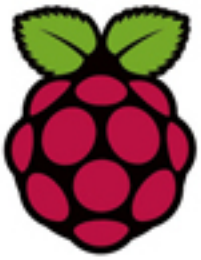


A Punnet of Berries - Documentation

TeamPi: Adrian Zielonka, Alyssa Biasi, Zach Ryan

COSC1114 - Operating Systems Principles: Friday 14:30

CONTENTS

CHAPTER

ONE

SELF ASSESSMENT

1.1 Collaboration Tools

The collaboration tools utilised for the development of the Punnet of Berries project reflect the Open Source nature of the project.

Version Control: All code and documentation relating to the Punnet of Berries project is hosted in repositories on GitHub under the *rmit-teamPi* organisation. GitHub was chosen as it allows individual team members to use either Git or SVN depending on their own preferences.

Project Management: The project and bug-tracking was managed using Redmine, an open source web-based project management tool. The team's Redmine account was hosted on HostedRedmine, a service that hosts standard Redmine projects for free.

Group Communication: Aside from weekly group meetings, our main avenues for group communication were a TeamPi Facebook group and Skype chat sessions.

1.2 Milestone 01

1.2.1 Reflection

Milestone 1 involved following a cross-compiler recipe and becoming familiar working in a command line Unix environment.

The image was build by one TeamPi member, Alyssa Biasi. Alyssa already has considerable experience working on the command line. After spending some time getting to know the recommended tool, `tmux`, she followed the instructions given in the recipe and performed the sanity check. No problems were encountered during the build.

uClibc vs. glibc: Clibc was developed without consideration for other architectures and thus is a C library targeted directly for embedded Linux. glibc was intended for higher output performance and has additional features not required on a Raspberry Pi. The omitted features allow uClibc to function with a smaller amount of memory.

Cross-Compliers: A cross-compiler is critical in the creation of executables that are able to function on architectures that differ from the platform on which the code was developed.

1.2.2 Result

OSP Check sheet 1 – Cross Compiler

Input	Expected result / output	Verified
cd $HOME/cross/cross-tools/bin ldd arm-unknown-linux-uclibcgnueabi-ar | grep `whoami`	If this returns nothing then the student did not build the compiler as this user.	✓
cd $HOME/cross/rootfs	Should return 245	✓
find . -type d | wc -l	Should return 1815 245	✓
cd $HOME/cross/images	boot the image the student created.	✓
/share/tools/bin/qemu-system-arm -M versatilepb -cpu arm1176 -m 256 -kernel kernel-qemu -initrd fs.img -append "root=/dev/ram rdinit=/bin/sh console=ttyAMA0" -nographic	Should see a "#" prompt	✓
which find	Returns /usr/bin/find	✓
find . | wc -l	Returns 2101 2100	✓
vi -v	Should fail. Returns /bin/vi: invalid option -- v BusyBox v1.21.0 (2013-07-27 16:16:25 EST) multi-call binary. Usage: vi [OPTIONS] [FILE]... Edit FILE -c CMD Initial command to run ($EXINIT also available) -R Read-only -H List available features	✓
Ctrl-A x	Exit.	✓

Team name: Team P1 Lab time: Friday 14:30-15:30 Date of demonstration: 16/08/13

Result of demonstration: Requirements fulfilled ☑ Resubmit ☐ Demonstrator name: Peter George Signed:

CHAPTER

TWO

INTRODUCTION

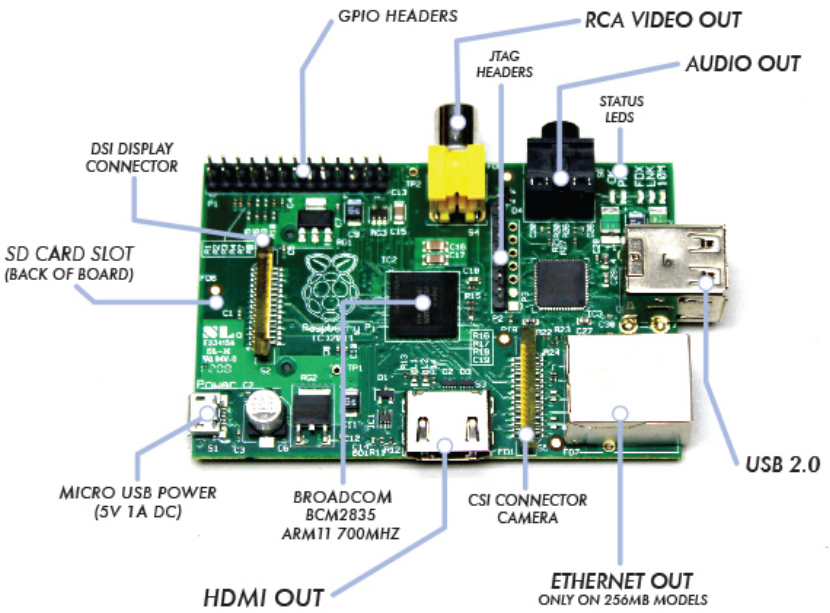
The aim of the **Punnet of Berries** project was to create a financially viable Beowulf cluster, evaluate the power of the Raspberry Pi and demonstrate scheduling concepts.

This document outlines the frameowrk on which the **Punnet of Berries** and its accompanying software were developed. System design, architecture and implementation are all covered in the subsequent sections.

2.1 The Punnet of Berries

A Beowulf cluster is a supercomputer built out of a collection of (typically) inexpensive computers. The computers are networked together by a local area network, usually Ethernet, and run a parallel processing software. The individual computers are combined together to form a single system, becoming the nodes of the supercomputer. This concept puts the power of supercomputing into the hands of small research groups and schools. It gives them the ability to access the computational power that normally only large corporations can afford.

The Raspberry Pi is a single-board, credit card sized device. It is capable of running Linux, and other light-weight operating systems, with its ARM processors.



The Raspberry Pi is a powerful, yet, low-cost "mini-computer". It was developed to educate and inspire the next generation of programmers. **TeamPi** believes that this makes the Raspberry Pi an ideal candiate to create a Beowulf cluster.

The **Punnet of Berries** is a *Raspberry Pi Beowulf cluster*.

2.2 The Berry Batch

In order for a compute cluster to function properly and fairly, its resources must be managed and monitored by a batch system scheduler. Batch systems enforce limits on the number of jobs running at one time and the resources available to them. Compute cluster users request resources by submitting jobs to the batch system, which then determines the scheduling that best utilises the resources available.

The **Berry Batch** is a custom batch system, which manages the Punnet of Berrries compute cluster.

As one of the aims of the Punnet of Berries is to demonstrate scheduling concepts, the Berry Batch implements several scheduling algorithms. The user is able to select which algorithm to use when the cluster is initialised.

Please note:
Due to unforeseen circumstances, the Berry Batch application has not been completed. Any reference to the Berry Batch is speculation based on the original design.

CHAPTER

THREE

GOALS AND OBJECTIVES

The key goals of the Punnet of Berries project were to:

- Create a Beowulf compute cluster
- Develop a basic batch system scheduler
- Demonstrate scheduling concepts
- Develop descriptive documentation so that others may recreate the Punnet of Berries project

A financially viable Beowulf cluster was created as a part of the Punnet of Berries project. The cluster is energy efficient and demonstrates the advantages of parallel computing.

During the development of the Punnet of Berries, three key design principles were kept in mind.

- **Scalability** A computer cluster is, by nature, designed to be scalable. It was important that this inherent design characteristic be adhered to in order to avoid any sort of strict node limitations.
- **Usability** The Punnet of Berries project is largely for educational purposes. As such, it was important that the cluster be easily constructed with high cohesion. TeamPi aimed to create an application interface that is readily accessible.
- **Flexibility** In order to best demonstrate scheduling concepts, the Berry Batch was created to allow customisation. This was achieved by offering the user different options for the main features of the Berry Batch.

  > Scheduling algorithm: First-Come-First-Served (FCFS), Round Robin (RR) or Priority scheduling.

  > I/O: Blocking or Non-Blocking.

CHAPTER

FOUR

CONSTRAINTS

4.1 Hardware: Raspberry Pi

The Raspberry Pi is a comparatively low end single-board computer. As such, the design process must accommodate for these hardware specifications:

- SoC (System on a Chip) Broadcom BCM2835
- 700 MHz ARM core
- VideoCore IV GPU
- 512 MB of SDRAM.

Aside from an SD card, the Raspberry Pi does not feature any sort of non-volatile storage. Efficiency is a concern when constrained to a low capacity memory card. There is also an interface bottleneck, as both the USB and Ethernet share the same bandwidth.

An additional concern is of power distribution in regards to scalability; this would likely be resolved with the utilization of custom power supplies, but nonetheless it is something to consider.

4.2 Software: Architectural Considerations and Code Compilation

The Raspberry Pi hosts a single-core ARM processor, operating at 700MHz (with overclocking available). Due to this, any software developed for a Raspberry Pi should be in languages that are high-level, architecturally dependent and need to be recompiled on the Pi itself.

4.2.1 Operating System

As there are several pre-configured Linux operating system images for the Raspberry Pi, it was decided that it was not necessary to create a custom Linux-From-Scratch image. In order to choose an appropriate operating system, it was necessary to evaluate the compute cluster's basic needs. The key factors in the decision making process were:

- Performance
- Size
- Compatibility

With this in mind, it was decided that each node in the Punnet of Berries would run the **Arch Linux Arm** operating system. Arch Linux is:

- ARM compatible.

- Light-weight
  - The entire image is ~150 MB.
  - The default installation is bare bones, with nothing extra included.
- Boots in around 10 seconds.
  - Allowing the entire cluster to *boot in under 20 seconds*.

4.2.2 Programming Language

When selecting a programming language for the development of the Berry Batch scheduler, the key concerns were:

- Efficiency
  - As mentioned in the previous section, performance is a key issue when construcing a compute cluster.
- The existence of a Message Passing Interface (MPI) library.
- Ease of use.

It was decided that **C** would best suit these requirements. C is:

- A high level language.
- Efficient.
- Lightweight.

There is also a readily available MPI implementation. *OpenMPI* is an open source implementation of the MPI-2 specification. The Berry Batch utilises OpenMPI for communication with the cluster.

CHAPTER

FIVE

ARCHITECTURE

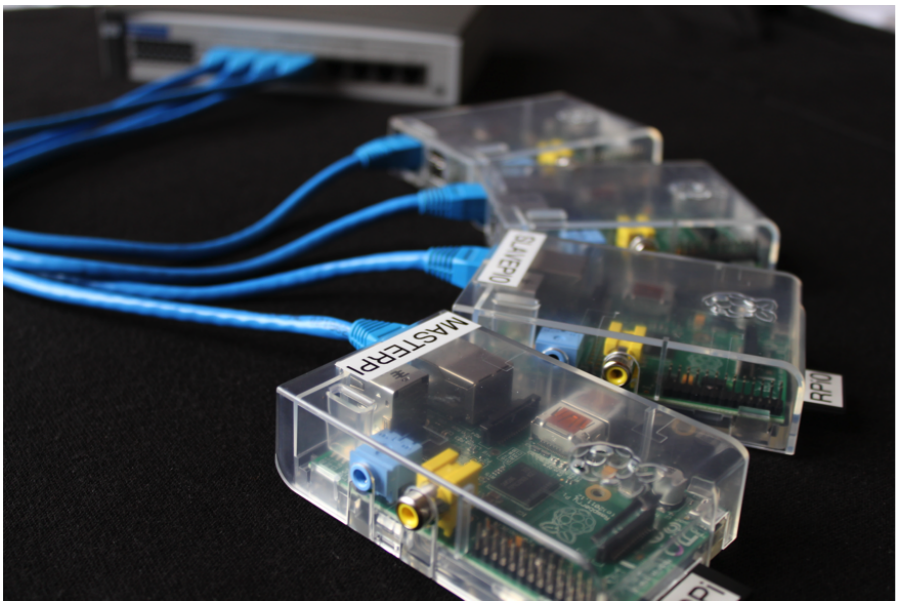
5.1 Hardware

Category	Qty	Description	Supplier	Unit Cost	Total Cost
Computer	6	Raspberry Pi Model B 512MB	element14	$41.80	$250.80
Storage	6	Assorted brands 8GB SD card	MULTIPLE	$12.00	$72.00
Enclosure/Case	6	Raspberry Pi Case (Clear)	element14	$9.79	$58.74
Power Supply	6	Power Supply USB 5V/1.2A	element14	$22.00	$132.00
Ethernet Cables	6	1.5m Cat-5e Patch Cable	MULTIPLE	$3.00	$18.00
Switch	1	8-Port 10/100 Mbps Ethernet Switch	UNKNOWN	$34.00	$34.00

The above table details the hardware components used to create the Punnet of Berries compute cluster.

5.2 System Design

A chain is only as strong as its weakest link.



In a distributed model super computer, this statement cannot be more true. No matter how quickly each individual node can process its allocated tasks, if the communications link between the system's master and each of the slave nodes becomes a bottleneck, then the entire system will suffer in performance.

In order to produce an efficient, flexible and reliable system, the selection of a networking topology which can fulfil those three specific characteristics is crucial. By selecting the star network topology as the basis for the Punnet of Berries super computer, the key requirements of efficiency, flexibility and reliability are met.

5.2.1 Efficiency

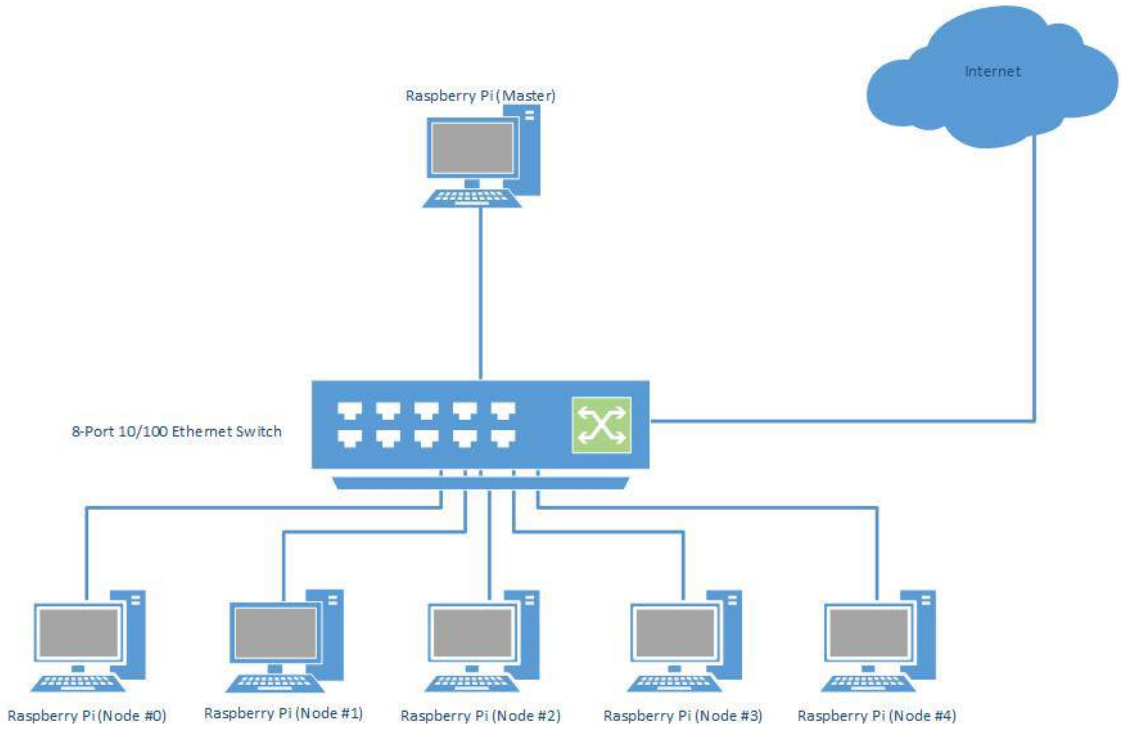
By design, star networks utilise a central device, such as a networking switch, to handle to flow of packets in a network. This allows data packets to only travel to those nodes which they are intended for, reducing the overall traffic in the network.

5.2.2 Flexibility

By utilising a central device for communications, a star network can easily be expanded to support additional nodes without affecting the entire network. The cluster can be expanded to accommodate as many nodes as required. However, in order to reduce the master pi's overhead, it is recommended that there be a 1:20 ratio between the master and slaves. Furthermore nodes can be removed at any time, if servicing is required.

5.2.3 Redundancy

The likelihood of total system failure is greatly reduced by using a central device for communication. The failure of an individual node wouldn't have any effect on the overall system. However, the central device failing would be catastrophic, bringing the entire system down.



CHAPTER

SIX

OPERATING SYSTEM

As discussed earlier, it was decided that **Arch Linux ARM** would be the Punnet of Berries' operating system.

The following section describes the steps that were taken to configure a basic Arch Linux image for use in the compute cluster. The base image was obtained from the Raspberry Pi downloads page (http://www.raspberrypi.org/downloads).

6.1 Setting up Arch Linux ARM

- Resizing:

```
# fdisk /dev/mmcblk0

  >> p

  >> d

  >> 2

  >> n

  >> e

  >> (return) = accept default partition number

  >> (return) = accept default start

  >> (return) = accept default end

  >> n

  >> l

  >> (return) = accept default start

  >> (return) = accept default end

  >> p

  >> w

# sync; reboot

# resize2fs /dev/mmcblk0p5
```

- Set the timezone:

  ```
  # ln -s /usr/share/zoneinfo/Australia/Melbourne /etc/localtime
  ```

- Update System:

  ```
  # pacman-key --init
  ```

  Press (Alt+F2) to switch to a 2nd virtual console, then enter the following command:

  ```
  # ls -R / && ls -R / && ls -R /
  ```

  Press (Alt+F1) to swtich back 1st virtual console. Check whether the "pacman-key –init" command has finished running.

  ```
  # pacman -Syu
  ```

- Users and Groups:

  Note: Use the following credentials when executing the steps below:

  Username: rpicluster

  Password: rpicluster

```
# useradd -m <username>

# passwd <username>

# groupadd admin

# gpasswd -a <username> admin
```

- Install the following using Arch Linux's package manager (pacman).

  1. **Sudo:**

     ```
     # pacman -S sudo
     ```

     – Give "admin" group sudo rights.

     ```
     # visudo
     ```

     Find "#%wheel ALL=(ALL) ALL". Change it to:

     ```
     %admin ALL=(ALL) ALL
     ```

  2. **Vim:** `# pacman -Syy vim`
  3. **GCC:** `# pacman -Syy gcc`
  4. **Make:** `# pacman -Syy make`
  5. **OpenMPI:** `# pacman -Syy openmpi`
  6. **OpenSSH:** `# pacman -Syy openssh`
  7. **NFS:** `# pacman -Syy nfs-utils`

CHAPTER

SEVEN

CLUSTER CONFIGURATION

- Set the hostname:

  Note: Enter the appropriate hostname for the device being setup

Device	Command
masterpi	`# echo "masterpi" > /etc/hostname`
slavepi0	`# echo "slavepi0" > /etc/hostname`
slavepi1	`# echo "slavepi1" > /etc/hostname`
slavepi2	`# echo "slavepi2" > /etc/hostname`
slavepi3	`# echo "slavepi3" > /etc/hostname`
slavepi4	`# echo "slavepi4" > /etc/hostname`

- Set a Static IP Address:

  Note: Enter the appropriate IP address for the device being setup

  ```
  # sudo vim /etc/conf.d/network
  ```

  interface=eth0
  netmask=255.255.255.0
  broadcast=172.20.32.255
  gateway=172.20.32.1

Device	Command
masterpi	address=172.20.32.82
slavepi0	address=172.20.32.83
slavepi1	address=172.20.32.84
slavepi2	address=172.20.32.85
slavepi3	address=172.20.32.86
slavepi4	address=172.20.32.87

  Note: Run the following on all nodes.

  ```
  # sudo vim /etc/systemd/system/network.service
  ```

  [Unit]
  Description=Network Connectivity
  Wants=network.target
  Before=network.target

  [Service]
  Type=oneshot

RemainAfterExit=yes
EnvironmentFile=/etc/conf.d/network
ExecStart=/sbin/ip link set dev ${interface} up
ExecStart=/sbin/ip addr add ${address}/${netmask} broadcast ${broadcast} dev ${interface}
ExecStart=/sbin/ip route add default via ${gateway}
ExecStop=/sbin/ip addr flush dev ${interface}
ExecStop=/sbin/ip link set dev ${interface} down

[Install]
WantedBy=multi-user.target

```
# sudo systemctl disable dhcpcd@eth0.service
# sudo systemctl enable network.service
```

- SSH Configuration:

  OpenSSH setup is a core requirement for OpenMPI functionality.

  - **slavepiX ONLY:**

    Follow the steps below for each of the slavepiX nodes.

    1. Generate an SSH Key Pair

       > Login as 'rpicluster'

```
# sudo ssh-keygen -t rsa -b 2048 -C "$(whoami)@$(hostname)-$(date -I)"
   >> (return) = accept default save location

   >> (return) = accept default 'blank' passphrase

   >> (return) = confirm default 'blank' passphrase
```

    2. Copy SSH Keys from Slave Nodes

```
# ssh-copy-id -i ~/.ssh/id_rsa.pub rpicluster@172.20.32.82
```

- NFS Configuration:

  - **Server Configuration [masterpi]**

```
# sudo mkdir /cluster_shared
```

    > Add the "cluster_shared" directory to NFS.

```
# sudo vim /etc/exports
```

      > Add the following line to the end of the file:

```
\/cluster_shared *(rw,sync)
```

```
# sudo chown -R nobody.nobody /cluster_shared
```

    > Edit the "nfs-common.conf" file.

```
# sudo vim /etc/conf.d/nfs-common.conf
```

      > Find "STATD_OPTS=". Change it to:

```
STATD_OPTS="-no-notify"
```

- **Client Configuration [slavepiX]**

  > Add the "cluster_shared" NFS share to the client.

  ```
  # sudo vim /etc/fstab
  ```

  > Add the following line to the end of the file:

  ```
  172.20.32.82:/cluster_shared /cluster_shared nfs
  defaults 0 0
  ```

- **Server Configuration [masterpi]**

```
# sudo systemctl enable sshd.service

# systemctl is-enabled sshd.service

# sudo systemctl enable nfsd.service

# systemctl is-enabled nfsd.service

# sudo systemctl enable rpcbind.service

# systemctl is-enabled rpcbind.service

# sudo systemctl enable rpc-idmapd.service

# systemctl is-enabled rpc-idmapd.service

# sudo systemctl enable rpc-mountd.serivce

# systemctl is-enabled rpc.mountd.service
```

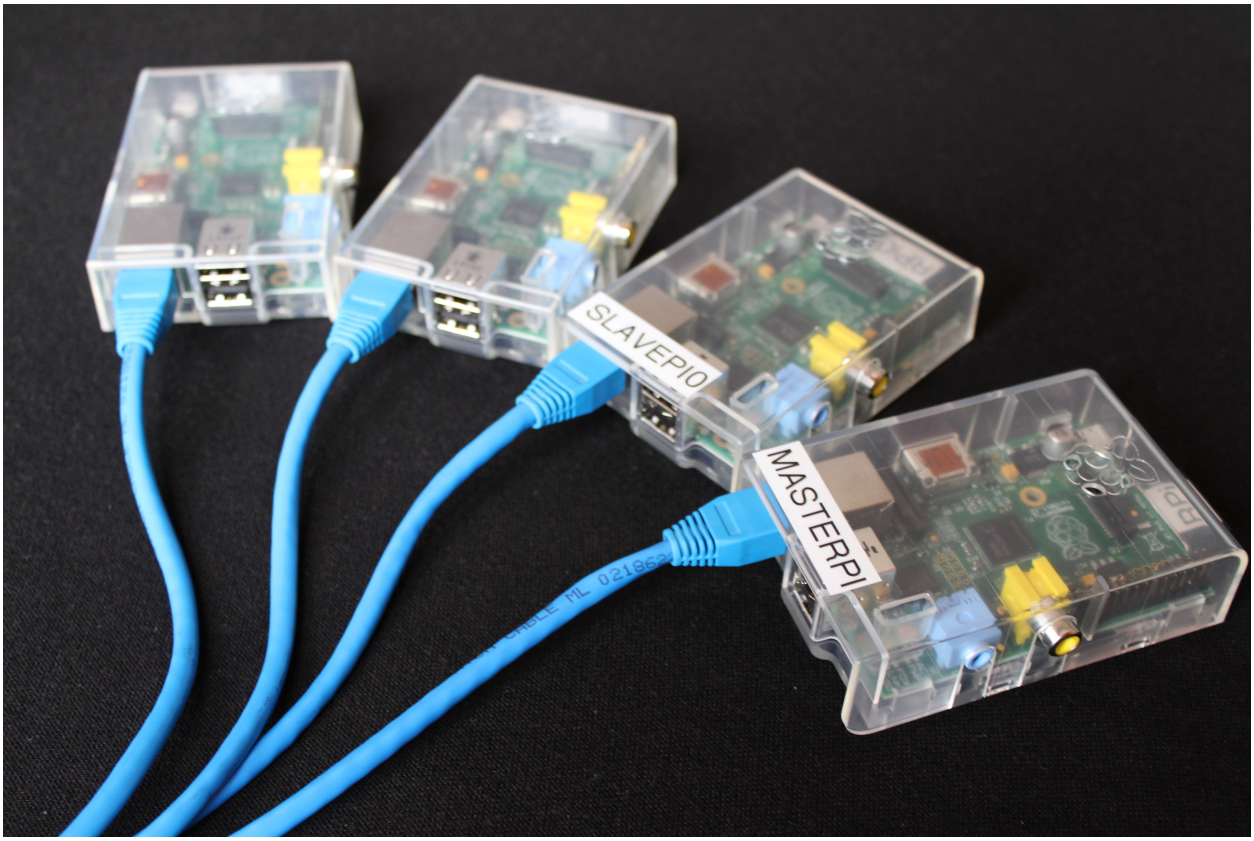
- **Client Configuration [slavepiX]**

```
# sudo systemctl enable sshd.service

# systemctl is-enabled sshd.service

# sudo systemctl enable rpcbind.service

# systemctl is-enabled rpcbind.service

# sudo systemctl enable rpc-idmapd.service

# systemctl is-enabled rpc-idmapd.service
```

CHAPTER

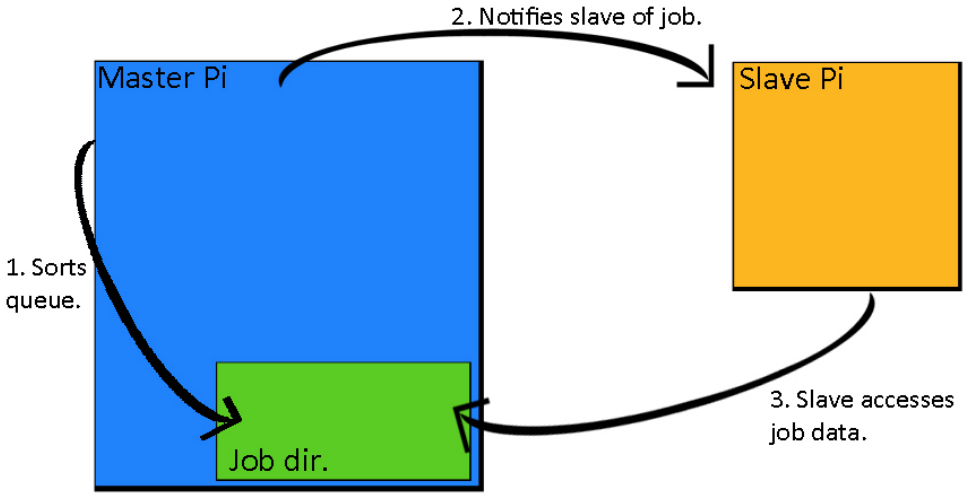
EIGHT

PROGRAM DESIGN



The Berry Batch consists of a centralised manager daemon and worker daemons. While the Punnet of Berries' centralised master node runs the manager daemon, each slave node runs a worker daemon.

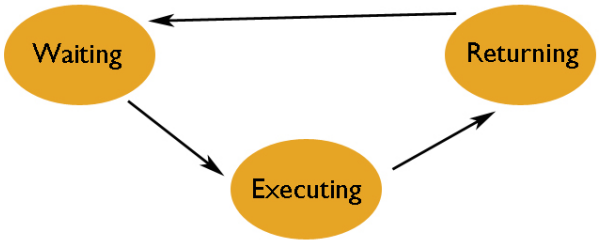
The queue is generated by parsing a "job directory" on the master and sorting the submitted jobs. The Berry Batch manager then iteratively pulls jobs from the queue and determines the most appropriate slave to carry out the job. This is done by monitoring the system's resources and the jobs running or waiting.



Rather than transferring the entire job file to and from the chosen slave, just the job's ID is sent. The slave accesses the job directory on the master and, using the job ID, gets just the information that it needs to complete the job. The process is completed by using OpenSSH and a Network File System (NFS). Using pre-generated RSA SSH keys, OpenSSH allows the connections between the master and the slaves to be established passwordless. The NFS allows the job directory to be shared over the network. The slaves can then access the directory as though it were on their own local system.

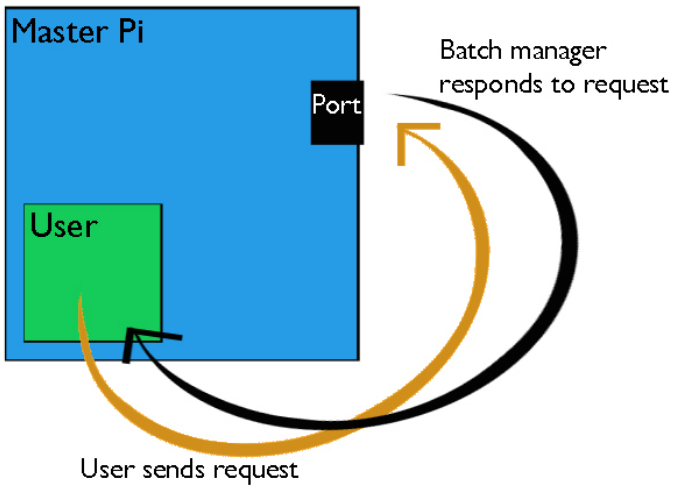
The workers exist to execute the jobs submitted to the system. They operate in a polling fashion. They will:

1. Wait to be assigned a job.
2. Execute the job.
3. Return the job's exit status.
4. Back to 1.



8.1 Users

Users are able to remotely connect to the Punnet of Berries using Secure Shell (SSH). They can then interact directly with the Berry Batch manager to manage their job requests. The manager daemon receives user requests by listening to a port. The user invokes a helper which sends a connect request to the port.



A user is able to:

- Submit jobs.
- View all queued jobs.
- View the status of all of a particular user's jobs.
- View the status of a particular job.
- Cancel their own jobs.

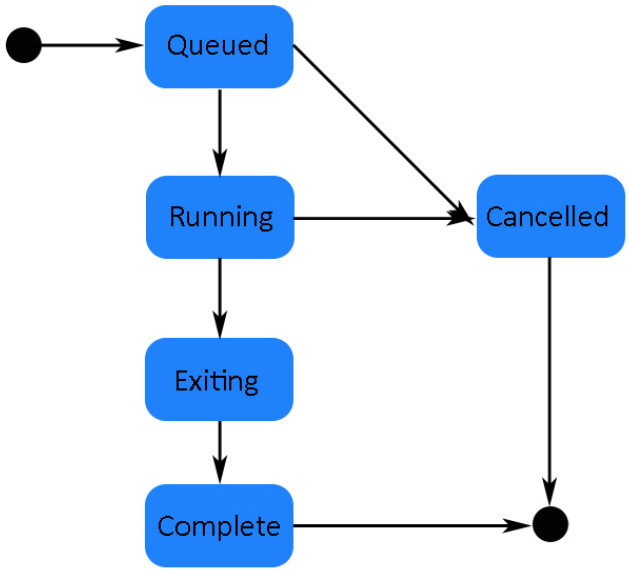
For example, to view the status of the jobs currently in the system:

```
# berrybatch status
ID      | Owner     | Name      | Walltime   | Status
-------------------------------------------------------
123     | user2     | MPI_Test  |            | Q
129     | user1     | job129    | 00:10:32   | C
139     | user3     | Test      | 00:00:32   | R
```

8.2 Job Execution

Once a job has been submitted, it moves through the following states:

- Queued
- Running
- Exiting
- Complete
- Cancelled



When a job has completed, a Berry Batch job summary file is written. This summary contains the details of the job's execution, such as the resources and walltime that were requested as well as what was actually used. Any standard output generated during execution is also included in the summary file. The file is saved to a directory within the owner's home directory.

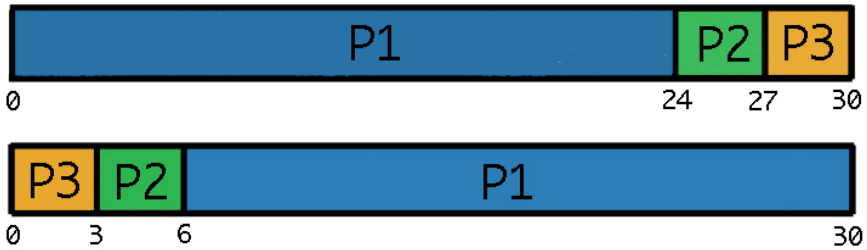
8.3 Scheduling Algorithms

As discussed earlier in this document, the user is able to select which scheduling algorithm the Berry Batch should use. The next few sections will outline the algorithms available.

8.3.1 First-Come-First-Served

The **First-Come-First Served (FCFS)** algorithm is very simple. As the name suggests, jobs are processed in the order that they are submitted. While the FCFS algorithm is very simple and easy to implement, its simplicity can also be its biggest flaw.

As shown in the following image, the waiting time for jobs in queue can vary greatly depending on the order jobs are submitted.



This is due to the running order being determined only by the job arrival time. Ignoring other factors, such as the estimated length of the job, often results in the CPU and device utilisation being lower than it could have been had shorter running jobs been scheduled first.

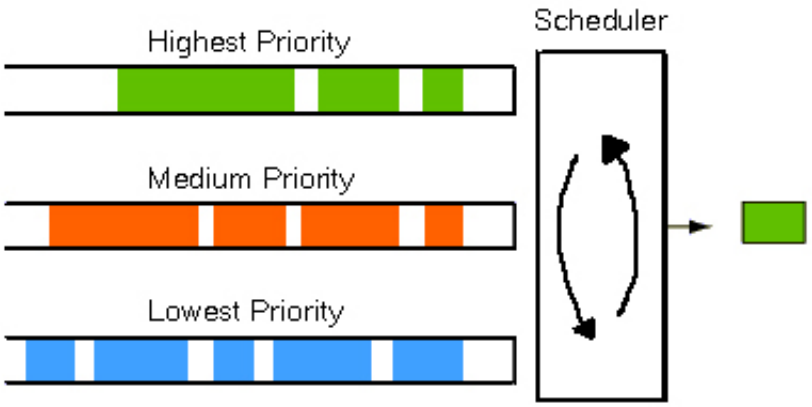
8.3.2 Priority Scheduling

The **Priority scheduling** algorithm involves each job being assigned a priority. Jobs are then run based on their priority, with the highest priority being run first.

When a job is submitted, the Berry Batch manager determines which priority queue the job should be assigned to. This is done by taking into account the estimated walltime and the resources requested. The priority queues are defined as:

Priority	Max Walltime (minutes)	Resources (no. nodes)
Low	20 mins	1 - 2
Medium	40 mins	1 - 3
High	> 60 mins	1 - 4
Special	> 60 mins	5

As jobs in the special queue require use of the entire cluster, they need special permission from the Punnet of Berries administrator before running.



The *special* queue has first priority, followed by the *high* queue, and so on. If the resources are not available for any job in the *special* queue, the manager looks in the *hight* queue for a suitable job, and so on. Within each queue, jobs are selected in a *First in First Out* fasion.

A problem that can occur with priority based scheduling is starvation. This means that low priority jobs are forced to wait indefinitely or are never run. This can occur when jobs with higher priority are submitted before the low priority job runs, blocking the lower priority job.

Two of the possible ways to fix this problem are:

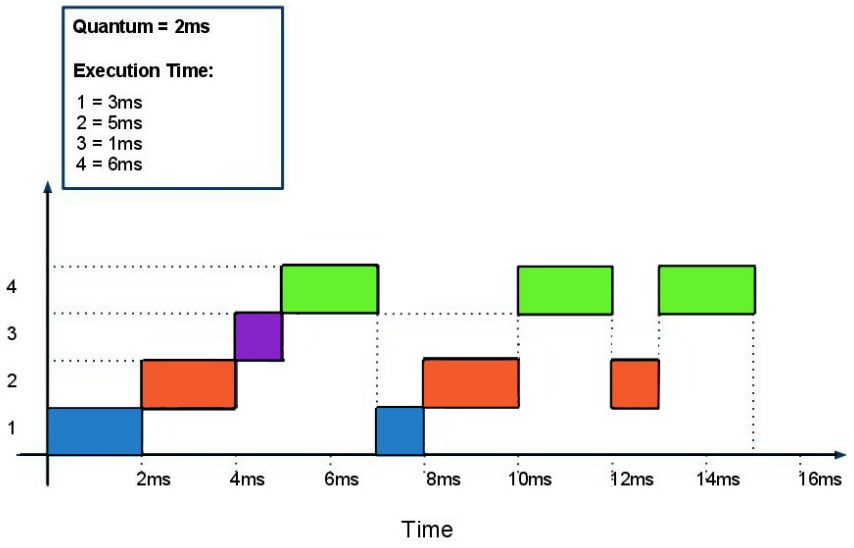
1. The job priorities can be re-evaluated based on how long they have been waiting. This would prevent low priority jobs from never running. After they reach a pre-defined wait threshold the job will be re-evalutated to a higher priority.

2. One or two of the compute cluster's nodes could be reserved for low priority jobs. These nodes would work their way through the low priority queue. Once the queue is empty, the reserved nodes can be opened up to service the other queues. After completing jobs from the higher priority queues, a check will be performed to determine if there are jobs waiting in low priority queue.

8.3.3 Round-Robin

As a part of the **Round-Robin (RR)** scheduling algorithm a time *quantum* is defined, in milliseconds. The job queue is a *First in First Out* queue, with new jobs added to the end of the queue. Each job in the queue is picked one at a time

and given running time. After a time interval of 1 quantum, *q*, the job is paused and the next in the queue is started. Once the end of the queue has been reached, the scheduler returns to the start of the queue, in Round-Robin fashion.



Each node is assigned a number of jobs from the FCFS queue, forming their own sub queues. The nodes iterate over their subset of jobs in an RR fashion.

As each job only gets small intervals of running time, the average waiting time for jobs can be longer. The job queue holds *n* jobs. Jobs with short walltimes can finish in a reasonable time. However, longer running jobs are continuously starting and stopping. These long running jobs must wait a maxiumum of *(n-1)/q* time units before each time it runs.

If the time quantum is large enough, the RR algorithm can turn into FCFS. If the quantum is extremely small, the RR algorithm can create the apperance of each job having its processor. However, the size of the quantum must make up for the overhead of stopping one job to start/ re-start another.

CHAPTER

NINE

SOURCE CODE

9.1 Punnet Scheduler

```
/*
    Main program daemon.

    1) Must include MPI header files and function prototypes.
    2) Initialize MPI environment
    3) Utilize message passing system.
    4) Terminate MPI environment.
*/

// These constants should ideally be defined in their own header file along with
// function prototypes
#include    "mpi.h"
#include    <stdio.h>
#include    <dirent.h>
#include    <pthread.h>
#include    <sys/types.h>
#include    <sys/socket.h>
#include    <sys/stat.h>
#include    <stdlib.h>
#include    <fcntl.h>
#include    <errno.h>
#include    <unistd.h>
#include    <syslog.h>
#include    <string.h>


#define     MASTER_NODE 0
#define     JOBFLAG 1
#define     KILLFLAG 2
#define     JOBDONEFLAG 3
#define     ALGORITHM_LONGEST_FIRST 4
#define     ALGORITHM_SHORTEST_FIRST 5
#define     ALGORITHM_FCFS 6
#define     ALGORITHM_ROUND_ROBIN 7
#define     FILENAME_MAX_LENGTH 20

static void init_master(void);
static void init_slave(int rank);
static worker_output_t do_job(worker_input_t);
static void process_work(worker_output_t);
```

```
static worker_input_t get_next_job(void);
static Job *jobQueue;

typedef struct
{
    int jobId;
    int status;
    char jobName[FILENAME_MAX_LENGTH];
    double walltime;
} Job;

int main(int argc, char *argv[])
{
    pid_t pid, sid; // Process ID & session ID
    int rank, initFlag, algorithmFlag, commFlag;
    char hostname[MAX_CHAR_HOSTNAME];
    pthread_t schedulerThread;

    // Fork of parent process
    pid = fork();
    if (pid < 0)
        exit(EXIT_FAILURE);

    if (pid > 0)
        exit(EXIT_SUCCESS);

    sid = setsid();
    if (sid < 0)
        exit(EXIT_FAILURE);

    // Change working directory of daemon
    // TODO: This must be changed to the working directory of OpenMPI
    if ((chdir("/")) < 0)
        exit(EXIT_FAILURE);

    // Daemon cannot interact with STDIN, STDOUR, or STDERR
    close(STDIN_FILENO);
    //close(STDOUT_FILENO);
    close(STDERR_FILENO);

    // --------------------------------
    // DAEMON IS INITIALIZED HERE
    // --------------------------------

    initFlag = MPI_Init(&argc, &argv);
    if (initFlag != MPI_SUCCESS)
    {
        printf("Error in initializing MPI environment. Terminating...");
        MPI_Abort(MPI_COMM_WORLD, initFlag);
    }
    // Initialize MPI environment.
    // The function accepts argc and argv pointers in order to differentiate
    // between command line arguments provided on "mpirun".

    MPI_COMM_RANK(MPI_COMM_RANK, &rank);
    // Allocates the rank of the calling node. Each node is defined a unique ID.

    MPI_Get_processor_name(hostname);
```

```
    // Gets hostname of calling node and assigns it to variable.

    if (rank == MASTER_NODE)
    {
        // Gather user input as to how the scheduler will operate.
        algorithmFlag = display_algorithm_menu();
        commFlag = display_comm_menu();
         // Create seperate thread for master scheduler
        pthread_create(&schedulerThread, NULL, init_master);
        gather_user_requests();
        // Merge main thread and master thread
        pthread_join(schedulerThread, NULL);
    }
    else
        init_slave(rank);

    // MPI environment must be destroyed.
    MPI_FINALIZE();
    return 0;
}

// This function allows the user to communicate via sockets to request
// scheduler statistics.
static void gather_user_requests(void)
{
    struct sockaddr_in address;
    int listen_fd, connection_fd;
    socklen_t address_length;
    char buffer[1024];

    // Create TCP/IP socket.
    // AF_NET: IPv4 address family
    // SOCK_STREAM: TCP type
    // 0: IP protocol
    // Function returns a file descriptor
    listen_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (listen_fd < 0)
    {
         perror("Failed socket creation");
         exit(1);
    }

    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(9999);

    // Call socket bind
    if (bind(listen_fd, (struct sockaddr *) &address, sizeof(address)) < 0)
    {
        perror("Bind Failed");
        exit(1);
    }
    // Listen.
    // The second argument is a maximum length to which the queue of pending
    // connections to the socket may grow.
    if (listen(listen_fd, 1) != 0)
    {
        perror("Listen Failed");
```

```
        exit(1);
    }
    // Accept incoming connections
    while((connection_fd = accept(listen_fd, (struct sockaddr *) &address, &address_length)) > -1)
    {
        // Here the user will communicate with the daemon scheduler.
        // The user will run some program to initiate the socket connection
        // and the daemon will be sent command line arguments to send back
        // the requested information.
        //read(connection_fd, buffer, 255);

        write(connection_fd, buffer, strlen(buffer));
    }

    close(listen_fd);
    return;
}

// This function will parse data read from the input socket
// in order to interpret user requests for scheduler diagnostics.
static void parse_user_input(void)
{


}

// This menu should provide the master node an option for the end user to
// specificy what scheduling technique to use and whether to use
// blocking/non-blocking IO.

// TODO: Fix terminating while condition
static int display_algorithm_menu(void)
{
    int algorithmOption;
    printf("Please specify the scheduling algorithm you want to employ.\n");
    printf("1) Next job waiting.\n2) Longest job first.\n3) Shortest job first.\n");
    scanf("%d", algorithmOption);
    do
    {
        switch(algorithmOption)
        {
            case ALGORITHM_FCFS:
                printf("Next job waiting selected.\n");
                break;
            case ALGORITHM_LONGEST_FIRST:
                printf("Longest job first selected.\n");
                break;
            case ALGORITHM_SHORTEST_FIRST:
                printf("Shortest job first selected.\n");
                break;
            default:
                printf("Please specify the scheduling algorithm you want to employ.\n");
                printf("1) Next job waiting.\n2) Longest job first.\n3) Shortest job first.\n");
                scanf("%d", algorithmOption);
                break;
        }
    } while();
}
```

```
// TODO: Fix terminating while condition
static int display_comm_menu(void)
{
    int commOption;
    printf("Please specify whether you want communication to be blocking or non-blocking.\n");
    printf("1) Blocking IO.\n2) Non-blocking IO.\n");
    scanf("%d", commOption);
    do
    {
        switch(commOption)
        {
            case 1:
                printf("Blocking IO selected.\n");
                break;
            case 2:
                printf("non-blocking IO selected.\n");
                break;
            default:
                printf("Please specify whether you want communication to be blocking or non-blocking
                printf("1) Blocking IO.\n2) Non-blocking IO.\n");
                scanf("%d", comm);
                break;
        }
    } while(comm != 1 || comm != 2);
}

// MASTER SECTION
// This function will be called after identifying the call device as a "manager".
// The manager should iteratively request a scheduled job from the queue and
// determine the most appropriate slave to undertake the job.
// After all processing has been complete, the master should recieve outstanding
// results from all slaves (sending a pull request ideally).
static void *init_master(void)
{
    int nodeNum, rank, jobCompletedNum = 0, jobID = -1, outstandingJobNum = 0;
    worker_input_t job;
    worker_output_t result;
    MPI_STATUS status;

    MPI_COMM_SIZE(MPI_COMM_WORLD, &taskNum);
    // Allocates the number of tasks in the provided communicator group.
    // As the communicator is defined as "world", it represents all available MPI
    // nodes. MPI_COMM_WORLD denotes all nodes in the MPI application

    if (nodeNum > 1)
        printf("MASTER: There are [%d] slave nodes.\n", nodeNum);
    else
        printf("MASTER: There is [%d] slave node.\n", nodeNum);

    // Seed slaves each one job. These jobs should be popped from the job queue
    // that has been established by the user.
    for (rank = 1; rank < nodeNum; rank++)
    {
        job = get_next_job();
        MPI_Send(&job, 1, MPI_INT, rank, JOBFLAG, MPI_COMM_WORLD);
        outstandingJobNum++;
    }
```

```
    while (outstandingJobNum != 0)
    {
        // Get result from workers
        MPI_Recv(&result, 1, MPI_UNSIGNED, MPI_ANY_SOURCE, DONE, MPI_COMM_WORLD, &status);
        outstandingJobNum--;

        // Determine which node completed that job.
        rank = status.MPI_SOURCE;

        job = get_next_job();

        // Assign a new job to now vacant node.
        MPI_Send(&job, 1, MPI_INT, rank, JOBFLAG, MPI_COMM_WORLD);
        outstandingJobNum++;
    }

    // Send a kill request to all workers, this signals a shutdown of cluster.
    for (rank = 1; rank < nodeNum; rank++)
    {
        MPI_Send(&s, 1, MPI_INT, s, KILLFLAG, MPI_COMM_WORLD);
    }
}

// SLAVE SECTION
// This will be called after identifying the calling device as a "worker".
// The worker node should operate in a polling fashion.
// The worker waits for messages from the master and proceeds to do the work
// and finally sends the result to the master.
static void init_slave(int rank)
{
    worker_input_t job; // Job buffer recieved by master
    worker_output_t result; // Result buffer after processing job
    MPI_STATUS status;

    // Recieve all messages from master node. This is blocking IO
    while(true)
    {
        // Recv(buffer, count, datatype, destination, tag, WORLD, status)
        // TODO: Alter arguments to match job script identifies
        MPI_Recv(&job, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status)

        // Check to see if the slave has been sent a kill command
        if (status.MPI_TAG == KILLFLAG)
            return;

        result = do_job(job);
        // Send(buffer, count, datatype, destination, tab, WORLD)
         // TODO: Alter arguments to match job script identifies
        MPI_Send(&result, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    }
}

// This is the function run to process a job on a worker node.
static worker_output_t do_job(worker_input_t job)
{

}
```

```
// This is a master function used to process the results returned by workers.
static void process_work(worker_output_t result)
{

}

// Function called by master in order to process next job in the queue.
// This function simply removes the next job from the queue and farms it to
// a worker.
static worker_input_t get_next_job(void)
{

}

// This function will be called by the user to add additional jobs to the queue.
// The queue determines what job will be issued to the workers next.
static void queue_job(int position)
{

}

static void parse_job_script(void)
{

}

// Check if job queue on master is empty.
static boolean is_queue_empty(void)
{

}

// This function reads the contents of the job directory, checks if a files
// is of the correct format then adds it to the job queue on the master node.
// Basically this function initializes the job default, unsorted job queue.
// Functionally, the queue must be able to be dynamically allocated filenames.

static boolean parse_job_directory(void)
{
    int i, fileCount = 0;
    DIR *dir;
    struct dirent *d;
    char *extension;
    // This should correspond to the current working directory of OpenMPI.
    dir = opendir(".");
    // Iterate over entire working directory.
    while ((d = readdir(dir)) != NULL)
    {
        // Check if file is regular.
        if (d->d_type == DT_REG)
        {
            // Tokenize file extension, delimited by last period.
            extension = strchr(d->d_name, '.');
            // If file extension matches that of a job script, increment file count.
            if (strcmp(extension, ".pjs") == 0)
                fileCount++;
        }
    }
```

```
    if (fileCount == 0)
        return false;
    closedir(dir);

    // Allocate memory for data structure containing all job structs.
    jobQueue = malloc(fileCount * sizeof(Job));

    dir = opendir(".");
    while ((d = readdir(dir)) != NULL)
    {
        if (d->d_type == DT_REG)
        {
            extension = strchr(d->d_name, '.');
            if (strcmp(extension, ".pjs") == 0)
            {
                jobQueue[i].jobId = i;
                jobQueue[i].jobName = ;
                i++;
            }

        }
    }
    return true;
}
```

9.2 User Interface

```
// This is the "client" application that utilizes sockets to communicate
// with the daemon scheduler in order to request current diagnostics of the
// job farming process.

#include <stdio.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>
#include <string.h>

int main(int argc, char *argv[])
{
    struct sockaddr_in address;
    int socket_fd, nbytes;
    char buffer[255];

    // Create socket.
    socket_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (socket_fd < 0)
    {
        printf("Error creating socket.");
        exit(1);
    }

    address.sun_family = AF_INET;
    address.sin_addr.s_addr = inet_addr("127.0.0.1");
    address.sin_port = htons(9999);

    // Conenct to daemon socket.
```

```
    if (connect(socket_fd, (struct sockaddr *) &address, sizeof (address)) < 0)
    {
        printf("Connection to daemon failed.");
        exit(1);
    }

    // No additional command line arguments.
    // TODOD: Change packet size.
    if (argc == 0)
    {
        nbytes = snprintf(buffer, 255, "SHOWALLSTATUS");
        write(socket_fd, buffer, nbytes);
        nbytes = read(socket_fd, buffer, 255);
        buffer[nbytes] = 0;
        printf(buffer);
    }

    close(socket_fd);
    return 0;
}
```

CHAPTER

TEN

TESTING

10.1 Tests Done

10.1.1 Cluster Performance Tests

In order to test the performance and functionality of the Punnet of Berries cluster, it was planned to test the system using a **High-Performance Linpack (HPL) Benchmark**. This benchmark provides informationon the processing performance of systems using estimates of how many GFLOPS (**G** iga **FL** oating-point **O** perations **P** er **S** econd).

Unfortunately, it proved to be extremely difficult to location a version of HPL that is compatible with Arch Linux ARM.

10.1.2 Batch System Tests

To prove that the Berry Batch is functioning correctly, it will undergo a range of tests. These include, but are not limited to, the following:

Test Cases	Expected Result
Initialize the 'Punnet Scheduler' application with a total of 6 nodes in the cluster.	A total of 5 worker nodes should report for duty.
Initialize the 'Punnet Scheduler' application with a total of 6 nodes in the cluster. Proceed to disconnect two nodes.	An alert should appear advising that two nodes have gone offline. The cluster should proceed to function as per normal.
Initialize the 'Punnet Scheduler' application with 2 nodes overclocked to 900MHz.	The application should report that 2 priority processing nodes are available if required.
View the worker nodes state table.	Details the instantaneous system resources of the worker nodes including processor and memory utilization for each.
Submit a collection of short jobs using the "Round Robin" algorithm. (Repeat: 10 times)	After running the test 10 times, some of the job assignments to each of the nodes should be the same in each test.
Submit a collection of assorted jobs using the "Round Robin" algorithm. (Repeat: 10 times)	After running the test 10 times, some of the job assignments to each of the nodes should be different in each test.
Submit a collection of short jobs using the "First Come First Serve" algorithm. (Repeat: 10 times)	After running the test 10 times, the jobs should be processed in the order the appear in the queue. This should be consistent for each test.
Submit a collection of assorted jobs using the "First Come First Serve" algorithm. (Repeat: 10 times)	After running the test 10 times, the jobs should be processed in the order the appear in the queue. This should be consistent for each test.
Overclock one node to 900MHz. Submit a collection of jobs, including one high priority job using the "Priority Scheduling" algorithm. (Repeat: 10 times)	The node overclocked running at 900MHz should be assigned the high priority job in every test which is run.
Assign one node as a priority processor. Submit a collection of jobs including one high priority job using the "Priority Scheduling" algorithm. (Repeart: 10 times)	The node assigned as a priority processor should be assigned the high priority job in every test which is run.
User views all current jobs.	List of jobs currently in the system.
User views a specific job.	Details of the given job.
User cancels a queued job.	Confirmation of job cancellation.
User cancels a running job.	Confirmation of job cancellation.
User cancels a completed job.	Warning of invalid state.
User cancels an already cancelled job.	Warning of invalid state.

CHAPTER

ELEVEN

ROLES AND RESPONSIBILITIES

The members of TeamPi are:

- **Alyssa Biasi:**

  Primary Role: Documentation, Testing and Management

  Secondary Role(s): Operating System Configuration

- **Adrian Zielonka**

  Primary Role: Hardward and Operating System Configuration

  Secondary Role(s): Application Development, Documentation

- **Zach Ryan**

  Primary Role: Application Development

  Secondary Role(s): Documentation

CHAPTER

TWELVE

WORK BREAKDOWN BY TEAM MEMBER

12.1 Alyssa Biasi's Log

12.1.1 Week 1

1. Raspberry PI cross complier recipe - Milestone 1.
2. Project research.

12.1.2 Week 3

1. Evaluation of project management tools

   > Trello

   > Gantter

   > Jira

   > Redmine
2. Setting up Redmine

   > Hosted by www.hostedredmine.com
3. Setting up repositories

   > hostedredmine.com only supports SVN

   > Repository hosted on www.github.com/rmit-teamPi

   - Using GitHub's "Organizations" to allow team access
   - GitHub provides support for SVN allowing individual members to pick their preferred method of version control.

12.1.3 Week 4

1. Arch Linux Arm image setup.

   > Partition layout changed in July 2013 (http://davidnelsono.me/?p=218)

12.1.4 Week 5

1. Setting up docutils, rst2pdf and Sphinx on a VM running Ubuntu for the generation of documentation.

    > apt-get install python-docutils

    > apt-get install rst2pdf

    > apt-get install python-sphinx

    > apt-get install texlive-latex-recommended

    > apt-get install texlive-latex-extra

    > apt-get install texlive-fonts-recommended

2. Work on design specification.

12.1.5 Week 6

1. Work on design specification.
2. Completed Milestone 2 - design specification.

12.1.6 Week 7

1. Setting up Arch Linux Arm images on 8GB SD cards.
2. Set up basic documents and sphinx for the final portfolio.

12.1.7 Week 8

1. Fixing SD cards to cluster specifications.

12.1.8 Week 11

1. Portfolio

12.1.9 Week 12

1. Portfolio

12.2 Adrian Zielonka's Log

12.2.1 Week 01

1. Brainstormed and researched viable project ideas.

12.2.2 Week 02

1. Brainstormed and researched viable project ideas.

12.2.3 Week 03

1. Brainstormed and researched viable project ideas.

12.2.4 Week 04

1. Brainstormed and researched viable project ideas.
2. Worked on Design Specification.

12.2.5 Week 05

1. Worked on Design Specification.

12.2.6 Week 06

1. Worked on Design Specification.
2. Complete Milestone 2 - design specification.
3. Installed ArchLinux ARM (for Raspberry Pi) image (8GB SD Card).
4. Setup "masterpi" image for MASTER Raspberry Pi.

12.2.7 Week 07

1. Configured "masterpi" image to reduced unneeded packages.
2. Setup OpenMPI, OpenSSH and NFS sharing for the MASTER Raspberry Pi.

12.2.8 Week 08

1. Installed ArchLinux ARM (for Raspberry Pi) image (8GB SD Card).
2. Setup "slavepi0" image for SLAVE #0 Raspberry Pi.
3. Created baseline performance benchmark of single Raspberry Pi.

12.2.9 Week 09

1. Networked all Raspberry Pi nodes together
2. Setup OpenMPI, OpenSSH and NFS sharing for the SLAVE #0 Raspberry Pi.

12.2.10 Week 10

1. Portfolio
2. Added SLAVE #1 and SLAVE #2 to the cluster.
3. Setup OpenMPI, OpenSSH and NFS sharing for the SLAVE #1 Raspberry Pi.
4. Setup OpenMPI, OpenSSH and NFS sharing for the SLAVE #2 Raspberry Pi.

12.2.11 Week 11

1. Portfolio
2. Tested intercommunication between Raspberry Pi nodes

12.2.12 Week 12

1. Portfolio

12.3 Zach Ryan's Log

Note: As you can see, Zach created this empty template and never filled it in.

12.3.1 Week 1

12.3.2 Week 2

12.3.3 Week 3

12.3.4 Week 4

12.3.5 Week 5

12.3.6 Week 6

12.3.7 Week 7

12.3.8 Week 8

CHAPTER

THIRTEEN

SUMMARY

When setting out to build a Raspberry Pi computing cluster, one should never aim to create a system capable of making the list of the Top 500 supercomputers in the world. After all, such a cluster is no substitute for a true supercomputer.

So what's the point you ask?

While the sheer performance of a Raspberry Pi computer cluster will never come close to that of a modern day supercomputer, it's a fantastic and fundamental representation of a full scale supercomputer on a budget. Sure the performance per dollar ratio will never even come close to that of a true supercomputer, but there's one major benefit to creating an RPi cluster. It's relatively affordable for anyone who would like to build one. Traditionally supercomputers have been reserved for organisations and educational institutions who can afford them. This means that access to such systems is often limited and out of reach for most individuals.

With the birth and release of the Raspberry Pi onto the market, thousands of interesting projects were brought to life. With the cost of a single unit coming in at approximately $35 (depending on country), the idea of a building a distributed computing cluster on a budget was made a reality. With only a couple hundred dollars spent, individuals can now have access to a mini-supercomputer in their own home.

'A Punnet of Berries' is a project created by 'Team Pi' which employs the use of a Raspberry Pi computing cluster to perform batch scheduling and processing of jobs submitted to the 'Berry Batch' application software. This custom written piece of software utilises a number of technologies to assign and distribute jobs based on predefined user selectable algorithms. These scheduling algorithms include the "Round Robin", "First Come First Serve" and "Priority Scheduling" algorithms, with the software being easily expandable to incorporate a variety of other scheduling algorithms. Using a master and slave model of communication, the cluster utilises the CPU processing power of each the slaves nodes to execute jobs and report back their results to the master node.

With the Raspberry Pi CPU clock speed coming in at a decent but conservative 700MHz, future implementations of the 'Berry Batch' software could incorporate the use of the Raspberry Pi's onboard GPU to boost the raw processing power each node and as a result the raw processing power of the entire cluster.

CHAPTER

FOURTEEN

REFERENCES

14.1 Raspberry Pi

Murray, M. (2012, July 13). *Raspberry Pi*. Retrieved from http://www.pcmag.com/article2/0,2817,2407058,00.asp

RPi Easy SD Card Setup. Retrieved from http://elinux.org/Rpi_Easy_SD_Card_Setup

Build Guide - Linux From Scratch on the Raspberry Pi. Retrieved from http://www.intestinate.com/pilfs/guide.html

Linux From Scratch. Retrieved from http://www.linuxfromscratch.org/lfs/view/development

14.2 Project Research

About Beowulf. Retrieved from http://yclept.ucdavis.edu/Beowulf/aboutbeowulf.html

Vaughan-Nichols, S. (2013, May 23). *Build your own supercomputer out of Raspberry Pi boards*. Retrieved from http://www.zdnet.com/build-your-own-supercomputer-out-of-raspberry-pi-boards-7000015831

Kiepert, J. (2013, May 22). *RPiCluster*. Retrieved from http://coen.boisestate.edu/ece/files/2013/05/Creating.a.Raspberry.Pi-Based.Beowulf.Cluster_v2.pdf

The Annual Unnamed UD Internet Contest - Problem 6: Supercomputer Job Scheduling. Retrieved from http://www.eecis.udel.edu/~breech/contest.inet.fall.09/problems/sc-sched.html

SImple Linux Utility for Resource Management. Retrieved from https://computing.llnl.gov/linux/slurm/overview.html

Sample PBS Script for Serial Job. Retrieved from http://qcd.phys.cmu.edu/QCDcluster/pbs/run_serial.html

Bell, J. *Operating Systems: CPU Scheduling*. Retrieved from

http://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/5_CPU_Scheduling.html

Apparatus, D. (2012, April 18). *HPC High Performance Compute Cluster with MPI and Arch*. Retrieved from http://apparatusd.wordpress.com/2012/04/18/hpc-high-performance-compute-cluster-with-mpi-and-arch/

14.3 Arch Linux ARM

Arch Linux ARM - Download Page. Retrieved from http://archlinuxarm.org/platforms/armv6/raspberry-pi

Arch Linux ARM - Official Installation Guide. Retrieved from https://wiki.archlinux.org/index.php/Official_Installation_Guide

Arch Linux ARM Forums - Resizing SD Card via command line. Retrieved from http://archlinuxarm.org/forum/viewtopic.php?f=31&t=3119

Arch Linux ARM Wiki - Beginners' Guide. Retrieved from https://wiki.archlinux.org/index.php/Beginners'_Guide

Arch Linux ARM Wiki - Keyboard shortcuts. Retrieved from https://wiki.archlinux.org/index.php/Keyboard_Shortcuts

Arch Linux ARM Wiki - Pacman-key. Retrieved from https://wiki.archlinux.org/index.php/Pacman-key

Arch Linux ARM Wiki - SSH Keys. Retrieved from https://wiki.archlinux.org/index.php/SSH_Keys

Arch Linux ARM Wiki - Sudo. Retrieved from https://wiki.archlinux.org/index.php/Sudo

Arch Linux ARM Wiki - Users and Groups. Retrieved from https://wiki.archlinux.org/index.php/Users_and_Groups

ELinux.org - Install Guide. Retrieved from http://elinux.org/ArchLinux_Install_Guide

Nelson, D. (2013, June 16). *Growing the root filesystem on Arch Linux ARM for the Raspberry Pi*. Retrieved from http://davidnelson.me/?p=218

Norman (2013, June 4). *Beginner's Guide to Arch Linux on the Raspberry Pi*. Retrieved from http://qdosmsq.dunbar-it.co.uk/blog/2013/06/beginners-guide-to-arch-linux-on-the-raspbery-pi/

Norman (2013, June 12). *Beginner's Guide to Arch Linux on the Raspberry Pi - Part 2*. Retrieved from http://qdosmsq.dunbar-it.co.uk/blog/2013/06/beginners-guide-to-arch-linux-on-the-raspbery-pi-part-2/

Van Der Veen, B. (2012). *I booted and SSH'd into my Raspberry Pi!*. Retrieved from http://bvanderveen.com/a/rpi-booted-static-ip-ssh/

14.4 OpenMPI

OpenMPI.org - Running. Retrieved from http://www.open-mpi.org/faq/?category=running

Wikipedia.org - Message Passing Interface.

Retrieved from http://en.wikipedia.org/wiki/Message_Passing_Interface#Example_program