Modeling Galaxies: Barnes-Hut Approximation

Hans Montero, Rhys Murray
{hjm2133, ram2269}@columbia.edu

cs4995.003 Final Project
December 23rd, 2020

1 n-body Problem

Given a set of celestial bodies with mass, initial velocity, and initial position, we would like to simulate the motion of these bodies over time under the influence of gravity. Such simulations allow us to model the collisions and interactions of large-scale galaxy clusters. While there is a closed form solution for $n = 2$, no such formula exists for $n \geq 3$, so computationally expensive numerical solutions are required. These numerical methods vary in their approaches to calculating the effect of gravity on each body. We know from classical kinematics that the gravitational force on one body by another separated by distance r is given by the following (where G is the gravitational constant):

$$F = G\frac{m_1 m_2}{r^2}$$

A naïve algorithm would run in $\mathcal{O}(n^2)$ time, where for each time step, the algorithm calculates the net force on a given body by iterating over the entire set of bodies and accounting for every single body, regardless of distance. This algorithm clearly will not scale well at the galaxy-level with a huge number of bodies. Further overhead would be added by calculating the positions of the bodies at each step and displaying them. We must seek a more efficient algorithm if we wish to seamlessly model large systems over more fine-grained periods of time.

2 Barnes-Hut Approximation

The Barnes-Hut Approximation seeks to cut down computation by grouping very distant masses together into one larger mass. The first step is to divide up the n bodies into a quadtree (for 2D simulations) to group together nearby masses. Then, for each body in the tree, we calculate the contribution of other bodies in the same way as the naïve algorithm. However, if a group of bodies is sufficiently far away, we aggregate them and use their combined mass and center of gravity for our computation. By leveraging this approximation, the algorithm's time complexity improves to $\mathcal{O}(n \log n)$. Whether a region is considered "distant" or not depends on the ratio of its size to its distance from the body. If this ratio exceeds a threshold value, the region is approximated as above. This threshold value can be adjusted depending on desired speed or accuracy of the simulation.

2.1 Data Types

We defined three main data types for our implementation: `Body`, `QuadInfo`, and `QuadTree`.

The `Body` type tracks a body's position and velocity vectors, along with other physical properties:

```
data Body = Body {  mass :: Double -- For force calculation
                 ,  xCord :: Double
                 ,  yCord :: Double
                 ,  xVel :: Double
                 ,  yVel :: Double
                 ,  radius :: Double -- For visualization
                 }
```

The `QuadInfo` type holds information about a quadrant that we use in Quadtree insertion and force calculation:

```
data QuadInfo = QuadInfo {  xl :: Double -- Quadrant boundaries
                         ,  xr :: Double
                         ,  yb :: Double
                         ,  yt :: Double
                         ,  com :: CenterMass -- (x, y, mass)
                         }
```

We can now define the `QuadTree` type, which is a recursive algebraic type:

```
data QuadTree = QuadTree QuadTree QuadTree QuadTree QuadTree QuadInfo
              | QuadNode (Maybe Body) QuadInfo
```

This representation allows us to express the two cases for a quadrant: either it is occupied by at most 1 body, or it has been further divided into quadrants because there are at least two bodies in it.

2.2 Implementation

The main loop of the iterative Barnes-Hut algorithm works by approximating the acceleration due to gravity on every body in the tree, updating their velocity and positions, and recreating a new `QuadTree` out of the updated bodies for each timestep `dt`. Here is our main `barnesHut` function that allows us to run the algorithm iteratively:

```
barnesHut :: QuadTree -> Double -> QuadTree
barnesHut oldTree dt = newTree
  where oldbodyList = toList oldTree
        updatedBodyList = map (\b -> approximateForce oldTree b dt) oldbodyList
        movedBodyList = map (doTimeStep dt) updatedBodyList
        newTree = calcCOM $ fromList movedBodyList (getInfo oldTree)
```

Note two operations in this routine: `fromList` and `map ... approximateForce`. We will attempt to parallelize them later on.

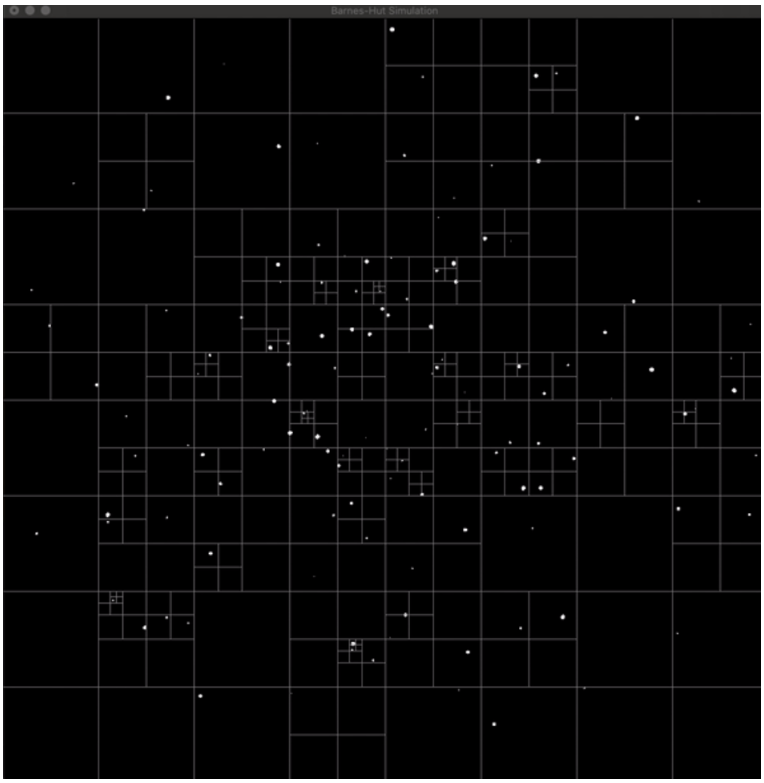
2.3 Animation

Rendering the simulation as a 2D animation is done via Haskell's `gloss` library which provides handy functions to draw on the screen and hides the details of working directly with OpenGL.

```haskell
runSimulation :: QuadTree -> (QuadTree -> Double -> QuadTree) -> IO ()
runSimulation qt updateFunc = simulate (InWindow "Barnes-Hut" (1500, 1500) (10, 10))
                              black 60
                              qt
                              (\ qt' -> pictures $ drawQuadTree qt' [])
                              (\_ dt qt' -> updateFunc qt' (float2Double dt))
```

Running the simulation via `gloss` is as simple as passing a list of `Pictures` to draw the current state and a function to generate a new state. In our case, we draw all of the bodies on the screen at their current positions and update our state (the `QuadTree`) by running the Barnes-Hut algorithm on it.

Here's a screenshot of the animation, showing a celestial system orbiting around a supermassive black hole in the center:



We also drew the quadrant boundaries in light grey to show that no two bodies share the same quadrant.

3 Parallelization

To further optimize this approximation algorithm, we can parallelize the two major computational steps. First, the quadtree construction can be delegated to four threads, as each "quadrant" of the tree can be constructed independently. We expect to see some minor speed up here, as we are not guaranteed to see even workloads for each of those quadrant constructing threads. Second, and more importantly, we can parallelize the quadtree traversal for calculating the gravitational force on a certain body. This is a perfect example of data parallelism, given the enormous amounts of bodies in realistic models and the fact that these traversals are independent of one another. Parallelizing this step should greatly speed up the runtime of the algorithm, much more so than the parallelization of the quadtree construction. We chose a reference simulation of 1000 bodies simulated for 500 timesteps as a benchmark for comparing our different attempts at parallelization.

What follows is a report of the different strategies we used and how they performed at improving the two aforementioned computational steps.

3.1 Force Computation

3.1.1 parMap and parBuf Strategies

To implement any of the parallel strategies, we must first create instances of the `NFData` type classes for our custom data types. After this small change, implementing `parMap` and `parBuffer` in our simulation is as simple as changing our update function to call the appropriate parallel function.

```
barnesHutParMap :: QuadTree -> Double -> QuadTree
barnesHutParMap oldTree dt = newTree
  where oldbodyList = toList oldTree
        updatedBodyList = parMap rdeepseq (\b -> approximateForce oldTree b dt) oldbodyList
        movedBodyList = map (doTimeStep dt) updatedBodyList
        newTree = calcCOM $ fromList movedBodyList (getInfo oldTree)
```

Note that we must use `rdeepseq` to deeply evaluate each of the bodies to normal form. This strategy creates sparks to evaluate the forces on each of the bodies in parallel. Spark results can be seen below:

```
SPARKS: 50000 (47371 converted, 0 overflowed, 0 dud, 2203 GC'd, 426 fizzled)
```

We can see that most sparks are being converted, but a significant percentage is being garbage collected or is fizzling, indicating that each work unit is too small. `parBuffer` reduces this problem by sparking only a certain number of sparks at a time (100 in our case), but we still ran into similar problems with unbalanced workloads. This problem can be eliminated entirely by chunking the list and giving each core more work. However, even with this highly unbalanced work load, we saw significant speedups that will be detailed below.

3.1.2 parListChunks and parBufChunks Strategies

Seeing how the previous two strategies were too fine-grained, we sought out chunking strategies to ensure that the CPUs have decently sized work items to compute. We tried the built in `parListChunks` strategy and set it up in a way that would allow us to programmatically try different chunk sizes so we could figure out which was the ideal size for a given CPU count:

```
barnesHutParListChunks :: Int -> QuadTree -> Double -> QuadTree
barnesHutParListChunks cz oldTree dt = newTree
  where oldbodyList = toList oldTree
        newTree = fromList (map (\b ->
            doTimeStep dt $ approximateForce oldTree b dt) oldbodyList
            `using` parListChunk cz rdeepseq) (getInfo oldTree)
```

This strategy would ensure us that less sparks would be created and that each spark would have more meaningful work to do. Surely enough, the spark stats showed that this strategy was indeed more successful:

```
SPARKS: 6500 (6500 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)
```

In the next section, we will show some more experiments that we ran for `parListChunks` to find its ideal chunk size across different CPU counts and to see its spark behavior through Threadscope.

Out of curiosity, we also developed a strategy of our own, named `barnesHutParBufChunks`. Seeing how well `parListChunks` performed, we were wondering if we could combine its benefits with `parBuffer`, which would ensure that we don't overwhelm the system with too many sparks at any given time. We imagined this would come in handy for larger datasets.

```
barnesHutParBufChunks :: Int -> QuadTree -> Double -> QuadTree
barnesHutParBufChunks cz oldTree dt = newTree
  where oldbodyList = toList oldTree
        updatedBodyList = concat (map (map (\b ->
            doTimeStep dt $ approximateForce oldTree b dt)) (chunksOf cz oldbodyList)
            `using` parBuffer 100 rdeepseq)
        newTree = calcCOM $ fromListPar updatedBodyList (getInfo oldTree)
```

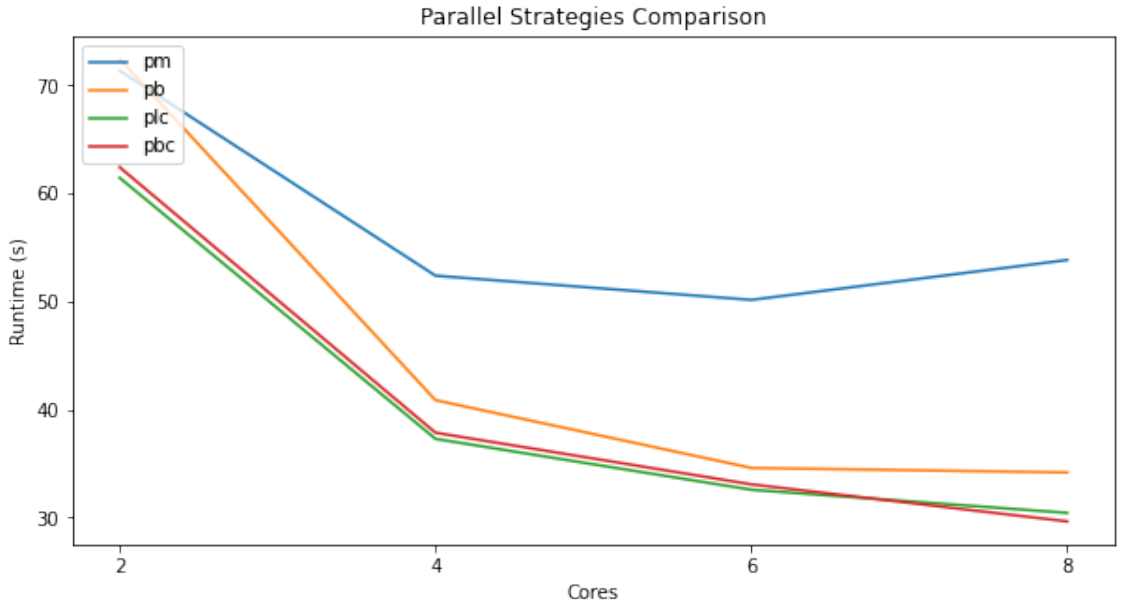
We had to manually replicate the chunking from `parListChunks` by using `chunksOf` to break up the dataset. The spark stats showed that this strategy was indeed helpful in controlling spark creation:

```
SPARKS: 4500 (4500 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)
```

In the next section, we'll compare the performances of `barnesHutParListChunks` and `barnesHutParBufChunks` to see which is actually more preferable in terms of runtime.

3.1.3 Analysis

To begin, we can plot all four of the strategies we mentioned on the same set of axes to see which one is worth investigating a little deeper. For the chunking strategies, we used their optimal chunk sizes for this experiment (see discussion below on optimal chunk size).



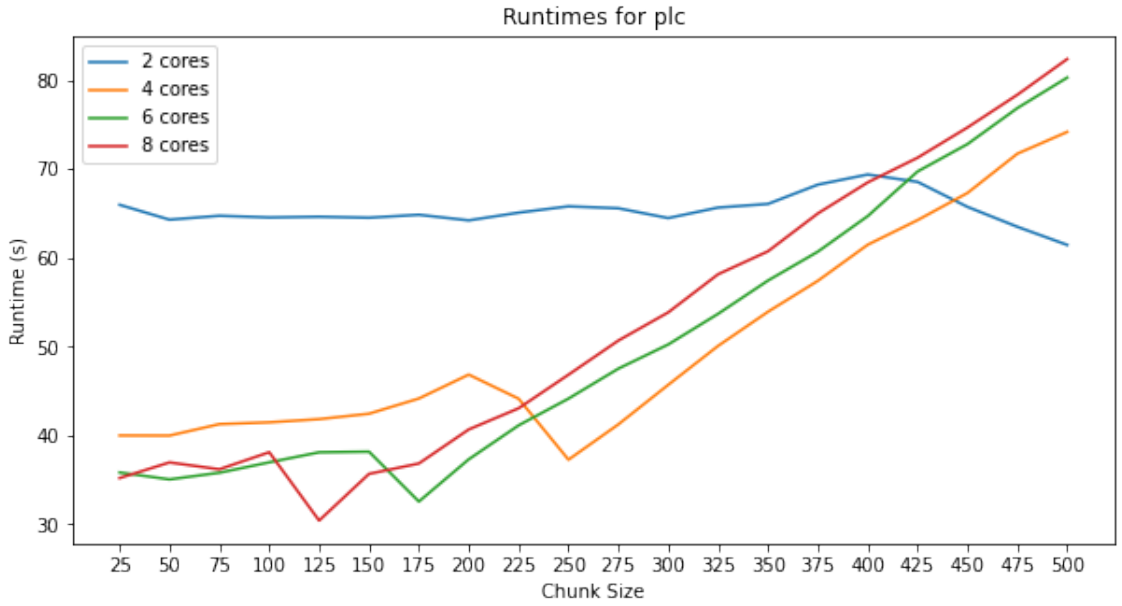
As expected, the chunking strategies perform significantly better than the non-chunking strategies. This is probably because the chunking strategies create less sparks and thus suffer less overhead in their creation.

We were also curious how `barnesHutParListChunks` and `barnesHutParBufChunks` would perform in our experiments. As we can see, they pretty much performed similarly, with the latter only winning out very slightly at 8 cores. Below, we tabulated the speedup results from all four strategies on different core counts compared to the original sequential algorithm. The speedups didn't match up with core count 1:1, but that was to be expected – there are still significant portions of the algorithm that aren't parallelized. Also, we believe the superlinear speedups, i.e. a 2.5x speedup for 2 cores, is due to increases in memory allocation when run with multiple cores. This leads to less frequent garbage collection and thus even faster runtimes.

Table 1: Speedups achieved

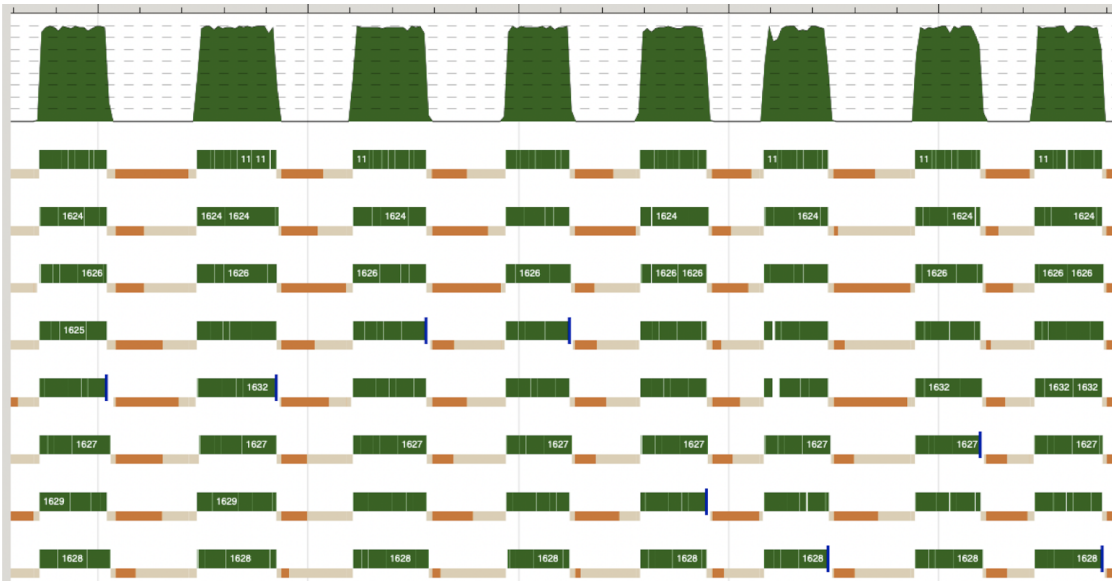
Cores	**pm**	**pb**	**plc**	**pbc**
2	2.4625	2.4261	2.8597	2.8148
4	3.3543	4.3000	4.7130	4.6420
6	3.5041	5.0795	5.3945	5.3130
8	3.2639	5.1419	5.7739	5.9278

The next question to ask here, seeing that the chunking strategies are the best, is "what is the ideal chunk size for chunking strategies?". We designed a new experiment where we would try different chunk sizes at different CPU counts for `barnesHutParListChunks`. Here are the results:



All four plots achieve their minimum at `chunkSize = numBodies/numCPUs`. This chunk size ensures that each core will have a balanced amount of work to do and that no other core is just standing by. Users of the chunking strategies should therefore calculate this ratio and use it as the chunk size to achieve optimal runtime and resource usage.

We've seen that `barnesHutParListChunks` has a perfect conversion rate and that we can configure it to use an optimal chunk size. Finally, we should analyze the performance of the algorithm using Threadscope to observe actual CPU usage.



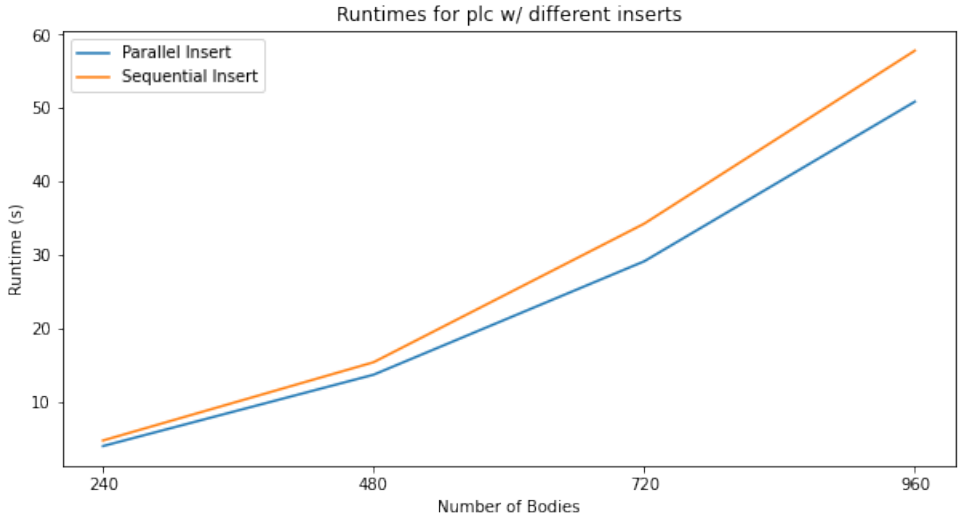
On the one hand, we're very pleased with the CPU usage. We are almost always using all eight cores to their fullest extent, which means the strategy really does ensure good balancing. There are some dips though, and this probably has to do with the single-threaded aspects of the algorithm (like `QuadTree` creation and printing out information). On the other hand, the pauses due to the garbage collector are a little troubling. Our implementation seems to use a very large amount of memory that often gets discarded, which means the GC will have to step in quite frequently. This has to do with the `QuadTree` creation per iteration. We can't easily update it given the pure nature of Haskell so we instead create an entirely new one. The sizes of the `QuadTree`s can be quite large, so it probably is costly to discard them so frequently like we are. Nonetheless, because `barnesHutParListChunks` saw excellent spark conversion, speedups, and CPU usage, we are quite proud of the implementation!

3.2 Parallel `QuadTree` Construction

As mentioned above, we can also exploit natural parallelism in the `QuadTree` construction. Each quadrant can be constructed independently of one another and then stitched back together in the end.

```
1  fromListPar :: [Body] -> QuadInfo -> QuadTree
2  fromListPar bs qi = QuadTree nw' ne' sw' se' qi
3      where (QuadTree nw ne sw se _) = emptyQTree minNum maxNum minNum maxNum
4            (minNum, maxNum) = squareBounds qi bs
5            makeTreeForQuad quad = (flip fromList (getInfo quad) . filter (inQuad quad)) bs
6            (nw', ne', sw', se') = runEval $ do
7                                              parNW <- rparWith rdeepseq (makeTreeForQuad nw)
8                                              parNE <- rparWith rdeepseq (makeTreeForQuad ne)
9                                              parSW <- rparWith rdeepseq (makeTreeForQuad sw)
10                                             parSE <- rparWith rdeepseq (makeTreeForQuad se)
11                                             return (parNW, parNE, parSW, parSE)
```

Runtimes for the plc strategy with and without parallel insertion can be seen below. `fromListPar` provides a noticeable yet very small decrease in runtime. This increases with more and more bodies in the simulation as we'd expect. However, once again the workload is unbalanced, and we began to see similar fizzling problems as with parMap and parBuffer. The number of bodies in each quadrant is vastly different in some timesteps than others, and this imbalance leads to occasional fizzles.



4 Conclusion

In our journey to optimize the runtime of the Barnes-Hut algorithm and parallelize it, we learned a few key lessons about parallelization. First, not all work is worth being parallelized. We saw this when we were implementing parallel `QuadTree` construction. If the work units are too small, the overhead from spark management will win out and reduce speedups. Next, we learned that more cores doesn't always mean better performance. This is very algorithm dependent and requires good work balancing. We observed this while experimenting for the ideal chunk size for `barnesHutParListChunks` – some chunk sizes are so non-ideal that they yield worse runtimes as you increase the number of cores. This all ties into the most important lesson we got out of this: parallelization in Haskell requires experimentation. In other languages, we would often just blackbox away parallelization mechanisms and just assume we'll see speedups. While that can still be true in Haskell, we have more power here in fine tuning that speedup. We are able to choose different strategies which, as we've proven above, yield different speedups. While the process of experimentation isn't as easy as simply blackboxing it all away, it is certainly more rewarding.

The project repository can be found here: `https://github.com/hmontero1205/barnes-hut`

5 Code Listing

Building from Source

```
$ # Requires stack.
$ stack install
$ # Assuming you have ~/.local/bin in your path,
$ barnes-hut -h
usage: barnes-hut [-r <min-radius> <max-radius> -n <num-bodies> -m <max-mass> |
                   -i <iterations> -n <numBodies> [pm|plc <chunk-size>|pb|pbc <chunk-size>]]
```

```haskell
{- Main.hs: Entrypoint for our implementation -}
module Main where
import QuadTree
import System.Random
import Physics
import Visualize
import Control.Parallel.Strategies(parMap, rdeepseq, parListChunk, using, parBuffer,
  ↪ Eval)
```

```haskell
import System.Environment (getArgs, getProgName)
import System.Exit
import Data.List.Split(chunksOf)

empty :: QuadTree
empty = emptyQTree (-20000) 20000 (-20000) 20000


barnesHutParMap :: QuadTree -> Double -> QuadTree
barnesHutParMap oldTree dt = newTree
  where oldbodyList = toList oldTree
        updatedBodyList = parMap rdeepseq (\b -> approximateForce oldTree b dt) oldbodyList
        movedBodyList = map (doTimeStep dt) updatedBodyList
        newTree = calcCOM $ fromList movedBodyList (getInfo oldTree)

barnesHutParBufChunks :: Int -> QuadTree -> Double -> QuadTree
barnesHutParBufChunks cz oldTree dt = newTree
  where oldbodyList = toList oldTree
        updatedBodyList = concat (map (map (\b -> doTimeStep dt $ approximateForce oldTree b dt)) (chunksOf cz oldbodyList) `using` parBuffer 100 rdeepseq)
        newTree = calcCOM $ fromListPar updatedBodyList (getInfo oldTree)

barnesHutParListChunks :: Int -> QuadTree -> Double -> QuadTree
barnesHutParListChunks cz oldTree dt = newTree
  where oldbodyList = toList oldTree
        newTree = fromListPar (map (\b -> doTimeStep dt $ approximateForce oldTree b dt) oldbodyList `using` parListChunk cz rdeepseq) (getInfo oldTree)

barnesHutParBuffer :: QuadTree -> Double -> QuadTree
barnesHutParBuffer oldTree dt = newTree
  where oldbodyList = toList oldTree
        updatedBodyList = map (\b -> approximateForce oldTree b dt) oldbodyList `using` parBuffer 100 rdeepseq
        movedBodyList = map (doTimeStep dt) updatedBodyList
        newTree = calcCOM (fromList movedBodyList (getInfo oldTree))

barnesHut :: QuadTree -> Double -> QuadTree
barnesHut oldTree dt = newTree
  where oldbodyList = toList oldTree
        updatedBodyList = map (\b -> approximateForce oldTree b dt) oldbodyList
        movedBodyList = map (doTimeStep dt) updatedBodyList
        newTree = calcCOM $ fromListPar movedBodyList (getInfo oldTree)

makeBHSystem :: Int -> Int -> QuadTree
makeBHSystem n spacing = calcCOM $ insert blackhole $ fromList orbiters (getInfo empty)
    where blackhole = Body 5000000 0 0 0 0 1
          orbiters = [(\x -> generateOrbiter blackhole (fromIntegral x) 10) (spacing + spacing * i) | i <- [0..(n-2)]]

makeBHSystemRandom :: Int -> [Double] -> [Double] -> [Double] -> QuadTree
makeBHSystemRandom n radii angles masses = calcCOM $ insert blackhole $ fromList [generateOrbiterAngle blackhole radius' mass' angle' | (radius', mass', angle') <- combinedList] (getInfo empty)
    where blackhole = Body 500000000 0 0 0 0 1
          combinedList = take n $ zip3 radii angles masses

```

```haskell
simpleLoop :: Int -> (QuadTree -> Double -> QuadTree) -> QuadTree -> Double -> QuadTree
simpleLoop n f tree dt
  | n > 0 = simpleLoop (n - 1) f (f (calcCOM tree) dt) dt
  | otherwise = calcCOM tree

simpleLoop' :: Int -> QuadTree -> Double -> Eval QuadTree
simpleLoop' n tree dt
  | n <= 0 = return tree
  | otherwise = do let oldBodyList = toList tree
                   newBodyList <- parListChunk 24 rdeepseq (map (\b -> doTimeStep dt $ approximateForce tree b dt) oldBodyList)
                   newBodyList' <- rdeepseq newBodyList
                   simpleLoop' (n - 1) (calcCOM $ fromList newBodyList' (getInfo tree)) dt

doUsage :: IO ()
doUsage = do progName <- getProgName
             die $ "usage: " ++ progName ++
                 "[-r <minRadius> <maxRadius> -n <numBodies> -m <maxMass> | -i <iterations> -n <numBodies> [pm|plc <chunk-size>|pb|pbc <chunk-size>]]"

randomlist :: Random a => a -> a -> IO [a]
randomlist a b = fmap (randomRs (a,b)) newStdGen

main :: IO ()
main = do
    args <- getArgs
    case args of
      ["-r", minRadius, maxRadius, "-n",
      numBodies, "-m", maxMass] -> do radii <- randomlist (read minRadius) (read maxRadius :: Double)
                                      angles <- randomlist 0 (2 * pi :: Double)
                                      masses <- randomlist 0 (read maxMass :: Double)
                                      runSimulation (makeBHSystemRandom (read numBodies) radii angles masses) (barnesHutParListChunks ((read numBodies) `div` 4))--(\qt _ -> qt)
      ["-i", its, "-n", nb] -> do radii <- randomlist (1000) (50000 :: Double)
                                  angles <- randomlist 0 (2 * pi :: Double)
                                  masses <- randomlist 0 (1000 :: Double)
                                  print $ simpleLoop (read its) barnesHut (makeBHSystemRandom (read nb) radii angles masses) 0.5
      ["-i", its, "-n", nb, "pm"] -> print $ simpleLoop (read its) barnesHutParMap (bhs (read nb)) 0.5
      ["-i", its, "-n", nb, "plc", cz] -> do radii <- randomlist (1000) (50000 :: Double)
                                             angles <- randomlist 0 (2 * pi :: Double)
                                             masses <- randomlist 0 (1000 :: Double)
                                             print $ simpleLoop (read its) (barnesHutParListChunks $ read cz) (makeBHSystemRandom (read nb) radii angles masses) 0.5
      ["-i", its, "-n", nb, "pbc", cz] -> do radii <- randomlist (1000) (50000 :: Double)
                                             angles <- randomlist 0 (2 * pi :: Double)
                                             masses <- randomlist 0 (1000 :: Double)
```

```
                                                print $ simpleLoop (read its)
                                                ↪  (barnesHutParBufChunks $ read cz)
                                                ↪  (makeBHSystemRandom (read nb) radii
                                                ↪  angles masses) 0.5
      ["-i", its, "-n", nb, "pb"] -> print $ simpleLoop (read its) barnesHutParBuffer
      ↪  (bhs (read nb)) 0.5
      _ -> doUsage
    where bhs nb' = makeBHSystem nb' 1000
```

```
{- QuadTree.hs: Quadtree definition and helpers -}
module QuadTree where
import Control.DeepSeq
import Control.Parallel.Strategies(rdeepseq, runEval, rparWith)

data Body = Body {  mass :: Double
                 ,  xCord :: Double
                 ,  yCord :: Double
                 ,  xVel :: Double
                 ,  yVel :: Double
                 ,  radius :: Double
                 }

instance NFData Body where
    rnf (Body m x y xv yv r) = rnf m `deepseq`
                               rnf x `deepseq`
                               rnf y `deepseq`
                               rnf xv `deepseq`
                               rnf yv `deepseq`
                               rnf r

instance Eq Body where
  b1 == b2 = (xCord b1 == xCord b2) && (yCord b1 == yCord b2)

data CenterMass = CenterMass {  cMass :: Double
                             ,  cx :: Double
                             ,  cy :: Double
                             }

instance NFData CenterMass where
  rnf (CenterMass m x y) = rnf m `deepseq` rnf x `deepseq` rnf y

instance Show CenterMass where
    show (CenterMass ma xx yy) = "COM " ++ show ma ++ " @ " ++ "(" ++ show xx ++ ", " ++
    ↪  show yy ++ ")"

data QuadInfo = QuadInfo {  xl :: Double
                         ,  xr :: Double
                         ,  yb :: Double
                         ,  yt :: Double
                         ,  com :: CenterMass
                         }

instance NFData QuadInfo where
    rnf (QuadInfo xl' xr' yb' yt' com') = rnf xl' `deepseq`
                                          rnf xr' `deepseq`
                                          rnf yb' `deepseq`
```

```
                                        rnf yt' `deepseq`
                                        rnf com'


instance Show QuadInfo where
    show (QuadInfo xxl xxr yyb yyt com') = "QI[ X:" ++ show xxl ++ "-" ++ show xxr ++ ",
    ↪  Y:" ++ show yyb ++ "-" ++ show yyt ++ ", "++ show com' ++ "]"

instance Show Body where
    show (Body m x y xVel' yVel' radius') = "body @ (" ++ show x ++ ", " ++ show y ++ ")
    ↪  -> mass: " ++ show m ++ ", vel: (" ++ show xVel' ++ ", " ++ show yVel' ++ ")" ++
    ↪  ", radius: " ++ show radius'

data QuadTree = QuadTree QuadTree QuadTree QuadTree QuadTree QuadInfo
              | QuadNode (Maybe Body) QuadInfo

instance NFData QuadTree where
    rnf (QuadTree nw ne sw se qi) = rnf nw `deepseq` rnf ne `deepseq` rnf sw `deepseq`
    ↪  rnf se `deepseq` rnf qi
    rnf (QuadNode (Just b) qi) = rnf b `deepseq` rnf qi
    rnf (QuadNode Nothing qi) = rnf qi

getCOMX :: QuadTree -> Double
getCOMX (QuadTree _ _ _ _ qi) = cx . com $ qi
getCOMX (QuadNode _ qi) = cx . com $ qi

getCOMY :: QuadTree -> Double
getCOMY (QuadTree _ _ _ _ qi) = cy . com $ qi
getCOMY (QuadNode _ qi) = cy . com $ qi

getCOMM :: QuadTree -> Double
getCOMM (QuadTree _ _ _ _ qi) = cMass . com $ qi
getCOMM (QuadNode _ qi) = cMass . com $ qi

toList :: QuadTree -> [Body]
toList (QuadNode Nothing _) = []
toList (QuadNode (Just b) _) = [b]
toList (QuadTree nw ne sw se _) = toList nw ++ toList ne ++ toList sw ++ toList se

squareBounds :: QuadInfo -> [Body] -> (Double, Double)
squareBounds qi bs = (minNum, maxNum)
    where xl' = min (xl qi) (minimum $ map xCord bs)
          xr' = max (xr qi) (maximum $ map xCord bs)
          yb' = min (yb qi) (minimum $ map yCord bs)
          yt' = max (yt qi) (maximum $ map yCord bs)
          minNum = min xl' yb' -- ensure we always have a square
          maxNum = max xr' yt'


fromList :: [Body] -> QuadInfo -> QuadTree
fromList bs qi
  | null bs = emptyQTree (xl qi) (xr qi) (yb qi) (yt qi)
  | otherwise = foldl (flip insert) empty bs
    where empty = emptyQTree minNum maxNum minNum maxNum -- Dynamically calculate bounds
    ↪  of new Quadtree
          (minNum, maxNum) = squareBounds qi bs

```

```haskell
getInfo :: QuadTree -> QuadInfo
getInfo (QuadTree _ _ _ _ qi) = qi
getInfo (QuadNode _ qi) = qi

fromListPar :: [Body] -> QuadInfo -> QuadTree
fromListPar bs qi = QuadTree nw' ne' sw' se' qi
    where (QuadTree nw ne sw se _) = emptyQTree minNum maxNum minNum maxNum -- Dynamically calculate bounds of new Quadtree
          (minNum, maxNum) = squareBounds qi bs
          makeTreeForQuad quad = (flip fromList (getInfo quad) . filter (inQuad quad)) bs
          (nw', ne',
           sw', se') = runEval $ do parNW <- rparWith rdeepseq (makeTreeForQuad nw)
                                    parNE <- rparWith rdeepseq (makeTreeForQuad ne)
                                    parSW <- rparWith rdeepseq (makeTreeForQuad sw)
                                    parSE <- rparWith rdeepseq (makeTreeForQuad se)
                                    return (parNW, parNE, parSW, parSE)

emptyQNode :: Double -> Double -> Double -> Double -> QuadTree
emptyQNode xl' xr' yb' yt' = QuadNode Nothing (QuadInfo xl' xr' yb' yt' (CenterMass 0 0 0))

emptyQTree :: Double -> Double -> Double -> Double -> QuadTree
emptyQTree xl' xr' yb' yt' = QuadTree nw ne sw se (QuadInfo xl' xr' yb' yt' (CenterMass 0 0 0))
                        where xm = (xr' + xl') / 2
                              ym = (yt' + yb') / 2
                              nw = emptyQNode xl' xm ym yt'
                              ne = emptyQNode xm xr' ym yt'
                              sw = emptyQNode xl' xm yb' ym
                              se = emptyQNode xm xr' yb' ym

mapQuads :: (QuadTree -> a) -> QuadTree -> [a]
mapQuads f qn@(QuadNode _ _) = [f qn]
mapQuads f (QuadTree nw ne sw se _) = [f nw, f ne, f sw, f se]

foldQuads :: (QuadTree -> a -> a) -> a -> QuadTree -> a
foldQuads f z qn@(QuadNode _ _) = f qn z
foldQuads f z (QuadTree nw ne sw se _) = foldQuads f (foldQuads f (foldQuads f (foldQuads f z se) sw) ne) nw

inQuad :: QuadTree -> Body -> Bool
inQuad qt b = xl qi <= x && xr qi >= x && yt qi >= y && yb qi <= y
            where x = xCord b
                  y = yCord b
                  qi = getInfo qt

combineBodies :: Body -> Body -> Body
combineBodies b1 b2 = b1 {mass = mass b1 + mass b2, xVel = xVel b1 + xVel b2, yVel = yVel b1 + yVel b2}

insert :: Body -> QuadTree -> QuadTree
insert b (QuadNode Nothing qi) = QuadNode (Just b) qi
insert b2 (QuadNode (Just b1) qi)
  | (xCord b1 == xCord b2) && (yCord b1 == yCord b2) = QuadNode (Just $ combineBodies b1 b2) qi
  | otherwise = insert b2 $ insert b1 $ emptyQTree (xl qi) (xr qi) (yb qi) (yt qi)
insert b (QuadTree nw ne sw se qi)
```

```
  | inQuad nw b = QuadTree (insert b nw) ne sw se qi
  | inQuad ne b = QuadTree nw (insert b ne) sw se qi
  | inQuad sw b = QuadTree nw ne (insert b sw) se qi
  | inQuad se b = QuadTree nw ne sw (insert b se) qi
  | otherwise = error "Couldn't find QuadTree to insert body"

traversePrint :: QuadTree -> Int -> String
traversePrint n@(QuadNode _ _) _ = "\\_ " ++ show n
traversePrint qt@(QuadTree _ _ _ _ qi) lvl = concat $ prInfo : branches
    where branches = mapQuads (\q -> "\n" ++ replicate lvl '-' ++ traversePrint q (lvl + 1)) qt
          prInfo = (if lvl /= 0 then "\\_ " else "") ++ show qi

instance Show QuadTree where
     show (QuadNode b qi) = show b ++ " " ++ show qi
     show qt = traversePrint qt 0
```

```
{- Physics.hs: Logic for calculating force and movement -}
module Physics where
import QuadTree

thetaThreshold :: Double
thetaThreshold = 1

g :: Double
g = 50

density :: Double
density = 1/10 -- Object of mass 10 is radius 100, in mass / radius

combineBodies :: Body -> Body -> Body
combineBodies b1 b2 = b1 {mass = mass b1 + mass b2, xVel = xVel b1 + xVel b2, yVel = yVel b1 + yVel b2}

calcCOM :: QuadTree -> QuadTree
calcCOM (QuadNode Nothing qi) = QuadNode Nothing qi
calcCOM (QuadNode (Just b) qi) = QuadNode (Just b) (qi {com = CenterMass (mass b) (xCord b) (yCord b)})
calcCOM qt@(QuadTree _ _ _ _ qi) = QuadTree nw' ne' sw' se' (qi {com = CenterMass totMass newX newY})
    where qs@[nw', ne', sw', se'] = mapQuads calcCOM qt
          totMass = foldr (\q tm -> tm + getCOMM q) 0 qs
          newX = foldr (\q wx -> wx + getCOMM q * getCOMX q) 0 qs / totMass
          newY = foldr (\q wy -> wy + getCOMM q * getCOMY q) 0 qs / totMass

approximateForce :: QuadTree -> Body -> Double -> Body -- Run Barnes Hut
approximateForce (QuadNode Nothing _) b _ = b -- nothing to compute
approximateForce (QuadNode (Just b1) _) b dt = if b == b1 then b else updateVelocity b b1 dt
approximateForce qt@(QuadTree _ _ _ _ qi) b dt
  | theta < thetaThreshold  = updateVelocity b referenceMass dt-- Treat this quadrant as a single mass
  | otherwise = foldQuads (\qt' b' -> approximateForce qt' b' dt) b qt
  where (xDiff, yDiff) = (xCord b - getCOMX qt, yCord b - getCOMY qt)
        distance = xDiff * xDiff + yDiff * yDiff
        theta = (xr qi - xl qi) / sqrt distance
```

```
        referenceMass = Body (getCOMM qt) (getCOMX qt) (getCOMY qt) 0 0 0 -- Consider the
        ↪  COM a body for calculation

doTimeStep :: Double -> Body -> Body
doTimeStep timeStep b = b {xCord = xCord b + xVel b * timeStep, yCord = yCord b + yVel b
↪  * timeStep}

updateVelocity :: Body -> Body -> Double -> Body
updateVelocity bodyToUpdate otherBody dt
  | bodyToUpdate == otherBody = bodyToUpdate
  | otherwise = bodyToUpdate {xVel = xVel bodyToUpdate - xVelChange * dt, yVel = yVel
  ↪  bodyToUpdate - yVelChange * dt}
  where (xDiff, yDiff) = (xCord bodyToUpdate - xCord otherBody, yCord bodyToUpdate -
  ↪  yCord otherBody)
        distance = xDiff * xDiff + yDiff * yDiff
        angleToBody = atan2 yDiff xDiff
        xVelChange = g * cos angleToBody * (mass otherBody / distance)
        yVelChange = g * sin angleToBody * (mass otherBody / distance)

circularVelocity :: Double -> Double -> Double
circularVelocity massSun radius' = sqrt (g * massSun / radius')

generateOrbiter :: Body -> Double -> Double -> Body
generateOrbiter sun radius' mass' = Body mass' (xCord sun + radius') (yCord sun) (xVel
↪  sun) (yVel sun + velocity) (mass' / density)-- Start at same y level
  where velocity = circularVelocity (mass sun) radius'

generateOrbiterAngle :: Body -> Double -> Double -> Double -> Body
generateOrbiterAngle sun radius' mass' angle = Body mass' (xPos) (yPos) (xVel') (yVel')
↪  (mass' / density)-- Start at same y level
  where velocity = circularVelocity (mass sun) radius'
        xVel' = xVel sun + velocity * sin (angle + pi / 2 :: Double)
        yVel' = yVel sun + velocity * cos (angle + pi / 2 :: Double)
        xPos = xCord sun + (sin angle) * radius'
        yPos = xCord sun + (cos angle) * radius'
```

```
{- Visualize.hs: Interface with gloss library -}
module Visualize where

import Graphics.Gloss
import QuadTree
import GHC.Float

drawBody :: Body -> Picture
drawBody b = Color white $ Translate x y (circleSolid (realToFrac $ radius b))
    where x = realToFrac $ xCord b
          y = realToFrac $ yCord b

drawQuadTree :: QuadTree -> [Picture] -> [Picture]
drawQuadTree (QuadNode Nothing qi) pics = drawBox qi : pics
drawQuadTree (QuadNode (Just b) qi) pics = drawBox qi : drawBody b : pics
drawQuadTree qt@(QuadTree _ _ _ _ qi) pics = drawBox qi : foldQuads drawQuadTree pics qt
↪  ++ pics

drawBox :: QuadInfo -> Picture
```

```haskell
drawBox qi = Color (greyN 0.5) $ Translate x y (rectangleWire (realToFrac (xr qi - xl qi)) (realToFrac (yt qi - yb qi)))
  where x = realToFrac (xr qi + xl qi) / 2
        y = realToFrac (yt qi + yb qi) / 2


runSimulation :: QuadTree -> (QuadTree -> Double -> QuadTree) -> IO ()
runSimulation qt updateFunc = simulate (InWindow "Barnes-Hut Simulation" (1500, 1500) (10, 10))
                              black 60
                              qt
                              (\ qt' -> pictures $ drawQuadTree qt' [])
                              (\_ dt qt' -> updateFunc qt' (float2Double dt))
```

```yaml
# package.yaml: Build configs
name:                barnes-hut
version:             0.1.0.0
github:              "hmontero1205/barnes-hut"
license:             BSD3
author:              "Hans Montero, Rhys Murray"
maintainer:          "hjm2133@columbia.edu, ram2269@columbia.edu"
copyright:           "2020 Nuss Tendie"

extra-source-files:
- README.md
- ChangeLog.md

description:         Please see the README on GitHub at <https://github.com/hmontero1205/barnes-hut#readme>

dependencies:
- base >= 4.7 && < 5
- gloss
- parallel
- deepseq
- split
- random

library:
  source-dirs: src

executables:
  barnes-hut:
    main:                Main.hs
    source-dirs:         app
    ghc-options:
    - -Wall
    - -O2
    - -threaded
    - -rtsopts
    - -eventlog
    dependencies:
    - barnes-hut
    - gloss

tests:
```

```
42  barnes-hut-test:
43    main:                Spec.hs
44    source-dirs:         test
45    ghc-options:
46    - -threaded
47    - -rtsopts
48    dependencies:
49    - barnes-hut
```