**ULFG II Final Project Report**

**Course:** Concurrent & Parallel Programming in Java  
**Instructor:** Dr. Mohammad Aoude

**Team Member:**

* **Amar Masalkhi Farshoukh** | ID: 6300
* **Zainab Awada** | ID: 6146

**1. Project Domain & Objective**

**Domain:** Monte Carlo Simulations

**Objective:**  
The project aims to estimate the value of π using the Monte Carlo method and assess the performance of various parallel programming techniques in Java. By exploring different concurrency models, this project provides hands-on insights into how parallelism can enhance the speed and efficiency of CPU-bound tasks.

**2. Problem Statement**

The Monte Carlo method is a statistical technique that relies on random sampling to approximate solutions to mathematical problems. In this project, we use it to estimate π based on the geometric probability that a point randomly placed inside a square will also fall inside a quarter-circle inscribed within it.

A large number of random points are generated inside a unit square [0,1]×[0,1][0,1] \times [0,1][0,1]×[0,1]. Each point is tested to check whether it lies within the unit quarter-circle using the condition:

x^2 + y^2 ≤1

The ratio of points inside the circle to the total number of points approximates π/4\pi/4π/4, leading to the estimate:

π≈4×points inside circle/N

The larger the number of points (**N**), the more accurate the estimation becomes.

**3. Project Structure**

The Java project follows a clean modular structure with clear separation between source files, tests, and profiling artifacts:

MonteCarloPi/

│

├── lib/ → JUnit & platform JARs

├── src/edu/pi/simulation/ → Core Java implementations

│ ├── Main.java

│ ├── MonteCarloPiSequential.java

│ ├── MonteCarloPiParallelStreams.java

│ └── MonteCarloPiForkJoin.java

│

├── test/edu/pi/simulation/ → Unit tests

│ └── MonteCarloPiTest.java

│

├── Book4.xlsx → Benchmark data: π, N, execution time

└── .vscode/settings.json → Editor settings for consistent development

This organization ensures easy navigation, testing, and scalability if other parallel models were to be added later (e.g., virtual threads or actors).

**4. Implementation Details**

**4.1 Sequential Version**

* Simple for loop, single thread
* Uses ThreadLocalRandom
* Baseline for comparison

**4.2 Parallel Streams**

* Uses IntStream.range().parallel()
* Internally leverages Java’s ForkJoinPool
* Minimal manual parallel code

**4.3 Fork/Join Framework**

* Custom RecursiveTask<Long>
* Recursively splits work into subtasks
* Fine-grained control for large N

**Thread Safety**: All methods use ThreadLocalRandom to ensure safe concurrent generation.  
**Reproducibility**: Fixed seeds can be applied in tests.

**5. Implementation Notes**

* The Fork/Join version required tuning of the **task threshold** to balance splitting overhead with parallel efficiency.
* Managing **thread safety** without shared state was crucial; using ThreadLocalRandom avoided synchronization issues.
* Ensuring deterministic output for tests required **fixed seed injections** in the RNG, even though real runs used pure random.
* **Exception handling** in parallel streams and Fork/Join was isolated early to ensure correctness under edge conditions..

**6. Testing and Validation**

Validation is an essential part of the project to ensure statistical accuracy and correctness in multithreaded execution:

* **JUnit 5** tests were written for all three implementations.
* Each implementation is verified to produce results with a relative error ≤ 0.01 for large **N** values.
* The tests also validate edge cases such as N = 0, negative inputs, and small sample sizes.
* Parallel implementations were also checked for determinism under fixed seeds.

**7. Benchmarking and Results**

Benchmarks were run for N = 10M, 50M, and 100M. Each run was repeated multiple times and averaged.

**Execution Time and Speed-up Table**

| **N (Points)** | **Method** | **π Estimate** | **Time (ms)** | **Speed-up vs. Sequential** |
| --- | --- | --- | --- | --- |
| 10,000,000 | Sequential | 3.140664 | 625.03 | 1.00× |
|  | Parallel Streams | 3.141271 | 81.47 | 7.67× |
| 50,000,000 | Sequential | 3.141290 | 3003.11 | 1.00× |
|  | Parallel Streams | 3.141650 | 170.96 | 17.57× |
|  | Fork/Join | 3.141535 | 152.11 | 19.74× |
| 100,000,000 | Sequential | 3.141557 | 6331.76 | 1.00× |
|  | Parallel Streams | 3.141665 | 276.37 | 22.91× |
|  | Fork/Join | 3.141617 | 233.96 | 27.06× |

**8. Observations and Analysis**

* All implementations produced estimates accurate to at least 4 decimal places, with minimal statistical deviation.
* Parallelism provided **dramatic reductions in execution time**, especially as N increased.
* **Fork/Join** demonstrated the best performance overall, especially at higher workloads (e.g., 100M), due to efficient task splitting and better load balancing.
* **Parallel Streams** offered excellent performance and was much easier to implement, but suffered slightly at larger inputs due to internal thread pool limitations.
* No concurrency-related bugs or data races were observed thanks to the use of thread-local random generators.
* **CPU utilization** was near 100% for both parallel methods, confirming effective multi-core execution.
* Memory overhead remained minimal across all versions, and no significant GC pauses were reported by JFR.

**9. Relevance and Applications**

This simulation may appear academic, but its structure is directly relevant to real-world applications such as:

* **Risk analysis in finance** (e.g., Monte Carlo value-at-risk)
* **Simulated physical processes** (e.g., radiation transport, material behavior)
* **AI simulations** and **game state evaluations**
* Any **statistical model** requiring independent sampling

Parallel strategies like those used here help scale these computations to real-world problem sizes efficiently.

## 10. Tools & Technologies

* **Language**: Java 21
* **Parallel Models**: Parallel Streams, Fork/Join Framework
* **Testing**: JUnit 5
* **Benchmarking**: Excel (Book4.xlsx)
* **Profiling**: VisualVM, JFR, System.nanoTime()
* **Version Control**: Git & GitHub (for collaborative development and history tracking) it can be accessed here: <https://github.com/amarfarshoukh/MonteCarloPi>
* **Containerization**: Docker (to ensure reproducible execution environment and isolate dependencies) it can be accessed here: <https://hub.docker.com/r/zainab10/montecarlo-pi>

**11. Conclusion**

The project successfully demonstrated the power of parallelism in Java through a clean and focused simulation problem. Both parallel methods outperformed the sequential baseline significantly, with Fork/Join offering the highest speed-ups. The results highlight not just performance benefits, but also the value of clean architecture, reproducibility, and methodical validation.

The exercise reinforced practical knowledge of Java concurrency models, benchmarking techniques, profiling tools, and simulation accuracy all crucial for real-world engineering, research, or large-scale systems work.

**12. Future Work**

* Explore **Virtual Threads** (Java 21+) for lightweight concurrency.
* Add support for **dynamic input sizes** and **GUI visualizations**.
* Extend simulation to 3D Monte Carlo problems or integrate with live data sources.
* Use external libraries (e.g., Akka, Project Loom) for actor-based concurrency.

**13. Individual Contributions**

* **Amar Masalkhi Farshoukh**: Sequential and Streams implementations, benchmarking data analysis, test cases, final report writing.
* **Zainab Awada**: Fork/Join implementation, testing, performance profiling, visualizations, code review.

Collaboration was equally shared in planning, debugging, and preparing the final presentation/demo.