DV1567 Lab 1

Group Members	
Sreemayi Reddy Rapolu	980417-1727
Venkata Satya Sai Ajay Daliparthi	970316-2637
Wenjie Zhan	971015-0237

Task 1

1. Terminal Command

The command we use in this Task is `lshw` and `cat /etc/os-release`

2. SUT Info

- **Processors**

  No. of Processors: 1

  Name: Intel Core i5-8259U

  Speed: 2.30GHz

  Cores: 4
- **Motherboard**

  Parallels Virtual Platform
- **Memory**

  2 GB
- **Storage**

  64 GB
- **Operating Systems**

  Ubuntu 18.04.1 LTS

Task 2

1. **performance metrics/counters to detect memory leak**

   commands and related flags: `Sar -r`

2. **metrics/counters that you can use to analyze the CPU utilization**

   commands and related flags: `sar -P ALL 1 15`

3. **metrics/counters that you can use to analyze disk throughput**

   commands and related flags: `sar -b`

4. **sample performance counters each 2 second?**

   ```
   Top -d <number of seconds> | sar 2 20 | watch -n 2 ps
   ```

5. **automatically stop the sampling after 3 minutes**

   ```
   Timeout 3m sar 1 20

   Timeout 3m top -d 1

   Timeout 3m watch -n 1 ps
   ```

6. **save the sampled data into a file and load it again**

   ```
   Save: sar -u 1 3 -o a.txt , load : sar -f a.txt

   Save: top -b -n 1 > b.txt , load: less b.txt

   Save: ps > c.txt , load: less c.txt
   ```

7. **sample statistics for a specific processor**

   ```
   sar -P 0 (CPUId-0)

   sar -P 1 (CPUId-1)
   ```

8. **sample statistics for a process/thread or group of processes/threads**

   ```
   Top -H | grep ‘<processName1>\| <processName2>\|….’
   ```

   (for group of threads , for single thread include only one name)

   ```
   Top | grep ‘<processName1>\| <processName2>\|….
   ```

   (for group of processes, for single process include only one process)

9. **track a specific process and related threads**

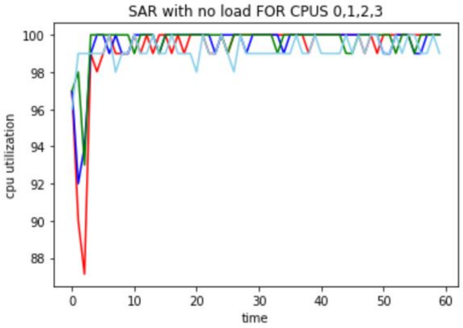
   ```
   ps -T -p <PID> (for thread)

   ps -p <PID> (for process)
   ```

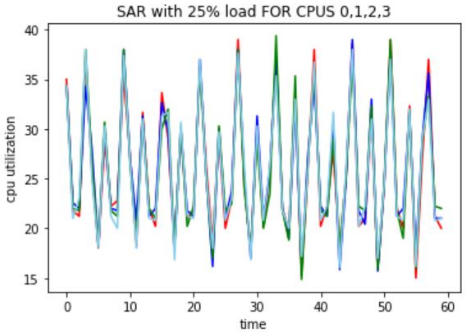
TASK 3

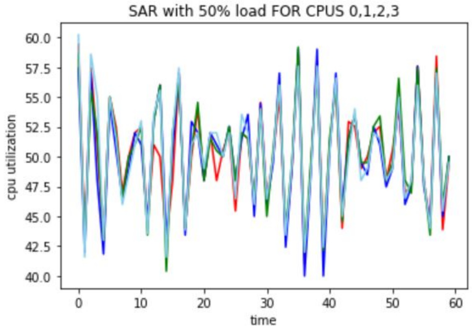
4.1.1

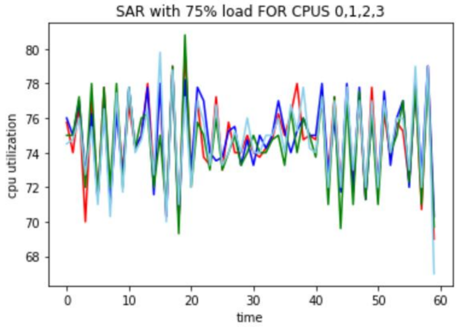
This task has been done for SAR values. The results obtained were:

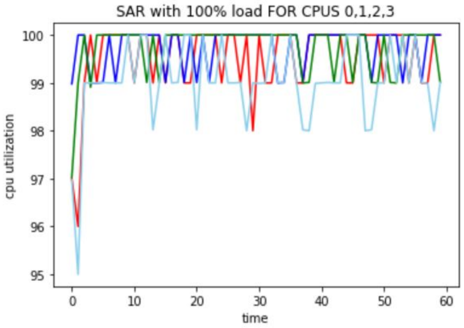
- Plot the time series of the CPU utilization for all the CPUs and the total CPU utilization



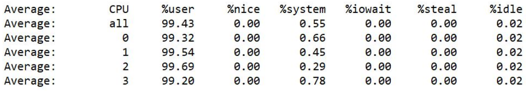




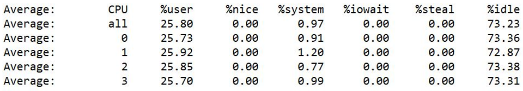




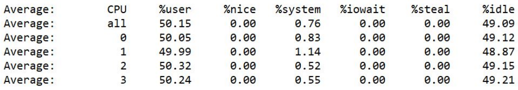
- Compute the average values and the 90,95, 99 percentiles



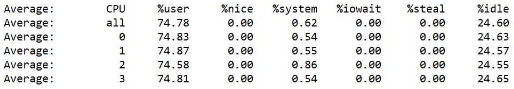
```
Average:        CPU     %user     %nice   %system   %iowait    %steal     %idle
Average:        all     99.43      0.00      0.55      0.00      0.00      0.02
Average:          0     99.32      0.00      0.66      0.00      0.00      0.02
Average:          1     99.54      0.00      0.45      0.00      0.00      0.02
Average:          2     99.69      0.00      0.29      0.00      0.00      0.02
Average:          3     99.20      0.00      0.78      0.00      0.00      0.02
```



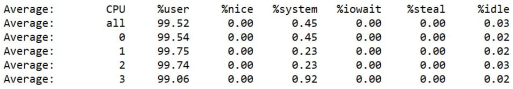
```
Average:        CPU     %user     %nice   %system   %iowait    %steal     %idle
Average:        all     25.80      0.00      0.97      0.00      0.00     73.23
Average:          0     25.73      0.00      0.91      0.00      0.00     73.36
Average:          1     25.92      0.00      1.20      0.00      0.00     72.87
Average:          2     25.85      0.00      0.77      0.00      0.00     73.38
Average:          3     25.70      0.00      0.99      0.00      0.00     73.31
```



```
Average:        CPU     %user     %nice   %system   %iowait    %steal     %idle
Average:        all     50.15      0.00      0.76      0.00      0.00     49.09
Average:          0     50.05      0.00      0.83      0.00      0.00     49.12
Average:          1     49.99      0.00      1.14      0.00      0.00     48.87
Average:          2     50.32      0.00      0.52      0.00      0.00     49.15
Average:          3     50.24      0.00      0.55      0.00      0.00     49.21
```



```
Average:        CPU     %user     %nice   %system   %iowait    %steal     %idle
Average:        all     74.78      0.00      0.62      0.00      0.00     24.60
Average:          0     74.83      0.00      0.54      0.00      0.00     24.63
Average:          1     74.87      0.00      0.55      0.00      0.00     24.57
Average:          2     74.58      0.00      0.86      0.00      0.00     24.55
Average:          3     74.81      0.00      0.54      0.00      0.00     24.65
```



```
Average:        CPU     %user     %nice   %system   %iowait    %steal     %idle
Average:        all     99.52      0.00      0.45      0.00      0.00      0.03
Average:          0     99.54      0.00      0.45      0.00      0.00      0.02
Average:          1     99.75      0.00      0.23      0.00      0.00      0.02
Average:          2     99.74      0.00      0.23      0.00      0.00      0.03
Average:          3     99.06      0.00      0.92      0.00      0.00      0.02
```

From the snippets, it can be seen that the average CPU utilization values are:
When no load is applied: 99.43%
When 25% load is applied: 25.80%
When 50% load is applied: 50.15%

When 75% load is applied: 74.78%
When 100% load is applied: 99.52%

The percentile values for each of them are:
When no load is applied:
90 percentile: 99.75%
95 percentile: 100.0%
99 percentile: 100.0%
When 25% load is applied:
90 percentile: 36.84%
95 percentile: 38.1%
99 percentile: 38.35%
When 50% load is applied:
90 percentile: 57.14%
95 percentile: 57.89%
99 percentile: 58.65%
When 75% load is applied:
90 percentile: 77.5%
95 percentile: 78.3%
99 percentile: 79.2%
When 100% load is applied:
90 percentile: 99.75%
95 percentile: 100.0%
99 percentile: 100.0%

- Do you observe any difference between the two options used to load the system? The key difference between the two options is the flexibility to control the load applied on the cores. ”–cpu” option does not provide this, whereas ”–cpu load” does.

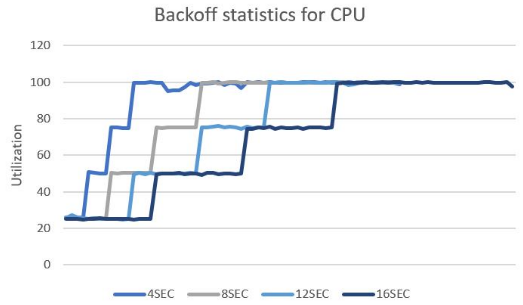
  When ”–cpu” option was used,the utilization of all the cpus were observed to be more than 98%, and the utilization was pretty consistent throughout the time for which the system was under test. However, the metric ”idle time” was 0 in almost all the cases, indicating that the cpu is running at its maximum capacity, implying that the demand for the cpu is high.
  However, when ”–cpu load” option was used, it was observed that the utilization of all the CPUs was around the specified load, with a fluctuation of +5% or -5%. However, the metric ”idle time” indicates that whenever there was a load assigned, the stressors took a considerable amount of time to decide how to balance the load between the cores and proceeded to implement it.
  Irrespective of the option used, the load was distributed evenly, putting equal stress on all the cores.

- When you use the –cpu-load option try to sample the system load metrics each 2 seconds, do you see any difference? What is the impact of the sampling interval on the results that you obtain?

  The only difference observed in the sampling interval is that the values are more closer to each other and the variation isn't as clear, to make any conclusions.

- Use the option –backoff N to progressively load the system from (about) 0% to 100% of the CPU utilization (N is in microseconds, 1sec = 1000000microseconds)

  - Try different values of N. Plot the time series and comment the results.

    From the graph, it is observed that the utilization gradually increases and reaches 100% towards the end of the observation. The progressive load on the system is observed by assigning load to one core at a time and adding another core everytime the backoff period is reached. This is done until all the cores are engaged and that is when the utilization is observed to reach its maximum.

4.1.2

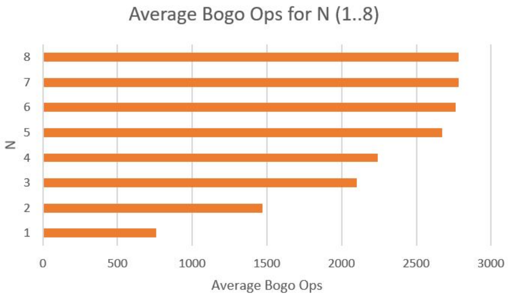
- Run stress-ng –crypt N –metrics-brief –perf -t 10s while increasing the value of N from 1 to 8 (step 1). Execute more than 1 run (at least 5 runs) for each test case to be sure that the number you obtain are reliable. Comment on the results.

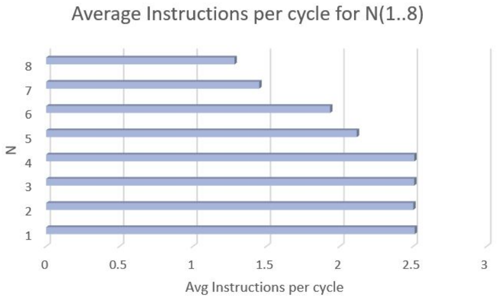
  - Does stress-ng statically assign workers to processors?

    No, stress-ng does not statically assign workers to processors. It does this dynamically.

- Comment about the scalability of the system. Use the bogo ops and the number of Instructions as performance counters for the comparison. Produce a plot to better explain your conclusions.

  The graphs are plotted considering the average values for 5 runs, for each value of N, ranging from 1 to 8. They are given as:

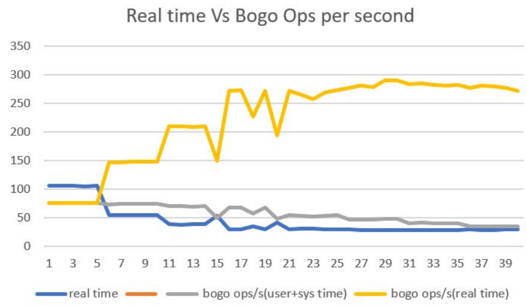
  

  We observed through the increase in the value of N (checking every N for 5 runs) that as the number of instructions increased, the system had decreased throughput, which was measured using the metric Bogo Ops. The decrease however, minimized as N value increased, but it was not proportional to number of instructions per CPU cycle. Hence, we conclude that the system is not scalable, in this case.

- Run stress-ng –crypt N –crypt-ops 8000 –metrics-brief –perf while increasing the value of N from 1 to 8 (step 1). Execute more than 1 run (at least 5 runs) for each test case to be sure that the number you obtain are reliable. What is/are the best metric(s) you can use to provide information on the system scalability? Is the stressor scalable? Produce a plot to better explain your conclusions.

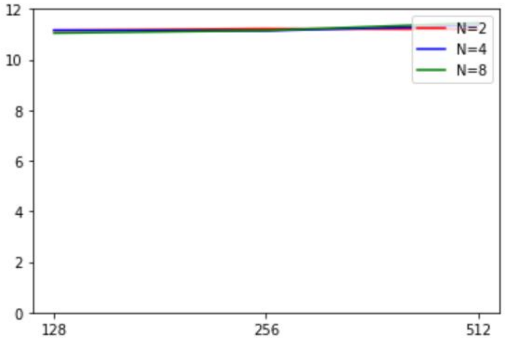
  The best metrics to provide information on the system's scalability will be : real time, bogo ops/s(real time) and bogo ops/s(user time and system time)
  The graph showcases the metrics as follows:



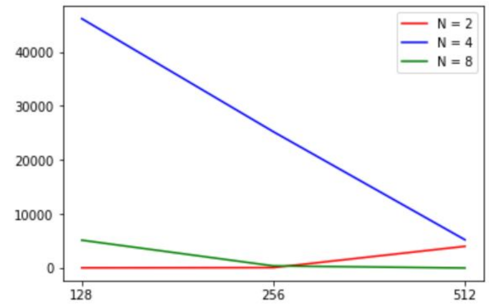
4.1.3

- Run stress-ng –matrix N –matrix-method prod –matrix-size M –metrics-brief –perf -t 10s while increasing the value of N (N=2, 4, 8) and, for each value of N, increasing M (M=128, 256, 512). Execute more than 1 run (at least 5 runs) for each test case to be sure that the number you obtain are reliable. Comment on the results.

  - What is the impact of the matrix size (M) on the memory consumption and on the cache performance (a cache miss means that the CPU accesses the memory to get the data rather than finding the data in the cache). Use plots to better explain your conclusions.

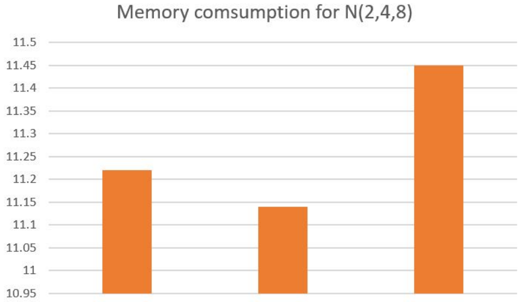
    From the above graph, we observe that even when the number of workers increased, the average memory consumption for the matrices of all sizes remains almost constant throughout, and the change is very slight, and almost negligible. On a whole, there is a slight increase as the size of the matrix increased.



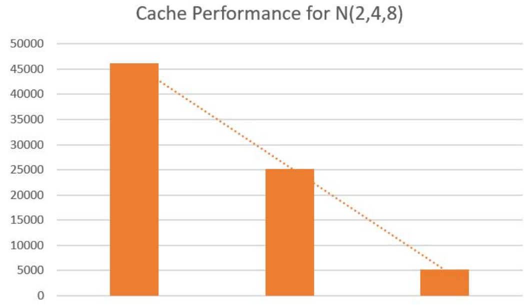
In the above graph, we observe that as number of workers increased, there was a drastic difference in each of the matrices cache performance. It increases when the number of workers are 2, and decreases when it is 4 and 8. On a whole, we observe that the cache performance decreases with increase in the number of workers in the system, and the rate of decrease is exponential as the number of workers increase gradually.

- Fix the matrix size. What is the impact of the number of threads on the memory consumption and on the cache performance. Use plots to better explain your conclusions.

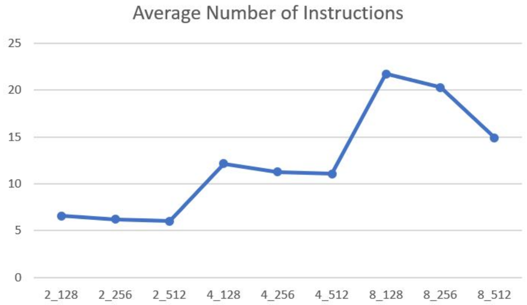
  As the number of threads increase, the average memory consumption seems to decrease at first and increase as number of workers increase. This is shown below:

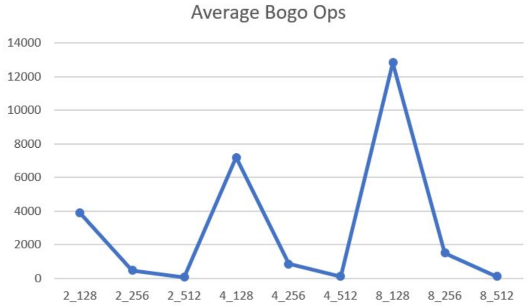
  

  However, a steady decrease in the cache performance is observed, with increase in thread number as well as number of workers, as demonstrated below.



- Is the stressor scalable with respect to the number of threads and matrix size? Use bogo ops and the number of instructions as performance counters for the comparison. Use plots to better explain your conclusions.

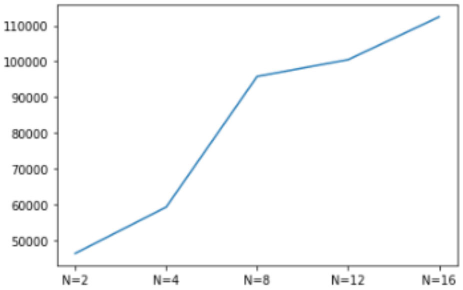




From the graphs, a steady rise and fall in the throughput is seen through Bogo Ops metric, where the peak is higher than before due to increase in the number of workers.
However, the highest scalability is observed when the matrix is of the size 128, and it remains consistent throughout the graph.
All of this is observed with a steady growth in the number of instructions used to load the system, which increases with workers.
Hence we can conclude that the system is scalable with respect to the above factors.

- Run stress-ng –pthread N –pthread-ops 1000000 –pthread-max 64 –perf

–metrics-brief with N ranging from 2 to 16 (2, 4, 8, 12, 16). Execute more than 1 run (at least 5 runs) for each test case to be sure that the numbers you obtain are reliable. Comment on the scalability and use plots to better explain your conclusions.

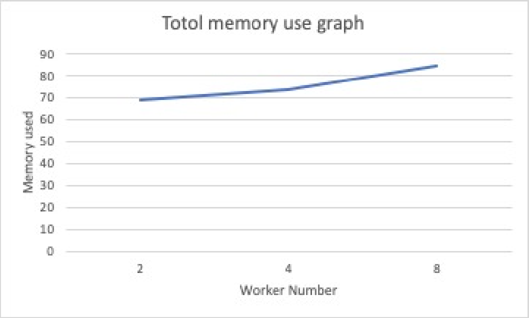


From above graph, it is observed that the system is scalable, as there is a throughput increase with increase in the number of threads(from 2 to 16).

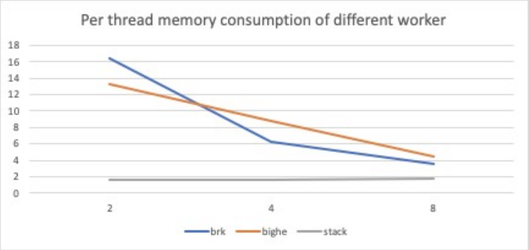
4.2.1

- Run stress-ng –brk N –stack N –bigheap N –metrics-brief –perf -t 60s with N ranging from 2 to 8 (2, 4, 8) and collect the global memory statistics with sar and the per thread memory statistics with top). Execute more than 1 run (at least 5 runs) for each test case to be sure that the number you get are reliable. Comment on the results you obtained using plots to better explain your conclusions.

  Total memory use: as the worker number increase, the total memory usage increases.

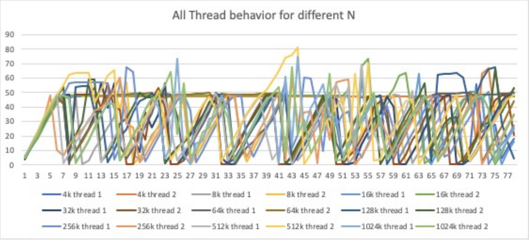
  

  For brk workers, when the worker number doubles, the memory use per thread will decrease by half or slightly more. For bigheap workers, when the worker number doubles, the memory use per thread will decrease by half or slightly less. For stack workers, when the worker number doubles, the memory use per thread will just keep the same.

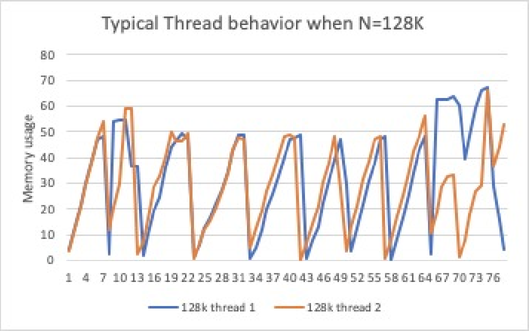


- Run stress-ng –bigheap 2 –bigheap-growth N –metrics-brief –perf -t 60s while increasing N from 4KB to 1MB (4K, 8K, 16K, 32K, 64K, 128K, 256K, 512K, 1024K). Comment on the behavior of the two threads and on the memory usage. Use plots to better explain your conclusions.

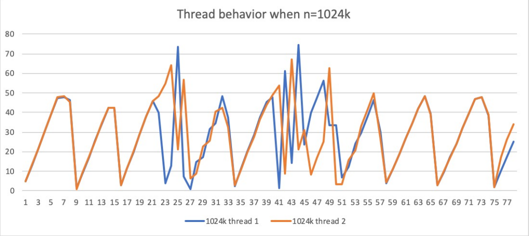
Behavior of 2 threads on memory consumption:
The bigheap stressor assign workers to grow their heaps by reallocating memory.
During the task running time, the 2 bigheap thread keeps reallocating memory and each eat up nearly half memory of the machine very rapidly within few seconds, and then they keep growing until get killed by the (out of memory)OOM. After getting killed, they eat up the memory again very rapidly as before and get killed again. That procedure loops until the set run time is over.



At the beginning of the run, the 2 thread usually get killed simultaneously, and they revived together and take half memory again until next kill. However, in later period of the run, the 2 thread is get killed asynchronously, The thread firstly killed revive with 0 memory use and goes up later, and the other thread will rise its memory usage beyond 50% to a higher usage until it get killed, and these procedure will loop until the run ends.



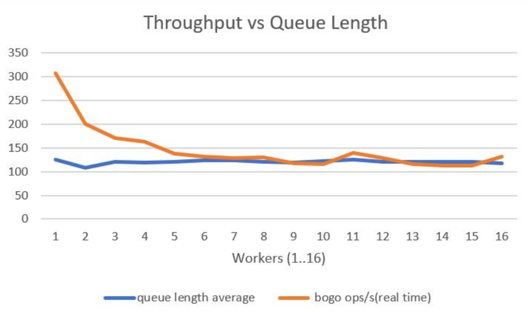
As Heap-growth unit gets bigger, the probability of the 2 thread to get asynchronously killed by OOM will go higher.



4.3.1

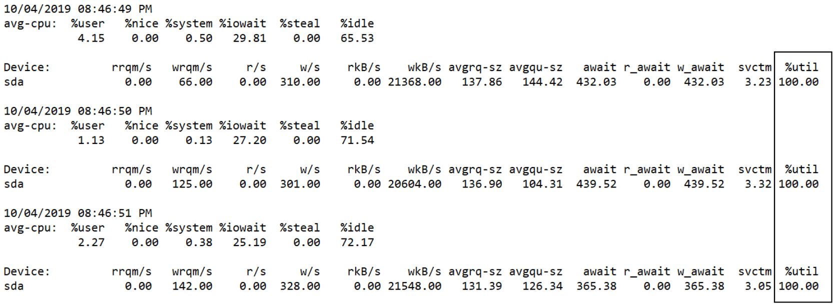
- Run stress-ng –hdd N –hdd-opts wr-rnd –metrics-brief -t 20s while increasing the value of N (from 1 to 16) to identify the maximum throughput that is achievable. Analyze the throughput and the queue length of the disk and comment on the scalability. Use plots to better explain your conclusions.

  A graph against throughput and queue length is plotted as:



It is observed that the queue length is almost constant from the beginning of the observation, whereas there is an exponential decrease in the

throughput of the system, as the assigned number of workers increase.
We suspect that the reason for this exponential decrease is a bottleneck that was created as the workers increased. This was observed through the % utilization metric which was always 100%. This is shown as:

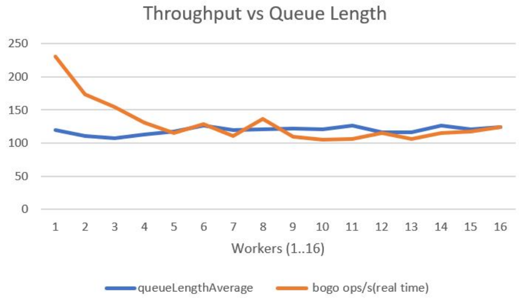


```
10/04/2019 08:46:49 PM
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           4.15    0.00    0.50   29.81    0.00   65.53

Device:         rrqm/s   wrqm/s     r/s     w/s    rkB/s    wkB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
sda               0.00    66.00    0.00  310.00     0.00 21368.00   137.86   144.42  432.03    0.00  432.03   3.23 100.00

10/04/2019 08:46:50 PM
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           1.13    0.00    0.13   27.20    0.00   71.54

Device:         rrqm/s   wrqm/s     r/s     w/s    rkB/s    wkB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
sda               0.00   125.00    0.00  301.00     0.00 20604.00   136.90   104.31  439.52    0.00  439.52   3.32 100.00

10/04/2019 08:46:51 PM
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           2.27    0.00    0.38   25.19    0.00   72.17

Device:         rrqm/s   wrqm/s     r/s     w/s    rkB/s    wkB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
sda               0.00   142.00    0.00  328.00     0.00 21548.00   131.39   126.34  365.38    0.00  365.38   3.05 100.00
```

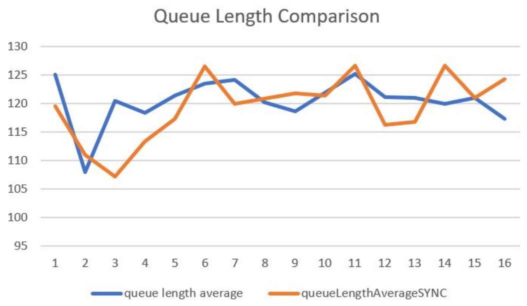
Due to these observations, we conclude that the performance of the disk is not scalable.

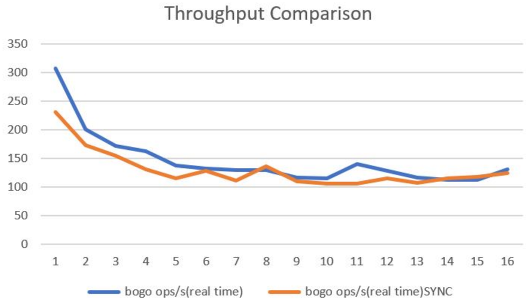
- Run stress-ng –hdd N –hdd-opts wr-rnd, sync –metrics-brief -t 20s while increasing the value of N (from 1 to 16). What is the difference with the previous test? Analyze the throughput and the queue length of the disk and comment on the scalability. Use plots to better explain your conclusions.

  The graph is plotted as:

  We stick to our previous explanation, in regards to scalability of the disk, as both the graphs show similar behaviour.
  However, if we were to compare the throughput and queue length separately, we found that, the throughput was higher in the first case whereas the queue length was higher in the latter.
  A graphical representation of this is:


Queue Length Comparison
130
125
120
115
110
105
100
95
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
queue length average
queueLengthAverageSYNC


Throughput Comparison
350
300
250
200
150
100
50
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
bogo ops/s(real time)
bogo ops/s(real time)SYNC