Rushit Pandya A20381916
Achal Lathia A20375208

Design Document

I. CPU Benchmark:

→**Libraries Used:** We have used pthreads library available in C programming language for creating multiple threads and achieving different concurrency levels.

→**Functions:**

i. main: This function is a starting point of CPU benchmark. As we want to test benchmarks at different concurrency levels, we are calling perform_IOPS and perform_FLOPS functions for 1,2,4,8 numbers of threads.

ii. perform_IOPS: When this function is called and executed by main function, it creates multiple threads by executing pthread_create() method and waits for results of each thread. As soon as pthread_join() method is executed by all the threads, it calculates total time taken during execution of the routine of each threads. Each thread is calling intoperations() routines to perform integer precision operations. Also, processor speed is calculated by **(no_operations / (total_time)) / 1e9** formula, which returns number of integer operations per seconds in GigaFlops/seconds.

iii. perform_FLOPS: This function is called and executed by main function. It creates multiple threads by executing pthread_create() method and waits for results of each thread. As soon as pthread_join() method is executed by all the threads, it calculates total time taken during execution of the routine of each threads. Each thread is calling fpoperations() routines to perform double precision operations. Also, processor speed is calculated by **(no_operations / (total_time)) / 1e9** formula, which returns no of double operation per seconds in GigaFlops/seconds.

iv. intoperations(): When this function is executed by each thread, it computes sum of the loop for 9,00,000,000 iterations. But as each iteration consists of 2 integer operations (addition and assignment instruction), it performs total 1,800,000,000 integer instructions. But to achieve **strong scaling**, no of instructions are divided among the threads. So, each thread performs **no_of_instructions/no_threads** integer instructions.

v. fpoperations(): When this function is executed by each thread, it computes sum of the loop for 9,00,000,000 iterations. But as each iteration consists of 2 double operations (addition and assignment instruction), it performs total 1,800,000,000 double instructions. But to achieve **strong scaling**, no of instructions are divided among the threads. So, each thread performs **no_of_instructions/no_threads** integer instructions.

→**AVX Instructions:** For running our CPU Benchmark for AVX instructions, we have implemented a separate file i.e. cpu_avx.c. This file has same functions as above but only intoperations() and fpoperations() function computes AVX integer instructions and AVX double instructions. We have tested it on baremetal and put our results in performance pdf for avx instructions.

II. Memory Benchmark:

→**Libraries Used:** We have used pthreads library available in c programming language for creating multiple threads and achieving different concurrency levels for different block_sizes.

→**Functions:**

i. main: This function is a starting point of Memory Benchmark. As we want to test benchmarks at different concurrency levels and for varying block sizes, we are calling perform_random_write(), perform_seq_write() and perform_seq() function for 1,2,4,8 numbers of threads and 8B,8KB,8MB,80MB blocks sizes.
ii. perform_random_write(): When this function is called and executed by main function, it creates multiple threads by executing pthread_create() method and waits for results of each thread. As soon as pthread_join() method is executed by all the threads, it calculates total time taken during execution of the routine of each threads. Each thread is calling random_write () routines to perform operations for writing into memory randomly. Also, throughput is calculated by **total_memory-write/total_time** formula, which gives output in MBPS. Inverse of throughput calculates latency.
iii. perform_seq_write():When this function is called and executed by main function, it creates multiple threads by executing pthread_create() method and waits for results of each thread. As soon as pthread_join() method is executed by all the threads, it calculates total time taken during execution of the routine of each threads. Each thread is calling seq_write () routines to perform operations for writing into memory sequentially. Also, throughput is calculated by **total_memory-write/total_time** formula, which gives output in MBPS. Inverse of throughput calculates latency.
iv. perform_seq():When this function is called and executed by main function, it creates multiple threads by executing pthread_create() method and waits for results of each thread. As soon as pthread_join() method is executed by all the threads, it calculates total time taken during execution of the routine of each threads. Each thread is calling seq_access() routines to perform operations for reading and writing into memory. Also, throughput is calculated by **total_memory-read-write/total_time** formula, which gives output in MBPS. Inverse of throughput calculates latency.
v. seq_write(): When this function is executed by each thread, it performs memset operation to write into memory sequentially in block_size fetched. Each thread structure is created which contains the starting, ending blocks to write the memory. This is required to achieve **strong scaling**.
vi. random_write(): When this function is executed by each thread, it performs memset operation to write into memory randomly in block_size fetched. Each thread structure is

created which contains the starting, ending blocks to write the memory. This is required to achieve **strong scaling**.

vii. seq_access(): When this function is executed by each thread, it performs memcpy operation to read and write into memory sequentially in block_size fetched. Each thread structure is created which contains the starting, ending blocks to write the memory. This is required to achieve strong **scaling**.

III. Disk Benchmark:

→**Libraries Used:** We have used pthreads library available in c programming language for creating multiple threads and achieving different concurrency levels for different block_sizes.

→**Functions:**

i. main: This function is a starting point of memory benchmark. As we want to test benchmarks at different concurrency levels and for varying block sizes, we are calling perform_random_read(), perform_seq_read() and perform_read_write() function for 1,2,4,8 numbers of threads and 8B,8KB,8MB,80MB blocks sizes.

ii. perform_random_read(): When this function is called and executed by main function, it creates multiple threads by executing pthread_create() method and waits for results of each thread. As soon as pthread_join() method is executed by all the threads, it calculates total time taken during execution of the routine of each threads. Each thread is calling random_read () routines to perform operations for reading from file randomly. Also, throughput is calculated by **total_size_file_read /total_time** formula, which gives output in MBPS. Latency is calculated by inverse of throughput.

iii. perform_seq_read():When this function is called and executed by main function, it creates multiple threads by executing pthread_create() method and waits for results of each thread. As soon as pthread_join() method is executed by all the threads, it calculates total time taken during execution of the routine of each threads. Each thread is calling seq_read () routines to perform operations for read from file sequentially. Also, throughput is calculated by **total_size_file_read/total_time** formula, which gives output in MBPS. Latency is calculated by inverse of throughput.

iv. perform_seq():When this function is called and executed by main function, it creates multiple threads by executing pthread_create() method and waits for results of each thread. As soon as pthread_join() method is executed by all the threads, it calculates total time taken during execution of the routine of each threads. Each thread is calling seq_access() routines to perform operations for reading and writing to file. Also, throughput is calculated by **total_size_file_read-write/total_time** formula, which gives output in MBPS. Latency is calculated by inverse of throughput.

v. seq_read(): When this function is executed by each thread, it opens the file and execute read() to read from file sequentially in block_size fetched. Each thread structure is created which contains the starting, ending blocks to read from file. This is required to achieve **strong scaling**.

vi. random_read(): When this function is executed by each thread, it opens the file and execute read() to read from file randomly in block_size fetched. Each thread structure is created which contains the starting, ending blocks to read from file. This is required to achieve **strong scaling**.

vii. create_testing_file(): This function creates a testing file of size 1GB.

viii. read_write(): When this function is executed by each thread, it opens the file and execute read() to read from file sequentially and then write() to write to the file sequentially in block_size fetched. Each thread structure is created which contains the starting, ending blocks to read and write to the file. This is required to achieve **strong scaling**.

IV. Network Benchmark:

→**Libraries Used:** We have used Thread Class library available in java programming language for creating multiple threads and achieving different concurrency levels for TCP and UDP Connection.

1. TCP Network Benchmark:

→To test this benchmark, network connection has been established between client and server using Sockets class in java. First of all, TcpServer is started at 9000 port. Any connection request made by client to localhost address and port 9000, is accepted by the TcpServer. All the clients requests are made by TcpClient class. Once main method for TcpClient is executed, Socket connection is established with server and it will create different threads to achieve concurreny levels of 1,2,4,8 threads. Each thread will make instance of ExecuteTcpClient class. Once run method is called by thread of that instance, it will create a stream of 64KB packet size and will start sending the data at the server end port. As total 1GB of data has to be send at the server side, client will transfer the data into 64KB chunks. Note that, **Strong Scaling** is achieved here which means total amount of data to be sent is divided among the threads. Total time is calculated by amount of time taken to send whole 1GB of data to the server. Also, throughput is calculated by total amount 1GB of data sent/total time taken to send 1GB of data. Latency is calculated by total time taken to send a chunk of data at the server side. Units of calculations for Throughput and Latency are MegaBytes per second and microseconds respectively.

Classes:

→Client Files: TcpClient.java

→Server File: TcpServer.java

2. UDP Network Benchmark:

→To test this benchmark, network connection has been established between client and server using Datagram class in java. First of all, UdpServer is started at 9000 port. Any connection request made by client to localhost address and port 9000, is accepted by the UdpServer. All the clients requests are made by UdpClient class. Once main method for UdpClient is executed, Datagram socket is established with server and it will create different threads to achieve concurreny levels of 1,2,4,8 threads. Each thread will make instance of ExecuteUdpClient class. Once run method is called by thread of that instance, it will create a Datagram Packet of 63KB packet size and will start sending the data at the server end port. As total 1GB of data has to be send at the server side, client will transfer the data into 63KB chunks. At the UdpServer side, once connection is established by the UdpClient it will start to receive Datagram packet from the UdpClient. Note that, **Strong Scaling** is achieved here which means total amount of data to be sent is divided among the threads. Total time is calculated by amount of time taken to send whole 1GB of data to the server. Also, throughput is calculated by total amount 1GB of data sent/total time taken to send 1GB of data. Latency is calculated by total time taken to send a chunk of data at the server side. Units of calculations for Throughput and Latency are MegaBytes per second and microseconds respectively.

Classes:

→Client Files: UdpClient.java

→Server File: UdpServer.java