UNIVERSITY OF LIMERICK
OLLSCOIL LUIMNIGH

Parallelising the Simulation of Forest Fire Spread and Recovery

CS5741 - Concurrency and Parallelism in Software Development

Lecturer: Faeq Alrimawi

Submitted by:

Mahika Jaguste (24016454)

Table of Contents

I. Abstract

A parallelised forest fire simulation across varying grid sizes and thread counts is implemented by decomposing the grid into horizontal segments. The states of individual workers are synchronised using Java's cyclic barrier whereas the global counters maintain integrity via atomic integers. Performance analysis results show that actual speedup aligns with Amdahl's predictions at low thread counts but diverges at higher counts due to synchronisation overhead. Gustafson-Barsis law highlights better scaling potential for larger problems. The Karp-Flatt metric identifies overhead as the primary bottleneck in higher thread configurations. While parallelisation significantly improves performance for medium-scale problems, it is limited by contention and diminishing returns in larger setups.

II. Introduction

Forest fire simulations involve modelling interactions between cells in a grid over multiple steps. Every cell in the grid is in a specific state, either empty, containing a healthy tree, or a burning tree. This state is updated based on probabilistic rules and the states of its neighbouring cells. The goal of this simulation is to develop correct and efficient parallel computation techniques to handle large-scale problems.

In this report, I aim to implement and analyse a parallel version of the forest fire grid-based simulation with inherent dependencies between neighbouring cells in Java. The key objectives of this study include:

1. Parallelisation: Decomposing the grid into independent chunks and managing dependencies between chunks.
2. Synchronisation: Ensuring the correctness of state updates across threads using Java's synchronisation primitives to avoid race conditions and ensure data consistency.
3. Performance Analysis: Evaluating the effectiveness of parallelisation and when it fails to perform in practical scenarios.

This report presents the design and analysis of the serial and parallel implementations of the simulation. It highlights the potential of parallel computing techniques in forest fire simulations and the limitations of scaling in practice.

III. Design

A. Parallelisation Strategy

The grid needs to be partitioned into chunks that different threads can process. Firstly, the dependency among cells should be identified. According to the rules, if a cell contains a healthy tree in step (i), its state in step (i+1) depends on its 8 neighbours. The grid can be divided into chunks of rows, or chunks of columns, or even smaller square subgrids.

Assuming the grid is divided into smaller subgrids, each subgrid will in turn depend on its 8 neighbouring subgrids to compute its state in the next step. There is dependency in the horizontal and vertical directions. For my solution, I decided to divide the grid only horizontally, producing chunks of rows. This keeps the dependency only in the vertical direction. A chunk of rows depends on the chunk below and above it to compute its next state.

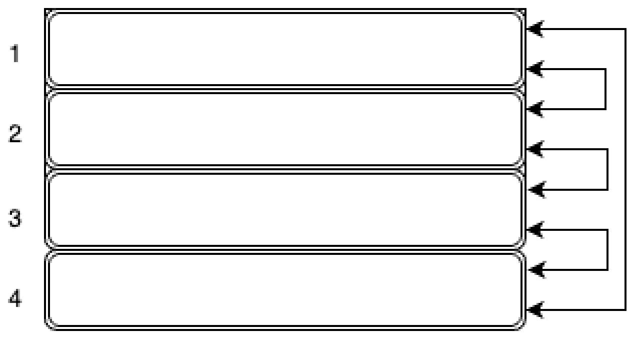


Figure 1. Dependency amongst chunks

Assume the grid is divided into 4 row-wise chunks (Figure 1). Chunk 1 depends on chunks 2 and 4, chunk 2 depends on chunks 1 and 3 and so on forming a cyclic dependency. It can be inferred that no chunk can update its next state until all other chunks have computed their next state. Each chunk worker can compute and store its next state in local storage in parallel and once all chunk workers have computed their next state, each can update its state in the shared grid storage in parallel. Storing the next state in intermediate local storage ensures that neighbouring chunks can access data safely from the shared grid. There is a need for a synchronisation mechanism that ensures that all workers reach a certain checkpoint in a simulation step before any of them can proceed. There are two checkpoints - first, all workers should compute their state and store it locally before updating the shared grid; secondly, all workers should update the shared grid before computing the state for the next simulation step (Figure 2).

The chunk size is taken to be 25 rows. For a 500*500 grid, there will be 500 rows /25 rows = 20 chunks, each consisting of 25 rows * 500 columns = 12500 cells. Hence the parallelisable part of the code ($F_{enhanced}$) for a 500*500 grid = (500 - 25)/500 = 0.95 and ($F_{enhanced}$) for a 1000*1000 grid = (1000-25)/1000 = 0.975.

When implementing the solution, the number of threads available is much less than the number of chunks. Hence, contiguous chunks are allotted to one thread to further reduce the inter-chunk dependency among threads. The implementation divides the rows among threads as shown in (Code Snippet 1). For a 500*500 grid and 8 threads, the rows will be divided such that 4 threads will handle 63 rows each and 4 threads will handle 62 rows each (4*63 + 4*62 = 500).

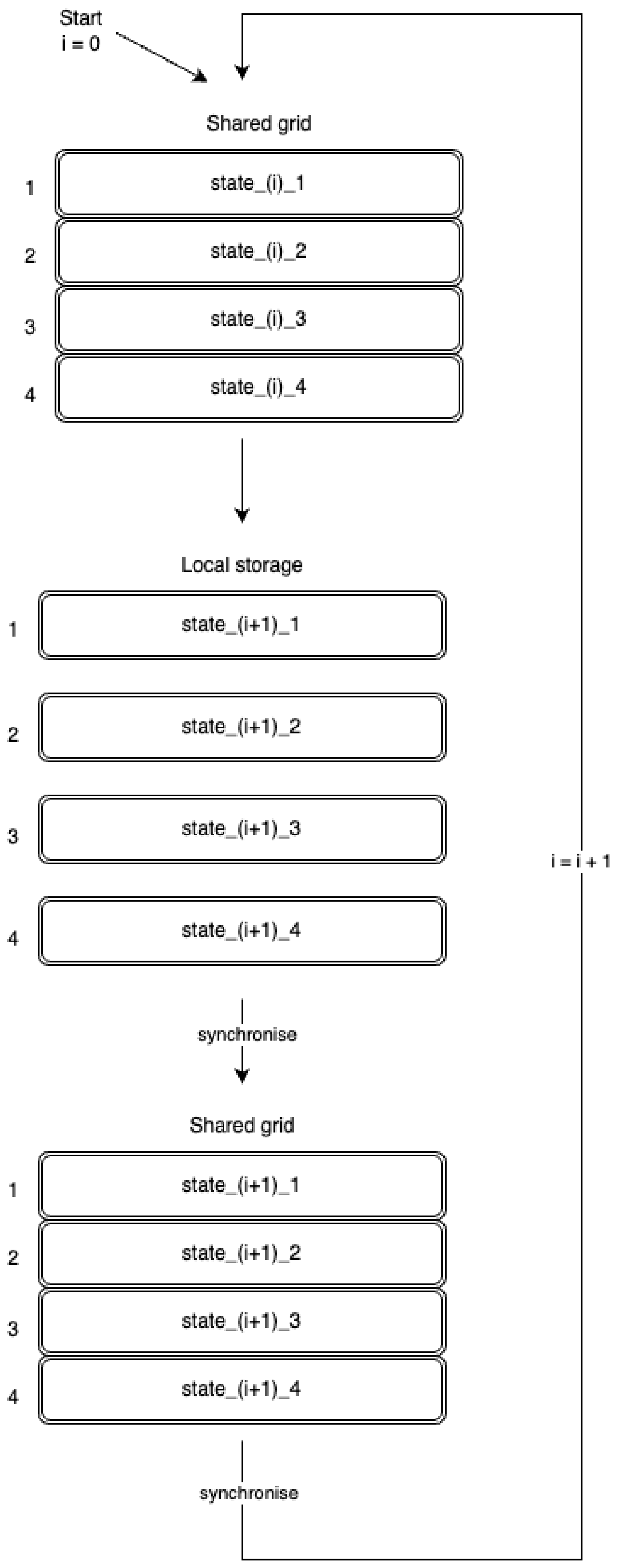
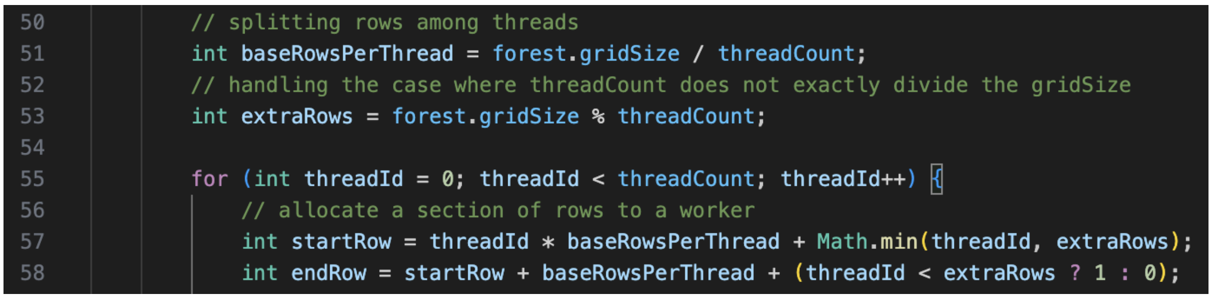


Figure 2. Parallel design



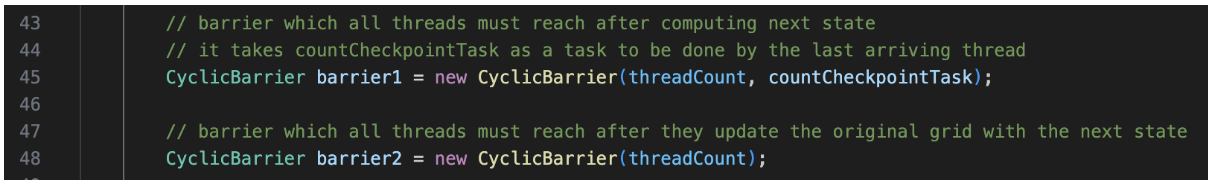
```java
// splitting rows among threads
int baseRowsPerThread = forest.gridSize / threadCount;
// handling the case where threadCount does not exactly divide the gridSize
int extraRows = forest.gridSize % threadCount;

for (int threadId = 0; threadId < threadCount; threadId++) {
    // allocate a section of rows to a worker
    int startRow = threadId * baseRowsPerThread + Math.min(threadId, extraRows);
    int endRow = startRow + baseRowsPerThread + (threadId < extraRows ? 1 : 0);
```

Code Snippet 1. Dividing rows among threads (ParallelForestFireSimulation.java)

The solution demonstrates task parallelism by dividing the simulation into distinct tasks, such as processing the state updates for each grid section and checkpointing the global cell counts (discussed ahead). It incorporates data parallelism by partitioning the 2D grid into chunks of rows, enabling multiple threads to process these partitions independently.

B. Synchronisation

As discussed in Figure 2., I need a mechanism through which each thread can wait until all other reads reach the same point in the execution. Java provides a synchronisation aid, CyclicBarrier, that allows a set of threads to all wait for each other to reach a common barrier point. To initialise a cyclic barrier, I need to provide the number of parties/threads that must wait for each other (Code Snippet 2). Each thread will block on barrier.await() until all parties reach the barrier, another thread interrupts or the await times out. The barrier is called cyclic because it can be reused after the waiting threads are released. This is particularly useful because I want to synchronise the threads for each simulation step.

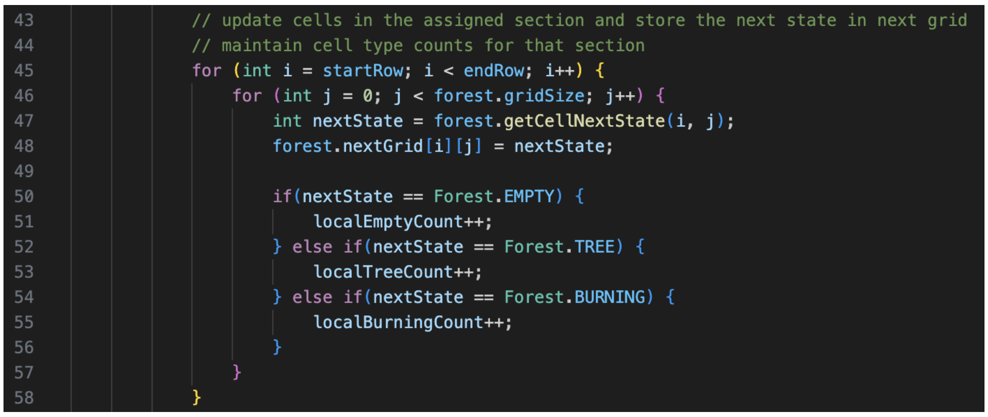


```java
// barrier which all threads must reach after computing next state
// it takes countCheckpointTask as a task to be done by the last arriving thread
CyclicBarrier barrier1 = new CyclicBarrier(threadCount, countCheckpointTask);

// barrier which all threads must reach after they update the original grid with the next state
CyclicBarrier barrier2 = new CyclicBarrier(threadCount);
```

Code Snippet 2. Initialising the CyclicBarrier (ParallelForestFireSimulation.java)

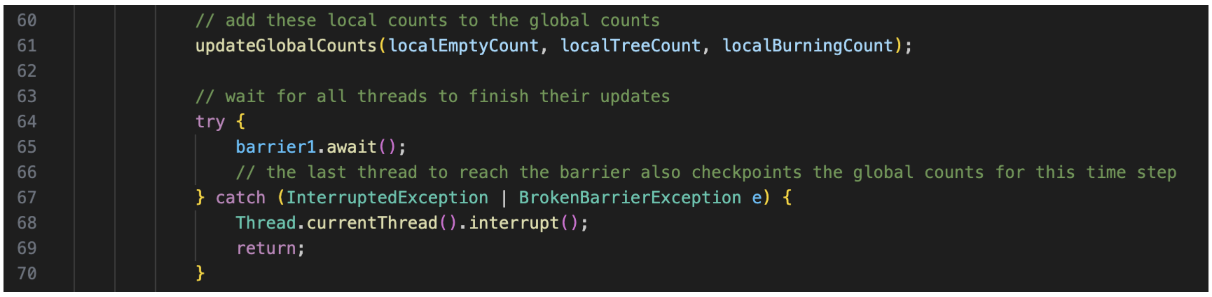
Inside the Worker.run() function, each thread computes the state of its allotted rows based on the neighbouring rows in the shared grid and stores this computed state in nextGrid. Simultaneously, it also keeps local counts of the cell types in this computed state (Code Snippet 3).

The key synchronisation concept here is the barrier.await() function which ensures that all threads have computed and stored their next state in nextGrid as well as updated the global counts with their local cell type counts. This ensures that no thread updates the shared grid before some other thread reads the shared grid at that simulation step (Code Snippet 4).



```java
            // update cells in the assigned section and store the next state in next grid
            // maintain cell type counts for that section
            for (int i = startRow; i < endRow; i++) {
                for (int j = 0; j < forest.gridSize; j++) {
                    int nextState = forest.getCellNextState(i, j);
                    forest.nextGrid[i][j] = nextState;

                    if(nextState == Forest.EMPTY) {
                        localEmptyCount++;
                    } else if(nextState == Forest.TREE) {
                        localTreeCount++;
                    } else if(nextState == Forest.BURNING) {
                        localBurningCount++;
                    }
                }
            }
```

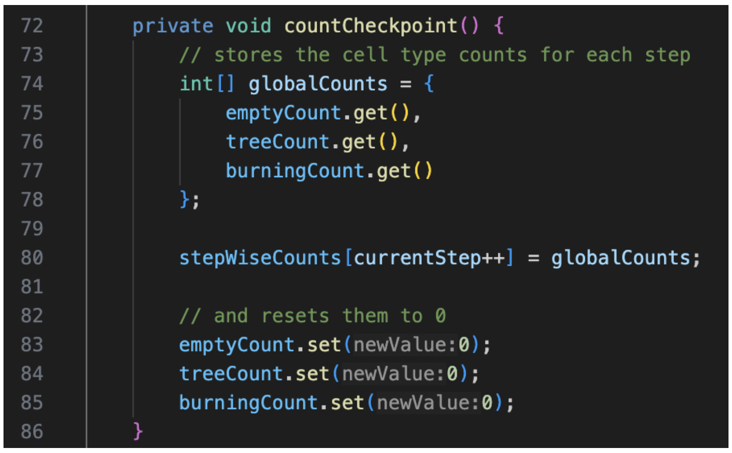
Code Snippet 3. Computing next state and local cell type counts (Worker.java)



```java
            // add these local counts to the global counts
            updateGlobalCounts(localEmptyCount, localTreeCount, localBurningCount);

            // wait for all threads to finish their updates
            try {
                barrier1.await();
                // the last thread to reach the barrier also checkpoints the global counts for this time step
            } catch (InterruptedException | BrokenBarrierException e) {
                Thread.currentThread().interrupt();
                return;
            }
```

Code Snippet 4. Updating global counts and waiting at the barrier (Worker.java)

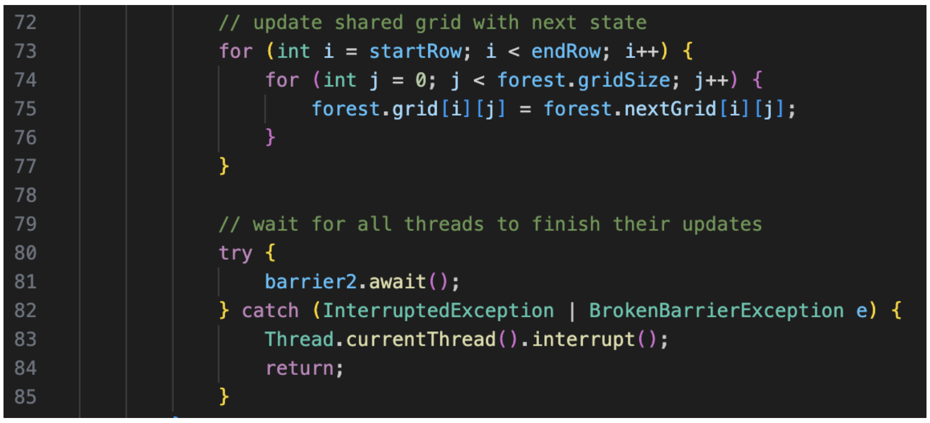
The CylicBarrier also takes in an optional Runnable task that is run once per barrier point, after the last thread in the party arrives, but before any threads are released. Once the last thread reaches the barrier point, the countCheckpointTask (Code Snippet 5) is run which stores the global cell type counts at this simulation step. After this, the threads waiting at the barrier are released and they can proceed further. The barrier is automatically reset for the next simulation step.



```java
    private void countCheckpoint() {
        // stores the cell type counts for each step
        int[] globalCounts = {
            emptyCount.get(),
            treeCount.get(),
            burningCount.get()
        };

        stepWiseCounts[currentStep++] = globalCounts;

        // and resets them to 0
        emptyCount.set(newValue:0);
        treeCount.set(newValue:0);
        burningCount.set(newValue:0);
    }
```

Code Snippet 5. Storing global cell type counts (ParallelForestFireSimulation.java)

Each thread now needs to update the shared grid with the next state that is stored in nextGrid. There is no dependency amongst threads to perform this action. However, there is again the need to synchronise the threads such that none of the threads can move to the next simulation step unless all of the threads have updated the shared grid for the current simulation step. I utilise a second cyclic barrier to ensure this (Code Snippet 6). Once all threads have updated their allocated rows in the shared grid, the barrier is tripped and all threads can proceed to the next simulation step. (I could have utilised barrier1 here if it did not take the countCheckPoint task as the optional Runnable parameter)

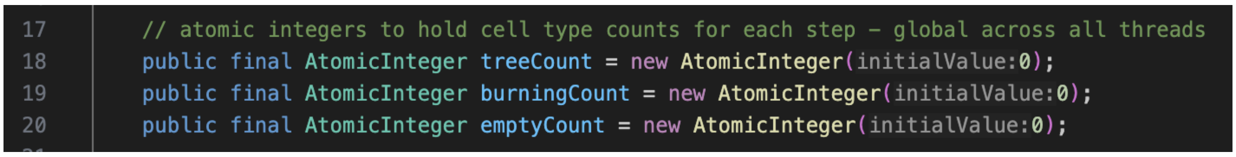


```java
            // update shared grid with next state
            for (int i = startRow; i < endRow; i++) {
                for (int j = 0; j < forest.gridSize; j++) {
                    forest.grid[i][j] = forest.nextGrid[i][j];
                }
            }

            // wait for all threads to finish their updates
            try {
                barrier2.await();
            } catch (InterruptedException | BrokenBarrierException e) {
                Thread.currentThread().interrupt();
                return;
            }
```

Code Snippet 6. Second barrier (Worker.java)

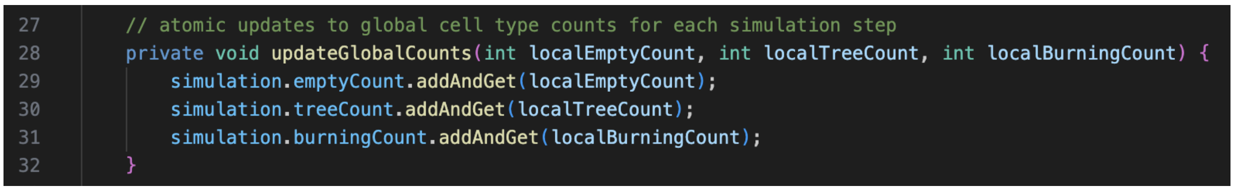
Other alternative synchronisation mechanisms in Java include CountDownLatch and Phaser. CountDownLatch allows one or more threads to wait until a set of operations being performed in other threads completes. However, it is a one-shot phenomenon and it cannot be reset after the count reaches zero. For my application, the synchronisation mechanism has to be re-used for each simulation step. The Phaser primitive offers more flexible usage but that is not needed for my use case.

When computing the next simulation state, each thread maintains local counters for cell types. After it has computed the next state for all its cells, it calls updateGlobalCounts to add these counts to the global counts. Since all threads will be updating the global counts, I need to employ synchronisation so that none of these updates is missed. Java offers several primitives like mutexes, semaphores and synchronized that can help in this situation; however, they also come with overheads and performance penalties. I decided to use Java's AtomicInteger (Code Snippet 7) to ensure that updates to the global counts are thread-safe. AtomicInteger is implemented using the C-A-S (CompareAndSwap or CompareAndSet) operation which is a primitive hardware operation. Its addAndGet method (Code Snippet 8) provides an efficient way to manage updates to the high-contention global counts without the overhead of locking. Since CAS is a non-blocking operation, the threads will not be put to sleep which reduces the overhead of context switching and thread management.



```java
    // atomic integers to hold cell type counts for each step - global across all threads
    public final AtomicInteger treeCount = new AtomicInteger(initialValue:0);
    public final AtomicInteger burningCount = new AtomicInteger(initialValue:0);
    public final AtomicInteger emptyCount = new AtomicInteger(initialValue:0);
```

Code Snippet 7. Atomic integers for global cell type counts (ParallelForestFireSimulation.java)

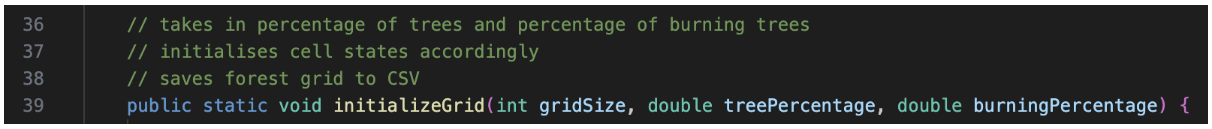


```java
    // atomic updates to global cell type counts for each simulation step
    private void updateGlobalCounts(int localEmptyCount, int localTreeCount, int localBurningCount) {
        simulation.emptyCount.addAndGet(localEmptyCount);
        simulation.treeCount.addAndGet(localTreeCount);
        simulation.burningCount.addAndGet(localBurningCount);
    }
```

Code Snippet 8. Thread safe updates without explicit synchronisation overhead (Worker.java)

C. Code Structure

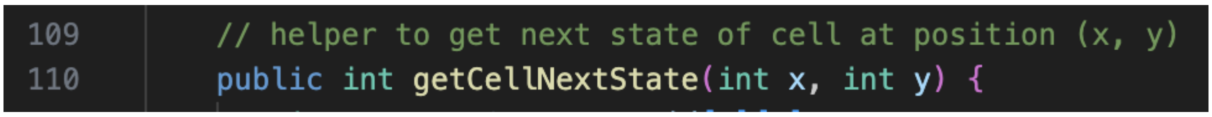
The Forest class provides a function to initialise the grid, taking in the percentage of cells that should be trees and the percentage of trees that should be burning at the start of the simulation. The initialisation of the forest affects the workload - depending on the position and count of healthy trees and burning trees. As I do performance analysis of forest fire simulation on serial and parallel implementations, I wanted to reduce the variability in workload. This led me to initialise the forest, store it in a CSV file and use the same forest to run all benchmarking tests. Hence, for this solution, the forest initialisation is not considered part of the simulation process. I use 2 grids, a 500*500 grid and a 1000*1000 grid. For both grids, initially, 70% of the grid cells are trees and the rest are empty. For both grids, initially, 0.05% of the trees are burning.



```java
    // takes in percentage of trees and percentage of burning trees
    // initialises cell states accordingly
    // saves forest grid to CSV
    public static void initializeGrid(int gridSize, double treePercentage, double burningPercentage) {
```

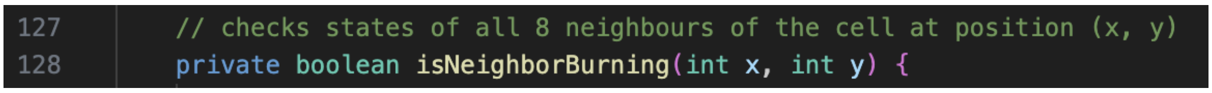
Code Snippet 9. Initialise and store grid (Forest.java)

The Forest class provides helper functions to get the state of the cell in the next step based on its neighbouring cells in the shared grid (Code Snippet 10). Our parallel solution design and synchronisation mechanisms ensure that no thread can update the shared grid before every thread has computed its next state locally, thus this helper function can access the shared grid without the need for any synchronisation mechanisms. The Forest class also implements a helper function to check all the neighbouring cells considering the boundary and corner conditions and treating the shared grid like a torus (Code Snippet 11).



```java
    // helper to get next state of cell at position (x, y)
    public int getCellNextState(int x, int y) {
```

Code Snippet 10. Get the cell's next state (Forest.java)

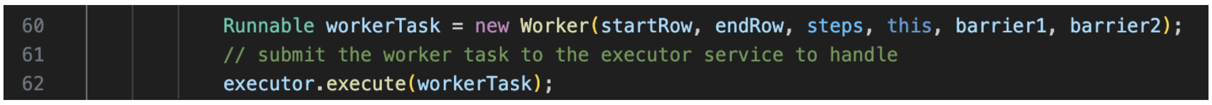


```java
    // checks states of all 8 neighbours of the cell at position (x, y)
    private boolean isNeighborBurning(int x, int y) {
```

Code Snippet 11. Check if any neighbour is burning (Forest.java)

The Worker class is a Runnable with the run() function containing the functionality run by each thread. Each thread computes its next state in nextGrid and updates the global cell type counts. After the first barrier is tripped, each thread updates the shared grid with the next state. After the second barrier is tripped, all threads move to the next iteration and this continues till all steps of simulation are completed.

The ParallelForestFireSimulation is the orchestrator for initialising the threads and synchronisation mechanisms and submits tasks to the threads. Java's ExecutorService takes the responsibility to manage threads and submit tasks. The executor service is initialised with the thread count. The 2 cyclic barriers are also initialised in this class. The grid is divided into chunks of rows based on the thread count. For each chunk/thread, a worker object is initialised with the startRow, endRow, steps and the cyclic barriers. This Runnable is submitted to the executor service to be assigned to a thread (Code snippet 12).



```
            Runnable workerTask = new Worker(startRow, endRow, steps, this, barrier1, barrier2);
            // submit the worker task to the executor service to handle
            executor.execute(workerTask);
```

Code Snippet 12. Submitting tasks to the executor service (ParallelForestFireSimulation.java)

The SerialForestFireSimulation performs the next state computation and grid updation serially. It also accumulates the cell type counts for each simulation step. This code is primarily used to compute the speedup obtained from our parallel implementation.

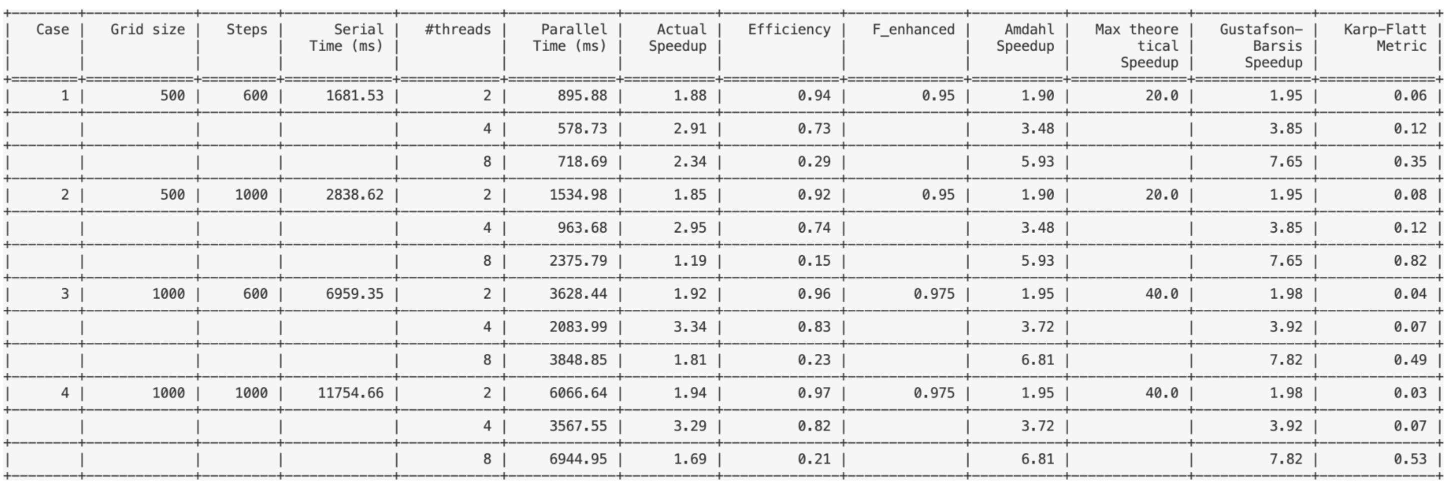
Finally, the Driver class initialises the simulation object based on whether I want to run the serial or parallel code and calls the simulate() function. It also records the time taken for the simulation and writes it to a file to be analysed.

The growth probability and burn probability affect the workload that the simulation needs to perform. A low growth probability means empty cells often remain empty and there is less regrowth, however, a higher growth probability means the forest recovers quickly and there is more load on the simulation. Similarly, a lower burn probability means less chance of a healthy tree being affected whereas a high burn probability indicates that healthy trees can be impacted easily causing more workload. For measuring performance, the growth probability is set to 0.15 and the burn probability is set to 0.25. These values fall in the middle of the suggested range ensuring the simulation workload is neither too idle nor too heavy.

IV. Performance Analysis

A. Execution Time

Using Java's built-in System.nanoTime() function, the execution time for the serial and parallel versions of the implementation are recorded. The simulation is run for two grids, 500*500 and 1000*1000 with 600 steps and 1000 steps for each. The program is run for 3 iterations and the average execution time is considered to minimise the effect of any irregularities or fluctuations due to system load or other factors. Table 1 captures the execution times in milliseconds.



Case	Grid size	Steps	Serial Time (ms)	#threads	Parallel Time (ms)	Actual Speedup	Efficiency	F_enhanced	Amdahl Speedup	Max theoretical Speedup	Gustafson-Barsis Speedup	Karp-Flatt Metric
1	500	600	1681.53	2	895.88	1.88	0.94	0.95	1.90	20.0	1.95	0.06
				4	578.73	2.91	0.73		3.48		3.85	0.12
				8	718.69	2.34	0.29		5.93		7.65	0.35
2	500	1000	2838.62	2	1534.98	1.85	0.92	0.95	1.90	20.0	1.95	0.08
				4	963.68	2.95	0.74		3.48		3.85	0.12
				8	2375.79	1.19	0.15		5.93		7.65	0.82
3	1000	600	6959.35	2	3628.44	1.92	0.96	0.975	1.95	40.0	1.98	0.04
				4	2083.99	3.34	0.83		3.72		3.92	0.07
				8	3848.85	1.81	0.23		6.81		7.82	0.49
4	1000	1000	11754.66	2	6066.64	1.94	0.97	0.975	1.95	40.0	1.98	0.03
				4	3567.55	3.29	0.82		3.72		3.92	0.07
				8	6944.95	1.69	0.21		6.81		7.82	0.53

Table 1. Execution times, speedup, efficiency and theoretical performance metrics

For all cases, the execution time is almost halved from the serial version to the 2-thread version. The 4-thread version performs the simulation in even less time. However, the 8-thread version performs worse, often taking more execution time than the 2-thread version. There is a need to analyse why the 8-thread version exhibits this poor result.

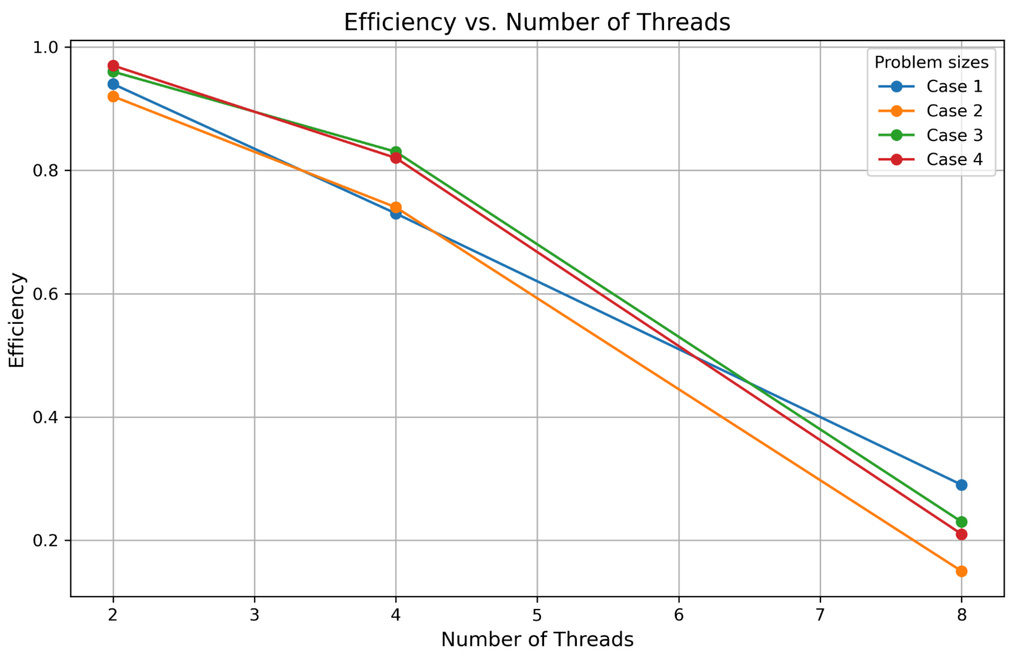
B. Actual Speedup & Efficiency

I divide the serial execution time with the parallel execution time to get the actual speedup. The efficiency is the speedup divided by the number of threads (Plot 1).

$$ActualSpeedup = \frac{serialExecutionTime}{parallelExecutionTime}$$

$$Efficiency = \frac{ActualSpeedup}{\#threads}$$

For all four problem sizes, the simulation with 2 threads achieves speedup close to 2 and efficiency close to 1. There are significant gains in switching from the sequential version to the 2-thread version. For the simulation with 4 threads, the efficiency decreases, aligning with diminishing returns tendency, and the speedup observed is close to 3. However, the simulation with 8 threads performs worse than the 2-thread version for three of the above four cases with extremely poor efficiency. The 2 and 4-thread versions show higher speedup for the larger problem size (case 3 and case 4) indicating better utilisation of resources and higher efficiency for larger problems.



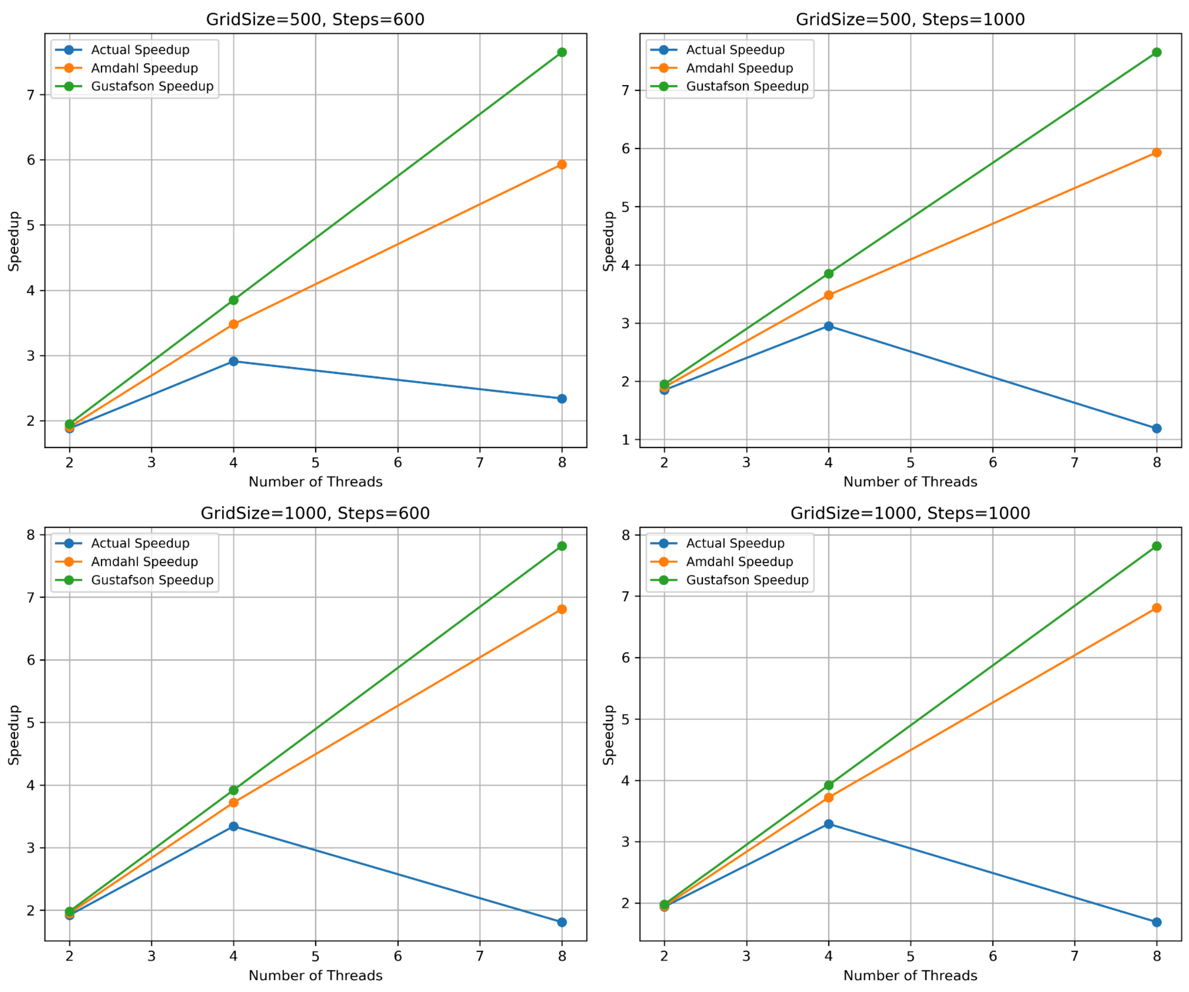
Plot 1. Efficiency vs thread count

C. Amdahl's Theoretical Speedup & Maximum Speedup

Amdahl's Law provides a formula to find the theoretical improvement in the performance of a system (i.e. speedup in latency) when only part of the system is improved. As discussed earlier, our solution design considers chunks of 25 rows for serial execution by a thread. For a 500*500 grid, $F_{enhanced} = 0.95$ and for a 1000*1000 grid, $F_{enhanced} = 0.975$.

$$OverallSpeedup = \frac{oldExecutionTime}{newExecutionTime} = \frac{1}{(1 - F_{enhanced}) + (\frac{F_{enhanced}}{\#threads})}$$

The theoretical speedup closely matches the actual speedup for the 2 and 4-thread versions, indicating efficient scaling (Plot 2). However, at 8 threads, the actual speedup falls significantly below the Amdahl speedup, suggesting that parallelisation overheads or contention dominate, diminishing the effectiveness of adding more threads.



Plot 2. Comparison of actual speedup, Amdahl's speedup and Gustafson-Barsis' speedup

Suppose that infinite resources can be utilised. In that case, the contribution of the parallel portion of the program to the speedup becomes 0 and we are limited only by the serial section. Table 1. shows the maximum theoretical speedup for each problem size. It highlights the theoretical upper limit for scaling, but practical constraints like synchronisation and thread management overhead prevent reaching these values.

$$MaxSpeedup = \lim_{\#thread \to \infty} \frac{1}{1 - F_{enhanced}}$$

D. Gustafson-Barsis' Speedup

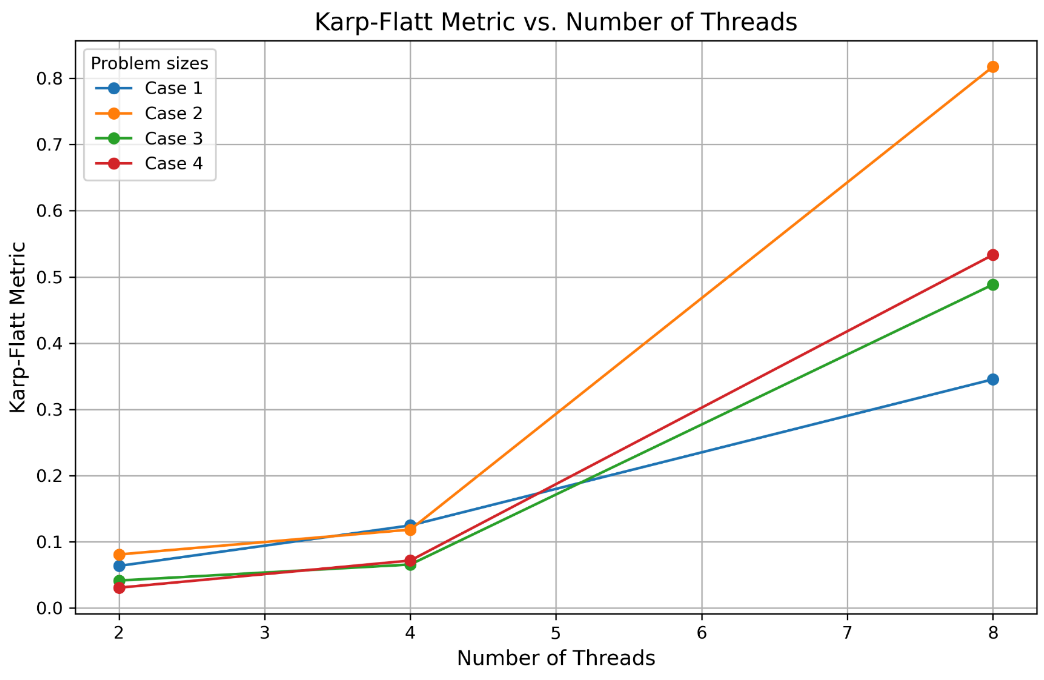
Amdahl's law applies to strong scaling as it assumes that the problem size remains fixed as we add more resources. Gustafson-Barsis Law addresses this limitation by considering that as the number of processors increases, the problem size typically grows (weak scalability). It assumes that as more processors are added, the size of the parallel task increases proportionally. This usually causes the Gustafson-Barsis speedup to be scaled up than the Amdahl speedup as observed in Plot 2. Gustafson-Barsis speedup remains higher than actual speedup, especially for the higher thread count, demonstrating the potential for better performance under weak scaling. However, as problem size and thread count increase, diminishing returns appear due to the inability to fully utilise additional threads where overhead or contention limits performance gains.

$$Speedup = (1 - F_{enhanced}) + (F_{enhanced} * \#threads)$$

E. Karp-Flatt Metric

The Karp-Flatt metric is used to identify how much of the performance bottleneck is due to the serial parts of the code and overhead introduced by parallelisation. A value close to 0 indicates that the code is very parallelisable with minimal overhead whereas a value close to 1 indicates that either there are non-parallelisable portions of the algorithm or the parallelisation overhead is the bottleneck.

$$\epsilon(\text{bottleneck from sequential code}) = \frac{\frac{1}{ActualSpeedup} - \frac{1}{\#threads}}{1 - \frac{1}{\#threads}}$$



Plot 3. Karp-Flatt metric across #threads

For the 2 and 4-thread versions, the value is relatively low signalling that parallelisation is effective (Plot 3). For the 8-thread version, the metric shows a significant increase. As I add more threads, the Karp-Flatt metric increases indicating that parallel overhead contributes to the poor speedup. If it had remained the same as I increased resources, then the poor performance could be attributed to the serial section of the algorithm. There are more parties involved in the cyclic barrier and each thread has to wait for a larger number of threads to reach the barrier before it can proceed. Also, the global counts are now accessed by numerous threads increasing contention. These synchronisation mechanisms increase the overhead as the number of threads is increased.

F. Machine specifications

CPU Model: Apple M3 Pro; Number of available cores: 11.
I ran experiments with the number of threads = 2, 4, 8; the number of threads does not exceed the number of cores demonstrating the potential for parallelisation without introducing thread contention. Background programs running on the system can affect the thread management and scheduling process.

V. Conclusion

This report explored the parallelisation of a forest fire simulation to improve computational performance on a grid-based probabilistic model. By partitioning the grid into horizontal chunks and employing Java's cyclic barriers for synchronisation, I ensured correctness in state transitions while enabling parallel execution. The use of atomic operations for global counters minimised contention during high-contention updates, reducing synchronisation overhead.

Challenges include weighing the different concurrency primitives available and selecting the best synchronisation mechanisms to ensure correctness along with efficiency. Another aspect which required tuning was setting the forest probabilities (growth and burn) to ensure that the simulation workload was balanced.

Performance analysis revealed significant speedups and high efficiency with 2 and 4 threads, demonstrating the viability of the parallel approach. However, with 8 threads, synchronisation overhead and contention reduced performance gains, highlighting the limitations of scaling. Strong scaling shows diminishing returns in speedup as thread count increases. However, even with weak scaling, excessive thread counts encounter overhead and diminishing returns.

Metrics such as Amdahl's and Gustafson-Barsis's speedup were employed to understand the trade-offs between parallelisation and overhead. The Karp-Flatt metric suggested that overhead becomes a dominant factor at higher thread counts. To conclude, parallelisation offers clear benefits up to a certain point but is constrained by diminishing returns and contention.

References

[1] Lecture Slides

[2] Parallel Computing Stanford CS149, Fall 2023 Lecture 4: Parallel Programming Basics. (n.d.). Available at: https://gfxcourses.stanford.edu/cs149/fall23content/media/progbasics/04_progbasics.pdf [Accessed 23 Nov. 2024].

[3] Oracle.com. (2024). Java Platform SE 8. [online] Available at: https://docs.oracle.com/javase/8/docs/api/index.html?java/util/concurrent/CyclicBarrier.html [Accessed 23 Nov. 2024].