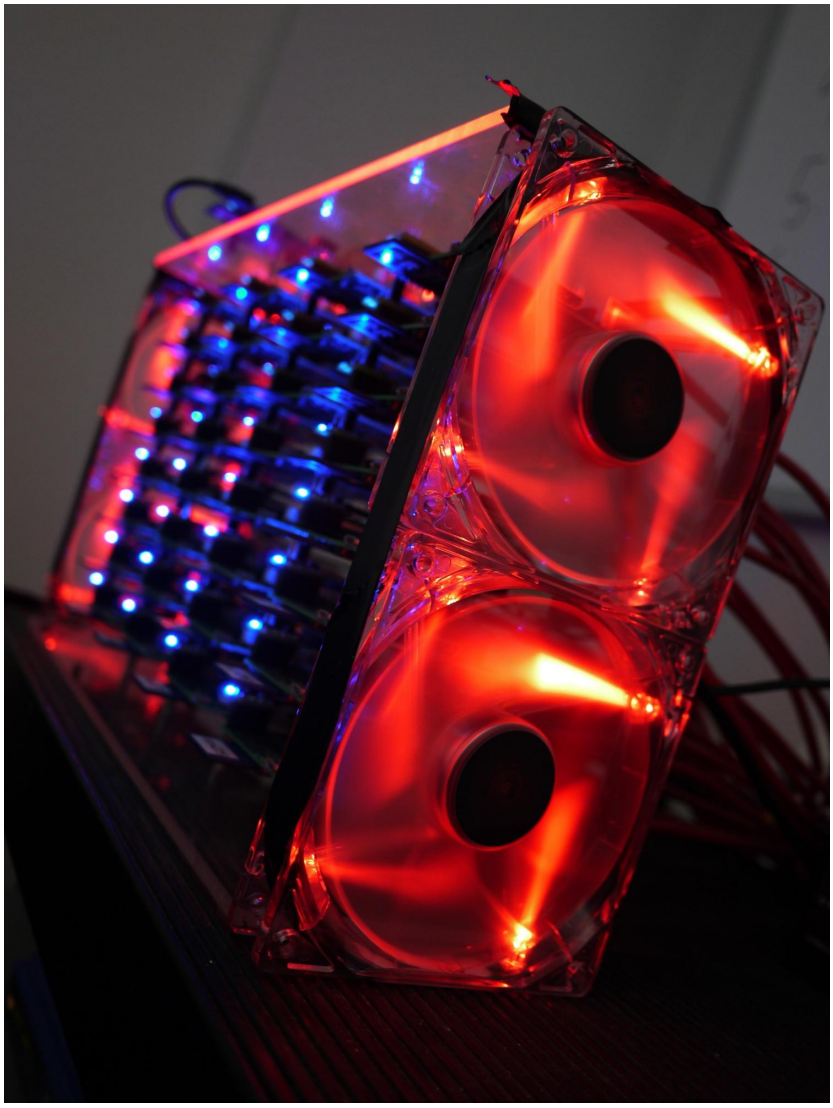
Creating a Raspberry Pi-Based Beowulf Cluster

Joshua Kiepert

Updated: May 22nd, 2013



BOISE STATE UNIVERSITY

Introduction

Raspberry Pis have really taken the embedded Linux community by storm. For those unfamiliar, however, a Raspberry Pi (RPi) is a small (credit card sized), inexpensive single-board computer that is capable of running Linux and other lightweight operating systems which run on ARM processors. Figure 1 shows a few details on the RPi capabilities.

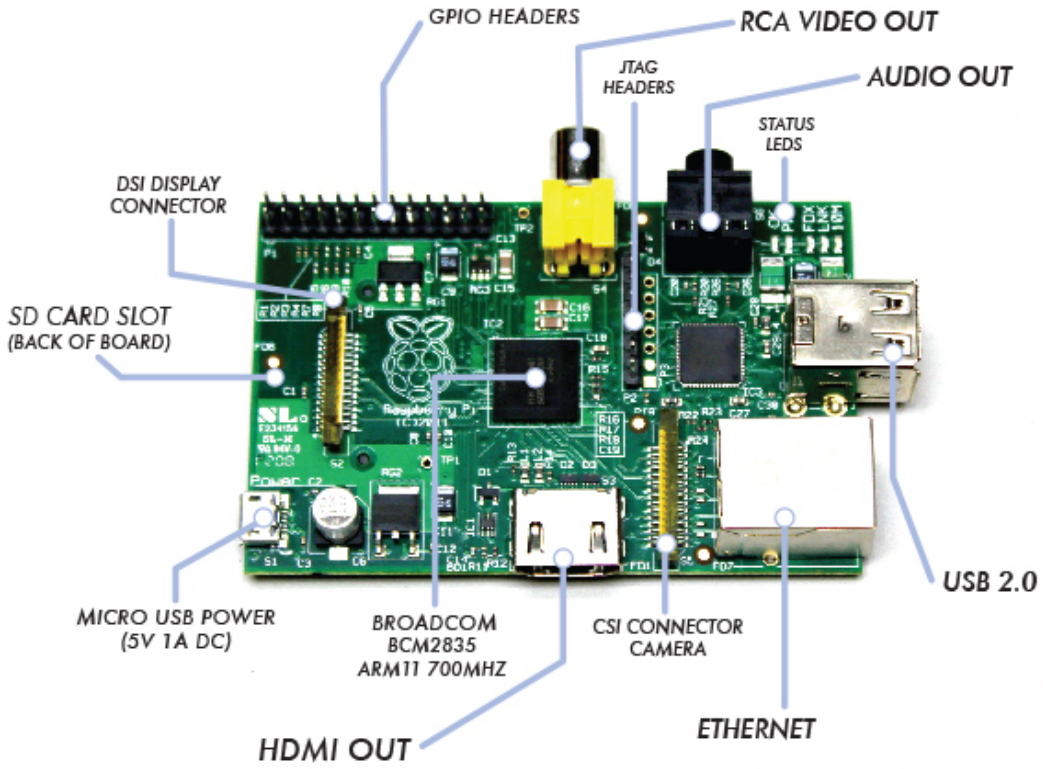


Figure 1: Raspberry Pi Model B (512MB RAM)

The RPiCluster project was started a couple months ago in response to a need during my PhD dissertation research. My research is currently focused on developing a novel data sharing system for wireless sensor networks to facilitate in-network collaborative processing of sensor data. In the process of developing this system it became clear that perhaps the most expedient way to test many of the ideas was to create a distributed simulation rather than developing directly on the final target embedded hardware. Thus, I began developing a distributed simulation in which each simulation node would behave like a wireless sensor node (along with inherent communications limitations), and as such, interact with all other simulation nodes within a LAN. This approach provided true asynchronous behavior and actual network communication between nodes which enabled better emulation of real wireless sensor network behavior.

For those who may not have heard of a Beowulf cluster before, a Beowulf cluster is simply a collection of identical, (typically) commodity computer hardware based systems, networked together and running some kind of parallel processing software that allows each node in the cluster to share data and computation. Typically, the parallel programming software is MPI (Message Passing Interface), which utilizes TCP/IP along with some libraries to allow programmers to create parallel programs that can split

a task into parts suitable to run on multiple machines simultaneously. MPI provides an API that enables both asynchronous and synchronous process interaction. While my simulation did not necessarily need a Beowulf cluster, since it simply required a number of computers on a common network, a cluster environment provided a very convenient development platform for my project thanks to its common file system and uniform hardware.

Here at Boise State University we have such a cluster in the “MetaGeek Lab” (or Onyx Lab), which is run by the Computer Science Department. The Onyx cluster currently utilizes 32 nodes, each of which has an 3.1GHz Intel Xeon E3-1225 quad-core processor and 8GB of RAM. Clearly, this setup has the potential to offer quite good performance for parallel processing.

So, with Onyx available to me, you may ask why I would build a Beowulf cluster using Raspberry Pis? Well, there are several reasons. First, while the Onyx cluster has an excellent uptime rating, it could be taken down for any number of reasons. When you have a project that requires the use of such a cluster and Onyx is unavailable, there are not really any other options on campus available to students aside from waiting for it to become available again. The RPiCluster provides another option for continuing development of projects that require MPI or Java in a cluster environment. Second, RPis provide a unique feature in that they have external low-level hardware interfaces for embedded systems use, such as I^2C, SPI, UART, and GPIO. This is very useful to electrical engineers requiring testing of embedded hardware on a large scale. Third, having user only access to a cluster is fine if the cluster has all the necessary tools installed. If not however, you must then work with the cluster administrator to get things working. Thus, by building my own cluster I could outfit it with anything I might need directly. Finally, RPis are cheap! The RPi platform has to be one of the cheapest ways to create a cluster of 32 nodes. The cost for an RPi with an 8GB SD card is ~$45. For comparison, each node in the Onyx cluster was somewhere between $1,000 and $1,500. So, for near the price of one PC-based node, we can create a 32 node Raspberry Pi cluster!

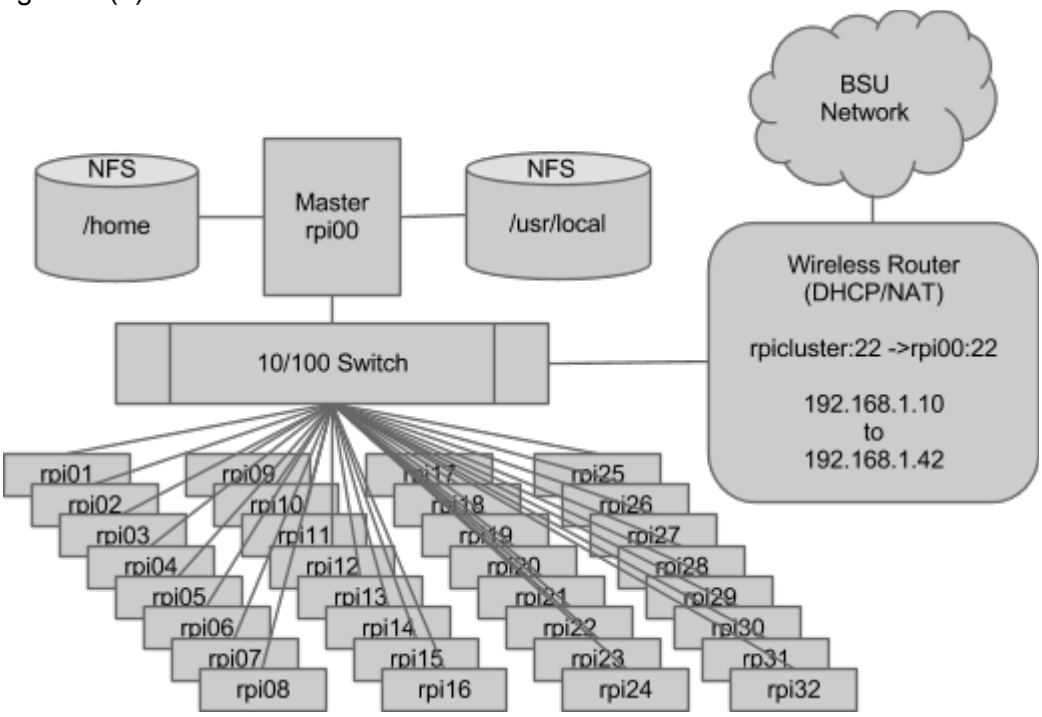
Although an inexpensive bill of materials looks great on paper, cheaper parts come with their own set of downsides. Perhaps the biggest downside is that an RPi is no where near as powerful as a current x86 PC. The RPi has a single-core ARM1176 (ARMv6) processor, running at 700MHz (though overclocking is supported). Additionally, since the RPi uses an ARM processor, it has a different architecture than PCs, i.e. ARM vs x86. Thus, any MPI program created originally on x86 must be recompiled when deployed to the RPiCluster. Fortunately, this issue is not present for java, python, or perl programs. Finally, because of the limited processing capability, the RPiCluster will not support multiple users simultaneously using the system very well. As such, it would be necessary to create some kind of time-sharing system for access if it ever needed to be used in such a capacity.

The overall value proposition is pretty good, particularly if cluster program development is focused on distributed computing rather than parallel processing. That is, if the programs being developed for the cluster are distributed in nature, but not terribly CPU intensive. Compute-intensive applications will need to look elsewhere, as there simply is not enough “horse power” available to make the RPi a terribly useful choice for cluster computing.

Building the System

There are really only five major components needed for a working cluster: computer hardware, Linux OS, an MPI library, an ethernet switch, and possibly a router. Of course, we have already chosen the

RPi as the computer hardware. Figure 2 (a) shows the overall network architecture. The system design includes 32 RPi nodes, 48-port 10/100 switch, Arch Linux ARM, and MPICH3. The bill of materials can be seen in Figure 2 (b).



(a) RPiCluster Network Architecture

Category	Quantity	Description	Supplier	Part Number	Cost	Extended
Computer	33	Rasberry Pi Model B	MCM Electronics	83-14421	$35.00	$1,155.00
Storage	33	8 GB Class 10 SD Card	Amazon	B003VNKNEG	$8.79	$290.07
Rack	74	Standoff M/F M3 25mm BB, 33mm OAL	Digi-Key	AE10782-ND	$0.40	$29.79
Rack	8	HEX NUT 0.217" M3	Digi-Key	H762-ND	$0.04	$0.31
Rack	8	MACHINE SCREW PAN PHILLIPS M3	Digi-Key	H744-ND	$0.07	$0.55
Rack	16	WASHER FLAT M3 STEEL	Digi-Key	H767-ND	$0.04	$0.58
Rack	2	Frame to mount stacks securely (15"x5"x1/4" Clear Plexiglass)	N/A			
Ethernet Cables	4	3 ft Cat-6 Network Ethernet Patch Cable - Red (Cat6) (Pack of 10)	Amazon	B0049VQ7P8	$16.90	$67.60
Switch	1	Cisco SF200-48 Switch 48 10/100 Ports (Managed)	Amazon	B004GHMU4W	$283.53	$283.53
Power Breakout	32	PTC RESETTABLE 33V 1.10A 1812L	Digi-Key	F3486CT-ND	$0.45	$14.32
Power Breakout	32	Tri-Color LED (LED RGB 605/525/470NM DIFF SMD)	Digi-Key	160-2022-1-ND	$0.28	$8.93
Lighting	2	Red LED Strip 12V (30 LED) 20"	Digi-Key	365-1503-1-ND	$12.29	$24.58
Power Supply	2	Thermaltake TR2 W0070 430W ATX12V (5V@30A)	Newegg	N82E16817153023	$36.00	$72.00
Cooling	4	Rosewill RFA-120-RL 120mm 4 Red LED Case Fan	Amazon	B00552Q8CM	$4.99	$19.96
Total:						$1,967.21

(b) Bill of Materials

Figure 2: Cluster Design

There are several operating systems available as pre-configured Linux images for the RPi: Raspbian “wheezy” (based on Ubuntu 11.10) and Arch Linux for ARM (aka “alarm”) (raspberrypi.org). Raspbian is available in both hard and soft float versions. “Hard-float” implies it is compiled with support for the hardware floating-point unit in the ARM processor, whereas soft-float implies it will use software libraries to implement floating-point operations. Clearly, hard float is desirable for performance, but for compatibility reasons soft-float is available. In the case of Raspbian, hard-float is not yet supported by

Oracle's Java Development Kit (JDK)/Java Virtual Machine (JVM) for ARM. It is worth noting that the Oracle JDK is not the only Java implementation available, so there is no need to limit performance with soft-float. The beauty of open source is that you can simply compile the program for your platform even if the vendor has not taken the time to do so. In this case, OpenJDK is the open source implementation of Java, and it is available on all Linux distributions, including Raspbian and Arch. Arch LInux for ARM is only available in the hard-float configuration.

So, which Linux distribution makes sense for a cluster of RPis? This depends greatly on how comfortable you are with Linux. Raspbian provides a turnkey solution in that it comes pre-installed with the LXDE (a full desktop environment), and being Debian based, there is a huge set of packages precompiled and available for the RPi. This is great for getting started with Linux and with the RPi in general. The downside to this distribution is its weight. With support for so many features to begin with, there is a staggering amount of daemons running all the time. As such, boot time is much longer than it has to be, and you have many packages installed that you most likely will never need or use.

Arch LInux on the other hand, takes the minimalist approach. The image is tiny at ~150MB. It boots in around 10 seconds. The install image has nothing extra included. The default installation provides a bare bones, minimal environment, that boots to a command line interface (CLI) with network support. The beauty of this approach is that you can start with the cleanest, fastest setup and only add the things you need for your application. The downside is you have to be willing to wade through the learning process of a different (but elegant) approach to Linux, which is found in Arch Linux. Fortunately, Arch Linux has some of the best organized information out there for learning Linux. An excellent place to start is the Arch Linux Beginner's Guide. I could go on about all the things I appreciate about Arch Linux, but I digress. As you might have guessed from my previous statements, and being a "Linux guy" myself (Linux Mint, primarily), I chose Arch Linux for the RPiCluster.

As for the MPI implementation, I chose MPICH primarily due to my previous familiarity with it through Parallel Computing classes (OpenMPI is available also). If you are running Raspbian, MPI can be installed by executing something like:

```
$ sudo apt-get install mpich2
```

That's it. However, on Arch Linux MPICH must be compiled from source, or installed from the AUR database. I did some looking on AUR, and only found an older version of MPICH (built for an old version of gcc). So, I went with the compile-from-source method. You can get the source from mpich.org. OpenMPI is available in the Arch Linux repositories, and it can be installed as follows:

```
$ sudo pacman -Syy openmpi
```

In the case of MPICH, compiling large amounts of source on a 700MHz machine can be tedious. So, why not utilize our Core i7 Quad-Core to do the job? Enter: QEMU. QEMU is a CPU architecture emulator and virtualization tool. It so happens that it is capable of emulating an ARM architecture very similar to the RPi. This allows us to boot an RPi image directly on our x86 or x86_64 system! There is one caveat, you need an RPI kernel for QEMU that knows how to deal with the QEMU virtualized hardware which is not normally part of the RPi kernel image. This involves downloading the Linux kernel source, applying a patch to support the RPi, adding the various kernel features and drivers

needed, and compiling. The compile process for the kernel only took about 5 minutes on my i7 laptop. Downloading the source from the Git repository, however, took much longer (~1.5GB).

There are several tutorials available which provide details for running QEMU with an RPI system image. A couple references I found useful are Raspberry Pi under QEMU and Raspberry Pi with Archlinux under QEMU. Aside from the need for a custom kernel, there are a couple system configuration changes needed within the RPi image to allow it to boot flawlessly. The changes primarily have to do with the fact that the RPi images assume the root file system is on /dev/mmcblk0p2 and the boot partition is on /dev/mmcblk0p1. QEMU makes no such assumptions so you have to map /dev/sda devices to mmcblk0 devices on boot. With the RPi system image adjusted and the custom kernel built, starting QEMU is something like the following:

```
$ qemu-system-arm -kernel ./zImage -cpu arm1176 -m 256 -M versatilepb -no-reboot
-serial stdio -append "root=/dev/sda2 panic=0 rw" -hda archlinux-hf-2013-02-11.img
```

Once you have a kernel image (zImage) that is suitable for QEMU you can point it at the new kernel and the RPi system image. Running an RPi image via QEMU may be significantly faster than working on the RPi, of course, this depends on the computer being used to run QEMU. Figure 3 shows the boot screen of Arch Linux ARM under QEMU.

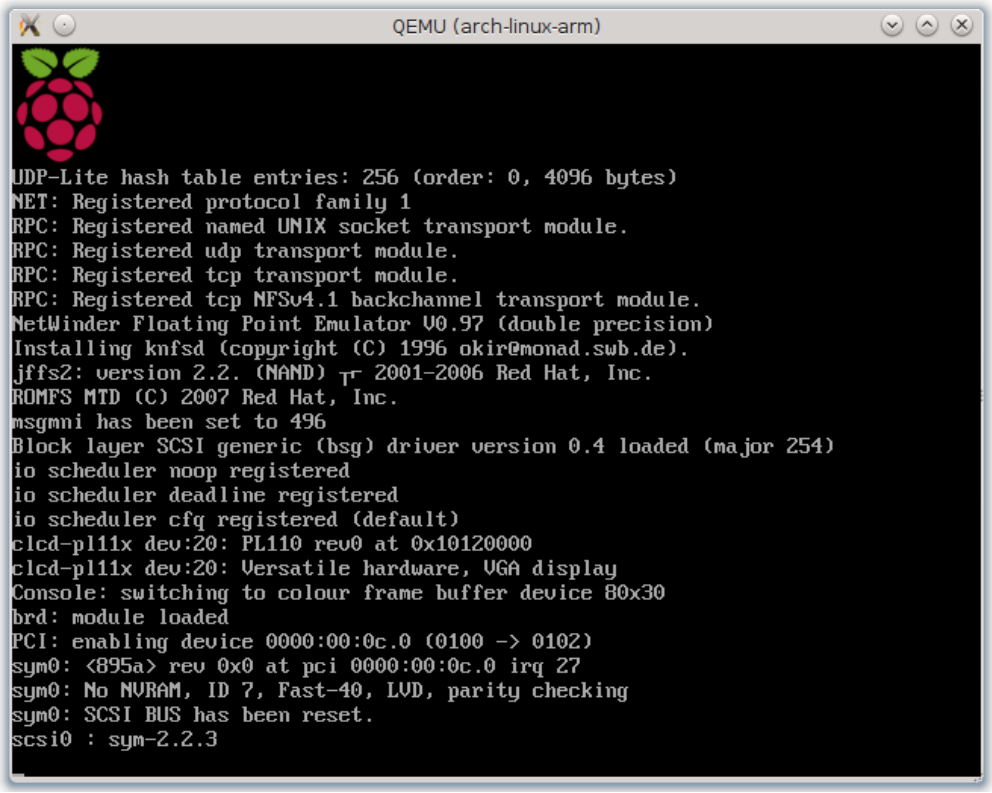


Figure 3: Arch Linux ARM under QEMU

With a working Arch Linux image under QEMU it is possible to update to the latest packages, install the necessary packages for the cluster environment, and compile MPICH for parallel computing, all from the comfort of you x86 Linux system. I used QEMU for all of the aforementioned tasks. Once

configuration of the image was completed with QEMU, I was able to simply write the resulting image to each RPi SD card (after modifying hostname and IP address, unique to each).

Below are a few of the steps involved for configuring an Arch Linux image for use in a cluster:

Change root password from default (root):

```
# passwd
```

Full system update:

```
# pacman -Syyu
```

Install typically needed packages, bold packages are need for compiling MPICH and for using NFS or OpenMPI:

```
# pacman -S nfs-utils base-devel openmpi sudo adduser nano vim gdb bc minicom

Set the local timezone:
# timedatectl set-timezone America/Boise
```

Adds a new user...<user> in this case (be sure to add groups wheel,audio,video,uucp when prompted):

```
# adduser <user>
```

Allow users that are part of the wheel group to use superuser permissions:

```
# chmod 600 /etc/sudoers
# nano /etc/sudoers
```

(Uncomment the line: %wheel = ... and save)

```
# logout
```

(Login as user who will be using MPI)

```
$ ssh-keygen -t rsa
$ ssh-keygen -t dsa
$ ssh-keygen -t edsa
```

Cluster setup is basically just configuring a few files so that all nodes can find each other and have password-less SSH access to one another.

Key configuration files for cluster setup under Arch Linux:

- /etc/exports
- /etc/idmapd.conf
- /etc/hosts
- /etc/hostname
- /etc/fstab
- /etc/network.d/ethernet-static
- /etc/conf.d/netcfg

Performance Comparison

Initially, I created a four-node test platform with RPis. So, it was possible to do some preliminary testing of performance before creating the full cluster. The first test was to determine the general performance of a single node running multiple threads, using the MPI libraries.

The MPI test program I used is one I developed during a parallel computing class at BSU. The program calculates pi using the Monte Carlo method. In this method, essentially the more random numbers you generate, the more accurate your resulting estimation of pi. It is an embarrassingly parallel way to calculate pi (i.e., scales easily to multiple independent processes, near 100% performance gain for each additional process added). All that is needed is a way to parallelize the generation of random numbers so that each process does not repeat the random sequence (remember, we are dealing with pseudorandom numbers here). In this case, I use a library called *prand*, developed by Dr. Jain and Jason Main at BSU.

Table 1 shows the hardware specifications of the various platforms tested for comparison.

Table 1: Test Hardware Specifications

Platform	Cost	OS	CPU	Cores	MHz
Onyx Node	$1,000	Fedora 16	Intel Xeon E3-1225 (x86_64)	4	3,100
Chromebook	$250	Arch Linux ARM	Samsung Exynos 5250 (ARMv7)	2	1,700
Raspberry Pi	$45	Arch LInux ARM	BCM2708 ARM1176JZF-S (ARMv6)	1	700

Figure 4 shows the execution time for 900 Million iterations of the Monte Carlo pi calculation program (PMCPI).

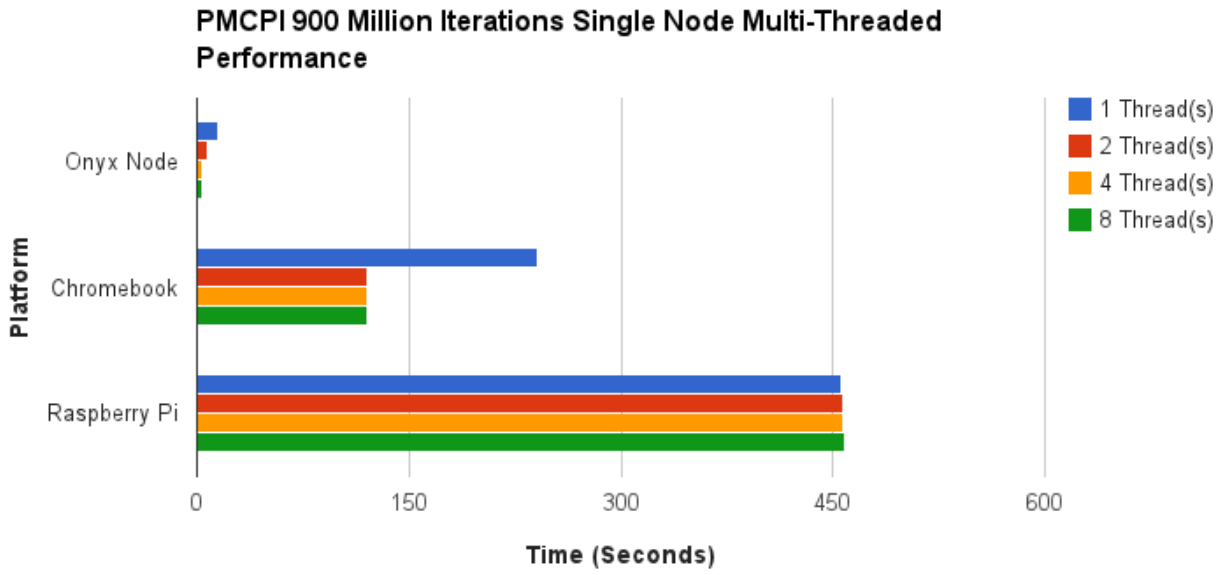


Figure 4: MPI Single Node Performance[1]

[1] This is running the MPI program on a single node with multiple threads, not in parallel with other nodes in a cluster.

If you can not tell from Figure 4, that is a maximum of 14.6 seconds for a single thread on an Onyx node. With four threads it goes down to 3.85 seconds thanks to four processing cores. With eight threads, it goes up to 3.90 seconds. Compare that to our lowly RPi, which requires 456 seconds for single thread. Additional threads do not help due to its single core design.

Figures 5 and 6 show the performance improvement as we add additional RPi nodes to participate in the calculation, in execution time and speed up, respectively.

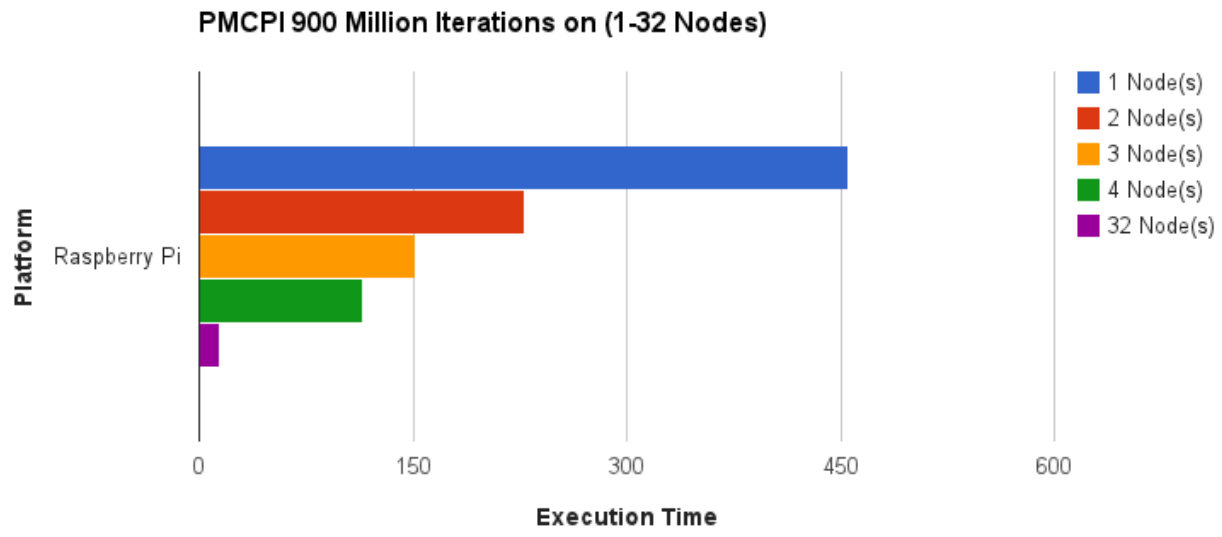


Figure 5: Execution Time Reduction (1-32 nodes)

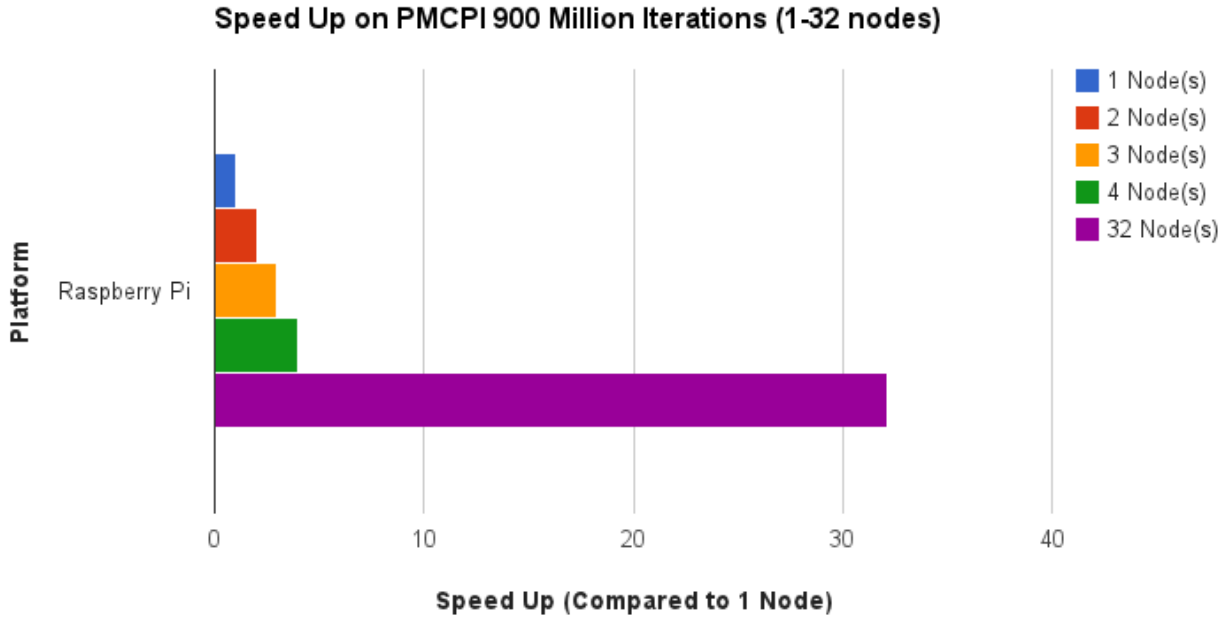
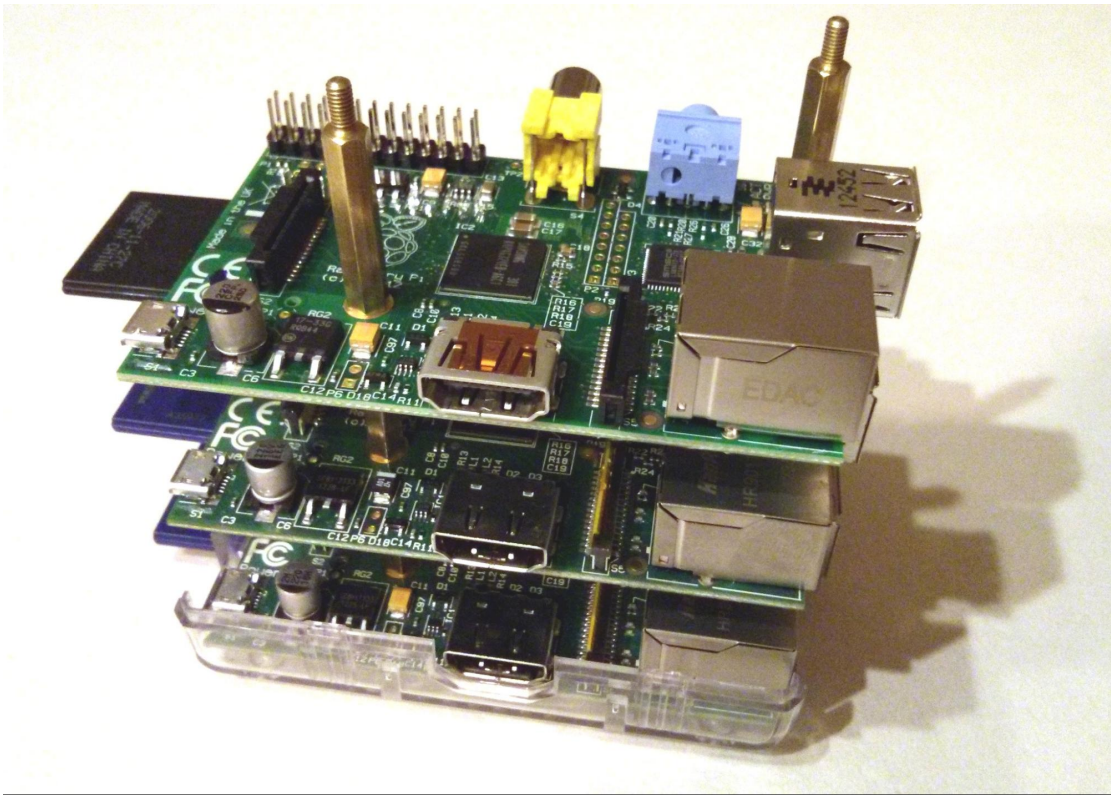


Figure 6: Speed-Up as Compared to 1 Node

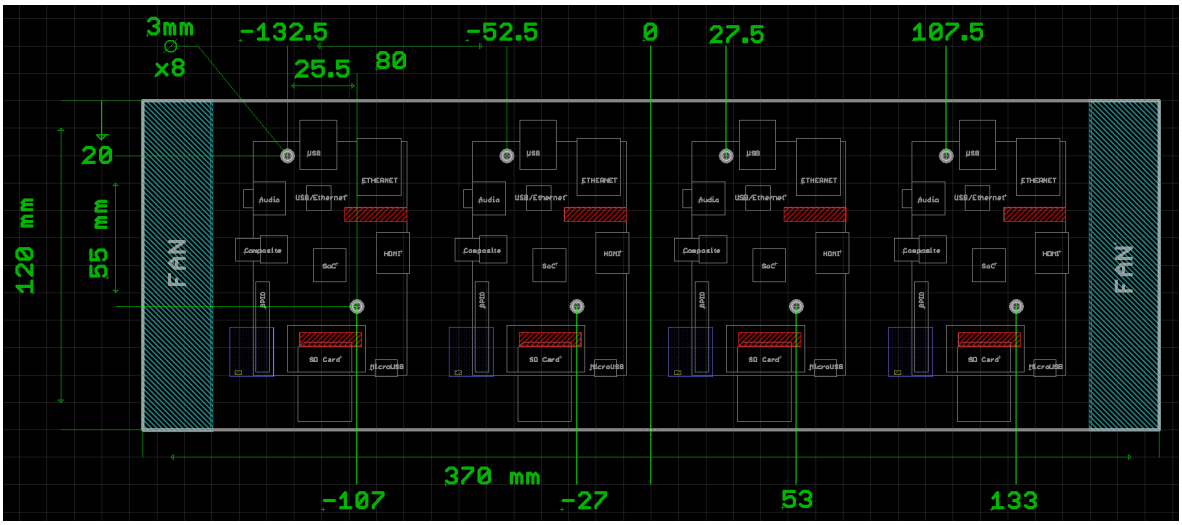
As is seen in Figures 5 and 6, we clearly get the expected performance improvement as we add additional nodes to participate in the calculations.

Rack and Power Design

One aspect of the cluster design that required quite a lot of thought was the rack mounting system and power distribution method. In order to keep the cluster size to a minimum while maintaining ease of access, the RPis were stacked in groups of eight using PCB-to-PCB standoffs with enough room in between them for a reasonable amount of air flow and component clearance. This configuration suited our needs for power distribution very well since it allowed for a power line to be passed vertically along each stack. Using this orientation, four RPi stacks were assembled and mounted between two pieces of ¼" acrylic. This created a solid structure in which the cluster could be housed and maintain physical stability under the combined weight of 32 Ethernet cables. Figure 7 (a) shows some initial experimentation using this method of mounting. The plexiglass layout for mounting each RPi stack was designed using EagleCAD, along with the power/LED PCB. Figure 7 (b) shows the layout.



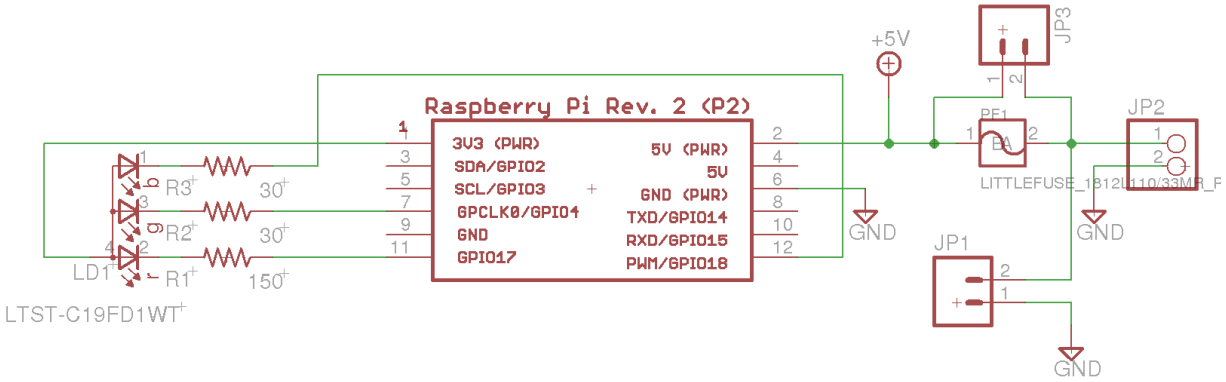
(a) Rack Mounting RPis Using PCB Stand-offs



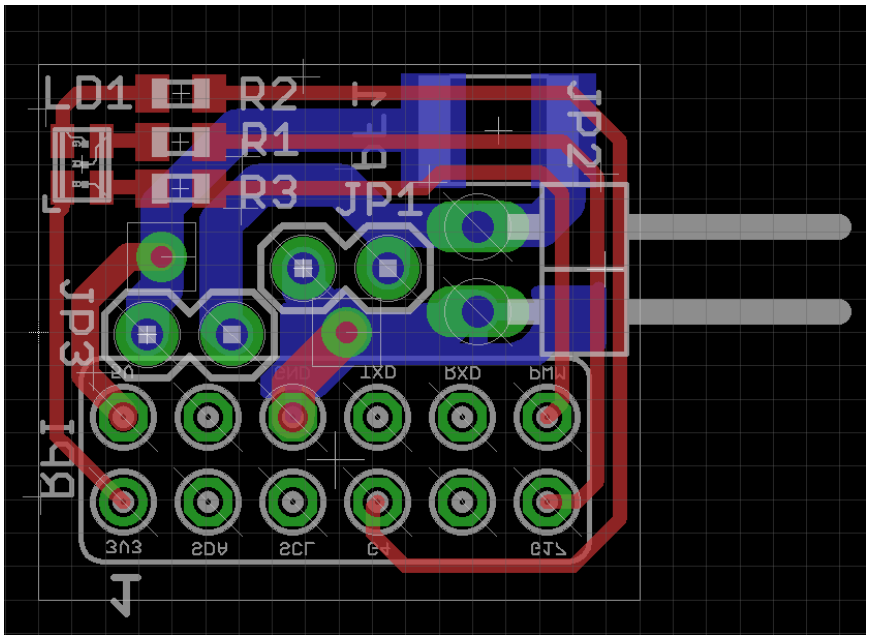
(b) Plexiglass Layout (Top/Bottom of RPiCluster Rack)

Figure 7: RPiCluster Rack Components

There are two methods of powering an RPi. Each RPi has a microUSB port that enables it to be powered from a conventional powered USB port. Using this method would require a microUSB cable and a powered USB port for every RPi in the cluster, which would add a lot more cabling to deal with. It also means that there would be cabling on both sides of the cluster. Alternatively, there is an I/O header on the side of the RPi which contains a 5V pin that can be used to power the board externally. The latter option was chosen as it would also allow for additional customization and control of each node. A custom PCB was created to fit the I/O header to provide power and an RGB LED (since electrical engineers must, of course, have LEDs to show that their project is working). Figure 8 (a) and (b) show the circuit and PCB layouts, respectively.



(a) RPiCluster Power/LED Board Schematic



(b) RPiCluster Power/LED Board PCB Layout
Figure 8: RPiCluster Power/LED Board

As seen in Figure 8 (a), aside from a RGB LED and some connectors, there is also a poly fuse (PF1). This was included to maintain short-circuit protection in the event of a board failure. The RPi already has a poly fuse between the USB power connector and the 5V rail. That fuse is bypassed when using the 5V pin to power the board. JP1 provides a 5V interconnect vertically between the RPis in each stack.

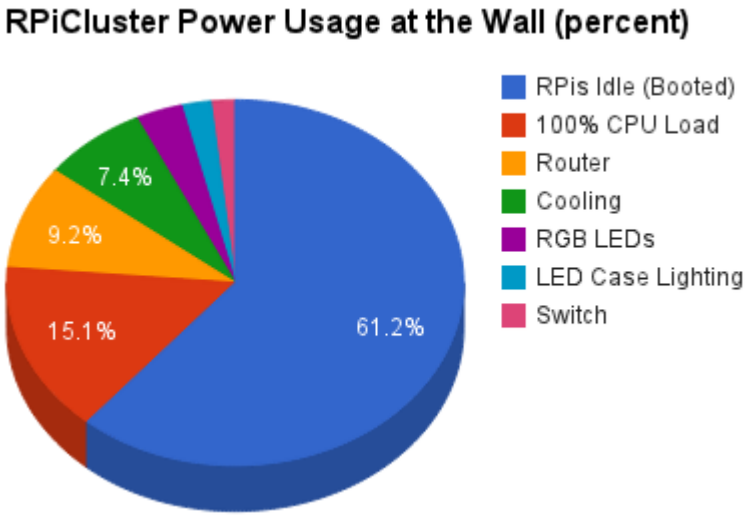
With the power being directly distributed via the Power/LED board, it was necessary to find a good source of 5V with sufficient amperage to drive the whole cluster. Each RPi draws about 400mA at 5V (2W), thus we needed a minimum of 13A of 5V (65W) (and more for overclocking). You could, of course, buy a dedicated high output 5V power supply, but a great option is to use a standard PC power supply. PC power supplies are already designed for high loads on their 5V rails, and they are relatively cheap. The 430W Thermaltake (430W combined output for 5V, 12V, etc) we selected is rated to provide 30A at 5V (150W) and cost $36. We opted to purchase two supplies to keep the overall load very low on each and allow for overclocking or future expansion.

Final Tweaks

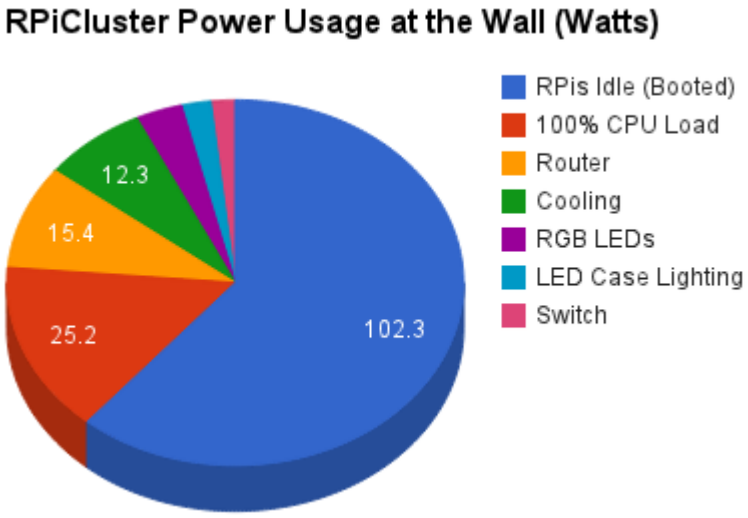
As mentioned earlier, the RPi processor supports overclocking. Another noteworthy feature is that the 512MB of RAM is shared by both GPU and CPU, and the amount of RAM reserved for the GPU is configurable. Both of these features are configurable through modification of parameters in /boot/config.txt. After some testing, the whole cluster was set to run in “Turbo Mode” which overclocks the ARM core to 1GHz and sets the other clocks (SDRAM etc..) to 500MHz. The result was a proportional increase in performance (~30%) and power draw. Since the cluster does not require a desktop environment, the GPU RAM reserve was reduced to 48MB. 48MB seems to be the smallest amount of RAM allowed for the GPU on the current version of Arch Linux ARM for the RPi (it wouldn’t boot with anything less).

Power Consumption

One of the great things about creating a cluster with ARM-based processors is low power consumption. As discussed earlier, each RPi uses about 2W of power (when running at 700MHz). A number of power measurements were made at the wall with the RPiCluster in various operational states. This allowed the individual component power usage to be determined without taking each item off-line to measure power draw individually. As I have overclocked the cluster to 1GHz core frequency and 500MHz for SDRAM etc., the power consumption is higher. Figures 9 (a) and (b) show the overall power use (as measured at the wall) in relative proportion (percent) and watts, respectively (overclocked to 1GHz).



(a) RPiCluster Component Power Useage (percent)



(b) RPiCluster Component Power Usage (Watts)
Figure 9: RPiCluster Power Consumption

As is seen in Figure 9 (b), there is about 15% difference in power consumption between idle and full CPU utilization. Additionally, about 12% of the total power used is attributed to the cooling portion of the cluster (four 120mm fans). The maximum total power usage of the cluster at the wall is about 167

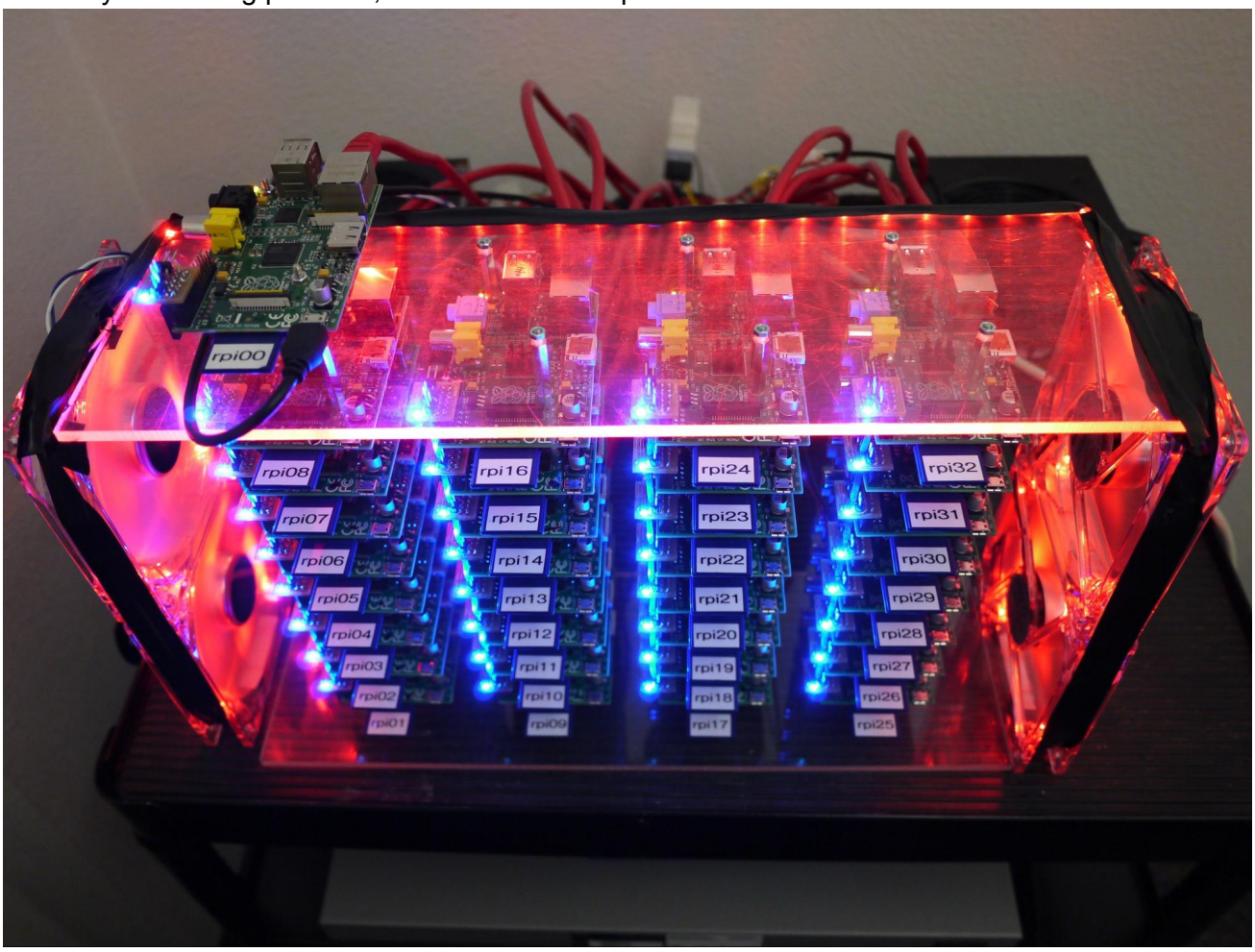
Watts. It is worth noting that active cooling is technically unnecessary. During my testing, the SoC temperature did not exceed 60° C while under 100% load, and the SoC is rated to operate up to 85° C. With cooling active, however, the SoC temperatures did not exceed 35° C. Cooling was retained because it seemed better for the SoC to operate at lower temperatures (and it looks nice).

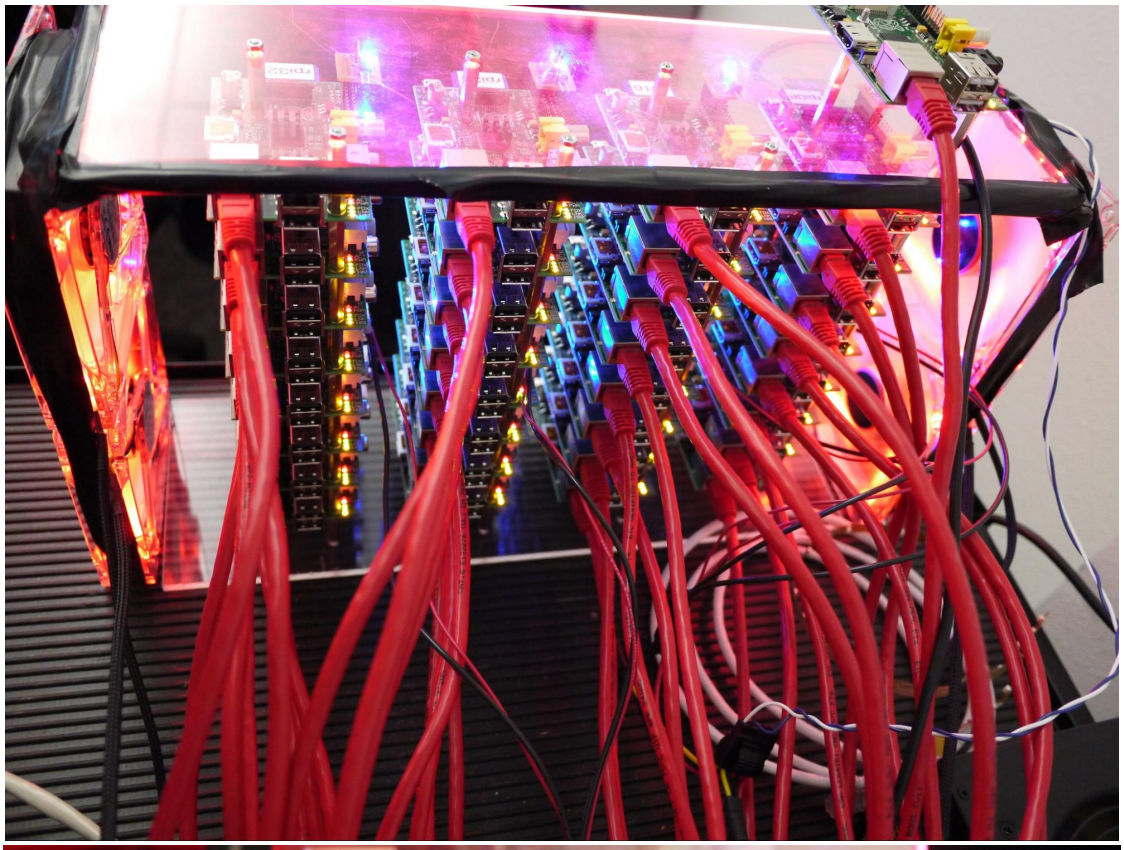
Conclusion

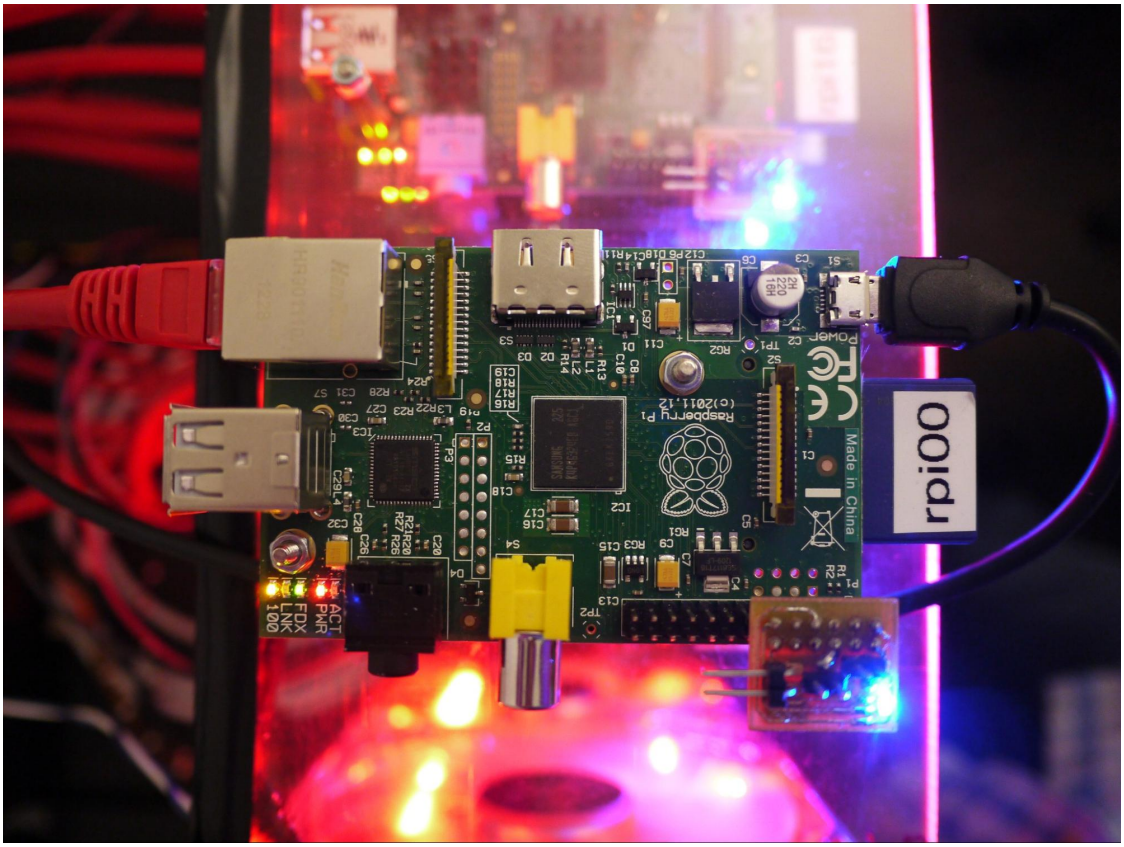
Overall, the RPiCluster has proved quite successful. Since completing the build I have moved my dissertation work exclusively to the RPiCluster. I have found performance perfectly acceptable for my simulation needs, and have had the luxury of customizing the cluster software to fit my requirements exactly.

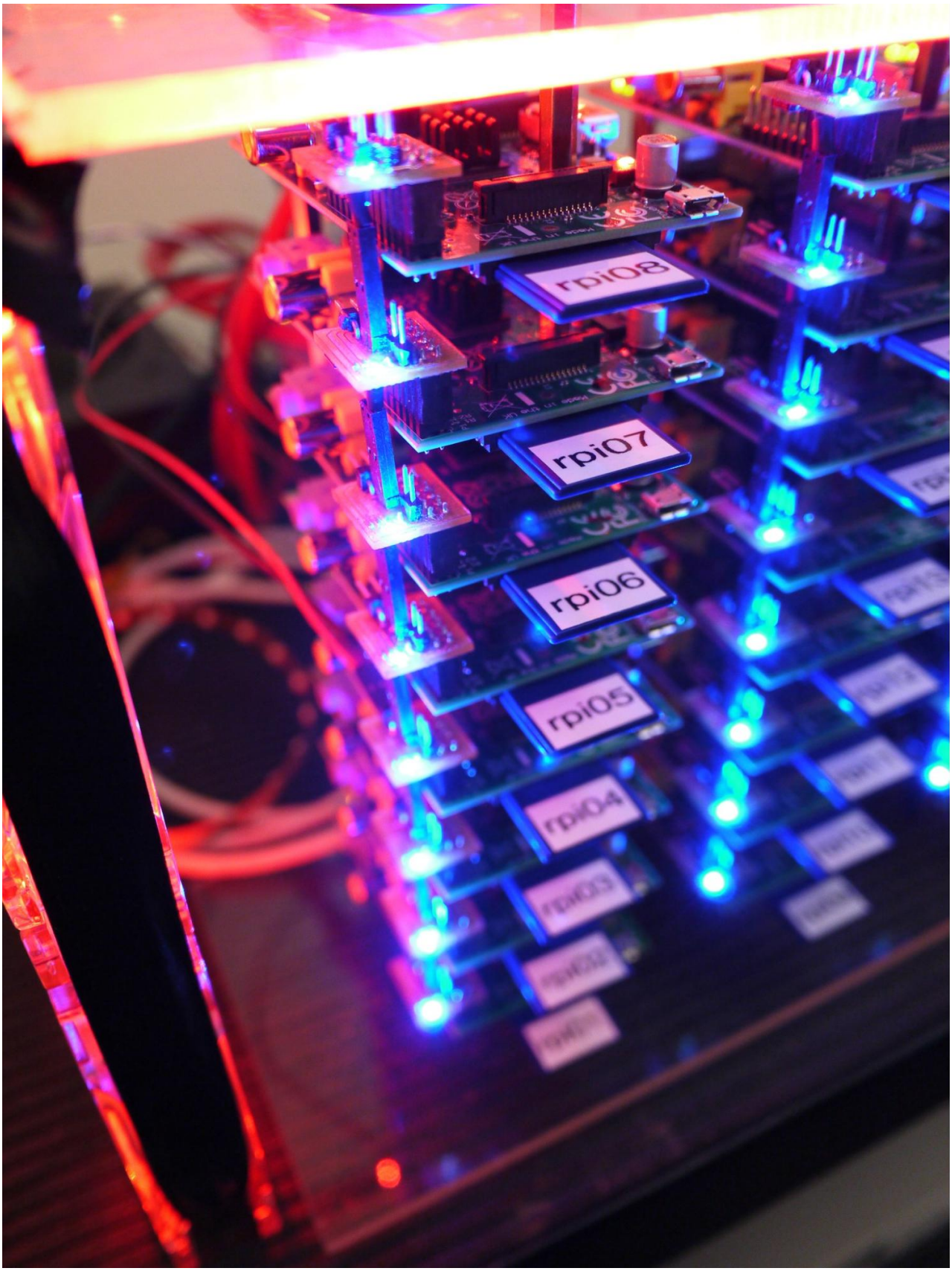
I would like to thank all the guys I work with in the Hartman Systems Integration Laboratory for all their help implementing the cluster design, especially Michael Pook, Vikram Patel, and Corey Warner. Additionally, this project would not have been possible without support from Dr. Sin Ming Loo and the Department of Electrical and Computer Engineering.

...and for your viewing pleasure, I have a few more photos:






rpi00
Made in China


rpi08
rpi07
rpi06
rpi05
rpi04