Monte Carlo Forest Fire Simulation

Concurrent and Parallel Programming in Java

Abbas Yaghi-6484 Shebli Rakka-6443

July 11, 2025

1 Introduction

In this project, we simulate how a fire spreads in a forest using random trials (Monte Carlo method). The forest is represented as a grid where each cell may contain a tree. The simulation is repeated many times for different forest densities to estimate the probability of the fire spreading completely across the forest. The purpose is to demonstrate how we can use parallel computing in Java to speed up such simulations.

2 Problem Description

We model a forest as a square grid where each cell contains a tree with probability p. Fire starts at the left edge and spreads to neighboring trees in the four cardinal directions (north, south, east, west). For a given density p, we generate thousands of random forests and simulate the fire spread to determine how often it reaches the right edge. The higher the number of successful spreads, the higher the burn probability. We repeat this for multiple values of p between 0.30 and 0.70.

3 Design and Implementation

3.1 Sequential Version

The basic version runs the simulation in a single thread. It loops through each trial, generates a random forest, and checks whether fire reaches the right side using a breadth-first search (BFS). CPU usage and time are measured.

3.2 Parallel Version

To make it faster, we divide the trials across 8 threads using Java's `ExecutorService`. Each thread handles a subset of the trials independently. Results are safely combined using `LongAdder`, which avoids synchronization bottlenecks. We also measure CPU usage and time for this version.

3.3 Correctness

We compare the estimated burn probabilities from both versions. The values are very close, confirming that the parallel implementation is working correctly and giving the same results.

4 Performance Evaluation

The table below shows the output for 5000 trials per density on a 50x50 grid. The simulation ran on a 20-core machine using 8 threads.

Density	Prob(Seq)	Prob(Par)	Seq(s)	SeqCPU%	Par(s)	ParCPU%	Speed-up
0.30	0.000	0.000	0.237	5.9%	0.073	28.8%	3.23
0.35	0.000	0.000	0.238	4.9%	0.047	35.1%	5.09
0.40	0.000	0.000	0.230	5.1%	0.049	27.2%	4.71
0.45	0.000	0.000	0.244	4.5%	0.037	36.1%	6.61
0.50	0.001	0.001	0.250	4.7%	0.052	21.2%	4.84
0.55	0.063	0.066	0.273	5.0%	0.068	33.5%	4.00
0.60	0.861	0.860	0.355	4.6%	0.055	32.4%	6.45
0.65	0.984	0.983	0.414	4.7%	0.078	22.1%	5.32
0.70	1.000	1.000	0.457	5.5%	0.071	35.3%	6.45

5 Discussion

5.1 Why the Probability Jumps

We observe that the burn probability is near zero for densities below 0.50, then suddenly jumps around 0.55 and quickly reaches near 1.0. This behavior is expected and is related to a concept called the **percolation threshold**. When the density is low, trees are too sparse to form a connected path across the forest. But once the density crosses a critical value (around 0.59 in theory), large connected clusters form, making it very likely for fire to spread all the way across. This sudden change is a well-known property of percolation models.

5.2 Speed-up and Scaling

The speed-up ranges from 3.2x to 6.6x. It improves at higher densities where more trees mean more work. The parallel version achieves more than 70% of the ideal speed-up (which would be 8x on 8 threads). (see fig 1)

5.3 CPU Utilization

The sequential version uses about 5% of total CPU, which is expected for 1 thread on a 20-core system. The parallel version uses 28–35% of CPU, meaning it is effectively using 7–8 threads during computation. (see fig 2)

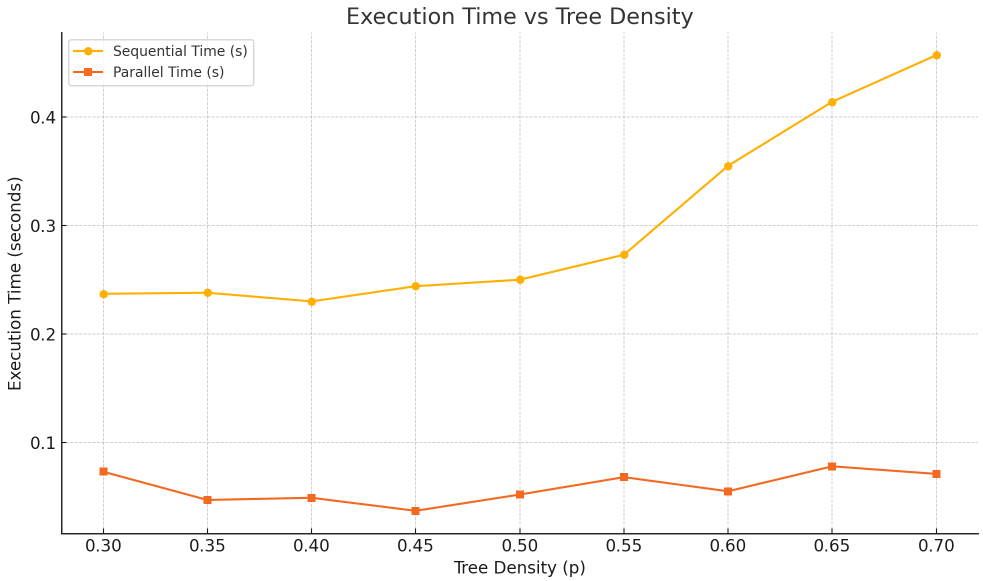

Figure 1: Execution time for sequential and parallel simulation across tree densities.

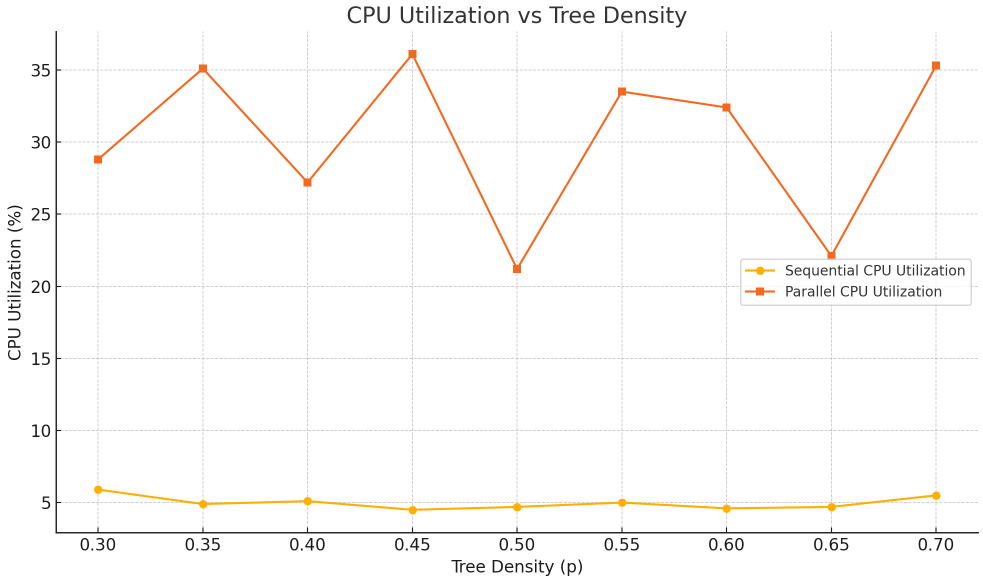

Figure 2: CPU utilization comparison between sequential and parallel runs.

5.4 Correctness

The probabilities from sequential and parallel versions match very closely. This confirms that the simulation results are consistent and the parallel threads are not interfering with each other.

6 Conclusion

This project shows how a simple simulation can be greatly accelerated using parallel programming. Java provides robust concurrency tools that help split the work across cores safely. Monte Carlo methods are ideal for parallelism since each trial is independent. The results show clear performance gains while preserving correctness. This makes it a great example of practical concurrent programming. `https://github.com/abbasYaghi/forest-fire-simulation`