Jesi Merrick
Ron Pyka
CS553-01

Design Document

Introduction

For this assignment, we started by reviewing Ron Pyka's code from CS451. Jesi Merrick improved and added to the previous CPU, memory, and disk benchmarks, while Ron Pyka improved on the previous network benchmark and developed the GPU benchmark from scratch. Additionally, all the results from below were from a machine of the following specifications:

Operating System: Ubuntu 12.04 LTS 64-bit
Processor: AMD A8-4500M APU (1900MHz 4 cores)
Memory: 8 GiB 1600MHz DDR3
Harddrive: 750 GB spinning disk
Network: Gigabit integrated ethernet

The results of the GPU benchmark are from it being ran on Jarvis.

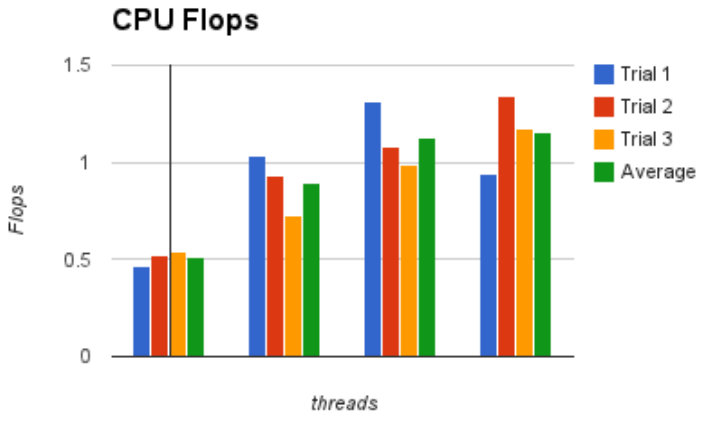
We have also included all of our data (including standard deviations) in an Excel spreadsheet.

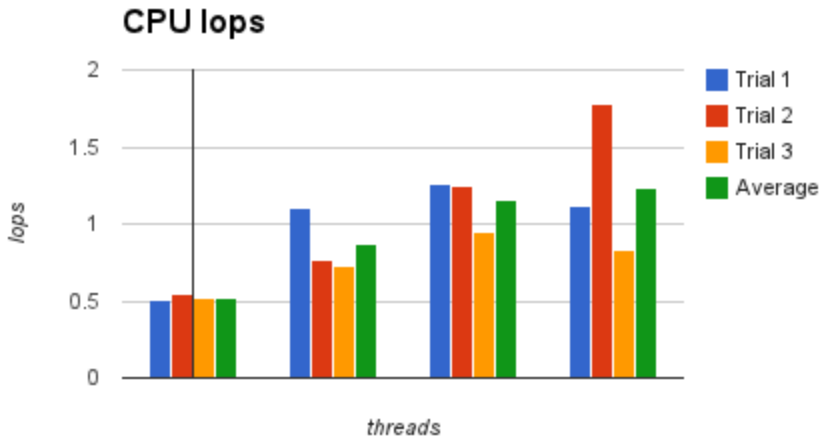
CPU Benchmark (Jesi Merrick)

Design

The CPU benchmark was the simplest to complete. A set of floats and a set of ints are randomly generated, to be use in the operations.

Results





Analysis/Theoretical

By the average, four threads provides the best performance, followed closely by three threads.

Improvements/Extensions

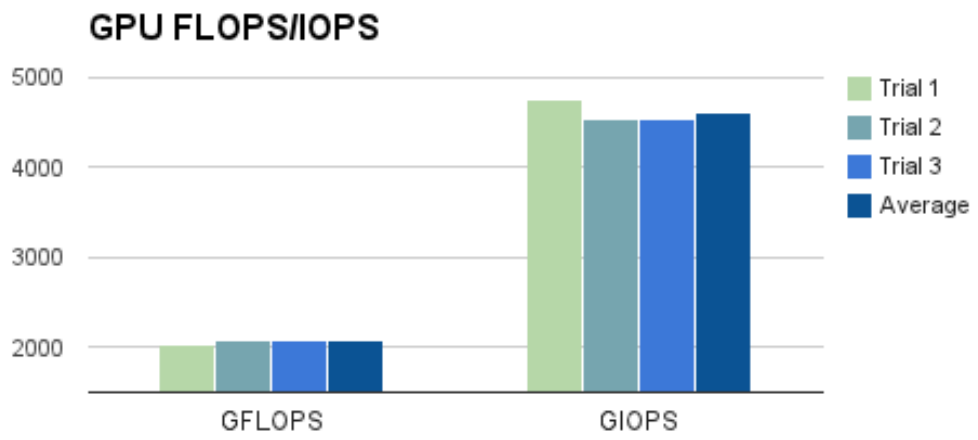
We don't think we would improve on this much.

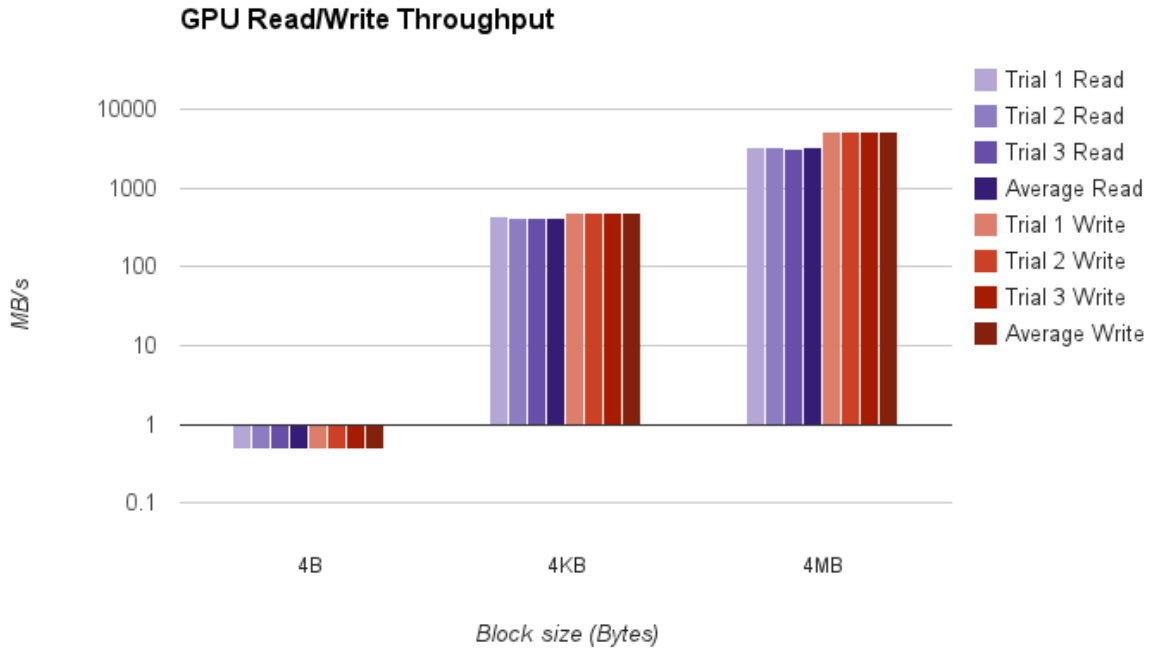
GPU Benchmark (Ron Pyka)

Design

This was a simple design, where for each block size, that amount of data was written to the GPU memory for some number of times (so that it took a reasonable amount of time) to measure write speed, and then the data was copied back for a number of iterations for read speed. To measure the number of operations per second, arrays of both floats and integers were copied over, and basic float/integer operations were performed on each. The time for the functions to complete could be used to calculate the number of operations per second.

Results





Analysis/Theoretical

Not sure what the theoretical would be for Jarvis' hardware.

Improvements/Extensions

More time to test, more information on Jarvis, and possibly running on different cards for comparison all could have helped.

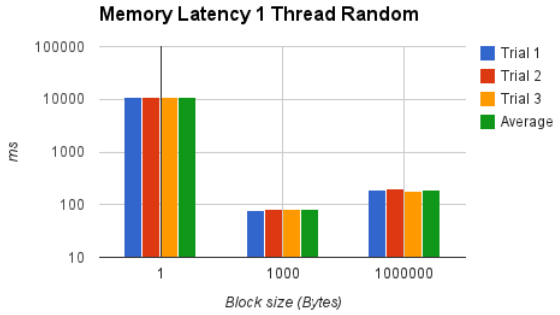
Memory Benchmark (Jesi Merrick)

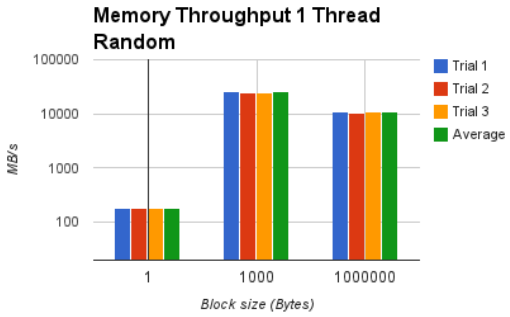
Design

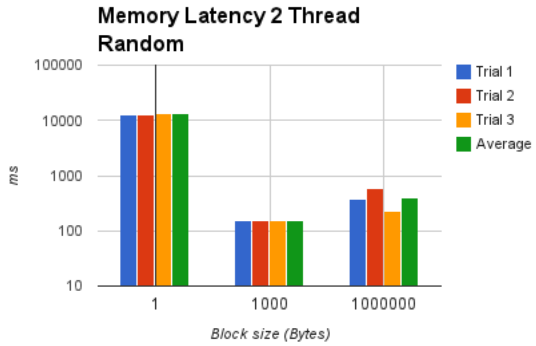
This benchmark was completed using CS451 code. However, it was necessary to implement sequential read and write, as well as improve the previous random read and write code. The main improvements to random read and write was to limit the code to only timing the actual operations - previously the malloc and free actions were include within the timer. These were moved outside of the timer to improve accuracy. The read and write actions themselves were implemented using memcopy.

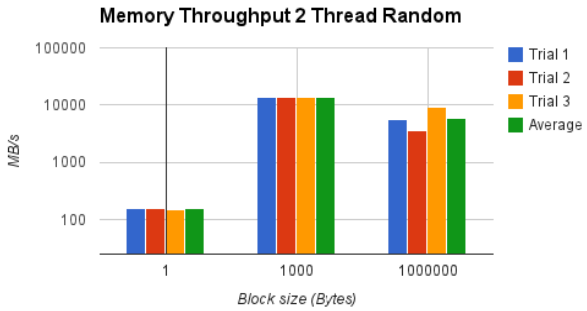
Sequential read and write was implementing by calling malloc with a much larger size, and then using pointer variables to move by a single block size at a time through the two pieces of memory, using memcopy at each step (from block A to block B).

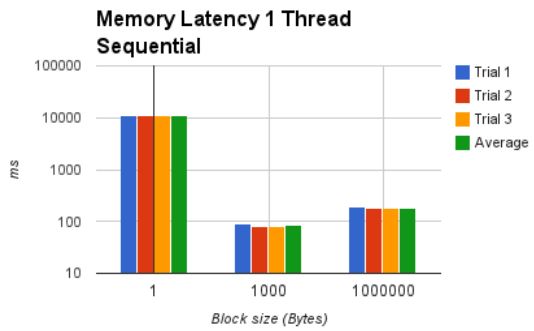
Results

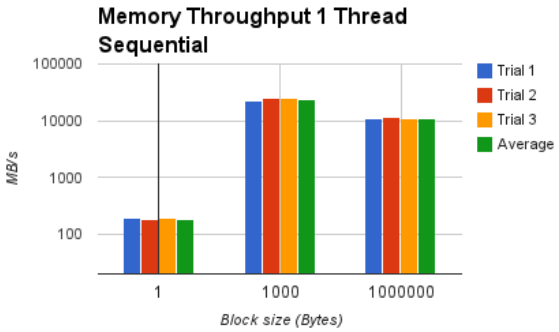


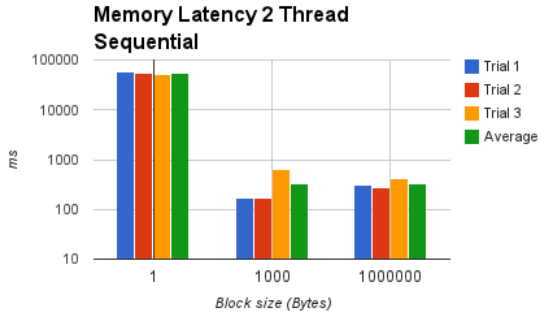


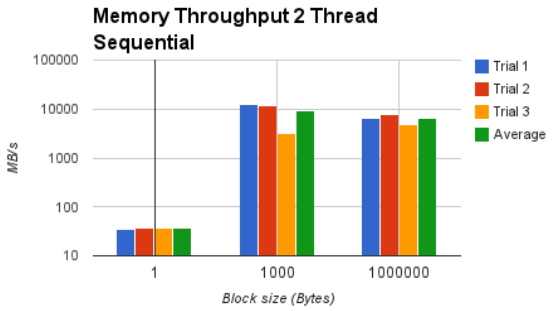












Analysis/Theoretical

Our memory tests showed no improvement from 1 to 2 threads, and actually some loss, likely due to memory being able to be written by one thread at max speed, and two threads needing to switch or share. other processes on the machine could have interfered as well.

Improvements/Extensions

Cannot think of any improvements to this.

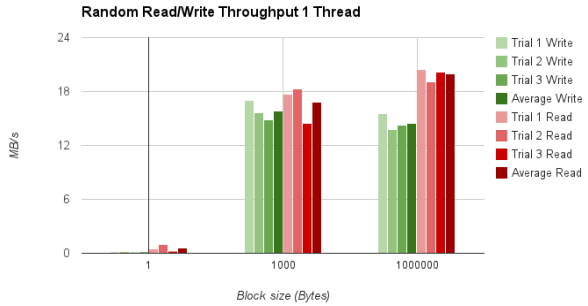
Disk Benchmark (Jesi Merrick)

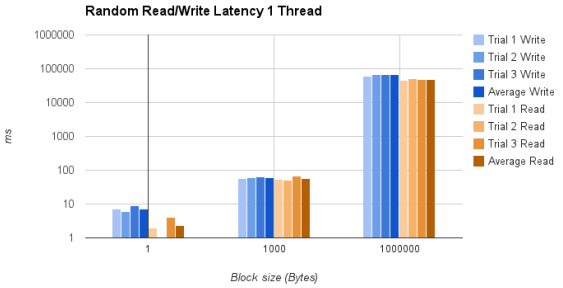
Design

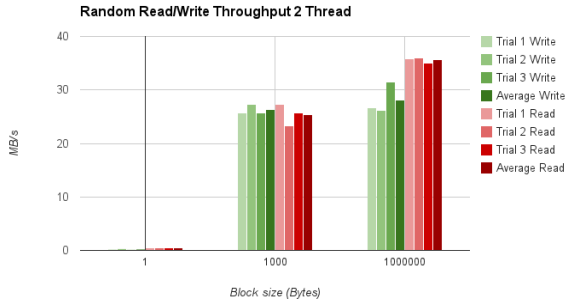
This benchmark was completed using CS451 code. However, it was necessary to implement sequential read, as well as random read and write, as the code only dealt with sequential write previously. Both sequential write and read are straight-forward. For sequential write, a new file is created and written to multiple times in the appropriate block sizes. Sequential read then uses this file, to read through.

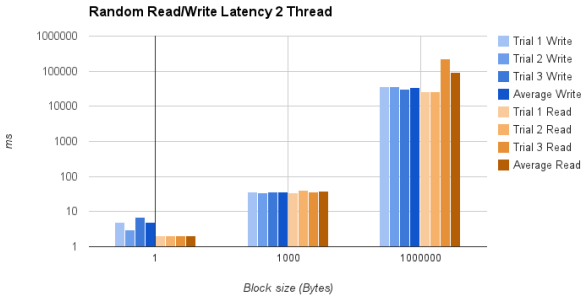
For random write and read, a set of random integers were generated. These integers are used with fseek to randomly move the pointer through the document, and then either write or read (depending on which parameter is being tested). Both random write and read use the file initially created by sequential write, which is an appropriate size.

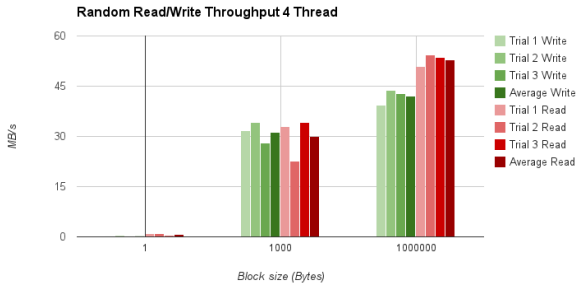
Results

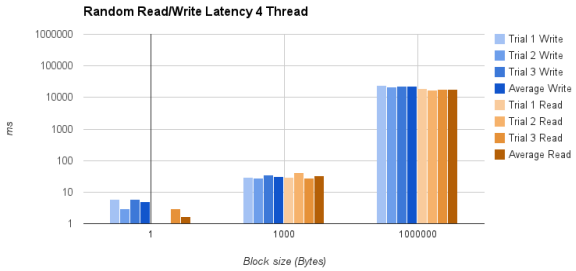

Random Read/Write Throughput 1 Thread
MB/s
24
18
12
6
0
1
1000
1000000
Block size (Bytes)
Trial 1 Write
Trial 2 Write
Trial 3 Write
Average Write
Trial 1 Read
Trial 2 Read
Trial 3 Read
Average Read

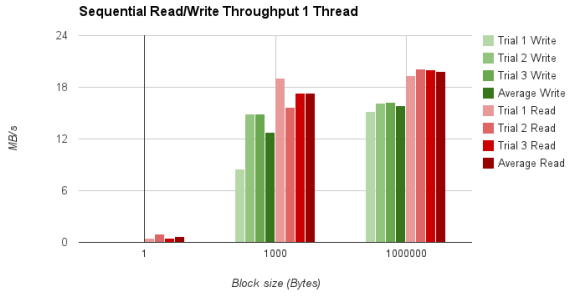

Random Read/Write Latency 1 Thread
ms
1000000
100000
10000
1000
100
10
1
1
1000
1000000
Block size (Bytes)
Trial 1 Write
Trial 2 Write
Trial 3 Write
Average Write
Trial 1 Read
Trial 2 Read
Trial 3 Read
Average Read

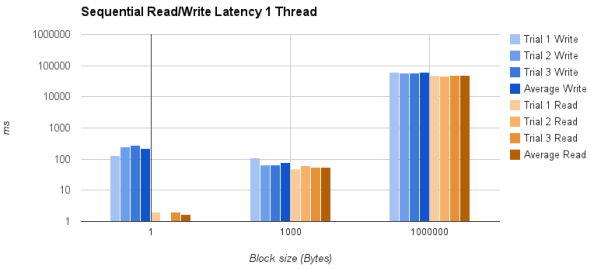

Random Read/Write Throughput 2 Thread
MB/s
40
30
20
10
0
1
1000
1000000
Block size (Bytes)
Trial 1 Write
Trial 2 Write
Trial 3 Write
Average Write
Trial 1 Read
Trial 2 Read
Trial 3 Read
Average Read

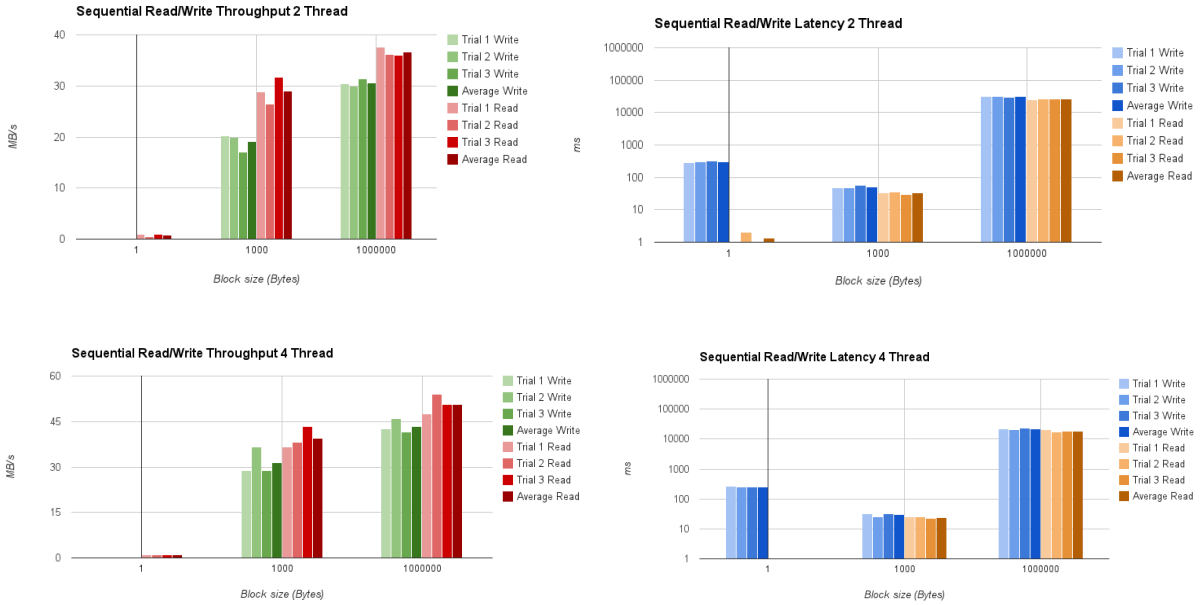

Random Read/Write Latency 2 Thread
ms
1000000
100000
10000
1000
100
10
1
1
1000
1000000
Block size (Bytes)
Trial 1 Write
Trial 2 Write
Trial 3 Write
Average Write
Trial 1 Read
Trial 2 Read
Trial 3 Read
Average Read


Random Read/Write Throughput 4 Thread
MB/s
60
45
30
15
0
1
1000
1000000
Block size (Bytes)
Trial 1 Write
Trial 2 Write
Trial 3 Write
Average Write
Trial 1 Read
Trial 2 Read
Trial 3 Read
Average Read


Random Read/Write Latency 4 Thread
ms
1000000
100000
10000
1000
100
10
1
1
1000
1000000
Block size (Bytes)
Trial 1 Write
Trial 2 Write
Trial 3 Write
Average Write
Trial 1 Read
Trial 2 Read
Trial 3 Read
Average Read


Sequential Read/Write Throughput 1 Thread
MB/s
24
18
12
6
0
1
1000
1000000
Block size (Bytes)
Trial 1 Write
Trial 2 Write
Trial 3 Write
Average Write
Trial 1 Read
Trial 2 Read
Trial 3 Read
Average Read


Sequential Read/Write Latency 1 Thread
ms
1000000
100000
10000
1000
100
10
1
1
1000
1000000
Block size (Bytes)
Trial 1 Write
Trial 2 Write
Trial 3 Write
Average Write
Trial 1 Read
Trial 2 Read
Trial 3 Read
Average Read



Analysis/Theoretical

By the average, four threads showed the best performance, and both 2 and 4 threads were similar in performance. The disk might have had multiple arms enabling it to write in two locations at once, allowing that spike in efficiency.

Improvements/Extensions

We could probably have tried other libraries, or looked into some way to further reduce overhead.

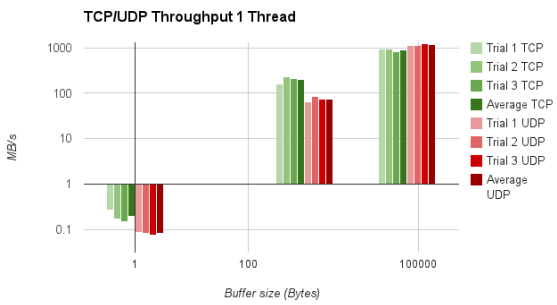
Network Benchmark (Ron Pyka)

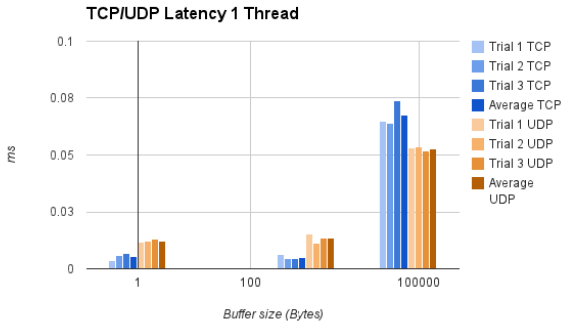
Design

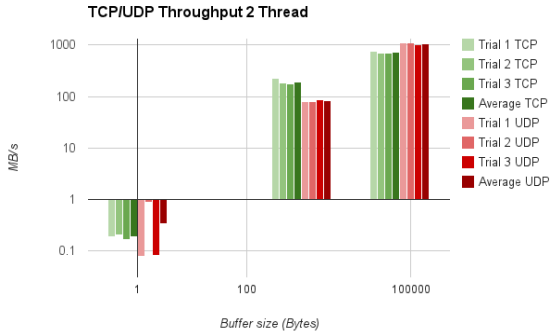
The network benchmark started with the existing 451 code for building individual packets in C++, but as that seemed fairly inefficient, we decided to switch to sockets instead. With the sockets, we first establish a TCP “server” with a listening socket, then a TCP “client” that sends an initial message used to establish a connection using a new socket on the server. From there we have the client send several buffers full of data to the server and time that to determine throughput and latency. We then do similar for UDP, but instead of establishing a connection on a new socket (since UDP is connectionless) the server just listens on the initial socket.

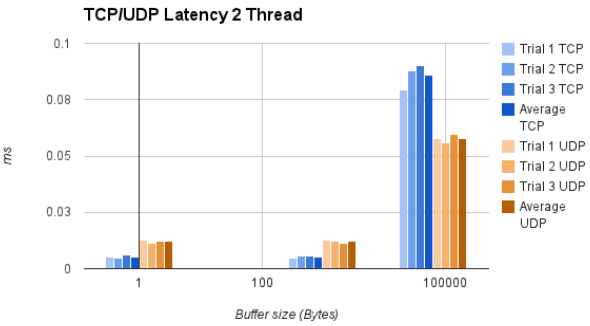
We hit a lot of problems with the network code in getting it working, and still had two when we finished. One was that the UDP server sometimes misses some data and will wait to fill it’s buffer even after the client has finished sending, and the program will hang waiting. The other is that our system did not like establishing and closing sockets repeatedly, and would block connections made too quickly after others were closed, forcing us to make our program take parameters so we could wait between runs.

Results









Analysis/Theoretical

Since the machine we tested on had a gigabit interface, we could expect 125 MB/s, or if the interface allows reading/writing on the second set of wires, would theoretically reach 250MB/s. But since the testing was run on the loopback interface, and never actually has to transmit across the physical network, the process might simply be CPU bound, and only take the time needed to put the data in packets and then open it up again, which could explain our higher speeds.

Improvements/Extensions

This could have been improved by resolving the connection refused errors and figuring out how to successfully exit the UDP server.