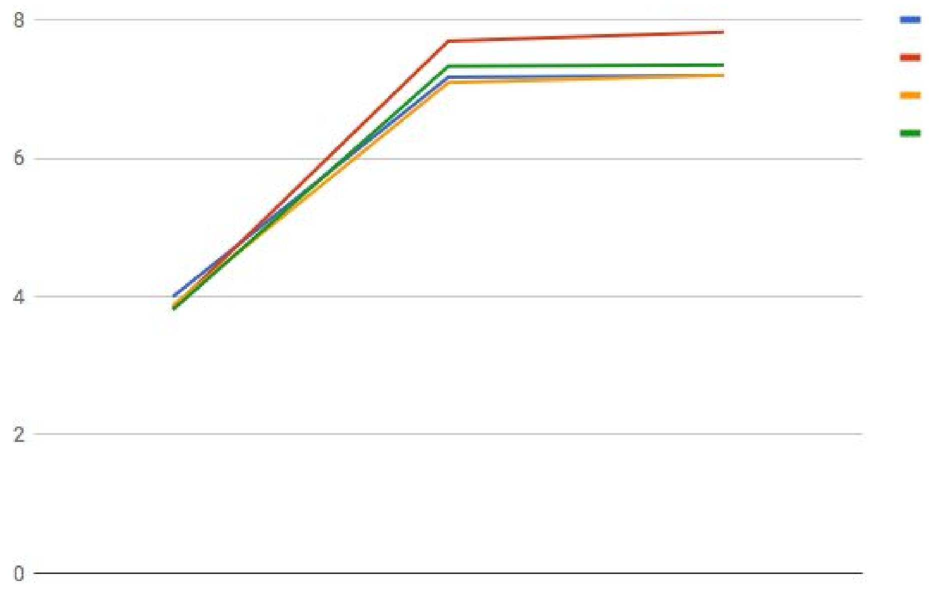
Cloud Assignment #1

CPU Benchmarking:

Workload	Concurrency	MyCPUBench Measured Ops/Sec (GigaOPS)	HPL Measured Ops/sec (GigaOPS)	Theoretical Ops/Sec (GigaOPS)	MyCPUBench Efficiency(%)	HPL Efficiency (%)
QP	1	3.996243	N/A	73.6	5.429677	N/A
QP	2	7.169763	N/A	73.6	9.739080	N/A
QP	4	7.194356	N/A	73.6	9.774864	N/A
HP	1	3.832785	N/A	73.6	5.207552	N/A
HP	2	7.691308	N/A	73.6	10.450146	N/A
HP	4	7.819654	N/A	73.6	10.624103	N/A
SP	1	3.870563	N/A	73.6	5.258918	N/A
SP	2	7.089364	N/A	73.6	9.632287	N/A
SP	4	7.197613	N/A	73.6	9.779366	N/A
DP	1	3.805855	37.5237	73.6	3.89	50.98
DP	2	7.327419	51.6862	73.6	7.03	70.22
DP	4	7.345443	57.1173	73.6	6.96	71.60

Benchmarking of type of precisions:



X-Axis : Concurrency (Number of Threads) | Y Axis : Giga Ops over 1 trillion operations

Blue: Quarter Precision

Red: Half Precision

Yellow: Single Precision

Green: Double Precision

As observed in the graph, all types of precision show very similar kind of behaviour with increase in concurrency (no of threads). Multithreading is the feature of cpu processor, to execute multiple threads or processes parallely. This is a great feature as it provides ability to execute instructions simultaneously. As seen in the graph, the GigaOps (10^9 operations /sec) increases with increase in number of threads, it almost double when concurrency is increase from 1 to 2, thus taking half the time for same number of operations. Since one node on the hyperion has two cpu cores, increasing number of threads from 2 to 4, does not really make much difference in terms of performance, this behaviour is also observed in the results by HPL benchmarking tool.

Quarter Precision operates over 1 byte char-type data size.
Half Precision operates over 2 byte short-type data size.
Single Precision operates over 4 byte int-type data size.
Double Precision operates over 8 byte double-type data size.
Since the address bus is 64 bits (8 bytes) , operating over size less than 8 bytes, does not make much difference because the the address bus and data bus used by cpu to access memory and other peripherals is 8 bytes, if the data is less than 8 bytes, it pads the data to make it 8 bytes. Thus quarter precision does not take ¼ performance of double precision(as it sounds).
The theoretical Giga Ops obtained is 73.6. This is obtained through the formula :

```
Giga Ops = (Number Of Cores) * (Address Bus Size) * (CPU Clock Rate)
```

Number Of Cores: 4 `// cpuinfo` command on the cluster,

Address bus is 64 bits or 8 bytes `// cpuinfo` command on the cluster,
CPU Clock Rate is 2.29 Ghz `// cpuinfo` command on the cluster.

The efficiency is calculated by :

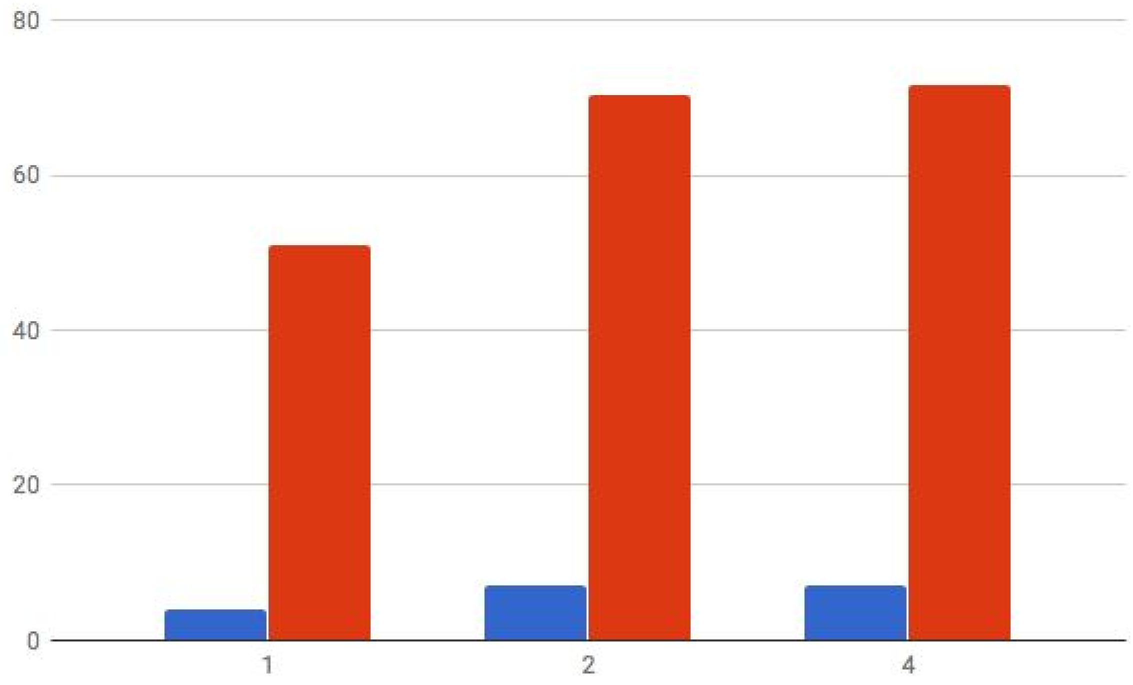
```
    Efficiency : (Obtained Giga Ops) / (Theoretical Giga Ops) * 100
```

The following chart compares the efficiency of the Benchmark code and HPL with the theoretical efficiency, but the HPL efficiency does not increase more than theoretical efficiency.
Clearly the HPL Benchmark performs much better, but the increase with concurrency behaviour remains same for both the cases.

X-Axis : Concurrency (Number of Threads) | Y Axis : Efficiency of Giga Ops w.r.t Theoretical value

Blue: Benchmark in C Efficiency

Red: HPL LinPack Efficiency



The efficiency cannot be equal to peak, because of theoretical peak is calculated with assumption that the processor performs at the speed of light, which is not practically possible.

2. Memory Benchmarking:

The Memory Benchmark operates 100 times over 1 GB of data. This data is reserved in the DRAM (Dynamic Memory), then copied and pasted into another locations allocated dynamically (Read + Write Operation). There is two type of operations, Read/Write Sequentially and Read/Write Randomly. The concurrency is 1,2 and 4 threads.

Table 2.1

Work -load	Concurrency	Block Size	MyRAMBench Measured Throughput (GB/sec)	pmbw Measured Throughput (GB/sec)	Theoretical Throughput (GB/sec)	MyRAMBench Efficiency (%)	pmbw Efficiency
RWS	1	1KB	4.896	15.53	34.128	14.367	45.50
RWS	1	1MB	3.751	17.73	34.128	11.060	51.9
RWS	1	10MB	6.644	18.54	34.128	19.494	54.3
RWS	2	1KB	8.111	22.16	34.128	23.801	64.93
RWS	2	1MB	4.243	25.34	34.128	12.449	74.24
RWS	2	10MB	10.725	29.46	34.128	31.468	86.32
RWS	4	1KB	8.828	29.89	34.128	25.903	87.58
RWS	4	1MB	6.974	31.26	34.128	20.464	91.59
RWS	4	10MB	12.053	29.23	34.128	35.365	85.64
RWR	1	1KB	2.628	18.00	34.128	7.712	52.74
RWR	1	1MB	4.532	16.70	34.128	13.296	48.93
RWR	1	10MB	6.957	13.80	34.128	20.413	40.43
RWR	2	1KB	1.962	25.73	34.128	5.815	75.39
RWR	2	1MB	8.268	28.92	34.128	24.261	84.73
RWR	2	10MB	7.574	26.88	34.128	22.223	78.76
RWR	4	1KB	1.936	19.68	34.128	5.679	48.87
RWR	4	1MB	8.899	23.38	34.128	26.111	68.50
RWR	4	10MB	12.582	21.23	34.128	36.920	62.20

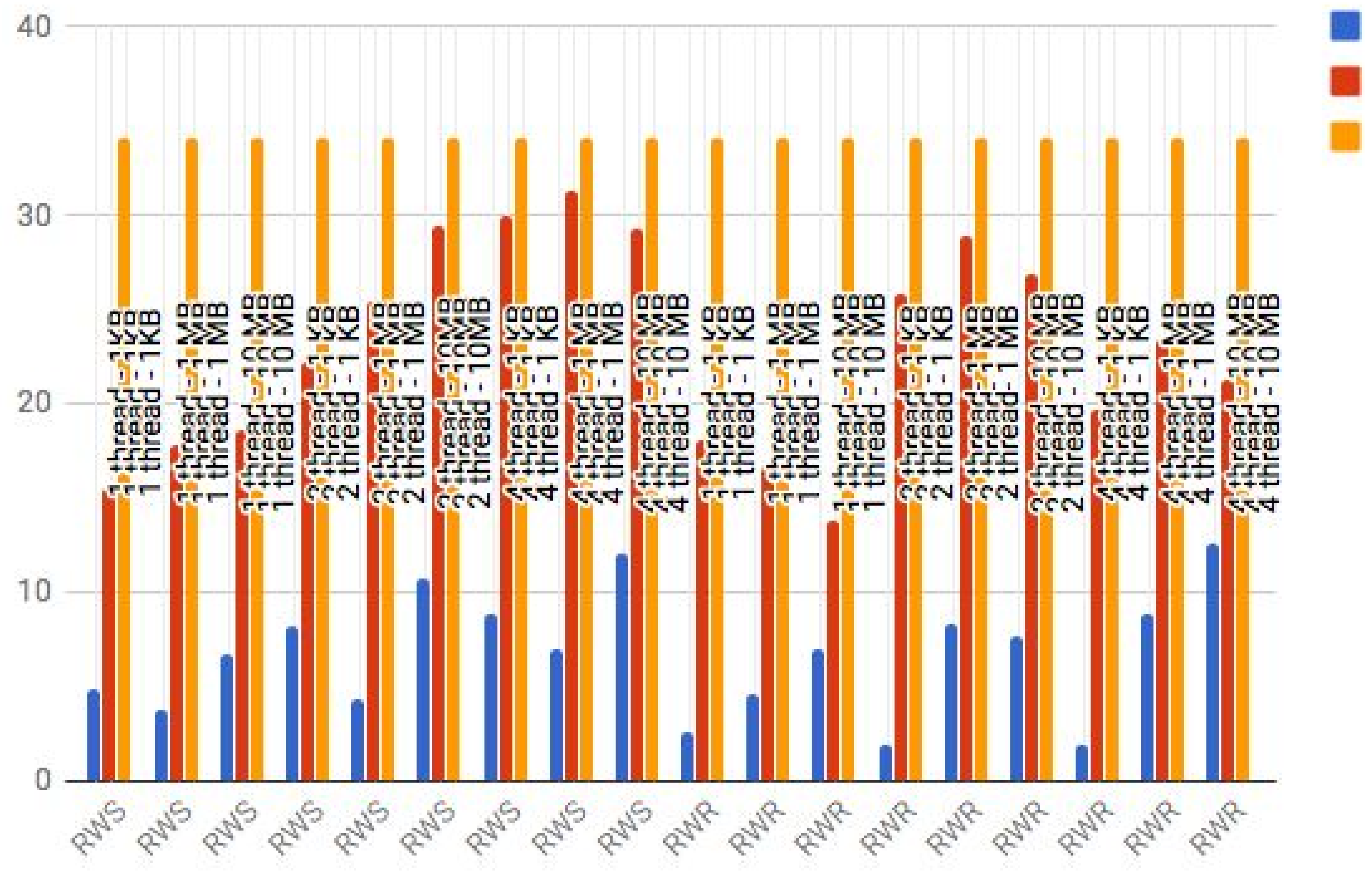
Memory Latency:-

Table 2.2:

Work -load	Concurrency	Block Size	MyRAMBench Measured Latency (uS)	pmbw Measured Latency (uS)	Theoretical Latency (uS)	MyRAMBench Efficiency (%)	pmbw Efficiency
RWS	1	1B	0.005060	N/A	0.057	10.265	N/A
RWS	2	1B	0.002696	N/A	0.057	20.141	N/A
RWS	4	1B	0.002689	N/A	0.057	20.198	N/A
RWR	1	1B	0.003165	N/A	0.057	17.011	N/A
RWR	2	1B	0.003850	N/A	0.057	13.804	N/A
RWR	4	1B	0.004419	N/A	0.057	11.899	N/A

Comparing the values in table 2.1, we get the following graph:

X-Axis : Type of operation | Y Axis : Throughput (GB/sec)



Blue: Benchmark in C
Red: PMBW
Yellow: Theoretical Peak Throughput

The graph compares the throughput in GigaBytes per second of Sequential Read/ Write and Random Read/ Write operations, with various data block sizes (1KB, 1 MB, 10 MB) and different concurrencies (1,2 and 4), with the throughput obtained through PMBW benchmarking tool and the theoretical throughput. As the graph suggests the throughput increases with increase in concurrency with exceptional case, RWR operation using block size 1 KB, the throughput decreases with increase in number of threads. Another observation can be made is, sequential read/write operation perform better than random read/write. Accessing data sequentially is much faster than accessing it randomly because reading randomly involves a higher number of seek operations than does sequential reading, while the sequential does not require a seek operation as such, since the next operation address is next address on the memory, thus can easily be found using an counter or incrementer.

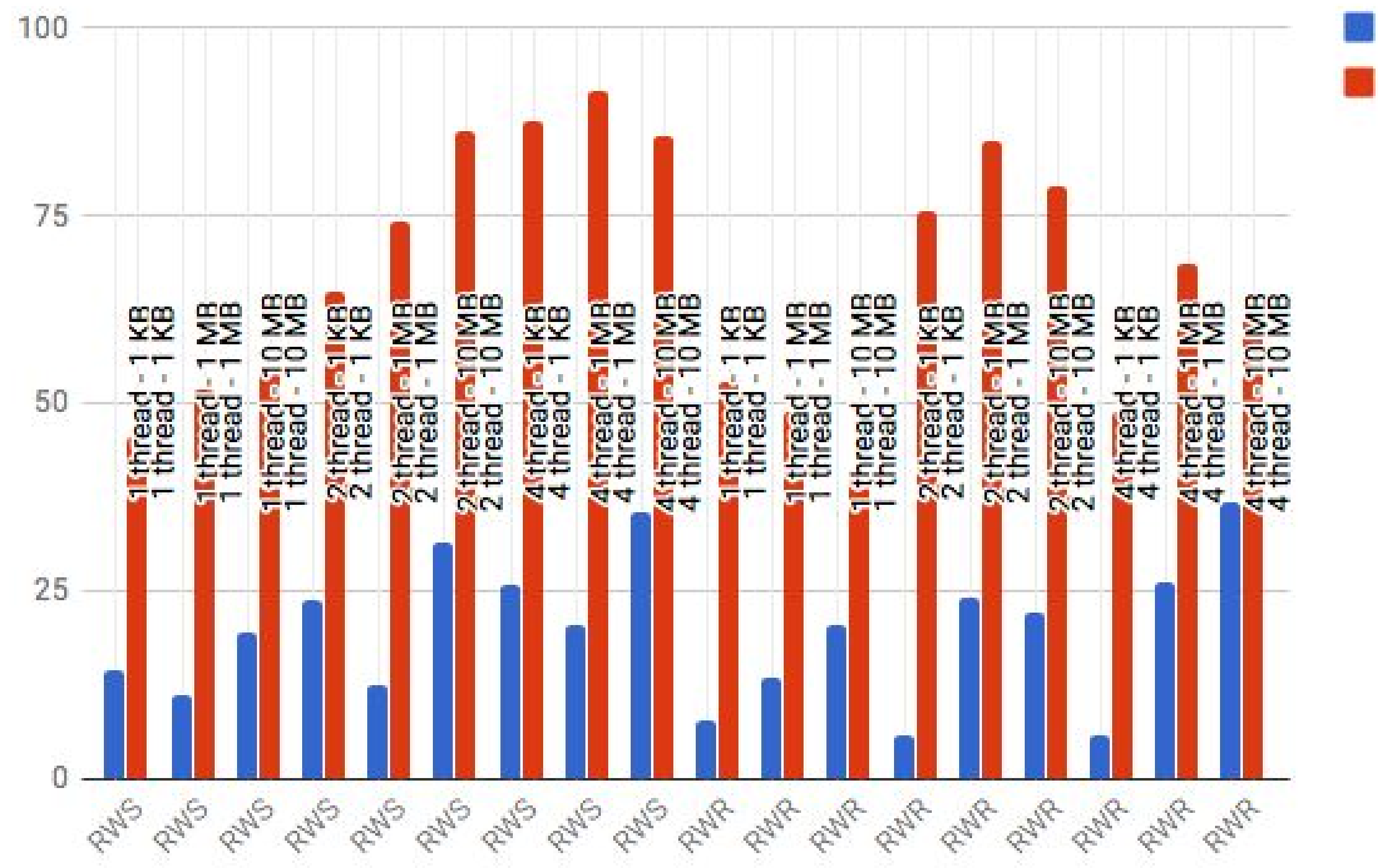
The theoretical throughput can be obtained using the formula :
Channel width (bits/transfer) × transfers/second x no. of CPUS= bits transferred/second

*Assuming the cluster has 2133 MHz DDR3 DRAM with 64 bit address and data bus. Since I could not access the memory information on the cluster due to not having root privileges. Also the cluster has Sandy Bridge that supports DDR3 DRAM.
Intel Website for Sandy Bridge

Channel width = 64 bits or 8 bytes,
Transfers/second = 2.133 x 10^9 (2133 MHZ),
Number of CPUS per node = 2.
Thus Theoretical value = 2 * 8 * 2.133 = 34.128 GB/sec.

Latency is the time it takes the memory to respond to a command request. For the benchmark (read+write) we use 1 byte of data, and the result obtained is in terms of time(microseconds) per access. The total number of such operations performed are 100 million operations. The results are filled in Table 2.2. The theoretical latency value of 2133 MHz is taken as 57 nanoseconds (7-10 cpu cycles). As observed from the table the efficiency increases with increase in concurrency for Sequential Read and decreases for Random Read. Latency could not be obtained for PMBW tool.

The following graph compares efficiency of results obtained through benchmark code and PMBW tool:
Blue: Throughput obtained through benchmark in C
Red: Throughput obtained through PMBW tool
X-Axis : Type of operation | Y Axis: Efficiency of Throughput(GB/sec) w.r.t Theoretical value



As seen the graph the PMBW performs much better. Even in result obtained through PMBW, for RWR operation using block size, the throughput decreases with increase in concurrency, as seen the value almost halves for 1 KB with 4 threads compared to 1 KB with 2 threads.

3. Disk Benchmarking:

The Disk Benchmark operates 10 GB of data. A file of size 10 GB is created and stored on the disk. The Read operation is performed over this file. The operations is of Sequential Read or Random Read. The Write operation is performed by creating a file of size of 10 GB and measuring the throughput obtained (MB/sec), again it can Sequential Write or Random Write. The concurrency is 1,2 and 4 threads.

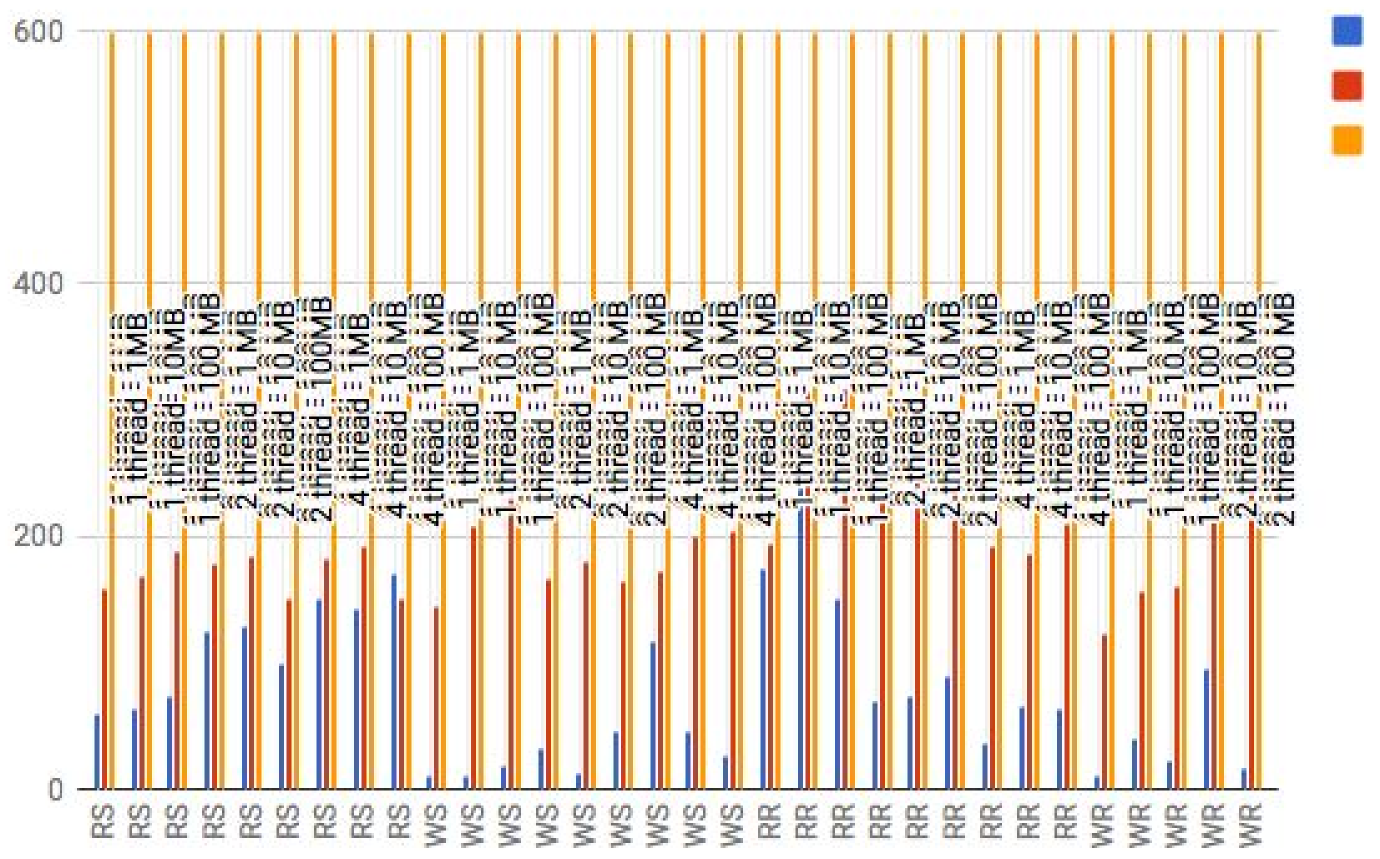
Table 3.1

Work load	Concurrency	Block Size	MyDiskBench Measured Throughput (MB/sec)	IOZone Measured Throughput (MB/Sec)	Theoretical Throughput (MB/Sec)	MyDiskBench Efficiency(%)	IOZone Efficiency (%)
RS	1	1 MB	59.5365	158.44	600	9.92	26.40
RS	1	10 MB	63.9441	169.25	600	10.65	28.20
RS	1	100 MB	73.3884	188.39	600	12.23	31.39
RS	2	1 MB	125.5187	179.01	600	20.91	29.83
RS	2	10 MB	128.6011	185.19	600	21.43	30.86
RS	2	100 MB	99.0778	151.45	600	16.51	25.24
RS	4	1 MB	150.5163	182.23	600	25.08	30.37
RS	4	10 MB	142.6561	191.87	600	23.77	31.97
RS	4	100 MB	170.7277	151.59	600	28.45	25.26
WS	1	1 MB	10.2473	144.13	600	1.70	24.02
WS	1	10 MB	10.7787	207.51	600	1.79	34.58
WS	1	100 MB	18.2857	229.54	600	3.04	38.25
WS	2	1 MB	32.5097	167.33	600	5.41	27.88
WS	2	10 MB	12.5798	181.23	600	2.09	30.20
WS	2	100 MB	45.6794	165.54	600	7.59	27.59
WS	4	1 MB	116.3636	172.34	600	19.39	28.72
WS	4	10 MB	46.5454	200.28	600	7.75	33.38

WS	4	100 MB	26.6756	204.25	600	4.44	34.04
RR	1	1 MB	174.60	195.23	600	29.10	32.53
RR	1	10 MB	280.36	314.05	600	46.41	52.34
RR	1	100 MB	150.10	317.14	600	25.01	52.85
RR	2	1 MB	70.87	242.58	600	11.81	40.43
RR	2	10 MB	73.68	277.30	600	12.28	46.21
RR	2	100 MB	89.59	301.66	600	14.93	50.27
RR	4	1 MB	37.25	192.92	600	6.20	32.15
RR	4	10 MB	65.98	185.63	600	10.99	30.93
RR	4	100 MB	64.44	210.13	600	10.74	35.02
WR	1	1 MB	10.5899	122.42	600	1.76	20.40
WR	1	10 MB	39.3846	157.14	600	6.56	26.19
WR	1	100 MB	23.2439	159.88	600	3.87	26.64
WR	2	1 MB	inf	193.72	600		32.28
WR	2	10 MB	95.3000	211.92	600	15.88	35.32
WR	2	100 MB	17.4014	269.15	600	2.90	44.85
WR	4	1 MB	inf	201.80	600		33.63
WR	4	10 MB	inf	141.94	600		23.67
WR	4	100 MB	34.0571	200.94	600	5.6761	33.49

**Note: Readings that are in red, have been adjusted, because when executed on the cluster, threw an error of DATA LIMIT for READ/WRITE operations.
Some readings also gave infinite value due to the unstable cluster, they executed fine on local machine.

Comparing Table 3.1 we get the following graph:
Blue: Benchmark in C
Red: IOZone
Yellow: Theoretical Peak
X-Axis : Type of operation | Y Axis: Throughput(MB/sec) w.r.t Theoretical value



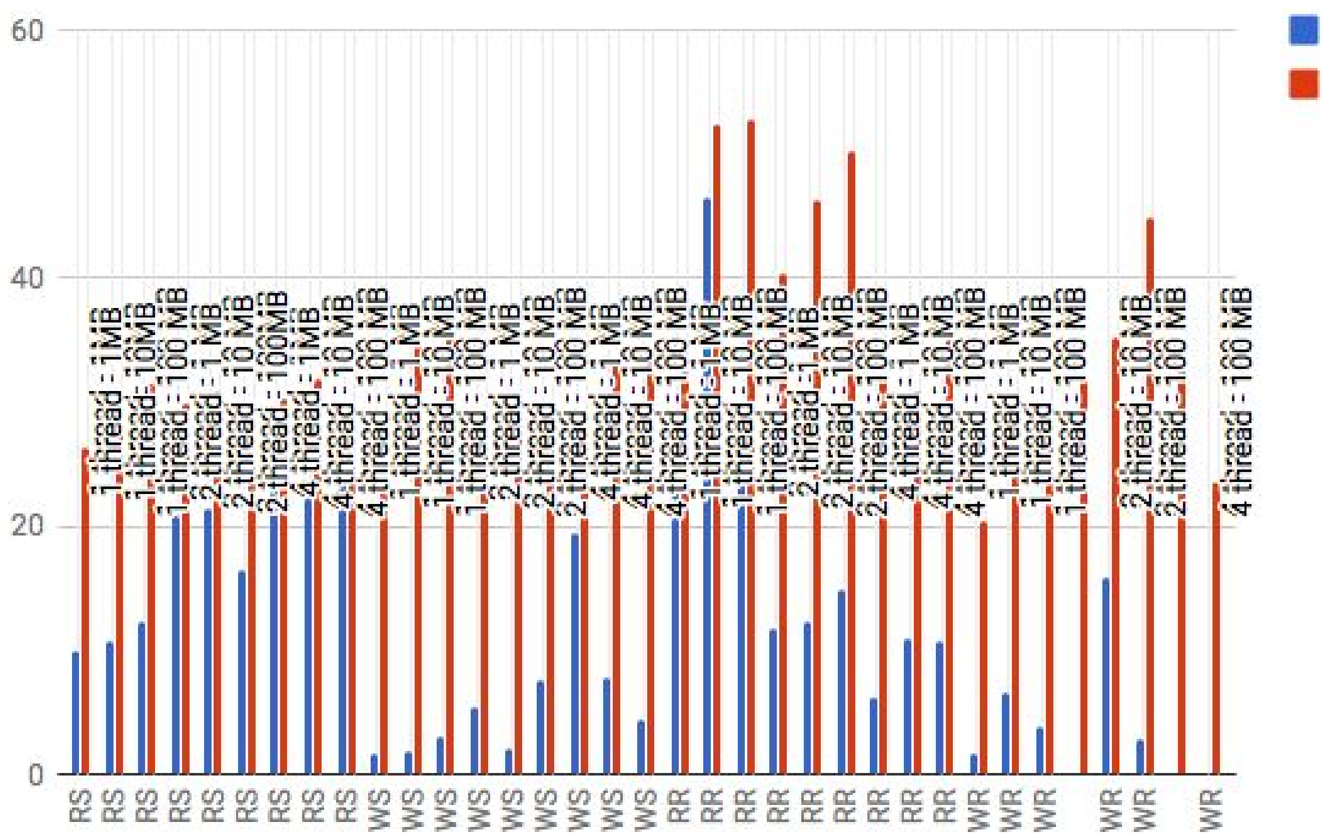
Since the disk storage of the cluster is of type SSD(Solid State Drive), it is much faster than usual Hard Disk type of storage. The workload is 10 GB data file, the operations performed are Sequential Read, Sequential Write, Random Read and Random Write. The operations are performed over 1 MB, 10 MB and 100 MB data transfer block sizes and with different concurrencies(1,2 and 4).

As seen in the graph, the read operation is much faster than the write operations. Since the write operations involved twice as much access to disk by the cpu compared to read operations, thus takes more time. For the sequential read function performance increases with increase in councurrency. The performance also increases with increase in data file block size. This interpolation is not that consistent with random read and write and function. Since the disk is one

resource accessed by multiple cpu cores and thread in the multi channel architecture, this leads to conflict in accessing memory, if the threads try to access same offset value. Since `mutex` function is used, which locks a thread that is executing and is not released till the thread has finished executing. Due to this, the thread that is trying to access the same offset, has to wait for the previous thread to finish. Thus the performance is not so consistent with increase in concurrency. The random read operation performance decreases with increase in block size and concurrency, again the same reason.
The IOZone tool performs much better. Even the IOZone results are not so consistent for random read/write operations with respect to block sizes and concurrency.

Theoretical Peak: The theoretical peak of the SSD disk of the cluster is assumed to be 600 MB/sec. This means theoretically the throughput is 600 MB/sec, about 600 MB of data can be accessed and transferred between the disk and cpu.
https://en.wikipedia.org/wiki/Solid-state_drive, as provided in this web page, I have used 600 MB/sec as the peak throughput of the SSD.

Graph comparing the efficiency of the benchmarking in C and IOZone tool:
Blue: Throughput obtained through benchmark in C
Red: Throughput obtained through IOZone tool
X-Axis : Type of operation | Y Axis: Efficiency of Throughput(MB/sec) w.r.t Theoretical value



As seen in the graph for sequential read the results obtained are close to the result obtained by the IOZone. Also random read operation for block size of 10 MB and 100 MB are irregular readings in both benchmark in C and IOZone. The IOZone efficiency is also no so consistent with random operations. Thus it can be concluded that sequential read/write is much robust and consistent for the cluster hardware specs.

Latency: Latency is the time it takes the disk to respond to a command request. For the benchmark (read+write) we use 1 byte of data, and the result obtained is in terms of time(microseconds) per access. The total number of such operations performed are 1 million operations. The results are filled in Table 3.2.
Theoretical latency: Assuming 4.12 ms as theoretical peak of SSD.
IOPS(In/Out operations per second) can be calculated by:

```
IOPS Theoretical = Theoretical Throughput / Block Size.
```

**Considering theoretical throughput to be 712 Mb/sec

Table 3.2

Work load	Concurrency	Block Size	MyDiskBench Measured Latency (ms)	IOZone Measured Latency(ms)	Theoretical Latency (uS)	MyDiskBench Efficiency(%)	IOZone Efficiency (%)
RR	1	1 KB	8.72	6.43	4.16	47.70	64.69
RR	2	1 KB	16.13	5.97	4.16	25.79	69.68
RR	4	1 KB	21.56	6.01	4.16	19.29	69.21
RR	8	1 KB	7.89	4.9	4.16	52.72	84.89
RR	16	1 KB	30.74	4.05	4.16	13.53	102.71
RR	32	1 KB	46.42	2.8	4.16	8.96	148.57
RR	64	1 KB	46.54	2.89	4.16	8.93	143.94
RR	128	1 KB	63.19	2.16	4.16	6.58	192.59
WR	1	1 KB	6.14	4.95	4.16	67.75	84.04
WR	2	1 KB	6.38	3.84	4.16	65.20	108.33
WR	4	1 KB	8.71	3.68	4.16	47.76	113.04
WR	8	1 KB	16.31	3.01	4.16	25.50	138.23
WR	16	1 KB	12.34	2.87	4.16	33.71	144.94

WR	32	1 KB	9.83	2.17	4.16	42.31	191.70
WR	64	1 KB	14.36	1.98	4.16	28.96	210.10
WR	128	1 KB	21.57	1.7	4.16	19.28	244.70

Table 3.3

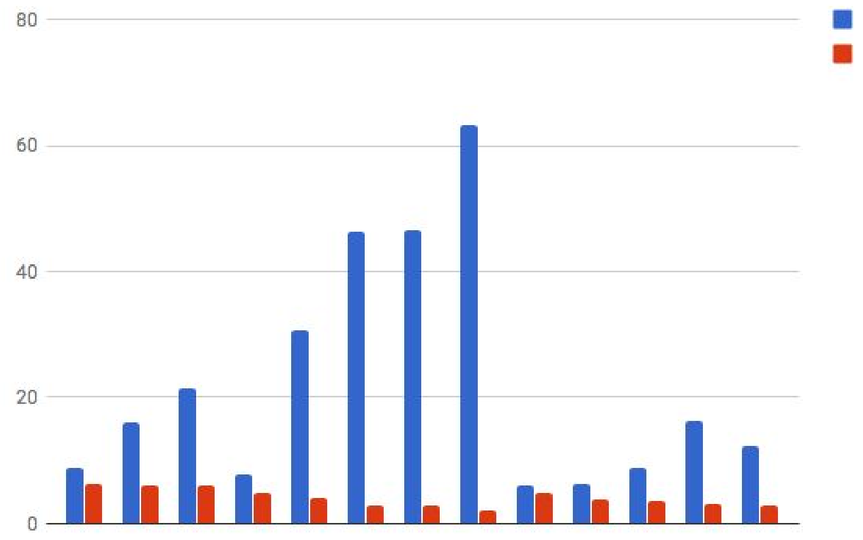
Work load	Concurrency	Block Size	MyDiskBench Measured IOPS	IOZone Measured IOPS	Theoretical IOPS	MyDiskBench Efficiency(%)	IOZone Efficiency (%)
RR	1	1 KB	118743.23	416182.31	695312	17.07	28.53
RR	2	1 KB	54645.34	832267.34	695312	7.85	6.56
RR	4	1 KB	55234.34	559767.16	695312	8.24	9.86
RR	8	1 KB	178233.34	2131348.45	695312	25.63	8.36
RR	16	1 KB	34345.45	3356181.12	695312	4.93	1.02
RR	32	1 KB	23675.12	1004579.98	695312	5.40	2.35
RR	64	1 KB	25099.56	1918234.23	695312	3.60	1.30
RR	128	1 KB	18344.12	4098207.98	695312	1.63	0.44
WR	1	1 KB	145123.34	194123.87	695312	20.87	74.32
WR	2	1 KB	117564.45	75233.24	695312	16.23	156.21
WR	4	1 KB	170984.13	2908930.87	695312	24.59	5.87
WR	8	1 KB	52094.34	308345.35	695312	7.49	16.89
WR	16	1 KB	98975.32	160884.23	695312	14.23	61.51
WR	32	1 KB	164424.234	1187123.23	695312	23.64754729	13.85
WR	64	1 KB	80834.234	2170234.35	695312	11.62560606	3.72
WR	128	1 KB	57563.234	2180234.12	695312	8.278763203	2.64

The following graphs compare IOZone and Benchmark in C in terms of latency and latency(ms) and IOPS.

Blue: Throughput in ms obtained through benchmark in C

Red: Throughput in ms obtained through IOZone tool

X-Axis : Type of operation | Y Axis: Throughput(ms)

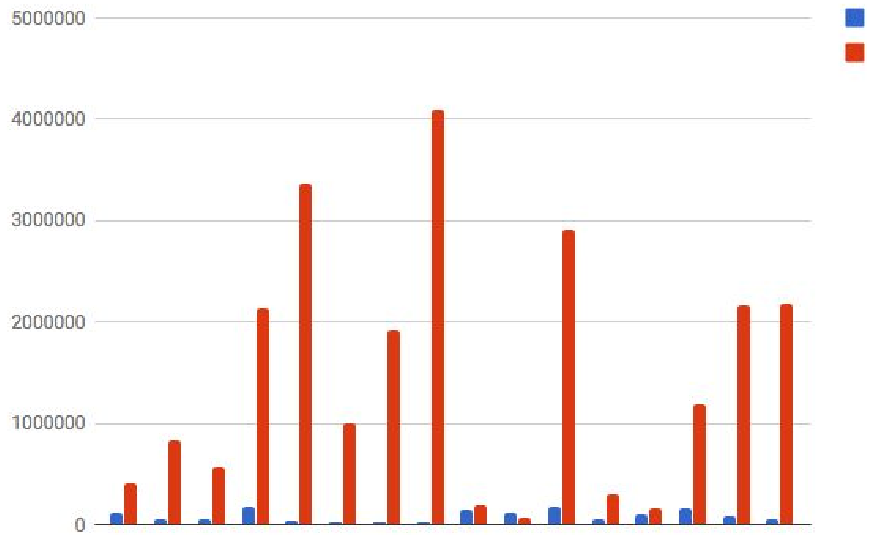


As seen in the graph the latency obtained by the IOZone tool is much less compared to benchmark in C.

Blue: Throughput in ms obtained through benchmark in C

Red: Throughput in ms obtained through IOZone tool

X-Axis : Type of operation | Y Axis: Throughput(ms)



As seen in the graph, the IOZone IOPS is much greater than benchmark in C.

4. Network Benchmarking:

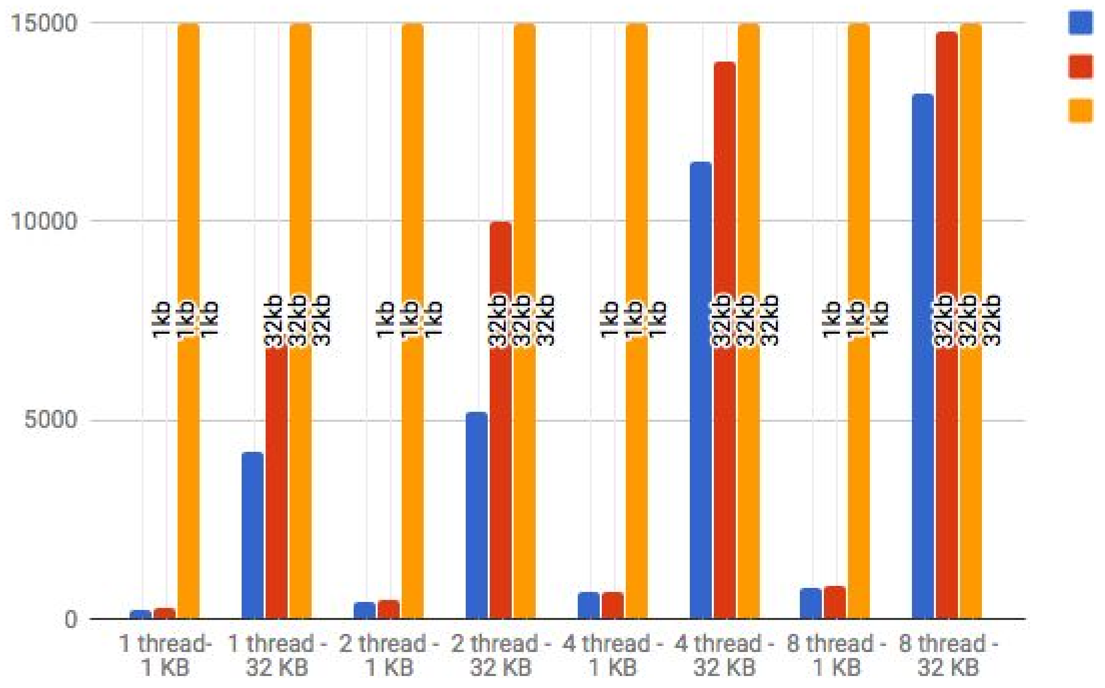
The Network Benchmark operates 100 times over 1 GB of data. The data is transferred over the network from the client to the server using TCP (Transmission Control Protocol) and UDP (User Datagram Protocol). The data is transferred using block size of 1KB and 32KB. The concurrency is 1,2,4 and 8 threads.

Table 4.1

Protocol	Concurrency	Block Size	MyNETBench Measured Throughput (Mbps)	Iperf Measured Throughput (Mb/s)	Thereotical Throughput (Mb/s)	MyNETBench Efficiency (%)	Iperf Efficiency (%)
TCP	1	1kb	233.96	270.00	15000.00	1.559733333	1.8
TCP	1	32kb	4225.43	7160.00	15000.00	28.16953333	47.73333333
TCP	2	1kb	424.30	468.00	15000.00	2.828666667	3.12
TCP	2	32kb	5202.91	10000.00	15000.00	34.68606667	66.66666667
TCP	4	1kb	659.97	680.00	15000.00	4.3998	4.533333333
TCP	4	32kb	11494.25	14000.00	15000.00	76.62833333	93.33333333
TCP	8	1kb	757.81	832.00	15000.00	5.052066667	5.546666667
TCP	8	32kb	13234.43	14800.00	15000.00	88.22953333	98.66666667
UDP	1	1kb					
UDP	1	32kb					
UDP	2	1kb					
UDP	2	32kb					
UDP	4	1kb					
UDP	4	32kb					
UDP	8	1kb					
UDP	8	32kb					

Comparing values of the throughput obtained through Benchmark in C and iPerf.

X-Axis : Number of Threads/ Block Size | Y Axis : Throughput (Gigabits/sec)
Blue: Benchmark in C
Red: IOZone
Yellow: Theoretical Peak

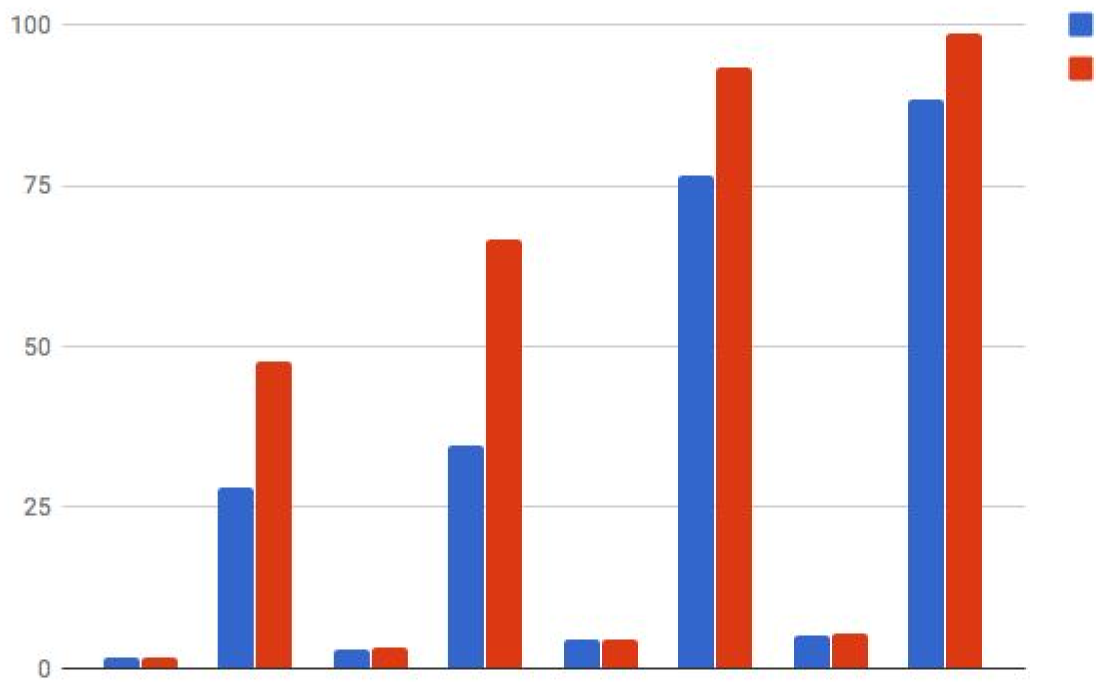


The output obtained are in Megabits/sec: To convert them to MegaBytes/sec, divide the value by 8.
The theoretical peak is 15000 Megabits/sec. Since this the benchmarking is executed on the local cluster intranode system using the loopback address, it never reaches the internet traffic thus it is very fast. The TCP protocol is very robust protocol unless the UDP.
As seen in the graph, the throughput is very large for block size of 32 KB compared to 1 KB, thus block size of 1 KB creates a bottleneck and the throughput is so poor. The performance increases exponentially for increase in number of threads, throughput for block size 32 KB with 8 threads almost reaches the theoretical value for benchmark in C and iperf tool.
Using the `srun` command, two different blue compute nodes are set up for server and client each. Then server and client are each setup on those nodes and communication takes place.

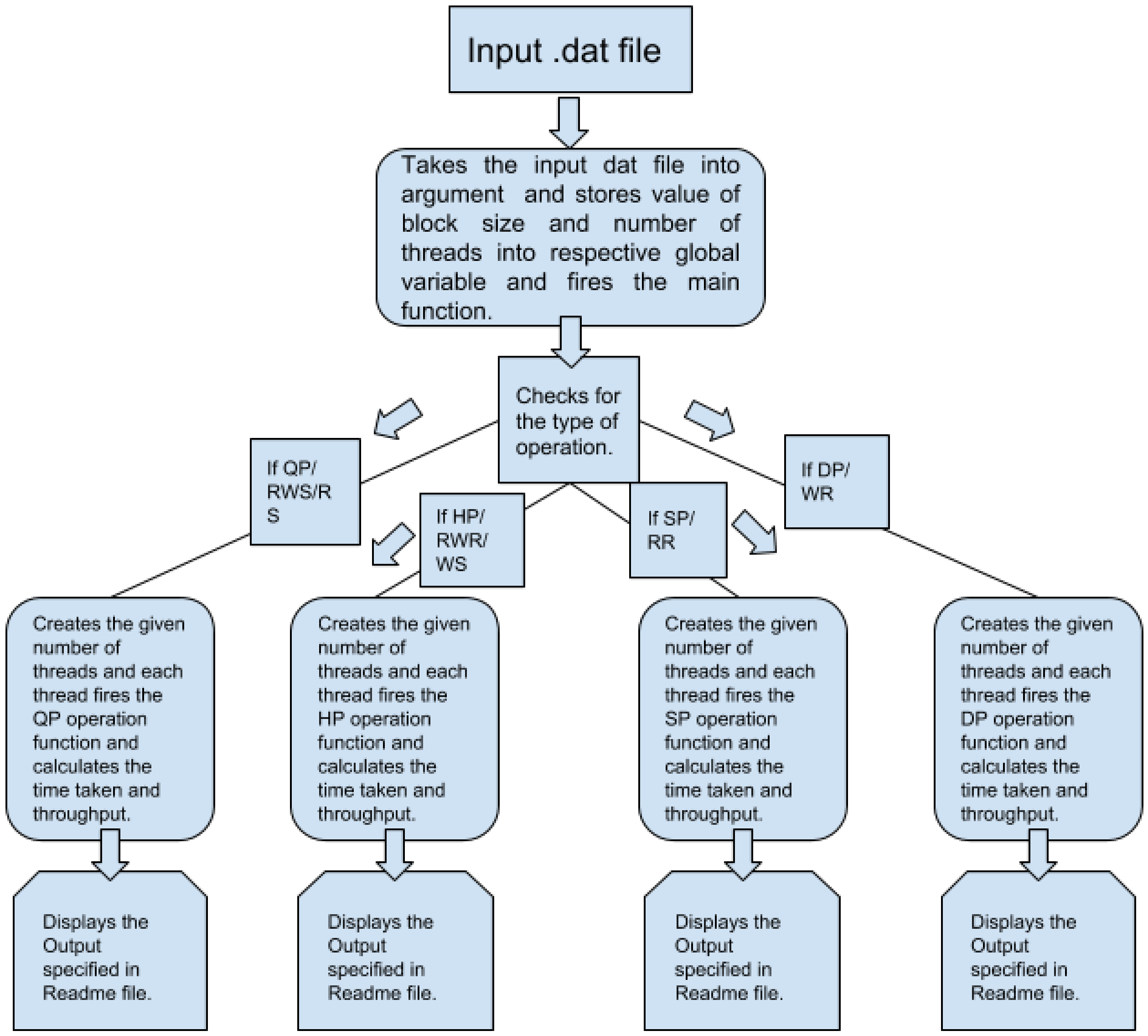
Comparing iperf performance with the benchmark code in C. Clearly iperf performs better for lesser block size and threads but the performance of both are similar with bigger block size and higher concurrency.
The throughput cannot be greater than the theoretical peak because there is always some TCP overhead, even though it is a loopback.

From table 4.1, graph to compare efficiencies of iperf and benchmark in C with respect to the theoretical peak.

Blue: Efficiency of throughput obtained through benchmark in C w.r.t to theoretical peak.
Red: Efficiency of throughput obtained through iperf w.r.t to theoretical peak.
X-Axis : Type of tool(iperf and benchmark in C) | Y Axis: Efficiency

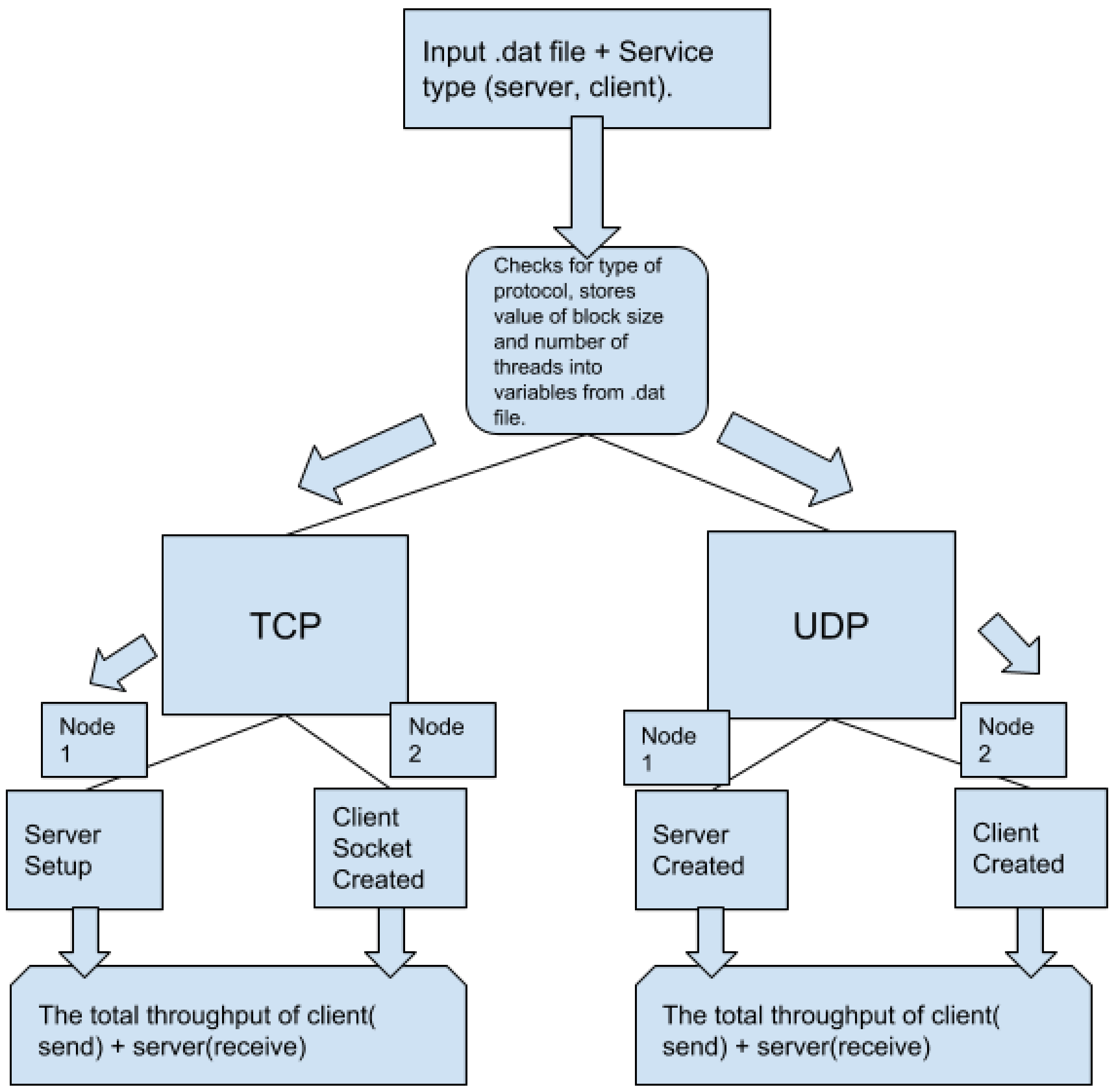


Code Design Document:
The program flow is very easy and robust:


Input .dat file
Takes the input dat file into argument and stores value of block size and number of threads into respective global variable and fires the main function.
Checks for the type of operation.
If QP/ RWS/R S
If HP/ RWR/ WS
If SP/ RR
If DP/ WR
Creates the given number of threads and each thread fires the QP operation function and calculates the time taken and throughput.
Creates the given number of threads and each thread fires the HP operation function and calculates the time taken and throughput.
Creates the given number of threads and each thread fires the SP operation function and calculates the time taken and throughput.
Creates the given number of threads and each thread fires the DP operation function and calculates the time taken and throughput.
Displays the Output specified in Readme file.
Displays the Output specified in Readme file.
Displays the Output specified in Readme file.
Displays the Output specified in Readme file.

This flow is for the CPU benchmark. The flow remains similar except for Memory and Disk benchmarking plus the latency part.

The TCP and UDP has a bit different flow:
The server side of code needs to be deployed on different node while the client needs to be deployed on the another node, for the communication to take place.


Input .dat file + Service type (server, client).
Checks for type of protocol, stores value of block size and number of threads into variables from .dat file.
TCP
UDP
Node 1
Node 2
Node 1
Node 2
Server Setup
Client Socket Created
Server Created
Client Created
The total throughput of client(send) + server(receive)
The total throughput of client(send) + server(receive)

The design is very workload/ protocol specific. Code snippet example :

```
if(strncmp(type, "RWS", 3) == 0){
        if (pthread_mutex_init(&mutex, NULL) != 0)
        {
            printf("\n mutex init has failed\n");
            return 1;
        }
        for(i = 0; i < no_of_threads; i++){
            pthread_create(&tid, NULL, sequentialReadWrite, (void
*)i);
            //if(rc){
            //printf("Counld not create a thread\n");
            //}
        }
        for(i = 0; i < no_of_threads; i++){
           pthread_join(tid, NULL);
           //if(rc){
           //printf("Cound not join the thread\n");
                  //}
        }
        pthread_mutex_destroy(&mutex);
 }
```

The above code takes the input argument from .dat file, compares the workload type and then fires the number of threads given in .dat file.

This creates threads that execute following function parallely :
Example:

```
void *sequentialReadWrite(void *args){
    pthread_mutex_lock(&mutex);
    int o,m;
    int offset = 0;
    if(block_size > 1) {
        //printf("%d\n", no_of_threads );
        for(o=0;o<100/no_of_threads;o++) {
            for (m = 0; m<SIZE/block_size * sizeof(char); m++) {
                memcpy(dest + offset, src + offset, block_size*
sizeof(char));
                offset = offset + block_size * sizeof(char) ;
                if (offset == SIZE) {
                    offset = 0;
                }
                if (offset > SIZE) {
                    break;
                }
            }
        }
    }
    else if (block_size == 1) {
        //printf("Latency Calculation");
        for(o=0;o<50000000/no_of_threads;o++) {
            memcpy(dest + offset, src + offset, block_size *
sizeof(char));
            offset = offset + block_size ;
            if (offset == SIZE) {
                offset = 0;
            }
            if (offset > SIZE) {
                    break;
            }
        }
    }
    pthread_mutex_unlock(&mutex);
}
```

Once this function is finished executed the output is calculated and then displayed. Each type of workload function has its own logic and uses various functions. The logic is according to the requirements mentioned in the assignment document provided. Most of the workload functions uses for loop to iterate over required number of operations.