1. Design Report

All the tests are performed on amazon EC2 instances having following configuration.

Configuration:
Processor: Intel Xeon E5-2670v2 @2.50Ghz.
No of Cores: 1
No of Threads: 1
Instruction/Cycle: 8
Instance Type: t2.micro

All the experiments are performed for 3 time than the average result is taken into consideration.

1) For CPU Benchmark:

I have measured the Integer and Floating Point Operation considering 1, 2 and 4 threads. Overall performance vary a little due to the single core processor on Amazon EC2, while performing the same operation on the multithreaded local system, There was a wide difference in performance of program with respect to the number of threads running.

Floating point operation took little higher time and Integer points get good result compare to Floating point operation. Overall I have performed 6 different Experiments.

I inherited the Thread Class considering that the user will give input with their choice of thread number and integer or floating point operation. Then as per entered parameter number of threads are created and appropriate operations are called in run() method. This whole procedure is taken into consideration for CPU performance to time is measured for entire process.

2) For Disk Benchmark:

I had measured disk performance on concurrent level on 1 and 2 thread with the following operation mode.

a. Sequential Read
b. Sequential Write
c. Random Write
d. Random Read

During performance I have considered 3 different block size for operation which were 1 Byte, 1 KB and 1 MB. Apart from using Thread class, I have Used FileChannel class and RandomAccessFile class for reading and writing operation. Also in order to send data to file from array I have used ByteBuffer Class.

Run() method consist of calling appropriate operation residing outside the Run() method but in same class. Since we are measuring the disk reading/writing speed, I am counting time inside each function and passing it further for calculation. I did the operation in iteration of thousand to overcome the overhead process execution duration.

Performance was better on local system because it contained multiple thread. While running same program on Amazon EC2 cloud performance was degraded for 2 threads. When Block size is increased performance was getting better.

3) For Network:

I have made two program here different for TCP and UDP.

- TCP consist of two class as client and server. TCP is connection oriented so once connection has been made, it will be connected until we close it. Thus we need to provide IP/Hostname and Port of server, which we pass as an argument by user from client to the server.
  Here I am testing it for Round Trip packet transmission. So I am counting time on Client Side. In the iteration of 1000 I am sending data packet to the server with the size input by user. User is passing two arguments as number of threads and block size to be sent.
  Server is receiving the packet and send back the same packet to the client. Once the packet is send client measure the time for the packet RTT. Which later I used for the calculation of throughput and latency.

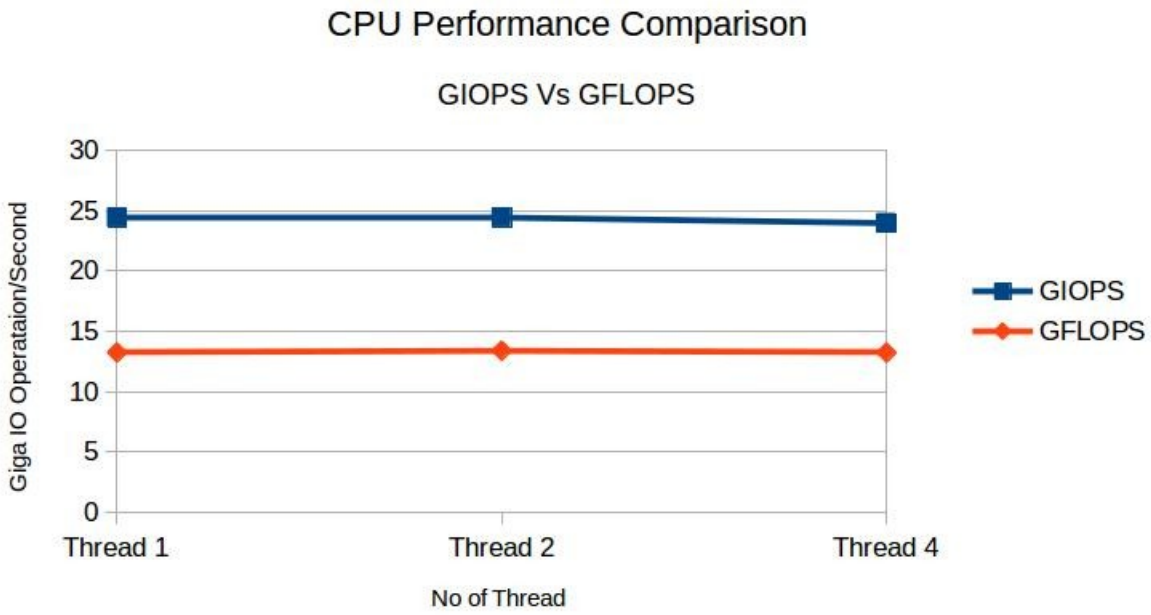
- UDP consist of two program of Server and client. UDP is connectionless so both the classes works as client and server, instead of calling it as Client/Server it is more or of sender/receiver program.
  UDP used DatagramPacket class to send/receive data. UDP can send max data packet of 65507 Bytes, so here I expect from user that it will enter 65507 instead of 65536 bytes at the input screen, otherwise this program will throw and error for connection reset.
  Program consist of same arguments are number of thread, block size to send and Hostname for other class.

- In future we can make it more advance by considering the Timeout, which is currently set to the default. Another option we can consider is to send back the acknowledgement of packet to other side.

2. Performance Evaluation:

1. CPU:

A. Processor speed on Amazon EC2 is as below:



While doing operation in Floating point I observed the speed of 13 Giga-Flop/Second across the Thread ranges from **1-4**, While doing operation in Integer point, I observer the speed around **24** Giga-Iops/Second.

B. Processor Speed at varying level of concurrency:

Integer Operations:

Integer Operation	Reading 1	Reading 2	Reading 3	Average	Std Dev
1 Thread	24.39	24.48	24.39	24.42	0.305505046
2 Thread	24.69	24.09	24.49	24.42333333	0.305505046
4 Thread	23.71	23.8	24.39	23.96666667	0.369368831

Floating Operations:

Floating Operation	Reading 1	Reading 2	Reading 3	Average	Std Dev
1 Thread	13.24	13.1	13.33	13.22333333	0.115902258
2 Thread	13.3	13.36	13.45	13.37	0.075498344
4 Thread	13.21	13.2	13.27	13.22666667	0.037859389

C. Theoretical Peak Performance of Processor in Flops/Sec:

Peak Performance = CPU Speed in Ghz * CPU Cores * Instruction per Cycle * CPU per Node
= 2.5 * 1 * 8 * 2
= 40 * 10E9 flop/Sec

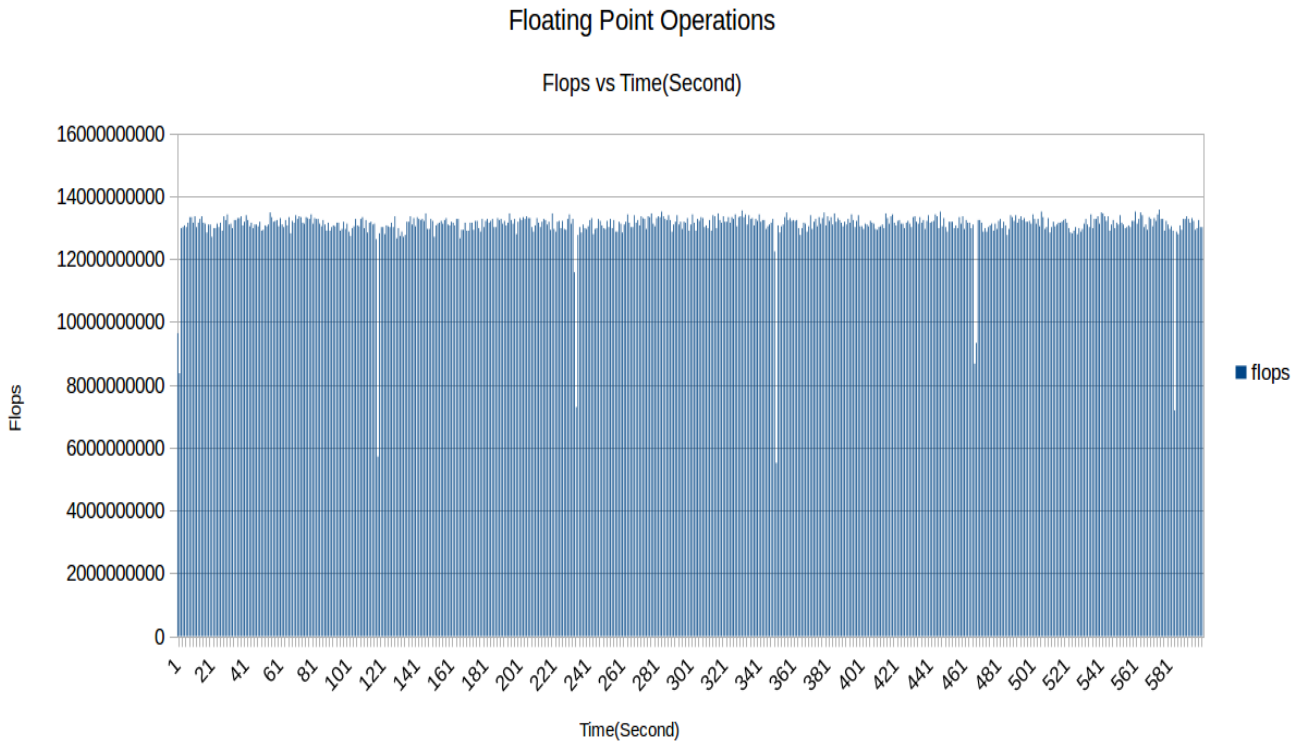
source:http://www.novatte.com/our-blog/197-how-to-calculate-peak-theoretical-performance-of-a-cpu-based-hpc-system

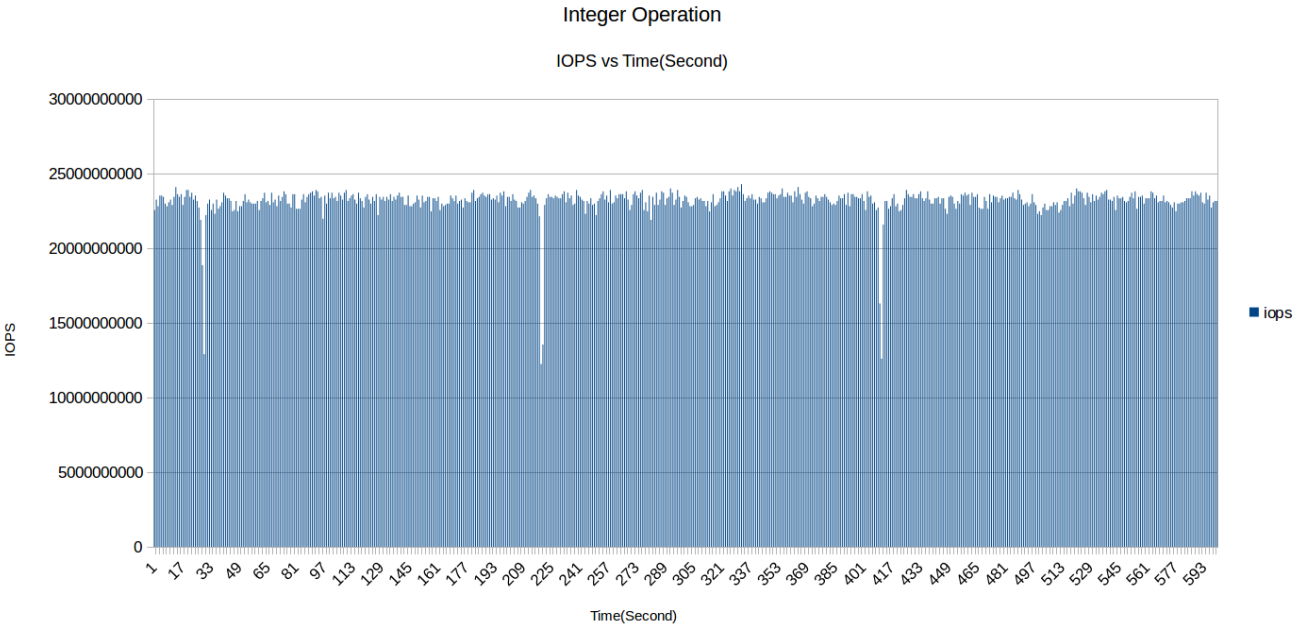
D. Efficiency achieved compared to Theoretical performance

I have achieved peak performance of 13.37 Giga-FIOP/Sec which is 25% of the theoretical performance.

E. Benchmark Result:

Benchmark are taken for 4 Thread 600 samples each.





F. Lin Pack Performance Evaluation:

Highest Peak Performance that I have achieved is 20.4658 GFlops which is 51.15% of theoretical performance

```
(ec2-user) ec2-52-87-226-0.compute-1.amazonaws.com – Konsole
[ec2-user@ip-172-31-58-61 linpack]$ ls
help.lpk      lininput_xeon32  lininput_xeon64_ao  runme_mic     runme_xeon64     xhelp.lpk     xlinpack_xeon32
lininput_mic  lininput_xeon64  lin_xeon64.txt      runme_xeon32  runme_xeon64_ao  xlinpack_mic  xlinpack_xeon64
[ec2-user@ip-172-31-58-61 linpack]$ ./runme_xeon64
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 16:40:13 UTC 2016
Intel(R) Optimized LINPACK Benchmark data

Current date/time: Fri Feb 12 16:40:13 2016

CPU frequency:    2.847 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      0.051      13.0753  9.900691e-13 3.376390e-02   pass
1000   1000   4      0.038      17.5163  9.900691e-13 3.376390e-02   pass
1000   1000   4      0.038      17.5545  9.900691e-13 3.376390e-02   pass
1000   1000   4      0.040      16.8401  9.900691e-13 3.376390e-02   pass
2000   2000   4      0.289      18.5000  4.053480e-12 3.526031e-02   pass
2000   2000   4      0.289      18.4644  4.053480e-12 3.526031e-02   pass
5000   5008   4      4.159      20.0474  2.336047e-11 3.257429e-02   pass
5000   5008   4      4.168      20.0076  2.336047e-11 3.257429e-02   pass
10000  10000  4      32.404     20.5800  1.124127e-10 3.963786e-02   pass
10000  10000  4      32.767     20.3517  1.124127e-10 3.963786e-02   pass

Performance Summary (GFlops)

Size   LDA    Align.  Average  Maximal
1000   1000   4       16.2465  17.5545
2000   2000   4       18.4822  18.5000
5000   5008   4       20.0275  20.0474
10000  10000  4       20.4658  20.5800

Residual checks PASSED

End of tests

Done: Fri Feb 12 16:41:34 UTC 2016
[ec2-user@ip-172-31-58-61 linpack]$
(ec2-user) ec2-52-87-226-0.compute-1.amazonaws.com
(ec2-user) ec2-52-90-237-215.compute-1.amazonaws.com
```

3. DISK:

A. Measure the Disk Speed:

- **1 Thread :– Speed Measurement:**

1 Thread: Throughput(1B)

	Reading 1	Reading-2	Reading-3	Average	Std Dev
Write_Sequential	0.841278	0.881959	0.854687	0.859308	0.0207304407
Read_Sequential	3.146985	3.1199408011	3.1414724326	3.1361327446	0.0142909543
Write_Random	0.8390548097	0.8429258588	0.8451209314	0.8423672	0.0030714057
Read_Random	1.4019908972	1.4209903391	1.4206188372	1.4145333578	0.0108636777

1 Thread: Throughput(1KB)

	Reading 1	Reading-2	Reading-3	Average	Std Dev
Write_Sequential	502.4387	524.1249	523.4396	516.6677333333	12.3274673483
Read_Sequential	1422.468	1386.372	1407.838	1405.5593333333	18.155565134
Write_Random	63.18732	64.71992	61.70278	63.20334	1.5086337943
Read_Random	54.47872	57.12697	53.12465	54.9101133333	2.03573488

1 Thread: Throughput(1 MB)

	Reading 1	Reading-2	Reading-3	Average	Std Dev
Write_Sequential	1135.173	1144.954	864.5163	1048.2144333333	159.1624017564
Read_Sequential	1749.025	1819.0348	2136.664	1901.5746	206.5807651217
Write_Random	1614.818	1726.584	1886.423	1742.6083333333	136.5097192523
Read_Random	310.5358	336.0151	313.4874	320.0127666667	13.9367854875

- **2 Thread :– Speed Measurement:**

2 Thread: Throughput(1B)

	Reading 1	Reading-2	Reading-3	Average	Std Dev
Write_Sequential	0.5079453	0.4991907	0.5375612	0.5148990667	0.0201082066
Read_Sequential	1.430121	1.6738958	1.4309064	1.5116410667	0.1405172697
Write_Random	0.3230364	0.3315832	0.7121133	0.4555776333	0.2222075002
Read_Random	0.8437458	0.78240289	0.93569896	0.8539492167	0.0771557098

2 Thread: Throughput(1 KB)

	Reading 1	Reading-2	Reading-3	Average	Std Dev
Write_Sequential	283.3838	288.7776	268.0161	280.0591666667	10.7726440888
Read_Sequential	485.1352	468.3341	458.7753	470.7482	13.3447364571
Write_Random	37.5206	41.1923	42.8823	40.5317333333	2.7412073915
Read_Random	24.8929	13.42995	21.1039	19.8089166667	5.8401662691

2 Thread: Throughput(1 MB)

	Reading 1	Reading-2	Reading-3	Average	Std Dev
Write_Sequential	846.4275	781.7741	768.6456	798.9490666667	41.6382108394
Read_Sequential	1277.956	1161.039	1208.525	1215.84	58.8007494935
Write_Random	631.903	691.8571	761.6319	695.1306666667	64.9263740928
Read_Random	243.2472	205.4197	184.8606	211.1758333333	29.6158499473

- **1 Thread :- Latency Measurement:**

1 Thread: Latency(1B)

	Reading 1	Reading-2	Reading-3	Average	Std Dev
Write_Sequential	0.042733	0.037306	0.031224	0.0370876667	0.0057576056
Read_Sequential	0.070033	0.005688	0.008395	0.0280386667	0.0363933371
Write_Random	0.0011366	0.0011313	0.0011284	0.0011321	4.158124577258E-006
Read_Random	6.80E-004	6.71E-004	6.71E-004	6.74E-004	5.201099855434E-006

1 Thread:Latency(1 KB)

	Reading 1	Reading-2	Reading-3	Average	Std Dev
Write_Sequential	0.030971	0.031979	0.032591	0.031847	0.0008180269
Read_Sequential	0.004415	0.005304	0.006837	0.0055186667	0.0012251867
Write_Random	0.014078	0.01663	0.073635	0.034781	0.0336727362
Read_Random	0.021345	0.017107	0.020101	0.0195176667	0.0021783869

1 Thread: Latency(1MB)

	Reading 1	Reading-2	Reading-3	Average	Std Dev
Write_Sequential	1.029664	0.86553	0.991573	0.9622556667	0.0859047264
Read_Sequential	0.668664	0.673604	0.627461	0.6565763333	0.0253353089
Write_Random	0.688594	0.641534	0.507378	0.612502	0.0940316565
Read_Random	3.669532	3.034647	3.091741	3.2653066667	0.3512314355

- **2 Thread :– Latency Measurement:**

2 Thread: Latency(1B)

	Reading 1	Reading-2	Reading-3	Average	Std Dev
Write_Sequential	0.02103	0.04266	0.026475	0.030055	0.0112506233
Read_Sequential	0.004712	0.06515	0.00352	0.0244606667	0.0352430362
Write_Random	0.0572543	0.002212	0.003291	0.0209191	0.0314718307
Read_Random	0.0017361	0.0014799	0.001746	0.001654	0.0001508563

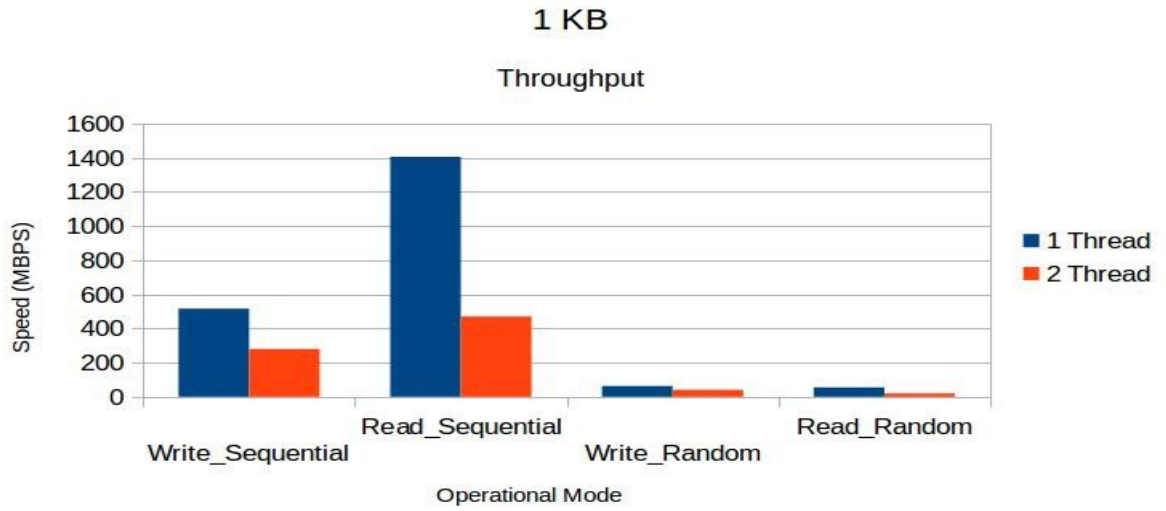
2 Thread: Latency(1KB)

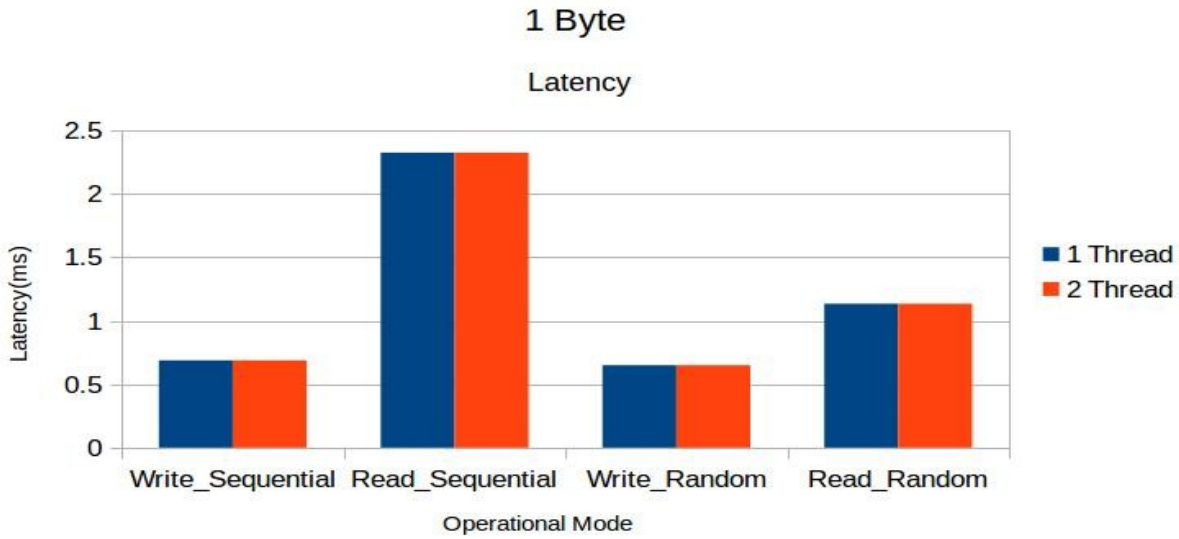
	Reading 1	Reading-2	Reading-3	Average	Std Dev
Write_Sequential	0.0291015	0.008011	0.013051	0.0167211667	0.0110138499
Read_Sequential	0.0044755	0.0042635	0.004275	0.004338	0.0001192172
Write_Random	0.0135758	0.0423099	0.024813	0.0268995667	0.0144802433
Read_Random	0.0453837	0.0635172	0.082205	0.0637019667	0.0184113453

2 Thread:Latency(1 MB)

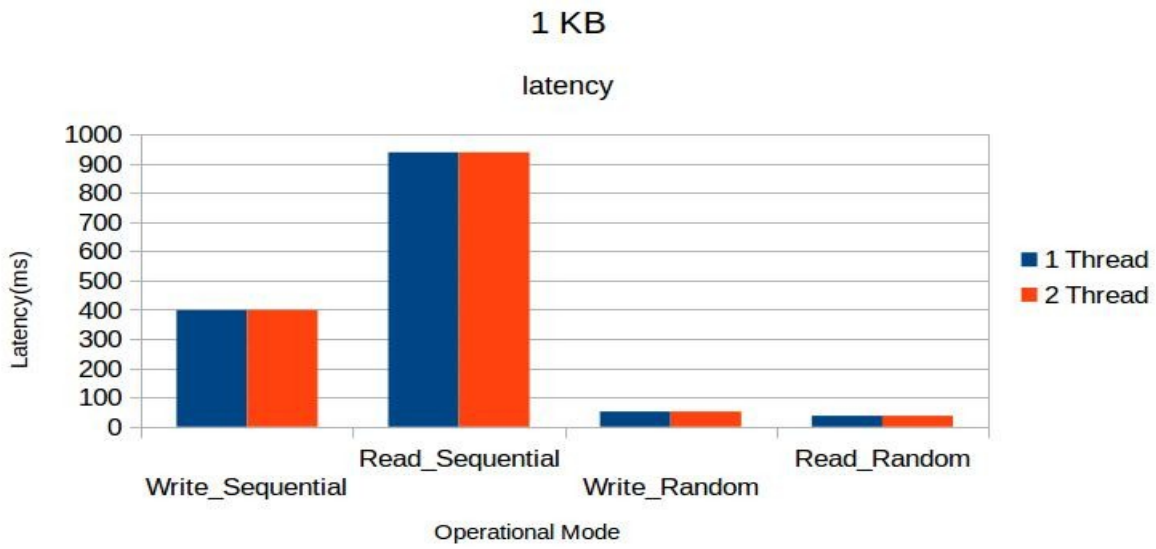
	Reading 1	Reading-2	Reading-3	Average	Std Dev
Write_Sequential	0.834116	0.828887	0.773699	0.812234	0.0334745467
Read_Sequential	0.5702245	0.6773553	0.472446	0.5733419333	0.1024902147
Write_Random	1.7647535	1.514529	1.161995	1.4804258333	0.3028229215
Read_Random	4.6521477	4.305275	5.638524	4.8653155667	0.691714268

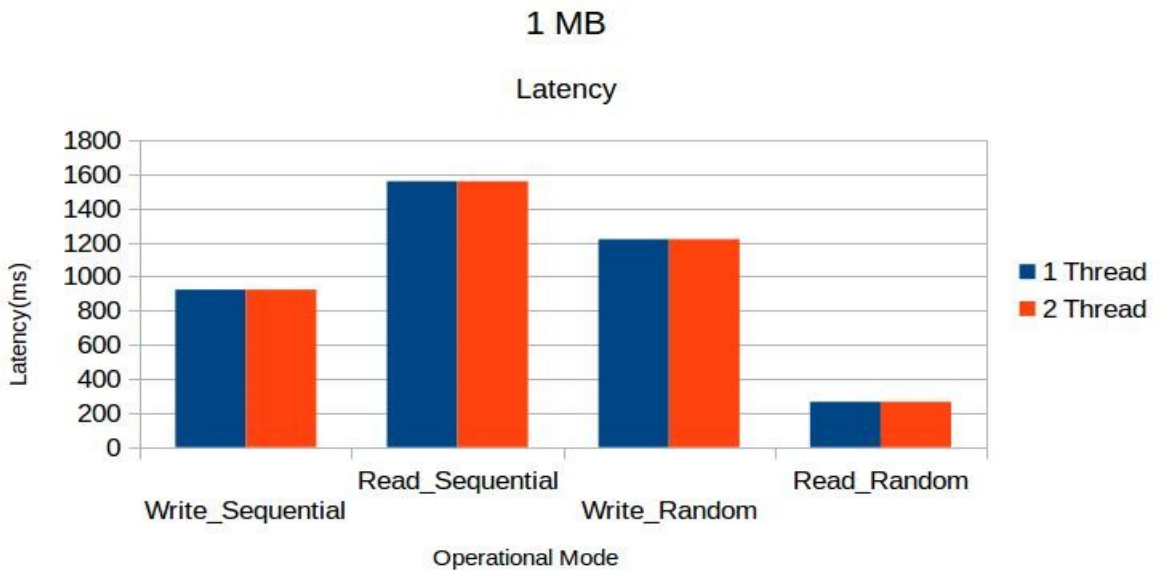
- **Comparison of performance: 1 Thread Vs 2 Thread:**
  - **Throughput:**





- **Latency:**





Here, while Evaluating Disk Performance I Observed the following things:

1) Amazon EC2 Contains Processor with one core only, hence the overall performance is degraded while running the code with two thread instead of running with Single thread.
So **I get the better performance with single thread.**
2) Overall **latency remains almost equal** for doing operation single threaded and multi-threaded

4. NETWORK:

1) TCP

- **Latency:**

1 Thread:

	Reading 1	Reading 2	Reading 3	Average	Std Dev
1 Byte	0.002	0.011	0.007	0.006666667	0.00450925
1 KB	0.005	0.006	0.009	0.006666667	0.002081666
1 MB	0.0771	0.085	0.089	0.0837	0.006055576

2 Thread:

	Reading 1	Reading 2	Reading 3	Average	Std Dev
1 Byte	0.003	0.00013	0.008	0.00371	0.00398275
1 KB	0.012	0.015	0.019	0.015333333	0.003511885
1 MB	0.167	0.21	0.142	0.173	0.034394767

- **Throughput:**

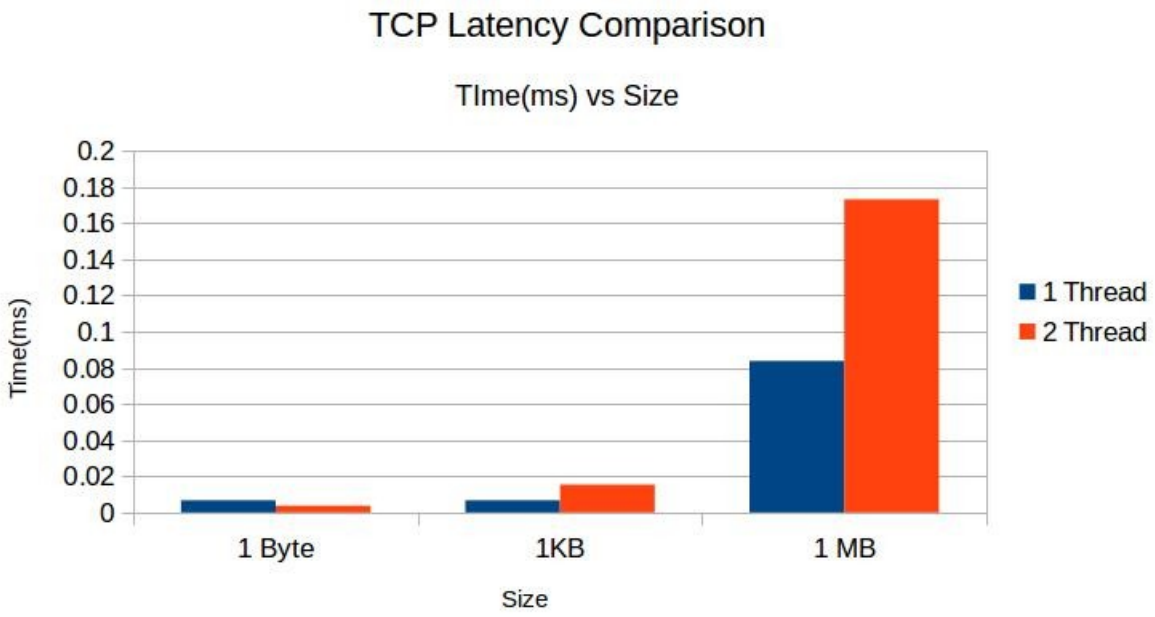
1 Thread:

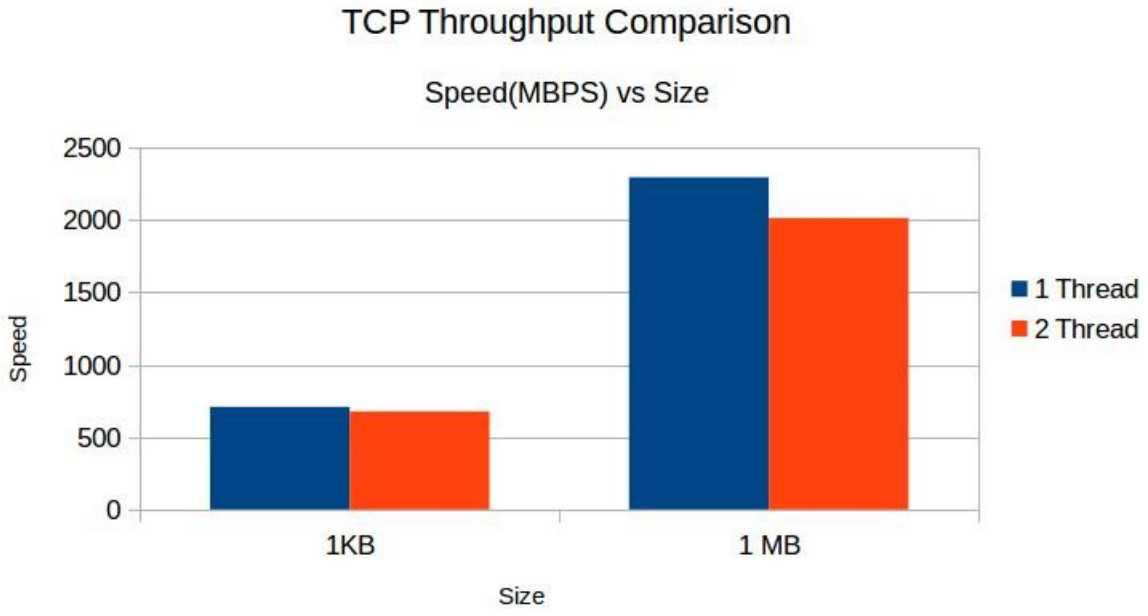
1 Thread	Reading 1	Reading 2	Reading 3	Average	Std Dev
1 Byte	5.123	6.4217	5.932	5.825566667	0.655859332
1 KB	748.92	684.241	691.473	708.2113333	35.43969684
1 MB	2231.91	2301.21	2348.42	2293.846667	58.60297802

2 Thread:

2 Thread	Reading 1	Reading 2	Reading 3	Average	Std Dev
1 Byte	3.34	4.241	3.793	3.791333333	0.450502312
1 KB	673.25	700.213	655.284	676.249	22.61413852
1 MB	2012.53	1985.03	2040.24	2012.6	27.60506656

- **TCP Performance Comparison:**



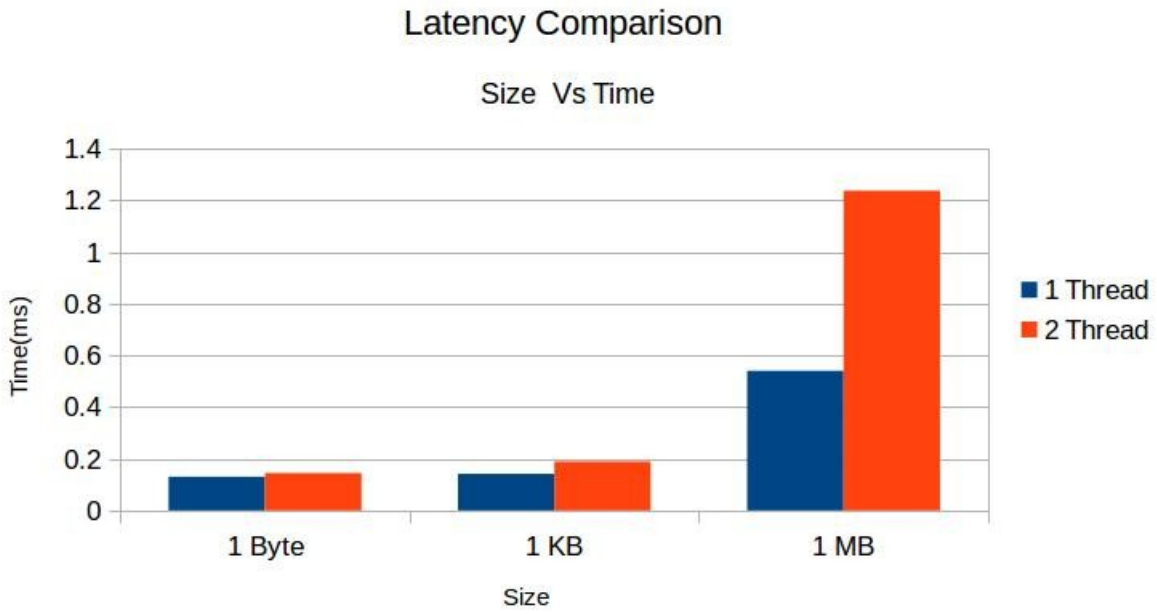


2) UDP

- **Latency:**

1Thread	Reading 1	Reading 2	Reading 3	Average	Std Dev
1 Byte	0.12	0.11	0.16	0.13	0.026457513
1 KB	0.149	0.134	0.141	0.141333333	0.007505553
1 MB	0.55	0.56	0.51	0.54	0.026457513

2 Thread	Reading 1	Reading 2	Reading 3	Average	Std Dev
1 Byte	0.072	0.29	0.072	0.144666667	0.125862359
1 KB	0.177	0.201	0.189	0.189	0.012
1 MB	1.21	1.14	1.36	1.236666667	0.112398102



- **Throughput:**

1Thread	Reading 1	Reading 2	Reading 3	Average	Std Dev
1 Byte	0.12	0.11	0.093	0.107666667	0.013650397
1 KB	102.13	116.45	110.71	109.7633333	7.206783841
64 KB	1823.27	1782.36	1902.32	1835.983333	60.9821452

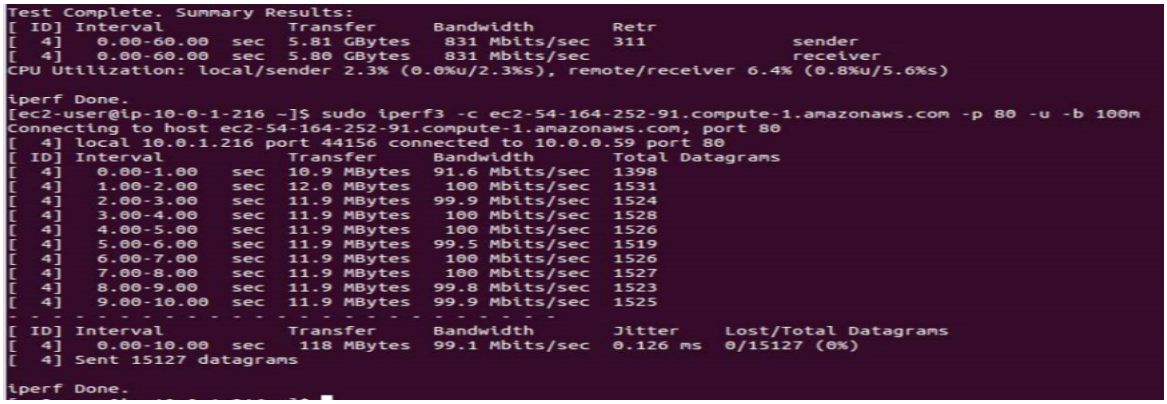
2 Thread	Reading 1	Reading 2	Reading 3	Average	Std Dev
1 Byte	0.0512	0.045	0.058	0.0514	0.006502307
1 KB	90.92	81.26	82.23	84.80333333	5.319345198
64 KB	791.64	858.23	747.61	799.16	55.6920901

From the Above given data regarding the network performance we can see here that the system is single core system and hence performance is degraded when the system tries to run the code with 2 thread rather than single thread operation.

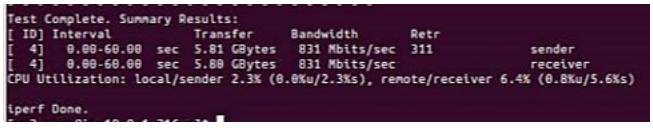
1) **Throughput is decreased and Latency is increased while running system for 2 threads.**

Extra Credit:
After Running IPerf on Amazon EC2, I achieved the following performance.

For UDP Test:
For transaction of118 Mbytes it took 15127 Packets and achieved speed of 99.1 Mbit/Sec, packet delay variation was 0.126 ms.



For TCP Test:
For transaction of 5.81 Gytes of data it achieved average speed of 831 Mbit/Second.
CPU utilization was 2.3% on sender side and 6.4% on receiver side.



References:
1)http://www.ryanchapin.com/fv-b-4-653/Java--How-To-Use-RandomAccessFile-and-FileChannel-to-
2) Write-to-a-Specific-Location-in-a-File.html
https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#isDaemon()
3)http://www.rgagnon.com/javadetails/java-0542.html
4)http://www.kdgregory.com/index.php?page=java.microBenchmark