Design Report

- Cluster Details:
    - CPU, Memory and Network: Hyperion
    - Disk: Prometheus.

1. **CPU**:
    a. This benchmark is implemented using c programming language.
    b. File name: MyCPUBench.c
    c. I have used pthread, POSIX thread library available in c for multi-threading and achieving multiple concurrency levels.
    d. Use of structures to store thread information.
    e. Use of FILE for reading input files and writing output to a file.
    f. Strong scaling is achieved by dividing the problem size equally among different threads.
    g. main() method: This function is starting point of the program execution. It takes input as input file name and writes the output to a different file.
        i. There are two parameters taken as input from the file that is read, they are: workload type and concurrency.
        ii. After that as per the workload type, different methods are called to perform respective operations (QP, HP, SP or DP).
    h. **performQPOps**() : This method will execute Quarter Precision operations .This method gets called from the main method.. It generates number of threads (1,2,4) based on the input parameter received and starts executing the operations.
        i. pthread_create() will generate number of threads based on the concurrency level.
        ii. Start the timer before operations starts executing for each thread.
        iii. Execute 1 trillion operations.
        iv. End the timer after each thread completes their execution.
        v. Use of pthread_join() will ensure that all the threads have completed their execution before it moves forward in the program execution.
        vi. Once the operations are completed, calculate,

            totalTime = endTime – startTime,

            taken by all the threads.
        vii. Calculate GigaOps using **(no_operations / (total_time)) / 1e9.**
        viii. This will return number of Quarter Precision operations per seconds in GigaOps/Sec.
    i. **performHPOps** () : This method will execute Half Precision operations .This method gets called from the main method.. It generates number of threads (1,2,4) based on the input parameter received and starts executing the operations.
        i. pthread_create() will generate number of threads based on the concurrency level.

ii. Start the timer before operations starts executing for each thread.
iii. Execute 1 trillion operations.
iv. End the timer after each thread completes their execution.
v. Use of pthread_join() will ensure that all the threads have completed their execution before it moves forward in the program execution.
vi. Once the operations are completed, calculate,

totalTime = endTime – startTime,

taken by all the threads.
vii. Calculate GigaOps using **(no_operations / (total_time)) / 1e9.**
viii. This will return number of Half Precision operations per seconds in GigaOps/Sec.

j. **performSPOps**() : This method will execute Single Precision operations .This method gets called from the main method.. It generates number of threads (1,2,4) based on the input parameter received and starts executing the operations.
i. pthread_create() will generate number of threads based on the concurrency level.
ii. Start the timer before operations starts executing for each thread.
iii. Execute 1 trillion operations.
iv. End the timer after each thread completes their execution.
v. Use of pthread_join() will ensure that all the threads have completed their execution before it moves forward in the program execution.
vi. Once the operations are completed, calculate,

totalTime = endTime – startTime,

taken by all the threads.
vii. Calculate GigaOps using **(no_operations / (total_time)) / 1e9.**
viii. This will return number of Single Precision operations per seconds in GigaOps/Sec.

k. **performSPOps**() : This method will execute Single Precision operations .This method gets called from the main method.. It generates number of threads (1,2,4) based on the input parameter received and starts executing the operations.
i. pthread_create() will generate number of threads based on the concurrency level.
ii. Start the timer before operations starts executing for each thread.
iii. Execute 1 trillion operations.
iv. End the timer after each thread completes their execution.
v. Use of pthread_join() will ensure that all the threads have completed their execution before it moves forward in the program execution.
vi. Once the operations are completed, calculate,

totalTime = endTime – startTime,

taken by all the threads.
vii. Calculate GigaOps using **(no_operations / (total_time)) / 1e9.**
viii. This will return number of Single Precision operations per seconds in GigaOps/Sec.

l. **performDPOps**(): This method will execute Double Precision operations .This method gets called from the main method.. It generates number of threads (1,2,4) based on the input parameter received and starts executing the operations.
    i. pthread_create() will generate number of threads based on the concurrency level.
    ii. Start the timer before operations starts executing for each thread.
    iii. Execute 1 trillion operations.
    iv. End the timer after each thread completes their execution.
    v. Use of pthread_join() will ensure that all the threads have completed their execution before it moves forward in the program execution.
    vi. Once the operations are completed, calculate,
    totalTime = endTime – startTime,
    taken by all the threads.
    vii. Calculate GigaOps using **(no_operations / (total_time)) / 1e9.**
    viii. This will return number of Double Precision operations per seconds in GigaOps/Sec.

m. **writeToFile():**
    i. This method will write the output to an output file.

2. Memory:

a. This benchmark has been implemented in C language.
b. Filename: MyRAMBench.c
c. Use of POSIX Threads to achieve different concurrency levels.
d. Type of operations:
    i. RWS – Read-Write Sequential.
    ii. RWR – Read-Write Random
e. Various Block Sizes: 1B,1KB, 1MB and 10MB.
f. Strong scaling is used which divides the overall workload among the number of threads available for execution.
g. Use of structures to maintain thread information for calculating offset for different threads.
h. **Main**():
    i. This method is the starting point of the program and the execution of this benchmark begins here.
    ii. It reads input as a file and outputs the data to an output file.
    iii. There are 3 parameters that are read from the file:
        1. Workload type: RWS or RWR.
        2. Block Size: 1B(for latency), 1KB, 1MB or 10MB.
        3. Number of threads: 1,2 or 4.
    iv. Based on the wokload type, different methods are called to perform calculations.

i. **performRWR**()

   i. This method is called from main method for performing Random Read and Write operations.
   ii. Use of **pthread_create**() to create multiple threads.
   iii. Divide the 1GB data by BlockSize and num of threads and operate over it for 100x times to calculate the required throughput.
   iv. Use of **memcpy**(…) function in c to read + write data in random order.
   v. Randomizatoin is achieved by using
      randomOffset = rand()%blocksPerThread + (start);
   vi. Use of **pthread_join()** to ensure all the threads completed execution before the endTime is calculated.
   vii. Once the operations are completed, calculate,
      totalTime = endTime – startTime,
   viii. Calculate Throughput using:
      Throughput = Total Memory Read+Write / totalTime
   ix. This will give output in GBPS.
   x. Latency is calculated by operating over a block of 1B for 100 million times. Formula:
      Latency = 1 / throughput
   xi. The latency values are calculated in micro-seconds.

j. **performRWS** ()
   i. This method is called from main method for performing Sequential Read and Write operations.
   ii. Use of **pthread_create**() to create multiple threads.
   iii. Divide the 1GB data by BlockSize and num of threads and operate over it for 100x times to calculate the required throughput.
   iv. Use of **memcpy**(…) function in c to read + write data in sequential order.
   v. Use of **pthread_join()** to ensure all the threads completed execution before the endTime is calculated.
   vi. Once the operations are completed, calculate,
      totalTime = endTime – startTime,
   vii. Calculate Throughput using:
      Throughput = Total Memory Read+Write / totalTime
   viii. This will give output in **GBPS**.
   ix. Latency is calculated by operating over a block of 1B for 100 million times. Formula:
      Latency = 1 / throughput
   x. The latency values are calculated in **micro-seconds**.

3. Disk:

a. This benchmark has been implemented in C language.
b. Filename: MyDiskBench.c

c. Use of pthreads for achieving concurrency.
d. Strong scaling is achieved by dividing the workload to number of threads.
e. Type of Operations:
    i. RS: Read Sequential
    ii. WS: Write Sequential.
    iii. RR: Read Random.
    iv. WR: Write Random.
f. Various Block Sizes: 1KB,1MB, 10MB and 100MB.
g. Use of structures to maintain thread information for calculating offset for different threads.
h. Usage of flush to ensure data is read from disk and not cache.
i. **Also, since /tmp folder is local to compute nodes, I have performed read and write operations on the /tmp folder for each compute nodes.**
j. For calculating throughput, the experiments are conducted over 10GB of data.
k. For latency calculation, the experiment is ran on 1KB of data for 1 million times.
l. **Main**():
    i. This method is the starting point of the program and the execution of this benchmark begins here.
    ii. It reads input as a file and outputs the data to an output file.
    iii. There are 3 parameters that are read from the file:
        1. Workload type: RS, WS, RR or WR.
        2. Block Size: 1KB(for latency), 1MB, 10MB or 100MB.
        3. Number of threads: 1,2,4, 8, 16, 32, 64 and 128.
        4. Based on the wokload type, different methods are called to perform calculations
        5. Throughput is calculated using formula,
           Throughput = total file size Read/Write / Total Time
        6. The throughput value will be in **MBPS** and latency is in **millisecond**.
        7. Latency is calculated using,
           Latency = 1/throughput
    iv. **randomReadOperation** ():
        1. This method is for performing RR operation, which is Random Read operation.
        2. It uses various Block Sizes, number of threads to perform calculation.
        3. Pthread_create takes care of thread creation.
        4. Use of File read command (fread) to read from file.
        5. Use of fseek or lseek for pointing file pointers to appropriate position.
        6. Randomizatoin is achieved by using
           randomOffset = rand()%blocksPerThread + (start);

7. The calculations are performed over 10GB data.
8. The number of operations varies, based on blocksize and number of threads.
9. Pthread_join ensures that all threads have completed execution before total time calculation is done.
10. For throughput, block size varies from 1MB, 10MB and 100MB.
11. Number of threads varies from 1,2 and 4.
12. For latency, the block size is fixed to 1KB and the threads vary from 1 to 128.

v. **sequentialReadOperation** ():

1. This method is for performing RS operation, which is Sequential Read operation.
2. It uses various Block Sizes, number of threads to perform calculation.
3. Pthread_create takes care of thread creation.
4. Use of File read command (fread) to read from file.
5. Use of fseek or lseek for pointing file pointers to appropriate position.
6. Pthread_join ensures that all threads have completed execution before total time calculation is done.
7. For throughput, block size varies from 1MB, 10MB and 100MB.
8. The number of operations varies, based on blocksize and number of threads.
9. Number of threads varies from 1,2 and 4.
10. For latency, the block size is fixed to 1KB and the threads vary from 1 to 128.

vi. **randomWriteOperation** ():

1. This method is for performing WR operation, which is Random Write operation.
2. It uses various Block Sizes, number of threads to perform calculation.
3. Pthread_create takes care of thread creation.
4. Use of File write command (fwrite / write) to write data to file.
5. Use of fseek or lseek for pointing file pointers to appropriate position.
6. Randomizatoin is achieved by using
   randomOffset = rand()%blocksPerThread + (start);
7. Pthread_join ensures that all threads have completed execution before total time calculation is done.
8. The number of operations varies, based on blocksize and number of threads.
9. For throughput, block size varies from 1MB, 10MB and 100MB.

10. Number of threads varies from 1,2 and 4.
11. For latency, the block size is fixed to 1KB and the threads vary from 1 to 128.

vii. **sequentialWriteOperation** ():

1. This method is for performing WS operation, which is Sequentisl Write operation.
2. It uses various Block Sizes, number of threads to perform calculation.
3. Pthread_create takes care of thread creation.
4. Use of File write command (fwrite / write) to write data to file.
5. Use of fseek or lseek for pointing file pointers to appropriate position.
6. Pthread_join ensures that all threads have completed execution before total time calculation is done.
7. The number of operations varies, based on blocksize and number of threads.
8. For throughput, block size varies from 1MB, 10MB and 100MB.
9. Number of threads varies from 1,2 and 4.
10. For latency, the block size is fixed to 1KB and the threads vary from 1 to 128.

4. **Network**:

a. This benchmark has been implemented in JAVA language.
b. Filename: MyNETBenchTCP.java and MyNETBenchUDP.java.
c. Type of operations:
   i. TCP
   ii. UDP
d. Various Packet Sizes: 1B,1KB and 32KB.
e. Various concurrency levels ranging from 1,2,4 and 8.
f. **Instead of localhost, I have executed the program for TCP and UDP on two different nodes by using proper SBATCH parameters. Code for this can be found in slurm files.**
g. **Get the available nodes list and start server on any compute node. Write the server node name to a file on login node. When you start your client, fetch the server name from that file which is stored on login node to establish connection or to send datagrams.**
h. Use of Thread libraries of JAVA like Thread class and Runnable interface for achieving various concurrency levels.
i. Strong scaling is used which divides the overall workload among the number of threads available for execution.
j. **TCP:**
   i. Server and Client both are implemented in the same file.

- ii. When value=1 is passed in input parameter in initiates server and value=2 initiates client mode.
- iii. Use of **Thread t = new Thread(…)**; for achieving concurrency.
- iv. Use of SOCKET programming in java for creation of socket and data transfer.
- v. For server, Main() will call server method which will create server instance based on the number of threads.
- vi. For Client, main() will call client method which creates client instance as per the number of threads passed.
- vii. For multithreads, it is assumed that number of servers = number of clients.
- viii. Transfer of 1GB of data is done 100 times for calculating the throughput.
- ix. Transfer of 1B of data is done 1 million times to calculate latency of TCP network.
- x. Throughput is calculated in Mbps and is given by,

  Throughput = total data transferred / total time taken.

  Latency = 1 / throughput
- xi. The latency is calculated in milliseconds.

k. **<u>UDP</u>**:

- i. Server and Client both are implemented in the same file.
- ii. When value=1 is passed in input parameter in initiates server and value=2 initiates client mode.
- iii. Use of **Thread t = new Thread(…)**; for achieving concurrency.
- iv. Use of SOCKET programming in java for creation of UDP socket and datagram for transfer.
- v. For server, Main() will call server method which will create server instance based on the number of threads.
- vi. For Client, main() will call client method which creates client instance as per the number of threads passed.
- vii. For multithreads, it is assumed that number of servers = number of clients.
- viii. Transfer of 1GB of data is done 100 times for calculating the throughput.
- ix. Transfer of 1B of data is done 1 million times to calculate latency of TCP network.
- x. Throughput is calculated in Mbps and is given by,

  Throughput = total data transferred / total time taken.

  Latency = 1 / throughput
- xi. The latency is calculated in milliseconds.