ParLife: A Parallel Implementation of Conway's Game of Life

Adam Fowler
ajf2177

ParLife is a parallel implementation of Conway's Game of Life, a well-known cellular automaton modeling discrete computation. Conway's Game of Life is typically represented as a gameboard consisting of cells identified by their coordinate pair. Each cell can be either 'alive' or 'dead' and the subsequent state of the game board (i.e., a generation) is computed from the current state starting with the initial seed state.

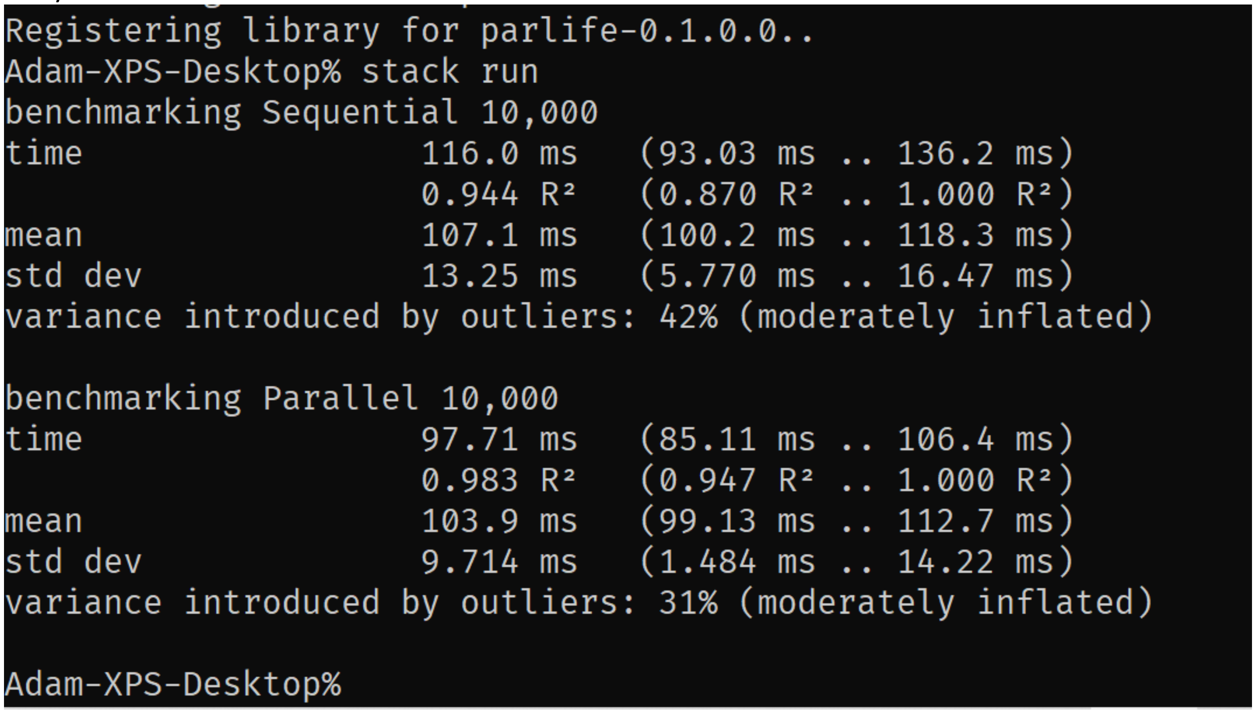
ParLife is implemented using a sparse representation of the board state, where each living cell is represented by a coordinate pair of type (Int, Int) contained within a list. The next generation of the board state is given by first expanding each cell in the population list to a list including each of its neighbor cells. The resulting list of lists of coordinate pairs is then concatenated to flatten the nested structure into a single list. This list of coordinate pairs is then sorted and grouped to organize each repetition of a cell together. This sorted and grouped list is then mapped over to produce a list of cells and how many times that cell appears in the generated neighborhood list. This list is then filtered by applying the rules for Conway's Game of Life, namely if a cell is already a member of the population set and also appears in 3 neighborhoods (including its own) the cell continues to live or if the cell appears in 4 neighborhoods that cell is now alive irrespective of its prior state. This lookup to determine whether a cell is alive is O(log n) as the population list is transformed into a Set at the start of each function call which is then used to determine population membership.

The difference between the parallel implementation and the sequential implementation is the usage of parMap rseq to parallelize both the generation of the list of neighborhoods as well and the final parsing of the list of cells and their neighborhood occurrence counts. These two pieces of the algorithm lend themselves well to parallelization as there are no interdependencies between the cells which would interfere with the resulting lists.

The testing strategy used was to take a few known seed states which either repeat, stabilize, or die off and test that they reach the intended state after an expected number of intervals. Future considerations include more test cases in addition to adding some method to randomize the start state and also a function to write the end state of the board out as a simple bitmap image.

Performance

Speedup of 1.19x seen from parallelization with -N and 10,000 iterations
This is most likely not an accurate measurement of speedup due to the small amount of time, most likely this is a fluke but I've included it as a base line.

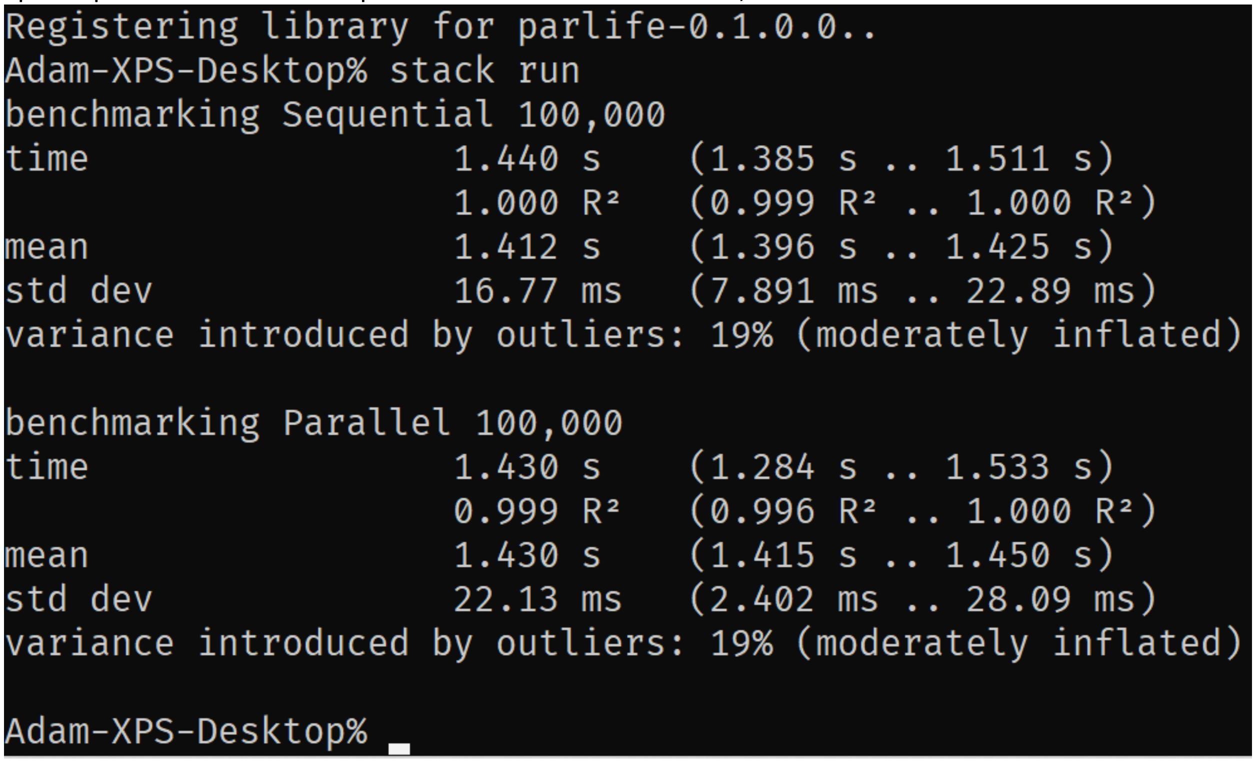


```
Registering library for parlife-0.1.0.0..
Adam-XPS-Desktop% stack run
benchmarking Sequential 10,000
time                 116.0 ms   (93.03 ms .. 136.2 ms)
                     0.944 R²   (0.870 R² .. 1.000 R²)
mean                 107.1 ms   (100.2 ms .. 118.3 ms)
std dev              13.25 ms   (5.770 ms .. 16.47 ms)
variance introduced by outliers: 42% (moderately inflated)

benchmarking Parallel 10,000
time                 97.71 ms   (85.11 ms .. 106.4 ms)
                     0.983 R²   (0.947 R² .. 1.000 R²)
mean                 103.9 ms   (99.13 ms .. 112.7 ms)
std dev              9.714 ms   (1.484 ms .. 14.22 ms)
variance introduced by outliers: 31% (moderately inflated)

Adam-XPS-Desktop%
```

Speedup of 1.007x seen from parallelization with -N4 and 100,000 iterations



```
Registering library for parlife-0.1.0.0..
Adam-XPS-Desktop% stack run
benchmarking Sequential 100,000
time                 1.440 s    (1.385 s .. 1.511 s)
                     1.000 R²   (0.999 R² .. 1.000 R²)
mean                 1.412 s    (1.396 s .. 1.425 s)
std dev              16.77 ms   (7.891 ms .. 22.89 ms)
variance introduced by outliers: 19% (moderately inflated)

benchmarking Parallel 100,000
time                 1.430 s    (1.284 s .. 1.533 s)
                     0.999 R²   (0.996 R² .. 1.000 R²)
mean                 1.430 s    (1.415 s .. 1.450 s)
std dev              22.13 ms   (2.402 ms .. 28.09 ms)
variance introduced by outliers: 19% (moderately inflated)

Adam-XPS-Desktop% _
```

Code Listings

Lib.hs

```
module Lib (parLife, life, driver, testDriver, testLifeFunc,
testNeighbors, originSquare, neighbors, Cell) where

import Control.Arrow
import Control.Parallel.Strategies
import Data.List
import qualified Data.Set as Set

-- datatypes
type Cell = (Int, Int)
type Board = [Cell]

testDriver::([Cell] -> [Cell]) -> Int -> [Cell] -> [Cell]
testDriver lf num startState = last $ take num $ iterate lf startState
-- driver
driver::([Cell] -> [Cell]) -> Int -> [Cell]
driver lf num = last $ take num $ iterate lf initState
  where
      initState = [(1, 2), (2, 2), (2, 1), (2, 3), (3, 3)]
    -- initState = [(2, 0), (7, 0), (0, 1), (1, 1), (3, 1), (4, 1), (5,
1), (6, 1), (8, 1), (9, 1), (2, 2), (7, 2)]

-- non parallel implementation for benchmarking
life :: [Cell] -> [Cell]
life population = map fst . filter rules . tally $ concatMap neighbors
population
  where
    popSet = Set.fromList population
    rules (cl, n) = (cl `Set.member` popSet && n == 3) || n == 4
    tally = map (\x -> (head x, length x)) . (group . sort)

-- See https://lbarasti.com/post/game_of_life/
-- count occurrences of each (Int, Int) live cell in populations
-- cells appearing in 3 'neighborhoods' and also present in population
will continue to live
-- cells appearing in 4 'neighborhoods' will become 'alive'
-- use Set to reduce lookup time per Professor Edwards
parLife :: [Cell] -> [Cell]
parLife population = parMap rseq fst . filter rules . tally . concat $
parMap rseq neighbors population
```

```haskell
  where
    popSet = Set.fromList population
    rules (cl, n) = (cl `Set.member` popSet && n == 3) || n == 4
    tally = map (\x -> (head x, length x)) . (group . sort)

originSquare::[Cell]
originSquare = [
  (-1, 1), (0, 1), (1, 1),
  (-1, 0), (0, 0), (1, 0),
  (-1,-1), (0,-1), (1,-1)
  ]

-- Get all neighbor coords of given cell
--   [(-1, 1), (0, 1), (1, 1),
--    (-1, 0), (0, 0), (1, 0),
--    (-1,-1), (0,-1), (1,-1)
--]
neighbors :: Cell -> [Cell]
neighbors (x, y) = [(x + dx, y + dy) | (dx, dy) <- originSquare]


testNeighbors::Cell -> [Cell] -> Bool
testNeighbors cl expected = (sort (neighbors cl)) == (sort expected)

testLifeFunc::[Cell] -> ([Cell] -> [Cell]) -> Int -> [Cell] -> Bool
testLifeFunc startState lifeFunc num expectedState = (testDriver
lifeFunc num startState) == expectedState
```

App.hs

```haskell
module Main where

import Lib
import System.Environment
import Criterion
import Criterion.Main


main = defaultMain
    [ bench "Sequential 100,000" $ whnf (driver life) 1000000
    , bench "Parallel 100,000"    $ whnf (driver parLife) 10000000]
```

Spec.hs

```
import Test.HUnit
import Test.Framework
import Test.Framework.Providers.HUnit
import Lib

dyingTromino::[Cell]
dyingTromino = [(0,0), (0,1), (1,2)]

dyingBoardTest::Assertion
dyingBoardTest = assertBool "Dying tromino state should die after 3
moves" (testLifeFunc dyingTromino parLife 3 [])

deadBoardTest::Assertion
deadBoardTest = assertBool "Dead state should remain dead" (testLifeFunc
[] parLife 1 [])

stableBoardTest::Assertion
stableBoardTest = assertBool "Stable state should remain same"
(testLifeFunc [(1,2), (2,1), (2,3), (3,2), (3,3)] parLife 1 [(1,2),
(2,1), (2,3), (3,2), (3,3)])

main :: IO ()
main = defaultMainWithOpts
       [testCase "deadBoardTest" deadBoardTest,
       testCase "dyingBoardTest" dyingBoardTest,
       testCase "stableBoardTest" stableBoardTest
       ]
       mempty
```