Performance Testing a Cluster of Raspberry Pis

Prepared for: Dr. Apan Qasem
Associate Professor
Department of Computer Science
Texas State University

Prepared by: William Brooks
Amber Guevara
Cody Wright

May 15, 2019

Abstract

"Performance Testing a Cluster of Raspberry Pis"

Prepared by: Will Brooks, Amber Guevara, Cody Wright

The purpose of this project is to evaluate and compare the performance of a cluster of Raspberry Pis. To test for performance, we run matrix multiplication with HPL, a performance software, on different combinations of threads and nodes. Each combination was run with four different problem sizes (2000, 4000, 6000, 8000). Each of these problem sizes went through three cycles and the average values were recorded. The gathered data consisted of Cache Miss Rate, CPU Execution Time, GFLOPs (one billion floating point operations per second), Bandwidth, and Power Consumption. Once testing was complete, our team compared the different combinations to evaluate for any performance improvement.

Our findings were a mixture of predictable and unexpected conclusions. General cache miss rates and execution times built off the problem sizes as one would expect. The larger the problem, more cache misses are likely, and the program takes longer to finish. GFLOPs and bandwidth were more situational in performance. This data favored individual nodes in small problems and clustered nodes in larger problems.

Keywords: computer performance, clustering, speedup, evaluation.

Table of Contents

Introduction **1**
Methodology **2**
I. Assembling the Cluster 2
II. Installing HPL Benchmark 3
III. Installing Perf on Raspbian 3
IV. Running Tests on the Cluster 4
Results **5**
I. Cache Miss Rate 5
II. CPU Execution Time 7
III. GFLOPs 9
IV. Bandwidth 11
Conclusions **13**
I. Cache Miss Rate 13
II. CPU Execution Time 13
III. GFLOPs 13
IV. Bandwidth 13
References **14**

Introduction

Computer performance is believed to increase through parallel programming. One way to achieve parallel programming is to create a network of computers that communicate with each other and share data. Running multiple threads allows the computers to take advantage of their multicore processors and split up tasks, dividing them among the group to be completed. Using these concepts, we constructed a cluster of Raspberry Pis with parallelized software that allowed us to manipulate how many nodes were used at a time and how many threads were run.

The testing process utilized a benchmark program that solved a large order linear system of equations on the cluster. The data collected would be an assortment of execution times, cache actions, and power consumption. The ability to choose the nodes accessed and threads available with these tests would give insight into specific performances. When compared, different combinations produced better performance than others. The main goals of this project were to evaluate how performance is affected by networking together Raspberry Pis and evaluate the performance benefits of parallel programming.

Methodology

In this section, we will detail the process to complete the performance testing on a clustering of three Raspberry Pis. Our goal in this section is allow this experiment to be replicated. We will include any issues we encountered and our troubleshooting efforts.

I. Assembling the Cluster

To be able to run performance benchmark tests on multiple Raspberry Pis simultaneously, we needed to complete a process called “clustering”. This refers to connecting computers to work together to perform a single task. They can be viewed as a single system once connected.

For our experiment, we used two Raspberry Pi 3 Model B+ computers [1] and one Raspberry Pi 3 Model B computer [2]. The main difference between the two being the B+ CPU is built with better thermals which allows a higher clock rate. The specifications for each model are in the table below.

Model	CPU	Clock Rate	Core Count	RAM Size
Raspberry Pi 3 Model B+	Broadcom BCM2837B0	1.4 GHz	4	1 GB
Raspberry Pi 3 Model B	Broadcom BCM2837	1.2 GHz	4	1 GB

The following are a reduced set of important steps in the build from tutorials we followed [3] [4]:

Step 1. Configurate the head Pi for Clustering
From raspi-config, select expand filesystem, enable ssh, change the hostname to node0, and set the GPU memory split to 16MB

Step 2. Update and update Linux packages

Step 3. Install Fortran which is used during MPICH install

Step 4. Install MPICH
MPICH is an open source implementation of a Message Passing Interface library [5]

Step 5. Edit .bashrc and add the path to mpi

Step 6. Make disk images of the head node, to run on other nodes
We used the installed Raspbian OS *SD Card Copier*

Step 7. Assign static IP addresses to the nodes

Step 8. Assign node hostnames to node1, and node2

Step 9. Create a host file to associate the static IP addresses with the hostnames

Step 10. Write a Machinefile which is a list of nodes you want to execute a program

Step 11. Set up ssh using RSA keys to bypass password login

II. Installing HPL Benchmark

In order to test performance on this cluster of Raspberry Pis, we decided to use the High-Performance Linpack Benchmark, or HPL. The algorithm used by HPL "solves a (random) dense linear system in double precision arithmetic on distributed-memory computers" [6].

The following is a reduced set of instructions as followed from tutorial [7]:

Step 1. Install Fortran and MPI library as detailed in Method I. *Assembling the Cluster* if not yet completed

Step 2. Download and build the linear algebra library ATLAS [8]

Step 3. Disable CPU throttling on the Pi

Step 4. Download HPL [6]

Step 5. Edit the paths to the MPI & ATLAS libraries in Make.rpi

Step 6. Compile HPL

Step 7. Edit HPL.dat to tune the benchmark for your system

III. Installing Perf on Raspbian

While the HPL package would provide us with GFLOPs information, we wanted to also measure CPU Execution Time and Cache Miss Rate. Unfortunately, the "perf" command does not come factory installed with the Raspbian OS.

We followed these steps to obtain the perf command [9]:

Step 1. We installed supplement software with the commands:
sudo apt-get update
sudo apt-get install bc flex bison

Step 2. We then cloned the Linux kernel specific to Raspberry Pi
git clone https://github.com/raspberrypi/linux.git

Step 3. Once cloned, we were able to compile perf using
cd linux
make ARCH=arm -C tools/perf/

Step 4. Copy the compiled program to */usr/bin*

Step 5. The compiled perf was found in *where_you_clone_the_git/linux/tools/perf*

IV. Running Tests on the Cluster

To measure performance of the cluster, we decided on the following metrics to analyze data:

- Cache Miss Rate
- CPU Execution Time
- GFLOPs
- Bandwidth
- Power Consumption

The executable file, *xhpl*, was run with the following command to capture all necessary data:

mpiexec -n 1 -f machinefile perf stat -e cache-misses,cache-references ./xhpl

In the above executable, "-n" determined the number of threads the program was to be run on. "Machinefile" referenced the list of nodes used to run the program. The program calculated and printed GFLOPs information into an output file. The perf stat information displayed on the screen and had to be captured manually after each trial. Because of the parallel execution, the perf stats were printed in an out-of-order fashion, and had to be pieced together carefully.

In HPL.dat, we had to specify the following information for each configuration:

- Problem Size – "N"
- Thread Configuration – PxQ where P x Q = Total Number of Threads
- 1-Node Configurations - PxQ
    - 1x1
    - 1x2, 2x1
    - 1x3, 3x1
    - 1x4, 4x1
- 2-Node Configurations – PxQ
    - 1x2, 2x1
    - 1x4, 4x1
    - 1x6, 6x1
    - 1x8, 8x1
- 3-Node Configurations – PxQ
    - 1x3, 3x1
    - 1x6, 6x1
    - 1x9, 9x1
    - 1x12, 12x1

Each configuration was run with a problem size of 2000, 4000, 6000, and 8000. We ran each executable at a specific configuration three times and calculated the average. The data presented in the next section refers to the averages we calculated.

Before we continue to our results, a side note about data collection: Some of the thread configurations and/or problem sizes were unable to run on our cluster. The programs either timed out or displayed an error after several minutes of attempting to run. We were unable to successfully collect data on 12 threads being run on three nodes, for undetermined reasons. We also ran into issues with nine threads being run on three nodes. This is likely due to the sensitivity of the HPL benchmark to the system and tuning variables needed to run.

Results

In this section, we will present the results of our data collection using the executable file and perf stat to record data. For each metric, we will provide details on the metric itself, then provide charts showing results for weak and strong scaling.

I. Cache Miss Rate

An important factor in evaluating performance of a system is the cache miss rate. When a cache miss occurs, data must be brought in from a lower level of data. Cache miss rate refers to the ratio of cache misses versus cache references. The charts below display the weak scaling results when configured with one thread per node and three threads per node.

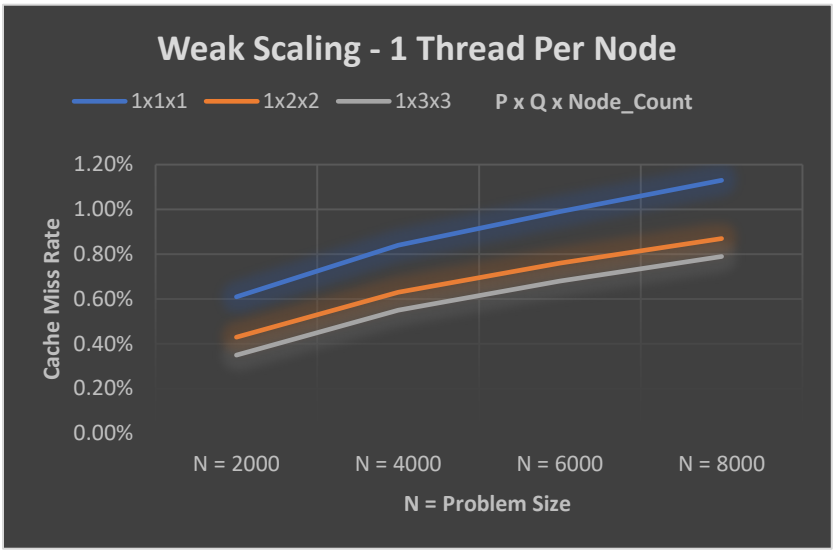

Figure 1.1.1 Cache Miss Rate
Weak Scaling
1 Thread Per Node

Results of Cache Miss Rate when running program on 1 thread per node for problem sizes 2000-8000

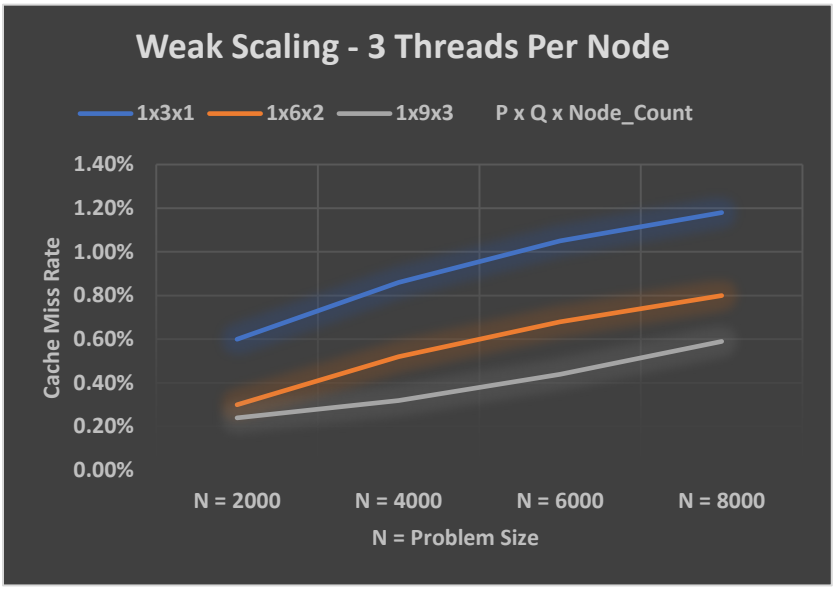

Figure 1.1.2 Cache Miss Rate
Weak Scaling
3 Threads Per Node

Results of Cache Miss Rate when running program on 3 threads per node for problem sizes 2000-8000

The charts below display strong scaling results for cache miss rate on problem sizes 2000 and 8000.

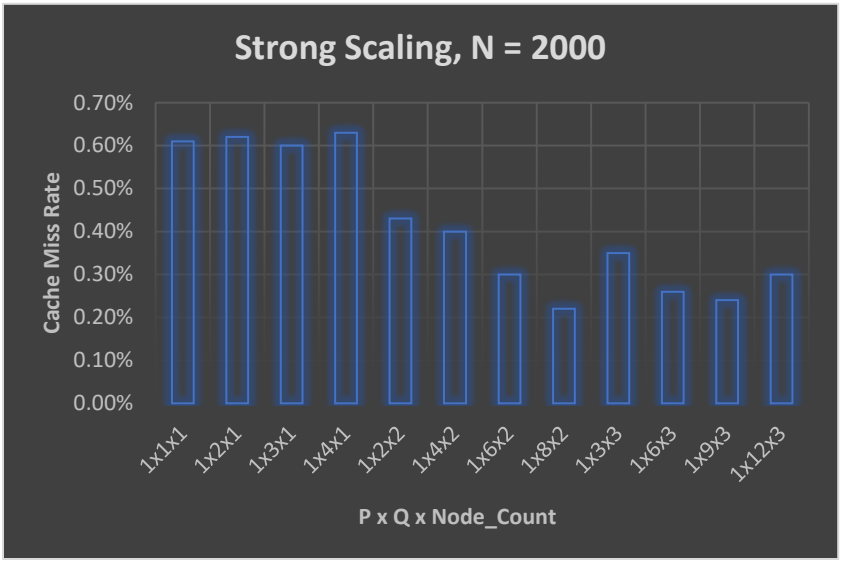

Figure 1.1.3 Cache Miss Rate
Strong Scaling
Problem Size - 2000

Results of Cache Miss Rate when running program on a problem size of 2000 with increasing nodes/threads

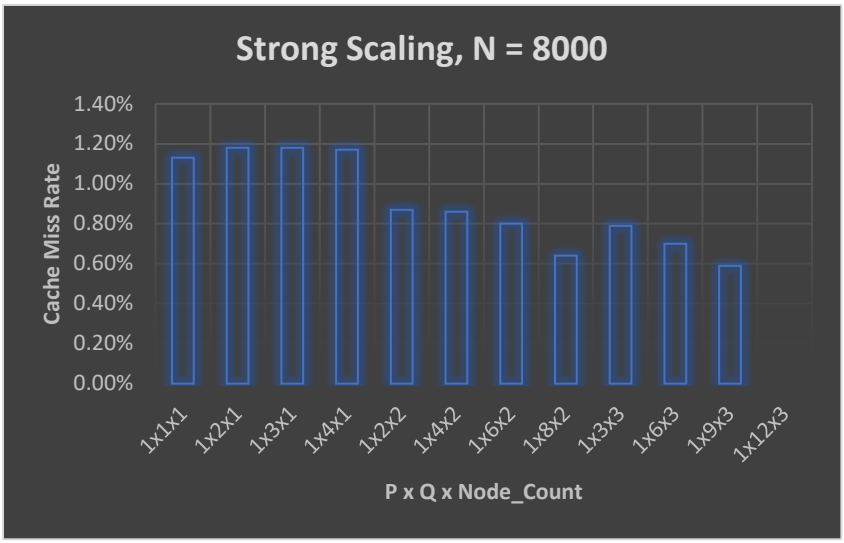

Figure 1.1.4 Cache Miss Rate
Strong Scaling
Problem Size - 8000

Results of Cache Miss Rate when running program on a problem size of 8000 with increasing nodes/threads

II. CPU Execution Time

CPU execution time is the time a computer system spends on a given program, ignoring other system factors. For this metric, we used perf stat to capture the time for each iteration of the program. Below are charts displaying weak scaling results of CPU Execution Time over each problem set with one thread per node versus three threads per node.

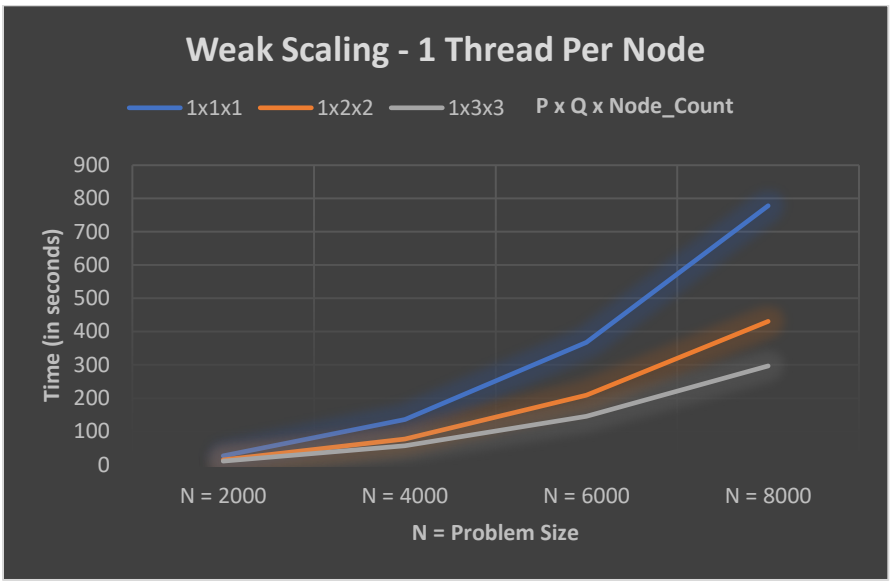

Figure 1.2.1 CPU Execution Time
Weak Scaling
1 Thread Per Node

Results of CPU Execution Time when running program on 1 thread per node for problem sizes 2000-8000

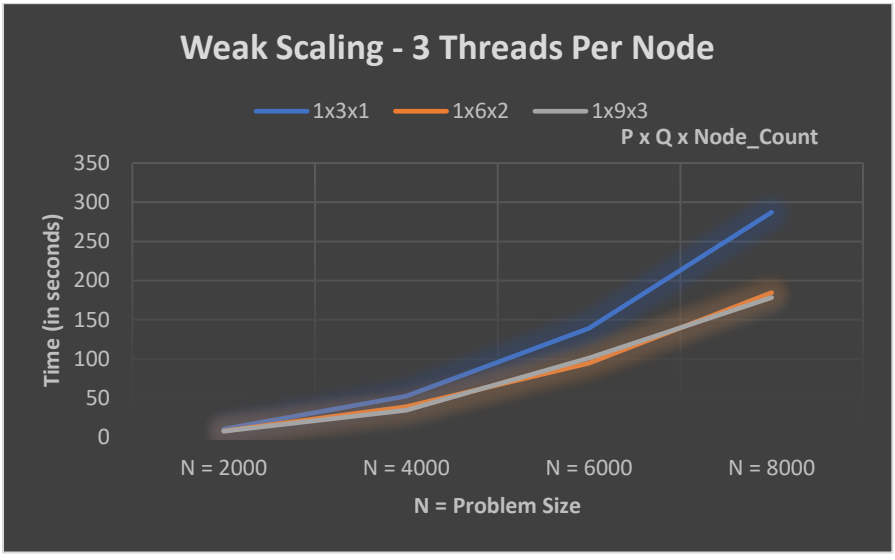

Figure 1.2.2 CPU Execution Time
Weak Scaling
3 Threads Per Node

Results of CPU Execution Time when running program on 3 threads per node for problem sizes 2000-8000

We also examined strong scaling for CPU Execution Time. Those results are below, using problem sizes 2000 and 8000.

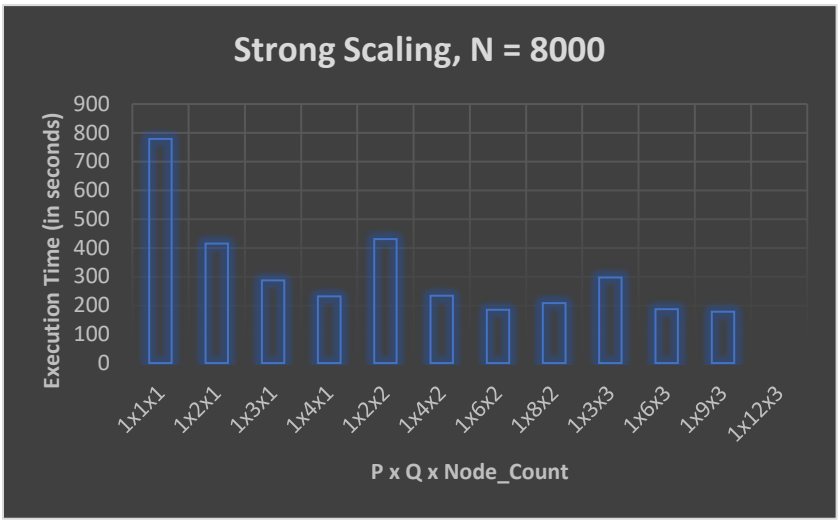

Figure 1.2.3 CPU Execution Time
Strong Scaling
Problem Size - 2000

Results of CPU Execution Time when running program on a problem size of 2000 with increasing nodes/threads

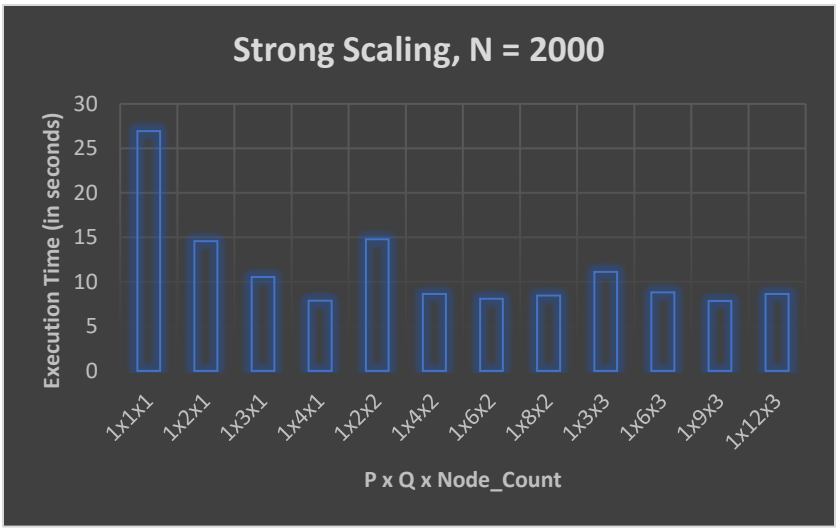

Figure 1.2.4 CPU Execution Time
Strong Scaling
Problem Size - 8000

Results of CPU Execution Time when running program on a problem size of 8000 with increasing nodes/threads

III. GFLOPs

GFLOPs refers to the number of billion floating-point operations were completed each second. The HPL Benchmark is used specifically for this measurement on computer systems. Below are the weak scaling results for the cluster.

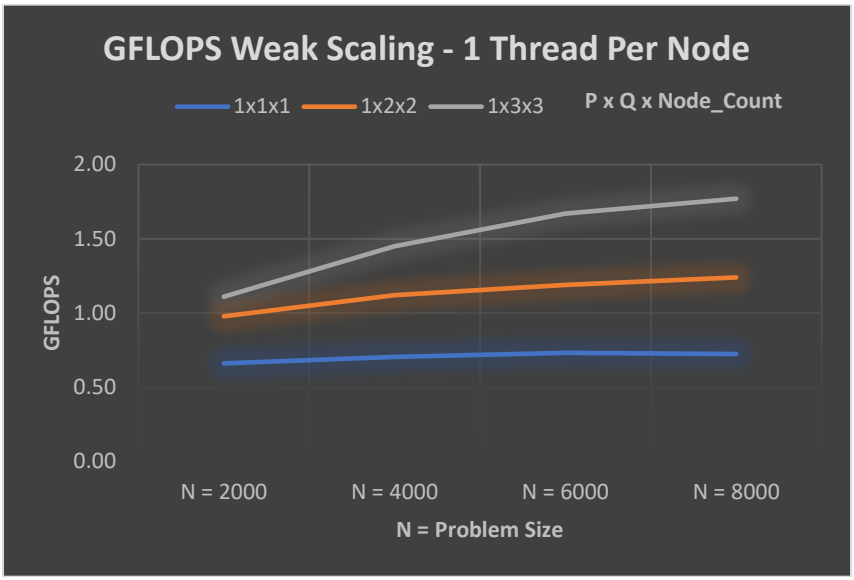

Figure 1.3.1 GFLOPs
Weak Scaling
1 Thread Per Node

Results of GFLOPs when running program on 1 thread per node for problem sizes 2000-8000

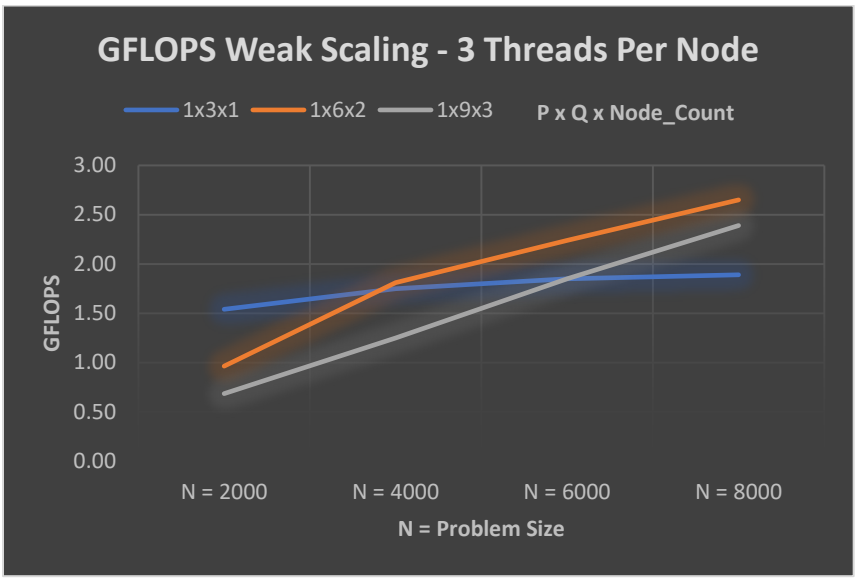

Figure 1.3.2 GFLOPs
Weak Scaling
3 Threads Per Node

Results of GFLOPs when running program on 1 thread per node for problem sizes 2000-8000

The charts for strong scaling of GFLOPs are below. A sample of data was used to demonstrate speedup using two nodes and two threads as the basis of comparison for a small and large problem size. Data for two nodes was used as the sample because data for four threads per node across three nodes was unavailable.

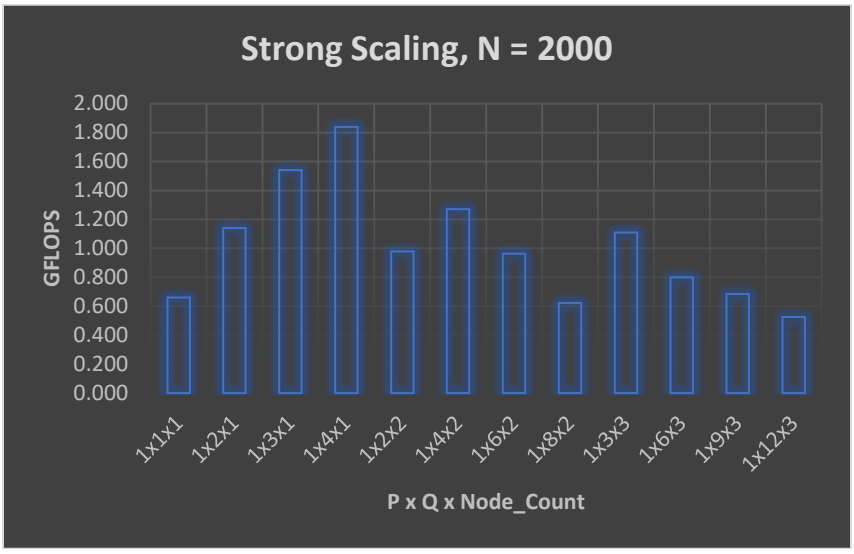

Figure 1.3.3 GFLOPs
Strong Scaling
Problem Size - 2000

Results of GFLOPs when running program on a problem size of 2000 with increasing nodes/threads

2 Nodes	N = 2000
# of Threads	Speedup
2	1.00
4	1.30
6	0.98
8	0.44

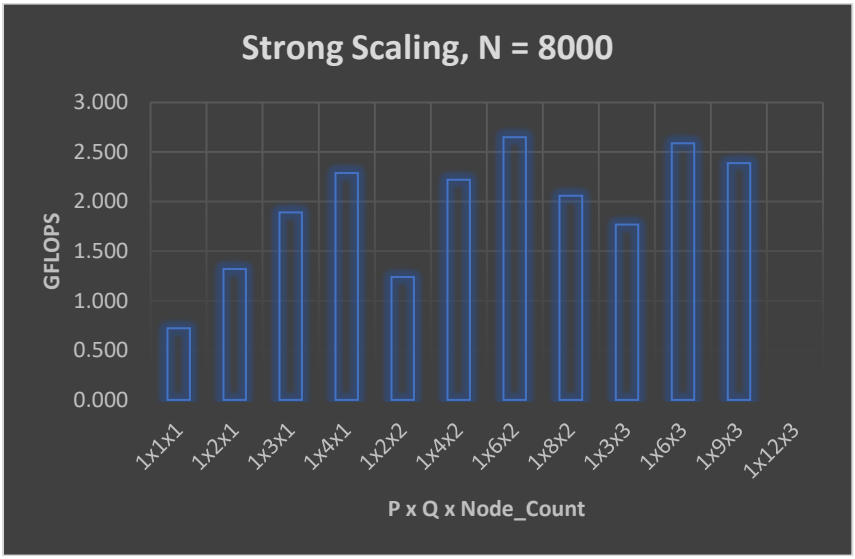

Figure 1.3.4 GFLOPs
Strong Scaling
Problem Size - 8000

Results of GFLOPs when running program on a problem size of 8000 with increasing nodes/threads

2 Nodes	N = 8000
# of Threads	Speedup
2	1.00
4	1.79
6	2.14
8	1.66

IV. Bandwidth

Bandwidth refers to the amount of memory accessed per a unit of time. We calculated this by taking the size of the problem as it relates to memory usage per HPL's documentation and dividing it by the time it took for the program to complete. The weak scaling results are shown below.

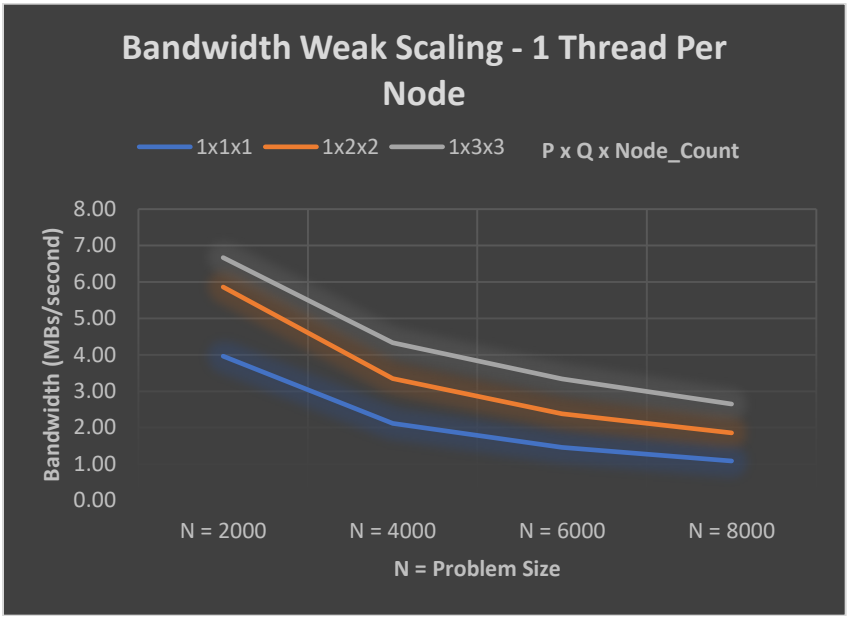

Figure 1.4.1 Bandwidth Weak Scaling 1 Thread Per Node

Results of Bandwidth when running program on 1 thread per node for problem sizes 2000-8000

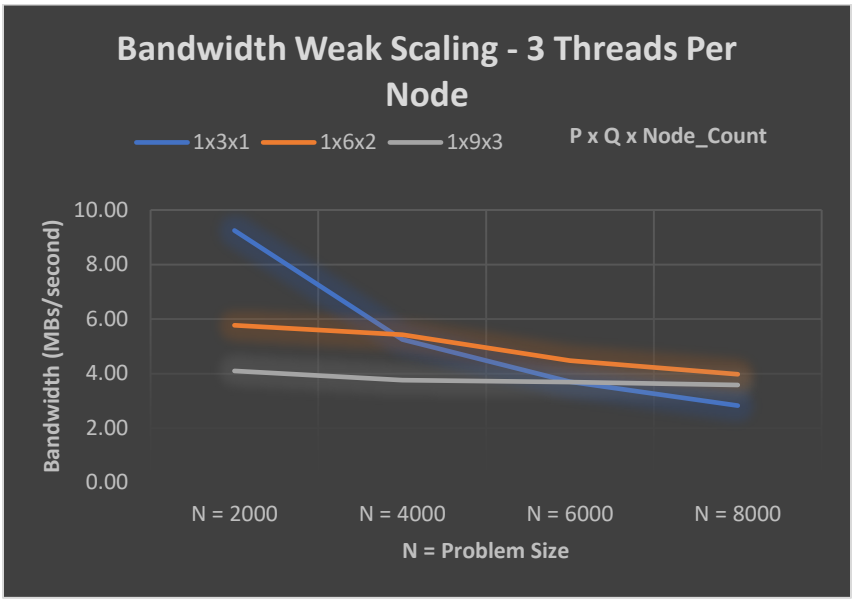

Figure 1.4.2 Bandwidth Weak Scaling 3 Threads Per Node

Results of Bandwidth when running program on 3 threads per node for problem sizes 2000-8000

The results of strong scaling experiments done regarding bandwidth are below.

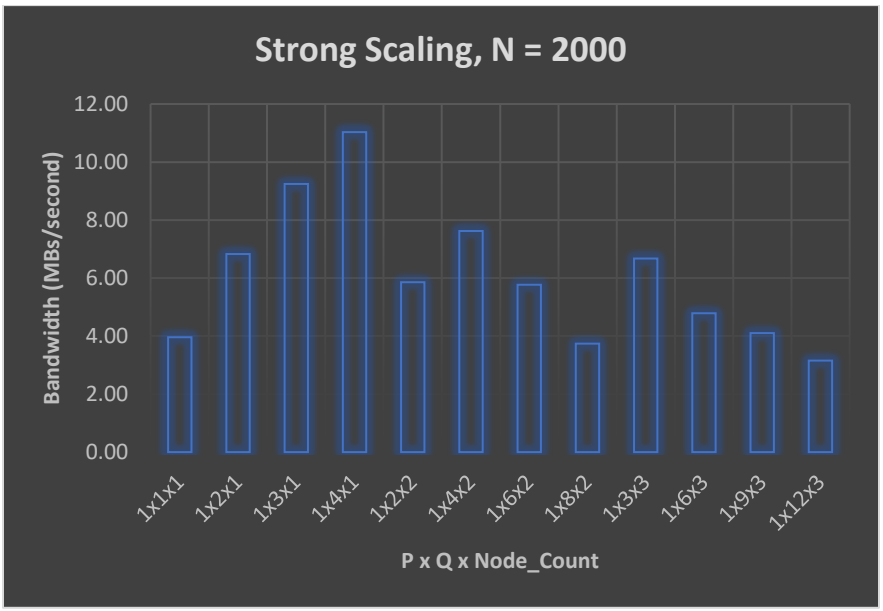

Figure 1.4.3 Bandwidth
Strong Scaling
Problem Size - 2000

Results of Bandwidth when running program on a problem size of 2000 with increasing nodes/threads

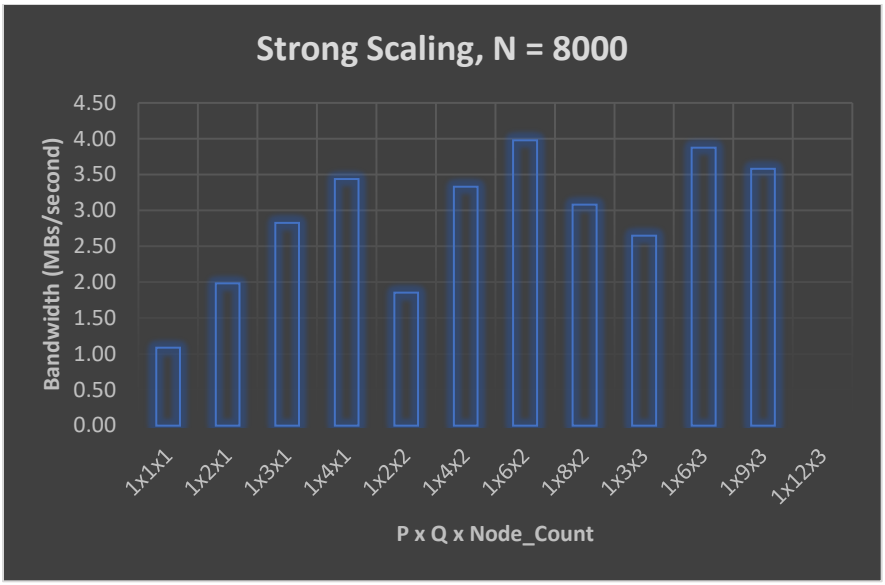

Figure 1.4.4 Bandwidth
Strong Scaling
Problem Size - 8000

Results of Bandwidth when running program on a problem size of 8000 with increasing nodes/threads

Conclusions

In this section, we present our conclusions based on the metrics tested throughout our experiment. Generally, we found that at a small problem size a single node performed better; once the problem size became adequately large, we achieved speedup.

I. Cache Miss Rate

The cache miss rates stayed consistent throughout all the testing. As the problem size increased, so too did cache misses. This outcome was predictable since a larger problem required more interaction with data, resulting in more overall misses. However, there was a decrease in misses when more nodes were active. This is possible due to each node taking less of the problem resulting in fewer misses. It should also be stated that the difference between two and three nodes working in parallel resulted in a trivial difference in miss rate. We assume this is due to the miss rate reaching such a low percentage that it has bottomed out and will not get lower. Increasing threads to an individual node didn't affect misses, as they had to access the data regardless of how many threads were working on the problem.

II. CPU Execution Time

Like cache miss rates, as the problem size increased it required more time to complete the task. The increase of parallel programming from one to three nodes didn't change the execution time of small problem sizes. This outcome is not surprising since a small problem does not take much time to complete. As the problem increased, there was a clear indication that multiple nodes have a faster execution time than a single node.

III. GFLOPs

The GFLOPs for an individual node increase as the threads increase for a low problem size. When the number of nodes and threads increase the network actually loses some performance small problem sizes. We began to see speedup when the problem size neared 8000. We suspect that this is because the reported time is skewed by network latency, as it is a large portion of the execution time for small problem sizes. We were unable to gather data for problem sizes greater than 8000 because of hardware constraints. We suspect larger problem sizes would have shown performance results more in line with expectations as the communication time was marginalized. We anticipate the best performance comparison would require tests of a problem size of 10000 or greater to see more accurate results.

IV. Bandwidth

Bandwidth had similar results to GFLOPs, where it showed favoritism to individual nodes at small problem sizes. The results started to favor multiple nodes at higher problem sizes. Again, we anticipate that a smaller step size at a larger problem size could confirm this, but our cluster was not able to successfully run problem sizes larger than 8000 on multiple nodes.

References

[1] “Raspberry Pi 3 Model B” *Raspberry Pi*, www.raspberrypi.org/products/raspberry-pi-3-model-b/.

[2] “Raspberry Pi 3 Model B+” *Raspberry Pi*, www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/.

[3] Braun, Kurt. “Raspberry Pi 3 Cluster Computer - MPICH2 & MPI4Py.” YouTube, 10 Aug. 2016, www.youtube.com/watch?time_continue=48&v=vflPGqcXJkY.

[4] Cox, Simon. *Steps to Make Raspberry Pi Supercomputer*, www.southampton.ac.uk/~sjc/raspberrypi/pi_supercomputer_southampton.htm.

[5] *MPICH*, www.mpich.org/static/downloads/3.2.1/.

[6] *HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers*, www.netlib.org/benchmark/hpl/.

[7] “Building HPL and ATLAS for the Raspberry Pi.” *Compute Nodes*, computenodes.net/2018/06/28/building-hpl-an-atlas-for-the-raspberry-pi/.

[8] “Automatically Tuned Linear Algebra Soft.” *SourceForge*, sourceforge.net/projects/math-atlas/.

[9] Ren, Jingyao. “Compile Perf on Raspberry Pi, Mac and Ubuntu.” *Dummy Coder's Dummy Code*, jireren.github.io/blog/2016/09/19/Compile-Perf/.