

**University of Arkansas – CSCE Department**

**Capstone II – Final Report– Spring 2014**

# Affordable Raspberry Pi Cluster

**Brenna Blackwell, Nicolas Edward, Joshua Ross**

## Abstract

Building a Raspberry Pi cluster computer can solve the problem of affordable performance based computing by combining multiple Pis, along with other peripherals, to create a self-contained computing system that demonstrates performance comparable to market laptops available. Additionally, the system would provide a portable, cost-effective teaching tool focused on distributed computing.

## 1.0 Problem

While there are products on the market geared towards affordable, ubiquitous computing, the market is sparse. The products available require the user to have a basic functional knowledge in order to create more powerful tools from these parts, and affordable distributed computing systems are even more sparse.

## 2.0 Objective

To build a computer system using inexpensive parts, such as Raspberry Pis and other peripherals, in order to create a more powerful computing and teaching tool.

## 3.0 Background

### 3.1 Key Concepts

Networking - The four Pis are going to have to work together, and we intend to build the network using an Ethernet router. The individual Pis will be running their own instance of Raspian, the Debian-based Linux operating system optimized for the Raspberry Pi.

Concurrent/Parallel Computing - Because we want to demonstrate the parallel nature of the completed Raspberry Pi cluster, we intend to develop a problem for the cluster to solve concurrently. A message passing interface will be used in order to distribute the work, and an MPI will coordinate the activities for every Pi node in the cluster. The de facto standard approach for MPIs uses TCP/IP and socket connections.

Node Failure Management - The system must remain operational in the case of node failure. To keep the system operational, a node may be shut down or disallowed access to shared resources.

### 3.2 Related Work

Joshua Kiepert, a Ph. D. student at Boise State University’s Electrical and Computer Engineering Department, designed and built a Beowulf cluster consisting of Raspberry Pis in order to further his research. He published the method that he used, comparisons with other clusters computers, and the results of testing [1].

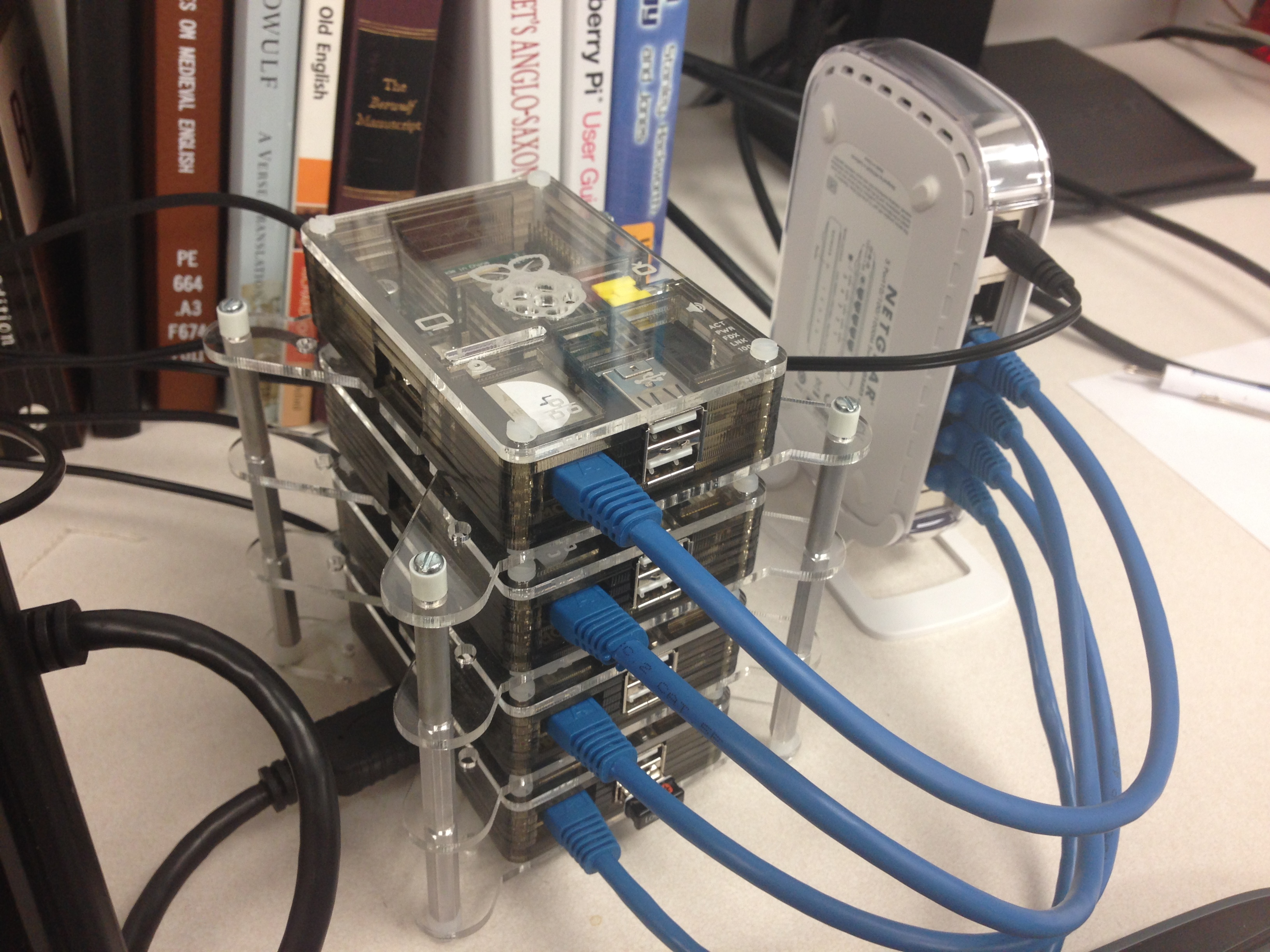
Additionally, Simon Cox, Professor of Computational Methods at Southampton University published his design on how to make a Raspberry Pi cluster, which can be found at the University’s website [2].

## 4.0 Design

### 4.1 Design Goals

The design goal is to build a more powerful computer using inexpensive, widely available components, such as Raspberry Pis, than can be achieved with those components individually.

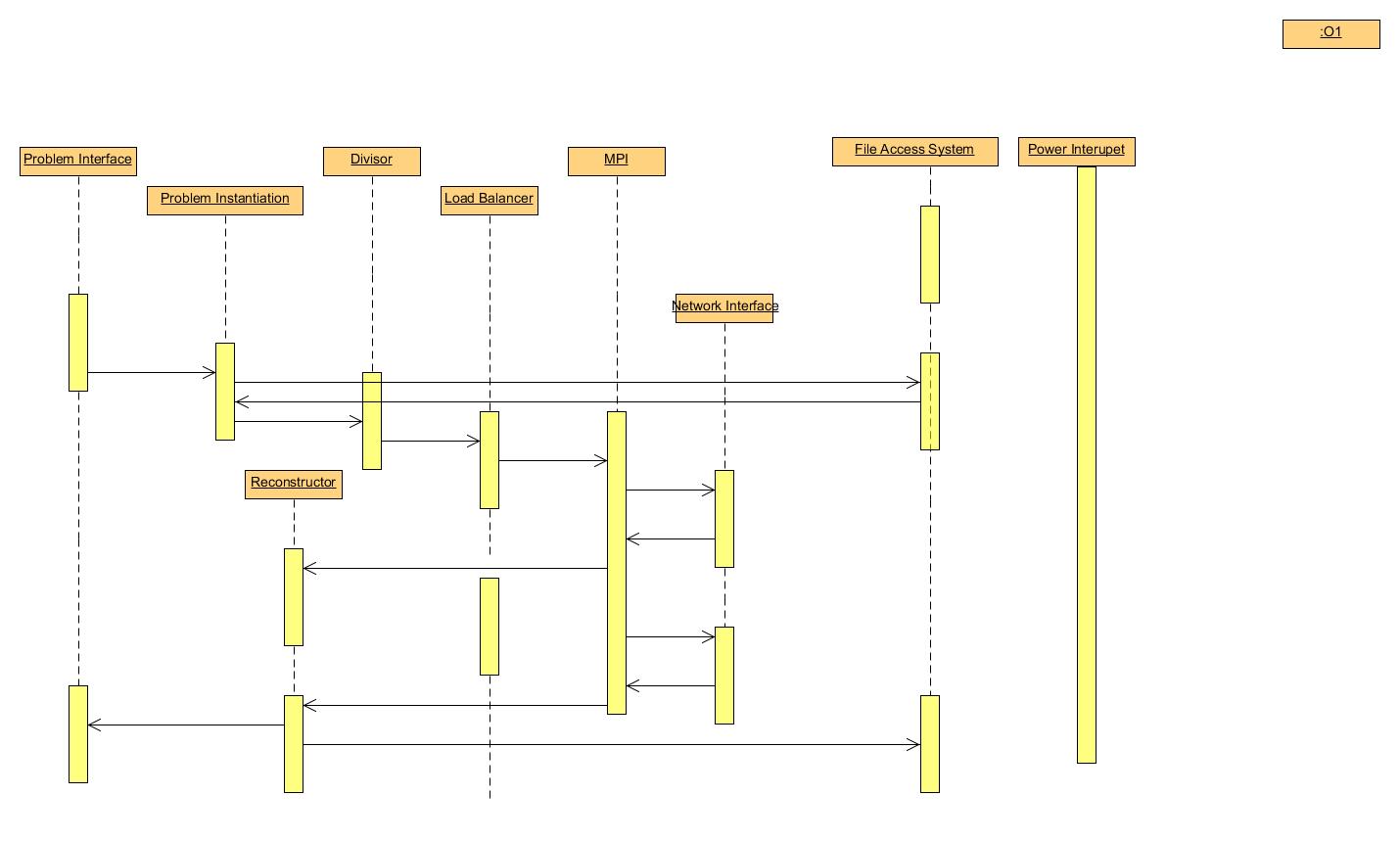
### 4.2 High Level Architecture



#### Image 1: The completed Raspberry Pi Cluster Connected to the Switch

#### Hardware

The hardware architecture consists of one Pi acting as the head node, with three leaf node Pis connected to it via a switch. Power will be routed to each Pi using independent 5V power supplies in order to avoid relying on a single unit. Use of the GPIO pins may be done to integrate some data passage between Pis. Additionally, we have included a 7” 1280x800 (720p) display, powered by its own 9V power supply, can be connected to the head node via HDMI cable. While any HDMI display could be connected, this small, lightweight option enhanced the portability aspect of the project’s objective.



#### Fig. 1: Conceptual Architecture Chart of the Cluster

#### Software:

The host Pi is in charge of data and instruction communication, while the leaf Pis perform the assigned work. MPICH2 [3] has been installed on all Pis in order to facilitate intra-cluster communication. MPI4Py [4], a library of MPI bindings for the Python programming language has also been installed on each Pi in the cluster. Because the problem program will be one of image processing, Pillow, an open source fork of the Python Imaging Library, [5] has also been installed on all the Pis. After some initial experimentation, we decided to use OpenMPI [6] instead of MPICH, as it worked better on the cluster.

#### Problem Program

After researching the Raspberry Pi, we decided to use Python in order to write our program. We are looking into image processing. The two programs that we wrote are i) to send a whole image to each leaf Pi, with the individual leaves performing a unique operation on the image, so that when it returns to the head node, three different operations have been done to the image, and ii) divide a single image into three parts, sending a single part to each Pi in order for the same operation to be applied, at which point the parts are sent back to the head node and a single image is reconstructed. Additionally, in order to emphasize testing of the supercomputer architecture aspect of the cluster, a program was written performing the same operations as program ii, except it was shell script communication based rather than MPI based, so as to demonstrate the differences in performance between MPI communication and network communication.

#### Results

After writing the problem programs described above, they were each run several times using a variety of subject images so that their performance could be examined and evaluated. A series of fifteen images were manipulated via the divide and recreate program implementing mpi4py, and the results are in the table as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Image | Time | Real | User | System |
| 1 | 0.301923036575 | 3.714 | 1.530 | 0.540 |
| 2 | 0.302290916443 | 2.670 | 1.640 | 0.450 |
| 3 | 0.269831180573 | 2.655 | 1.500 | 0.560 |
| 4 | 0.302568912506 | 3.439 | 1.540 | 0.520 |
| 5 | 0.588696002960 | 2.639 | 1.560 | 0.470 |
| 6 | 0.478888034821 | 3.048 | 1.490 | 0.570 |
| 7 | 0.486144065857 | 4.691 | 1.490 | 0.590 |
| 8 | 0.288335084915 | 3.500 | 1.490 | 0.570 |
| 9 | 0.465758085251 | 3.168 | 1.530 | 0.520 |
| 10 | 0.656569004059 | 3.596 | 1.470 | 0.520 |
| 11 | 0.509644985199 | 3.608 | 1.550 | 0.480 |
| 12 | 0.389425039291 | 3.881 | 1.550 | 0.510 |
| 13 | 0.496536016464 | 3.730 | 1.510 | 0.520 |
| 14 | 0.310142993927 | 3.761 | 1.470 | 0.630 |
| 15 | 0.314999103546 | 3.605 | 1.620 | 0.420 |
|  |  |  |  |  |
| Total: | 6.161752462387 | 51.705 | 22.940 | 7.870 |
| Average: | 0.410783497492 | 3.447 | 1.529 | 0.525 |

Table 1: Execution Time Data of the Divide and Recreate Algorithm Implementing MPI

The values in the “Time” column were found inside the program itself, using the native Python timing library. A timer was started before the image was loaded, and stopped and measured after the image was recombined and saved. The remaining three columns, “Real”, “User”, and “System” were measurements generated by the Linux time command. “Real” measures the amount of time that has passed while the entire process completes, and is sometimes referred to as “wall clock time.” “User” is the amount of time the CPU spent executing the process, and “System” is the amount of time spent on system calls for the process. The last two combined show how much actual CPU time is spent on the process. On average, the total time spent per picture was 2.054 s.

It is important to note that these measurements were only taken on the head node. This is because our goals for measurement were to see how well the system performed from the point of view of a person using the system like they would a standard computer. In future work, we would like to increase our measurement scope and break down exactly how much time is spent per node per task.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Image | # of times Image passed | Total Time | Time per Image | Real | User | System |
| 1 | 1 | 1.533 | 1.533 | 3.899 | 1.970 | 1.090 |
| 1 | 2 | 2.261 | 1.130 | 4.795 | 2.290 | 1.410 |
| 1 | 3 | 2.915 | 0.972 | 6.069 | 2.520 | 1.720 |
| 1 | 4 | 3.792 | 0.948 | 6.394 | 2.950 | 1.990 |
| 1 | 5 | 4.338 | 0.868 | 6.893 | 3.230 | 2.140 |
| 1 | 6 | 4.834 | 0.806 | 7.319 | 3.070 | 2.700 |
| 2 | 1 | 1.416 | 1.416 | 3.093 | 2.010 | 0.980 |
| 2 | 2 | 2.095 | 1.048 | 5.015 | 2.120 | 1.420 |
| 2 | 3 | 2.730 | 0.910 | 6.018 | 2.320 | 1.760 |
| 2 | 4 | 3.789 | 0.947 | 6.931 | 2.930 | 1.580 |
| 2 | 5 | 3.722 | 0.744 | 6.025 | 2.700 | 2.020 |
| 2 | 6 | 4.572 | 0.762 | 6.892 | 3.100 | 2.290 |

Table 2: Execution Time Data for the Pipeline Algorithm Implementing MPI

The above results for testing the pipeline program were obtained by running a single image through the pipeline a number of times in an attempt to achieve consistent results. “Total Time” is the time as taken by the Python script itself while “Time per Image” is this value divided by the number of copies of the image processed. “Real,” “User,” and “System” are the same as described above. As such, the times listed by the time command are of the total time to process all of the given images (in this case, one image a number of times) since the Python script is only run once to process multiple images.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Job Size (in bytes) | # of Jobs | Average Image/Job | Total Clock Time | Total Run Time | Average Time/Picture |
| 30000 | 161 | 1.19 | 76.43s | 6m 34.242s | 2.064s |
| 60000 | 130 | 1.47 | 70.56s | 6m 0.447s | 1.887s |
| 120000 | 94 | 2.03 | 60.56s | 5m 6.930s | 1.606s |
| 240000 | 67 | 2.85 | 56.23s | 4m 38.461s | 1.457s |
| 480000 | 42 | 4.55 | 52.91s | 4m 11.404s | 1.316s |
| 960000 | 27 | 7.07 | 48.17s | 3m 44.165s | 1.173s |
| 1920000 | 12 | 15.92 | 44.53s | 3m 16.483s | 1.028s |
| 3840000 | 7 | 27.29 | 43.88s | 3m 13.177s | 1.011s |
| 9680000 | 4 | 47.75 | 43.79 | 3m 13.479s | 1.012s |

Table 3 Execution Time Data for the Divide and Recreate Algorithm Run via Shell Scripts

For the shell scripted program, we stored 191 images, equaling approximately 41MB of data in our head node. Once the program starts, it will SSH into each Pi as well as clean out the directory structure. Then it goes through the image files and creates jobs consisting of groups of pictures under a certain size constraint, storing this list of in a text file. Individual files that were over the size constraint were relegated to their own individual job. After creating a list of jobs, the shell program then sends out a node execution script to the Pis, which is subsequently launched. The nodes run this script in a wait state until a job is sent to their directory, at which point the node’s working directory opens it and requests the specified files from the head node using SCP file transfer. The files that the node receives are then manipulated with a Python script and saved locally to a results folder. When the leaf node is done with the job, it deletes that job and launches a message script on the head node using an SSH call. When all the jobs are complete, the head node sends out a job to each node that makes the nodes exit their scripts. The head node then downloads all files back to a central results folder and timing is stopped.

Testing on different job sizes allowed us to adjust the average amount of images per job. An increase in average image/job leads to an increase in leaf node usage. It would appear that the best job size is between 384000 bytes and 9680000 bytes. As we used a small sample set of approximately 41 MB, our results might be better shown with a larger sample set and larger job sizes. In addition, integration of aspects of the shell script with those of the previous two problem sets could lead to a more user-friendly method for cluster usage.

### 4.3 Risks

|  |  |
| --- | --- |
| **Risk** | **Risk Reduction** |
| Overloaded Power Supply | Distributing power supply by using individual supplies instead of an entire-unit supply. |
| Overheating | Install a small fan unit if necessary. |
| Incorrect GPU Computations | Use unit testing and incremental development. |
| Idle CPU/GPU | Use a monitor program to load balance the Pis. |

### 4.4 Tasks

**Understand/gain background:** Read Raspberry Pi documentation. DONE

**Design:** Design system layout in order to integrate the four Pis and other necessary peripherals. Software design software flow to allow load balancing and data coherency among Pis when while the cluster is running the problem program. DONE

**Implementation:** Hook up the four Pis and peripherals according to designed specifications. DONE

**Test:** We are planning on testing each step of the implementation as we progress, so that when we move forward to the next step, the previous step has been thoroughly tested. Our major divisions in testing will occur when we go from coding for a single unit and move to concurrent computing in all units. DONE

**Demonstrate:** Run the problem program on the cluster, and display the results. DONE

**Document:** Like testing, we plan on writing documentation for the cluster as we are working on a specific task, so that by the time the task is complete, its documentation is written. DONE

### 4.5 Schedule

|  |  |  |
| --- | --- | --- |
|  | **Fall** | **Spring** |
| 1. Understanding |  |  |
| 2. Design |  |  |
| 3. Implement |  |  |
| 4. Test |  |  |
| 5. Demonstrate |  |  |
| 6. Document |  |  |

### 4.6 Deliverables

* Design Document: Upon completion of the project, we will submit a final design document outlining the cluster.
* Code: Any code that we write to test or run on the cluster will be provided.
* Final Report: A final report will be written discussing the process of designing and implementing the cluster.

## 5.0 Key Personnel

**Brenna Blackwell:** Blackwell is a senior Computer Engineering major in the Computer Science and Computer Engineering Department of the University of Arkansas. She has completed or is enrolled in Computer Organization and Embedded Systems. She is currently an iOS developer for the Biological and Agricultural Engineering Department at the University. She assisted the assembly of the cluster, configured the cluster software, wrote project documentation and instructions, curated the GitHub repository, and wrote the divide and recreate image processing code in Python.

**Nicolas Edwards:** Edwards is a senior Computer Engineering major in the Computer Science and Computer Engineering Department at the University of Arkansas. He has completed relevant courses in Embedded Systems, Computer Architecture, and Open Source Hardware. Edwards has gained experience in this area by using Arduino to detect door knocks and broadcast image from integrated camera to homeowners, as well as Light pulse detection was implemented as well to provide novel unlocking of door. He was the primary assembler of the cluster, configured cluster software, wrote administrative shell scripts to control cluster, and wrote a program using series of shell scripts to divide and manipulate an image.

**Joshua Ross:** Ross is a senior Computer Science major in the Computer Science and Computer Engineering Department at the University of Arkansas.  He has completed courses in Computer Organization, Algorithms, and Operating Systems.  Ross interned at FIS at their Little Rock, AR campus under a Programming Analyst where he developed a productivity-enhancing file cross-checker for the technical reviewers.  He wrote the pipeline image processing code in Python.

## 6.0 References

[1] Creating a Raspberry Pi Beowulf Cluster, http://coen.boisestate.edu/ece/files/2013/05/Creating.a.Raspberry.Pi-Based.Beowulf.Cluster\_v2.pdf

[2] Raspberry Pi at Southampton, http://www.southampton.ac.uk/~sjc/raspberrypi/

[3] MPICH, http://www.mpich.org/

[4] MPI4Py, http://mpi4py.scipy.org/

[5] Pillow, http://pillow.readthedocs.org/en/latest/

[6] OpenMPI, http://www.open-mpi.org/