Journal of Physics: Conference Series **396** (2012) 042050 doi:10.1088/1742-6596/396/4/042050

Using Hadoop File System and MapReduce in a small/medium Grid site

H Riahi[1], G Donvito[2], L Fanò[1], M Fasi[1], G Marzulli[3], D Spiga[4] and A Valentini[1]

1 INFN Perugia, IT
2 INFN Bari, IT
3 INFN Bari and GARR, IT
4 CERN

E-mail: hassen.riahi@pg.infn.it

Abstract. Data storage and data access represent the key of CPU-intensive and data-intensive high performance Grid computing. Hadoop is an open-source data processing framework that includes fault-tolerant and scalable distributed data processing model and execution environment, named MapReduce, and distributed File System, named Hadoop distributed File System (HDFS).

HDFS was deployed and tested within the Open Science Grid (OSG) middleware stack. Efforts have been taken to integrate HDFS with gLite middleware. We have tested the File System thoroughly in order to understand its scalability and fault-tolerance while dealing with small/medium site environment constraints. To benefit entirely from this File System, we made it working in conjunction with Hadoop Job scheduler to optimize the executions of the local physics analysis workflows. The performance of the analysis jobs which used such architecture seems to be promising, making it useful to follow up in the future.

1. Introduction

The MapReduce[1] is a simple programming model that applies to many large scale computing problems. Typically when reading a lot of data, the Map step extracts from each record data of interest and then the Reduce aggregates the Map output to produce the result. Both Map and Reduce functions operates on key/value pairs. The outline for solving the problems stays the same while the Map and Reduce functions change to fit the problem.

Hadoop[2] is data processing system that follows the MapReduce paradigm for scalable data analysis. It includes a fault-tolerant and scalable execution environment, named MapReduce, and a distributed File System, named Hadoop Distributed File System (HDFS). The largest Hadoop-based cluster is installed at Facebook to manage nearly 31 PB of online disk data [3]. Other companies, such as Yahoo and Last.Fm, are also making use of this technology.

The mission of the WLCG[4] project is to build and maintain a distributed data storage and analysis infrastructure, of 15 Petabytes of data annually generated by the LHC[5], for the entire High Energy Physics (HEP) community around LHC at CERN[6]. To satisfy the computing and storage requirements of LHC experiments, the Grid approach is used to interconnect the regional computing centers. These are organized in Tiers where the medium/small size Grid

sites are dedicated to Monte Carlo event simulation and to end-user analysis. LHC data is stored, analyzed, and visualized using experiment specific frameworks based on ROOT[7].

Data storage and access represent the key of CPU-intensive and data-intensive high performance Grid computing. However, the small/medium size Grid sites are in particular constrained to use often commodity hardware which exposes them to hardware failure.

The final goal of this work is the deployment of Hadoop-based solution for data storage and processing in HEP Grid sites.

2. Advantages in deploying Hadoop in HEP

The deployment of Hadoop-based solution for data storage and processing represents many advantages. The most important characteristics of this solution are:

- Reliability: HDFS allows the deployment of commodity hardware. To deal with unreliable servers, Hadoop uses replication and handles task resubmission on failure.
  The files splitting into HDFS blocks fits well with replication for providing fault tolerance and availability. To ensure against corrupted blocks and disk and machine failure, each block is replicated to a configurable number on physically separated machines (typically three). If a block becomes unavailable, a replica can be read from another location in a transparent way. Hadoop's strategy is to place the first replica on the client node in case that the client is inside the Hadoop's cluster. Otherwise, the first replica is placed randomly trying the best to not pick nodes that are full or too busy. The second and the third replicas are placed on different nodes on a different rack than the first replica.
  When a task execution attempt fails, Hadoop reschedules execution of the task transparently to the user. It will try to avoid rescheduling the task on a node where it has previously failed. The maximum number of attempts is a configuration parameter set by the Hadoop administrator. If a task fails more than the maximum number of attempts, it will not be retried further.
- Scalability: Hadoop MapReduce splits the data computation into fine grained "map" and "reduce" tasks. The splits are then processed in parallel. If the splits are small:
  - the time taken to process each split is small compared to the time to process the whole input;
  - the processing of the input is better load-balanced inside the cluster since, of course, a faster machine will be able to process more splits of the job than a slower machine;
  - the time required to recover the task from failure is fast.

  Furthermore, this solution allows to benefit of Hadoop data locality optimization. Actually, Hadoop tries to schedule the task close to the closest replica of the input, particularly, when it is configured to use the Fair scheduler or the Capacity scheduler.

3. Hadoop-based Grid Storage Element in gLite environment

On the WLCG, the protocol used for data transfers between sites is GridFTP [8] and the protocol used for metadata operations is SRM [9] ; both are necessary for interoperability between site storages in the Grid. To deploy Hadoop as Grid Storage Element (SE) in Open Science Grid (OSG)[10] sites, a plugin has been developed to allow the interaction between GridFTP server and HDFS storage system, and Berkeley Storage Manager (BeStMan) SRM server has been utilized to allow SRM access [11].

For the deployment of Hadoop as SE in gLite[12], the first choice was to use Storm[13] as Grid SRM since it is widely deployed in this environment. But investigating this solution, it was noticed that Storm cannot work with File System not supporting Access Control List (ACL). Since HDFS does not support such control, the BeStMan SRM server was used.

The following steps were required to setup HDFS as a Grid SE in gLite environment:

- setup GridFTP server: HDFS-GridFTP library developed for OSG sites is recompiled in gLite environment and then gLite GridFTP server is started using this library as Data Storage Interface (dsi): globus-gridftp-server -p 2811 -dsi hdfs ;
- setup SRM server: HDFS is mounted using Fuse[14]. Next, BeStMan is installed and configured correctly to be able to authenticate users using Gridmap-file and to manage the experiments storage area in HDFS;
- to allow to read and write into HDFS using ROOT software program, the Xrootd service has been setup as interface to HDFS for ROOT files access. The xrootd-hdfs rpm used in OSG site is installed with –nodeps option to bypass the credential check required for OSG sites;
- publish SE Information to Site Berkeley Database Information Index (BDII)[15] to make them available in the Grid. A provider script is developed to publish dynamic information to gLite Information Service. SRM-PING is called to get required information.

The Hadoop-based SE architecture in gLite environment and the interactions between its components are shown in Fig. 1.

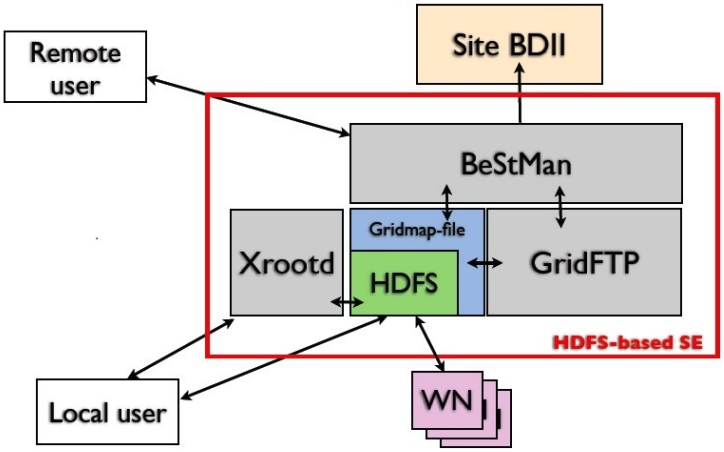

Figure 1. Architecture of Hadoop-based SE in gLite environment and the interactions between the SE components. The interactions of the SE with others gLite Grid components. The services involved in the typical interactive accesses by users to SE.

4. Fault tolerance and scalability tests of HDFS for small/medium size Grid site

One of the main interesting features of Hadoop is the capability to replicate data in order to avoid any data loss both in case of hardware or software problems. The data replication is automatically managed from the system and it is easy to configure the number of the replicas that the system should guarantee. This is important because, the most critical data should be guaranteed with a sufficiently high number of replicas.
All the files hosted in the HDFS File System are split in chunk (the size of the chunk could be configured from the site admin), and each file could be registered in a different disk or host.

The testing activities was focused on how the replication mechanism works and if this could really be a solution for medium-sized computing centers that have to provide an high available data access infrastructure, without the need of buying an expensive and complex hardware solution.
Several different kind of failures were simulated in order to check how HDFS reacts to each kind of problem. In particular, the two main topics were metadata failures and data failures.

Both issues may happen for different reason, so we have tested: temporary network failure, complete disk failures, host failures, and File System corruption.
Starting from metadata failure: from the design it is clear that the NameNode (that is the metadata server) is the single point of failures in the HDFS file-system. A failure on the metadata, could lead to unavailability of files. Looking at the design of the system it is possible to have a Secondary NameNode that could sync the metadata from the primary NameNode on a different disk.

The Secondary NameNode, indeed, will dump the status of the primary every given amount of seconds that could be configured from the site admin. It has been measured that the dump takes less than one second with hundreds of files to be synchronized. It was tested that if the NameNode gets corrupted or its file-system is deleted, it is easy to import the metadata from the backup on the Secondary NameNode. While the NameNode is not available, every activity is frozen both on the client side as on the DataNode side, but as soon as the NameNode is restarted, all the processes reconnect and the activities proceed. This is achieved because the HDFS client can be configured to retry every operation in case of failures. In particular, using a configuration parameters, it is possible to specify the number of times that the client should retry a write operation before really failing.
This is very useful as this provides fault-tolerance also in case of DataNodes failures. Indeed, a client that is writing a file on a specific DataNode, could, in case of failure of that node, retries the write operation on another DataNode as many times as it is configured from the site admin, until a working DataNode will be found and the transfer is completed successfully.
For what about the read operation, it has been tested that if a DataNode fails during a read operation, the client gets only a warning and the operation can continue using a node that hosts the replica. This failure automatically triggers a copies of the chunk hosted in the failed node, in order to guarantee that the number of available replicas is always the same.

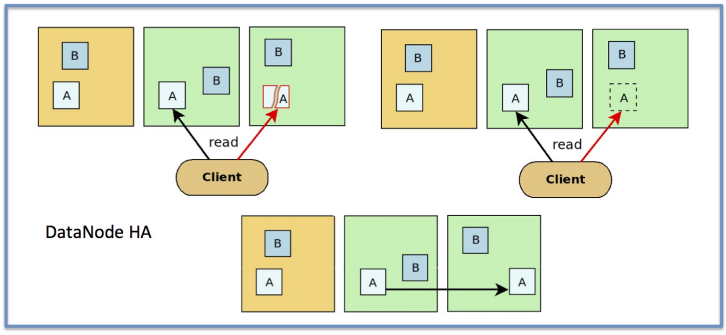

Figure 2. High availability of the DataNode mechanism when reading corrupted data

It has been also tested what happens in case of data corruption. It has been intentionally corrupted a chunk of a file and try to read back that file. The HDFS system automatically understands that there is a problem on that chunk, flags it as failed, reads it from another replica and deletes the corrupted chunk so that it is automatically replicated. This mechanism is shown schematically in Fig. 2.
All these features, are able to reduce the effort needed to maintain the system in production and this is very important as it means a low Total Cost of Ownership.
The INFN-Bari group is also involved in testing the scalability of the HDFS File System, testing a solution with up to 130 different DataNodes. In this test the idea is to prove that a solution based on HDFS could be easily scaled up in terms of datanode in the cluster, by means of simply adding new nodes to it. This was typically called "horizontal scalability" in opposite to the "vertical scalability" that requires that a single, or few nodes are upgraded in order to

obtain grater performance from each of those.
From the test it is evident that a system based on HDFS is highly scalable and we can easily get up to two Gbyte/s of aggregate bandwidth among the nodes.

5. Comparison of Hadoop MapReduce with respect to Fuse performances when accessing HDFS

Hadoop has been deployed successfully as SE in the OSG sites. Within the production setup, the analysis jobs access ROOT files stored in HDFS through Fuse. Before starting the design and implementation of the solution to analyze ROOT files using MapReduce, it was worth to quantify the gain compared to the current production setup. To achieve this, the performance in reading two GB Text files and count how often words occur using MapReduce with respect to Fuse have been measured.

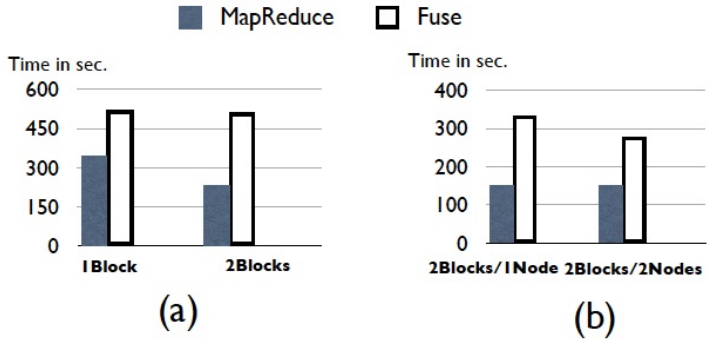

Figure 3. (a) Time in seconds to process two GB total size of one Text file with MapReduce and Fuse as function of the number of HDFS blocks forming the file. (b) Time in seconds to process two GB total size of one Text file split in two HDFS blocks with MapReduce and Fuse as function of the number of nodes hosting the blocks.

The size of HDFS block is a configuration parameter set by the site admin. Files stored in HDFS are shopped up into chunks. Each chunk corresponds to an HDFS block.

By design, MapReduce splits and submits tasks over each HDFS block of the data sets accessed. The blocks are accessed in parallel by MapReduce tasks. Even the HDFS blocks are hosted by the same node, MapReduce submits independent tasks running in parallel over these blocks. Using Fuse, HDFS blocks are accessed sequentially by a single process. This makes the time spent to process the Text files using Fuse is higher, proportionally to number of HDFS blocks, than when processing the Text files using MapReduce. The performances to process one file, sized two GB and split into HDFS blocks, using MapReduce and Fuse are shown in Fig. 3. MapReduce can split data sets per file if requested. As shown by Fig. 4, within this configuration, the time spent to process Text files using Fuse is higher, proportionally to number of files, than when processing files using MapReduce. As previously mentioned, in general, the execution of the task takes place in the node hosting the block running over. In a large infrastructure and when analyzing larger data sets, such optimization can increase considerably the gain of using MapReduce compared to Fuse.

6. ROOT files analysis with Hadoop MapReduce

To take advantage of the parallel processing that Hadoop provides, it is needed to express the ROOT files analysis data flow as a MapReduce job.

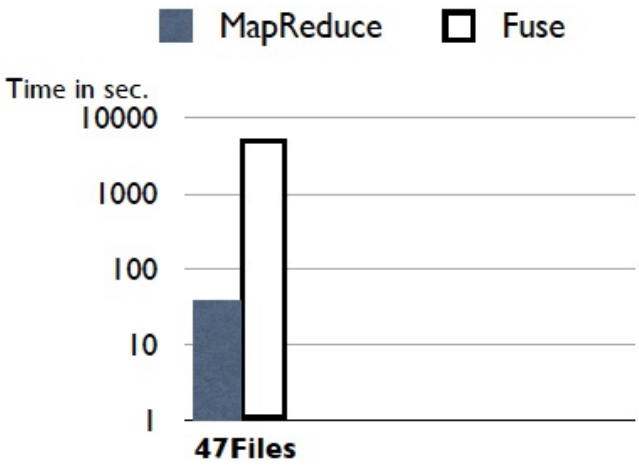

Figure 4. Time in seconds to process two GB total size split into 47 Text files with MapReduce and Fuse.

MapReduce works by breaking the processing into two phases: the "map" phase and the "reduce" phase. Each phase has key-value pairs as input and output, the types of which may be chosen by the programmer. The programmer also specifies two functions: the "map" function and the "reduce" function. In this step of the project, only the "map" function is considered to express the common required step to analyze ROOT files, namely read and write of the files.

The input to the "map" phase is usually one or more text files. The default input format is Text input format. It gives each line in the text files given in input as a text value. The output is the result of the "map" execution over the input files formatted as key-value pairs.

For ROOT files analysis, the input is one or more ROOT files. The Analyzer is usually a C++ analysis code that takes as input the ROOT files names. It opens them for processing by the mean of ROOT primitives call and then produces output ROOT files.

MapReduce and ROOT files Analysis data flows are illustrated in Fig. 5.

The implementation of the ROOT files analysis data flow as a MapReduce job requires the support of ROOT files as input to the "map" function. To achieve that, it was required to develop a dedicated input format, named RootFilesInputFormat, inheriting from the super-class FileInputFormat.

FileInputFormat is the base class for all implementations of InputFormat that use files as their data source. It provides two things: a place to define which files are included as the input to a job, and an implementation for generating splits for the input files. The job of dividing splits into records is performed by the subclasses. The FileInputFormat class calls the RecordReader class to effectively open and read the files contents from the File System.

The RootFilesInputFormat allows the splitting of the input per file instead of line per file. The input files will be then opened and processed inside the Analyzer, implemented as "map" function, producing as result a ROOT file. Within RootFilesInputFormat, the input files contents will not be read and passed as input stream to the "map" function, as it is done for Text files, but only the input split path will be passed to the "map" function. The "map.input.file" environment variable is set to the input split path by the RootFilesInputFormat object and then read by the "map" function. See the Appendix A for the implementation details.

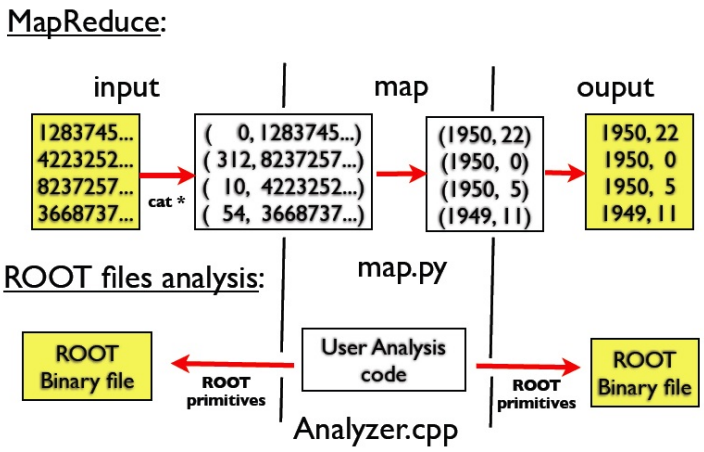

Figure 5. Typical ROOT files analysis and MapReduce data flows: in MapReduce the content of the input files are passed to the "map" function which analyzes it and then produces the output. For the ROOT files analysis, the Analyzer gets as input the files names, open it for analysis, and then produces the output.

7. Analysis workflow performance

Small-scale tests are performed to measure the performance of the implemented Analysis data flow. During these tests, the Analysis workflow is split by file and submitted for execution. The performance of its execution, using Hadoop MapReduce and HDFS, and Portable Batch System (PBS)[16] coupled with HDFS are compared.

7.1. Tests description

A testbed composed of the three machines, (A), (B), and (C), has been setup. Each machine has 24 Cores, 24 GB of RAM, and 500 GB of disk storage.

The machine (A) has been configured as a Master node for both Hadoop MapReduce and HDFS. This machine has been installed and configured as JobTracker, NameNode, and SecondaryNameNode. While the machines (B) and (C) have been configured as TaskTrackers, and DataNodes. The TaskTrackers are configured to accept and execute 48 parallel tasks. While for PBS, the machine (A) was configured as resources manager and job scheduler, and the other two, (B) and (C), as 48 PBS nodes.

The MapReduce job is submitted from the machine (A) to the JobTracker, running in the same machine, which splits the MapReduce job into tasks and schedule them to TaskTrackers for execution. The JobTracker queues the tasks if the TaskTrackers are full. Tasks are scheduled for execution once news slots become available in the TaskTrackers.

The PBS jobs are created and submitted in parallel from the machine (A) to the PBS job scheduler which schedules them to available PBS nodes. Jobs stay in the scheduler queue if all PBS nodes are busy.

The Analyzer submitted for execution in these tests reads a "Tree" from a ROOT file stored in HDFS and write to HDFS an histogram from one of its variables in a new output ROOT file. The size of the files read and the interface used to access HDFS were varied during these tests.

7.2. Performance test of HDFS ROOT files processing through Fuse

Currently, the typical access to HDFS in HEP sites is performed through Fuse, the Posix-like interface of the File System. The aim of this test is to understand the performance of

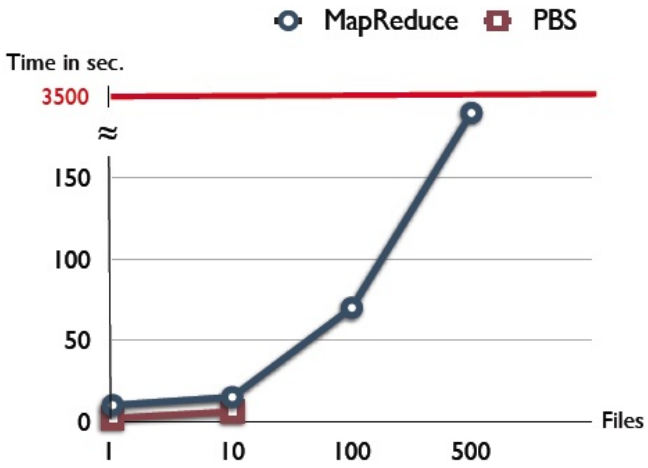

Figure 6. Duration of the workflow execution when submitted using PBS (square) and Hadoop MapReduce (circle) over 100 MB size per file.

the read/write operations performed by workflows submitted using PBS and MapReduce when accessing through Fuse ROOT files stored in HDFS.

For this test, HDFS is mounted through Fuse, using the default value of rdbuffer option which is ten MB. The size of a single file read is 100 MB. Fig. 6 shows the duration of the workflow execution when submitted using PBS and Hadoop MapReduce.

For one file and ten files, the executions of workflows submitted using PBS have shown better performance than the ones submitted using MapReduce. However, for 100 files and 500 files, almost all the jobs submitted using PBS have crashed for Fuse input/output error caused by the I/O of the jobs. Moreover, increasing the value of rdbuffer option, the executions of workflows submitted using PBS continue to show a bit better performance than the ones submitted using MapReduce but they continue to crash with input/output error by increasing the size of the files read. So, the jobs submitted with MapReduce handle better the interaction with Fuse.

7.3. Performance test of HDFS ROOT files processing using ROOT and HDFS libraries

In this test, ROOT files are read from HDFS using the ROOT's HDFS plugin. In particular, issues have been met to make use of this plugin since it requires the manual export of the CLASSPATH environment variable needed by Hadoop. This CLASSPATH uses a particular syntax to point to Hadoop configuration files. To fix that, a patch has been developed and submitted to ROOT developers to export correctly and automatically the CLASSPATH variable when loading the plugin. Since this plugin supports only read operation, the libhdfs library of Hadoop, is used to interact with HDFS for the write operation. Fig. 7 shows in (a) the duration of the workflow execution when submitted using PBS and Hadoop MapReduce over 100 MB size per file, while in (b) the duration of executions over one GB size per file.

The workflows submitted using PBS have shown better performance when executing over 100 MB size per file. However, only a small overhead is introduced by the MapReduce framework for job splitting and processing.

Regarding jobs executed over one GB size per file, MapReduce has shown better performance than PBS. Furthermore, the high load on the PBS nodes when processing 50 files has caused the abort of PBS jobs.

Plots (a) and (b) in Fig. 7 allow to conclude that MapReduce scales better when processing

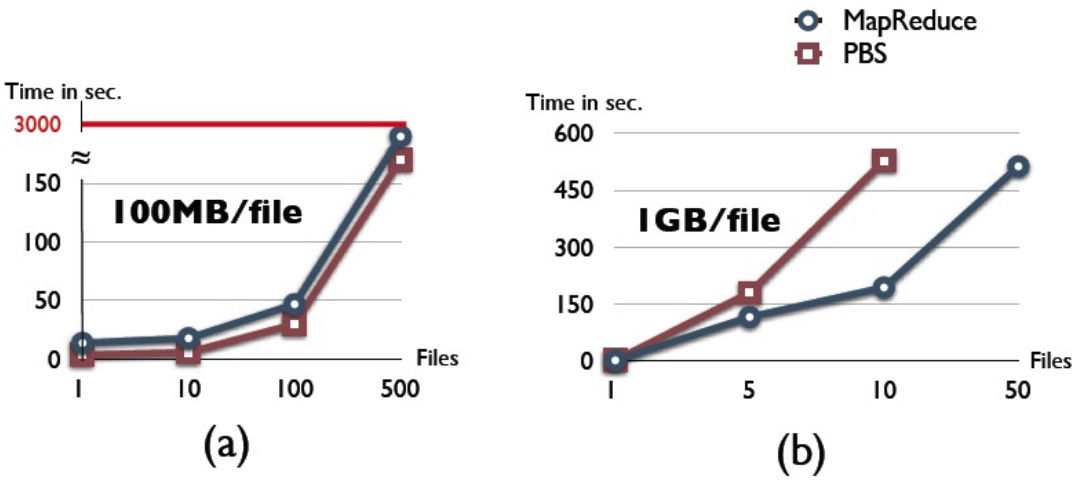

Figure 7. Workflow execution time as a function of the number of files accessed using two submission schedulers: PBS (square) and Hadoop MapReduce (circle). Two cases are considered corresponding to files of size equal to 100 MB (a) and to one GB (b).

large files compared to when processing small ones.

In this test, it is noticed that ~ 95 % of jobs submitted using MapReduce have been executed in the node where the data is located. This feature of Hadoop has made the MapReduce jobs performing less remote read than PBS jobs, in particular for processing big files, the MapReduce jobs were making less use of the CPU and memory resources on the compute node. So, it is expected that proportionally to the increase of the cluster size, the gain of using MapReduce will increase compared to using PBS.

8. Futur work

The next effort will be spent to extend the ROOT files analysis workflow implemented using Hadoop to include the "reduce" function. In general, the output ROOT files are merged by the end-user by hand in a single file for analysis. The "reduce" function can automate this step decreasing the delays supported currently by the end-user analysis.

The Grid Computing Element (CE) is a set of gLite services that provide access through plugins for Grid jobs to a local resource management system. The required plugins for the deployment of Hadoop-based CE will be developed.

9. Conclusions

HDFS has been deployed successfully in gLite environment and has shown satisfactory performance in term of scalability and fault tolerance while dealing with small/medium site environment constraints.

ROOT files Analysis workflow has been deployed using Hadoop and has shown promising performance.

Tests were performed in a cluster composed of three nodes. It is expected that the gain seen in this small infrastructure will increase as the cluster grows.

References

[1] Jeffrey D and Sanjay G 2008 Mapreduce: simplified data processing on large clusters *Commun. ACM* vol 51 pp 107-113

[2] Apache hadoop 2009 `http://hadoop.apache.org/`
[3] Thusoo A *et al.* 2010 Data warehousing and analytics infrastructure at facebook ACM SIGMOD conf pp 1013-1020
[4] Shiers J D 2007 The worldwide LHC computing Grid (worldwide LCG) *Computer Physics Communications* vol 177 pp 219-223
[5] `KantiP` 2009 Black Holes at the LHC *Lecture Notes in Physics* vol 769 pp 387-423
[6] Fanti V *et al.* 1998 A new measurement of direct CP violation in two pion decays of the neutral kaon *Physics Letters B* vol 465 pp 335-348
[7] Moneta L *et al.* 2008 Recent developments of the ROOT mathematical and statistical software *J.Phys.: Conf. Ser.* **119** 042023
[8] Mandrichenko I, Allcock W and Perelmutov T 2005 Gridftp v2 protocol description http://www.ggf.org/documents/GFD.47.pdf
[9] Shoshani A *et al.* 2007 Storage resource managers: Recent international experience on requirements and multiple co-operating implementations 24th IEEE Conference on Mass Storage Systems and Technologies pp 47-59
[10] Open Science Grid `http://www.opensciencegrid.org/`
[11] Bockelman B 2009 Using Hadoop as a grid storage element *J. Phys.: Conf. Ser.* **180** 012047
[12] gLite middleware `http://glite.cern.ch/`
[13] Carbone A, dell'Agnello L, Forti A, Ghiselli A, Lanciotti E, Magnoni L, Mazzucato M, Santinelli R, Sapunenko V, Vagnoni V, Zappi R 2007 Performance Studies of the StoRM Storage Resource Manager *e-science* pp 423-430
[14] Filesystem in userspace 2009 `http://fuse.sourceforge.net`
[15] Site Berkeley Database Information Index `https://twiki.cern.ch/twiki/bin/view/EGEE/BDII`
[16] Bode B, Halstead D, Kendall R and Lei Z *et al.* The Portable Batch Scheduler and the Maui Scheduler on Linux Clusters Proceedings of 4th Annual Linux Showcase and Conference pp 27-27

Appendix A. Implementation details

The implementation of the most important methods of RootFileInputFormat class is shown in Fig. A1 while in Fig. A2 it is shown the implementation of RootFileRecordReader class.

```
package org.apache.hadoop.streaming;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.BytesWritable;
import org.apache.hadoop.mapred.InputFormat;
import org.apache.hadoop.mapred.InputSplit;
import org.apache.hadoop.mapred.JobContext;
import org.apache.hadoop.mapred.RecordReader;
import org.apache.hadoop.mapred.TaskAttemptContext;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileSplit;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.mapred.JobConf;
import java.io.IOException;
import org.apache.hadoop.fs.FileSystem;

public class RootFileInputFormat
        extends FileInputFormat<NullWritable, BytesWritable> {
        protected boolean isSplitable(FileSystem fs, Path filename) {
                return false;
        }
```

Figure A1. Implementation of RootFileInputFormat class.

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.BytesWritable;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.mapred.InputSplit;
import org.apache.hadoop.mapred.RecordReader;
import org.apache.hadoop.mapred.TaskAttemptContext;
import org.apache.hadoop.mapred.FileSplit;
import java.io.IOException;
import org.apache.hadoop.io.IOUtils;

class RootFileRecordReader implements RecordReader<NullWritable, BytesWritable> {
...
public RootFileRecordReader(FileSplit fileSplit, Configuration conf) throws IOException {
...
public boolean next(NullWritable key, BytesWritable value) throws IOException {
        if (!processed) {
                byte[] contents = new byte[0];
                Path file = fileSplit.getPath();
                FSDataInputStream in = null;
                value.set(contents, 0, contents.length);
                conf.set("map.input.file", conf.get("mapred.input.dir") + "/" + file.getName());
                processed = true;
                return true;
                }
        return false;
}
...
```

Figure A2. Implementation of RootFileRecordReader class.