Tiny Data with Raspberry Pi's: An Exploration of Low-Cost, Low-Energy MapReduce Clusters

Kuan-Lin Chen[1], Jeremy Feinstein[1] and Brian Kutsop[1]

***Abstract*— As data centers have increased in size, there has been a push to create clusters out of cheaper, more affordable commodity parts that can easily be replaced upon failure, and that create more affordable data centers overall. However, such large clusters are still outside of feasibility for individuals and small businesses. It is a worthwhile exercise to see if much smaller clusters could be created for such applications, and to compare their performance / price measure to that of traditional datacenters. In our case, we explored creating such a cluster with Raspberry Pis which are $35 credit-card-sized, single-board computers. More specifically, we built a distributed data processing architecture in Python that runs on a cluster of four Raspberry Pi's and closely resembles Google's MapReduce architecture. In order to profile the performance of the system, we wrote several example MapReduce jobs such as counting words, calculating baseball statistics, and counting n-gram frequency for text documents.**

This project will be extended to create a type of plug-and-complete networking project that can be used to teach and introduce networking concepts in one of Cornell's primary systems class: CS 3410 or CS 4410. This will also include coming up with a complete instruction set and set of guidelines to support students completing the project. During the implementation process, all members of our team learned previously unknown skills, including how to create a cluster, programming the infrastructure that lies under a single, physical switch, and analyzing system throughput.

I. INTRODUCTION

A. *Raspberry Pi*

The Raspberry Pi is a credit-card-sized, single-board computer whose initial purpose was to enable students learning computer science (really at any education level) to have more hands on experience. Since their launch, they have become incredibly popular among in the computer science and maker movement communities and have been used to make everything from home automation systems to full-fledged laptops to monitoring devices. For this project, we were generously given four Raspberry Pis to create a little computing cluster, in other words a mock data center. We chose to use the Raspberry Pi platform because of their price, availability, and support community. At $35, one would be hard pressed to find a cheaper, more established solution for low cost, hands on computing. Raspberry Pis are also readily available online and Professor Weatherspoon even had the four that he lent us lying around in his office for opportunities such as this one. The community surrounding the Raspberry Pi also proved to be invaluable, offering an unparalleled amount of online support during the installation of our cluster. Additionally, other members of the community had set-up similar clusters and we readily pulled from their own experiences, shared online through forums and blog posts.

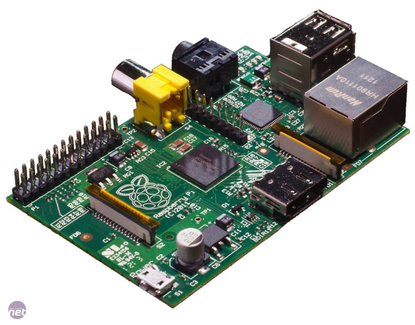


Fig. 1. A Raspberry Pi model B

B. *MapReduce*

MapReduce is a programming model popularized by Google in the mid-2000s and used for processing large amounts of data. It is inspired by the functional programming map and reduce patterns and is designed to be easily parallelizable, allowing programmers with little to no knowledge of parallel or distributed computing to write jobs that run on tens to thousands of machines in the cloud. The interface for a job (the term for a MapReduce program that is run on the system) itself requires two functions, which we will call map and reduce. Their interfaces have the following types:

- map (k1, v1) $\rightarrow$ list(k2, v2)
- reduce (k2, list(v2)) $\rightarrow$ list(v2)

An example of a word-counting job in pseudo-code where the input is a text document and each initial value is a line of text would look as follows:

```
map (key, line):
    for word in line.split( ):
    yield (word, 1)

reduce (word, values):
    return sum(values)
```

The overall MapReduce architecture can be seen in Figure 2. On the left hand side is the initial data set split into several pieces. Each piece is sent to one of several parallel workers who apply the map function each line or atom of the piece. The result is then written locally to disk on the worker. From there, the results are then piped to another worker who applies the reduce function to the intermediate

[1] Department of Computer Science, Cornell Univerist

data. Once this is done, the output is written to a file, usually on a distributed file system. There is a single master that coordinates the data flow and a user program that invokes the master with the MapReduce job to run and what data to run it on.

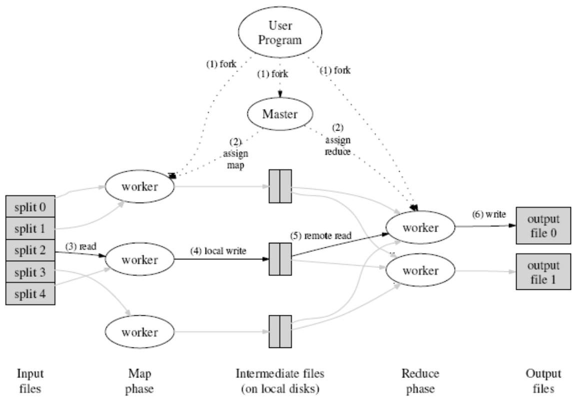


Fig. 2. The Google MapReduce architecture

Googles own implementation of MapReduce is closed source, but several open source projects have emerged as the programming model was popularized, such as Hadoop. We followed the architecture laid out in Googles MapReduce white paper closely, but took some liberties wherever we saw fit.

II. MOTIVATION

Our main motivation for this project is to push the limits of low-cost, commodity computing. With the rise of commodity server-based data centers, we saw the next frontier as constructing these data centers out of just about the cheapest boards commercially available. This would lower the barrier to entry for cluster computing and make it a reality that can be realized by anybody in their own home (although one could argue that this is already on the rise via advancements in GPU programming accessibility). We also want to profile this type of cluster and compare it to other solutions available today, specifically Cornells own Computer Science Undergraduate Lab (CSUG) servers. Furthermore, our project goes hand-in-hand with the Raspberry Pi Foundations initial mission of making physical computing more accessible to students learning computer science at all levels. Much in that way, we aim to make cluster computing more accessible to computer science students at Cornell University by turning our project into a project for an undergraduate course, where students will complete the Distributed File System and MapReduce and implementations presented here.

III. DESIGN

A. System Architecture

The entire system can be broken down into three parts: the client, the master, and the followers. The client is what a user would speak to the system through. It communicates directly with the master machine to interact with the system. We provide this client library so that the user never has to worry about correctly communicating with the master and can do so through very simple calls in the shell, which will be shown later. The master and follower parts are run on computers in the cluster Communication between the different parts is done through sockets, much like the labs earlier in the semester. We designed our own chat-like protocol through which the different parts can talk to each other autonomously. In this way, each part of the system runs its own chat server and sends messages to other parts chat servers. Communication between the client and master is initiated by the user, whereas communication between the master and followers happens completely autonomously and is initiated after client calls to master.

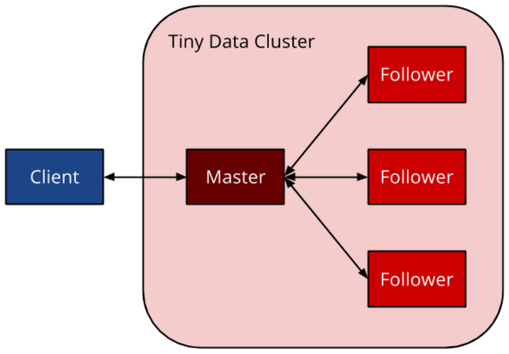


Fig. 3. The layout of our cluster

B. Distributed File System

The distributed file system exposes a simple interface with commands such as: ls, mkdir, rm, mv, cat, and upload. We will show the flow for each of these commands below. The state of the file system is stored in memory on the master and is currently not backed up on disk. Each file in the file system is composed of chunks, or pieces of the file which contain some number of lines of data. Each chunk is assigned a UUID which is then logged as belonging to the file. These chunks are stored on disk on the followers in a dedicated directory. Followers do not need to know which file a chunk belongs to; they are solely used to store, read, and perform operations on chunks when requested by the master. Communication flows for the distributed file system commands can be seen in Figures 4, 5, and 6.

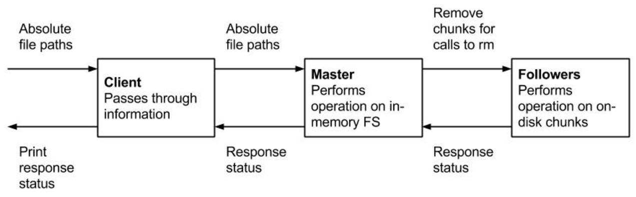

Absolute file paths
Client
Passes through information
Print response status
Absolute file paths
Master
Performs operation on in-memory FS
Response status
Remove chunks for calls to rm
Followers
Performs operation on on-disk chunks
Response status

Fig. 4. Communication flow for DFS commands: ls, rm, mkdir, mv

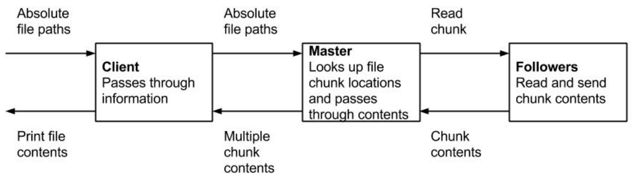


Fig. 5. Communication flow for DFS command cat

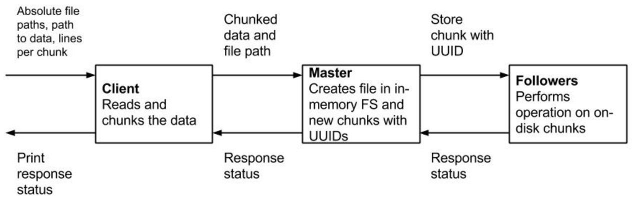

Absolute file paths, path to data, lines per chunk
Print response status
Client
Reads and chunks the data
Chunked data and file path
Response status
Master
Creates file in in-memory FS and new chunks with UUIDs
Store chunk with UUID
Response status
Followers
Performs operation on on-disk chunks

Fig. 6. Communication flow for DFS command upload

C. *MapReduce*

The MapReduce system is accessed via a command similar to those of the DFS, which we call map_reduce. As arguments, it takes the path in the DFS of the data set (it must already be uploaded), the path in the DFS to store the results, and a MapReduce job file. The job file is a normal python file which can use all of the standard Python libraries, but not third party ones like numpy (although this could easily be changed by installing those libraries on the master and followers). The file must define two functions, map and reduce, and may also implement an optional third function, combine. Map takes a string which is a line from the data set and returns a list of keys and values. Combine takes a key and a list of values associated with that key from the map phase and returns a new value. Reduce takes a key and a list of values associated with that key from either the map or combine phases and returns a new value. This closely follows the Google MapReduce interface as described above. Map and combine are executed on the followers in separate threads (not the chat server thread), while reduce is executed on the master in a separate thread. The communication flow for MapReduce can be seen in Figure 7.

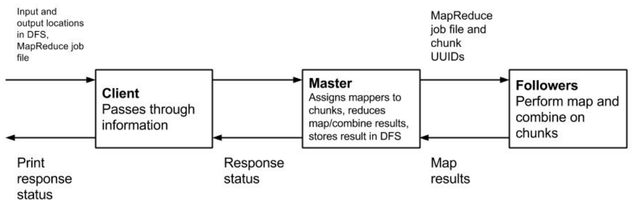

Input and output locations in DFS, MapReduce job file
Print response status
Client
Passes through information
Response status
Master
Assigns mappers to chunks, reduces map/combine results, stores result in DFS
MapReduce job file and chunk UUIDs
Map results
Followers
Perform map and combine on chunks

Fig. 7. Communication flow for DFS command map_reduce

IV. SETUP

The experimental setup consists of four raspberry pis running Raspbian version September 2014. All four pis are connected to a Netgear WGPS606 54 Mbps Wireless Print Server w/4-port Switch by ethernet cables, where ports are 10/100 autosensing RJ-45 ports. The switch is then wirelessly connected to eduroam, Cornell's secure Wi-Fi network. Each pi is configured to own a static IP address for ease of use (thus saving us from modifying the code every time pis get new IP address by DHCP). Pi 2 is currently running our master program while the rest of pis are running the follower program. We then run client requests from our own machine to test our system. Side note on setting up a static IP:

1) Gather network information. Specifically, we need current IP address (which will be made static later), gateway IP, netmask, network, and broadcast address.
2) Disable DHCP client and add above information to the network interface file.
3) Restart the network

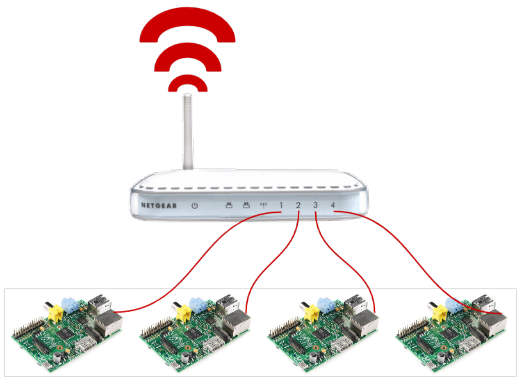


Fig. 8. The physical setup of our cluster

V. IMPROVEMENTS

Since the previous report, a number of improvements were made. First a few minor bugs in the code were fixed. For example, the cat function for viewing file contents now ensures that it only displays 1 copy of the data in the correct order (not a random shuffle of the chunks as before), and the MapReduce function now terminates gracefully by sending the proper response to the client. As will be seen in the results section, the performance of the Pi cluster ranges between 10% faster and 300% faster than in the progress report. This was caused by an improvement in the way the port select function was being handled, which was causing up to a 10 second delay between write/reads during MapReduce. Finally, the code was exported to a different set of servers for experimentation and testing. Although we were unable to have our code easily communicate between one of Cornells Cloud Computing clusters due to its security settings, we were able to port it to the Cornell CSUG machines with a few minor adaptations (since not all Python libraries used were supported on these machines). In addition to allowing for a testing comparison, the successful porting of the code to CSUG could prove crucial for its adaptation into future undergraduate class project.

VI. RESULTS

A. *MapReduce Application*

All performance testing on the created cluster was performed using a simple natural language processing application that is well-suited for the MapReduce architecture. Essentially the application counts all n-grams in a text document for n = 1,2,3, where an n-gram is defined as a word sequence of length n. The application performs the following actions:

1) Uploads a pre-processed version of the King James Bible with 1 sentence (verse) per line, which the distributed file systems spreads throughout the cluster.
2) Sends MapReduce request messages that contain the necessary Map and Reduce code modules to the master.
3) Waits for a success message to be received from the cluster master indicating that the Map and Reduce modules have completed running. All of the 1-gram, 2-gram, and 3-gram counts can then be seen in the results file through the cat command (all ngrams are placed into the same file).

For most tests, a smaller subset of the King James Bible was used, which contained the first 8192 versess (each on its own line) for a total file size of 1.20 MB (about 146 bytes/line). Using this data, the following experiments were run both on our commodity cluster of 4 Pis, and on 4 Cornell CSUG nodes.

B. *Explanation of Performance Statistics Gathered*

For all experiments, the following statistics were gathered using the "timeit" library in Python:

Upload Time: The time necessary to upload the data file to the master and all Pis in the cluster. This gives a reasonable idea of how long it takes for the data to traverse to main nodes (client to master, and master to follower).

Full MapReduce Time: The time necessary for the entire MapReduce job to be completed

Max Map Time: The maximum time that any one of the follower took to complete all of its assigned jobs. Only time spent within each Map function call was considered.

Reduce Time: The time the master node spent reducing the results from all worker node results.

Networking Time: Because all other processes of the MapReduce job were trivial, the networking time was taken to be the remaining time of the full MapReduce job. Therefore, we have: Networking Time = [Full MapReduce Time] - [Max Map Time + Reduce Time]

Note on Calculation: We believe this is relatively accurate for the networking time. On the Pis this includes serialization of the Map and Reduce modules being sent to the master, although this overhead is not included on the CSUG machines since the libraries are not supported and that functionality removed. It also includes the time required to send these serialized/unserialized MapReduce function modules to all the cluster nodes, the ssh connection overhead, and the fact that our laptop was using a USB over Ethernet connection. In addition, the calculation does not take into account concurrent networking time occurring simultaneously that could be added onto each other for total networking time – which would indicate that it is possibly a lower estimate and should slightly offset any other process overhead included.

C. *Experiment 1: Varying File Size*

For the first experiment, we varied the size of the file on which we were performing the MapReduce application. This would allow us to see the effect on shear data size on processing performance and networking time. Because we had 1 master and 3 single-core worker Pis, we decided to keep a constant number of 3 chunks for the experiment (one per Pi), and varied the number of file lines per chunk accordingly to keep the number of chunks set at 3. (A chunk is one section of the files stored on 1 of the system nodes.) The results are shown in Figure 9 below.

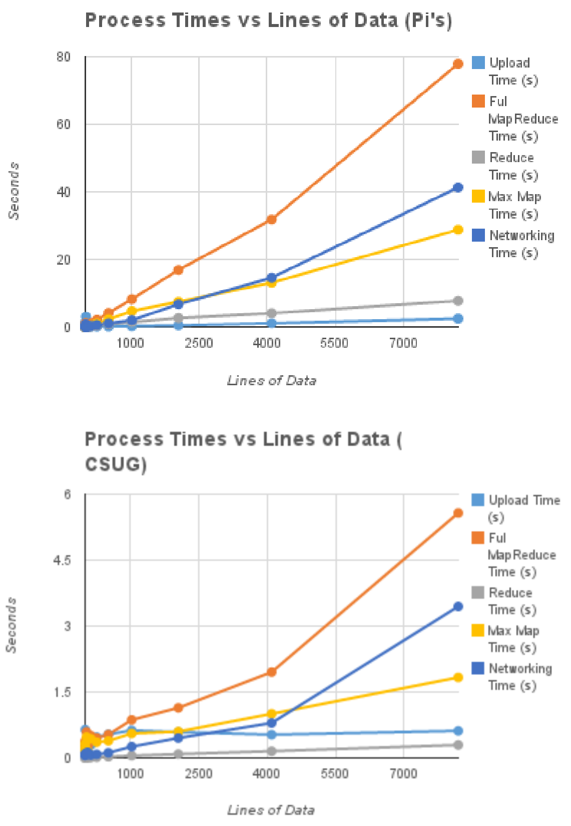


Fig. 9.

The first comparison to note is that the Pis have about 12% the processing throughput per time interval. This should be expected, given that the Pis have a slower single-core processor. However, despite the slower performance of the

Pis that creates a much larger range on the y-axis, it is quite interesting to see that the profiles of the Pi and CSUG cluster matches nearly exactly, indicating that the commodity Pis do a great job at mimicking the performance balance of a larger machine (at least in this experiment).

In both cases, the networking time begins to surpass processing time as the size of the data increases and the number of workers are kept constant. Logically, we believe this makes sense: the processor can predictably handle the larger job without much context switching, whereas the networking must handle more (possibly) unsuccessfully delivered packets and bandwidth limitations. Therefore, it appears that given a limited set of workers, increasing network bandwidth might be a good approach towards increasing the performance of the system.

D. *Experiment 2: Varying Number of Workers*

For the second experiment, we varied the number of workers performing the MapReduce job, with the job being split equally with 1 chunk per worker. The size of the file was kept constant at our current maximum file size of 1.20 MB, and the lines per chunk were varied accordingly to keep the invariant stated above. The results are shown in Figure 10 below.

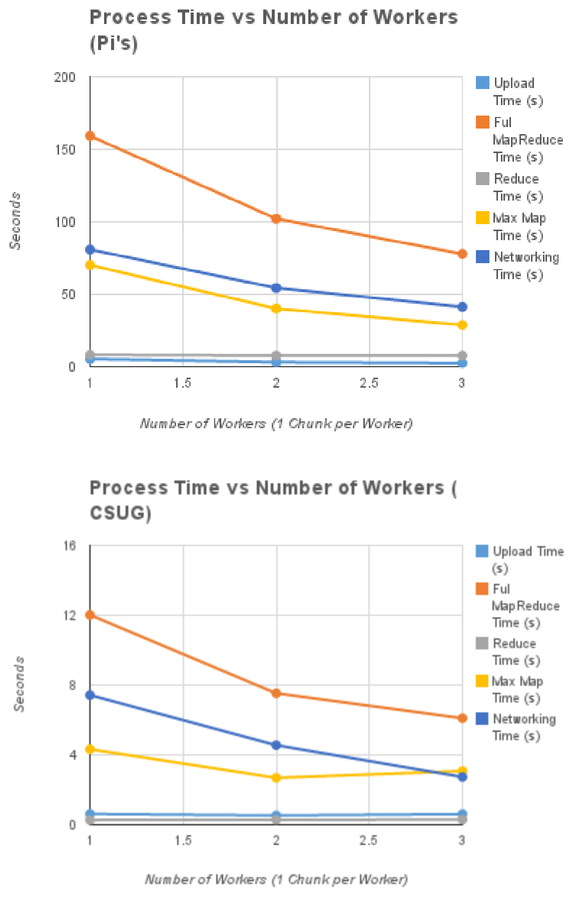


Fig. 10.

As in Experiment 1, we see the same relative slower performance between the Pis and lab machines, and also similarly matching performance profiles. As expected, as the number of workers increases, the total MapReduce time decreases. It is somewhat surprising that the networking that the network time also decreases as the number of workers increases, although this is likely caused by the fact that less per-worker data is being sent, and these messages can be sent simultaneously as long as they do not congest the same network path. Less data being sent to each worker at a time also means it is less likely for their buffers to become saturated.

E. *Experiment 3: Varying the Number of Chunks*

For the last experiment, we varied the number of chunks a mapreduce job was split into by varying the number of file lines per chunk. The size of the file was kept constant at our current maximum file size of 8192 lines (1.20 MB), and three worker nodes were used. The results are shown in Figure 11 below.

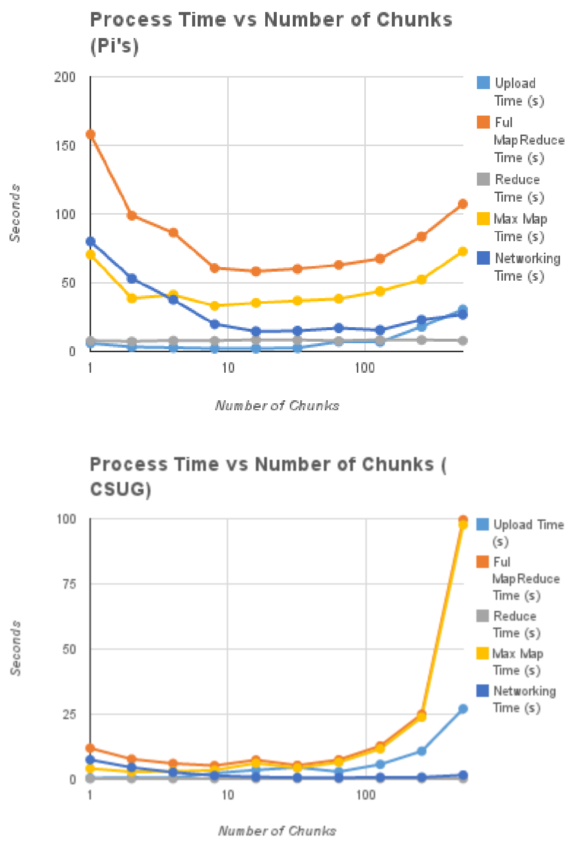


Fig. 11.

The results show that for our system, as the number of chunks increased, the processing time initially increased, as the jobs were more evenly distributed over idle worker threads. However, as the number of chunks continued to increase, the processing time began to increase linearly – becoming very large. Initially, the team believed this was due to the extra networking overhead. However, as can be seen in the results (and particularly for the CSUG machines), the networking time remains relatively low, while the processing time accounts for the great majority of the MapReduce time. This implies that as the number of chunks or small jobs increases, the jobs themselves and the multitude of context switching between these jobs is not handled efficiently on the number of machines at hand. Therefore, in the case of a datacenter similar to ours where an increasing number of small jobs are being seen, it may be wisest to increase performance by investing in more processing power instead of bandwidth.

F. Economic and Reliability Comparison

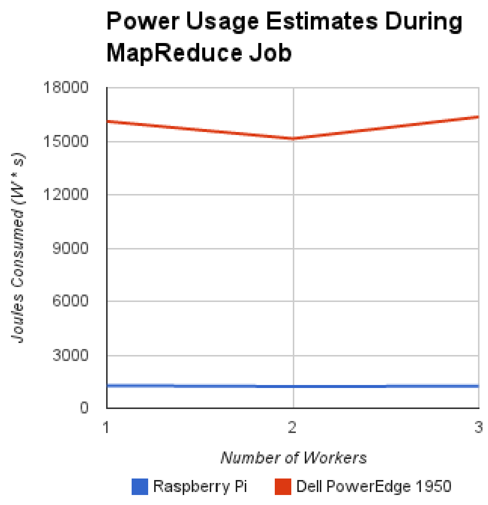
The economic feasibility of the Pis is compared against two scenarios: a user renting computation from a cloud service such as from Amazon, and that of creating ones own server.

A small Amazon EC2 general purpose on-demand server is currently priced at \$0.013/hr, which leads to a yearly cost of \$114 if run continuously. Under the assumption that we have 4 of these running year-round for an architecture similar to our own, then the total cost/year would be \$456. Energy costs are non-existant to the consumer, as these costs are covered by Amazon. As stated earlier, the Raspberry Pis cost about \$35 each, for a total of \$140 (excluding the relatively cheap switch and Ethernet cables). Let us also assume that the Amazon Web Service instances perform just as well as those in Cornells lab. Then although the price over 1 year is \$456/\$140 = 3.26 times as much for Amazon over the Pis the Pis in general perform at least 10 times worse when measured in time. Therefore, you pay a little over 3 times as much for over 10 times performance gains – so Amazon clearly is the clear winner over our Pis for the price/throughput ratio. Without considering energy costs, the cost of 4 servers would be \$2000 versus the Pis at \$140, so the Pis clearly do win out economically when considering performance/\$ in this case, especially since energy costs (discussed below) could add up to a couple hundred dollars per year for each Dell server in this case.

However, the above analysis does not take into account the reliability of the Pis. During experimentation, the Pis crashed many times and were quite unreliable. In order to monitor the cluster and keep a reasonable number of them up at any given moment, a full-time hire who costs around \$40,000/year would most likely be needed. Therefore, even when creating ones own cluster, it would be wiser to choose more reliable components.

G. Power Considerations

Now we consider comparing against the creation of our own cluster. One typical enterprise server such as the Dell PowerEdge 1950 in CSUG is rated at 650W, versus a Pi which has been reported to run at about 3W of power. At this rate, the energy costs could add up to a couple hundred dollars per year for each Dell server in this case. For our total energy consumption based on the different running length of the jobs on the Pis and CSUG, the Pis use about 1000 Joules for the job, while the Dell PowerEdge server uses about 16,000 Joules – over an order of magnitude difference! Therefore, if the Pis could be made reliable, and we could tolerate longer running times, then the Pis are an excellent option for conserving a lot of power, and the costs associated with that power!



VII. Future Work

Due to the success of the distributed file system and simple MapReduce framework established, the team plans to form the project into an assignment that can be used for undergraduate students. It will be very useful to expose them to basic networking components using TCP/IP connections, the concept of MapReduce over a distributed file system, and the necessary synchronization to keep processes coordinated. It will also increase their experience with Linux commands to ssh and work on the remote machines through the terminal. However, due to the unreliability of the Pis, it might be wisest to have the project hosted on the CSUG machines. On the CSUG machines, the level of accessibility and reliability would create a much more solid project base for students.

VIII. Conclusion

By experimenting with creating the Pi cluster and profiling the process times of computational MapReduce jobs, he team has arrived at a number of conclusions. As the pure size of individual jobs and documents being passed through network increases, networking time appears to become the bottleneck, and performance would likely benefit most by increasing bandwidth. As the number of working nodes in the cluster increases, both overall networking time and processing time of jobs tends to decrease. Finally, as the number of small size jobs increase, processing time accounts for the majority of the MapReduce job time due likely to much context switching between simultaneous processes, and therefore investing in more nodes/processing power is key to performance gains.

Economically, the Pis cannot compete against the consumer costs of a cloud service like Amazon on a dollar/throughput basis. Although using the Pis to create ones own server is significantly cheaper than creating ones own cluster using enterprise-level equipment when considering product and energy requirements, the economic gains are hardly noticed when the labor required to keep the Pi servers up and running is taken into account. Therefore, we deem commodity parts as unreliable as the Pis not to be fit for actual personal datacenter use.

References

- RPi Setting up a static IP in Debian
- Dell PowerEdge 1950 Server Specifications
- Raspberry Pi Power Consumption
- Amazon EC2 Pricing
- Raspberry Pi
- MapReduce: Simplified Data Processing on Large Clusters

Appendix

A. Instructions to Run

On Master

1) In the top-level Tiny Data directory run `python master`
2) Look up and write down the public-facing IP address of this machine

On Followers

1) In the top-level Tiny Data directory run `python follower master_ip` where `master_ip` is the IP address recorded before.

On Client

1) In the top-level Tiny Data directory run `python client master_ip` where `master_ip` is the IP address recorded before. This will print out a list of commands that are available.
2) Here's an example of every command. All paths in the DFS *must* be provided as absolute paths, starting at root (/)
   - ls - `python client master_ip ls /`
   - rm - `python client master_ip rm /some_data_file`
   - mkdir - `python client master_ip mkdir /my_dir`
   - cat - `python client master_ip cat /some_results_file`
   - upload - `python client master_ip upload /data_path_on_dfs /data_on_my_computer 1000`
   - map_reduce - `python client master_ip map_reduce /data_path /store_results_here map_file.py reduce_file.py --combine combine_file.py`

B. Word Count MapReduce Job File

```
import re

def map_fn(line):
    words = re.findall(r"[\w']+", line)
    return [(word.strip().lower(), 1) for word in words]

def reduce_fn(key, values):
    return sum(values)
```