CS 553
Cloud Computing
Programming Assignment 1
Design Document
Submitted By:
Sumedha Gupta (A20377983)
Aditya Jadhav (A20377887)

1. Introduction to the document

a. This document is designed to specify the overall designs of the programs that have been implemented to do the benchmarking for CPU, GPU, Memory, Disk and Network, the design tradeoffs that have been considered and the future improvements that can be done to enhance the efficiency.

b. Problem

This assignment required us to perform the benchmarking of CPU, GPU, Memory, Disk and Network on the Chameleon cloud.
For CPU benchmarking, we have used Open Stack KVM cloud
Instances used:

Flavor Details: m1.medium

ID	3
VCPUs	2
RAM	4GB
Size	40GB

In the CPU benchmarking, we have to calculate GFLOPS, GIOPS with varying level of concurrency i.e. 1 thread, 2 threads, 4 threads, 8 threads. In addition to this, the program has to be run for 10 minutes to take samples of no. of instructions per second for 8 threads. We have to do an efficiency comparison of our results with theoretical performance and Linpack benchmark.

In the GPU benchmarking, we have to calculate GLOPS, GIOPS, GQOPS, GHOPS, GPU speed with full concurrency and comparison of our results with the theoretical speed and Linpack benchmark.
For GPU Benchmarking, we have used Bare Metal
Instances used:

Flavor Details: baremetal

ID	baremetal
VCPUs	4
RAM	31.3GB
Size	100GB

In the memory benchmarking, there are different functions need to be performed such as read and write sequentially, write sequentially, write randomly with varying block sizes- 8B, 8KB, 8MB, 80MB and varying concurrency- 1 thread, 2 threads, 4 threads, 8 threads. Comparison of our results with Stream benchmark.

In the disk benchmarking, we have to measure the disk speed for different functions need to be performed such as read and write sequentially, read and write randomly, read sequentially, read randomly with varying block sizes- 8B, 8KB, 8MB, 80MB and varying concurrency- 1 thread, 2 threads, 4 threads, 8 threads. Comparison of our results with IOZone benchmark.

In the network benchmarking, we have to measure the network speed over the loopback interface card with 1 node and 2 processes with parameter space including TCP protocol stack, UDP, fixed packet/buffer size (64KB), and varying the concurrency- 1 thread, 2 threads, 4 threads, 8 threads. Comparison of our results with IPerf benchmark.

2. Overall Program Design

*****For all the experiments, strong scaling is used which ensures a fair comparison in case of multithreading.***

a. CPU Benchmarking

To perform the CPU benchmarking, a C program has been implemented to calculate GFLOPS and GIOPS. The program is designed in such a way that it performs 40 arithmetic operations with in a loop that runs for 1000000000 times for float and integer datatypes separately.

```
for(itr = 0; itr < numiter; itr++) {
  total = 5.66 + 7.888 + 4.67 + 8.99
        + 5.69 + 7.898 + 4.47 + 8.39
        + 5.66 + 7.888 + 4.67 + 8.99
        + 5.69 + 7.898 + 4.47 + 8.39
        + 5.66 + 7.888 + 4.67 + 8.99
        + 5.66 + 7.888 + 4.67 + 8.99
        + 5.69 + 7.898 + 4.47 + 8.39
        + 5.66 + 7.888 + 4.67 + 8.99
        + 5.69 + 7.898 + 4.47 + 8.39
        + 5.66 + 7.888 + 4.67 + 8.99;
}
pthread_exit(NULL);
}
```

```
for(itr=0; itr < numiter; itr++){
  total = 566 + 7888 + 467 + 899
        + 569 + 7898 + 447 + 839
        + 566 + 7888 + 467 + 899
        + 569 + 7898 + 447 + 839
        + 566 + 7888 + 467 + 899
        + 569 + 7898 + 447 + 839
        + 566 + 7888 + 467 + 899
        + 569 + 7898 + 447 + 839
        + 566 + 7888 + 467 + 899
        + 569 + 7898 + 447;
}
```

GFLOPS and GIOPS were calculated by measuring the time required to perform the computation. The code also includes multi-threading using pthread library in C to measure performance at varying level of concurrency- 1 thread, 2 threads, 4 threads, 8 threads.

Design Tradeoffs: The loops are designed in such a way that they will the operations as specified in each iteration and will not be simplified by the compiler.

Possible improvements and extensions: AVX instructions can be used to improve the performance.

b. GPU Benchmarking (Extra Credit)

To perform the GPU benchmarking, a CUDA C program has been implemented to calculate GFLOPS and GIOPS. The program is designed in such a way that it performs 40 arithmetic operations with in a loop that runs for 1000000000 times for double precision and single precision separately.
GFLOPS and GIOPS were calculated by measuring the time required to perform the computation.

```
for (i = 0; i < NUM_ITERATIONS; i++) {
    d_out[x] =  5.66 + 7.888 + 4.67 + 8.99
              + 5.69 + 7.898 + 4.47 + 8.39
              + 5.66 + 7.888 + 4.67 + 8.99
              + 5.69 + 7.898 + 4.47 + 8.39
              + 5.66 + 7.888 + 4.67 + 8.99
              + 5.66 + 7.888 + 4.67 + 8.99
              + 5.69 + 7.898 + 4.47 + 8.39
              + 5.66 + 7.888 + 4.67 + 8.99
              + 5.69 + 7.898 + 4.47 + 8.39
              + 5.66 + 7.888 + 4.67 + 8.99;
}
```

```
for (i = 0; i < NUM_ITERATIONS; i++) {
    d_out[x]  = 566 + 7888 + 467 + 899
              + 569 + 7898 + 447 + 839
              + 566 + 7888 + 467 + 899
              + 569 + 7898 + 447 + 839
              + 566 + 7888 + 467 + 899
              + 569 + 7898 + 447 + 839
              + 566 + 7888 + 467 + 899
              + 569 + 7898 + 447 + 839
              + 566 + 7888 + 467 + 899
              + 569 + 7898 + 447;
}
```

Design Tradeoffs and assumptions:

- We have called empty kernel function to initialize the CUDA environment to overcome the delay while working in parallel environment.
  (Reference: https://devtalk.nvidia.com/default/topic/392429/first-cudamalloc-takes-long-time-/ */)
- CUDA kernel launches are asynchronous so, we have used cudaDeviceSynchronize() after the CUDA kernel launch to facilitate synchronization and to get correct output.

c. Memory Benchmarking

In this benchmark, a C program with four separate functions to read and write sequentially, read and write randomly, write sequentially and write randomly from the memory have been implemented.
A memory size of 2 GB has been allocated. Each function performs the operations in the block sizes of 8B, 8KB, 8MB, 80MB with no. of threads varying from 1,2,4 and 8.
The metrics that have been measured are throughput(MB/sec) and latency(ms).
Below are functions that have been designed:

```
//Read - Write Sequentially
void *readWriteSeq(void *blockSize){
  long size = *(long *) blockSize;
  long int number_iter= ((long)memorySize/size)/numThreads;
  int index=0;
  long i=0;
  for (i=0; i<number_iter;i++){
    memcpy(&dest[index], &source[index],size);
    index= (index+size)%memorySize; //reading sequentially
  }
  pthread_exit(NULL);
}

/*Read - Write Randomly*/
void *readWriteRand(void *blockSize){
  long size =*(long *) blockSize;
  long int number_iter= ((long)memorySize/size)/numThreads;
  int index=0;
  long i=0;
  for (i=0; i<number_iter;i++){
    index =  rand()% (memorySize- (size+1)); //pointing to a random loaction
    memcpy(&dest[index], &source[index],size);
  }
  pthread_exit(NULL);
}
```

```
/*Write Randomly*/
void *writeRand(void *blockSize){
  long size = *(long *)blockSize;
  long int number_iter= ((long)memorySize/size)/numThreads;
  int index=0;
  long i=0;
  for (i=0; i<number_iter;i++){
    index =  rand()% (memorySize- (size+1)); //pointing to a random loaction
    memset(&dest[index],'a',size);
  }
  pthread_exit(NULL);
}

/*Write Sequentially*/
void *writeSeq(void *blockSize){
  long size = *(long *)blockSize;
  long int number_iter= ((long)memorySize/size)/numThreads;
  int index=0;
  long i=0;
  for (i=0; i<number_iter;i++){
   memset(&dest[index],'a',size);
    index= (index+size)%memorySize; //writing sequentially
  }
  pthread_exit(NULL);
}
```

- **Design Tradeoffs and Assumptions:** The memory size is fixed and the level of varying concurrency i.e. number of threads and the block size is taken as a command line argument from user.

- **Possible improvements and extensions:** To get more accurate results, the effect of spatial locality can be disregarded.

d. Disk Benchmarking

In this benchmark, a C program with four separate functions to read and write sequentially, read and write randomly, write sequentially and write randomly from a file on disk have been implemented.
A file size of 10 GB has been allocated. Each function performs the operations in the block sizes of 8B, 8KB, 8MB, 80MB with no. of threads varying from 1,2,4 and 8.
The metrics that have been measured are throughput(MB/sec) and latency(ms).
Below are functions that have been designed. ***fread and fwrite have been used as these provided the best performance.***

```c
//Read - Write Sequentially
void *readWriteSeq(void *bufferSize) {
    long int numIter = ((long)FILE_SIZE / blockSize) / numThreads;
    char *buf = malloc(blockSize * sizeof(char));
    long i;
    FILE *fileRead = fopen (fileName, "r"); //open file in read mode
    for (i = 0; i < numIter; i++) {
        fread(buf, blockSize, 1, fileRead);
    }
    free(buf); //free buffer
    fclose(fileRead); //close the file
}

/*Read - Write Randomly*/
void *readWriteRand(void *bufferSize) {
    long int numIter = ((long)FILE_SIZE / blockSize) / numThreads;
    char *buf = malloc(blockSize * sizeof(char));
    long i;
    FILE *fileRead = fopen (fileName, "r"); //open file in read mode
    for (i = 0; i < numIter; i++) {
        //pointing to a ramdom location
        fseek(fileRead, (rand() % sizeof(buf)), SEEK_SET);
        fread(buf, blockSize, 1, fileRead);
    }
    free(buf); //free buffer
    fclose(fileRead); //close the file
}
```

```c
void *writeSeq(void *bufferSize) {
    long int numIter = ((long)FILE_SIZE / blockSize) / numThreads;
    char *buf = malloc(blockSize * sizeof(char));
    long i;
    FILE *fileWrite = fopen (fileName, "a");//open file in append mode
    for (i = 0; i < numIter; i++) {
        fwrite(buf, blockSize, 1, fileWrite);
    }
    free(buf);
    fclose(fileWrite);
}

void *writeRand(void *bufferSize) {
    long int numIter = ((long)FILE_SIZE / blockSize) / numThreads;
    char *buf = malloc(blockSize * sizeof(char));
    long i;
    FILE *fileWrite = fopen (fileName, "a");//open file in append mode
    for (i = 0; i < numIter; i++) {
        //pointing to a ramdom location
        fseek(fileWrite, (rand() % sizeof(buf)), SEEK_SET);
        fwrite(buf, blockSize, 1, fileWrite);
    }
    free(buf);
    fclose(fileWrite);
}
```

- **Design Tradeoffs and Assumptions:** The file size is fixed and the level of varying concurrency i.e. number of threads and the block size is taken as a command line argument from user.

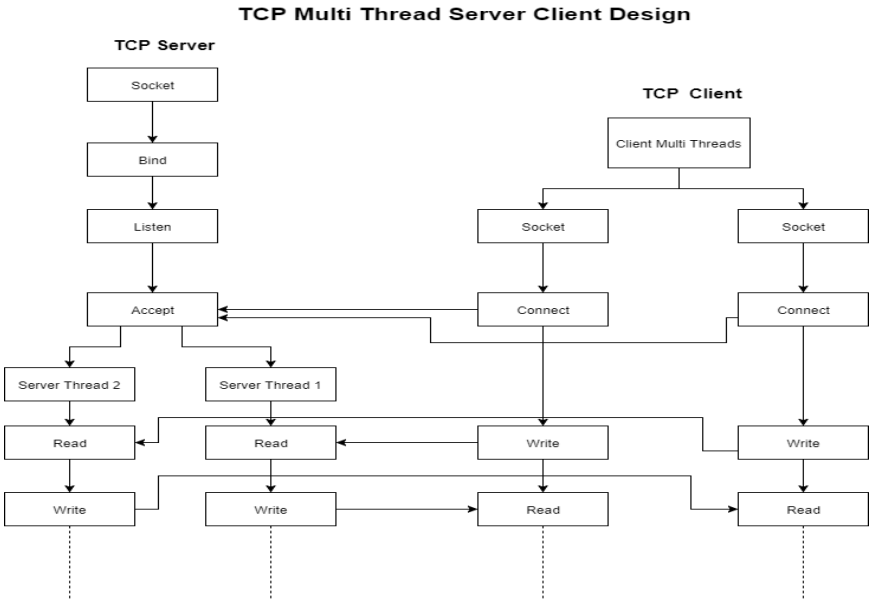
- **Possible improvements and extensions:**
  - The block size is limited to 80MB, we can increase the block size to get even more better performance.
  - Also, there might be some bad sectors that are inaccessible or unwriteable due to permanent damage to the disk, that can be considered to get more accurate performance.

e. Network Benchmarking

In this benchmark, network speed over the loopback interface card with 1 node and 2 processes on the same node is measured with a fixed packet size of 64KB with no. of threads varying from 1,2,4 and 8. This has been tested for both TCP protocol stack and UDP. Also, multithreading is implemented on both server and client side. When a connection is established between server and client, server creates a thread to handle client request. The metrics that have been measured are throughput(Mb/sec) and latency(ms).

Extra Credit: The above experiment has been done for 2 nodes as well.

The below diagram shows the overview of my program for TCP protocol with multithreading on both server and client.



- Similar design has been implemented for UDP as well.
- Design Tradeoffs and Assumptions:
    - Number of threads at both and client and server are assumed to be same.
    - The packet size of 64KB is also assumed to be fixed.
- Possible improvements and extensions: We can vary the packet size to get even more better performance.