Ptcls: n-Body Stimulation

Rong Bai (rb3512)

December 23, 2021

1 n-Body Stimulation

1.1 Introduction

The N-body problem is a common physical simulation problem. In the N-body system, each particle body interacts with the rest of the other particles, resulting in corresponding physical phenomena. Celestial body simulation is a very classic N-body system. The trajectory of a celestial body ultimately depends on the combined force of the gravitational forces of all the remaining celestial bodies on it.

The goal is to give a set of particles and simulate their coordinate changes over time. The interaction force between particles can be expressed by eq. (1), and then by eq. (2) and eq. (3), we can get the speed of the particle at the next time, and finally calculate the new coordinate by Δt.

$$F = G\frac{Mm}{r^3} \tag{1}$$

$$F = ma \tag{2}$$

$$\Delta v = a\Delta t \tag{3}$$

1.2 Direct Solution

for every particle p1 in list do

for do

end

end

Algorithm 1 Direct Solution

$list \leftarrow$ particles
$\Delta t \leftarrow$ time span of stimulation
for every particle $p1$ in list **do**
  coordinate $\leftarrow$ current position of p1
  for every other particle p2 in list **do**
    $F \leftarrow$ gravity force of $p2$ over $p1$
    $a \leftarrow F$ / mass of $p1$
    v $\leftarrow$ velocity of $p1 + a * \Delta t$
    coordinate $\leftarrow coordinate + v * \Delta t$
  end for
  update old coordinate with new value
end for

Time complexity of direct solution can be $O(N^2)$. There are some other approximate methods for n-body problem, this project will use Barnes-Hut algorithm as an improvement.

1.3 Barnes-Hut Approximation

The simulation volume is usually divided up into squares via an quadtree, so that only particles from nearby squares need to be treated individually, and particles in distant squares can be treated as a single large particle centered at the square's center of mass. This can dramatically reduce the number of particle pair interactions that must be computed. The time complexity is $O(NlogN)$.

Algorithm 2 Barnes-Hut

Build quad tree from particle list
for every particle p in list **do**
  F <- calculate tree force over particle
  update velocity of particle
  update position of particle
end for
Update entire particle list

2 Implementation

2.1 Data Structure

1. `Vec` is a two mension vector in space, which is used to indicate position.
2. `Particle` is the object that this project want to stimulate. It has attributes of coordinate, velocity and mass.
3. `Squard` is a square region in space that contains several particles inside. `center` is the mass center of total particles and `mass` is the sum mass. `topleft` and `bottomright` shows the boundary of each `Squard`.

4. `Tree` can either be a intern node `Tree` with four sub trees or a leaf node `Leaf` which contains only one particle. Each tree node has a `Squard` obtaining necessary computing attributes. As tree level grows, the size of squard is decreasing.

```haskell
data Vec = Vec {x :: Double, y :: Double} deriving (Eq)

data Particle = Particle {coord :: Vec, v :: Vec, m :: Double} deriving (Eq)

data Squard = Squard {center :: Vec, topleft :: Vec, bottomright :: Vec,
                        mass :: Double} deriving (Eq)

data Tree = Tree {subtree1 :: Tree, subtree2 :: Tree, subtree3 :: Tree,
                    subtree4 :: Tree, squard :: Squard}
          | Leaf {particle :: Maybe Particle, squard :: Squard} deriving (Eq)
```

2.2 Barnes-Hut

2.2.1 Build Tree

1. Add particles to an empty tree. Given a particle `p1`. If add to `Tree`, then add this particle to the sub tree where it belongs in space. If add to `Leaf (Just p2)`, then turn this leaf node into `Tree`, put `p1` and `p2` in sub trees. If add to `Leaf Nothing`, then update the leaf.
2. Calculate `squard` for tree nodes bottom up. Use the mass center info of sub trees to update its own `squard`.

```haskell
addParticle :: Particle -> Tree -> Tree
addParticle p (Leaf Nothing s)
  | p `isInSquard` s = Leaf {particle = Just p, squard = s}
  | otherwise = Leaf {particle = Nothing, squard = s}
addParticle p (Leaf (Just p') s)
  | p `isInSquard` s = addParticle p $
                        addParticle p' (emptyTree (topleft s) (bottomright s))
  | otherwise = Leaf (Just p') s
addParticle p (Tree t1 t2 t3 t4 s)
  | p `isInSquard` squard t1 = Tree (addParticle p t1) t2 t3 t4 s
  | p `isInSquard` squard t2 = Tree t1 (addParticle p t2) t3 t4 s
  | p `isInSquard` squard t3 = Tree t1 t2 (addParticle p t3) t4 s
  | p `isInSquard` squard t4 = Tree t1 t2 t3 (addParticle p t4) s
  | otherwise = Tree t1 t2 t3 t4 s
```

```haskell
calcSquard :: Tree -> Tree
calcSquard (Leaf Nothing s) = Leaf Nothing s
calcSquard (Leaf (Just p) s) = Leaf (Just p) s {center = coord p, mass = m p}
calcSquard tree@(Tree t1 t2 t3 t4 s) = Tree t1' t2' t3' t4' s'
  where
    subtrees@[t1', t2', t3', t4'] = mapTree calcSquard tree
    s' = s {center = Vec newX newY, mass = totalMass}
    totalMass = foldr (\t acc -> acc + getMass t) 0 subtrees
    newX = foldr (\t acc -> acc + getCenterX t * getMass t) 0 subtrees / totalMass
    newY = foldr (\t acc -> acc + getCenterY t * getMass t) 0 subtrees / totalMass
```

2.2.2 Update Particles

1. Update velocity. When compute a tree's force over a particle, consider their relative distance. If it is far enough than threshold, the entire tree will be calculated as one big particle.
2. Update position. Use new velocity and delta time to compute particle's new position.

```haskell
updateParticle :: Tree -> Double -> Double -> Particle -> Particle
updateParticle tree g dt p = updateP (updateV p tree g dt) dt
```

```haskell
updateP :: Particle -> Double -> Particle
updateP p dt = p {coord = coord p + v p *. dt}
```

```haskell
updateV :: Particle -> Tree -> Double -> Double -> Particle
updateV p (Leaf Nothing _) _ _ = p
updateV p1 (Leaf (Just p2) s) g dt
  | coord p1 == coord p2 = p1
  | otherwise = p1 {v = v p1 + deltaV p1 p2 g dt}
updateV p1 tree@(Tree _ _ _ _ s) g dt
  | isCongregate = (p1 {v = v p1 + deltaV p1 p2 g dt})
  | otherwise = foldTree (\t p -> updateV p t g dt) p1 tree
  where
    p2 = defaultParticle {coord = center s, m = mass s}
    r = dist (coord p1) (center s)
    squardSize = distX (topleft s) (bottomright s)
    theta = abs $ squardSize / r
    isCongregate = theta < thetaThreshold
```

3 Parallelism

3.1 simpleRpar

```
calcSquardPar :: Tree -> Tree
calcSquardPar (Leaf Nothing s) = Leaf Nothing s
calcSquardPar (Leaf (Just p) s) = Leaf (Just p) s {center = coord p, mass = m p}
calcSquardPar tree@(Tree t1 t2 t3 t4 s) = runEval $ do
  t1' <- rpar $ calcSquardPar t1
  t2' <- rpar $ calcSquardPar t2
  t3' <- rpar $ calcSquardPar t3
  t4' <- rpar $ calcSquardPar t4
  rdeepseq t1'
  rdeepseq t2'
  rdeepseq t3'
  rdeepseq t4'

  totalMass <- rpar $ foldr (\t acc -> acc + getMass t) 0 [t1', t2', t3', t4']
  rdeepseq totalMass
  newX <- rpar $ foldr (\t acc -> acc + getCenterX t * getMass t) 0 [t1', t2', t3', t4'] / totalMass
  newY <- rpar $ foldr (\t acc -> acc + getCenterY t * getMass t) 0 [t1', t2', t3', t4'] / totalMass
  rdeepseq newX
  rdeepseq newY
  return $ Tree t1' t2' t3' t4' s {center = Vec newX newY, mass = totalMass}
```

```haskell
updateParticlePar :: Tree -> Double -> Double -> Particle -> Particle
updateParticlePar (Tree t1 t2 t3 t4 _) g dt p = runEval $ do
  p1' <- rpar $ updateP (updateV p t1 g dt) dt
  p2' <- rpar $ updateP (updateV p t2 g dt) dt
  p3' <- rpar $ updateP (updateV p t3 g dt) dt
  p4' <- rpar $ updateP (updateV p t4 g dt) dt
  rdeepseq p1'
  rdeepseq p2'
  rdeepseq p3'
  rdeepseq p4'
  return p {coord = coord p1' + coord p2' + coord p3' + coord p4' - 3 * (coord p), v = v p1' + v p2' + v p
```

Both `addParticle` and `calcSquard` is implemented with recursive tree traverse, so they can be converted to a parallel variation.

```
SPARKS: 347892 (29276 converted, 0 overflowed, 0 dud, 249449 GC'd, 69167 fizzled)
```

It shows that this strategy creates too many sparks that most of them are fizzled, and garbage collection is the most time-consuming part. Sparks are generated so fast and so many and each spark occupy quite a big heap storage. However the heap is not large enough to hold them all. The run-time is even worse than non-parallel version.

Table 1: simpleRpar: 1,000 iter, 500 particles

core #	1	2	3	4
Time (sec)	13.060	16.530	18.210	19.250
Converted	0	15258	25426	28312
Overflowed	0	0	0	0
Dud	0	0	0	0
GC'd	326198	273857	255714	249704
Fizzled	2515	58967	71340	78090

3.2 parListChunk

```haskell
bhstepParListChunk :: Vec -> Vec -> Double -> Double -> [Particle] -> [Particle]
bhstepParListChunk tl br g dt particles = particles'
  where
    tree = calcSquard $ fromList particles tl br
    particles' = map (updateParticle tree g dt) particles `using` parListChunk 20 rdeepseq
```

With `parListChunk`, `bhStep` will split particle lists into chunk of a fixed size, then run the calculation parallel. The performance is improved dramatically with parallelism increasing. There're still some fizzled sparks, which indicates some sparks are not evaluated before being referred.

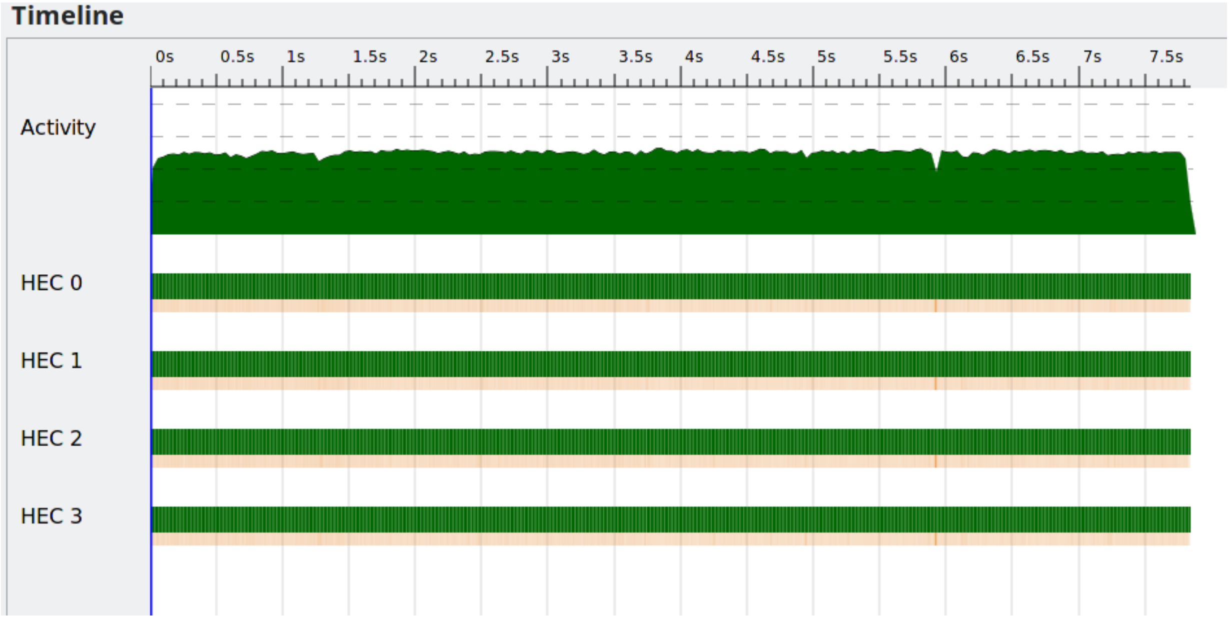

Figure 1: parListChunk threadscope: 1,000 iter, 500 particles.

Figure 2: parListChunk threadscope: 1,000 iter, 500 particles, zoom in.

Table 2: parListChunk: 1,000 iter, 500 particles

core #	1	2	3	4
Time (sec)	23.511	12.971	10.520	7.751
Converted	0	22969	22979	22976
Overflowed	0	0	0	0
Dud	0	0	0	0
GC'd	1	0	0	0
Fizzled	22999	31	21	24

As threadscope shows, parChunkList has a rather promising parallel performance, both GC and sparks running distributed balancely between CPUs.

Zoom out the timeline, besides the GC time, there are still some sequential work during iterations. It might because the tree building process. This part has not been paralleled.

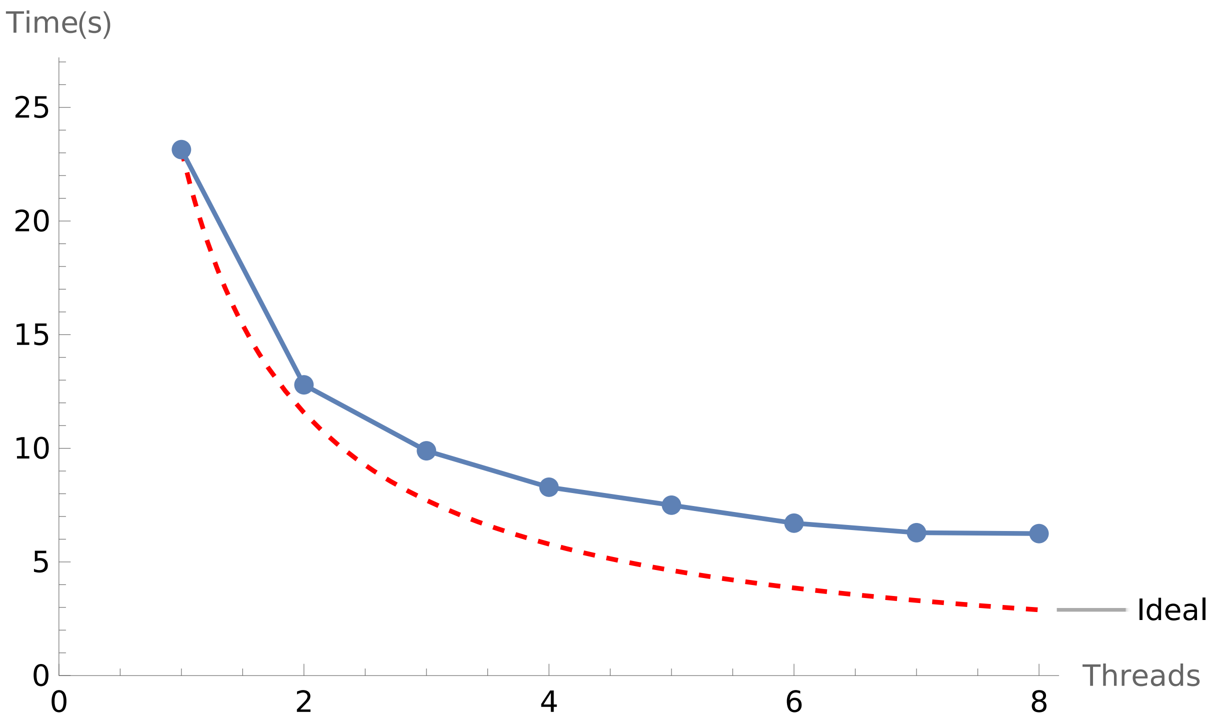

Figure 3: parListChunk threadscope: 1,000 iter, 500 particles, performance.

A Code Listing

Github repo: `https://github.com/adobemomo/ptcls-barnes-hut-haskell`

```
module DataStructs where

import Control.DeepSeq

--------------------------------------------------------------------------------
----------Vec-------------------------------------------------------------------
--------------------------------------------------------------------------------

data Vec = Vec {x :: Double, y :: Double} deriving (Eq)

instance Show Vec where
  show (Vec x y) = "[" ++ show x ++ "," ++ show y ++ "]"

zeroVec = Vec 0 0

dist :: Vec -> Vec -> Double
dist (Vec x1 y1) (Vec x2 y2) = sqrt $ (x1 - x2) ^ 2 + (y1 - y2) ^ 2

distX :: Vec -> Vec -> Double
distX (Vec x1 y1) (Vec x2 y2) = x1 - x2

scalar :: Vec -> Double
```

```haskell
scalar v = dist v zeroVec

instance Num Vec where
  (+) (Vec a b) (Vec c d) = Vec (a + c) (b + d)
  (-) (Vec a b) (Vec c d) = Vec (a - c) (b - d)
  (*) (Vec a b) (Vec c d) = Vec (a * c) (b * d)
  abs (Vec a b) = Vec (abs a) (abs b)
  signum (Vec a b) = Vec (signum a) (signum b)
  fromInteger a = Vec (fromInteger a :: Double) (fromInteger a :: Double)

instance NFData Vec where
  rnf (Vec a b) = rnf a `deepseq` rnf b

(*.) :: Vec -> Double -> Vec
(Vec x y) *. n = Vec (x * n) (y * n)

(/.) :: Vec -> Double -> Vec
(Vec x y) /. n = Vec (x / n) (y / n)

--------------------------------------------------------------------------------
----------Particle--------------------------------------------------------------
--------------------------------------------------------------------------------

data Particle = Particle {coord :: Vec, v :: Vec, m :: Double} deriving (Eq)

instance Show Particle where
  show (Particle c v m) = "[P] pos=" ++ show c ++ ", v=" ++ show v ++ ", m=" ++ show m

instance NFData Particle where
  rnf (Particle c v m) = rnf c `deepseq` rnf v `deepseq` rnf m

defaultParticle = Particle zeroVec zeroVec 1

--------------------------------------------------------------------------------
----------Squard----------------------------------------------------------------
--------------------------------------------------------------------------------

data Squard = Squard {center :: Vec, topleft :: Vec, bottomright :: Vec, mass :: Double} deriving (Eq)

instance Show Squard where
  show (Squard c tl br m) = "[S] center=(" ++ show (x c) ++ "," ++ show (y c) ++ "), size=" ++ show (abs (

instance NFData Squard where
  rnf (Squard c tl br m) = rnf c `deepseq` rnf tl `deepseq` rnf br `deepseq` rnf m

--------------------------------------------------------------------------------
----------Tree------------------------------------------------------------------
--------------------------------------------------------------------------------
```

```haskell
data Tree
  = Tree {subtree1 :: Tree, subtree2 :: Tree, subtree3 :: Tree, subtree4 :: Tree, squard :: Squard}
  | Leaf {particle :: Maybe Particle, squard :: Squard}
  deriving (Eq)

instance NFData Tree where
  rnf (Tree a b c d e) = rnf a `deepseq` rnf b `deepseq` rnf c `deepseq` rnf d `deepseq` rnf e
  rnf (Leaf (Just a) b) = rnf a `deepseq` rnf b
  rnf (Leaf Nothing b) = rnf b

mapTree :: (Tree -> a) -> Tree -> [a]
mapTree f (Tree a b c d _) = [f a, f b, f c, f d]
mapTree f leaf@Leaf {} = [f leaf]

foldTree :: (Tree -> a -> a) -> a -> Tree -> a
foldTree f acc (Tree t1 t2 t3 t4 _) = foldTree f (foldTree f (foldTree f (foldTree f acc t1) t2) t3) t4
foldTree f acc leaf@Leaf {} = f leaf acc

printTree :: Tree -> Int -> String
printTree tree@(Tree _ _ _ _ s) level = concat $ sq : branches
  where
    sq = (if level /= 0 then "\\_" else "") ++ show s
    branches = mapTree (\t -> "\n" ++ replicate level '-' ++ printTree t (level + 1)) tree
printTree leaf@Leaf {} _ = "\\_" ++ show leaf

instance Show Tree where
  show tree@(Tree {}) = printTree tree 0
  show (Leaf p s) = show p ++ " " ++ show s

getMass :: Tree -> Double
getMass (Tree _ _ _ _ s) = mass s
getMass (Leaf (Just p) _) = m p
getMass (Leaf Nothing _) = 0

getCenter :: Tree -> Vec
getCenter (Tree _ _ _ _ s) = center s
getCenter (Leaf (Just p) _) = coord p
getCenter (Leaf Nothing _) = zeroVec

getCenterX :: Tree -> Double
getCenterX tree = x $ getCenter tree

getCenterY :: Tree -> Double
getCenterY tree = y $ getCenter tree

emptyLeaf :: Vec -> Vec -> Tree
emptyLeaf topleft bottomright = Leaf {particle = Nothing, squard = Squard {center = Vec 0 0, topleft = top

emptyTree :: Vec -> Vec -> Tree
```

```haskell
emptyTree topleft bottomright =
  Tree
    (emptyLeaf topleft (Vec xmid ymid))
    (emptyLeaf (Vec xmid ymid) bottomright)
    (emptyLeaf (Vec xmid ymin) (Vec xmax ymid))
    (emptyLeaf (Vec xmin ymid) (Vec xmid ymax))
    (Squard {center = Vec xmid ymid, topleft = topleft, bottomright = bottomright, mass = 0})
  where
    xmin = x topleft
    xmax = x bottomright
    ymin = y topleft
    ymax = y bottomright
    xmid = (xmin + xmax) / 2
    ymid = (ymin + ymax) / 2
```

```haskell
module BarnesHut where

import Control.Parallel.Strategies (parBuffer, parListChunk, rdeepseq, rpar, rparWith, rseq, runEval, usin
import DataStructs

thetaThreshold :: Double
thetaThreshold = 1

--------------------------------------------------------------------------------
----------BH Algo---------------------------------------------------------------
--------------------------------------------------------------------------------
fromList :: [Particle] -> Vec -> Vec -> Tree
fromList [] tl br = emptyTree tl br
fromList (p : ps) tl br = foldl (flip addParticle) (fromList ps tl br) [p]

toList :: Tree -> [Particle]
toList (Leaf Nothing _) = []
toList (Leaf (Just p) _) = [p]
toList (Tree t1 t2 t3 t4 _) = toList t1 ++ toList t2 ++ toList t3 ++ toList t4

toListPar :: Tree -> [Particle]
toListPar (Leaf Nothing _) = []
toListPar (Leaf (Just p) _) = [p]
toListPar (Tree t1 t2 t3 t4 _) = runEval $ do
  t1' <- rpar $ toListPar t1
  t2' <- rpar $ toListPar t2
  t3' <- rpar $ toListPar t3
  t4' <- rpar $ toListPar t4
  rseq t1'
  rseq t2'
  rseq t3'
  rseq t4'
  return $ t1' ++ t2' ++ t3' ++ t4'
```

```haskell

-- is particle in squard
isInSquard :: Particle -> Squard -> Bool
isInSquard (Particle (Vec x y) _ _) (Squard (Vec cx cy) (Vec tlx tly) (Vec brx bry) _) =
  x >= tlx && x <= brx && y >= tly && y <= bry

-- add particle to tree
addParticle :: Particle -> Tree -> Tree
addParticle p (Leaf Nothing s)
  | p `isInSquard` s = Leaf {particle = Just p, squard = s}
  | otherwise = Leaf {particle = Nothing, squard = s}
addParticle p (Leaf (Just p') s)
  | p `isInSquard` s = addParticle p $ addParticle p' (emptyTree (topleft s) (bottomright s))
  | otherwise = Leaf (Just p') s
addParticle p (Tree t1 t2 t3 t4 s)
  | p `isInSquard` squard t1 = Tree (addParticle p t1) t2 t3 t4 s
  | p `isInSquard` squard t2 = Tree t1 (addParticle p t2) t3 t4 s
  | p `isInSquard` squard t3 = Tree t1 t2 (addParticle p t3) t4 s
  | p `isInSquard` squard t4 = Tree t1 t2 t3 (addParticle p t4) s
  | otherwise = Tree t1 t2 t3 t4 s

-- compute ntForce of particle2 on particle1 F=G*m1*m2/r^2
ntForce :: Particle -> Particle -> Double -> Vec
ntForce (Particle {coord = pos1, v = _, m = m1}) (Particle {coord = pos2, v = _, m = m2}) g = Vec (const *
  where
    const = g * m1 * m2 / r
    r = dist pos1 pos2
    dx = x pos2 - x pos1
    dy = y pos2 - y pos1

-- compute acceleration of particle2 on particle1 a=F/m
acceleration :: Particle -> Particle -> Double -> Vec
acceleration p1 p2 g = ntForce p1 p2 g /. m p1

-- compute delta v of particle2 on  particle1 dv=a*dt
deltaV :: Particle -> Particle -> Double -> Double -> Vec
deltaV p1 p2 g dt = acceleration p1 p2 g *. dt

-- update velocity of particle based on tree
updateV :: Particle -> Tree -> Double -> Double -> Particle
updateV p (Leaf Nothing _) _ _ = p
updateV p1 (Leaf (Just p2) s) g dt
  | coord p1 == coord p2 = p1
  | otherwise = p1 {v = v p1 + deltaV p1 p2 g dt}
updateV p1 tree@(Tree _ _ _ _ s) g dt
  | isCongregate = (p1 {v = v p1 + deltaV p1 p2 g dt})
  | otherwise = foldTree (\t p -> updateV p t g dt) p1 tree
  where
    p2 = defaultParticle {coord = center s, m = mass s}
```

```haskell
    r = dist (coord p1) (center s)
    squardSize = distX (topleft s) (bottomright s)
    theta = abs $ squardSize / r
    isCongregate = theta < thetaThreshold

-- update position of particle
updateP :: Particle -> Double -> Particle
updateP p dt = p {coord = coord p + v p *. dt}

-- update particle based on tree
updateParticle :: Tree -> Double -> Double -> Particle -> Particle
updateParticle tree g dt p = updateP (updateV p tree g dt) dt

updateParticlePar :: Tree -> Double -> Double -> Particle -> Particle
updateParticlePar (Tree t1 t2 t3 t4 _) g dt p = runEval $ do
  p1' <- rpar $ updateP (updateV p t1 g dt) dt
  p2' <- rpar $ updateP (updateV p t2 g dt) dt
  p3' <- rpar $ updateP (updateV p t3 g dt) dt
  p4' <- rpar $ updateP (updateV p t4 g dt) dt
  rdeepseq p1'
  rdeepseq p2'
  rdeepseq p3'
  rdeepseq p4'
  return p {coord = coord p1' + coord p2' + coord p3' + coord p4' - 3 * (coord p), v = v p1' + v p2' + v p

-- calculate squard for tree
calcSquard :: Tree -> Tree
calcSquard (Leaf Nothing s) = Leaf Nothing s
calcSquard (Leaf (Just p) s) = Leaf (Just p) s {center = coord p, mass = m p}
calcSquard tree@(Tree t1 t2 t3 t4 s) = Tree t1' t2' t3' t4' s'
  where
    subtrees@[t1', t2', t3', t4'] = mapTree calcSquard tree
    s' = s {center = Vec newX newY, mass = totalMass}
    totalMass = foldr (\t acc -> acc + getMass t) 0 subtrees
    newX = foldr (\t acc -> acc + getCenterX t * getMass t) 0 subtrees / totalMass
    newY = foldr (\t acc -> acc + getCenterY t * getMass t) 0 subtrees / totalMass

calcSquardPar :: Tree -> Tree
calcSquardPar (Leaf Nothing s) = Leaf Nothing s
calcSquardPar (Leaf (Just p) s) = Leaf (Just p) s {center = coord p, mass = m p}
calcSquardPar tree@(Tree t1 t2 t3 t4 s) = runEval $ do
  t1' <- rpar $ calcSquardPar t1
  t2' <- rpar $ calcSquardPar t2
  t3' <- rpar $ calcSquardPar t3
  t4' <- rpar $ calcSquardPar t4
  rdeepseq t1'
  rdeepseq t2'
  rdeepseq t3'
  rdeepseq t4'
```

```haskell
  totalMass <- rpar $ foldr (\t acc -> acc + getMass t) 0 [t1', t2', t3', t4']
  rdeepseq totalMass
  newX <- rpar $ foldr (\t acc -> acc + getCenterX t * getMass t) 0 [t1', t2', t3', t4'] / totalMass
  newY <- rpar $ foldr (\t acc -> acc + getCenterY t * getMass t) 0 [t1', t2', t3', t4'] / totalMass
  rdeepseq newX
  rdeepseq newY
  return $ Tree t1' t2' t3' t4' s {center = Vec newX newY, mass = totalMass}

--------------------------------------------------------------------------------
----------BH Algo---------------------------------------------------------------
--------------------------------------------------------------------------------

-- bh algorithm stimulation
bhstep :: Vec -> Vec -> Double -> Double -> [Particle] -> [Particle]
bhstep tl br g dt particles = particles'
  where
    tree = calcSquard $ fromList particles tl br
    particles' = map (updateParticle tree g dt) particles

bhstepRpar :: Vec -> Vec -> Double -> Double -> [Particle] -> [Particle]
bhstepRpar tl br g dt particles = particles'
  where
    tree = calcSquardPar $ fromList particles tl br
    particles' = map (updateParticlePar tree g dt) particles

bhstepParListChunk :: Vec -> Vec -> Double -> Double -> [Particle] -> [Particle]
bhstepParListChunk tl br g dt particles = particles'
  where
    tree = calcSquard $ fromList particles tl br
    particles' = map (updateParticle tree g dt) particles `using` parListChunk 20 rdeepseq
```

```haskell
module Main where

import BarnesHut
import DataStructs

p :: Particle
p = defaultParticle

ps :: [Particle]
ps = [p {coord = Vec x' y', v = zeroVec, m = 1000000000} | x' <- [-10 .. 10], y' <- [-10 .. 10]]

tl :: Vec
tl = Vec (-10) (-10)

br :: Vec
br = Vec 10 10
```

```haskell

g :: Double
g = 6.67e-11

dt :: Double
dt = 0.01

main :: IO ()
main = do
  loop 0 ps
  where
    loop :: Int -> [Particle] -> IO ()
    loop 1000 particles = do
      print particles
      return ()
    loop n particles = do
      loop (n + 1) $ bhstepParListChunk tl br g dt particles
      -- loop (n + 1) $ bhstepRpar tl br g dt particles
      return ()
```