VERKFRÆÐISTOFNUN
HÁSKÓLA ÍSLANDS

Analysing High-Energy Physics Data Using the MapReduce Paradigm in a Cloud Computing Environment

Fabian Glaser, Helmut Neukirchen

Second, extended edition, July 10, 2012

Report nr. VHI-01-2012, Reykjavík 2012

Fabian Glaser, Helmut Neukirchen. Analysing HEP Data Using the MapReduce Paradigm in a Cloud Computing Environment,
Engineering Research Institute, University of Iceland, Technical report VHI-01-2012,
Second, extended edition, July 2012

Engineering Research Institute, University of Iceland, Hjarðarhagi 2-6, IS-107 Reykjavík, Iceland

Abstract

Huge scientific data, such as the petabytes of data generated by the Large Hadron Collider (LHC) experiments at CERN are nowadays analysed by Grid computing infrastructures using a hierarchic filtering approach to reduce the amount of data. In practise, this means that an individual scientist has no access to the underlying raw data and furthermore, the accessible data is often outdated as filtering and distribution of data only takes places every few months. The advent of Cloud computing promises to make a "private computing Grid" available to everyone via the Internet. Together with Google's MapReduce paradigm for efficient processing of huge sets of data, the Cloud may be a viable alternative for a scientist to perform analysis of huge scientific data.
This project investigates the applicability of the MapReduce paradigm, in particular as implemented by Apache Hadoop, for analysing LHC data and at investigating the feasibility of Cloud computing, e.g. as provided by the Amazon Cloud, for this task instead of Grid computing. Even though it was not possible to assess the applicability to petabyte-sized problems, this project serves as a feasibility study to demonstrate that this approach is applicable at all. This includes:

1. Adaptation of the existing LHC data analysis to the MapReduce paradigm.
2. Implementation of the adapted analyses for being usable with Apache Hadoop.
3. Investigation of the applicability of the approach by setting up and running the analyses in the Amazon EC2 public Cloud computing and storage environment.

Contents

1 Introduction

The Large Hadron Collider (LHC) experiments conducted at CERN produce vast amounts of data. The produced data consists of millions of events that describe particle collisions in the LHC and their evaluation typically includes statistical analysis of selected events. To select from the vast number of events the ones of interest, a lot of computational power is needed. Because we can look at each event separately, the analysis process can be highly parallelized.
Currently, Grid computing is used to store and analyze the LHC data. The Grid infrastructure was designed as a decentralized system to distribute the load of handling the huge data volume produced by the LHC experiments. To make efficient use of available resources, the computing models implemented by the different collaborations impose certain restrictions, such as access to only partial event information for the full data sample, or full information for only small subsets of events which are selected in a centrally organized pre-processing step. Depending on the scheduling of these activities, certain physics analysis easily can suffer several months delay. Therefore, the RAVEN [8] project aims at addressing the problem of the analysis of huge sets of data and is looking for promising approaches to this problem.
In the last years, a new paradigm for parallel programming and processing huge amounts of data arose. This report investigates the applicability of this paradigm to high-energy physics (HEP) analysis. The paradigm is called *MapReduce* and was first introduced in a whitepaper by Google [3]. While Google is using a proprietary implementation of MapReduce in their own computing facilities, there exists a popular implementation by the Apache Software Foundation, which is called *Hadoop*. Hadoop provides a framework to develop and execute programs according to the MapReduce paradigm on large compute clusters.
Often local computational facilities are limited and Grid or Cloud Computing can be used to utilize or rent resources remotely. This document describes a study to parallelize a sample analysis of data related to the LHCb[1] experiment with MapReduce and Hadoop and utilize the Amazon Web Services for Cloud Computing to deploy a cluster which satisfies our requirements.
The data to be analysed is a toy study based on Monte Carlo events generated with a standalone version of the PYTHIA-8 simulation program and made available by colleagues from the Max Planck Institute for Nuclear Physics (MPIK) in Heidelberg, who are members of the LHCb collaboration. For this study, detector simulation and reconstruction were emulated by simple fiducial cuts and some momentum smearing for charged tracks. Neutral particles were not considered. The data is stored as ROOT files (see section 2.1) in a compact binary format developed at MPIK which is also used for fast exploratory access to real data or centrally produced Monte Carlo with detailed detector simulation and the full reconstruction chain.

1.1 Goals and contributions

The goal of the project described in this report was to investigate the applicability of the MapReduce paradigm for HEP analyses and the speed-up that can be expected. We start with a sample analysis provided by a small C++ program that is used at the MPIK to analyse data such as the data produced by the LHCb experiment at CERN. This C++ program makes use of the ROOT HEP analysis software framework. The goal of this project is to port the

[1] http://lhcb-public.web.cern.ch/lhcb-public/

existing analysis to the Hadoop framework and assure its scalability. Due to the fact that local computational capabilities with superuser privileges (needed to install and configure Hadoop and the ROOT framework without having to depend on a system administrator of a cluster) were limited we also show how the Cloud Computing environment offered by Amazon can be used to deploy a scalable cluster, which can be used for the analysis. This project contributes in the following points:

- Show how the Map Reduce programming model can be applied to the LHCb data.
- Describe a scalable implementation of the existing data analysis from the LHCb experiment with the Hadoop framework.
- Show how the Amazon Web Services for Cloud Computing can be used to deploy a Hadoop Cluster that satisfies the requirements.
- Evaluate the provided solution and point out some ideas for future research.

1.2 Outline

The document is organized as follows: After this introduction, we give an overview of the tools utilized for the analysis. In chapter 3, the implementation of the ROOT-based analysis on top of the Hadoop framework is described in more detail. Chapter 4 shows how we utilized the Amazon Web Services to deploy a Hadoop cluster, while chapter 5 provides an evaluation of the provided solution. Finally, chapter 6 concludes and points out further directions.

1.3 Notations used in this document

Whenever a new term is introduced, it is written in *italic* letters. Class names and variables, when appearing in the text, are written in `typewriter` style.

2 Background

This section provides an overview on the foundations and the tools used in this project. It introduces briefly the *ROOT framework*, *Map Reduce*, *Hadoop* and the *Amazon Web Services*.

2.1 The ROOT Framework

ROOT [2] is a C++ software framework developed at CERN, that provides tools to handle and analyze large amounts of data. The data is defined in a specialized set of objects. ROOT provides functionality for histograms, curve fitting, minimization, graphics and visualization (There is also a parallel processing version of ROOT available, which is called PROOF – however it was not used in this project.). The ROOT architecture consists of around 1200 classes organized in 60 libraries and 19 categories (modules).
Because the events we are going to analyze throughout this project are stored in a proprietary ROOT file format, we are going to use the framework to read from and work on the input data. A ROOT file contains objects that can be hierarchically organized in several levels similar to a UNIX directory. ROOT files provide random access to objects. This is managed through special objects called `TKeys`, which provide an index for the stored data. Data stored in ROOT files can be organized in trees and compression can be applied. ROOT's compression algorithm is based on *gzip*. More information can be found in the ROOT user guide [9].

2.2 MapReduce and the Google File System

MapReduce is a paradigm and framework for implementing data intensive application that was first introduced by Jeffrey Dean and Sanjay Ghemawat at Google in 2004 [3] and attracted a lot of attention since then. In a divide-and-conquer manner the programmer writes code in terms of a `map` and `reduce` function. Both functions take a *key/value* pair as input. The `map` function takes an input key/value pair generated from the input data and emits one or more key/value pairs, which are then passed to the `reduce` function.
The original example provided by Dean and Ghemawat for the application of the MapReduce paradigm is counting the appearance of words in a large text file. The pseudocode is given in listing 2.1. The map function takes the document name as the key and file contents

```
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));
```

Listing 2.1: Word Count example [3].

as a value. For each word in the context it emits a key/value pair, whereby the key is the word itself and the value is the current count of the word, which is "1". These pairs are then passed to the reduce function. For each key (in this example: each word) the reduce function is called once and the count of the words is summed up. In the end, the reduce function emits a final key/value pair, whereby the key is still the word and value holds the count of the word's appearance in the document. As this is done for every unique key, a histogram of all words occurring in the input file is computed.

2.2.1 Architecture

Typically, the input data is split up and on each such a split, a map task is executed in parallel to other map tasks thus leading to a speed-up in comparison to non-parallel processing; the same applies to the reduce tasks that are started as soon as output from the map tasks is available. Figure 2.1 shows the typical execution procedure in the MapReduce framework. A master node is assigning map and reduce tasks to worker nodes. The input data is partitioned into splits (of typically 64 MB size) and each split is passed to a worker node executing the map task. These tasks write intermediate key/value pairs to the local disk of the worker node, which are then read remotely by workers executing reduce tasks. These workers then produce the final output.

2.2.2 The Google File System

Network bandwidth is a limiting resource in the execution described above. Dean and Ghemawat describe how the *Google File System* (GFS) [7] is used to preserve bandwidth usage. The Google File System implements a distributed way of storing large data files on a cluster. The file is hereby split up into blocks (typically 64 MB in size) and each block is stored several times (typically three times) on different nodes in the cluster. Google's MapReduce implementation now makes use of the locality information of these file blocks by assigning a map task operating on one of the blocks to a node which holds the block already or is nearby to a node that holds the block (e.g. in the same rack of a cluster). In this way huge amounts of bandwidth usage can be saved by doing the processing where the data is stored.
Google's implementation of the MapReduce framework is proprietary and not available for the public. The next section will introduce *Hadoop* which is an open-source framework implementing the MapReduce programming model.

2.3 Hadoop and HDFS

Hadoop [6] is a Java-based open-source framework developed by the Apache Software Foundation, that implements the MapReduce programming model. It has been deployed on huge clusters and is used by, e.g., *Facebook*, *Amazon* and *AOL*[1]. It comes with the *Hadoop Distributed File System* (HDFS) which mimics the properties of the Google File System described above. After describing Hadoop's Architecture and HDFS in this section, we will introduce *HadoopPipes* and *HadoopStreaming*, two possibilities to parallelize applications with Hadoop, which are not implemented in Java.

[1] http://wiki.apache.org/hadoop/PoweredBy

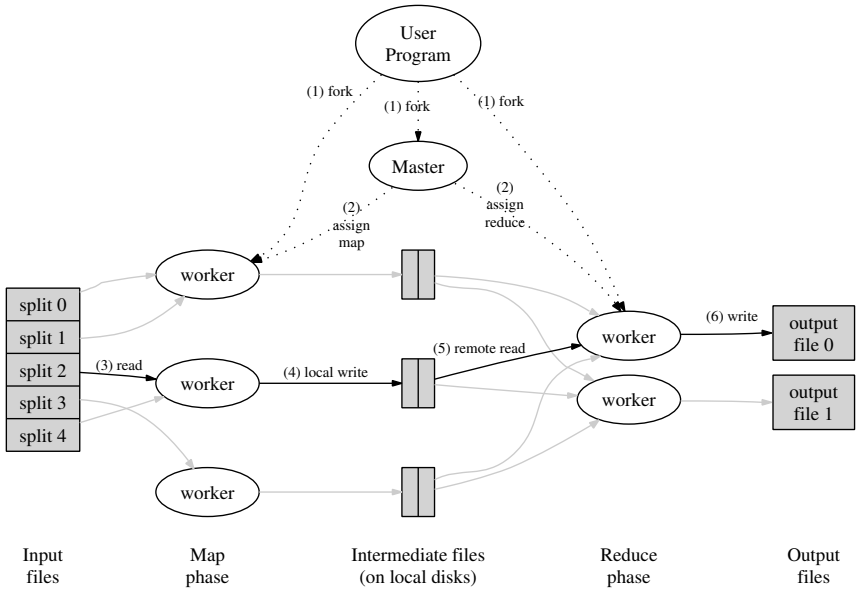

Figure 2.1: Execution overview [3].

2.3.1 Architecture

The architecture of a cluster running Hadoop is based on Google's approach shown in figure 2.1. It is a master-slave architecture, where one or two master nodes are assigning tasks to several worker/slave nodes. The master nodes are serving as *NameNode*s and *JobTracker*s, while the worker nodes are serving as *DataNode*s and *TaskTracker*s. These daemons are described in the following.

- **NameNode**: The `NameNode` is keeping track of the data in the Hadoop File System (HDFS). It does not store the data itself, but manages the add/copy/delete and move operations that are executed on the data distributed over the `DataNodes`. It is a single point of failure in the current Hadoop implementation, because a failure of the `NameNode` causes the whole file system to become unavailable. As a remedy, Hadoop provides a optional daemon, which is called `SecondaryNameNode`. Nevertheless it does not provide full redundancy, because it only creates snapshots of the namespace. There are some plans to implement a `Backup Node` and an additional `Checkpoint Node` in future versions of Hadoop, but they have not been realized in the current version of Hadoop (Hadoop-1.0.0).

- **DataNode**: The data stored in the Hadoop cluster is stored on the `DataNodes`. Data stored in HDFS is internally split up into blocks (64 MB) and then distributed and replicated around the `DataNodes`.

- **JobTracker**: The `JobTracker` is responsible for assigning map and reduce tasks to nodes running `TaskTrackers`. It therefore asks the `NameNode` where the data resides and tries to assign the corresponding tasks to `TaskTrackers` which are nearby. The

`JobTracker` monitors the work of all `TaskTrackers` and updates its status when a job is completed or failed.

- **TaskTracker**: To each node that is configured as a `TaskTracker`, the `JobTracker` can assign certain tasks. The number of tasks a `TaskTracker` can accept is limited by the number of *slots* it offers. Each task will get assigned to a certain slot. The `TaskTracker` spawns a separate Java Virtual Machine for each task to make sure, that it does not get effected if a task crashes. While assigning the tasks to certain nodes the `JobTracker` takes care that tasks that work on certain data sets get executed on the `DataNode` where the data persists. If the corresponding `TaskTracker` does not offer a free slot, it will chose another `TaskTracker` running on a machine in the same rack (assuming that a cluster is organized into racks where machines of the same rack have a faster network connection than to machines of a different rack).

2.3.2 The Hadoop File System

The Hadoop File System (HDFS) is based on the description of GFS given in [7]. HDFS is a distributed file system, designed to store large amounts of data on clusters build of commodity hardware. As in GFS the data is split up into blocks and replicated to several nodes of the cluster. While HDFS is mostly POSIX compatible it relaxes a few of its requirements to enable streaming access to data. Its design goals and implementation are described in the HDFS Architecture Guide [1] and can be summarized as follows:

- **Hardware Failure**: In large clusters with thousands of nodes, the chances are pretty high that several cluster nodes suffer from hardware failure. Therefore, replication is used to take care that HDFS data is still available even if an HDFS node fails. HDFS aims to detect and recover from these failures as fast as possible. The `DataNodes` described above are sending regular heartbeat-messages to the `NameNode`, informing it that they are still available and which data chunks they hold. If the `NameNode` does not receive any message from a certain `DataNode` for a certain period of time, it initializes a replication of the data that was stored on the node to other `DataNodes`.
- **Streaming Data Access**: HDFS is designed to enable high throughput sequential access to data stored in the cluster, rather than enabling low latency random access. Its usage scenario is batch processing of jobs working on huge amounts of data without the need for interactivity.
- **Large Data Sets**: The typical size of files stored in HDFS are a few gigabytes up to some terabytes. While the files are split up into blocks and replicated over several `DataNodes`, they still appear to the user as single files.
- **Simple Coherency Model**: HDFS is designed for data which is written once and read many times. This leads to a simplified data coherency model.
- **"Moving computation is cheaper than moving data"**: As we have seen in the description of the Google File System (section 2.2.2), we can speed up computation by moving it to a worker node where or close to where the data is stored. Therefore HDFS provides an interfaces which can be used by applications to obtain these locality information.
- **Portability across Heterogeneous Hardware and Software Platforms**: HDFS is written an Java and aims to be deployable on all mayor platforms.

The size of the file blocks in HDFS is set to 64 MB by default. The number of replications of each file block can be controlled by the *replication factor*. A replication factor of two, would mean that the HDFS cluster always keeps two copies of each file block.

2.3.3 Hadoop Streaming

Hadoop Streaming is an API that enables the user to work with map and reduce functions that are not written in Java. Both functions can be any applications that read input from the standard input stream (stdin) and write their results to standard output stream (stdout). The standard input format hereby is text based and read line-by-line, whereby each line represents a key/value pair separated by a tab. Hadoop hereby makes sure that the lines passed to the reduce function are sorted by keys. Programs that are written in this manner can be simply tested with the help of UNIX Pipes (of course, only sequential processing is used as Hadoop is not involved):

```
cat input | map | reduce
```

Splitting up the input and parallelizing map and reduce calls in the cluster, whereby using the locality information of the input data is done with the help of a special input format for Hadoop. For a more detailed discussion, please take a look at the implementation chapter, section 3.2.

2.3.4 Hadoop Pipes

A second way to integrate map and reduce functions that are not implemented in Java with Hadoop are *Hadoop Pipes*. Programs written with Hadoop Pipes link against a thin Hadoop C++ library which enables the written functions to be spawned as processes and communicate over sockets. The C++ interface is SWIG[2] compatible and can be generated for other source languages as well. Unlike Streaming, Hadoop Pipes does not use the standard input and output channels. The key/value pairs are passed to the mapper and reducers as C++ Standard Template Library (STL) strings so any data that should be used as keys or values need to be converted beforehand. To use Hadoop Pipes the code needs to inherit from certain base classes and reimplement some of there functionality. During this project some effort was spent on developing an implementation based on Hadoop Pipes, but it was discarded, because the implementation with Hadoop Streaming turned out to be faster and more straight-forward!

2.4 Amazon Web Services

Amazon Web Services (AWS) provides a scalable, on demand, Infrastructure-as-a-Service environment, which can be used to provide large amounts of computing power, storage and data management infrastructure on a very short timescale. AWS thereby consists of several different services which can be used separately or in combination to provide the desired functionality. After introducing the concept of *Availability regions*, this section will describe the services, that were used in the project, namely the *Simple Storage Service (S3)*, *Elastic Cloud (EC2)* and *Elastic Map Reduce (EMR)*. In the end we will comment on the usage fees.

[2]More information on SWIG can be found on http://www.swig.org/

2.4.1 Availability zones

Each service of AWS is bound to an *Availability zone*. Availability zones are designed to be independent of the failure of one another. Multiple availability zones form together a *region*. Currently there are seven available regions: US East (Northern Virginia), US West (Oregon), US West (Northern California), EU (Ireland), Asia Pacific (Singapore), Asia Pacific (Tokyo) and AWS GovCloud (US). Zones and regions also limit the availability of certain service instances to one another.

2.4.2 Simple Storage Services and Elastic Block Store

The AWS mainly contain two services that are used for persistent data storage in the Amazon cloud: the *Amazon Simple Storage Service (S3)* and the *Amazon Elastic Block Store (EBS)*.
S3 is the general storage mechanism for AWS. It provides accessibility over the web and is theoretically unlimited in size from the users viewpoint. All objects stored in S3 are contained in *buckets*. Each bucket has a unique name and can be made accessible over HTTP. Objects in buckets can be grouped into subfolders. Amazon provides access control to buckets in two ways: access control lists (ACLs) and bucket policies. ACLs are permission lists for each bucket while policies provide a more generalized way of managing permissions that can be applied to users, user groups or buckets.
EBS provides block storage and can in particular be used to supply persistent storage for the Amazon Elastic Cloud (see next section). The EBS volumes can be formatted with a desired file system and are suitable for applications that need to use persistent storage e.g. databases. Each volume is created in a particular availability zone and can only be attached to virtual machines within the same zone. They can be anything from one GB to one TB in size. Amazon EBS provides the ability to back-up snapshots of the current volume state to S3. Snapshots get stored incrementally that means that for consecutive snapshots only the changed blocks are backed-up and not the whole volume again.

2.4.3 Elastic Cloud

Amazon Elastic Compute Cloud (EC2) is a web service that provides scalable computing resources on demand. The basic building blocks of EC2 are the *Amazon Machine Images (AMIs)* that provide templates for the boot file system of virtual machines that can be hosted in the Elastic Cloud. Users can launch several instances of a single AMI as different instance types, which represent different hardware configuration. The CPU power of the instances is measured in *Amazon EC2 Compute Unit*s. According to Amazon, one unit provides the capacity of a 1.0-1.2 GHz 2007 Intel Opteron or Xeon processor. Amazon and several third party developers provide AMIs, which can be used for free or in exchange for a fee. During the time this document was created, there were around 3500 different AMIs to choose from. In case the available AMIs do not fulfill the users requirements, it is possible to create own AMIs, which can be made available to public or just used for private purposes. It is also possible to import local virtual machine images created by a Citrix Xen, Microsoft Hyper-V or VMware vSphere virtualization platform into EC2 and launch them as instances.
To provide access control to the launched instances, Amazon introduces the concept of *security groups*. A security group is a collection of rules that define what kind of traffic should be accepted or discarded by an instance. Instances can be assigned to numerous groups. Changes to the rules can be made at any time and are propagated to all instances automatically.

EC2 uses both the Amazon Simple Storage Service (S3) and the Amazon Elastic Block Store (EBS) as background storage. When designing a custom AMI or choosing from the available AMIs one need to choose the kind of storage that should be used by the launched instances. If an AMI only uses *ephemeral storage*, also sometimes referred as *instance storage* the data stored on the instance is only available until the instance is terminated. For persistent storage (i.e. even when an instance is terminated), EBS-backed AMIs can be used. The instances of these AMIs are backed up by an EBS volume, which is connected to the instance on start up. Amazon also provides services for database storage, address mappings and monitoring of instances which will not be further described in this document.

2.4.4 Elastic Map Reduce

Amazon *Elastic Map Reduce (EMR)* offers users that have written their code for the Hadoop Framework to run their programs in a Map Reduce environment on top of EC2. EMR therefore automatically launches AMIs that have a preinstalled Hadoop framework and takes care of the users job submission. For the time being EMR supports Hadoop version 0.18 and 0.20. Jobs can be specified by the user over a web interface, a command line tool or the *Elastic Map Reduce API*. The user specifies either a jar file for running a Java job on Hadoop or a mapper and reducer written in another programming language to run as Hadoop Streaming Job. EMR does not offer the utilization of Hadoop Pipes over its interface so far. The jar files, scripts or binaries that should be executed, need to be uploaded to S3 beforehand. For configuration purposes EMR offers the possibility to the user to define *Bootstrap Actions*, which are scripts that get executed on each Hadoop node before the job is finally started. These scripts can be written in any language that is already installed on the AMI instances, such as Ruby, Perl or bash.
Elastic Map Reduce typically utilizes a combination of the following filesystems: HDFS, *Amazon S3 Native File System (S3N)*, local file systems and the *legacy Amazon S3 Block File System*. S3N is a file system which is implemented to read and write files on S3 and the local file system refers to the file system of the AMI instance itself. The user can choose on which kind of AMI instance types his job should be executed depending on the type of filesystem that is used to store the data.

2.4.5 Usage fees

The usage fees incurring when using AWS are differing by regions. In this project we were using the region EU (Ireland). During the time of the project the pricing was calculated as follows:

- **S3 and EBS**: For data stored in S3 or EBS Amazon charges a fee of 0.125$ per GB and month. Transferring data to the cloud was free. Transferring out of the cloud data was free for the first GB per month, the next 10 TBs of data cost 0.120$ per GB.
- **EC2**: The costs for the instances started in EC2 depend on the instance type that is started and are calculated per hour. For example a small Linux instance (1.7 GB memory, 1 EC2 Compute Unit, 160 GB instance storage, 32 Bit or 64 Bit) was priced 0.09$ per hour, while a large Linux instance (7.5 GB of memory, 2 Virtual Cores with 2 EC2 Compute Units each, 850 GB instance storage, 64 Bit) was 0.36$ per hour.
- **EMR**: While EMR uses EC2 internally, the costs for the instances automatically started by EMR are less than the ones started in EC2 manually. For the time of the project a small instance was 0.015$ per hour, while a large instance was 0.06$ per hour.

The actual pricing can be found on the websites for S3[3], EC2[4] and EMR[5].

[3]http://aws.amazon.com/s3/
[4]http://aws.amazon.com/ec2/
[5]http://aws.amazon.com/elasticmapreduce/

3 Implementation in the Hadoop framework

The C++ LHCb analysis code makes use of ROOT to read the data from a file and scans the data for particles of a certain mass. The particle masses in a certain range are recorded and a histogram of their quantity is created.
During the project work, Hadoop Streaming was used to port the C++ LHCb data analysis code to Hadoop. The first part of this section provides the application of the MapReduce paradigm to the LHCb data, while the second part describes the implementation of the analysis with Hadoop Streaming.

3.1 Applying MapReduce to LHCb data analysis

In general, the data to be analysed consists of thousands or even millions of events. Each event holds the data for a single collision and the tracks of the involved particles. In our case we need to scan through all events, looking for particles with a certain characteristic (e.g. with a certain mass). So we need to touch every event, loop through every track and calculate the parameters of interest of the involved particles. In a second step we are taking the obtained data and calculate some statistics (e.g. a histogram). It is not uncommon that events that hold information of interest are only a small fraction of those provided as input data. The main computational effort is spent on calculating the characteristics of particles involved in an event and on the selection process. Looping through all events can be highly parallelized, because each event can be processed independent from the other events.
The idea is to split up the input data into its events and pass each event to a map function. The map function than calculates the characteristics of the involved particles and in case there is a particle of interest, it passes the data to the reduce function that is then doing the statistical evaluation.

3.2 Implementation with Hadoop Streaming

This section gives a small overview of the steps taken to implement the above analysis with the help of Hadoop Streaming: the map and reduce functions need to implement the interface of Hadoop Streaming, input format classes need to be provided to decide on where to split the input data to allow parallel processing by multiple map jobs, and the existing C++ LHCb analysis code needs to be transformed into map and reduce functions.

3.2.1 Interface of map and reduce functions

As described in section 2.3.3, the mapper and reducer for Hadoop Streaming are implemented as independent executables. The code modularity can therefore be chosen freely, it must only be assured that both programs read their input from the standard input stream line by line, whereby each line corresponds to a key/value pair. In addition the mapper needs to write its intermediate key/value pairs to the standard output stream one pair per line. In the implementation created in this project, both mapper and reducer are implemented as single classes.

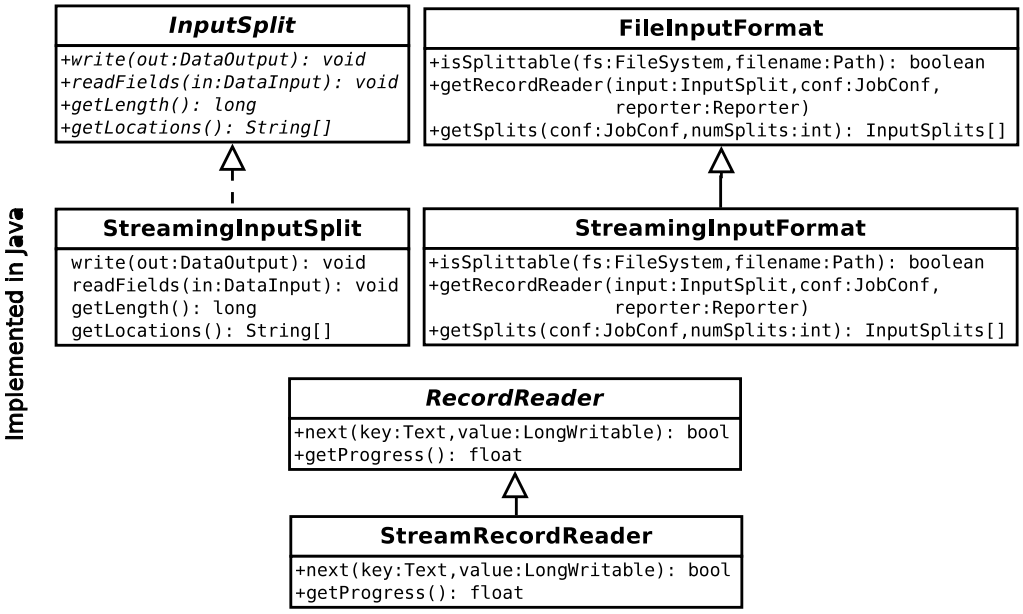

Figure 3.1: Class hierarchy for StreamingInputFormat.

3.2.2 Input format

The initial idea was to pass whole events to the map function. Therefore some time was spend on developing an algorithm that is able to serialize a whole event into a string and deserialize the string again to an event. This solution worked, but it turned out, that this way of handling the data was too time consuming and it was discarded. The second approach, which has then been adopted uses simply the input file name as the key and the index of event as a value for the mapper.
The input format for a HadoopStreaming job can be defined by implementing certain Java classes. Figure 3.1 shows the class hierarchy for an input format definition. The class `InputFormat` defines how the input is divided into `InputSplits` and returns them by a call to its `getSplits()` method. The class `FileInputFormat` is Hadoop's base class for file based input formats. Each `InputSplit` defines the input that is processed by a single map job. It provides the information on which nodes of the cluster the data can be found (method `getLocations()`). Each mapper uses a `RecordReader` class which knows how the `InputSplit` needs to be read. The `RecordReader` offers the method `next()`, which is called by the Hadoop framework to get the next key/value pair. A `RecordReader` can also implement the `getProgress()` method, which reports the progress through the `InputSplit`, which is processed by the `RecordReader`. The method `isSplittable()` offered by the `FileInputFormat` returns whether an input file is splittable or whether it should rather be processed as a whole by a single mapper. Note that the implementation as it is described here uses a deprecated API to maintain compatibility with Hadoop Pipes. The implementation according to the new API is however similar.
In our scenario the need for the implementation of the `InputFormat` in Java implies certain problems: The ROOT framework is written in C++ and cannot be used easily by Java classes. To be able to read ROOT files within the input format class (to decide on a split), the Java

library *jhepwork*[1] is therefore used to read files in ROOT format. Because this library can not read ROOT files from HDFS directly, but only using the ordinary POSIX filesystem API, HDFS was mounted with the help of *fuse*[2] to appear as a normal Linux file system that can be accessed using ordinary file operations. Reading data from HDFS with the help of fuse is 20% to 30% slower than reading it directly using the HDFS API, so this is only done in the input format classes for deciding the splits. The actual processing of data by the map jobs reads directly from HDFS using the ROOT framework that allows access to ROOT format files stored in HDFS[3].

3.2.3 The mapper

Given the C++ code of the original LHCb data analysis the implementation of the mapper for Hadoop Streaming is relatively straight forward. The looping through the events is transformed by reading the standard input line by line, where each line provides the information from which file an event should be read and the index of the event. The map function used then ROOT to read the event of the given index.[4] The mapper loops through all particle tracks in one event, calculates the particle mass and if the mass is in a certain range, writes the value to the standard output, from which it is read by the reducer.

3.2.4 The reducer

The reducer in our sample implementation is only used to collect the data from different mappers and write it to a single output file. In general the reducer could be used for further data processing or to produce representative plots of the data obtained (e.g. histograms).

[1] http://jwork.org/jhepwork/

[2] http://fuse.sourceforge.net/

[3] Mappers are implemented in C++ and can therefore utilize ROOT to read files in ROOT format – in contrast to the Java input format classes that do the splitting of the input data.

[4] Note that it is the responsibility of the ROOT framework – and thus out of our control – how the ROOT input file is traversed to move to the event of the given index, e.g. whether the file needs to be traversed sequentially byte by byte from the beginning or whether it is possible to skip irrelevant bytes and move immediately to the desired event. Therefore, the input data is read multiple times in the worst case: first sequentially by the input format class (using jhepwork) to decide on the split, and then again sequentially by the map function (using ROOT) to reach the desired event of the given index. Only after this, the actual data to be processed can be read. At least, the file accesses initiated by the map functions are typicall fast local reads because a map function is preferably started on a node where the data is locally stored.

4 Deploying a Hadoop cluster on AWS

This chapter focuses on the deployment of a Hadoop cluster on top of the Amazon Web Services for Cloud computing. To build a cluster, we need to customize the AMI that should be used as a base for our cluster nodes. Conveniently Hadoop comes with a bunch of scripts that can be used to deploy a Hadoop cluster on EC2[1]. The solution presented in this chapter was build on top of these scripts.

4.1 Remarks on Elastic Map Reduce

Initially, Amazon Elastic Map Reduce (EMR) was considered to deploy a Hadoop cluster in the Cloud. After several trials it turned out that its API does not provide all the functionality that is needed for the project: the Hadoop versions used in EMR are not state of the art and installing all the required software by Bootstrap Actions (see section 2.4.4) is not very handy – for the project, a non-trivial installation of the ROOT framework and all its dependencies is needed on the cluster nodes. Therefore, the Hadoop cluster was deployed on top of EC2 manually as described in the following.

4.2 Building a customized Amazon Machine Image

To be able to launch nodes for the cluster automatically, we need to customize an AMI to fit our requirements. Each node needs to hold a copy of Hadoop and the ROOT framework. Additionally, the master needs fuse to provide access to HDFS for the Java input format classes (see section 3.2.2). The ROOT framework has dependencies to other libraries[2] which need to get installed. Furthermore, for accessing HDFS, we need to compile and deploy the HDFS C++ library to be able to use the HDFS API. As described in section 2.4.3 we need to use an EBS backed AMI to be able to restart instances and keep their internal storage even when we stop them in between, i.e. to have a persistent HDFS distributed file system that stores our data to be analysed. In the end, the following AMI configuration was chosen:

System:	Fedora 8
Platform:	64 Bit
Kernel version:	2.6.21
Hadoop version:	1.0.0
Root version:	5.32
Storage:	10 GB EBS

Each newly registered AMI on EC2 gets a unique ID which can be used to launch instances of its type. The Hadoop configuration is done by a startup script which is passed to the launching instance via the command line interface.

[1] They can be found in `$HADOOP_HOME/src/contrib/ec2/bin/`, where `$HADOOP_HOME` is the root directory of the Hadoop installation.

[2] http://root.cern.ch/drupal/content/build-prerequisites

4.3 Deploying the cluster

After building a customized AMI, the launch procedure for a cluster is quite convenient with the help of the scripts that come with Hadoop. The scripts are altered so that the newly created instance type is launched. Master and worker nodes are assigned to different Security groups (see section 2.4.3), so that the master node is accessible via `ssh` from the outside and the worker nodes can be accessed via `ssh` from the master node. The launch-up-scripts have been altered in a way that all environment variables and the Hadoop configuration is set as desired. When launching the master node, it automatically starts its `NameNode` and `JobTracker` daemons and worker nodes automatically register themselves at the master node and start their `DataNode` and `TaskTracker` daemons. After launching the cluster, one can log in via `ssh` and use the Hadoop command line interface to submit jobs.

5 Evaluation

For the evaluation of the applicability of MapReduce for HEP analysis, we used a Hadoop cluster built with Amazon EC2. Each node in the cluster is provided by a small virtual machine instance (1.7 GB of memory, 10 GB EBS storage, 1 Amazon EC2 Compute Unit). During this project, we were limited to running 20 EC2 instances at the same time. This is the default limit Amazon puts on EC2 users which can be extended by an application form. For the purpose of our initial experiments a total cluster size of 20 instances was regarded to be sufficient.
We varied the input size from 2×10^5 events (ca. 500 MB) to 2×10^6 events (ca. 5 GB) and the cluster size from 5 to 19 worker nodes. We used one extra instance that served as a master node. To reduce the impact of fluctuations the measurement for each value was repeated three times and the mean value was calculated. We evaluated two different versions of the input format: the first one was not able to utilize the locality information of the data blocks, while the second one did, i.e. doing processing of data on those nodes where the data is locally stored. As a third step, we evaluated the effect of increasing the replication factor in HDFS.

5.1 Input data

The data used in this evaluation is Monte Carlo simulation data comparable to the data used in the LHCb experiment. We had access to a file with 2×10^4 simulated events (ca. 45 MB), which we duplicated several times (as separate HDFS files) to achieve the desired input size.

5.2 Output data

After having transformed the original C++ analysis program into a map and a reduce program, we compared the output created by our MapReduce approach with the output of the original C++ analysis program: the calculated result (histograms of particle masses within a specified interval) was the same for the same test input data. This shows that it is possible to achieve the same functionality using the MapReduce paradigm. In the remainder of this chapter, we will therefore move the focus and discuss the performance of the MapReduce-based approach.

5.3 Overhead introduced by Hadoop framework

Hadoop spawns one Java Virtual Machine for each executed map and reduce job. These Virtual Machines are used to run a Java wrapper, which executes the provided map/reduce executables. Additionally Hadoop introduces overhead by generating and distributing the logical input splits. To evaluate this overhead, we compared the execution time of the implementation using the Hadoop framework to call ROOT with the original analysis (just ROOT, no Hadoop involved) on a single node. The input size was set to 2×10^5, 5×10^5, 1×10^6 and 2×10^6 events respectively. Hadoop was running a maximum of three map jobs in parallel on the worker node. The results are shown in figure 5.1. It is obvious that Hadoop introduces overhead to the data processing, but we can see that the overhead decreases relatively with growing input size: 76% of additional execution time when processing 2×10^5

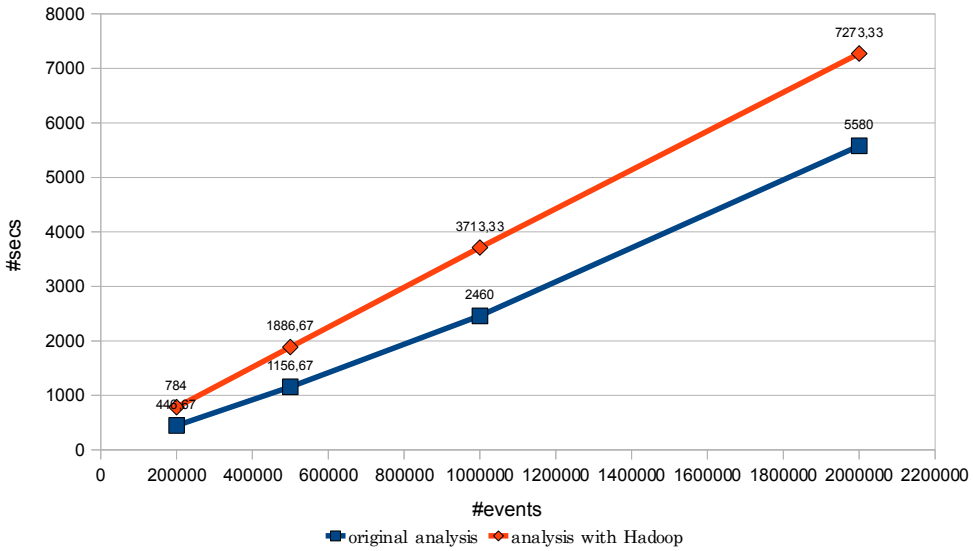

Figure 5.1: Execution time of the original analysis and the Hadoop implementation on a single node.

events compared to 33% overhead with 2×10^6 evaluated events. Since Hadoop is built for the analysis of very large input data, we expect the relative overhead to decrease further with growing input size. As we will see in the following, using the parallel processing approach of Hadoop yields despite the overhead a significant speed-up already with just 5 nodes (which was the smallest cluster size we used for evaluation).

5.4 Scaling the cluster and input size

In this evaluation we used a cluster size with 5, 10, 15 and 19 worker nodes. The input size was set to 2×10^5, 5×10^5, 1×10^6 and 2×10^6 events respectively. For the initial evaluation only one copy of each file was stored on the cluster (HDFS replication factor of 1). Locality information was not taken into account, i.e. to read data, a map job needs to read data from remote HDFS nodes via slow network accesses (local data processing might however have occurred accidentally).
Figure 5.2 shows how long the analysis of different input sizes took on different clusters. The blue plot represents a cluster size with five worker nodes, the red plot shows the outcome on a cluster with 10 worker nodes, the cluster with fifteen worker nodes is colored yellow and the results on a cluster with nineteen worker nodes are represented by the green plot. Each InputSplit assigned to a single mapper instance corresponds to 20.000 events and each worker node was processing a maximum of five mappers at the same time.
We can see that the execution time was growing linearly with the input size. Additionally, we were able to almost halve the execution time by doubling the amount of worker nodes. This result is already very promising.[1] The results can be even improved by taking data

[1]We made initial measurements in a local, non-cloud environment consisting of only one node to compare the overhead added by using Hadoop in comparison to the original C++ analysis program: the Hadoop-based

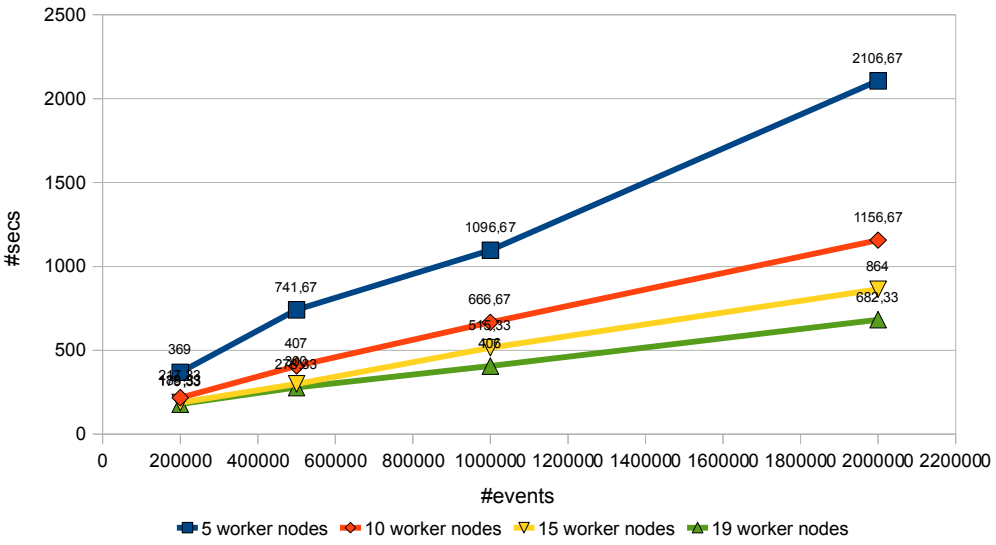

Figure 5.2: Execution time with varying input and cluster size.

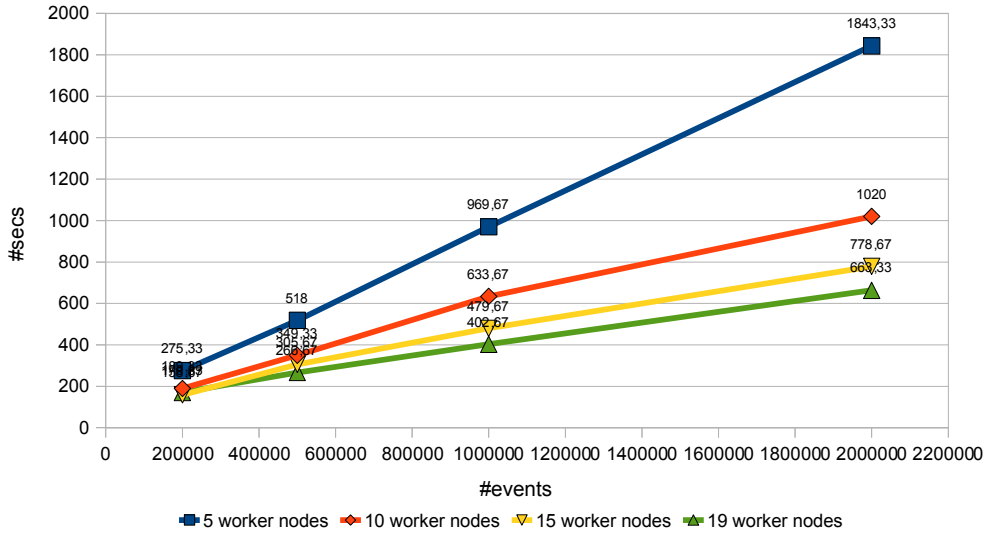

Figure 5.3: Execution time with varying input and cluster size utilizing locality information in input format.

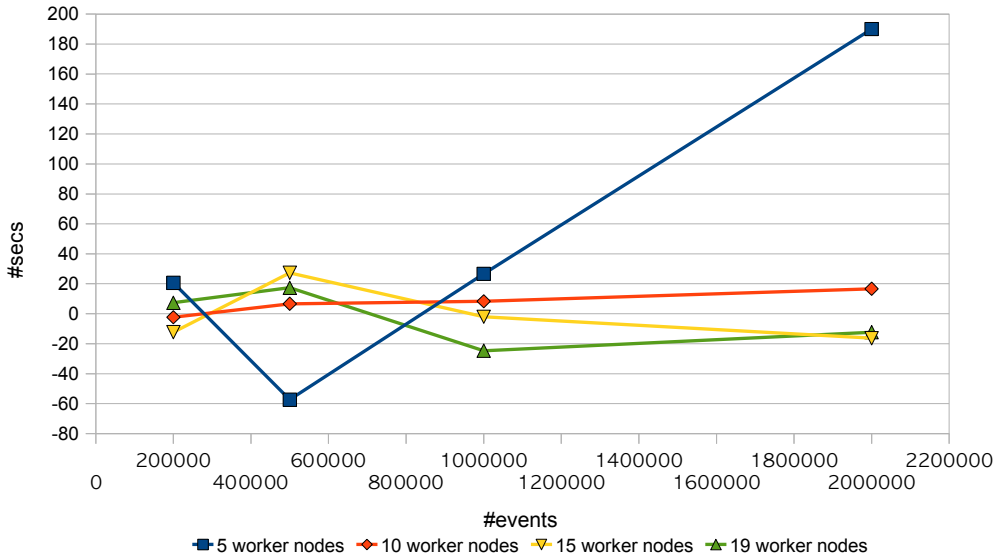

Figure 5.4: Time differences between replication factor = 1 and replication factor = 2.

locality into account as shown in the next section.

5.5 Effect of data locality

In this measurement we repeated the first evaluation, but included locality information in the input format (still using an HDFS replication factor of 1), i.e. map jobs were started on those HDFS nodes that store locally the data to be analysed. Figure 5.3 shows the outcome of the measurements. We can see that the total execution time dropped in all measurements when comparing to the first evaluation. In fact we saved up to 25% of execution time by including locality information of the data.

5.6 Effect of replication factor

While we were using an HDFS replication factor of 1 in the first two evaluations, we increased the HDFS replication factor to 2 in the third experiment. The differences to the outcome of the experiment shown in figure 5.3 are presented in figure 5.4: A negative value means that the third evaluation was running slower, while a positive value indicates that the increased replication factor led to a speed up. We can see that a higher replication factor does not have a positive effect on the execution speed in general (we explain this by the fact that in the bigger clusters, i.e. 10 worker nodes and more, not enough map jobs were available to saturate the worker nodes in a way that a second worker that holds a replica of the data could increase the degree of parallel execution).
We have no explanation why the execution speed even slows down (e.g. 5 worker nodes

analysis was approximately 2.6 times slower than the original C++ analysis program. Due to the fact that our implementation scales so well, we can be sure that the scalability outweighs easily the involved overhead as soon as we move to a cluster consisting of a sufficient number of nodes.

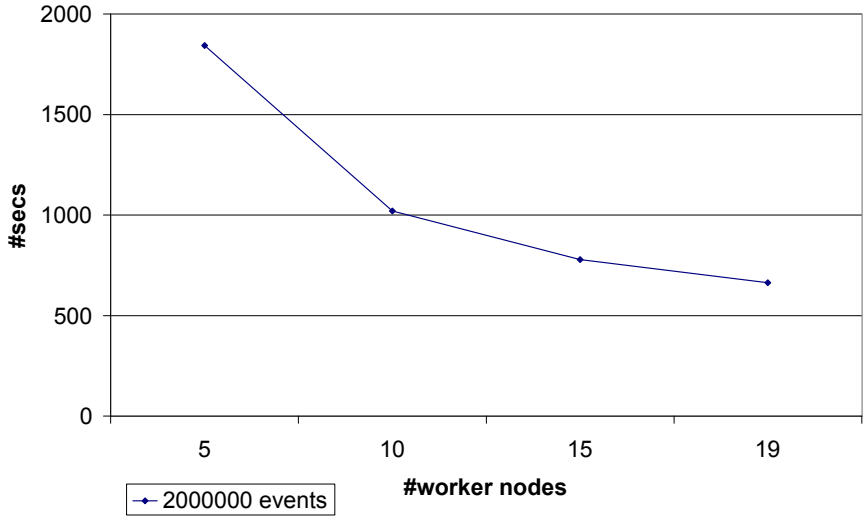

Figure 5.5: Speed-up of computation due to parallel processing

processing 50.000 events) with a higher replication factor. It might be connected to traffic that is incurring in the EC2 internally and cannot be controlled by the user.
We got a significant speed up for the biggest input size on the smallest cluster: a bigger input size leads to a higher number of map jobs assigned to each worker node and it becomes more likely that a worker node needs to work on data, which is not locally available. Here the higher replication factor reduces the number of map jobs that work on non-local data and therefore leads to a speed up.

5.7 Speed-up of parallel processing

Figure 5.5 shows how the speed-up due to parallel processing develops over the number of used worker nodes (problem size: 2×10^6 events). While computation time decreases almost by a factor of two when going from 5 to 10 nodes, the speed-up with a higher number of nodes is not that high anymore. This can either be explained by increased overheads or simply by the fact that even if all map jobs run in parallel, the overall duration cannot be shorter than the time of the slowest map job.
While it might be suspected that a saturation occurs if more than 19 worker nodes are used, it has to be noted that the fastest MapReduce execution occurs if as many worker nodes as splits are used, i.e. all splits are processed in parallel by the map jobs. In the evaluation, only a relatively small input size (and thus a small number of map jobs) is used: it can be expected that for a much bigger problem size, a higher number of worker nodes can be reasonably utilized before a saturation occurs.

In summary, it can be concluded that using Hadoop to execute multiple ROOT jobs in parallel on a cluster enabled us to achieve a much faster data analysis than using just

ROOT without parallel processing: We started our evaluation in section 5.3 with using just a single node to have a baseline for comparison. There, 5580 seconds were needed to process 2×10^6 events using ROOT without Hadoop and due to overhead 7273 seconds using ROOT with Hadoop on a single node (see figure 5.1). Using ROOT with Hadoop in a cluster lead to significant shorter execution times ranging from 1843 seconds (5 worker nodes) to 663 seconds (19 worker nodes): this corresponds to a speed-up factor ranging from 3.0 (5 worker nodes) to 8.4 (19 worker nodes) in comparison to plain ROOT.

6 Conclusions and future directions

The previous chapter presented the execution time evaluations of performing HEP analyses using the MapReduce paradigm by running the analyses using Apache Hadoop in the Amazon Cloud computing environment. This chapter summarizes the results and makes some remarks on possible future directions.

6.1 Results

The results show that:

- the analysis of the data from LHCb can be highly parallelized with MapReduce and Hadoop.
- we can utilize the Amazon Web Services for Cloud computing in order to deploy a Hadoop cluster to analyse HEP data.
- we can save computation time by distributing the data over the cluster in a clever way by making use of locality. The chance of being able to do data processing locally increases when replication is used (in particular if significantly more data splits then worker nodes are available).
- the cluster size can be scaled easily: computation time scales linear to problem size. With respect to the number of worker nodes, computation time does not scale linear, but decreases with the number of nodes. Using a maximum number of 19 worker nodes, a saturation can only be suspected for a small problem size. This is promising and in contrast to other work that shows a saturation already with 10 or 12 nodes with a big problem size [5].

6.2 Future directions

While this document shows that it is possible to analyze data from the LHCb experiment with help of the MapReduce programming model and Hadoop, it opens up a lot of new questions and future directions that can be worked on. A few ideas are listed below.

6.2.1 Development of a full-featured Hadoop input format for ROOT files

Because the input file was relatively small in size (ca. 45 MB) it was copied multiple times to obtain more input data as needed for the scalability evaluation. However, the duplication of the input data was not achieved by concatenating the data into one huge file, but by creating simply multiple copies of the initial 45 MB file. Each file is stored in HDFS and since the file size (45 MB) is smaller than the HDFS block size (64 MB), each file fits into one HDFS block. Therefore, a split of the input data that is generated by the input format class never crosses the border of an HDFS block. The current input format class is able to split up a data file into logical file splits with a desired number of events per split and also provide its location in HDFS. The location in HDFS is defined by the location of the file itself, which is stored as a whole on one or more data nodes in HDFS. For the mentioned reasons, it was never considered (and is thus still an open question) how to handle the splitting of large ROOT files in HDFS. The splitting into blocks is done sequentially and does not take the

internal structure of the ROOT file into account. Due to the structure of ROOT files and due to its internal data compression, this may result in some data dependencies between two or more HDFS data blocks stored on different data nodes in the cluster. Obviously this is not desirable, because single map jobs should be able to work on data which is stored entirely local. Related to this problem is the fact that the same data may be read twice: first by the input format class to find a good location for a split, then by map job to navigate to the start of the split. A carefully designed input format could solve this problem and make use of the full potential of the MapReduce paradigm. A further assessment of the the full potential of the MapReduce paradigm would also include an evaluation on a very big input size (hundreds of terabytes).

6.2.2 Development of a WebService that offers analysis of ROOT files with help of Hadoop

We have shown that we can utilize the Amazon Web Services to deploy a Hadoop cluster which can be used to run analysis of ROOT files. To make this procedure even more convenient to use, a WebService could be implemented, that takes the input data (ROOT files) and the algorithm for analysis in a specified format as input and uses EC2 or a local Hadoop cluster internally to analyse the data.

6.2.3 Deployment and study of physical Hadoop cluster for LHCb experiment analysis

We have seen that we can deploy a Hadoop cluster on top of the Amazon Elastic Cloud. No information is provided by Amazon on how these virtual machines are launched in the data centers itself (in particular whether the instance storage is really local to a launched instance as required by Hadoop to ensure local data processing). It would be interesting to study the performance on a non-virtualized Hadoop cluster and compare it to the performance in the cloud. A local cluster also offers more control on where the data is physically stored and how the nodes in the cluster are connected.

6.2.4 Comparative studies of different parallelization approaches and frameworks

As mentioned in section 2.1, PROOF provides a parallel version of ROOT. This framework provides another approach to parallelize the analysis of the LHCb data. There are also other frameworks available that implement the MapReduce programming paradigm. For example Ekanayake et al. [4, 5] claim that Hadoop is not suitable for scientific computing and introduce *Twister* and alternative runtime environment for the MapReduce paradigm. All these different approaches could be deployed and compared either on a local cluster or on EC2.

Acknowledgments

The Cloud computing resources have been provided by Amazon.

Prof. Dr. Jens Grabowski, head of the Software Engineering for Distributed Systems group at Fabian Glaser's home university, the University of Göttingen, Germany, has initiated the collaboration between Fabian Glaser and Helmut Neukirchen. Furthermore, he took care that Fabian Glaser's travel to Heidelberg got funded. M.Sc. Thomas Rings from the Software Engineering for Distributed Systems group gave initial advice on using Hadoop in the Amazon cloud.

We have to thank Prof. Dr. Michael Schmelling and Dr. Markward Britsch from the LHCb group at the Max Planck Institute for Nuclear Physics (MPIK) in Heidelberg, Germany, for providing the case study: they spent valuable time with us in fruitful discussions and introduced us into the world of HEP data analysis. They prepared for us artificial test data and Markward Britsch created and shared with us the sample C++ analysis program that is underlying the whole work described in this report.

Bibliography

[1] Dhruba Borthakur. *Hadoop Architecture Guide*, March 2012.

[2] CERN. Root: A data analysis framework. http://root.cern.ch/drupal/, April 2012.

[3] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6*, page 10, Berkeley, CA, USA, 2004. USENIX Association.

[4] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 810–818, New York, NY, USA, 2010. ACM.

[5] Jaliya Ekanayake, Shrideep Pallickara, and Geoffrey Fox. MapReduce for Data Intensive Scientific Analyses. In *Proceedings of the 2008 Fourth IEEE International Conference on eScience*, ESCIENCE '08, pages 277–284, Washington, DC, USA, 2008. IEEE Computer Society.

[6] The Apache Software Foundation. Apache Hadoop. http://hadoop.apache.org/, March 2012.

[7] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. *SIGOPS Oper. Syst. Rev.*, 37:29–43, October 2003.

[8] Michael Schmelling, Markward Britsch, Nikolai Gagunashvili, Hans Kristjan Gudmundsson, Helmut Neukirchen, and Nicola Whitehead. RAVEN - Boosting Data Analysis for the LHC Experiments. In Kristján Jónasson, editor, *PARA (2)*, volume 7134 of *Lecture Notes in Computer Science*, pages 206–214. Springer, 2010.

[9] The ROOT team. *ROOT User Guide 5.26*, December 2009.