Newtonian Particles

Nathan Cuevas (njc2150)
December 2021

Introduction

This report will discuss my findings as I implement a Newtonian particle simulation program in parallel Haskell. The simulation consists of *m* particles (of even mass and size) that are enclosed in a rectangular 2-D container, the particles each have an initial position and velocity, and are free to bounce around the container and each other. Intuitively, since particle motion is largely independent of other particles (until there is a collision), it is tempting to explore the extent it can be parallelized.

Given the initial conditions for each particle, this program will generate a list of lists (can be thought of as a *n x m* matrix) where the rows of the matrix will represent the state of each particle (position and velocity) and the columns of the matrix will represent the state of a particle at each step that the simulation was run for. This program will also have an animation component, which allows the user to have a tangible understanding of the data. The animation feature is more of an "add-on" and will not be subject to parallel optimizations and benchmarking. Each particle will adhere to basic Newtonian physics and will be influenced by gravitational forces, conservation of energy and conservation of linear momentum, friction and inelastic energy loss. Factors such as spin, angular momentum, aerodynamic drag will not be simulated here.

Numerical Simulation

The simulation technique that will be used here (albeit not an extremely sophisticated one), sees similar techniques used for many real world problems in computation. These techniques form the basis for numerical analysis, which models real world phenomena (particle motion, heat transfer, fluid flow, etc.) by iterating their governing equations to best approximate its behavior. These simulations can be computationally

intensive, and it is not uncommon to see real world numerical simulations run for hours or days.

Simulations similar to these are also used in games where having realistic physics is required. Games also need this simulation process to be performant since slow implementations can lead to losing valuable framerate and/or latency in a multiplayer setting.

Model

This section will discuss the governing rules that are required to simulate this system. These rules mostly follow real life physics but are not completely exact; however, these approximations are enough to get realistic particle motion. This model also isn't based on any particular algorithm, just my recollection of college Physics.

Throughout this report, I will define the "state" of a particle as a data type that contains following four floating point numbers: x velocity (v_x) y velocity (v_y) x position (x) y position (y), which is represented in my Haskell program as the `ParticleState` record. The movement of the particles is governed by fundamental physics. Let's discuss the 3 computations that are involved in this simulation.

Basic Motion:

The following are the equations that calculate the state of a particle at step n, written in terms of the state of the particle at $n - 1$:

- $x^{(n)} = x^{(n-1)} + v_x^{(n)} \Delta t$
- $y^{(n)} = y^{(n-1)} + v_y^{(n)} \Delta t$
- $v_x^{(n)} = v_x^{(n-1)}$
- $v_y^{(n)} = v_y^{(n-1)} - g\,\Delta t$

Where g is the acceleration due to gravity. The above logic is implemented in the function `updateState`.

Wall Collision:

We can say that a particle has collided with a wall if one of the following is satisfied:

- $x - r <$ left wall x position
- $x + r >$ right wall x position
- $y - r <$ bottom wall y position
- $y + r >$ top wall y position

Where r is the radius of the particle. If one of the above conditions is satisfied, we will update the state using the following instructions:

- negate the velocity normal to the wall and multiply by α
- multiply the velocity tangent to the wall by β
- move the particle back into the box by shifting its position into the box by the same amount of distance it went into the wall

Where $0 < \beta \leq 1$ is the friction loss coefficient and $0 < \alpha \leq 1$ is the elastic coefficient. `adjustForWallBounce` implements this wall bounce logic.

Particle Collision:

Finally, the last case that can happen in this simulation is a collision between two particles. Two particles *A* and *B* are colliding if the distances of their centers is less than $2r$. The calculations of this case are similar to the previous wall bounce case, except with the extra complication of momentum transfer between the particles. Also the calculations will be done in the normal and tangential basis (instead of the x and y basis) to simplify the math. First define the new basis by finding the normal and tangential component

- getting the normal vector: This is the unit vector that is in the direction of the line that connects the center points of the 2 colliding particles.
- getting the tangent vector: define the tangent vector as the unit vector that is perpendicular to the normal vector

The directions of these vectors are arbitrary, just be sure to stick with it for the remainder of the calculation. Now update the states of the colliding particles *A* and *B*.

- In the tangential direction: the velocities of A and B will be scaled by β.
- In the normal direction: the velocities of A and B will be swapped, then scaled by α.

The above logic for particle collisions is implemented in `collision`.

Sequential Implementation

The first challenge is designing a robust sequential implementation for this program. One of the most powerful things about Haskell is the ease of parallelization once a sequential implementation is created. To begin, an initial condition is fed into the program's core algorithm (which resides in Compute.hs) which is of type `[ParticleState]`. We get this initial condition by parsing a csv file whose path is specified as an argument to the executable. The entry point to the algorithm is `computeMatrix` or `compute` (the latter only spits out a list of the final states of each particle while the former computes the entire *n x m* matrix). The following is the core algorithm:

First Attempt at Sequential Algorithm:

1. For each particle:
    - update the state of the particle by using the basic motion equations
2. For each particle:
    - if wall collision:
        - update the state using the wall collision logic
3. For each particle:
    - if collided with another particle:
        - update the state of the colliding particles by using the particle collision logic
4. Save updated state
5. goto (1.)

As the program runs, we are accumulating the updated state at each step.

During my first implementation of this algorithm, I ran into excessive garbage collecting by the ghc compiler. The below figure shows the threadscope result of my first go at implementing this algorithm.

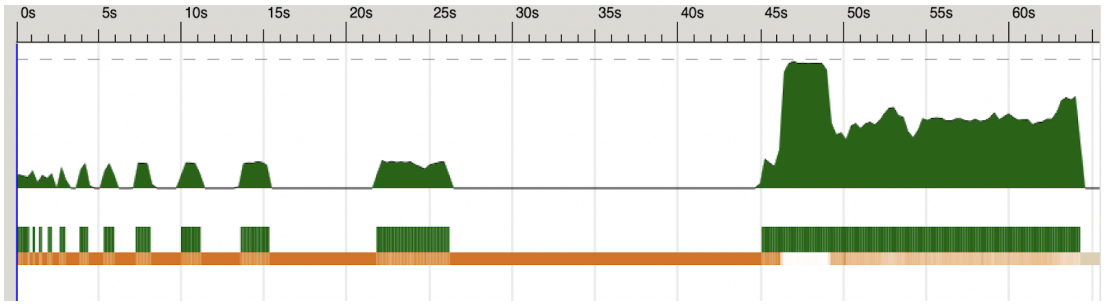

Figure 1:
Threadscope Profile of First Sequential Implementation

As observed, a significant amount of time is spent garbage collecting. It turns out that the reason for this is because of Haskell's laziness. Since we don't actually evaluate the output of `computeMatrix` or `compute` until the very end of the program (in my case, I print the result of the particle's final states to force evaluation), we basically have a chain of thunks that are evaluated all at once at the end causing significant activity by the garbage collector and hereby draining system resources.

Forcing the algorithm to do a strict evaluation at each iteration of the algorithm is critical to prevent the evaluation of the thunk chain all at once. There is a convenient function in the `Control.deepSeq` module called `force` that performs a strict evaluation of the arguments. Now we can rewrite the sequential algorithm as:

Updated Sequential Algorithm:

1. For each particle:
    - update the state of the particle by using the basic motion equations
2. For each particle:
    - if wall collision:
        - update the state using the wall collision algorithm
3. For each particle:
    - if collided with another particle:
        - update the state of the colliding particles by using the particle collision algorithm
4. **Do a strict evaluation of the updated state** then save
5. goto (1.)

Rewriting this algorithm with this simple change caused significant speedup:

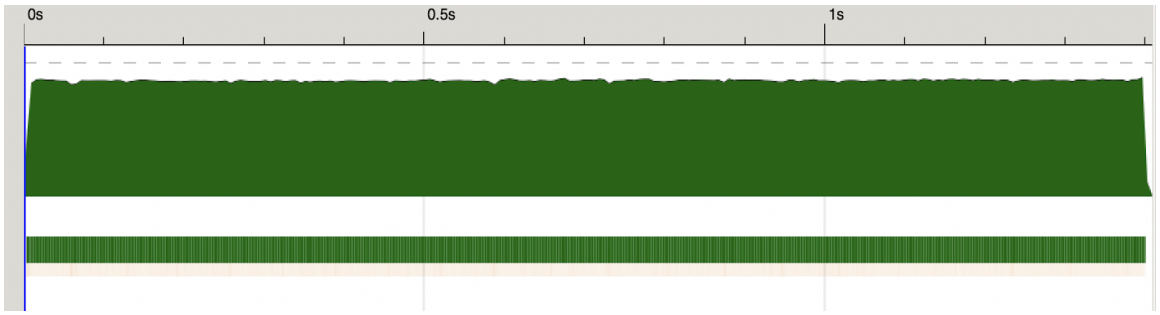

Figure 2:
Threadscope Profile of Updated Sequential Impl

Notice that this implementation now runs in ~1.5s as opposed to ~65s before. The time spent garbage collecting is now negligible.

When implementing `force` into the algorithm, the `ParticleState` record had to be updated to be an instance of `NFData`. Doing so will provide `force` with instructions on how to strictly evaluate `ParticleState`.

```haskell
data ParticleState = ParticleState
  { xPos :: !Float
  , yPos :: !Float
  , xVel :: !Float
  , yVel :: !Float
  } deriving Show

instance NFData ParticleState where
  rnf (ParticleState x' y' vx' vy') =
    rnf x' `seq` rnf y' `seq` rnf vx' `seq` rnf vy'
```

The following are the performance figures of the sequential algorithm running the simulation for *n = 2700*.

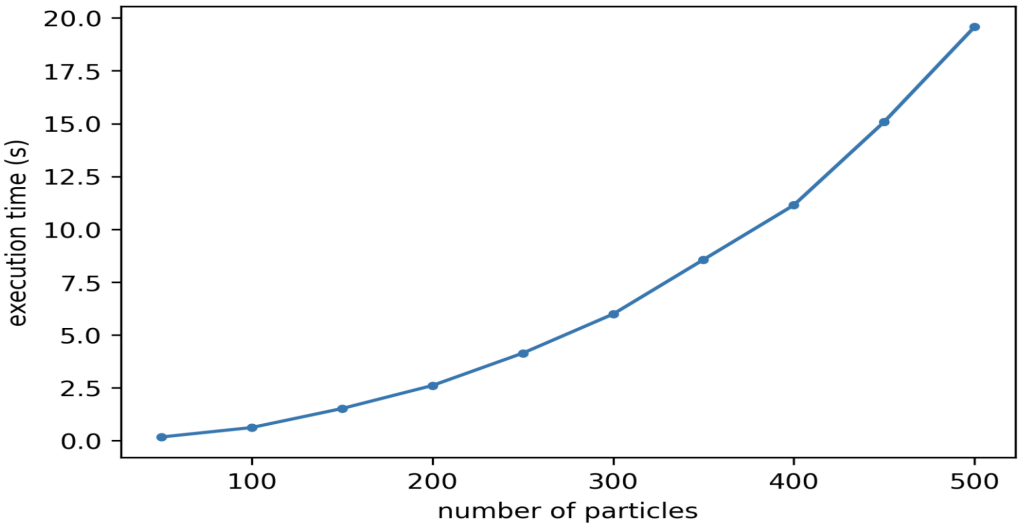

Figure 3:
Runtime of Sequential Impl w/ Varying Number of Particles

Notice that the plot seems to run in polynomial time w.r.t the input size. This intuitively makes sense since step 3 of the sequential algorithm is essentially a “double for loop” since checking for collisions involves checking the distance to each individual.

Parallel Implementation

In the source code, the function called `nextStep` is what runs each step of our sequential algorithm.

```
nextStep :: Float -> Config -> [ParticleState] -> Int ->
[ParticleState]
nextStep dt config currStates step = force $
adjustForWallBounce stepped config
  where
    postCollisions, stepped :: [ParticleState]
    postCollisions = adjustForCollisions currStates config
    stepped = map (updateState dt config) postCollisions
```

Which is a subroutine of the `compute` function (which gets our initial states as an argument).

```
compute :: [ParticleState] -> Float -> Int -> Config ->
[ParticleState]
compute initial dt nSteps config =
  foldl (nextStepParCollision dt config 50) initial [1..nSteps]
```

(Note that the order of steps 1-3 in `nextStep` is slightly different from the order in the pseudo code for the sequential implementation.This order doesn't actually matter too much)

In this section, I will be discussing two major tests I did to parallelize this simulation algorithm. To test a new parallel algorithm, just put it in place of `nextStep` inside of the `compute` function.

First Attempt at Parallelizing:

The first thing I tried to parallelize is step 1 of the sequential algorithm. I thought this was an obvious place to parallelize since this step doesn't require any interactions with the other particles, so the computations for the state updates can be done in parallel. I created a new function called `nextStepChunkedDeep` which is a chunked parallel version of `nextStep` which uses the `rdeepseq` strategy on step 1. I then put this function in place of `nextStep`. Running some quick benchmark testing on it, it was clear that this was slower than the sequential implementation.

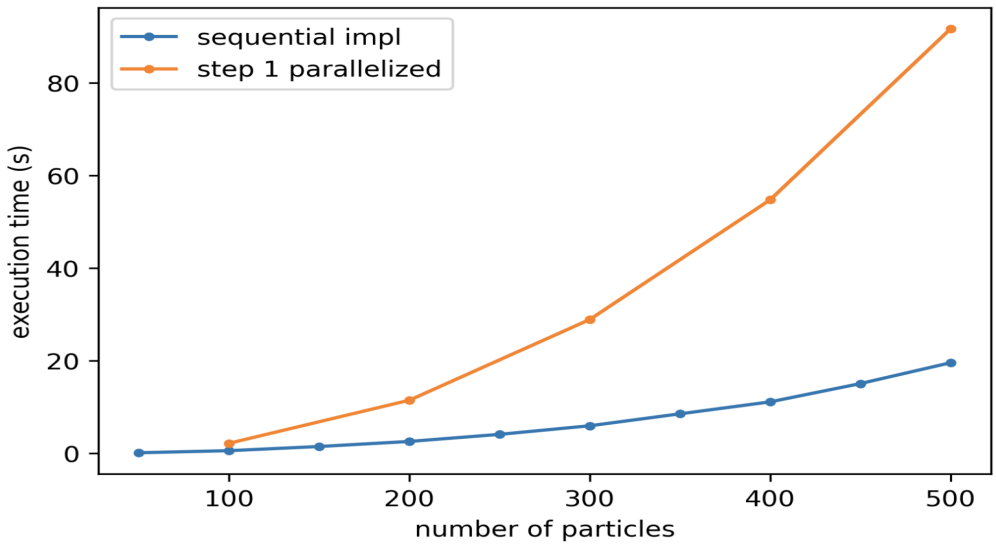

Figure 4:

Benchmark of First Parallel Attempt Compared to Sequential

(Note: the chunk size for this parallel tests was arbitrarily fixed to (num particles)/10, the number of cores was fixed to 4)

This poor parallel performance is likely due to the overhead of creating the sparks outweighing the cost of just running it sequentially. Since step 1 is a pretty lightweight computation in its own right (just simple arithmetic), it appears to be cheaper to *not* parallelize it.

A Much Better Parallelizing Attempt:

I decided to scrap parallelizing step 1 and step 2 completely. This is because these computations are inexpensive and it is clear from the results in the previous step that there is negative performance gain when parallelizing since the overhead of spark creation dominates. However, step 3 in the sequential algorithm could potentially be parallelized. Step 3 can be imagined as a "double for loop". This is because checking for particle collisions requires us to loop through all the other particles and check if there is another particle within collision range. This "inner loop" can be run in parallel to save time; however, there is a tradeoff that some computations might be done more than once (imagine the case that there is a collision between 2 particles, the computation for the new state will happen twice because we are evaluating in parallel). By defining a new function called `nextStepParCollision`, which is a parallel chunked implementation of step 3 using the `rdeepseq` strategy, I was able to

benchmark this new optimization and conclude that the program runs significantly faster as a result.

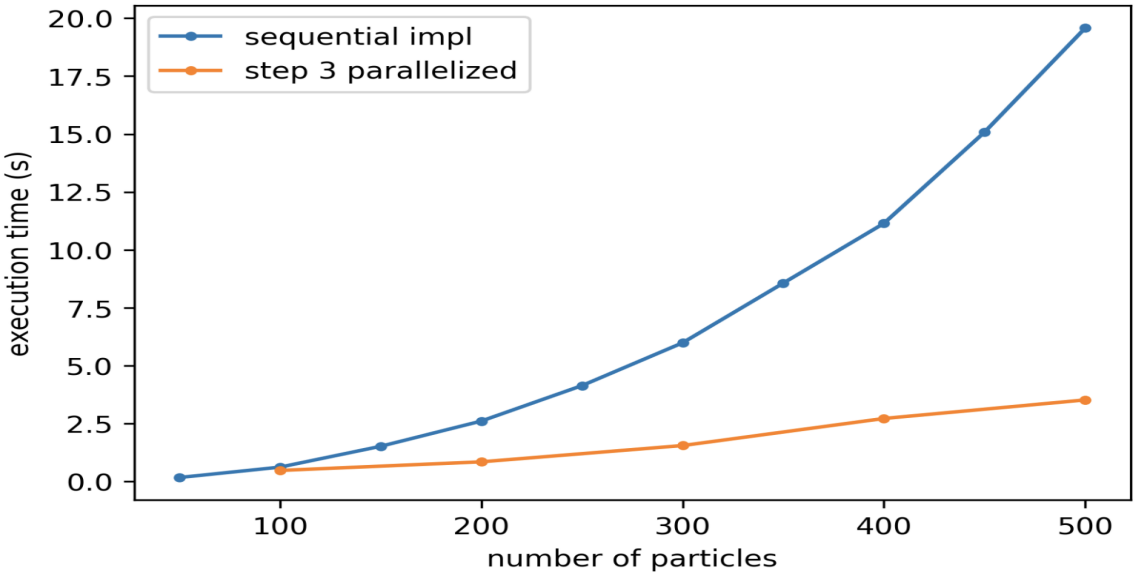

Figure 5:

Benchmark of New Parallel Algorithm Compared to Sequential

(Note: the chunk size for this parallel tests was arbitrarily fixed to (num particles)/10, the number of cores was fixed to 4)

It is clear that the performance of this new parallel optimization makes the program run significantly faster compared to the sequential version. It also seems as if the program runs in essentially linear time, which follows intuition since we are parallelizing the “inner for loop.”

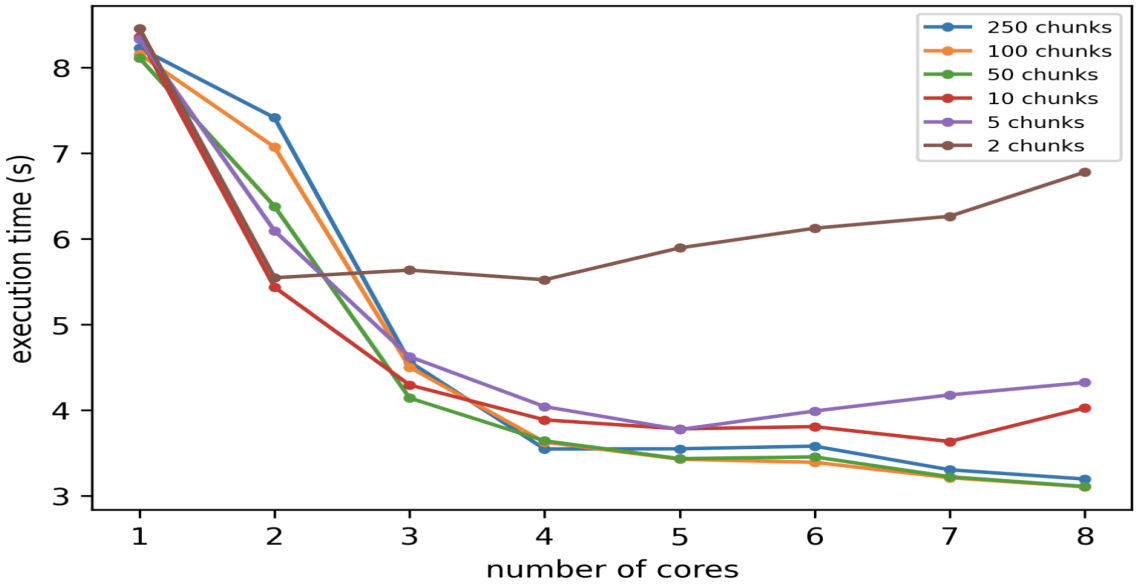

Figure 6:
Execution Time of The New Parallel Algorithm vs. Num Cores. Ran w/ 500 Particles.

On my machine with 4 physical cores with up to 8 core hyper threading, I observed that at as low as 10 chunks (for 500 particles) is sufficient enough to see these performance gains (see figure 5). In the case of 4 cores, 500 particles and a chunk size of 50, around 95% of the sparks were converted. Judging from the threadscope profile, the load balancing was also very good. Overall, the benchmarking stats of this new parallel algorithm looks healthy.

```
SPARKS: 135000 (129407 converted, 0 overflowed, 0 dud, 3205
GC'd, 2388 fizzled)
```

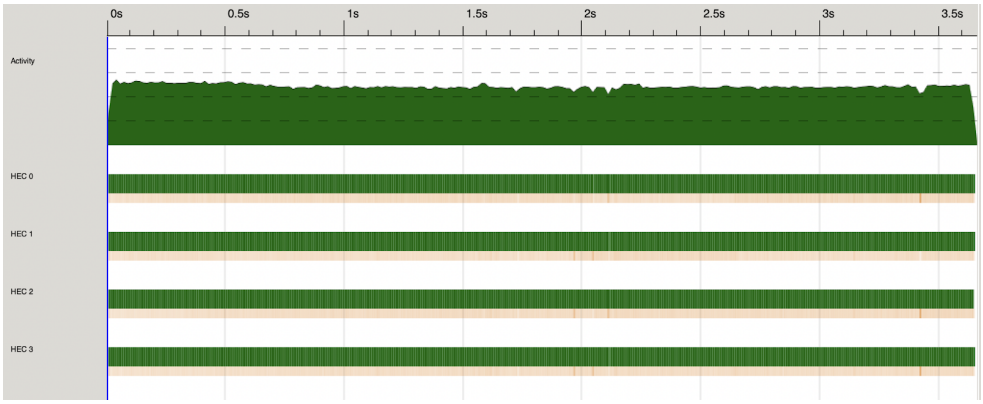

Figure 7:
Threadscope Profile of Better Parallel Attempt

Conclusion

Parallelization does indeed give significant performance gains over the sequential version. It is seen from this exercise that there is a tradeoff when choosing to parallelize a part of a program. Additionally, the parts that seem to parallelize well at first inspection, sometimes don't work out in the end, so it is crucial to benchmark parallel prototypes to justify the performance gains (if any). I also found that while Haskell's laziness is one of the best features of the language, it leads to detrimental performance losses if not understood properly.

Source Code

Main.hs

```
module Main where

import System.Environment(getArgs, getProgName)
import System.Exit(die)
import Animate
import Compute
import Parse
import Types


totalTime ,steps, maxSize :: Int
totalTime = 60
steps = totalTime * fps
```

```
maxSize = 7000000

dt :: Float
dt = 1 / (fromIntegral fps)


extractPosVectors :: [[ParticleState]] -> [[PosVector]]
extractPosVectors pSll = map helper pSll
  where
    helper :: [ParticleState] -> [PosVector]
    helper pSl = map helper2 pSl
    helper2 :: ParticleState -> PosVector
    helper2 pS = PosVector (xPos pS) (yPos pS)


main :: IO ()
main =
  do
    args <- getArgs
    (filename, configPath, animate) <-
      case args of
        [f, c] ->
          return (f, c, False)
        [f, c, "-animate"] ->
          return (f, c, True)
        _   ->
          do
            pn <- getProgName
            die $ "Usage: "++pn++" <filename> <config>
[-animate]"
    contents <- readFile filename
    configContents <- readFile configPath
    let
      config = extractConfig configContents
      preset = contentsToData contents
      m = length preset
    if steps * m > maxSize then
      error "exceeded max size"
    else if animate then
      runAnimation $ extractPosVectors $ computeMatrix preset
dt steps config
    else
```

```
      print $ compute preset dt steps config
```

Compute.hs

```
module Compute where

import Types
import Control.Parallel.Strategies
import Control.DeepSeq


getParticleData :: ParticleState -> (Float, Float, Float,
Float)
getParticleData pS = (xPos pS, yPos pS, xVel pS, yVel pS)


getConfigData :: Config -> (Float, Float, Float)
getConfigData cG = (g cG, alpha cG, beta cG)


updateState ::  Float -> Config -> ParticleState ->
ParticleState
updateState dt cG prevState =
  prevState {xPos = xPrev + vxPrev * dt, yPos = yPrev + vyNew *
dt, yVel = vyNew}
  where
    (xPrev, yPrev, vxPrev, vyPrev) = getParticleData prevState
    vyNew = vyPrev - g * dt
    (g, _, _) = getConfigData cG


inWall :: Float -> Float -> Maybe Wall
inWall x y
  | x - r < leftWallLoc   = Just LeftWall
  | x + r > rightWallLoc  = Just RightWall
  | y + r > topWallLoc    = Just TopWall
  | y - r < bottomWallLoc = Just BottomWall
  | otherwise = Nothing
  where r = (fromIntegral radius) :: Float


adjustForWallBounce :: [ParticleState] -> Config ->
```

```
[ParticleState]
adjustForWallBounce pSl cG = map bounce pSl
  where
    bounce :: ParticleState -> ParticleState
    bounce pS =
      case inWall x y of
        Just LeftWall   ->
          ParticleState (x+2*(leftWallLoc-x+r)) y
(alpha*(negate vx)) (beta*vy)
        Just RightWall  ->
          ParticleState (x-2*(x+r-rightWallLoc)) y
(alpha*(negate vx)) (beta*vy)
        Just TopWall    ->
          ParticleState x (y-2*(y+r-topWallLoc)) (beta*vx)
(alpha*(negate vy))
        Just BottomWall ->
          ParticleState x (y+2*(bottomWallLoc-y+r)) (beta*vx)
(alpha*(negate vy))
        Nothing         -> pS
      where
      (x, y, vx, vy) = getParticleData pS
      (_, alpha, beta) = getConfigData cG
      r = (fromIntegral radius) :: Float


collision :: ParticleState -> ParticleState -> Config ->
(ParticleState, ParticleState)
collision pS1 pS2 cG
  | d >= 2 * r = (pS1, pS2)
  | otherwise = (new1, new2)
  where
    (_, alpha, beta) = getConfigData cG
    r = (fromIntegral radius) :: Float
    d = sqrt $ (x2-x1)^2 + (y2-y1)^2
    (x1,y1,vx1,vy1) = getParticleData pS1
    (x2,y2,vx2,vy2) = getParticleData pS2
    nx = x2 - x1
    ny = y2 - y1
    thetaN = posAtan2 ny nx
    theta1 = posAtan2 vy1 vx1
    theta2 = posAtan2 vy2 vx2
    phi1 = theta1 - thetaN
```

```
    phi2 = theta2 - thetaN
    mag1 = sqrt $ vx1^2 + vy1^2
    mag2 = sqrt $ vx2^2 + vy2^2
    vn1 = mag1 * (cos phi1)
    vt1 = mag1 * (sin phi1)
    vn2 = mag2 * (cos phi2)
    vt2 = mag2 * (sin phi2)
    vt2' = beta * vt2
    vt1' = beta * vt1
    vn1' = alpha * vn2
    vn2' = alpha * vn1
    mag1' = sqrt $ vn1'^2 + vt1'^2
    mag2' = sqrt $ vn2'^2 + vt2'^2
    phi1' = posAtan2 vt1' vn1'
    phi2' = posAtan2 vt2' vn2'
    theta1' = thetaN + phi1'
    theta2' = thetaN + phi2'
    angle = posAtan2 ny nx
    pen = 2 * r - d
    new1 = ParticleState x1 y1 (mag1' * (cos theta1')) (mag1' *
(sin theta1'))
    new2 = ParticleState (x2 + pen * cos angle) (y2 + pen * sin
angle) (mag2' * (cos theta2')) (mag2' * (sin theta2'))
    posAtan2 :: Float -> Float -> Float
    posAtan2 y x
      | res < 0   = 2 * pi + res
      | otherwise = res
      where res = atan2 y x


split :: Int -> [a] -> [[a]]
split numChunks xs = chunk (length xs `quot` numChunks) xs


chunk :: Int -> [a] -> [[a]]
chunk n [] = []
chunk n xs = as : chunk n bs
  where (as,bs) = splitAt n xs


adjustForCollisions :: [ParticleState] -> Config ->
[ParticleState]
```

```
adjustForCollisions [] _          = []
adjustForCollisions (pS1:[])  _   = [pS1]
adjustForCollisions (pS1:rem) cG = pS1New:(adjustForCollisions
remNew cG)
  where
    (pS1New, remNew) = helper pS1 rem
    helper :: ParticleState -> [ParticleState] ->
(ParticleState, [ParticleState])
    helper pS [] = (pS, [])
    helper pS (x:xs) = (a, newX : r)
      where
        (newPS, newX) = collision pS x cG
        (a, r) = helper newPS xs


adjustForCollisions2 :: [ParticleState] -> Config ->
[ParticleState]
adjustForCollisions2 pSl cG = map helper pSl
  where
    helper :: ParticleState -> ParticleState
    helper pS = foldl helper2 pS pSl
      where
        helper2 :: ParticleState -> ParticleState ->
ParticleState
        helper2 s e
          | e == pS     = s
          | first == pS = s
          | otherwise   = first
            where (first, _) = collision pS e cG


adjustForCollisions2Chunked :: [ParticleState] -> Config -> Int
-> [ParticleState]
adjustForCollisions2Chunked pSl cG numChunks = concat (map (map
helper) splitted `using` parList rdeepseq)
  where
    splitted = split numChunks pSl
    helper :: ParticleState -> ParticleState
    helper pS = foldl helper2 pS pSl
      where
        helper2 :: ParticleState -> ParticleState ->
ParticleState
```

```
        helper2 s e
          | e == pS     = s
          | first == pS = s
          | otherwise   = first
            where (first, _) = collision pS e cG


nextStep :: Float -> Config -> [ParticleState] -> Int ->
[ParticleState]
nextStep dt config currStates step = force $
adjustForWallBounce stepped config
  where
    postCollisions, stepped :: [ParticleState]
    postCollisions = adjustForCollisions currStates config
    stepped = map (updateState dt config) postCollisions


nextStepChunkedForce :: Float -> Config -> Int ->
[ParticleState] -> Int -> [ParticleState]
nextStepChunkedForce dt config numChunks currStates step =
  force $ adjustForWallBounce stepped config
    where
      postCollisions, stepped :: [ParticleState]
      postCollisions = adjustForCollisions currStates config
      splitted = split numChunks postCollisions
      stepped = concat (map (map (updateState dt config))
splitted `using` parList rseq)


nextStepChunkedDeep :: Float -> Config -> Int ->
[ParticleState] -> Int -> [ParticleState]
nextStepChunkedDeep dt config numChunks currStates step =
  adjustForWallBounce stepped config
    where
      postCollisions, stepped :: [ParticleState]
      postCollisions = adjustForCollisions currStates config
      splitted = split numChunks postCollisions
      stepped = concat (map (map (updateState dt config))
splitted `using` parList rdeepseq)


nextStepParCollision :: Float -> Config -> Int ->
```

```
[ParticleState] -> Int -> [ParticleState]
nextStepParCollision dt config numChunks currStates step =
force $ adjustForWallBounce stepped config
  where
    postCollisions, stepped :: [ParticleState]
    postCollisions = adjustForCollisions2Chunked currStates
config numChunks
    stepped = map (updateState dt config) postCollisions


compute :: [ParticleState] -> Float -> Int -> Config ->
[ParticleState]
compute initial dt nSteps config =
  foldl (nextStep dt config) initial [1..nSteps]


computeMatrix :: [ParticleState] -> Float -> Int -> Config ->
[[ParticleState]]
computeMatrix initial dt nSteps config =
  reverse $ foldl helper [initial] [1..nSteps]
    where
      helper :: [[ParticleState]] -> Int -> [[ParticleState]]
      helper matrix@(front:_) step = (nextStep dt config front
step) : matrix
      helper _ _ = error "computeMatrix helper error"
```

Types.hs

```
module Types where
import Control.DeepSeq


data ParticleState = ParticleState
  { xPos :: !Float
  , yPos :: !Float
  , xVel :: !Float
  , yVel :: !Float
  } deriving Show


instance NFData ParticleState where
```

```
  rnf (ParticleState x' y' vx' vy') =
    rnf x' `seq` rnf y' `seq` rnf vx' `seq` rnf vy'


instance Eq ParticleState where
  pS1 == pS2 =
    (xPos pS1 == xPos pS2) &&
    (yPos pS1 == yPos pS2) &&
    (xVel pS1 == xVel pS2) &&
    (yVel pS1 == yVel pS2)


data PosVector = PosVector
  { xComp :: !Float
  , yComp :: !Float
  } deriving Show


data Wall =
    LeftWall
  | RightWall
  | TopWall
  | BottomWall
  deriving Show


data Config = Config
  { g     :: !Float
  , alpha :: !Float
  , beta  :: !Float
  } deriving Show


defaultConfig :: Config
defaultConfig =
  Config { g = 9.81, alpha = 0.93, beta = 0.98 }


fps, width, height, radius :: Int
fps = 45
width = 740
height = 740
```

```
radius = 3


rightWallLoc, leftWallLoc, topWallLoc, bottomWallLoc :: Float
rightWallLoc = fromIntegral $ width `div` 2
leftWallLoc = negate rightWallLoc
topWallLoc = fromIntegral $ height `div` 2
bottomWallLoc = negate topWallLoc
```

Parse.hs

```
module Parse where

import Data.List
import Types


splitComma :: String -> [String]
splitComma s =
  case span (/= ',') s of
    (start , "")      -> [start]
    (start , ',':rem) -> start:(splitComma rem)
    _                 -> error "parse error in splitComma"


listToParticleState :: [String] -> ParticleState
listToParticleState (x:y:vx:vy:[]) =
  ParticleState (read x) (read y) (read vx) (read vy)
listToParticleState _ = error "mismatched dimensions in data
file"


contentsToData :: String -> [ParticleState]
contentsToData contents =
  map (listToParticleState . splitComma) $ words contents


extractConfig :: String -> Config
extractConfig contents =
  foldl helper defaultConfig $ splitComma contents
  where
    helper :: Config -> String -> Config
```

```
    helper c s
      | isPrefixOf "g=" s     =
        c { g = stripPrefixFloat "g=" s }
      | isPrefixOf "alpha=" s =
        c { alpha = stripPrefixFloat "alpha=" s }
      | isPrefixOf "beta="  s =
        c { beta = stripPrefixFloat "beta=" s }
      | otherwise = error "invalid config file"
    stripPrefixFloat :: String -> String -> Float
    stripPrefixFloat prefix s=
      case stripPrefix prefix s of
        Just post -> read post
        Nothing   -> error "invalid config file"
```

Animate.hs

```
module Animate where

import Graphics.Gloss
import Graphics.Gloss.Data.ViewPort
import Types


offset :: Int
offset = 100


window :: Display
window = InWindow "Particles" (width, height) (offset, offset)


background :: Color
background = black


update :: ViewPort -> Float -> [[PosVector]] -> [[PosVector]]
update _ _ []     = []
update _ _ (_:ps) = ps


render :: [[PosVector]] -> Picture
render []    = blank   -- when the simulation is done, show a
```

```
blank screen
render (p:_) =
  pictures $ map getTranslation p
  where
    getTranslation :: PosVector -> Picture
    getTranslation state =
      translate x y $ c $ circleSolid $ fromIntegral radius
        where
          x = xComp state
          y = yComp state
          c = color red


runAnimation :: [[PosVector]] -> IO ()
runAnimation ds = simulate window background fps ds render
update
```

The source code can also be found on:
github.com/nathanjcuevas/Newtonian-Particles
I will be tagging my submission on github as well so it can easily be reverted to if needed.