3.

Introduction

1.1. Purpose

This document provides an overview of how benchmarking of different parts of computer/system can be carried out.

1.2. Problem Statement

To benchmark different parts of computer as CPU, MEMORY, DISK, NETWORK, etc.

CPU Benchmarking

- Measure the processor speed, in terms of double precision floating point operations per second (Giga FLOPS, 109 FLOPS) and integer operations per second (Giga IOPS, 109 IOPS).
- Measure the processor speed at varying levels of concurrency (1 thread, 2 threads, 4 threads, 8 threads).

Memory Benchmarking

- Measure the memory speed of your host
- Includes read+write operations (e.g. memcpy), sequential write access (e.g. memset), random write access, varying block sizes (8B, 8KB, 8MB, 80MB), and varying the concurrency (1 thread, 2 threads, 4 threads, and 8 threads).
- The metrics to be measured are throughput (MB/sec) and latency (ms); note that the 8B block case can be used to measure latency, and the 8KB, 8MB, and 80MB cases can be used to measure throughput.

Disk Benchmarking

- Measure the disk speed.
- Includes read+write operations, sequential read access, random read access, varying block sizes (8B, 8KB, 8MB, 80MB), and varying the concurrency (1 thread, 2 threads, 4 threads, 8 threads).
- The metrics to be measured are throughput (MB/sec) and latency (ms); note that the 8B block case can be used to measure latency, and the 8KB, 8MB, and 80MB cases can be used to measure throughput.

4.

For these per system the benchmarking results are compared with the standard third party benchmarking tools –

1. CPU benchmarking – Linpack (http://en.wikipedia.org/wiki/LINPACK)
2. Memory Benchmarking – Stream (http://www.cs.virginia.edu/stream/)
3. Disk Benchmarking – Iozone (http://www.iozone.org/)
4. Network Benchmark – Iperf (http://en.wikipedia.org/wiki/Iperf)

Design Overview

2.1. CPU Benchmarking

A C program is implemented for CPU Benchmarking. In order to calculate GFLOPS & IOPS, a definite loop is designed to run number of times that calculates addition, multiplication, etc operations. This execution time is captured for GFLOPS and IOPS.

Number of threads can be passed via command line argument to maintain concurrency.

A separate program is written for AVX instructions utilization. In this, FLOPS and IOPS operations are conducted using AVX instructions to get optimized output.

2.2. Memory Benchmarking

A CPP program is implemented to read block of data from memory and copy to other block of memory using memcpy() construct. Also data is to be written in sequential and random format to a block of memory using memset() construct. All this operation is done in a definite number of loop count. Execution time is measured based on start and end of thread processing. Speed of memory is calculated based on total data transferred.

Command line arguments are used to configure different switch cases. It accepts operation number, no of threads, block size.

2.3. Disk Benchmarking

A C program is implemented for Disk benchmarking, which writes the data to file, reads sequentially and randomly from file. Read and write operation done specific times in a loop based on file size and maintaining of thread concurrency and strong scaling. The different use cases can be configured using command line argument that includes operation, block size, thread and file name to be read.

5.

2.4 Network Benchmarking

A Java program is implemented for Network benchmarking, which sends data via TCP and UDP protocol between Server and Client on loopback address. We are sending 8GB of data from Client to Server with fixed block size of 64KB with thread concurrency and strong scaling. We can config the Client application to send messages to specifying the server IP in command line.

Implementation

3.1. CPU Benchmarking

- Function performing floating point arithmetic operations.

```
void *sum_flops_general(void* arg)
{

    float sum;
    int i;

    for(i=0; i<iterations;i++)
    {

        sum = 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 + 0.12+
        12.0;

    }

    pthread_exit(NULL);
}
```

6.

- Function performing integer arithmetic operations

```
void *sum_iops_general(void* arg)
{
    int sum=1,i;

    for(i=0; i<iterations;i++)
    {
        sum = 43 + 54 +
        62 + 32+
        52 + 55+
        52 + 54+
        52 + 58+
        52 + 52+
        52 + 56+
        52 + 54+
        52 + 59+
        52 + 34+
        23;

    }
    pthread_exit(NULL);

}
```

- Function performing AVX floating point arithmetic operations

```
void *sum_flops_avx(void* arg){
    int i;

    __m256 a = _mm256_set_ps(2.0, 4.0, 6.0, 8.0, 10.0, 12.0, 14.0, 16.0);
    __m256 b = _mm256_set_ps(1.0, 3.0, 5.0, 7.0, 9.0, 11.0, 13.0, 15.0);

    for(i=0; i<iterations;i++)
    {
        __m256 result = _mm256_add_ps(a, b);
        __m256 result_sub = _mm256_sub_ps(a, b);

    }
    pthread_exit(NULL);
}
```

7.

- Function performing AVX integer arithmetic operations

```
void *sum_iops_avx(void* arg)
{
    int i;

    __m128i a = _mm_set_epi32(12, 14, 16, 28);
    __m128i b = _mm_set_epi32(11, 13, 45,17);

    for(i=0; i<iterations;i++)
    {
        __m128i result = _mm_add_epi32(a, b);
        __m128i result_sub = _mm_sub_epi32(a, b);
        __m128i result_mul = _mm_mul_epi32(a, b);
        __m128i result_1 = _mm_add_epi32(a, b);
    }
    pthread_exit(NULL);
}
```

- Dynamic thread creation

```
for (i = 0; i < number_of_threads[k]; i++){
    pthread_create (&pthread_1[i], NULL, sum_flops_general,NULL);
}

for (i=0; i<number_of_threads[k]; i++){
    pthread_join(pthread_1[i], NULL);
}
```

3.2. Memory Benchmarking

- Read Write to memory

```
//MEMCPY
void *seq_read_write(void* arg) {

    long param = (long) arg;

    char* mem_source = new char[iterations];
    char* mem_dest = new char[iterations];

    long index;

    for (index = 0; index < param; index++) {

        memcpy(mem_dest + index * BLOCK_SIZE, mem_source + index * BLOCK_SIZE,BLOCK_SIZE);
    }
    pthread_exit(NULL);

}
```

- Sequential write to memory

```
void *seq_write(void* arg) {
    long param = (long) arg;

    char* mem_dest = new char[iterations];
    long index;

    for (index = 0; index < param; index++) {

        memset(mem_dest + index * BLOCK_SIZE,'1',BLOCK_SIZE);
    }
    pthread_exit(NULL);
}
```

- Random writes to memory

```
void *random_write(void* arg) {
    long param = (long) arg;

    char* mem_dest = new char[iterations];
    long index;

    for (index = 0; index < param; index++) {

        int random_count = rand() % param;

        memset(mem_dest + random_count * BLOCK_SIZE,'1',BLOCK_SIZE);
    }
    pthread_exit(NULL);
}
```

3.3. Disk Benchmarking

- Read Write to disk

```
void *seq_read_write(void* arg) {

    long param = (long) arg;

    FILE *fptr = fopen("new_file.txt", "w");
    FILE *fptr_read = fopen(file_name, "r");
    int index;
    long limit_new;

    long denominator = BLOCK_SIZE * number_of_threads;

    limit_new = file_size / denominator;

    fseek(fptr_read, param, SEEK_SET);
    fseek(fptr, param, SEEK_SET);

    for (index = 0; index < limit_new; index++) {
        fread(total_buffer,1,BLOCK_SIZE, fptr_read);
    }

    for (index = 0; index < limit_new; index++) {
        fwrite(total_buffer, 1,BLOCK_SIZE, fptr);
    }

    fclose(fptr);
    pthread_exit(NULL);

}
```

- Sequential read to disk

```
void *seq_read(void* arg) {
    long param = (long) arg;

    FILE *fptr_read = fopen(file_name, "r");
    int index;
    long limit_new;

    long denominator = BLOCK_SIZE * number_of_threads;

    limit_new = file_size / denominator;

    fseek(fptr_read, param, SEEK_SET);

    for (index = 0; index < limit_new; index++) {
        fread(total_buffer, 1, BLOCK_SIZE, fptr_read);
    }

    fclose(fptr_read);
    pthread_exit(NULL);
}
```

- Random writes to disk

```
void *random_read(void* arg) {

    long param = (long) arg;

    FILE *fptr_read = fopen(file_name, "r");
    int index;
    long limit_new;

    long denominator = BLOCK_SIZE * number_of_threads;

    limit_new = file_size / denominator;

    int random_count = rand() % 100;

    fseek(fptr_read, random_count, SEEK_SET);

    for (index = 0; index < limit_new; index++) {
        fread(total_buffer, 1, BLOCK_SIZE, fptr_read);
    }
```

3.3. Network Benchmarking

- TCP Client

```
class TCPClientWorker implements Runnable {

    Socket clientSocket;
    int threadCount;
    String IPAddress;
    int port;

    public TCPClientWorker(String IPAddress, int port, int threadCount) {
        this.IPAddress = IPAddress;
        this.port = port;
        this.threadCount = threadCount;
    }

    @Override
    public void run() {

        ObjectOutputStream toServer = null;
        ObjectInputStream fromServer = null;

        try {
            clientSocket = new Socket(IPAddress, port);
            toServer = new ObjectOutputStream(clientSocket.getOutputStream());
            fromServer = new ObjectInputStream(clientSocket.getInputStream());

            byte[] message = new byte[65536];
            byte[] reply = new byte[65536];
            Arrays.fill(message, (byte) 1);

            long dataSizeInBytes = 8589934592L;
            long limit = dataSizeInBytes / (65536 * threadCount);
            for (long index = 0; index < limit; index++) {
                try {
                    toServer.writeObject(message);
                    try {
                        reply = (byte[]) fromServer.readObject();
                    } catch (ClassNotFoundException e) {
                        e.printStackTrace();
                    }
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }

        } catch (IOException e) {
            e.printStackTrace();
```

- TCP Server

```
    public static void main(String args[]) throws Exception {

        int port = 8080;
        ServerSocket myServer = new ServerSocket(port);
        System.out.println("Server started at port: " + port);
        while (true) {
            Socket connSocket = myServer.accept();
            TCPServerWorker worker = new TCPServerWorker(connSocket);
            Thread thread = new Thread(worker);
            thread.start();
        }
    }
}

class TCPServerWorker implements Runnable {

    Socket socket;

    public TCPServerWorker(Socket socket) {
        this.socket = socket;
    }

    @Override
    public void run() {

        try {
            ObjectInputStream fromClient = new ObjectInputStream(socket.getInputStream());
            ObjectOutputStream toClient = new ObjectOutputStream(socket.getOutputStream());
            byte[] inputData = new byte[65536];

            while (true) {
                try {
                    inputData = (byte[]) fromClient.readObject();
                    toClient.writeObject(inputData);
                } catch (EOFException e) {
                    break;
                } catch (ClassNotFoundException e) {
                    e.printStackTrace();
                }
                inputData = null;
            }

        } catch (IOException e) {
            e.printStackTrace();
        }
```

- UDP Client

```
class UDPClientWorker implements Runnable {

    DatagramSocket clientSocket;
    int threadCount;
    InetAddress IPAddress;
    byte[] sendMessage;
    byte[] receiveMessage;

    public UDPClientWorker(String ipAddress, int threadCount) throws UnknownHostException {
        this.IPAddress = InetAddress.getByName(ipAddress);
        this.threadCount = threadCount;
    }

    @Override
    public void run() {

        sendMessage = new byte[64512];
        receiveMessage = new byte[64512];
        long dataSizeInBytes = 8589934592L;
        long limit = dataSizeInBytes / (64512 * threadCount);

        Arrays.fill(sendMessage, (byte) 1);

        try {
            clientSocket = new DatagramSocket();
        } catch (SocketException e) {
            e.printStackTrace();
        }

        for (long index = 0; index < limit; index++) {
            DatagramPacket sPacket = new DatagramPacket(sendMessage, sendMessage.length, IPAddress, 6666);
            try {
                clientSocket.send(sPacket);
            } catch (IOException e) {
                e.printStackTrace();
            }
            DatagramPacket rPacket = new DatagramPacket(receiveMessage, receiveMessage.length);
            try {
                clientSocket.receive(rPacket);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        clientSocket.close();
    }
```

- UDP Server

```
class UDPServerWorker implements Runnable {
    DatagramSocket serverSocket;
    byte[] receiveMessage;
    byte[] sendMessage;

    public UDPServerWorker() throws SocketException {
        this.serverSocket = new DatagramSocket(6666);
    }

    @Override
    public void run() {

        while (true) {
            receiveMessage = new byte[64512];
            sendMessage = new byte[64512];

            DatagramPacket rPacket = new DatagramPacket(receiveMessage, receiveMessage.length);
            try {
                serverSocket.receive(rPacket);
            } catch (IOException e) {
                e.printStackTrace();
            }
            sendMessage = rPacket.getData();

            InetAddress clientIPAddress = rPacket.getAddress();
            int clientPort = rPacket.getPort();
            DatagramPacket sPacket = new DatagramPacket(sendMessage, sendMessage.length, clientIPAddress, clientPort);
            try {
                serverSocket.send(sPacket);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

}
```

4. Assumptions and Constraints

- Actual results are the noise as time required for other parts of the code is ignored
- It's very hard to achieve same results as theoretical values, as we are implementing benchmarks in high level programming languages.
- Program achieves 20 – 50% of the theoretical performance for respective benchmarks.
- Benchmarking results are close to standard benchmarking tools