CS 553 CLOUD COMPUTING

Programming Assignment -1

RONAKKUMAR MAKADIYA (CWID: A20332994)
KAUSTUBH BOJEWAR (CWID: A20329244)
SOURABH CHOUGALE (CWID: A20326997)

Source Code

****

Contribution:

CPU Benchmarking: (RONAKKUMAR MAKADIYA)

GPU Benchmarking: (RONAKKUMAR MAKADIYA, KAUSTUBH BOJEWAR, SOURABH CHOUGALE)

Memory Benchmarking: (RONAKKUMAR MAKADIYA, KAUSTUBH BOJEWAR, SOURABH CHOUGALE).

Disk Benchmarking: (KAUSTUBH BOJEWAR)

Network Benchmarking: (SOURABH CHOUGALE)

1. CPU Benchmarking:

```
// main.cpp

// CPU Benchmarking

// Created by Ronakkumar Makadiya (CWID - A20332994)

// Created on 09/09/2104

// Copyright (c) 2014 Ronakkumar Makadiya. All rights reserved

#include <pthread.h>

#include <stdio.h>

#include <time.h>                    //Including header files

#include <limits.h>

struct arg_struct {                  // defining structure

int thread_id;

int choice;

};

double gflops_avg=0.0;

double flops_avg=0.0;

double giops_avg=0.0;

double iops_avg=0.0;

void *calculateBenchMark(void *arguments)    // Calculating  Benchmarrking

{

clock_t t1,t2;

struct arg_struct *args = (struct arg_struct *)arguments;

int i=0;

double emptyloop,floatingloop=0,flops,gflops;

double temp_float_value;

t1=clock();
```

```
for(i=0;i<INT_MAX;i++)                                    // calcuating  Empty loop timing
{
}
t1 = clock() - t1;
emptyloop =((double)t1)/CLOCKS_PER_SEC;

/* ...........For Flops ..........................*/
if(args -> choice == 0)
{
t1=clock();
// calculating floatig point operation time
for(i=0;i<INT_MAX;i++)
{
temp_float_value +=0.5;
}
t1 = clock() - t1;
floatingloop =((double)t1)/CLOCKS_PER_SEC;
flops=INT_MAX/((floatingloop-emptyloop));
gflops=(flops/1000000000);
gflops_avg += gflops;
flops_avg += flops;
}
//----------------For IOPS .......................*/
if(args -> choice == 1)
{
t2=clock();
int temp_int_value;
for(i=0;i<INT_MAX;i++)
{
temp_int_value +=1;                                    // calculating interger operation time
}
t2 = clock() - t2;
double intloop =((double)t2)/CLOCKS_PER_SEC;
double iops=INT_MAX/((intloop-emptyloop));
double giops=(iops/1000000000);
giops_avg += giops;
iops_avg += iops;
}
}

int select_num_thread(int no_of_threads)
{
printf("\n Enter the number of threads:");                     // Accepting number of threads from user
scanf("%d",&no_of_threads);
return no_of_threads;

}

void operation(int no_of_threads,int choice)
{
volatile int i=0;
pthread_t threads[no_of_threads];
```

```
struct arg_struct args;
int result;
for(i = 0; i < no_of_threads; i++)
{
args.thread_id = i;
if(choice==0) {
args.choice = 0;
} else
{
args.choice=1;
}
result = pthread_create(&threads[i], NULL,calculateBenchMark,(void *)&args);    // Creating Thr
if (result)
{
printf("Error:unable to create thread %d",result);

}
}

for(i=0;i<no_of_threads;i++)
{
pthread_join(threads[i],NULL);
}

if(choice==0)
{
```

```
printf("\nGFLOPS:%f \n",gflops_avg/no_of_threads);

printf("\nFLOPS:%f \n",flops_avg/no_of_threads);

}


if(choice==1)

{


printf("\nGIOPS:%f \n",(-(giops_avg/no_of_threads)));

printf("\nIOPS:%f \n",(-(iops_avg/no_of_threads)));

}

pthread_exit(NULL);

}

int main ()

{

int no_of_threads=0;

int choice;

do {

printf("..Menu...\n");

printf("1.FLOPS\n");

printf("2.IOPS\n");

printf("3.EXIT\n");

printf("\nEnter your choice :");

scanf("%d",&choice);

switch(choice)

{
```

```
case 1: printf("You have selected FLOPS\n");
no_of_threads=select_num_thread(no_of_threads);
operation(no_of_threads,0);
break;
case 2: printf("You have selected IOPS \n");
no_of_threads=select_num_thread(no_of_threads);
operation(no_of_threads,1);
break;

case 3:
break;
}
}
while(choice<3);
return 0;
}
```

2. **GPU Benchmarking:**

i. **flops.cu**

```
// GPU Benchmarking FLOPS
// flops.cu
// Created by Ronakkumar Makadiya (CWID - A20332994) Kaustubh Bojewar(CWID: A20329244) Sourabh Chougale(CWID: A20326997)
// Created on 09/09/2104
// Copyright (c) 2014 Ronakkumar Makadiya Kaustubh Bojewar Sourabh Chougale. All rights reserved.


#include <sys/time.h>
#include <iostream>
#include <cuda.h>
#include <ctime>
using namespace std;
#define SIZE 10000000



__global__ void emptyLoopTime(int n)
{

long int i=0;
int a=0;
for(i=0;i<n;i++)
{
}


}


__global__ void flopsCUDA(long int* total,int n)
{
//clock_t t1,t2,total_time=0;
long int i=0;
int a=0;

for(i=0;i<n;i++)
{
a=a+0.5;
}

}



void calculateFlops()
{

long int total=0;
long int *d_total;
double time=0;
double *d_time;
```

```
//cudaError_t cudaStatus;
cudaMalloc(&d_total, sizeof(long int));
cudaMalloc(&d_time, sizeof(double));

cudaMemcpy(d_total,&total, sizeof(long int),cudaMemcpyHostToDevice);
cudaMemcpy(d_time,&time, sizeof(double),cudaMemcpyHostToDevice);
//---------------------------------------------------------------------
cudaEvent_t empty_start, empty_stop;
cudaEventCreate(&empty_start);
cudaEventCreate(&empty_stop);

// Start record

cudaEventRecord(empty_start, 0);

emptyLoopTime<<<1,1>>>(SIZE);

cudaEventRecord(empty_stop, 0);

cudaEventSynchronize(empty_start); //optional
cudaEventSynchronize(empty_stop);

float emptyloop;
cudaEventElapsedTime(&emptyloop,empty_start,empty_stop);
```

```
cudaEventDestroy(empty_start);

cudaEventDestroy(empty_stop);

//cout << "Empty Loop time:"<<emptyloop<<endl;


//----------------------------------------------------------------

cudaEvent_t start,stop;

cudaEventCreate(&start);

cudaEventCreate(&stop);

//        cudaStatus = cudaDeviceSynchronize();




cudaEventRecord(start, 0);

flopsCUDA<<<1,1>>>(d_total,SIZE);

cudaEventRecord(stop, 0);


cudaEventSynchronize(start); //optional

cudaEventSynchronize(stop);




float elapsedTime;

cudaEventElapsedTime(&elapsedTime, start, stop); // that's our time! Clean up:


cudaEventDestroy(start);

cudaEventDestroy(stop);
```

```
//------------------------------------------------------------------------

//cout << "Time elapsed:"<<elapsedTime<<endl;


cudaMemcpy(&total,d_total,sizeof(long int),cudaMemcpyDeviceToHost);

cudaMemcpy(&time,d_time,sizeof(double),cudaMemcpyDeviceToHost);


long double flops=SIZE/(emptyloop-elapsedTime);


cout<<"\nFLOPS:"<<flops<<endl;


double gflops=flops/1000000000;


cout<< "\n The GFLOPS:"<<gflops;

//cout<<"\nThe answer is "<<total<<endl;

//cout<<"The answer is "<<b<<endl;


cudaFree(d_total);

cudaFree(d_time);




}




int main(){
```

```
cout <<"\n\n---------------------GFLOPS CUDA Benchmarking---------------------------------------
\n\n";

calculateFlops();



//cout <<"\n\n---------------------GIOPS CUDA Benchmarking---------------------------------------
\n\n";

//calculateIops();




return 0;

}
```

```
}
```

ii. <u>iops.cu</u>

```
// GPU Benchmarking IOPS

// iops.cu

// Created by Ronakkumar Makadiya (CWID - A20332994) Kaustubh Bojewar(CWID: A20329244) Sourabh Chougale(CWID: A20326997)

// Created on 09/09/2104

// Copyright (c) 2014 Ronakkumar Makadiya Kaustubh Bojewar Sourabh Chougale. All rights reserved.



#include <sys/time.h>

#include <iostream>

#include <cuda.h>

#include <ctime>

using namespace std;

#define SIZE 10000000



__global__ void iopsCUDA(long int* total,int n)

{

//clock_t t1,t2,total_time=0;

long int i=0;

int a=0;

for(i=0;i<n;i++)

{

a=a+i;

}
}

__global__ void emptyLoopTime(int n)
{

long int i=0;
int a=0;
for(i=0;i<n;i++)
{
}

}

__global__ void flopsCUDA(long int* total,int n)
{
//clock_t t1,t2,total_time=0;
long int i=0;
int a=0;

for(i=0;i<n;i++)
{
a=a+0.5;
}

}
```

```
void calculateIops()
{

long int total=0;
long int *d_total;
double time=0;
double *d_time;

//cudaError_t cudaStatus;
cudaMalloc(&d_total, sizeof(long int));
cudaMalloc(&d_time, sizeof(double));

cudaMemcpy(d_total,&total, sizeof(long int),cudaMemcpyHostToDevice);
cudaMemcpy(d_time,&time, sizeof(double),cudaMemcpyHostToDevice);
//------------------------------------------------------------------------
cudaEvent_t empty_start, empty_stop;
cudaEventCreate(&empty_start);
cudaEventCreate(&empty_stop);

// Start record

cudaEventRecord(empty_start, 0);
```

```
emptyLoopTime<<<1,1>>>(SIZE);

cudaEventRecord(empty_stop, 0);

cudaEventSynchronize(empty_start); //optional
cudaEventSynchronize(empty_stop);

float emptyloop;
cudaEventElapsedTime(&emptyloop,empty_start,empty_stop);

cudaEventDestroy(empty_start);
cudaEventDestroy(empty_stop);
//cout << "Empty Loop time:"<<emptyloop<<endl;

//---------------------------------------------------------------
cudaEvent_t start,stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
//        cudaStatus = cudaDeviceSynchronize();



cudaEventRecord(start, 0);
iopsCUDA<<<1,1>>>(d_total,SIZE);
cudaEventRecord(stop, 0);
```

```
cudaEventSynchronize(start); //optional

cudaEventSynchronize(stop);



float elapsedTime;

cudaEventElapsedTime(&elapsedTime, start, stop);


cudaEventDestroy(start);

cudaEventDestroy(stop);


//----------------------------------------------------------------------

//cout << "Time elapsed:"<<elapsedTime<<endl;


cudaMemcpy(&total,d_total,sizeof(long int),cudaMemcpyDeviceToHost);

cudaMemcpy(&time,d_time,sizeof(double),cudaMemcpyDeviceToHost);


long double iops=SIZE/(emptyloop-elapsedTime);


cout<<"\n IOPS:"<<iops<<endl;


double giops=iops/1000000000;


cout<< "\n The GIOPS:"<<giops;

//cout<<"\nThe answer is "<<total<<endl;

//cout<<"The answer is "<<b<<endl;
```

```
cudaFree(d_total);

cudaFree(d_time);



}


void calculateFlops()

{


long int total=0;

long int *d_total;

double time=0;

double *d_time;


//cudaError_t cudaStatus;

cudaMalloc(&d_total, sizeof(long int));

cudaMalloc(&d_time, sizeof(double));


cudaMemcpy(d_total,&total, sizeof(long int),cudaMemcpyHostToDevice);

cudaMemcpy(d_time,&time, sizeof(double),cudaMemcpyHostToDevice);

//------------------------------------------------------------------------

cudaEvent_t empty_start, empty_stop;

cudaEventCreate(&empty_start);

cudaEventCreate(&empty_stop);
```

```
// Start record

cudaEventRecord(empty_start, 0);

emptyLoopTime<<<1,1>>>(SIZE);

cudaEventRecord(empty_stop, 0);

cudaEventSynchronize(empty_start); //optional
cudaEventSynchronize(empty_stop);

float emptyloop;
cudaEventElapsedTime(&emptyloop,empty_start,empty_stop);

cudaEventDestroy(empty_start);
cudaEventDestroy(empty_stop);
//cout << "Empty Loop time:"<<emptyloop<<endl;

//--------------------------------------------------------------
cudaEvent_t start,stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
//      cudaStatus = cudaDeviceSynchronize();
```

```
cudaEventRecord(start, 0);

flopsCUDA<<<1,1>>>(d_total,SIZE);

cudaEventRecord(stop, 0);


cudaEventSynchronize(start); //optional

cudaEventSynchronize(stop);



float elapsedTime;

cudaEventElapsedTime(&elapsedTime, start, stop); // that's our time! Clean up:


cudaEventDestroy(start);

cudaEventDestroy(stop);


//--------------------------------------------------------------------------------------

//cout << "Time elapsed:"<<elapsedTime<<endl;


cudaMemcpy(&total,d_total,sizeof(long int),cudaMemcpyDeviceToHost);

cudaMemcpy(&time,d_time,sizeof(double),cudaMemcpyDeviceToHost);


long double flops=SIZE/(emptyloop-elapsedTime);


cout<<"\nFLOPS:"<<flops<<endl;
```

```
double gflops=flops/1000000000;

cout<< "\n The GFLOPS:"<<gflops;
//cout<<"\nThe answer is "<<total<<endl;
//cout<<"The answer is "<<b<<endl;

cudaFree(d_total);
cudaFree(d_time);


}


int main(){

//calculateFlops();

cout <<"\n\n---------------------GIOPS CUDA Benchmarking------------------------------------\n\n";
calculateIops();

return 0;
}
```

Memory GPU

```
//
```

```
// MemoryCuda.cu
// GPU Memory Benchmarking
//
// Created by Ronakkumar Makadiya (CWID - A20332994)
// Created on 22/09/2104
// Copyright (c) 2014 Ronakkumar Makadiya. All rights reserved.
//


#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

void calclulate(float timeTaken, int no_of_threads){
	float data = 0.001;
	printf("\n%f GB/sec", (data / (timeTaken / 1024.0))*no_of_threads);

}


__global__ void readwritebyte(char *str, int *size){

	char* a_to_z = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
	char str2 = a_to_z[0];
	char *s = &str2;
	int index = threadIdx.x + blockIdx.x * blockDim.x;

	for (int i = 0; i < 1024*1024; i++){
		memcpy(&str[index], s, sizeof(char));
	}

	//free(s);
}



void startkernal(int threads,int blocks,int* size)
{

	//cudaError_t cudaStatus;
	cudaEvent_t start, stop;
	char *str_d;
	int *size_d;
	float time;
```

```
        cudaMalloc((void**)&str_d, *size * sizeof(char));
        cudaMalloc((void**)&size_d, sizeof(int));
        cudaMemcpy(size_d,size, sizeof(int), cudaMemcpyHostToDevice);



        cudaEventCreate(&start);
        cudaEventCreate(&stop);
        cudaEventRecord(start, 0);

        readwritebyte <<<blocks, threads >>>(str_d,size_d);

        cudaEventRecord(stop, 0);
        cudaEventSynchronize(stop);
        cudaEventElapsedTime(&time, start, stop);
        printf("\nTime for read and write one Byte kernel: %f ms", time);

        calclulate(time, blocks*threads);
}


int main()
{

        int num_of_blocks=1024;
        int num_of_threads_block=1024;
        int mem_size=1024*1024*1024;

        startkernal(num_of_threads_block,num_of_blocks,&mem_size);

        return 0;

}
```

3. Memory Benchmarking:

i. memoryonebyte.c

```
// Memory Benchmarking - For 1 byte
// memoryonebyte.c
// Created by Ronakkumar Makadiya (CWID - A20332994) Kaustubh Bojewar(CWID: A20329244) Sourabh Chougale(CWID: A20326997)
// Created on 09/09/2104
// Copyright (c) 2014 Ronakkumar Makadiya Kaustubh Bojewar Sourabh Chougale. All rights reserved.
#include <iostream>
#include <cstdlib>
#include <pthread.h>
#include <stdio.h>
#include <time.h>
#include <limits.h>
#include<string.h>
double sequentialThroughput=0,randomThroughput=0;
double sequentialLatency=0,randomLatency=0;
struct arg_struct {
int thread_id;
int blockSize;
};

int random_number(int limit)
{
```

```
int result=rand()%limit;

return result;

}


char* initializeBlock(long int blocksize)

{


const char* const a_to_z = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

long int allocate_size=blocksize*10*1024;

char* buffer = new char[allocate_size];

long int i=0;

for(i = 0; i < allocate_size; i++)

{

buffer[i] = a_to_z[random_number(26)];

}


return buffer;

}


void sequential()

{

char* block1=initializeBlock(1);

char* block2=initializeBlock(1);

char* temp = new char[10*1024];
```

```
char* block1_temp=block1;

char* block2_temp=block1;

char* temp_temp=temp;



volatile long int i,j;

clock_t t1,t2,total_time=0;

for(i=0;i<100;i++)

{


for(j=0;j<10*1024;j++)

{


t1 = clock();

memcpy(temp, block1, 1); //1 Read and 1Write operation.

memset(block1, '-',1); // 1 WriteOperation

memcpy(block1, block2,1); //1 Read and1 Write operation.

memset(block2, '#',1); // 1 WriteOperation

memcpy(block2, temp,1); // // 1 Readand 1 Write operation.

t2 = (clock() - t1);

total_time += t2;


block1 += 1;

block2 += 1;

temp += 1;
}

block1 = block1_temp;
block2 = block2_temp;
temp = temp_temp;

}

double timeTaken =((double)total_time)/CLOCKS_PER_SEC;
long bytesTransfer=(long)1 * 100 *10*1024*8;

double throughput_seq=bytesTransfer/(timeTaken*1024*1024);
//printf("\nseq:%f\n",throughput_seq);
long double temp_latency=(timeTaken*1000)/(100 *10*1024*8);
//printf("\nlat:0.8g%\n",temp_latency);
sequentialThroughput+=throughput_seq;
sequentialLatency+=temp_latency;
}

void randomrw()
{

char* block1=initializeBlock(1);
char* block2=initializeBlock(1);
```

```
char* temp = new char[10*1024];


char* block1_temp=block1;

char* block2_temp=block1;

char* temp_temp=temp;




volatile long int i,j;

clock_t t1,t2,total_time=0;

for(i=0;i<100;i++)

{


for(j=0;j<10*1024;j++)

{


t1 = clock();

memcpy(temp, block1, 1); //1 Read and 1Write operation.

memset(block1, '-',1); // 1 WriteOperation

memcpy(block1, block2,1); //1 Read and1 Write operation.

memset(block2, '#',1); // 1 WriteOperation

memcpy(block2, temp,1); // // 1 Readand 1 Write operation.

t2 = (clock() - t1);

total_time += t2;


block1 += (random_number(1024*10)-1)*1;
block2 +=(random_number(1024*10)-1)*1;
temp += (random_number(1024*10)-1)*1;


block1 = block1_temp;
block2 = block2_temp;
temp = temp_temp;

}


}
double timeTaken =((double)total_time)/CLOCKS_PER_SEC;
long bytesTransfer=(long)1 * 100 *10*1014*8;

double temp_latency=(timeTaken*1000)/(100 *10*1024*8);
double throughput=bytesTransfer/(timeTaken*1024*1024);
//printf("thru:ra:%f\n",throughput);

randomThroughput+=throughput;
randomLatency+=temp_latency;
}


void *calculateBenchmark(void *arguments)
```

```
{

sequential();
randomrw();

return NULL;

}


main()
{

int no_of_threads=0;

printf("\nEnter the number of threads :");
scanf("%d",&no_of_threads);

pthread_t threads[no_of_threads];

int i=0;
int result=0;
for(i = 0; i < no_of_threads; i++)
{
```

```
result = pthread_create(&threads[i], NULL,calculateBenchmark,(void *)&no_of_threads);
if (result)
{
printf("Error:unable to create thread %d",result);
//exit(-1);
}
}
for(i=0;i<no_of_threads;i++)
{
pthread_join(threads[i],NULL);
}
printf("\n\n Throughput of Sequential read/write :%f",sequentialThroughput/no_of_threads);
printf("\n\n Latency of Sequential read/write :%f",sequentialLatency/no_of_threads);

printf("\n\n Throughput of Random read/write :%f",randomThroughput/no_of_threads);
printf("\n\n Latency of Random read/write :%f",randomLatency/no_of_threads);
}
```

ii. memoryonekb.c

```
// Memory Benchmarking - For 1 kb
// memoryonekb.c
// Created by Ronakkumar Makadiya (CWID - A20332994) Kaustubh Bojewar(CWID: A20329244) Sourabh Chougale(CWID: A20326997)
// Created on 09/09/2104
// Copyright (c) 2014 Ronakkumar Makadiya Kaustubh Bojewar Sourabh Chougale. All rights reserved.
#include <iostream>
#include <cstdlib>
#include <pthread.h>
#include <stdio.h>
#include <time.h>
#include <limits.h>
#include<string.h>

double sequentialThroughput=0,randomThroughput=0;

double sequentialLatency=0,randomLatency=0;

struct arg_struct {
int thread_id;
int blockSize;
};

int random_number(int limit)
{
int divisor = RAND_MAX/(limit+1);
int retval;
do {
retval = rand() / divisor;
} while (retval > limit);

return retval;

}


char* initializeBlock(long int blocksize)

{


const char* const a_to_z = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

long int allocate_size=blocksize*10*1024*10;


if(blocksize==1024*1024)

{

allocate_size=(blocksize+5)*10*50;

}


char* buffer = new char[allocate_size];

long int i=0;

for(i = 0; i < allocate_size; i++)

{

buffer[i] = a_to_z[random_number(26)];

}


return buffer;

}
```

```
void sequential(char* block1, char* block2, char* temp, char* block1_temp, char*
block2_temp, char* temp_temp)

{


volatile long int i,j;

clock_t t1,t2,total_time=0;

for(i=0;i<100;i++)

{


for(j=0;j<10*1024;j++)

{


t1 = clock();

memcpy(temp, block1, 1024); //1 Read and 1Write operation.

memset(block1, '-',1024); // 1 WriteOperation

memcpy(block1, block2,1024); //1 Read and1 Write operation.

memset(block2, '#',1024); // 1 WriteOperation

memcpy(block2, temp,1024); // // 1 Readand 1 Write operation.

t2 = (clock() - t1);

total_time += t2;


block1 += 1024;

block2 += 1024;

temp += 1024;
}

block1 = block1_temp+0;

block2 = block2_temp+0;

temp = temp_temp+0;

}

double timeTaken =((double)total_time)/CLOCKS_PER_SEC;

long bytesTransfer=(long)1024 * 100 *10*1024*8;

double temp_latency = (timeTaken)/(1024 * 1024 * 8);

                                                  double
throughput=bytesTransfer/(timeTaken*1024*1024);

sequentialThroughput+=throughput;

randomLatency+=temp_latency;

}

void random(char* block1, char* block2, char* temp, char* block1_temp, char* block2_temp,
char* temp_temp)

{

int max;
```

```
volatile long int i,j;

clock_t t1,t2,total_time=0;

for(i=0;i<100;i++)

{


for(j=0;j<10*1024;j++)

{


t1 = clock();

memcpy(temp, block1, 1024); //1 Read and 1Write operation.

memset(block1, '-',1024); // 1 WriteOperation

memcpy(block1, block2,1024); //1 Read and1 Write operation.

memset(block2, '#',1024); // 1 WriteOperation

memcpy(block2, temp,1024); // // 1 Readand 1 Write operation.

t2 = (clock() - t1);

total_time += t2;

block1 += (random_number(10))*1024;

block2 +=(random_number(10))*1024;

temp += (random_number(10))*1024;


}


block1 = block1_temp+0;

block2 = block2_temp+0;

temp = temp_temp+0;
}


double timeTaken =((double)total_time)/CLOCKS_PER_SEC;
long bytesTransfer=(long)1024 * 100 *10*1024*8;


double throughput=bytesTransfer/(timeTaken*1024*1024);
double temp_latency = (timeTaken)/(1024*1024*8);
randomLatency=temp_latency;
randomThroughput+=throughput;
}



void *calculateBenchmark(void *arguments)
{
char* block1=initializeBlock(1024);
char* block2=initializeBlock(1024);
char* temp = new char[10*1024*1024*10];


char* block1_temp=block1;
char* block2_temp=block1;
char* temp_temp=temp;


sequential(block1, block2, temp,block1_temp,block2_temp,temp_temp);


random(block1, block2, temp,block1_temp,block2_temp,temp_temp);
```

```
}



main()
{

int no_of_threads=0;

printf("\nEnter the number of threads :");
scanf("%d",&no_of_threads);

pthread_t threads[no_of_threads];
int i=0;
int result=0;
for(i = 0; i < no_of_threads; i++)
{

result = pthread_create(&threads[i], NULL,calculateBenchmark,(void *)&no_of_threads);
if (result)
{
printf("Error:unable to create thread %d",result);
//exit(-1);
}
}
```

```
for(i=0;i<no_of_threads;i++)

{

pthread_join(threads[i],NULL);

}


printf("\n\n Throughput of Sequential read/write :%f",sequentialThroughput/no_of_threads);


printf("\n\n Latency of Sequential read/write :%0.11f",sequentialLatency/no_of_threads);


printf("\n\n Throughput of Random read/write :%f",randomThroughput/no_of_threads);


printf("\n\n Latency of Random read/write :%0.11f",randomLatency/no_of_threads);


pthread_exit(NULL);

}
```

iii. memoryonekb.c

```
// Memory Benchmarking - For 1 Mb

// memoryoneMb.c

// Created by Ronakkumar Makadiya (CWID - A20332994) Kaustubh Bojewar(CWID: A20329244) Sourabh Chougale(CWID: A20326997)

// Created on 09/09/2104

// Copyright (c) 2014 Ronakkumar Makadiya Kaustubh Bojewar Sourabh Chougale. All rights reserved.

#include <iostream>
```

```
#include <cstdlib>

#include <pthread.h>

#include <stdio.h>

#include <time.h>

#include <limits.h>

#include<string.h>


double sequentialThroughput=0,randomThroughput=0;

double sequentialLatency=0,randomLatency=0;


int random_number(int limit)

{

int result=rand()%limit;

return result;

}


char* initializeBlock(long int blocksize)

{


const char* const a_to_z = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

long allocate_size;


allocate_size=(blocksize)*100;


char* buffer = new char[allocate_size];
```

```
long int i=0;

for(i = 0; i < allocate_size; i++)

{

buffer[i] = a_to_z[random_number(26)];


}


return buffer;

}


void sequential(char* block1, char* block2, char* temp, char* block1_temp, char*
block2_temp, char* temp_temp)

{


volatile long int i,j;

clock_t t1,t2,total_time=0;

for(i=0;i<10;i++)

{


for(j=0;j<10;j++)

{


t1 = clock();

memcpy(temp, block1, 1024*1024); //1 Read and 1Write operation.

memset(block1, '-',1024*1024); // 1 WriteOperation
memcpy(block1, block2,1024*1024); //1 Read and1 Write operation.
memset(block2, '#',1024*1024); // 1 WriteOperation
memcpy(block2, temp,1024*1024); // // 1 Readand 1 Write operation.
t2 = (clock() - t1);
total_time += t2;

block1 += 1024*1024;
block2 += 1024*1024;
temp += 1024*1024;

}

block1 = block1_temp;
block2 = block2_temp;
temp = temp_temp;

}

double timeTaken =((double)total_time)/CLOCKS_PER_SEC;
long bytesTransfer=(long)1024*1024*10*10*8;
double temp_latency = (timeTaken*10)/(1024*1024*8);
double throughput=bytesTransfer/(timeTaken*1024*1024);

sequentialThroughput+=throughput;
sequentialLatency+=temp_latency ;
```

```
}

void randomrw()
{



char* block1=initializeBlock(1024*1024);
char* block2=initializeBlock(1024*1024);
long temp_size=1024*1024*100;
char* temp = new char[temp_size];

char* block1_temp=block1;
char* block2_temp=block1;
char* temp_temp=temp;
int max;
volatile long int i,j;
clock_t t1,t2,total_time=0;
for(i=0;i<10;i++)
{

for(j=0;j<10;j++)
{

t1 = clock();
memcpy(temp, block1, 1024*1024); //1 Read and 1Write operation.

memset(block1, '-',1024*1024); // 1 WriteOperation

memcpy(block1, block2,1024*1024); //1 Read and1 Write operation.

memset(block2, '#',1024*1024); // 1 WriteOperation

memcpy(block2, temp,1024*1024); // // 1 Readand 1 Write operation.

t2 = (clock() - t1);

total_time += t2;


//printf("%d\n",random_number(100));

block1 += (random_number(100)-1)*1024*1024;

block2 +=(random_number(100)-1)*1024*1024;

temp += (random_number(100)-1)*1024*1024;




block1 = block1_temp;

block2 = block2_temp;

temp = temp_temp;

}

}


double timeTaken =((double)total_time)/CLOCKS_PER_SEC;

long bytesTransfer=(long)(1024 *1024)*10*10*8;


double throughput=bytesTransfer/(timeTaken*1024*1024);
```

```
}



double temp_latency = (timeTaken*10)/(1024*1024*8);


randomThroughput+=throughput;

randomLatency += temp_latency ;

}



void *calculateBenchmark(void *arguments)

{

char* block1=initializeBlock(1024*1024);

char* block2=initializeBlock(1024*1024);

long temp_size=1024*1024*100;

char* temp = new char[temp_size];


char* block1_temp=block1;

char* block2_temp=block1;

char* temp_temp=temp;



sequential(block1, block2, temp,block1_temp,block2_temp,temp_temp);


randomrw();


}
```

```
main()

{


int no_of_threads=0;


printf("\nEnter the number of threads :");

scanf("%d",&no_of_threads);


pthread_t threads[no_of_threads];


int i=0;

int result=0;

for(i = 0; i < no_of_threads; i++)

{


result = pthread_create(&threads[i], NULL,calculateBenchmark,(void *)&no_of_threads);

if (result)

{

printf("Error:unable to create thread %d",result);

//exit(-1);

}

}

for(i=0;i<no_of_threads;i++)
{
pthread_join(threads[i],NULL);
}

printf("\n\n Throughput of Sequential read/write :%f",sequentialThroughput/no_of_threads);
printf("\n\n Latency of Sequential read/write :%0.11f",sequentialLatency/no_of_threads);

printf("\n\n Throughput of Random read/write :%f",randomThroughput/no_of_threads);
printf("\n\n Latency of Random read/write :%0.11f",randomLatency/no_of_threads);
pthread_exit(NULL);
}
```

4. Disk Benchmarking:

```
// Disk Benchmarking
// Created by        KAUSTUBH BOJEWAR          (CWID: A20329244)
// Created on 09/09/2104
// Copyright (c) 2014 Kaustubh Bojewar. All rights reserved

#include<stdio.h>
#include<math.h>
#include<limits.h>
#include<time.h>
#include<pthread.h>
#include<string.h>
//#include<fstream.h>

#define BYTE 1
#define KB 1024
#define MB 1048576
#define MAX_BYTE 10000000
#define MAX_KB 1000
#define MAX_MB 10
#define MAX_BYTE_WR 100000
#define MAX_KB_WR 1000
#define MAX_MB_WR 10

double throughput,latency,latency1,throughput1;
```

```
pthread_mutex_t lock;

int no_of_threads;

struct disk_str

{

int thread_id;

long int type;

long int max;

long int max_wr;

};

void *operate_random(void *arguments1)

{

int ch1,j;

pthread_t threads[no_of_threads];

int result;

int thread_args[no_of_threads];

struct disk_str *args = (struct disk_str *)arguments1;

FILE *fp,*fp1;

//Random Read and Write 1Byte Data

char buffer[1024*1024];

clock_t t1,t2,total_time=0,total_time1=0;

double throughput=0.0;

int k,l;

fp= fopen("test.txt", "r");

//fseek(fp,r,SEEK_SET);

//long temp=0;
```

```
int s,r;
for(k=0;k<args->max;k++)
{
s=rand()%3000;
t1=clock();
fseek(fp,s,SEEK_SET);
fread(buffer,1,args->type,fp);
t1=clock()-t1;
total_time+=t1;
}
fclose(fp);
double intloop =((double)total_time)/CLOCKS_PER_SEC;
//printf("%f\n",intloop);
latency=latency+intloop;
fp=fopen("test.txt","r");
fp1=fopen("rand_test.txt","r+");
fseek(fp,0,SEEK_SET);
for(l=0;l<args->max_wr;l++)
{
r= rand()%3000;
fread(buffer,1,args->type,fp);
t1=clock();
fseek(fp1,r,SEEK_SET);
fwrite(buffer,1,args->type,fp1);
t1=clock()-t1;
total_time1+=t1;
```

```
}
fclose(fp1);
fclose(fp);
double intloop1=((double)total_time1)/CLOCKS_PER_SEC;
latency1=latency1+intloop1;
}
void *operations(void *arguments)
{
//volatile int i=0;
int ch1,j;
pthread_t threads[no_of_threads];
int result;
int thread_args[no_of_threads];
struct disk_str *args = (struct disk_str *)arguments;
FILE *fp,*fp1;
//Sequential Read and Write 1Byte Data
char buffer[1024*1024];
clock_t t1,t2,total_time=0,total_time1=0;;
double throughput=0.0;
int k,l;
fp= fopen("test.txt", "r");
fseek(fp,0,SEEK_SET);
for(k=0;k<args->max;k++)
{
t1= clock();
fread(buffer,1,args->type,fp);
```

```
t1=clock()-t1;

total_time+=t1;

}

fclose(fp);

double intloop =((double)total_time)/CLOCKS_PER_SEC;

//printf("%f\n",intloop);

latency=latency+intloop;

fp=fopen("test.txt","r");

fp1=fopen("test1.txt","w+");

fseek(fp,0,SEEK_SET);

for(l=0;l<args->max_wr;l++)

{


fread(buffer,1,args->type,fp);

t1=clock();

fwrite(buffer,1,args->type,fp1);

t1=clock()-t1;

total_time1+=t1;

}

fclose(fp1);


double intloop1=((double)total_time1)/CLOCKS_PER_SEC;

latency1=latency1+intloop1;

}
```

```
int main()

{

struct disk_str args;

int ch,ch1,ch2,i,result;

//int thread_args[no_of_threads];

pthread_t threads[no_of_threads];

pthread_mutex_init(&lock, NULL);


printf("..Menu...\n");

printf("1.SEQUENTIAL\n");

printf("2.RANDOM\n");

printf("3.EXIT\n");

printf("\nEnter your choice :");

scanf("%d",&ch);

switch(ch)

{

case 1:printf("You have selected SEQUENTIAL ACCESS\n");

        printf("\nEnter the number of threads (1,2,4)");

scanf("%d",&no_of_threads);

pthread_setconcurrency(no_of_threads);

printf("\n..MENU..");

printf("\n1. 1 BYTE");

printf("\n2. 1 KB");

printf("\n3. 1 MB");

printf("\n Enter your choice");

scanf("%d",&ch1);
```

```
switch(ch1)

{

case 1: printf("\nSequential Read and Write Operations for 1BYTE");

for(i = 1; i <= no_of_threads; i++)

{


args.thread_id = i;

args.type=BYTE;

args.max= MAX_BYTE;

args.max_wr=MAX_BYTE_WR;

result = pthread_create(&threads[i], NULL,operations,(void *)&args);

}

for (i=1; i<=no_of_threads; ++i)

{

result = pthread_join(threads[i], NULL);

}

double f_latency= (latency*1000)/MAX_BYTE;

printf("\nLatency = %f ms\n",f_latency);

double avg = latency/no_of_threads;

throughput=(MAX_BYTE)/(avg*1024*1024);

printf("Throughput = %f MB/sec\n",throughput);

printf("Write Sequential operations for 1BYTE\n");


double f_latency1=(latency1*1000)/MAX_BYTE_WR;

printf("\nLatency= %f ms\n",f_latency1);
```

```
double avg1 = latency1/no_of_threads;
throughput1=(MAX_BYTE_WR)/(avg1*1024*1024);
printf("Throughput = %f MB/sec\n",throughput1);
pthread_mutex_destroy(&lock);
break;
case 2: printf("\nSequential Read and Write Operations for 1KB\n");
for(i = 1; i <= no_of_threads; i++)
{

args.thread_id = i;
args.type=KB;
args.max=MAX_KB;
args.max_wr=MAX_KB_WR;
result = pthread_create(&threads[i], NULL,operations,(void *)&args);

}
for (i=1; i<=no_of_threads; ++i)
{
result = pthread_join(threads[i], NULL);
}
f_latency=(latency*1000)/MAX_KB;
printf("Latency = %f ms\n",f_latency);
avg = latency/no_of_threads;
throughput=(MAX_KB)/(avg*1024);
printf("Throughput = %f MB/sec\n",throughput);
```

```
f_latency1= (latency1*1000)/MAX_KB_WR;
printf("\nLatency= %f ms\n",f_latency1);
avg1 = latency1/no_of_threads;
throughput1=(MAX_KB_WR)/(avg1*1024);
printf("Throughput = %f MB/sec\n",throughput1);
pthread_mutex_destroy(&lock);
break;

case 3: printf("\nSequential Read and Write Operations for 1MB\n");
for(i = 1; i <= no_of_threads; i++)
{

args.thread_id = i;
args.type=MB;
args.max=MAX_MB;
args.max_wr=MAX_MB_WR;
result = pthread_create(&threads[i], NULL,operations,(void *)&args);

}
for (i=1; i<=no_of_threads; ++i)
{
result = pthread_join(threads[i], NULL);
}
f_latency=(latency*1000)/MAX_MB;
printf("Latency = %f ms\n",f_latency);
avg = latency/no_of_threads;
```

```
throughput=(MAX_MB)/(avg);

printf("Throughput = %f MB/sec\n",throughput);


f_latency1=(latency1*1000)/MAX_MB_WR;

printf("\nLatency= %f ms\n",f_latency1);

avg1 = latency1/no_of_threads;

throughput1=(MAX_MB_WR)/(avg1);

printf("Throughput = %f MB/sec\n",throughput1);
        pthread_mutex_destroy(&lock);

break;

}

break;


case 2:printf("You have selected RANDOM ACCESS \n");

printf("\nEnter the number of threads (1,2,4)");

scanf("%d",&no_of_threads);

pthread_setconcurrency(no_of_threads);


printf("\n..MENU..");

printf("\n1. 1 BYTE");

printf("\n2. 1 KB");

printf("\n3. 1 MB");

printf("\n Enter your choice");

scanf("%d",&ch2);

switch(ch2)

{

case 1:printf("\nRandom Read and Write Operations for 1BYTE");
```

```
printf("\n\nRandom Read\n");                                                                        for(i =
1; i <= no_of_threads; i++)
{
args.thread_id = i;
args.type=BYTE;
args.max=10000000;
args.max_wr=100000;
result = pthread_create(&threads[i], NULL,operate_random,(void *)&args);


}
for (i=1; i<=no_of_threads; ++i)
{
result = pthread_join(threads[i], NULL);
}
double f_latency= (latency*1000)/10000000;
printf("\nLatency = %f ms\n",f_latency);
double avg = latency/no_of_threads;
throughput=(10000000)/(avg*1024*1024);
printf("Throughput = %f MB/sec\n\n",throughput);
printf("Random Write\n");


double f_latency1= (latency1*1000)/100000;
printf("\nLatency= %f ms\n",f_latency1);
double avg1 = latency1/no_of_threads;
throughput1=(100000)/(avg1*1024*1024);
printf("Throughput = %f MB/sec\n",throughput1);
pthread_mutex_destroy(&lock);
```

```
break;


case 2: printf("\nRandom Read and Write Operations for 1 KB");

printf("\n\nRandom Read");

for(i = 1; i <= no_of_threads; i++)

{


args.thread_id = i;

args.type=KB;

args.max=150000;

args.max_wr=1000000;

result = pthread_create(&threads[i], NULL,operate_random,(void *)&args);


}

for (i=1; i<=no_of_threads; ++i)

{

result = pthread_join(threads[i], NULL);

}

f_latency= (latency*1000)/150000;

printf("\nLatency = %f ms\n",f_latency);

avg = latency/no_of_threads;

throughput=(1000)/(avg*1024);

printf("Throughput = %f MB/sec\n\n",throughput);

printf("Random Write\n");


f_latency1= (latency1*1000)/1000000;
```

```
printf("\nLatency= %f ms\n",f_latency1);

avg1 = latency1/no_of_threads;

throughput1=(1000)/(avg1*1024);

printf("Throughput = %f MB/sec\n",throughput1);

pthread_mutex_destroy(&lock);

break;


case 3: printf("\nRandom Read and Write Operations for 1 MB");

printf("\n\nRandom Read");

for(i = 1; i <= no_of_threads; i++)

{


args.thread_id = i;

args.type=MB;

args.max=100;

args.max_wr=100;

result = pthread_create(&threads[i], NULL,operate_random,(void *)&args);


}

for (i=1; i<=no_of_threads; ++i)

{

result = pthread_join(threads[i], NULL);

}

f_latency= (latency*1000)/100;

printf("\nLatency = %f ms\n",f_latency);

avg = latency/no_of_threads;
```

```
throughput=(100)/(avg*1024);
printf("Throughput = %f MB/sec\n\n",throughput);
printf("Random Write\n");
f_latency1= (latency1*1000)/100;
printf("\nLatency= %f ms\n",f_latency1);
avg1 = latency1/no_of_threads;
throughput1=(100)/(avg1*1024);
printf("Throughput = %f MB/sec\n",throughput1);
pthread_mutex_destroy(&lock);
break;
//operation(no_of_threads,ch);
break;
}
case 3: //exit();

break;

}

return 0;

}
```

5. Network Benchmarking:

```
// menu.java

// Created by Sourabh Chougale  (CWID: A20326997)

// Created on 09/09/2104

// Copyright (c) 2014 Sourabh Chougale. All rights reserved
```

i. menu.java

```
// Created by Sourabh Chougale  (CWID: A20326997)

// Created on 09/09/2104

// Copyright (c) 2014 Sourabh Chougale. All rights reserved

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;


public class menu {


public static void main(String args[]) throws IOException, InterruptedException {

BufferedReader a;

a = new BufferedReader(new InputStreamReader(System.in));

while(true){

System.out.println("\nSelect transmission protocol");

System.out.println("\n1. TCP");

System.out.println("\n2. UDP");

System.out.println("\n3. Exit");

String s1 = a.readLine(); // taking users preference for transmission protocol.
```

```
int ch = Integer.parseInt(s1);

switch (ch) {

case 1:


System.out.println("Please Enter Number of Thread");

int threadCount=Integer.parseInt(a.readLine());// Taking number of threads.

int portNumberArray[]=new int[threadCount]; // start and initialize port number array to take
port number for each port.

for(int index=0;index<threadCount;index++)

{

System.out.println("Please Enter portNumber for running Server "+(index+1));

portNumberArray[index]=Integer.parseInt(a.readLine());// Taking port number from user.

}

int choice =selectBlockSize(a);        // Take a block size to be transfer.

for (int i = 0; i < portNumberArray.length; i++) {

tcpServer ser=new tcpServer(portNumberArray[i]);

ser.setName("Server"+(i+1));

ser.start();

Thread client =new Thread(new tcpClient(portNumberArray[i],choice,ser));;

client.start();// Starting thread

client.join();// joining multiple threads in case of multithreading to achieve synchronization.

}

break;

case 2:

System.out.println("Please Enter Number of Thread");
int threadCount1=Integer.parseInt(a.readLine());// Taking number of threads.
int portNumberArray1[]=new int[threadCount1];// start and initialize port number array to take port number for each port.
for(int index=0;index<threadCount1;index++){
System.out.println("Please Enter portNumber for running Server "+(index+1));
portNumberArray1[index]=Integer.parseInt(a.readLine()); // Taking port number from user.
}
int choice1 =selectBlockSize(a);   // Take a block size to be transfer.
int blockSize=choice1== 1 ? 1:(choice1==2?1024:65530);
for (int i = 0; i < portNumberArray1.length; i++) {
UdpServer ser=new UdpServer(portNumberArray1[i],blockSize);
ser.setName("Server"+(i+1));
Thread client =new Thread(new UdpClient(portNumberArray1[i],choice1,ser));;
client.start();// Starting thread
client.join();// joining multiple threads in case of Multithreading to achieve synchronization.
}
break;
case 3:
System.out.println("Exiting The Application");
System.exit(0);
break;
default:
System.out.println("Please Enter valid input");
break;
}
```

```
}
}

private static int selectBlockSize(BufferedReader a) throws NumberFormatException,
IOException {
System.out.println("Please Select the Operation");
System.out.println("\nSelect Block Size to Send to the Servers");
System.out.println("\n1. 1byte");
System.out.println("\n2. 1kilobyte");
System.out.println("\n3. 64Kilobytes");
int choice =Integer.parseInt(a.readLine());
return choice;
}
}
```

ii. __TcpClient.java__

```
// TcpClient.java
// Created by Sourabh Chougale  (CWID: A20326997)
// Created on 09/09/2104
// Copyright (c) 2014 Sourabh Chougale. All rights reserved
import java.net.*;
import java.util.ArrayList;
import java.util.List;
import java.io.*;
//import java.util.*;
public class tcpClient implements Runnable
```

```
{
private static int count=-1;
private int portNumber;
private int choice;
private tcpServer thread;

public void client(int portNumber,int choice,tcpServer thread ) throws IOException{

Socket client=new Socket("localhost",portNumber);
DataInputStream b = new DataInputStream(client.getInputStream());
double blockSize;

switch (choice) {
case 1:
blockSize = 1;
List<Double> avg =new ArrayList<Double>();
List<Double> avgthr1= new ArrayList<Double>();

avgthr1=sendAndReadByte(client, b, blockSize, avg, avgthr1);
double avgTime=calculateAvgTime(avg);
double avgThrb=calculateAvgTime(avgthr1);
System.out.println("\n*********BenchMarking Result for "+thread.getName()+"*************");
System.out.println(" Average Time Taken = "+ avgTime);
System.out.println("Average Throughput = "+ avgThrb);
```

```
thread.getServer().close();
break;

case 2:
blockSize=1024;
List<Double> average= new ArrayList<Double>();
List<Double> avgthr2= new ArrayList<Double>();
sendReceivekb(client, b, blockSize, average, avgthr2);
double avgTimekb=calculateAvgTime(average);
double avgThrkb=calculateAvgTime(avgthr2);
System.out.println("\n*********BenchMarking Result for "+thread.getName()+"*************");
System.out.println("Average Time Taken = "+ avgTimekb);
System.out.println("Average Throughput = "+ avgThrkb);
thread.getServer().close();
break;

case 3:
blockSize=65536;
List<Double> averages= new ArrayList<Double>();
List<Double> avgthr3= new ArrayList<Double>();
sendRead64kb(client, b, blockSize, averages, avgthr3);
double avgTime64kb=calculateAvgTime(averages);
double avgThr64kb=calculateAvgTime(avgthr3);
System.out.println("\n*********BenchMarking Result for "+thread.getName()+"*************");
System.out.println("Average Time Taken = "+ avgTime64kb);
System.out.println("Average Throughput = "+ avgThr64kb);
break;
default:

break;
}
}



private static void sendRead64kb(Socket client, DataInputStream b,
double blockSize, List<Double> averages,List<Double> avgthr3) throws IOException {
++count;
byte [] z1 = new byte[65536];
for(long d=0;d<65536;d++)
{
z1[(int) d]= (byte) (1);
};
DataOutputStream p = new DataOutputStream(client.getOutputStream());

long t1 = System.nanoTime();

p.write(z1);
```

```
p.flush();
String s2 = b.readUTF();
System.out.println(":  " + s2);
long t2 = System.nanoTime();

long t3 = t2 - t1;

double t4 = t3 * Math.pow(10, -9);
averages.add(new Double(t4));

double throughput = ((blockSize*8) / (t4 * 1024 * 1024));
avgthr3.add(new Double (throughput));

if(count<10){
sendReceivekb(client, b, blockSize, averages, avgthr3);
}
}
private static void sendReceivekb(Socket client, DataInputStream b,
double blockSize, List<Double> average, List<Double> avgthr2) throws IOException {
byte [] kb = new byte[1024];
for(int e=0;e<1024;e++){
kb[e]= (byte)(1);
};
DataOutputStream p = new DataOutputStream(client.getOutputStream());
for (int j = 0; j < 10; j++) {
```

```
long t1 = System.nanoTime();

p.write(kb);

long t2 = System.nanoTime();

long t3 = t2 - t1;

double t4 = t3 * Math.pow(10, -9);

average.add(new Double(t4));


double throughput = ((blockSize*8) / (t4 * 1024 * 1024));

avgthr2.add(new Double (throughput));


}

}


private static double calculateAvgTime(List<Double> avg) {

double sum=0;

for (Double value : avg) {

sum= sum+value.doubleValue();

}

return sum/avg.size();


}


private static List<Double> sendAndReadByte(Socket client, DataInputStream b,

double blockSize, List<Double> avg,

List<Double> avgthr1) throws IOException {
```

```
byte[] z = new byte[] { 1 };
DataOutputStream p = new DataOutputStream(client.getOutputStream());
for (int j = 0; j < 10; j++) {
long t1 = System.nanoTime();
p.write(z);
p.flush();
b.readUTF();

long t2 = System.nanoTime();

long t3 = t2 - t1;

double t4 = t3 * Math.pow(10, -9);
avg.add(new Double(t4));

double throughput = ((blockSize*8) / (t4 * 1024 * 1024));
avgthr1.add(new Double(throughput));

}
return (avgthr1);

}
public void run() {
try{
client(this.portNumber , this.choice ,this.thread);
} catch (IOException e){

e.printStackTrace();

}


}

public tcpClient(int portNumber,int choice,tcpServer thread){

this.portNumber=portNumber;

this.choice = choice;

this.thread=thread;

}


}
```

iii. TcpServer.java

```
// TcpServer.java

// Created by Sourabh Chougale  (CWID: A20326997)

// Created on 09/09/2104

// Copyright (c) 2014 Sourabh Chougale. All rights reserved


import java.net.*;

import java.io.*;

//import java.util.*;
```

```
public class tcpServer extends Thread implements Runnable {
private int portNumber;
public ServerSocket server;

public void startServer(int portNumber) throws IOException {
server=new ServerSocket(portNumber);

Socket client = server.accept();
DataOutputStream e=new DataOutputStream(client.getOutputStream());
DataInputStream b = new DataInputStream(client.getInputStream());

while(true){
b.read();
String st2="Data received successfully";
e.writeUTF(st2);
e.flush();
}


}
public void run() {
try {
startServer(this.portNumber);
} catch (IOException e) {
// TODO Auto-generated catch block
e.printStackTrace();
}
}
public tcpServer(int portNumber) {
this.portNumber=portNumber;
}
public ServerSocket getServer() {
return server;
}
public void setServer(ServerSocket server) {
this.server = server;
}
}
```

iv. UdpClient.java

```
// UdpClient.java
// Created by Sourabh Chougale  (CWID: A20326997)
// Created on 09/09/2104
// Copyright (c) 2014 Sourabh Chougale. All rights reserved

import java.io.*;
import java.net.*;
import java.util.ArrayList;
import java.util.List;
```

```
class UdpClient implements Runnable

{

private int portNumber1;

private int choice1;

private UdpServer serv;

String serverHostname = new String ("127.0.0.1");

public UdpClient(int portNumber1, int choice1, UdpServer ser) {

this.portNumber1=portNumber1;

this.choice1=choice1;

this.serv=ser;


}


public void udpclient(int portNumber1,int choice1,UdpServer thread)throws IOException{

DatagramSocket Sock= new DatagramSocket();

switch (choice1) {

case 1:


List<Double> avg =new ArrayList<Double>();

List<Double> avgthr1= new ArrayList<Double>();

send1Byte(portNumber1, Sock,avg, avgthr1);

double avgTime=calculateAvgTime(avg);

double avgThrb=calculateAvgTime(avgthr1);

System.out.println("\n*********BenchMarking Result for
"+thread.getName()+"*************");
```

```
System.out.println("Average Time Taken = "+ avgTime);
System.out.println("Average Throughput = "+ avgThrb);
thread.getSock().close();
break;
case 2:
List<Double> average =new ArrayList<Double>();
List<Double> avgthr2= new ArrayList<Double>();
send1KB(portNumber1, Sock,average,avgthr2);
double avgTime1=calculateAvgTime(average);
double avgThrb1=calculateAvgTime(avgthr2);
System.out.println("\n*********BenchMarking Result for
"+thread.getName()+"*************");
System.out.println("Average Time Taken = "+ avgTime1);
System.out.println("Average Throughput = "+ avgThrb1);
thread.getSock().close();
break;

case 3:
List<Double> averages =new ArrayList<Double>();
List<Double> avgthr3= new ArrayList<Double>();
send64KB(portNumber1, Sock,averages,avgthr3);
double avgTime2=calculateAvgTime(averages);
double avgThrb2=calculateAvgTime(avgthr3);
System.out.println("\n*********BenchMarking Result for
"+thread.getName()+"*************");
System.out.println("Average Time Taken = "+ avgTime2);
System.out.println("Average Throughput = "+ avgThrb2);

thread.getSock().close();

break;

}


}


private void send1KB(int portNumber1, DatagramSocket Sock,List<Double>
average,List<Double> avgthr2)

throws SocketException, UnknownHostException, IOException {

double blockSize=1;

serv.start();


byte[] buffer1 = new byte[1024];

for(int e=0;e<1024;e++){

buffer1[e]= (byte)(1);

};

for (int j = 0; j < 10;j++) {

long t1 = System.nanoTime();

DatagramPacket sendPacket1 =

new DatagramPacket(buffer1, buffer1.length,
InetAddress.getByName(serverHostname),portNumber1);

Sock.send(sendPacket1);

long t2 = System.nanoTime();

long t3 = t2 - t1;

double t4 = t3 * Math.pow(10, -9);
```

```
average.add(new Double(t4));
double throughput = ((blockSize*8) / (t4 * 1024 * 1024));
avgthr2.add(new Double (throughput));
}
}
private void send64KB(int portNumber1, DatagramSocket Sock,List<Double>
averages,List<Double> avgthr3
) throws SocketException, UnknownHostException,
IOException {
double blockSize3=61500;
serv.start();

byte[] buffer2 = new byte[61500];
for(int e=0;e<61500;e++){
buffer2[e]= (byte)(1);
};
for (int j = 0; j < 10;j++) {
long t1 = System.nanoTime();
System.out.println("\nStart time :  " + t1);
DatagramPacket sendPacket2 =
new DatagramPacket(buffer2, buffer2.length,
InetAddress.getByName(serverHostname),portNumber1);
Sock.send(sendPacket2);
long t2 = System.nanoTime();
long t3 = t2 - t1;
double t4 = t3 * Math.pow(10, -9);
```

```
averages.add(new Double(t4));

double throughput = ((blockSize3*8) / (t4 * 1024 * 1024));

avgthr3.add(new Double (throughput));

}

}

private void send1Byte(int portNumber1, DatagramSocket Sock,

List<Double> avg, List<Double> avgthr1) throws SocketException, UnknownHostException,

IOException {

double blockSize=1;

serv.start();

byte[] buffer = new byte[] {1};


for (int j = 0; j < 10; j++)

{

long t1 = System.nanoTime();

DatagramPacket sendPacket =

new DatagramPacket(buffer, buffer.length,
InetAddress.getByName(serverHostname),portNumber1);

Sock.send(sendPacket);

long t2 = System.nanoTime();

long t3 = t2 - t1;

double t4 = t3 * Math.pow(10, -9);

avg.add(new Double(t4));

double throughput = ((blockSize*8) / (t4 * 1024 * 1024));

avgthr1.add(new Double (throughput));
```

```
}

}

private static double calculateAvgTime(List<Double> avg)

{

double sum=0;

for (Double value : avg) {

sum= sum+value.doubleValue();

}

return sum/avg.size();

}

public void run() {

try {

udpclient(this.portNumber1,this. choice1, this.serv);

} catch (IOException e) {

e.printStackTrace();

}

}


}
```

v. **UdpServer.java:**

```
// UdpServer.java

// Created by Sourabh Chougale  (CWID: A20326997)

// Created on 09/09/2104

// Copyright (c) 2014 Sourabh Chougale. All rights reserved
```

```
import java.io.IOException;

import java.net.*;

class UdpServer extends Thread implements Runnable

{

private int portNumber1;

DatagramSocket Sock;

DatagramPacket Dp;

byte[] receiveData ;

public void startUDPServer(int portNumber1) throws IOException

{

DatagramPacket receivePacket =

new DatagramPacket(receiveData, receiveData.length);

Sock.receive(receivePacket);



byte[] sentence = (byte[])(receivePacket.getData());

InetAddress IPAddress = receivePacket.getAddress();

int port = receivePacket.getPort();

System.out.println ("From: " + IPAddress + ":" + port);

System.out.println ("Received" + sentence.length +"bytes");

}



public UdpServer(int portNumber1,int size) throws SocketException {



this.portNumber1=portNumber1;
Sock = new DatagramSocket(portNumber1);

receiveData=new byte[size];

}


public void run() {

try {

startUDPServer(this.portNumber1);

} catch (IOException e) {

e.printStackTrace();

}

}

public DatagramSocket getSock() {

return Sock;

}

public void setSock(DatagramSocket sock) {

Sock = sock;

}
```