Cloud Computing

CS553

Programming Assignment 1

Benchmarking

Student

Nikhil Nere

A20354957

nnere@hawk.iit.edu

Table of Contents

1. Introduction

1.1. Purpose

This document provides a detailed design of the benchmarking application. It provides an overview how benchmarking of different parts of a computer system is done.

1.2. Problem Statement

To benchmark different parts of a computer system like the CPU, Memory and Disk. The benchmarking experiment is to be carried out on AWS t2.micro instance. A program should run different usecases for different parts as below –

CPU Benchmarking – To calculate GFLOPS and GIOPS with varying level of concurrency involving different number of concurrent threads (1 thread, 2 threads and 4 threads). A separate experiment with 4 threads which will run the arithmetic operations for 10 mins and collect the readings per second.

Memory Benchmarking – Measuring the memory speed with different usecases like sequential access, random access, varying block size (1byte, 1KB, 1MB) and varying concurrency (1 thread, 2 thread)

Disk Benchmarking – Measuring the disk speed with different usecases like sequential access, random access, varying block size (1byte, 1KB, 1MB) and varying concurrency (1 thread, 2 thread). Separate experiments for read and write operations.

For each system the benchmarking results are compared with the third party benchmarking tools – linpack for CPU benchmarking, STREAM for memory benchmarking and IOZone for disk benchmarking.

2. Design Overview

2.1. CPU Benchmarking

A C programming is written to calculate the GFLOPS and GIOPS. The program runs addition operations in a loop for definite number of times. The time is calculated for the loop processing. The GFLOPS and IOPS are calculate by using total additions done and the time required to do the total additions.
The code can be run with different levels of concurrency which can be configured by passing command line argument (no of threads as command line argument).

A separate program is written which runs 4 threads who run the addition operations in an infinite loop and record each operation completed in variable. The main thread keeps track of time and at every second reads the variable updated by each thread and kills all the threads after 10 mins collecting total 600 samples.

2.2. Memory Benchmarking

A C program is written which reads a block of data from memory and copies it to the other block of memory using memcpy() function. This operation is done in a loop and which runs for definite number of times. The time is measured that the loop took to execute. The time and total data transferred is used to measure the speed of the memory.
The program accepts command line arguments to configure different usecases. It accepts no of threads, no of iteration the loop runs, block size.
To nullify the cache role, the memory blocks are declared greater than the L1, L2, L3 cache size.

2.3. Disk Benchmarking

A C program is written which writes to the disk and the other part of it reads from the disk. With every usecase a different file is used so that the disk pages are not available in the memory. Which will ensure the actual IO read – write times. The read and write operations with a specific block size are done several times by using a loop which runs for several times and the time required for the loop is recorded.
The different usecases can be configured using command line arguments which include no of threads, block size, no of iterations.

3. Implementation

3.1. CPU Benchmarking

- Loop that performs floating point arithmetic operations.

```
for (i = 0; i < 1000000000; i++){
    add = 0.2 + 0.5 +
    0.2 + 0.2+
    0.2 + 0.5+
    0.2 + 0.4+
    0.2 + 0.8+
    0.2 + 0.2+
    0.2 + 0.6+
    0.2 + 0.4+
    0.2 + 0.9+
    0.2 ;
}
```

- Loop that performs integer arithmetic operations

```
for (i = 0; i < 1000000000; i++){
    add = 43 + 54 +
    62 + 32+
    52 + 55+
    52 + 54+
    52 + 58+
    52 + 52+
    52 + 56+
    52 + 54+
    52 + 59+
    52 ;
}
```

- Loop that creates threads dynamically

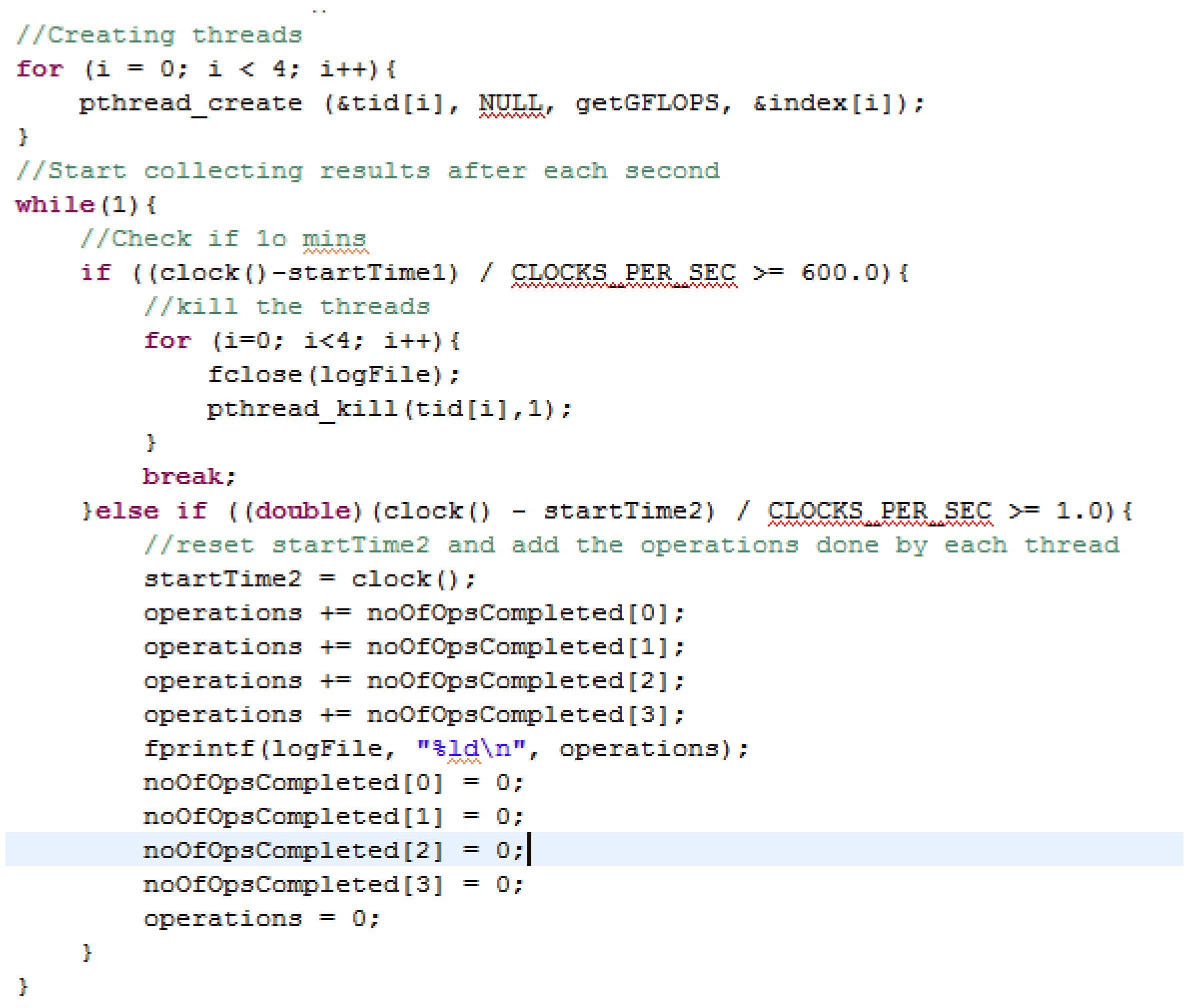
```
for (i = 0; i < noThreads; i++){
    pthread_create (&tid[i], NULL, getGIOPS,NULL);
}
//Waiting for the threads to finish processing
for (i=0; i<noThreads; i++){
    pthread_join(tid[i], NULL);
}
```

- The searchFile method searches for the file in the index

```
ArrayList<String> peerList = fileIndex.get(fileName);
```

3.2. CPU Benchmarking Experiment 2

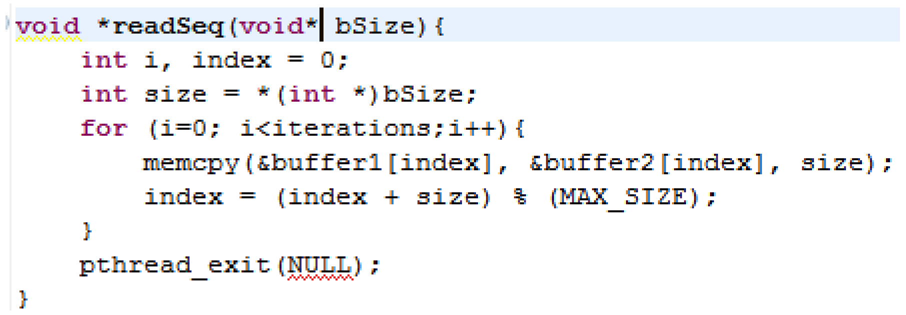
- Below code snippet tells how the experiment 2 is achieved which is responsible for creating 4 independent threads and pulling the number of operations every second and kills the threads after 10 mins.



```
//Creating threads
for (i = 0; i < 4; i++){
    pthread_create (&tid[i], NULL, getGFLOPS, &index[i]);
}
//Start collecting results after each second
while(1){
    //Check if 1o mins
    if ((clock()-startTime1) / CLOCKS_PER_SEC >= 600.0){
        //kill the threads
        for (i=0; i<4; i++){
            fclose(logFile);
            pthread_kill(tid[i],1);
        }
        break;
    }else if ((double)(clock() - startTime2) / CLOCKS_PER_SEC >= 1.0){
        //reset startTime2 and add the operations done by each thread
        startTime2 = clock();
        operations += noOfOpsCompleted[0];
        operations += noOfOpsCompleted[1];
        operations += noOfOpsCompleted[2];
        operations += noOfOpsCompleted[3];
        fprintf(logFile, "%ld\n", operations);
        noOfOpsCompleted[0] = 0;
        noOfOpsCompleted[1] = 0;
        noOfOpsCompleted[2] = 0;
        noOfOpsCompleted[3] = 0;
        operations = 0;
    }
}
```

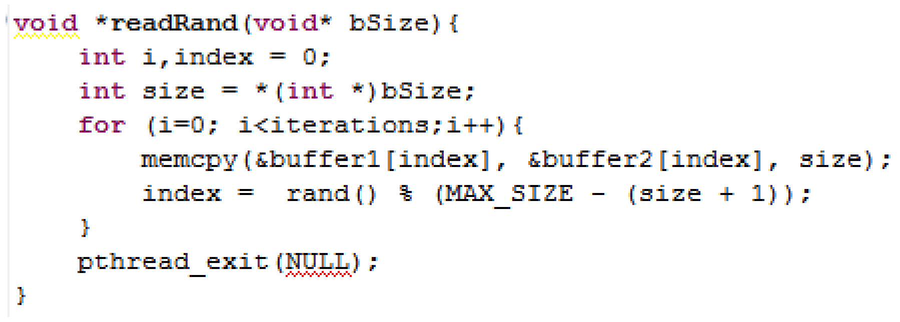
3.3. Memory Benchmarking

- Loop which performs sequential read – write operations on memory for infinite number of times



```
void *readSeq(void* bSize){
    int i, index = 0;
    int size = *(int *)bSize;
    for (i=0; i<iterations;i++){
        memcpy(&buffer1[index], &buffer2[index], size);
        index = (index + size) % (MAX_SIZE);
    }
    pthread_exit(NULL);
}
```

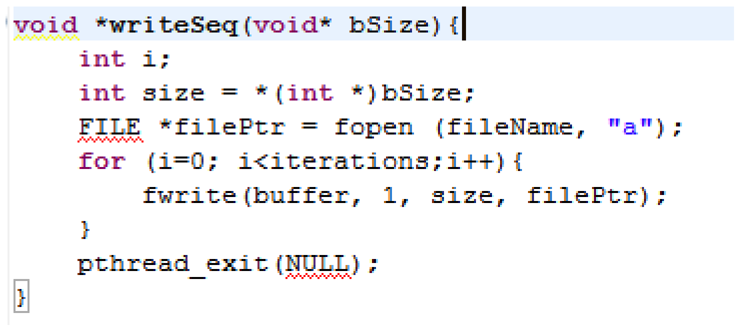
- Loop which performs random read – write operations on memory for infinite number of times



```
void *readRand(void* bSize){
    int i,index = 0;
    int size = *(int *)bSize;
    for (i=0; i<iterations;i++){
        memcpy(&buffer1[index], &buffer2[index], size);
        index =  rand() % (MAX_SIZE - (size + 1));
    }
    pthread_exit(NULL);
}
```

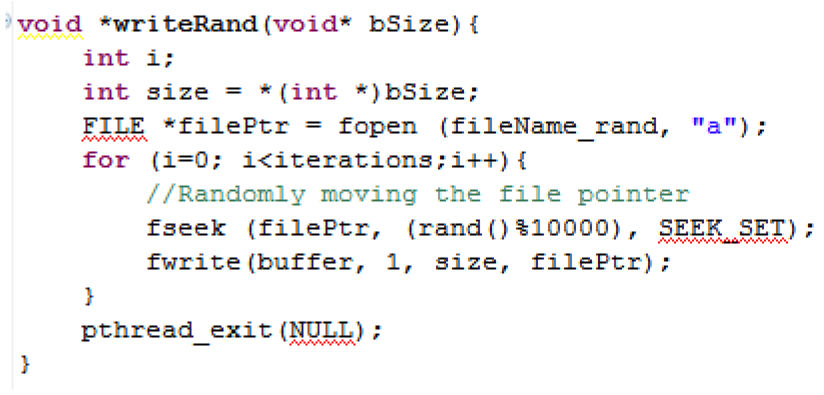
3.4. Disk Benchmarking

- Loop for sequential writes to disk



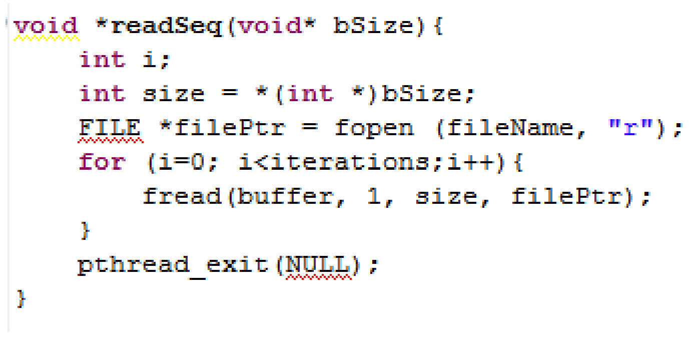
```
void *writeSeq(void* bSize){
    int i;
    int size = *(int *)bSize;
    FILE *filePtr = fopen (fileName, "a");
    for (i=0; i<iterations;i++){
        fwrite(buffer, 1, size, filePtr);
    }
    pthread_exit(NULL);
}
```

- Loop for random writes to disk



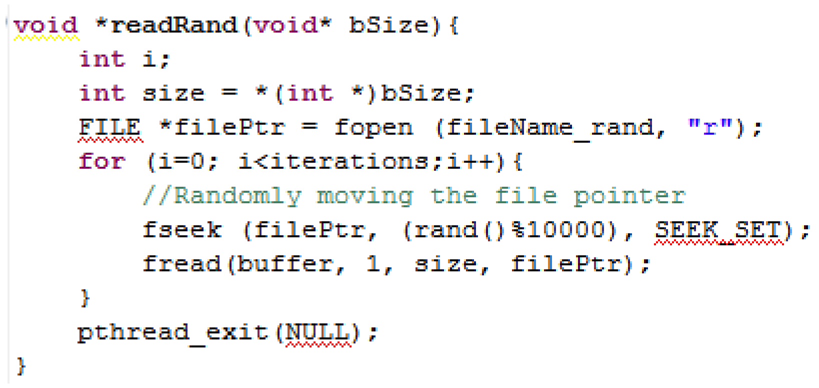
```c
void *writeRand(void* bSize){
    int i;
    int size = *(int *)bSize;
    FILE *filePtr = fopen (fileName_rand, "a");
    for (i=0; i<iterations;i++){
        //Randomly moving the file pointer
        fseek (filePtr, (rand()%10000), SEEK_SET);
        fwrite(buffer, 1, size, filePtr);
    }
    pthread_exit(NULL);
}
```

- Loop for sequential reads to disk



```c
void *readSeq(void* bSize){
    int i;
    int size = *(int *)bSize;
    FILE *filePtr = fopen (fileName, "r");
    for (i=0; i<iterations;i++){
        fread(buffer, 1, size, filePtr);
    }
    pthread_exit(NULL);
}
```

- Loop for random reads to disk



```c
void *readRand(void* bSize){
    int i;
    int size = *(int *)bSize;
    FILE *filePtr = fopen (fileName_rand, "r");
    for (i=0; i<iterations;i++){
        //Randomly moving the file pointer
        fseek (filePtr, (rand()%10000), SEEK_SET);
        fread(buffer, 1, size, filePtr);
    }
    pthread_exit(NULL);
}
```

4. Assumptions and Constraints

- The time required for the other parts of the code are ignored and produce noise in the actual results.
- Since we are working with high level language it is very difficult to get the figures close to the theoretical values.
- The programs give only up to 20% of the theoretical values.
- The benchmarking results are 90% - 50% of the third party benchmarking tools.
- Please go through the performance document for the detail reading comparisons.