoop interface expects the developer to add the *void write(DataOutput out)* and *void readFields(DataInput in)* to the class. The latter's purpose is to serialize the fields of the object to the given *DataOutput* stream, while the first is the reverse operation, i.e. it deserializes the fields of the object. A crucial limitation of *DataOutput* is that its write operations are limited to Java's primitive types. This implies that user-defined classes cannot usually be serialized without effort. The two operations are typically complemented by a third static method *read(DataInput in)* that instantiates an object of the class and calls *readFields()* on it to populate the object.
This relates to the internal differences in both approaches. Java does not assume that the class of the stored objects is known, Hadoop on the other hand does assume this because the application creates the instance of the class before calling *readFields()*. Therefore, because the type of the object does not have to be stored in the byte stream, Hadoop serialization is less input/output-intensive. Since MapReduce is very input/output-intensive due to the fact that objects regularly need to be transferred across the network, reducing the required communication is paramount. [19]
The differences outlined here have the important implication that both serialization approaches are not interchangeable, therefore not allowing plain *Serializable* classes to be used in the context of Hadoop operations.

This is due to the way they are implemented: Some of the classes contain inner classes and multiple class members, making it hard to serialize them in this scheme. More concretely, part of the problem is that because Hadoop's *DataOutput* supports only primitive types, complex types need to be converted into one of these types first, e.g. a byte array. This, however, requires a prior serialization of the object using Java's serialization which again is only practicable for classes supporting it, which does not apply to all of those used in the

trainer classes.
Hence, to avoid making substantial changes to CrossPare, we have to limit ourselves to the *WekaTraining* class that does not add any additional members to its base class (*WekaBaseTraining*) and thus only contains the members as depicted in Figure 6.1. Though this trainer does not cover all experiments in use with CrossPare, it covers a great part of them and thus a speedup-involving distribution of them would contribute to an overall runtime improvement.

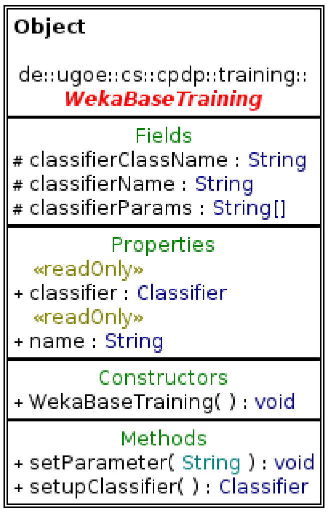

Object
de::ugoe::cs::cpdp::training::
WekaBaseTraining
Fields
# classifierClassName : String
# classifierName : String
# classifierParams : String[]
Properties
«readOnly»
+ classifier : Classifier
«readOnly»
+ name : String
Constructors
+ WekaBaseTraining( ) : void
Methods
+ setParameter(String) : void
+ setupClassifier( ) : Classifier

Figure 6.1.: WekaBaseTraining Class Diagram

We split up our discussion of the serialization into two parts, the training data and the actual trainers. Finally, we will show how to make the serialized data available to the MapReduce portion of the program.

Training data

We recall that *traindata* is of type *weka.core.Instances* which is a Java *Serializable* because Weka has been designed to support serialization in its core types. Since the training data is relevant to all of our *map* tasks instead of just a specific one as in the case of the trainers, we aim to make it available globally. Hadoop supports this via its concept of the *Configuration* object used for setting up a job. This object can be assigned values via its *set(String name, String value)* operation. In our case, we can utilize this concept in the following way:

1. Convert the instances to a *ByteArrayOutputStream* via an *ObjectOutputStream*.
2. Convert this byte array to a Base64 String using the `Apache Commons` library [4].

3. Set this string as the value for the configuration option ”traindata”.

Now, this serialized version of the training data is available to all *map* tasks. We will see how the mapper class accesses it in the next paragraph.

Trainers
The second class of objects to serialize are the trainers. Listing 6.2 shows the corresponding code.

Listing 6.2: Serialization of Trainers

```
1  ArrayList<String> inputFiles = new ArrayList<String>();
2  for(ITrainingStrategy i : config.getTrainers()) {
3          WekaTraining trainer = (WekaTraining) i;
4          try {
5                  String filename = Long.toString(System.nanoTime())
                       + ".ser";
6                  Path serializationFile = new Path(
                       serializationTrainerInputDir, filename);
7                  OutputStream os = fs.create(serializationFile);
8                  os.write(SerializationHelper.serialize(trainer));
9                  os.close();
10                 inputFiles.add(serializationFile.toString());
11         } catch (Exception e) {
12                         System.out.println("Exception␣serializing␣
                               to␣disk:␣" + e);
13         }
14 }
```

We iterate over the trainers and cast them to *WekaTraining* by our assumption that we only use trainers of that type in this context. Our aim is to serialize the trainers to HDFS-backed files that can be read by the *map* tasks. The order is arbitrary, so a random filename (line 5) suffices, for which the corresponding HDFS path is setup (line 6) and *OutputStream* opened (line 7). The trainer then gets serialized and the resulting binary stream is written to the file (line 8). Finally, the file is added to the list of filenames that should be processed. The *SerializationHelper* class called in line 8 is simply a helper class for wrapping the serialization to and deserialization from *Writable*s (see Appendix B.5). For brevity, we don't show the corresponding *write()* and *readFields()* of the *WekaTraining* class here (see Appendix B.6 for the complete code).

At this point, all needed objects are serialized, notably in different fashion: While the training data is available to the Hadoop tasks via its *Configuration* object, the trainers have been serialized to disk. Conceptually, we could have serialized the former to disk, too, but not the other way around. However, since there is an overhead involved in serializing to disk, our approach is preferable and more straightforward to implement.

4. Implement Mapper Class In our context, the task of the mapper class (see Appendix B.7 for the complete class code) is to apply a trainer to the training data and transfer back the result of this operation to the program. Like in all Hadoop applications, our class implements the *map()* method that is called by the framework. Additionally, we implement the *setup()* method which is called once prior to the execution to the actual map operations.

Listing 6.3: setup() Method of TrainerMapper

```
 1  protected void setup(Context context) {
 2          conf = context.getConfiguration();
 3          serializationTrainerOutputDir = new Path(conf.get("
                outputpath"));
 4          String serializedTraindata = conf.get("traindata");
 5          try {
 6          byte b[] = Base64.decodeBase64(serializedTraindata);
 7          ByteArrayInputStream bi = new ByteArrayInputStream(b);
 8          ObjectInputStream si = new ObjectInputStream(bi);
 9          traindata = (Instances) si.readObject();
10      } catch( Exception e ) {
11          System.out.println(e);
12      }
13  }
```

In *setup()* (Listing 6.3), we retrieve two parameters from the *Configuration* object of the job. The first is the path where the processed trainers should be stored. We will discuss this point in the next paragraph. The other parameter is the serialized training data. Line 4 reads this data as a Base64-formatted *String* and deserializes it back with the help of Apache Commons (lines 5-8). The resulting *Instances* object is assigned to a *static* variable of the class, thereby making it available to all map tasks.

Listing 6.4: map() Method of TrainerMapper

```
public void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {
    // Read serialized trainer from HDFS
    Path inputFile = new Path(value.toString());
    FileSystem fs = FileSystem.get(conf);
    FSDataInputStream in = fs.open(inputFile);
    final int available = in.available();
    byte serializedTrainer[] = new byte[available];
    in.readFully(0, serializedTrainer);

    // Deserialize and apply trainer
    WekaTraining trainer = SerializationHelper.asWritable(
        serializedTrainer, WekaTraining.class);
    trainer.apply(traindata);

    // Serialize trainer and write to HDFS
    String filename = Long.toString(System.nanoTime()) + ".ser";
    Path serializationFile = new Path(
        serializationTrainerOutputDir, filename);
    OutputStream os = fs.create(serializationFile);
    os.write(SerializationHelper.serialize(trainer));
    os.close();
}
```

The concept of the *map()* method is to read a previously serialized trainer from the HDFS, deserialize it, execute it using its *apply()* method and serialize it back. Notably, the *apply()* method is the time-intensive task that trains the model. Which trainer the particular map task reads depends on the contents of the *value* parameter, which is fed in by the Hadoop framework. The *SerializationHelper* again assists in the serialization/deserialization process. Processed trainers are saved in the directory specified by the static *serializationTrainerOutputDir* with a random file name. By this procedure, the non-distributed part of the application can simply collect the results of the distributed processing as we will see below.

5. Prepare MapReduce Job Having the serialized data and the mapper class allows the setup of a MapReduce job. This setup is performed in the class in which we also serialized the needed objects and which usually executes the trainers non-distributedly, namely *Experiment*. Listing 6.5 shows our code for the job setup.

Listing 6.5: MapReduce Job Setup

```
Configuration conf = new Configuration();
(... trainer serialization ...)

// Create input index file (using NLineInputFormat)
Path indexFilePath = new Path(serializationSequenceInputDir, "
    input.idx");
```

```
 6 OutputStream os = fs.create(indexFilePath);
 7 for(String filename : inputFiles) {
 8         filename += "\n";
 9         os.write(filename.getBytes());
10 }
11 os.close();
12 
13 // Job execution
14 conf.set("outputpath", serializationTrainerOutputDir.toString());
15 Job job = Job.getInstance(conf, "Weka␣Trainers");
16 job.setJarByClass(TrainerMapper.class);
17 job.setMapperClass(TrainerMapper.class);
18 job.setInputFormatClass(NLineInputFormat.class);
19 NLineInputFormat.addInputPath(job, serializationSequenceInputDir);
20 NLineInputFormat.setNumLinesPerSplit(job, 1);
21 FileOutputFormat.setOutputPath(job, serializationOutputDir);
22 job.waitForCompletion(true);
```

Line 1 creates the aforementioned instance of the *Configuration* class. Following, in lines 4-10 the input file for the Hadoop job is created by writing the filenames of the serialized trainers to it separated by newlines. We will not elaborate on the concept of input formats here other than to mention that Hadoop supports different input specifications of which the *NLineInputFormat* we employ is one. This format creates input splits – which can be thought of as the portion of input data a map task receives – from n lines of the specified input file, where n is set to 1 in our case (line 19) [20]. That way, each map task receives exactly one filename as its *value*, thus giving it the information necessary to complete the processing of one trainer.
The actual job creation occurs in lines 15-21. Using the *Configuration* object, we can create a job instance (line 15) and set its mapper class (lines 16-17). The job's input format is then set to the mentioned *NLineInputFormat* and its input and output path configured. As the names indicate, these paths inform the input format about where it should read the input from and write its output to.
Finally, line 22 launches the job and blocks the program flow until it is finished.

6. Deserialize Processed Data The outcome of the successfully executed MapReduce job is the set of processed trainer objects that the mapper class has serialized to the HDFS storage. The final step in our scheme is to integrate the processed data back into the non-parallel part of the application. This is performed right after the job execution in the *Experiment* class. Listing 6.6 shows the necessary code.

Listing 6.6: Trainer Deserialization

```java
RemoteIterator<LocatedFileStatus> iter = fs.listFiles(
   serializationTrainerOutputDir, false);
        while(iter.hasNext()) {
        LocatedFileStatus stat = iter.next();
        Path inputFile = stat.getPath();
        FSDataInputStream in = fs.open(inputFile);
        final int available = in.available();
        byte serializedTrainer[] = new byte[available];
        in.readFully(0, serializedTrainer);
        WekaTraining trainer = SerializationHelper.asWritable(
           serializedTrainer, WekaTraining.class);
        processedTrainers.add(trainer);
}
```

We iterate over the filenames in the HDFS output directory, read them and deserialize the contents back to *WekaTraining* objects. The deserialized trainers are added to the container *processedTrainers*. This container then replaces the previously unprocessed trainers for the remainder of the program which only applies evaluators to the trainers to generate output files for the user. Since this evaluation step is computationally inexpensive, the program comes to a halt shortly after its MapReduce job.

6.3. Evaluation

In this section we evaluate the performance of the MapReduce'd version of CrossPare, both against the local execution of the unmodified program and against the distributed execution using the Hadoop Streaming approach we developed. Again, for the same reasons described earlier, the *speedup* is our only criterion for the evaluation.
Also, we use the same experiment configurations as defined in Section 5.3.2, but unlike in the earlier evaluation of our approach, we this time also utilize the *MEDIUM* and *LARGE* configurations. The rationale for this is that while the parallelization of our approach is **file-based**, i.e. the elementary unit is an experiment configuration, the MapReduce version is **trainer-based**, i.e. the elementary unit is a trainer. Therefore, it can only launch as many tasks as there are trainers if multiple parallel MapReduce jobs (i.e. one per input configuration) are not desired. Because the *SMALL* configurations contains just four trainers, it cannot fully utilize our test cluster with just one experiment configuration.
Again, as in our evaluation in Section 5.3.3, it is important to note that there may be variations in runtime caused by differing workload on the hardware underlying our virtual machines smoothed out by taking the average.

Streaming vs. MapReduce A balanced comparison between our Streaming-based approach and the MapReduce variant demands the use of differently structured input experiment configurations for the two. As mentioned before, the former requires multiple inputs for parallel processing while the latter solely takes into account the number of trainers. Hence, a scenario with only one input file cannot lead to any speedup with Streaming and thus multiple input configurations are required. On the other hand, the MapReduce variant creates one job per input file, which is an unnecessary overhead if the same experiment can be represented with a single configuration. Exactly this is the case with our configurations. For example, configuration *LARGE* simply contains the trainers of *SMALL*, but repeated four times. Thus, using *4x SMALL* or *1x LARGE* as input is functionally equivalent aside from the fact that in the first case the model output will be split across four output files and aggregated into one in the latter. We therefore use the most efficient combination of inputs for our comparison to rule out the possibility that runtime differences are caused by suboptimal input choice.
Table 6.1 summarizes our comparison where the left side of the first column represents the input for the streaming approach and the right side the input for the MapReduce version.

Scenario (Input 1 | Input 2)	*Streaming Ø*	*MapReduce Ø*	*Speedup*
2x SMALL | 1x MEDIUM	12:00h	12:31h	0.96
4x SMALL | 1x LARGE	11:58h	12:59h	0.92
8x SMALL | 1x XLARGE	23:44h	25:29h	0.93

Table 6.1.: Streaming vs. MapReduce

As can be seen in the table, in no scenario the MapReduce version outperforms our approach. This can be explained with the fact that the runtime of the MapReduce version is determined by the longest-running trainer: Even if most trainers finish faster than in the given runtime and there are free nodes available for further processing, the whole process cannot exit before the last trainer is finished. The same applies to the Streaming-based approach, however without the overhead induced by our transformation to a MapReduce problem which appears to slow down the computation linearly by problem size, however not by much.

Local vs. MapReduce Similar considerations regarding the input files for the evaluation apply to our second comparison. In our test environment, each node has two cores and these are both utilized by the unmodified CrossPare application due to its thread-based parallelism, given that there are at least two input files. However, unlike in the case of the MapReduce version, there is only a negligible difference between using e.g. *8x SMALL* or *2x LARGE* as the input for local execution, because more inputs in this environment do not lead to a higher degree of parallelization. We therefore employ the same scenarios. Table 6.2 shows the results.

Scenario (Input 1 | Input 2)	*Local Ø*	*MapReduce Ø*	*Speedup*
2x SMALL | 1x MEDIUM	16:00h	12:31h	1.28
4x SMALL | 1x LARGE	24:10h	12:59h	1.86
8x SMALL | 1x XLARGE	52:32h	25:29h	2.06

Table 6.2.: Local vs. MapReduce

The results show that the MapReduce version is superior compared to local execution and that the benefits of MapReduce are greater for larger inputs. This conforms to our expectations, because the runtime for the local execution is expected to grow proportionally to the input size, while at this scale the growth for MapReduce is less than proportional.

7. Related Work

To the best of our knowledge, there is no work similar to the approach presented in this thesis so far in the narrower sense. We here define the narrower sense as the combination of both software for the automated deployment of a Hadoop computation cluster in a Cloud environment and helper tools for executing a broad variety of software on it in a parallel fashion with negligible effort on the user side.
Therefore, we will give an overview over related work in this chapter that addresses certain similar aspects of our approach. First, in Section 7.1 we will present cloud orchestration frameworks for the deployment of applications in the cloud. Following, in Section 7.2 we refer to solutions for the automated setup of Hadoop clusters.

7.1. Application Deployment

`cloudinit.d` [11], part of the Nimbus project [35], is ”a tool designed for launching, controlling, and monitoring complex environments in the cloud” [11] that is compatible with different Cloud providers and IaaS stacks. Users define so-called *launch plans* that instruct cloudinit.d which virtual machine images and types should be launched in a specified order, thereby seeking to build a Cloud equivalent to the `init.d`-style systems known from the Linux world. Further, it offers the user to specify programs that should be executed upon booting and termination of the machines, which allows to perform setup, teardown and other operations. cloudinit.d aims to provide monitoring and automatic error correction of the deployed system based on the hierarchization of runlevels. By the usage of specially crafted VM images and respective setup scripts, cloudinit.d can be employed to deploy arbitrary software.
`Neptune` [53] is a Ruby-based domain-specific language ”that automates configuration and deployment of existing HPC software via cloud computing platforms” [53]. It operates at the cloud platform layer to control both the system and application level. For the job management side it utilizes `AppScale` [5] which is an open source Cloud computing platform for running `Google App Engine` [16] applications. Being targeted towards HPC software, Neptune provides some modules for its DSL for popular scientifically used frameworks such as MPI. We were not able to test Neptune because its latest release seems incompatible with current versions of `AppScale` and its development appears to have been discontinued with the latest change having been committed in 2012.

7.2. Simplified Hadoop Deployment and Management

There is a wide variety of possibilities to deploy Hadoop clusters in Cloud environments. At the extreme in terms of manual work involved, there are commercial offerings for ready-to-use clusters such as Amazon's *Elastic MapReduce (EMR)* [1] or Google's Compute Engine with a click-to-deploy offering [24] for Hadoop. Naturally, these can only be used on the respective provider's Cloud platform.
For the deployment on provider-independent platforms there exist a number of software packages. A prominent commercial one is *Cloudera Enterprise* [9] by CLOUDERA [8]. The vendor also provides a free version called `Cloudera Express` that also includes its Hadoop distribution and its proprietary management tool, but comes with less functionality. Both aim to ease the deployment and management tasks as well as extending `Hadoop` further with enterprise features.
On the non-commercial side, Apache's `Ambari` ”is aimed at making Hadoop management simpler by developing software for provisioning, managing, and monitoring Apache Hadoop clusters” [3]. For providing the cluster monitoring it leverages the established projects `Ganglia` [15] and `Nagios` [34].
While Ambari makes no assumptions about the underlying infrastructure, the OpenStack project `Sahara` provides means ”to provision a data-intensive application cluster (Hadoop or Spark) on top of OpenStack” [38]. Hence, a running OpenStack installation is a prerequisite for utilizing Sahara. Given that, users can describe their desired cluster in the form of templates and have the deployment handled by Sahara.

8. Conclusion and Outlook

We have developed a Hadoop-based approach for automated deployment and distributed execution of applications in Cloud environments with minimal effort on the user side. An important aspect of our work was to impose as few requirements as possible onto the set of potential applications that can be executed using our framework to design it as generic as possible. Besides Hadoop, we employed tools for the provisioning and configuration of virtual computing resources.

To evaluate our approach, we performed two case studies using applications developed by researchers of the Software Engineering for Distributed Systems group of this university. Our interest besides investigating the effort needed to deploy and execute applications using our solution was to measure the speedup gained in comparison to a non-distributed execution of the application, i.e. how much time the users can save from using the Cloud-based approach in relation to running the application on their local working machine. In both cases, our results showed that our approach is favourable. First, the applications could be adapted easily to fit our modest requirements. Second, under most scenarios we observed a speedup which was varying in size depending on the structure of the particular scenario.

Since our approach is based on Hadoop Streaming, which is a wrapper to execute arbitrary applications using the MapReduce framework, a logical consequence was to compare our approach's performance with that of a native MapReduce application to be able to draw qualified conclusions about either option's advantages and disadvantages. Again, the speedup was our primary criterion of interest. Moreover, we also documented the effort needed to modify an existing application to employ MapReduce for its compute-intensive tasks. Naturally, due to the heterogeneity of software, it is only possible to examine these questions on a per-case basis. For the context of this thesis we chose one of our case study applications as the object of investigation.

Again, the results were in favour of our approach. Though the MapReduce version performed better than the reference situation,i.e. the non-distributed execution, and thus is an improvement in itself, our approach still outperformed it, though not by much. Also, the development work the user needs to invest is by far larger than to adapt the application to our requirements.

Summarizing, the approach developed in the context of this thesis is an applicable solution for automatically deploying and executing applications in Cloud environments with the goal of reducing the overall execution time and without the user having to learn about Cloud Computing internals.

8.1. Future Research Directions

Currently, there is a fixed one-file-one-task-mapping implying that each node processes exactly one file at a time. As we have seen in one of our case studies, it might be beneficial to give the user the possibility to influence this mapping while retaining the simplicity of the configuration. A basic solution may introduce an additional parameter in the job description for specifying the files-per-task-mapping.

Further, the cluster size is currently set by the user prior to any job execution and then remains constant. However, depending on the particular job it may be desirable to add or remove compute resources to either shorten the execution time or to save costs. While our deployment software is already prepared for scaling of the cluster, the required scaling logic is a topic for future research.
Here, both a user-configurable and a machine learning-based approach is conceivable.

Throughout this thesis we used an OpenStack Cloud and our software is currently hard-coded to work with this IaaS platform. For use with other systems a Cloud-agnostic implementation would be necessary. Supporting other providers would only require modest changes to our approach as long as there is a Vagrant plugin available for the desired provider.

Finally, we did not consider the issue of keeping the software on the cluster up-to-date or security issues in general. Rather, we treat the clusters as just short-lived compute resources that are created and destroyed on-demand. For real-world application where security considerations are warranted our approach needs to be extended in this direction.

A. Abbreviations and Acronyms

API	Application Programming Interface
ASF	Apache Software Foundation
AWS	Amazon Web Services
DSL	Domain-Specific Language
HDFS	Hadoop Distributed File System
EMR	Elastic MapReduce
GWDG	Gesellschaft für wissenschaftliche Datenverarbeitung Göttingen
HPC	High Performance Computing
IaaS	Infrastructure as a Service
IOPS	Input/Output Operations Per Second
JSON	JavaScript Object Notation
LOC	Lines of Code
MPI	Message Passing Interface
NFS	Network File System
NFV	Network Functions Virtualization
PaaS	Platform as a Service
POSIX	Portable Operating System Interface
RAID	Redundant Array of Independent Disk
SaaS	Software as a Service
SCM	Software Configuration Management

SIMD	Single Instruction Multiple Data
STDIN	Standard Input
STDOUT	Standard Output
VM	Virtual Machine
YARN	Yet Another Resource Negotiator

B. Source Code Excerpts

B.1. Vagrantfile Template

Listing B.1: Vagrantfile Template for Cluster Provisioning

```ruby
# -*- mode: ruby -*-
# vi: set ft=ruby :

# Vagrantfile template for cluster.py tool.
# DO NOT EDIT UNLESS YOU KNOW WHAT YOU ARE DOING.

VAGRANTFILE_API_VERSION = "2"
SLAVES_COUNT = $count

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
    config.vm.box = 'dummy'
    config.ssh.username = 'root'
    config.ssh.private_key_path = '$key_path'
    config.vm.synced_folder '.', '/vagrant'
    config.vm.boot_timeout = 600

    config.vm.provider :openstack do |os|
        os.openstack_auth_url = 'https://api.cloud.gwdg.de:5000/v2
            .0/tokens'
        os.username           = '$username'
        os.password           = '$password'
        os.tenant_name        = '$tenant_name'
        os.flavor             = '$flavor'
        os.image              = '7d9da5d4-2b56-477c-bb03-4
            e731ad59f3e'
        os.floating_ip_pool   = 'public'
        os.keypair_name       = '$key_name'
    end

    config.vm.define "namenode" do |namenode|
        namenode.vm.hostname = "namenode"
    end

    SLAVES_COUNT.times do |i|
        config.vm.define "slave#{i+1}" do |slave|
            slave.vm.hostname = "slave#{i+1}"
        end
    end

    config.vm.define "resourcemanager" do |resourcemanager|
        resourcemanager.vm.hostname = "resourcemanager"

        resourcemanager.vm.provision :ansible do |ansible|
            ansible.verbose = "v"
            ansible.sudo = true
            ansible.playbook = "$MTT_HOME/cluster/deployment/site.
               yml"
            ansible.limit = "all"

            slaves = (1..SLAVES_COUNT).to_a.map {|id| "slave#{id
               }"}
            ansible.groups = {
                "NameNode" => ["namenode"],
                "ResourceManager" => ["resourcemanager"],
                "Slaves" => slaves
            }
        end
    end
end
```

B.2. cluster --project_init Ansible Code

Listing B.2: Ansible Play Code for Project Initialization

```
---
# Playbook for synchronization step in cluster --project_init
- hosts: namenode
  remote_user: root
  tasks:
    - name: Synchronize input data to NameNode
      synchronize: src={{ dir_path }} dest=/home/hadoop delete=yes
          recursive=yes
    - name: Change ownership to Hadoop user
      shell: chown -R hadoop:hadoop /home/hadoop/{{ dir_name }}

- hosts: namenode
  remote_user: root
  sudo: yes
  sudo_user: hadoop
  tasks:
    - name: Import input data to HDFS
      shell: cp -R /home/hadoop/{{ dir_name }} /mnt/hdfs/ # TODO:
          Move instead of copy
```

B.3. cluster --project_destroy Ansible Code

Listing B.3: Ansible Play Code for Project Removal

```
---
# Playbook for server-side data removal in cluster --
    project_destroy
- hosts: namenode
  remote_user: root
  sudo: yes
  sudo_user: hadoop
  tasks:
    - name: Remove directory from HDFS
      shell: rm -rf /mnt/hdfs/{{ dir_name }}
    - name: Remove directory from NameNode
      shell: rm -rf /home/hadoop/{{ dir_name }}
```

B.4. job --start Ansible Code

Listing B.4: Ansible Play Code for Job Execution

```
---
# job --start playbook

- hosts: namenode
  remote_user: root
  tasks:
    - name: Synchronize input data to NameNode
      synchronize: src={{ project_path }} dest=/home/hadoop delete
         =yes recursive=yes
    - name: Change ownership to Hadoop user
      shell: chown -R hadoop:hadoop /home/hadoop/{{ project_name
         }}
    - name: Make MapReduce mapper executable
      file: path=/home/hadoop/{{ project_name }}/.job_config/{{
         job }}/mr_job.py mode="u+rwx" owner=hadoop

- hosts: namenode
  remote_user: root
  sudo: yes
  sudo_user: hadoop
  tasks:
    - name: Remove temporary files from HDFS
      shell: rm -rf /mnt/hdfs/{{ project_name }}/.job_config /mnt/
         hdfs/{{ project_name }}/transient /mnt/hdfs/{{
         project_name }}/out
    - name: Wait 3 seconds
      pause: seconds=3
    - name: Import input data to HDFS
      shell: cp -R /home/hadoop/{{ project_name }}/.job_config /
         home/hadoop/{{ project_name }}/transient /home/hadoop/{{
         project_name }}/out /mnt/hdfs/{{ project_name }}

- hosts: all
  remote_user: root
  tasks:
  - name: Unmount temporary directory (1/3)
```

```
    mount: src=/tmp/local_job_out name=/mnt/hdfs/{{ project_name }}/out fstype=bind state=unmounted
  - name: Unmount temporary directory (2/3)
    mount: src=/tmp/local_job_out name=/mnt/hdfs/{{ project_name }}/out fstype=bind state=unmounted
  - name: Unmount temporary directory (3/3)
    mount: src=/tmp/local_job_out name=/mnt/hdfs/{{ project_name }}/out fstype=bind state=unmounted

- hosts: namenode
  remote_user: root
  sudo: yes
  sudo_user: hadoop
  tasks:
  - name: Remove temporary output directory
    shell: rm -rf /mnt/hdfs/mr_tmp
  - name: Execute MapReduce task
    shell: screen -dmS mr_job sh -c "/opt/hadoop/bin/hadoop jar /opt/hadoop/share/hadoop/tools/lib/hadoop-streaming-*.jar -D mapreduce.map.cpu.vcores=2 -input /{{ project_name }}/.job_config/{{ job }}/input_list.txt -output /mr_tmp -mapper /mnt/hdfs/{{ project_name }}/.job_config/{{ job }}/mr_job.py -inputformat org.apache.hadoop.mapred.lib.NLineInputFormat && touch /mnt/hdfs/{{ project_name }}/.job_config/jobdone; sleep 30"
```

B.5. SerializationHelper Class

Listing B.5: SerializationHelper Class (taken from [41])

```
package de.ugoe.cs.cpdp.mapreduce;

import org.apache.commons.io.IOUtils;

import java.io.*;

import org.apache.hadoop.io.Writable;

public class SerializationHelper {
    public static byte[] serialize(Writable writable) throws
        IOException {
         ByteArrayOutputStream out = new ByteArrayOutputStream();
         DataOutputStream dataOut = null;
         try {
             dataOut = new DataOutputStream(out);
             writable.write(dataOut);
         } catch (Exception e) {
             System.out.println(e);
         }
         finally {
             IOUtils.closeQuietly(dataOut);
         }
         return out.toByteArray();
    }

    public static <T extends Writable> T asWritable(byte[] bytes,
        Class<T> clazz)
             throws IOException {
         T result = null;
         DataInputStream dataIn = null;
         try {
             result = clazz.newInstance();
             ByteArrayInputStream in = new ByteArrayInputStream(
                 bytes);
             dataIn = new DataInputStream(in);
             result.readFields(dataIn);
         } catch (InstantiationException e) {
             // should not happen
             assert false;
         } catch (IllegalAccessException e) {
             // should not happen
             assert false;
         } finally {
             IOUtils.closeQuietly(dataIn);
         }
         return result;
    }
}
```

B.6. WekaTraining Class

Listing B.6: WekaTraining Class

```
package de.ugoe.cs.cpdp.training;

import java.io.*;
import java.util.Arrays;
import java.util.List;
import java.util.logging.Level;

import org.apache.commons.io.output.NullOutputStream;

import de.ugoe.cs.util.console.Console;
import weka.core.Instances;
import weka.classifiers.Classifier;

import org.apache.hadoop.io.Writable;

/**
 * Programmatic WekaTraining
 *
 * first parameter is Trainer Name.
 * second parameter is class name
 *
 * all subsequent parameters are configuration params (for example
     for trees)
 * Cross Validation params always come last and are prepended with
    -CVPARAM
 *
 * XML Configurations for Weka Classifiers:
 * <pre>
 * {@code
 * <!-- examples -->
 * <trainer name="WekaTraining" param="NaiveBayes weka.classifiers
    .bayes.NaiveBayes" />
 * <trainer name="WekaTraining" param="Logistic weka.classifiers.
    functions.Logistic -R 1.0E-8 -M -1" />
 * }
 * </pre>
 *
 */
public class WekaTraining extends WekaBaseTraining implements
   ITrainingStrategy, Writable {

        @Override
        public void apply(Instances traindata) {
                PrintStream errStr       = System.err;
                System.setErr(new PrintStream(new NullOutputStream
                   ()));
                try {
                        if(classifier == null) {
                                Console.traceln(Level.WARNING,
                                   String.format("classifier null
                                   !"));
                        }
                        classifier.buildClassifier(traindata);
                } catch (Exception e) {
                        throw new RuntimeException(e);
                } finally {
                        System.setErr(errStr);
                }
        }

        public void write(DataOutput out) throws IOException {
                ByteArrayOutputStream bo = new
                   ByteArrayOutputStream();
                ObjectOutputStream so = new ObjectOutputStream(bo)
                   ;

                // Write classifier (Instance of Classifier)
                so.writeObject(classifier);
                so.flush();
                byte[] b = bo.toByteArray();
                int length = b.length;
                out.writeInt(length);
                out.write(b);

                // Write classifierClassName / classifierName
                out.writeUTF(classifierClassName);
			out.writeUTF(classifierName);

			// Write classifierParams (Instance of String[], convert to List before)
			bo = new ByteArrayOutputStream();
			so = new ObjectOutputStream(bo);
			List<String> classifierParamsList = Arrays.asList(classifierParams);
			so.writeObject(classifierParamsList);
			so.flush();
			b = bo.toByteArray();
			length = b.length;
			out.writeInt(length);
			out.write(b);
	}

	public void readFields(DataInput in) throws IOException {
			// Read classifier
			final int classifierLength = in.readInt();
			byte classifierArray[] = new byte[classifierLength];
			in.readFully(classifierArray);
			ByteArrayInputStream bi = new ByteArrayInputStream(classifierArray);
			ObjectInputStream si = new ObjectInputStream(bi);
			try {
					classifier = (Classifier) si.readObject();
			} catch(Exception e) {
					System.out.println("Exception caught trying to deserialize classifier");
			}

			// Read classifierClassName / classifierName
			classifierClassName = in.readUTF();
			classifierName = in.readUTF();

			// Read classifierParams
			final int classifierParamsLength = in.readInt();
			byte classifierParamsArray[] = new byte[classifierParamsLength];
```

```java
        in.readFully(classifierParamsArray);
        bi = new ByteArrayInputStream(classifierParamsArray);
        si = new ObjectInputStream(bi);
        try {
            List<String> classifierParamsList = (List<String>) si.readObject();
            classifierParams = (String[]) classifierParamsList.toArray();
        } catch(Exception e) {
            System.out.println("Exception caught trying to deserialize classifierParams");
        }
    }

    public static WekaTraining read(DataInput in) throws IOException {
        WekaTraining w = new WekaTraining();
        w.readFields(in);
        return w;
    }

    public String[] getParams() {
        return classifierParams;
    }
}
```

B.7. TrainerMapper Class

Listing B.7: SerializationHelper Class

```java
package de.ugoe.cs.cpdp.mapreduce;

import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.OutputStream;

import org.apache.commons.codec.binary.Base64;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.io.LongWritable;
import weka.core.Instances;
import weka.classifiers.Classifier;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import de.ugoe.cs.cpdp.training.WekaTraining;


public class TrainerMapper
        extends Mapper<LongWritable, Text, Object, Object> {
    private static Configuration conf = null;
    private static Instances traindata = null;
    private static Path serializationTrainerOutputDir = null;

    protected void setup(Context context) {
        conf = context.getConfiguration();
        serializationTrainerOutputDir = new Path(conf.get("
           outputpath"));
        String serializedTraindata = conf.get("traindata");
        try {
            byte b[] = Base64.decodeBase64(serializedTraindata);
            ByteArrayInputStream bi = new ByteArrayInputStream(b);
            ObjectInputStream si = new ObjectInputStream(bi);
            traindata = (Instances) si.readObject();
        } catch( Exception e ) {
            System.out.println(e);
        }
    }

    public void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {
        // Read serialized trainer from HDFS
        Path inputFile = new Path(value.toString());
        FileSystem fs = FileSystem.get(conf);
        FSDataInputStream in = fs.open(inputFile);
        final int available = in.available();
        byte serializedTrainer[] = new byte[available];
        in.readFully(0, serializedTrainer);

        // Deserialize and apply trainer
        WekaTraining trainer = SerializationHelper.asWritable(
           serializedTrainer, WekaTraining.class);
        trainer.apply(traindata);

        // Serialize trainer and write to HDFS
        String filename = Long.toString(System.nanoTime()) + ".ser
           ";
        Path serializationFile = new Path(
           serializationTrainerOutputDir, filename);
        OutputStream os = fs.create(serializationFile);
        os.write(SerializationHelper.serialize(trainer));
        os.close();
    }
}
```

List of Figures

Listings

List of Tables

Bibliography

[1] Amazon Elastic MapReduce. Online: http://aws.amazon.com/de/elasticmapreduce/. Accessed: 2015-04-15.

[2] Ansible is Simple IT Automation. Online: http://www.ansible.com/home. Accessed: 2015-04-15.

[3] Apache Ambari. Online: https://ambari.apache.org/. Accessed: 2015-04-15.

[4] Apache Commons. Online: http://commons.apache.org/. Accessed: 2015-04-15.

[5] AppScale. Online: http://www.appscale.com/. Accessed: 2015-04-15.

[6] AutoQUEST. Online: https://autoquest.informatik.uni-goettingen.de. Accessed: 2015-04-15.

[7] Chef - Code Can. Online: https://www.chef.io/. Accessed: 2015-04-15.

[8] Cloudera. Online: http://www.cloudera.com/. Accessed: 2015-04-15.

[9] Cloudera Enterprise. Online: http://www.cloudera.com/content/cloudera/en/products-and-services/cloudera-enterprise.html. Accessed: 2015-04-15.

[10] Cloudera Express. Online: http://www.cloudera.com/content/cloudera/en/products-and-services/cloudera-express.html. Accessed: 2015-04-15.

[11] cloudinit.d. Online: http://www.nimbusproject.org/doc/cloudinitd/latest/. Accessed: 2015-04-15.

[12] coreutils - GNU Core Utilities. Online: http://www.gnu.org/software/coreutils/. Accessed: 2015-04-15.

[13] Crosspare Application. Online: https://crosspare.informatik.uni-goettingen.de/svn/crosspare/trunk/CrossPare/. Accessed: 2015-04-15.

[14] ETSI - Network Functions Virtualisation. Online: http://www.etsi.org/technologies-clusters/technologies/nfv. Accessed: 2015-04-15.

[15] Ganglia. Online: http://ganglia.sourceforge.net/. Accessed: 2015-04-15.

[16] Google App Engine. Online: https://cloud.google.com/appengine. Accessed: 2015-04-15.

[17] Hadoop. Online: http://hadoop.apache.org/. Accessed: 2015-04-15.

[18] Hadoop. HDFS Architecture. Online: http://hadoop.apache.org/docs/r2.6.0/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html. Accessed: 2015-04-15.

[19] Hadoop Mailing List - Why not use Serializable? Online: http://www.mail-archive.com/hadoop-user@lucene.apache.org/msg00378.html. Accessed: 2015-04-15.

[20] Hadoop Main 2.6.0 API - NLineInputFormat Class. Online: https://hadoop.apache.org/docs/stable/api/org/apache/hadoop/mapred/lib/NLineInputFormat.html. Accessed: 2015-04-15.

[21] Hadoop Main 2.6.0 API - Writable Interface. Online: https://hadoop.apache.org/docs/current/api/org/apache/hadoop/io/Writable.html. Accessed: 2015-04-15.

[22] Hadoop. MapReduce. Online: http://wiki.apache.org/hadoop/HadoopMapReduce. Accessed: 2015-04-15.

[23] Hadoop MapReduce tutorial. Online: http://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html. Accessed: 2015-04-15.

[24] Hadoop on the Google Cloud Platform. Online: https://cloud.google.com/solutions/hadoop/. Accessed: 2015-04-15.

[25] Hadoop. What is the Purpose of the Secondary Name-node? Online: http://wiki.apache.org/hadoop/FAQ#What_is_the_purpose_of_the_secondary_name-node.3F. Accessed: 2015-04-15.

[26] Hadoop Wiki - How Many Maps And Reduces. Online: http://wiki.apache.org/hadoop/HowManyMapsAndReduces. Accessed: 2015-04-15.

[27] Hadoop. YARN. Online: http://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html. Accessed: 2015-04-15.

[28] HashiCorp. Online: https://hashicorp.com/. Accessed: 2015-04-15.

[29] Java Documentation - Serializable Objects. Online: http://docs.oracle.com/javase/tutorial/jndi/objects/serial.html. Accessed: 2015-04-15.

[30] Java Platform SE7 - Serializable Interface. Online: http://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html. Accessed: 2015-04-15.

[31] JavaScript Object Notation. Online: http://json.org/. Accessed: 2015-04-15.

[32] Meet Infrastructure as Code- DevOps.com. Online: http://devops.com/blogs/meet-infrastructure-code/. Accessed: 2015-04-15.

[33] Message Passing Interface (MPI). Online: http://www.mcs.anl.gov/research/projects/mpi/. Accessed: 2015-04-15.

[34] Nagios. Online: http://www.nagios.org. Accessed: 2015-04-15.

[35] Nimbus Project. Online: http://www.nimbusproject.org/. Accessed: 2015-04-15.

[36] OpenMP. Online: http://openmp.org/. Accessed: 2015-04-15.

[37] OpenStack. Online: http://www.openstack.org/. Accessed 2014-09-26.

[38] OpenStack Sahara. Online: https://wiki.openstack.org/wiki/Sahara. Accessed: 2015-04-15.

[39] Puppet Labs: IT Automation Software for System Administrators. Online: http://puppetlabs.com/. Accessed: 2015-04-15.

[40] Ruby Programming Language. Online: https://www.ruby-lang.org/. Accessed: 2015-04-15.

[41] Stack Overflow - How To Unit Test Hadoop Writable. Online: http://stackoverflow.com/questions/13288214/how-to-unit-test-hadoop-writable. Accessed: 2015-04-15.

[42] Tera-Promise Repository. Online: http://openscience.us/repo/. Accessed: 2015-04-15.

[43] TOP 500 Supercomputer Sites - Statistics. Online: http://www.top500.org/statistics/. Accessed: 2015-04-15.

[44] TORQUE Resource Manager. Online: http://www.adaptivecomputing.com/products/open-source/torque/. Accessed: 2015-04-15.

[45] Vagrant. Online: https://www.vagrantup.com/. Accessed: 2015-04-15.

[46] Vagrant OpenStack Cloud Provider. Online: https://github.com/ggiamarchi/vagrant-openstack-provider. Accessed: 2015-04-15.

[47] Weka 3 - Data Mining with Open Source Machine Learning Software in Java. Online: http://www.cs.waikato.ac.nz/ml/weka/. Accessed: 2015-04-15.

[48] What is DevOps? Online: http://radar.oreilly.com/2012/06/what-is-devops.html. Accessed: 2015-04-15.

[49] YAML Specification. Online: http://www.yaml.org/spec/. Accessed: 2015-04-15.

[50] G. S. Almasi and A. Gottlieb. *Highly Parallel Computing.* Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1989.

[51] M. Armbrust, A. Fox, R. Griffith, et al. Above the Clouds: A Berkeley View of Cloud Computing. Technical Report UCB/EECS-2009-28, University of California at Berkeley, February 2009.

[52] C. Boyd. Data-parallel computing. *Queue*, 6(2):30–39, 2008.

[53] C. Bunch, N. Chohan, C. Krintz, and K. Shams. Neptune: a domain specific language for deploying hpc software on cloud platforms. In *Proceedings of the 2nd international workshop on Scientific cloud computing*, pages 59–68. ACM, 2011.

[54] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[55] M. Flynn. Some Computer Organizations and Their Effectiveness. *Computers, IEEE Transactions on*, C-21(9):948–960, Sept 1972.

[56] P. Harms and J. Grabowski. Usage-Based Automatic Detection of Usability Smells. In *Human-Centered Software Engineering*, pages 217–234. Springer, 2014.

[57] P. Harms, S. Herbold, and J. Grabowski. Trace-based task tree generation. In *ACHI 2014, The Seventh International Conference on Advances in Computer-Human Interactions*, pages 337–342, 2014.

[58] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[59] S. Nanda and T. Chiueh. A Survey on Virtualization Technologies. Technical Report TR-179, Stony Brook University, February 2005.

[60] National Institute of Standards and Technology. The NIST Definition of Cloud Computing. *NIST Special Publication*, (800-145), 2011.

[61] P. Pacheco. *An Introduction to Parallel Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011.

[62] S. Wadkar, M. Siddalingaiah, and J. Venner. *Pro Apache Hadoop.* Apress, Berkely, CA, USA, 2nd edition, 2014.