SUPER SCALING PROOF TO VERY LARGE CLUSTERS

M. Ballintijn, G. Roland, K. Gulbrandsen, MIT, Cambridge, MA 02139, USA
R. Brun, F. Rademakers, CERN, Geneva, Switzerland
P. Canal, FNAL, Batavia, IL 60510, USA

Abstract

The Parallel ROOT Facility, PROOF, enables a physicist to analyze and understand very large data sets on an interactive time scale. It makes use of the inherent parallelism in event data and implements an architecture that optimizes I/O and CPU utilization in heterogeneous clusters with distributed storage. Scaling to many hundreds of servers is essential to process tens or hundreds of gigabytes of data interactively. This is supported by the industry trend to pack more CPU's into single systems and to create bigger clusters by increasing the number of systems per rack. We will describe the development of a standardized benchmark for PROOF clusters. The benchmark is self contained and measures the network, I/O and processing characteristics of a cluster. We will present the comprehensive results of the benchmark for several clusters, demonstrating the performance and scalability of PROOF on very large clusters.

INTRODUCTION

The Parallel ROOT Facility, PROOF, is a part of the well known ROOT system (http://root.cern.ch/) [1]. It allows for easy and transparent analysis of large collections of ROOT files in parallel on remote clusters of computers.

The main design goals for the PROOF system are transparency, scalability and adaptability. The use of the system is transparent in that there is as little difference as possible between a local ROOT client based session and a remote parallel PROOF session, both being interactive and giving the same results. By scalability we mean that the basic architecture will not put any implicit limitations on the number of computers that can be used in parallel. In this paper we will assess how well we have achieved this goal so far. By adaptability we mean that the system should be able to adapt itself to variations in the remote environment (changing load on the cluster nodes, network interruptions, etc.).

Being an extension of the ROOT system, PROOF is designed to work on objects in ROOT data stores. These objects can be individually *keyed* objects as well as `TTree` based object collections. By logically grouping many ROOT files into a single object, very large data sets can be created. In a local cluster environment these data files can be distributed over the disks of nodes within the cluster or made available via a NAS or SAN solution. Remote file access protocols available in ROOT, like rootd, dCache or RFIO can be used as well.

By employing Grid technologies, PROOF has been extended from single clusters to virtual global clusters. In such an environment the processing might take longer (not interactive), but the user will still be presented with a single result, like the processing was done locally.

The development of PROOF is a joint effort between CERN and MIT. A more extensive introduction to the use of PROOF is available in [2]

As we deployed PROOF on larger and larger clusters it be came clear we needed a way to validate the algorithms used to distribute work over the slaves as well as to monitor the performance of the network, I/O and CPU resources in the clusters. In this paper we will describe the extensions to PROOF which make this possible and show the results obtained by applying them to PROOF running on very large clusters.

BENCHMARK PACKAGE

The benchmark package consists of three components, test data generation, standard test queries, and the system performance monitoring. Together they provide a standalone package that can be used to validate the configuration of a (new) PROOF cluster, provide a quick way to measure its performance, or to study in dept the performance characteristics of its components. The package is fully self contained, a standard ROOT installation is all that is required. All the files of the benchmark can be found in `$ROOTSYS/test/ProofBench`. We will now discuss each of the components in some detail.

Test data generation

The test data consists of ROOT trees containing objects of the standard ROOT `Event` example class. This class is a realistic model of a non trivial HEP event and uses modern ROOT features like the `TRef` smart pointer. The shell script `make_event_par.sh` is provided to create a PROOF archive or PAR file. This archive will be uploaded to the PROOF slaves making the `Event` class available in the slave servers. The test data is generated in situ on the slaves using PROOF by running a simple Monte Carlo simulation on each of the slaves. The user runs the script `make_event_trees.C`, specifying the directory on each node where the file(s) will be written, the number of events per file and the number of files per node. The script determines the list of nodes and the available slaves and then runs the commands to generate the files. The companion script `make_tdset.C` is used to create the `TDSet` that describes the generated test data, which can be used to run queries on the data. An example session looks like:

```
% make_event_par.sh
% root
root[0] gROOT->Proof()
root[1] .X make_event_trees.C("/tmp",100000,4)
root[2] .L make_tdset.C
root[3] TDSet *d = make_tdset("/tmp",1)
```

Standard test queries

The benchmark package currently contains three `TSelector` based queries. The first one, `EventTree_NoProc.C` measures the overhead of the PROOF system, it does not read any data from the tree. The second one, `EventTree_Proc.C` reads all the data and creates the maximum load on the I/O system. The third one, `EventTree_ProcOpt.C`, reads about 20 % of each event, more closely simulating a real life physics analysis. The queries that read data fill histograms that are returned to the client. This simulates normal usage and provides a cross check on the correct execution of the queries.

Performance measurement extensions

PROOF has been extended with a comprehensive performance monitoring and measurement system that provides easy global monitoring of a query as well as the ability to record the detailed sequence of events and interactions between the components of the system.

The global monitoring is provided by a set of histograms collected by the PROOF master server. These histograms measure for each slave server a number of basic quantities. The histograms are returned to the client at the end of the query in the standard output list. They can also be selected for dynamic feedback during the query, providing a real-time overview of work going on in the slaves. Table 1 lists the histograms that are currently implemented. The monitoring histograms are enabled by calling:

```
root[4] gEnv->SetValue("Proof.StatsHist",1);
```

Table 1: PROOF Performance Histograms

Name[1]	**Description**
PacketsHist	Packets processed per slave
EventsHist	Events processed per slave
NodeHist	Slaves per file serving node
LatencyHist	GetPacket latency per slave
ProcTimeHist	Packet processing total Time per slave
CpuTimeHist	Packet processing CPU time per slave

The detailed system event trace stores interesting events and associated parameters. It is implemented using a ROOT tree that contains objects of the `TPerfEvent` class. The event trace can be enabled in the master only, which provides information from the packetizer, or in master and slaves in which case additional information for each file open and read operation is stored. This last mode provides very detailed information but also produces a large amount of data. Table 2 lists the the currently implemented event types. The event trace and slave event traces are enabled by calling:

```
root[5] gEnv->SetValue("Proof.StatsTrace",1);
root[6] gEnv->SetValue("Proof.SlaveStatsTrace",1);
```

Table 2: PROOF Trace Event Types

Name	**Source**	**Description**
Start	M, S	Start of query
Stop	M, S	End of query
Packet	M	Details of a processed packet
File	M	File processing starts or finishes
FileOpen	S	A file is opened
FileRead	S	Details of a single file read

The performance monitoring and measurement system provides an extensible framework. It is easy to add new histograms or new event types. It is expected that as we gain experience with these tools and with the complex environment of distributed PROOF clusters new features will be added.

It is also important to note that this system is implemented using only the powerful features and tools already available in ROOT and PROOF. The trees produced are easily analyzed using the standard ROOT analysis and visualization tools.

Benchmark support scripts

A number of scripts are included in the benchmark package that facilitate running the benchmark on a cluster. They can be used out of the box or may be modified for specific benchmark goals.

- **Run_Node_Tests.C** For a sequence of increasing number of slaves, runs each selector a given number of times. This gives quick validation of the configuration and a comprehensive overview of the performance of the cluster.
- **SavePerfInfo.C** Saves the monitoring results to a file for later analysis.
- **Draw_Slave_Access.C** Draw a profile of the number of slaves active over the duration of the query.
- **Draw_Time_Hists.C** Draw the basic timing histograms.
- **Draw_PerfProfiles.C** Create graph of the results of a run of **Run_Node_Tests.C**.

MEASUREMENTS OF LARGE CLUSTERS

In order to do meaningful and repeatable tests on large clusters it is imperative to have exclusive use of a cluster for

[1] Actual names have "PROOF_" prepended

a block of time. For the first set of measurements we were grateful for the access provided by Prof. Frank Wuerthwein, who is in charge of CDF computing. Being able to run on 150 nodes of their new cluster, before it went into production was extremely helpful. We learned a number lessons from this initial test. It is important to use local disk rather than NFS for tests of more than a few tens of nodes. The precise number depends of course on the NFS servers, but even light access by each node will quickly add up to an overwhelming load on the servers. Automating all steps of the test provides efficiency and repeatability. We also found bugs in the packetizer and the trace package. We then decided to run a comprehensive set of tests on a slightly smaller cluster, Pharm, on which we will report below followed by another large scale test, this time using the Phobos RCF cluster, also discussed below.

Pharm cluster, 24 nodes

The Pharm cluster is a privately run cluster, owned by Phobos. It is typical for many small analysis cluster currently in use by HEP groups. It contains several generations of machines and was only recently upgraded to full Gigabit Ethernet. Initial tests on this cluster failed because the nodes were connected to two switches with just a 100 Mbit link connecting them. This layout makes it difficult to use all nodes in a single PROOF session as one or two slaves can already saturate this link. The configuration after the upgrade which was used for the tests is listed in Table 3.

Table 3: Description pharm nodes

Num.	CPU	Mem	Disk	Network
6	2x P3 730 MHz	512 Mb	ATA	Gbit
6	2x P3 930 MHz	512 Mb	ATA	Gbit
12	2x P4 1.8 GHz	1 Gb	ATA	Gbit

Figure 1 shows the total time to run a query as a function of the number of slaves (gray points). The setup uses one data file of 100 Mbyte per slave, the amount of data grows with the number of slaves. The triangle, square and diamond markers correspond to the three standard selectors described above. Each point consists of the average of four measurements. The three regions correspond to the three types of machines in the cluster. The black and white markers show the time to run the query in a local root session. The black markers correspond to a query on a single file local to that node while the white markers correspond to a query run over all data (44 files), normalized to a single file. Figure 1 was obtained with a configuration of one slave per node. In this setup PROOF performs very well, the fact that the total query time remains below the local query time for the slower nodes indicates that the faster nodes successfully contribute to the slower nodes. In Figure 2 a similar measurement is shown, but now using a configuration of two slaves per node. It can be observed that the faster nodes

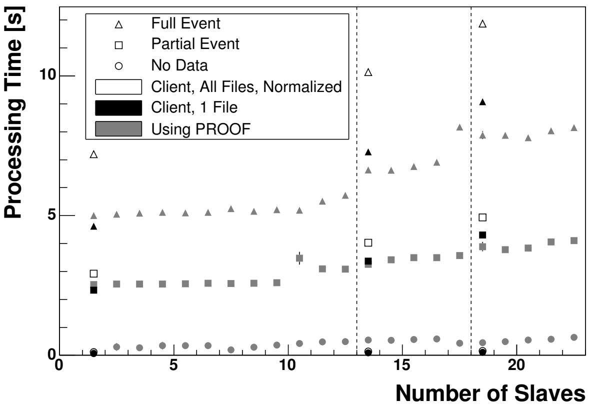

Figure 1: Total query time, 1 slave per node on pharm

have enough I/O and memory bandwidth to reach the same query time. However, the slower nodes reach a bottleneck and the total query times increases significantly. As a result the faster nodes cannot contribute because the rootd processes on the slow nodes are competing for the same resources.

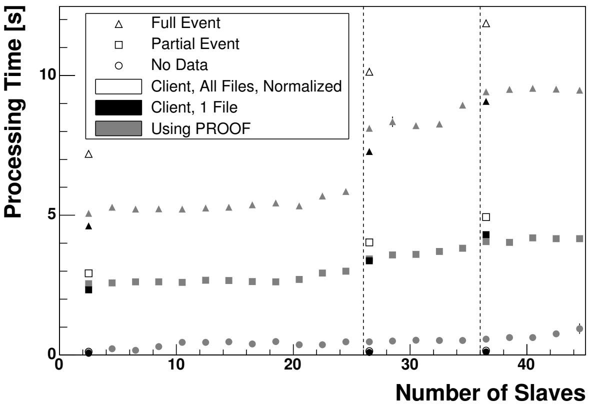

Figure 2: Total query time, 2 slaves per node on pharm

In order to better understand the latency associated with reading a remote file using rootd we also enabled the trace tree on the slaves. In Figure 3 the distribution of latencies is shown for the cases of two slaves per node, forty four slaves total. The average of 1.5 ms should be compared to 200 μs latency for local reads.

RCF cluster, 192 nodes

The Phobos RCF cluster is one of the four large, centrally managed clusters at BNL. This cluster also contains a number of generations of servers. These servers are again connected to a number of switches. The servers used in our test are listed in Table 4. Each group is connected to a switch and the switches are connect through four bonded Gigabit links.

Figure 4 shows again the query time as a function of the number of slaves. The system performs very well up to 100

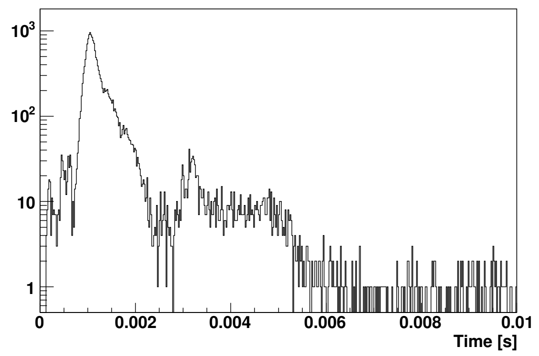

Figure 3: Remote file read latency

Table 4: Description RCF nodes

Num.	CPU	Mem	Disk	Network
18	2x P4 1.4 Ghz	1 Gb	ATA	100 Mbit
99	2x P4 2.4 Ghz	1 Gb	ATA	100 Mbit
75	2x P4 3 Ghz	1 Gb	ATA	Gbit

slaves, reaching an aggregate throughput of 800 MByte/s. But above 100 slaves a dramatic increase in the time per query can be observed. To start to understand what is happening we can look at Figure 5. These graphs show the number of active slaves during the query. We see the number of slaves quickly ramping up, followed by a phase where all slaves are processing. At the end, when no more work remains, the number of active slaves drops to zero. Looking at the bottom graph of Figure 5 we see that a single slave, processing the last packet, remains active for many seconds. This appears to be a general effect that occurs as the number of slaves exceeds the threshold. The effect seems to be caused by high latency remote file reads, but the root cause is not yet fully understood. We are currently evaluating several improvements that will allow us to avoid triggering this effect.

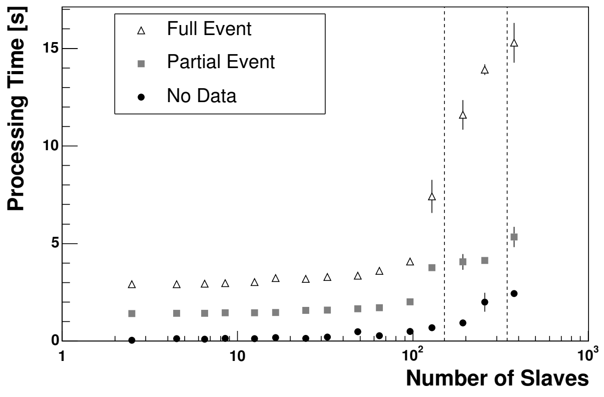

Figure 4: Total query time, 2 slaves per node on RCF

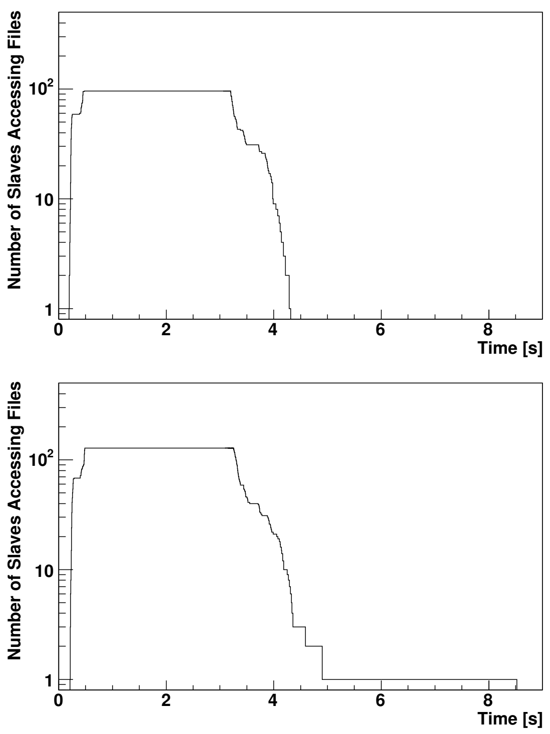

Figure 5: Active slaves during query, 96 & 128 slaves

CONCLUSIONS AND PLANS

The ability to measure the performance of PROOF and to study in detail different aspects of the system has been very valuable. The benchmark package has allowed us to setup and run these tests in a very short amount of time. This was a key factor in testing on the large clusters.

PROOF was shown to scale to well over a hundred slaves at RCF. It is clear that a good understanding of the underlying system and network architecture is key. More work is required on both PROOF and system architectures to reach the next goal a 1000 slaves.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 0219063 and No. 0313002.

REFERENCES

[1] Rene Brun and Fons Rademakers, ROOT - An Object Oriented Data Analysis Framework, Proceedings AIHENP'96 Workshop, Lausanne, Sep. 1996, Nucl. Inst. & Meth. in Phys. Res. A 389 (1997) 81-86. See also http://root.cern.ch/.

[2] M. Ballintijn, R. Brun, F. Rademakers and G. Roland, The PROOF Distributed Parallel Analysis Framework based on ROOT, Proceedings CHEP03 Workshop, La Jolla, California, Mar. 2003, http://arxiv.org/abs/physics/0306110 .