

Lebanese University

Faculty of Engineering III

Electrical and Electronic Department

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

monte carlo pandemic simulation

using sir model, JAVA

by

**Alaa Fadlallah, 6445**

**Batoul Abbas, 6502**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Spring 2024-2025**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Concurrent Programming**

**Dedicated to : Dr. Mohammad Aoude**

monte carlo pandemic simulation

**1. Introduction**

**1.1 Problem Statement**

The SIR (Susceptible-Infected-Recovered) model is a fundamental epidemiological model used to simulate the spread of infectious diseases. The motivation behind this project is to explore how parallel computing can improve the efficiency of running large-scale SIR simulations, especially in scenarios that require millions of iterations for probabilistic modeling. The problem lies in the time complexity of sequential simulation runs which can become a bottleneck. This project implements and compares a sequential approach & two parallelization approaches in Java: ExecutorService and ForkJoinPool, and visualizes their performance through Python

**1.2 Motivation**

Modern CPUs with multiple cores are underutilized in sequential Monte Carlo simulations, where only one core performs all computations. The SIR simulation exhibits high parallelism since each run is independent. By leveraging Java's concurrent programming features, we can significantly reduce simulation time and maximize resource usage

**1.3 Objectives**

* Implement sequential and parallel versions of SIR simulation
* Use ExecutorService and ForkJoinPool for multithreading
* Analyze speedup and efficiency
* Validate correctness of parallel output
* Visualize performance metrics using Python

**2. Design and Implementation**

**2.1 Problem Domain Analysis**

Each simulation follows a Monte Carlo approach to disease spread:

1. Initialize population with 1 infected, rest susceptible
2. Each day:

• Infected individuals infect susceptibles with probability β

• Infected individuals recover with probability γ

1. Update S, I, R states
2. Record metrics (peak infected, total days)
3. Stop when I = 0 or 365 days passed

**2.2 Parallel Strategies**

**2.2.1 ExecutorService - Thread Pool Approach**

**• Algorithm**: Fixed-size thread pool

• **Decomposition:** Assign simulations evenly to threads

• **Synchronization:** Futures with result collection

• **Memory:** Local object reuse to minimize GC

**2.2.2 ForkJoinPool - Recursive Task Approach**

**• Algorithm:** Recursive task splitting

• **Decomposition:** Split simulation count until threshold

• **Threshold**: 50,000 simulations per task (empirically chosen)

• **Synchronization**: Fork/Join handles work-stealing and merging

**2.3 Data Structures and Memory Management**

• **SIRSimulation:** Encapsulates SIR logic (1000 population, β=0.3, γ=0.1)

• **SimulationStats**: Holds duration and peak infection count

• **Futures:** Used in ExecutorService result collection

• **RecursiveTask**: ForkJoin subtask result structure

**2.4 Synchronization Mechanisms**

• **ExecutorService**: Synchronization via Futures

• **ForkJoinPool**: Implicit work-stealing and task joining

• **Thread-safety:** No shared state, simulation objects are isolated

**3. Implementation Details**

**3.1 Technical Challenges and Solutions**

**3.1.1 Optimal Splitting Strategy**

**Problem** Too small = overhead, too large = underutilization

**Solution**: Threshold set to 50,000 for ForkJoin.

**3.1.2 Result Aggregation Problem:**

**Problem**: Efficiently combining results without locks

**Solution**: Use return values from Callable and RecursiveTask

**3.1.3 Memory Efficiency Problem:**

**Problem**: High object creation

**Solution**: Object reuse within threads

**3.2 Code Architecture**

Main.java

├─ Sequential baseline (runSimulationsSequential)

├─ Parallel using ExecutorService (SimulationTask)

└─ Parallel using ForkJoinPool (SimulationForkTask)

**4. Testing Methodology**

**4.1 Correctness Verification**

Reproducibility with fixed random seeds

• Output comparison: duration and peak infected match

• Edge cases tested: β = 0, γ = 1

**4.2 Performance Testing Protocol**

* **Hardware**: 12-core CPU system
* **Thread Configurations**: Time measured using System.nanoTime() (Threads tested: 1, 2, 4, 8, 16, 32)

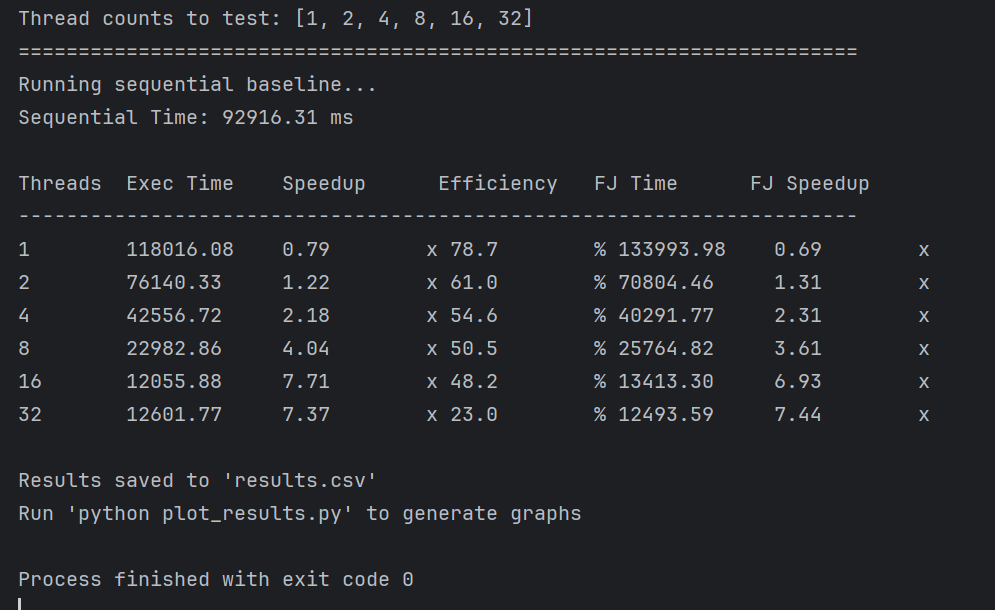
• **Monitored CPU utilization**

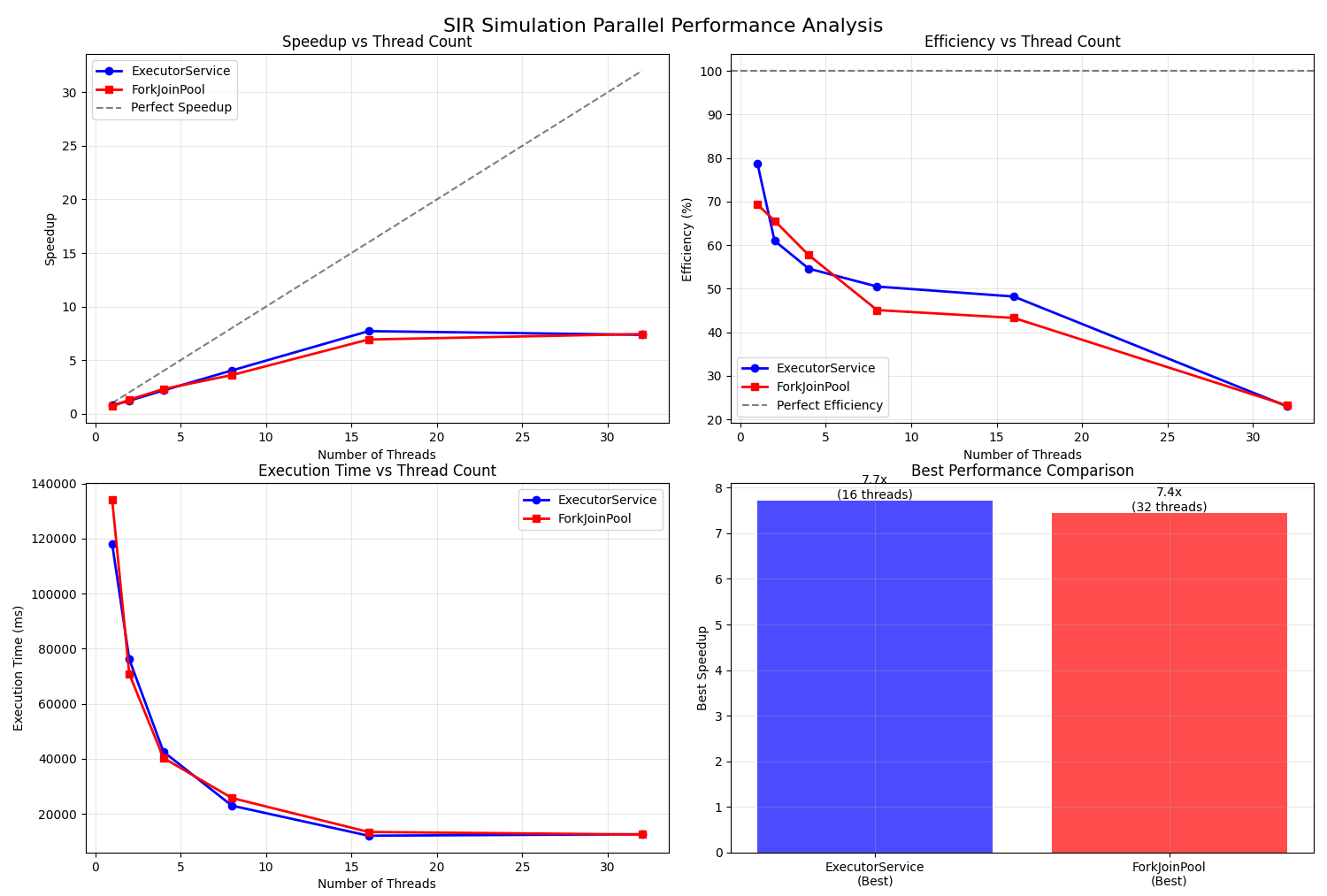
**4.3 Metrics Collection**

* **Execution Time**: ms
* **Speedup Calculation**: S = T\_sequential / T\_parallel
* **Efficiency**: E = Speedup / Number\_of\_threads

**5. Results and Analysis**

**5.1 Performance Results**

****

****

**\***efficiency drops due to overhead or uneven workload

**5.2 Scalability Analysis**

• ForkJoin and ExecutorService showed steady speedup with diminishing efficiency

• ExecutorService peaked at 7.71x speedup, ForkJoin peaked at 7.44x

• Efficiency dropped after 8 threads due to management overhead

* Speedup nearly linear up to 8 threads, then plateaus
* Efficiency decreases with more threads, especially past 16
* Execution time drops steeply initially, then stabilizes
* Best performance: ExecutorService (16 threads), ForkJoin (32 threads)

**6. Pros and Cons**

**Executor Service**:

* Easy to implement
* Good performance up to 16 threads
* Less dynamic load balancing

**ForkJoin:**

* Efficient task stealing
* Clean recursive splitting
* Slightly higher overhead

**7. Bottleneck Analysis**

**7.1 Identified Bottlenecks**

• Task overhead from small thresholds

• ForkJoin management at high thread counts

• CPU saturation beyond hardware core count

**7.2 Optimization Opportunities**

• Threshold tuning (50K)

• Balanced task distribution

• Avoided thread contention by isolating simulation objects

**8. Conclusion and Future Work**

* 1. **Project Outcomes**

• ExecutorService: 7.71x speedup at 16 threads

• ForkJoinPool: 7.44x speedup at 32 threads

• Validated correctness and strong scaling behavior

**8.3 Future Enhancements**

• GPU acceleration

• Adaptive load balancing thresholds

• Multi-node simulation with distributed systems

**8.4 Real-world Applications**

• Public health simulation

• Agent-based modeling

• Real-time risk forecasting