DC Project

N.A.M.E

April 17, 2024

1 Primary Objective

The primary objective of the **Multi-threaded Matrix Benchmark project** is to create a comprehensive bench-marking tool capable of stressing both the **CPU** and **RAM** of a computing system.

By employing **multi-threading techniques** and **simulating intensive** operations on matrices, the project seeks to emulate real-world scenarios where applications perform concurrent computations and I/O operations, thereby providing valuable insights into system performance and resource utilization.

2 Key Features

2.1 Multi-threaded CPU Stress Testing

The project leverages **multi-threading** to simulate **CPU-intensive tasks**, with each **thread** operating on its own matrix **instance**. By executing concurrent computations across multiple **threads**, the benchmark assesses the system's ability to handle parallel processing and **thread synchronization effectively**.

2.2 I/O Operations Simulation

In addition to **CPU stress testing**, the project incorporates **threads** for simulating **I/O operations** such as **reading** from and **writing** to disk. This aspect of the benchmark enables the evaluation of system performance under scenarios involving both **computational** and **I/O-bound tasks**, providing a holistic perspective on resource utilization.

2.3 User-Input Parameters

The benchmark offers flexibility through **User-Input Parameters**, allowing users to specify matrix size, the number of threads, and the intensity of operations. This enables **tailored benchmarking scenarios** to suit diverse use cases and system configurations.

3 Programming

The project will be done in the **java** programming language and it will have the structure of a program / app , where the user can launch the bench-marking tool and test **their machine** in various ways.

3.1 Program Baseline

The baseline for the benchmark package will establish an interface serving as a point of abstraction, allowing clients to interact with the benchmark functionality through simplified method calls such as **run()** and **initialize()**.

This approach aims to shield the internal implementation details of the benchmark features, providing clients with a clear and intuitive interface while concealing complexities. The Timer class encapsulates methods for initiating and halting the timer, as well as retrieving the elapsed time. Upon invocation of the **start()** method, the timer records the current system time in nanoseconds as the starting point.

Subsequently, the **stop()** method captures the current system time as the endpoint, enabling the calculation of the elapsed time. Finally, the **getElapsedTime()** method facilitates access to the duration of the timer's execution, expressed in nanoseconds.

Listing 1: Program Baseline

```java
import java.util.Random;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class MemoryAllocator {

    public static void main(String[] args) {
        int size = 10; // specify the size in MB
        int threads = 4; // number of threads

        ExecutorService executor = Executors.newFixedThreadPool(threads);

        for (int i = 0; i < threads; i++) {
            executor.execute(new MemoryAllocationTask(size));
        }

        // Shutdown the executor after all tasks are completed
        executor.shutdown();
    }

    static class MemoryAllocationTask implements Runnable {
        private final int size;

        MemoryAllocationTask(int size) {
            this.size = size;
        }

        @Override
        public void run() {
            allocateMemory(size);
        }

        private void allocateMemory(int size) {
            byte[] array = new byte[size * 1024 * 1024];
            // Allocate memory in bytes

            Random random = new Random();

            for (int i = 0; i < array.length; i++) {
                array[i] = (byte) random.nextInt(256);
                // Fill memory with random data
            }
        }
    }
}
```