Installation of PROOF-lite and ROOT on NERSC systems

Anastasia Pierce McTaggart
NERSC
University of Massachusetts at Amherst
Amherst USA
apmctaggart@gmail.com

***Abstract*—As many improvements in modern computing relate to massive parallelism, software which can easily run in parallel is highly useful. High energy nuclear and particle physicists often use the software ROOT, which has a built in utility, PROOF, to automate parallelism of physics analyses. The feasibility of running ROOT with PROOF-lite, a single node paradigm of PROOF, on two NERSC systems, the serial PDSF, and the massively parallel Hopper, will be discussed in detail. Benchmarks were run in order to determine the performance. Using PROOF-lite on Hopper led to a significant net improvement for IO benchmarks, and much better use of memory. CPU results were affected more by clock speed of the processors compared to IO results, so there wasn't an improvement. Additionally, a traditional physics analysis was run on both PDSF and Hopper to show the speedup in a more tangible manner.**

***Keywords*-ROOT;PROOF;Parallel Computing;NERSC**

I. Introduction

In the modern era of big data and complicated code, efficient computing is essential to productive science. Parallel computing is one method to increase the efficiency of certain types of problems. By simultaneously running tasks, efficiency is greatly improved. However, parallelism cannot be exploited for all types of problems, as some types of complex problems defy attempts to parallelism them, such as sequential approximations of π. Parallel code is also more difficult to write. It is for these reasons and many others that the National Energy Research Scientific Computing Center (NERSC) at the Lawrence Berkeley National Laboratory (LBNL) has many different systems to support the unique needs of over 5000 users. Some of those users include High Energy Nuclear and Particle Physicists (HENP), who at present, use the system known as PDSF (Parallel Distributed Systems Facility), the longest continuously running Linux cluster in the world [1]. PDSF is a scientific Linux cluster, with the goal of supporting serial IO (input-output) intensive workloads. However, at present, to profit from the benefits of parallel computation on PDSF, users must run many instances of their code at once. This approach uses a large amount of memory, and requires nontrivial work for HENP physicists to ensure their code works properly.

NERSC has another system, Hopper. Hopper is a petaflop system which runs a limited Linux kernel and is designed for large scale parallel computations. As HENP physicists often have mostly embarrassingly parallel code, it would drastically improve efficiency to use Hopper for this type of analysis. Furthermore, it would provide another resource for HENP physicists to use, and Hopper is more cost-effective per core than PDSF to run. However, at present, Hopper supports MPI as a way to create a distributed environment to run tasks and jobs in parallel on multiple nodes. This requires rewriting the code for an analysis, which can be complicated. HENP physicists often use the program ROOT, from CERN, to do analyses. ROOT includes a built in utility to enable event level parallelism known as PROOF, that presents an alternative framework to MPI Event level parallelism is essential for many HENP physics analyses as they frequently run the same code over many different events in order to determine important characteristics. PROOF provides a framework to write code, and then takes care of the complicated multithreading automatically, greatly simplifying code production. ROOT had not been tested for Hopper, so the purpose of this paper is to explore the feasibility of running and using ROOT and PROOF-lite with Hopper. PROOF-lite is the multicore single node version of PROOF that will be run on a single user's account. Expanding beyond PROOF-lite would require system software modifications, and therefore is not practical at this point in time. The feasibility and efficiency of using PROOF were determined using the standard PROOF benchmarks for both CPU and IO intensive tasks on a single node on Hopper.

II. Parallel Computation

Parallel computation is an effective way to increase the efficiency of many types of problems. Parallel computing works by running many parts of the same problem concurrently on multiple processors or cores (hencefore referred to simply as processors), eliminating the wait between instructions common in serial, or standard, computation. For instance, in serial computation, one could have a loop iterating over thousands of objects and doing the same task to all of them individually, one after another. In parallel computation, all of the objects could have the task done simultaneously, drastically improving efficiency. There are two main ways to determine the predicted speedup from a switch to a parallel system.

A. Amdahl's Law

Amdahl's law [2] is one method to determine the speedup for a problem of fixed size, but variant time. Amdahl's law predicts the speedup as in figure 1.

$$S = \frac{1}{1 - P + \frac{P}{N}}$$

Figure 1. S is the speedup, N is the number of processors, P is the proportion of code which is parallel

As HENP physicists often have highly parallel code, the speedup can be quite large. For example, a 90% parallel program, not unreasonable for an embarrassingly parallel case, on a single 24 core node such as those on Hopper, would lead to a speedup of approximately 727%, almost 3 order of magnitude.

B. Gustafson's Law

Another method to determine the speedup by parallelism is Gustafson's law [3] , which is applicable for the case of fixed time, but variable problem size. As people can usually find more calculations to perform with their compute time, this is the more commonly used law. Gustafson's law predicts the increase in calculations to be as in figure 2.

$$S = N - \alpha(N - 1)$$

Figure 2. S is the speedup, α is the proportion of code which cannot be parallelized, N is the number of processors

For example, a 90% parallel program, not unreasonable for a highly parallel case, on a single 24 core node such as those on Hopper, would lead to a speedup of approximately 21.7 times the previous number of calculations. Unlike Amdahl's law, Gustafson's law does not assume that as the processors increase to infinity, that the speed must as well. This better reflects the reality where a small section of code, that cannot be made parallel, can be the majority of compute time.

III. PROOF

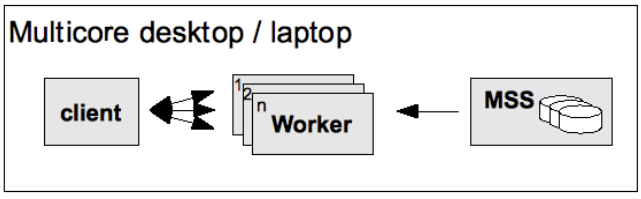
A. Background and motivation

PROOF is an extension of the ROOT software which allows ROOT analyses to easily run in parallel. PROOF uses the Selector framework to automate parallelization of embarrassingly parallel tasks. PROOF works in conjecture with ROOT, eliminating the need for the programer to learn an entire new programming environment, such as MPI or OpenMP. However, only some systems are capable of using PROOF at present, for example, GPU systems require knowledge of OpenMP to run C++ code, as opposed to a pure CPU system like Hopper. The synergy between PROOF and ROOT allows the same analysis code to be run as before, without extensive changes. PROOF is highly scalable, thus making it effective for a large system such as Hopper. Additionally, PROOF is highly adaptable, providing robustness in the event of changing loads on shared nodes or occasional network interruptions [4]. Although PROOF and PROOF-lite are often used interchangeably, there are specific benefits and disadvantages to each daemon. PROOF-lite was chosen because PROOF requires being installed along with ROOT on the entire system, as opposed to simply being run out a user's directory, as PROOF-lite can be. Hopper deliberately has few programs installed by default, because of the diverse users, so installing ROOT and PROOF would be a disadvantage for the non physicists using Hopper. The major disadvantage to using PROOF-lite as opposed to PROOF is that PROOF-lite can only be used on one node. This is not an issue for Hopper, because ROOT cannot be run on multiple nodes without rewriting the source code. ROOT can only be run on one node at a time due to a limitation of both the CCM (Cluster Compatibility Mode), the Cray software used to simulate a full Linux system, and the default limited kernel. The final limitation is that both systems disable shared multinode libraries by default. In order to use multinode libraries, the CINT interpreter used by ROOT would need to be modified to use a -dynamic flag in the linking, which would require modifying the ROOT source code. This prevents ROOT from easily enabling shared libraries between nodes, and as the benefits to parallelism on a single node are still great, was not done.

B. How PROOF works

PROOF works by implementing a multi-tier master-worker architecture. PROOF-lite, as shown in figure 3 from the ROOT website [5] implements the same architecture, but with two tiers.

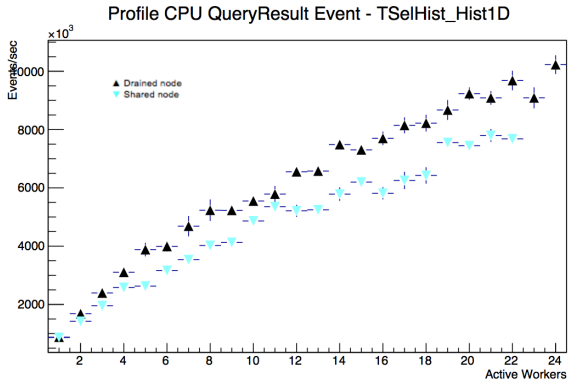
Figure 3. PROOF-lite architecture



The architecture handles requests from the user by sending the requests to the PROOF master, which then distributes the requests to the worker threads. At the end, the master collects and merges the results. PROOF enables event level parallelism by using the Selector framework. The Selector framework consists of three logical steps: begin, process and

terminate. PROOF parallelizes all the embarrassingly parallel parts of the process stage automatically. Additionally, to better handle the varying loads on shared nodes, PROOF has a built in scheduler to increase efficiency. Benchmarks were run on both PDSF and Hopper to determine a baseline and any improvements from using Hopper. They consisted of a CPU and IO benchmark. The CPU benchmark consists of creating 16 1d histograms, each filled with 30,000 times the number of workers random numbers. The number of workers, is by default, the number of threads on the node, or cores to a node. The IO benchmark consists of making 30,000 times number of cores event files, and then inputting the files into memory, scanning over the files, and outputting the event files to test IO rates. This should lead to a linear increase with number of processors, as expected from Gustafson's law. However, as shown in figure 4, from the benchmarking, the scheduler still results in delays when PROOF encounters other threads for the CPU benchmark. This was not an issue on Hopper, because each node is devoted to only one user, unlike on PDSF. Those delays are shown in figure 4, where the upside down triangle is the node shared with other users, and results in a lower peak number of interactions due to conflicts with other worker threads in comparison to the drained node case, in which the expected linear increase occurs. This difference was found by running the benchmarks on PDSF on both drained and regular nodes.

Figure 4. Benchmarking encountering other threads on PDSF



IV. Components and Configuration

The configurations of PDSF and Hopper are described below.

A. PDSF

PDSF is a Scientific Linux distributed computing cluster that is optimized for serial jobs. It is used frequently by HENP physicists groups. There are 1100 cores distributed over 205 compute nodes, which are shared between users. Some of the nodes have 8 cores, like those on the debug queue, but many of the newer nodes have 12 cores, which with hyperthreading enabled, results in 24 virtual cores. PDSF jobs have access to Mendel, a more modern expansion, as well, so there's a chance of getting a Mendel node, with 32 virtual cores. The default memory allocated to each job is 1.1 GB, but one can allocate as much as is needed, up to a maximum of 47 GB per node. PDSF is optimized for IO because it has 450TB of local disk, which is considerably faster to write to, and also has an GPFS file system for IO intensive tasks, which is called project. However, local disk is usually only used as a temporary SCRATCH space for running jobs, as it requires time to upload to it, which can be considerable.

B. PDSF IO

PDSF uses a system where it can write to a local directory, or to a disk outside, such as the project directories. The limiting factor comes from the router which only supports 20 Gb/sec uplink and limiting access to 1 Gb/second. This limitation is due to the maximum bandwidth. The expected rates to the project directories are 30 GB/sec (240 Gb), and are the same as on Hopper, but the greatest differences in results occur from the queue used, as the 30 GB/sec is shared between all users.

C. PDSF queues

1) Standard Queue: On the standard, shared, PDSF queue, nodes are shared between users. This can cause overlapping and a small delay when multiple users are using the same node. The original nodes have 12 real cores, which with hyperthreading results in 24 virtual cores.

2) Debug Queue: The debug queue sends jobs to the debug nodes, which have 8 cores for the exclusive use of a single job.

3) Drained nodes: Although draining a node is not usually done in a production job, it is an effective way to test a benchmark. It allows exclusive access to one of the standard nodes, such as those in the regular queue, with 24 virtual cores. No Mendel cores were drained for the purposes of this project. Standard drained nodes provide an accurate scale-up of the debug queue results.

D. Hopper

Hopper is a 1.28 petaflop/sec system which runs a limited Linux kernel and is designed for large scale parallel computations. It has 153,215 compute cores and 6,384 compute nodes. Each compute node is made up of 2 twelve core processors, which results in 24 virtual cores, as on PDSF. All of the cluster is in the same configuration. The standard nodes have a maximum of 32 GB of memory, but there are large memory nodes with up to 64 GB of memory. IO depends highly on which directory it is writing to, hence there are local and global SCRATCH directories used.

E. Hopper IO

1) Hopper SCRATCH and SCRATCH 2: SCRATCH directories still have a bandwidth limit that prevents IO of greater than 35 GB/sec (280Gb/sec), but are encouraged for IO intensive jobs, in contrast to the home directories or project directories. The default quota is 5TB, which is unlikely to be exceeded in a reasonable run, but if it is, there is the global SCRATCH option, and the quota can be increased upon request. Files left on the SCRATCH directories are purged after 12 weeks. The location of the SCRATCH directories varies per user, but can be found via the variable $SCRATCH and $SCRATCH2.

2) Hopper Global SCRATCH: Global SCRATCH has reduced IO speeds of 15 GB/sec (120Gb) compared to local SCRATCH, but has a default quota of 20TB and allows users to run code on multiple platforms.

3) Hopper Project Directories: Hopper Project directories work the same way as PDSF project directories and are stored on the same global file system, thus enabling cross system access.

F. A description of Batch jobs

Batch jobs were used to run all of the programs. A batch system is used to execute jobs without any interactive input. In the process of debugging code, interactive or even login nodes were used, but for the actual programs to run, the batch system was used, both on Hopper and PDSF. Batch processing is done so that many jobs can be run at once and so that the code does not rely on human typing speed for input, or other easily modifiable factors. Batch systems also allow the system to allocate resources properly, preventing one job from hogging all of them, which is very difficult to control in a real time online system. It also means that for long running code, such as some of the benchmarks, that one can submit the code, go home, wait for the computer to find it an appropriate slot, and then read the results upon returning to the office 12 hours later as opposed to waiting to see the results online instantaneously. Using a batch system does add another layer of complexity, as the required commands can differ slightly than using ROOT interactively. In particular, ”-> ”and ”.” are interchangeable in the interactive version of ROOT, but not in the batch version. However, the benefits of using the batch system outweigh any negatives, and most of the ROOT macros work perfectly fine on batch systems without any changes.

1) PDSF Batch: PDSF uses the UGE (Univa Grid Engine) which is a fork of the SGE (Sun Grid Engine) in order to effectively run batch jobs. If jobs are run for more than 24 hours, UGE will kill them, as this exceeds the wallclock limit. This did not occur with the benchmarks. One can submit a job to UGE by using the qsub command with assorted variables, as is described on the NERSC webpage [6]. These can include memory required, the queue, the node, the IO resources, etc. PDSF required more memory than the minimum in order to do the CPU benchmark because the default is set for a job which occupies a single core. PDSF is also less efficient than Hopper with memory because it was not designed to run jobs in parallel. Only standard benchmarks were run on PDSF to determine a baseline, but proper allocation could increase efficiency for IO intensive tasks. Furthermore, the queue and node can be specified, allowing one to use a single user node or a debug node to have a node solely to the program running. PDSF will not allow one to submit a job if one is over quota, so it is important to not use the home directory for PROOF runs.

2) Hopper Batch: Hopper uses Torque as the batch system, and requires one to submit a .pbs script with information such as the walltime for the program, the queue, the number of cores, the number of nodes, the location of the program, and a shell script to open the program. The jobs on Hopper were run inside a special module, CCM, which emulates a standard computing environment. Hopper also allows one to request more memory, but this was not an issue in running the benchmarks. Hopper also allows one to run jobs in serial, but this was not done because it would defeat the purpose of using Hopper as a fast parallel system.

V. Building Root

ROOT was built on both PDSF and Hopper from source according to the default directions from the ROOT webpage [7]. The default configure methods worked on Hopper, as it is 64 bit x86 and close enough to Linux to be detected. In the event of using multicore machines, it is suggested to run make with the -j flag, which allows one to add the number of processors. This does not result in a difference in results, because PROOF automatically does multithreading. As soon as ROOT is installed, PROOF-lite can be run by creating a PROOF-lite session.

VI. Running jobs

A. PDSF

Jobs on PDSF are run by simply submitting a shell script to open ROOT and run the macro for benchmarking or other purposes. The jobs are finally submitted with the qsub command. Macros and shell scripts remain the same as on Hopper, and are shown in listings 2 and 3. The minimum memory was changed on PDSF, in qsub via the flag l h_vmem, where h_vmem is set to the amount needed, up to 15 GB in the benchmarks due to the serial nature of the Linux cluster. It should be noted that on PDSF, the same shell script was used, with correct paths, as on Hopper. The command qsub on PDSF takes the shell script as an argument, in the same way aprun does in the .pbs script on Hopper, however, qsub is used directly as opposed to with the .pbs script.

B. Hopper

Running a job on Hopper is a bit more complicated, as it involves a .pbs script to set all the variable names and in CCM mode, requires importing environmental variables manually as opposed to directly from the local environmental variables. The need to use CCM led to errors in the initial attempted production of the benchmarks as ROOT requires the CCM mode and uses ccmrun instead of aprun. On Hopper, the default number of processes was set to one so that PROOF could create as many threads as needed (which was the number of worker threads declared in RunCPU, or RunDataSet, or by default, the number of processors detected by PROOF-lite). The greatest difficulties were had in running the dataset benchmarks, as PROOF by default writes to the home directory, which has a very low bandwidth limit. The .pbs script that was used to submit the program is shown in listing 1.

Listing 1. Script to submit code

```
#PBS -q ccm_queue
#PBS -l mppwidth=24
#PBS -l walltime=12:00:00
#PBS -N myscript
#PBS -e myscript.$PBS_JOBID.err
#PBS -o myscript.$PBS_JOBID.out


cd $PBS_O_WORKDIR

export  CRAY_ROOTFS=DSL
module load ccm
module load openmpi_ccm

ccmrun mpirun -np 1 \\
/global/project/projectdirs/\\
pdsf/amct/hopper/root/shell3.sh
```

It should be noted that the last 3 lines are all on one, but due to formatting, are placed on separate lines here. Listing 2 depicts the shell script used to call ROOT to open with a specific macro, again with line breaks added with \\ added to denote them.

Listing 2. Script to submit code

```
#/bin/bash
echo $HOSTNAME
. /global/project/projectdirs/
\\pdsf/amct/pdsf/root/bin/thisroot.sh
root -b -q /global/project/
\\projectdirs/pdsf/amct/pdsf/macro.c
```

Listing 3 depicts the macro used to run the benchmarks. Note that the IO methods may be run by uncommenting them, and commenting out the CPU, and the location of the PROOF sandbox can be changed.

Listing 3. Script to submit code

```
#include <iostream>
using namespace std;
//macro.c
void  macro(){
   TStopwatch t;//create the timer object
   t.Start();//start the timer
  gEnv->SetValue("ProofLite.Sandbox",
\\"/global/project/projectdirs
\\/pdsf/amct/proofSand1");
  TProofBench pb("");
  // pb.MakeDataSet();
  //pb.RunDataSet("BenchDataSet",1,24,1);
   pb.RunCPU(1000000,1,24,1);
 t.Stop();//stop the timer
   t.Print();//print info
}
```

The same method of line breaks, in the code, as in listings 1 and 2, is used in listing 3.

VII. Benchmarks

In order to correctly measure improvement, the baseline must be determined. Benchmarking is one way to do that. Benchmarks are standard codes that are run on different systems to determine an expected level of performance. Benchmarks were run on both PDSF and Hopper, to determine the overall improvement for both CPU and IO intensive tasks, and if Hopper was in fact, an improvement. The results of each benchmark are expected to increase linearly according to the number of worker threads, as expected by Gustafson's law. However, the results do not always follow expectations. The amount of time that each code took to run was determined via the ROOT class TStopwatch. The memory used was found on PDSF via the Linux command qacct. On Hopper, the memory per job is listed at https://www.nersc.gov/users/job-logs-and-analytics/completed-parallel-jobs-cray-aprun/

A. PDSF Benchmarking

Benchmarking was done on PDSF at first to establish the baseline on PDSF. To avoid the case where local writes are done, improving IO performance drastically, all writes were done to the project directories, excluding the case mentioned below of local directories on PDSF. Local IO is not the standard as it requires uploading to the compute node, which takes an excessive amount of time for standard data amounts. For instance, a 1TB file would take over 2 hours, at minimum, to upload to a PDSF compute node with a 1 Gb uplink bandwidth. Local IO was compared here for a comparison with Hopper and Global filesystems. Using different queues produced different results, as well.

Figure 5. CPU on pc2634, shared queue

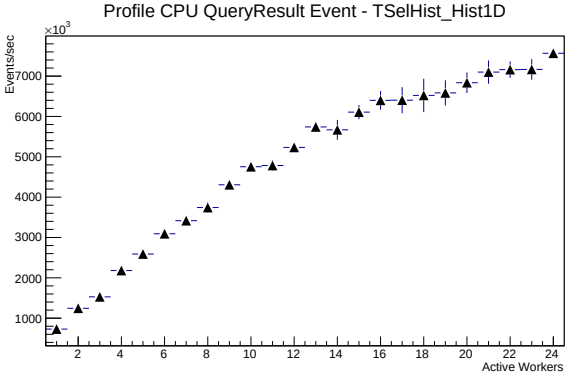


Figure 6. IO mb on pc2634, shared queue

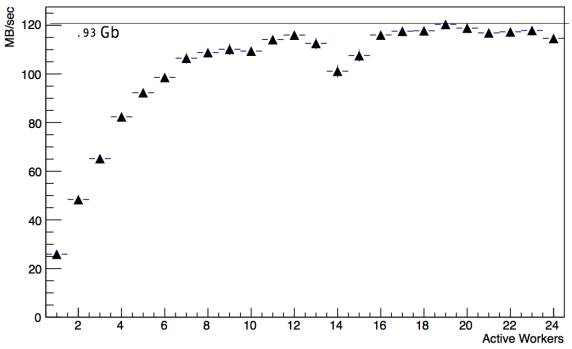


Figure 7. IO on pc2634, shared queue

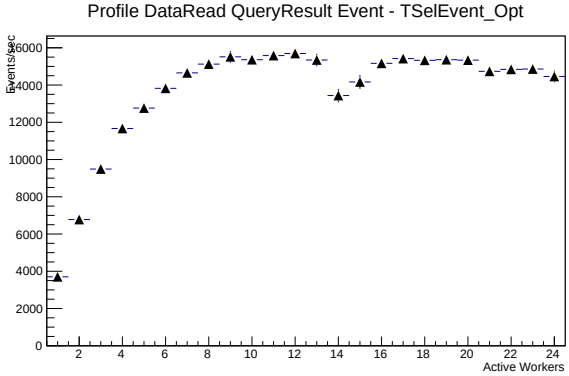
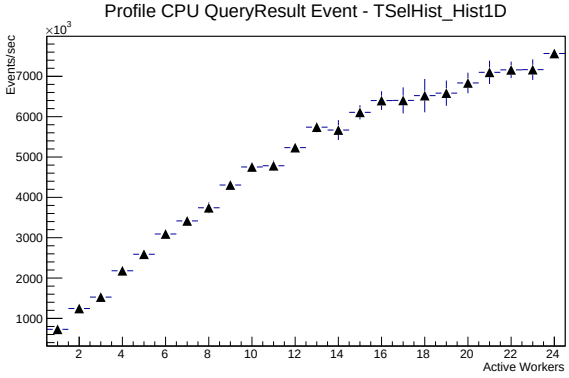


Figure 8. CPU on mc0103, shared queue



1) Regular queue: The results from submitting the benchmarking to the regular queue deviated from the expected linear form. This is because the threads interact with other jobs on the same node. The benchmarks show deviations from the expected linear behavior when PROOF encounters another thread, as shown in figure 5 for CPU benchmarking. The deviations are due to the system scheduler pausing the PROOF processes, resulting in a delay between events and a lower number of calculations per second being completed. This is because there are more processes than available cores.

The CPU benchmark took 3:05 minutes to run, and used 5.055 GB of memory, as is shown in figure 5. However, as shown in figures 6 and 7 for IO benchmarks, the uplink bandwidth was hit, at 1 Gb per second, impacting results. The IO benchmark took 1:23 hours to run and used 8.304 GB of memory.

2) Shared queue, Mendel nodes: Using Mendel produces different results. Due to the more modern architecture, Mendel results in much more efficient processing, being comparable to a drained node despite interacting with other threads.

The CPU benchmark took 4:05 minutes to run, and is shown in figure 8. It used 6.640 GB of memory

Figure 9. IO mb on mc0124, shared queue

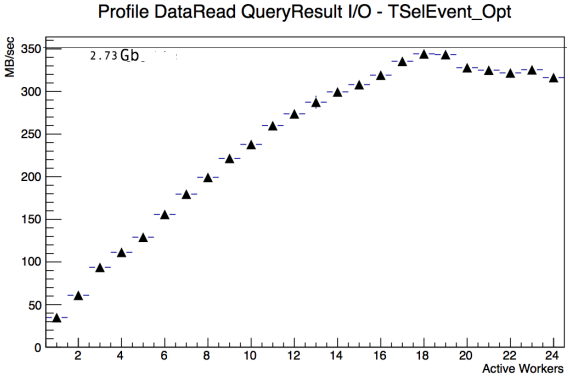


Figure 10. IO on MC0124, shared queue

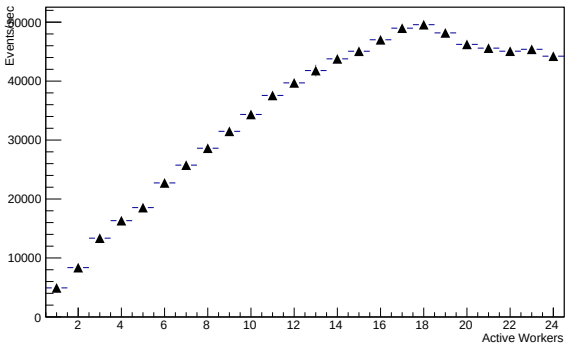


Figure 11. CPU on pc2030, debug

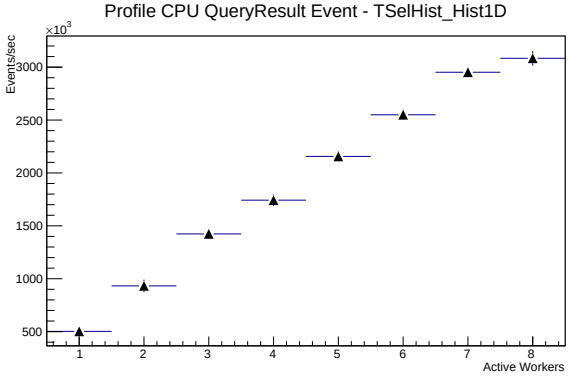
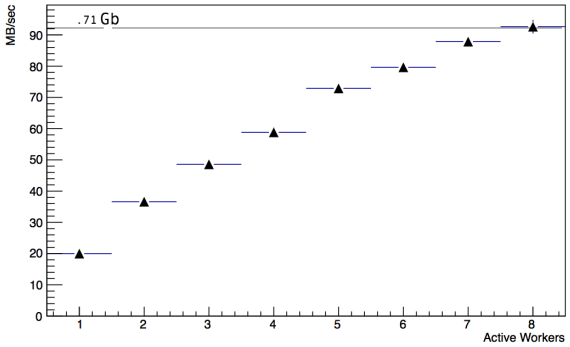


Figure 12. IO on pc2026, debug,mb



However, as shown in figures 9 and 10, the IO shows a linear increase up until 16 workers. The uplink bandwidth was not hit, as Mendel has more modern architecture. IO is dependent on the number of real cores, and depends on chip architecture, not the number of multithreaded cores, so it is not suprising that it tapers off at 16 workers.

The IO benchmark took 36:18 minutes to run and used 10.153 GB of memory.

3) Debug queue: As a result of this issue, the debug queue was the next one used as it would provide a single node to use. The debug queue nodes only have 8 cores, in contrast to the 16 multithreaded to 32 cores on standard PSDF nodes and this particular code was run on pc2026 for the IO, and pc2030 for CPU. These results should not be compared to the results on Mendel, but the standard results on a non drained node. As is shown in figure 11, the issue with the CPU benchmark disappears, and the IO benchmark remains the same as shown in figures 12 and 13, but with fewer nodes, so the gains are not as great. It took 18:06 minutes to run. It used 2.934 GB of memory. The uplink limit was not hit for the IO benchmark. The CPU benchmark took 1:35 minutes to run and used 1.938 GB of memory.

4) Drained nodes: As a result of the knowledge that PROOF encountered difficulties when it encountered other threads, a node pc2634 was drained and only had the benchmark running on it. The results are shown in figure 14 for CPU and 15 and 16 for IO.

The CPU benchmark took 3:35 minutes to run. It consumed approximately 5.302 GB of memory.

The IO benchmark took 1:17 hours to run and consumed

Figure 13. IO on pc2026, debug

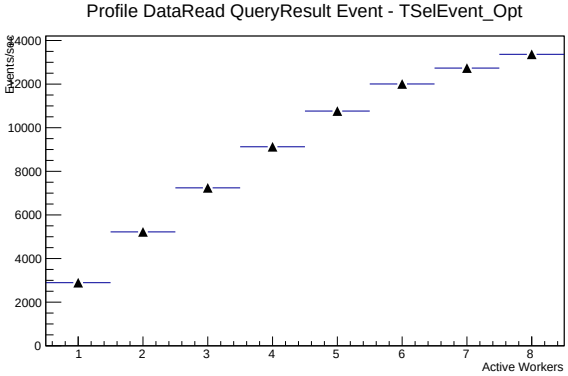


Figure 14. CPU on pc2634, drained node

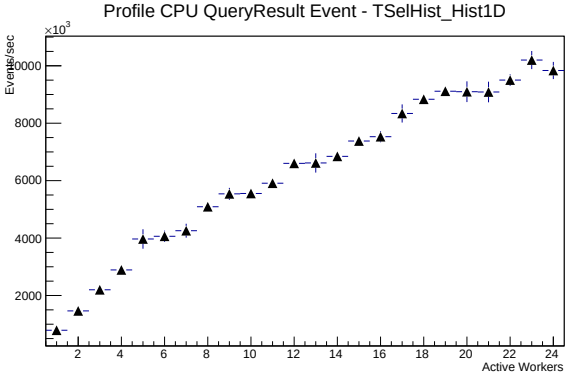
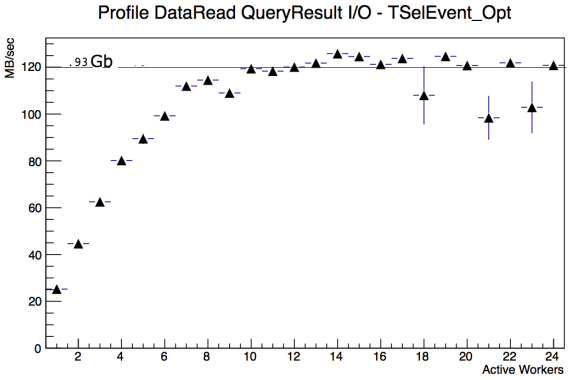


Figure 15. IO on pc2634, drained node



8.327 GB of memory. It is shown in figure 10.

5) Local IO: PDSF is better with local IO, for the obvious reason of locality. It requires uploading input data to the compute node, so in practice, external systems are used more regularly. The CPU benchmark, in figure 17, better shows the different rates of increase after 12 threads have been created. The TMPDIR variable was used to write locally,

Figure 16. IO on pc2634, drained node

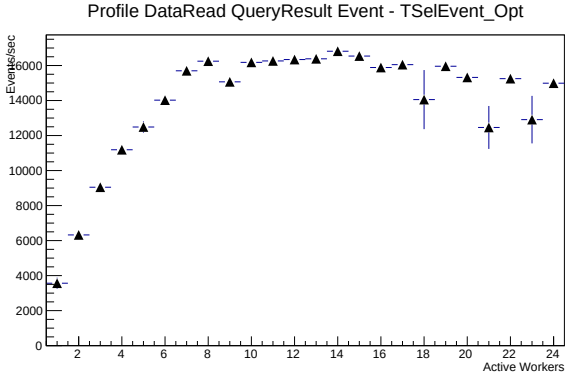
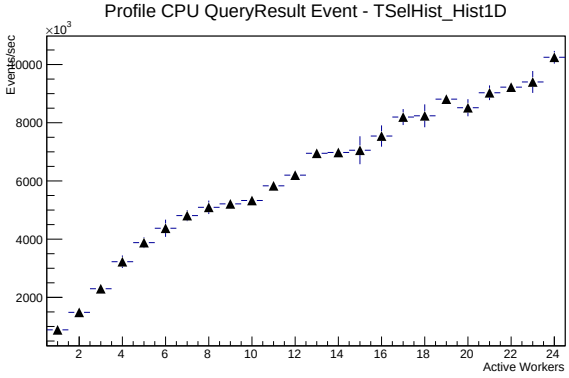


Figure 17. CPU on pc2634, local



and although there is a major difference in IO, from the external project directories, as shown in figures 18 and 19 due to different loads on the project filesystems, there is not a major difference for CPU performance, compared to the drained nodes. CPU on the local directory took 5.302 GB of memory and took 5:34 minutes to run. Additionally, the IO uplink limit was not hit, so the local IO shows a taper at 12 workers in contrast to the uplink limit stopping before that. IO on the local directory took 1:04 hours and used 6.643 GB of memory

B. Hopper Benchmarks

Benchmarks were also run on Hopper. The major differences occurred depending on which working directory was used for the PROOF sandbox, as all the nodes on Hopper have the same architecture. The directory which PROOF uses as a sandbox can be modified by inserting in the ROOT macro gEnv- >SetValue("ProofLite.Sandbox", "/location/of/directory"); which will modify the PROOF- lite sandbox.

1) SCRATCH and SCRATCH2: It is beneficial to have two SCRATCH directories so that if one wants to run multiple jobs at once, one does not run into the issue of

Figure 18. IO on pc2634,local

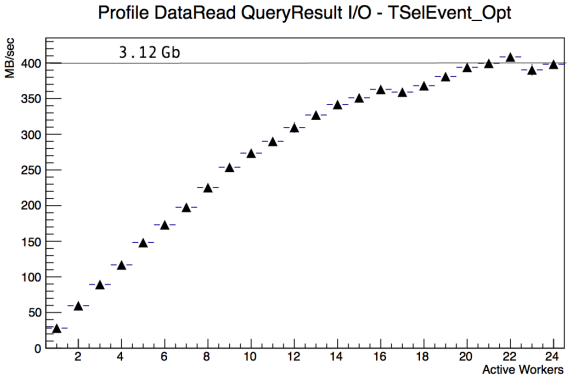


Figure 19. IO on pc2634, local

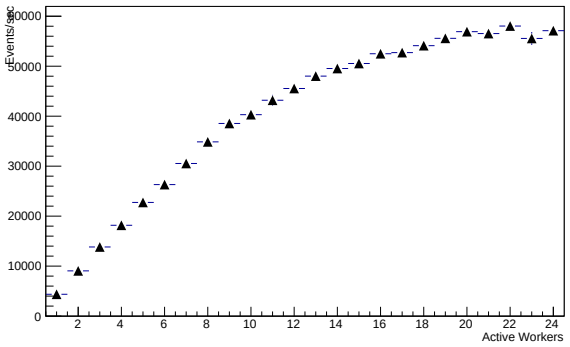


Figure 20. CPU Benchmarking on SCRATCH, nid06094

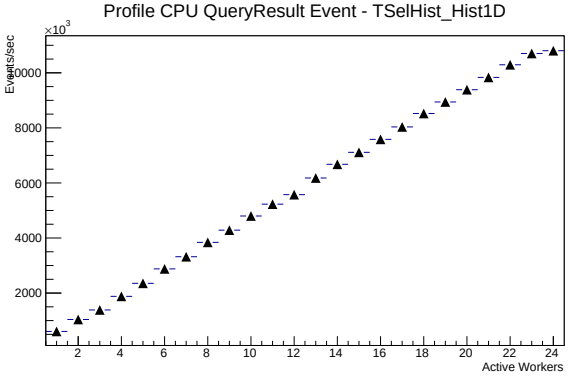


Figure 21. CPU Benchmarking on SCRATCH2, nid05322

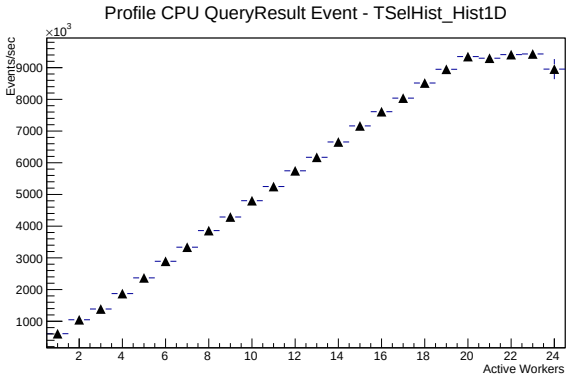


Figure 22. IO Benchmarking on SCRATCH,mb

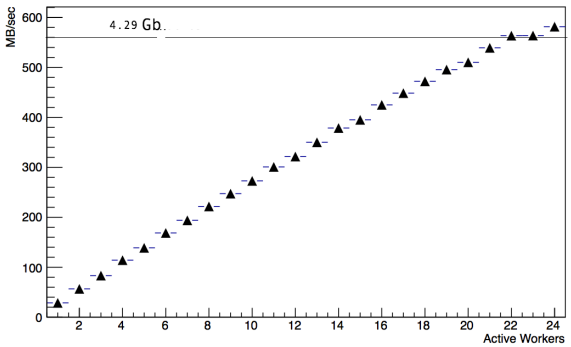
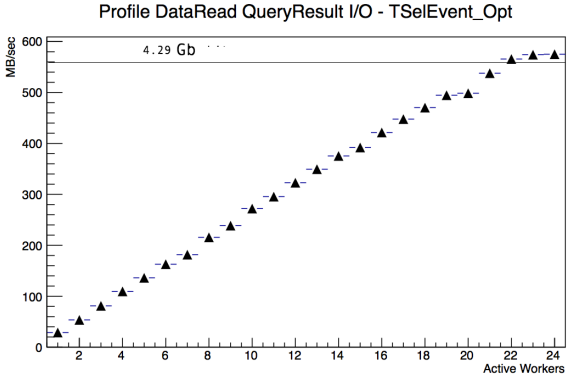


Figure 23. IO Benchmarking on SCRATCH2,mb



the jobs stepping on each other, as was the issue with PDSF on the shared queue and shared nodes. This overlap can cause jobs to crash on Hopper.

The result of the CPU benchmarks are shown in figures20 and 21, for both SCRATCH and SCRATCH2. This shows an drastic increase in CPU performance from 1 worker to 24.

As suspected due to the similar nature of the SCRATCH systems, there is no major difference between SCRATCH and SCRATCH2 for IO, or for CPU. The only differences are due to different loads on the system As shown in figure 22 and 23, the results are better on the SCRATCH directories for IO, and it does not change the overall results if SCRATCH2 is used instead of SCRATCH.

Figure 24. IO Benchmarking on SCRATCH

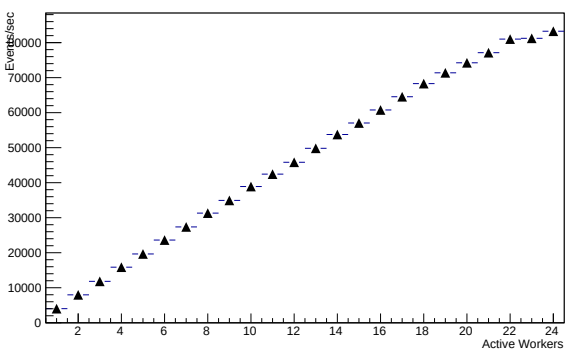
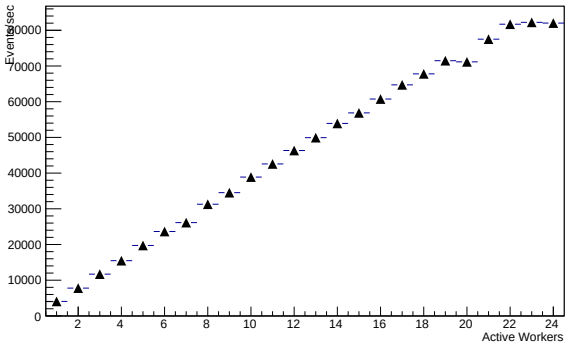


Figure 25. IO Benchmarking on SCRATCH2



There is a slight stall at the 19th worker on SCRATCH2 for reasons related to other users using the system at the same time and the load varying per system. As the SCRATCH and global SCRATCH directories are designed to withstand heavy loads and improved regularly, any deviations can be considered a result of the current load on the system, not the underlying system itself. Despite there being 24 available cores, PROOF must be explicitly told with the RunCPU or RunDataSet to run over all of the worker threads, and declaring it in the constructor for TProofBench is ineffectual. This was unclear from the documentation.

This particular benchmark took 33:05 minutes on SCRATCH and 33:35 minutes on SCRATCH2. This increase in time is likely due to loading the CCM module, which appeared to take close to 30 minutes. It used .1 GB of memory on SCRATCH and SCRATCH2 for both the CPU and IO benchmarks, a great improvement over PDSF.

2) Global SCRATCH: Interestingly, there is a dropoff at about 18 workers for CPU on global SCRATCH. This is due to the load on the global filesystem affecting IO of CPU

Figure 26. CPU Benchmarking on Global SCRATCH

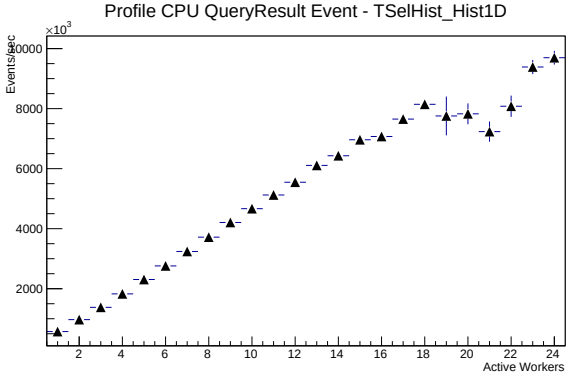


Figure 27. IO Benchmarking on Global SCRATCH

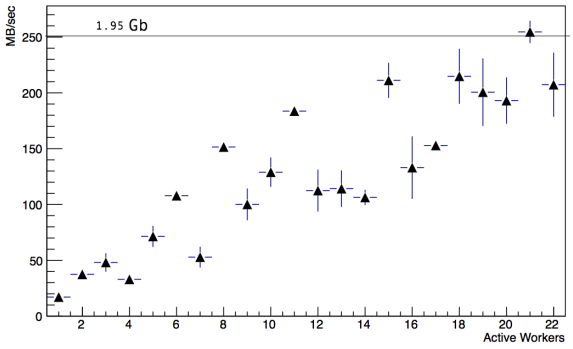
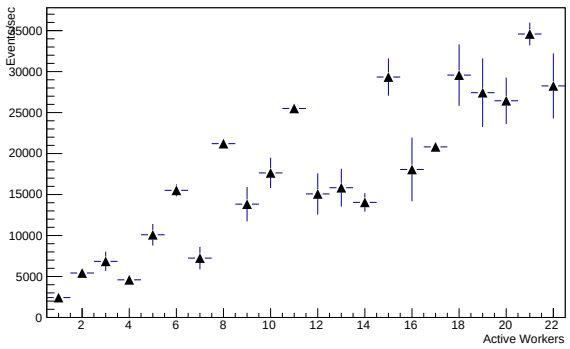


Figure 28. IO Benchmarking on Global SCRATCH



results. CPU is shown in figure 26.

As shown in figures 27 and 28, there is considerable variance due to the bandwidth limitations in the IO benchmark results. This results in a large error bar being displayed, and is best corrected by switching to a file system that doesn't involve the lower bandwidth limits, and ideally has a lighter load, for IO intensive tasks.

Figure 29. CPU Benchmarking on project directories

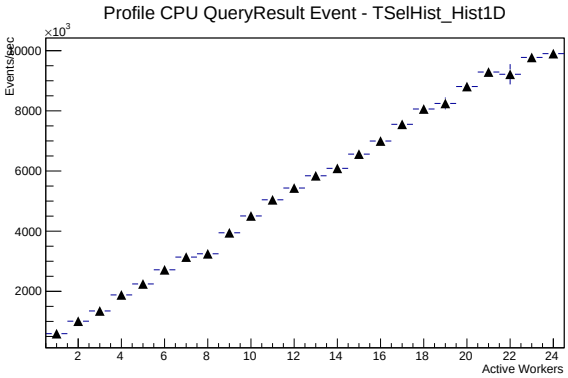
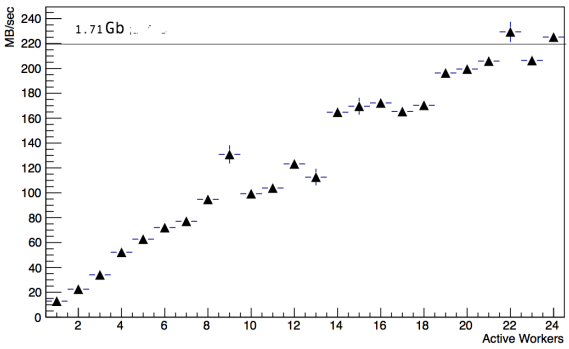


Figure 30. IO Benchmarking on project directories,mb



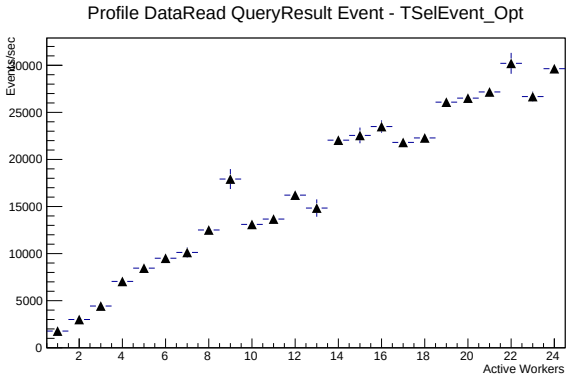
3) Project Directories: For comparison with PDSF, the benchmarks run out of a project directory are shown below. Interestingly, the CPU results remain similar, regardless of the file system in use, as seen in figure 29. However, the IO is where the real differences between file systems become visible, as shown in figures **??** and **??**.

IO is not as rapid because the project directory has a maximum IO of 30 GB/sec (240 Gb/sec), shared between all users. The project directory is not designed for IO intensive tasks. The greatest benefit to this is the lack of purging, and that it allows projects running on multiple NERSC machines to easily share data. The use of the project directory also allows comparison with PDSF results.

VIII. Discussion

In doing the benchmarks, the assumption that Hopper would be automatically faster was challenged. In reality, clockspeed has the largest impact on embarrassingly parallel computing, with the same number of cores, multithreaded or real. Therefore, Mendel nodes on PDSF are slightly faster than Hopper for the CPU benchmark. However, where Hopper shines is in memory usage, using roughly a

Figure 31. IO Benchmarking on project directories



hundredth of the memory that PDSF's Mendel uses. It is also noted that for data IO, the queues matter on PDSF, with a drained node using a third of the memory that the shared queue uses, due to a lack of interactions with other threads. It was also shown, due to Hopper's more modern chip architecture and lustre file system, that IO was considerably faster, even in comparison to local IO on PDSF. Local IO on PDSF obviously has locality, but Hopper's efficiencies still overcome that major benefit, and Hopper is nearly 4 times as efficient for writing to the SCRATCH directories. Even when comparing apples to apples, as with the project directories, Hopper has a max IO rate of 1.73 Gb in contrast to PDSF's .93 Gb. Mendel reached comparable results to the project directories on Hopper, but not to SCRATCH on Hopper, which still outperformed PDSF. This difference is due to different maximum bandwidths on the different PDSF systems, compared to Hopper. Hopper has a maximum bandwidth of 12 Gb per second, rather as the older parts of PDSF only have a maximum bandwidth of 1 Gb per second.

However, the benefits of parallelism are still apparent, even for CPU results, as shown by the differences between PDSF's debug queue and the final results. More threads are generally better, and a 24 real core system will be more efficient than the 12 real cores per node that PDSF has. The differences is most pronounced between the SCRATCH directories and the drained node, as the SCRATCH directories display less interference from other jobs than the global directories on Hopper, creating a greater contrast between PDSF and Hopper. Additionally, project directory results were comparable because of bandwidth limitations. Hopper also showed the benefits of parallelism in the IO results, because it has 24 real cores in contrast to the standard PDSF nodes with 12 real cores, which control the IO. Hopper also has a more modern architecture than PDSF, resulting in faster IO even with a similar number of cores, such as on Mendel.

IX. Conclusion

In conclusion, this research enables NERSC and other high performance computing facilities to use ROOT in parallel on various systems, making data analysis easier for all involved, and embarrassingly parallel code automatically multithreaded. It shows how HENP physicists can expand easily to more powerful parallel computers at NERSC, which frees up resources for NERSC to allocate, reduces the cost in running a cluster for physicists, and enables data analysis to be much more easily done. The benefits of parallelism are once more shown, especially with IO results on Hopper in contrast to PDSF, with Hopper coming out roughly a Gb per second ahead of PDSF's local IO, despite the benefits from locality on PDSF. Hopper compared similarly to PDSF's newest nodes, Mendel, for CPU benchmarks, despite having a lower clock speed, due to the single user nature of Hopper's nodes, and Hopper was considerably more efficient in memory utilization. As parallel computation becomes ubiquitous in computer science, the impact of it increases, and enabling groups such as HENP physicists to use parallel computation effectively becomes increasingly important. Additionally, this will enable HENP physicists to gradually shift from cluster based computing with PDSF at NERSC to the more easily maintained parallel model, like Hopper. Finally, PROOF-lite automatically detects number of available cores and utilizes them efficiently without requiring physicists to make any changes to their analysis code. This speeds up code production in contrast to a manually multithreaded code. There is considerable room for NERSC to optimize ROOT for Hopper, but presently, the benefits in terms of user cost, CPU time, and memory usage make Hopper the better choice for using ROOT for parallel physics applications at NERSC.

Acknowledgment

The author would like to thank her project advisor, Iwona Sakrejda for the extensive assistance she gave. Additionally, she would like to thank Lisa Gerhardt, Jack Deslippe, Helen He, and Katie Antypas for assorted technical support. Finally, she would like to thank Aimee Tabor for organizing the TRUST (Team for Research in Ubiquitous Secure Technology) program, and the National Science Foundation (NSF award number CCF- 0424422) for funding her REU via TRUST.

References

[1] "National Energy Research Scientific Computing Center - Wikipedia, the free encyclopedia." [Online]. Available: http://en.wikipedia.org/wiki/National_Energy_Research_Scientific_Computing_Center

[2] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," *Proceedings of the April 18-20, 1967, spring joint computer conference on - AFIPS '67 (Spring)*, p. 483, 1967. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1465482.1465560

[3] J. L. Gustafson, "REEVALUATING AMDAHLS LAW," pp. 4–6, 1988.

[4] "Proof." [Online]. Available: http://root.cern.ch/drupal/content/proof

[5] "Multi-Tier Master-Worker Architecture ROOT." [Online]. Available: http://root.cern.ch/drupal/content/multi-tier-master-worker-architecture

[6] "Submitting Jobs on PDSF." [Online]. Available: http://www.nersc.gov/users/computational-systems/pdsf/using-the-sge-batch-system/submitting-jobs/

[7] "Installing ROOT from Source ROOT." [Online]. Available: http://root.cern.ch/drupal/content/installing-root-source

[8] "More about Proof." [Online]. Available: http://root.cern.ch/drupal/content/more-about-proof