<u>ILLINOIS INSTITUTE OD TECHNOLOGY</u>

<u>CLOUD COMPUTING</u>

PROGRAMMING ASSIGNMENT 1

Due by 08th October 2017

PERFORMANCE EVALUATION

&

DESIGN DOCUMENTATION

Indranil Thakur – A20377114

Abhay Nagaraj - A20382006

DESIGN DOCUMENT

This programming assignment asks us to establish 4 different but essential benchmarks.
1. CPU Benchmark
2. Memory Benchmark
3. Disk Benchmark and
4. Network Benchmark.

All the experiments have been performed on
Chameleon Openstack instance m1.medium of Linux version CentOS7 – 1602.0

General design overview implemented for all the programs are as follows:

- Each program contains a main() which serves as the entry point to the program as well as runs the code for variants of block sizes and thread levels.
- There would be functions defined that can be reused for varying inputs provided by the main() which carry out the repetitive Integer and Floating point operations.
- And these methods are clocked with the help of built in functions available.

1. CPU BENCHMARKING:

- All the code is written in C programming language for CPU Benchmarking.
- This program has 4 functions and a main () method.
- 2 functions = (1 for Integer operations + 1 for floating point operations).
- 2 functions = (1 each for allocating threads for integer and floating ops and joining them.
- The program has to perform operations such as addition, subtraction, multiplication and division in a loop of 1 Billion to get the maximal values for number of operations.
- A maximum of 8 threads are used to perform concurrent operations.
- At the end, Linpack experiment is executed and compared with the theoretical values.

Improvements or extensions:

- Could have used AVX instructions to achieve higher performance.
- Could have implemented GUI
- More complex operations could have been performed like Digital signal processing matrix convolutions to test harder on the system's limits.

2. DISK BENCHMARKING:

- The structure and flow of the program remains the same as that of CPU benchmarking.
- The design includes implementation for 4 varying block sizes i.e. 8B, 8KB, 8MB and 80 MB each for Sequential and Random operations.
- Functions for sequential read and write and random write have been implemented.

- A maximum of 8 threads are used to perform concurrent operations and calculate the throughput and latency.

3. MEMORY BENCHMARKING:

- The design includes implementation for 4 varying block sizes i.e. 8B, 8KB, 8MB and 80 MB each for Sequential and Random operations.
- A maximum of 8 threads are used to perform concurrent operations and calculate the throughput and latency.
- The benchmarking also calculates the throughput and latency for the varying concurrencies block sizes of 8B, 8KB, 8MB and 80MB.

4. NETWORK BENCHMARKING:

- All the code is written in **JAVA** programming language for Network Benchmarking.
- The benchmarking is done for both TCP as well as UDP protocol.
- Connection is established between the client and the server. Packets are then exchanged at a fixed block size of 64KB over the network and also between the processes.

Improvements or extensions:

- Add on network cards could be used to increase the network bandwidth available which would have helped us increase the speed of data packet transfer.
- Compression techniques could be implemented before sending the data over the network in order to attain higher data throughput. For example: Huffman encoding.

Performance Evaluation

We have to evaluate the performance of four Benchmarks namely:

1. **CPU**
2. **DISK**
3. **MEMORY**
4. **NETWORK**

The specifications of the machine used are mentioned below:

```
[cc@pa1-neil ~]$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                2
On-line CPU(s) list:   0,1
Thread(s) per core:    1
Core(s) per socket:    1
Socket(s):             2
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 42
Model name:            Intel Xeon E312xx (Sandy Bridge)
Stepping:              1
CPU MHz:               2299.996
BogoMIPS:              4599.99
Hypervisor vendor:     KVM
Virtualization type:   full
L1d cache:             32K
L1i cache:             32K
L2 cache:              4096K
NUMA node0 CPU(s):     0,1
[cc@pa1-neil ~]$ _
```

1. CPU Benchmark

a) Strong scaling is used wherein we keep the **workload and block size the same**, but **distribute** the work **equally between the threads** as we increase the number of threads.

b) We measure the processor speeds in GFLOP/s and GIOP/s at varying concurrency of 1, 2, 4, and 8 threads.
Formula for which is as follows:
FLOP/s = sockets * cores per socket * Cycles per second * Instructions per cycle * FLOPs per cycle

c) The total time is calculated for the instructions executed with all the variants of threads and FLOP/s is computed using the above formula to obtain the following results:

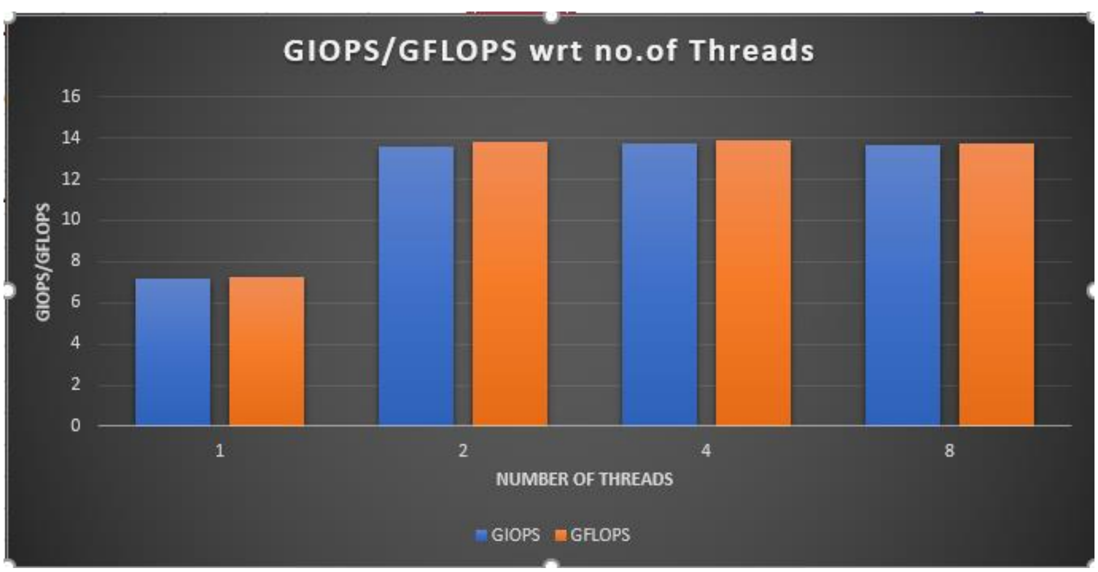
Average & Standard Deviation for FLOPS:

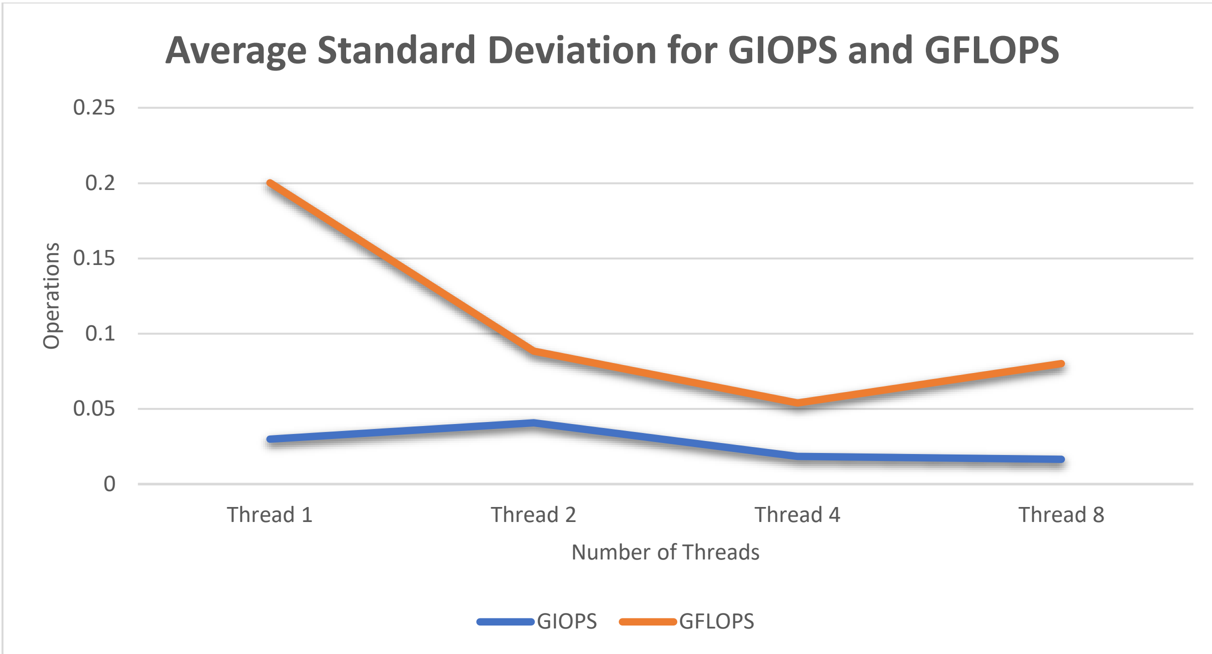
No. of Threads	Experiment :1	Experiment:2	Experiment:3	Average	Standard Deviation
1	7.181762	7.397569	7.061536	7.213622	0.170267
2	13.807019	13.891578	13.810610	13.836402	0.047817
4	13.882545	13.915692	13.844545	13.880927	0.035601
8	13.711689	13.820850	13.710246	13.747595	0.063445

Average & Standard Deviation for GIOPS:

No. of Threads	Experiment :1	Experiment:2	Experiment:3	Average	Standard Deviation
1	7.128434	7.188280	7.155693	7.157469	0.029963
2	13.522522	13.599539	13.583506	13.568522	0.040636
4	13.746735	13.763713	13.727084	13.748544	0.018331
8	13.678757	13.708282	13.680478	13.689172	0.016572

Graphical representation of the values obtained above are as follows:





Observations:

- There is a sharp and significant increase in the performance from 1 thread to 2.
- But almost reaches a saturations on further increase in the number of threads as seen in the graphs which can be attributed to the measuring capacity of the clock available to us.
- Since the maximum precision attainable by the clock is in ms, even if the performance got better with strong scaling, it goes un noticed as the changes are not representable.
- The theoretical peak performance of the processor in flops/sec:
  Number of Cores * Number of Instruction per cycle * Clock Speed = 2*8*2.3 = **36.8** GFLOP/s which is way higher than what we have managed to achieve.
- Efficiency achieved when compared to the theoretical performance= **(13.88/36.8)*100 = 37.8%.**

LINPACK EXPERIMENT was run and following results were obtained.

```
Number of equations to solve (problem size) : 10000
Leading dimension of array                  : 10000
Number of trials to run                     : 4
Data alignment value (in Kbytes)            : 4

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
10000  10000  4      9.592      69.5208  9.187981e-11 3.239775e-02   pass
10000  10000  4      9.427      70.7415  9.187981e-11 3.239775e-02   pass
10000  10000  4      9.585      69.5707  9.187981e-11 3.239775e-02   pass
10000  10000  4      14.697     45.3735  9.187981e-11 3.239775e-02   pass

Performance Summary (GFlops)

Size   LDA    Align.  Average  Maximal
10000  10000  4       63.8016  70.7415

Residual checks PASSED

End of tests

[cc@pa1-neil code]$
```

Efficiency when compared to the theoretical value :
(70.7415/36.8)*100 = 192.23%

Linpack makes use of AVX instructions to achieve high efficiency in FLOPS in which almost 256 bits of data can be fed in parallel to the CPU juxtaposed with the 64 bit instructions normally, to carry out the operations on top of the data which drastically boosts the performance of the processor and hence can be attributed to the performance difference.

2. MEMORY Benchmark

a. Strong scaling is used wherein we keep the **workload and block size the same**, but **distribute** the work **equally between the threads** as we increase the number of threads.

b. We have to calculate the throughput and latency for measuring the memory speed. Latency is measured in ms and throughput is measured in MBPS.

c. Basically for sequential and random read+write operations, we use memcpy(dest, src, no of bytes), wherein it copies the data from the source address location and writes it to the destination address location pointed to by the first 2 arguments to memcpy()

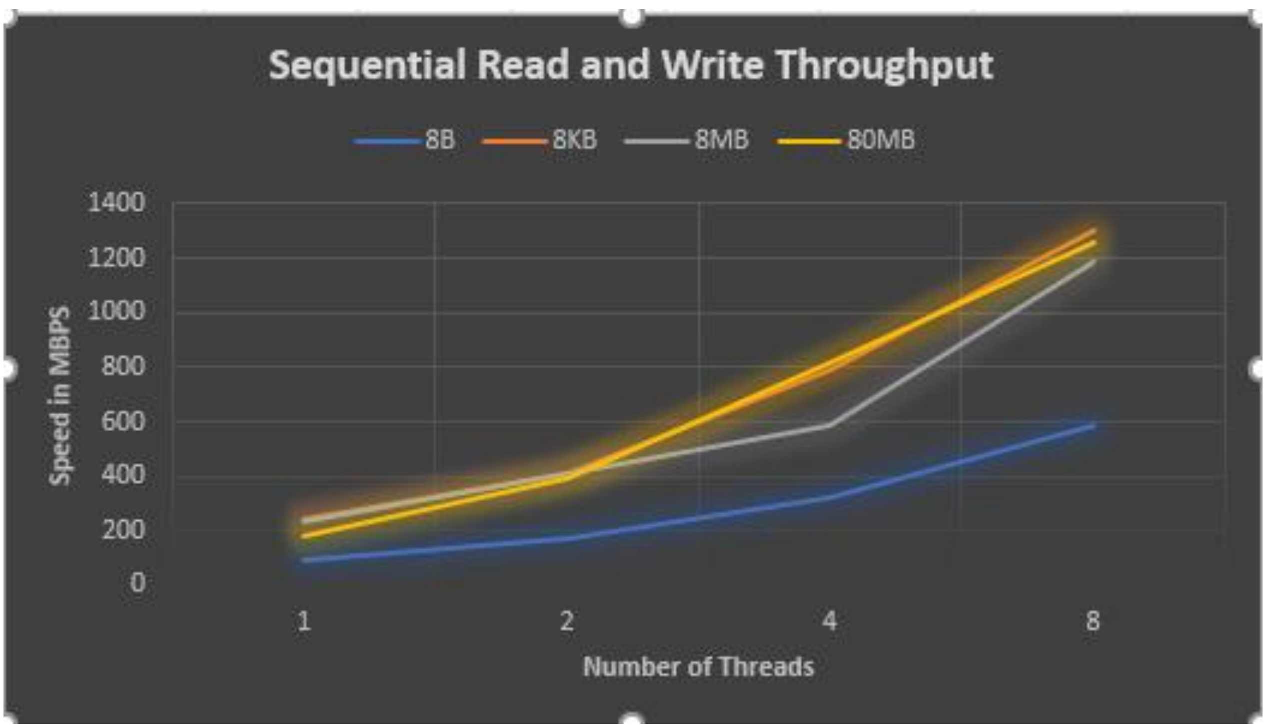
d. Whereas for sequential write alone, we have used memset(pointer, char, no of bytes) Where is copies the char mentioned through the no of bytes starting from the pointer's address location in the memory.

We obtain the following values for various combinations of random and sequential accesses.

For Sequential Read and Write:

Throughput readings are mentioned below:

Threads	8B	8KB	8MB	80MB
1	89.358	243.595	234.362	180.362
2	168.855	411.652	407.910	391.625
4	323.658	793.638	587.329	816.250
8	584.561	1306.621	1184.362	1254.362



Latency readings are mentioned below:

Threads	8B in uSec
1	11.625
2	5.2922
4	2.6584
8	2.125

Observations:

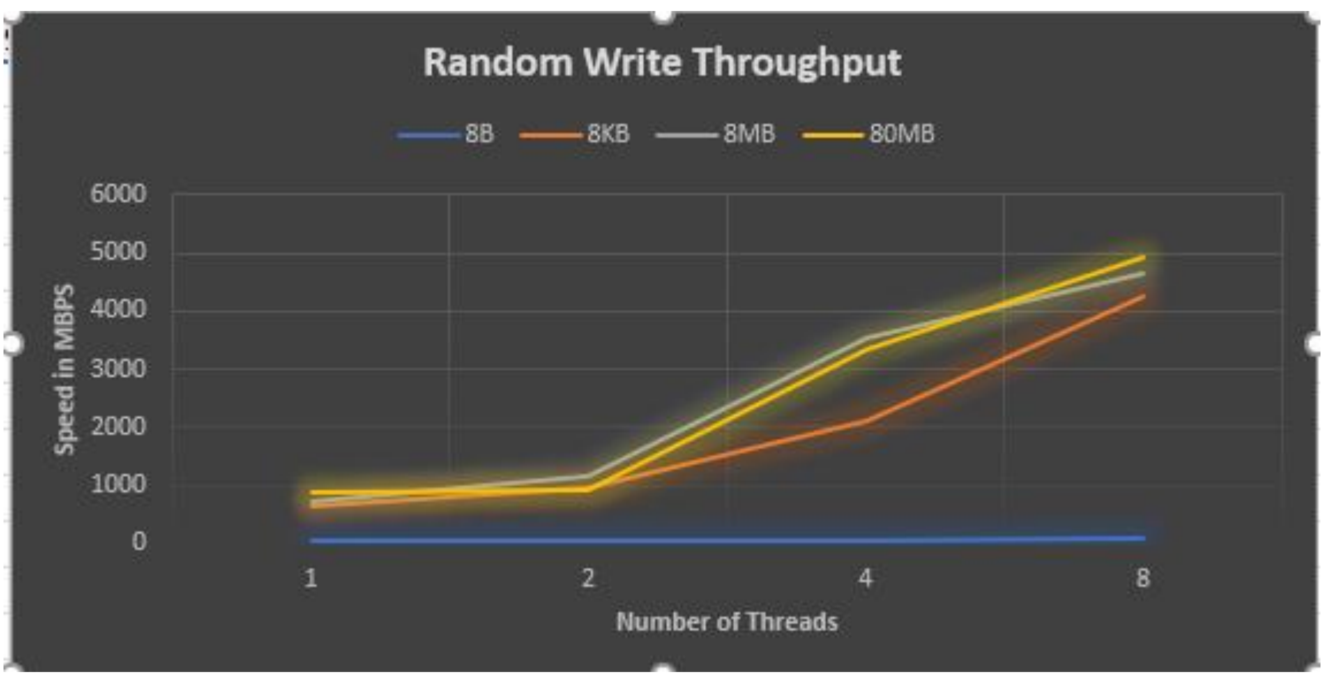
For all the block sizes:

- As we can see that the throughput increases with increase in the number of threads.
- Whereas, the latency reduces with increase in the number of threads as expected.

For Random Write:

Throughput reading are mentioned below:

Threads	8B	8KB	8MB	80MB
1	54.025	633.75	742.25	884.26
2	51.575	986.55	1156.98	946.32
4	52.637	2125.96	3552.12	3356.80
8	83.981	4263.1741	4635.49	4946.32



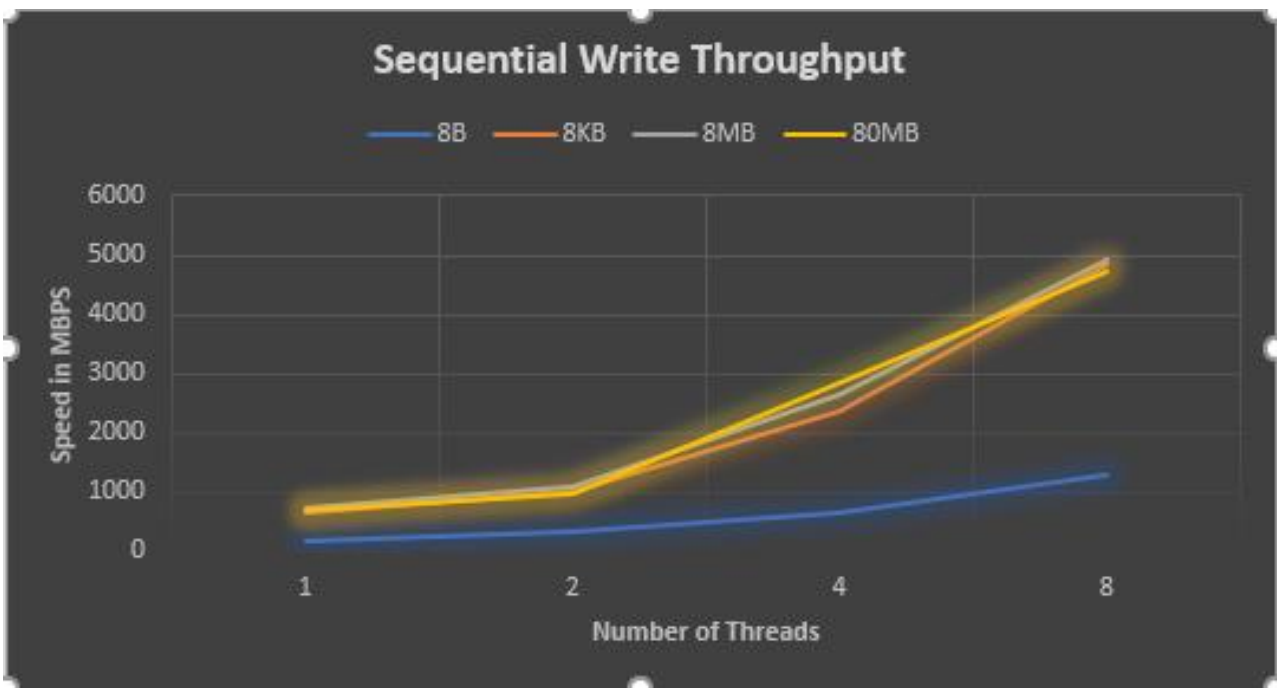
Latency readings are mentioned below:

Threads	8B in uSec
1	17.6551
2	28.4821
4	19.6254
8	12.6517

For Sequential Write:

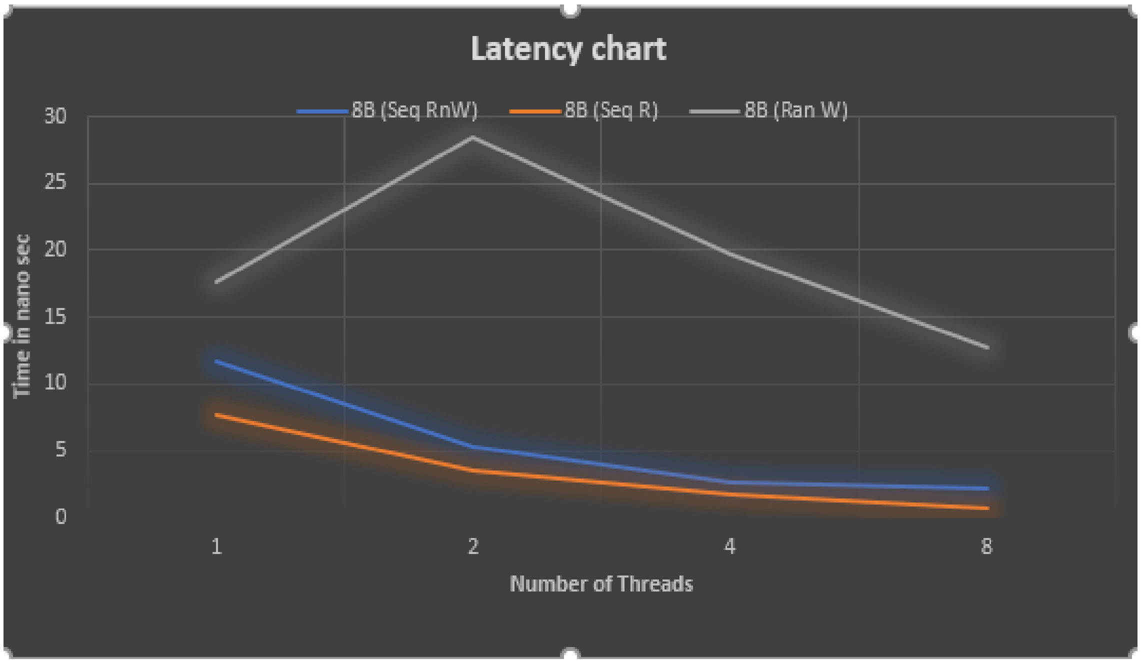
Throughput reading are mentioned below:

Threads	8B	8KB	8MB	80MB
1	164.025	659.751	737.252	714.26
2	318.575	997.559	1089.987	981.326
4	652.639	2375.964	2652.121	2856.80
8	1288.96	4851.641	4915.495	4716.32



Latency readings are mentioned below:

Threads	8B in uSec
1	7.6194
2	3.4762
4	1.6964
8	0.6801



g. We ran the **stream benchmark** on Chameleon instance and below results were obtained.

```
[cc@pa-neil code]$ gcc stream.c
[cc@pa-neil code]$ ./a.out
-------------------------------------------------------------
STREAM version $Revision: 5.10 $
-------------------------------------------------------------
This system uses 8 bytes per array element.
-------------------------------------------------------------
Array size = 10000000 (elements), Offset = 0 (elements)
Memory per array = 76.3 MiB (= 0.1 GiB).
Total memory required = 228.9 MiB (= 0.2 GiB).
Each kernel will be executed 10 times.
 The *best* time for each kernel (excluding the first iteration)
 will be used to compute the reported bandwidth.
-------------------------------------------------------------
Your clock granularity/precision appears to be 1 microseconds.
Each test below will take on the order of 30691 microseconds.
   (= 30691 clock ticks)
Increase the size of the arrays if this shows that
you are not getting at least 20 clock ticks per test.
-------------------------------------------------------------
WARNING -- The above is only a rough guideline.
For best results, please be sure you know the
precision of your system timer.
-------------------------------------------------------------
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:           5937.4     0.030311     0.026948     0.038453
Scale:          5819.9     0.029553     0.027492     0.031844
Add:            8458.5     0.033764     0.028374     0.051148
Triad:          8011.5     0.035110     0.029957     0.050823
-------------------------------------------------------------
Solution Validates: avg error less than 1.000000e-13 on all three arrays
-------------------------------------------------------------
[cc@pa-neil code]$
```

Observations:

- The throughput increases with the number of threads.
- The latency is inversely proportional to the number of threads.
- The read access usually proves to be faster than the write access to the memory.
- The throughput is also directly proportional to the block size, which is obvious most of the times as when the block size is small, there has to be more number of transfers each carrying smaller amount of data compared to when the block size is larger where a larger amount of data is transferred per transfer provided the total data to be read or written being the same in both the cases.
- Sequential reads and writes are faster than the random reads and writes, as when the block sizes are smaller, there needs to more calculations performed to determine the next address location from where the data has to be read or written to as the data is not sequentially available on the memory like in the sequential access case.

For example: if a song is to be read from memory and is distributed sequentially on the memory, then the entire read needs just one calculation which would be the start address of the song unlike in the random access.

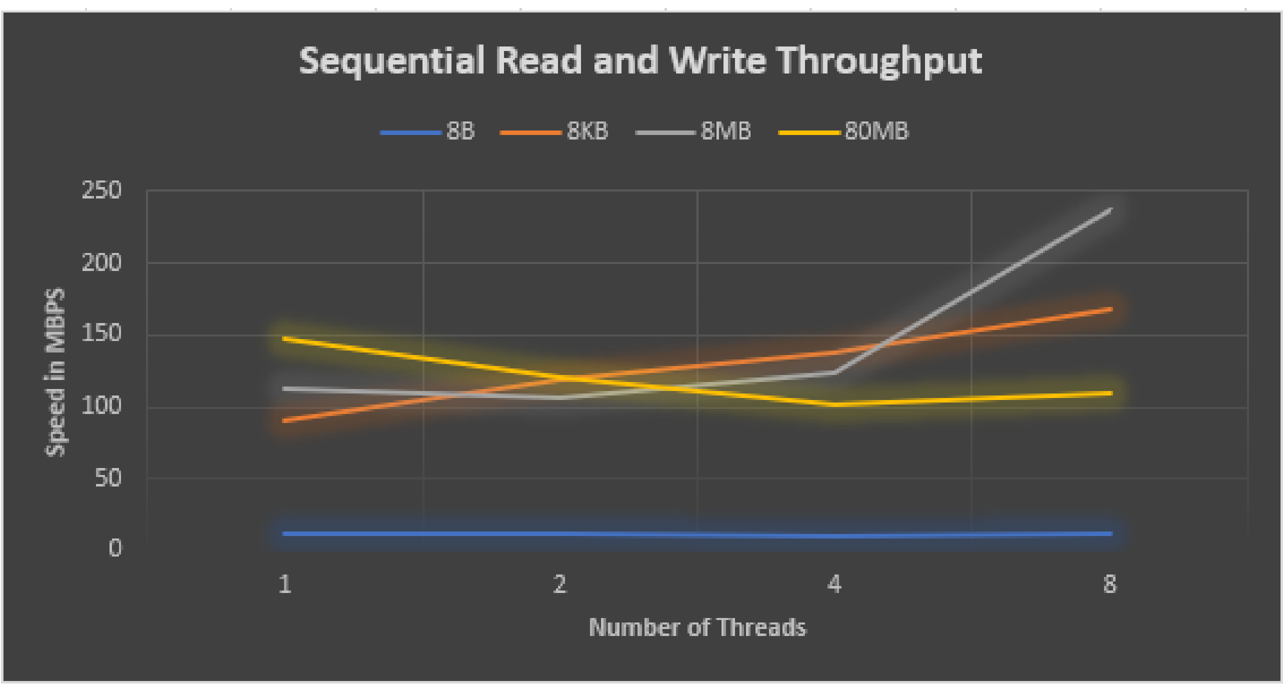
3. DISK Benchmarking

- Strong scaling is used wherein we keep the **workload and block size the same**, but **distribute** the work **equally between the threads** as we increase the number of threads.
- We have to calculate the throughput and latency for measuring the memory speed. Latency is measured in seconds and throughput is measured in MBPS.
- The experiment is carried out for varying bock sizes and at different levels of concurrency along with the combinations of the access types like random, sequential, read, write.

For Sequential Read and Write:

Throughput readings are mentioned below in MBPS:

Block size	8B (MBps)	8KB(MBps)	8MB(MBps)	80MB(MBps)
1	11.9261	91.2658	112.5842	147.2015
2	12.3725	118.2596	105.5841	120.5869
4	10.2543	138.2543	123.5896	102.2357
8	12.2659	167.9851	237.5680	109.5642



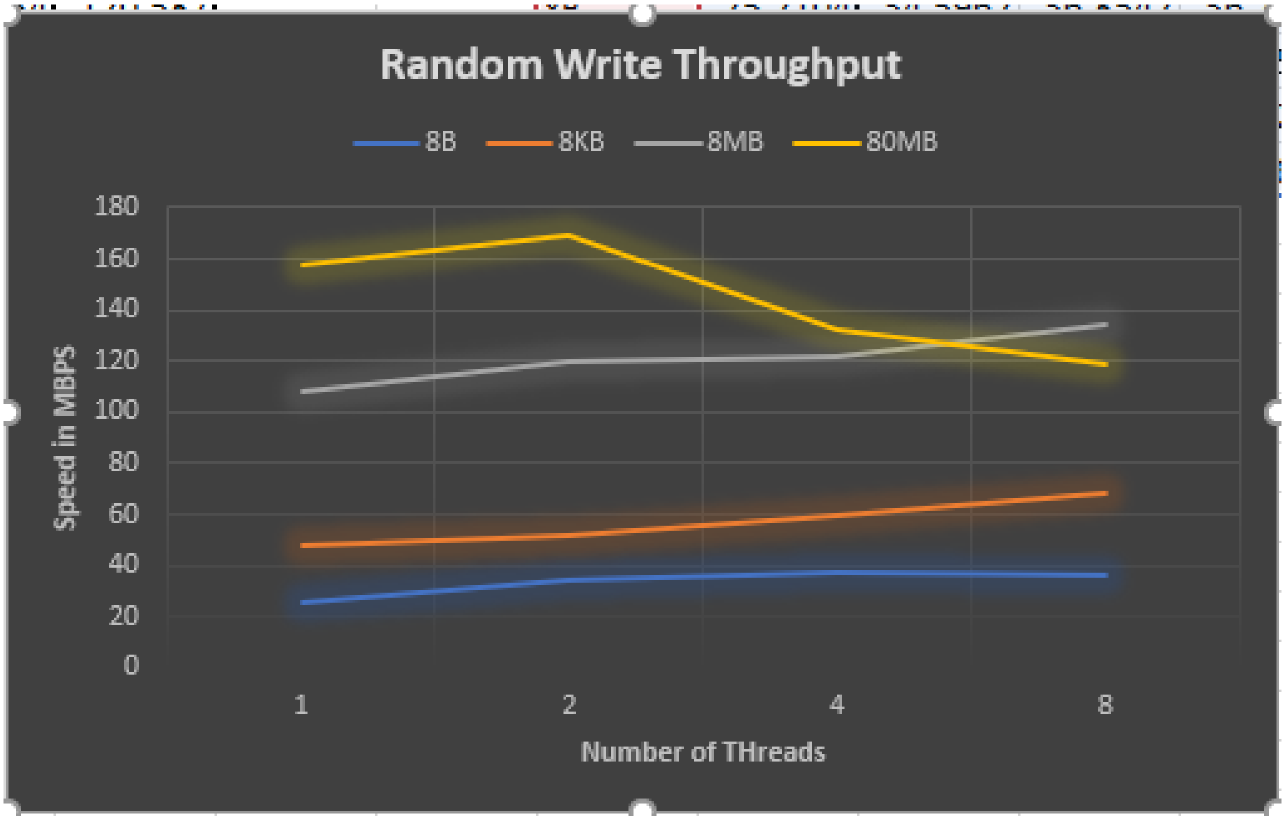
Latency readings are mentioned below:

Threads	8B in mSec
1	0.000658
2	0.095801
4	86.2354
8	118.2589

For Random Write:

Throughput reading are mentioned below in MBPS:

Threads	8B	8KB	8MB	80MB
1	25.2104	48.3520	108.5320	157.9620
2	34.5962	51.6853	119.3651	169.3521
4	36.8542	59.3681	121.5842	132.5983
8	36.2015	68.2596	134.2683	118.6583



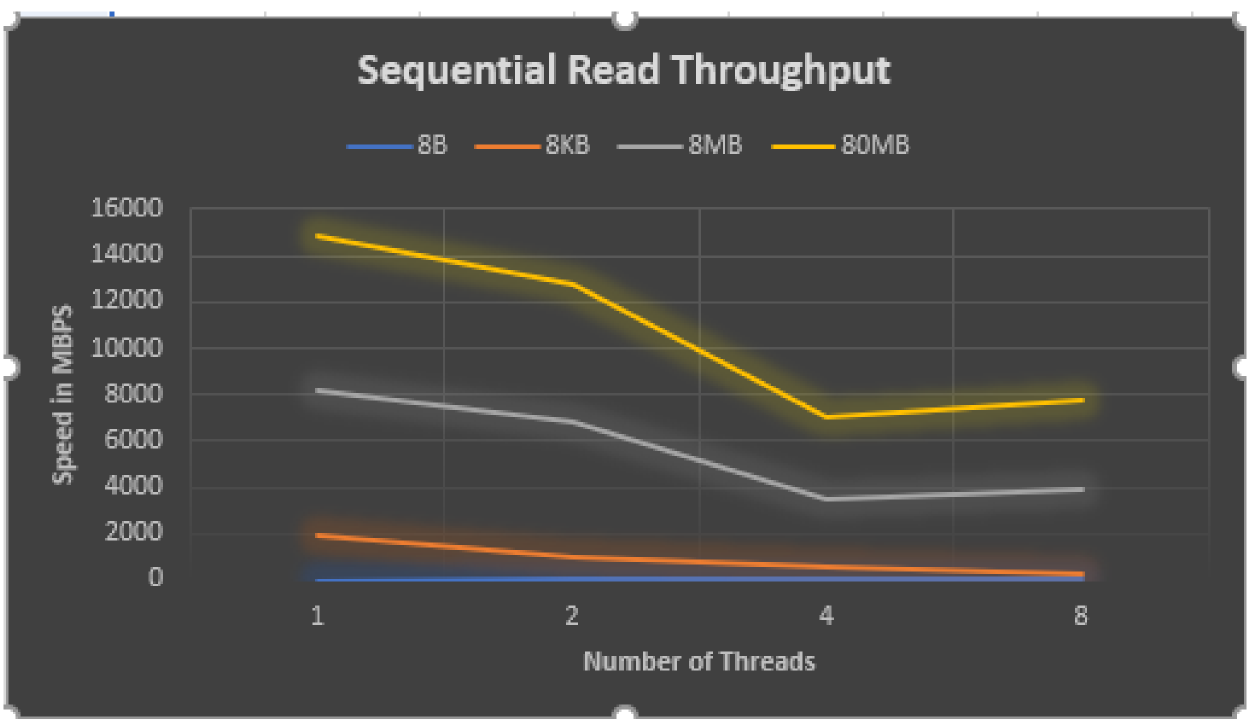
Latency reading are mentioned below:

Threads	8B in mSec
1	35.362581
2	12.63501
4	15.26830
8	22.36584

For Sequential Read:

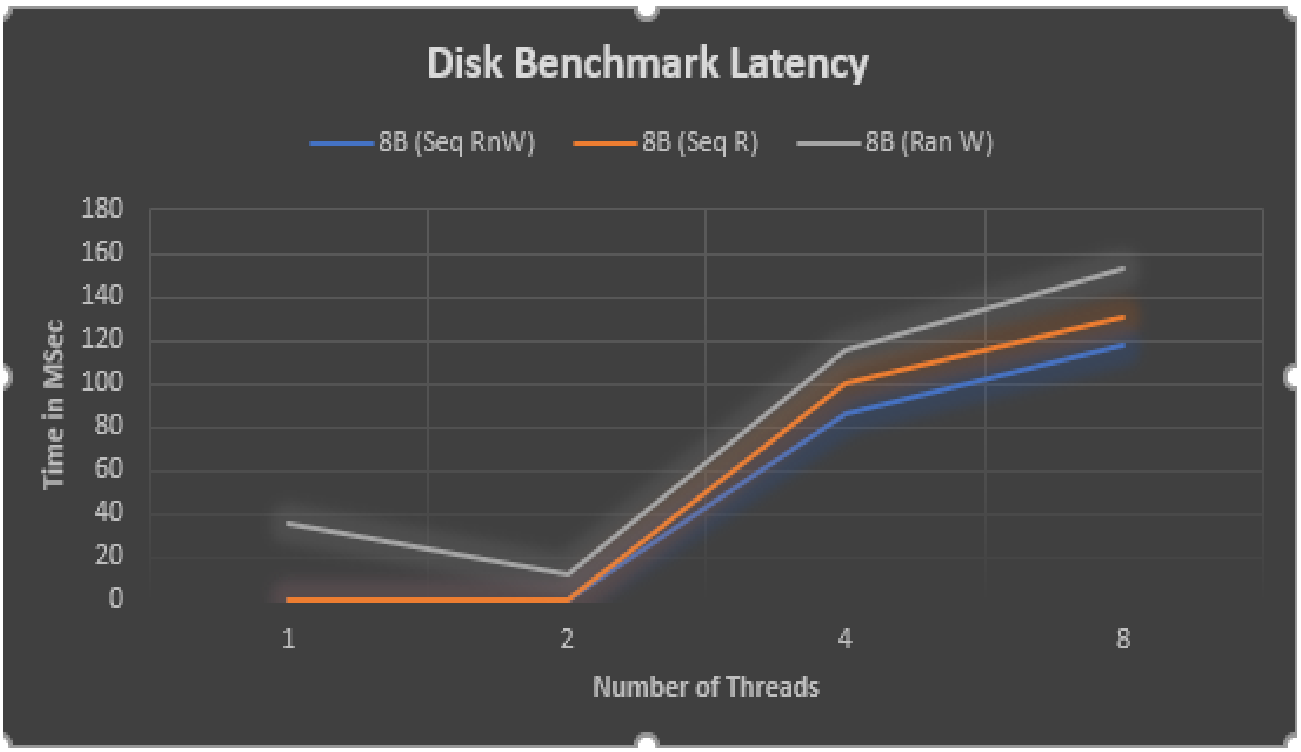
Throughput reading are mentioned below in MBPS:

Threads	8B	8KB	8MB	80MB
1	21.6950	1948.256	6238.1247	6625.107
2	28.5972	980.2543	5768.256	5983.2541
4	26.5940	495.25630	2935.2147	3584.2147
8	34.2591	198.6872	3624.1572	3924.157



Latency reading are mentioned below:

Threads	8B in mSec
1	0.00598
2	0.026587
4	14.2567
8	12.5873



We ran the iozone experiment and obtained the below results:

```
[cc@pa-neil current]$ ./iozone -g# -s 8192
        Iozone: Performance Test of File I/O
                Version $Revision: 3.394 $
                Compiled for 64 bit mode.
                Build: linux

        Contributors:William Norcott, Don Capps, Isom Crawford, Kirby Collins
                     Al Slater, Scott Rhine, Mike Wisner, Ken Goss
                     Steve Landherr, Brad Smith, Mark Kelly, Dr. Alain CYR,
                     Randy Dunlap, Mark Montague, Dan Million, Gavin Brebner,
                     Jean-Marc Zucconi, Jeff Blomberg, Benny Halevy, Dave Boone,
                     Erik Habbinga, Kris Strecker, Walter Wong, Joshua Root,
                     Fabrice Bacchella, Zhenghua Xue, Qin Li, Darren Sawyer.
                     Ben England.

        Run began: Tue Oct 10 02:14:42 2017

        Using maximum file size of 4 kilobytes.
        File size set to 8192 KB
        Command line used: ./iozone -g# -s 8192
        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    bk
wd   record   stride
              KB  reclen   write rewrite    read    reread    read   write    re
ad  rewrite     read   fwrite frewrite   fread  freread
            8192       4 1057977 1807949  3282106  4177574 3589681 1935344 27723
31  2602260  3273040  1891034  1513057 3243073  3901055

iozone test complete.
[cc@pa-neil current]$
```

Observations:

- As expected, the throughput of the disk increases with the increase in the number of threads as well as the block sizes.
- But the difference between the memory and the disk is that, we need to allocate a larger memory space for disk benchmarking, if not the processor just writes to its memory and reads it from the memory unless we force the processor to write to the disk, which happens only after the memory get filled up.
- Therefore in contrast to the memory size considered in memory, we here in disk consider a larger disk size of around 1+ GB
- And also it should be noted that the stream benchmarking tool gets a better through put compared to our throughputs which can be attributed to the efficient algorithms used by the stream.

- Also it is noteworthy that, for all the block sizes unlike memory benchmarking we don't see a monotonic graph for all the threads. As we can see that for this particular set of combinations tried, 2 threads when used give us the best **optimal performance** overall for all the variants of the number of threads.

4. Network Benchmarking

- The code for Network Benchmarking has been written in JAVA programming language.
- We make use of the Java APIs such as

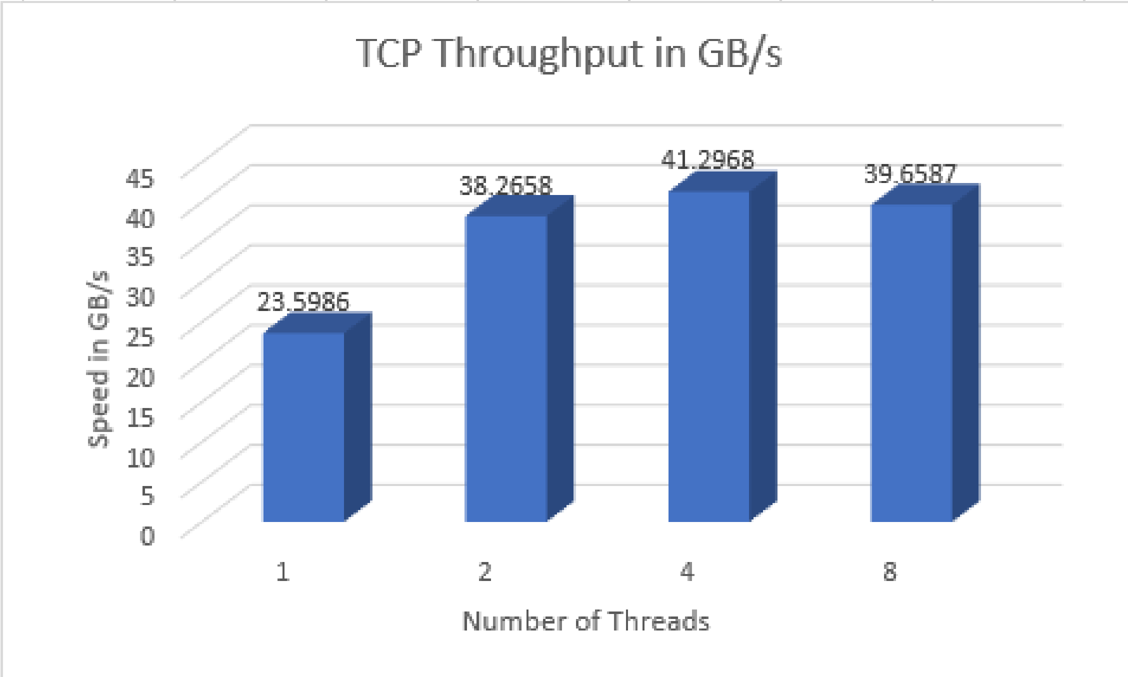
```
java.io.DataInputStream;
java.io.DataOutputStream;
java.io.IOException;
java.io.InputStream;
java.io.OutputStream;
java.net.ServerSocket;
java.net.Socket;

Socket = (IP address + port address) a unique combination to reach a process run on a system connected to the internet.
Sockets are created for each thread and are activated.
Data packets of fixed size of 64KB as mentioned in the assignment are generated up till 1GB of data gets transferred.
```

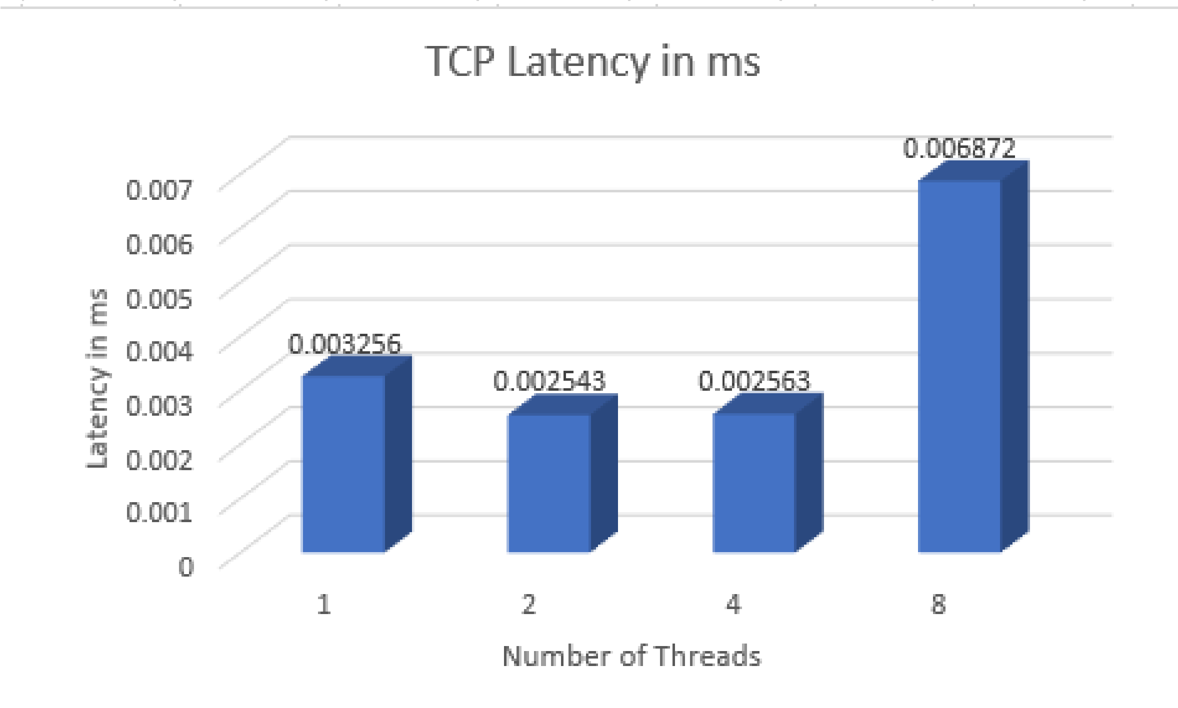
- The network throughput is calculated for both TCP and UPD in similar ways using the Socket APIs in java to obtain the following results.

TCP

Threads	Throughput in GBps	Latency(mSec)
1	23.5986	0.003256
2	38.2658	0.002543
4	41.2968	0.02563
8	39.6587	0.006872



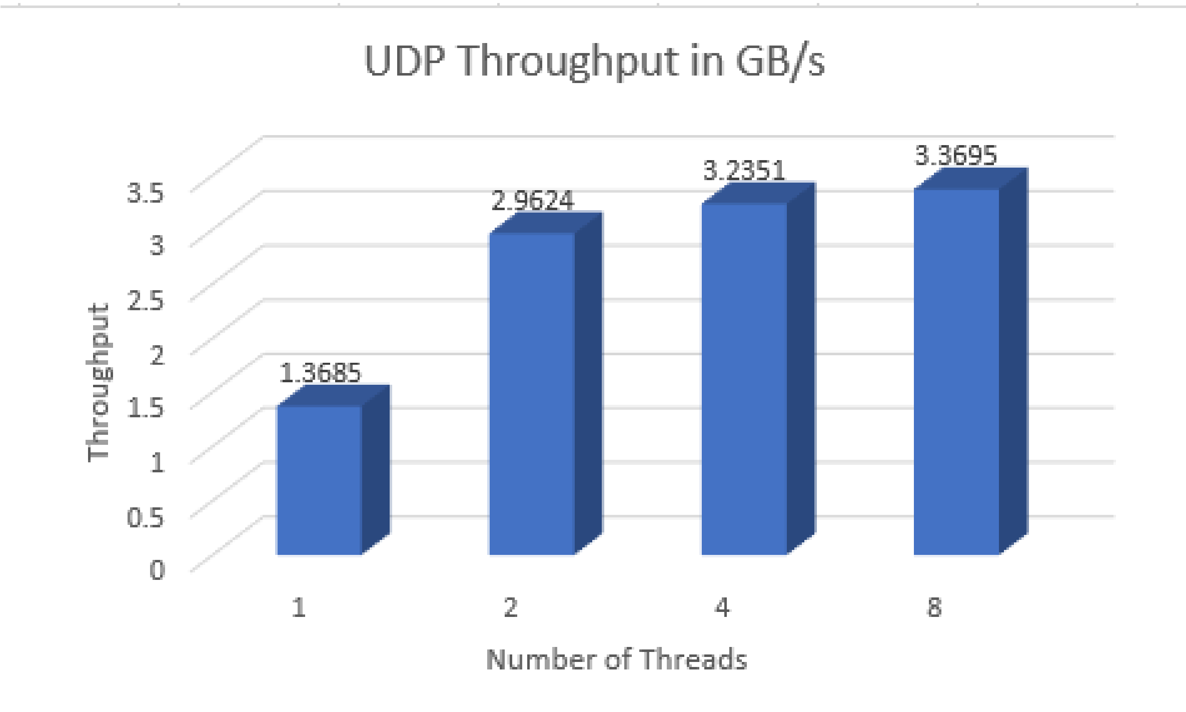
TCP Throughput in GB/s
Speed in GB/s
45
40
35
30
25
20
15
10
5
0
23.5986
38.2658
41.2968
39.6587
1
2
4
8
Number of Threads

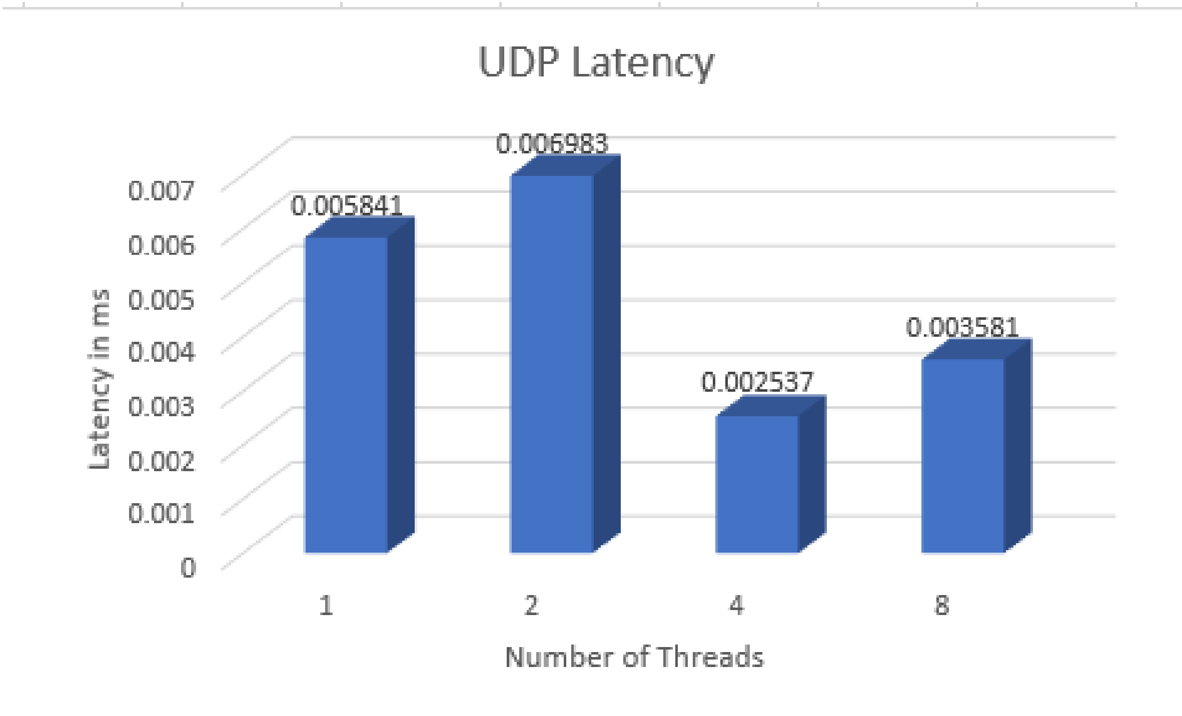


TCP Latency in ms
Latency in ms
0.007
0.006
0.005
0.004
0.003
0.002
0.001
0
0.003256
0.002543
0.002563
0.006872
1
2
4
8
Number of Threads

UDP -

Threads	Throughput in GBps	Latency(mSec)
1	1.3685	0.005841
2	2.9624	0.006983
4	3.2351	0.002537
8	3.3695	0.003581




UDP Latency
0.006983
0.005841
0.003581
0.002537
0.007
0.006
0.005
0.004
0.003
0.002
0.001
0
Latency in ms
1
2
4
8
Number of Threads

We ran the Iperf experiment and obtained the below results:

TCP Iperf Client

```
$ ssh -i /home/ithak/cloud.key cc@129.114.32.57
Enter passphrase for key '/home/ithak/cloud.key':
Last login: Tue Oct 10 16:43:13 2017 from 104.194.112.158
[cc@pal-neil ~]$ sudo su
[root@pal-neil cc]# sudo iperf -c 127.0.0.1 -P 4 -i 1 -t 10 -p 6566
------------------------------------------------------------
Client connecting to 127.0.0.1, TCP port 6566
TCP window size: 2.50 MByte (default)
------------------------------------------------------------
[  6] local 127.0.0.1 port 57584 connected with 127.0.0.1 port 6566
[  4] local 127.0.0.1 port 57582 connected with 127.0.0.1 port 6566
[  5] local 127.0.0.1 port 57583 connected with 127.0.0.1 port 6566
[  3] local 127.0.0.1 port 57581 connected with 127.0.0.1 port 6566
[ ID] Interval       Transfer     Bandwidth
[  6]  0.0- 1.0 sec  2.20 GBytes  18.9 Gbits/sec
[  4]  0.0- 1.0 sec  2.23 GBytes  19.2 Gbits/sec
[  5]  0.0- 1.0 sec  2.17 GBytes  18.7 Gbits/sec
[  3]  0.0- 1.0 sec  2.15 GBytes  18.4 Gbits/sec
[SUM]  0.0- 1.0 sec  8.76 GBytes  75.2 Gbits/sec
[  6]  1.0- 2.0 sec  2.46 GBytes  21.1 Gbits/sec
[  4]  1.0- 2.0 sec  2.49 GBytes  21.4 Gbits/sec
[  5]  1.0- 2.0 sec  2.43 GBytes  20.9 Gbits/sec
[  3]  1.0- 2.0 sec  2.40 GBytes  20.6 Gbits/sec
[SUM]  1.0- 2.0 sec  9.78 GBytes  84.0 Gbits/sec
[  4]  2.0- 3.0 sec  2.46 GBytes  21.2 Gbits/sec
[  5]  2.0- 3.0 sec  2.41 GBytes  20.7 Gbits/sec
[  3]  2.0- 3.0 sec  2.38 GBytes  20.5 Gbits/sec
[  6]  2.0- 3.0 sec  2.44 GBytes  21.0 Gbits/sec
[SUM]  2.0- 3.0 sec  9.70 GBytes  83.3 Gbits/sec
[  6]  3.0- 4.0 sec  2.43 GBytes  20.9 Gbits/sec
[  4]  3.0- 4.0 sec  2.45 GBytes  21.1 Gbits/sec
[  5]  3.0- 4.0 sec  2.41 GBytes  20.7 Gbits/sec
[  3]  3.0- 4.0 sec  2.39 GBytes  20.5 Gbits/sec
[SUM]  3.0- 4.0 sec  9.69 GBytes  83.2 Gbits/sec
[  6]  4.0- 5.0 sec  2.33 GBytes  20.0 Gbits/sec
[  4]  4.0- 5.0 sec  2.35 GBytes  20.2 Gbits/sec
[  5]  4.0- 5.0 sec  2.31 GBytes  19.8 Gbits/sec
[  3]  4.0- 5.0 sec  2.28 GBytes  19.6 Gbits/sec
[SUM]  4.0- 5.0 sec  9.27 GBytes  79.6 Gbits/sec
[  6]  5.0- 6.0 sec  2.40 GBytes  20.6 Gbits/sec
[  4]  5.0- 6.0 sec  2.42 GBytes  20.8 Gbits/sec
[  5]  5.0- 6.0 sec  2.38 GBytes  20.4 Gbits/sec
[  3]  5.0- 6.0 sec  2.36 GBytes  20.2 Gbits/sec
[SUM]  5.0- 6.0 sec  9.55 GBytes  82.0 Gbits/sec
[  6]  6.0- 7.0 sec  2.48 GBytes  21.3 Gbits/sec
[  4]  6.0- 7.0 sec  2.49 GBytes  21.4 Gbits/sec
[  5]  6.0- 7.0 sec  2.43 GBytes  20.9 Gbits/sec
[  3]  6.0- 7.0 sec  2.41 GBytes  20.7 Gbits/sec
[SUM]  6.0- 7.0 sec  9.82 GBytes  84.3 Gbits/sec
[  6]  7.0- 8.0 sec  2.50 GBytes  21.5 Gbits/sec
[  4]  7.0- 8.0 sec  2.54 GBytes  21.8 Gbits/sec
[  5]  7.0- 8.0 sec  2.48 GBytes  21.3 Gbits/sec
[  3]  7.0- 8.0 sec  2.43 GBytes  20.9 Gbits/sec
[SUM]  7.0- 8.0 sec  9.95 GBytes  85.4 Gbits/sec
[  6]  8.0- 9.0 sec  2.42 GBytes  20.8 Gbits/sec
[  4]  8.0- 9.0 sec  2.45 GBytes  21.0 Gbits/sec
[  5]  8.0- 9.0 sec  2.39 GBytes  20.5 Gbits/sec
[  3]  8.0- 9.0 sec  2.37 GBytes  20.3 Gbits/sec
[SUM]  8.0- 9.0 sec  9.63 GBytes  82.7 Gbits/sec
[  6]  9.0-10.0 sec  2.49 GBytes  21.4 Gbits/sec
[  4]  9.0-10.0 sec  2.51 GBytes  21.5 Gbits/sec
[  5]  9.0-10.0 sec  2.46 GBytes  21.1 Gbits/sec
[  3]  9.0-10.0 sec  2.44 GBytes  20.9 Gbits/sec
[SUM]  9.0-10.0 sec  9.89 GBytes  84.9 Gbits/sec
[  6]  0.0-10.0 sec  24.2 GBytes  20.7 Gbits/sec
[  4]  0.0-10.0 sec  24.4 GBytes  20.9 Gbits/sec
[  3]  0.0-10.0 sec  23.6 GBytes  20.3 Gbits/sec
[  5]  0.0-10.0 sec  23.9 GBytes  20.5 Gbits/sec
[SUM]  0.0-10.0 sec  96.2 GBytes  82.4 Gbits/sec
[root@pal-neil cc]# |
```

TCP Iperf Server

```
[root@pa1-neil cc]# iperf -s -p 6566
------------------------------------------------------------
Server listening on TCP port 6566
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  4] local 127.0.0.1 port 6566 connected with 127.0.0.1 port 57581
[  5] local 127.0.0.1 port 6566 connected with 127.0.0.1 port 57582
[  6] local 127.0.0.1 port 6566 connected with 127.0.0.1 port 57583
[  7] local 127.0.0.1 port 6566 connected with 127.0.0.1 port 57584
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0-10.0 sec  23.6 GBytes  20.3 Gbits/sec
[  6]  0.0-10.0 sec  23.9 GBytes  20.5 Gbits/sec
[  5]  0.0-10.0 sec  24.4 GBytes  20.9 Gbits/sec
[  7]  0.0-10.0 sec  24.2 GBytes  20.7 Gbits/sec
[SUM]  0.0-10.0 sec  96.2 GBytes  82.3 Gbits/sec
```

Observations:

- As can be seen from the results, the throughputs of both TCP and the UDP shoot up with the increase in the number of threads until 2 threads.
- But then they reach a saturation point due to the bottle neck offered by the network bandwidth available.
- In addition, we can also notice that, the UDP being an unreliable connectionless protocol which doesn’t include an acknowledgement mechanism unlike TCP, results in loss of packets sometime. But is definitely a faster protocol relative to the TCP.
- The throughput of the UDP is higher than that of TCP for a particular block size or packet size being considered because, UDP does not have a huge overload unlike TCP does, as TCP needs extra data to be carried along with the payload for error detection, connection oriented delivery, Heavy TCP Header etc.
- Therefore, UDP is more preferable over TCP in scenarios where the speed of delivery is more important than the reliability and the integrity of the data to be delivered like in the multiplayer real time online gaming systems.
- The iPerf’s performance is way better than that of our program which is evident from the results seen above.