Benchmarking - Design Document

In this assignment, we have to benchmark different parts of a computer system – CPU, memory, disk and network as described in the manual.

CPU

In CPU, I have used around 22 Integer operations and around 77 floating point operations which are iterating for a loop of 10^9. And for the second experiment I have used same no. of instructions and iterated through a loop of 10 minutes to plot 600 points for each IOPS and FLOPS with 4 threads. I am taking no. of threads as command line arguments.

Program Flow:

Main Class:

1. Get the no. of threads from command line arguments
2. Create a thread array with that many no. of threads
3. Iterate the array to initialize each thread of another inner class "Nthread" and start them.

Nthread Class:

1. Perform integer operations for loop of 10^9 and display the IOPS.
2. Perform floating point operations for loop of 10^9 and display the FLOPS.

I have create similar java file for second experiment which will have 4 threads and it will perform operations for 10 minutes and save the output in a text file.

DISK

In Disk, I have created four different functions for random and sequential read and write.

For sequential, I am writing the data into a byte array and then writing into a file and for reading, the data will read and store in a byte array.

For random, I am writing the data into a byte array and storing it at a random location using seek() of RandomAccessFile class of Java. For random read, I am seeking to a random location and reading the data.

I have created an array for different block size which will be iterated to read different block size of data automatically.

In this experiment as well, I am taking no. of threads in the command line arguments.

Program Flow:

Main Class:

1. Get the no. of threads from command line arguments
2. Create a thread array with that many no. of threads
3. Create an integer array for block size and initialize it with the different block size as mentioned.
4. Iterate the block size array to iterate thread array which will initialize each thread of another inner class "Nthread" and start them.

Nthread Class:

1. Create a file object with the input file on which read write needs to be performed
2. Call random write and random read functions which will do their job and print the result
3. Call sequential write and read functions which will also perform their job and print the result.

NETWORK

In Network, I have created two different java files for server and client. The server will send the files of different sizes to the client and client will receive them. I have used both TCP and UDP connections to data to the client.

As required by the professor, I have created equal no. of threads at both the ends for 1:1 mapping.

Program Flow:

Server

Main Class:

1. Create file objects of files which needs to be sent
2. Create Server Socket and establish connection with the client
3. Create input/output stream of the client to perform file operations
4. Get the no. of threads from command line arguments
5. Create a thread array with that many no. of threads
6. Create an integer array for block size and initialize it with the different block size as mentioned
7. Iterate the block size array to iterate thread array which will initialize each thread of another inner class “Nthread” and start them.

Nthread Class:

1. Create a byte array (buffer) of block size which needs to be receive
2. Read the data from the client using client’s input stream and store it in the buffer
3. Write the data onto the client side using output stream
4. Receive the packet using Datagram socket into a buffer
5. Create a Datagram packet with the same buffer and client’s ipaddress and port no.
6. Send the datagram packet to the client.

Similar program flow is done at the client side.

Client

Main Class:

1. Create file objects of files to receive from server
2. Create Socket and establish connection with the server
3. Create input/output stream of the client to perform file operations
4. Get the no. of threads from command line arguments
5. Create a thread array with that many no. of threads
6. Create an integer array for block size and initialize it with the different block size as mentioned

7. Iterate the block size array to iterate thread array which will initialize each thread of another inner class “Nthread” and start them.

Nthread Class:

1. Create a byte array (buffer) of block size to send data
2. Send the data using server’s output stream stored in the buffer
3. Read the data using server’s input stream and store into a file
4. Record the time and display TCP result
5. Create a Datagram packet with data and server’s ipaddress and port no.
6. Send datagram packet to the server
7. Receive the packet using Datagram socket and save it in the buffer
8. Record the time and display UDP result

Tradeoffs made:

I used Java as a programming language to code to avoid memory issues with C. Since C is faster than Java I could have achieved much better results. The code is very straight forward and can be made more dynamic and to achieve maximum utilization for benchmarking. I have used multithreading and avoided code redundancy as far as I can.

I also learned a lot of new things in this assignment such as thread barrier in Java and 1:1 mapping of threads.

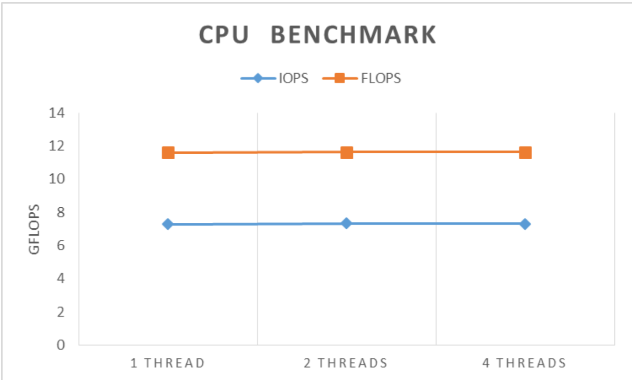
Performance Evaluation

In this experiment, I have performed various experiment on t2 micro instance of Amazon AWS to benchmark different parts of a computer system such as CPU, Disk, Memory and Network.

CPU BENCHMARK

The theoretical performance of CPU on t2 instance is 10 GFLOPS since it's a single core with 2.5GHZ of clock speed and 4 IPC (Instruction per cycle). I have achieved 70% of the theoretical performance in integer operations and 40% in floating operations of it. As per my evaluation with increase in no. of threads, the summation of IOPS and FLOPS for all the threads give me the same result which I get with a single thread which is as expected since CPU performs same no. of integer and floating operations per second in IPC.

Below is the graph for CPU benchmark followed by the table which shows by varying number of threads, I have achieved same no. of GIOPS and GFLOPS.



	1 Thread	2 Threads	4 Threads
IOPS	7.272209	7.32087	7.305676
FLOPS	4.324515	4.310709	4.322285

<u>Standard Deviation & Average</u>

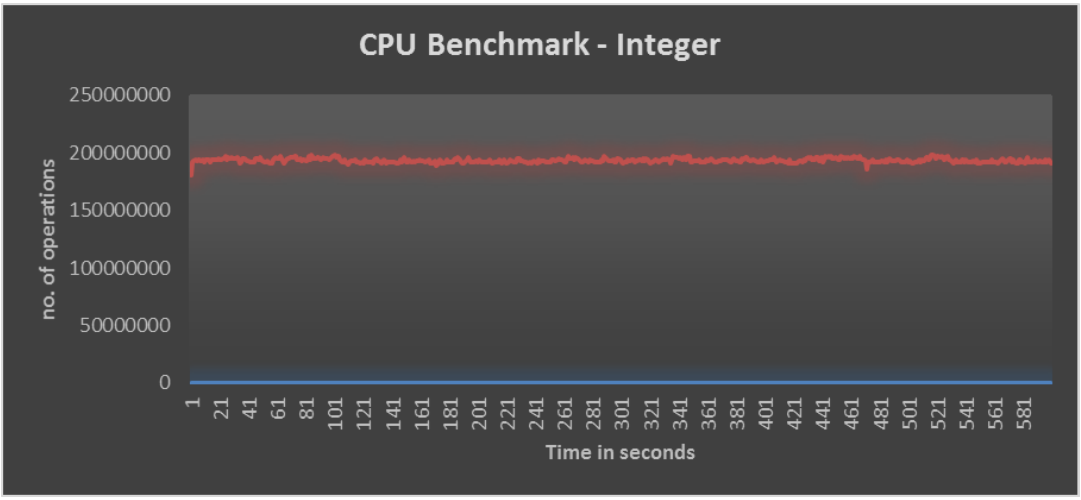
I performed the same operations 3 times to compute average and standard deviation. As per the below table, there is <0.5 of deviation amongst the three results. And the average of the three test scenarios falls under the range of 7.27-7.9 GIOPS and 4.3 – 4.7 GFLOPS

	Standard Deviation		
IOPS	0.545865	0.501169	0.563036
FLOPS	0.34183	0.366418	0.355247

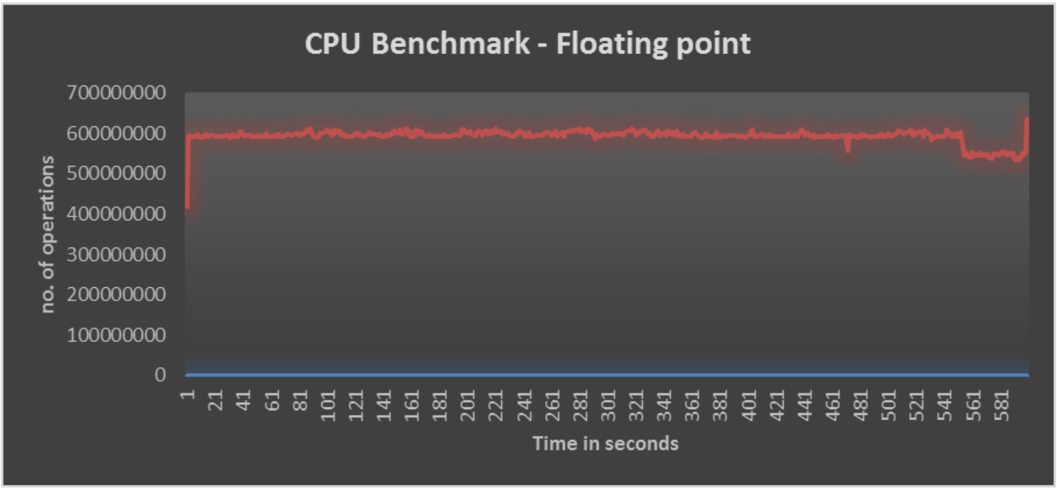
	Average		
IOPS	7.902516	7.898927	7.95409
FLOPS	4.718777	4.733654	4.732091

Experiment 2:

For experiment 2, I have performed the same no. of operations for 10 minutes and traced the no. of operations performed every sec. The below graph shows the integer operations performed per second for 10 minutes. Ideally, it should be a straight line but since we cannot get 100% CPU utilization the graph looks little distorted. But as no. of operations per second remains almost the same I got this output.

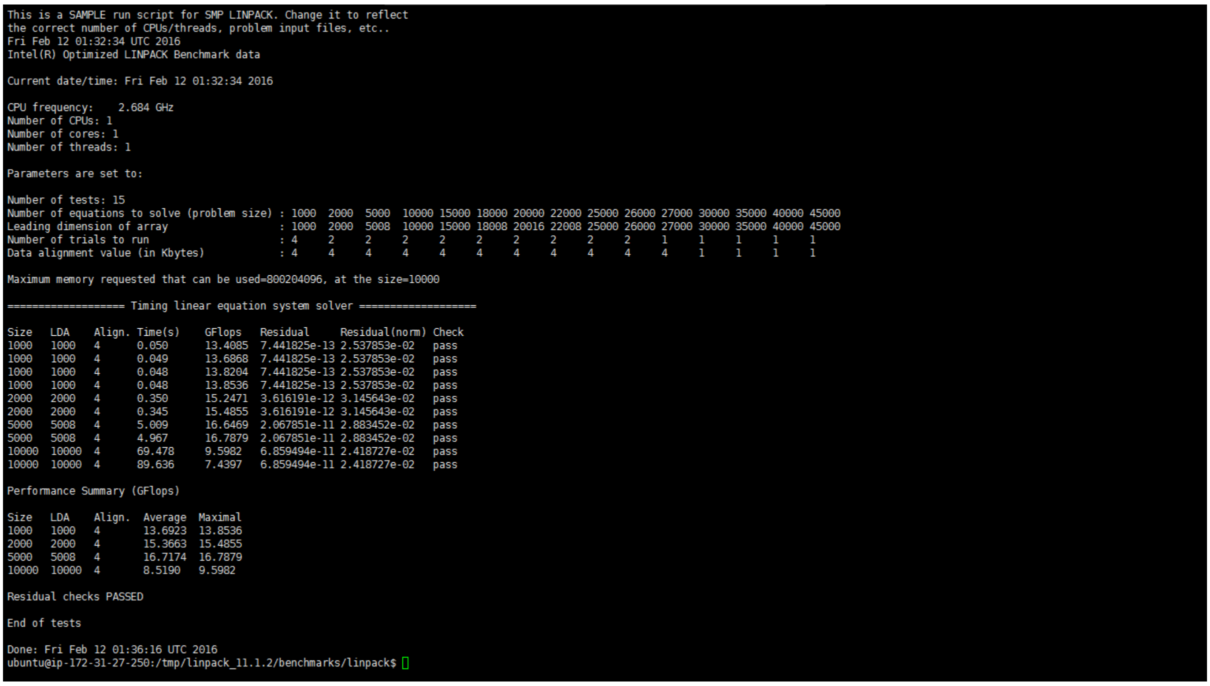


The below graph shows the floating point operations performed per second for 10 minutes. As I mentioned above for integer operations, as no. of operations per second remains almost same I got the below output.



Extra Credit:

I have also ran the Linpack benchmark tool for CPU on AWS and I got maximum of 16 GFLOPS and average 15GFLOPS as it can be seen in screenshot, which is more than 100% since it runs in burst mode by prioritizing the operations.



```
This is a SAMPLE run script for SMP LINPACK. Change it to reflect
the correct number of CPUs/threads, problem input files, etc..
Fri Feb 12 01:32:34 UTC 2016
Intel(R) Optimized LINPACK Benchmark data

Current date/time: Fri Feb 12 01:32:34 2016

CPU frequency:    2.684 GHz
Number of CPUs: 1
Number of cores: 1
Number of threads: 1

Parameters are set to:

Number of tests: 15
Number of equations to solve (problem size) : 1000  2000  5000  10000 15000 18000 20000 22000 25000 26000 27000 30000 35000 40000 45000
Leading dimension of array                  : 1000  2000  5008  10000 15000 18008 20016 22008 25000 26000 27000 30000 35000 40000 45000
Number of trials to run                     : 4     2     2     2     2     2     2     2     2     2     1     1     1     1     1
Data alignment value (in Kbytes)            : 4     4     4     4     4     4     4     4     4     4     4     1     1     1     1

Maximum memory requested that can be used=800204096, at the size=10000

=================== Timing linear equation system solver ===================

Size   LDA    Align. Time(s)    GFlops   Residual     Residual(norm) Check
1000   1000   4      0.050      13.4085  7.441825e-13 2.537853e-02   pass
1000   1000   4      0.049      13.6868  7.441825e-13 2.537853e-02   pass
1000   1000   4      0.048      13.8204  7.441825e-13 2.537853e-02   pass
1000   1000   4      0.048      13.8536  7.441825e-13 2.537853e-02   pass
2000   2000   4      0.350      15.2471  3.616191e-12 3.145643e-02   pass
2000   2000   4      0.345      15.4855  3.616191e-12 3.145643e-02   pass
5000   5008   4      5.009      16.6469  2.067851e-11 2.883452e-02   pass
5000   5008   4      4.967      16.7879  2.067851e-11 2.883452e-02   pass
10000  10000  4      69.478     9.5982   6.859494e-11 2.418727e-02   pass
10000  10000  4      89.636     7.4397   6.859494e-11 2.418727e-02   pass

Performance Summary (GFlops)

Size   LDA    Align.  Average  Maximal
1000   1000   4       13.6923  13.8536
2000   2000   4       15.3663  15.4855
5000   5008   4       16.7174  16.7879
10000  10000  4       8.5190   9.5982

Residual checks PASSED

End of tests

Done: Fri Feb 12 01:36:16 UTC 2016
ubuntu@ip-172-31-27-250:/tmp/linpack_11.1.2/benchmarks/linpack$
```

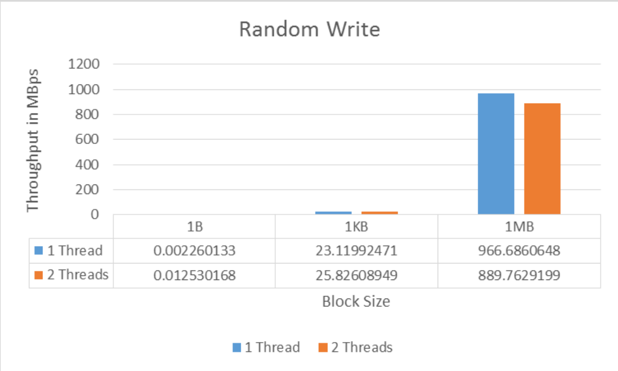
DISK BENCHMARK

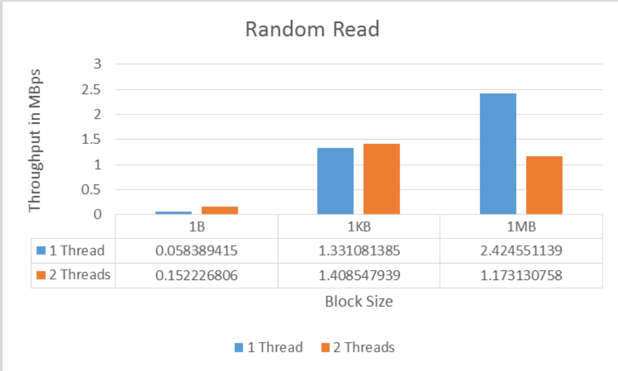
For Disk, I am performing four different ways of read write operations on disk. I am reading and writing randomly and sequentially to the disk for calculating throughput and latency. I have kept file size more than all the cache sizes so that it gives me optimum results.

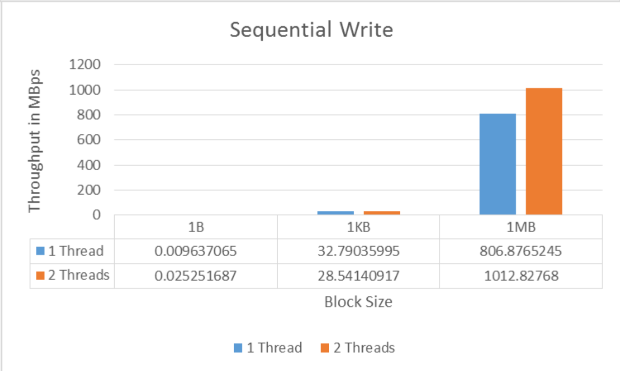
Throughput:

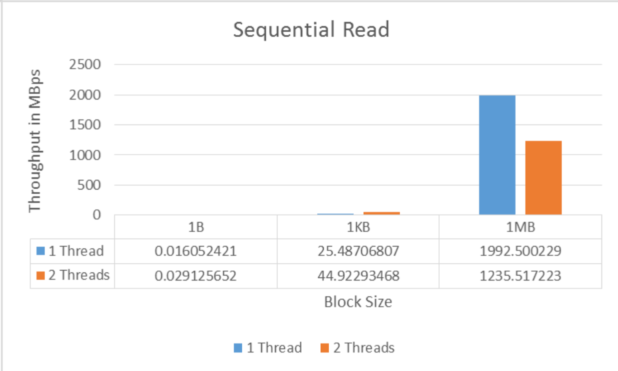
I have calculated the throughput in Mega Bytes per second (MBps). As per my evaluation, for random read and write I am getting throughput of around 900MBps and 2MBps to transfer 1MB of file and 25MBps and 1MBps for 1KB of file. As it can be seen from the figure as well, as I increase no. of threads the speed remains almost constant for 1KB but it slightly decreases for 1MB.

For sequential read and write I am getting throughput of around 800MBps and 2000MBps to transfer 1MB of file and 30MBps and 25MBps to transfer 1KB of file. As it can be seen from the graph with increase in no. of threads the speed has been decreased for read but increased for the write.



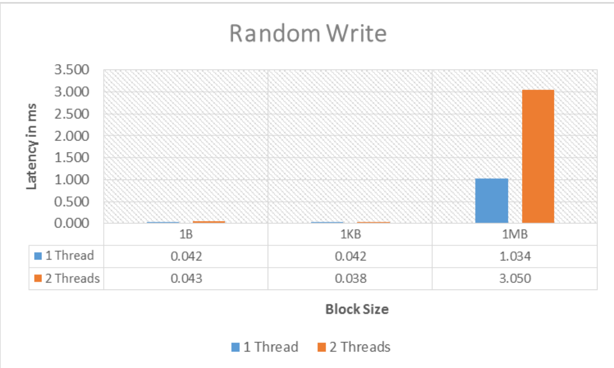


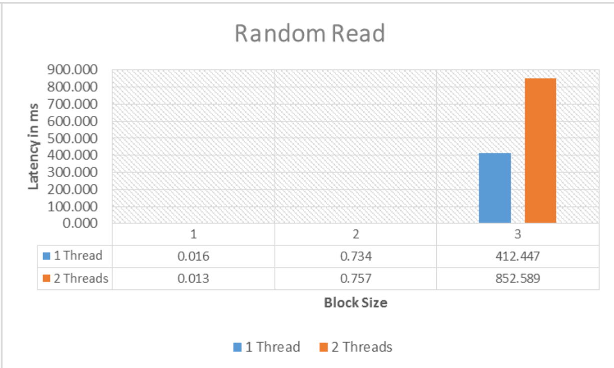


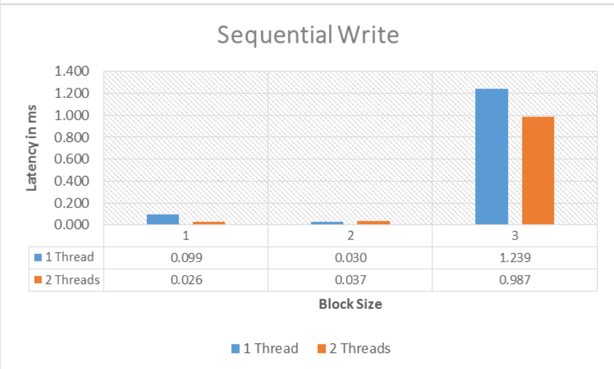


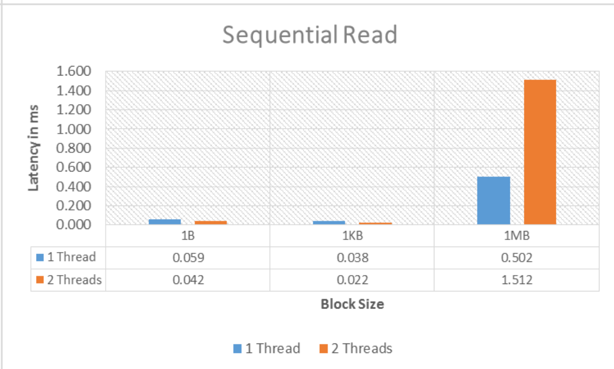
<u>Latency:</u>

The latency for 1B and 1KB files falls under the range 0.001 - 0.5ms for random and sequential read write while for 1MB file it's slightly more ranging from 1-3ms. The random read takes more time to read from a random location hence it comes down to around 400ms.









Average:

The below tables shows the average reading I achieved after running the codes for three times

	Average - Throughput							
	Thread 1				Thread 2			
Size	RW	RR	SW	SR	RW	RR	SW	SR
1B	0.001921	0.039149	0.020402	0.01537	0.025497	0.309727	0.051228	0.056676
1KB	16.15586	1.116369	20.27212	35.07422	49.83252	2.761735	59.05838	90.64876
1MB	921.6627	2.325894	776.5338	1805.256	1048.624	2.279175	1400.065	2535.318
	Average - Latency							
	Thread 1				Thread 2			
Size	RW	RR	SW	SR	RW	RR	SW	SR
1B	0.023089	0.008982	0.052751	0.032764	0.082567	0.02673	0.05172	0.08661
1KB	0.023089	0.403204	0.017991	0.02036	0.076417	1.513552	0.065798	0.042775
1MB	0.570537	226.8367	0.68774	0.275524	4.101661	1708.893	5.204928	2.65771

Standard Deviation

After performing the experiments three times I found minor deviation for throughput ranging from 0.0007 - 1.5MBps and for latency the deviation falls under range of 0.001 – 4ms which gives me consistent results.

	Standard Deviation - Throughput							
	Thread 1				Thread 2			
Size	RW	RR	SW	SR	RW	RR	SW	SR
1B	0.000777	0.033406	0.01458	0.000815	0.000708	0.012315	0.003696	0.002933
1KB	1.533893	0.381735	1.979665	0.833771	0.506105	2.767058	0.930887	1.990842
1MB	0.550298	0.171836	0.375258	0.360825	1.779526	2.335893	0.549302	0.681948
	Standard Deviation - Latency							
	Thread 1				Thread 2			
Size	RW	RR	SW	SR	RW	RR	SW	SR
1B	0.027082	0.010396	0.065348	0.037684	0.003973	0.000206	3.96E-05	0.002536
1KB	0.027082	0.467336	0.016675	0.025394	0.506105	2.767058	0.011558	0.000996
1MB	0.656089	2.624932	0.78009	0.320119	1.779526	2.335893	4.567915	0.517939

As per my evaluation I found sequential read is faster than random read which is the expected outcome since random has to read from a random location.

Extra Credit:

I also ran the IOZone benchmark tool for disk on AWS and found that for 1MB of file transfer it takes 1.4GBps and 7GBps to write and read sequentially whereas 3GBps and 8GBps randomly.

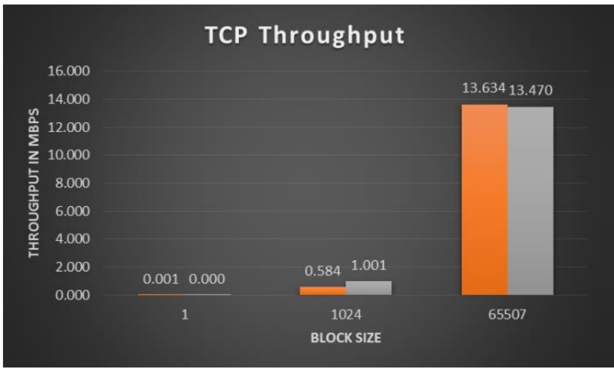
```
        Auto Mode
        Command line used: ./iozone -a
        Output is in Kbytes/sec
        Time Resolution = 0.000001 seconds.
        Processor cache size set to 1024 Kbytes.
        Processor cache line size set to 32 bytes.
        File stride size set to 17 * record size.
                                                              random  random    bkwd   record   stride
              KB  reclen   write rewrite    read    reread    read   write    read  rewrite     read   fwrite frewrite   fread  freread
              64       4 1049372 3363612 10821524 10402178 7100397 3363612 4988978  4274062  7940539  3165299  3203069 5860307 10821524
              64       8 1460430 3541098 12902017 15972885 3363612 3738358 6421025  4897948  9006179  4018152  3738358 7940539 12902017
              64      16 1638743 3738358 12902017 1597288512902017 3203069 5735102  3588436  8182586  2892445  4018152 7940539 10821524
              64      32 1679761 4274062 15972885 1597288512902017 4564786 7100397  4988978  7940539  3541098  5860307 7940539 12902017
              64      64 1768283 3203069 15972885 2096219112902017 4564786 7100397  3363612  8182586  3541098  3363612 7940539  9318832
             128       4 1346196 2784517 11720614 12842051 7582312 3895854 5122535  4267461  3759450  3053774  3560017 6406138 10567140
             128       8 1346196 3124872 12542043 1420079411470204 3560017 8548124  4934216  8548124  3468030  3759450 8548124 12842051
             128      16 1471662 4596273 14200794 1420079411720614 4717434 8414153  4759253  8036304  3657016  4596273 9129573 12842051
             128      32 1391557 4596273 12842051 15881078 9129573 4267461 9129573  4889281  7917784  4267461  5122535 9977956  5603747
             128      64 1905110 4407601 16365173 1588107812842051 5122535 9129573  4717434  7582312  3359523  708219710567140 14200794
             128     128 1911894 4596273 16365173 1254204314200794 4934216 7582312  4267461  7582312  5122535  5122535 8036304 12842051
             256       4 1506359 3236056  9434868 11569783 8271916 4054829 7314033  5117791  7518900  3605511  3823789 7735574  6554972
             256       8 1707589 4652146 11091721 1416439512228612 4422226 8534922  5938650  9868433  2849590  3511189 9114515 13454450
             256      16 1967260 4929815 10651598  7735574 6761355 4929815 7518900  5938650 11091721  4350555  4652146 9114515 13454450
             256      32 1897721 4572895 11091721 13454450 9868433 5022044 8534922  4572895  6554972  4652146  5117791 9192546 13625180
             256      64 1985448 4907284 11091721 1607260811091721 5569035 9192546  5687020 11091721  5242734  584172210651598 13454450
             256     128 2081678 4907284 10245071 1416439510245071 5117791 9778562  4907284  9518507  3950402  7314033 8534922 13454450
             256     256 1802167 4819184 13454450 1345445010651598 5347168 8888172  4907284 11091721  5455847  545584710651598 12812277
             512       4 1394645 4163357 10485468 11374040 8844453 4123387 8394979  5509113  6246213  3736016  3970896 7871843 10235583
             512       8 1789859 4457157  8844453 13872122 8265729 4784885 8992598  6652558  9107004  4000485  4195896 9145790 13522711
             512      16 2065273 4973263 10235583 1378308811374040 527889210044089  6736026 10235583  4571003  453241410235583 13190469
             512      32 2022482 5214798 10694336 1462806711943357 556623110235583  6999490 13190469  4827914  492761710641343 10235583
             512      64 2337255 4701087 10911694 1424006912215097 556623110911694  6999490 13109944  5115422  549501610434519 14240069
             512     128 2105777 5439343 12499490 1387212212146009 576032910911694  6319739 12499490  5453155  5886650 8394979 13872122
             512     256 2297251 4532414 12215097  968182311683444 5566231 9859630  6246213 11943357  4030519  7513788 9814569 13109944
             512     512 2132967 5127637 10044089 1310994411138071 5624545 8665997  4068701 11374040  5684095  562454510434519 12797441
            1024       4 1432225 4014717  7160597 11249091 8914314 3808251 7577494  5472650  8750850  3937426  4048778 8183918 10796646
            1024       8 1648862 3849207  9855234 1386342211489839 485413610230845  6650556 11249091  4570058  445163911018226 13305116
            1024      16 2069060 5024495 11249091 12828238 8895850 455069010769573  6819511 11132462  3804877  4654248 9464330  9743447
            1024      32 2151996 5390231 12173747 1423090210906310 553613711741116  6692005 12037272  4876180  451244110906310 12983353
            1024      64 2245375 5219904 10134284 1442204511018226 588508312037272  7420395 14618393  3984917  487618011773301 14422045
            1024     128 2207295 3447541  7020149 1386342212639479 3447541 6393168  6776473 13142265  3179559  3241960 6874084 13472053
            1024     256 1695065 2822283  6733974 1294422410906310 3591693 5564829  5175871 10255274  3835457  3579719 6830356 12313351
            1024     512 1809317 3055164  9402175 1298335312983353 3748425 6128613  4232305  9064828  3324778  5251818 6402699 12944224
            1024    1024 1828574 3210456 12492426 1314226511018226 3066069 6529233  3618930  6172653  3631168  3151562 6692005 13102174
            2048       4 1430271 2557437  6298503  9621281^C
iozone: interrupted

exiting iozone

ubuntu@ip-172-31-27-251:~/iozone3_394/src/current$
```

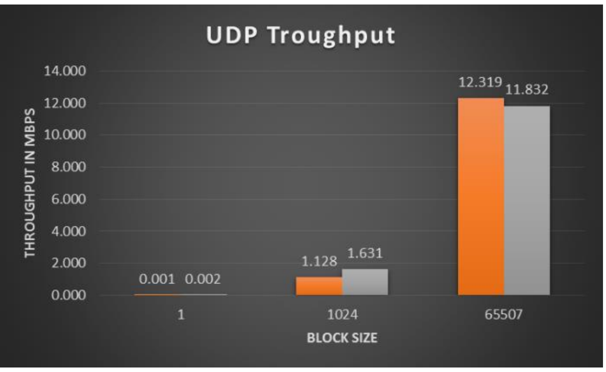
NETWORK BENCHMARK

In Network, I am sending different blocks of data from client to server and then downloading back from server to calculate the roundtrip time. I have used both TCP and UDP protocols to perform this experiment. I have calculated throughput and latency over the network. I created two instances of t2 micro to act as server and client and perform this experiment.

Throughput:

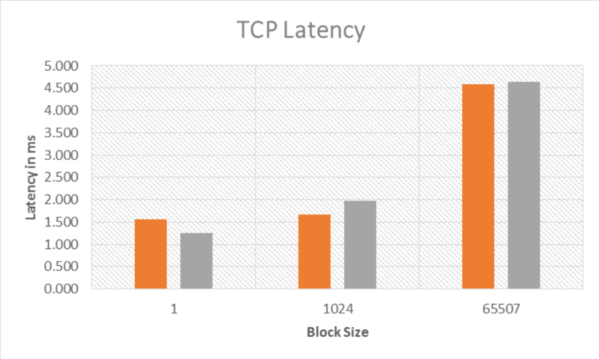
For 64KB of block size I got throughput of 13MBps in TCP and 12MBps in UDP. It remains almost same even if I increased no. of threads. Whereas for 1KB of file I got around 1MBps in both TCP and UDP. As expected, UDP should be faster than TCP but for such a small block size the throughput appears equal but there is difference in the latencies.

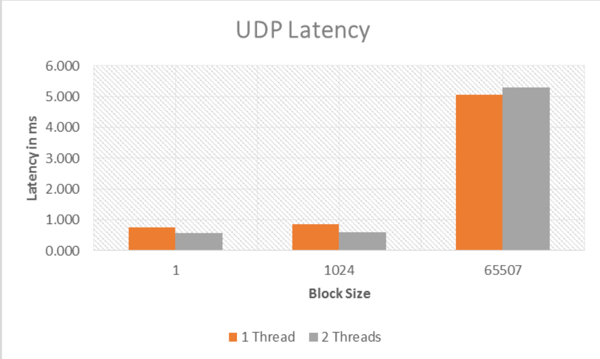




Latency:

As It can be seen from the below graphs, I can make out that UDP is faster than TCP which is obvious since it's a connectionless protocol which will not wait for acknowledgement. With TCP, I am getting 4.5ms latency to transfer 64KB whereas with UDP, I am getting around <4ms.





Average:

The below table gives the average results achieved by executing the code three times.

Average					
TCP - Throughput			UDP - Throughput		
Block size	1 Thread	2 Threads	Block size	1 Thread	2 Threads
1	0.000856	0.000647	1	0.001135	0.001906
1024	0.672182	1.197309	1024	1.616043	2.02554
65507	15.27683	16.912	65507	17.56942	15.95911
TCP - Latency			UDP - Latency		
Block size	1 Thread	2 Threads	Block size	1 Thread	2 Threads
1	1.204198	1.039396	1	0.635753	0.493813
1024	1.478065	1.33736	1024	0.66434	0.511394
65507	4.136107	3.848295	65507	3.900642	4.200702

Standard Deviation

The below table shows deviation resulted when the experiment was performed thrice. It can be seen that deviation is less than 2ms in throughput and latency giving optimum results.

Standard Deviation					
TCP - Throughput			UDP - Throughput		
Block size	1 Thread	2 Threads	Block size	1 Thread	2 Threads
1	0.000320397	0.000269	1	0.000699	0.001856815
1024	0.309122068	1.136813	1024	0.879786	1.142124144
65507	0.673717298	1.481095	65507	0.520057	1.676296571
TCP - Latency			UDP - Latency		
Block size	1 Thread	2 Threads	Block size	1 Thread	2 Threads
1	0.498790295	0.599333	1	0.163699	0.208775045
1024	0.273882962	1.816869	1024	0.284738	0.272642697
65507	0.630890216	2.243778	65507	1.655538	3.084972182

Extra Credit

I ran the Iperf benchmark tool for Network on amazon AWS and achieved the below results for TCP and UDP.

For TCP, minimum 128KB takes 13.4Gbps whereas for UDP, it takes 1.05Mbps to transfer 1KB or 64KB of data.

```
ubuntu@ip-172-31-27-251:~$ sudo iperf -c 52.36.110.108 -n 1
------------------------------------------------------------
Client connecting to 52.36.110.108, TCP port 5001
TCP window size:  325 KByte (default)
------------------------------------------------------------
[  3] local 172.31.27.251 port 47864 connected with 52.36.110.108 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 0.0 sec   128 KBytes  18.4 Gbits/sec
ubuntu@ip-172-31-27-251:~$ sudo iperf -c 52.36.110.108 -n 1024
------------------------------------------------------------
Client connecting to 52.36.110.108, TCP port 5001
TCP window size:  325 KByte (default)
------------------------------------------------------------
[  3] local 172.31.27.251 port 47865 connected with 52.36.110.108 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 0.0 sec   128 KBytes  13.4 Gbits/sec
ubuntu@ip-172-31-27-251:~$ sudo iperf -c 52.36.110.108 -n 65507
------------------------------------------------------------
Client connecting to 52.36.110.108, TCP port 5001
TCP window size:  325 KByte (default)
------------------------------------------------------------
[  3] local 172.31.27.251 port 47866 connected with 52.36.110.108 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 0.0 sec   128 KBytes  15.0 Gbits/sec
```

```
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[  3] local 172.31.27.251 port 60214 connected with 52.36.110.108 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 0.0 sec  1.44 KBytes  1.04 Mbits/sec
[  3] Sent 1 datagrams
[  3] Server Report:
[  3]  0.0- 0.0 sec  1.44 KBytes   273 Mbits/sec   0.000 ms    0/    1 (0%)
ubuntu@ip-172-31-27-251:~$ sudo iperf -c 52.36.110.108 -u -n 1024
------------------------------------------------------------
Client connecting to 52.36.110.108, UDP port 5001
Sending 1470 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[  3] local 172.31.27.251 port 36729 connected with 52.36.110.108 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 0.0 sec  1.44 KBytes  1.03 Mbits/sec
[  3] Sent 1 datagrams
[  3] Server Report:
[  3]  0.0- 0.0 sec  1.44 KBytes   291 Kbits/sec   0.000 ms    0/    1 (0%)
ubuntu@ip-172-31-27-251:~$ sudo iperf -c 52.36.110.108 -u -n 65507
------------------------------------------------------------
Client connecting to 52.36.110.108, UDP port 5001
Sending 1470 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[  3] local 172.31.27.251 port 35272 connected with 52.36.110.108 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 0.5 sec  64.6 KBytes  1.05 Mbits/sec
[  3] Sent 45 datagrams
[  3] Server Report:
[  3]  0.0- 0.5 sec  64.6 KBytes  1.08 Mbits/sec  14.363 ms    0/   45 (0%)
ubuntu@ip-172-31-27-251:~$
```

Conclusion:

From these experiments, I benchmark various components of computer system and was able to achieve 50-70% of the theoretical performances. I also installed and ran the various benchmarking tools on t2 instance to compare the performances with the theoretical and the percentage I achieved. For future developments, I can utilize this knowledge to benchmark my code and utilize computer's hardware performances to its maximum.