Using MapReduce for High Energy Physics Data Analysis

Fabian Glaser*, Helmut Neukirchen†, Thomas Rings* and Jens Grabowski*
* *Institute of Computer Science, University of Göttingen, Germany*
Email: {fglaser, rings, grabowski}@cs.uni-goettingen.de
† *Faculty of Industrial Engineering, Mechanical Engineering and Computer Science, University of Iceland*
Email: helmut@hi.is

***Abstract*—At the *Large Hadron Collider* (LHC) *High Energy Physics* (HEP) experiment at CERN, 15 PB of raw data is recorded per year. As it was considered inconvenient to store, access and process this data using the traditional hardware and software tools, this data gets reduced to 10–200 TB per year. This paper investigates the applicability of the MapReduce paradigm for analyzing HEP data. In a case study, a sample HEP analysis that makes use of the HEP analysis framework *ROOT* has been re-implemented using the MapReduce implementation Apache Hadoop. In addition, a Hadoop input format has been developed that takes storage locality of the ROOT file format into account. This approach was evaluated in a cloud computing environment and compared to data analysis with the *Parallel ROOT Facility* (PROOF).**

***Keywords*-MapReduce, Hadoop, Input format, ROOT, PROOF, High Energy Physics, Cloud computing**

I. Introduction

High Energy Physics (HEP) experiments generate huge amounts of data that need to be analyzed. For example, the events measured at the *Large Hadron Collider* (LHC) experiment at *Conseil Européen pour la Recherche Nucléaire* (CERN) create 15 PB of raw data annually from which the events to be analyzed get reconstructed [1]. In practice, a physicist works on event data that is further reduced to about 10–200 TB per year [2]. Physicists complain that even with this reduced data, HEP analysis tools take long to analyze the data or that it would be desirable to work on the non-reduced datasets.

The MapReduce [3] paradigm promises a speed-up of processing big data provided that processing can take place locally where the data is stored in a distributed file system that is spanning the nodes of a cluster. HEP analyses can be typically parallelized on the event-level and are thus local with respect to event-data. Therefore, it is worthwhile to investigate the applicability of MapReduce. To this aim, a sample HEP analysis that makes use of the *ROOT* HEP analysis framework [4] has been rewritten to be usable within the context of the MapReduce implementation Apache Hadoop [5]. ROOT uses its own file format for storing event data. To be able to move processing to the nodes where event data is stored, Hadoop needs to know where particular data of a current event is stored. To provide this information, a so called *input format* has been developed for the ROOT file format. To evaluate the correctness and the performance and scalability of our approach and implementation, we apply it with varying cluster sizes and compare it with a distributed implementation based on PROOF [2], the *Parallel ROOT Facility*. We use cloud computing to provide the cluster resources.

This paper is structured as follows: subsequent to this introduction, we provide foundations. Afterwards, in Section III, we describe the sample HEP analysis that we use in our MapReduce-based approach. In Section IV, we present our ROOT-specific input format for Hadoop and the MapReduce implementation of the sample HEP analysis. Results from running our implementation in a cloud-based cluster are given in Section V and subsequently discussed in Section VI. Related work is surveyed in Section VII, before we conclude with a summary and an outlook in Section VIII.

II. Foundations

In this section, we provide an overview on HEP analysis, on the *Worldwide LHC Computing Grid* (WLCG), on the ROOT framework, and on the *Parallel ROOT Facility* (PROOF) and the *Structured Cluster Architecture for Low Latency Access* (SCALLA) filesystem. Furthermore, we introduce MapReduce and the Apache Hadoop implementation together with the *Hadoop File System* (HDFS). Finally, we cover the principles of cloud computing.

A. High Energy Physics Data Analysis

In *High Energy Physics* (HEP) experiments, such as the *Large Hadron Collider* (LHC) [6] at CERN, the collisions of particles are observed using detectors. Each of the LHC experiments (the four big experiments are ATLAS, CMS, ALICE, and LHCb) uses a detector that fits the purpose of the particular experiment. The raw data produced by a detector is used to reconstruct data about *events*, such as a single bunch crossing of two proton beams.

In, for example, the LHCb [7] experiment, the typical amount of data per event is approximately 50 KB. With 2×10^{10} events per year, the annual data volume amounts to $\mathcal{O}(1)$ PB—the other LHC experiments produce even more data. All events are independent from each other, thus the analysis of the reconstructed data is embarrassingly parallel. Basically, an analysis includes two steps: first, the data set is scrutinized for events containing a specific signature. In the second step, events with this signature are analyzed in detail. The quantities of interest are probabilities or probability density functions for a certain HEP process or configuration to occur: numerical estimates are obtained by means of histograms—simple counters for how often a certain condition is observed [8].

B. The Worldwide LHC Computing Grid

In the case of the LHC experiments, the data is stored and processed using a computing grid with a hierarchical architecture, the *Worldwide LHC Computing Grid* (WLCG) [1]. The raw data produced at LHC is stored (and preprocessed) at the Tier-0 center at CERN. Distributed copies are sent to Tier-1 centers around the globe. The Tier-1 centers redistribute reduced data to Tier-2 centers. Tier-2 centers are meant to provide capabilities for end-user analysis and Monte Carlo simulations. The local workstations and smaller clusters of a department can be considered Tier-3 level. In our work, we focus on problems that arise at Tier-2/Tier-3 level. Typically, batch processing takes place within the WLCG.

C. ROOT, PROOF and SCALLA

ROOT [4] is a C++ framework and library developed at CERN. It provides powerful functionality for analyzing and visualizing data. ROOT defines its own data structures for storing data in main memory and in files. For implementing HEP analyses, a physicist typically writes a C++ program that uses ROOT functionality to access and process data stored in ROOT file format.

The *Parallel ROOT Facility* (PROOF) [2] is used to run analyses of ROOT files in parallel on computing clusters: work is broken down by a master node into packets that are processed in parallel by the worker nodes of a cluster. A packet can be as small as the basic independent unit of embarrassingly parallel processing: the HEP event.

While ROOT and PROOF are independent from a particular file system that is used to store the ROOT files, the *Structured Cluster Architecture for Low Latency Access* (SCALLA) file system [9] is often used together with PROOF. SCALLA allows to store files in a distributed manner: each node of a cluster stores different files of the SCALLA filesystem on its local hard disk thus enabling huge file systems that store huge amounts of data. On one hand, the distribution is handled transparently, thus a user does not need to know on which node a file is physically stored. On the other hand, PROOF can exploit the distribution information to assign a packet to that worker node that has the relevant file locally available. Only if that node is busy, another node gets the work assigned which needs then to fetch the data from the busy node via network access.

D. MapReduce, Hadoop and HDFS

MapReduce [3] is a general purpose paradigm for parallel data processing in clusters of commodity PCs. It is based on the assumption that a distributed filesystem is deployed in a cluster and thus, the data to be processed is stored in a distributed manner. In this setup, network traffic can be minimized by moving the processing of data to the nodes where the data to be processed is actually stored—or, if the current node is already busy processing, at least to a node that is close to the storage node in terms of network distance. Processing takes place in two consecutive steps, a *map* and a *reduce* step. The map step works preferably on local data (to achieve a speed-up) and produces intermediary results that are stored locally. These intermediary results are locally partitioned into as many partitions as reduce subtasks will be later executed. The reduce step collects a partition of the intermediate results remotely from all the worker nodes, condenses the partition, and writes the reduced result as file to the distributed filesystem. Since the map steps are independent from each other, they can be executed embarrassingly parallel on the nodes of the cluster. As soon as all map subtasks finish, the intermediate results are stable and the reduce subtasks are executed.

The map and reduce functions have to be user defined, both take a key/value pair as input. The map function takes as input a key/value pair generated from the input data and emits one or more key/value pairs as intermediate result. To achieve locality of processing, the key/value pair used as input for the mapper needs to refer to a split of the data that is locally available. To enable this, MapReduce needs to be aware of the file format that is processed and needs to know, which node from the distributed filesystem stores the relevant chunk of data. To support partitioning of the intermediate result as needed by the reduce step, the MapReduce framework automatically sorts the intermediate result based on the key. The reduce function takes as input a key and a list of all the values for that key from the intermediate result. This list of values is typically reduced to a smaller list of values that is emitted as value for the current key.

Apache Hadoop [5] is a popular open-source framework that implements the MapReduce paradigm. Hadoop is implemented in Java and thus, the map and reduce functions are preferably implemented in Java.

MapReduce assumes a distributed filesystem. Hadoop comes with the *Hadoop File System* (HDFS) that stores data distributed over the nodes of a cluster. Just like with SCALLA, the distribution is transparent to the end user. In contrast to SCALLA, HDFS splits individual files into blocks of equal size (typically 64 MB). This allows huge files spanning multiple nodes. Furthermore, copies of each block are stored on different nodes in the cluster thus enabling fault-tolerance: if a node fails, a remaining replica is automatically copied to another node to take care that a pre-defined replication factor is adhered to. This enables also loadbalancing of map subtasks as there exist multiple nodes where the same data is locally available.

HDFS can be used outside the context of MapReduce. For example, it is possible to access HDFS files from ROOT or PROOF via an HDFS plugin for ROOT.

E. Cloud Computing

Cloud computing is an approach that gives the ability to scale an IT infrastructure up and down by only using and paying for just as many resources as currently needed (“elastic” and “pay-per-use”) [10].

In the *Infrastructure as a Service* (IaaS) cloud computing service model [10], the computational resources are typically provided by *Virtual Machine*s (VMs). A popular IaaS cloud provider is the Amazon *Elastic Compute Cloud* (EC2) [11]. EC2 provides only VMs. Therefore, it used together with further services of the *Amazon Web Services* (AWS) family, for example for storage. Each EC2 VM instance comes with an *instance storage* (Amazon states that it is based on disks that are physically attached to the host computer). However, instance storage is not persistent. In contrast, the *Elastic Block Store* (EBS) provides persistent block-based storage that itself is independent from a VM, but can be attached to a VM so that it appears like a local storage. EBS does not affect the network traffic of the (virtualized) network of a VM.

An IaaS cloud is attractive because it is easy to install and configure own software (such as Hadoop and HDFS): in contrast to a traditional cluster, an IaaS user is superuser of each VM instance. Furthermore, it is possible to create arbitrary cluster sizes (only limited by the number of currently available VM instances). However, an external cloud provider charges for usage.

III. Sample High Energy Physics analysis

To investigate the applicability of MapReduce to HEP data analysis, we use an example analysis that represents many of the tasks conducted at WLCG Tier-2/Tier-3 facilities. The example ROOT-based C++ analysis has been developed by researchers from the *Max-Planck-Institut für Kernphysik* (MPIK) in Heidelberg. They also provided a toy Monte Carlo event generation program that uses the PYTHIA-8 [12] event generator. The output data of this simulation contains the traces of the involved particles for each event as they are recorded by a detector. These traces are called *tracks* and contain additional information, such as the charge and momentum of the traced particles.

The provided example HEP analysis processes the event data and counts the appearance of a specific particle. This particle is identified by its decay into two further particles: a positively charged one and a negatively charged one. The tracks of these two particles is used to determine the mass of the specific particle of interest. Therefore, each pair of tracks where one track belongs to a positively charged particle and the other belongs to a negatively charged particle is considered. For each of these pairs, the *point of closest approach* is calculated. If this distance is small enough, the point of closest approach is considered to be the point where the particle of interest decayed into the two further particles. The outcome of the analysis is a histogram that depicts the reconstructed particle masses in a certain range.

IV. Using Hadoop for High Energy Physics analysis

Since the events are analyzed independently, we can speed-up the analysis by processing them in parallel. Each event is checked whether it contains particles with certain characteristics (in the example analysis: a certain mass), intermediate results for the matching particles are produced and finally merged. This can be formulated in a MapReduce manner: events are passed to map functions, which perform the event-level analyses and produce new key/value pairs (in the example analysis: representing the mass of a matching particle), which are passed to reduce functions that do the statistical analyses (in the example analysis: producing histograms).

Since we process multiple input files and the events are numbered inside the file scope, we provide this information as input key/value pairs to the mapper as follows: $<\text{path to event file}, \text{event number in file}>$.

The intermediate key/value pairs need to be defined in a way that they represent the characteristics of interest. In the example analysis, we search for particles with a certain mass. Therefore, the intermediate key/value pairs are: $<\text{mass of the particle}, \text{number of the observed particles with that mass}>$. These are fed into the reduce phase, where a histogram is created using ROOT.

A. The Physical Structure of the ROOT Input Data

To distribute processing of data stored in a ROOT-based file format using Hadoop, we need to consider its low-level structure. ROOT uses a complex binary file format to stream data objects to disk. Internally, a tree structure where member variables of classes are split into different branches is utilized. Variables inside the same branch are written to fixed-size buffers, which are compressed and written to disk when their capacity is reached. Since member variables can be of complex compound types themselves, subbranches with associated output buffers can be created. The depth, to which subbranches are created can be controlled when a branch is initialized.

Fig. 1 shows the distribution of events in two ROOT files. Each file contains 2×10^4 events and has a total size of about 50 MB. We define the *byte position* as the offset of a byte from the beginning of the file. Using the byte position, we calculate metrics for the minimum, maximum, median, and average of the byte positions of all bytes belonging to each respective single event.

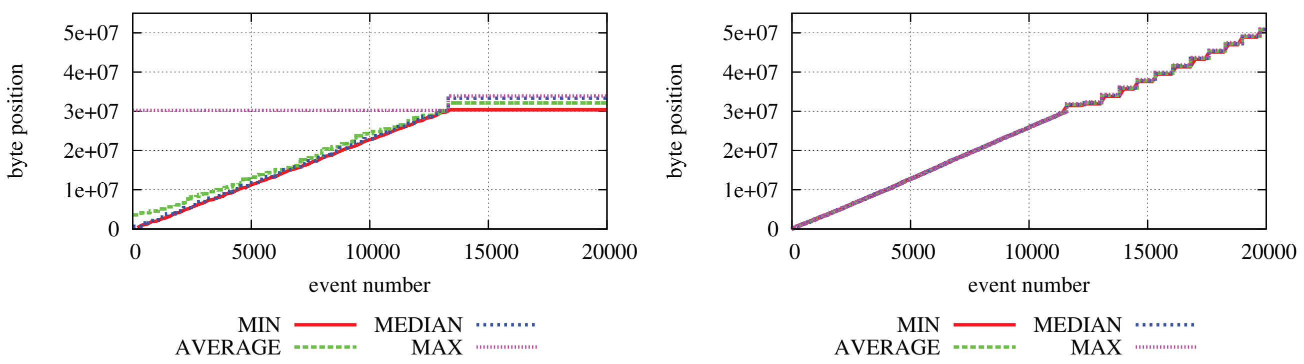
The file in Fig. 1a is created with a maximum number of subbranches, the file in Fig. 1b with no subbranches. The most important observations are:

- The higher the event number, the later in the file the per-event data is stored.
- Even if the data belonging to one event is not stored continuously inside a file, the event-associated data is clustered around certain byte positions.

We use these observations to define how the processing of input data is distributed by the Hadoop framework.

B. Hadoop Input Format

In Hadoop, the part of data that is processed by a single mapper is defined as an *input split*. Input splits are created by subclasses of the Hadoop class `InputFormat`. It is responsible for cutting the physical data at logical boundaries. For example, an input format for text files splits at line-breaks. Depending on the duration of a map



(a) With maximum number of subbranches

(b) With no subbranches

Figure 1. Per-event data distribution in an example file with 2×10^4 events.

subtask, an input split should either match an HDFS block or partition an HDFS block into multiple input splits. Due to the big HDFS block size, an input split contains typically data that is turned into several key/value pairs as map input (such as multiple lines of text with each line giving one key/value pair).

Typically, the input data for a ROOT analysis consists of several files. Therefore, splitting the input data on a file-level and representing each file by a single input split is possible, but has considerable drawbacks:

1) We have no control over the granularity of the subtasks assigned to each mapper. If we work with big input files (files involved in a single HEP analysis can be $\mathcal{O}(1)$ TB each), the subtasks assigned to the mappers take a long time to complete. This results in situations where analysis completion is delayed by a single, huge map subtask.
2) If the number of input files is smaller than the total number of worker nodes, some of the nodes do not get any subtasks assigned, which results in idle nodes and unnecessary low parallelity of processing.
3) Hadoop prefers to schedule subtasks to worker nodes where the corresponding input data is stored locally. This means that input splits should refer to data that is stored on a single node. If the input file size and the corresponding input split are larger than the block size in HDFS, then local storage is not exploited anymore.

Therefore, we need to define how a ROOT file is divided into several input splits that are efficiently processed by a Hadoop system. Due to the physical structure of the ROOT files, there is a good chance that all the data or at least most of the data belonging to a single event is stored on a single block in HDFS. Thus, we calculate the distribution metrics (minimum, maximum, average, or median byte position of the per-event data) for each event and assign the events to a certain block in HDFS based on the value of the chosen metric. The events assigned to a single block in HDFS then define the input splits that are determined by our ROOT-specific `InputFormat` class.

C. *Implementation of Input Format and Analyses*

Input formats are implemented by inheriting from the Hadoop Java class `InputFormat`. To provide suitable input splits, our ROOT-specific input format needs the byte positions of events. To this aim, we implemented a standalone C++ tool that uses ROOT to traverse the input files and to generate all the metrics described at the end of Section IV-B. These metrics are stored by the tool in helper files which we call *event maps*. The tool needs to be executed once the ROOT input files are available or whenever they change. If a Hadoop run is started, the event map files are read by our input format implementation and thus the input splits can be determined accordingly. From the available metrics, we use the average byte position to determine the HDFS block targeted by the input split.

The original analysis is a C++ program that links to the ROOT library to use its functionality. To be able to reuse parts of the original C++ code, we utilize *Hadoop Streaming* which provides the possibility to implement map and reduce functions in other languages than Java. Thereby, the key/value pairs are read line-by-line from standard input. The per-event data analysis and calculation of the particle mass is implemented in the map function, while the reduce function collects the mass values from the different mappers and creates the final histogram.

To compare our implementation with a PROOF-based analysis, we implemented the same analysis for PROOF. In PROOF, parallelizing a ROOT analysis that reads data from ROOT files is done with the *Selector* framework. A C++ code skeleton is automatically generated and the data stored in specified tree-branches is automatically split for parallel processing. The implementation then comes down to calls of the original analysis code at the right place.

V. Evaluation

In this section, we evaluate our solutions. We describe the clusters that we deployed using Amazon's EC2 and outline the generation of the data used for the evaluation. Finally, we evaluate our input format and compare the Hadoop-based analysis with the PROOF-based analysis.

Table I
HARDWARE CONFIGURATION OF CLUSTER NODES.

Instance type:	m1.medium
CPU:	2 *EC2 Compute Units* (ECUs)
Architecture:	64-Bit
Boot volume:	EBS
Memory:	3.75 GB RAM
EBS-storage:	100 GB

Table II
SOFTWARE CONFIGURATION OF CLUSTER NODES.

Ubuntu:	12.04 with Linux kernel 3.2.0-27-virtual
SCALLA:	3.2.0
Hadoop:	1.0.4
ROOT/PROOF:	5.34/05
PYTHIA:	8.1
Ganglia:	3.5.2

A. Deployment

We evaluated the performance of EC2 instance storage and EBS by measuring the execution time of the original analysis on a small dataset and were not able to detect a significant speed difference. Hence, we decided to use EBS, since it provides data persistence in case the VMs are stopped and restarted. The hardware configuration of the cluster nodes is given in Table I.

For collecting cluster metrics, we used the Ganglia monitoring system [13]. Hadoop and PROOF use both a master/slave architecture: in our configuration, the master daemons and the Ganglia meta-daemon are deployed on dedicated VMs. The deployed software is listed in Table II. The maximum number of subtasks running on a node in parallel was set to 2 for Hadoop and PROOF. The HDFS replication factor was set to 3.

B. Data Generation

The data generation is a two-step procedure. In the first step, Monte Carlo data is generated with help of the event generator PYTHIA-8 [12]. In the second step, a detector chain is simulated that reconstructs the particle tracks. The output of the data generation is stored in a ROOT-based file format, called *Simple Data Format* (SDF), which is used by the researchers at MPIK.

For the evaluations, we created three different datasets: a small one, a middle sized one, and a big dataset. The small dataset is used for evaluating the input formats and contains a series of differently sized files. It contains single files of different size that are stored in HDFS: 7.5×10^5 events (1.8 GB), 1.875×10^6 events (4.4 GB), 3.75×10^6 events (8.8 GB), and 7.5×10^6 events (18 GB) respectively.

The middle and big datasets contain files with fixed size and are used for comparing the Hadoop-based analysis with the PROOF-based analysis. The number of events per file is 3×10^5. Generating one file takes around 25 minutes on a m1.medium EC2 instance and the individual output file size is 750 MB. Since the HDFS block size is set to 64 MB throughout all our experiments, an individual file occupies 12 HDFS blocks. Each generated file was written as a separate ROOT file to HDFS and SCALLA.

The middle sized dataset contains 250 of such ROOT files that accumulate to 75×10^6 events resulting in 190 GB of data. A cluster with 15 worker nodes took about 8 hours to generate this data. The creation of the event map files on a single EC2 instance took about 1 hour.

The big dataset contains 1500 ROOT files totaling 450×10^6 events occupying 1 TB. The data generation with 120 worker nodes took about 7.5 hours. The event map generation for this dataset was parallelized with the same number of worker nodes using Hadoop Streaming and took 750 seconds to complete.

C. Input Format Evaluation

For the input format evaluation, we compare two different custom input format implementation:

- `StreamingInputFormat`: This class generates input splits with equal size, without taking any locality information into account.
- `ROOTFileInputFormat`: The input splits correspond to blocks in HDFS and are generated according to the approach described in Section IV-B.

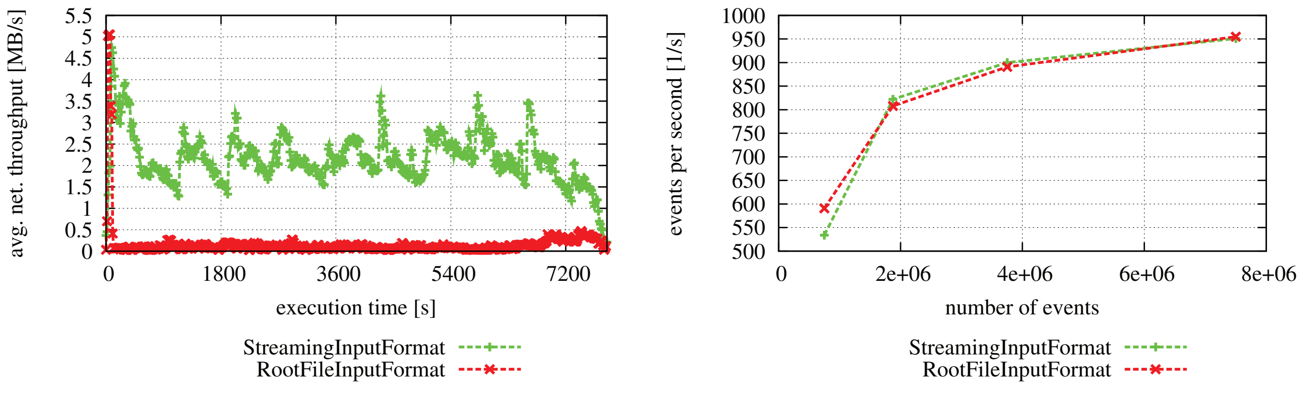
The cluster size was fixed to 15 worker nodes. Using the different sized files from the small dataset, the `StreamingInputFormat` generates 30, 75, 150, and 300 input splits respectively leading to an average number of maps assigned to each worker node of 2, 5, 10, and 20 respectively. We repeated the evaluations two times for each input size to be able to calculate a mean average.

Fig. 2a shows the network throughput during the analysis of the input file with 7.5×10^6 events: The `RootFileInputFormat` causes considerably less network traffic in comparison to the `StreamingInput-Format`. This demonstrates that our approach successfully assigns the map subtasks to that nodes that are responsible for storing the corresponding event data.

A closer investigation of the network utilization within the overall Hadoop workflow reveals the following for the `RootFileInputFormat`: during the initial setup, reading in the event map files causes a high peak in the network traffic. This is due to the fact that these files are stored in HDFS and the Hadoop framework reads them remotely to calculate the input splits. Furthermore, we observe for the `RootFileInputFormat` a small increase in the network utilization towards the end of the analysis when the results are combined by the reducer (starting from 6800 seconds in Fig. 2a). The reason is that the intermediate key/value pairs are read remotely. During the actual HEP analysis (the map subtasks), most of the data is read locally when using the `RootFileInputFormat`.

In contrast, the `StreamingInputFormat` leads to a lot of remote reads to provide the map subtasks with event data. During the reduce phase, no event data is read anymore; only the intermediate key/value pairs are transmitted via the network to the reducer.

Fig. 2b shows analyzed events per second for both input formats: both show a similar performance and the overall performance increases with the input size. This indicates that Hadoop needs a minimum number of maps per worker node to perform efficiently. While the result from Fig. 2a is



(a) Network throughput (b) Analyzed events per second

Figure 2. Performance comparison of the different input formats.

that the `RootFileInputFormat` leads to less network transmission than the `StreamingInputFormat`, it is remarkable that the duration of the overall HEP analysis is almost the same for both input formats. Regarding our example HEP analysis, we conclude therefore that its total performance is rather affected by the computing capabilities than by the I/O performance which does not measurably contribute to the duration.

D. Comparison of Hadoop and PROOF

We ran the Hadoop-based and PROOF data analyses successfully on different cluster sizes and dataset sizes: both implementations yield the same output values as the original non-parallel ROOT analysis. For comparing the performance of the Hadoop-based data analysis with the analysis in the PROOF framework, we used the middle and big datasets.

PROOF was used with two different file systems for reading the input data: SCALLA and HDFS. Using the middle-sized dataset, we started with a cluster size of 15 worker nodes and scaled-up to 30, 60 and 120 nodes. The analysis of the generated events was done with both Hadoop and PROOF and was repeated two times for each cluster size to be able to calculate average means of the monitored cluster metrics. After each scale-up of the cluster size, the data stored in HDFS and SCALLA was rebalanced such that each data node roughly stores the same amount of data.

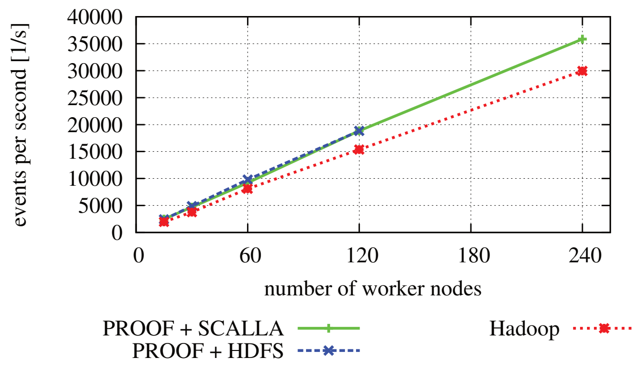
We used the big dataset to evaluate the performance of a Hadoop cluster with 120, 240, and 480 worker nodes. Additionally, we evaluated the performance of PROOF reading the data from SCALLA on a 120 and 240 worker node cluster (480 nodes were not possible for PROOF due to limited funding of cloud resources; for the same reason, we were not able to repeat the measurements for PROOF with HDFS). After each scaling of the cluster, the data stored in HDFS and SCALLA was rebalanced.

Fig. 3a shows the analyzed events per second for the three different analysis software and file system configurations. The values for the smaller cluster sizes (15, 30, 60, and 120 worker nodes) correspond to the analysis of the middle-sized dataset, while the values for the 240 node cluster correspond to the big dataset. The number of the analyzed events per second on a 120 node cluster is similar for both datasets analyzed with that cluster size (Hadoop: 15396 vs. 14666 events per second, PROOF + SCALLA: 18844 vs. 18671 events per second). The result for the 480 worker node Hadoop cluster is omitted in the figure—it is 50664 events per second and follows the same trend as the results for the smaller cluster sizes.

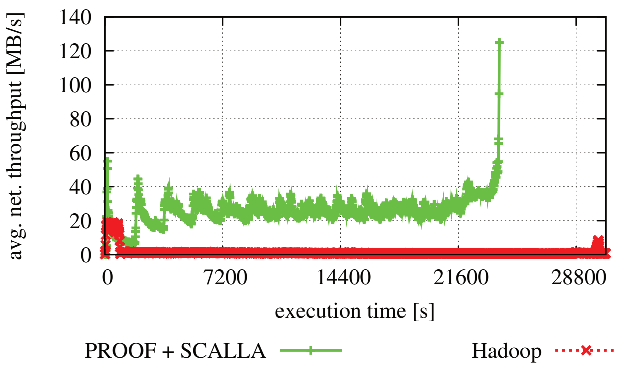
Our results show that Hadoop introduces overhead to the calculations. Comparing its performance to the PROOF-based solutions, we identify a deceleration by 14%–29%. When using PROOF, the performance differences between the two underlying file systems are negligible.

We observe that the number of analyzed events per second almost doubles when we double the cluster size. It can thus be said that all three solutions *scale well* on the investigated cluster sizes. Nevertheless, when scaling the Hadoop cluster size from 240 to 480 worker nodes, the performance did not double, but the gain was only about 70%. We can think of two reasons: First, the setup phase does not benefit from parallelization; the generation of the input splits in Hadoop is done by one of the master daemons and cannot be parallelized within the framework. Secondly, also the reduce step does in our HEP analysis not benefit from parallelization as only one reduce subtask is used. (However, the same applies for the final step in the PROOF approach). Because these sequential parts of the overall workflow do not experience any speedup, they relatively gain weight in the total execution time when scaling the cluster size. This could be an explanation of the reduced performance gain when we scale the cluster size from 240 to 480 nodes. Nevertheless, the achieved performance gain is still satisfying.

Fig. 3b shows the average network throughput on a 120 worker node cluster during the analysis of the middle-sized dataset with Hadoop/HDFS and with PROOF reading data from SCALLA. Hadoop causes some network traffic during the setup phase and at the end when the intermediate data of the map subtasks is retrieved by the reducer. During the analysis by the map subtasks, the data



(a) Analyzed events per second



(b) Network throughput (120 worker nodes)

Figure 3. Performance of analysis in a cluster using PROOF and Hadoop.

is mainly read locally. In contrast, PROOF with SCALLA causes significantly more network traffic, since it is not able to schedule all subtasks to the nodes that are in charge of storing the data. Nevertheless, this is not reflected in the total execution time. This reaffirms that our example analysis rather depends on CPU power than on network performance. Measurements for PROOF with HDFS are not shown as this requires a complete different scale for the plot: the network traffic was significantly larger than for the two other configurations. The reason is that PROOF with HDFS does not take any storage locality into account.

VI. Discussion

The above evaluation shows that our Hadoop input format for ROOT is able to significantly reduce network load by allocating map subtasks to the nodes that are in charge of storing the data to be analyzed. We are not aware of any other MapReduce input format that achieves this for ROOT files.

Furthermore, our evaluation shows that both, Hadoop with our input format and PROOF, scale well when increasing the cluster size. Finally, the evaluation shows that our Hadoop-based implementation is slower than our PROOF-based implementation of the HEP analysis even though PROOF with the SCALLA filesystem uses just a file-based notion of data locality. Our explanation is threefold: first, the sample HEP analysis that we performed is rather CPU-bound, not I/O-bound, and thus, speed-up due reduced network load is negligible in comparison to the duration of the actual HEP analysis. Second, Hadoop is implemented in Java while PROOF is implemented in C++ which is typically faster than Java, and in the case of Hadoop, each subtask on a worker node is started in a new *Java Virtual Machine* (JVM) which adds further overhead. Third, the ROOT-based map and reduce subtasks that are called by Hadoop are implemented in C++ and thus, we used for the involved Java/C++ communication the standard input/output to pass key/value parameters in and out line-by-line (Hadoop Streaming): this includes encoding all values to ASCII and parsing them back which is slow.

Even though the Hadoop-based implementation is slower than the special-purpose PROOF-based implementation, using Hadoop/HDFS has two main advantages: First, it has a bigger user- and developer-base. Thus, it is more likely that bugs are fixed and new features are added. Also, more documentation is available for Hadoop and HDFS than for PROOF and SCALLA. Furthermore, Hadoop and HDFS are installed at many sites and can be used out-of-the box (for example, the Amazon cloud provides a ready-to-use Hadoop service). For the same reasons, it is more likely to find staff that is familiar with Hadoop than with PROOF. Second, Hadoop and HDFS are fault-tolerant. For example, in case of a failing node, HDFS takes care that a configurable replication factor is adhered to by automatically replicating data and Hadoop automatically resubmits a failed map or reduce subtask. In contrast are PROOF-based solutions or simple ROOT batch jobs: with these, physicists suffer from analyses that fail as a whole because of failing nodes or processes.

VII. Related Work

Besides the evaluation of the applicability of MapReduce to scientific computing in general, Ekanayake et al. [14] describe also an adoption of Hadoop/HDFS for HEP data analysis. However, they do not address the problem of splitting the data efficiently using a ROOT-specific input format. Instead, data files are processed as a whole. This leads to the problems described in Section IV-B. They evaluate their solutions on small clusters (up to 12 compute nodes) and claim that for sizes larger than 10 nodes, the speed-up diminishes. This contradicts our results: using our ROOT-specific input format, we are able to achieve satisfying speed-ups even when we scale the cluster up to 480 compute nodes.

Riahi et al. [15] describe how Hadoop/HDFS can be utilized for computing within the WLCG in a small/medium Grid site (Tier-2/Tier-3). The problem of a ROOT-specific input format is not solved. Instead, whole files are assigned to single mappers. Similar to Ekanayake et al., they limit their evaluation to very small cluster sizes (three compute nodes with a total of 72 cores).

HDFS is not limited to Hadoop and can be used as a reliable distributed file system without exploiting the MapReduce paradigm. Bockelman [16] identifies HDFS as a possible storage solution in the context of the CMS experiment. He concludes that HDFS is a viable solution for grid storage elements inside the WLCG, especially with regards to scalability, reliability and manageability. He is also the author of the HDFS plugin for ROOT.

VIII. Summary and Outlook

We investigated the applicability of the MapReduce paradigm for HEP data analysis that is based on the ROOT analysis framework. To this aim, we used the general purpose MapReduce implementation Apache Hadoop/HDFS. For Hadoop, we developed a so called input format that is aware of locality inside the ROOT file format and thus enables Hadoop to assign map workload to the nodes that are responsible for storing the data to be processed.

We evaluated our implementation for different cluster sizes and compared it to traditional ROOT-based data analysis that is parallelized using PROOF. To ease setting up a cluster, we used cloud computing resources.

Our evaluation indicates that Hadoop offers a viable alternative to the traditional methods: Both scale well and while our Hadoop-based solution is slower, it has some advantages: Hadoop is a wide-spread general purpose approach and adds fault tolerance for storage and processing.

For those that want to use PROOF for their analyses, using HDFS to achieve fault tolerance at least with respect to storage is a good choice. It is thus promising to develop an HDFS plug-in for PROOF to make PROOF aware of HDFS with respect to storage locality. To achieve this, the approach from our ROOT input format is applicable.

Concerning performance improvements, it is worthwhile to investigate the performance of a pure C++ MapReduce framework implementation that avoids any Java-related overheads or to use a different big data approach and framework beyond plain MapReduce. Another speed-up might be achieved by using *Graphics Processing Unit*s (GPUs) [17] instead of CPUs.

Last but not least, it needs to be investigated how other HEP analyses than the one we used benefit from applying MapReduce. This includes determining consumed I/O time and CPU time and investigating their ratio.

Acknowledgments

The authors like to thank Michael Schmelling and Markward Britsch from the LHCb group at the *Max-Planck-Institut für Kernphysik* (MPIK) in Heidelberg, Germany, for providing the example for the case study, consisting of a toy Monte Carlo program for event data generation and the ROOT-based sample C++ analysis code used in this paper. The cloud resources were provided by Amazon. The work of Fabian Glaser is partially funded by the Joint Centre of Simulation Technology (SWZ) of the University of Göttingen and the Technical University of Clausthal (project 11.4.1).

References

[1] The LCG TDR Editorial Board, "LHC Computing Grid Technical Design Report," CERN, Tech. Rep. LCG-TDR-001/CERN-LHCC-2005-024, June 2005.

[2] G. Ganis, J. Iwaszkiewicz, and F. Rademakers, "Data Analysis with PROOF," in *Proceedings of XII International Workshop on Advanced Computing and Analysis Techniques in Physics Research*, ser. Proceedings of Science (PoS), no. PoS(ACAT08)007, 2008.

[3] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," in *Proceedings of the 6th Symposium on Operating Systems Design & Implementation.* Berkeley, CA, USA: USENIX Association, 2004.

[4] R. Brun and F. Rademakers, "ROOT - An Object Oriented Data Analysis Framework," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 389, no. 1, pp. 81–86, 1997, see also http://root.cern.ch.

[5] Apache Software Foundation, "Apache Hadoop," [Online; http://hadoop.apache.org/ fetched on 08/24/2013].

[6] L. Evans and P. Bryant (Eds.), "LHC Machine," *Journal of Instrumentation (JINST)*, vol. 3, no. S08001, 2008.

[7] A. Augusto Alves Jr, L. M. Andrade Filho, A. F. Barbosa *et al.*, "The LHCb Detector at the LHC," *Journal of Instrumentation (JINST)*, vol. 3, no. S08005, 2008.

[8] M. Schmelling, M. Britsch, N. Gagunashvili, H. K. Gudmundsson, H. Neukirchen, and N. Whitehead, "RAVEN – Boosting Data Analysis for the LHC Experiments," in *Applied Parallel and Scientific Computing: 10th International Conference, PARA 2010, Reykjavik, Iceland, June 6-9, 2010, Revised Selected Papers, Part II*, ser. Lecture Notes in Computer Science (LNCS), vol. 7134. Springer, 2012.

[9] A. Hanushevsky and D. L. Wang, "Scalla: Structured Cluster Architecture for Low Latency Access," in *IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW).* IEEE, 2012, pp. 1168–1175.

[10] P. Mell and T. Grance, "The NIST definition of Cloud Computing," *NIST special publication*, vol. 800-145, 2011.

[11] Amazon, "Elastic Compute Cloud," [Online; http://aws.amazon.com/ec2/ fetched on 08/24/2013].

[12] T. Sjöstrand, S. Mrenna, and P. Skands, "A brief introduction to PYTHIA 8.1," *Computer Physics Communications*, vol. 178, no. 11, pp. 852–867, 2008.

[13] M. Massie *et al.*, "The Ganglia Monitoring System," [Online;http://ganglia.sourceforge.net/ fetched on 08/24/2013].

[14] J. Ekanayake, S. Pallickara, and G. Fox, "MapReduce for Data Intensive Scientific Analyses," in *Proceedings of the 2008 Fourth IEEE International Conference on eScience*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 277–284.

[15] H. Riahi, G. Donvito, L. Fanò *et al.*, "Using Hadoop File System and MapReduce in a small/medium Grid site," in *Journal of Physics: Conference Series*, vol. 396, no. 042050. IOP Publishing, 2012.

[16] B. Bockelman, "Using Hadoop as a grid storage element," in *Journal of physics: Conference series*, vol. 180, no. 012047. IOP Publishing, 2009.

[17] J. A. Stuart and J. D. Owens, "Multi-GPU MapReduce on GPU Clusters," in *Parallel & Distributed Processing Symposium (IPDPS).* IEEE, 2011, pp. 1068–1079.