Parallel Computing

Damon Alcorn, Peter McPherson, Elizabeth Wanic

OS 3070 - Operating Systems

8 March 2017

# Do You Parallel?

In this day and age, all computer science professionals have at least a cursory awareness of parallel computing. Almost everyone has heard that many processors working in parallel can accomplish more than one alone, and they have heard the stories of massive, expensive super computers that take advantage of this concept. Yet, for most, the implementation details remain shrouded in mystery. In this project, our group set out to answer the following questions: What are appropriate, reasonable applications for parallel computing? How can a common computer user construct a cluster of parallel processors to perform a task?

# Common Applications

A single processor can do almost anything that multiple processors can do; additional processors simply add speed. Super computers are popularly utilized on problems that require intensive mathematical computation or when the total amount of data that must be analyzed is unusually large. The latter scenario is common for scientists studying nature, as the sheer scope and complexity of our universe naturally results in an almost unlimited amount of data and types of analysis that can be done. Examples where parallel computing is utilized include weather prediction, finite element analysis, and genetic engineering. There are also plenty of human-oriented applications, such as modeling the world economy, sorting through genetic code, or simulating and analyzing the performance of a nuclear weapon.

Some may think that if parallelization speeds up processes, it should be utilized for every task. However, there are problems that do not lend themselves to parallelization, in particular, those that depend mostly on sequential, serial algorithms. In these cases, additional processors cannot add speed due to the fact that each subsequent phase is dependent on the previous phase’s completion. Whether you had ten processors or one, each would need to wait for the tasks to be completed in sequential order. Many mathematical equations, such as encryption algorithms, operate in this way.

But the fact is, many algorithms can be sped up to some degree with parallelization. To make the ultimate decision on whether the technology should be used to solve a particular problem, the potential increase in speed must be weighed against the cost of implementing or renting a supercomputer. Although processing power today is absurdly cheap compared to 30 years ago,[[1]](#footnote-1) costs can still add up. Every supercomputer requires some amount of design, materials, setup time, extra electricity, and maintenance (as our group found out first hand). These are a few reasons why parallel computing has been slow to take off for the average consumer. The availability of inexpensive hardware such as Arduino and Raspberry Pi are starting to change this, allowing regular users to construct their own version of a supercomputer, such as we did in this project. But as we found out, in the end, you usually get what you pay for.

# It’s All About Memory

Not all parallel computers are created equal. Some have higher quality processors, or greater numbers of them, but there are a many other considerations that can dramatically affect performance. Far and away the biggest factor is how each processor accesses the data it needs to operate. Super computers are broken down into two categories:

* Shared memory systems--popularly known as multiple core machines--in which all processors are clumped together within one “CPU”
* Distributed systems, in which multiple independent systems, each having their own memory resources, are connected to form a computational network.

On a shared memory system, all processors have a short path to access the data they will need to read or the locations to which they will write during the course of execution of a process, as they will utilize RAM or long-term storage connected directly to the computer. In the distributed model, however, this is not the case. The data will be either accessible from only one of the nodes in the network, or it will be located on a central server. Either way, most of the processors in the network will have to fetch data and load it into their local memory before they can work on it, which often results in considerable overhead. For some jobs, this overhead may be too costly, thus defeating any possible gains from parallelization.

Distributed systems are popular with consumers, because they are relatively easy to build by linking together various pre-existing machines and using open-source software to make them communicate. Engineering shared memory systems is beyond the ability of most consumers, so they must be purchased from retailers at a high price. The performance of distributed systems will usually be slower than shared memory systems, and the former will not work well on algorithms which require community data to be edited frequently.

# Setting up our own Supercomputer

In order to investigate this topic in a hands-on manner, we decided to build our own supercomputer composed of four quad-core Raspberry Pi boards. This kind of setup is called a cluster, but Raspberry Pi aficionados call it a bramble. Our idea was to run parallel code and to analyze the increase in speed gained by using various numbers and combinations of the 16 available processors.

After obtaining the Raspberry Pi units, our first task involved deciding which operating system we would utilize for our project. We discussed the need to use a simple operating system, which would allow us to easily download and incorporate the modules required for our experiment. After exploring two other operating systems at length, we ultimately chose Ubuntu Mate as we had all worked with Ubuntu operating systems for various projects in the past.

Once the operating system had been installed on each of the four Raspberry Pis, we connected them to each other via an Ethernet switch, and statically assigned each Pi its own IP address using the Ubuntu Network Manager application. We also updated the /etc/hosts file on each of the Pis to include IP addresses for each of the other nodes so that their hostnames could be resolved to their actual IPs. Our next hurdle involved setting up SSH, without passwords, so that the nodes would be able to communicate with each other. Ordinarily, this should have been a somewhat trivial task, involving simply downloading OpenSSH and creating and sharing RSA encryption keys. However, this proved to be a major hurdle in our case, as no matter how many different ways we tried to share keys or which settings we changed, we were unable to enable password-less SSH. Ultimately, we did discover the proper settings for encryption and key sharing in the /.ssh/sshd\_config file, but we also learned that there is a dearth of accurate information on this particular topic. Lesson number one: when building your own cluster, do not expect setup to go smoothly, no matter how many tutorials you have read.

With the Pis now able to communicate without passwords, it was time to set up a folder on the master node, which would be shared with all the slaves. Once the file sharing system was installed, whatever code and data was necessary to run the program needed only to be present on the master node, but would be accessible by all nodes. We used the NFS module for this part of the project, which provided a straightforward and painless setup process.

The last setup task was to install an implementation of the Message Passing Interface (MPI), which is a proven API that has been used for decades to facilitate communication among distributed nodes in supercomputers. When MPI methods and commands are used in C, Python, or Java programs, the module automates most of the parallelization of the code. We chose to install the MPICH implementation, which is one of the most popular.

# Analyzing Performance

Next, we had to decide which task to give to our cluster. Given the distributed memory concerns discussed earlier, we chose to avoid data-centric problems, instead focusing on finding a task that was math-based and highly parallelizable. Of all the tasks we considered, performing hashes on many passwords seemed like a good fit. We adapted code from our NPS classmate, Alexis Peppas, to systematically compute SHA hashes of all possible combinations of upper-case letters, lower-case letters and digits, and then check these hash values against values in a known password hash file.

We computed passwords of length 4, which produced 14,776,336 passwords as well as an equal amount of hashes - a lot of work to do. With only one of our processors performing the entire task serially, it took approximately 24 seconds. When we split the task between two slave processors, with a third as a master node to handle the administrative work, the total time was 12 seconds. Since the bulk of the code for this task fell in the parallelized section, our cluster was able to take full advantage of parallelization, which could be seen by the time being cut in half. In this case, we were utilizing three processors on the same node, essentially a shared-memory situation. We then conducted experiments across the nodes, to see whether there were noticeable inefficiencies introduced by the distribution process. Our timing method was perhaps not granular enough, but we were not able to observe a significant difference while the master and two slaves were on the same node, versus having the master on one node and two slaves on another.

We continued to divide the work between more and more processors in our cluster, and observed that the computation time continued to decrease. Once we advanced to utilizing 8 processors, the time to compute all the hashes was down to 3 seconds. From this point on, the benefit of adding additional processors began to be negligible. There was no observable increase in speed when we utilized more than 9 processors as slaves. After this point, the completion time levelled off at 2 seconds. Even with 15 processors participating, and 1 administrating, the completion time was still close to 2 seconds. We did not find this surprising however, as we were using a distributed system for a task that was not overly time consuming to begin with. At some point, the overhead of dividing the tasks among all nodes and sending them the minimal data that they required to do the hashes took almost equal time as computing 1 million more or less of the hashes.

# Results

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Slaves** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| **Seconds** | 24 | 23 | 12 | 8 | 6 | 6 | 5 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |

Above is a table showing the number of slave processors (i.e. *n* slave processors used, plus one uncounted master processor, for an actual total of *n + 1* processors) utilized and the resulting time it took to complete the task (in seconds). Below, this data is graphed.

Improvements in performance through the adding of slave processors are visibly impressive after the addition of just a few, but quickly level off as noted above. An interesting aspect of the plotted data is the subtle waveform it creates. The meaning could be interpreted in various ways, but a reasonable conclusion is that the use of each additional processor does not result in a uniform reduction in the total time it takes to complete the task. Rather, numerous variables governing each processor’s operation, and therefore the operation of the cluster as a whole, make for somewhat varied improvements in overall performance when a slave processor is added.

For example, when an additional processor (processor *k*) is housed on a different CPU/motherboard than processors *1* through *k-1*, this could result in larger signal transmission times to and from processor *k*. Additionally, the use of processor *k* can result in an even or uneven distribution of the parallelizable task (i.e. with 2 versus 3 slaves, etc.). This may affect the speed with which certain processes complete, possibly affecting aspects of operation such as process queuing and the nature of bus/wire traffic, as well as data splitting and consolidation. Numerous factors, including processor placement, bus/wire distance between a slave and the master processor, process start and completion times, variations in CPU clock speeds, etc., all play a part in just how much performance is gained with the addition of a slave processor. As a next step for a more in depth project, it would be interesting to further investigate this phenomenon, as well as perhaps attempt to account for it.

# 

# Was it Worth It?

As we discovered, there is a large amount of work that goes into setting up a system to run parallel code. The amount of overhead required from a hardware, software and integration perspective is quite large. In addition, the user needs to consider whether the code can in fact be parallelized and after that, must (re)write the code so it can be executed in this fashion. With this in mind, most problems are too small to make parallelization worthwhile, as the long-term benefits will likely not outweigh the costs. Nevertheless, there are jobs for which parallelization is favorable, such as the ones mentioned earlier, which might take days, months or even years to complete if run serially. In these cases, the time given to preparation and setup, is well worth the savings gained when executing the code in parallel.

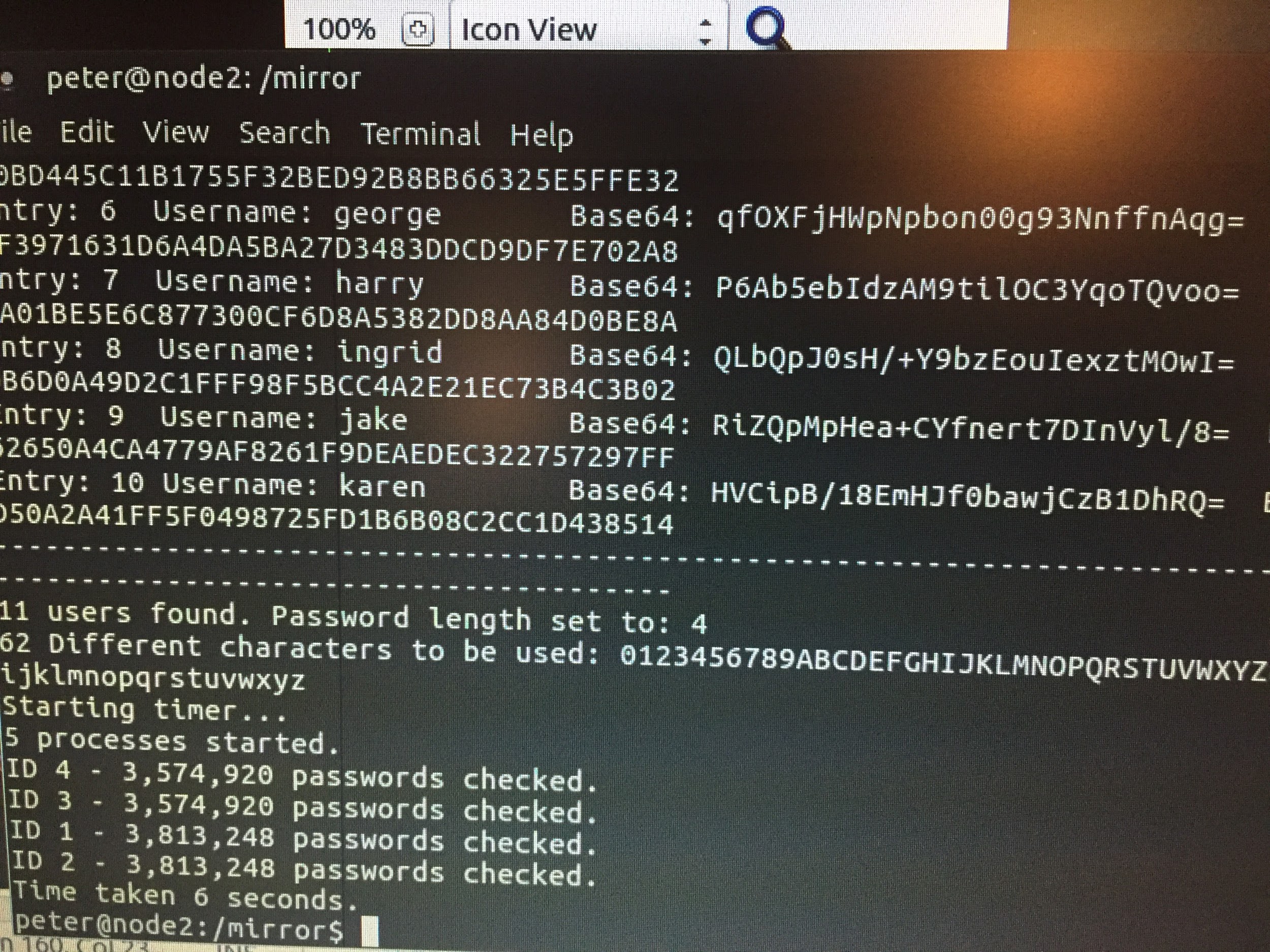
Amdahl’s law discusses the idea of maximum increase in speed of a program, depending on the portion which can be parallelized. Using P as the proportion which can be made parallel and thus 1 - P as that which must remain serial, he states that the maximum speed up using N processors is 1/((1-P)+(P/N)). For a job including a large proportion which must remain serial, the speedup even with use of a very large number of processors will remain minimal. We observed this law in effect in our experiment. The job completion time was sped up with an increase in processors, even though there was a point at which it levelled out. As the job was comparatively not that large, we reached a point where adding additional processors did not bring about additional increase in speed. Had we more time, we could have conducted other experiments to see how this law held for other, much larger, jobs and compared results of those that were mostly parallelizable and others that required a large part to remain serial.

Overall, we learned a great deal about this important aspect of computer science. While there is much more to discover with respect to this topic, our project helped us develop a new understanding of parallel systems, their construction and use.

# Photos



*Our cluster of 4 Raspberry Pi units connected via Ethernet switch. Each Pi is on and running, but we only interact with the master node to run each process.*



*An example of our output. The number of processes is shown. Here it is five, including the master node as administrator and four workers. You can see the number of passwords checked by each of the worker nodes, along with their corresponding ID number. The total time taken is indicated on the last line.*

1. This is an understatement. The Intel 4004 in 1970 cost $200 ($1250 inflation adjusted), and executed 92.600 instructions per second. (<https://www.extremetech.com/computing/105029-intel-4004-the-first-cpu-is-40-years-old-today>). In comparison, Intel’s i7 chip in 2016 can compute approximately 177,730 million instructions per second, and costs $300. (<https://www.quora.com/How-many-machine-instructions-can-excute-per-second-on-a-modern-processor>) [↑](#footnote-ref-1)