Final Project Report: Parallel Monte Carlo Simulation for Estimating π

Course: Concurrent Programing

Instructor: Dr. Mohammed Aoudi

Team Members:

- **Hasan Hijazi** (ID: 5922)
- **Morhaf Najjar** (ID: 5980)

---

1. Introduction

This project aims to estimate the value of π using a Monte Carlo simulation. The core concept involves randomly generating points in a 2D space and checking how many fall inside a quarter circle. This project implements both sequential and parallel versions of the simulation and presents the results through a user-friendly JavaFX interface.

---

2. Problem Statement & Domain

- **Domain:** Monte Carlo Simulations
- **Problem:** Estimating π by simulating random point generation inside a unit square and counting the proportion of points falling inside a quarter circle.
- **Formula:**
 π≈4×Points inside quarter circleTotal points generated\pi \approx 4 \times \frac{\text{Points inside quarter circle}}{\text{Total points generated}}π≈4×Total points generatedPoints inside quarter circle

---

3. Why This Problem Has Parallelism Potential

This problem is ideal for parallelization because:

- Each random point is generated and evaluated independently.
- No dependencies or communication are required between iterations.
- Minimal synchronization needed—perfect for multi-core CPUs.
- Allows nearly linear speedup as cores increase.

---

4. Project Goals and Metrics

- **Target Speed-up:** ≥ 3× on an 8-core CPU
- **CPU Utilization:** ≥ 85%
- **Memory Overhead:** ≤ 2×
- **Tools Used for Monitoring:** Java Flight Recorder, VisualVM, CSV plotting
- **Measurement:** Execution time and CPU usage in both sequential and parallel runs.

---

5. Sequential Algorithm Overview

- Generate `N` random (x, y) points in range [0, 1].
- Count how many satisfy x2+y2≤1x^2 + y^2 \leq 1x2+y2≤1.
- Estimate π using the formula above.
- Measure time using `System.nanoTime()`.

---

6. Parallel Strategy

- Use `ExecutorService` with a fixed thread pool based on available CPU cores.
- Each thread:
    - Generates a batch of random points using `ThreadLocalRandom`.
    - Calculates local hit count.
- Use `LongAdder` for safe and efficient result aggregation.
- Synchronization handled using `CountDownLatch`.

---

7. UI Design Using JavaFX

A JavaFX GUI was created to allow:

- Selection between sequential or parallel simulation modes.
- A "Run" button to initiate the simulation.
- A "Refresh" button to reset results.
- Labels to show:
    - Estimated value of π.
    - Response time in milliseconds.

UI Layout:

- ToggleGroup with two RadioButtons (Sequential, Parallel)
- Two Buttons (Run, Refresh)

- Two Labels (Result, Response Time)
- Layout managed by VBox with spacing and padding.

---

8. Java Code Summary

Main Class: `MonteCarloPiSimulator.java`

Key Components:

Sequential Path:

java
CopyEdit
```
for (int i = 0; i < NUM_POINTS; i++) {
    double x = rand.nextDouble();
    double y = rand.nextDouble();
    if (x * x + y * y <= 1.0) inside++;
}
```

Parallel Path:

- Detect available CPU cores.
- Create thread pool.
- Submit tasks to compute local hit counts.
- Use `ThreadLocalRandom` to avoid contention.
- Use `LongAdder` and `CountDownLatch` for efficient aggregation and synchronization.

Performance Timing:

- Execution time calculated using:

java
CopyEdit

```
long start = System.nanoTime();
// ... computation ...
long end = System.nanoTime();
```

- Time displayed in the UI:

java
CopyEdit
```
timeLabel.setText(String.format("Response Time: %.3f ms", durationNano
/ 1_000_000.0));
```

---

9. CPU Utilization & Performance

- **CPU Usage in Parallel Mode:** Achieved over 85% usage during peak execution, confirmed via VisualVM.
- **Speed-up Achieved:** Nearly 4x faster on an 8-core CPU compared to the sequential version.
- **Memory Usage:** Kept under 2× increase, as per project constraints.
- **Response Time:** Reduced significantly in parallel execution, dropping from several seconds to under one second for 10 million points.

---

10. Timeline Overview

Week	Milestone
1	Proposal submission
2	Sequential implementation
3	Parallel implementation
4	Profiling & optimization
5	Final report writing

6	Demo preparation & peer review

---

11. Conclusion

This project successfully demonstrates the power and efficiency of parallel programming for computational simulations. The Monte Carlo method, being embarrassingly parallel, allowed straightforward and effective scaling. The addition of a GUI made the project interactive and intuitive. Both goals—accuracy and performance—were achieved, with thorough profiling and monitoring validating the outcomes.

---

12. Future Enhancements

- GPU-based implementation for even faster performance.
- Adjustable number of points via GUI.
- Real-time plotting of π estimation convergence.