CS-553 CLOUD COMPUTING

PROGRAMMING ASSIGNMNET -1

BENCHMARKING

By

Teja Maripuri (A20376276)

Abhishek Vijhani (A20377670)

Source Code

CPU Benchmarking (Abhishek Vijhani)

```
#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<time.h>

#include<pthread.h>

#include<immintrin.h>

#include<limits.h>


long long ITR = INT_MAX;

struct thread_args {

   long long total;

};


void *compute_flops(void *arguments);

void *compute_iops(void *arguments);


int main( int argc, char *argv[])

{

            clock_t start_time,end_time, s_t, e_t;

            int choice = atoi(argv[1]);

            pthread_t *pt;

            pt = (pthread_t*)malloc(sizeof(pthread_t)*8);

            char str[32];



                        if(choice == 0){
```

```
        exit(0);
}


struct thread_args args;
long long total = ITR / choice;
args.total = total;


int i=0;
start_time = clock();
for(i=0; i < choice; i++)
{
        pthread_create(&pt[i],NULL,compute_flops, (void *)&args);
}
for(i=0; i < choice; i++)
{
        pthread_join(pt[i],NULL);
}
end_time = clock();


double time_taken = (double)(end_time - start_time) / (CLOCKS_PER_SEC);
printf("\nGFLOPS for %d Thread: %f\n", choice, (((12 * ITR)/time_taken)/1000000000));




start_time = clock();
for(i=0; i < choice; i++)
{
        pthread_create(&pt[i],NULL,compute_iops, (void *)&args);
}
```

```
        for(i=0; i < choice; i++)
        {
            pthread_join(pt[i],NULL);
        }
        end_time = clock();
        time_taken = (double)(end_time - start_time) / (CLOCKS_PER_SEC);
        printf("\nGILOPS for %d Thread: %f\n\n\n", choice, (((12 *
ITR)/time_taken)/1000000000));

        return 0;


}
void *compute_flops(void *arguments)
{
    struct thread_args *args = (struct thread_args *)arguments;
    long long total = args -> total;
    __m256d a = _mm256_set_pd(2.0, 4.0, 6.0, 8.0);
  __m256d b = _mm256_set_pd(1.0, 3.0, 5.0, 7.0);

    __m256d c = _mm256_set_pd(2.0, 4.0, 6.0, 8.0);
  __m256d d = _mm256_set_pd(1.0, 3.0, 5.0, 7.0);

    __m256d e = _mm256_set_pd(2.0, 4.0, 6.0, 8.0);
  __m256d f = _mm256_set_pd(1.0, 3.0, 5.0, 7.0);
    int i;
    for(i=0; i < total; i++){
        __m256d result = _mm256_add_pd(a, b);
        result = _mm256_add_pd(c, d);
        result = _mm256_add_pd(e, f);
    }
}

void *compute_iops(void *arguments)
{
    struct thread_args *args = (struct thread_args *)arguments;
    long long total = args -> total;
    __m256i a = _mm256_set_epi64x(1, 2, 3, 4);
    __m256i b = _mm256_set_epi64x(5, 6, 7, 8);

    __m256i c = _mm256_set_epi64x(1, 2, 3, 4);
    __m256i d = _mm256_set_epi64x(5, 6, 7, 8);

    __m256i e = _mm256_set_epi64x(1, 2, 3, 4);
    __m256i f = _mm256_set_epi64x(5, 6, 7, 8);
    int i;
    for(i=0; i < total; i++){
        __m256i result = _mm256_add_epi64(a, b);
        result = _mm256_add_epi64(c, d);
        result = _mm256_add_epi64(e, f);
    }
}
```

Memory Benchmarking (Teja Maripuri)

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<pthread.h>
#include<string.h>
```

```
#include<memory.h>


int b = 1;
int kb = 1024;
int mb = 1024 * 1024;
int gb = 1024 * 1024 * 1024;
char buff_mb[8388608];
char buff_gb[1024 * 1024 * 1024];
char types[5][20] = {"8 Byte Block", "8 Kilo Bytes Block", "8 Mega Bytes Block", "80 Mega Bytes Block"};
int blocks[] = {8*1, 8*1024, 8*1048576, 80*1048576};

int threads[] = {1, 2, 4, 8};
clock_t start_time,end_time;


FILE *fp;

struct thread_args {
   long long total;
   long block;
   int curr_part;
   int flag;
};



void *write(void *arguments);
void *read(void *arguments);
void *read_write(void *arguments);


void *write_random(void *arguments);

void *read_random(void *arguments);


int main()

{

    pthread_t *pt;

    pt = (pthread_t*)malloc(sizeof(pthread_t)*8);

    int itr;

    for (itr = 0; itr < 4; itr++)

    {

        printf("\n\nSequential Read/Write for %s\n\n", types[itr]);

        int t_iter;


        for (t_iter = 0; t_iter < 4; t_iter++)

        {

            if(itr == 0)

            {

                int i;

                //printf("Throughput ");

                struct thread_args args;

                long long trueTotal = 8*mb;

                args.total = trueTotal/threads[t_iter];

                args.block = blocks[itr];

                args.flag = 0;

                start_time = clock();

                for (i=0; i< threads[t_iter]; i++)

                {
                    args.curr_part = i;
                    pthread_create(&pt[i],NULL, &read_write, (void *)&args);
                }
                for (i=0; i< threads[t_iter]; i++)
                {
                    pthread_join(pt[i],NULL);
                }
                end_time = clock();
                double time_taken = (double)(end_time - start_time) / 10000000;
                printf("The Latency for Write+Read with %d thread(s): %f us\n",
threads[t_iter], time_taken);

            }
            else
            {
                int i;
                //printf("Throughput ");
                struct thread_args args;
                long long trueTotal = 1*gb;
                args.total = trueTotal/threads[t_iter];
                args.block = blocks[itr];
                args.flag = 1;
                start_time = clock();
                for (i=0; i< threads[t_iter]; i++)
                {
                    args.curr_part = i;
                    pthread_create(&pt[i],NULL, &read_write, (void *)&args);
                }
                for (i=0; i< threads[t_iter]; i++)
                {
                    pthread_join(pt[i],NULL);
                }
                end_time = clock();
                double speed = (double) (trueTotal/mb) / ((end_time - start_time) /
CLOCKS_PER_SEC) ;
                printf("The Throughput for Write+Read with %d thread(s): %.2f
Mbps\n", threads[t_iter], speed * 100);

                start_time = clock();
                for (i=0; i< threads[t_iter]; i++)
                {
                    args.curr_part = i;
                    pthread_create(&pt[i],NULL, &write, (void *)&args);
                }
                for (i=0; i< threads[t_iter]; i++)
                {
                    pthread_join(pt[i],NULL);
                }
                end_time = clock();
                speed = (double) (trueTotal/mb) / ((end_time - start_time) /
CLOCKS_PER_SEC) ;
                printf("The Throughput for Write      with %d thread(s): %.2f Mbps\n\n",
threads[t_iter], speed * 100);
            }
        }
    }

    for (itr = 0; itr < 4; itr++)
    {
        printf("\n\nRandom Read/Write for %s\n\n", types[itr]);
        int t_iter;

        for (t_iter = 0; t_iter < 4; t_iter++)
        {
            if(itr == 0)
            {
                int i;
                //printf("Throughput ");
                struct thread_args args;
                long trueTotal = 8*kb;
                args.total = trueTotal/threads[t_iter];
                args.block = blocks[itr];
                args.flag = 0;
                start_time = clock();
                for (i=0; i< threads[t_iter]; i++)
                {
                    args.curr_part = i;
                    pthread_create(&pt[i],NULL, &write_random, (void *)&args);
                }
                for (i=0; i< threads[t_iter]; i++)
                {
                    pthread_join(pt[i],NULL);
                }
                end_time = clock();
                double time_taken = (double)(end_time - start_time)  / 10000000;
                printf("The Latency for Write with %d thread(s): %f us\n",
threads[t_iter], time_taken);
            }
            else
            {
                int i;
                //printf("Throughput ");
                struct thread_args args;
                long trueTotal = 1*gb;
                args.total = trueTotal/threads[t_iter];
                args.block = blocks[itr];
                args.flag = 1;
                start_time = clock();
                for (i=0; i< threads[t_iter]; i++)
                {
                    args.curr_part = i;
                    pthread_create(&pt[i],NULL, &write_random, (void *)&args);
                }
                for (i=0; i< threads[t_iter]; i++)
                {
                    pthread_join(pt[i],NULL);
                }
                end_time = clock();
                double speed = (double) (trueTotal/mb) / ((end_time - start_time) /
CLOCKS_PER_SEC) ;
                printf("The Throughput for Write with %d thread(s): %.2f Mbps\n",
threads[t_iter], speed * 100);


            }
        }
    }
```

```
        return 0;
}

void *write(void *arguments)
{
        struct thread_args *args = (struct thread_args *)arguments;
        long total = args -> total;
        long block = args -> block;
        int curr_part = args -> curr_part;
        int flag = args -> flag;
        long i, j;
        long iter = total/block;
        for(j=0; j < 100; j++)
        {
                for(i=0; i < iter; i++)
                {
                        memset(&buff_gb[((curr_part * total) + (i*block))], 'a', block);
                }
        }

}

void *read_write(void *arguments)
{
        struct thread_args *args = (struct thread_args *)arguments;
        long total = args -> total;
        long block = args -> block;
        int curr_part = args -> curr_part;
        int flag = args -> flag;
```

```
    char *item = NULL;

    item = malloc (total * sizeof *item);

    long i, j;

    long iter = total/block;

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

    {

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

        {

            if(flag == 1)

            {

                memcpy(&buff_gb[((curr_part * total) + (i*block))],&item[(i*block)], block);

            }

            else

            {

                memcpy(&buff_mb[((curr_part * total) + (i*block))],&item[(i*block)], block);

            }

        }

    }


    free(item);

}





void *write_random(void *arguments)

{

    struct thread_args *args = (struct thread_args *)arguments;
```

```
        long total = args -> total;

        long block = args -> block;

        int curr_part = args -> curr_part;

        int flag = args -> flag;

        long i;

        long j;

        long iter = total/block;

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

        {

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

                {

                        if(flag == 1)

                        {

                                long rand_num = rand() % iter;

                                memset(&buff_gb[((curr_part * total) + (rand_num*block))], 'a', block);

                        }

                        else

                        {

                                long rand_num = rand() % iter;

                                memset(&buff_mb[((curr_part * total) + (rand_num*block))], 'a', block);

                        }

        }

        }


}
```

Disk Benchmarking (Teja Maripuri)

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<pthread.h>
#include<string.h>

int b = 1;
int kb = 1024;
int mb = 1024 * 1024;
int gb = 1024 * 1024 * 1024;
//long long f_size = 10737418240ULL;
long long f_size = 1024 * 1024 * 1024;
char *buff;
char types[5][20] = {"8 Byte Block", "8 Kilo Bytes Block", "8 Mega Bytes Block", "80 Mega Bytes Block"};
int blocks[] = {8*1, 8*1024, 8*1048576, 80*1048576};

int threads[] = {1, 2, 4, 8};
clock_t start_time,end_time;



FILE *fp, *fp1;

struct thread_args {
   long long total;
   long block;
   int curr_part;
   FILE *fp;
};
```

```
void *read_write(void *arguments);
void *read(void *arguments);
void write_first(FILE *fp, long long f_size);

void *write_random(void *arguments);
void *read_random(void *arguments);

int main()
{
    fp1 = fopen("test1.txt", "w+");
    write_first(fp1, f_size);
    fclose(fp1);
    pthread_t *pt;
    pt = (pthread_t*)malloc(sizeof(pthread_t)*8);
    int itr;
    for (itr = 0; itr < 4; itr++)
    {
        printf("\n\nSequential Read/Write for %s\n\n", types[itr]);
        int t_iter;

        for (t_iter = 0; t_iter < 4; t_iter++)
        {
            if(itr == 0)
            {
                int i;
                //printf("Throughput ");
                struct thread_args args;
                long long trueTotal = 8*kb;
                args.total = trueTotal/threads[t_iter];
                args.block = blocks[itr];
                fp = fopen("test.txt", "w+");
                args.fp = fp;
                start_time = clock();
                int k = 0;
                for (k=0; k< 10; k++)
                {
                    for (i=0; i< threads[t_iter]; i++)
                {
                    args.curr_part = i;
                    pthread_create(&pt[i],NULL, &read_write, (void *)&args);
                }
                for (i=0; i< threads[t_iter]; i++)
                {
                    pthread_join(pt[i],NULL);
                }
                }

                end_time = clock();
                fclose(fp);
                double time_taken = (double)(end_time - start_time) / 10;
                printf("The Latency for Read+Write with %d thread(s): %.3f ms\n", 
threads[t_iter], time_taken/1000);
            }
            else
            {
                int i;
                //printf("Throughput ");
                struct thread_args args;
                long long trueTotal = f_size;
                args.total = trueTotal/threads[t_iter];
                args.block = blocks[itr];
                fp = fopen("test.txt", "w+");
                args.fp = fp;
                start_time = clock();
                for (i=0; i< threads[t_iter]; i++)
                {
                    args.curr_part = i;
                    pthread_create(&pt[i],NULL, &read_write, (void *)&args);
                }
                for (i=0; i< threads[t_iter]; i++)
                {
                    pthread_join(pt[i],NULL);
                }
                end_time = clock();
                fclose(fp);
                //double speed = (double)(threads[t_iter] *
(args.total/args.block))/mb/((end_time - start_time) / CLOCKS_PER_SEC);
                double time_taken = (double)(end_time - start_time)/
CLOCKS_PER_SEC;
                double speed = (trueTotal/mb) / time_taken ;
                printf("The Throughput for Read+Write with %d thread(s): %.2f
Mbps\n", threads[t_iter], speed);

                fp = fopen("test1.txt", "r");
                args.fp = fp;
                start_time = clock();
                for (i=0; i< threads[t_iter]; i++)
                {
                    args.curr_part = i;
                    pthread_create(&pt[i],NULL, &read, (void *)&args);
                }
                for (i=0; i< threads[t_iter]; i++)
                {
                    pthread_join(pt[i],NULL);
                }
                end_time = clock();
                fclose(fp);
                time_taken = (double)(end_time - start_time) / CLOCKS_PER_SEC;
                speed = (double) (trueTotal/mb) / time_taken ;
                printf("The Throughput for Read  with %d thread(s): %.2f Mbps\n\n", threads[t_iter], speed);
            }
        }
    }

    for (itr = 0; itr < 4; itr++)
    {
        printf("Random Read for %s\n\n", types[itr]);
        int t_iter;

        for (t_iter = 0; t_iter < 4; t_iter++)
        {
            if(itr == 0)
            {
```

```
int i;
//printf("Throughput ");
struct thread_args args;
long trueTotal = 8*kb;
args.total = trueTotal/threads[t_iter];
args.block = blocks[itr];

fp = fopen("test1.txt", "r");
args.fp = fp;
start_time = clock();
int k = 0;
for (k=0; k< 10; k++)
{
        for (i=0; i< threads[t_iter]; i++)
{
        args.curr_part = i;
        pthread_create(&pt[i],NULL, &read_random, (void *)&args);
}
for (i=0; i< threads[t_iter]; i++)
{
        pthread_join(pt[i],NULL);
}
}

end_time = clock();
fclose(fp);
double time_taken1 = (double)(end_time - start_time) / 10;
printf("The Latency for Read  with %d thread(s): %.3f ms\n", threads[t_iter], time_taken1/1000);
```

```
            }
            else
            {
                int i;
                //printf("Throughput ");
                struct thread_args args;
                long trueTotal = f_size;
                args.total = trueTotal/threads[t_iter];
                args.block = blocks[itr];

                fp = fopen("test1.txt", "r");
                args.fp = fp;
                start_time = clock();
                for (i=0; i< threads[t_iter]; i++)
                {
                    args.curr_part = i;
                    pthread_create(&pt[i],NULL, &read_random, (void *)&args);
                }
                for (i=0; i< threads[t_iter]; i++)
                {
                    pthread_join(pt[i],NULL);
                }
                end_time = clock();
                fclose(fp);
                double time_taken = (double)(end_time - start_time) /
CLOCKS_PER_SEC;
                double speed = (double) (trueTotal/mb) / time_taken ;
                printf("The Throughput for Read  with %d thread(s): %.2f Mbps\n\n",
threads[t_iter], speed);
```

```
            }
        }
    }
    return 0;
}

void write_first(FILE* fp, long long f_size)
{
    long block = (100 * mb);
    char *item = NULL;
    item = malloc (block * sizeof *item);
    long long i;
    int p_size = (int) f_size / block;
    for(i = 0; i < p_size; i++)
    {
        fseek(fp, i * block,SEEK_SET);
    fwrite(item,block,1,fp);
    }
    free(item);

}

void *read_write(void *arguments)
{
    struct thread_args *args = (struct thread_args *)arguments;
    long total = args -> total;
    long block = args -> block;
    int curr_part = args -> curr_part;
    FILE *fp = args -> fp;
```

```
        char *item = NULL;
        FILE *fp1;
        fp1 = fopen("test1.txt", "r");
        item = malloc (block * sizeof *item);
        //fp = fopen("test.txt", "w+");
        long i;
        long iter = total/block;
        for(i=0; i < iter; i++)
        {
                fseek(fp1,((curr_part * total) + (i*block)),SEEK_SET);
        fread(item,block,1,fp1);
                fseek(fp,((curr_part * total) + (i*block)),SEEK_SET);
        fwrite(item,block,1,fp);
        }
        //fclose(fp);
        free(item);
}

void *read(void *arguments)
{
        struct thread_args *args = (struct thread_args *)arguments;
        long total = args -> total;
        long block = args -> block;
        int curr_part = args -> curr_part;
        FILE *fp = args -> fp;
        char *item = NULL;
        item = malloc (block * sizeof *item);
        //char item[block];
        //fp = fopen("test.txt", "r");
```

```
    long i;
    long iter = total/block;
    for(i=0; i < iter; i++)
    {
        fseek(fp,((curr_part * total) + (i*block)),SEEK_SET);
    fread(item,block,1,fp);
    }
    //fclose(fp);
    free(item);
}

void *write_random(void *arguments)
{
    struct thread_args *args = (struct thread_args *)arguments;
    long total = args -> total;
    long block = args -> block;
    int curr_part = args -> curr_part;
    FILE *fp = args -> fp;
    char *item = NULL;
    item = malloc (block * sizeof *item);
    //char item[block];
    //fp = fopen("test.txt", "w+");
    long i;
    long iter = total/block;
    for(i=0; i < iter; i++)
    {
        long rand_num = rand() % iter;
        fseek(fp,((curr_part * total) + (rand_num*block)),SEEK_SET);
    fwrite(item,block,1,fp);
    }
    //fclose(fp);
    free(item);
}

void *read_random(void *arguments)
{
    struct thread_args *args = (struct thread_args *)arguments;
    long total = args -> total;
    long block = args -> block;
    int curr_part = args -> curr_part;
    FILE *fp = args -> fp;
    char *item = NULL;
    item = malloc (block * sizeof *item);
    //char item[block];
    //fp = fopen("test.txt", "r");
    long i;
    long iter = total/block;
    for(i=0; i < iter; i++)
    {
        long rand_num = rand() % iter;
        fseek(fp,((curr_part * total) + (rand_num*block)),SEEK_SET);
    fread(item,block,1,fp);
    }
    //fclose(fp);
    free(item);
}
```

Network Benchmarking (Abhishek Vijhani)

```
#include <iostream>
#include <netdb.h>
#include <stdio.h>
#include <sys/types.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>    /* Contains constants and structures needed for internet domain addresses. */
#include <stdlib.h>
#include <thread>
#include <chrono>
#include <arpa/inet.h>


using namespace std;

typedef std::chrono::duration<long, std::ratio<1,1000>> millisecs;   /* 2 Milliseconds */
template <typename T>
long duration(std::chrono::time_point<T> time)
{
  auto differance = std::chrono::system_clock::now() - time;
  return std::chrono::duration_cast<millisecs>( differance ).count();
}

void error(const char *msg)
{
  perror(msg);
  exit(1);
```

```
}



int TCPserver(int iterations, int portnum)
{
        int buffsize = 64000;
  socklen_t clilen;
  char buffer[64000];
  struct sockaddr_in serv_addr, cli_addr;

  int sockfd = socket(AF_INET, SOCK_STREAM, 0);
  if (sockfd < 0)
  {
    error("ERROR Opening Socket");
  }
  bzero((char *) &serv_addr, sizeof(serv_addr));
  int portno = portnum;
  serv_addr.sin_family = AF_INET;
  serv_addr.sin_addr.s_addr = INADDR_ANY;
  serv_addr.sin_port = htons(portno);
  /* Bind the socket to an address using the bind() system call.*/
  if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
  {
    error("Binding Error");
  }
  /* Listen for connections with the listen() system call*/
  listen(sockfd,5);
  clilen = sizeof(cli_addr);
   /* Accept a connection with the accept() system call*/
  int newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
  if (newsockfd < 0)
  {
    error("Acception Error");
  }
  bzero(buffer,buffsize);
  int i=0;
  while (i < iterations)
  {
  int n = read(newsockfd,buffer,buffsize);
    if (n < 0)
              {
                            error("Reading from Socket Error");
              }
              i =  i + 1;
  }

  close(newsockfd);
  close(sockfd);
  return 0;
}



int TCPclient(int iterations, int portnum)
{
  struct sockaddr_in serv_addr;
  struct hostent *server;
  int buffsize = 64000;
  char buffer[64000];
  int portno = portnum;
  /* Parameters = address domain of the socket, second argument is the type of socket, The third
argument is the protocol. */
  int sockfd = socket(AF_INET, SOCK_STREAM, 0);
  if (sockfd < 0)
  {
    error("Error on Opening the Socket");
  }
  server = gethostbyname("127.0.0.1");
  if (server == NULL)
  {
    fprintf(stderr,"ERROR, There is no such Host\n");
    exit(0);
  }
            /* The function bzero() sets all values in a buffer to zero. It takes two arguments, the first is a
pointer to the buffer and the second is the size of the buffer.*/
  bzero((char *) &serv_addr, sizeof(serv_addr));

  serv_addr.sin_family = AF_INET;

  bcopy((char *)server->h_addr, (char *)&serv_addr.sin_addr.s_addr, server->h_length);

  serv_addr.sin_port = htons(portno);


  if (connect(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
  {
    error("Error in Connection");
  }
```

```
 /*send first message to server to initialize socket on that end   */
 int n = write(sockfd,"Abhishek",8);
 cout << "Initialized the Socket" << endl;

 bzero(buffer,buffsize);
 int i = 0;
 while (i<buffsize)
 {
  buffer[i]='A' + random()%26;
       i = i+1;
 }

 i = 0;
 while (i<iterations)
 {
  n = write(sockfd,buffer,buffsize);
              if (n < 0)
              {
               error("Writing to Socket Error");
              }
       i=i+1;
 }

 bzero(buffer,buffsize);

 close(sockfd);
 return 0;
}
```

```
int UDPserver(int iterations, int portnum)
{
  int buffsize = 64000;
  int newsockfd;
  socklen_t clilen;
  char buffer[64000];
  struct sockaddr_in serv_addr, cli_addr;
  int sockfd = socket(AF_INET, SOCK_DGRAM, 0);
  if (sockfd < 0)
  {
    error("ERROR opening socket");
  }
  bzero((char *) &serv_addr, sizeof(serv_addr));
  int portno = portnum;
  serv_addr.sin_family = AF_INET;
  serv_addr.sin_addr.s_addr = INADDR_ANY;
  serv_addr.sin_port = htons(portno);
   /* Bind the socket to an address using the bind() system call.*/
  if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
  {
    error("ERROR on binding");
  }
   /* Listen for connections with the listen() system call*/
  listen(sockfd,5);
  clilen = sizeof(cli_addr);

  bzero(buffer,buffsize);
```

```
 int i = 0;
 while (i<iterations)
 {
  int n = recv(sockfd,buffer,buffsize,MSG_WAITALL);
  if (n < 0)
       {
              error("ERROR reading from socket");
       }
       i = i + 1;
 }

 close(newsockfd);
 close(sockfd);
 return 0;
}



int UDPclient(int iterations, int portnum)
{
       int buffsize = 64000;
 struct sockaddr_in serv_addr;
 struct hostent *server;

 char buffer[64000];
 int portno = portnum;
 int sockfd = socket(AF_INET, SOCK_DGRAM, 0);
 if (sockfd < 0)
 {
  error("ERROR opening socket");
```

```
}
server = gethostbyname("127.0.0.1");
if (server == NULL)
{
  fprintf(stderr,"ERROR, no such host\n");
  exit(0);
}
bzero((char *) &serv_addr, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
bcopy((char *)server->h_addr, (char *)&serv_addr.sin_addr.s_addr, server->h_length);
serv_addr.sin_port = htons(portno);
if (connect(sockfd,(struct sockaddr *) &serv_addr,sizeof(serv_addr)) < 0)
{
  error("ERROR connecting");
}
cout << "socket Has been Initialized" << endl;
bzero(buffer,buffsize);
int i = 0;
while (i<buffsize)
{
  buffer[i]='A' + random()%26;
            i = i + 1;
}
i = 0;
while (i < iterations)
{
  int n = send(sockfd,buffer,buffsize,MSG_CONFIRM);
  if (n < 0)
            {
```

```
      error("ERROR writing to socket");
              }
              i = i + 1;
  }

  bzero(buffer,buffsize);

  close(sockfd);
  return sockfd;
}



int main(int argc, char* argv[])
{
  if (argc<3){
    cout << "Enter ./net Port and  NumberofThreads" << endl;
    return 0;
  }



  int portnum = atoi(argv[1]);
  int numthreads = atoi(argv[2]);
  int buffsize = 64000;
  int iterations = 10000;

  thread threads[numthreads*2];

  if (numthreads==1){
    cout << "TCP 1 thread" << endl;
```

```
threads[0] = thread(TCPserver,(2*iterations),portnum);

auto beforeTCP = std::chrono::system_clock::now();

threads[1] = thread(TCPclient,iterations,portnum);

threads[1].join();

auto time_elapsedTCP = duration(beforeTCP);
cout << "Latency: "<<time_elapsedTCP << "ms" <<endl;
ulong latencyTCP = (iterations/time_elapsedTCP) * 1000 / 1000;
        cout << "---------------------------------------------" << endl;
        cout << "Througput: " <<latencyTCP*buffsize / 8000 << "Mbps" << endl;

threads[0].join();

cout << "Waiting 10 sec before UDP..." << endl;
sleep(10);
cout << "UDP 1 Thread" << endl;

threads[0] = thread(UDPserver,iterations,portnum);

auto beforeUDP = std::chrono::system_clock::now();

threads[1] = thread(UDPclient,iterations,portnum);

threads[1].join();
```

```
  auto time_elapsedUDP = duration(beforeUDP);
          cout << "Latency: "<<time_elapsedUDP << "ms" <<endl;
  ulong latencyUDP = (iterations/time_elapsedUDP) * 1000 / 1000;
          cout << "----------------------------------------" << endl;
  cout << "Througput: " <<latencyUDP*buffsize / 8000 << "Mbps" << endl;exit(0);


  threads[0].join();
          exit(0);
}
  else if(numthreads==2){
  cout << "TCP 2 Threads" <<endl;


  threads[0] = thread(TCPserver,iterations,portnum);
  threads[1] = thread(TCPserver,iterations,portnum+1);


  auto beforeTCP = std::chrono::system_clock::now();


  threads[2] = thread(TCPclient,(iterations/2),portnum);
  threads[3] = thread(TCPclient,(iterations/2),portnum+1);


  threads[3].join();
  threads[2].join();


  auto time_elapsedTCP = duration(beforeTCP);
  cout << "Latency: "<<time_elapsedTCP << "ms" <<endl;
  ulong latencyTCP = (iterations/time_elapsedTCP) * 1000 / 1000;
          cout << "----------------------------------------" << endl;
          cout << "Througput: " <<latencyTCP*buffsize / 8000 << "Mbps" << endl;
  threads[1].join();
  threads[0].join();

  cout << "Waiting 15 sec before UDP in attempt to prevent system from refusing connection" << endl;
  sleep(15);
  cout << "UDP 2 Threads" << endl;

  threads[0] = thread(UDPserver,iterations,portnum);
  threads[1] = thread(UDPserver,iterations,portnum+1);

  auto beforeUDP = std::chrono::system_clock::now();

  threads[2] = thread(UDPclient,(iterations/2),portnum);
  threads[3] = thread(UDPclient,(iterations/2),portnum+1);

  threads[3].join();
  threads[2].join();

  auto time_elapsedUDP = duration(beforeUDP);
          cout << "Latency: "<<time_elapsedUDP << "ms" <<endl;
  ulong latencyUDP = (iterations/time_elapsedUDP) * 1000 / 1000;
          cout << "---------------------------------------------" << endl;
  cout << "Througput: " <<latencyUDP*buffsize / 8000 << "Mbps" << endl; exit(0);



  threads[1].join();
  threads[0].join();
          exit(0);
}
```

```
else if(numthreads==4){
  cout << "TCP 4 Threads" <<endl;


  threads[0] = thread(TCPserver,iterations,portnum);
  threads[1] = thread(TCPserver,iterations,portnum+1);
          threads[2] = thread(TCPserver,iterations,portnum+2);
          threads[3] = thread(TCPserver,iterations,portnum+3);

  auto beforeTCP = std::chrono::system_clock::now();


  threads[4] = thread(TCPclient,(iterations/4),portnum);
  threads[5] = thread(TCPclient,(iterations/4),portnum+1);
  threads[6] = thread(TCPclient,(iterations/4),portnum+2);
  threads[7] = thread(TCPclient,(iterations/4),portnum+3);

  threads[7].join();
  threads[6].join();
          threads[5].join();
          threads[4].join();

  auto time_elapsedTCP = duration(beforeTCP);
  cout << "Latency: "<<time_elapsedTCP << "ms" <<endl;
  ulong latencyTCP = (iterations/time_elapsedTCP) * 1000 / 1000;
          cout << "------------------------------------------" << endl;
          cout << "Througput: " <<latencyTCP*buffsize / 8000 << "Mbps" << endl;

  threads[3].join();
          threads[2].join();
          threads[1].join();
```

```
threads[0].join();

cout << "Waiting 20 sec before UDP in attempt to prevent system from refusing connection" << endl;
sleep(20);
cout << "UDP 4 Threads" << endl;

threads[0] = thread(UDPserver,iterations,portnum);
threads[1] = thread(UDPserver,iterations,portnum+1);
        threads[2] = thread(UDPserver,iterations,portnum+2);
        threads[3] = thread(UDPserver,iterations,portnum+3);

auto beforeUDP = std::chrono::system_clock::now();

        threads[4] = thread(UDPclient,(iterations/4),portnum);
threads[5] = thread(UDPclient,(iterations/4),portnum+1);
threads[6] = thread(UDPclient,(iterations/4),portnum+2);
threads[7] = thread(UDPclient,(iterations/4),portnum+3);

threads[7].join();
threads[6].join();
        threads[5].join();
        threads[4].join();

auto time_elapsedUDP = duration(beforeUDP);
        cout << "Latency: "<<time_elapsedUDP << "ms" <<endl;
ulong latencyUDP = (iterations/time_elapsedUDP) * 1000 / 1000;
        cout << "-------------------------------------------" << endl;
cout << "Througput: " <<latencyUDP*buffsize / 8000 << "Mbps" << endl;exit(0);
```

```
  threads[3].join();
          threads[2].join();
          threads[1].join();
  threads[0].join();
          exit(0);

}
else{
            cout << "TCP 8 Threads" <<endl;

  threads[0] = thread(TCPserver,iterations,portnum);
  threads[1] = thread(TCPserver,iterations,portnum+1);
          threads[2] = thread(TCPserver,iterations,portnum+2);
          threads[3] = thread(TCPserver,iterations,portnum+3);
          threads[4] = thread(TCPserver,iterations,portnum+4);
          threads[5] = thread(TCPserver,iterations,portnum+5);
          threads[6] = thread(TCPserver,iterations,portnum+6);
          threads[7] = thread(TCPserver,iterations,portnum+7);

  auto beforeTCP = std::chrono::system_clock::now();

  threads[8] = thread(TCPclient,(iterations/8),portnum);
  threads[9] = thread(TCPclient,(iterations/8),portnum+1);
  threads[10] = thread(TCPclient,(iterations/8),portnum+2);
  threads[11] = thread(TCPclient,(iterations/8),portnum+3);
          threads[12] = thread(TCPclient,(iterations/8),portnum+4);
          threads[13] = thread(TCPclient,(iterations/8),portnum+5);
          threads[14] = thread(TCPclient,(iterations/8),portnum+6);
          threads[15] = thread(TCPclient,(iterations/8),portnum+7);

threads[15].join();
threads[14].join();
          threads[13].join();
          threads[12].join();
          threads[11].join();
          threads[10].join();
          threads[9].join();
          threads[8].join();

auto time_elapsedTCP = duration(beforeTCP);
cout << "Latency: "<<time_elapsedTCP << "ms" <<endl;
ulong latencyTCP = (iterations/time_elapsedTCP) * 1000 / 1000;
          cout << "-------------------------------------------" << endl;
          cout << "Througput: " <<latencyTCP*buffsize / 8000 << "Mbps" << endl;

threads[7].join();
          threads[6].join();
          threads[5].join();
          threads[4].join();
          threads[3].join();
          threads[2].join();
          threads[1].join();
threads[0].join();

cout << "Waiting 20 sec before UDP in attempt to prevent system from refusing connection" << endl;
sleep(20);
```

```
cout << "UDP 8 Threads" << endl;


threads[0] = thread(UDPserver,iterations,portnum);

threads[1] = thread(UDPserver,iterations,portnum+1);

        threads[2] = thread(UDPserver,iterations,portnum+2);

        threads[3] = thread(UDPserver,iterations,portnum+3);

        threads[4] = thread(UDPserver,iterations,portnum+4);

        threads[5] = thread(UDPserver,iterations,portnum+5);

        threads[6] = thread(UDPserver,iterations,portnum+6);

        threads[7] = thread(UDPserver,iterations,portnum+7);


auto beforeUDP = std::chrono::system_clock::now();


        threads[8] = thread(UDPclient,(iterations/8),portnum);

threads[9] = thread(UDPclient,(iterations/8),portnum+1);

threads[10] = thread(UDPclient,(iterations/8),portnum+2);

threads[11] = thread(UDPclient,(iterations/8),portnum+3);

        threads[12] = thread(UDPclient,(iterations/8),portnum+4);

        threads[13] = thread(UDPclient,(iterations/8),portnum+5);

        threads[14] = thread(UDPclient,(iterations/8),portnum+6);

        threads[15] = thread(UDPclient,(iterations/8),portnum+7);


threads[15].join();

threads[14].join();

        threads[13].join();

        threads[12].join();

        threads[11].join();

        threads[10].join();

        threads[9].join();
```

```
            threads[8].join();

    auto time_elapsedUDP = duration(beforeUDP);
            cout << "Latency: "<<time_elapsedUDP << "ms" <<endl;
    ulong latencyUDP = (iterations/time_elapsedUDP) * 1000 / 1000;
            cout << "--------------------------------------------" << endl;
    cout << "Througput: " <<latencyUDP*buffsize / 8000 << "Mbps" << endl;exit(0);



    threads[7].join();
            threads[6].join();
            threads[5].join();
            threads[4].join();
            threads[3].join();
            threads[2].join();
            threads[1].join();
    threads[0].join();
            exit(0);
  }
  exit(0);
}
```