ULFG III

Black Scholes (Monte-Carlo Sim)

Concurrent Programming

by

Mohammad Kansoun , 6123

Spring 2024-2025

Supervisor:

Prof Mohamad AOUDE

Black–Scholes Monte Carlo: Sequential vs. Parallel in Java

1. Introduction

Accurate pricing of European call options under the Black–Scholes model often relies on closed-form formulas. However, for more complex derivatives or path-dependent payoffs, Monte Carlo (MC) simulation is the go-to numerical method. This project implements and evaluates a high-throughput MC simulator in Java, leveraging modern concurrent APIs to exploit multi-core CPUs.

Goals:

1. Develop a correct sequential MC baseline.

2. Design and implement a parallel version for speed up.

3. Provide a lightweight GUI for parameter entry and timing.

2. Design

Algorithms:

Core Algorithm:

$$S_T = S0\exp((r - \frac{1}{2}\sigma^2)T + \sigma\sqrt{T}Z), Z \sim N(0,1).$$

- Payoff: max (ST−K,0), averaged over N paths and discounted by e^{-rT} .

- Sequential: simple for-loop over paths, accumulating payoffs.

- Parallel: split paths equally across threads; each task returns its local sum; master sums via LongAdder.

Data Structures & Synchronization:

- ExecutorService with a fixed-size pool to reuse threads.

- Callable<Double> Tasks & Future<Double>

-

Justification:

- Per-Thread Local Sums avoid any locking or atomic updates inside the inner Monte Carlo loops, maximizing throughput.

- exactly one Callable per thread (each handling paths/threads simulations) ensures low scheduling overhead compared to submitting thousands of tiny tasks.

- ExecutorService Reuse cuts out the repeated cost of thread startup/shutdown, crucial for GUI - driven, repeated runs.

3. Implementation Notes

Key Classes:

Sequential core compute(...) and runAndTime(...) as provided in source.

Parallel core uses ExecutorService, Callable<Double> Tasks & Future<Double> to coordinate threads.

Obstacles & Solutions:

- Avoiding Shared-Accumulator Bottlenecks: switch to per - thread local sums returned by each Callable<Double>. This removes any shared writes inside the hot loop and simplifies the code

- Thread startup cost: reused a single ExecutorService instead of rebuilding per click.

- GUI responsiveness: separated compute(...) from timing (runAndTime(...)) so Swing event-dispatch remains fluid.

4. Testing Methodology

Correctness:

- Ran both versions with parameters: S0=100, K=100, r=0.05, σ=0.2, T=1; paths in {10^6,10^7}.

- Confirmed relative difference <0.1%.

- N=10^6 Price sequential= 10.44 | Price parallel = 10.439 (4 threads)

Performance:

- Machine A: i5-8350U, 4 cores, 16 GB RAM.

- Warm-up: one 10,000-path "dry run" before timing.

5. Results

Results table and charts to be inserted:

Paths (N)	Threads (P)	ParallelTime (ms)	Sequential Time (ms)
1000000	2	55.72	67.75

1000000	4	48.04	66.15
10000000	2	514.88	656.02
10000000	4	474.81	661.05

Bottlenecks:

- For small N, parallel overhead dominates (speed-up <1).

- At large N, amortized overhead yields efficiencies ≈ 0.8–0.9 on 4 cores(since my laptop has small number of cores sequential is faster than the multithreading).

6. Comparison & Trade-offs

Metric comparison between sequential and 4-thread parallel: price, time, speed-up, efficiency.

Wins: faster for N=10^7. GUI integration allows real-time exploration.

Trade-offs: Initial overhead makes it unsuitable for small N; more complex code and slight memory overhead.

7. Conclusion & Future Work

This project demonstrates that with careful task sizing and low-contention accumulation, Java’s ExecutorService + LongAdder achieves near-linear scaling on multi-core hardware.

Future directions: Fork/Join, parallel streams, GPU offloading, auto-tuning at runtime.