Raspberry Pi Cluster Computer

Trendley, Skylar
stbrb@mst.edu

Ward, George
gcwzf4@mst.edu

December 7, 2018

Abstract

A cluster computer is a set of loosely connected computers that work together to form a single system. Cluster computing has gained popularity in recent years due to the increasing availability of inexpensive high-performance microprocessors and high-speed networks, allowing for a cost-effective alternative to single computers without sacrificing speed or reliability. Computer clusters tend to have each node on the cluster divide a work load into processes executed in parallel. By running the processes in parallel, cluster computing provides a way to accelerate the computational time of a solution by employing multiple hardware resources to work simultaneously. In our study, we have employed the use of seven Pi 3 B+'s loaded with the Raspbian OS to form an OctoPi cluster. The cluster then communicates between nodes using MPI4PY, a message passing interface that allowed us to pass processes back and forth between Pi's. By utilizing the cluster to perform a series of matrix multiplication operations, we attempt to test, record, and analyze the speed up times at which parallel computing executes processes across a cluster.

1 Introduction

1.1 A History of Cluster Computing

The first inspiration for cluster computing was developed in the 1960's by IBM as an alternative of linking large mainframes to provide a more cost-effective form of commercial parallelism [1]. However, cluster computing did not become a popular means for high-performance computing until the 1980's, when the need for high-performance microprocessors, high-speed networks, and the tools to develop a method of distributed computing surmounted. As the means for acquiring the hardware for high-performance computing increased, cluster computers quickly emerged as leaders in the industry. By utilizing a cluster computer to harness the combined computing power of multiple processors in a parallel configuration, one may expedite the rates at which a process is executed [2].

1.2 The Utilization of a Cluster Computer

Cluster computers are designed with intent to take large sets of data or programs and divide them into tasks for each node of the cluster to process. When the task is complete, the node will then pass its data back to the controller, and the controller will re-synchronize the data. The message passing interface between nodes is typically deployed on a high-speed local area network. Cluster computers can be classified into three different types: fail-over, high-performance, and load-balancing [3]. Fail-over clusters are used to provide an uninterrupted stream of data services to an end user. Fail-over clusters offer a level of redundancy that allows for a node to fail and the service to be restored onto a different node in the cluster without service interruption. Fail-over clusters are often built with two nodes, correlating to the minimum number of nodes required to provide a level of redundancy. Load balancing clusters can be built with fail-over utilities but often prioritize sending tasks to machines that are the least "busy" executing other processes. High-performance clusters often utilize load balancing techniques but focus on paralleling processes into routines that can be run on separate machines synchronously instead of waiting for each process to complete before beginning the next. By utilizing parallel computing to complete large data set processing tasks, the power of cluster computing can rival if not surpass the speed of a singleton computer. To test the theory of parallelism expediting the execution speed processing tasks on cluster computers, our team created a high-performance raspberry Pi cluster that would execute matrix multiplication tasks. Each task would be recorded and tested against a single computer to compare the results of computation and execution speed times between systems. Our team predicted that by increasing the number of processes to match the number of cores on each Pi, the efficiency of the processing would be increased and therefore the maximum speed up time would occur when the number of cores matched the number of processes.

2 Experimental Environment

2.1 Configuration of the Cluster

Our environment for testing cluster computing is based on a cluster consisting of 7 Raspberry PI 3 Model B+′s, each utilizing 1GB of LPDDR2 SDRAM and a 4 core Broadcom BCM2837B0, Cortex-A53 64-bit SoC processor running at 1.4 GHz.8 16GB SanDisk Class 4 SDHC Flash Memory Cards 7 16GB SanDisk Class 4 SDHC Flash Memory Cards were formatted and loaded with Raspbian Lite to give to the Pi′s. After the initial controller Pi, designated as Pi1, was flashed with Raspbian Lite, we installed MPICH 3.1 onto the SD card to allow for multiprocessing communication between computers as well as MPI4PY for the message passing interface and Python to compile our code. It should be noted that the OS for Pi1 was backed up after downloading all the utilities needed for pro-

gram execution and copied over to the other Pi′s SD cards for consistency. After the Pi′s was configured, it was placed onto the cluster and connected to a GearIT Cat6 3-foot Ethernet cable, which ran to a NetGear 8-Port Gigabit Ethernet Switch to allow for communication as well as SSHing into each Pi. Finally, each Pi was connected to a USB cable which ran to an 8 USB port surge protector to provide power to each Pi in the cluster.

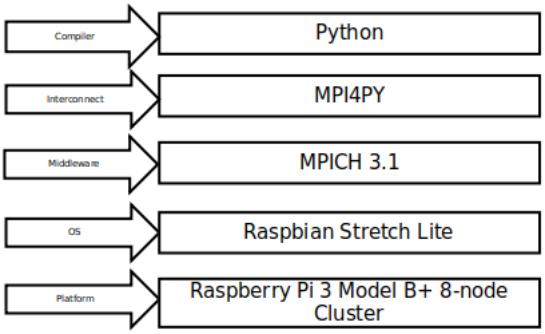


Figure 1: architectural stack of the testing environment

2.1.1 Setting Up the Speed Performance Test

To distinguish between Pi′s on the network, nmap was installed on the Pi Controller to scan for Pi′s on the network. Once each Pi′s IP address had been recorded, we decided to SSH into each of the Pi′s and change its configuration to differentiate the names as Pi1 through Pi8. Next, a machine file was created to contain all the Pi′s IP addresses so the MPI4PY could read and distinguish between them when designating tasks. Finally, we downloaded a matrix multiplication program written in Python to test the results of parallel computing on the Pi′s.

2.2 Problems with Permissions

Once we attempted to run the **mpiexec** command to execute the matrix multiplication program, we encountered an error involving the restriction of access rights to the other six raspberry Pi′s from the controller. To remedy this, we decided to create and swap authentication keys between the Pi′s. By using SSH to keygen a random art image encryption key onto an authorized key list on each of the Pi′s, we were able to gain access rights to each of the Pi′s for parallel communication.

2.3 Testing the Cluster

Once the cluster had been properly configured and was ready for testing, we utilized the mpiexec command to begin the parallel communication process. The command, as shown in figure 2, shows the parameters required of mpiexec to allow for parallel computing. The “—machinefile“ flag indicates that the proceeding file should contain a list of IP addresses of machines to divide processes between. The “—n“ flag indicates the number of processes to be generated for parallel execution. We decided to test the results of three different numbers of processes—4, 14, and 28. Finally, the command requires the name of an executable file to run. For our matrix multiplication program, we decided to

test the outputs of six different sizes of matrices—24x24, 56x56, 80x80, 144x144, 180x180, and 280x280. By changing the size of each matrix, we were able to record and analyze the speed up times by each number of processes.

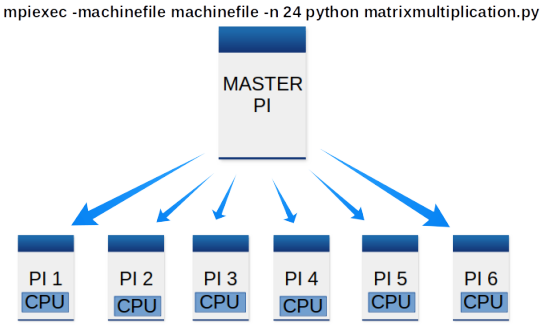


Figure 2: distribution of processes among Pi′s

2.4 Serial Versus Parallel Communication

To compare the results of the speed up time to a single computer, we decided to test the first six matrix sizes using 4 processes. The data was then transferred to the designated Pi serially to be executed on each of its cores. Through serial communication, datum is transferred from node A to node B via a singular byte stream. This byte stream will then enter a processing queue that will continue once the current byte stream has finished transferring. In parallel communication, multiple byte streams can be transferred and processed synchronously, allowing for faster completion times. By utilizing MPI4PY to communicate between each of the Pi′s, we were able to establish parallel communication across the cluster. Each Pi could then divide a portion of the work for processing and return its result back to the controller.

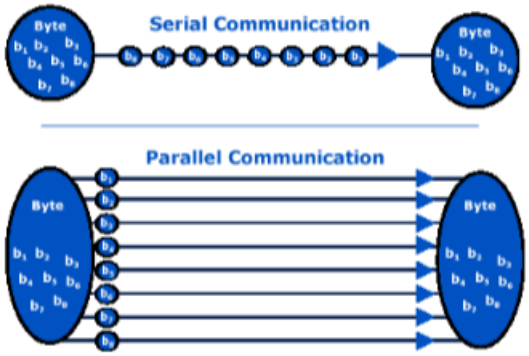


Figure 3: A representation of a data stream transferring serially vs. in parallel

3 Results

3.1 How Parallel Computing Affects Processing Time

Once the command was executed, the Pi controller would then access the program, convert the process into a byte stream, and send the data to each Pi, including itself, to process the multiplication task. As seen in Figure 4, the trial run with 4 processes took the longest amount of time to execute. By increasing the dimensions of each matrix, the processing time that was required increased

dramatically. As expected, sending datum serially over a byte stream and waiting for each task to conclude before processing another resulted in reducing the speed at which a task could be processed. Next, we attempted to test parallelization by increasing the number of processes to 14. These processes were then divided by the controller and sent to each of the Pi′s in the cluster to complete. Each process was executed on an individual core inside of the Pi and sent back to the controller. While increasing the size of the matrices, the processing time required to complete the tasks only increases by a fraction of a second, making the 14 processes to seemingly be the second most efficient of the tests. Finally, 28 processes were given to the controller, divided, and sent to each Pi to be processed. The results of this operation were like that of 14 processes but with an increase of efficiency. Initially, we predicted that having a process for each of the cores would be the most efficient use of parallelization because every core on every Pi has a process being executed. This would prevent thrashing while attempting to switch between the cores of a Pi to process a task as well as prevents the under-utilization of processors. For a 14-process task, every single core is not being utilized and therefore potential processing time is wasted. According to the results of the three tests, the data did in fact reflect our hypothesis.

Figure 4: The results of varying the dimensions of matrix multiplication operations on the number of processes.

TEST 1						
processes	28x28	56x56	80x80	144x144	180x180	280x280
4	0.0404	0.3018	0.867	5.1602	10.3072	38.6905
14	0.0104	0.0689	0.2092	1.2123	2.2196	8.7283
28	0.0114	0.0726	0.1225	1.109	1.9348	6.9833

TEST 2						
processes	28x28	56x56	80x80	144x144	180x180	280x280
4	0.0395	0.3006	0.8722	5.3085	10.1742	38.971
14	0.01	0.0708	0.2048	1.22	2.2501	8.7001
28	0.0055	0.0692	0.1398	1.0931	1.7382	6.837

TEST 3						
processes	28x28	56x56	80x80	144x144	180x180	280x280
4	0.0403	0.3091	0.8902	5.2079	10.4164	38.5956
14	0.0103	0.0708	0.2066	1.2306	2.2313	8.772
28	0.0096	0.0706	0.1244	1.0688	1.9003	7.3031

Figure 5: The results of the first test.

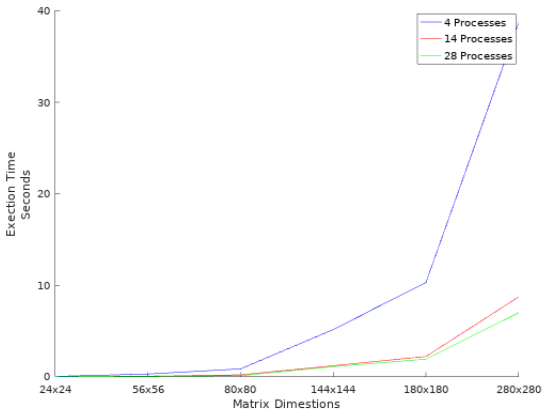


Figure 6: The results of the second test.

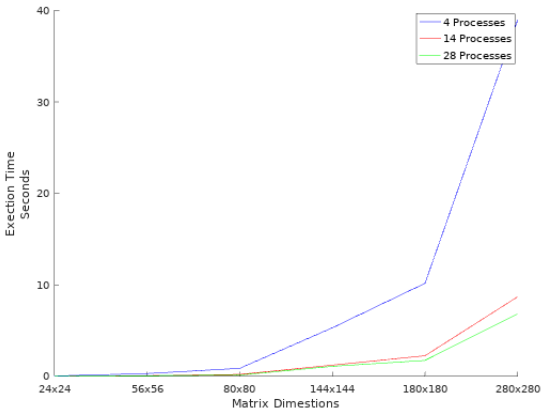
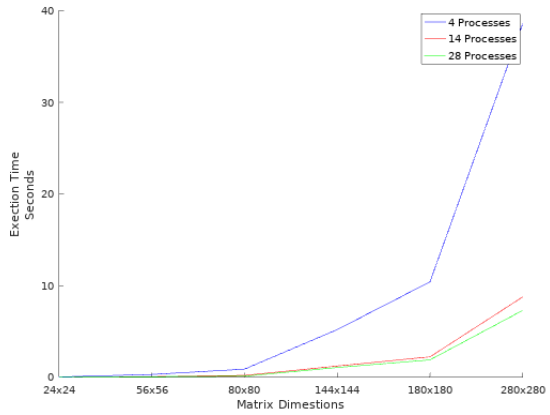


Figure 7: The results of the third test.



3.2 Conclusion

Cluster computing and parallelization has the ability to increase the speed at which a task can be completed compared to a single processor. By utilizing a controller to send datum via parallel communication to different systems, each system can process the data synchronously, therefore reducing the amount of time required to finish each task. Although computer clusters allow for a potential increase of processing speed, the number of processes being executed on each system can impact the efficiency of parallel computing. The performance increase may also be dependent on the configurations of the cluster and the nature of the program being executed on the cluster. Our high-performance cluster did increase the speed at which the processes were executed, proving our hypothesis that suggested that matching the number of cores to the number of processes would result in maximum efficiency. If we were to continue the experiment, we would like to compare our data to other clusters utilizing parallel computing to determine if our data deviates from other findings. We would also attempt to replicate our results on a larger data pool to confirm its consistency and reliability. If our initial hypothesis remains uniform with our results, we would design a future experiment to test the usage of a cluster computer on more complex tasks such as panoramic pictures or security system monitoring.

References

[1] Buyya, R. *High Performance Cluster Computing: Architectures and Systems.*, Prentice Hall, 1999.

[2] IBM. *Clustering: A Basic 101 Tutorial. Retrieved from IBM.com*, https://www.ibm.com/developerworks/aix/

tutorials/clustering/clustering.html,
(2002, 4 3).

[3] Sharma, P., Kumar, B., &; Gupta, P. *An Introduction To Cluster Computing Using Mobile Nodes*, https://ieeexplore-ieee-org.libproxy.mst.edu/document/5395041/, (2009, December 16).

[4] Cabilla, Jordi, *Matrix Multiplication and MPI4PI-Examples, Retrieved from github.com*, https://github.com/JordiCorbilla/mpi4py-examples/blob/master/src/examples /matrix%20multiplication/matrixmultiplication.py, (2016, August 16).