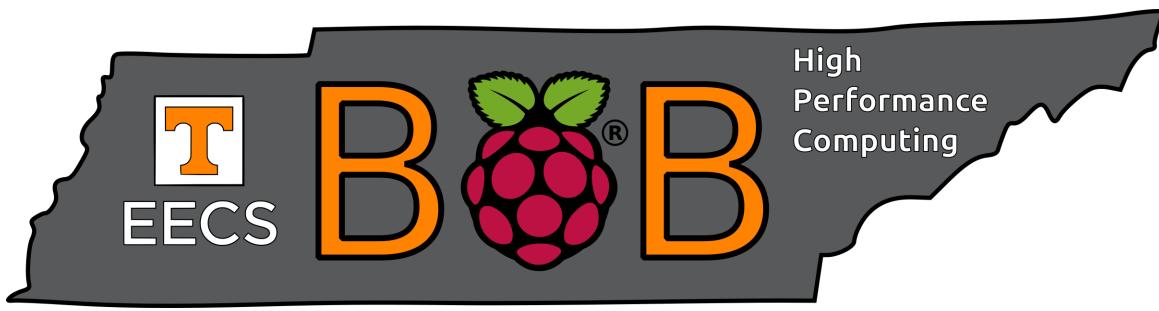


THE UNIVERSITY OF TENNESSEE
KNOXVILLE

CS 594

SUPERCOMPUTER DESIGN AND ANALYSIS

Big Orange Bramble+
BOB and ALICE



Supervisor:
Dr. Mark DEAN

Team Leader:
Kelley DEUSO

Team:
J. Parker MITCHELL Gregory SIMPSON
Ellias PALCU Caleb WILLIAMSON
Jordan SANGID Aaron YOUNG

December 8, 2016

Abstract

This project was an expansion of The University of Tennessee’s Big Orange Bramble (BOB) project from Summer 2016, which aimed to develop a high performance computing cluster from 68 quad-core ARMv8 64-bit Raspberry Pi 3s for faculty and students of the Electrical Engineering and Computer Science department to use for research and education. The success of the BOB project led to a proposal for a secondary cluster, which was intended to be combined with BOB to form a larger, heterogeneous cluster. This secondary cluster was created from 32 64-bit Pine64s and 12 Nvidia TX1 GPUs. It was also a requirement to perform benchmarking on each subset of architecture and provide a performance comparison to BOB. Additionally, several more applications were developed for BOB, including a facial recognition application, a weather forecasting, and a Gillespie algorithm application. An installation and usage guide was also developed for these applications for anyone wishing to utilize them.

N.B.: For the purposes of simplification for this document, the 68-node Raspberry Pi cluster will be referred to as “BOB”, the 32 Pine64 + 12 Nvidia TX1 GPU cluster will be referred to as “ALICE”, and both clusters together will be “BOB+”. The previous document for BOB can be found at <http://web.eecs.utk.edu/~markdean>.

Table of Contents

1	Introduction	1
2	Requirements	1
3	Assumptions	2
4	List of Parts and Materials	2
4.1	Preliminary Parts List	2
5	Project Plan	2
6	Implementation	4
6.1	Hardware Development	4
6.1.1	Daughter Card for Pine64	4
6.1.2	Enclosure	6
6.1.3	Monitor Node	8
6.1.4	Fan & Reset Controller	8
6.2	Operations & Systems Development	11
6.2.1	Pine64	11
6.2.2	Nvidia Jetson TX1	11
6.2.3	Heterogeneous System	11
Phase One	12
Phase Two	13
6.2.4	Slurm Accounting	15
6.3	Standard Packages	16
6.3.1	Pine64	16
6.3.2	Nvidia Jetson TX1	16
6.4	Frameworks and Tools Development	17
6.4.1	Apache Spark on Raspberry Pi	17
6.4.2	Octave on Raspberry Pi	17
6.4.3	CUDA on Nvidia Jetson TX1	17
6.4.4	Monitoring	17
6.5	Applications Development	19
6.5.1	Facial Recognition	19
6.5.2	Weather Forecasting	22
6.5.3	Gillespie Algorithm	23
7	Testing Results	24
7.1	HPL Benchmark	24
7.1.1	Pine64 Cluster	24
7.1.2	Nvidia TX1 Cluster	26
7.2	HPCG Benchmark	27
7.2.1	Pine64 Cluster	28
7.2.2	Nvidia TX1 Cluster	29

7.3	Facial Recognition	29
7.4	WRF	32
7.5	FDS	34
7.6	Gillespie Algorithm	37
8	Challenges and Future Considerations	38
8.1	Pine64	38
8.1.1	Monitor Issues	39
8.1.2	Rebooting Issues	39
8.1.3	Networking Issues	39
8.1.4	Software Difficulties	40
8.1.5	Cache Issues	40
8.1.6	Reliability Issues	40
8.2	Nvidia	41
9	Conclusion	41
References		43
Appendices		44
Appendix A Hardware/Systems Build Guide		44
A.1	Hardware	44
A.1.1	Daughter Card	44
A.1.2	Temperature Sensor MCP9808	45
A.2	Software	46
A.2.1	Setup MySQL for usage with Slurm	46
A.2.2	Migrate LDAP	46
A.2.3	Backup NFS Drive	47
A.2.4	Setup MySQL Tables for Slurm Accounting	47
Appendix B Application Install and Usage Guide		48
B.1	HPL	48
B.2	HPCG	49
B.3	FDS	49
B.4	WRF	51
B.5	Gillespie Algorithm	51

List of Figures

1	Daughter Card for Raspberry Pi from BOB	4
2	Daughter Card with added Pull-Up Resistors (circled).	5
3	Raspberry Pi Daughter Card attached to Pine64.	6
4	The Bare Enclosure	7
5	The Power Supply Shelf	7
6	The Pine64 nodes and Nvidia TX1s.	8

7	Arduino Mega Shield Modified for Fan Power Distribution.	9
8	Fan Controller	9
9	Fan Wall inside Enclosure.	10
10	Pine64 port containing reset pin	10
11	Hardware Diagram	12
12	Overview of the Job Pack Investigation	14
13	Software Stack	16
14	Main Monitor Page	18
15	Heat Map Page	18
16	Simple example of facial detection with OpenCV.	19
17	Altered function to accommodate facial prediction.	20
18	Text file read in for modifying machine-learning labels.	21
19	Dynamic process management vs. collective communication	22
20	Pine64 Performance across varying block sizes with $N = 6000$	24
21	HPL Performance across Pine64 nodes	25
22	TX1 Performance across varying block sizes	26
23	HPL Performance across TX1 nodes	26
24	HPCG Performance across Pine nodes	28
25	HPCG Performance across TX1 nodes	29
26	KNN facial prediction results.	30
27	Facial Recognition parallelization across BOB.	31
28	Facial Recognition parallelization across ALICE.	32
29	Hurricane Katrina Test Results	34
30	FDS – simple room into multiple meshes on the Raspberry Pi cluster	35
31	FDS – simple room into multiple meshes on ALICE	36
32	<code>example.txt</code> Run # 1 Results	37
33	<code>example.txt</code> Run # 2 Results	38
34	<code>example_inf.txt</code> Run Results - Truncated	38

List of Tables

2	HPL Performance for Pine64 Nodes	25
3	HPL Performance for TX1 Nodes	27
4	HPCG Performance for Pine Nodes	28
5	HPCG Performance for TX1 Nodes	29
6	Relative image count speedups experienced on BOB.	31
7	Relative image count speedups experienced on ALICE.	32
8	FDS – simple room into multiple meshes on BOB	35
9	FDS – simple room into multiple meshes on ALICE	36

1 Introduction

This report outlines the work completed by the group from start to finish in the build and development of BOB's sibling "A Linux Integrated Computing Environment (ALICE)." It will be demonstrated that issues were discovered that either prevented the accomplishment of a requirement or altered the path to achieve specific successes. The beginning sections discuss the framework and organization for the project. This will be followed by details of implementation and the categorization of obstacles of this process. The appendices include both a Build and Usage Guide, for those wishing to continue the endeavors of the group.

2 Requirements

1. Hardware
 - 1.1. The additional cluster must incorporate and function with 12 Nvidia TX1 GPUs.
 - 1.2. The additional cluster must incorporate and function with 32 Pine64s.
 - 1.3. The additional cluster must have hardware for power monitoring and cooling.
 - 1.4. As a stretch goal, cooling features must be extended to prevent thermal throttling.
2. Systems
 - 2.1. Slurm accounting must be added to BOB.
 - 2.2. Slurm accounting must be added to the additional cluster.
 - 2.3. A performance comparison of 32 Pine64s and 32 Raspberry Pi 3's must be completed.
 - 2.4. An evaluation of Nvidia GPUs must be performed through benchmarking.
3. Software
 - 3.1. Octave must be implemented on BOB.
 - 3.2. Spark or Hadoop must be implemented on BOB.
 - 3.3. A facial recognition application must be developed and implemented.
 - 3.4. A weather forecasting application must be developed and implemented.
 - 3.5. A Gillespie algorithm application must be developed and implemented.
 - 3.6. FDS must be expanded on the additional cluster.

3 Assumptions

4 List of Parts and Materials

4.1 Preliminary Parts List

Quantity	Item
32	Pine64
12	Nvidia Jetson TX1
32	15mm x 15mm x 8mm Aluminum Heatsink Cooler Heat Cooling Fin
40	64GB SD Card Ultra
32	Custom Daughter Card for Power Monitoring
14	Anker 3-pack 3ft Micro-USB
16	5 pack, 6ft Flat Ethernet Cables
10	USB 7-Port Hubs
2	5V, 20A Adjustable Power Supplies
6	8-port USB Charging Stations
10	COUGAR CF-V12HPB Vortex Fan
2	PDU - 12 Outlet
2	D-Link Systems 52-Port Gigabit Web Smart Switch including 4 Gigabit SFP Ports (DGS-1210-52)
8	GLC-T Cisco 1000BASE-T SFP Transceiver Module
2	4TB USB HDD
1	Custom 19" Rack Enclosure
1	USB Ethernet adapter
1	Arduino Mega 2560

5 Project Plan

Date	Milestones
8/18/2016	<ol style="list-style-type: none">1. Discuss hardware setup2. Discuss individual roles and interests
8/23/2016	<ol style="list-style-type: none">1. Order parts2. Assign individual roles3. Discuss applications

8/25/2016	<ol style="list-style-type: none"> 1. Begin assembling new cluster 2. Begin applications research and dev on BOB
8/30/2016	<ol style="list-style-type: none"> 1. Daughter card evaluation 2. Hadoop/Spark investigation and discussion 3. Applications dev on BOB
9/6/2016	<ol style="list-style-type: none"> 1. 2 Pine system completion + 2 Pi + 2 GPUs 2. Begin benchmarking 3. Begin documentation 4. Applications dev on BOB
9/8/2016	<ol style="list-style-type: none"> 1. Complete assumptions, parts list, and project plan sections 2. Applications dev on BOB
9/13/2016	<ol style="list-style-type: none"> 1. 8 Pine+Pi+GPU system completion 2. Applications dev on BOB 3. Begin implementation, testing, and results discussions and graphics/diagrams sections
9/15/2016-9/21/2016	<ol style="list-style-type: none"> 1. Applications dev on BOB 2. Continue implementation, testing, and results discussions and graphics/diagrams sections
9/22/2016-10/24/2016	<ol style="list-style-type: none"> 1. 16 Pine+2 Pi+12GPU Pine system completion 2. Applications dev on BOB 3. Continue implementation, testing, and results discussions and graphics/diagrams sections

10/25/2016	<ol style="list-style-type: none"> 1. BOB + 32 Pine + 12 GPU system completion 2. Applications dev on BOB 3. Continue implementation, testing, and results discussions and graphics/diagrams sections
11/22/2016	<ol style="list-style-type: none"> 1. Complete Applications and testing 2. Complete testing and results sections 3. Finalize BOB+ build
12/8/2016	<ol style="list-style-type: none"> 1. Complete ISC paper 2. Complete HPC paper 3. Finalize BOB+ documentation

6 Implementation

6.1 Hardware Development

6.1.1 Daughter Card for Pine64

The intent of the Daughter Card is to provide access to the input power of each individual node for a current sense and voltage sense device. The Daughter Card was initially designed to compliment the physical layout of the Raspberry Pi in the previous project, BOB. The card uses a Texas Instruments INA219 to monitor the voltage drop across a high side, current sense resistor size of $10m\Omega$. The card also sends the measurements to the parent node through I²C. The specifics of the Daughter Card design can be found in the original BOB report and a complete user guide to programming and implementing the Daughter Card is located in [Appendix A.1.1](#))

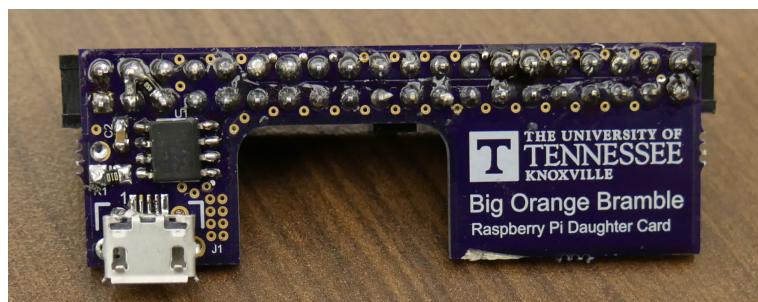


Figure 1: Daughter Card for Raspberry Pi from BOB

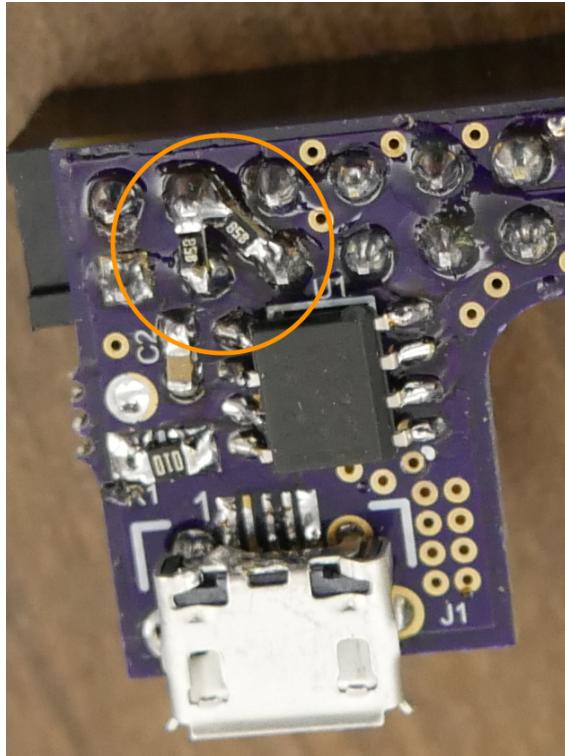


Figure 2: Daughter Card with added Pull-Up Resistors (circled).

After brief evaluations, it was determined the originally designed Daughter Card could be compatible with the Pine64s. The Pine64 includes a port called “PI-2 Connector,” which is identical in function and organization to the Raspberry Pi’s GPIO. Once connected together, the Pine64 was having some inconsistencies in communication with the Daughter Card. This was examined with an oscilloscope and shown that the initial status of the SCL and SDA pins of the I²C were undefined and floating. This was an oversight in the original design of the Daughter Card for BOB considering the I²C controller on the Raspberry Pi had default definitions for the states of these signals. It was concluded that the Pine64s possibly did not include pull-ups or pull-down resistors internal to the I²C controller. To correct this error, two pull-up resistors were tied between SCL and SDA and VDD (5V). This modification can be seen in **Figures 1 & 2**. Once the modifications were performed, 34 Daughter Cards were successfully installed on ALICE (See **Figure 3**. Furthermore, all 68 Raspberry Pis on BOB were fitted with Daughter Cards in addition to the initial 16 that were equipped at the time of the previous BOB report.

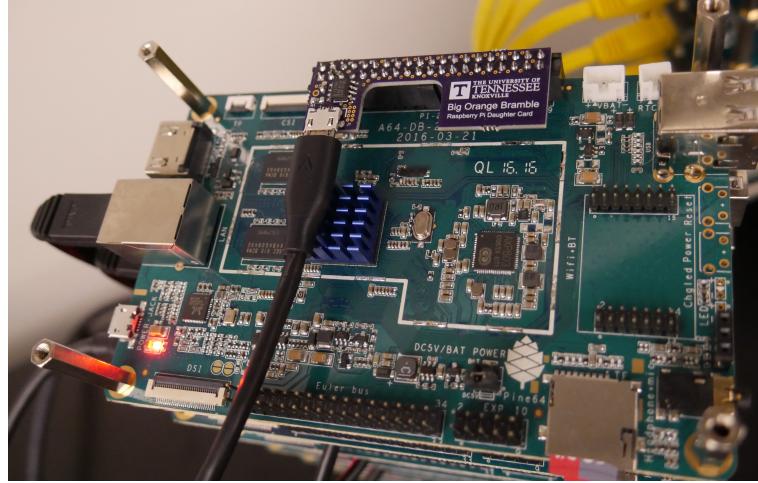


Figure 3: Raspberry Pi Daughter Card attached to Pine64.

It was originally believed that the firmware used for BOB would easily transfer to ALICE. However, after the hardware issues with the daughter card were addressed, certain software issues began to arise. The firmware for the daughter cards relies on two interconnected libraries, one from Adafruit written in C++ for the INA219 [4], and one written in Python available from Github [16] that overlays the Adafruit library and provides high level functions for easier access to the functionality of the INA219.

It was discovered that the Python library overlay made several assumptions, not least of which was that the library would only be used on Raspberry Pi devices. It was necessary to remove several references to Raspberry Pi specific resources and replace them with more generic Linux I²C tools. As such, the firmware Python overlay library for the INA219 should now work for any Pi-like Linux-based board.

6.1.2 Enclosure

The enclosure was designed for the purpose of protecting the nodes from environmental threats and wandering hands. The enclosure was designed within a large 4ft rack, which has locking casters so that the machine could easily be moved if necessary (See **Figure 4**). Rack shelves were used to hold the 4 towers of 8 Pine64s, 3 groups of 4 Nvidia TX1s, and power supplies. The power supplies used were USB charging stations that had 8 separate 5.5V channels capable of supplying a 2.5A load each. Two 6V switch mode power supplies were placed in series to provide power to the 12V fans. The power supply shelf is shown in **Figure 5**. The Pine64s and Nvidia TX1s are shown in **Figure 6**.



Figure 4: The Bare Enclosure

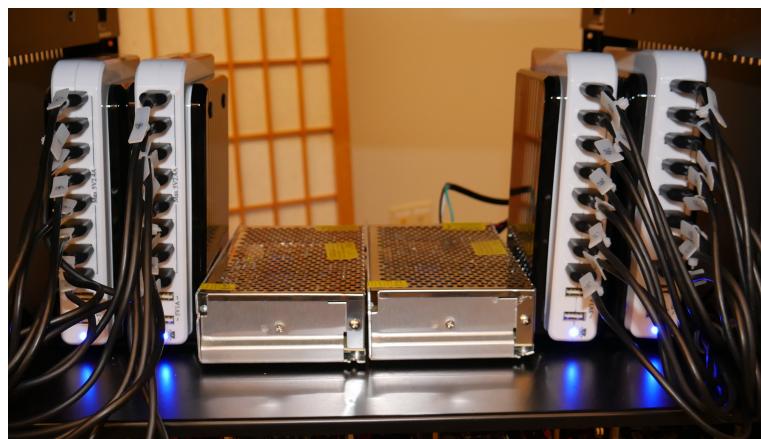


Figure 5: The Power Supply Shelf

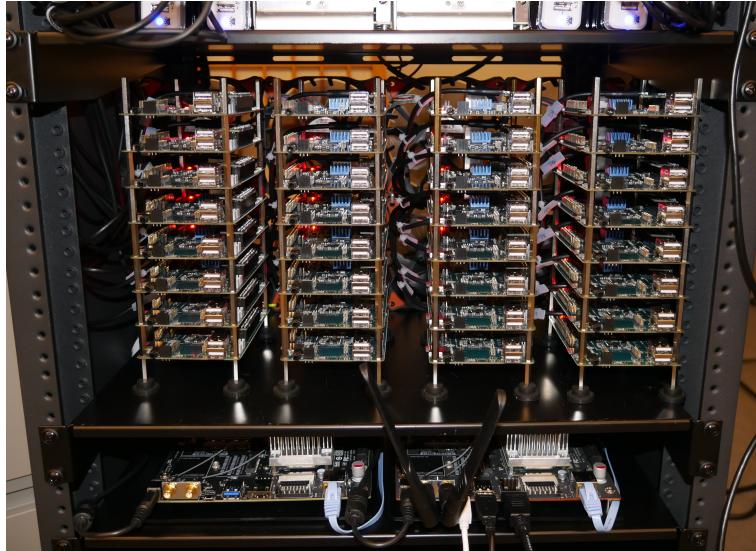


Figure 6: The Pine64 nodes and Nvidia TX1s.

6.1.3 Monitor Node

The monitor node consists of a single Pine64 mated to a 7" touchscreen display. The purpose of the node is to receive information 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 6.4.4**.

6.1.4 Fan & Reset Controller

The Fan & Reset Controller was initially intended to provide active cooling to the Pine64s. Each individual Pine64 would relay core temperature to the monitor node, which would pass the information to an Arduino Mega 2560. The Arduino could then use the information to make a decision to increase or decrease the air flow rate of one or more of the 12 12V fans through PWM control. This would be a closed loop temperature control system.

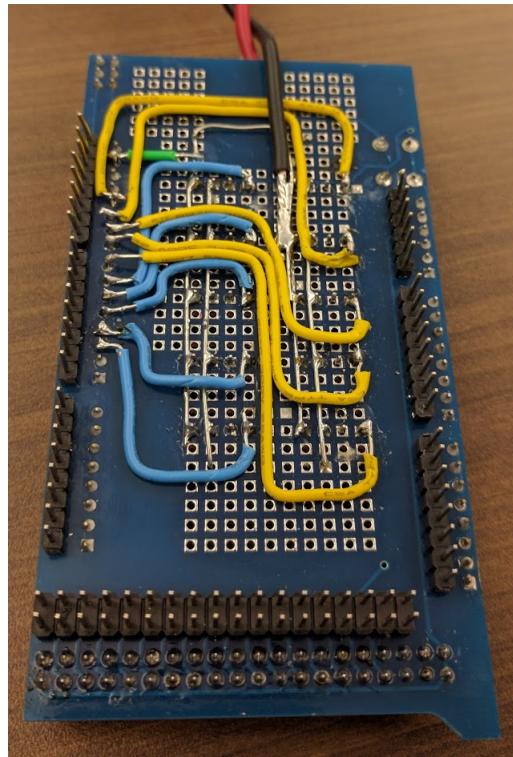


Figure 7: Arduino Mega Shield Modified for Fan Power Distribution.

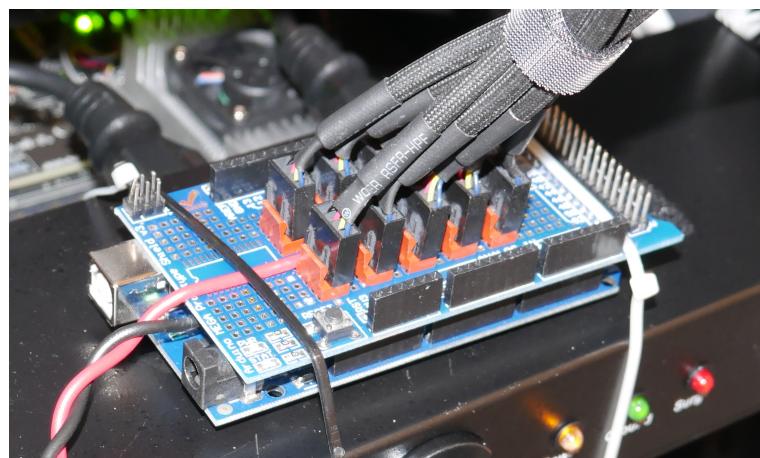


Figure 8: Fan Controller

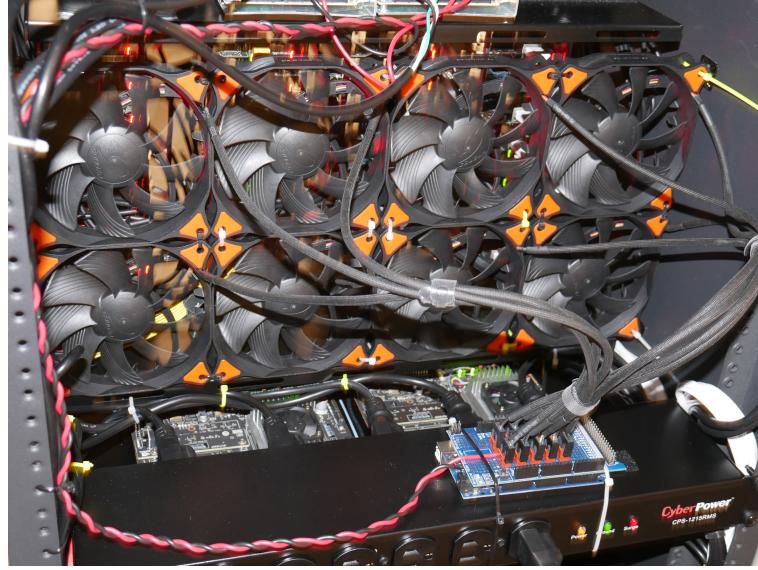


Figure 9: Fan Wall inside Enclosure.

An Arduino shield was modified to send 12V from the power supplies and a separate PWM signal from the Arduino to each of the 12 fans (See **Figures 7 & 8**). The fans were placed behind the Pine64s (See **Figure 9** and connected to the controller. In addition to the CPU core voltages from the Pine64s, the option exists to use the MCP9808 temperature sensor to gather information of the ambient temperature within the enclosure. These sensors were not placed in the system, but the opportunity does exist to be utilized at a later time. These sensors could be placed strategically throughout the enclosure to monitor temperatures in determined critical spaces. An installation guide for the MCP9808 can be referenced in **Appendix A.1.2**.

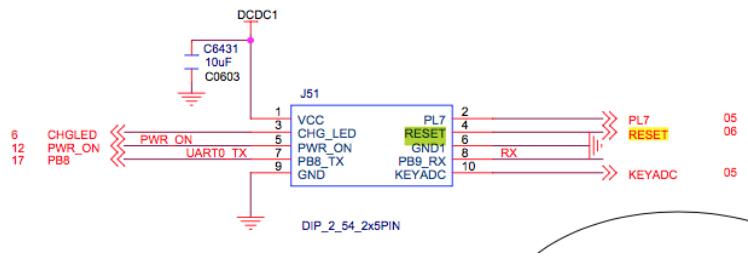


Figure 10: Pine64 port containing reset pin

A secondary function of the Arduino controller is to be able to reset individual Pine64s if an error has been detected. The need for this task is due to inconsistencies in the boot-up of the Pine64s. By connecting a digital output pin of the Arduino to the reset pin on the Pine64, shown in **Figure 10**, the controller will simply drive the pin to logic-low and the Pine64 will enter reboot. It was experimentally shown that the digital ports on the controller must be defined as logic high prior to connecting to the Pine64s, or the Pine64 will enter a reset loop. All 32 Pine64 worker nodes have been connected to the Digital Output pins of the Arduino.

6.2 Operations & Systems Development

6.2.1 Pine64

During the set up of the Pine64s many issues were uncovered. These issues are expounded on in [Appendix 8.1](#). The majority of these issues stem from the fact that the Pine64 is a new device with out much field testing. The issues can be separated into two categories: hardware problems and available software for the device. Many obstacles could be overcome and other problems could be diminished to the point that a functioning cluster made out of Pine64 nodes was able to be created. The system was setup in much the same way as BOB and the operation of ALICE is very similar. Both systems have a login node that is used to access the system. From the login node the other nodes can be accessed via SSH or jobs can be run using the Slurm workload manager. User accounts are handled with LDAP and each user has their own home directory on a NFS drive that is shared on all the nodes.

6.2.2 Nvidia Jetson TX1

The Nvidia Jetson TX1 boards offered a different set of challenges due to its unique Tegra X1 SoC. The Tegra X1 incorporates a quad core ARM Cortex A57 unit with a quad core ARM Cortex A53 unit in an ARM big.LITTLE configuration. Additionally, the X1 offers a Maxwell architecture GPU featuring a single SM with 256 CUDA cores. The CPU and GPU share 4 GB of LPDDR4 RAM.

When first working with the TX1 boards, each must be flashed to the latest software image. The current image runs Ubuntu 16.04 LTS with CUDA 8.0 and is a part of the Jetpack 2.3 release. In addition to the provided software, each TX1 node runs OpenMPI 2 with CUDA-aware support. Similar to the Pine64s, the Nvidia nodes may have jobs scheduled through Slurm on the ALICE head node by selecting the Nvidia partition. The nodes use the LDAP user account service as well as the NFS shared network storage to allow for seamless usage of either partition of the cluster.

6.2.3 Heterogeneous System

The initial goal for New BOB was to create a large heterogeneous cluster made up of Raspberry Pis, Pine64s, and Nvidia TX1s. This larger goal can be broken into two steps or phases. Phase one is to get the different architectures to communicate and function as independent compute partitions. This phase would test the capabilities of each architecture individual and would allow for communication and launching of jobs on nodes of the same type. Phase two would be to combined all of the compute partitions into one large cluster and to setup systems that would allow jobs to be run across nodes of different types.

Phase One

The BOB machine from the previous project has been shown to be a stable system with limited bugs and an overall successful implementation of a Raspberry Pi cluster. Since BOB is functional and needed for additional app development, a DBOB (Development BOB) cluster was setup. This new development cluster became known as ALICE. ALICE was designed to be the Pine64 and Nvidia portion of the larger system. ALICE has its own login/head-node which is an Pine64. It also has the 32 Pine64 worker nodes and 12 Nvidia nodes connected with a Gigabit switch. Additionally, ALICE has a NFS storage drive on a Pine64 and a Raspberry Pi monitoring node. ALICE was designed and setup with the plan of eventually connecting BOB and ALICE together to create a larger heterogeneous cluster (BOB+). To this end, ALICE also has a pseudo BOB head-node which would act like the head-node from the BOB cluster so that connecting the two clusters could be as simple as removing the pseudo head-node and connecting BOB and ALICE together. Ansible is used to manage the cluster and install new software. For ALICE, the Ansible playbook was set up with slight differences to support multiple node architectures. It now has groups parameters that are used to specify parameters that are specific to a certain architecture. It also used an encrypted Ansible vault to store passwords for the different devices. Furthermore, inventory files are now included in the playbook instead of in `/etc/ansible/`. This allows the nodes of the cluster to be selected when running Ansible commands and stored in the repository. One inventory file can be used for BOB and the other for ALICE, when BOB and ALICE are put together, their inventory files can be combined so that it specifies all of the nodes.

Figure 11 shows a diagram of ALICE as just described.

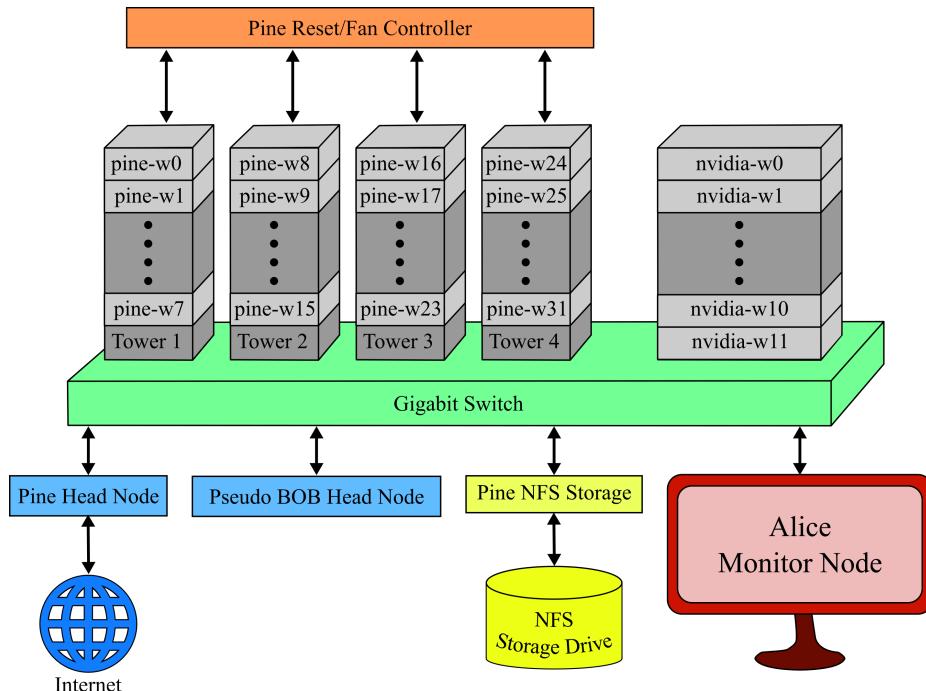


Figure 11: Hardware Diagram

The Pine64s and the Nvidia boards each presented their own challenges, as seen in **Sections 6.2.1 and 6.2.2**, however, each computer node group was able to be set up to work as a cluster of like nodes. Each node type was added to Slurm as a separate partition and Slurm is able to run jobs on one partition at a time.

Phase Two

For phase two the goal was to unify the different compute nodes so that jobs could be run heterogeneously across the entire cluster. The main challenges with this phase are to verify that the software versions on each of the nodes are compatible with the versions on the other nodes, verify that there is a way to launch and manage runs on all of the nodes at the same time, and finally verify that the jobs are put into the same MPI comm world when they are launched. Since Slurm was used successfully on BOB, it was decided to first check if Slurm had support for launching runs on a heterogeneous cluster. A presentation was found that discussed a new feature being added to Slurm called job packs which would allow for separate executables to be launched across the cluster, with each job being placed in the same MPI comm world.[10] Job packs have all the features needed to manage runs across a heterogeneous cluster. Furthermore the presentation stated that jobpacks would be available in Slurm 16.05, which corresponds to the version released in May, 2016. After building and installing Slurm 16.05 on ALICE, a job pack run was tried using pine-w0 and nvidia-w0. The job pack command was not recognized by Slurm and further investigation revealed that job packs were not included in version 16.05. A message was then posted to the Slurm mailing list inquiring about the state of job packs and how to use them. A reply from the mailing list verified that job packs were not in the current version of Slurm, and mentioned an updated job packs presentation that was from the most recent user group meeting. Although not online at the time, the presentation was subsequently made available a week later. The email and the presentation both stated that the job pack feature is now planned for Slurm 17.02.[9] The email also mentioned that the git repository for Slurm has a branch called jobpack which could have the job packs implemented.[13]

The pre-release version of Slurm 17.02 was investigated and shown to not contain job packs implemented in it at this time. The jobpack branch was also tried and was able to successfully launch separate executables for each of the different device partitions; however, the different nodes were not put in the same MPI comm world. Even when launching the jobs on multiple cores of the same node, the different threads of execution were in separate MPI comm worlds. The version of MPICH2 on the nodes was outdated, so new versions of MPICH2 and OpenMPI, were both tried; however, the new versions behave in the same manor and neither were able to launch jobs in the same com world. The conclusion was reached that the job pack feature was not added to the current version of Slurm because it had not been completed. The jobpack branch was also deemed to be not working. Therefore, phase two of the heterogeneous system will have to wait until job packs are finished or a different workload manager is chosen. **Figure 12** shows a graphical flowchart of the job pack investigation elaborated on above.

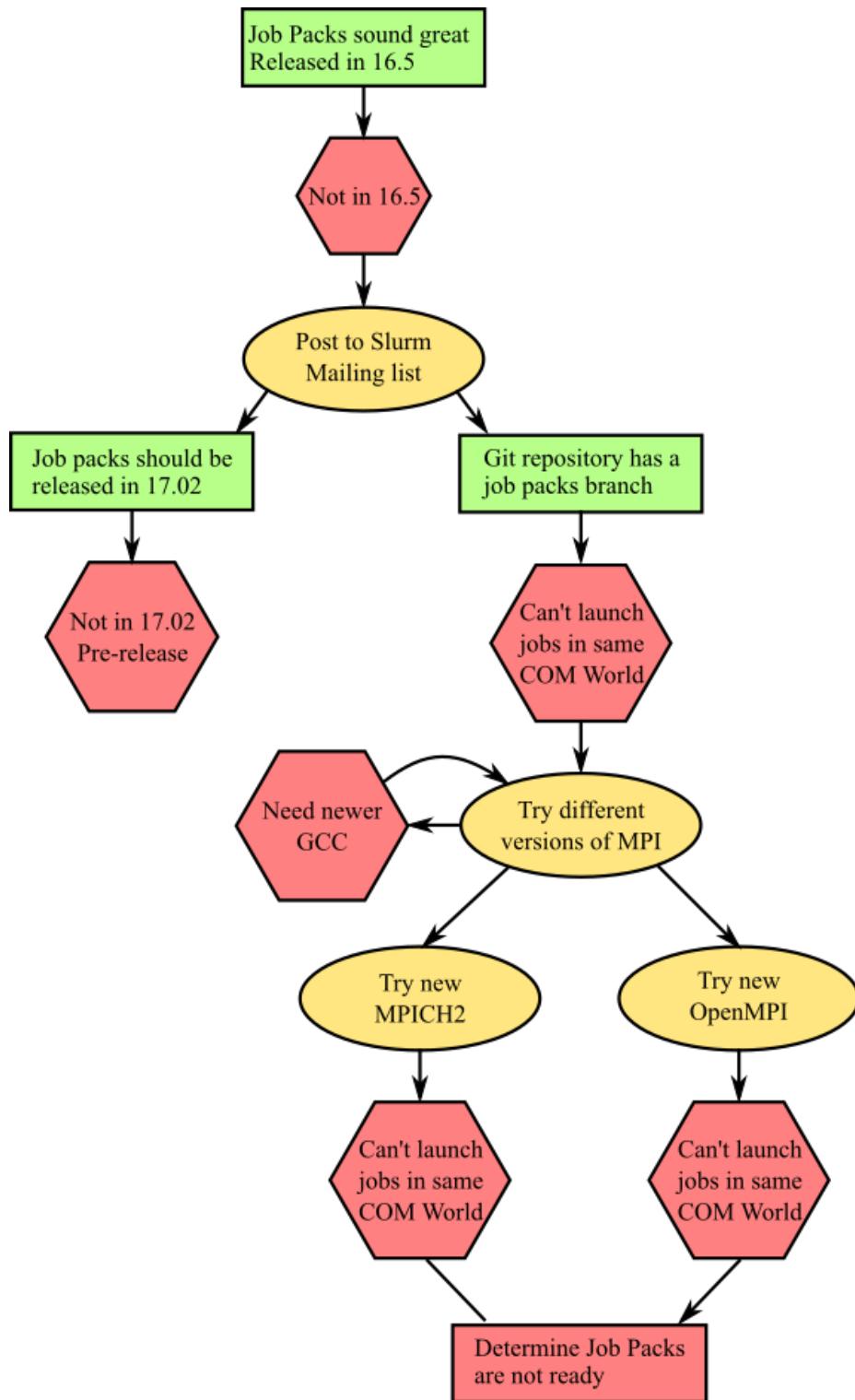


Figure 12: Overview of the Job Pack Investigation

6.2.4 Slurm Accounting

Slurm can be configured to perform accounting for the cluster. This accounting allows for collection of information for every job or job step, report usage information, and enforce cluster usage limits.[1] The decision was made to setup accounting on ALICE first since ALICE was under development and BOB is functional and stable. Since MySQL and MariaDB are listed as the preferred databases for Slurm accounting, MySQL was chosen. Upon attempting to install `mysql-server` package via apt, it was discovered that the package has missing dependencies and can not be installed on the Pine64s. After searching through the packages in the apt repository, the package `mysql-server-5.6` was found. This package is a newer version of MySQL, version 5.6 over version 5.5, and it could be installed successfully on the Pine64s. However, after configuring the Slurm database service and setting up the tables in the MySQL database, the Slurm database service gave a fatal error saying that it could not “initialize accounting_storage/mysql accoiunting storaga plugin.” Slurm determines which plugins to include when the configure script is run before building Slurm. The configure script did not pick up on MySQL since the development tools and libraries for MySQL were not installed. The apt repository has `libmysqlclient-dev` and `libmysqld-dev`; though, both of these packages are for MySQL 5.5 and similarly had broken dependencies preventing their installation. There are currently no development library packages in the Pine64 apt repository for MySQL 5.6. So in order to proceed, MySQL will have to be installed from source on the Pine64 or a Raspberry Pi node will have to run the accounting database. The Raspberry Pis can install `mysql-server`, `libmysqlclient-dev`, and `libmysqld-dev` from the apt repository without any problems. The Slurm configuration files can be found on the Bitbucket page[2] and the setup of MySQL tables can be found in **Appendix A.2.4**.

Accounting on BOB has not been implemented due to time constraints determined by the end of the semester and the reallocation of time used to have ALICE in a functioning condition.

6.3 Standard Packages

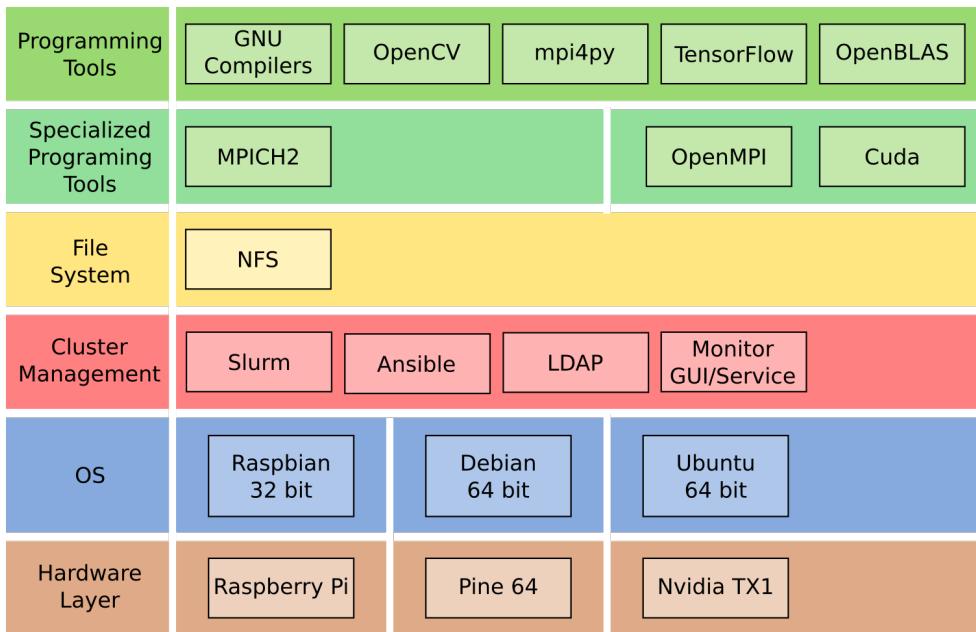


Figure 13: Software Stack

6.3.1 Pine64

- gcc 4.9, 6.2
- MPICH 3.2
- OpenBLAS
- Slurm 16.05.6
- Ansible 2.2.0.0

6.3.2 Nvidia Jetson TX1

- gcc 5.4
- CUDA 8.0
 - Nvidia CUDA compiler (nvcc)
 - Nvidia Profiler (nvprof)
 - CUDA optimized BLAS (cuBLAS)
 - CUDA accelerated deep learning library (cuDNN)

- OpenMPI 2.0.1
- OpenACC
- OpenBLAS
- Slurm 16.05.6

6.4 Frameworks and Tools Development

6.4.1 Apache Spark on Raspberry Pi

Apache Spark is being used as a replacement for Hadoop, which was unable to run on more than 1 node on BOB. Like Hadoop, Spark is an open source large data computing framework; however, Spark has many differences.

6.4.2 Octave on Raspberry Pi

GNU Octave is a language and software package that is an open source replacement for Matlab. Octave is an interpreted language for numerical calculations and experimentation. Similar to Matlab, it is also capable of producing graphics for data visualization and has its own GUI. Octave was requested to be installed by Dr. Materassi of The University of Tennessee so that the previously developed BOB Monte Carlo application could take advantage of running Octave files in addition to C/C++ and Python files. Because Octave was developed by GNU, there exists a version for Raspbian, a Debian based operating system, which is the current OS for BOB.

6.4.3 CUDA on Nvidia Jetson TX1

Nvidia CUDA is an API for general compute on Nvidia GPUs. CUDA offers a lightweight threading model which enables highly parallel applications to utilize the parallelism of GPUs in order to accelerate computation. On the Nvidia Jetson TX1 cluster, CUDA 8.0, the latest version of the API, is installed and configured. Each node includes the full CUDA toolkit including the CUDA compiler (nvcc), CUDA profiler (nvprof), cuBLAS, cuDNN, and nvGRAPH.

6.4.4 Monitoring

In order to easily monitor the status of the cluster, a monitor GUI was created to display information about the current status of each node. 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 and 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. **Figure 14** shows this first page. 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. **Figure 15** shows the heat map page.

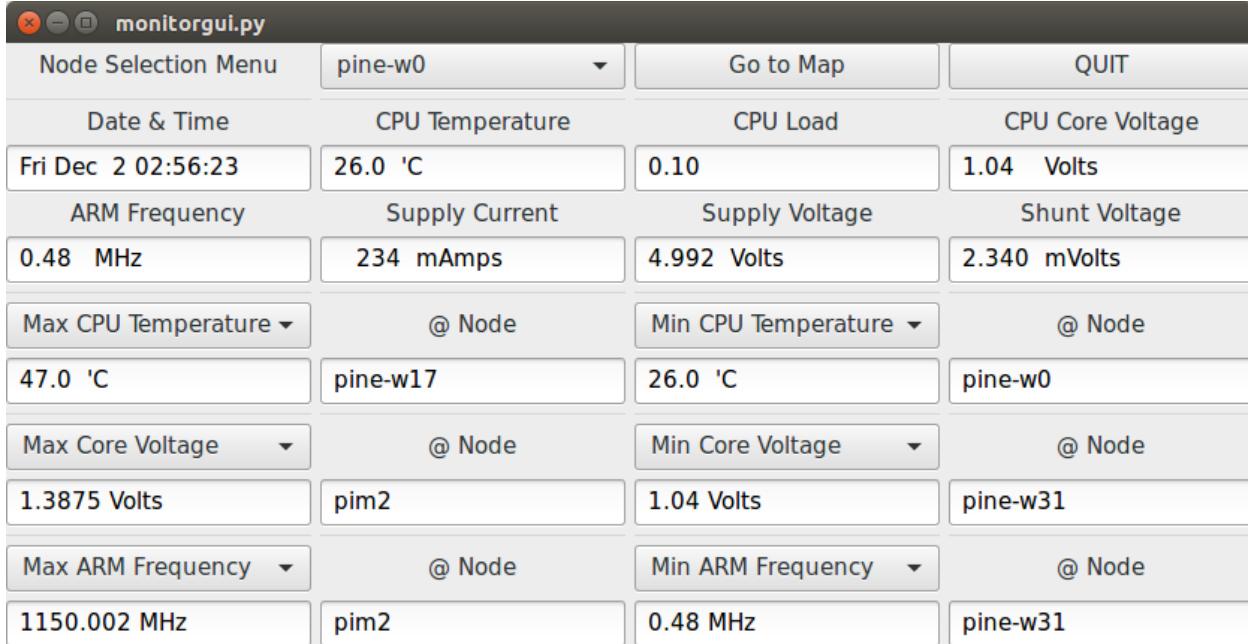


Figure 14: Main Monitor Page

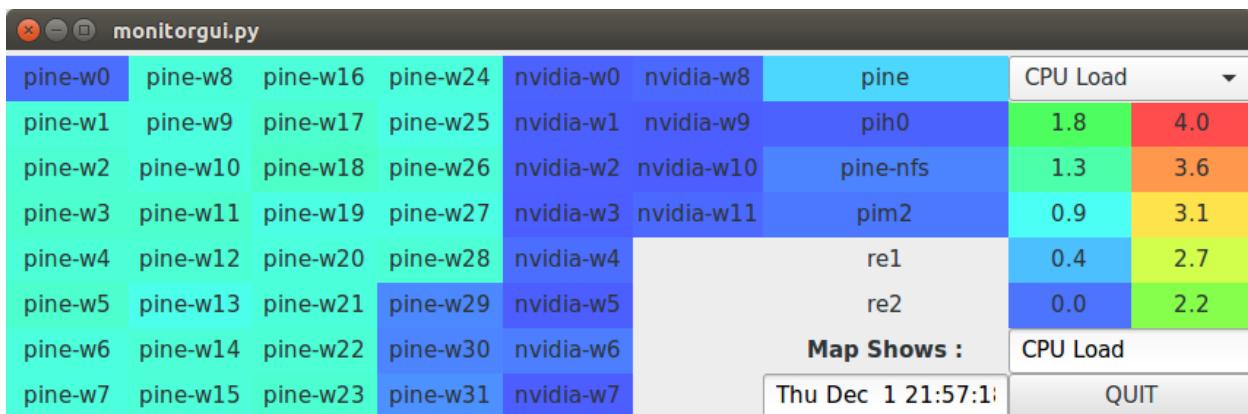


Figure 15: Heat Map Page

6.5 Applications Development

6.5.1 Facial Recognition

In utilizing the resources available within modern HPC systems, many applications can experience performance increases otherwise unattainable by a conventional laptop or desktop computer. One such application is facial recognition. By taking advantage of a HPC's interconnected nodes, time-consuming tasks such as image processing and I/O can be run concurrently. Theoretically, by being able to parallelize the processing of images, the notion of facial prediction can be sped up tremendously. Therefore, the ultimate goal is to successfully implement efficient parallelization for the eventual facial prediction to take place; however, to accomplish this, the development must go through multiple phases. Specifically, these phases pertain to utilizing a proper facial recognition software, establishing facial recognition, arranging the image data for facial prediction, and then spreading the image processing across the entire cluster to achieve full parallelization.

Of the plethora of facial recognition software available for Python, OpenCV was selected due to its comparative efficiency, thorough documentation, and available online forum support. In the concluding stages of installing OpenCV from source, the user can specify the number of cores to allow the OpenCV to run on. For BOB and ALICE, each node is either a Raspberry Pi 3 or a Pine64; therefore, each node has a potential of four cores to be specified. This effectively ensures initial parallelization, as all cores across each node contribute to the processing of images.

```
cascade = "cascades/haarcascades/haarcascade_frontalface_default.xml"
c = cv2.CascadeClassifier(cascade)

def open_images(im):
    ri = cv2.imread(im)
    gi = cv2.cvtColor(ri, cv2.COLOR_BGR2GRAY)
    return gi

def detect_faces(im):
    gi = open_images(im)
    flag = cv2.CASCADE_SCALE_IMAGE
    faces = c.detectMultiScale(gi, scaleFactor=1.1, minNeighbors=10,
        minSize=(100, 100), maxSize=(150, 150), flags=flag)
    print "Number of detected faces: {}".format(len(faces))
    return faces

if __name__ == "__main__":
    images = ["image.jpg"]
    im = Pool(4).map(detect_faces, images)
```

Figure 16: Simple example of facial detection with OpenCV.

The initial objective was simply be able to detect faces in images. As can be viewed in **Figure 16**, the process begins with accessing the function *detect_faces*. As can initially be seen, data parallelism is already performed by making use of *Pool(4)* from Python’s Multiprocessing module. Upon taking in larger numbers of images, this function parallelizes the performing of facial detection across four worker processes. Moving into the *detect_faces* function, the instruction *detectMultiScale* is encountered. By utilizing *haarcascades_frontalface_default.xml* from above, *detectMultiScale* can process each input image with facial patterns explicitly defined in the cascade. By specifying size thresholds for the algorithm, only faces that fall within a min size of 100-by-100 pixels and max size of 150-by-150 are considered valid faces. This claims particular importance in relation to facial prediction, as will be further expounded upon in the following section.

```
def detect_faces(im):
    gi = open_images(im)
    flag = cv2.CASCADE_SCALE_IMAGE
    faces = c.detectMultiScale(gi, scaleFactor=1.1, minNeighbors=10,
        minSize=(100, 100), maxSize=(150, 150), flags=flag)
    print "Number of detected faces: {}".format(len(faces))

    for (x, y, w, h) in faces:
        if len(faces) == 1:
            sample = cv2.resize(gi[y:y+h, x:x+w], (125, 125)).
                astype(np.float32)
            label = int(os.path.basename(im).split('.')[0])
        return (sample, label)
```

Figure 17: Altered function to accommodate facial prediction.

From the **Figure 17** above, it is apparent that certain changes must be made to accommodate facial prediction. These changes correspond to how machine learning algorithms fundamentally function. Specifically, data must be broken down into two data sets: samples and labels. In the code snippet below, once a face is detected, the sample data and label data are extrapolated. By calling *cv2.resize(gi[y:y+h, x:x+w], (125, 125)).astype(np.float32)*, all of the samples trained on by the machine-learning algorithms are of the same length. Ideally, all images taken would be the same pixel size, resulting in equivalent image vector lengths; however, this is not possible. As aforementioned, by specifying concise min/max pixel thresholds and resizing the vectors to fall perfectly in between those ranges, the potential for tampering image resolutions is mitigated. Once the samples are resized, they must be assigned a label. Because the machine-learning only accept integer labels, each label was simply determined by using the image’s name. For example, the sample vectors attained from image “128.jpg” would simply be assigned the label “128”. Upon returning the sample and label information, the *map* function, from **Figure 16**, simply appends the tuple into a massive list containing all associated samples and labels for each corresponding image.

However, before the data can be input into the algorithms for training, further modifications

must be made to the labels. Because many of these pictures correspond to the same individual, the algorithms must be aware of this. By reading in a text file indicating which images correspond to which individuals, the labels can be modified. **Figure 18** is an example text file read into the application. For example, in the second line, the application sees that images *80.jpg*, *81.jpg*, *82.jpg*, and *83.jpg* correspond to the same individual, namely Greg. Therefore, upon encountering a label within this image range, the proper label of “1” is assigned. By specifying which labels correspond to which image vectors, the machine-learning algorithms will then be able to extrapolate common characteristics present within each individual, and ultimately be able to predict on prospective images.

```
0, Elias, 39, 40, 41, 45, 46, 47, 126, 127
1, Greg, 80, 81, 82, 83
2, Aaron, 84, 85, 87, 88, 89, 90, 91, 92
3, Jordan, 93, 94, 95, 96, 97, 98
4, Caleb, 99, 100, 101, 102, 103, 104
5, Dean, 105, 107, 108, 111
6, Parker, 112, 113
7, Kelley, 120, 121, 122, 123, 124, 125
```

Figure 18: Text file read in for modifying machine-learning labels.

The final step in realizing facial prediction across the cluster is to implement parallelization. Of the many prospective manners of parallelizing the I/O of the image processing, two strategies were explored: dynamic process management and collective communication. For dynamic process management, once the master process reads in the list of images to process, it sends out the entire glob of data to each worker process. As each worker performs processing on the images, it will update the master process, and reduce the image lists across all of the workers. This results in heavy reliance on interconnect speeds among the worker nodes. For collective communication, the list of images is sliced into equal parts and scattered across the nodes. This results in each worker performing image processing on its own small set of images. Once each process completes its list of images, they are gathered back by the master process. This method relies heavily on the RAM available for the master process. **Figure 19** displaying the interaction differences between the two parallelization methods.

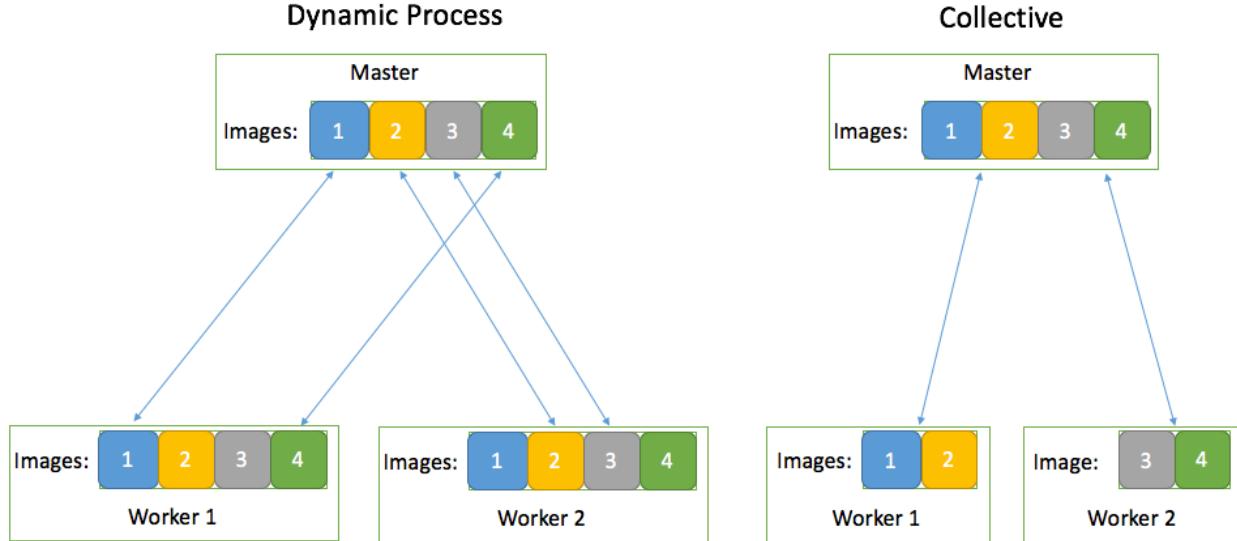


Figure 19: Dynamic process management versus collective communication parallelization.

After experimentation with both methods, the collective communication technique was ultimately selected. In comparing the two, it consistently achieved a higher overall performance speedup. This can be attributed to the interconnect speeds across the nodes for both clusters. Because they operate in the realm of only 100 Mbps, the dynamic process management is not experiencing optimal performance. Because this method requires heavy communication once each image is processed, it experiences weaker performances as the interconnect speeds are simply not efficient enough. Therefore, the collective communication is the superior method, as it limits the use of cross-node communication to the initial scattering and ultimate gathering. However, as pointed out previously, there is a drawback to this approach. Once the gathering of images takes place, the master process is subjected to an influx of processed images; therefore, depending on the RAM available for the process, there may result in a limited image count that can be processed.

6.5.2 Weather Forecasting

Parallel weather forecasting was implemented using The Weather Research and Forecasting (WRF) Model[15], an open-source weather prediction system developed by National Center for Atmospheric Research (NCAR)[14], the National Oceanic and Atmospheric Administration (NCEP), and several other government and academic institutions. WRF provides full support for parallel processing using MPICH, making it an excellent candidate for testing the scalability of both BOB (The Big Orange Bramble) and the new cluster.

The program is able to be parallelized by dividing the geographic region which is to be modeled into a grid. A collection of connected grid cells are then assigned to each parallel process. At a regular interval, each grid cell is updated concurrently by the responsible worker node based on the current state of its neighbors. Each worker node then updates the head node with the new state information for each grid cell. The head node then provides each

worker node with the updated state information of the necessary neighboring cells, and the process is repeated.

WRF scaling was tested on BOB using the Hurricane Katrina test case provided by the University Corporation for Academic Research. Scaling results will be discussed in detail in the Testing Results section. Because both BOB and the new cluster utilize worker nodes leveraging ARM processors, several tweaks to different configuration files were necessary to successfully install both WRF and all of its dependencies. Specific configuration settings will be detailed in the build guide at the end of this report.

6.5.3 Gillespie Algorithm

The Gillespie algorithm is one broadly used in biology, chemistry, and even economics for stochastic simulations. In the 1970s, Dan Gillespie and Joseph Doob created the Gillespie algorithm to simulate chemical systems under the limited computational power of the time. The algorithm's innovation lies within the discrete and stochastic method of simulation utilizing few reactants without having to iterate in real time due to its Monte Carlo steps. The Gillespie model ultimately creates an accurate model of a solution for the specified master equation set.

The Gillespie algorithm is implemented in several steps, which are elaborated below:

1. Initialization: The concentration of each species is recorded as well as the rates of the reaction equations.
2. Propensities: The likeliness that each reaction could occur is calculated.
 - A propensity is calculated by $\mu_i = R_i \sum_{j \in S} C_j$ where i is a reaction equation, C_j is the concentration of a reactant species (on left side of the arrow) in the reaction i , and R_i is the rate of the reaction i .
 - If the sum of propensities ever reaches 0, no more reactions can occur in the system.
3. Monte Carlo - Time: The next time step is chosen using $\Delta t = \mu_{total} \ln k$ where $k \in (0, 1)$ and $\mu_{total} = \sum \mu_i$
4. Monte Carlo - Reaction: A reaction is chosen randomly, but weighted by propensities.
5. Update: Concentrations are updated based on which reaction is chosen in previous step, and the iteration repeats at step 2.

It should be noted that depending on the number of species and molecules, some simulations can become computationally expensive. The idea for the Gillespie expansion of the original BOB Monte Carlo application is not necessarily to diminish the computational expense through parallelization, but to run multiple simulations at the same time. There exists a Python package which implements Gillespie algorithms called StochPy, however, the BOB Gillespie application is intended to be modular according to user input as well as to functionality specified by user. The ability to optimize code is also important particularly

for longer simulations, thus a tailored program was a better choice over the existing Python library. The current application was developed in Python, but a stretch goal to create a similar program in C++ has been added to compare run time and performance.

7 Testing Results

7.1 HPL Benchmark

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[8]. Dense linear algebra calculations are applicable to many problems and a good method to measure peak performance for a system.

7.1.1 Pine64 Cluster

The Pine64 partition of ALICE demonstrated overall disappointing performance in HPL even after very extensive tuning. For the Pine cluster, OpenBLAS is used as the linear algebra library. OpenBLAS features optimized linear algebra routines for ARMv8 AARCH64 processors.

WARNING: HPL tends to fail the residual calculation when running large matrices across 16 or more nodes. The performance values are being reported here anyway, but there is some evidence to suggest that the Pine64 boards have some level of instability causing errors in the HPL calculations.

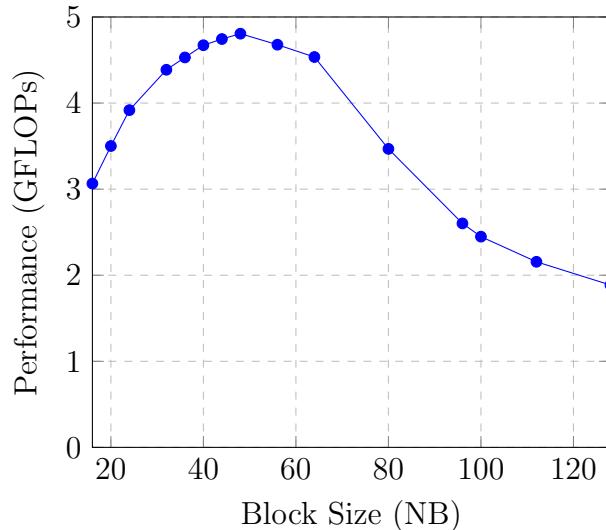


Figure 20: Pine64 Performance across varying block sizes with $N = 6000$

As demonstrated in **Figure 20**, the relationship between block size (NB) and performance achieved (GFLOP/s) drops dramatically after $NB = 64$. Using such a small block size greatly limits the performance of the cluster and negatively impacts the scaling across additional nodes. One hypothesis for why this behavior occurs is that the memory hierarchy — specifically the L2 cache — is not functioning properly. For an example of an expected block size versus performance curve, see **Figure 22** which shows the TX1’s behavior across varying NB values.

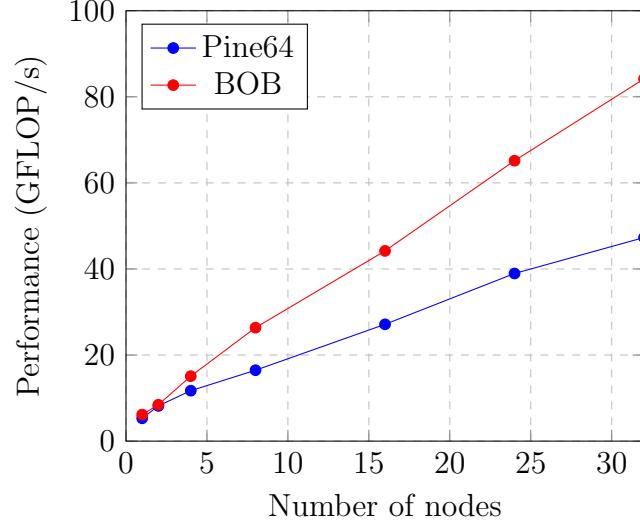


Figure 21: HPL Performance across Pine64 nodes

Figure 21 shows the final HPL results from the Pine64 nodes. Comparing with the results from BOB, the Pine cluster offers decreased scaling on top of lower single node performance. One would initially expect both systems to perform similarly up to 32 nodes, but the challenges and issues offered by the Pine64 boards result in a disappointing regression in performance relative to BOB.

# Nodes	N	NB	GFLOPs	Scaling Efficiency
1	14500	48	5.302	100%
2	21000	48	8.184	77.2%
4	29000	48	11.72	55.3%
8	41000	48	16.47	38.8%
16	58000	48	27.14	32.0%
24	71000	48	38.95	30.6%
32	79000	48	47.31	27.89%

Table 2: HPL Performance for Pine64 Nodes

7.1.2 Nvidia TX1 Cluster

For the TX1 cluster, performance was a distinct improvement in comparison to BOB even when only looking at CPU results. Due to the more complex and higher clock rate Cortex A57 cores in the TX1, HPL performance is much higher than with BOB or the Pines. On the TX1 nodes, OpenBLAS was used as the linear algebra library.

Unlike with the Pine64 boards, the TX1 boards demonstrated the expected performance curve across varying block sizes. To evaluate appropriate block sizes, a range of NB values from 16 to 256 were run on a matrix of $N = 6000$. **Figure 22** shows the results of this testing.

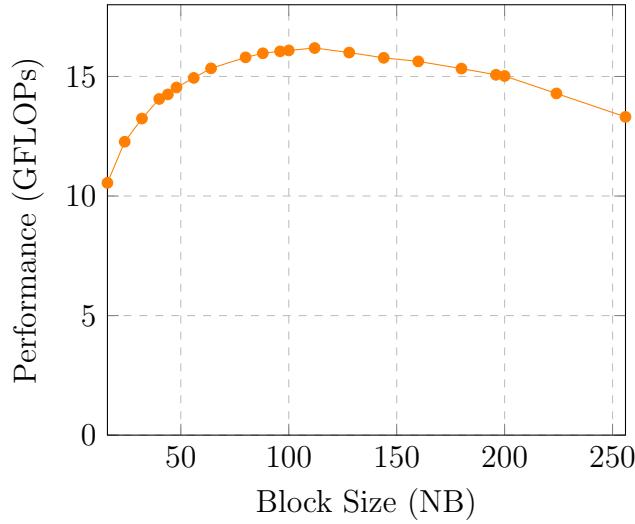


Figure 22: TX1 Performance across varying block sizes

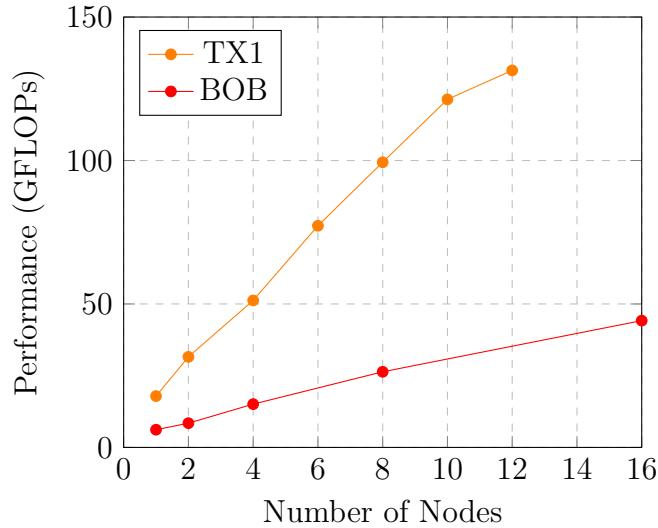


Figure 23: HPL Performance across TX1 nodes

As demonstrated in **Figure 23**, even just 12 TX1 nodes are able to achieve solid performance. At 12 nodes, it offers 131.4 GFLOPs which is almost identical to 56 Pi3 nodes in BOB which can achieve 133.8 GFLOPs.

# Nodes	N	NB	GFLOPs	Scaling Efficiency
1	18000	128	17.90	100%
2	25000	128	31.56	88.2%
4	36000	128	51.23	71.6%
6	44000	128	77.25	71.9%
8	51000	128	99.39	69.4%
10	57000	128	121.3	67.8%
12	62000	128	131.4	61.2%

Table 3: HPL Performance for TX1 Nodes

Table 3 shows a table summary of the HPL results achieved on the TX1 nodes.

It should be noted that HPL can be modified to take advantage of Nvidia GPUs with CUDA. Preliminary testing with two different modified HPL variants did not demonstrate any speedup and often were actually slower. The Nvidia TX1’s Maxwell GPU is optimized for FP16 and FP32 calculations, and running FP64 which is required for HPL incurs a high penalty. Optimization of a single precision (FP32) HPL for the TX1 would be a useful future task. However, this was not able to be accomplished during the duration of the semester.

7.2 HPCG Benchmark

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[7]. 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.

For the HPCG testing on BOB and ALICE, the reference implementation is used. In the future, one possible task could be attempting to optimize the HPCG algorithms for distributed ARM clusters.

7.2.1 Pine64 Cluster

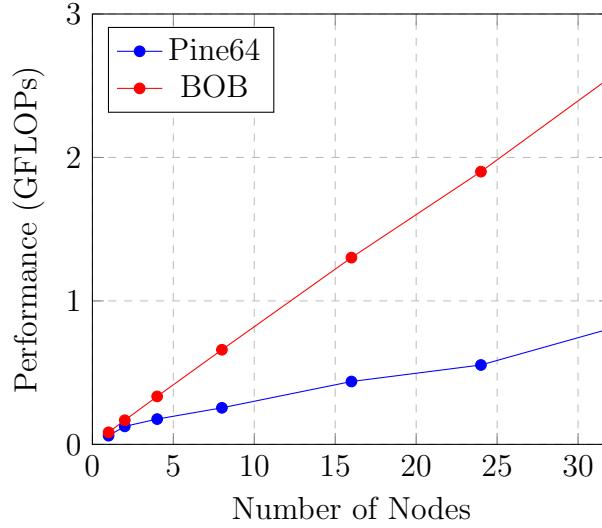


Figure 24: HPCG Performance across Pine nodes

# Nodes	GFLOPs	Scaling Efficiency
1	0.060712	100%
2	0.125465	103.3%
4	0.176229	72.6%
8	0.254834	52.5%
16	0.438145	45.1%
24	0.553385	40.0%
32	0.80700	41.5%

Table 4: HPCG Performance for Pine Nodes

As seen in **Table 4** and **Figure 24**, the performance of the Pine cluster in HPCG is again a less than ideal showing. In the comparison to BOB, the Pine64 nodes demonstrate close to 30% less performance at a single node with the gap widening as the node count increases. Unlike BOB which scaled above 93% across all HPCG tests, the Pine nodes show quite poor scaling especially for an application like HPCG which generally scales well.

7.2.2 Nvidia TX1 Cluster

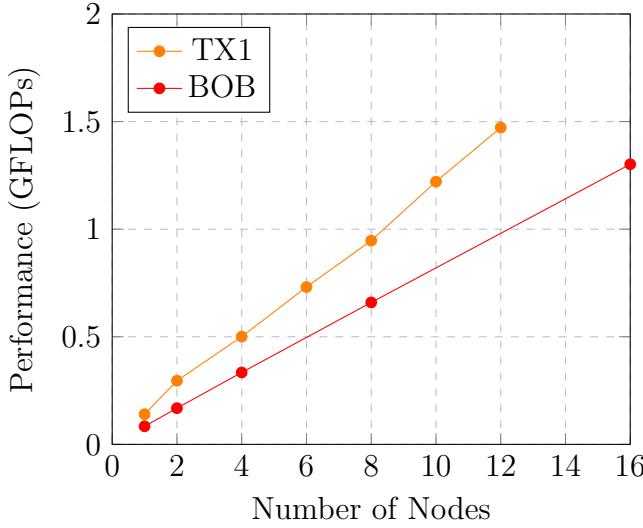


Figure 25: HPCG Performance across TX1 nodes

Looking at the BOB comparison in **Figure 25**, the TX1 nodes continue to significantly outperform BOB. However, the margin between the two systems is much closer here than in HPL. This is not unexpected though. Because the reference implementation of HPCG is memory bound, the architectural advantages of the Tegra X1’s CPU are likely of minimal usefulness. Because HPCG attempts to capture a lower bound on real world application performance, this result is reasonable and acceptable.

# Nodes	GFLOPs	Scaling Efficiency
1	0.14021	100%
2	0.295739	105.5%
4	0.500745	89.3%
6	0.731083	86.9%
8	0.946748	84.4%
10	1.22104	87.1%
12	1.47204	87.5%

Table 5: HPCG Performance for TX1 Nodes

Table 4 shows the table summarizing the HPCG tests on the TX1 nodes.

7.3 Facial Recognition

As the primary focus of this application was to achieve efficient image processing parallelization across BOB and ALICE, less emphasis was placed on fine-tuning the machine learning for

accurate facial prediction. Therefore, of the implemented algorithms, only k-nearest neighbors was the most heavily experimented on. K-nearest neighbors, or KNN for short, is a supervised, classification machine-learning algorithm. By specifying the number of neighbors to consider, it selects the neighbors based on their distances from one another. Essentially, if two images share similar characteristics with one another, then their relative distances will be short. From the aforementioned text file of individuals and their associated images, various image counts were tested to expose potential correlations. In **Figure 26** below, the prediction accuracy corresponding to each image count can be viewed. Interestingly enough, possible correlations can be developed. As the image count per individual decreases, the prediction accuracy decrease as well. This can be attributed to the machine-learning algorithm simply not having enough image samples to compare against once a prediction is to be made. By maintaining a per-individual image count of above 8, KNN is capable of performing very accurate facial prediction.

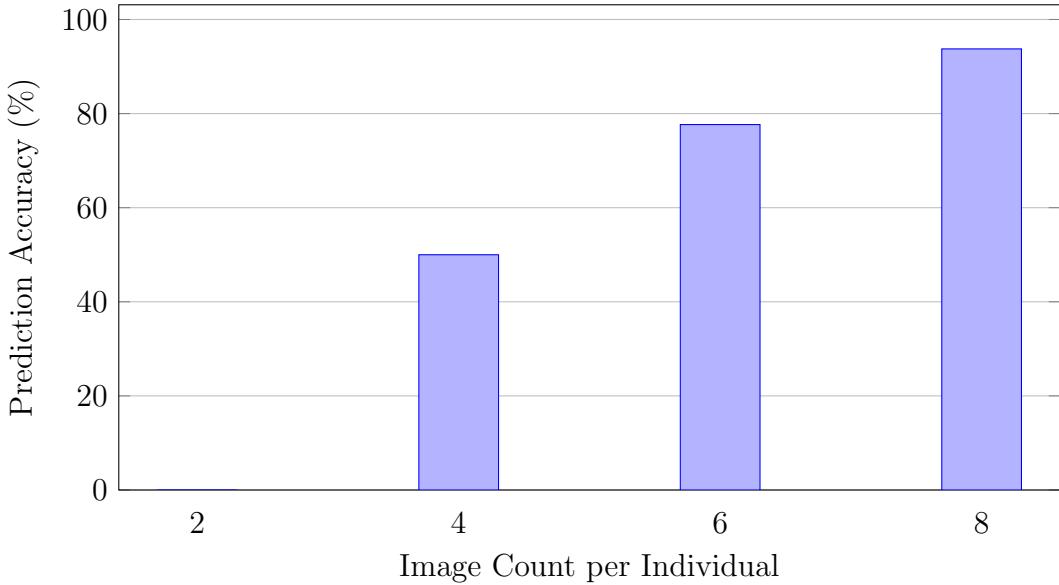


Figure 26: KNN facial prediction results.

As previously noted, the technique chosen for parallelization across the nodes was collective communication. Because both clusters suffer from inefficient interconnect among the nodes, the processing of images would have to take this into account; therefore, the images are initially scattered across the nodes and then gathered once processing completes. The first cluster tested on was BOB. **Figure 27** below details the performance results for BOB. As can be seen in the legend, BOB is not capable of processing over 320 images. This is due to the 1 GB RAM drawback experienced upon gathering the processed images.

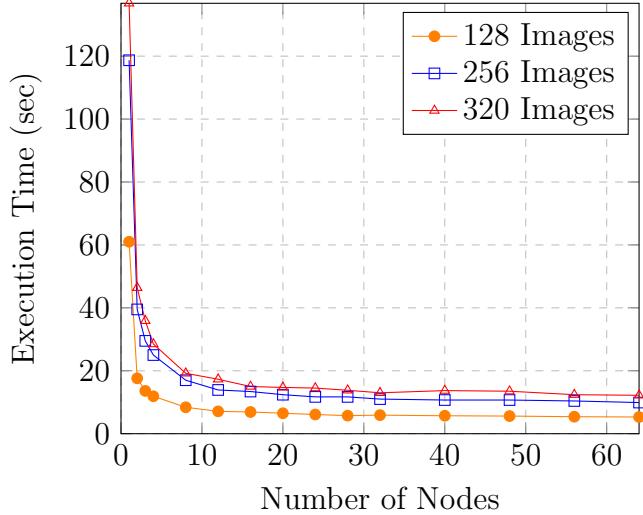


Figure 27: Facial Recognition parallelization across BOB.

As can be seen from **Figure 27** above, BOB successfully achieves parallelization across its 64 nodes. Furthermore, BOB experiences noticeable speedups until it reaches 16 nodes. At this point the speedups, although gradual, begin to plateau. **Table 6** displays the relative speedups experienced by each image count. It can be concluded that, across its 64 nodes, BOB experiences speedups in the range of 11 to 12.

Image Count	Speedup
128	11.5
256	12
320	11.2

Table 6: Relative image count speedups experienced on BOB.

Just as for BOB, collective communication was deployed on ALICE. As can be seen from **Figure 28** below, ALICE was capable of processing larger image counts than BOB. This is due to the fact that ALICE sports 2 GB RAM per node, in comparison to BOB's 1 GB. Therefore, as can be deduced from the same figure, ALICE's image count caps at 448 images. Just as for BOB, ALICE also experiences a plateau effect near 16 nodes; however, just as with BOB, ALICE continues to gradually scale across all of its nodes.

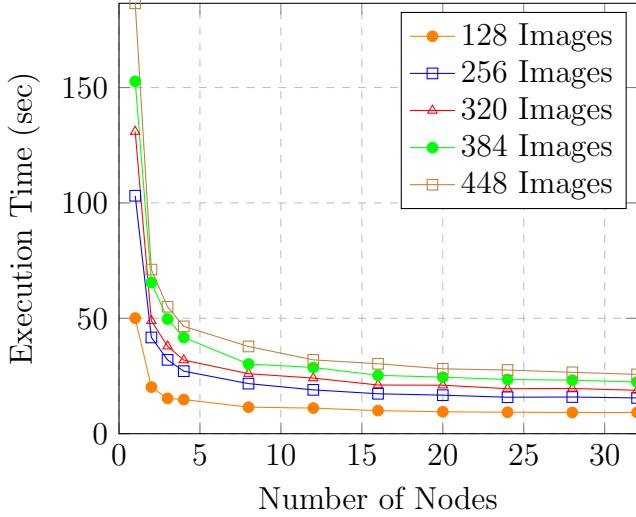


Figure 28: Facial Recognition parallelization across ALICE.

Image Count	Speedup
128	5.4
256	6.7
320	7
384	6.8
448	7.3

Table 7: Relative image count speedups experienced on ALICE.

From the tabulated results of the relative speedups across ALICE, in **Table 7**, it can be seen that ALICE does not achieve the speedups experienced by BOB. Whereas BOB experienced speedups of 12, ALICE only attained speedups near 7. This can be further noticed in **Figures 27** and **28** as they scale across the nodes. Although ALICE exhibits stronger single node processing performance, BOB emerges superior once the processing of images is extended across its worker nodes.

7.4 WRF

Testing of WRF performance on BOB was conducted using the Hurricane Katrina test data made available by the University Corporation for Academic Research[14]. Several pre-processing steps are necessary before the test model is ready to be run across the cluster. However, the data pre-processing is a serial procedure that cannot be processed by multiple nodes simultaneously. Therefore, only the actual running of the model by the WRF executable was measured to determine the level of speed-up provided by running the process across a greater number of worker nodes.

The WRF settings input file was edited as suggested by UCAR to provide respectable performance numbers while still operating on the model at a level of granularity such that the program provides useful and accurate results. Using the suggested input numbers, the test case was set to be modeled over a period of 12 hours at intervals of 3 minutes.

The initial intent of testing was to begin by running the model on a single node using 4 parallel processes and doubling the number of worker nodes for each subsequent run. This worked as intended up to 32 worker nodes, and expected results were obtained. Specifically, a diminishing return in speedup was observed as the number of worker nodes was increased, expected behavior based on Amdahl's Law, which reminds us that the execution time of a program is limited by the portion of the program which cannot be parallelized. However, less unanticipated behavior was observed when attempting to run the program across 64 worker nodes. Given Amdahl's Law and the poor return we observed when doubling the number of worker nodes from 16 to 32, we expected a similar run time when moving from 32 worker nodes to 64. Instead, the attempted run using 64 worker nodes never halted.

Our initial assumption after observing the failure of a 64 node task to complete was that the cluster was bottlenecking on the gigabit connection between the two switches which each contain 32 worker nodes. To confirm this, the model was run on 40 worker nodes, utilizing all 32 from the first switch and 8 from the second switch. This resulted in an increased run time versus the 32 node run, seeming to confirm our suspicions. Subsequent runs of 48 and 56 worker nodes returned even greater run times, further confirming our switch bottleneck hypothesis. However, more fine-grained runs at intervals of 4 nodes were then run, and a run of 36 nodes (32 on the first switch and 4 on the second) returned a (slightly) better execution time than the 32 node run. This result suggested that the increased execution time was due not to a bottleneck between the two switches but instead at the head node, which is limited much more with only a 100 mb Ethernet connection available to communicate with all worker nodes across the cluster. Regardless of the specific reason for the bottleneck and the inability of the program to complete execution when using all 64 worker nodes, the scaling results confirm that using this number of nodes would be an unnecessary use of resources, as there is little speedup even when increasing the number of worker nodes from 16 to 32 at 4 processes per node. Again, this is not to say that WRF scales poorly. In fact, these numbers are exactly what we would expect from a program for which the majority of its execution can be parallelized. Exact results are shown in **Figure 29** below. The blue portion of the line graph indicates that execution time was decreased from one run to the subsequent run using a greater number of worker nodes. The red portion of the line graph indicates that the execution time of the program inversely scaled as more worker nodes were used, most likely due to the bottleneck at the cluster's head node.

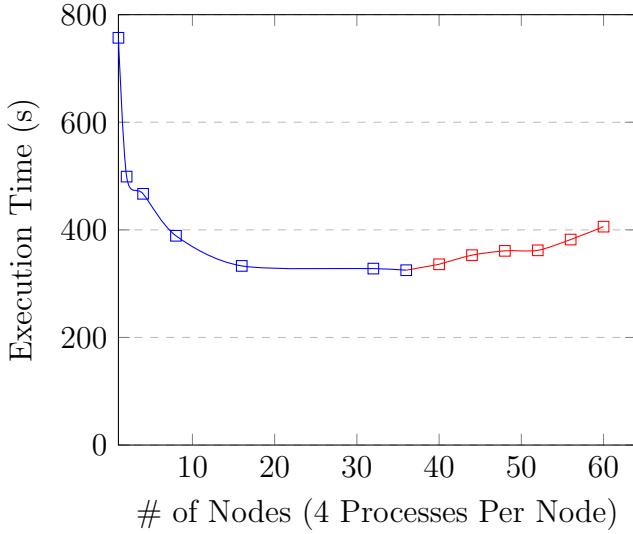


Figure 29: Hurricane Katrina Test Results

7.5 FDS

Fire Dynamics Simulator (FDS) is a computational fluid dynamics (CFD) model of fire-driven fluid flow. It is a simulation program that solves numerically a large eddy simulation form of the Navier–Stokes equations appropriate for low-speed, thermally-driven flow, with an emphasis on smoke and heat transport from fires, to describe the evolution of fire. A simple FDS file which models a room with a fire and an HVAC system was used originally for testing scalability across the BOB 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, and 32 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 BOB. 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.

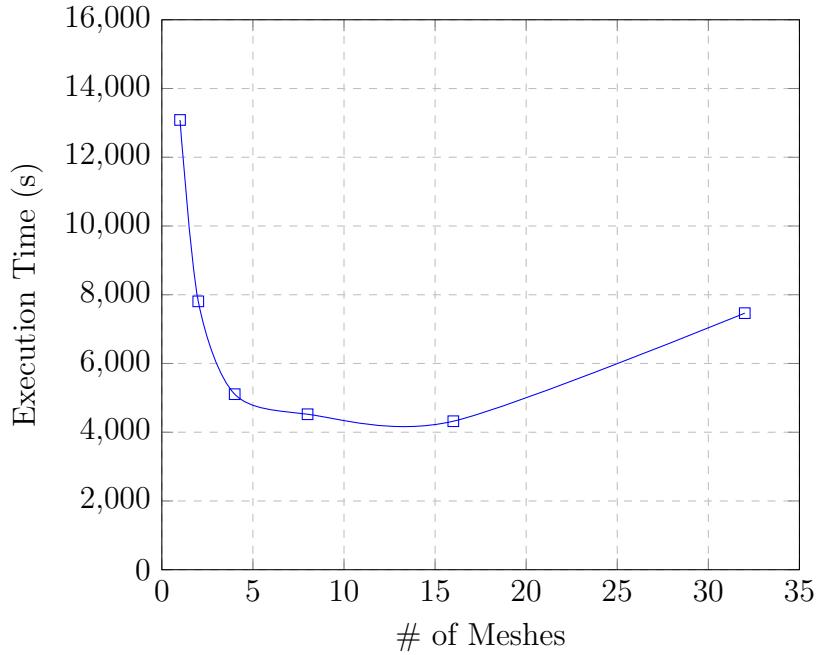


Figure 30: Graphical results of tests dividing a simple room into multiple meshes on the Raspberry Pi cluster.

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 8: Tabular results of tests dividing a simple room into multiple meshes on BOB.

This same test procedure was used on ALICE to test relative performance to BOB. Although one would expect better or at least similar results on ALICE, the measured results were in fact very different than the performance given by BOB. The following chart and table reflect the performance given by ALICE on the same test cases that were previously run on BOB.

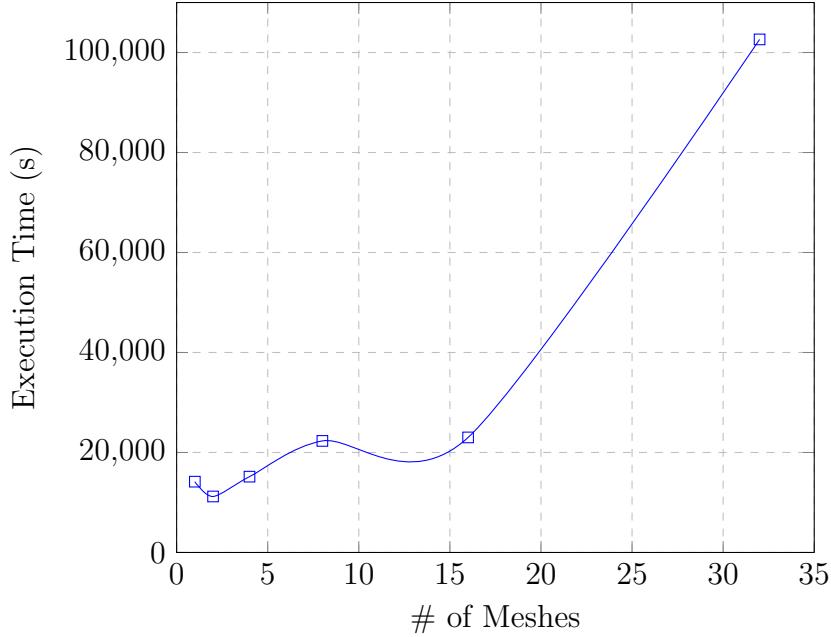


Figure 31: Graphical results of tests dividing a simple room into multiple meshes on ALICE.

Number of Meshes	Total Elapsed Wall Clock Time (s)
1	14160.73
2	11197.44
4	15164.194
8	22312.124
16	23008.796
32	102622.535

Table 9: Tabular results of tests dividing a simple room into multiple meshes on ALICE.

The results appear to have no clear trend and follow no anticipated pattern of scaling. It is left purely to supposition as to what could cause this kind of erratic and irregular behavior, as the logs from FDS runs show only that records the time it took for each individual simulation time step to successfully complete its work allotment. What can be noted is that the work allotment for each time step increases later in the simulation as the fire spreads throughout the room, creating more interactions between meshes. It is during these later time steps that FDS runs slowed significantly on ALICE, suggesting that ALICE may be experiencing network communication failure between nodes, potentially due to some kind of undiscovered packet loss or other network anomaly.

7.6 Gillespie Algorithm

Testing for the Gillespie application involved running through example test cases, which are also available in the repository as `example.txt` and `example_inf.txt`, to see if the functionality operates as intended. Similarly, these example cases were performed with `gill_simp.py` with `mc_batch.py`. **Figure 32** and **Figure 33** show converging results for single runs of `gill_simp.py`. **Figure 34** shows truncated results for `example_inf.py`, which is truncated because of non-converging (cyclical) behavior of the equations. More test runs of these examples were performed, but it seemed unnecessary to put all of them in the report as the examples are merely a proof of concept. Also, the outcome of the two example files were accurate to the results that were expected. More testing with actual master equation test sets where the solutions are known would be beneficial, but current results are promising.

A single file where the average concentration of each species at a specified interval of time was requested in order to limit the number of files created as well as minimize file size. Additions to unify output to a single file have been attempted, but results at this time show that the overhead of recreating a file repeatedly in Python too large to be beneficial. Instead, the current output file produced from `gill_simp.py` with the Monte Carlo application (`python mc_batch.py -i gill_simp.py -o output.csv -r 64`) is a concatenated series of the results for each run. It should be noted that if simulations run indefinitely, this method most likely will not be acceptable to the user.

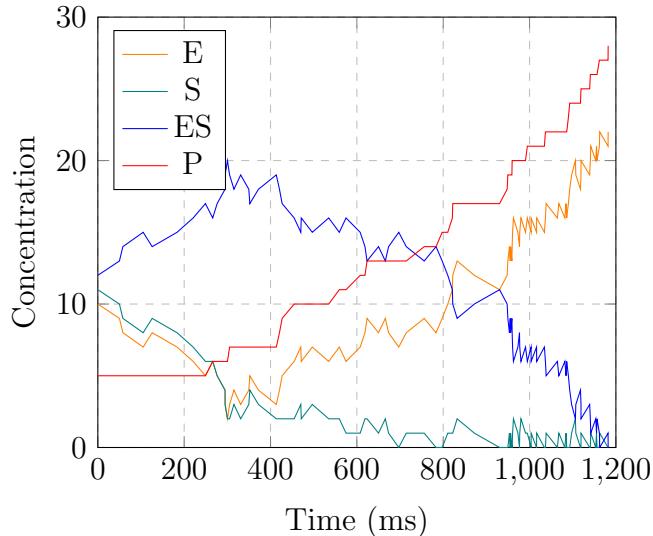


Figure 32: `example.txt` Run # 1 Results

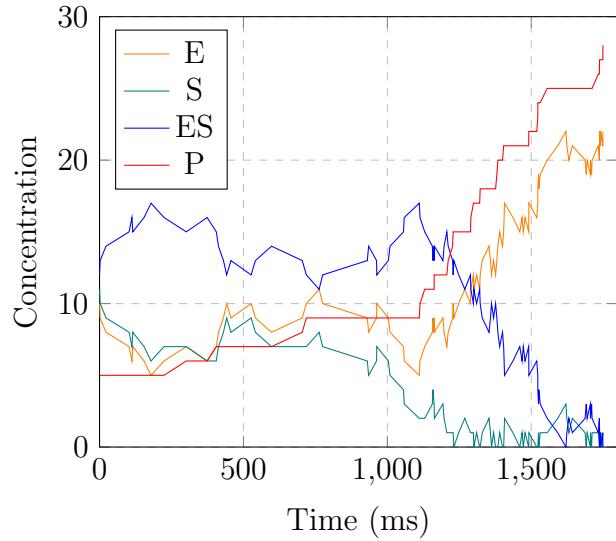


Figure 33: `example.txt` Run # 2 Results

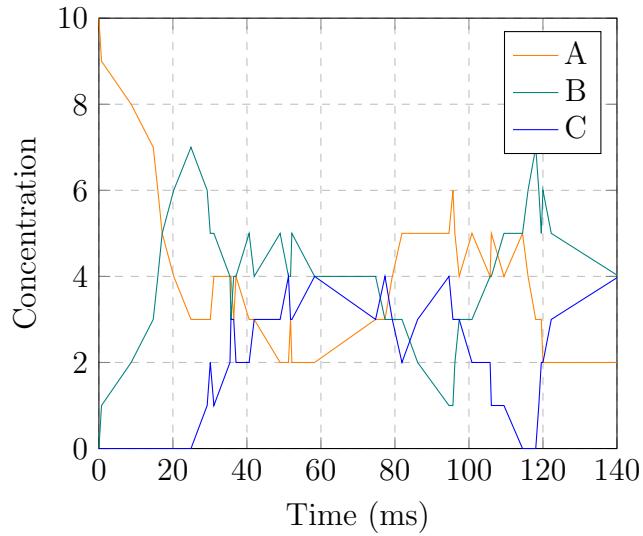


Figure 34: `example_inf.txt` Run Results - Truncated

8 Challenges and Future Considerations

8.1 Pine64

Since the Pine64 is a new device, there were many issues that we uncovered while trying to build a cluster with them. The main issues we uncovered are talked about in detail in the following sections. Although alternate methods were found to diminish the hindrance for many of the issues, some of the issues are still plaguing the system. Until all the issues are

solve, it is not recommended to build a cluster with Pine64s at the time that this report is being written.

8.1.1 Monitor Issues

When the Pine64s were first plugged into a monitor, nothing appeared on the screen. Upon trying other monitors, some of the monitors showed an image with incorrect colors and a cropped low resolution screen. With additional investigation, it was determined that the Pine64s have trouble with HDMI to DVI adapters. Although the HDMI to DVI adapter worked on one of the newer monitors, the rest of the monitors would not show anything on the screen. The pines worked better when connected directly to the monitor via HDMI without an adapter.

The color issue was resolved. The Pines output their colors in YCbCr while most monitors expect colors in RGB. After changing the color settings of the monitor to YCbCy the colors were correct. The resolution problem was also solved. The resolution is set in `/boot/uEnv.txt` by the line “`optargs=disp.screen0_output_mode=720p60`”. If this line is removed then the resolution defaults to 1080p. Most modern monitors do a better job of displaying the 1080p resolution and this fixed the cropped low resolution screen issue.

8.1.2 Rebooting Issues

The Pine64s also have trouble booting successfully when the power is cycled. If one of the Pine64 boards has power cycled multiple times, some small percentage of reboot instances will not be successful. When it fails to boot, nothing appears on the screen and there is no network activity. The log files also provide no information as to the source of this issue. If one were working with one node this issue could be a simple annoyance. When working with 34 nodes, the problem becomes much larger. In order to facilitate bringing the cluster online the restart controller was added (See [Section 6.1.4](#)). The controller will allow a selective restart of the nodes that have not successfully booted or have gone offline. The alternative is to have someone sit by the machine and continually cycle power on the nodes that are down until all the nodes are up. The power cannot be cycled on the entire cluster because the probability that all the nodes will come up is small and the nodes will just take turns being down.

8.1.3 Networking Issues

When trying to network the Pine64 boards together additional problems were discovered. The first of which is that the Pine boards get their MAC address from the `/boot/uEnv.txt` file. In order to be able to connect multiple pine boards together, the line

`ethaddr=36:c9:e3:f1:b8:05`

needs to be changed to give each board a unique MAC address.

The second problem is that the Pine64 boards are incapable of reliable communication running at 1 Gigabit Ethernet. When running at this speed, ping would show many consecutive packets being dropped. These drops prevent any service from running across the network, including SSH. Different boards seemed to have fewer drops than others, and there are instructions online for how to tune the boards to perform better at the high speeds. However, since a cluster uses many boards and needs reliable communication, the switch was configured to use 100 Mbit Ethernet on the ports that the pine are connected to. The Pines are able to run at the slower speed without dropping packets, however doing so greatly hurts the performance that can be expected from the cluster.

8.1.4 Software Difficulties

The Pine64s apt package repository is limited and is missing many of the needed packages for setting up a cluster. Other packages are listed but have broken dependencies when installation is attempted. Therefore, it is necessary to install the majority of the packages from source. This task is made more difficult given that it is unlikely that anyone else has tried to install the software before and installation difficulties are likely to occur.

8.1.5 Cache Issues

One hypothesis for the strange performance characteristics demonstrated by the Pine boards is that the L2 cache is not functioning properly or is completely disabled. This conjecture is based off of testing varying block sizes for HPL.

In **Figure 20**, it can be seen that after $NB = 64$ the performance dramatically decreases. The Allwinner A64 SoC features 32kb of L1 data cache which corresponds to a 64×64 matrix of double precision floating point values. The chip also features 512kb of L2 cache which should enable block sizes similar to what was used with BOB. However, in a test which a Raspberry Pi 3 would achieve greater than 6 GFLOPs, the Pine64 boards were recording less than half the Pi 3's performance.

8.1.6 Reliability Issues

While operating the Pine64 cluster it was discovered that the nodes are unreliable and prone to stop responding. When the headnode stops responding, the screen is frozen and it cannot be logged into via SSH. All of the pines are subject to this issue and it seems to be made worse when the system is under load. The only known way to unfreeze a node once it goes down is to force a power cycle.

Another reliability issue to note is the likelihood of application failure for applications with long execution times. With HPL, tests utilizing 16 or more nodes often failed the residual calculation which verifies the calculated solution's integrity. No root cause has been established for this behavior yet, but one hypothesis is that the frequency/voltage relationship in the firmware for the Pine64 is operating the Allwinner A64 chip in an unstable state.

Because HPL heavily utilizes the vector units, it tends to reveal any instability in a system which is why PC overclockers often use LINPACK or similar benchmarks to test stability.

8.2 Nvidia

When using the Nvidia Jetson TX1 boards, several issues and challenges appeared. When first loading the OS image onto the boards, the current Jetpack version (2.2) was only compatible with Ubuntu 14.04 and 16.04 as the host system, so team members had to load a compatible version of Ubuntu. After that, CUDA and the appropriate CUDA toolkit were not being installed properly by the Jetpack system. However, during the semester, Nvidia released Jetpack 2.3 with CUDA 8.0 which resolved all outstanding Jetpack issues.

Another challenge was determining how to best utilize the boards. After a bit of difficulty, OpenMPI 2.0.1 was finally compiled and installed with CUDA-aware functionality. However, at that point, there was not enough time to properly develop a sufficient benchmark to take advantage of the TX1's unique capabilities and unique SoC architecture.

There are several potential opportunities for future work. A modified single precision HPL should be developed which can utilize the 512 theoretical GFLOPs of FP32 performance. The cluster of 12 TX1 nodes should be able to achieve in the TFLOP range of performance on a reduced precision benchmark. HPCG can also be optimized rather than using the reference implementation.

Another application which could take advantage of this system is deep learning. TensorFlow and CAFFE are examples of deep neural network systems which can use the cuDNN library to accelerate deep learning. Using this across a cluster could further improve performance and yield some very useful power.

9 Conclusion

Given the considerable amount of complex challenges faced during this project, there is no small amount of improvements that could be performed on ALICE. It has been suggested that the Pine64s possibly be abandoned and replaced with either more Raspberry Pi 3s or Nvidia TX1s. Either of these options have proven to be worthwhile in regards to setup and performance, on top having a great deal of support through their respective company and developer communities. The performance of the Raspberry Pi 3s and Nvidia TX1s has also shown to be reliable, which is an important aspect of this cluster since it is being used for research and analysis. One of the main improvements would be to install and test the Slurm Job Packs when they are released in order to finally unify BOB and ALICE into BOB+. Even though there was a seemingly endless supply of issues which developed during this project, the experience gained by the team was invaluable. There is now a more detailed list of future improvements to the cluster, and there exists even more applications on BOB for users who are interested in either running them or looking for examples to develop their own BOB applications. BOB remains a great tool for high performance computing and parallel

computing education, and it is the hope of the team that BOB+ continues to evolve for the benefit of the students and faculty at The University of Tennessee.

References

- [1] *Accounting and Resource Limits.* SchedMD. URL: <https://slurm.schedmd.com/accounting.html> (visited on 12/07/2016).
- [2] *Big Orange Bramble Bitbucket.* University of Tennessee. URL: <https://bitbucket.org/bigorangebramble/> (visited on 12/06/2016).
- [3] Dearden, Dr. Chris. *Through the Wind and the Rain.* URL: <http://www.chrisdearden.net/> (visited on 08/25/2016).
- [4] DiCola, Tony. *Adafruit INA219.* Adafruit. URL: https://github.com/adafruit/Adafruit_INA219 (visited on 12/08/2016).
- [5] DiCola, Tony. *Adafruit Python MCP9808.* Adafruit. URL: https://github.com/adafruit/Adafruit_Python_MCP9808 (visited on 10/18/2016).
- [6] Forney, Glenn P. *Smokeview, A Tool for Visualizing Fire Dynamics Simulation Data.* 6th ed. Special Publication: 1017-1, Git Revision: Git-r21-3-gc0135a7. National Institute of Standards and Technology. Aug. 2016.
- [7] *HPCG.* URL: <http://www.hpcg-benchmark.org> (visited on 08/02/2016).
- [8] *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).
- [9] Razafinjatovo, Andry et al. *Job Packs – A New Slurm Feature For Enhanced Support of Heterogeneous Resources.* Slurm User Group. 2016. URL: https://slurm.schedmd.com/SLUG16/Job_Packs_SUG_2016.pdf (visited on 12/06/2016).
- [10] Schultz, Rod et al. *Heterogeneous Resources and MPMD (aka Job Pack).* Slurm User Group. 2015. URL: https://slurm.schedmd.com/SLUG15/Heterogeneous_Resources_and_MPMD.pdf (visited on 12/06/2016).
- [11] Simpson, Adam. *Tiny Titan.* Oak Ridge Leadership Computing Facility. URL: www.github.com/TinyTitan (visited on 07/02/2016).
- [12] Simpson, Gregory et al. “Big Orange Bramble: Supercomputer Design and Analysis”. MA thesis. The University of Tennessee, Knoxville, Aug. 2016. URL: [http://web.eecs.utk.edu/~markdean/B0B_documentation%20\(1\).pdf](http://web.eecs.utk.edu/~markdean/B0B_documentation%20(1).pdf).
- [13] *Slurm.* SchedMD. URL: <https://github.com/SchedMD/slurm> (visited on 12/06/2016).
- [14] *The University Corporation for Atmospheric Research.* URL: <http://www2.ucar.edu> (visited on 08/24/2016).
- [15] *The Weather Research and Forecasting Model.* URL: <http://www.wrf-model.org/index.php> (visited on 08/22/2016).
- [16] Williamson, Scott. *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 Daughter Card

To attach a daughter card to a node, first run the following to get I²C utilities:

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

Connect the daughter card to the Pine64 GPIO pins (NOTE: be sure to place it on the Pine64’s “Pi GPIO” pins and not the Euler Bus) and transfer the micro USB cable to the daughter card.

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 I²C 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 [16]:

```
$ 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 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.1.2 Temperature Sensor MCP9808

To attach a temperature sensor to a node, first run the following to get I²C utilities:

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

Connect the temperature sensor to the Pine64 GPIO pins (NOTE: be sure to place it on the Pine64's "Pi GPIO" pins and not the Euler Bus).

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: -- -- -- -- -- -- -- 18 -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- --
```

The I²C bus should be /dev/i2c-1 and the device should be connected at 0x18.

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 MCP9808 [5]:

```
$ cd git
$ git clone https://cwill133@bitbucket.org/bigorangebramble/
temp_sensor_mcp9808.git
```

Install the following packages:

```
$ sudo apt-get install python-dev python-smbus
```

Now enter the temp_sensor_mcp9808 directory:

```
$ cd daughter-card
```

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

```
$ python simpletest.py
```

Note: The wrapper.py program used to query the status of the daughter card on multiple nodes can be modified to also take measurements from the temperature sensor.

A.2 Software

A.2.1 Setup MySQL for usage with Slurm

1. Install the following packages:

```
$ sudo apt install my-sql-server libmysqlclient-dev libmysqld-dev
```

2. Configure MySQL for slurmdbd

```
mysql -u root -p
create database slurm_acct_db;
create user 'slurm'@'localhost';
set password for 'slurm'@'localhost';
grant usage on *.* to 'slurm'@'localhost';
grant all privileges on slurm_acct_db.* to 'slurm'@'localhost';
flush privileges;
```

3. Run to ansible playbook slurm-database to finish the setup.

A.2.2 Migrate LDAP

This section goes over the commands needed to dump the contents of the LDAP directory to a dump file, modify the dump file so that it can be used as an input file, and inputting the dump file into a new LDAP database. This process can be used to migrate a LDAP database to a new machine and this method was used when the LDAP database was migrated from BOB to the pseudo BOB head node on ALICE.

Steps to Migrate an LDAP directory.

1. Dump the contents of the LDAP directory to a dump file.

```
$ sudo slapcat > slapcat.dump
```

2. Remove all the lines that start with “entry CSN:”


```
$ egrep -v '^entry CSN:' < slapcat.dump > ldapdump
```
3. Add the modified dump files entries to the LDAP database.


```
$ sudo slapadd -l ldapdump
```

For ALICE, the home directories of all the users was change to be located on the Pine64 NFS node. The steps shown below were used to modify the home directories of all the users after the data had been migrated.

1. Dump the contents of the LDAP directory to a dump file.


```
$ sudo slapcat > slapcat.dump
```

2. Use vim to write out all the identifier lines to a file.
3. Add the following lines to the file.

```
changetype:modify
replace:homeDirectory
homeDirectory:/home
```

4. Commit changes with

```
$ ldapmodify -x -D cn=admin,dc=bob,dc=eeecs,dc=utk,dc=edu -W -f
slapcat.dump
```

A.2.3 Backup NFS Drive

The following command is used to backup the NFS network drive on BOB to an external USB hard drive. This command uses rsync and assumes you have a laptop connected directly to the BOB switch with the backup folder location /media/ayoung48/data1.

```
$ sudo rsync -avz -e "ssh" --rsync-path="sudo rsync"
pi@192.168.50.5:/mnt/nfs/ /media/ayoung48/data1
```

A.2.4 Setup MySQL Tables for Slurm Accounting

The commands below were used to create a Slurm user, setup privileges, and create tables for Slurm to use.

```
$ mysql -u root -p
$ create database slurm_acct_db;
$ create user 'slurm'@'localhost';
$ set password for 'slurm'@'localhost' = password('mypass');
$ grant usage on *.* to 'slurm'@'localhost';
$ grant all privileges on slurm_acct_db.* to 'slurm'@'localhost';
$ flush privileges;
```

Appendix B Application Install and Usage Guide

B.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 ARMv8 binary.

```
$ 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.alice
```

6. Modify the generic Makefile to match the appropriate paths and options for Debian/Ubuntu and OpenBLAS.

7. Make HPL

```
$ make arch=alice
```

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

- (a) Matrix Size (N): $N = \sqrt{\text{nodes} \times 14000^2}$ for Pine64, $N = \sqrt{\text{nodes} \times 18000^2}$ for TX1
- (b) Block Size (NB): 48-64 on Pine64, 100-200 (128 is a good choice) for TX1
- (c) Process Grid (P, Q): $P \times Q = \text{nodes}$, typically best to be as close to square as possible
- (d) NBMIN, NDIV: Combinations of 2, 4, and 8 can all be good. Going with 2 for both is usually safe.
- (e) Broadcast (BCAST): 1RingM (1)
- (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 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 hpcg-3.0.tar
```

- Create a Makefile for your architecture

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

- Edit the Makefile to have correct paths to MPI

- Create your build directory

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

- Compile HPCG

```
$ make
```

- Run HPCG with Quick Path flag

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

B.3 FDS

1. Installing FDS

Because one purpose of running FDS on ALICE is to compare its performance to BOB, it is necessary to install the 32-bit version of FDS as this was what was installed on BOB. 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_deprecated.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 = number 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.

B.4 WRF

This install guide assumes that MPICH and Ansible have already been properly installed and configured on the target cluster.

1. Clone all NetCDF contents of the Ansible repository on Bitbucket and run the NetCDF Ansible playbook to install the NetCDF dependency across all worker nodes.
2. Download WRF from the UCAR website. This guide recommends that you use WRF version 3.2.1. Newer versions of WRF have not been confirmed to successfully compile on ARM.
3. Create a new directory called “WRF” in your home directory and unpacked WRF to the newly created directory.
4. Edit the configure_new.defaults file in the unpacked WRF directory by replacing each instance of “x86_64” with “armv7l”.
5. Run the configure script and select the option for “gfortran dmpar”.
6. Edit the newly created configure.wrf file by adding the “-cpp” flag to the end of the lines that begin with “FORMAT_FIXED =” and “FORMAT_FREE =”.
7. Run the compile script.

After being successfully compiled, WRF can be tested using the Hurricane Katrina test data available at the UCAR website. Special thanks to Dr. Chris Dearden at www.chrisdearden.net[3] for specific instructions to allow WRF to successfully compile on the Raspberry Pi.

B.5 Gillespie Algorithm

1. Clone the Monte Carlo repository from BOB on Bitbucket (`BOB/app-monte-carlo`).
2. For a single simulation:
 - (a) Change the input file path, time interval, and iterations variables in the `gill_simp.py` file to suit your simulation needs. To allow the simulation to run indefinitely, set the iterations to 0.

- (b) From commandline, run `python gill_simp.py > output.csv`. If no file redirection is included, output will print within the terminal.

3. For multiple simulations:

- (a) Change the input file path, time interval, and iterations variables in the `gill_simp.py` file to suit your simulation needs. To allow the simulation to run indefinitely, set the iterations to 0.
- (b) From commandline, use the Monte Carlo application to run `gill_simpl.py` as follows:

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

Argument Descriptions:

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

The input file must be specified in a particular format. This format is as follows:

- Species and Concentration
 - Each species must be on its own line with a lowercase “s” at the beginning of the line.
 - Following the “s” flag, species variables must only be letters and must be a single character or string, such as “E” or “EST”. Note that the reaction calculations are case sensitive, thus it is important to be consistent.
 - A concentration can be added after the species variable name. If one is not specified, a random value $\in [0, 100]$ will be chosen during initialization.
- Equation and Rate
 - Each reaction equation must be on its own line with a lowercase “r” at the beginning of the line.
 - The equation itself must have no spaces.
 - Each reactant and product must be separated by a “” if there is more than one.
 - The reactants must be separated from the products by “->”, again with no spaces.
 - The rate must be added after the equation by a space.

Example input file `example.txt`:

```

s E 10
s S 11
s ES 12
s P 5
r E+S->ES 0.4
r ES->E+S 0.35
r ES->E+P 0.25

```

Within the `gill_simp.py` file, the input file and time sample interval must be specified as the default is `example.txt` and 100 (ms). A maximum numbers iterations can also be specified in order to limit infinite loops. The default number of iterations is 1000, however, if `iters=0`, there is no breakpoint unless propensities converges to 0.

Example of a single output file (`python gill_simp.py > output.csv`):

```

-,E,S,ES,P
0,10,11,12,5
100,6,7,16,5
200,4,5,18,5
400,6,7,16,5
500,5,6,17,5
700,5,4,17,7
700,6,4,16,8
800,7,4,15,9
900,5,1,17,10
1000,8,0,14,14
1100,16,2,6,20
1200,16,1,6,21
1300,18,0,4,24

```

An output file from the Monte Carlo application looks similar except with multiple run results listed within the same file, along with timestamps at the beginning and end of each batch. Several example test runs are given in the Testing section of this document.