CS5741 – Concurrency & Parallelism in Software Development
Assignment 2: Parallelising the Simulation of Forest Fire Spread and Recovery
Instructor: Dr. Faeq Alrimawi

Report Submission

Rohan Sikder
24165816
November 26, 2024

Table of Contents

Introduction

Objective

The main goal of this assignment is to design and implement a parallelized simulation of forest fire spread and recovery using data and task parallelism. The simulation represents the condition of a forest grid in which every cell contains either the state of a tree of an empty space or of a burning tree.

The parallel implementation is expected to:

- Use multiple threads to process the forest grid efficiently.
- Ensure thread consistent updates with proper synchronization.
- Measure and analyse the performance gains with speedup, efficiency and theoretical scalability based on Amdahl's Law, Gustafson's Law and Karp-Flatt metric.

Background

Forest fire simulates fire spreading through a forest as a 2D grid with each cell having one of three states:

Empty - There are no trees in this cell.
Tree - This cell has a healthy tree.
Burning - A tree is burning in this cell.

Following transitions occur at each time step:

- **Spread of Fire**: Burning trees become dead cells. If adjacent trees have a specified probability of catching fire, they will catch fire.
- **Growth of Trees:** If the growth probability is high enough, empty cells may grow into trees.
- **Boundary Conditions:** The grid completely wraps around its edges.

Design and Implementation

Sequential Implementation

The first sequential solution is used as a starting point for implementing and analysing the parallel version of the forest fire simulation. It is a two dimensional grid with each cell having one of 3 states. (EMPTY, TREE or BURNING)

Initialization of the grid

The grid is made up of two 2D arrays: *forest* for the current state and *nextForest* for the next state. During initialization:

1. Each cell is randomly assigned a state.
2. (TREE) 50% chance of being a tree.
3. (BURNING) 2% chance of trees beginning as burning .
4. (EMPTY) remaining cells are empty.
5. States are assigned by random number generation.

Probability Parameters Growth Probability (*growthProbability*): The chance that an empty cell will develop into a tree in the next step. Set to 0.01.

Burn Probability (*burnProbability*): Probability of a tree cell burning when at least one neighbour is burning. Set to 0.50.

Simulation Rules
Each time step:

1. **Burning Cells**: Transition to EMPTY.
2. **Tree Cells**: Catch fire if any neighbor is burning, based on burnProbability.
3. **Empty Cells**: Grow trees based on growthProbability.

The sequential implementation applies these rules through nested loops iterating over the entire grid.

During the simulation, states can be visually seen. Data on number of cells (No. Of EMPTY, TREE or BURNING) can logged with Boolean flags:

- **Visualization**: The simulation includes an option to visualize the grid state which shows the forest being on fire and simulating at each time step . This can be controlled with the Boolean flag: *visualize*
- **State Tracking**: The number of cells in each state (TREE, EMPTY, BURNING) can be logged at each time step. This is controlled with the Boolean flag: *trackStates*

Parallel Implementation

The parallel version is optimized for performance, utilizing both data parallelism and task parallelism.

Data Parallelism

The forest grid is segmented into chunks of rows, and each thread processes a subset of rows. This method reduces thread dependencies since each thread updates its own rows depending on the state of the forest at the moment.

```
// Divide the grid into tasks for the threads
 int rowsPerThread = Math.max(1, gridHeight / threads);
 UpdateTask[] tasks = new UpdateTask[threads];

 for (int t = 0; t < threads; t++) {
    int startRow = t * rowsPerThread;
    int endRow = (t == threads - 1) ? gridHeight : (t + 1) * rowsPerThread;
    tasks[t] = new UpdateTask(startRow, endRow, t);
 }

 // Run the tasks using the ForkJoin framework
 pool.invoke(new ParallelSimulation(tasks));
```

Task Parallelism

Tasks like updating cell states and logging simulation states are executed in parallel. Each time step is followed by logging (e.g. counting TREE, BURNING, and EMPTY cells) which reduces contention on the grid update.

The ForkJoin framework divides the grid into smaller tasks that can be executed independently. The UpdateTask class implements recursive splitting, This recursive approach ensures balanced task distribution and dynamic load balancing.

```
@Override
protected void compute() {
   if (endRow - startRow <= THRESHOLD) {
      processRows();
   } else {
      int midRow = (startRow + endRow) >>> 1;
      invokeAll(new UpdateTask(startRow, midRow, threadId), new
UpdateTask(midRow, endRow, threadId));
   }
}
```

Synchronization

To keep updates consistent in between threads:

Thread-local Random Generators: In this implementation, thread-local random generators were used to minimize contention when generating random probabilities for cell state updates:

```
threadLocalRandoms = new Random[threads];
for (int i = 0; i < threads; i++) {
   threadLocalRandoms[i] = new Random(initRandom.nextLong());
}
```

Grid Swapping: Grid swapping ensures that all threads work on the current state of the forest while avoiding conflicts. This approach reduces the need for explicit locks or semaphores by maintaining data locality and avoiding shared state dependencies:

```
int[][] temp = forest;
forest = nextForest;
nextForest = temp;
```

ForkJoin Framework

The ForkJoin framework seperates tasks recursively into smaller sized sub tasks until they're manageable in size. This implementation uses ForkJoinPool and a customized **RecursiveAction:** Tasks with RecursiveAction: Each UpdateTask runs just a subset of rows. In case the subset is bigger compared to threshold, it's recursively divided.

Justification for Synchronization Approach

The implementation of synchronization implicitly using grid swapping, thread-local random number generators and ForkJoin. These methods minimize the overhead of explicit locks (contention and the cost of acquiring and releasing locks) without sacrificing correctness. For example, grid swapping guarantees that all threads run on the same state (forest) while updating the next state (nextForest) without data races. Thread-local randomness also removes shared state dependencies, further reducing contention. Mutexes and semaphores can provide synchronization, but their use here would probably hurt performance due to the additional overhead, especially in very parallel situations. In this implementation, through implicit synchronization mechanisms, achieves high scalability and performance while remaining in line with the concepts of efficient parallel programming.

Performance Analysis

Experimental Setup

The parallelized forest fire simulation was performed with following configurations.

Grid Sizes: Tests have been performed on 100x100, 500x500, 1000x1000, and 2000x2000 grids.

Steps in Simulation: All configurations used a fixed number of time steps (500).

Threads Counts: Scalability and efficiency have been analysed by running the simulation using one (serial), 2, 4 and 8 threads.

Specifications for Hardware: Tests were performed using a multicore computer with an M2 MacBook Air with the following specs:
Processor: Apple M2 (8 core CPU/10-core GPU)
Memory: Unified Memory (8 GB)

Parameters of Probability:
Growth: growthProbability = 0.01
Burn: burnProbability 0.5

The performance parameters like execution time, speedup, effectiveness along with theoretical scalability variables were gathered into CSV to look at the performance of the parallel implementation.

Metrics

Execution Time

Execution time for every configuration was evaluated as the time necessary to finish the simulation. It was compared over various thread counts to evaluate parallelism performance gains.

Speedup

Speedup (S) shows the performance gain with multiple threads compared with the baseline serial implementation. Tserial is the execution time of the simulation using 1 thread. Tparallel is the execution time using multiple threads:

$$S = \frac{Tserial}{Tparallel}$$

Efficiency

Efficiency (E) evaluates how effectively the threads are utilized in the parallel implementation. P is the number of threads. Efficiency decreases as thread count increases due to overhead and contention:

$$E = \left(\frac{S}{P}\right) \times 100$$

Amdahl's Law

Amdahl's Law estimates the theoretical speedup based on the fraction of the program that can be parallelized (f) shows the diminishing returns of parallelism as the number of threads increases especially if the parallelizable portion (f) is limited.

$$S_{Amdahl} = \frac{1}{(1-f) + \frac{f}{P}}$$

Gustafson's Law

Gustafson's Law accounts for the scalability of parallelism with increasing problem size, This law shows that as the problem size grows, the effective speedup improves due to the increased contribution of parallelizable tasks:

$$S_{Gustafson} = (1 - f) + f \times P$$

Karp-Flatt Metric

The Karp-Flatt metric quantifies parallel overhead and helps identify non-parallelizable components (ϵ) represents the parallel overhead fraction. A smaller value of ϵ indicates efficient parallelism:

$$\epsilon = \frac{\frac{1}{S} - \frac{1}{P}}{1 - \frac{1}{P}}$$

Results

Grid Size	Steps	Threads	Time (s)	Speedup	Efficiency	Amdahl	Gustafson	KarpFlattMetric
100x100	500	1	0.09	1	100	1	1	0
100x100	500	2	0.08	1.13	56.55	1.82	1.9	0.55
100x100	500	4	0.05	1.97	49.13	3.08	3.7	0.325
100x100	500	8	0.05	2.04	25.45	4.71	7.3	0.2125
500x500	500	1	1.65	1	100	1	1	0
500x500	500	2	1.34	1.24	61.85	1.82	1.9	0.55
500x500	500	4	0.88	1.87	46.85	3.08	3.7	0.325
500x500	500	8	1.07	1.54	19.27	4.71	7.3	0.2125
1000x1000	500	1	7.18	1	100	1	1	0
1000x1000	500	2	5.57	1.29	64.46	1.82	1.9	0.55
1000x1000	500	4	10.7	0.67	16.78	3.08	3.7	0.325
1000x1000	500	8	4.55	1.58	19.74	4.71	7.3	0.2125
2000x2000	500	1	28.38	1	100	1	1	0
2000x2000	500	2	16.88	1.68	84.03	1.82	1.9	0.55
2000x2000	500	4	26.13	1.09	27.14	3.08	3.7	0.325
2000x2000	500	8	23.21	1.22	15.28	4.71	7.3	0.2125

Summary of Results Execution time:

The parallelized forest fire simulation performance was assessed for various grid sizes, thread counts and fixed time steps (500). Scalability and drawbacks of parallelization are shown by the main metrics.
Simulation time dropped with decreasing thread count. However, this particular performance was significantly better for smaller grid sizes and fewer threads.

Speedup: This speedup roughly matches theoretical predictions for smaller grid sizes but decreases with increasing thread count (particularly for bigger grids).
Efficiency: Efficiency decreases with thread count, because of overhead and depleting parallel advantages for several workloads.
Scalability: Amdahl's and Gustafson's laws describe the practical and theoretical consequences of parallelization, showing the outcome of the parallelizable fraction of the simulation.

Key Metrics and Observations

- **Small Grid (100x100)**:
  - **Serial Execution**: 0.09 seconds.
  - **Speedup**: Upped to a maximum of 2.04 using eight threads.
  - **Efficiency**: Reduced from 56.55% (two threads) to 25.45% (8 threads) because of greater parallel overhead for such a tiny issue size. The Karp-Flatt metric shows moderate overhead over 4 threads at most.

- **Medium Grid (500x500)**:
  - **Serial Execution**: 1.65 seconds.
  - **Speedup**: Peaked at 1.87 with four threads but dropped to 1.54 with eight threads because of overhead and contention.
  - **Efficiency**: Reduced to 19.27% with 8 threads, greater thread counts on medium grid sizes yield diminishing returns.

- **Huge Grid (1000x1000)**:
  - **Serial Execution**: 7.18 seconds.
  - **Speedup**: Hit 1.58 with 8 threads, but performance varies wildly (4 threads had slower times, for example).
  - **Efficiency**: Decreased significantly on account of grid size/thread count mismatch/parallel overhead mismatch.

- **Very Large Grid (2000x2000)**:
  - **Serial Execution**: 28.38 seconds.
  - **Speed**: Best overall performance with 2 threads (1.68). Bigger thread counts showed diminishing returns and overhead.
  - **Efficiency**: Dropped to 15.28% with 8 threads (because of synchronization and task management expenses).

Impact of parallelizable fraction (parallelFraction = 0.90)
A high parallelizable fraction (f=0.90) was presumed because of the simulation. This implies that 90% of the work could be parallelized. But:

- **Amdahl's Law**: Even at f=0.90, the maximum speedup plateaus at a particular number of threads. For instance, the theoretical speedup for 8 threads is:

$$S_{\text{Amdahl}} = \frac{1}{(1 - 0.90) + \frac{0.90}{8}} = 4.71$$

Practical speedup missed this due to overhead.

- **Gustafson's Law**: This high f allows better scaling for bigger grids (where the problem size increases). As parallel work dominated the serial portion, greater effective speedups were obtained for bigger grids in the simulation.

Key Findings

- **Small Grids**: The overhead of thread management in addition to synchronization, outweighs the benefit of parallelization, especially at high thread counts.

- **Medium Grids**: Moderate grid sizes benefit from faster threads but still experience diminishing returns.

- **Huge Grids**: Parallelization is best for big grids where the large parallelizable fraction (f=0.90) allows large portions of the simulation to operate simultaneously. But overheads like task allocation and synchronization eat into efficiency

- **Scalability**: Based on Gustafson's Law, bigger grid sizes could further enhance scalability by making the most of the excessive f.

Speedup Trends Graph

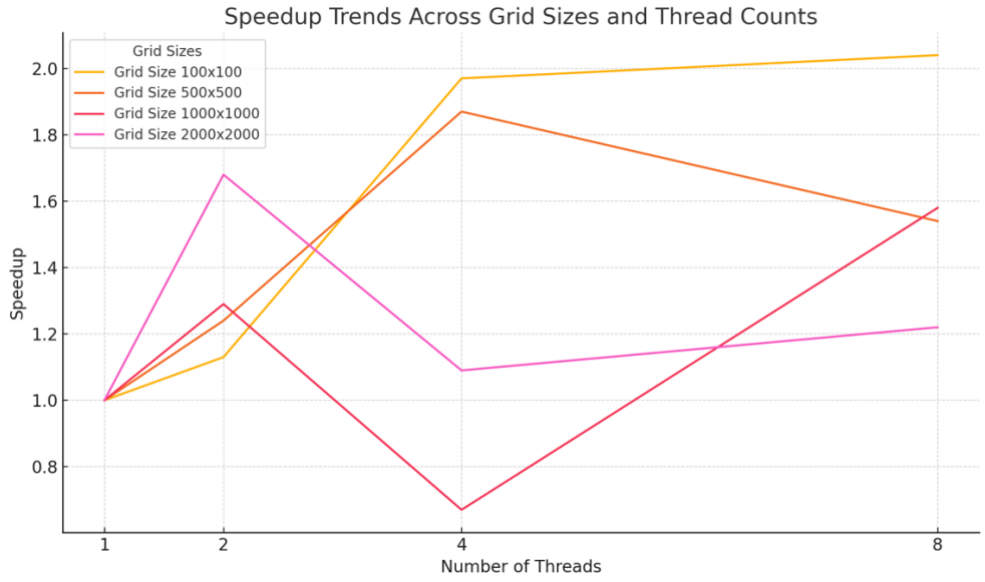

Figure 1: Speedup trends across grid sizes and thread counts.

Efficiency Trends Graph

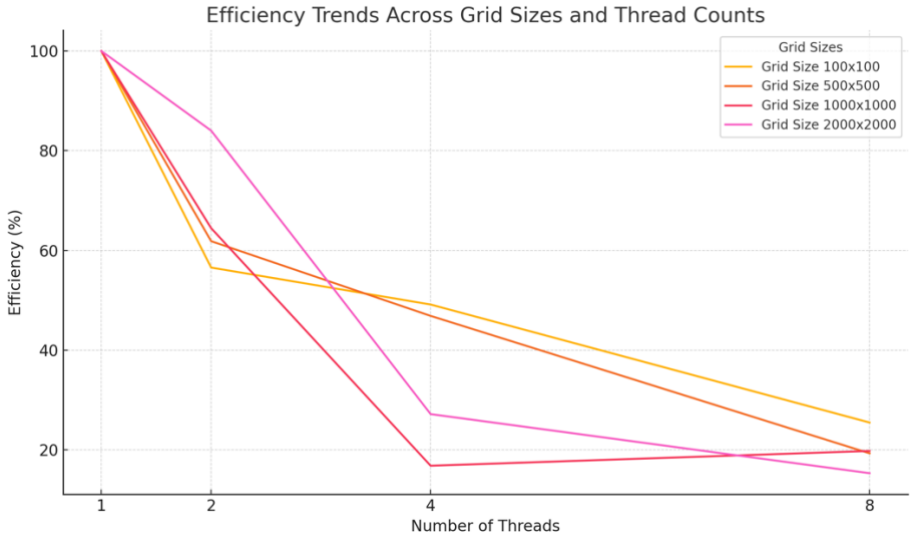

Figure 2: Efficiency trends across grid sizes and thread counts.

Conclusion

The parallel simulation of forest fire spread and recovery showed the advantages and drawbacks of parallelism in computationally intensive tasks.

Performance Gains: Large grid sizes with moderate thread counts (2 or maybe 4 threads) saw substantial execution time reductions. The advantages of parallelism had been limited for smaller grids because of the synchronization overhead and task management costs.

Scalability: The simulation scaled well for big grids and speedups have been in good agreement with Gustafson's Law. Nonetheless, Amdahl's Law discovered the limitations of the serial parts of the simulation.

Limitations on Efficiency: Efficiency dropped as thread count increased, especially for medium or small grid sizes, due to unbalanced load utilization and overheads. Negative effects of Parallel fraction: scalability and performance had been influenced by assumed parallelizable Fraction (f = 0.90).

This big parallel fraction helped bigger grids, but the theoretical speedup bounds defined by Amdahl's Law highlighted diminishing returns with way too many threads.

Implications for Practical Use: This demonstrates that parallelization techniques must be customized to the issue size and hardware restrictions. In computational simulations such as forest fire spread, parallelization has apparent benefits but is in need of focus on overhead and synchronization to get best outcomes.