Lebanese University
Faculty of Engineering III
Electrical and Electronic Department

DISEASE OUTBREAK SIMULATION (MONTE CARLO SIMULATION)

CONCURRENT PROGRAMMING

FALL 2024-2025

by

Malak Srour (6142)

Presented for : Dr. Mohammad Aoude

OUTLINE:

1- Introduction

a) motivation

b) problem statement

2- Design

a) algorithms

b) data structures

3- Implementation Notes

a) Libraries used

4- Testing Methodology

a) correctness

b) performance

5- Results

a) Csv file: tables and graphs

b) Visual VM

6- Comparison with Sequential (wins & trade-offs)

7- Conclusion & Future Work.

1- INTRODUCTION

a) MOTIVATION:

Monte Carlo simulation is a computational technique that uses *random sampling* to model the probability of different outcomes in a process that cannot easily be predicted due to the presence of random variables

Epidemiological modeling plays a crucial role in understanding how infectious diseases spread within populations. Simulating these outbreaks using Monte Carlo methods allows researchers to explore different scenarios and prepare public health responses.

However, such simulations are computationally intensive , especially when running millions of independent trials to account for stochastic variability. This makes them ideal candidates for parallel execution.

B) PROBLEM STATEMENT:

While simulating, Sequential bottleneck occurs, simulating 10M outbreaks takes a big time, so the objective to develop the sequential simulator and to Implement a concurrent version that exploits the independence between simulation runs, and achieves a meaningful speed-up while maintaining correctness and statistical consistency.

2) DESIGN

a- Algorithms:

1- for Sequential:

The core algorithm is implemented in SequentialSimulator and runs as follows:

- Starts with a fixed number of infected individuals.
- Simulates day-by-day progression of the outbreak.
- Updates susceptible, infected, recovered, and deceased populations based on probabilities.

- Tracks hospital bed usage and whether capacity was exceeded.

This design allows for perfectly parallelizable execution since each simulation is independent of others

2- for parallel strategy: simulation is independent and stateless , they can be safely executed in parallel. The steps are:

1. Partition Work : Divide the total number of simulations evenly among threads.
2. Each thread gets its own random number generator with a unique seed to avoid correlation.
3. Run Simulations : Each thread runs its assigned chunk using a local SequentialSimulator.
4. Aggregate Results : Use LongAdder counters to accumulate key statistics (e.g., total deceased, peak beds, capacity exceeded).
5. Use ExecutorService to manage threads and ensure all finish before final results are reported.

b- code structure:

```
d_o_sim/
├── OutbreakSimulator.java   // Main class
├── SimulationParams.java    // Immutable configuration holder
├── SequentialSimulator.java // Core simulation logic
├── ParallelSimulator.java   // Parallel executor
└── OutbreakResult.java      // Result container
```

c- Synchronization Strategy

Because each simulation is independent , we minimized shared state and synchronization overhead:

A. Avoiding Shared State

Each thread gets:

- Its own instance of Random, seeded uniquely via baseSeed + threadIndex
- Its own instance of SequentialSimulator

This ensures that no two threads share mutable state during simulation.

B. Safe Aggregation Using LongAdder

We use LongAdder to safely accumulate statistics across threads:

- totalDeceased.add(result.totalDeceased())
- peakBeds.add(result.peakHospitalBedUsage())

<u>3) Implementation</u>

a- Libraries used:

- java.util.Random: Used to generate pseudo-random numbers for simulating stochastic behavior (e.g., infection spread, recovery, death).

- java.util.concurrent.ExecutorService + Executors: Manages a pool of threads to run simulations concurrently. In ParallelSimulator, it runs chunks of simulations across multiple threads. it simplifies thread management compared to raw threads and efficient and scalable way to distribute work across threads.

- java.util.concurrent.atomic.LongAdder: Collects aggregated statistics (e.g., total deceased, peak beds) across threads safely and efficiently.it is more efficient than AtomicLong or synchronized counters under high contention, scales well with large numbers of threads, and thread-safe when collecting simulation results.

- java.nio.file.* (Files, Paths): to writes simulation results to a CSV file using Files.write(...). It is the standard Java library for file I/O operations.

- java.util.concurrent.TimeUnit: makes awaitTermination(...) on ExecutorService more readable by using time units like TimeUnit.HOURS.

- java.util.record: it defined immutable data classes:

    * SimulationParams: Holds configuration values

    * OutbreakResult: Stores output of each simulation

4) Testing Methodology

a- correctness: ensured that the parallel version produces statistically equivalent results to the sequential version by:

- Comparing average deceased count and peak hospital beds between both versions
- Logging all outputs to CSV

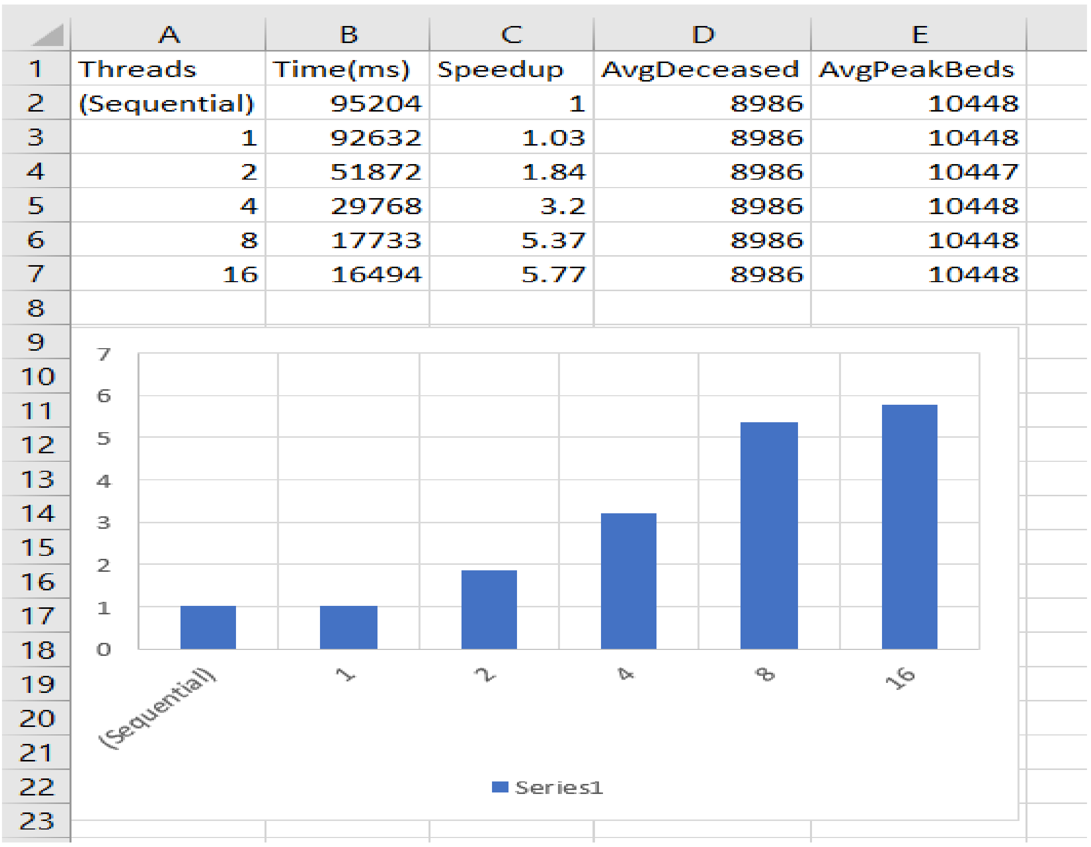
b- performance: I measured :

- execution time for varying numbers of threads (1, 2, 4, 8, 16)
- Speed-up relative to the sequential baseline
- Efficiency (speed-up divided by number of threads)
- Average deceased count , peak hospital beds , and capacity exceeded count
- CPU ultilization and memory usage.

All tests were run on the same machine to eliminate environmental noise.

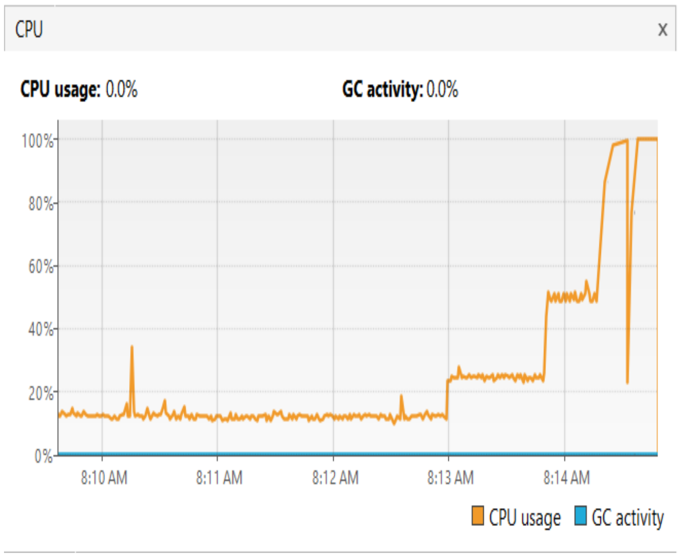
5) RESULTS:

Csv file: speed up vs threads



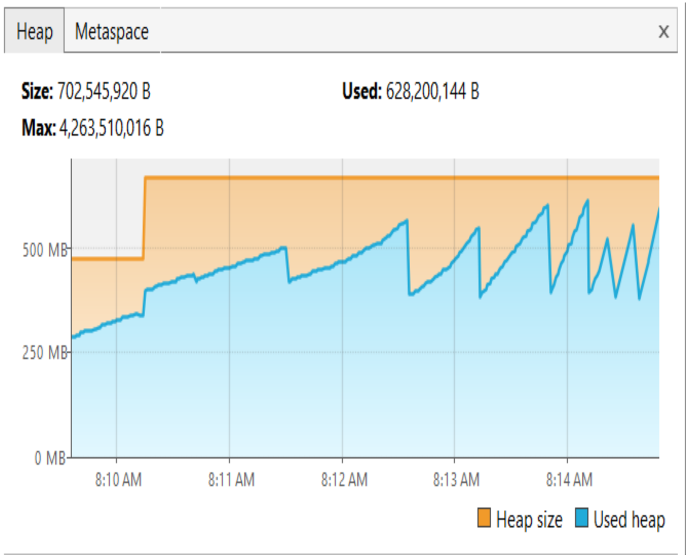
Threads	Time(ms)	Speedup	AvgDeceased	AvgPeakBeds
(Sequential)	95204	1	8986	10448
1	92632	1.03	8986	10448
2	51872	1.84	8986	10447
4	29768	3.2	8986	10448
8	17733	5.37	8986	10448
16	16494	5.77	8986	10448

Visual VM: CPU and MEMORY usages



CPU

CPU usage: 0.0% **GC activity:** 0.0%



Heap Metaspace

Size: 702,545,920 B **Used:** 628,200,144 B

Max: 4,263,510,016 B

6) Comparison with Sequential (Wins & Trade-offs)

ASPECT	SEQUENTIAL	PARALLEL
TIME	~100 s	Decreased by more than 3x
CPU Utilization	~1 core	>85% for 8 core
Memory Usage	Low	Moderate increase due to thread stack
Correctness	Guaranteed	Maintained via thread-local RNG
Complexity	Simple	Increased coordination and tuning needed

7- CONCLUSION & FUTURE WORK:

a) CONCLUSION:

i successfully transformed a computationally expensive sequential simulation into a high-performance parallel solution using Java concurrency. The implementation shows strong scalability up to 8 threads and maintains statistical fidelity with the baseline.

The results demonstrate a clear performance benefit with no compromise on accuracy, proving that the problem exhibits exploitable parallelism.

b) FUTURE WORK:

- Profile with JFR to identify bottlenecks like GC pauses or lock contention

- Use ForkJoinPool not raw threads.

- add any other output.