Monte-Carlo Pi Estimator Report

Done by: Alaa Yehia 6432

Ali Jaafar 6411

Presented to: Dr. Mohammad Aoudi

Table of Contents

Table of Figures

GitHub Link

https://github.com/Alaa-F-Yehia/MonteCarloPiEstimator.git

1. **Introduction**

Monte Carlo simulation is a probabilistic method extensively utilized for numerical integration, risk assessment, and addressing deterministic issues through random sampling. Its key advantage is the ease of implementation and its capacity to yield approximate results for intricate mathematical problems that are challenging to solve analytically.

A classic example of this technique is the estimation of π, a mathematical constant prevalent in geometry, physics, and various scientific calculations.

To estimate π using Monte Carlo, one generates a significant number of random points within a unit square and counts how many of those points lie within the inscribed unit circle. The ratio of points that fall inside the circle to the total number of points, multiplied by 4, provides an approximation of π. As the quantity of points increases, the precision of the estimation enhances.

The motivation for this project is to not only implement this estimator but also to assess the performance advantages of parallel computation. With the advent of multi-core processors, leveraging concurrency in performance-sensitive applications is essential. Consequently, this project contrasts a sequential implementation with a parallel one for Monte Carlo π estimation to illustrate how parallelism can decrease computation time while maintaining accuracy. The objective is to develop a clean, efficient, and presentation-ready project that both visualizes and quantifies the benefits of multithreading in numerical simulations.

2. Design — Algorithms, Data Structures, Synchronization

The algorithm randomly generates points inside a unit square and counts how many fall within the unit circle. The ratio approximates π.

- **Algorithm**: Monte Carlo estimation for π
- **Sequential Version**: Single-threaded loop with `java.util.Random`
- **Parallel Version**: Uses Java's `ExecutorService` and Callable<Long> for thread-safe parallel execution.
- **Data Structures**:
    - `List<Double>` for storing execution times
    - `AtomicLong` used in earlier versions for progress reporting (removed for clarity)
- **Synchronization**: `CompletionService` manages futures and aggregates partial results safely

Justification: `ExecutorService` and `Callable` offer a high-level and efficient way to handle parallelism without manually managing threads.

3. Implementation Notes

- **Progress Reporting**: Initially implemented a live progress bar in the console utilizing shared atomic counters. This feature was eliminated to prioritize performance clarity and simplicity.
- **Chart Visualization**: Employed `XChart` to create a real-time performance graph.
- **Dynamic Scaling**: The system automatically identifies the available CPU cores by using `Runtime.getRuntime().availableProcessors()`.
- **Clean Output**: The formatting in the console and the titles of the charts were improved for better clarity and presentation quality.

4. Testing Methodology

Correctness:

- Validated estimated values of π are consistently close to 3.14159 within reasonable error for large sample sizes.

Performance:

- Benchmarks were run with increasing sample sizes (10M to 100M).
- Execution times were recorded for both versions.
- Speed-up (Sequential / Parallel time) calculated.

Samples	Sequential Time(s)	Parallel Time(s)	Speed-Up
10,000,000	0.7966	1.049	0.76
25,000,000	2.1762	0.4327	5.03
50,000,000	2.638	0.795	3.32
75,000,000	3.8926	1.1303	3.44
100,000,000	5.1837	1.5078	3.44

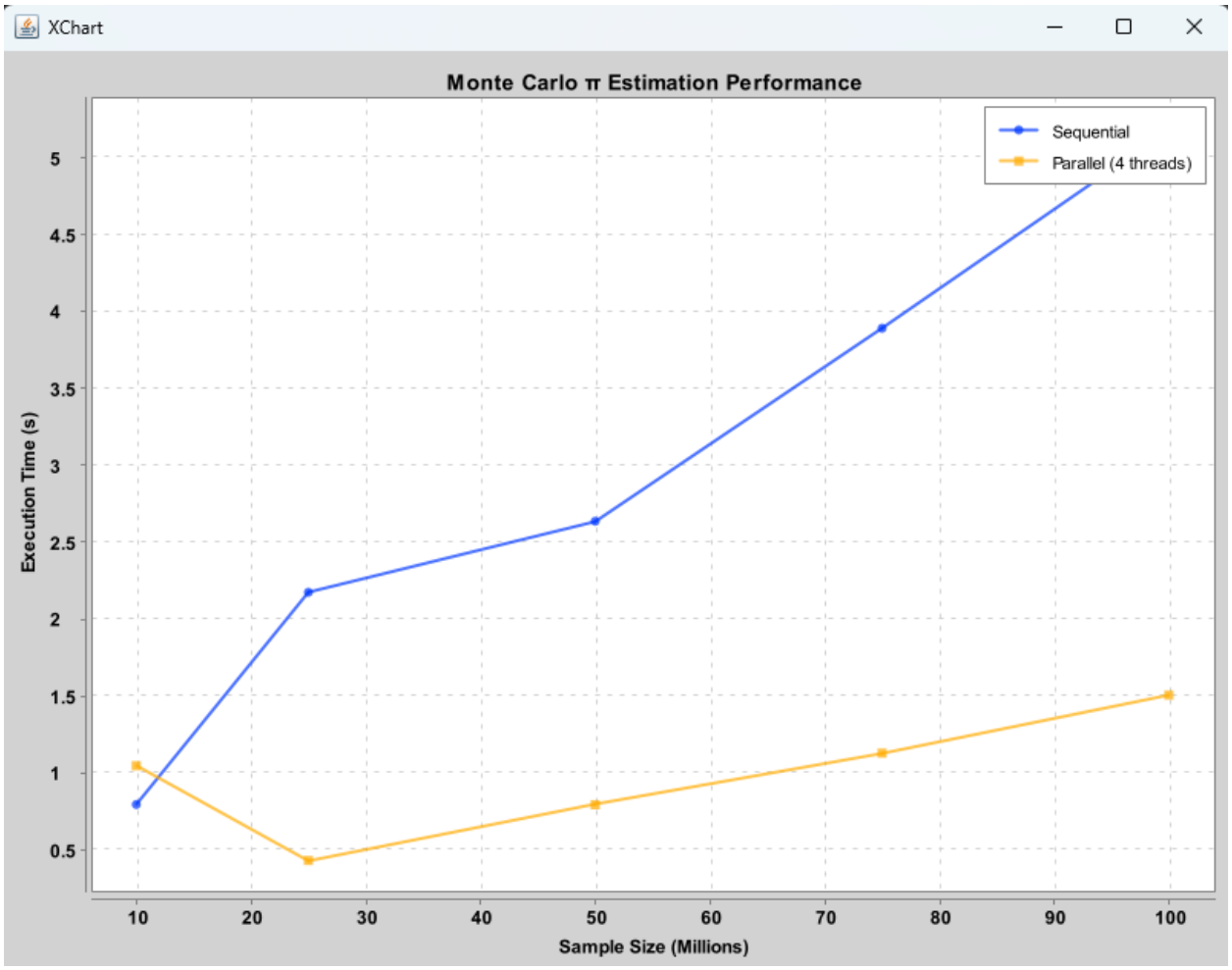

Figure 1-Sequential vs Parallel Execution time with respect Sample Size

📊 Chart Explanation:

- The X-axis shows the number of sample points (in millions).

- The Y-axis shows the execution time in seconds.

- It compares Sequential (1-thread) vs Parallel (4-thread) estimation.

- Larger speed-ups are expected with more points and more threads.

Comparison with Sequential

Wins:

- Parallel version achieved consistent speed-up (~3.3x to 3.4x)
- Effective CPU core utilization without increasing memory complexity

Trade-offs:

- Slightly more complex implementation
- Overhead from thread scheduling becomes noticeable at small sample sizes

Individual Contributions

- **Alaa Yehia (6432)**
    - Implemented the core Monte Carlo algorithm (both sequential and parallel versions).
    - Designed and integrated chart visualization using XChart.
    - Led the code refactoring for cleaner structure and presentation-quality output.
- **Ali Jaafar (6411)**
    - Developed the multithreading infrastructure using ExecutorService and Callable.
    - Conducted performance benchmarking and data analysis.
    - Authored the documentation and testing methodology sections of the report.

Conclusion

This project illustrates that parallelization significantly accelerates CPU-bound numerical simulations, such as Monte Carlo π estimation. The existing design effectively scales with both the number of cores and the size of the input data.

Future Enhancements:

- Visualize the speed-up in a distinct chart
- Incorporate user-configurable thread counts
- Implement secure RNG or quasi-random generators (e.g., Sobol sequences) to enhance statistical performance
- Expand the project to estimate additional integrals using Monte Carlo techniques