# INTRODUCTION TO PARALLEL PROCESSING WITH EIGHT NODE RASPBERRY PI CLUSTER

Greg Dorr, Drew Hagen, Bob Laskowski, Dr. Erik Steinmetz and Don Vo
Department of Computer Science
Augsburg College
707 21st Ave S, Minneapolis, MN 55454
dorrg@augsburg.edu, hagend@augsburg.edu,
laskowsr@augsburg.edu, steimee@augsburg.edu, vod@augsburg.edu

## Abstract

We built an eight node Raspberry Pi cluster computer which uses a distributed memory architecture. The primary goal of this paper is to demonstrate that Raspberry Pi's have the potential to be a cost effective educational tool to teach students about parallel processing. With the cluster we built, we ran experiments using the Message Passing Interface (MPI) standard as well as multiprocessing implemented in Python. Our experiments revolve around how efficiency in computing increases or decreases with different parallel processing techniques. We tested this by implementing a Monte Carlo calculation of Pi. We ran the experiments with eight nodes and one through four processes per node. For a comparison, we also ran the computations on a desktop computer. Our primary goal is to demonstrate how Raspberry Pis are an effective, yet comparatively cheap solution for learning about parallel processing.

## 1. Introduction

We became interested in parallel processing when our Association of Computing Machinery chapter toured the Minnesota Supercomputing Institute at the University of Minnesota. Walking through the rows of server racks filled with massive computing power and hearing about how the computers were working on problems such as predicting weather patterns and fluid mechanics fascinated us. This led us to exploring ways we could learn about parallel processing on a smaller and cheaper scale. What we were really interested in, as computer science students, was how to build a parallel processing environment, divide up difficult problems and compute results in an efficient manner. We had past experience with Raspberry Pi and discovered prior studies through the IEEE, detailing their use as small scale cluster computers (Abrahamsson [12] & Pfalzgraf [1]). We realized the Raspberry Pi would be our best option for building a parallel processing environment.

Once we built the cluster we sought out an easily parallelizable algorithm to implement. In our Numerical Mathematics and Computing class, we learned about Monte Carlo methods and decided a calculation of Pi using this technique would suit our experimental needs. We researched different ways to divide up the work which led us to MPI and Python's multiprocessing. In our experiments we wanted to determine which technique was the fastest. We also wanted to know how the computational power of eight Raspberry Pis compared to a modern desktop computer. Most of all, we wanted to learn as much as we could about parallel processing.

## 2. Building the Cluster

### 2.1. Supplies

The materials required to build a Raspberry Pi cluster are inexpensive compared to other cluster architectures. The parts we used and their costs can be seen in Table 1. We purchased everything from Amazon. These are the prices of the items we used as of March 15th, 2017 and do not include tax or shipping.

To begin we needed eight Raspberry Pis. We purchased the latest model, the Raspberry Pi 3 Model B, with 1 GB of DDR2 RAM and a 4x ARM Cortex-A53, 1.2GHz processor. Every Raspberry Pi needed a Micro USB card for the operating system and storage space. These we carefully selected to be speed class 10, also known as UHS speed class 1, to ensure the read/write speed of these would not be a limiting factor in the cluster's overall performance. Another factor we had to take into consideration was powering all of the Pis. Each Pi requires 2.5A at 5V from a micro-USB so we wanted to find a power source that provided sufficient power. We chose the Sabrent 60W USB charger which met all of our requirements. We purchased 1 ft. USB to MicroUSB cables to power the Pis.

| Part | Quantity | Individual Cost | Total Cost |
|---|---|---|---|
| Samsung 32GB Evo Plus UHS-1 microSDHC | 8 | $16.99 | $135.92 |
| Sabrent 60W 10-port USB Fast Charger | 1 | $32.99 | $32.99 |
| Netgear ProSAFE 16-Port 10/100 Desktop Switch (Model no. FS116) | 1 | $52.62 | $52.62 |
| Sabrent USB 2.0 A Male to Micro B 1 ft. Cables (6 pack) | 2 | $7.99 | $15.98 |
| Raspberry Pi 3, Model B, 1GB RAM | 8 | $35 | $280 |
| Mudder 8 Piece Black Aluminum Heatsink | 2 | $6.79 | $13.58 |
| | | **Total Cost:** | $531.09 |

Table 1: Cost Breakdown of Eight Node Pi Cluster

Now that we had the basics to get the Pis up and running individually, the next step was to connect them. We decided we would use a local area connection with ethernet for communication between Pis and Wi-Fi to connect the Pis to the internet. We believed the ethernet connection between the Pis would be faster and more stable for message passing associated with computations and the only internet access the Pis would need would be for secure shell (SSH), updates and downloading packages. We purchased a slightly larger switch than we needed in case we wanted to expand the cluster in the future. To connect the Pis to the switch we used ethernet cables we already had, cut to the exact length needed, so we did not include these in the price of the cluster. The last consideration we had was with the heat produced by the cluster. With heavy CPU loads during computation and the boards in close proximity, we were concerned about overheating. For this reason, we purchased the heatsinks and applied them to both the CPU and GPU of every Pi. The eight piece heatsink packages we purchased included four larger CPU heatsinks and four smaller GPU heatsinks, so this is why we needed two. As an additional precaution, we repurposed four fans from old computers and connected them to a separate power supply to blow over the Pis while they were running.

## 2.2 Hardware Setup

Like most projects, utility is only part of the problem. Our world demands that technology be both functional and aesthetically pleasing. Before beginning assembly we researched case options. In our computer science department we had access to a 3D printer. We created a case design and implemented the models in 3D Builder. The final product consisted of two stacks of four Pis connected at the middle. The whole cluster fits neatly on a square Lego base. All cables we as short as possible for cable management. We designed and printed labels to fit into the USB ports of the Pis so we could easily determine which Pi was which. This was very useful when one Pi needed to be reset occasionally. The final product can be seen in Figure 1.

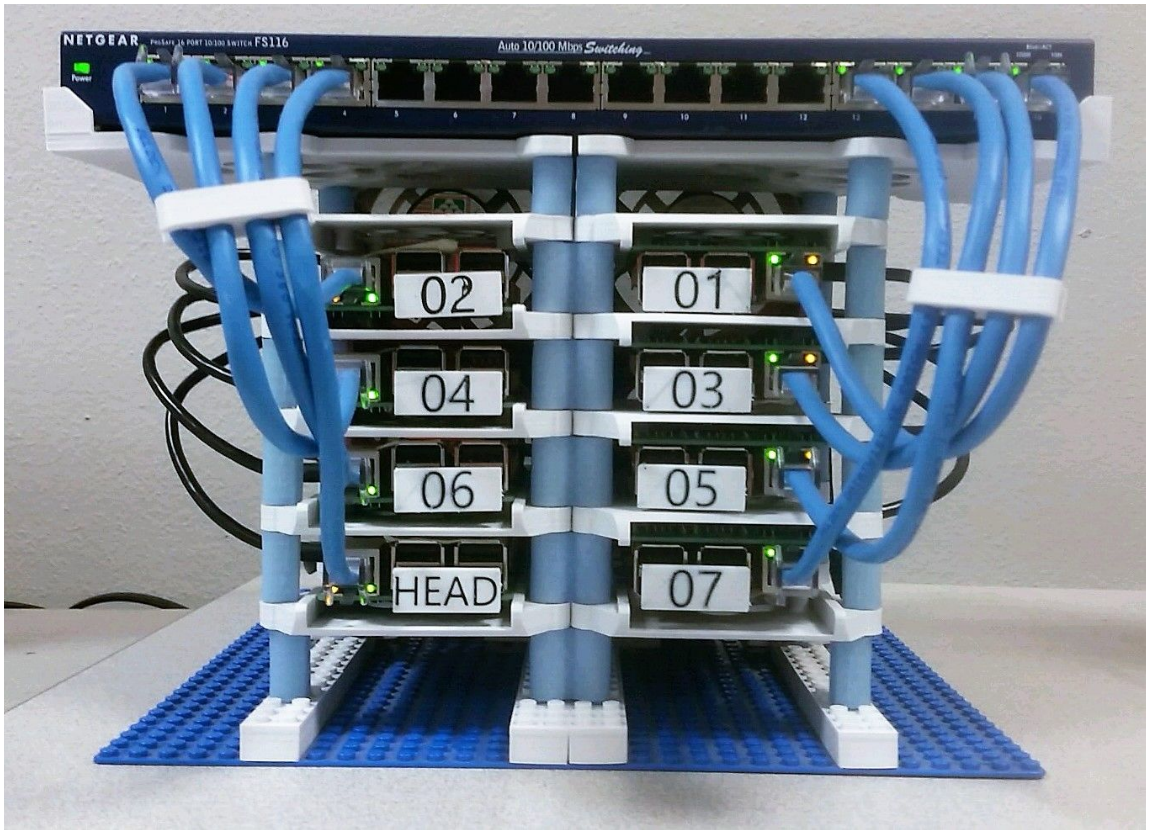


Figure 1: Raspberry Pi 8 Node Cluster

With the case done we moved on to assembling the hardware. This consisted of applying the heatsinks to the Pis, screwing the Pis into the case, connecting the power cables to the power block and connecting the ethernet cables to the switch. Now that we had all the hardware in place, we moved on to the software setup.

## 2.3 Software Setup

We chose to use Raspbian Jessie, a Debian based Linux operating system. From our past experiences, it seemed to be the most widely supported and stable. There were also many resources available for Raspbian, such as *Raspberry Pi Super Cluster* (Dennis [5]), which helped us throughout this process. We chose to designate one Pi as the head or master node and the remaining seven Pis as worker or slave nodes. On the master node we used Raspbian Jessie with Pixel, the version of the operating system with a graphical user interface. We thought having the GUI available would make some of the configuration easier. On the slave nodes we used Raspbian Jessie Lite, the version with only a command line interface. We chose this to reduce the overhead and because we would only ever be interacting with the slave nodes via the command line, primarily over SSH.

Instead of setting up all eight nodes individually, we set up the GUI operating system for the master node on one SD card and one command line operating system for the slave nodes. We then cloned the slave node configuration to the other six SD cards. Beginning with the master node, we downloaded the official Raspbian Jessie with Pixel from the official Raspberry Pi website's download page (Download [6]). Then we followed the instructions on the official Raspberry Pi documentations pages (Installing [8]) to put the operating system image on the SD card. Once the SD card was loaded with the image, we booted up the first Pi which was to become our master node. We ran *sudo raspi-config* to configure basic settings. After rebooting we followed the same process for the each slave node, only with the command line version of the operating system. We then used Win32DiskImager to read the image file from the slave node SD card and write it other the other six slave SD cards. We changed the host names to the appropriate ubXX for each slave node. Now we had the operating system up and running on all of the Pis and moved on to setting up the networking.

## 2.4 Network Configuration

The network configuration was the most complicated part of the setup process. The goal of this project was to have the master node connected to the internet via the Wi-Fi interface, wlan0, and connected to each of the slave nodes via ethernet, eth0. We wanted the slave nodes to be accessible via SSH from the master node but on their own subnet and not directly accessible. We also wanted the slave nodes to be able to get out to the internet for updates and downloading new packages as necessary. This led us to create the network topology as seen in Figure 2. Here the master node is acting as an intermediary between the slave nodes and the internet. This topology requires the master node to receive packets from the slave node and forward them out to the internet. To accomplish this we had to set up network address translation (NAT) on the master node. NAT essentially remaps one internet protocol (IP) address space to another. In our case, it maps packets on the cluster subnet to the router's subnet. We included a router in the network topology because we wanted the master node to have a static IP address. We did not have access to the school network's router and getting the school's IT department to assign a static IP for the master node's MAC address proved difficult. Almost every time we turned off the Pis for an extended period of time their IP address would change and we would have to hook a monitor up to the master node to figure out the new IP address. The router provided a workaround in that it always stayed online and therefore the IP address did not change. We assigned the master node a static IP on our router and setup port forwarding for port 22. Thus we essentially created a static IP address we could access from anywhere on campus via SSH.

To actually implement the network topology concept we created proved challenging. We began by connecting the master node to the router. This can be done via the command line or GUI, since we installed the full version of Raspbian on the master node. We chose to configure Wi-Fi via the command line following the guide found on the official Raspberry Pi documentation pages (Setting [14]). Now the master node can access the internet via the router. To configure the static IP and port forwarding for SSH (port 22), consult the specific instructions for your router, as they are all different. We chose to make our router network 192.168.11.X and assigned the router the IP address of 192.168.11.1 and the master node 192.168.11.101. The static IP for the master node was assigned on the router side, so no additional configuration needed to be done on the Pi for this. Note that all IP address range choices are arbitrary and can be changed as desired. All that matters is that the Wi-Fi and LAN networks have different IP address ranges.

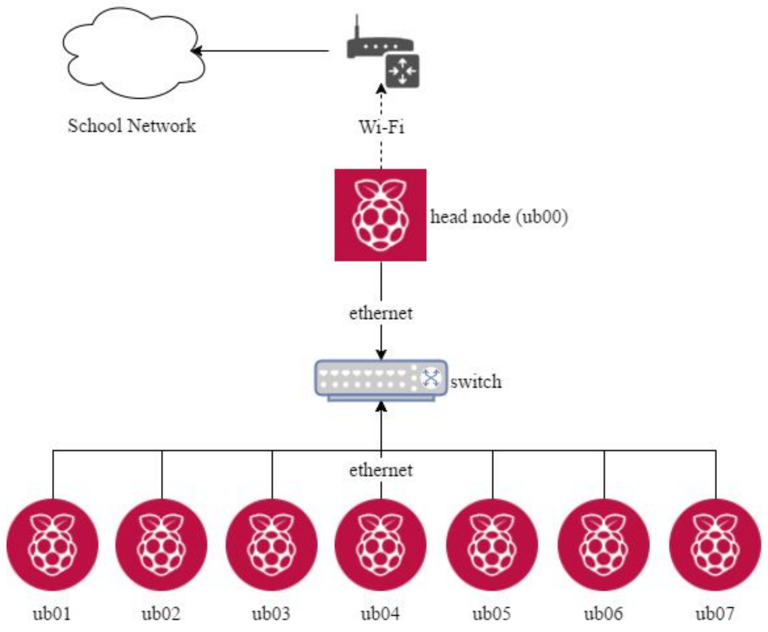


Figure 2: Cluster Network Topology

The next step was to configure all of the slave nodes to be on the same subnet as the master node, not yet worrying about the NAT. We started by first configuring the eth0 interface of the master node to a static IP address. We chose to make the subnet for inter-Pi communication 192.168.1.X with the master node acting as the default gateway for the slave nodes.

Now that the master node had a static IP address, we needed to configure the master node to act as a dynamic host configuration protocol (DHCP) server to hand out IP addresses to the slave nodes. We used the "Setting up a Raspberry Pi as a DHCP Server" guide found on NovelDevices.co.uk (Setting [13]). We chose to do this to make expanding the cluster in the future easier. If we want to add another node, we can simply add a line to the DHCP configuration file. To make each of the slave nodes get IP addresses from the DHCP server, we modified the */etc/network/interfaces* to use DHCP for the eth0 interface on every node. Now, all of the slave nodes get IP addresses from the master node on the inter-cluster subnet. We could now SSH into any of the slave nodes from the master node but the slave nodes could not yet access the internet.

To allow the slave nodes to access the internet for updates and installation of new packages, the final step was to configure the NAT on the master node. We tried many different things to get this working and eventually eventually found this guide on

Adafruit's website (Ada [2]). We used the section on page 21, "Configure Network Address Translation." Once we got it working, the NAT essentially tells the master node to forward packets it receives from the slave nodes over the ethernet interface to the wireless interface and out to the router. The only network configuration remaining was to set up RSA keys.

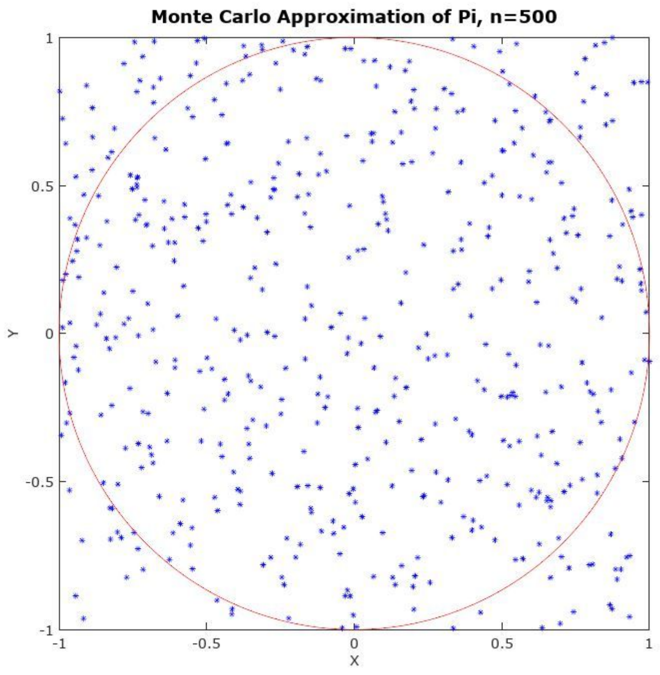
Setting up RSA keys allows the master node to SSH into the slave nodes without being prompted for a password every time. This will be essential in our parallel processing applications as the processes require a password-less SSH to compute its portion. To make the RSA key generation easier, we first set up the Pi's *hosts* file. In this file we mapped each Pi's hostname to its IP address to make accessing the different nodes easier. To create a RSA key, we entered the command *ssh-keygen*. We created a key for each of the nodes in your cluster. To copy the keys, we entered the following command on each node: *ssh-copy-id -i [key location] [user]@machine*. It required us to enter the *[user]@machine*'s password. This only needs to be done from the master node to slave nodes and vice versa, but not between every node, since the slave nodes do not need to communicate with each other.

## 3. Running Experiments

### 3.1 Monte Carlo Method

Given the necessity for a test that would highlight the parallel capabilities of the cluster, an easily divisible algorithm was chosen. In this task, "easily divisible" means that the algorithm has little to no functionality that must be processed in a single thread. In order to meet this criteria, the steps of an algorithm cannot depend on previously computed steps. In our Numerical Mathematics and Computing class, taught by Dr. Pavel Belik, we had recently studied Monte Carlo Methods. Wolfram Mathworld (Monte [9]) defines Monte Carlo Methods as "any method which solves a problem by generating suitable random numbers and observing that fraction of the numbers obeying some property or properties." Monte Carlo Methods can be applied to any number of problems and in general they satisfy the criteria of an embarrassingly parallel algorithm.

One particular application of a Monte Carlo Method we studied was an approximation of Pi. Pi ($\pi$) is an irrational mathematical constant describing the ratio of a circle's circumference to its diameter. Pi can be approximated using a Monte Carlo Method. To do so, we start with a circle of radius 1. The equation of such a circle is $x^2 + y^2 = 1$. Using the formula for the area of a circle, $A = \pi r^2$, we can easily determine the area of this circle is $A = \pi * (1)^2 = \pi * 1 = \pi$. We then proceed to enclose the circle in the tightest square possible, where one side of the square is equal to the diameter of the circle, in this case 2. We then approximate the area of the circle by generating random ($x$, $y$) points with both $x$ and $y$ in the range [-1,1]. For each point generated, we check to see if it is inside or on the circle. To check this we see if the point satisfies the inequality $x^2 + y^2 \leq 1$. Every time a point falls within or on the circle, we increment a counter. After the specified random number of points have been generated, we calculate the ratio of points inside or on the circle to total points generated and multiply this ratio by the area of the square. An example can be seen in Figure 3 using 500 points. There, 403 of the 500 points fell inside or on the circle. We approximate Pi by calculating the ratio and multiplying by the area of the square, in this case 4. The more points generated, the more accurate the approximation of Pi will be. For more information on how much accuracy is gained by use of additional points see Northeastern University's website (Feiguin). It has been proven that the error of a simulation is on the order of $1/\sqrt{n}$ where $n$ is the number of points used. Thus, as we increase the number of points we will increase the accuracy of our estimation of Pi but we will see diminished returns as $n$ grows large.



$$\frac{403}{500} * 4 = 3.224$$

Figure 3: Monte Carlo Approximation of Pi with 500 Points

## 3.2 Bash Scripting

We created several different Bash scripts to aid us while trying to work with eight computers at once. One of the most important scripts was our 'perkins' script, which implemented dynamic virtual terminal manager (DVTM). DVTM allowed us to SSH into all of the nodes at the same time and interact with them. Each connected node has its own subwindow in the terminal. This script allowed us to install new software or create files on all the Pis at once. While this allows us to run the same computations on each node at the same time, we could not find an easy way of collecting all of the data back to the master node.

In order to test parallel processing we needed a program that allowed us to SSH into each node at the same time, and have a way to collect results. We found a Python program called Parallel SSH (PSSH). This program allowed us to run the same computation on each node at the same time using a single command. PSSH requires a host file, username, timeout number, and a command to execute. The host file contains a list of all the hosts' IP addresses. The computed results are returned to the master node where they can be easily read and interpreted.

Another useful script we used was our copy script. This script would copy files from the master node to all the slave nodes. At its core, it uses Secure Copy (SCP). SCP on its own allows a user to copy to a remote host via SSH. We set it up so it could send a file or directory to all of the remote hosts. We needed this script because in order to run a PSSH or MPI  program, each node needed the exact same code file to execute.

We needed to determine the execution time of the scripts. We started out by using the *time* command built into to Linux. The *time* command works by clocking how long it takes for a command to execute. We ran into some issues with a small *n* for the Monte Carlo approximation of Pi. Any *n* below 100,000 appeared to give inaccurate timing because they all took between 0.6 and 0.7 seconds, not necessarily increasing as *n* grew. We believed the overhead of the starting PSSH took longer than the actual calculation. To test our hypothesis, we did a timing test of PSSH. This command took anywhere between 0.4 to 0.6 seconds to establish a connection. This explained the variability we saw with some results. It appears it took longer to establish a connection that it did for the command to execute with small values of *n*.

To calculate Pi, we created a script that used *time* and PSSH.  It took in one argument, the number times we wanted the Python code to loop. The first thing the script did was divide the argument by eight. Each node only has to calculate ⅛ the total number in get the result. We then called PSSH and passed in the Python file we wanted executed on each node. After each node finished its calculation, it returned its answer back to the master node. In order to pass the number we want into the script, we used regular expressions through *grep*. All of the results were returned in an array which we iterated through to add all of the results together. While trying to do the final calculation we ran into some limitations of bash. Bash only has two types of variables, integers and strings. To overcome this, we found a program called Bash Calculator (BC). With BC we were able to do floating point division and get our final answer.

## 3.3 Message Passing Interface

The Message Passing Interface (MPI) is a standard developed and agreed upon by many organizations and researchers, including Lawrence Livermore National Laboratory

(Barney [3]). The standard's primary function is to solve parallel processing problems primarily for high performance computing. It has been implemented in various languages including C, Fortran, C++ and Python. Messaging passing in itself is the sharing of data between threads or processes. We chose to use the Python implementation because we were the most familiar with that language and the installation process was the simplest. To set MPI up, we used a guide from SciPy.org (Dalcin [4]). In a very general sense, MPI works by executing the same script on every node. The node that began the execution becomes the master and has rank zero. The remaining slave nodes send messages to the communication world they exist in to determine their rank, which will all be integers greater than zero. Then the script proceeds on each node, usually checking to see whether the node is a master or slave, and executing the appropriate code.

In our implementation of a Monte Carlo approximation of Pi, we define a function *calculate.* This function generates *n* pseudo-random number pairs in the range [-1,1] and checks to see if the points generated are in or on the circle, as described in Section 4.1. Instead of actually calculating Pi, this function returns the number of points that landed within the circle because the master node will perform the calculation later. After the function definition every node establishes its rank by sending a message to the master node. The entire script takes one command line argument, which is the number of random points to generate. The script divides the total number of points by the number of nodes used and assigns each node a number of points. Essentially, if we used all the Pis in the cluster, each Pi would be doing ⅛ of the work. We then called the calculate function with the appropriate *n* on each node. Each node then checks to see if it is the master node and if it is it waits to receive the results from all the other nodes. If it is a slave node, it sends its results to the master node. The master node then computes the ratio and multiples by the area and prints the result.

## 3.4 Python Multiprocessing

We found that our MPI tests only used 25% of each Pis computational power available because only one of the four cores was used. In order to fully utilize the computational resources available, we needed to implement some sort of multiprocessing. We began by looking at multi-threading. Multi-threading gave us some eccentric results. It was faster to run a single thread than it was to run two. The more threads we added after that point always ran slightly faster than the previous amount. This only worked up until a point. It seemed to us that the overhead of creating a new thread was higher than just running it single processed. It also was still only using 25% of the CPU. From research we did on this problem we believe this is because of the Global Interpreter Lock (GIL). The Python interpreter holds a global lock, and no Python code can be executed without holding that lock. Therefore, no two Python instructions can execute at the same time and threading does not speed up performance.

Since multithreading was not our answer we decided to try multiprocessing. The difference between multiprocessing and multithreading is that threading shares the same memory space as what started them, while processes have separate memory locations. Processes also run independent of the program that spawned it. Spawning four processes per node allowed us to fully utilize the each node's four cores. This gave us the results we wanted.

We implemented multi-processing through Python's standard *multiprocessing* library. It works in a similar way to threading, but uses subprocesses instead of threads to avoid the GIL. Once the Python script was executed on the remote node, it spawned four processes to do all of that node's calculations. This allowed us to take advantages of the four cores in each Pi. Each node only has to calculate ⅛ the original value passed into the main script. Since we broke each node into four processes it now only has to do ¼ of the divided work. Overall each process only has to execute 1⁄32 of the original number of points, giving much faster run times.

The difference between multiprocessing and MPI lies in their intended uses. MPI is specifically designed to run on multiple machines. It passes messages between machines in an efficient manner (MPI [10]). Multiprocessing is designed to run on a single machine, as described in the application programming interface (API) associated with it (Multiprocessing [11]). For this reason, we had to use PSSH to make multiprocessing work with the cluster. Since the only messages we had to pass between nodes were to execute the script and return the result, multiprocessing worked just fine. We believe that if more message passing was required within the program's execution, MPI would have been a better choice.

# 4. Results

## 4.1 Pi Cluster

To test the efficiency of the cluster, we ran both the MPI and the multiprocessing scripts with one through four processes per node. Each test used all eight nodes. Our tests consisted of values of *n* for the Monte Carlo estimation of Pi from ten to one billion, increasing by a factor of ten every time. We used nine different values of *n* with four different numbers of processes for a total of 36 tests per method. To obtain more accurate results, we did three trials of every test and used the average of the three results. This resulted in 108 tests per method.

Our goal was to determine if one method was more efficient than the other. The results in Table 2 show the average computation time of all the tests run for a particular number of processes per node. We can see that the average run times for each is about the same, with an average increase in efficiency between processes of approximately 35%. This can be seen graphically in Figure 4. We believe the results are very similar because of the small number of messages being passed between machines. The only messages passed between the nodes are at execution time and at the end of the script to return the result. There are no intermediate results being passed or inter-node communication. Thus, MPI and our PSSH multiprocessing script are essentially doing the same thing, so it makes sense the results are so similar.

| Cluster Multiprocess, Average Time | | Cluster MPI, Average Time | |
|---|---|---|---|
| Processes | Time(Seconds) | Processes | Time(Seconds) |
| 1 | 78.61 | 1 | 98.41 |
| 2 | 47.35 | 2 | 49.56 |
| 3 | 32.01 | 3 | 33.07 |
| 4 | 24.39 | 4 | 24.85 |

Table 2: Cluster Multiprocess and MPI Average Times

## 4.2 Desktop

The test we ran on the desktop computer was similar to those we ran on the cluster. Our goal was to obtain the fastest execution results possible. The method we used was similar to the multiprocessing ran on the cluster, but no PSSH was required for the desktop since it is a single machine. We consider the PSSH process to be part of the overhead of a clustered environment. The computer we used has an Intel i7-2600K (8) @ 3.4GHz processor with 32GB of RAM. We ran all tests using Bash on Ubuntu on Windows. This allowed us to have a similar environment to the cluster for testing and required minimal modification of our code. As before, we used the Monte Carlo approximation of Pi with values of *n* from ten to one billion increasing by a factor of ten each time. The main difference with the desktop tests was that, since this CPU has eight cores, we tested with one through eight processes as opposed to one through four. As before, we ran three sets of each tests and used the average as our final value to increase accuracy. There were nine tests per number of cores used, each run three times for a total of 216 tests.

The results from the desktop computer were as expected, slow at first and quick as the number of processes increased. We expected this because the computer has four physical cores and four virtual cores, for a total of eight. In Figure 4 and Table 4 we can see that there was a significant increase when going from one to four processes, but not much when going from five through eight. The average amount of time it took to execute decreased by 34% on average as we increased from one to four processes and only 1% on average as we moved from five to eight. The time actually increased between seven and eight processes. What we found very interesting that very little return was seen after four processes. Each extra core utilized after that point had very little return. We believe this is due to the sharing of the physical cores. Another very interesting result was that, on both the cluster and the desktop, there was approximately a 35% decrease in execution time as a physical core was added. We were surprised this was consistent between machines.

As we expected, the desktop was faster. It has a much faster processor and four times the amount of RAM. We were surprised at how closely the cluster compared. When comparing one through four processes, the desktop was 22% faster on average. The fastest execution time on the cluster was only 29% slower than the fastest on the desktop. This shows the power and importance of cluster computing, as compared to a single system, as demand for computing power grows. Even with Raspberry Pis, the power of a modern desktop can almost be matched for a fraction of the price.

| Desktop, Average Time | |
|---|---|
| **Processes** | **Time(Seconds)** |
| 1 | 74.74 |
| 2 | 38.20 |
| 3 | 26.55 |
| 4 | 20.35 |
| 5 | 19.63 |
| 6 | 19.25 |
| 7 | 18.78 |
| 8 | 18.89 |

Table 3: Desktop Average Time

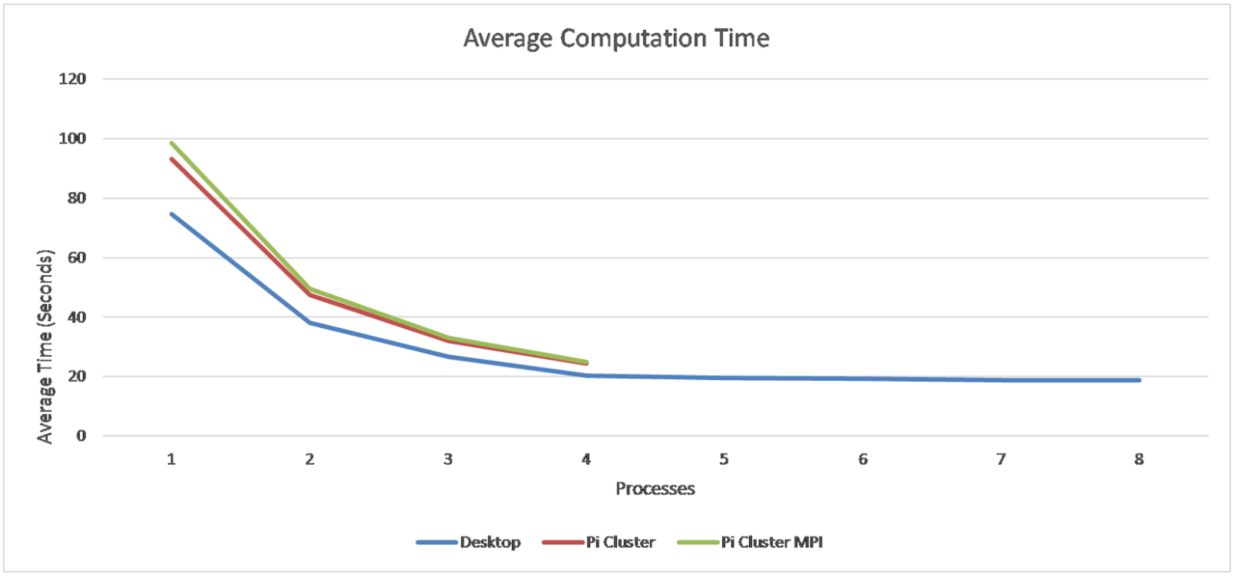


Figure 4: Average Computation Time

## 5. Conclusion

The primary goal we hoped to accomplish with building this cluster was to learn more about parallel processing, which we definitely accomplished. We knew next to nothing on the subject when we began and now have a decent understandings of the basics and why it is important. We implemented two different forms of parallel processing, multiprocessing and MPI. We learned that, under the conditions we tested with, their performance was comparable. In comparison to a desktop computer, the cluster was slightly slower. The cluster only had eight nodes and it would be interesting to see how adding more nodes would affect the results.

Throughout the process, we encountered many issues. One of the first issues we had was setting up the NAT for the nodes. We initially could not find a clear and concise guide on how to set it up. As we went through trial, error, and research, we found a clear guide that helped us figure it out. We also had issues setting up MPI for our cluster. We initially installed it in C, but the installation process was very long and complicated. We also ran into issues trying to implement our basic function in C, which led us to switching to Python. We found this version much easier to use and that it suited our needs.

If we were to further our research, we could create more computationally heavy algorithms to test on. Some of these algorithms could be solving a rubik's cube or the game Go Testing more computationally heavy algorithms could confirm or refute our findings. We could also do the same tests on another cluster to see if better hardware would decrease the runtime.

## 6. References

1) A. M. Pfalzgraf and J. A. Driscoll, "A low-cost computer cluster for high-performance computing education," *IEEE International Conference on Electro/Information Technology*, Milwaukee, WI, 2014, pp. 362-366. doi: 10.1109/EIT.2014.6871791 http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6871791&isnumber=6871745
2) Ada, Lady. "Setting up a Raspberry Pi as a WiFi Access Point." (n.d.): n. pag. *Adafruit*. Adafruit, 5 Dec. 2016. Web. 28 Nov. 2016.
3) Barney, Blaise. "Message Passing Interface (MPI)." *Message Passing Interface (MPI)*. Lawrence Livermore National Laboratory, 2 Feb. 2017. Web. 14 Jan. 2017.
4) Dalcin, Lisandro. "Installation of MPI4PY." *Installation — MPI for Python V1.3 Documentation*. SciPy, n.d. Web. 15 Jan. 2017.
5) Dennis, Andrew K. *Raspberry Pi Super Cluster*. Birmingham: Packt, 2013. Print.
6) "Download Raspbian for Raspberry Pi." *Raspberry Pi*. Raspberry Pi, n.d. Web. 16 Nov. 2016.
7) Feiguin, Adrian E. "Monte Carlo Error Analysis." *Monte Carlo Error Analysis*. Northeastern University, 4 Nov. 2009. Web. 24 Feb. 2017.
8) "Installing Operating System Images Using Windows." *Installing Operating System Images Using Windows - Raspberry Pi Documentation*. Raspberry Pi, n.d. Web. 16 Nov. 2017.

9) "Monte Carlo Method." *Monte Carlo Method -- from Wolfram MathWorld*. Wolfram, n.d. Web. 4 Feb. 2017.
10) "MPI for Python." Python Hosted. N.p., 18 Oct. 2015. Web. 15 Feb. 2017. <https://pythonhosted.org/mpi4py/usrman/>.
11) "Multiprocessing - Process-based Parallelism." Python 3.5.3 Documentation. Python Software Foundation, n.d. Web. 15 Feb. 2017.
12) P. Abrahamsson *et al*., "Affordable and Energy-Efficient Cloud Computing Clusters: The Bolzano Raspberry Pi Cloud Cluster Experiment," *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, Bristol, 2013, pp. 170-175. doi: 10.1109/CloudCom.2013.121 http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6735414&isnumber=6735374
13) "Setting up a Raspberry Pi as a DHCP Server." *Noveldevices.co.uk*. Novel Devices, n.d. Web. 28 Nov. 2016.
14) "Setting WiFi up via the Command Line." *Setting WiFi up via the Command Line - Raspberry Pi Documentation*. Raspberry Pi, n.d. Web. 20 Nov. 2016.