The University of Tennessee
Knoxville

ECE 599

Supercomputer Design and Analysis

Big Orange Bramble

Supervisor:
Dr. Mark Dean

Team Leader:
Gregory Simpson

Team:
Shawn Cox
Kelley A. Deuso
Patricia Eckhart
Chencheng Li
Liu Liu
J. Parker Mitchell
Taher Naderi
Jordan Sangid
Sepeedeh Sepehr
Caleb Williamson
Aaron Young

August 9, 2016

Abstract

This project involved the design and construction of a high performance cluster composed of 68 quad-core ARMv8 64-bit Raspberry Pi 3s. The primary intent of the project was to establish the operating environment, communication structure, application frameworks, application development tools, and libraries necessary to support the effective operation of a high performance computer model for the students and faculty in the Electrical Engineering and Computer Science Department of the University of Tennessee to utilize. As a foundation, the system borrowed heavily from the Tiny Titan[21] system constructed by the Oak Ridge National Laboratory, which was a similar but smaller-scale project consisting of 9 first generation Raspberry Pis. Beyond the primary target of delivering a functional system, efforts were focused on application development, performance benchmarking, and delivery of a comprehensive build/usage guide to aid those who wish to build upon the efforts of this project.

Table of Contents

List of Figures

List of Tables

1 Introduction

Currently, computing systems are progressing towards the exascale, device structures are being manufactured on the nanoscale scale, and Big Data continues to grow. However, complex phenomena such as weather, fluid and particle dynamics, astrophysics, as well as molecular behavior have been difficult to model with traditional computing systems due to their limited processing, communication, and storage capabilities. With the present technology and growing need for more advanced computing systems, High Performance Computing has become an increasingly significant vehicle for technological progress.

This document is the accumulation of design, implementation, and testing results of a high-performance parallel computing cluster, code-named Big Orange Bramble, or BOB, using the Raspberry Pi 3 single-board computer. The purpose of this document is to archive approaches and solutions used by the team while creating a Raspberry Pi-based Beowulf cluster in such detail that the entire project can be recreated and further developed by another team without the need for extensive understanding of parallel systems. The contents of this composition begins with an overview of the project followed by implementation details for each hardware and software component. Testing and performance results for each component are then discussed in detail. The remaining sections discuss the team's final conclusions as well as specific shortcomings of the system which could potentially be avoided by those who wish to construct a similar system using this guide. Finally, attached to the report is a complete **Hardware/Systems Build Guide** that can be found within **Appendix A** and an **Application Install and Usage Guide** within **Appendix B**.

The authoring group is comprised of graduate and undergraduate students enrolled at The University of Tennessee, Knoxville studying within the fields of Computer Science, Computer Engineering, and Electrical Engineering.

2 List of Parts and Materials

2.1 Primary Parts List

Quantity	Item
68	Raspberry Pi 3 Model B+
68	Addico/ Raspberry Pi Heatsink Set for B/B+ 2 and 3 (Set of 3 Aluminum Heat Sinks)
68	64GB SD Card Ultra
64	Custom Daughter Card for Power Monitoring
23	3 pack, 3ft USB 20AWG Charging Cables
16	GeauxRobot Raspberry Pi 3 Model B 4-layer Dog Bone Stack, Clear Case Box
16	5 pack, 6ft Flat Ethernet Cables
10	USB 7-Port Hubs
10	5V, 20A Trimmable Power Supplies
8	AC Infinity Multifan S3, Quiet 120mm USB Fan
2	PDU - 12 Outlet
2	TP-LINK TO-SG1048 48-Port 10/100/1000Mbps Gigabit 19-inch Rackmount Switch, 96 Gbps Switching Capacity
2	4TB USB HDD
1	Custom 19" Rack Enclosure
1	Wireless Microsoft Xbox 360 Controller
1	USB Ethernet adapter

TOTAL APPROXIMATE COST: $8000

2.2 Optional Parts List

Quantity	Item
1	HDMI Monitor
1	USB Keyboard
1	USB Mouse
1	Raspberry Pi 7" Touchscreen Display

3 Implementation

3.1 Hardware Development

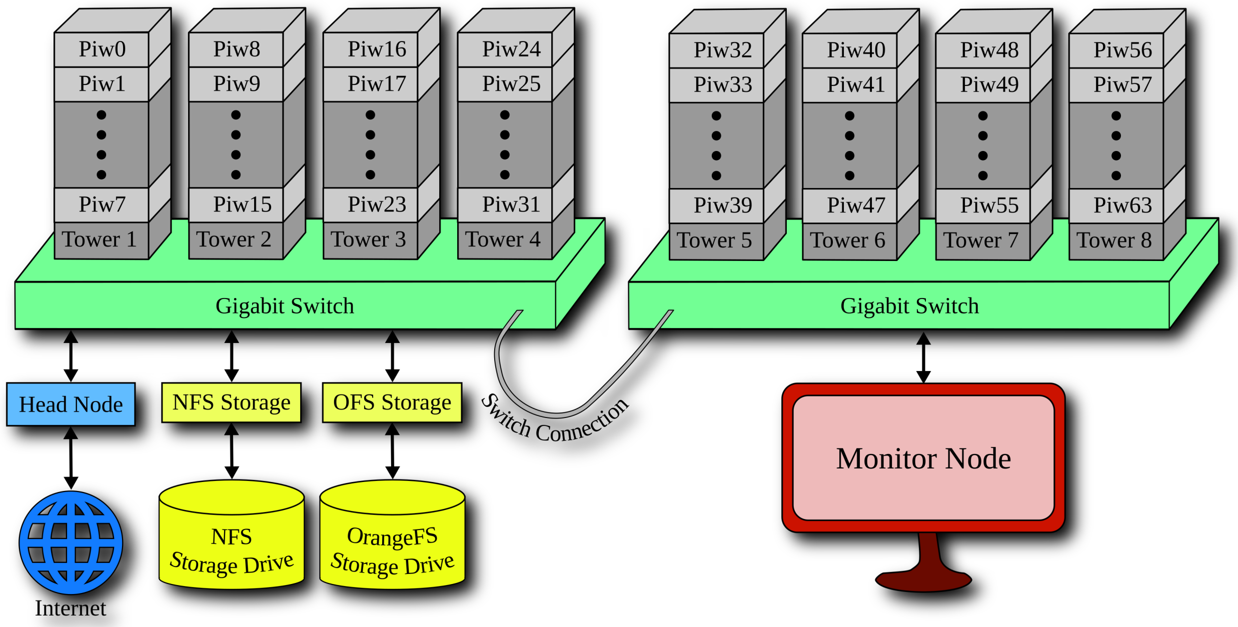


Figure 1: Hardware Diagram

3.1.1 Power Distribution Network

The main function of the power distribution network is to deliver the appropriate voltage and current to each Raspberry Pi node so that the performance could be maximized. If the power distribution were to provide less power than adequate, the nodes would fail to perform at an expected level. If the power distribution network were to provide more power than necessary, especially in the event of over-voltage, there would be an increased risk of board-level device failure resulting in irreparable damage to a node.

A secondary function of the power distribution network is to condition the power entering the nodes. Adequate noise filtering was to be designed so that the noise floor on the DC rails would be minimized. A large noise floor (greater than 20% of the DC level) could lead to the misbehavior of board-level components as well as corrupt the integrity of digital signals.

The design of a power distribution network began as a redesign of the originally implemented network. The original power network consisted of generic USB charge stations and generic 6 ft micro USB cables. This design was determined to be less than adequate during High-Performance Linpack (HPL) benchmarking. While the nodes were approaching maximum computation load, the voltage level measured on the boards would sag below the acceptable threshold for Ethernet-based communications causing the loss of multiple and often all nodes from the cluster. This under-power event was characterized by a sag in the V_{supply} measured locally from the Raspberry Pi 3's GPIO 5 V pins. It was shown that the steady state voltage measured locally on the Raspberry Pi 3 was on average 4.6 V. As the current increased, the voltage would begin to sag sometimes below 4 V with a duration greater than 1 s.

The original assumption was that the power supplies were not capable of providing the required current during HPL benchmarking. After performing a dissection of the power supplies, the supply voltage was measured 5.1 V with no load as well as with 8 of the channels under load of 2.5 A. This evidence indicated that the power supplies were not under-performing. The next logical culprit of this under-voltage was the extremely long USB cable transmitting the power between the power supplies and the nodes. A group of these cables were tested showing an average nominal impedance of 600 mΩ for each path. When connected to the power supply, the impedance of the send and return path was measured to be on average 1.2 Ω. The cables were cut and examined to show a gauge of 28 AWG which would be a very poor conductor under the maximum measured load of 750 mA during periods of high computation. Using the impedance of the cables and the max current at load, the expected voltage drop across the wire was $(1.2\,\Omega)(750\,\text{mA}) = 1\,\text{V}$. The calculated, expected voltage drop was still lower than the measured results.

A closer look at the anatomy of the power supplies showed that a 100 mΩ resistor was being used in a current sense feature of the supply (the channel would be clamped if a current > 2.7 A was detected). Furthermore,an evaluation of the Raspberry Pi board determined a 2.5 A fuse (MF-MSMF250) was being used with an expected resistance of $15\,\text{m}\Omega - 100\,\text{m}\Omega$ and a PMOS device (DMG2305UX) was being used for reverse polarity protection that contained an $R_{DS,ON}$ of approximately $68\,\text{m}\Omega - 200\,\text{m}\Omega$. When considering these newly discovered impedances the worst-case voltage drop under load could be calculated to be $(1.6\,\Omega)(750\,\text{mA}) = 1.2\,\text{V}$. Using this calculated voltage drop, the under-power of the power distribution network could be explained.

As a result of these evaluations, new power supplies with V-adjust capabilities, USB hubs, and shorter, heavier gauge USB cables were ordered. The new power supplies could reach a maximum average voltage of 5.8 V and supply greater than 10 A with no sag in voltage (100 W rated). The new cables used 20 AWG stranded wires for power lines, which at 3 ft decreased the nominal cable impedance to below 100 mΩ. However, the new USB hubs required some modification in order to be used in the power distribution network (See **Appendix A.1.1**). On the printed circuit boards (PCBs) within the USB hubs, the 5V trace width was measured to be approximately 100 mils (1 mil = 0.001 in). The trace width required to carry the desired current capability of 10 A is 370 mils. Therefore, the PCBs were modified by slicing the 5 V bus trace and placing 16 AWG wires across them to allow for current fanning. Additionally, 16 AWG wires were connected to the Ground plane of the PCB and the initial 5 V connection and brought out of the enclosure as a twisted pair to connect to the screw terminals of the new power supply.

The present power distribution network consists of 10 Switching Mode Power Supplies (SMPS), each of which provides up to 100 W. Every SMPS is responsible for powering up to 7 USB ports which gives 2.86 A of capacity per port which is safely above the expected draw of ~ 1 A under load. Each USB port has the option to have power toggled through a push button adjacent to the port. The power supply and USB hub are shown in **Figure 2**.

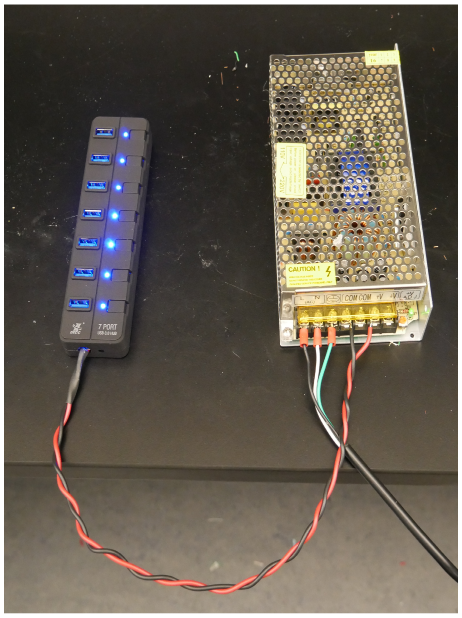


Figure 2: Power Supply and USB Hub.

3.1.2 Daughter Card

The intent of the Daughter Card is to provide access to the input power of each individual Raspberry Pi node for a current sense and voltage sense device. The power monitoring device selected was the Texas Instruments INA219. This device monitors the voltage drop across a high side, current sense resistor and will send this information along with the supply voltage from an integrated Analog-to-Digital Converter (ADC) to the Raspberry Pi through I^2C.

Because of the difficulties associated with attempting to place the I-sense shunt resistor in series with the Raspberry Pi's USB power entry, the best approach was determined to be to use the Raspberry Pi's General Purpose Input/Output (GPIO) ports as a power entry (**Figure 3**). Though advantageous for the power monitoring, this did bypass all of the Raspberry Pi's board-level power conditioning which included reverse polarity protection and over-current protection in addition to basic filtering. Consequently, the designed daughter card would have to include at minimum a power supply filter capacitor. Using Altium Designer, a schematic was drafted to implement the INA219 and Daughter Card (see **Appendices C and D**). A 3D model of the daughter card can be seen in **Figure 4**.

While the PCBs were being manufactured, the firmware team began to experiment with the INA219's I^2C interface with the GPIO of the Raspberry Pi 3 using the Adafruit INA219 Breakout Board (`https://www.adafruit.com/product/904`). The I^2C libraries provided by Adafruit for the INA219 current and voltage sensor were designed with the Raspberry 1 and 2 in mind. As a result, it became necessary to modify the library by hardcoding some values specific to the Raspberry Pi 3. The library and monitoring code were written in Python.

Currently there are two ways to read measurements from the daughter card. One way is

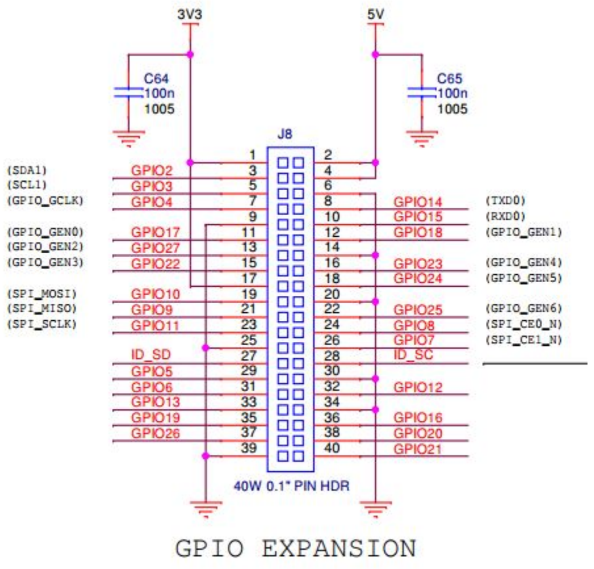
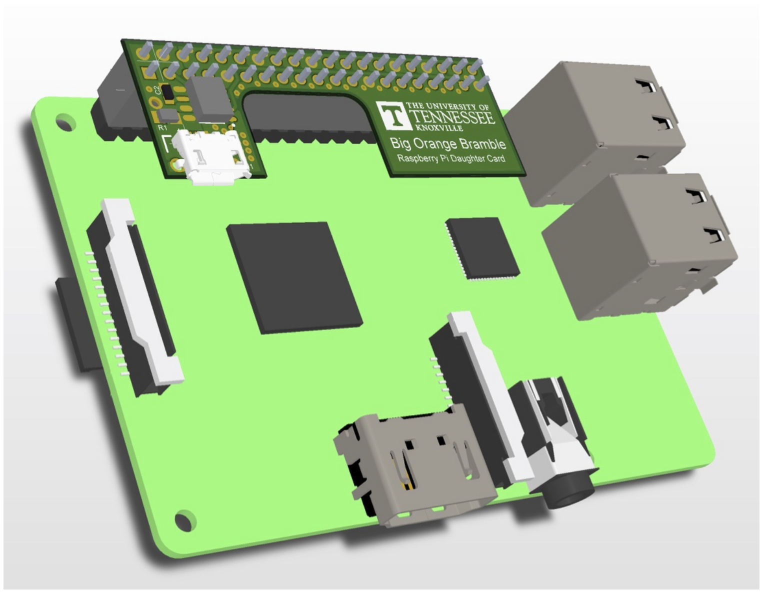


Figure 3: The GPIO connection of the Raspberry Pi 3



THE UNIVERSITY OF TENNESSEE KNOXVILLE
Big Orange Bramble
Raspberry Pi Daughter Card
R1

Figure 4: The Daughter Card 3D render

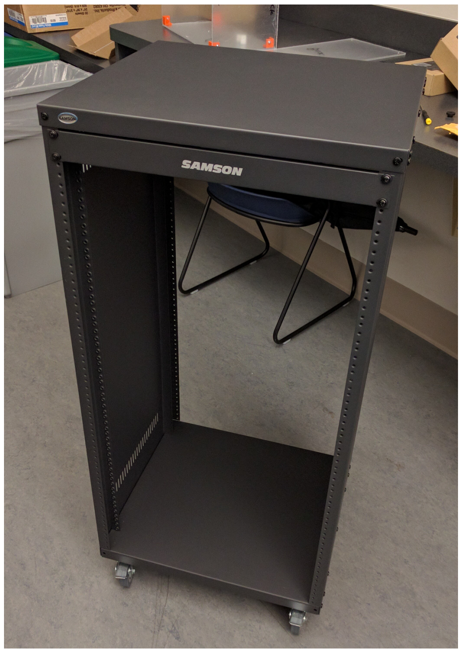


Figure 5: Rack for housing enclosure

to utilize the monitoring script (see **Section 3.2.7**). This functionality was added once the monitor node became fully operational. The second and original method was to use a wrapper Python script in which the IP addresses for each node had to be hard coded to allow for an SSH connection to each node. This was an inefficient method of reading the daughter card output. The monitoring script uses a much more efficient message passing interface via sockets. However, for the initially small number of daughter cards used (2) an SSH connection was a simple and easy to implement solution. Future work may include expanding the use of the daughter cards to monitor every node in the cluster, in which case the monitoring script will become more useful and easier to maintain over the wrapper script. The current build utilizes daughter cards on each of the worker nodes. (A complete user guide to programming and implementing the Daughter Card is located in **Appendix A.1.2**)

3.1.3 Enclosure

The enclosure was designed for the purpose of protecting the clusters from environmental threats (such as wandering hands), but the enclosure was to have aesthetic value as well. The housing of the enclosure is a large 4 ft rack, which has locking casters so that the computer can be easily moved if necessary (See **Figure 5**). Rack shelves were used to hold the network switches and power supplies, but a custom enclosure was made for the towers of Raspberry Pis (**Figure 6**).

The materials used for the custom enclosure consisted of 0.25 in sheets of Acrylic Plexiglass; in addition to 3D printed Polylactic Acid (PLA) corner braces and stainless steel hardware. The models for the 3D printed parts are shown in **Figure 7**. The Plexiglass was cut so that the enclosure would be 17 in $\times$ 8.75 in $\times$ 9.25 in. With these dimensions, there would be just

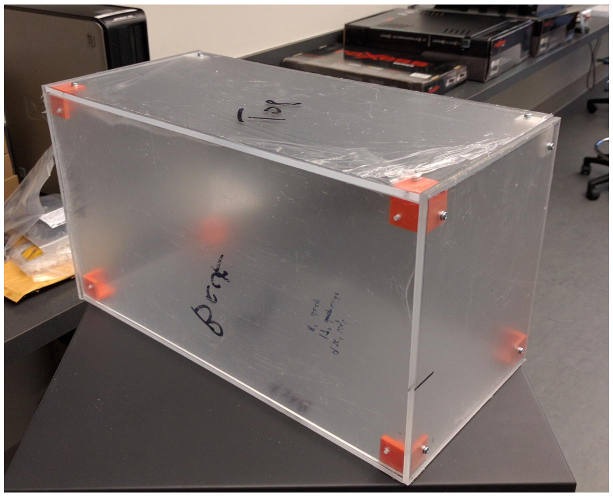


Figure 6: Custom-built Enclosure

enough room for 4 stacks of 8 Raspberry Pi units and all then necessary cabling (power and Ethernet). Cooling fans were attached to the rear of the enclosure as shown in **Figure 8**.

When laying out the enclosure, the power distribution network described in **Section 3.1.1** was routed first. This was done so that priority could be given to the safety considerations of having 110 $\sim$ 120 V AC throughout the rack. (*It should be noted that in some instances located near the power supplies, AC carrying wires are exposed and possess the* **risk for electrocution** *if care is not taken while performing maintenance on the system.*) Once the power distribution network was routed, the Ethernet switches and populated custom enclosures were added to the rack. Mechanical connections within the rack were formed with Velcro if there was enough surface area for contact or cable zip-ties otherwise. The zip-ties and Velcro cable-wraps were used to tidy the power and communication cables. The completed rack and enclosures are shown in **Figures 9 and 10**.

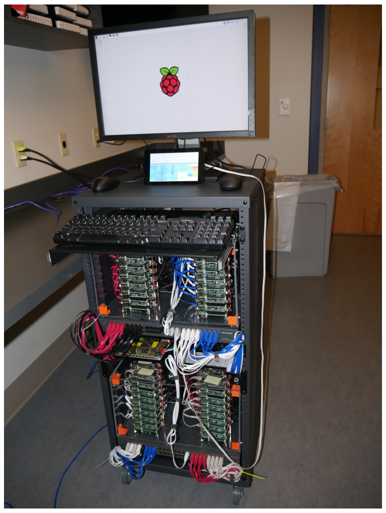
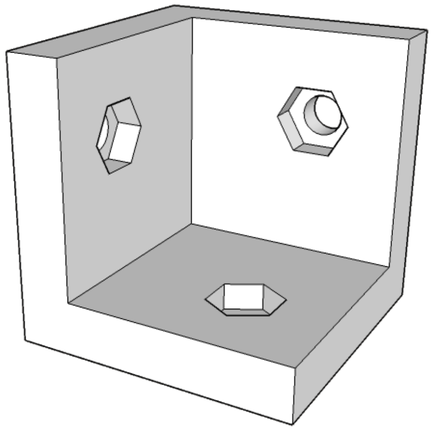
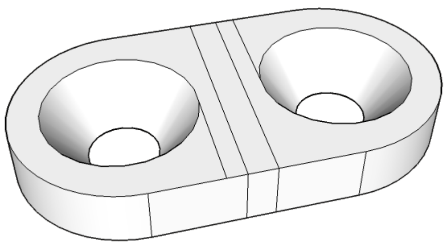
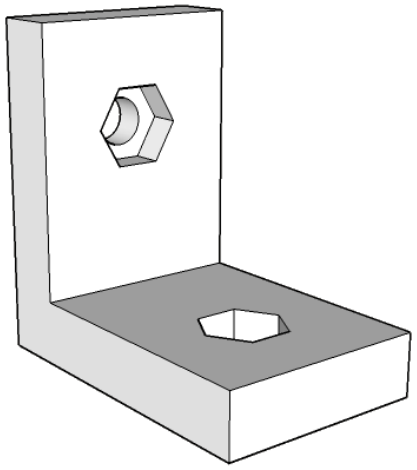


Figure 9: Front of Rack and Enclosures


(a) Corner Brace


(b) Fan Connector


(c) L Brace

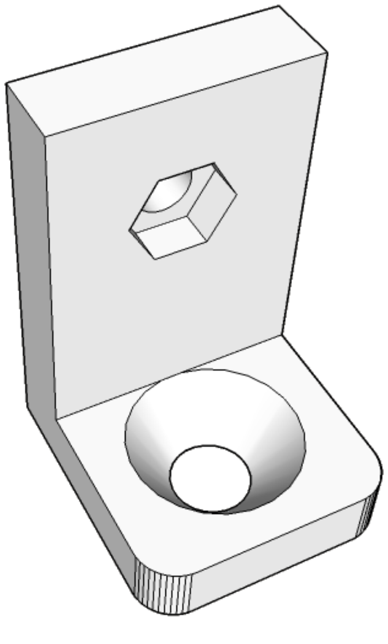

(d) Fan Brace

Figure 7: 3D Printed Parts

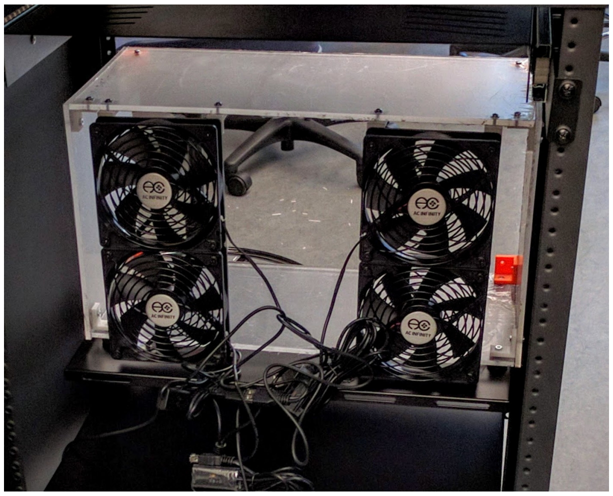


Figure 8: Rear of Custom Enclosure

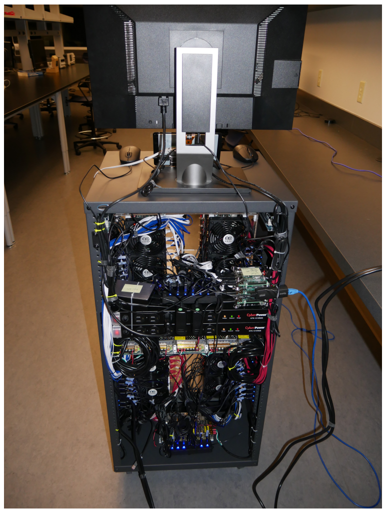


Figure 10: Rear of Rack and Enclosures

3.1.4 Monitor Node

The monitor node consists of a single Raspberry Pi mated to a 7" touchscreen display. The purpose of the node is to receive parameters from each node in the cluster, such as CPU temperature or load, and present these values to the user via an interactive GUI. The GUI offers a view of the status of all nodes simultaneously using a heatmap that represents a single parameter as well as a view that provides all details of a single node at once. Details regarding the implementation of the GUI and the methods used for receiving and processing the status of each node are discussed in detail in **Section 3.2.7**.

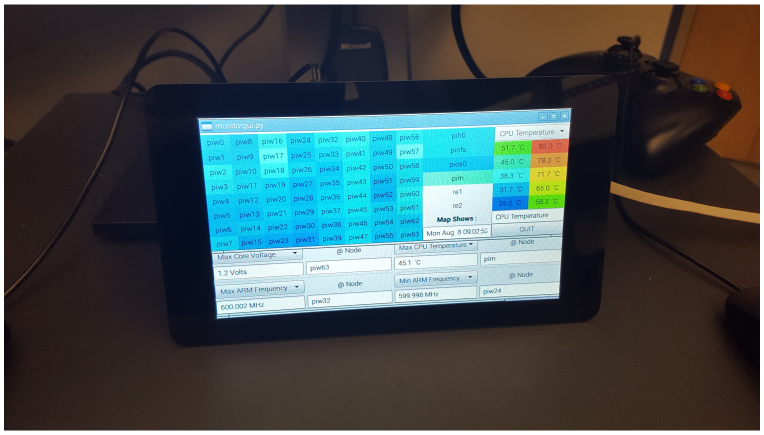


Figure 11: Monitor Node Display (front)

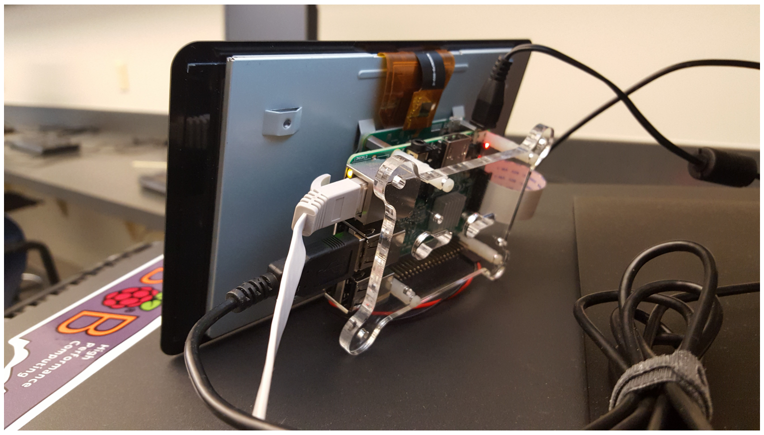


Figure 12: Monitor Node Display (back)

3.2 Operations & Systems Development

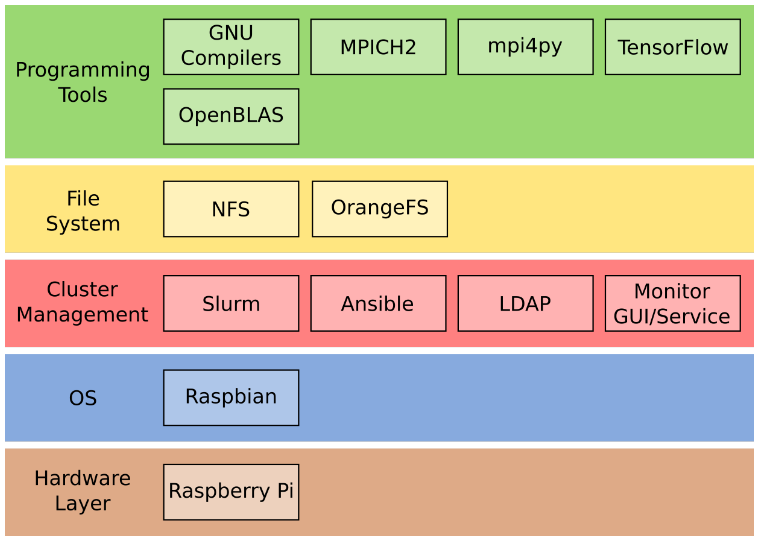


Figure 13: Software Stack

3.2.1 Raspbian

Raspbian is a fork of the common Linux distribution Debian. Specifically, for this cluster, Raspbian Jessie is being used which corresponds with Debian Jessie. It serves as a customized Linux environment optimized for the specific hardware of the Raspberry Pi board, and it includes all of the necessary drivers and hardware support for the full functionality of the Pi. However, at this time, the Broadcom BCM2837 SoC does not have support for a 64-bit kernel despite being a 64-bit processor. Instead, Raspbian Jessie utilizes a 32-bit ARMv7 kernel. This does provide lower memory utilization than 64-bit, but it also misses out on the ISA and performance improvements of ARMv8.

3.2.2 Slurm

Slurm stands for Simple Linux Utility for Resource Management.[23] Slurm is an open source system designed for highly scalable cluster management and job scheduling. Job scheduling and resource management is needed to allow multiple users to be able to use the cluster without interfering with any of the other users. A job scheduler is also needed so that the resources needed for a run can be specified and the job can be run when the resources are available. There are many cluster management systems (CMS) available, including Slurm, openSSI, and TORQUE. After looking into the other options, Slurm was chosen for a variety of reasons. Firstly, Slurm is a modern CMS and "as of the June 2016 Top 500 computer list, Slurm was performing workload management on five of the ten most powerful computers in the world including the number 2 system, Tianhe-2 with 3,120,000 computing cores."[22] Additionally, the Slurm daemon that runs on the worker nodes is very lightweight and only

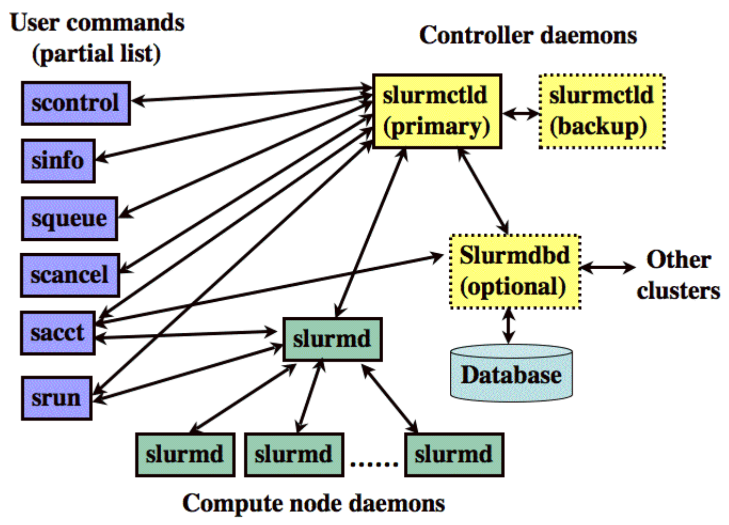


Figure 14: Slurm Architecture

runs on job start and signaling. Using a lightweight manager is particularly important on the Raspberry Pis as they do not have a significant amount of extra RAM to spare. Furthermore, Slurm does not require any kernel modifications and is relatively self-contained, which allows for easy installation. Slurm is able to provide all the functionality expected from a CMS and it can be easily installed and configured. There were no implementation challenges and Slurm was easily installed using the Debian package manager.

Figure 14 shows the architecture Slurm uses. Slurm has one centralized control manager which runs the `slurmctld` daemon. This daemon monitors the resources and work. An optional backup `slurmctld` daemon can be configured in case the main controller goes down. There can also be a dedicated database node that keeps up with accounting information for the cluster. Slurm can be used without an accounting node setup. Slurm can support an arbitrary number of worker nodes and the worker nodes can be configured into different network topologies. Each of the worker nodes runs the `slurmd` daemon.

Users and administrators interact with Slurm using the commands provided by Slurm. `scontrol` is an administrative tool that is used to view or change the state of the cluster. `sinfo` is used to report the state of the partitions and nodes in the cluster. `squeue` is used to report the state of jobs and to see jobs waiting to run. `scancel` is used to cancel a job. `sacct` is used to report accounting information. `srun` is used to run a job for real time execution. `sbatch` is used to submit a scripted batch job. `salloc` is used to allocate resources for interactive use.

3.2.3 Ansible

In order to be able to deploy configuration changes and facilitate initial setup, a method for parallel communication with worker nodes is needed. Parallel ssh (pssh) was first investigated. It allows for running ssh calls in parallel, but it lacked any built-in system management tools. The second looked into was Ansible.[1] Ansible was clearly the better tool for the job. Ansible runs over ssh connections. It can run in parallel and only needs to be installed on the head node. Additionally, Ansible is designed to deploy applications and manage systems. Ansible can be used to run ad-hoc commands to quickly run one-off commands such as copying an executable file or restarting the system. However, the real power of Ansible comes from the modular playbooks that defile system roles. Using these playbooks, different configurations can be setup and applied to different components of the cluster. One role is the headnode which sets up the configuration needed for the headnode in the cluster. Another role is worker which performs all the setup unique to worker nodes in the cluster. Other roles can define optional setups like using NFS or OrangeFS as the file system.

Ansible works using modules that define an operation to be performed. The modules are basically small programs that get copied to the target machine over an ssh connection and then run to perform a task. There are many built-in modules to perform any common action and custom modules can also be created. For example, the copy module is used to copy files, the command module is used to run a shell command, and the apt module is used to configure packages. The modules are smart and are only executed if there is a change to be made. After running a module the result is “OK” when there was no change to the system, “Changed” if a change was made to the system, and “failed” if the module execution encountered an error. Playbooks are used to organize and script the modules to be run. The playbooks are written in YAML which allows them to be stored in a format that is easy to read. The output from running a playbook is helpful to know what is going on and what has been changed. It also allows filtering on the target machines so that the playbook can be rerun only on the machines that failed to run. The team uses Ansible scripts to setup all of the nodes in the cluster and also to restart and shutdown the cluster.

Ansible was simple to setup and can be installed via either `apt` or `pip`. One caveat is that the version found in `pip` is much newer than the one found in `apt`. The group had to switch to newer version after modules we wanted were not available in the older version.

3.2.4 Security (LDAP)

One of the requirements for Slurm—and for most clusters—is to have a unified namespace for users, groups, and home areas. The cluster needed a way to synchronize accounts and to provide an easy structure for adding new users and groups. There are many ways to go about setting this up.

The most straight forward way is to keep the local files synchronized across the cluster. This can be done by writing scripts that will copy the `/etc/passwd`, `/etc/group`, etc. files to all the other nodes when a new group or user is added. Although this method is easy to setup and understand, it is limited in its flexibility and requires mass copy to be made any time the users or groups change.

A second method involves using a centralized directory such as LDAP, Windows Active

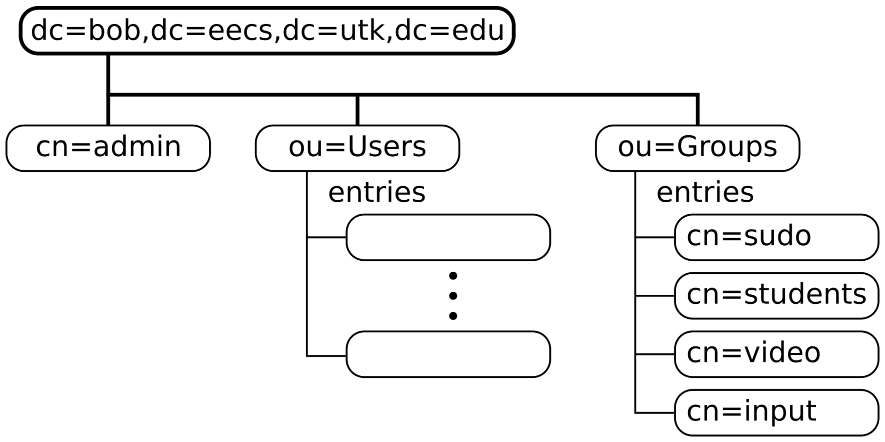


Figure 15: LDAP Stucture

Directory, or SAMBA. With this method, a uniform namespace is provided in a central directory and all users that sign in are authenticated by this directory. Using a centralized directory allows for greater flexibility and power. Their main downside is the complexity in setting up and the initial learning curve.

Of the possible methods, LDAP (Lightweight Directory Access Protocol) seemed to be best choice. LDAP is the most commonly used in enterprise Linux environments and it is designed to provide user and group data to systems. LDAP also allows changes to the users and groups without having to change anything on the clients. After initial setup, LDAP is much easier to use and allows easier changing of users and groups. There are also scripts that can be easily installed to automate the adding, removing, and modifying of groups and users.

LDAP is a standard application protocol for accessing a distributed directory over the internet. There are multiple back end databases that can be chosen; however, to the protocol the database structure is organized into tree structure with each entry in the tree having a unique identifier or distinguished name (DN). Each DN is a child of another DN with the top level being the database root. Each entry contains a set of attributes which defines the entry. Entries in the directory have different types. Some are purely for organizing the directory. Others represent POSIX groups or users. **Figure 15** shows how the LDAP structure looks. The tools for modifying the LDAP directory directly are cumbersome and only need to be used to setup LDAP. Once setup, scripts can be used to automate all the commands needed to change the users and groups. With the scripts, it is just as easy to use LDAP as it is to use the built-in commands for users and groups.

There were a couple challenges in setting up LDAP. The main challenge was that there are many tutorials available online making it hard to find a good tutorial that would work on Raspberry Pi and set LDAP up as desired. In the end, parts of multiple tutorials were used to create a method for setting up LDAP for the cluster.

Another challenge was adding LDAP users to groups that exist locally. To do this, a group with the same name and GID needs to be added to LDAP, then the LDAP user can be added to the LDAP group. Since the LDAP group has the same GID as the local group, the groups are essentially the same and the user is in the local group. If the names of the local group and the LDAP group are the same but the GID does not match, the groups will be two separate groups even though they appear to be the same. This can be confusing since users will not have permissions that it appears they should have.

3.2.5 File System

The Raspberry Pi cluster implemented in this study performs highly distributed parallel computations across four cores of 64 processors and appears to the user as a single computer. This cluster architecture, which behaves as a virtual supercomputer, is commonly known as a Beowulf cluster. As with any computing system, the organization of data storage and the control of data retrieval contributes significantly to the performance of the system. The file system has the potential to drastically improve or impede system performance. The highly parallel and distributed nature of the Raspberry Pi Cluster requires the file system to be extremely efficient and consistently reliable in storing and retrieving data to gain maximum performance. A wide variety of file systems exist and each one caters to distinct properties. The desired properties of the file system to be implemented on this Raspberry Pi Cluster included the ability to scale to larger clusters, demonstrate read and write speeds of at least 100 Mbps on small and large files, provide access, replication, and concurrency transparency to the clients, and to be compliant with MPICH. OrangeFS, GlusterFS, and NFS were considered as possible options that would meet the desired properties necessary for the Raspberry Pi Cluster. A brief discussion on the operational basics of each the three file systems and why each was considered a viable options is provided below. A further discussion as to why OrangeFS was the file system ultimately deployed on the cluster and the challenges that were encountered while bringing the deployment of OrangeFS to fruition follows.

Network File System (NFS)

Network File System (NFS) is a purely distributed file system protocol initially developed for Sun Microsystems. It has since become the standard file system in traditional Unix-based networks. Being purely distributed, the file system uses a protocol to access storage instead of sharing block level access to the storage. This access can be restricted on both clients and servers through the use of access control lists. Files are accessed using the same interfaces and semantics as local files. NFS aims to be transparent in access, location, concurrency, and failure while being heterogeneous and scalable. The most current version of NFS is NFSv4 which is designed to support clustered server deployments and has the ability to provide scalable parallel access to files distributed across multiples servers.

Raspbian Jessie, the operating system on the Raspberry Pi, uses the standard Linux kernel and is pre-installed with NFSv4. Thus, NFS only needed to be instantiated on the Raspberry Pis to be the implemented file system. NFS required no installation and minimal setup to become functional. Therefore, it was the first file system deployed on the first subset of operational nodes from the cluster. The simplicity of the NFS setup allowed for application development and power analysis to begin almost immediately. NFS was later benchmarked using the IOR bench marking platform. The results are discussed later in this report.

OrangeFS (OFS)

OrangeFS (OFS) is an object-based file system with a client-server architecture such that data is divided and distributed to one or more servers. These characteristics make it known as a distributed and parallel file system. The object-based design provides a layer of abstraction that further classifies it as virtual. OrangeFS is designed to focus on metadata and data distribution as opposed to disk storage management. Files are stored in objects that can be accessed in parallel and may reside on multiple servers. The servers provide a metadata service and a data service within one process. The metadata service sends and receives information about the directories and logical files in the network. The data service sends and receives data for the objects stored on the server.[16] The computation/client nodes in the cluster are configured with an interface designed to specifically communicate with the servers. This interface finds the location of a particular file's objects using the metadata service and then accesses the file's objects using the data service.

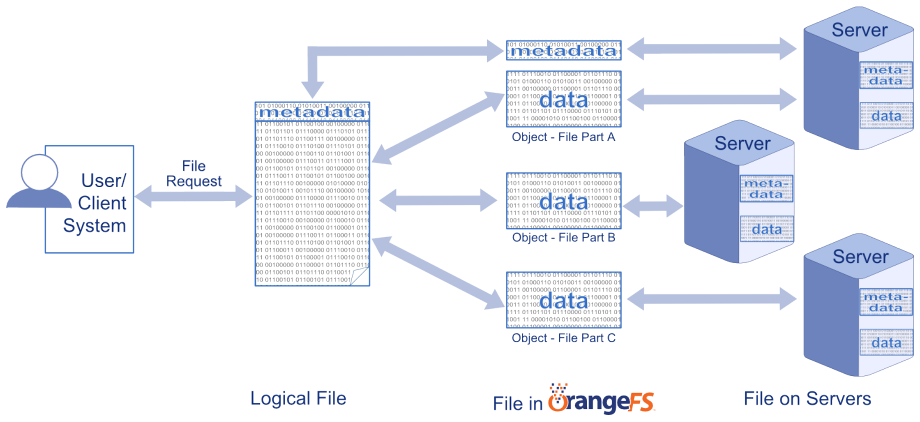


Figure 16: How OrangeFS Works
[16]

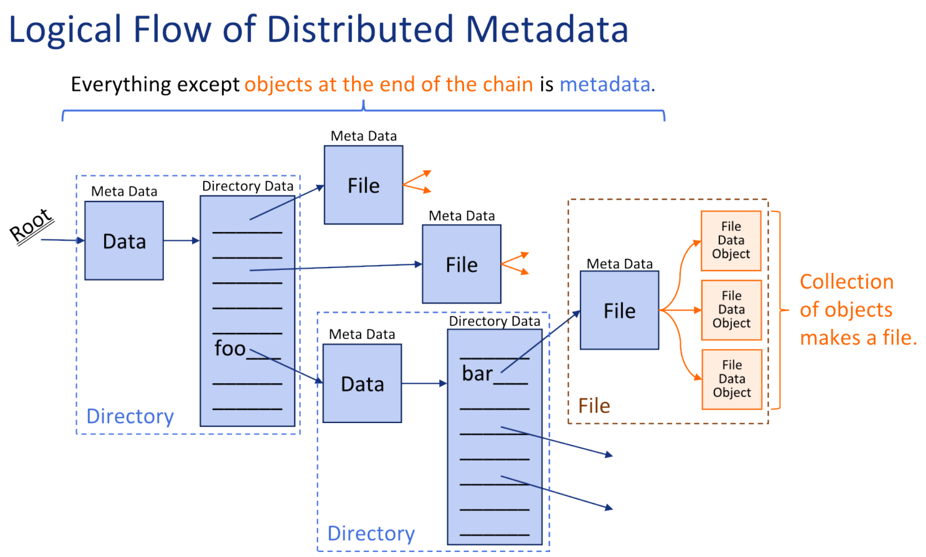


Figure 17: File in OrangeFS
[16]

OrangeFS was thought to be a viable option because it claimed to provide the sought after access, location, and concurrency transparency, MPI-IO support, and the ability to scale to large numbers of clients and servers due to it's object-based, multi-server structure and modular architecture. OrangeFS was well documented and claimed to have the ability to implement Hadoop MapReduce without HDFS. Implementing the Hadoop Framework was another goal for the Raspberry Pi Cluster in this study. For these reasons, OrangeFS was chosen to be implemented on the cluster. After successful installation, it was benchmarked using the IOR benchmarking platform. The results are discussed later in this report.

GlusterFS

GlusterFS is a distributed file system defined in user space. It eliminates metadata and claims to scale linearly as the number of servers is increased. The functionality of GlusterFS comes from it use of translators. Clients are stateless and do not communicate with each other. GlusterFS relies on an elastic hashing algorithm rather than a metadata model.[7] It was not determined if GlusterFS supports MPI-IO, as GlusterFS documentation was not abundant. Due to time constraints and lack of reliable information, GlusterFS was only researched and not implemented on the Raspberry Pi cluster.

Implementing OrangeFS

The implementation of OrangeFS on the Raspberry Pi cluster began before the cluster was a fully functional system. The Raspberry Pi Cluster began as two distinct small clusters each with one control node, eight worker nodes, and one storage node mounted to an external hard drive. One cluster was labeled the development tower and the other cluster was labeled the application tower. The entire implementation and testing of OrangeFS took place on the development tower. While OrangeFS was in the initial stages of implementation, the configuration manager, job scheduler, and directory access protocol were also in a developmental state on the development tower as well. The development tower was also being used for investigating network and power issues during this time.

The online documentation for the implementation of OrangeFS appeared to be straightforward so the team began following it by previewing the system requirements, security options, and the OrangeFS configuration file. According to the OrangeFS 2.9 Documentation, the OrangeFS build and server nodes require a Linux system and support the most common Linux distributions. The installation of several nonstandard Linux software packages are necessary for the build. The build will create the executable software to be distributed to all the servers and clients. The OrangeFS client interface can be configured such that it is compatible with several different operating systems including Ubuntu.[16] Based on this information, it was thought the Raspbian Jessie operating system used by the Raspberry Pi 3 would be supported by OrangeFS as it is similar to Ubuntu. However, a quick internet search of discussion forums revealed others had encountered issues compiling OrangeFS 2.9 on the Raspberry Pi. This caused some hesitation in the decision to implement OrangeFS, but the appeal of OFS was so great it was decided to move forward with the implementation in hopes a workaround could be found if we encountered an issue. Initially, a key-based security scheme was chosen to be incorporated into the build of the file system.

First, the additional required packages were installed on the cluster and then an attempt to build the executable software was made. The development tower's storage node was chosen to be the OrangeFS build node. More servers would be added if necessary after the initial implementation on the development tower. The eight worker nodes on the development tower were to be OrangeFS client nodes. After overcoming several challenges, which are discussed in the following sections, the build on the storage node was ultimately a success.

As directed by the installation guide, the configuration file was generated with the OrangeFS default settings using the executable `pvfs2-genconfig` program created during the build. This interactive program walked the system administrator through the configuration settings by asking for some basic information including the communication protocol, port number, desired locations for the server to store the the data and metadata, and the location of the log file.[16] A mount point for the external hard drive was created on the server node. The location of the mount point was given to the configuration program as the location for the storage of the data and metadata for OFS.

Before distributing the OFS executable software to the all the client nodes, an attempt to run the server was made using the executable pvfs2-server program which took a required parameter of the previously generated configuration file. This program created the storage space for the server and started the server. The output messages from this executable indicated the storage creation and server start was a success. The executable software and configuration file was then distributed to the client nodes.

The client nodes were also successfully built and configured after working through some minor challenges. A file was created on each of the client nodes to direct the file system through the server URL to the configured location of the data and metadata storage on the server. The executable pvfs2fuse, created during the build and located on each client, was then run to start the client process and finally mount OrangeFS on each client. Once the clients and the server were running, a few simple commands were run to ensure that the file system was in fact operational. These tests confirmed that the clients were able to read and write to the OrangeFS server. The final step before declaring the file system a success was to add MPI-IO capabilities. The clients were then reconfigured to include the necessary OFS configuration parameters to support the message passing interface. MPICH 3.2 was also recompiled to enable linkage to the OrangeFS file system when compiling and running applications.

Implementation Challenges

As we had found in our initial research, OrangeFS had difficulty compiling on the Raspberry Pi. The first step in the installation required a Makefile be generated using a configure script provided in the download. The configure script took several parameters which dictated where the portable directory was to be located, the path to the Linux kernel, and the security options to incorporate into the build.

The configure script successfully generated the appropriate modules and Makefiles necessary to make and install the build, but when the make command was run, several of the make modules failed to compile. The error messages indicated that there were errors in some function calls located in the `pvfs2-kernel.h` and the `pvfs2-utils.c` files. After quite some time and no successful workaround, the team reached out Dr. Walt Ligon and the OrangeFS developers at Clemson University. We were told that currently the kernel module for the clients could not be built on the Raspberry Pi because there were function calls being made that were designed for Intel processors and not ARM processors. A possible solution suggested by Dr. Ligon was to try using the FUSE client module to allow OFS to work outside of the Linux kernel in userspace. The Clemson team did not claim this was in fact a verified fix, but they believed it had a good possibility of working. Following Dr. Ligon's advice, we ran the configure script omitting the kernel directive. This cleared the build errors and we were able to make and install the executable software to distribute to the cluster without incorporating the Linux kernel module into the build.

The next steps of the installation, which included distributing the build directory to the clients, running the configure script including the FUSE parameter, creating the OFS storage space on the server, and running the server. This all succeeded with no error messages or indication of failures. However, when mounting the OrangeFS file system by running the `pvfs2fuse` command on the clients, an error message occurred indicating there was a failure to initialize any BMI methods for the URL of the server and the protocol was not available. This error message and the fact that the pvfs2-server command printed a message stating it started lead us to believe the issue was with the client node configuration and not with the server. Much time was spent working with the client node trying to resolve the error. Eventually a deeper look into the server was made. It was discovered that if the `-d` flag was set when starting the server with `pvfs2-server` command, the process stayed in the foreground and printed messages to the terminal instead of running in the background and printing messages to a log. This resulted in more informative messages which indicated the server had not in fact started and had conflicts. The messages indicated there was no ServerKey definition in the configuration file and the security module could not be initialized. This was quite puzzling because prior to resolving the kernel module issue and building the executable software, the choice to include security in OFS was foregone. The security option had not been included in the build attempts for quite some time and the Make clean command was run after every failed build attempt of the Linux kernel module. However, it appeared that there was still a remnant from the key-based security scheme from weeks prior. To resolve this issue, all OrangeFS files and directories were deleted from all the nodes and the installation procedure was started fresh with no kernel module and no security options, but with the FUSE option set. This allowed for the build node to build successfully, the server node to create the storage space, the server process to start successfully, the clients to be configured successfully, and the clients to mount to the OrangeFS file system through the FUSE interface successfully.

The last step of declaring OFS a success on the development tower was to incorporate MPICH into the build of the clients. The problem that occurred here was not so much with OrangeFS, but with the initial setup of the cluster. It was thought the cluster was running MPICH and not OpenMPI. When testing the basic functionality of the newly compiled MPICH configuration which enabled OFS, it was discovered that OpenMPI was inadvertently installed on the cluster during early setup and was set as the default message passing interface. The update-alternatives command was run to change the default message passing interface from OpenMPI to MPICH. After the update, the functionality of MPICH with OrangeFS was verified with a few very basic MPI commands. The implementation of OrangeFS and MPICH were declared a successful implementation on the development cluster.

Several other challenges were encountered when OrangeFS was finally deployed on the full 64 node cluster. These challenges were more network based challenges and not OrangeFS problems. The deployment of OrangeFS exposed the network problems and allowed successful solutions to be implemented. The progression of events is discussed in the following paragraph.

Testing the OrangeFS system had previously revealed that one storage node was sufficient to allow the maximum throughput offered by 100 Mbps Ethernet when the cluster consisted of only eight client nodes as shown in **Table 8** and **Table 9**. However, when we scaled the system to 64-four core client nodes, one storage server was no longer sufficient. A second server node was introduced to the cluster. This server node had been previously configured as a headnode on the development tower. This new server node was thought to have been completely void of all previous headnode functions and reconfigured as only a server node. However, a DHCP service some how remained active on this node and it was distributing the same IP address to the storage device handling the NFS file system containing user accounts and the initial OrangeFS storage device. Testing the performance of OrangeFS across all 64 nodes required access to both of the storage devices, allowing the conflict to expose itself. This conflict caused the NFS storage node to fail on one occasion and the OrangeFS storage node to fail on a second occasion, bringing the cluster to a halt each time. Once the cause of the storage failures was discovered the resolution was to remove the second OrangeFS server from the cluster to completely reimage and reconfigure it as a server node. This resolved the issue.

3.2.6 Networking

All of the Raspberry Pis are connected to each other via an Ethernet switch. Each of the devices needs and easy way to communicate with each other. There are two primary methods to set up the IP addressing of the devices. The first approach is to assign each of the Raspberry Pis with a static IP address. This method has low overhead but requires that each Pi is configured individually by hand.

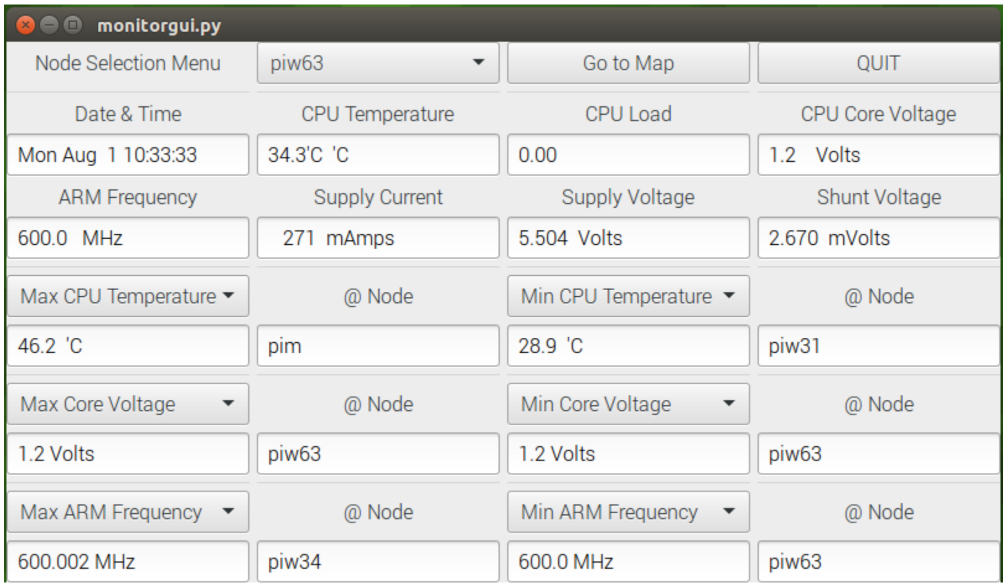
A second option is to use the Dynamic Host Configuration Protocol (DHCP) to assign IP addresses dynamically. With this method, a DHCP server would run on the headnode and it would assign an IP address to the clients when they are connected to the switch. The DHCP server can be configured to assign a specified IP address to a certain MAC address. Using this method of IP assignment allows all of the networking configuration to be handled on the head node. It also makes adding new nodes much easier. When you plug in a new node, the DHCP server provides it with an IP lease. By looking in the leasing file, the MAC address of that device can be found. The desired IP address for that device can then be specified in the DHCP configuration. This way that device will always be assigned the same IP address. This configuration method is powerful when coupled with the hosts file. The hosts files can map a name to an IP address. Using this method it is now possible to reference all the nodes in the cluster by a human readable name instead of its IP address. The DHCP method also makes it trivial to swap out compute nodes between clusters. Once the DHCP configuration file is setup with the node's MAC address, the node can be moved around to different clusters and it will be setup with the correct IP address and it can be accessed by its name in the hosts file.

In order to make the installation of packages easier on the worker nodes, it is also useful to set up network address translation on the headnode to allow the worker nodes to connect to the internet via the headnode.

In order to secure the cluster, the headnode is setup to only allow ssh connections from the outside network. This protects all of the services running on the local network. Additionally we have `fail2ban`[5] setup which will blacklist any IP address that fails to login 5 times.

3.2.7 Monitoring

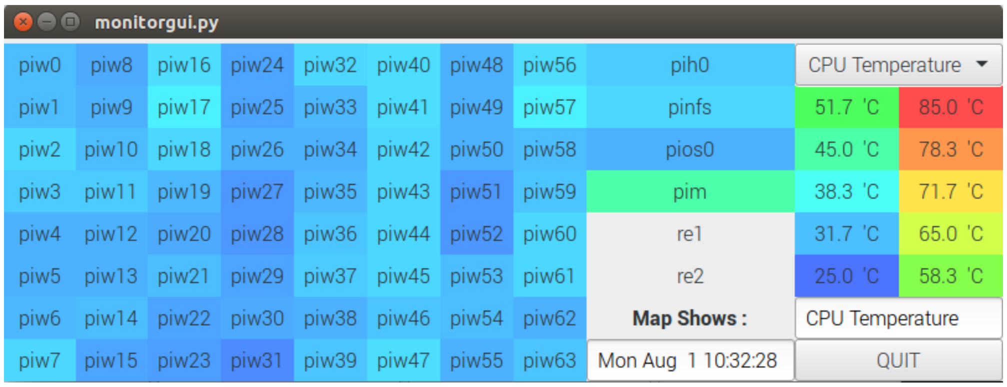
In order to easily monitor the status of the Pis, a monitor GUI was created to display information about the current status of the nodes in the cluster. A service runs on each of the nodes collecting the data. Each of the monitor services sends status information to the monitor Pi via UDP packets. Once collected on the Monitor Pi, the information is saved to a comma separated value file (CSV format) where it can then be read and displayed in the GUI. The GUI has been written using Python, Gtk and Glade which consists of two pages. In the first page, the user is able to view the numeric values of different parameters such as CPU temperature, CPU core voltage, CPU load, and CPU frequency of each node as well as supply current and voltage of nodes equipped with daughter cards in addition to the maximum and minimum of the aforementioned parameters. The second page consists of a color-coded map of the above-mentioned parameters for all nodes so that the user can have a better understanding of what is happening on all nodes in the cluster simultaneously.



monitorgui.py

Node Selection Menu	piw63	Go to Map	QUIT
Date & Time	CPU Temperature	CPU Load	CPU Core Voltage
Mon Aug 1 10:33:33	34.3'C 'C	0.00	1.2 Volts
ARM Frequency	Supply Current	Supply Voltage	Shunt Voltage
600.0 MHz	271 mAmps	5.504 Volts	2.670 mVolts
Max CPU Temperature	@ Node	Min CPU Temperature	@ Node
46.2 'C	pim	28.9 'C	piw31
Max Core Voltage	@ Node	Min Core Voltage	@ Node
1.2 Volts	piw63	1.2 Volts	piw63
Max ARM Frequency	@ Node	Min ARM Frequency	@ Node
600.002 MHz	piw34	600.0 MHz	piw63

Figure 18: Monitor GUI



monitorgui.py

piw0	piw8	piw16	piw24	piw32	piw40	piw48	piw56	pih0	CPU Temperature	
piw1	piw9	piw17	piw25	piw33	piw41	piw49	piw57	pinfs	51.7 'C	85.0 'C
piw2	piw10	piw18	piw26	piw34	piw42	piw50	piw58	pios0	45.0 'C	78.3 'C
piw3	piw11	piw19	piw27	piw35	piw43	piw51	piw59	pim	38.3 'C	71.7 'C
piw4	piw12	piw20	piw28	piw36	piw44	piw52	piw60	re1	31.7 'C	65.0 'C
piw5	piw13	piw21	piw29	piw37	piw45	piw53	piw61	re2	25.0 'C	58.3 'C
piw6	piw14	piw22	piw30	piw38	piw46	piw54	piw62	**Map Shows :**	CPU Temperature	
piw7	piw15	piw23	piw31	piw39	piw47	piw55	piw63	Mon Aug 1 10:32:28	QUIT	

Figure 19: Monitor Map

3.3 Standard Packages

(List of packages provided by default, dependencies for applications/frameworks)

- Vim
- MPICH2
- Xbox driver
- Libglew-dev
- Sshpass
- Libav-tools
- Libatlas-base-dev
- Libopenblas-base
- Libmpich-dev
- Gfortran
- Python-pip
- Python-dev
- Python-mpi4py
- Bison
- Flex
- libssl-dev
- gcc
- make
- libfuse-dev
- autoconf

3.4 Frameworks and Tools Development

3.4.1 Hadoop

Hadoop is a framework developed by Apache that is utilized for various types of processing on large data sets. Hadoop is typically implemented by utilizing its own distributed file system partition (HDFS) to increase bandwidth across a cluster. Hadoop is open-source and contains four basic modules: Hadoop Common, HDFS, Hadoop YARN, and Hadoop MapReduce. Hadoop Common is a set of libraries and utilities used by the other three modules. Hadoop YARN manages the resources of the cluster, including job scheduling. Hadoop MapReduce is a resource for parallel processing over the large data sets. Because of Hadoop's distributed nature, it was a clear choice of framework to install on the cluster and run some example tests using MapReduce to determine Hadoop's efficiency. Since there were two different file systems that were undergoing testing at the same time, attempts to install Hadoop were made on a small development cluster of eight nodes using the OrangeFS file system and the main cluster of eight nodes using NFS file system.

Hadoop on NFS

During the installation of Hadoop on the NFS server, our group encountered several issues. The version of Maven installed on the cluster had seemingly non-deterministic behavior, failing to complete the installation on one attempt, but claiming a successful installation on the next attempt when the trying to reacquire the error messages from the previous attempt. After this "successful" installation, several key components of Hadoop were seemingly missing from the installation, namely the configuration directories for all of the Hadoop Components. Our attempts to recreate the configuration directories were unsuccessful, either through reinstalling Hadoop or through copying a successful build from another system. Following these unsuccessful install attempts, efforts were focused on installing Hadoop on OrangeFS.

Hadoop on OrangeFS

The OrangeFS client nodes can be configured to run Apache Hadoop version 1.2.1 or 2.6.0 in conjunction with the OrangeFS file system to replace Hadoop's HDFS file system. The OrangeFS documentation states the process of creating an OrangeFS-Hadoop client is four steps: install system software, configure Hadoop to use OrangeFS, copy the client build system software to the other clients, and then start Hadoop on the clients.[17] While seemingly straightforward, the creation of OrangeFS-Hadoop clients has yet to be realized on the Raspberry Pi cluster.

According to the OrangeFS community, for a file system to link with Hadoop, it must appear as a file system native to HDFS, and the file system must provide a working implementation of the org.apache.hadoop.fs.FileSystem class. To accomplish this, OrangeFS uses the abstracted file system class found in MapReduce and a JNI shim found in the OrangeFS client.[17] However, after configuring the OrangeFS client as instructed in the OFS documentation and building the JAR file such that it contains the appropriate class definitions from the JNI shim and distributing it to the other clients, Hadoop MapReduce fails. A Java exception is thrown stating the org.apache.hadoop.fs.FileSystem class cannot be found. However, a search of the file system produces the path to the class that is said to not be found. It is thought that the `HADOOP_CLASSPATH` environment variable is being rewritten at some point during the configuration process which ultimately results in an invalid class path and the class in fact not being found or the `system.xml` file read by Maven is incorrect. A dependency issue also occurred initially, but it was resolved after discovering the `core-site.xml` file was written with an error in one of its properties. Time did not permit for the confirmation or elimination of the redirection of the `HADOOP_CLASSPATH` environment variable or the exploration of the `system.xml` file to be possible causes of the java exception.

3.4.2 TensorFlow

TensorFlow[4], an open source software library for numerical computation using data flow graphs, is a newly developed tool by Google. It was developed mainly for the purposes of conducting machine learning and deep neural networks research.

One issue with deep learning is that it needs a massive amount of data to train a good model, which is time consuming. A typical example is a convolutional neural network (CNN). CNN normally requires millions of images to converge to a good model. This process is done through stochastic gradient descent, as there is no way to fit the enormous amount of training data into memory and train the model. The training dataset is divided into small batches, and the model is trained using one batch at at time.

Not only will the training dataset be too large to load or train at one time, the model itself will potentially be too large to fit on a single machine. One characteristic of deep learning is to exploit the "depth" of the model to extract more meaningful features to perform computer vision-related tasks. For example, the latest residual network would have over 10 million parameters to train with 1,001 layers, and the deeper the network goes, the more parameters will have to be trained. It is entirely possible that a single machine cannot fit all the parameters, especially considering the hardware limitations of the Raspberry Pi. It is therefore necessary to consider a distributed structure of the model in the future, but our immediate focus is on “data parallelism”, i.e., one single node can handle the entire model. There are several ways to train a deep learning model on a distributed system. The simplest approach is to share all the model parameters across all workers while paralleling data and gradient updates. In a synchronized fashion, batches of data are used to train the model simultaneously. The update on the parameters are averaged based on all the working nodes for one iteration; while in an asynchronized fashion, every worker node will update the parameters of the model once it finishes the training process. Thus the asynchronized approach is more flexible comparing to its counterpart.

By using GRPC as the communication protocol to perform the message passing, Distributed TensorFlow is able to pass messages and updates. The distributed TensorFlow model normally consists of parameter servers and workers, among which parameter servers store the parameters of the model while workers perform the computationally intensive tasks.

The asynchronized approach based on LeNet as seen in **Figure 20** was implemented, containing two convolution layers, two pooling layers, and two fully connected layers.

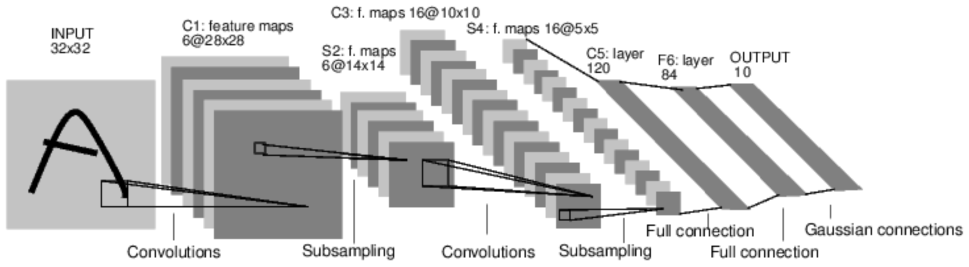


Figure 20: LeNet Structure

The dataset used was the famous MNIST dataset which is used for handwritten number recognition. Slurm was used to manage node resources. Because TensorFlow needs to specify the specific nodes for specific task (ps or worker), the nodes were hard-coded in the script to make sure each task ran on its designated node.

3.5 Applications Development

3.5.1 SPH

Provided by the Tiny Titan project at ORNL were several demonstration applications, the first of which is the Smoothed Particle Hydrodynamics (SPH). This application performs a distributed fluid dynamics simulation with user input providing perturbations in the system. One node performs rendering and task distribution, while the remaining nodes perform the particle interaction calculations. Parameters including gravity, viscosity, density, pressure, and elasticity are configurable inside the application, while parameters like the number of nodes used and the size and count of particles can only be configured in the source code.

The application has been altered from its original state to compile and run under BOB's cluster configuration without errors or warnings. MPICH provides the MPI support between workers and the master node, while Slurm handles node allocation and reservation. One major consideration when adapting SPH to BOB was that of the master node. When using Slurm under normal conditions, all requested nodes are from the pool of worker nodes. However, SPH requires access to both input peripheries and a display for the user. Most cluster nodes would not have access to these utilities, so a launch script manages a Slurm job allocation call for N nodes, manually launches the SPH executable across $N + 1$ nodes where the final node is the host node itself, and cleans up the Slurm allocation and local files. This workaround allows the host node's peripheries to be used at the same time as Slurm's reservation.

3.5.2 PiBrot

Another application provided by the Tiny Titan project is the parallel Mandelbrot set race (PiBrot). This application demonstrates the relative speedup that running an application can achieve when running a process in a massively parallel manner. When given N nodes, one node is assigned the MPI rank of 0 and handles the communication and rendering of an image of a Mandelbrot set by the worker nodes. One node computes the set on its own, relaying the information to the master node to be rendered on the left side of the screen. The remaining $N-2$ nodes perform the same calculation in parallel, relaying their set information to the right side of the screen. Due to bandwidth and CPU power limitations, the number of lines that can be rendered per second limits the operation speed of the cluster, but the relative speed is still clear as the right side of the screen fills roughly $N - 2$ times faster than that of the left side.

Similar to SPH, PiBrot was designed to be run on the original Tiny Titan cluster. As a result, the same workaround must be used to allow both PiBrot and Slurm to work together to provide the visuals for the user. Additionally, the launch script for PiBrot includes the ability to specify the number of nodes to use for the calculation, which allows for easy comparison between a varying number of parallel processes.

3.5.3 Parallel Pi

The degree to which the mathematical constant Pi can be accurately calculated is often used to demonstrate the performance of modern computers. One of the simplest methods of approximating the value of Pi is through the use of a convergent series. This method is particularly applicable to the testing of BOB because subsections of the chosen summation can be assigned to any number of available worker nodes in parallel, as the resulting sum of each subsection is independent of the others. Therefore, each worker node should be able to simply compute the sum of their subsection and return the result to the master node, where all results are then summed for a final approximation of Pi. If the cluster is operating as intended, a linear increase should be observed in runtime performance of the approximation as the number of available worker nodes increases.

Initially, BOB's Pi approximation application utilized the Leibniz formula for calculating Pi:

$$\pi = 4\sum_{i=0}^{\infty} \frac{(-1)^i}{2i+1}$$

However, the formula converges extremely slowly, requiring the processing of billions of terms to achieve a correct approximation of only 10–12 digits. Eventually, the Bailey-Borwein-Plouffe formula, originally discovered by Simon Plouffe, was chosen to replace the Leibniz formula due to its much faster convergence rate.

$$\pi = \sum_{i=0}^{\infty} \frac{1}{16^i} \left(\frac{4}{8i+1} - \frac{2}{8i+4} - \frac{1}{8i+5} - \frac{1}{8i+6} \right)$$

The BBP formula, in combination with Python's decimal module, which allows users to specify floating-point precision, resulted in an approximation of Pi that was accurate to thousands of decimal places in a fraction of the time required for Leibniz. Performance results will be discussed in detail in the Testing and Results section below.

3.5.4 Monte Carlo Simulations

A Monte Carlo simulation is a method of running an application or program with random (pseudo-random) values for a variable (or set of variables) in order to produce and analyze results. This type of simulation is useful for many applications, specifically for those in physics and mathematics which involve probabilistic variables. The Monte Carlo application was developed at the request of a faculty member who wished to run an input model in parallel in order to obtain an output set much faster than running the same model on a single processor. The application can be run from either the command line or a supporting GUI and allows the user to specify the input model, total number of runs, number of nodes, and maximum run time. The application supports models written in C/C++ and Python at this time.

GNU Parallel

Initially, GNU Parallel was installed in order to launch a single Monte Carlo job to run each task in parallel. GNU Parallel works with SLURM to handle any interrupts during the runs, as well as appropriately assign jobs to nodes. The idea was that one script could be created based on the user's input parameters, and using GNU Parallel, a single command could parallelize and manage all iterations. Although this seemed to be a straightforward approach, it failed to work properly during testing. For example, a batch script was created invoking the `parallel` function from GNU Parallel, but after the script would run once properly for a user, it would then not work again. In fact, all scripts invoking GNU Parallel afterwards would no longer work for that user. An example of this script is shown below. It should also be noted that in the testing of this method, errors occurred from the Perl code present in this script. The Raspberry Pis were all using the Great Britain language packages which were the default settings during setup. Therefore, the United States standard unicode had to be updated across all nodes.

```
#!/bin/bash
#SBATCH -n 12
#SBATCH -o ot.txt
#SBATCH -t 12:00:00

srun="srun -n1 -N1 --exclusive"
parallel="parallel --delay 1 -j 30 --joblog test.log --resume"
$parallel "$srun ./a.out"
```

Figure 21: An Example of a Batch File Using GNU Parallel

Multiple Runs in Single Batch File

Rewriting the Monte Carlo application to create multiple runs in a single batch file is the current solution to the GNU Parallel issues. This implementation divides up the total number of runs into chunks according to the number of available cores, creates an `srun` call per chunk, and then submits the file as a job. Each `srun` appends output to a single file utilized by all batches specified with the `--output` flag. This is the same file that the user specifies either on the command line or in the GUI for output. The output file and the standard error file will be stored in the same directory that the script is submitted. An example of a batch file which runs 100 simulations on 8 CPUs is shown below. While this method seems more brute force than the GNU Parallel solution, SLURM should still manage any interrupts. The output file will see no race conditions between `srun` calls because the previous `srun` will end before the next one is started. This would have been an issue if multiple batch files were created to append to the same output file.

```
#!/bin/bash

#SBATCH -N 8
#SBATCH --job-name=montecarlo_dice_roll
#SBATCH --output=dice_output.txt
#SBATCH --open-mode=append
#SBATCH --cpus-per-task=1
#SBATCH -e montecarlo_dice_roll_err.txt
#SBATCH --time="00:00:30"

srun -n 4 python dice_roll.py
srun -n 32 python dice_roll.py
srun -n 32 python dice_roll.py
srun -n 32 python dice_roll.py
```

Figure 22: An Example of Multiple Runs in Single Batch File

3.5.5 Numeric Integration

In mathematics, a simple way to estimate the area under a curve is to use Riemann sums. Riemann sums divide up a specified section of a function's domain by a set width and use rectangles to calculate sections of the total area. There are several types of Riemann sums: left, right, and middle. Examples of each type are shown below in **Figure 23**. Riemann sums can be easily parallelized because each subsection of the domain can be given to a different node which can then calculate that section's area. The master node can then sum the results from each node to produce a final total for the area. The numeric integration program on BOB utilizes right Riemann sums, though this could easily be adapted to left or middle by changing what x value is passed to the function during area calculation. The following steps show how the Numeric Integration application functions in more detail.

1. User inputs function, domain, and # samples.
2. MPI gets # available cores. Parallelization begins after this step.
3. Width of each rectangle is calculated.
4. Domain start and end points for the previous rectangle calculated for current core.
5. Areas are calculated for each sample in current core.
6. MPI sums and reduces all areas into one variable.
7. Error is calculated using scipy.integrate.quad().
8. Results are displayed to `stdout`.

The challenges for this implementation were few since it is a short and straightforward program. Ensuring that the correct libraries were installed across all nodes on BOB was the only obstacle, which was easily fixed. It should be noted that this program was adapted from the `riemann-parallel.py` Numeric Integration application listed on TinyTitan's GitHub page (`https://github.com/TinyTitan/Numeric-Integration`).

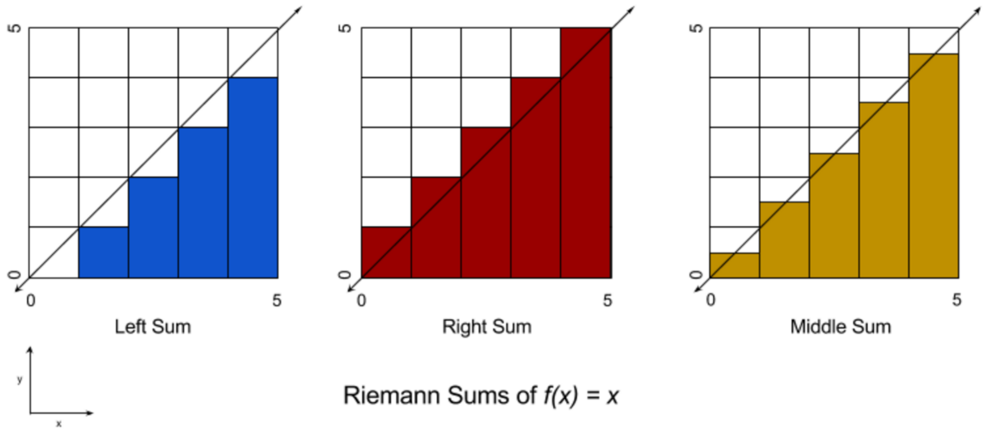


Figure 23: Riemann Sums

3.5.6 DANNA Evolutionary Optimization

DANNA, or Dynamic Adaptive Neural Network Array, is a neuromorphic computing architecture developed by the University of Tennessee.

Neural networks are too complex to feasibly be programmed by hand. DANNA uses evolutionary optimization to find solution networks to problems.

The input neurons and output synapses are affixed a priori, and then a population of randomly programmed networks is generated. For a given application, a set of training instances are executed on each network in the population, and the success of each network is quantified by a fitness function. The best networks are then chosen from the population, and used to generate the next population via mutation and crossover operations.[14]

To run on multiple nodes, each node has to be a sub-population, sharing the best networks they find with each other. With multiple nodes running EO, one may search the solution space much faster and find a solution network faster.[20]

3.5.7 Fire Dynamics Simulator (FDS6 by NIST)

Fire modeling and fluid dynamics simulations are traditionally intensive computing applications due to the large nature of both the physical structures, such as whole houses or buildings, and the fluid dynamics calculations of fire flow. Because of this, new computing methods and architectures are worth looking into to determine whether the complexity can be lessened in order to reduce run time, particularly in situations where time is a factor, like life-threatening circumstances or lawsuits.

The National Institute of Standards and Technology (NIST) has developed an open source fire simulation software fire dynamics simulator (FDS) which reduces computational and time complexity by using a technique which divides sections of the computation into meshes. A mesh is a section within a domain, e.g. a room or building, that is made up of rectilinear volumes. Then each mesh is divided into cells, the number of which depends on the specified resolution for the simulation. (FDS User Guide) Each mesh can then be assigned to multiple processors in parallel, which can significantly reduce the execution time necessary to create a complete model. The figure below shows previous research performed by C. Salter from Hoare Lea with an FDS benchmark over multiple cores on Amazon's EC2 cloud computing resources.

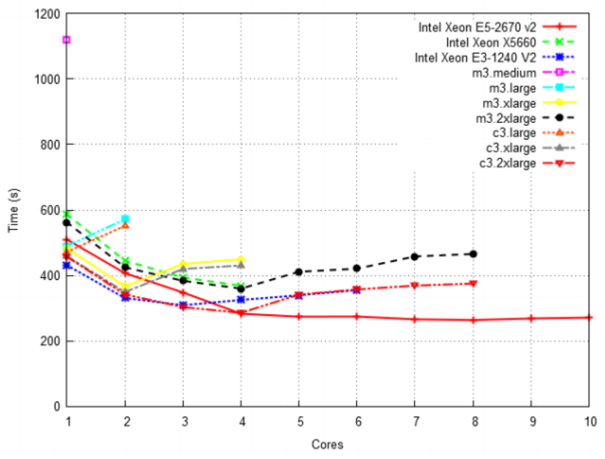


Figure 24: Results of bench1.fds Using OpenMP (on AWS) from Salter[19]

"This shows that, in general, as you add more cores, the time taken to run the model decreases, something that is validated by the modeling conducted by the developers of, as discussed earlier. The developers state that the time taken decreases by about 50% with about four cores for the model. There is a point reached where the addition of more cores actually increases the time taken to run the model - this can clearly be seen in the case of the m3.large instance, where the addition of a second core actually decreases the speed."[19] The results from Salter are consistent with other parallel models unrelated to FDS, thus this became an expectation of the Raspberry Pi Cluster's FDS runs and research.

This portion of this project builds upon the work put forth by Donald Collins in his master's thesis for The University of Tennessee at Knoxville.[3] Collins' research focused on ways in which FDS modeling work could be efficiently and quickly distributed across multiple processing cores. His work included a Python script to evenly divide a single mesh into multiple meshes for insertion into an FDS file. The usage and functionality of this script is expanded upon in the appendix provided with this paper. Fundamentally, his work allowed for one complex task to be subdivided for execution in a parallel environment, ideal for an application on BOB.

3.5.8 HPL - High Performance Linpack

HPL (High Performance Linpack) is a benchmark to determine the maximum double precision floating point performance on a distributed parallel system. It solves a random dense linear system $Ax = b$ with LU decomposition, and after the calculation, it checks the accuracy to ensure a valid result. Dense linear algebra calculations are applicable to many problems and a good method to measure peak performance for a system.[9]

HPL requires an MPI and BLAS library. The BOB system is using MPICH2 for message passing and OpenBLAS as the linear algebra library. OpenBLAS was selected over some alternatives due to its high performance on ARMv7 CPUs. In this case, OpenBLAS resulted in nearly an order of magnitude higher performance than ATLAS for the Raspberry Pi 3 SoC.

The Raspberry Pi 3 utilizes a Broadcom BCM2837 SoC with a quad core ARM Cortex A53 CPU clocked at 1.2 GHz.[18] This CPU supports VFP and NEON instructions to accelerate the processing of vector floating point instructions. BOB's OpenBLAS configuration is using the VFP unit for the calculations because the ARMv7 NEON unit is not IEEE floating point compliant.[2]

3.5.9 HPCG - High Performance Conjugate Gradient

HPCG, or High Performance Conjugate Gradient, is a newer benchmark designed to be used in conjunction with HPL to provide a more complete picture of the performance of a cluster.

Like HPL, HPCG solves $Ax = b$. However, HPCG uses a sparse matrix representation using an iterative conjugate gradient method rather than LU decomposition. The end result of these changes is a benchmark which tries to capture lower bound performance rather than peak performance. Using both HPL and HPCG, one can characterize the upper and lower bounds for typical cluster applications. Most programs will not achieve HPL levels of performance, but most programs will also be able to extract more performance than HPCG. This makes the combination of the two especially valuable when evaluating the performance of a cluster system. [8]

4 Testing Results

Testing results for the cluster's overall computational power, file system performance, network performance, and application-specific performance are discussed in the following subsections. A general trend to observe is that scaling performance of applications with intensive I/O is artificially constrained by the Raspberry Pi's 100 Mbps Ethernet, a major concern since the beginning of the system's construction. For embarrassingly parallel tasks with little required I/O, however, scaling results are in line with what we would hope to see in such a distributed system, with a diminishing return in performance per node only becoming a significant factor when a large collection of worker nodes is utilized.

4.1 HPL Benchmark Results

HPL is a ubiquitous benchmark in high performance computing, and as such, it serves as a useful comparison point to demonstrate the performance of this cluster against other systems throughout history.

Each worker node consists of a Raspberry Pi 3 which uses a quad core ARM Cortex A53 CPU. In the cluster, each core is clocked at 1150 Mhz. Assuming the Cortex A53 demonstrates similar floating point performance as the Cortex A9 core, the theoretical maximum floating point throughput can be estimated. For double precision (64 bit) FP values, the CPU can execute a scalar addition every cycle and a scalar multiplication over every two cycles. This can be summarized as about 1.5 FLOP/cycle. Using this information, it is possible to calculate a R_{peak} for both each node and the overall cluster.

$$R_{peak,node} = 1.5\,\text{FLOPs/Hz/core} \times 4\,\text{cores} \times 1150\,\text{Mhz} = 6.9\,\text{GFLOPs}$$

For the cluster, the R_{peak} is simply the single node maximum scaled to 64 total nodes.

$$R_{peak,cluster} = 6.9\,\text{GFLOPS} \times 64\,\text{nodes} = 441.6\,\text{GFLOPs}$$

However, this estimation is unrealistic. While the single node performance can come within 4% of the theoretical value, the cluster cannot achieve perfect scaling due to communication overhead.

The current R_{max} is 148.8 GFLOPs which places the cluster as the fastest supercomputer in the world in November of 1994. This is also good enough to be one of the Top 500 supercomputers in June of 2002.

Some values for throttled operation are provided as a comparison point. It is important to note that the performance is not necessary half of the full speed performance as the number of nodes increases beyond a single node. This is because of the high overhead of MPI calls when constrained to a low bandwidth interconnect. While the parallel compute performance might be doubled by doubling the frequency, the network communication time is a constant factor and unavoidable with the design of the Raspberry Pi. The scaling efficiency numbers also appear to be greater as a result of the increased ratio of computation speed to communication speed when throttled.

# Nodes	N	NB	Time (s)	GFLOPs	Scaling Efficiency
1	10000	256	184.33	3.618	100%
4	20000	256	553.70	9.633	66.6%
8	28000	256	968.21	15.12	52.2%
16	38000	256	1278.98	28.60	49.4%
24	48000	256	867.97	42.93	49.4%
32	55000	256	2024.93	54.78	47.3%
40	64000	256	2611.28	66.93	46.2%

Table 1: HPL Performance when throttled

# Nodes	N	NB	Time (s)	GFLOPs	Scaling Efficiency
1	10500	100	115.96	6.657	100%
2	15000	100	246.13	9.143	68.7%
4	21000	100	453.73	13.51	50.7%
8	30000	100	694.20	25.93	48.7%
16	42000	100	1093.02	45.61	42.8%
24	51000	100	1339.88	66.00	41.3%
32	59000	100	1592.44	85.98	40.4%
40	66000	100	1906.13	100.6	37.8%
48	73000	100	2153.39	120.4	37.7%
56	76000	100	2242.90	130.5	35.0%
64	80000	100	2412.44	141.5	33.2%

Table 2: HPL Performance at 1150 Mhz—Open Air

# Nodes	N	NB	Time (s)	GFLOPs	Scaling Efficiency
1	10500	100	125.21	6.165	100%
2	15000	100	266.35	8.449	68.5%
4	21000	100	409.25	15.09	61.2%
8	30000	100	682.86	26.36	53.4%
16	42000	100	1117.47	44.20	44.8%
24	51000	100	1357.19	65.16	44.0%
32	59000	100	1626.52	84.18	42.7%
40	66000	100	1890.25	101.4	41.1%
48	73000	100	2131.70	121.7	41.1%
56	79000	100	2454.77	133.8	38.8%
64	84000	100	2655.07	148.8	37.7%

Table 3: HPL Performance at 1150 Mhz—Enclosure

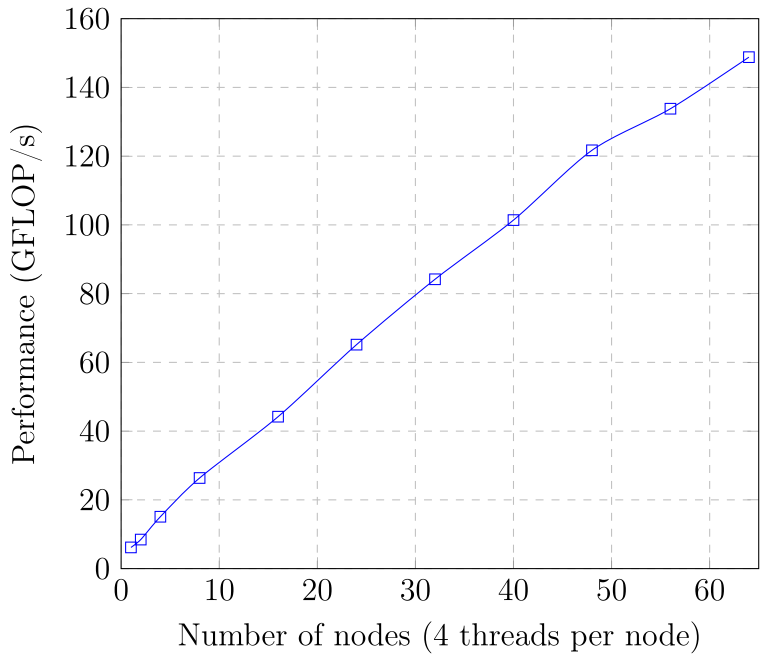


Figure 25: HPL Performance at 1150 Mhz—Enclosure

It should be noted that when comparing **Table 3** showing results from the enclosure to **Table 2** showing result from open air testing there is a dropoff in performance primarily seen at the single node data point. This appears to be due to some isolated thermal throttling causing by sub-optimal airflow in the new enclosure. This issue is masked at higher node counts because the communication overhead is significant. The communication time is constant regardless of computation time, and the communication time allows for a period of lower CPU utilization allowing the nodes to cool somewhat.

4.2 HPCG Benchmark Results

As HPCG is not an attempt to capture maximal floating point throughput, it should be noted these numbers are expected to be much lower. However, contrary to HPL, the scaling factor is both quite linear and nearly optimal.

Nodes	GFLOP/s	Scaling
1	0.08402	100.00%
2	0.16824	100.12%
4	0.33422	99.44%
8	0.65953	98.12%
16	1.30178	96.83%
24	1.90108	94.27%
32	2.55881	95.17%
48	3.77325	93.56%
64	5.07949	94.46%

Table 4: Reference Implementation HPCG Performance

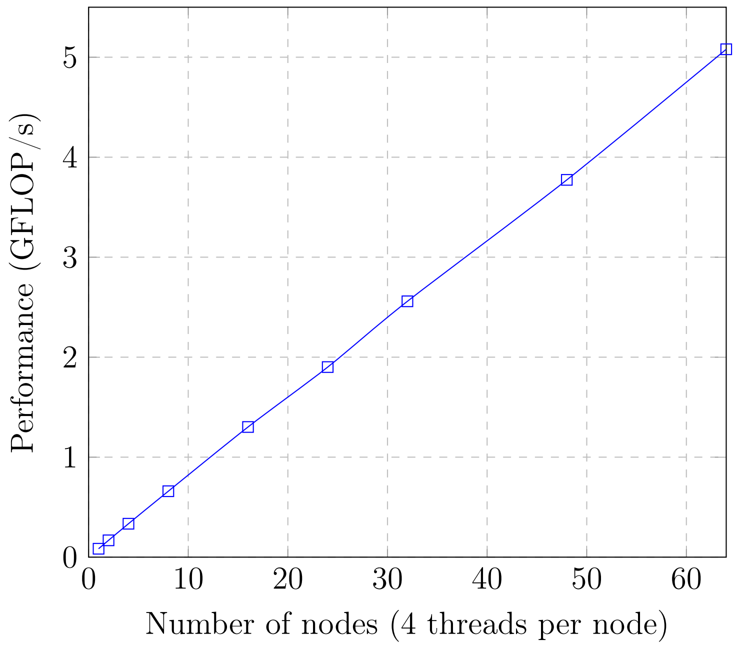


Figure 26: Reference Implementation HPCG Performance

Figure 26 demonstrates the linear scaling of HPCG from a single node to 64 nodes on BOB.

4.3 File System Benchmark Results

The Raspberry Pi 3 cluster began as two distinct clusters consisting of one control node, one storage node, and eight worker nodes all networked over 100 Mbps Ethernet. One cluster was used for systems development/testing and the other was used primarily for application development. After the initial build of both clusters, NFS was implemented to handle the file system servicing on the application cluster and the development of the implementation of OrangeFS was devoted to the development cluster.

An initial benchmark of NFS was completed by testing the sequential read and write times of both the local SD card and the NFS storage device on the application cluster as soon as it became functional. The benchmark was performed by utilizing the Linux utility `dd` on one client node. For testing write performance, the following command was used:

```
$ dd if=/dev/zero of=tempfile bs=1M count=1024 conv=fdatasync,notrunc
```

For testing read performance, the file created by writing after clearing the cache was used by executing the following commands:

```
# echo 3 > /proc/sys/vm/drop_caches
$ dd if=tempfile of=/dev/zero bs=1M count=1024
```

System Tested	Sequential Read	Sequential Write
Local SD Card	22.6 MB/s	10.8 MB/s
NFS Drive	11.2 MB/s	10.1 MB/s

Table 5: Sequential Single Node NFS Storage Performance

After successful implementation of OrangeFS with eight FUSE clients and one server node on the development tower, OrangeFS was benchmarked using the Interleaved Or Random (IOR) benchmark. IOR is designed to measure POSIX and MPI-IO performance on parallel file system I/O performance. IOR, developed by Lawerence Livermore National Lab, is a parallel program that performs reads/writes from/to files under several sets of conditions and reports the throughput rates.[10] The IOR benchmark test was used in this study for two reasons. First, to show that OrangeFS was in fact operating correctly by writing and reading large files to and from the server/storage node, and secondly, to show the I/O read/write performance of the cluster was enhanced by implementing OrangeFS as an alternative to NFS on the Raspberry Pi3 Beowulf Cluster. The IOR benchmark used on the Raspberry

Pi3 Cluster was modeled after the IOR NERSC-8 /Trinity Benchmark used by the National Energy Research Scientific Computing Center(NERSCS) to test sequential and parallel I/O performance.[26] This benchmark runs 2 sets of 3 POSIX tests and 2 sets of 3 MPI-IO tests for a total of 12 tests. One set of POSIX tests and one set of MPI-IO tests measure throughput on one shared file and the other sets measures throughput one file per process. All 12 tests use fixed block sizes of 1MB, a fixed user-defined segment count, and three different transfer sizes of 10KB, 100KB, and 1MB. The segment count dictates how many data sets will be contained in each file. This value needs to be declared in such a way that it ensures the IO benchmark files are not written to the client's local DRAM and is calculated such that the aggregate file size is at least 1.5 times the size of the available DRAM on the client.

$$\text{Aggregate File Size} = \text{segment count} * \text{block size} * \text{number of processes}$$

The IOR benchmark throughput values recorded and discussed in this report came from running a subset of the IOR N8/Trinity Benchmark[10]. 8 nodes implementing NFS on the application tower and 8 nodes implementing OrangeFS on the development tower of the Raspberry Pi3 Cluster were used to run the set of POSIX tests designed for one file per process of the IOR N8/Trinity Benchmark. 8 nodes implementing NFS on the application tower were used to run only the 10KB transfer size MPI-IO test for one file per process from the IOR N8 Trinity Benchmark. 8 Nodes using OrangeFS on the development tower were used to run the full set MPI-IO tests for one file per process using the IOR N8/Trinity Benchmark. The entire set of one file per process POSIX tests of the IOR N8/Trinity Benchmark was used to three times on the full 64 node Raspberry Pi3 Cluster after final integration of all hardware and systems, including OrangeFS. Once using 64 nodes implementing NFS, once using 64 clients and one server implementing OrangeFS, and a final time using 64 clients and two servers implementing OrangeFS. The segment count was set to 188 when testing 8 nodes and 23 when testing 256 nodes to create an aggregate file size of approximately 5.6 GB in all recorded tests. The sets of shared file tests were not run to expedite the testing process. The results of the various tests are discussed and shown below.

10KB MPI-IO Transfers, NFS vs OFS

File System	Read	Write
NFS	4.21	11.40
OFS	11.67	9.79

Table 6: MPI-IO Read and Write Rates (MB/s)

Table 6 displays the results of running the MPI-IO one file per process IOR N8/Trinity benchmark test with 10 KB transfers on NFS with no attribute caching(noac) enabled and OFS with OrangeFS default configurations and one server. This test was completed because NFS fundamentally does not work with MPI-IO by default but can be manipulated to work with MPI-IO by enabling no attribute caching. This feature requires that no file attributes are cached locally and all file attributes must be retrieved from the server. With this feature enabled, system performance is significantly decreased due to the increase in network overhead as verified in **Table 6**.

8 Node MPI-IO OrangeFS Test

Xfer Size	OFS Reads	OFS Writes
10 KB	11.67	9.79
100 KB	11.10	11.30
1 MB	10.44	11.65

Table 7: OFS MPI-IO Read and Write Rates (MB/s)

Table 7 displays the throughput of MPI-IO calls on OrangeFS while running the one file per process IOR N8/Trinity benchmark test set on eight clients nodes and one server using the default OrangeFS configurations. This test was used to verify OrangeFS was functional for MPI-IO calls. This test was also used to determine the capability of OrangeFS throughput under OrangeFS default configurations with one server and eight clients of the Raspberry

Pi3 cluster. **Table 7** shows that OrangeFS was not only functional for MPI-IO calls, but was achieving the theoretical maximum throughput rates of 11 MB/s possible for 100 Mbps Ethernet. This confirmed the belief that OrangeFS would perform better then NFS when making MPI-IO calls.

8 Node NFS vs OFS POSIX Test

Xfer Size	NFS Read	OFS Read
10 KB	10.41	10.78
100 KB	9.16	11.46
1 MB	9.65	11.12

Table 8: Read Rates (MB/s)

Xfer Size	NFS Writes	OFS Writes
10 KB	11.42	9.73
100 KB	11.25	11.31
1 MB	11.34	11.44

Table 9: Write Rates (MB/s)

The amount of performance gain OrangeFS may provide as well as its functional ability on POSIX operations was still in question, so the one file per process POSIX test set of the IOR N8/Trinity benchmark was run on the development tower of eight clients and one server implementing OrangeFS under its default configurations and the application tower of eight nodes and one server running NFS under its default conditions. In **Table 8** and **Table 9**, it can be seen that overall OrangeFS outperformed NFS in both reading and writing from and to the server using POSIX calls. The 10 KB transfers on OrangeFS writes did not exceed NFS performance. This is expected as NFS should outperform OrangeFS on smaller files and only one server. This test also showed that OrangeFS achieved the theoretical maximum throughput of 11 MB/s for 110 Mbps Ethernet. These results allowed the team to conclude that OrangeFS would enhance the performance of the Raspberry Pi3 Cluster using POSIX operations assuming the performance results scaled.

64 Node and One OrangeFS Server POSIX Test

Xfer Size	NFS	OFS
10 KB	8.4	5.64
100 KB	8.7	6.96
1 MB	8.5	7.85

Table 10: Read Rate (MB/s)

Xfer Size	NFS	OFS
10 KB	11.59	10.05
100 KB	11.57	11.51
1 MB	11.64	11.54

Table 11: Write Rate (MB/s)

OrangeFS was initially deployed on the full Raspberry Pi3 cluster exactly as it had been deployed on the development tower using only one server node. The results shown in **Table 10** and **Table 11** indicate that the high OrangeFS read throughput did not scale to 64 nodes as hoped. The throughput rate for reads using the OrangeFS default configurations and one server provided approximately half the throughput rates as seen on the eight node development cluster.

64 Node and Two OrangeFS Servers POSIX Test

Xfer Size	NFS Read	OFS Read
10 KB	8.4	11.79
100 KB	8.7	8.93
1 MB	8.5	10.18

Table 12: Read Rate (MB/s)

Xfer Size	NFS Writes	OFS Writes
10 KB	11.59	15.85
100 KB	11.57	21.73
1 MB	11.64	20.81

Table 13: Write Rate (MB/s)

"OrangeFS is designed to run on large clusters, composed of many storage nodes networked together"[16], so adding a second storage node to allow OrangeFS to run as it was designed seemed like the easiest and most logical way to achieve the performance seen on the eight node tower. The initial Raspberry Pi3 cluster design included two storage nodes. Therefore, adding a second storage node to the cluster would not violate the design requirements and it allowed OrangeFS to operate as it is intended with multiple servers.**Table 12** and **Table 13** demonstrates that adding a second server allowed OrangeFS to achieve comparable throughput results to the eight node development tower on reads and exceeded throughput results on writes.

4.4 Networking Benchmark Results

Networking is a critical aspect of any cluster. In the case of BOB, the networking performance is severely limited in comparison to other clustered systems due to the hardware utilized.

In order to examine the networking performance of the system, we used a utility called iperf to determine the node-to-node TCP (Transmission Control Protocol) throughput. In addition, the networking performance is indirectly tested with each parallel application as the message passing performance is a key aspect of many applications.

Node Type	# Threads	Destination	Throughput (Mbit/s)
Worker Node	1	Head Node	94.1
Worker Node	2	Head Node	94.1
Worker Node	4	Head Node	94.1

Table 14: Single Node, Threaded Network Performance

In **Table 14**, it can be seen that the number of simultaneous threads attempting to communicate does not negatively impact the aggregate TCP networking performance.

# Nodes	Destination	Per Node (Mbit/s)	Aggregate (Mbit/s)
1	Head Node	94.1	94.1
2	Head Node	47.1	94.2
4	Head Node	23.58	94.3
8	Head Node	11.8	94.4
16	Head Node	5.91	94.62
24	Head Node	3.95	94.85
32	Head Node	2.98	95.23

Table 15: Multiple Node Internal Network Performance

Table 15 demonstrates the networking performance when multiple worker nodes attempt to communicate with the head node. The aggregate throughput meaning the sum of the throughput achieved on each node tested. The per node throughput reported is the average of the results from each node tested.

Node Type	# Nodes	Destination	Aggregate Throughput (Mbit/s)
Head Node	1	com1598	190
Worker Nodes	1	com1598	93.7
Worker Nodes	2	com1598	93.3
Worker Nodes	4	com1598	93.6
Worker Nodes	8	com1598	94.3

Table 16: External Network Performance

Table 16 demonstrates the throughput available to both the head node and the worker nodes when trying to communicate with a system external to BOB. The destination system acting as the iperf host (com1598) is another system on the UT network attached via Gigabit Ethernet. From this experiment, it can be seen that there is practically no performance dropoff due to the low overhead of the head node's NAT system.

Additionally, the head node is able to achieve higher performance due to its USB to Gigabit Ethernet interface. This USB NIC cannot run at a full gigabit due to the limited maximum throughput of the USB 2.0 and the hardware architecture of the Pi. It can, however, achieve higher throughput than the internal NIC on the Pi 3.

4.5 Applications and Frameworks Benchmark Results

4.5.1 SPH

As a demo program, SPH was designed to allow users to adjust the number of nodes in use to affect the frame rate of the simulation's rendering node. However, the program was originally designed for the original Raspberry Pis used by the Titan Titan cluster, which have far fewer and less powerful nodes. The Raspberry Pi 3s that were used by our cluster had double the clock speed and has four times as many cores.

As a result of the increase in power, our cluster was able to maintain sixty FPS on a single core at the default particle configuration. By increasing the number of particles, we were able to achieve a slight drop in frame rate only when not using the smoothed fluid view (i.e. using the individual particle render option). However, when run across multiple nodes as intended, SPH was unable to achieve a loss in frame rate until the stability of the simulation was compromised. Increasing the default particle count from 1,500 to 10,000, the frame rate would begin to drop as fewer nodes were utilized. Unfortunately, the physics calculations themselves began to fail, as the increased particle density forced some particles inside others, resulting in erratic behavior. Examples would typically include fluid particles bouncing out of and across the surface of the fluid without any external influence as well as particles that were forced outside of the bounding box of the simulation.

4.5.2 PiBrot

PiBrot is the second demo program adopted from the Tiny Titan project. The performance of the rendering node is, seemingly, entirely dependent on the network. Regardless of the number of nodes utilized in the race, the entire simulation runs in approximately the same length of time. The master node is forced to read each line from the fractals across the network individually and then re-render the entire screen. Due to this, the worker nodes are able to perform the Mandelbrot Set check for their rows much quicker than the master node can render them to the user.

As a result, the number of rows rendered per second is essentially a fixed value, with the distribution of these rows varying based on the number of nodes used for the parallel portion of the code. With a total of four nodes, the master node renders the right side approximately twice as fast as the left side, resulting in half of the left screen rendered when the right side finishes. When doubling the number of nodes used in parallel is doubled (i.e. $N = 6$ instead of $N = 4$), the left side will only have completed a quarter of the screen when the right side finishes. Since the interconnect bandwidth and re-render times are already a bottleneck for the simulation, the relative time to completion scale nearly linearly with the nodes on the right while the overall time to completion of both sides of the screen remains stable.

4.5.3 Parallel Pi

Parallel Pi results were obtained from test runs using between 4 and 256 parallel processes, with each test using twice the number of processes of the previous test. As the division of the series into roughly equal subtasks is considered embarrassingly parallel, positive scaling results were anticipated. Test results are presented in the following graph. Please note that the x-axis is scaled logarithmically.

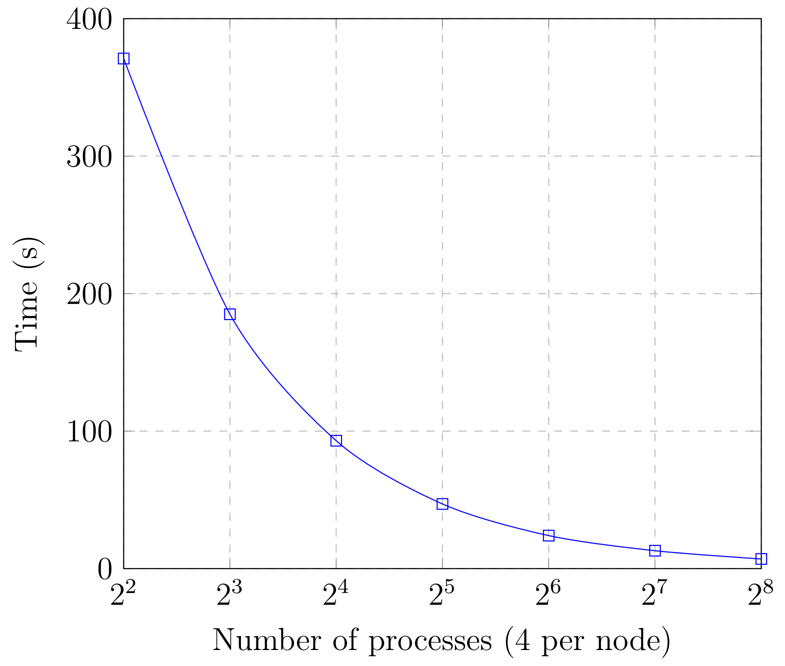


Figure 27: Parallel Pi Scaling Results

As can be seen from the results above, a diminishing return in speedup can be observed as the application utilizes more processes. However, these numbers are not surprising when considering the effects of Amdahl's Law, which states that speedup of a task is limited by the portion of the task that can not be parallelized, with each additional parallel process providing less performance improvement than the addition of the previous. Amdahl's Law provides us with the following formula:

$$S = \frac{1}{(1-p) + \frac{p}{n}}$$

where S is expected task speedup, p is the percentage of the task that can be parallelized, and n is the number of parallel processes. Using a rough approximation of $p \approx 99.87$ yields an expected speedup of approximately 3.98 for 4 parallel processes and 192.3 for 256 processes, which supports the results shown in **Figure 27**.

4.5.4 Monte Carlo

The testing of the Monte Carlo application focused on ensuring appropriate scalability. The program that acted as the user executable was a simple Python dice roll program which printed a random number between one and six. The blue line in the graph below shows the results of 100 random dice rolls on 4, 8, 16, 32, and 64 CPUs. The orange line shows the same test except with 1,000 random dice rolls. The variation using 4 CPUs for the two tests is large with the 100 rolls running approximately 3.13 seconds, while the 1,000 rolls clocks at 31.46 seconds. Both tests converge towards 64 CPUs, yet it is clear that the 1,000 runs benefited more from additional CPUs than the 100 runs. This can be seen in the table below where from 32 to 64 CPUs, the 100 runs only speeds up 8.3% compared to the previous 38.1% speedup from 16 to 32 CPUs. The total speedup from 4 to 64 CPUs for 100 runs is 82.2%. For the 1,000 runs, the speedup is consistently around 50% for all additional CPU tests. The speedup is 93.2% from 4 to 64 CPUs for 1,000 runs.

# Nodes	100 Dice Rolls Run Time (seconds)	% Speedup	1,000 Dice Rolls Run Time (seconds)	% Speedup
1	11.7004	-	113.4626	-
4	3.1275	73.2702	31.4619	72.2712
8	2.1206	32.1947	14.8163	52.9072
16	0.9785	53.8559	7.76318	47.6037
32	0.6057	38.1015	4.1045	47.1290
64	0.5554	8.3066	2.1351	47.9801
$1 \rightarrow 64$	-	95.2533	-	98.1182

Table 17: Test Results for Monte Carlo Application

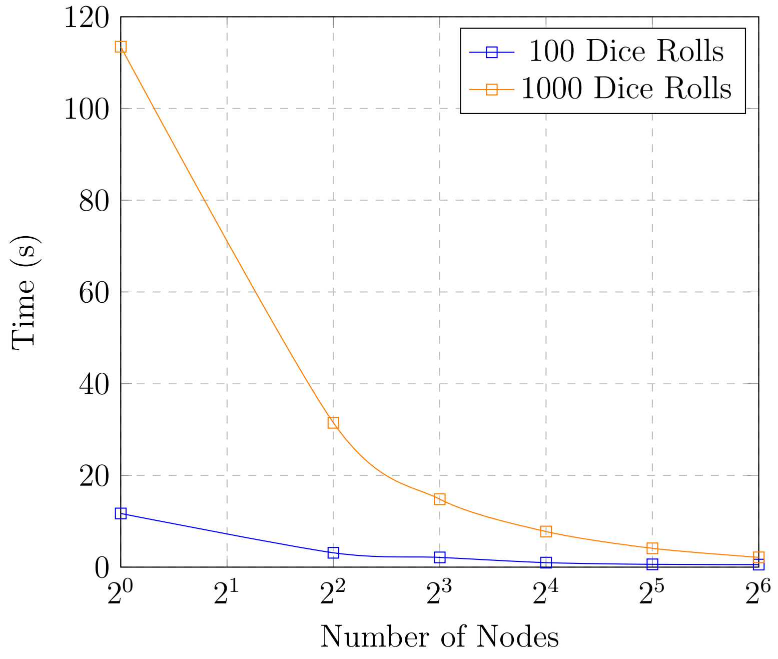


Figure 28: Monte Carlo Scaling Results

This test data reflects other test results that have been shown on BOB. The communication bottlenecks tend to mostly affect smaller programs (which most likely do not need to be parallelized) and programs that utilize a large number of reads and writes. With this Monte

Carlo application, results are likely to vary depending on the user specified executable, however, the results show improvement with the addition of CPUs for these test runs.

4.5.5 Numeric Integration

The Numeric Integration Application utilizing right Riemann sums demonstrated expected results throughout testing. Similar to the results in Parallel Pi, there was a diminishing return present in the Percent Error results, yet again due to Amdahl's Law, and this is shown in both **Figure 29** and **Figure 31**. Even though the figure showing run time looks appropriate to the expectations, there were frequently trials which took significantly longer than expected. This may be due to any number of factors since testing was performed using SLURM in order to specify the number of cores for a run. Though gathering data to show timing results was important for testing, it is worth mentioning that run time for this application should be approximated as results were inexact and, at times, inconclusive. Also, please note that **Figure 31** utilizes data from both the 10-sample runs and 100-sample runs because it seeks to show the percent error for the total rectangles while not emphasizing the number of cores used to obtain the results. This was done to demonstrate the effectiveness of the Riemann sum calculation aspect of the application and not its parallelization.

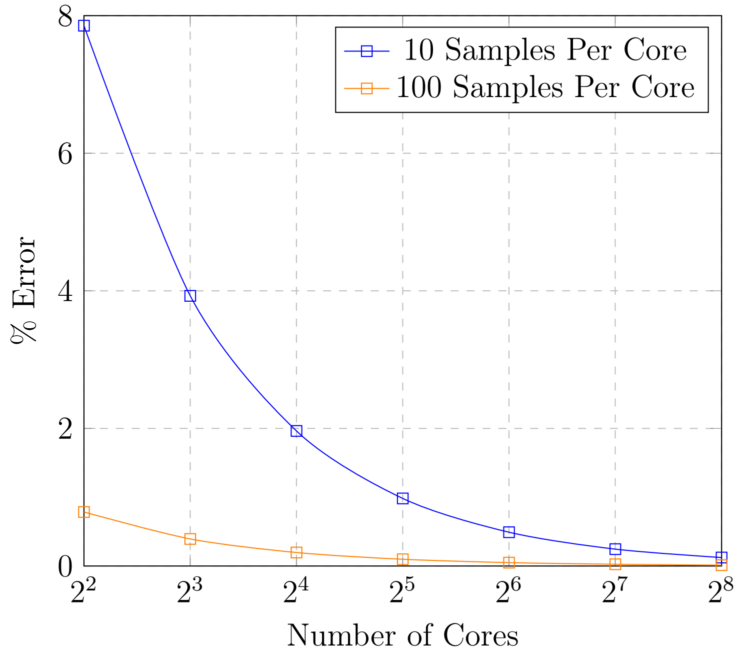


Figure 29: Percent Error vs. Number of Cores

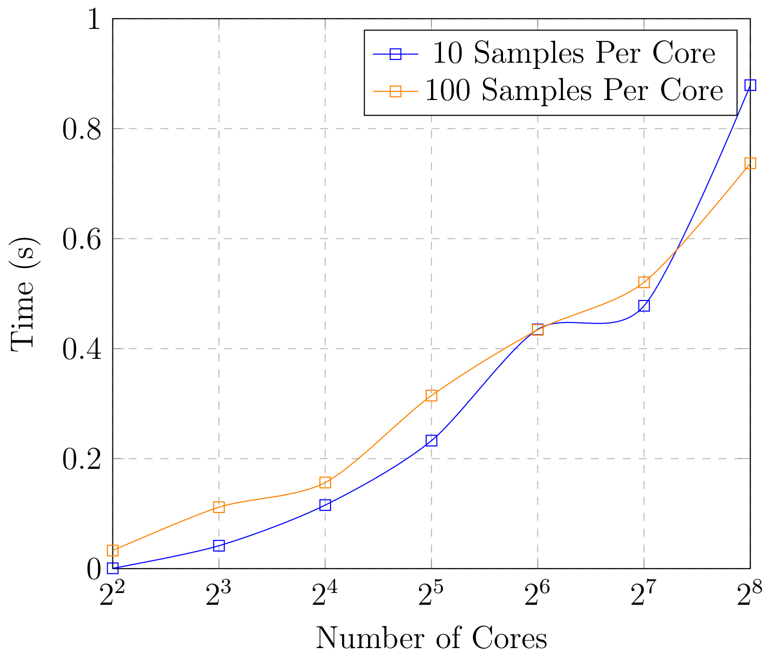


Figure 30: Run Time vs. Number of Cores

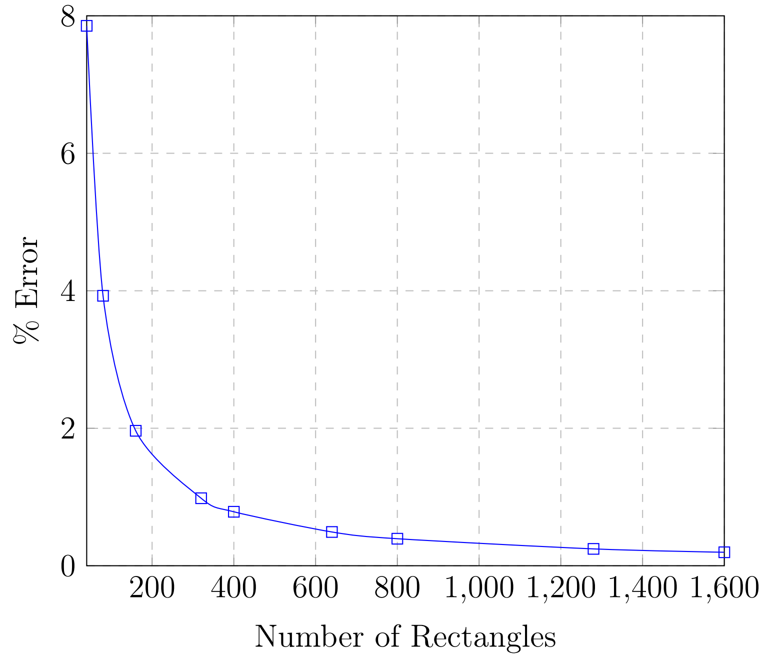


Figure 31: Percent Error vs. Number of Rectangles

# Cores	10 Samples % Error	10 Samples Run Time (s)	100 Samples % Error	100 Samples Run Time (s)
4	7.854	0.00058	0.785	0.03285
8	3.927	0.0416	0.393	0.1117
16	1.963	0.11552	0.196	0.15657
32	0.982	0.23274	0.098	0.31466
64	0.491	0.43496	0.049	0.43414
128	0.245	0.47786	0.025	0.52061
256	0.123	0.87908	0.012	0.73723

Table 18: Test Results for Numeric Integration Application
(Note: # samples are per core)

4.5.6 DANNA

To test the distributed DANNA Evolutionary Optimization performance, a pole balancing neural network was generated with a known seed over varying node counts to demonstrate the scaling of the cluster's performance. The network must survive at least 5 minutes of pole balancing on the fitness tests.

Please note that EO is not an ideal benchmark because of the random aspect of the network generation. Each time metric listed is the average of 5 runs performed with different random starting seed values.

# Nodes	Distribution Algorithm	Time (s)	Scaling Factor
1	None	5930	1.00
8	Master-Slave	2105.8	2.82
16	Master-Slave	1244.6	4.76
32	Master-Slave	911.8	6.50
64	Master-Slave	897.8	6.61

Table 19: DANNA EO Performance for Pole Balancing

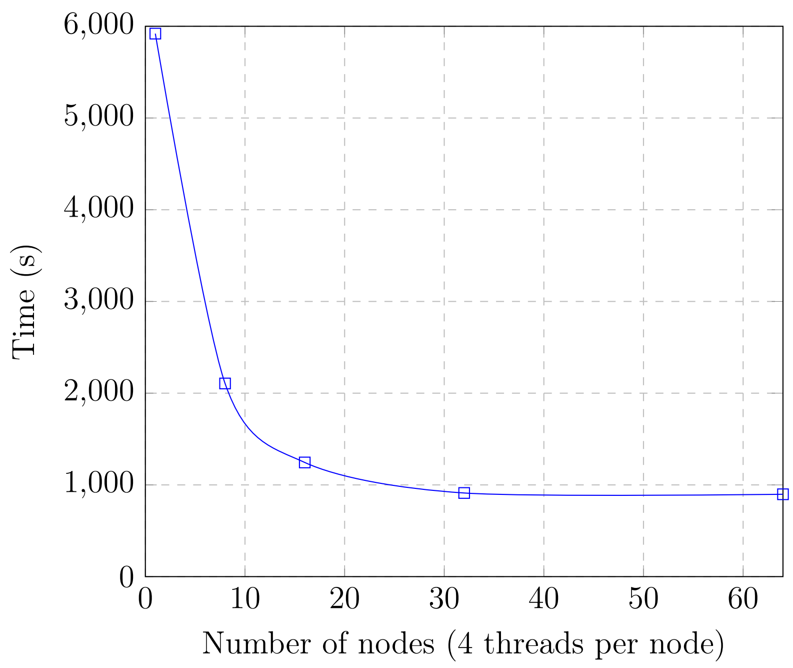


Figure 32: DANNA EO Performance for Pole Balancing

4.5.7 Fire Dynamics Simulator Results

A simple FDS file which models a room with a fire and an HVAC system was used for testing scalability across the cluster. The original file contains a single mesh encompassing the entire room. The mesh properties were entered into the `mesh_slicer.py` program which generates series of test files. The files generated include one mesh, two meshes, four meshes, 16 meshes, 32 meshes, 64 meshes, and 128 meshes. These numbers were chosen to make it easier to evenly divide the original mesh while still providing adequate data points to test performance scaling.

The following chart and table show the results of the tests that were run on the Raspberry Pi cluster above. Although the increase in performance from one to four meshes is significant, there is little advantage in running it on more cores on the cluster. In fact, shortly after 16 cores the networking overhead of the cluster overwhelms the performance gains of more cores.

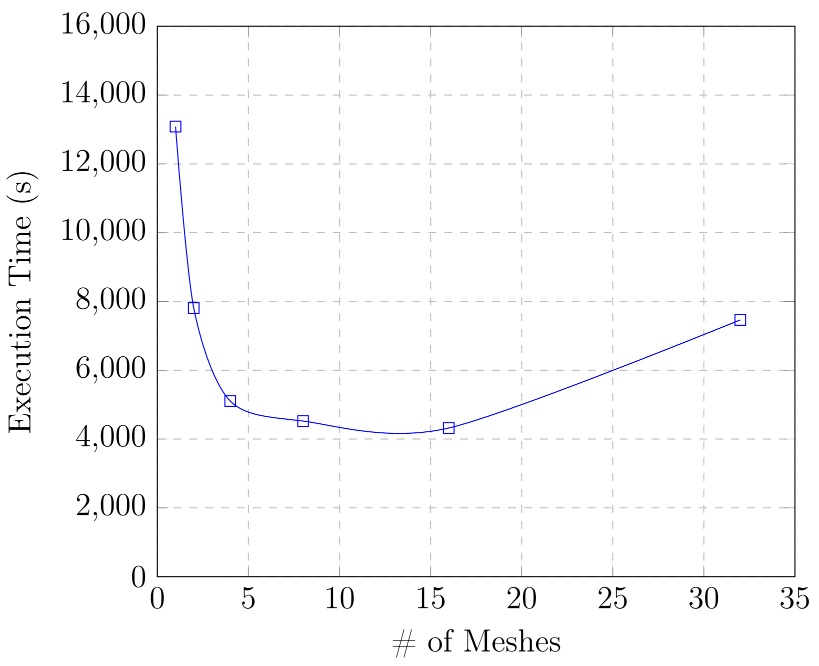


Figure 33: Graphical results of tests dividing a simple room into multiple meshes on the Raspberry Pi cluster.

Number of Meshes	Total Elapsed Wall Clock Time (s)
1	13084.685
2	7807.769
4	5106.836
8	4522.845
16	4320.457
32	7464.152

Table 20: Tabular results of tests dividing a simple room into multiple meshes on a Raspberry Pi cluster.

These results are not entirely unexpected, and in fact, they reflect the results that Salter experienced in his research on distributed computing with FDS on cloud-based resources. The advantage of a Raspberry Pi cluster is that it provides a large number of cores for a relatively inexpensive price. The cores themselves are computationally weak compared to traditional x86 cores. Each core adds a performance increase as well as an overhead cost, and because the Raspberry Pi cluster is limited to a 48-port switch, there is an unclear bottleneck past 48 Raspberry Pis. While the FDS overhead cost is virtually the same between x86 and ARM cores, the performance per core is not. This explains why the Raspberry Pi cluster's scaling ability plateaus more quickly than a typical multicore x86 system.

4.5.8 TensorFlow

LeNet[11], a type of convolutional neural network (CNN) used to recognize handwritten digits (0–9), was implemented on BOB. The dataset used for our tests is the MNIST handwritten number dataset. For each of our tests, the parameters are stored on a node (parameter server), and the heavy computation is assigned among worker nodes. Asynchronized parameter update was used here, i.e., the parameters are updated whenever a worker node finishes its current training iteration.

Although deep learning (CNN) is famous for its outstanding representation power, its heavy computation is notorious as well, which is mainly caused by the convolution operation and large number of parameters in the model. In this case, with 1 ps (parameter nodes) and 3 worker (worker nodes), training one epoch takes 1 hour to finish. The testing result is shown in **Table 21**. Note that only the training time for 1 epoch is listed.

ps nodes	worker nodes	training time
1	3	$58'$
1	7	$1:51'$
1	15	$4:23'$

Table 21: TensorFlow Performance

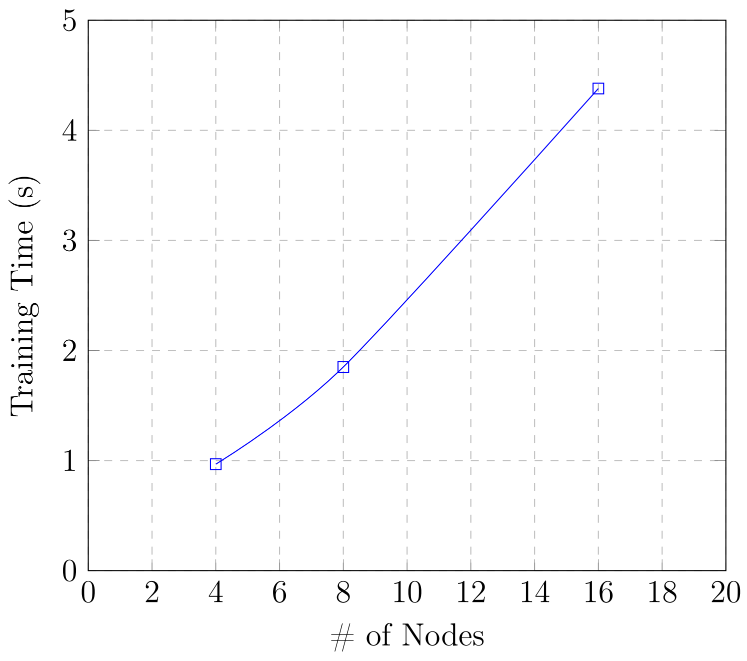


Figure 34: Graph of TensorFlow Performance

The result is controversial. It is expected with more nodes, the total time needed for training should be reduced. However the testing result is opposite to our expectation. (Possible reason: the overwhelming overhead, communication, because the workers have to transmit the updates to the parameter server in order to update the parameters, and then the parameter server has to transmit the parameters back to all the workers to train the next batch of the dataset). More experiments will be conducted to examine the computation time and communication between nodes in the cluster.

5 Challenges, Conclusions, and Future Considerations

Many of the complications encountered during the construction and testing of BOB were due to inherent shortcomings of the Raspberry Pi itself. Inter-node communication was limited by the Pi's 100 Mb Ethernet as well as the reliance on a single connection between the two network switches. Absence of support for a 64-bit operating system required us to rely on an OS which utilizes a 32-bit ARMv7 kernel, meaning that we were unable to exploit many of the performance improvements offered by ARMv8. A limit of 1 GB of on-board RAM also contributed to a lack of single node performance and more reliance on the storage nodes, increasing the strain on the storage nodes and the network as a whole. Storage speed was also bottlenecked due to reliance on the Pi's USB 2.0 ports. The ability to connect the external hard disks via USB 3.0 or SATA would certainly provide a measurable increase in storage speed. Were one of these options available, abandoning hard disk drives in favor of solid-state drives would be ideal given a sufficient budget.

CPU throttling of the Pis was an issue throughout the project due to both inadequate power supply as well as insufficient cooling methods. While throttling due to voltage sag was solved by the replacement of the original power supplies and USB power cables, individual nodes still sometimes experience thermal throttling while under heavy load due to BOB's current cooling configuration.

Security concerns came to light after networking issues were experienced late into the project. Server logs showed several IP addresses originating in China had attempted an unsophisticated brute-force SSH attack in an attempt to gain access to BOB. Although root access via SSH was already disabled, Fail2Ban was implemented as an extra security measure, blacklisting IP addresses after five successive failed login attempts. These security concerns, along with the often unstable nature of BOB during the construction process, served as a continuous reminder of the importance of the famous Git mantra: "Commit early, commit often".

Many potential improvements remain to be implemented as the first phase of BOB's construction comes to an end. More intensive application development is planned for the Fall 2016 semester, potentially focusing on the successful installation of Hadoop on OrangeFS via OFS FUSE Client, the optimization of HPCG, as well as the development of new applications in the areas of weather modeling and fluid dynamics. OrangeFS can be further developed by investigating and manipulating the available user-defined configuration parameters and by incorporating LDAP or key-based security in the initial build configuration of the portable OrangeFS directory. Slurm accounting, used for collecting information about current and previously executed Slurm jobs, remains to be configured. Improved cooling solutions may also be explored due to aforementioned thermal throttling of some worker nodes when under prolonged load. Finally, we are in possession of 32 Pine64+ single-board computers, which boast Gigabit Ethernet and double the RAM of the Raspberry Pi 3. We have considered adding these to the current system, as well as the possibility of "cloning" BOB and comparing the performance of the two systems.

Overall, we consider the initial construction of the Big Orange Bramble to be a success. BOB is a fully functional cluster with multiple working applications demonstrating its ability to efficiently parallelize tasks. The cluster can be easily scaled as the addition of further worker nodes is easily handled through the use of Ansible playbooks. BOB's HPL results easily outperform those of previous Raspberry Pi 2 clusters[13], and to our knowledge, this is the first instance of OrangeFS being successfully implemented on any type Raspberry Pi cluster[12]. While the Raspberry Pi's low price makes it an appealing option for constructing a budget computing cluster, building and testing of BOB has continually revealed fundamental limitations of the Pi which make it a less attractive choice for serious high performance computing applications in comparison to other off-the-shelf options. In spite of these inherent shortcomings, however, BOB succeeds in acting as an effective learning environment for students and faculty who wish to deploy, test, and evaluate distributed frameworks and applications.

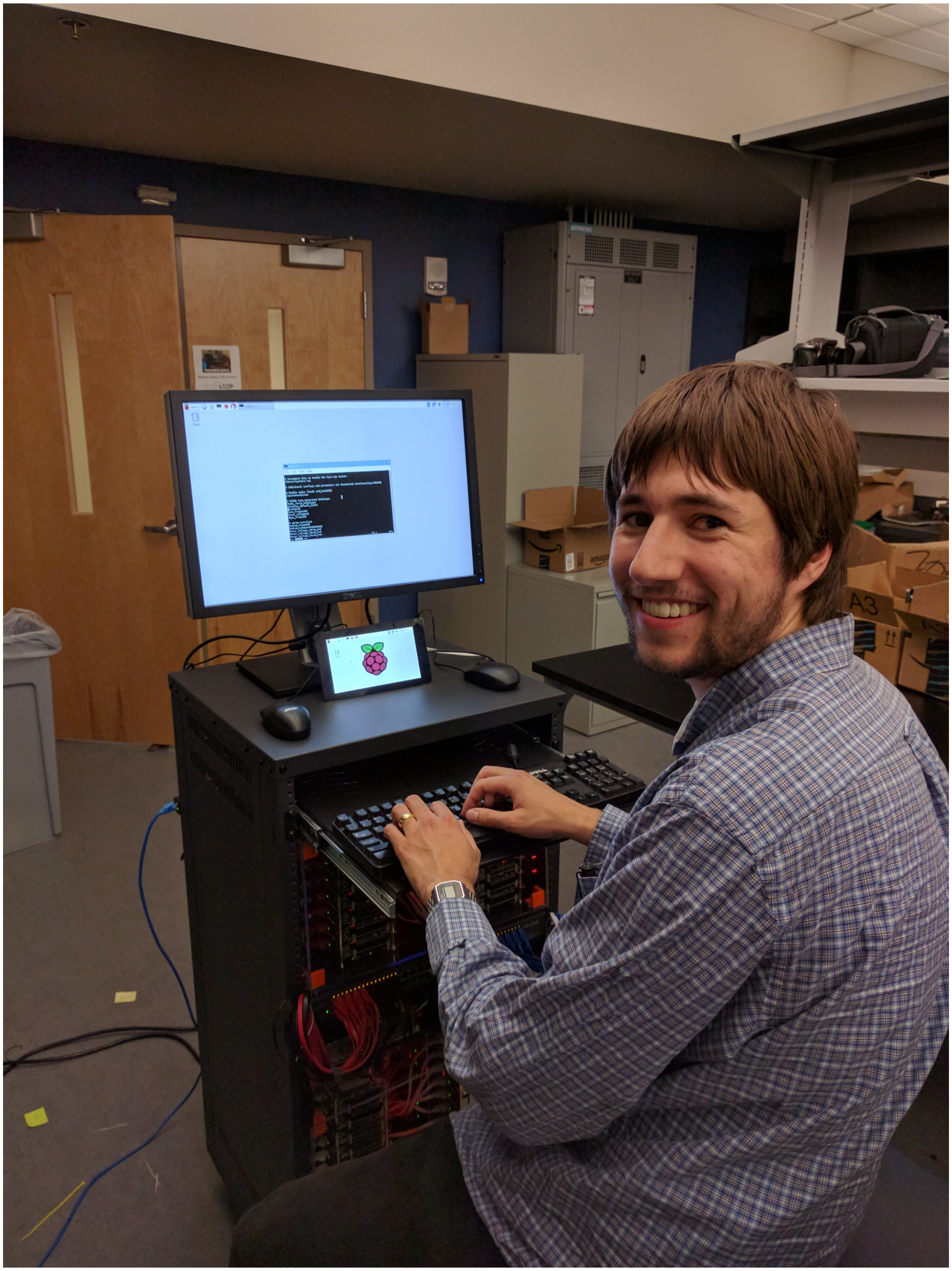


Figure 35: Success.

References

[1] *Ansible is Simple IT Automation*. Red Hat. URL: https://www.ansible.com/ (visited on 08/05/2016).

[2] *ARM Options Using the GNU Compiler Collection*. URL: https://gcc.gnu.org/onlinedocs/gcc/ARM-Options.html (visited on 06/16/2016).

[3] Donald Collins. *Dividing and Conquering Meshes within the NIST Fire Dynamics Simulator (FDS) on Multicore Computing Systems*. The University of Tennessee. URL: http://trace.tennessee.edu/cgi/viewcontent.cgi?article=4916&context=utk_gradthes (visited on 07/14/2016).

[4] *Distributed TensorFlow*. URL: https://www.tensorflow.org/versions/r0.10/how_tos/distributed/index.html.

[5] *Fail2Ban*. URL: http://www.fail2ban.org/ (visited on 08/03/2016).

[6] Jason Floyd. *FDS-SMV*. NIST: National Institute for Standards and Technology. URL: https://github.com/firemodels/fds-smv (visited on 07/12/2016).

[7] *Gluster Docs*. URL: http://gluster.readthedocs.io/en/latest/Developer-guide/Developers-Index/ (visited on 08/05/2016).

[8] *HPCG*. URL: http://www.hpcg-benchmark.org (visited on 08/02/2016).

[9] *HPL A Portable Implementation of the High-Performance Linpack Benchmark for Distributed Memory Computers*. URL: http://www.netlib.org/benchmark/hpl (visited on 06/14/2016).

[10] *IOR*. URL: http://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/ior/ (visited on 08/05/2016).

[11] Yann Lecun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE*. 1998, pp. 2278–2324.

[12] Walt Ligon. *Re: Orange FS inquiry*. Email. Message to Patricia Eckhart. June 13, 2016.

[13] V. M. Weaver M. F. Cloutier C. Paradis. "A Raspberry Pi Cluster Instrumented for Fine-Grained Power Measurement. (in submission)".

[14] *Neuromorphic Computing at Tennessee Research Overview*. URL: http://neuromorphic.eecs.utk.edu/pages/research-overview (visited on 07/27/2016).

[15] *OpenLDAP Server*. Ubuntu. URL: https://help.ubuntu.com/lts/serverguide/openldap-server.html (visited on 08/05/2016).

[16] *OrangeFS Documentation*. URL: http://docs.orangefs.com/home/index.htm (visited on 08/05/2016).

[17] *OrangeFS Information*. URL: http://orangefs.com/information/ (visited on 08/05/2016).

[18] *Raspberry Pi 3 now on sale at $35*. URL: https://www.raspberrypi.org/blog/raspberry-pi-3-on-sale (visited on 06/02/2016).

[19] C. Salter. "Fire Modelling Within Cloud Based Resources". In: *Fire Technology* 51.3 (Oct. 2014), pp. 491–497. DOI: `10.1007/s10694-014-0433-2`. URL: `http://link.springer.com/article/10.1007/s10694-014-0433-2`.

[20] C. D. Schuman et al. "Parallel Evolutionary Optimization for Neuromorphic Network Training". In: *Machine Learning in HPC Environments*. Salt Lake City, Utah, 2016.

[21] Adam Simpson. *Tiny Titan*. Oak Ridge Leadership Computing Facility. URL: `www.github.com/TinyTitan` (visited on 07/02/2016).

[22] *Slurm Commercial Support and Development*. SchedMD. URL: `http://www.schedmd.com/` (visited on 08/06/2016).

[23] *Slurm Workload Manager*. SchedMD. URL: `http://slurm.schedmd.com/` (visited on 08/05/2016).

[24] *TensorFlow*. URL: `https://github.com/samjabrahams/tensorflow-on-raspberry-pi` (visited on 08/05/2016).

[25] *TensorFlow Example*. URL: `https://github.com/tensorflow/tensorflow/blob/master/tensorflow/models/image/mnist/convolutional.py` (visited on 08/05/2016).

[26] *Trinity Benchmarks*. URL: `http://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/` (visited on 08/05/2016).

[27] Scott Williamson. *subfact*. URL: `https://github.com/scottjw/subfact_pi_ina219` (visited on 07/05/2016).

Appendix A Hardware/Systems Build Guide

A.1 Hardware

A.1.1 USB Hub Modifications

The purchased IXCC 7-port USB 3.0 hub was originally designed for data transmission rather than power distribution. In order to ensure that each hub would be capable of delivering 5 V and 10 A, some modifications were required. As described in **Section 3.1.1**, the trace width required to carry the desired current would be 370 mils (1 mil = 0.001 in) whereas the positive power bus trace was measured to be 100 mils on the board.[1]

To minimize the current traveling through the positive power bus, the trace was sliced in the approximate midpoint using an Exacto-Knife (see **Figure 36**. By doing this, the current would fan out such that the max current through a trace would be 2.5 A. A 16 AWG stranded wire was used to connect the Power Supply to the two current fan-out points (See **Figure 37**). No modifications were needed for the ground return path since all free space on the board was a copper ground plane (GND polygon pour).

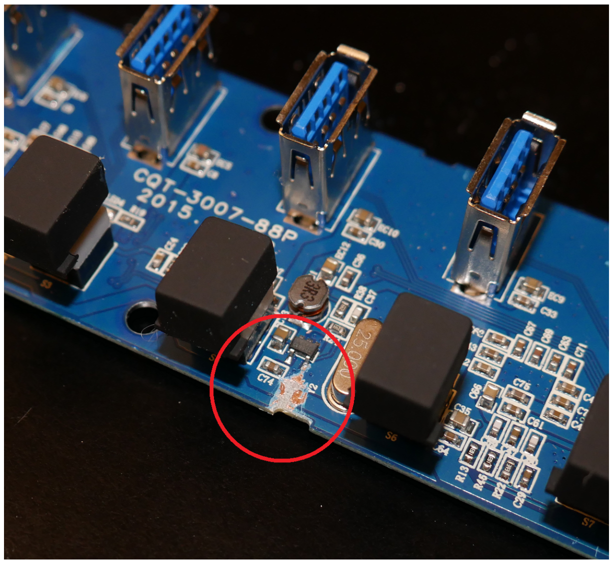


Figure 36: USB Hub: Sliced Power Bus

[1]Trace width was calculated using Advanced Circuit's online trace calculator. Reference the URL: `http://www.4pcb.com/trace-width-calculator.html`

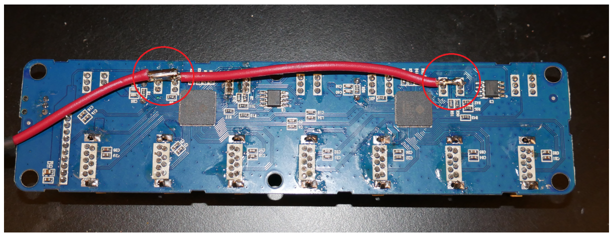


Figure 37: USB Hub: Power Rail Connections

Using diagonal pliers, the right angle header used as the USB input to the hub on the PCB was removed. With the same Exacto-Knife that was used previously to slice the positive power bus, the solder mask above the copper ground plane was scrapped in order to expose the copper and a solder pad could then be created for the ground connection to the power supply. Once both wires for power input to the hub were connected, the pairs were twisted and electrical heat-shrink wrap was used to create a mechanical connection between the two (see **Figure 38**).

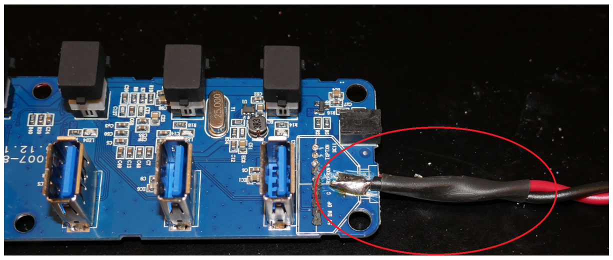


Figure 38: USB Hub: Ground connection and Heat-shrink

Once these modifications were complete, the PCB was recovered with the original plastic enclosure. The power input, twisted pair was then connected to the correct terminal on the power supply. After these connections were made, the power supply and USB hub were ready to be used (see **Figure 39**).

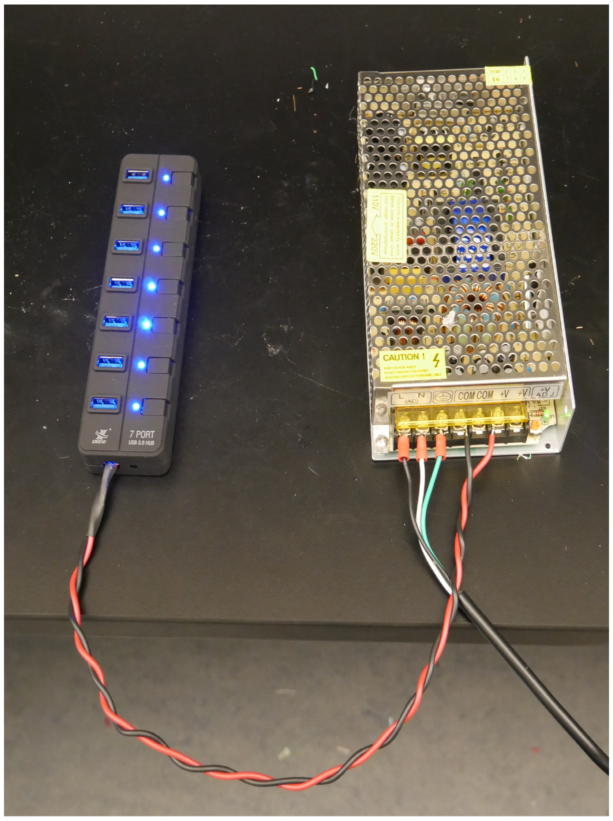


Figure 39: USB Hub: Complete with Power Supply

A.1.2 Daughter Card

To attach a daughter card to a node, first run the following to get I^{2}C utilities:

```
$ sudo apt-get install i2c-tools
```

Now enable the I^{2}C interface on your Raspberry Pi by running this command:

```
$ sudo raspi-config
```

Select Advanced Options, then select I^{2}C, then select yes. Finish the raspi-config. Connect the device to Pin 3 (SDA), Pin 4 (SCL), and Pin 6 (GND) at this point. It is also possible to connect an external power supply to the Pi on Pin 1 (5V Power) and Pin 9 (GND). The Raspberry Pi may require a restart before it will recognize that an I^{2}C device is connected.

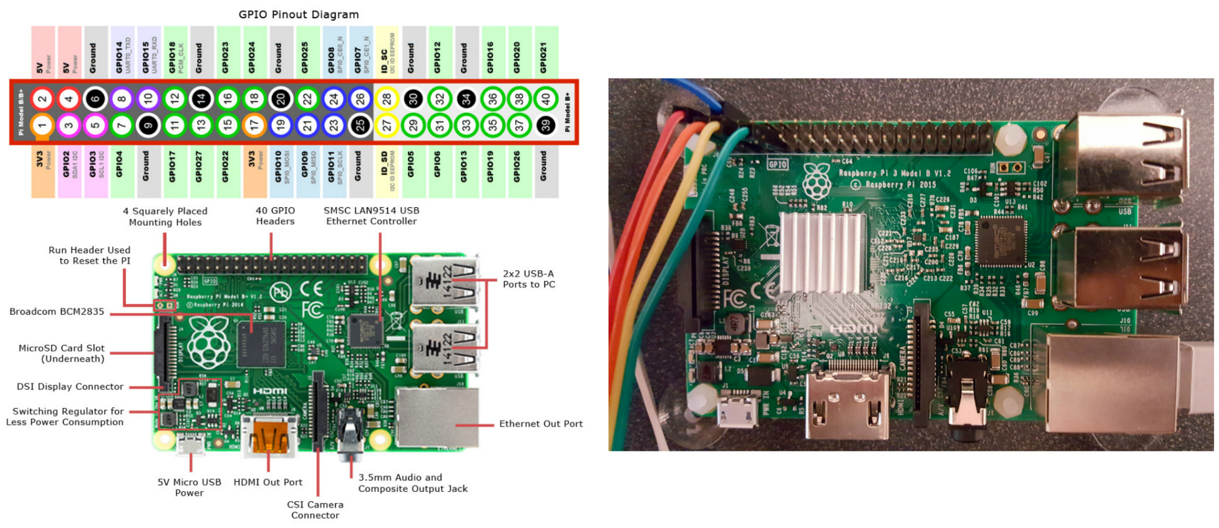

GPIO Pinout Diagram
4 Squarely Placed Mounting Holes
40 GPIO Headers
SMSC LAN9514 USB Ethernet Controller
Run Header Used to Reset the PI
2x2 USB-A Ports to PC
Broadcom BCM2835
MicroSD Card Slot (Underneath)
DSI Display Connector
Switching Regulator for Less Power Consumption
Ethernet Out Port
5V Micro USB Power
HDMI Out Port
3.5mm Audio and Composite Output Jack
CSI Camera Connector

Figure 40: GPIO Diagram

Now check that the device is properly connected with these commands:

```
$ ls /dev/*i2c*
/dev/i2c-1

$ i2cdetect -y 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: 40 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

The I^{2}C bus should be `/dev/i2c-1` and the device should be connected at 0x40.

If you don't already have a git directory in your home folder, make one:

```
$ cd home
$ mkdir git
```

Now enter the git directory and clone the python library for the INA219 [27]:

```
$ cd git
$ git clone https://cwill133@bitbucket.org/bigorangebramble/

daughter-card.git
```

Install the following packages:

```
$ sudo apt-get install python-dev python-smbus python-cffi
$ sudo apt-get install python-cryptography python-paramiko
```

Now run the following installations separately:

```
$ sudo pip install cffi
$ sudo pip install cryptography
```

Now enter the daughter-card directory:

```
$ cd daughter-card
```

If you are on a node with an attached daughter card, test that you can take measurements by running the measurement program:

```
$ python measure.py
```

If you are on the master node, run the wrapper.py program to query available nodes.

Note: When a new daughter card is added to a new node and after the node has run the above configuration, the wrapper function will need to be updated with a handler for the IP address of the new node.

A.2 Systems

A.2.1 Cluster Setup

1. Connect to the internet
2. `sudo apt update`
3. `sudo apt upgrade`
4. Create ssh key. `ssk-keygen`
5. Add your key to yourself.
6. Clone the ansible playbook.
   `git clone git@bitbucket.org:bigorangebramble/ansible_playbook.git`
7. `sudo apt install python-dev`
8. `sudo pip install markupsafe`
9. `sudo pip install ansible`
10. Add `127.0.0.1 pih0` to hosts file.
11. Run playbook for the headnode.
12. Connect headnode to the rest of the nodes.

13. Setup the LDAP server. See A.2.2
14. Copy ssh key to other nodes in the cluster.
15. Copy pios0 key to pih0 for users pi and root.
16. Make sure all the nodes can be communicated with. `ansible cluster -m ping`
17. Run playbook for cluster
18. You will be asked for your system password twice when you download the build directories from the storage node.

Note: If the playbook fails when trying to mount the NFS server, log into the server node and check the status of the nfs-kernal-server. You may have to reboot to load it into the kernel. You may also have to restart rpcbind if portmapping is not up.

A.2.2 Security (LDAP)[15]

Setup the LDAP server on the head node.

1. `sudo apt install slapd ldap-utils`
   (a) Set admin password when prompted
2. `sudo dpkg-reconfigure slapd`
   (a) Omit OpenLDAP server configuration: no
   (b) Set domain name when prompted
   (c) Set the organization name when prompted
   (d) Set admin password when prompted
   (e) Set database type: MDB
   (f) Remove on Purge: No
   (g) Move old: Yes
   (h) Allow LDAPv2: No
3. Setup server structure for scripts
   (a) create add_content.ldif

```
dn: ou=Users,dc=example,dc=com
objectClass: organizationalUnit
ou: Users

dn: ou=Groups,dc=example,dc=com
objectClass: organizationalUnit
ou: Groups
```

   (b) Add content `sudo ldapadd -x -D cn=admin,dc=example,dc=com -W -f add_content.ldif`

   (c) `vim /etc/pam.d/common-session`

      i. add `session required pam_mkhomedir.so` to end.

Setup LDAP scripts to allow for easy creation of new users

1. `sudo apt install ldapscripts`
2. Edit `/etc/ldapscripts/ldapscripts.conf`
   (a) `SERVER="ldap://localhost"`
   (b) Set `SUFFIX` to domain name dc
   (c) Uncomment `GSUFFIX`, `USUFFIX`, and `MSUFFIX`
   (d) Set `BINDDN` to admin cn
   (e) Set `UTEMPLATE="/etc/ldapscripts/ldapadduser.template"`
3. `sudo cp /usr/share/doc/ldapscripts/examples/ldapadduser.template.sample /etc/ldapscripts/ldapadduser.template`
4. Edit `/etc/ldapscripts/ldapadduser.template`
   (a) `homeDirectory: /mnt/nfs/<user>`
5. Add password to the script
   (a) `sudo sh -c "echo -n 'password' > /etc/ldapscripts/ldapscripts.passwd"`
   (b) `sudo chmod 400 /etc/ldapscripts/ldapscripts.passwd`

A.2.3 OrangeFS and MPICH2

1. **Prepare system for OrangeFS Software Installation**
   (a) Install required packages on all nodes in cluster.

   ```
   $ sudo apt-get install bison flex libfuse-dev libdb-dev
   libssl-dev
   ```

   (b) Download orangefs-2.9.5.tar.gz from `www.orangefs.org` to the designated build node.

   ```
   $ wget http://download.orangefs.org/current/source/orangefs-2.9.5.tar.gz
   ```

   ```
   $ tar -xzf orangefs-2.9.5.tar.gz -C /dir
   ```
   where *dir*= path to desired location to place OFS source directory

(c) Create a directory on all the nodes in the cluster to mount OFS

```
$ sudo mkdir /mnt/orangefs
```

(d) Create a system configuration file on all the nodes that will contain the files system information

```
$ sudo touch /etc/pvfs2tab
```

(e) Edit the pvfs2tab file to contain the location of the server's OFS mount point

```
$ sudo echo "storage /mnt/orangefs pvfs2 defaults,noauto 0,0" >>
/etc/pvfs2tab
```

where *storage*=/dev/sda1 for server nodes or tcp://*server1*:3334/orangefs for client nodes

2. **Build and Configure Nodes**

(a) **Servers Nodes**

i. Navigate to directory containing OrangeFS source files.

```
$ cd /dir/orangefs-2.9.5
```

ii. Configure the build node by running:

```
$ /dir/orangefs-2.9.5/configure --prefix=/opt/orangefs
    --enable-fuse --enable-shared
```

where *dir*/orangefs-2.9.5= path to source files

```
$ sudo make
$ sudo make install
```

iii. Copy configured source files from build node to the node designated as the client build node

```
$ sudo scp -r /dir/orangefs-2.9.5 node:/dir/
```

where *dir* = path to source files

(b) **Client Nodes**[2]

i. Configure the client build node by running:

```
$ /dir/orangefs-2.9.5/configure --prefix=/opt/orangefs
--disable-opt --disable-usrint --disable-server --enable-fuse
--enable-shared
```

where *dir*= path to copied OFS build from server

```
$ sudo make
$ sudo make install
```

[2]Rasbpian Jessie/linux kernel 4.4.11v7 cannot build the kernel module necessary to run the clients. The clients must be reconfigured as FUSE clients from the original build node to bypass the linux kernel. This requires the clients to be rebuilt from the original build node configuration.

ii. Copy the newly created executable software from the client build node to all the client nodes in the cluster

```
$ sudo scp -r /opt/orangefs node:/opt/
```

3. **Install and Configure MPICH2**

   (a) Download the MPICH 3.2 binary tar.gz file from MPICH

   ```
   $ tar http://www.mpich.org/static/downloads/3.2/mpich-3.2.tar.gz -C /dir/
   ```

   where *dir* = path to MPICH2 source files

   (b) Avoid possible MPICH and PVFS compilation conflict by running:

   ```
   $ sed -i s/ADIOI_PVFS2_IReadContig/NULL/ src/mpi/romio/adio/ad_pvfs2/ad_pvfs2.c
   $ sed -i s/ADIOI_PVFS2_IWriteContig/NULL/ src/mpi/romio/adio/ad_pvfs2/ad_pvfs2.c
   ```

   (c) Compile MPICH2 to run with OrangeFS support.

   ```
   $ /dir/configure --prefix=/opt/mpich2 --enable-romio
   --enable-shared --with-pvfs2=/opt/orangefs
   --with-file-system=pvfs2
   $ sudo make all install
   ```

   (d) Set library to path to point to MPICH libraries

   ```
   $ export LD_LIBRARY_PATH=/opt/mpich-3.0.4/lib:$LD_LIBRARY_PATH
   ```

   (e) Open /etc/profile.d/mpich.sh in a text editor and add
   `export LD_LIBRARY_PATH=/opt/mpich-3.0.4/lib:$LD_LIBRARY_PATH`
   to the file.

4. **Initialize and run OrangeFS on the cluster**

   (a) **Server**

   i. Navigate to the OrangeFS installation drive on the server node.

   ```
   $ cd /opt/orangeFS
   ```

   ii. Generate the configuration file specifying the file system parameters

   ```
   $ sudo bin/pvfs2-genconfig /opt/orangefs/orangefs.conf
   ```

   iii. Initialize the OrangeFS data and metadata storage locations

   ```
   $ sudo sbin/pvfs2-server -f /opt/orangefs/orangefs.conf
   ```

   -f creates the data and meta storage space
   -r removes the data and meta storage space

   iv. Start the OrangeFS server

   ```
   $ sudo sbin/pvfs2-server /opt/orangefs/orangefs.conf
   ```

   (b) **Clients**

i. On each client mount the OrangeFS file system

```
$ sudo /opt/orangefs/bin/pvfs2fuse /mnt/orangefs -o
   fs_spec=tcp://server1:3334/orangefs
```

A.2.4 Monitor Service

1. Clone the node_monitor repository.
   (a) `git clone git@bitbucket.org:bigorangebramble/node_monitor.git`
2. Run the included ansible playbook to install and start the service.
   (a) `ansible-playbook start_service.yml`

A.2.5 Power on BOB

1. Turn on the two power strips.
2. Wait for the system to boot.
3. Check that the system is booted.
   (a) Login as pi user.
   (b) `$ ansible cluster -m ping`
   If all of the nodes respond then your know that the system is booted.
4. Restart the services.
   (a) Login as pi user.
   (b) `$ ansible-playbook ~/git/ansible_playbook/restart_services.yml`

A.2.6 Restart BOB

1. Run the restart script.
   (a) Login as pi user.
   (b) `$ ansible-playbook ~/git/ansible_playbook/reboot.yml`
2. Wait for the system to reboot.
3. Check that the system is booted.
   (a) Login as pi user.
   (b) `$ ansible cluster -m ping`
   If all of the nodes respond then your know that the system is booted.
4. Restart the services.

   (a) Login as pi user.
   (b) `$ ansible-playbook ~/git/ansible_playbook/restart_services.yml`

A.2.7 Shutdown BOB

1. Run the shutdown script.
   (a) Login as pi user.
   (b) `$ ansible-playbook ~/git/ansible_playbook/shutdown.yml`
2. Wait for the system to shutdown.
   (a) System is shutdown once both lights on the Pi are solid for all of the Pis.
3. Turn off the two power strips.

A.2.8 LDAP Usage

To manage users and groups, use the LDAP Scripts. The available commands are:

- ldapadd
- ldapaddgroup
- ldapaddmachine
- ldapadduser
- ldapaddusertogroup
- ldapcompare
- ldapdelete
- ldapdeletegroup
- ldapdeletemachine
- ldapdeleteuser
- ldapdeleteuserfromgroup
- ldapexop
- ldapfinger
- ldapgid
- ldapid
- ldapinit
- ldapmodify
- ldapmodifygroup
- ldapmodifymachine
- ldapmodifyuser
- ldapmodrdn
- ldappasswd
- ldaprenamegroup
- ldaprenamemachine
- ldaprenameuser
- ldapsearch
- ldapsetpasswd
- ldapsetprimarygroup
- ldapurl
- ldapwhoami
- lsldap

Use the man pages to learn how to use the commands.

A.2.9 Start Monitor GUI

This guide goes over how to start the monitoring GUI on the monitoring node.

1. Open python2 IDE
   (a) Click on applications.
   (b) Go to programming.
   (c) Click on python2 IDE.
2. Open the monitoring program.
   (a) Click `File` → `Open`.
   (b) Open the file `~/git/Monitor_Gui/monitorgui.py`
3. Run the program.
   (a) click `Run` → `Run Module`

A.2.10 Monitor GUI via X11 forwarding

The monitor GUI can be run remotely using X11 forwarding. This usage guide goes over how to launch the monitoring GUI. In order for this method to work you will need ssh and an X11 server. Linux and OS X have both built in.

1. Connect to BOB with ssh compression and X11 forwarding.
   (a) `$ ssh -C -X <username>@bob.eecs.utk.edu`
2. You will need to add your ssh key to pim. You only have to do this step once.
   (a) `$ ssh-keygen`
   (b) Use the default for all the options and keep pressing enter until the key is generated.
   (c) `$ ssh-copy-id pim`
3. Run the monitor GUI.
   (a) `$ monitorgui`
   The program may take up to a minute to launch. It will run in the background of the shell by default, so please be patient while it launches.

Appendix B Application Install and Usage Guide

B.1 Frameworks

B.1.1 TensorFlow

1. **Installation of TensorFlow**

   (a) Install dependencies for TensorFlow.

   ```
   For Python 2.7:
   $ sudo apt-get install python-pip python-dev

   For Python 3.3+:
   $ sudo apt-get install python3-pip python3-dev
   ```

   (b) Download wheel file from repository and install it.

   ```
   For Python 2.7:
   $ wget https://github.com/samjabrahams/tensorflow-on-raspberry-pi/
     raw/master/bin/tensorflow-0.9.0-cp27-none-linux_armv7l.whl
   $ sudo pip install tensorflow-0.9.0-cp27-none-linux_armv7l.whl

   For Python 3.3+:
   $ wget https://github.com/samjabrahams/tensorflow-on-raspberry-pi/
     raw/master/bin/tensorflow-0.9.0-py3-none-any.whl
   $ sudo pip install tensorflow-0.9.0-py3-none-any.whl
   ```

   More information about installation of TensorFlow on the Raspberry Pi can be found by visiting the GitHub repository URL provided in the references section[24].

   (c) Clone BOB TensorFlow application from BitBucket.

   ```
   $  https://bitbucket.org/bigorangebramble/app-disttf.git
   ```

2. **Distributed TensorFlow Execution**
   Begin the program by running the included sbatch script.

   ```
   $ sbatch run_mnist_asyn.sh
   ```

   By default the script uses 9 nodes, including 1 ps (parameter server) node and 8 worker nodes. The script can be edited to utilize more nodes for distributed purpose. Currently, node assignment is hard-coded in the script as TensorFlow needs to specify each node for the designated task. This script calls a Python program, mnist_asyn.py, which is largely based on the example program found on Tensorflow's repository[25].

B.2 Applications

B.2.1 HPL

1. Clone the latest OpenBLAS repository from Github.

   ```
   $ git clone https://github.com/xianyi/OpenBLAS.git
   ```

2. Build OpenBLAS by navigating to its directory and running its included Makefile. This makefile automatically recognizes the ARM processor and will build an optimized ARMv7 binary with VFPv3-32 instructions.

   ```
   $ cd OpenBLAS
   $ make
   ```

3. Install the OpenBLAS library to your location of choice.

   ```
   $ make install PREFIX=/home/user/OpenBLAS
   ```

4. Download the HPL source code from Netlib and extract it to a directory.

   ```
   $ wget http://www.netlib.org/benchmark/hpl/hpl-2.2.tar.gz
   $ tar xvf hpl-2.2.tar.gz
   ```

5. Prepare the HPL Makefile for the Raspberry Pi.

   ```
   $ cd hpl-2.2
   $ setup/make_generic
   $ cp setup/Make.UNKNOWN Make.rpi
   ```

6. Modify the generic Makefile to match the appropriate paths and options for Raspbian and OpenBLAS.

7. Make HPL

   ```
   $ make arch=rpi
   ```

8. Configure the HPL parameters. Here are some guidelines:

   (a) Matrix Size (N): $N = \sqrt{\text{nodes} \times 10500^2}$

   (b) Block Size (NB): 100

   (c) Process Grid (P, Q): $P \times Q = $ nodes, typically best to be as close to square as possible

   (d) NBMIN, NDIV: 2 or 4

   (e) Broadcast (BCAST): Blong (4), Helps compensate for the low bandwidth of the 100 Mbit Ethernet interconnect

   (f) Swapping Threshold should match the block size.

9. To run HPL, ensure each node can access both HPL.dat and XHPL. Then, execute XHPL through SLURM.

```
$ srun -N number_of_nodes ./xhpl
```

B.2.2 HPCG

- Download and extract the HPCG source.

```
$ wget http://www.hpcg-benchmark.org/downloads/hpcg-3.0.tar.gz
$ gunzip hpcg-3.0.tar.gz; tar -xvf hcpg-3.0.tar
```

- Create a Makefile for your architecture

```
$ cp setup/Make.MPI_GCC_OMP setup/Make.PI
```

- Edit the Makefile to have correct paths to MPI
- Create your build directory

```
$ cd hpcg-3.0
$ mkdir build
$ cd build
$ ../configure PI
```

- Compile HPCG

```
$ make
```

- Run HPCG with Quick Path flag

```
$ srun -N (nodes) ./xhpcg --rt=0
```

B.2.3 SPH

1. To begin, clone the SPH repository from Bitbucket.

```
$ git clone https://bitbucket.org/bigorangebramble/app-sph.git
```

2. The included makefile should work on any Raspberry Pi 3 cluster with the proper dependencies installed. Using the `make` command, the executables `sph.out` and `slurm_target` should be compiled into the `app-sph/bin` directory. The compilation will likely take a few minutes to complete.

```
$ cd app-SPH
$ make
$ ls bin
```

3. (Optional) This application supports an Xbox 360 controller through the use of `xboxdrv`, a driver available through `apt-get`

```
$ sudo apt-get install xboxdrv
```

Once installed, the Xbox controller needs to be configured using the provided configuration files. Using `controller_1.cnf` allows the user to control all parameters from within the program without the ability to terminate it, ideal for a long running display. `controller_2.cnf` functions identically to the first, with the exception that it can exit the program through the select button.

In order to launch the driver run either of the following commands from the app-SPH directory:

```
$ sudo xboxdrv -s -d -c controller_1.cnf &
```

or

```
$ sudo xboxdrv -s -d -c controller_1.cnf &
```

In the case of a successful launch, you should see a message similar to the following:

```
Your Xbox/Xbox360 controller should now be available as:
  /dev/input/js0
  /dev/input/event2
```

If your controller has been configured as a different event number (e.g. `/dev/input/event3` due to additional peripheral devices), you will need to modify the source code in `egl_utils.c` on line 181 to match your event number.

In case of a failure in configuring the controller, follow the directions provided by the driver to resolve your issue.

4. With the executables ready, the launch script `SPH_Launch.sh` can be run from the host node, `pih0` in our case.

   ```
   $ sh SPH_Launch.sh
   ```

   This will make the necessary `salloc`, `squeue`, and `mpirun` calls to launch the process. This will begin the simulation, allowing the user to perturb the system as they see fit. Various presets can be reached with the `abxy` buttons, the variables can be modified directly with the D-pad, and the number of nodes is controlled with the left and right bumper buttons.

B.2.4 PiBrot

1. To begin, clone the PiBrot repository from Bitbucket.

   ```
   $ git clone https://bitbucket.org/bigorangebramble/app-PiBrot.git
   ```

2. The included makefile should work on any Raspberry Pi 3 cluster with the proper dependencies installed. Using the `make` command, the executables `pibrot` and `slurm_target` should be compiled into the `app-PiBrot/bin` directory. The compilation will likely take a few minutes to complete.

```
$ cd app-PiBrot
$ make
$ ls bin
```

3. (Optional) This application supports an Xbox 360 controller through the use of `xboxdrv`, a driver available through `apt-get`

   ```
   $ sudo apt-get install xboxdrv
   ```

   Once installed, the Xbox controller needs to be configured using the provided configuration file. In order to launch the driver run the following command from the app-PiBrot directory:

   ```
   $ sudo xboxdrv -s -d -c controller.cnf &
   ```

   In the case of a successful launch, you should see a message similar to the following:

   ```
   Your Xbox/Xbox360 controller should now be available as:
     /dev/input/js0
     /dev/input/event2
   ```

   If your controller has been configured as a different event number (e.g. `/dev/input/event3` due to additional peripheral devices), you will need to modify the source code in `egl_utils.c` on line 174 to match your event number.

   In case of a failure in configuring the controller, follow the directions provided by the driver to resolve your issue.

4. With the executables ready, the launch script `SPH_Launch.sh` can be run from the host node, `pih0` in our case.

   ```
   $ sh SPH_Launch.sh
   ```

   This will make the necessary `salloc`, `squeue`, and `mpirun` calls to launch the process. From within the application, navigate the cursor over the start button and press `A` to launch the race.

B.2.5 Parallel Pi

Begin by cloning the Parallel Pi repository from Bitbucket.

```
$ git clone https://bitbucket.org/bigorangebramble/app-PiBrot.git
```

The repository contains two files: `bbp.py`, which is responsible for completing the actual pi calculation, and `slurm_pi.sh`, a Slurm batch script which manages options such as the number of worker nodes and the output file. The `bbp.py` script's only dependency is the `mpi4py` Python package. Therefore, the script can be run directly from the command line using the `mpirun` command. However, it is strongly recommended that you utilize Slurm and the included batch script, which is invoked from the command line using the sbatch command.

```
$ sbatch slurm_pi.sh
```

The included batch script defaults to 8 nodes with 1 CPU per task, for a total of 32 parallel tasks. The script can be edited directly to use a different number of nodes, or the default node number can be overridden when calling sbatch using the -N flag. The default output file is `pi.txt`, which can also be changed by editing the batch script or by using the -o flag when calling sbatch from the command line. The output file will contain the calculated value of pi, time elapsed during the calculation, the set precision of Python's decimal module, and the number processors utilized by the script. Within the actual `bbp.py` script, decimal precision and the length of the finite series can both be modified if the user wishes to calculate a more accurate approximation of pi. However, increasing decimal precision can quickly lead to an exponentially greater runtime with minimal increase in the accuracy of the approximation.

B.2.6 Monte Carlo

Before running your program using the Monte Carlo application, please note that the current version does not allow any command line arguments to your program. It also only supports C/C++ or Python executables. The path to the output file from the Monte Carlo application will be placed either in the same directory that you run the GUI from or, if you are using command line, the directory where the batch file is submitted.

After logging into BOB, copy the Monte Carlo application to your own directory.

```
username@pih0:~ $ cp -R hduser/montecarlo mymontecarlo
username@pih0:~ $ cd mymontecarlo
```

For the GUI:
Run the GUI by entering `python mc_gui.py`, then input the executable path, maximum run time, output filename, and number of CPUs requested. Once you submit your information, you can check your job by exiting the GUI and typing `squeue`.

For command line:
The name of the python script to create and submit a batch file for the Monte Carlo application is `mc_batch.py`. The options to run are the same as for the GUI, but they can be specified as follows:

```
usage: python mc_batch.py [-h] [-i INPUTAPP] [-o OUTFILE] [-r NUMRUNS]
    [-max [MAXRUNTIME]] [-dl [DEADLINE]] [-N [CPUS]]
```

Argument Descriptions:

```
  -h, --help         For Help
  -i INPUTAPP        REQUIRED: Path to executable
  -o OUTFILE         REQUIRED: Name of output file
  -r NUMRUNS         REQUIRED: Total number of runs
  -max [MAXRUNTIME]  OPTIONAL: maximum total run time
  -dl [DEADLINE]     OPTIONAL: deadline time
  -N [CPUS]          OPTIONAL: numbers of CPUs
```

B.2.7 Numeric Integration

1. After logging in, navigate to the numeric-integration folder. Open the `reimann-parallel.py` program and type the function you want to integrate in the `func()` subroutine.

   ```
   #DEFINE YOUR FUNCTION HERE!
   def func(x):
       y = numpy.cos(x)
       return y
   ```

2. In the main subroutine, set the beginning of the domain you want to integrate over in the `xStart` variable. Then set the end of the domain you want to integrate over in the `xEnd` variable.

   ```
   #SET THE DOMAIN BEGINNING AND END
   xStart = 0.0
   xEnd = 1.0 * scipy.pi
   ```

3. (Optional) If you want to create a higher amount of accuracy (or create a more calculation intensive program), you can change the number of rectangles per node in the `samplesPerRank` variable. The default number of rectangles is 10. The error between the Riemann Sums calculation and the actual integral calculation produced by the scipy library will print at the end of the run.

   ```
   #SET NUMBER OF SAMPLES (rectangles & accuracy)
   samplesPerRank = 10
   ```

4. Save and quit the `riemann-parallel.py` file.

5. (Optional) Open the `slurm_riemann.sh` file and edit number of nodes to be used for your calculation. Save this file.

   ```
   #SBATCH -N 32
   ```

6. (Optional) You can also change the name of the output file produced by your program run. Save and quit this file.

   ```
   #SBATCH --output=out.txt
   ```

7. From command line, you can now run a batch job by typing `sbatch slurm_riemann.sh`. When the job is finished, you can find your results in the specified output file.

B.2.8 DANNA

1. The DANNA simulator and evolutionary optimization systems are not currently open source. Please contact the Neuromorphic Computing group at the University of Tennessee, Knoxville to inquire about obtaining the system.

2. Create a folder to house the different repositories

```
~$ mkdir danna_repos
```

3. Clone the DANNA repository and the DANNA distributed EO repository into this folder.

```
~$ cd danna_repos
~/danna_repos$ git clone (danna repo)
~/danna_repos$ git clone (danna eo repo)
```

4. Build the DANNA simulator, EO, and applications.

```
~/danna_repos$ cd danna
~/danna_repos/danna$ make sim
~/danna_repos/danna$ make eo
~/danna_repos/danna$ make apps
```

5. Build the distributed DANNA EO for the pole balancer application.

```
~/danna_repos/danna$ cd ../distributed_danna_eo
~/danna_repos/distributed_danna_eo$ make
~/danna_repos/distributed_danna_eo$ cd ../danna/Applications
~/danna_repos/danna/Applications$ cd danna-app-polebalancer
danna-app-polebalancer$ make bin/PBDEO
```

6. Run the distributed evolutionary optimization on the pole balancer app using SLURM to launch the job. The distributed system can use several technique to farm out the work. One of the following is recommended for BOB:

   - Master-Slave (ms)
   - Master-Master-Slave (mms)

```
$ srun -N (number of nodes) bin/PBDEO (seed) (ms or mms)
```

B.2.9 Fire Dynamics Simulator

1. **Installing FDS** Because there is currently no 64-bit operating system for the Raspberry Pi, it is necessary to install the 32-bit version of FDS. The latest FDS version supporting a 32-bit OS is 6.1.1. This guide assumes that mpich2 and slurm have both been properly installed and configured for submitting parallel jobs to the cluster. See other documentation on the Big Orange Bramble for details about how to do this.

   First, install git if you haven't already:

```
$ sudo apt-get install git
```

   Create a git directory if you don't have one:

```
$ cd ~
$ mkdir git
```

Clone the FDS-SMV repository maintained by NIST [6] into your git directory:

```
$ cd git
$ git clone https://github.com/firemodels/fds-smv.git
```

Now revert the repo back to its 6.1.1 state:

```
$ git reset --hard 898b35a
```

Navigate to the FDS_Compilation directory and list its contents. You should see the mpi_gnu_linux directory. If so, run the make command to build the FDS binary with the gnu compiler with mpich2 for 32-bit linux:

```
$ cd FDS_Compilation
$ make mpi_gnu_linux
```

After completion you should list the directory contents and see an fds_mpi_gnu_linux binary. You can create an alias in your bashrc to run this binary when you type the bash command:

```
$ fds test.fds
```

Otherwise you can just run the binary directly from the FDS_Compilation directory or move it to another directory.

2. **Testing the FDS Install**

   You can test the FDS install by running FDS on one of the examples provided in the Verification directory in the FDS-SMV repository. For example, you can run the water_ice_water.fds with this command from within the FDS_Compilation directory:

   ```
   $ ./fds_mpi_gnu_linux ../Verification/water_ice_water.fds
   ```

   This should run the water_ice_water FDS model with one mesh assigned to one thread.

3. **Running FDS With Slurm** Slurm handles splitting the FDS meshes into processes and threads for you. You can specify the number of cluster nodes to run the model on with the -N option, or you can specify the number of threads with the -n option. You specify which account the job will be assigned to with the -A option followed by your username. For example, to run a model with four meshes assigned to four nodes use the following command with an appropriate FDS file:

```
$ srun -A cwill133 -N 4 fds_mpi_gnu_linux sample_model_4_meshes.fds
```

This will dedicate one core from four separate nodes to each of the four meshes. Alternatively you can run the model on a single node on each of the four cores:

```
$ srun -A cwill133 -n 4 fds_mpi_gnu_linux sample_model_4_meshes.fds
```

4. **Notes:** To fully utilize each core on each node, it is necessary to have node x core count per node n̄umber of meshes. For example, to run on eight nodes with four cores each, you must have thirty-two meshes. It is also important to note that you can run n number of meshes on m number of threads so long as $n \geq m$. In other words, you can only run as many threads as there are meshes. It is possible to assign multiple meshes to a single thread if the number of meshes exceeds the number of available cores.

Appendix C Daughter Card Schematic

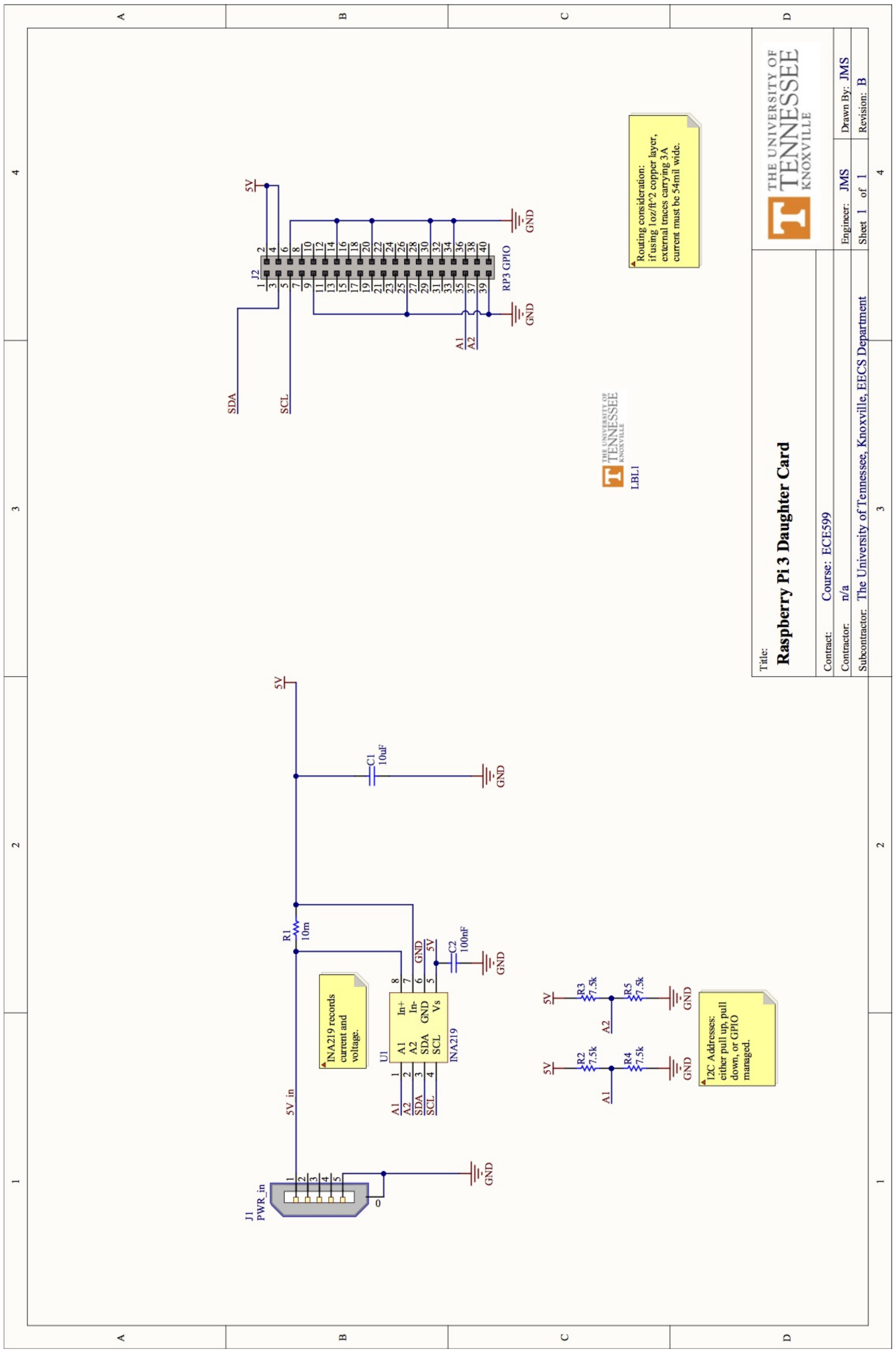
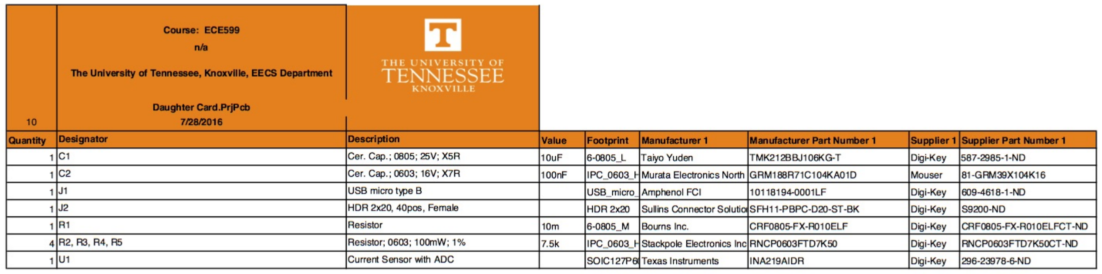

J2
RP3 GPIO
5V
GND
SDA
SCL
A1
A2
LBL1
Routing consideration: if using 1oz/ft^2 copper layer, external traces carrying 3A current must be 54mil wide.
THE UNIVERSITY OF TENNESSEE KNOXVILLE
Title:
Raspberry Pi 3 Daughter Card
Contract:
Course: ECE599
Contractor: n/a
Subcontractor: The University of Tennessee, Knoxville, EECS Department
Engineer: JMS
Drawn By: JMS
Sheet 1 of 1
Revision: B
C1
10uF
R1
10m
C2
100nF
U1
INA219
In+
In-
Vs
INA219 records current and voltage.
5V_in
J1
PWR_in
R2 7.5k
R3 7.5k
R4 7.5k
R5 7.5k
I2C Addresses: either pull up, pull down, or GPIO managed.

Figure 41: The Daughter Card Schematic Diagram

Appendix D Daughter Card Bill of Materials



Course: ECE599

n/a

The University of Tennessee, Knoxville, EECS Department

Daughter Card.PrjPcb

7/28/2016

10

Quantity	Designator	Description	Value	Footprint	Manufacturer 1	Manufacturer Part Number 1	Supplier 1	Supplier Part Number 1
1	C1	Cer. Cap.; 0805; 25V; X5R	10uF	6-0805_L	Taiyo Yuden	TMK212BBJ106KG-T	Digi-Key	587-2985-1-ND
1	C2	Cer. Cap.; 0603; 16V; X7R	100nF	IPC_0603_H	Murata Electronics North	GRM188R71C104KA01D	Mouser	81-GRM39X104K16
1	J1	USB micro type B		USB_micro_	Amphenol FCI	10118194-0001LF	Digi-Key	609-4618-1-ND
1	J2	HDR 2x20, 40pos, Female		HDR 2x20	Sullins Connector Solutio	SFH11-PBPC-D20-ST-BK	Digi-Key	S9200-ND
1	R1	Resistor	10m	6-0805_M	Bourns Inc.	CRF0805-FX-R010ELF	Digi-Key	CRF0805-FX-R010ELFCT-ND
4	R2, R3, R4, R5	Resistor; 0603; 100mW; 1%	7.5k	IPC_0603_H	Stackpole Electronics Inc	RNCP0603FTD7K50	Digi-Key	RNCP0603FTD7K50CT-ND
1	U1	Current Sensor with ADC		SOIC127P6	Texas Instruments	INA219AIDR	Digi-Key	296-23978-6-ND

Figure 42: The Daughter Card Bill of Materials