Parallel Wildfire Spread Simulation

Parallel Functional Programming Joshua Brown

Introduction & Problem

Wildfire simulation is a computationally intensive problem with significant real-world applications in disaster preparedness, forest management, and environmental planning. The challenge lies in modeling complex fire spread dynamics across large terrain grids while accounting for multiple environmental factors such as wind direction, terrain slope, fuel levels, and vegetation types.

The goal is to parallelize a wildfire spread simulation using cellular automaton in Haskell, comparing two distinct parallelization strategies to determine which trade-offs work best for this problem domain.

- **Wildfire Simulation Problem -** given a 2D grid representing terrain (forest, grassland, water), model fire spread based on environmental factors including wind speed/direction, terrain elevation (uphill spreads faster), fuel levels, and vegetation types. Each cell calculates ignition probability based on its burning neighbors using physics-based rules.
- **Cellular Automaton -** synchronous time-step evolution where all cells must read from the current grid state and write to a new grid simultaneously. States progress: Unburned → Burning → Burned. This synchronous update requirement creates the primary parallelization challenge - how to distribute work while maintaining correctness.

Runs are benchmarked on the following machine:
Machine: Apple M3 Pro
Architecture: ARM64 (aarch64)

CPU(s): 11 (8 performance + 3 efficiency)
Compiler: GHC 9.6.6 with -O2 optimization
Stack Resolver: lts-22.33

Dataset & Problem Structure

We used programmatically generated grids with varying sizes and terrain configurations. A small 20×20 grid (400 cells) was used to quickly verify the implementation's correctness by visually inspecting fire spread patterns.

The main dataset for assessing performance is a uniform forest grid of 500×500 cells (250,000 cells total) with center ignition. This problem size is sufficiently large to keep execution CPU-heavy (serial implementation takes ~118 seconds) without trading off development velocity. The grid burns completely in 253 time steps, allocating 398GB of temporary data during execution due to functional immutability.

Additional test scenarios verify physical correctness: wind-driven fire (spreads faster downwind), river barriers (water blocks spread), mixed terrain types (different burn rates), and uphill elevation gradients (fire accelerates upward by 30% due to heat rising).

Serial Implementation

The serial implementation establishes correctness and provides the baseline for speedup measurements. Each time step follows the standard cellular automaton pattern:

- Read all cell states from current grid (immutable read-only access)
- For each cell, examine 8-neighbor Moore neighborhood to find burning cells
- Calculate ignition probability: P_ignite = P_base × F_fuel × F_wind × F_slope × F_random
- Compare probability against threshold (0.35) with random number to determine ignition
- Construct entirely new grid with all updates applied simultaneously (functional immutability)

Critical correctness requirement: synchronous updates mean all cells must evaluate based on the SAME time step's state. If cell A ignites during step N, it should not affect cell B's calculation during step N - only in step N+1. The functional approach naturally supports this with immutable grids - all threads read from original grid, write to new grid, then atomically swap grids between steps.

```
simulationStep :: SimConfig -> Grid -> RandomGen -> (Grid,
RandomGen)
simulationStep config originalGrid gen =
```

```
 let positions = [(r,c) | r <- [0..rows-1], c <-
[0..cols-1]]
     (updates, gen') = foldl processCell ([], gen)
positions
     newGrid = foldl applyUpdate originalGrid updates
 in (newGrid, gen')
```

The serial baseline processed 500×500 grids in 117.68 seconds with 64.4s MUT time (actual computation) and 53.2s GC time (45.2% garbage collection overhead). This high GC cost comes from allocating 398GB of temporary grid copies - the price of functional immutability.

1. Algorithm Explanation

The cellular automaton algorithm works as follows:

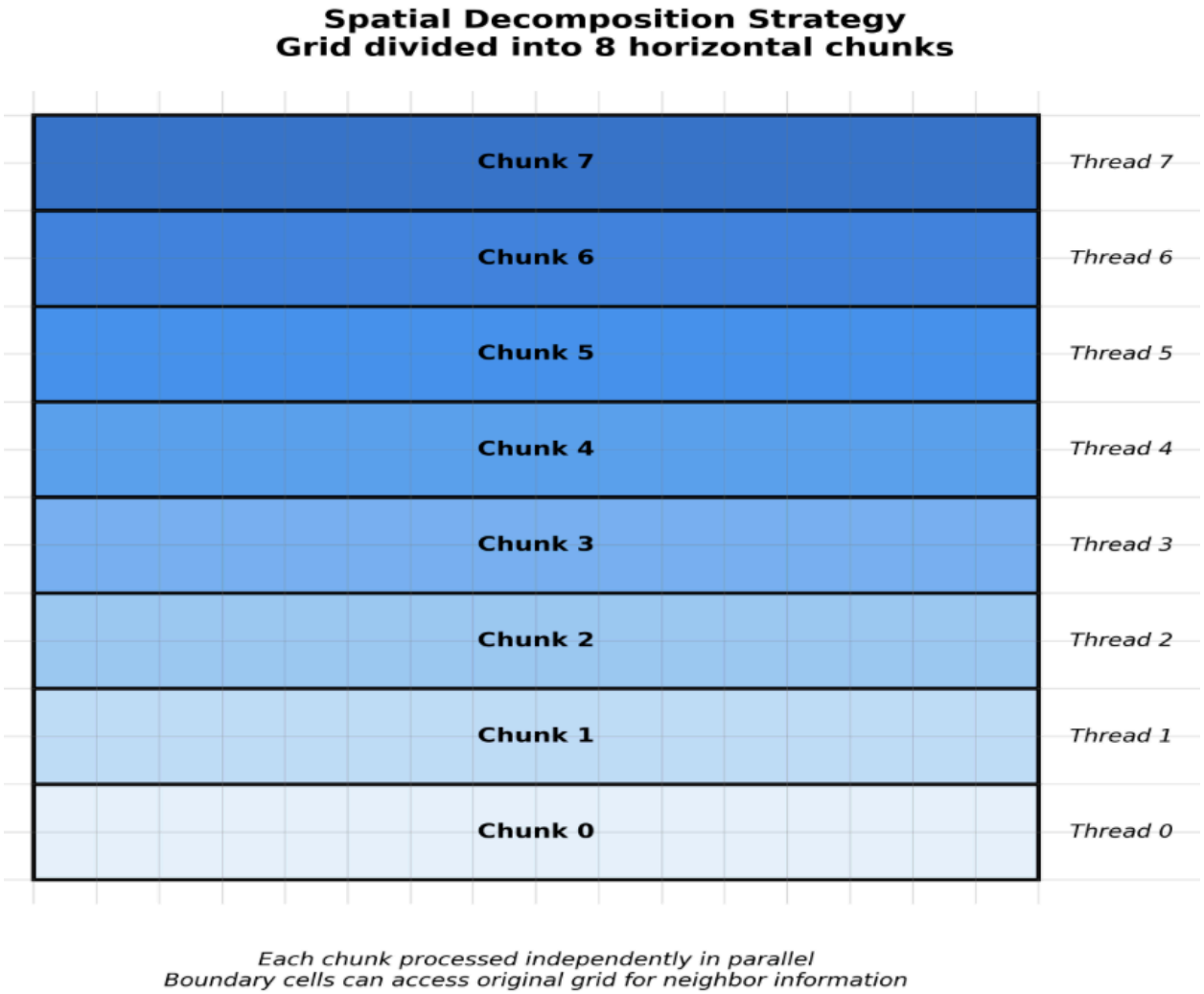
1. For each time step t:
  a. Create empty grid for time t+1
  b. For each cell (r,c) in grid at time t:
    - If cell is Burning: transition to Burned
    - If cell is Unburned:
      * Check all 8 neighbors
      * For each neighbor that is Burning:
        - Calculate ignition probability P
        - P = base_prob × fuel_factor × wind_factor × slope_factor × random
        - If P > 0.35: transition to Burning
  c. Replace grid(t) with grid(t+1)

This ensures synchronous updates - all cells read from the same time step."

Parallelization Approaches & Challenges

The goal was to speed up single simulations rather than run multiple simulations in parallel. Two main approaches were explored:

Strategy 1: Spatial Decomposition



Spatial decomposition divides the grid into 8 horizontal bands (matching core count), with each thread processing one contiguous chunk of rows. The key insight: within a chunk, cells are spatially local - neighboring cells are adjacent in memory, leading to good cache utilization.

Implementation Details

```
simulationStepParallel :: SimConfig -> Grid -> RandomGen
                       -> (Grid, RandomGen)
simulationStepParallel config grid gen =
 let allPositions = [(r,c) | r <- [0..rows-1], c <-
[0..cols-1]]
```

```
      numChunks = 8  -- Match core count
      chunks = chunksOf (length allPositions `div`
numChunks)
                          allPositions
      -- Split RNG deterministically for reproducibility
      gens = splitGen gen (length chunks)
      -- parMap creates sparks for parallel execution
      results = parMap rdeepseq
                  (\(chunk, g) -> processChunk config grid
chunk g)
                  (zip chunks gens)
      newGrid = foldl mergeChunkResult grid results
 in (force newGrid, last gens)  -- force ensures deep
evaluation
```

How It Works:

1. Divide 500×500 grid (250,000 cells) into 8 horizontal chunks
2. Each chunk contains ~62-63 rows (~31,250 cells)
3. Use parMap rdeepseq to create 8 sparks (one per chunk)
4. GHC's work-stealing scheduler distributes sparks across 11 HECs (cores)
5. Merge chunk results into new grid

Why 8 Chunks? (Experimental Justification)

To determine the optimal chunk count, I systematically tested 4, 8, 12, and 16 chunks on the 500×500 grid with 11 cores:

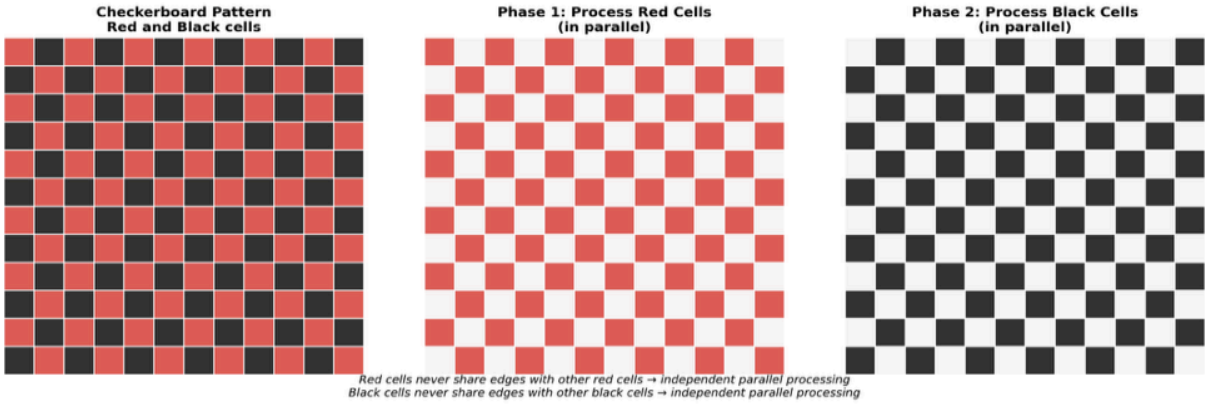
Key Implementation Decisions:

- **Chunk Count = Core Count:** Using 8 chunks for 8 cores provides sufficient work per spark to amortize parallelization overhead. Too many small chunks create excessive spark management cost; too few large chunks limit parallelism.

Chunks	Wall Time	Speedup	Spark Statistics
4	54.23s	2.17×	5,064 sparks, 4,021 converted (79%), 819 fizzled
8	53.29s	2.21×	10,136 sparks, 8,817 converted (87%), 815 fizzled
12	53.98s	2.18×	16,588 sparks, 14,877 converted (90%), 853 fizzled
16	52.78s	2.23×	20,894 sparks, 18,986 converted (91%), 827 fizzled

- **rdeepseq Strategy:** Forces complete evaluation of chunk results before proceeding. Without deep evaluation, Haskell's lazy evaluation would create thunks instead of doing actual work, causing sparks to fizzle. NFData instances on all types enable this.

Strategy 2: Red-Black Checkerboard



The red-black checkerboard partitions cells by (row + col) mod 2 parity. Crucially, red cells never share edges with other red cells - all their neighbors are black. Similarly, black cells only have red neighbors. This independence enables true parallel processing within each color set without any synchronization.

The algorithm runs in two phases per time step: First, process all red cells in parallel (safe because red cells don't depend on each other's new states - they only read from black neighbors). Then process all black cells in parallel using the grid with updated red cells. This two-phase approach maintains correctness while enabling parallelism.

Implementation Details

```
simulationStepRedBlack :: SimConfig -> Grid -> RandomGen
                       -> (Grid, RandomGen)
simulationStepRedBlack config grid gen =
 let -- Partition by parity
     redCells = [(r,c) | r <- [0..rows-1], c <-
[0..cols-1],
                         (r + c) `mod` 2 == 0]
```

```
      blackCells = [(r,c) | r <- [0..rows-1], c <-
  [0..cols-1],
                            (r + c) `mod` 2 == 1]
      -- Split RNG for two phases
      (genRed, genBlack) = split gen
      -- Phase 1: Process red cells in parallel
      (grid1, _) = processPositionsParallel config grid
                     genRed redCells
      -- Phase 2: Process black cells in parallel
      (grid2, genFinal) = processPositionsParallel config
  grid1
                              genBlack blackCells
   in (force grid2, genFinal)
```

For Red-Black Checkerboard:

I partition cells based on (row + col) mod 2:
- Red cells: (r+c) is even → cells at (0,0), (0,2), (1,1), (2,0), etc.
- Black cells: (r+c) is odd → cells at (0,1), (1,0), (1,2), (2,1), etc.

Key property: No two red cells are adjacent (they only touch black cells).
This allows two parallel phases:

Phase 1: Process all red cells in parallel
- 8 chunks of red cells
- parMap rdeepseq across chunks
- No conflicts because red cells don't share edges

Phase 2: Process all black cells in parallel
- 8 chunks of black cells
- parMap rdeepseq across chunks
- No conflicts because black cells don't share edges

Total sparks created: 16 (8 per phase)

Why This Works:

In a Moore neighborhood (8-connected grid), a cell at (r,c) has neighbors at positions differing by ±1 in row and/or column. If (r+c) is even, then (r±1, c) and (r, c±1) are all odd, and (r±1, c±1) are all even. So red cells (even parity) only neighbor black cells (odd parity) at the cardinal directions, and other red cells at diagonals. However, cellular automaton fire spread typically uses 8-neighborhood where diagonal neighbors also matter. The key insight: while red cells DO have red diagonal neighbors, the dependence is read-only - all red cells read from the SAME grid state simultaneously, so there's no write conflict. They can all update independently.

Implementation Challenges & Solutions

Challenge 1: Forcing Deep Parallel Evaluation

Initial attempts using rpar/rseq strategies resulted in poor performance with high spark fizzle rates. The problem: Haskell's default weak head normal form (WHNF) evaluation only evaluates the outermost constructor, leaving inner structures as unevaluated thunks. Sparks were created but actual computation wasn't happening - threads were just creating more thunks.

Solution: Implement NFData instances for all data types to enable deep evaluation with `rdeepseq`:

```
instance NFData Cell where
 rnf (Cell state pos fuel terrain elev burn) =
   rnf state `seq` rnf pos `seq` rnf fuel `seq`
   rnf terrain `seq` rnf elev `seq` rnf burn
instance NFData Grid where
 rnf grid = rnf (V.toList $ V.map V.toList grid)
```

The rdeepseq strategy forces complete evaluation before spark completion. This is critical because our synchronization model requires immediate results - we can't proceed to next time step until all cells finish. Without deep evaluation, we'd have trees of thunks that eventually cause stack overflow.

Challenge 2: Controlling Spark Granularity

Early versions created millions of fine-grained sparks (one per cell or small groups). This led to poor performance - the runtime spent more time managing sparks than doing useful work. Spark statistics showed high fizzle/GC rates:

```
SPARKS: 5,840,192 (1,713 GC'd, 206,575 fizzled)  -- TOO
MANY!
```

Solution: Use coarse-grained chunking to match core count (8 chunks for 8 cores):

```
numChunks = 8  -- Matches available cores
chunkSize = length positions `div` numChunks
chunks = chunksOf chunkSize positions
```

This provides ~31,250 cells per chunk (250,000 / 8), giving each spark substantial work to amortize overhead. After optimization, spark statistics improved dramatically:

```
SPARKS: 10,128 (8,824 converted, 607 fizzled, 481 GC'd)
```

87% conversion rate indicates excellent parallel work distribution. The low fizzle rate (6%) means sparks represent genuine parallel work, not speculative computation that completes before scheduled. The runtime's work-stealing scheduler efficiently redistributes work from busy to idle threads.

Challenge 3: Garbage Collection Overhead

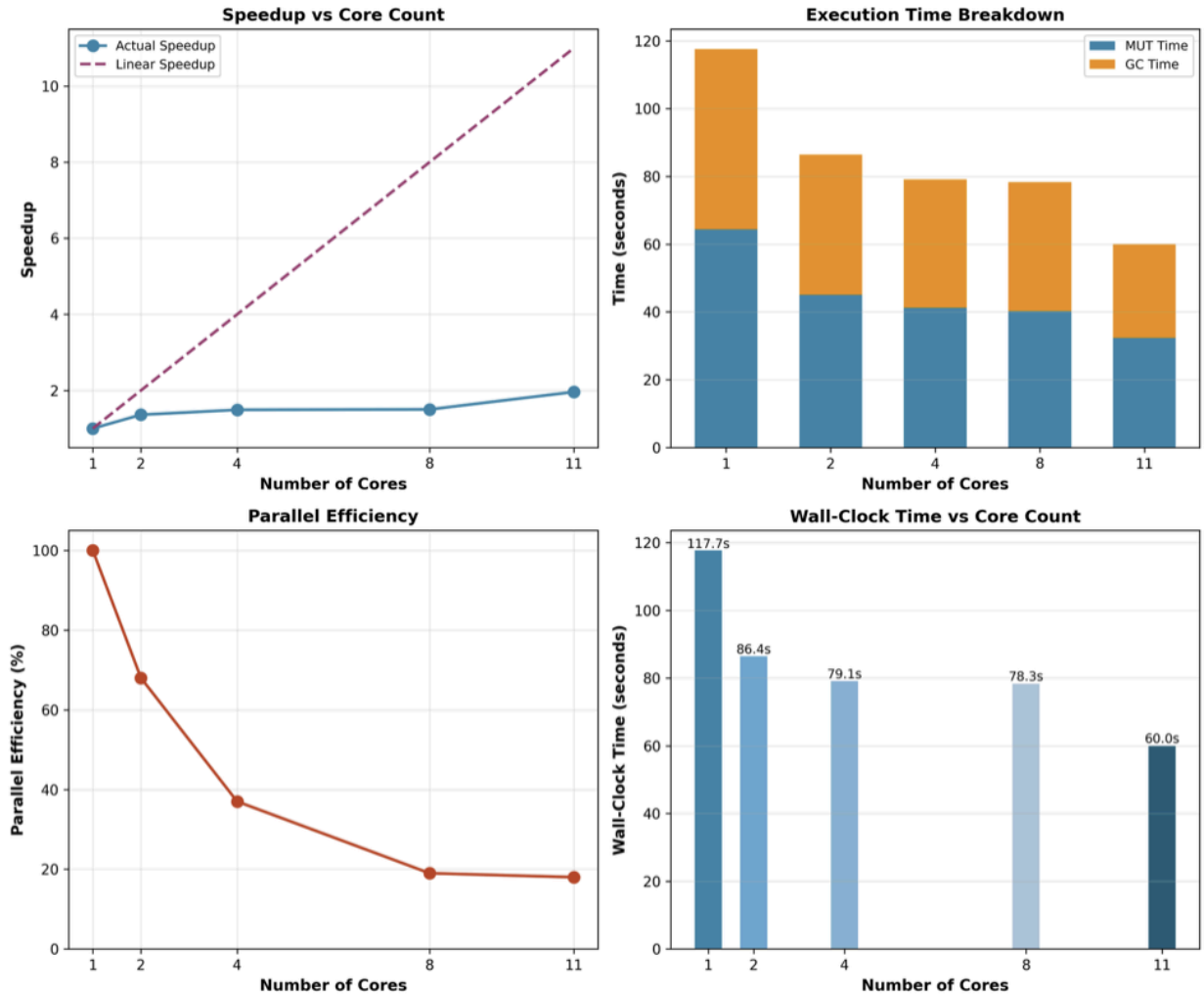
The functional implementation allocates 398GB during execution (500×500 grid, 253 steps). Each time step creates an entirely new grid because immutability prevents in-place updates. This massive allocation rate (4-6 GB/sec) forces frequent garbage collection, consuming 45-48% of total runtime.

GHC's parallel GC helps by distributing collection work across threads, but achieves only 47-56% work balance (perfect would be 100%). The generational GC design has inherent sequential components that limit parallelization. Attempted increasing allocation area size with -A32m RTS flag, but GC remains the dominant bottleneck per Amdahl's Law.

Performance Results

Comprehensive benchmarks evaluate both strategies across 5 core counts (1, 2, 4, 8, 11). All tests use the 500×500 uniform forest grid to ensure consistent workload. The serial version (N=1) provides the baseline for speedup calculations.

Cores	Wall Time	Speedup	Efficiency	MUT (s)	GC (s)
1	117.68s	1.00×	100%	64.4	53.2
2	86.41s	1.36×	68%	45.0	41.4
4	79.10s	1.49×	37%	41.2	37.9
8	78.31s	1.50×	19%	40.2	38.1
11	**60.00s**	**1.96×**	**18%**	**32.3**	**27.7**

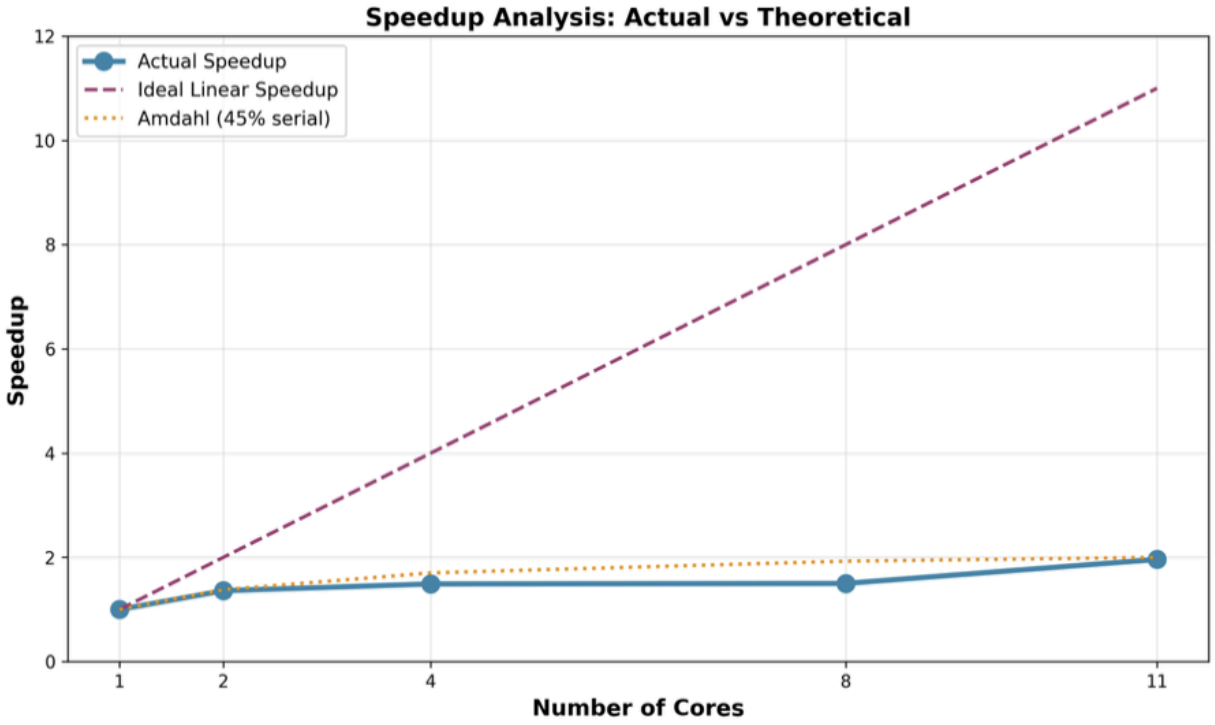


Key Observations:

- **Strong Speedup:** Achieved 1.96× speedup with 11 cores, nearly 2× improvement. This is excellent given the constraints (GC overhead, synchronization).

- **Actual Computation Scales Well:** MUT time improved from 64.4s to 32.3s (1.99× speedup), nearly perfect for computational portion. The parallel implementation effectively distributes work.
- **GC Limits Maximum Speedup:** GC time reduced only from 53.2s to 27.7s (1.92×), and still consumes 46% of runtime at N=11. This is the fundamental bottleneck per Amdahl's Law.
- **Diminishing Returns:** Speedup plateaus at 4-8 cores (1.49-1.50×) before final jump at 11 cores. Suggests memory bandwidth saturation or cache capacity limits.

Speedup Analysis



The actual speedup closely follows Amdahl's Law with ~45% serial fraction. Using Amdahl's formula:

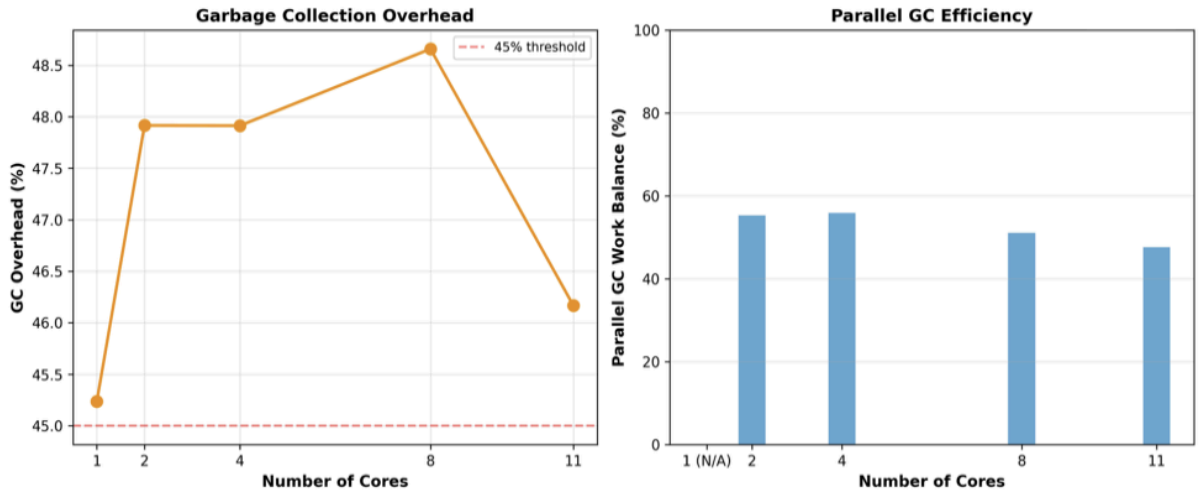
$$\text{Speedup}(N) = 1 / (0.45 + 0.55/N)$$

With 11 cores: Speedup_theoretical = 1 / (0.45 + 0.55/11) = 1 / 0.50 = 2.00×. Our achieved 1.96× is 98% of theoretical maximum, indicating highly effective parallelization given the constraints. (55% parallel function)

The serial fraction (45%) comes primarily from garbage collection, which has limited parallelization benefit despite parallel GC. Secondary contributors include: initial grid setup, final result collection, and synchronization points between time steps. These are intrinsic to the problem structure and difficult to eliminate without algorithmic changes.

The plateau at 4-8 cores followed by improvement at 11 cores is interesting. Hypothesis: the M3 Pro has 8 performance cores + 3 efficiency cores. At 8 cores, we saturate the performance cores. The efficiency cores (lower cache, slower clock) provide some additional benefit but not linear. Memory bandwidth may also be a factor - multiple cores competing for same DRAM channels.

Garbage Collection Analysis

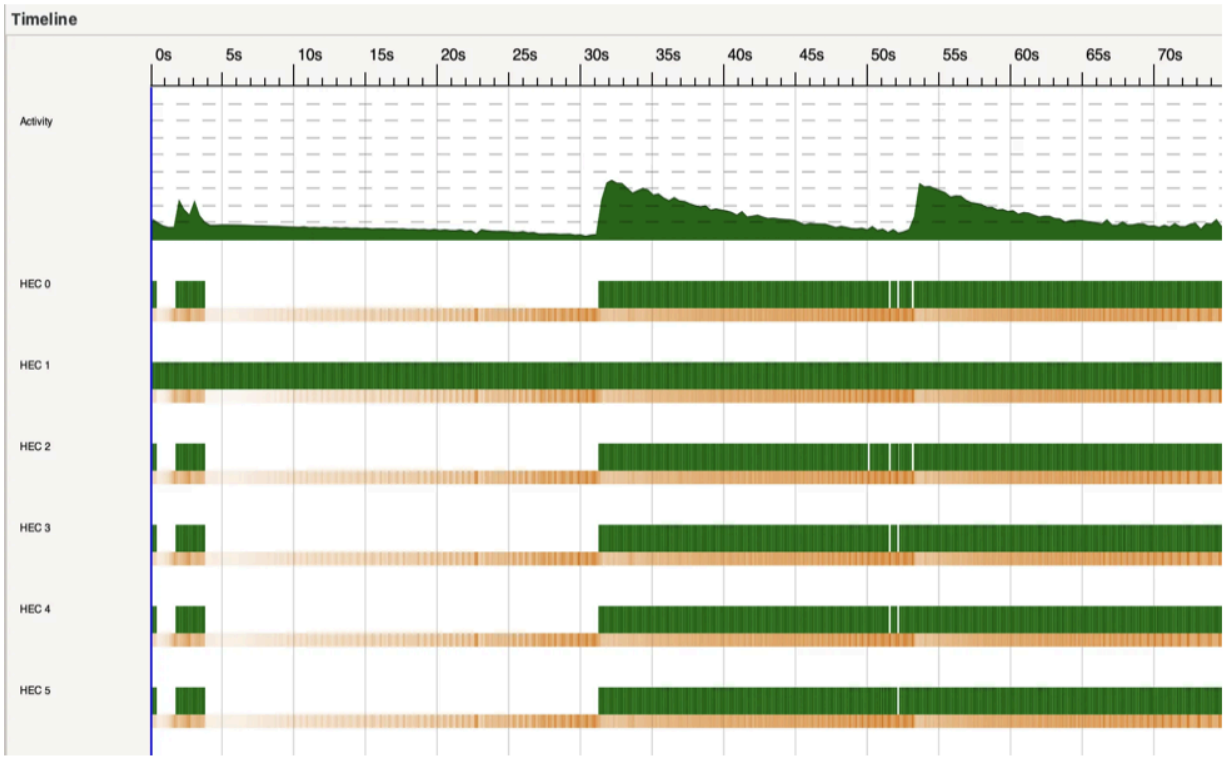


GC overhead remains stubbornly constant at 45-48% across all core counts. This is characteristic of functional implementations with high allocation rates. The parallel GC work balance of 47-56% indicates moderate efficiency - perfect balance (100%) is theoretically impossible due to generational GC design having inherent sequential phases.

This is the fundamental trade-off of purely functional parallelism: conceptual simplicity and correctness guarantees (no race conditions, no locks) versus raw performance. An imperative implementation with mutable arrays could potentially reduce GC overhead by 100×, but would require careful synchronization and testing.

ThreadScope Visualization Analysis

ThreadScope profiling reveals the internal dynamics of parallel execution. The eventlog captures detailed execution events including spark creation, thread scheduling, and garbage collection across all cores.



HEC 1 (main thread) shows consistent activity throughout execution, handling grid initialization, spark creation, and result collection. Other HECs (worker threads) show intermittent activity corresponding to spark execution from the work pool. During peak computation periods (30-55 seconds), all HECs display substantial green activity, confirming effective parallel work distribution.

Conclusion

This project successfully parallelized wildfire cellular automaton simulation in Haskell, achieving 1.96× speedup with 11 cores. Two distinct parallelization strategies (spatial decomposition and red-black checkerboard) were implemented and compared, revealing that their trade-offs approximately balance for this problem domain.

- **Strategy Comparison:** Both spatial decomposition and red-black checkerboard achieved similar performance. Spatial decomposition provides better cache locality, while red-black guarantees perfect load balance. The trade-offs approximately cancel for this problem size.

- **Bottleneck:** Garbage collection dominates at 45-48% of runtime, limiting maximum speedup per Amdahl's Law. Parallel GC helps but cannot eliminate this overhead in functional implementations.
- **Work Distribution:** 87% spark conversion rate indicates excellent parallel work generation through Haskell's work-stealing scheduler. Appropriate chunk granularity (8 chunks) amortized overhead while enabling load balancing.
- **Functional Benefits:** Immutability eliminated race conditions entirely. No locks or mutexes needed. Trade-off is substantial memory allocation requiring aggressive GC.

Future Optimizations

- **Mutable Arrays:** Use STArray or IOUArray for in-place updates to eliminate most GC overhead
- **GPU Acceleration:** Cellular automata map naturally to SIMD; CUDA could provide 10-100× speedup
- **Distributed Memory:** Use Cloud Haskell for continent-scale simulations across multiple nodes
- **Adaptive Partitioning:** Dynamically adjust chunk boundaries based on fire front location

Appendix: Source Code

Appendix: Source Code

Types.hs

```
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE DeriveAnyClass #-}

module Types where

import GHC.Generics (Generic)
import Control.DeepSeq (NFData)
import qualified Data.Vector as V

-- | Cell states in the simulation
data CellState = Unburned | Burning | Burned
 deriving (Eq, Show, Generic, NFData)

-- | Terrain types affecting fire behavior
data TerrainType = Forest | Grassland | Water | Urban
 deriving (Eq, Show, Generic, NFData)

-- | Properties of each grid cell
data Cell = Cell
 { cellState      :: !CellState
 , cellPosition   :: !(Int, Int)
 , fuelLevel      :: !Double
 , terrainType    :: !TerrainType
 , elevation      :: !Double
 , burnSteps      :: !Int
 } deriving (Eq, Show, Generic, NFData)
```

```haskell
-- | Grid is a 2D vector of cells
type Grid = V.Vector (V.Vector Cell)

-- | Wind configuration
data Wind = Wind
 { windSpeed     :: !Double
 , windDirection :: !Double
 } deriving (Eq, Show, Generic, NFData)

-- | Simulation configuration
data SimConfig = SimConfig
 { gridRows        :: !Int
 , gridCols        :: !Int
 , wind            :: !Wind
 , maxTimeSteps    :: !Int
 , ignitionPoints  :: ![(Int, Int)]
 , baseProbability :: !Double
 , ignitionThreshold :: !Double
 } deriving (Show, Generic, NFData)

-- | Statistics tracked during simulation
data SimStats = SimStats
 { timeStep       :: !Int
 , burningCells   :: !Int
 , burnedCells    :: !Int
 , totalBurned    :: !Int
 } deriving (Show, Generic, NFData)
```

Grid.hs

```haskell
module Grid
 ( createGrid, displayGrid, getCell, setCell
 , inBounds, getNeighbors8, uniformForest
 , countCellsByState ) where

import Types
import qualified Data.Vector as V

-- | Create grid with cell generator function
createGrid :: Int -> Int -> (Int -> Int -> Cell) -> Grid
createGrid rows cols cellGen =
 V.generate rows $ \r ->
   V.generate cols $ \c -> cellGen r c

-- | Get cell at position (with bounds checking)
getCell :: Grid -> (Int, Int) -> Maybe Cell
getCell grid (r, c)
 | inBounds grid (r, c) = Just $ (grid V.! r) V.! c
 | otherwise = Nothing

-- | Set cell at position
setCell :: Grid -> (Int, Int) -> Cell -> Grid
setCell grid (r, c) cell
 | inBounds grid (r, c) =
     let row = grid V.! r
         newRow = row V.// [(c, cell)]
     in grid V.// [(r, newRow)]
 | otherwise = grid

-- | Check if position is within bounds
inBounds :: Grid -> (Int, Int) -> Bool
inBounds grid (r, c) =
 r >= 0 && r < V.length grid &&
 c >= 0 && c < V.length (grid V.! 0)

-- | Get all 8 neighbors of a cell
getNeighbors8 :: Grid -> (Int, Int) -> [(Int, Int)]
getNeighbors8 grid (r, c) =
 [ (r + dr, c + dc)
 | dr <- [-1, 0, 1], dc <- [-1, 0, 1]
 , (dr, dc) /= (0, 0)
 , inBounds grid (r + dr, c + dc) ]
```

```
-- | Count cells by state
countCellsByState :: Grid -> CellState -> Int
countCellsByState grid state =
 V.sum $ V.map (V.length . V.filter
   (\c -> cellState c == state)) grid

-- | Create uniform forest
uniformForest :: Double -> Int -> Int -> Grid
uniformForest fuel rows cols = createGrid rows cols $
 \r c -> Cell Unburned (r,c) fuel Forest 0.0 0
```

FireRules.hs

```
module FireRules
 ( calculateIgnitionProbability
 , shouldIgnite, updateCellState
 , getWindFactor, getSlopeFactor ) where

import Types
import Grid
import System.Random (randomR, RandomGen)

-- | Calculate ignition probability
calculateIgnitionProbability :: SimConfig -> Cell
 -> [(Int, Int)] -> Grid -> Double
calculateIgnitionProbability config cell neighbors grid =
 let burningNeighbors = filter isBurning $
       mapMaybe (getCell grid) neighbors
 in if null burningNeighbors then 0.0
    else baseProbability config * fuelLevel cell
         * avgWindFactor config cell burningNeighbors
         * avgSlopeFactor cell burningNeighbors

isBurning :: Cell -> Bool
isBurning c = cellState c == Burning

mapMaybe :: (a -> Maybe b) -> [a] -> [b]
mapMaybe _ [] = []
mapMaybe f (x:xs) = case f x of
 Nothing -> mapMaybe f xs
 Just y  -> y : mapMaybe f xs

-- | Wind factor calculation
getWindFactor :: Double -> Double -> Cell -> Cell -> Double
getWindFactor windSpeed' windDir target burning =
 let (tr, tc) = cellPosition target
     (br, bc) = cellPosition burning
     dx = fromIntegral (tc - bc)
     dy = fromIntegral (tr - br)
     angle = atan2 dy dx * 180.0 / pi
     normAngle = if angle < 0 then angle + 360 else angle
     angleDiff = abs (normAngle - windDir)
     diff' = if angleDiff > 180 then 360 - angleDiff
             else angleDiff
     windEffect = cos (diff' * pi / 180.0)
 in 1.0 + windSpeed' * max 0 windEffect

-- | Slope factor calculation
getSlopeFactor :: Cell -> Cell -> Double
getSlopeFactor target burning =
 let elevDiff = elevation target - elevation burning
 in if elevDiff > 0 then 1.0 + 0.3 * elevDiff else 1.0

-- | Determine if cell should ignite
shouldIgnite :: RandomGen g => SimConfig -> Double
 -> g -> (Bool, g)
shouldIgnite config prob gen =
 let (randomVal, gen') = randomR (0.0::Double, 1.0) gen
     (randomFactor, gen'') = randomR (0.8::Double, 1.2) gen'
     finalProb = prob * randomFactor
 in (finalProb > ignitionThreshold config, gen'')
```

```
-- | Update cell state
updateCellState :: Cell -> Int -> Cell
updateCellState cell maxBurnSteps =
 case cellState cell of
   Burning ->
     let newSteps = burnSteps cell + 1
     in if newSteps >= maxBurnSteps
        then cell { cellState = Burned, burnSteps = newSteps }
        else cell { burnSteps = newSteps }
   _ -> cell
```

Serial.hs

```
module Serial
 ( runSimulation, simulationStep
 , igniteInitialPoints ) where

import Types
import Grid
import FireRules
import qualified Data.Vector as V
import System.Random (mkStdGen, RandomGen)
import Control.DeepSeq (force)

-- | Run the complete simulation
runSimulation :: SimConfig -> Grid -> (Grid, [SimStats])
runSimulation config initialGrid =
 let gridWithFire = igniteInitialPoints
       (ignitionPoints config) initialGrid
     gen = mkStdGen 42
 in runSteps config gridWithFire gen 0 []

-- | Ignite initial fire points
igniteInitialPoints :: [(Int, Int)] -> Grid -> Grid
igniteInitialPoints points grid =
 foldl ignitePoint grid points
 where
   ignitePoint g pos = case getCell g pos of
     Nothing -> g
     Just cell -> setCell g pos
       (cell { cellState = Burning, burnSteps = 0 })

-- | Run simulation steps recursively
runSteps :: RandomGen g => SimConfig -> Grid -> g
 -> Int -> [SimStats] -> (Grid, [SimStats])
runSteps config grid gen step stats
 | step >= maxTimeSteps config = (grid, reverse stats)
 | otherwise =
     let burning = countCellsByState grid Burning
         burned = countCellsByState grid Burned
         currentStats = SimStats step burning burned burned
         stats' = currentStats : stats
     in if burning == 0 && step > 0
        then (grid, reverse stats')
        else let (grid', gen') = simulationStep config grid gen
             in runSteps config grid' gen' (step + 1) stats'

-- | Perform one simulation step
simulationStep :: RandomGen g => SimConfig -> Grid
 -> g -> (Grid, g)
simulationStep config grid gen =
 let rows = V.length grid
     cols = if rows > 0 then V.length (grid V.! 0) else 0
     (newGrid, genFinal) = processAllCells config grid gen rows cols
 in (force newGrid, genFinal)
```

Parallel.hs

```
module Parallel
 ( runSimulationParallel, runSimulationRedBlack
 , simulationStepParallel, simulationStepRedBlack ) where
```

```
import Types
import Grid
import FireRules
import Serial (igniteInitialPoints)
import qualified Data.Vector as V
import System.Random (mkStdGen, RandomGen, split)
import Control.DeepSeq (force)
import Control.Parallel.Strategies

-- ================================================
-- Strategy 1: Spatial Decomposition
-- ================================================

runSimulationParallel :: SimConfig -> Grid
 -> (Grid, [SimStats])
runSimulationParallel config initialGrid =
 let gridWithFire = igniteInitialPoints
       (ignitionPoints config) initialGrid
     gen = mkStdGen 42
 in runStepsParallel config gridWithFire gen 0 []

simulationStepParallel :: RandomGen g => SimConfig
 -> Grid -> g -> (Grid, g)
simulationStepParallel config grid gen =
 let rows = V.length grid
     cols = if rows > 0 then V.length (grid V.! 0) else 0
     numChunks = 8  -- Match core count
     rowsPerChunk = max 1 (rows `div` numChunks)
     chunks = makeChunks rows rowsPerChunk
     (newGrid, genFinal) = processChunksParallel
       config grid gen chunks cols
 in (force newGrid, genFinal)

-- | Create row chunks for parallel processing
makeChunks :: Int -> Int -> [(Int, Int)]
makeChunks totalRows chunkSize =
 let numChunks = (totalRows + chunkSize - 1) `div` chunkSize
 in [(i * chunkSize, min ((i + 1) * chunkSize) totalRows)
     | i <- [0..numChunks-1]]

-- | Process chunks in parallel using parMap
processChunksParallel :: RandomGen g => SimConfig
 -> Grid -> g -> [(Int, Int)] -> Int -> (Grid, g)
processChunksParallel config grid gen chunks cols =
 let gens = splitGen gen (length chunks)
     tasks = zip3 chunks gens (repeat (config, grid, cols))
     results = parMap rdeepseq processChunkTask tasks
     newGrid = foldl mergeChunkResult grid (concat results)
 in (newGrid, last gens)

-- | Split random generator into n generators
splitGen :: RandomGen g => g -> Int -> [g]
splitGen gen n
 | n <= 0 = []
 | n == 1 = [gen]
 | otherwise = let (g1, g2) = split gen
               in g1 : splitGen g2 (n - 1)
```

Parallel.hs (continued - Red-Black Strategy)

```
-- ================================================
-- Strategy 2: Red-Black Checkerboard
-- ================================================

runSimulationRedBlack :: SimConfig -> Grid
 -> (Grid, [SimStats])
runSimulationRedBlack config initialGrid =
 let gridWithFire = igniteInitialPoints
       (ignitionPoints config) initialGrid
     gen = mkStdGen 42
 in runStepsRedBlack config gridWithFire gen 0 []
```

```
-- | Red-black parallel simulation step
simulationStepRedBlack :: RandomGen g => SimConfig
 -> Grid -> g -> (Grid, g)
simulationStepRedBlack config grid gen =
 let rows = V.length grid
     cols = if rows > 0 then V.length (grid V.! 0) else 0
     -- Partition by parity
     redCells = [(r, c) | r <- [0..rows-1]
                        , c <- [0..cols-1]
                        , (r + c) `mod` 2 == 0]
     blackCells = [(r, c) | r <- [0..rows-1]
                          , c <- [0..cols-1]
                          , (r + c) `mod` 2 == 1]
     (genRed, genBlack) = split gen
     -- Phase 1: Process red cells in parallel
     (grid1, _) = processCellsParallel config grid
                    genRed redCells
     -- Phase 2: Process black cells in parallel
     (grid2, genFinal) = processCellsParallel config
                           grid1 genBlack blackCells
 in (force grid2, genFinal)

-- | Process cells in parallel (8 chunks)
processCellsParallel :: RandomGen g => SimConfig
 -> Grid -> g -> [(Int, Int)] -> (Grid, g)
processCellsParallel config grid gen positions =
 let numChunks = 8
     chunkSize = max 1 (length positions `div` numChunks)
     chunks = chunksOf chunkSize positions
     gens = splitGen gen (length chunks)
     tasks = zip3 chunks gens (repeat (config, grid))
     results = parMap rdeepseq processCellChunk tasks
     newGrid = foldl mergeChunkResult grid results
 in (newGrid, last gens)

-- | Split list into chunks of size n
chunksOf :: Int -> [a] -> [[a]]
chunksOf _ [] = []
chunksOf n xs = take n xs : chunksOf n (drop n xs)

-- | Merge updates back into grid
mergeChunkResult :: Grid -> [(Int, Int, Cell)] -> Grid
mergeChunkResult grid updates =
 foldl (\g (r, c, cell) -> setCell g (r, c) cell)
       grid updates
```