ParBoids

Catelen Wu
cw3223@columbia.edu

Ethan Wu
ew2664@columbia.edu

1 Introduction

`Boids` ("bird-oids") is an artificial life program simulating the flocking behavior of birds developed by Craig Reynolds in 1986. It is an example of emergent behavior and swarm intelligence: each boid agent follows only a simple set of rules, but the interactions between them give rise to complex and unpredictable behavior mimicking that of flocks or herds of animals found in nature. In this project, we first develop a sequential `Boids` simulation in Haskell, then attempted to parallelize the program using different strategies.

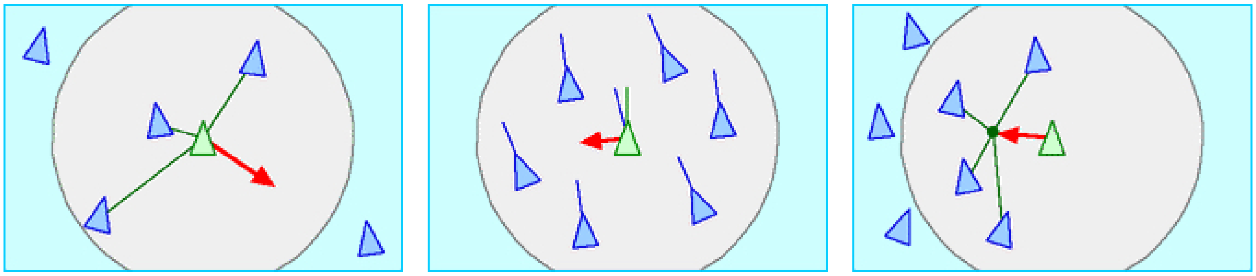

Figure 1: Steering forces of separation, alignment, and cohesion (left to right)

In `Boids`, each boid has its individual position, velocity, and mass. Each boid is also subject to three steering forces: **separation**, **alignment**, and **cohesion**. (Other rules can also be added to simulate more complex behavior, such as follow-the-leader or obstacle avoidance.) These forces are computed based on the relative positions and velocities of each boid to its local flockmates. Other boids beyond a certain radius are ignored. Then, every time step, the position and velocity of each boid is updated by a weighted sum of the three steering forces. These updates are repeated indefinitely until the program is terminated. The main update loop, run for n time steps on flock $\mathcal{B}$, is shown in Algorithm 1 below.

Algorithm 1 Reynolds's Flocking Algorithm

```
for $i \leftarrow 1, n$ do
    for each $b \in \mathcal{B}$ do
        nbs ← neighbors($b, \mathcal{B}$)                                  ▷ Get local flockmates
        $f_s \leftarrow$ separation($b$, nbs)
        $f_a \leftarrow$ alignment($b$, nbs)
        $f_c \leftarrow$ cohesion($b$, nbs)
        $b_v \leftarrow b_v + (k_s f_s + k_a f_a + k_c f_c)\Delta t / b_m$    ▷ Update boid velocity
        $b_x \leftarrow b_x + b_v \Delta t$                                ▷ Update boid position
    end for
end for
```

2 Sequential Implementation

A sequential implementation for Reynolds's flocking simulation is relatively simple. First, we define a `Boid` data type using Haskell's record syntax, as well as some utility functions for computing the displacement between two boids and finding a boid's local flockmates. We use `Linear.V2` from the `linear` package to represent each boid's position and velocity vectors in two dimensions.

```haskell
data Boid = Boid { bPos :: V2 Float, bVel :: V2 Float, bMass :: Float } deriving (Show)

between :: Config -> Boid -> Boid -> V2 Float
between cfg b bo = case wSize cfg of
  Size size -> wrapDisp size (bPos b) (bPos bo)
  Infinite -> bPos bo ^-^ bPos b

flockmates :: Config -> [Boid] -> Float -> Boid -> [(Boid, V2 Float)]
flockmates cfg flock r b = filter (\(bo, _) -> bPos bo /= bPos b) neighbors
  where
    neighbors = takeWhile (\(_, disp) -> norm disp < r) sorted
    sorted = sortOn (norm . snd) $ map (\bo -> (bo, between cfg b bo)) flock
```

We also define separate functions for each of the three steering forces that computes the effect on a boid from a neighbor. Each force is similarly represented with a two-dimensional `V2` vector.

```haskell
separation :: Boid -> (Boid, V2 Float) -> V2 Float
separation _ (_, disp) = negated disp ^/ (norm disp ** 2)

alignment :: Boid -> (Boid, V2 Float) -> V2 Float
alignment b (bo, _) = bVel bo ^-^ bVel b

cohesion :: Boid -> (Boid, V2 Float) -> V2 Float
cohesion _ (_, disp) = disp
```

Next, we define a `Steer` data type to collect the steering forces acting on a boid.

```haskell
data Steer = Steer { sSf :: V2 Float, sAf :: V2 Float, sCf :: V2 Float } deriving (Show)

initSteer :: Steer
initSteer = Steer zero zero zero
```

Then, for each boid, we have a function `updateBoid` that iterates through its local flockmates, accumulates the steering forces in a `Steer` record, and update's the boid's position and velocity with the net force.

```haskell
steerFrom :: Boid -> Steer -> (Boid, V2 Float) -> Steer
steerFrom b (Steer sf af cf) disp = Steer sf' af' cf'
  where
    sf' = sf ^+^ separation b disp
    af' = af ^+^ alignment b disp
    cf' = cf ^+^ cohesion b disp

updateBoid :: Config -> [Boid] -> Boid -> Boid
updateBoid cfg flock b = b {bPos = pos', bVel = vel'}
  where
    pos' = wrapPos cfg $ bPos b ^+^ 0.1 *^ vel'
    vel' = vBound (maxVel cfg) (bVel b ^+^ 0.05 *^ netf ^/ bMass b)
    netf = sn cfg *^ sf ^+^ an cfg *^ af ^+^ cn cfg *^ cf
    Steer sf af cf = foldl (steerFrom b) initSteer bs
```

```
    bs = flockmates cfg flock (radius cfg) b
```

Finally, we have our main simulation loop `runSimCollect` that runs recursively for `nIter` iterations, updating each boid in the flock using `updateBoid` in each iteration. The state of the flock after each iteration is collected in order to be written to file output or rendered in animation. In the alternative function `runSim`, only the last flock state is kept and all its predecessors are discarded.

```
runSim :: Config -> [Boid] -> Int -> [Boid]
runSim config flock0 nIter = case runSimCollect config flock0 nIter of
  [] -> flock0
  (flockN : _) -> flockN

runSimCollect :: Config -> [Boid] -> Int -> [[Boid]]
runSimCollect cfg flock0 nIter = foldl simLoop [flock0] [1 .. nIter]
  where
    simLoop :: [[Boid]] -> Int -> [[Boid]]
    simLoop [] _ = []
    simLoop flocks@(flock : _) _ = map (updateBoid cfg flock) flock : flocks
```

We also wrote a supplementary animation function using the `gloss` library to visualize the results of the simulation. Figure 2 below shows a frame from our animation of a flock of 50 boids.

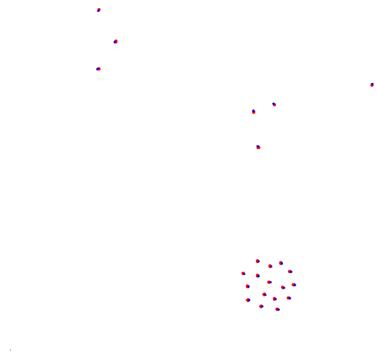

Figure 2: Example frame from an animation of a flock of 50 boids

3 Parallel Implementation

Reynolds's flocking algorithm actually lends itself naturally to parallelization because the update at each time step is only dependent upon the state of the flock at the previous time step. That is, given the previous state of the flock, each boid's update is independent. Therefore, our main approach to parallelizing our sequential implementation is to perform boid updates in parallel.

First, we refactored our `runSimCollect` function to allow us to easily plug in different update methods implemented with different strategies. We define a new data type `ParStrat` to indicate the strategy being used. The sequential `map` over boids from above is moved into the `updateSeq` function.

```
data ParStrat = Seq | TwoPart | Chunks Int | ParList

updateWith :: ParStrat -> Config -> [Boid] -> [Boid]
updateWith Seq = updateSeq
```

```
updateWith TwoPart = updateTwoPart
updateWith (Chunks n) = updateChunks n
updateWith ParList = updateParList

runSimCollect :: Config -> [Boid] -> Int -> [[Boid]]
runSimCollect cfg flock0 nIter = foldl simLoop [flock0] [1 .. nIter]
  where
    simLoop :: [[Boid]] -> Int -> [[Boid]]
    simLoop [] _ = []
    simLoop flocks@(flock : _) _ = updateWith Seq cfg flock : flocks

updateSeq :: Config -> [Boid] -> [Boid]
updateSeq cfg flock = map (updateBoid cfg flock) flock
```

The first strategy we attempted, `TwoPart`, is static two-way partitioning, where we split the flock into two sub-flocks and update each flock in parallel using `rpar`. Even though the work needed for each boid may differ depending on its number of neighbors, we don't expect this difference to be too great given the small radius of each boid's neighborhood. Morever, with a sufficient number of boids, the work needed to update the two sub-flocks will even out, so we believed this strategy to be a reasonable initial approach.

```
updateTwoPart :: Config -> [Boid] -> [Boid]
updateTwoPart cfg flock = runEval $ do
  as' <- rpar (force (map (updateBoid cfg flock) as))
  bs' <- rpar (force (map (updateBoid cfg flock) bs))
  _ <- rseq as'
  _ <- rseq bs'
  return (as' ++ bs')
  where
    (as, bs) = splitAt (length flock `div` 2) flock
```

A more sophisticated version of the above approach that we attempted next is `Chunks`. This approach uses `parListChunk` from `Control.Parallel.Strategies` to split the flock into a specified number of sub-flocks, spark an update for each sub-flock, and recombine the result.

```
updateChunks :: Int -> Config -> [Boid] -> [Boid]
updateChunks numChunks cfg flock = flock'
  where
    flock' = map (updateBoid cfg flock) flock `using` parListChunk numChunks rdeepseq
```

The final approach we attempted is `ParList`, using `parList` from `Control.Parallel.Strategies`. This sparks an update for each individual boid, equivalent to `Chunks` with a chunk size of 1.

```
updateParList :: Config -> [Boid] -> [Boid]
updateParList cfg flock = flock'
  where
    flock' = map (updateBoid cfg flock) flock `using` parList rdeepseq
```

Note that in the above approaches, for `force` and `rdeepseq` from `Control.DeepSeq` to be able to fully evaluate each `Boid` to normal form, we define a rather trivial `NFData` instance for `Boid`.

```
instance NFData Boid where
```

```
  rnf (Boid pos vel m) = rnf pos `seq` rnf vel `seq` rnf m
```

We discuss the results of our experimentation on these various approaches in the next section.

4 Results and Discussion

After implementing our various strategies as described above, we experimented with different parameters in order to gauge the effectiveness of each strategy. First, we experimented with varying the size of our flock, ranging from 50 boids to 5000 boids. Specifically, we measured the total time it took for the baseline sequential algorithm to simulate 100 iterations for these various flock sizes. We then compared this to the performance of static two-way partitioning running on two cores. The execution times are plotted in Figure 3 below, with example Threadscope profiles of the two strategies shown in Figure 4 and Figure 5.

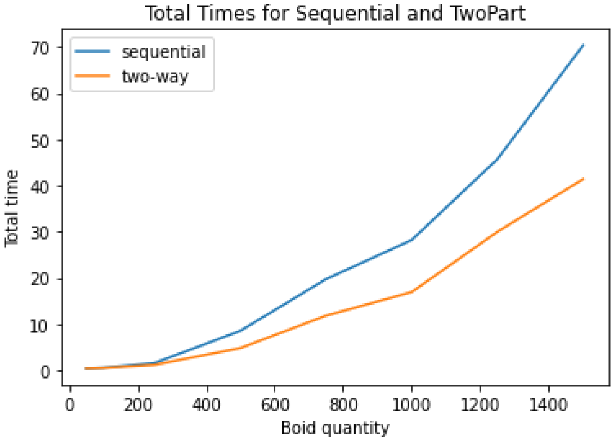

Figure 3: Results from `Seq` and `TwoPart` on different numbers of boids

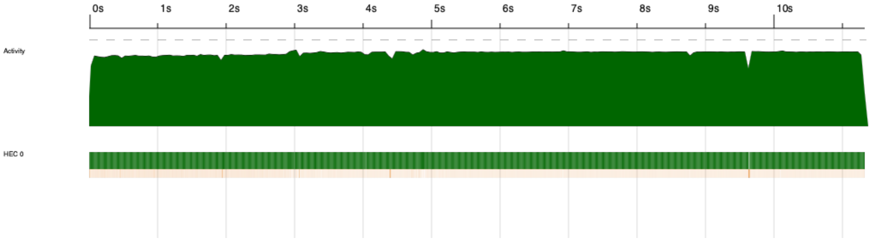

Figure 4: `Seq` Threadscope profile for 500 boids

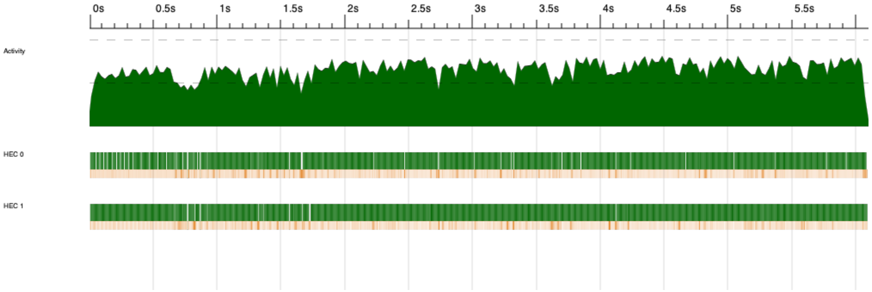

Figure 5: `TwoPart` Threadscope profile for 500 boids

First, we observe that regardless of parallelization, there appears to be a roughly $O(N^2)$ increase in execution time as the flock size increases. This is along the lines of what we expect, especially for large flock sizes, because the bottleneck in the algorithm becomes finding each boid's closest flock-mates. (Our implementation just uses a linear filter, though there may be more efficient solutions such as storing the boids in a quadtree structure.) We also see that two-way partitioning parallelizes work decently with both cores being utilized evenly. As we hoped, we did not encounter the problem of unbalanced partitions because work for each partition tends to even out as flock size increases. However, we do see that a big portion of time in both cores is taken up by garbage collection, causing activity to be spiky and significantly below the full potential.

Next, we moved on to testing our two other approaches `Chunks` and `ParList`, which use the `parListChunk` and `parList` evaluation strategies respectively. These two strategies can take advantage of more cores to hopefully provide further speedups. For each trial, we ran our parallel algorithm on 500 boids for 100 iterations. We experimented with both strategies using different numbers of cores, from 1 and up to 8. For the former, we also varied the number of chunks we used. The execution times are plotted in Figure 6 below, with example Threadscope profiles of the two strategies shown in Figure 7 and Figure 8.

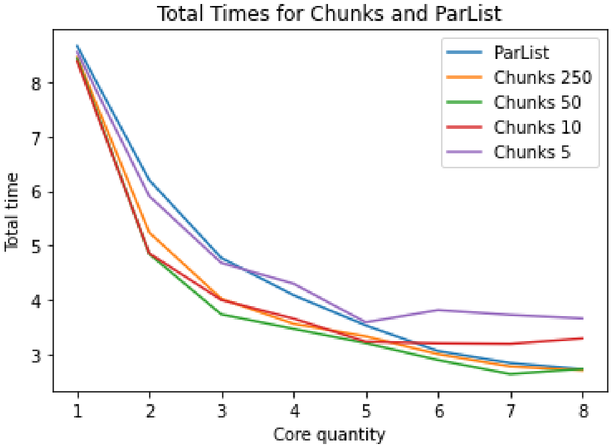

Figure 6: Results from `Chunks` and `ParList` using different numbers of cores and chunks

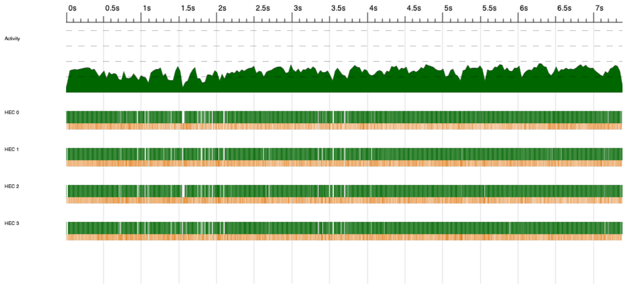

Figure 7: `Chunks 50` Threadscope profile for 500 boids

We can make a few interesting observations from Figure 6. First, all strategies generally decrease execution time as the number of cores increase. (An exception is `Chunks 5`, which as can be expected, stop receiving gains after more than 5 cores were used.) For the other strategies, there were also diminishing returns, often with a number of cores beyond which adding more cores is no longer produces a speed-up. For example, for `Chunks 50`, which split the flock into 10 chunks, using 7 cores was optimal; at 8 cores, the total execution time increased again. These diminishing returns are partly due to Amdahl's law, as our algorithm deals with a signficant amount of IO (e.g., reading and saving the state of the flock to file) that is inherently sequential. Furthermore, there is increased overhead and garbage collection: we found that as the number of cores increases, the amount of garbage collection increases noticeably, as seen in the Threadscope profiles.

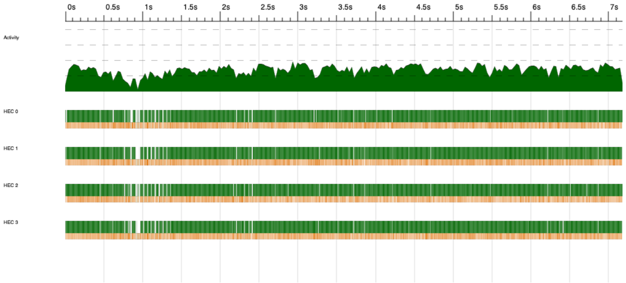

Figure 8: `ParList` Threadscope profile for 500 boids

In a similar regard, for `Chunks`, increasing the number of chunks initially produced better performance, up to a point where more chunks results in too many sparks, more overhead, and poorer performance. Where this point is may depend on the number of cores. For example, we see that `ParList` (equivalent to `Chunks 500` for our case of 500 boids) was initially one of the worst performers, but improved relative to the other strategies when we used higher numbers of cores that could more efficiently process the sparks it generated. Overall, we found the optimal number of chunks to be somewhere around 50, which produced consistently lower execution times for all numbers of cores; the corresponding optimal number of cores is around 7.

5 Code

5.1 `app/Main.hs`

```haskell
module Main (main) where

import Animate (runAnimation)
import BoidIO (loadFlock, saveFlock)
import Config (loadConfig)
import Control.Monad (foldM_, unless)
import GHC.Base (when)
import Options.Applicative
import Sim (runSimCollect)
import System.Exit (die)

data Args = Arguments
  { flockFile :: String,
    numIter :: Int,
    outputDir :: Maybe String,
    configFile :: Maybe String,
    animate :: Bool
  }

arguments :: Parser Args
arguments =
  Arguments
    <$> argument str (metavar "FILE" <> help "Initial flock data file")
    <*> option auto (long "num-iter" <> short 'n' <> metavar "INT" <> help "Number of iterations")
    <*> optional (strOption (long "out-dir" <> short 'o' <> metavar "DIR" <> help "Output directory"))
    <*> optional (strOption (long "config" <> short 'c' <> metavar "CONFIG" <> help "Configuration file"))
    <*> switch (long "animate" <> short 'a' <> help "Whether to run animation")

main :: IO ()
main = run =<< execParser opts
```

```
  where
    opts =
      info
        (arguments <**> helper)
        ( fullDesc
            <> progDesc ""
            <> header ""
        )

run :: Args -> IO ()
run args = do
  unless (numIter args > 0) $ die "num-iter must be a positive integer"
  flock0 <- loadFlock $ flockFile args
  config <- loadConfig $ configFile args
  print config
  let flocks = reverse $ runSimCollect config flock0 $ numIter args
  foldM_ (saveFlock $ outputDir args) 0 flocks
  putStrLn "simulation complete"
  when (animate args) $ do
    putStrLn "running animation"
    runAnimation flocks
  putStrLn "process complete"
```

5.2 src/Animate.hs

```
module Animate (runAnimation) where

import Boid (Boid, bPos, bVel)
import Graphics.Gloss
import Graphics.Gloss.Data.ViewPort (ViewPort)
import Linear.Vector ((*^), (^+^))
import Utils (vScaleTo, vxy)

background :: Color
background = white

window :: Display
window = InWindow "ParBoids" (800, 600) (200, 200)

update :: ViewPort -> Float -> [[Boid]] -> [[Boid]]
update _ _ [] = []
update _ _ (_ : flocks) = flocks

render :: [[Boid]] -> Picture
render [] = blank
render (flock : _) = pictures $ map draw flock

draw :: Boid -> Picture
draw boid =
  pictures
    [ translate x y $ color red $ circleSolid 3,
      translate x' y' $ color blue $ circleSolid 2
    ]
  where
    (x', y') = vxy $ scaleFac *^ bPos boid ^+^ vScaleTo 2 (bVel boid)
    (x, y) = vxy $ scaleFac *^ bPos boid
```

```haskell
    scaleFac = 20

runAnimation :: [[Boid]] -> IO ()
runAnimation flocks = simulate window background 60 flocks render update
```

5.3 src/Boid.hs

```haskell
module Boid (Boid, newBoid, updateBoid, bPos, bVel) where

import Config (Config (..), WorldSize (..))
import Control.DeepSeq (NFData (..))
import Data.List (sortOn)
import Linear (negated)
import Linear.Metric (Metric (norm))
import Linear.V2 (V2 (..))
import Linear.Vector (zero, (*^), (^+^), (^-^), (^/))
import Utils (vBound, vWrap, wrapDisp)

data Boid = Boid { bPos :: V2 Float, bVel :: V2 Float, bMass :: Float } deriving (Show)

instance NFData Boid where
  rnf (Boid pos vel m) = rnf pos `seq` rnf vel `seq` rnf m

newBoid :: [Float] -> Maybe Boid
newBoid [px, py, vx, vy, m] = Just $ Boid (V2 px py) (V2 vx vy) m
newBoid [px, py, vx, vy] = Just $ Boid (V2 px py) (V2 vx vy) 1
newBoid _ = Nothing

between :: Config -> Boid -> Boid -> V2 Float
between cfg b bo = case wSize cfg of
  Size size -> wrapDisp size (bPos b) (bPos bo)
  Infinite -> bPos bo ^-^ bPos b

flockmates :: Config -> [Boid] -> Float -> Boid -> [(Boid, V2 Float)]
flockmates cfg flock r b = filter (\(bo, _) -> bPos bo /= bPos b) neighbors
  where
    neighbors = takeWhile (\(_, disp) -> norm disp < r) sorted
    sorted = sortOn (norm . snd) $ map (\bo -> (bo, between cfg b bo)) flock

wrapPos :: Config -> V2 Float -> V2 Float
wrapPos cfg pos = case wSize cfg of
  Size size -> vWrap size pos
  Infinite -> pos

data Steer = Steer { sSf :: V2 Float, sAf :: V2 Float, sCf :: V2 Float } deriving (Show)

initSteer :: Steer
initSteer = Steer zero zero zero

steerFrom :: Boid -> Steer -> (Boid, V2 Float) -> Steer
steerFrom b (Steer sf af cf) disp = Steer sf' af' cf'
  where
    sf' = sf ^+^ separation b disp
    af' = af ^+^ alignment b disp
    cf' = cf ^+^ cohesion b disp

```

```
updateBoid :: Config -> [Boid] -> Boid -> Boid
updateBoid cfg flock b = b {bPos = pos', bVel = vel'}
  where
    pos' = wrapPos cfg $ bPos b ^+^ 0.1 *^ vel'
    vel' = vBound (maxVel cfg) (bVel b ^+^ 0.05 *^ netf ^/ bMass b)
    netf = sn cfg *^ sf ^+^ an cfg *^ af ^+^ cn cfg *^ cf
    Steer sf af cf = foldl (steerFrom b) initSteer bs
    bs = flockmates cfg flock (radius cfg) b

separation :: Boid -> (Boid, V2 Float) -> V2 Float
separation _ (_, disp) = negated disp ^/ (norm disp ** 2)

alignment :: Boid -> (Boid, V2 Float) -> V2 Float
alignment b (bo, _) = bVel bo ^-^ bVel b

cohesion :: Boid -> (Boid, V2 Float) -> V2 Float
cohesion _ (_, disp) = disp
```

5.4 src/BoidIO.hs

```
module BoidIO (loadFlock, saveFlock) where

import Boid (Boid, newBoid)
import System.IO (Handle, IOMode (ReadMode, WriteMode), hClose, hGetLine, hIsEOF, hPrint, withFile)

loadFlock :: String -> IO [Boid]
loadFlock file = withFile file ReadMode readFlockFile

readFlockFile :: Handle -> IO [Boid]
readFlockFile hdl = do
  isEOF <- hIsEOF hdl
  ( if isEOF
      then return []
      else
        ( do
            line <- hGetLine hdl
            bs <- readFlockFile hdl
            case newBoid $ map read (words line) of
              Just b -> return (b : bs)
              Nothing -> return bs
        )
    )

saveFlock :: Maybe String -> Int -> [Boid] -> IO Int
saveFlock outDir n = case outDir of
  Just dn -> \bs -> do
    withFile file WriteMode $ writeFlockFile bs
    return (n + 1)
    where
      file = dn ++ "/" ++ show n ++ ".txt"
  Nothing -> \_ -> return 0

writeFlockFile :: [Boid] -> Handle -> IO ()
writeFlockFile [] hdl = hClose hdl
writeFlockFile (b : bs) hdl = do
  hPrint hdl b
```

```
  writeFlockFile bs hdl
```

5.5 src/Config.hs

```haskell
module Config (Config (..), WorldSize (..), loadConfig) where

import System.IO (Handle, IOMode (ReadMode), hGetLine, hIsEOF, withFile)

data Config = Config
  { radius :: Float,
    sn :: Float,
    an :: Float,
    cn :: Float,
    maxVel :: Float,
    wSize :: WorldSize
  }
  deriving (Show)

data WorldSize = Infinite | Size Float

instance Show WorldSize where
  show Infinite = ""
  show (Size f) = show f

defaultConfig :: Config
defaultConfig =
  Config
    { radius = 5,
      sn = 1.8,
      an = 0.08,
      cn = 0.3,
      maxVel = 10,
      wSize = Infinite
    }

loadConfig :: Maybe String -> IO Config
loadConfig file = case file of
  Just fn -> withFile fn ReadMode readConfigFile
  Nothing -> return defaultConfig

readConfigFile :: Handle -> IO Config
readConfigFile hdl = do
  isEOF <- hIsEOF hdl
  ( if isEOF
      then return defaultConfig
      else
        ( do
            line <- hGetLine hdl
            cfg <- readConfigFile hdl
            let cfg' = case words line of
                  ["radius", arg] -> cfg {radius = read arg}
                  ["sn", arg] -> cfg {sn = read arg}
                  ["an", arg] -> cfg {an = read arg}
                  ["cn", arg] -> cfg {cn = read arg}
                  ["maxVel", arg] -> cfg {maxVel = read arg}
```

```haskell
                  ["wSize", arg] -> cfg {wSize = Size $ read arg}
                  _ -> cfg
           return cfg'
       )
   )
```

5.6 src/Sim.hs

```haskell
module Sim (runSim, runSimCollect) where

import Boid (Boid, updateBoid)
import Config (Config)
import Control.DeepSeq (force)
import Control.Parallel.Strategies (parList, parListChunk, rdeepseq, rpar, rseq, runEval, using)

data ParStrat = Seq | TwoPart | Chunks Int | ParList

updateWith :: ParStrat -> Config -> [Boid] -> [Boid]
updateWith Seq = updateSeq
updateWith TwoPart = updateTwoPart
updateWith (Chunks n) = updateChunks n
updateWith ParList = updateParList

runSim :: Config -> [Boid] -> Int -> [Boid]
runSim config flock0 nIter = case runSimCollect config flock0 nIter of
  [] -> flock0
  (flockN : _) -> flockN

runSimCollect :: Config -> [Boid] -> Int -> [[Boid]]
runSimCollect cfg flock0 nIter = foldl simLoop [flock0] [1 .. nIter]
  where
    simLoop :: [[Boid]] -> Int -> [[Boid]]
    simLoop [] _ = []
    simLoop flocks@(flock : _) _ = updateWith Seq cfg flock : flocks

updateSeq :: Config -> [Boid] -> [Boid]
updateSeq cfg flock = map (updateBoid cfg flock) flock

updateTwoPart :: Config -> [Boid] -> [Boid]
updateTwoPart cfg flock = runEval $ do
  as' <- rpar (force (map (updateBoid cfg flock) as))
  bs' <- rpar (force (map (updateBoid cfg flock) bs))
  _ <- rseq as'
  _ <- rseq bs'
  return (as' ++ bs')
  where
    (as, bs) = splitAt (length flock `div` 2) flock

updateChunks :: Int -> Config -> [Boid] -> [Boid]
updateChunks numChunks cfg flock = flock'
  where
    flock' = map (updateBoid cfg flock) flock `using` parListChunk chunkSize rdeepseq
    chunkSize = length flock' `div` numChunks

updateParList :: Config -> [Boid] -> [Boid]
updateParList cfg flock = flock'
```

```haskell
  where
    flock' = map (updateBoid cfg flock) flock `using` parList rdeepseq
```

5.7 src/Utils.hs

```haskell
module Utils (vBound, vScaleTo, vx, vy, vxy, wrapDisp, vWrap) where

import Data.Fixed (mod')
import Linear.Metric (Metric (norm), normalize)
import Linear.V2 (V2 (V2))
import Linear.Vector ((*^))

vBound :: Float -> V2 Float -> V2 Float
vBound lim v = vScaleTo (norm v `min` lim) v

vScaleTo :: Float -> V2 Float -> V2 Float
vScaleTo n v = n *^ normalize v

vWrap :: Float -> V2 Float -> V2 Float
vWrap size (V2 x y) = V2 (wrap x) (wrap y)
  where
    wrap a = (a + size / 2) `mod'` size - (size / 2)

vx :: V2 a -> a
vx (V2 x _) = x

vy :: V2 a -> a
vy (V2 _ y) = y

vxy :: V2 a -> (a, a)
vxy (V2 x y) = (x, y)

wrapDisp :: Float -> V2 Float -> V2 Float -> V2 Float
wrapDisp size p1 p2 = V2 dx' dy'
  where
    dx'
      | abs dx > 0.5 * size = dx + (if x2 > x1 then -size else size)
      | otherwise = dx
    dy'
      | abs dy > 0.5 * size = dy + (if y2 > y1 then -size else size)
      | otherwise = dy
    dx = x2 - x1
    dy = y2 - y1
    (V2 x1 y1) = vWrap size p1
    (V2 x2 y2) = vWrap size p2
```