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 $\Delta t$.

$$F = G\frac{Mm}{r^3} \quad (1)$$

$$F = ma \quad (2)$$

$$\Delta v = a\Delta t \quad (3)$$

1.2 Direct Solution

for every particle p1 in list do

for do

end

end
Algorithm 1 Direct Solution

\[
\text{\textit{list}} \leftarrow \text{particles} \\
\Delta t \leftarrow \text{time span of stimulation} \\
\text{for every particle } p1 \text{ in list do} \\
\quad \text{coordinate} \leftarrow \text{current position of } p1 \\
\quad \text{for every other particle } p2 \text{ in list do} \\
\quad \quad F \leftarrow \text{gravity force of } p2 \text{ over } p1 \\
\quad \quad a \leftarrow F / \text{mass of } p1 \\
\quad \quad v \leftarrow \text{velocity of } p1 + a \times \Delta t \\
\quad \quad \text{coordinate} \leftarrow \text{coordinate} + v \times \Delta t \\
\quad \text{end for} \\
\quad \text{update old coordinate with new value} \\
\text{end for}
\]

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 \(O(N\log N)\).

Algorithm 2 Barnes-Hut

\[
\text{Build quad tree from particle list} \\
\text{for every particle } p \text{ in list do} \\
\quad F \leftarrow \text{calculate tree force over particle} \\
\quad \text{update velocity of particle} \\
\quad \text{update position of particle} \\
\text{end for} \\
\text{Update entire particle list}
\]

2 Implementation

2.1 Data Structure

1. \textit{Vec} is a two mension vector in space, which is used to indicate position.

2. \textit{Particle} is the object that this project want to stimulate. It has attributes of coordinate, velocity and mass.

3. \textit{Squad} is a square region in space that contains several particles inside. \textit{center} is the mass center of total particles and \textit{mass} is the sum mass. \textit{topleft} and \textit{bottomright} shows the boundary of each \textit{Squad}.  

3
4. Tree can either be an 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.

```
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.

```
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
```
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 (\acc -> acc + getCenterX t * getMass t) 0 subtrees / totalMass
  newY = foldr (\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.

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

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

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

```haskell
1 calcSquardPar :: Tree -> Tree
2 calcSquardPar (Leaf Nothing s) = Leaf Nothing s
3 calcSquardPar (Leaf (Just p) s) = Leaf (Just p) s {center = coord p, mass = m p}
4 calcSquardPar tree@(Tree t1 t2 t3 t4 s) = runEval $ do
5   t1' <- rpar $ calcSquardPar t1
6   t2' <- rpar $ calcSquardPar t2
7   t3' <- rpar $ calcSquardPar t3
8   t4' <- rpar $ calcSquardPar t4
9   rdeepseq t1'
10  rdeepseq t2'
11  rdeepseq t3'
12  rdeepseq t4'
13
14   totalMass <- rpar $ foldr (\t acc -> acc + getMass t) 0 [t1', t2', t3', t4']
15   rdeepseq totalMass
16
17   newX <- rpar $ foldr (\t acc -> acc + getCenterX t * getMass t) 0 [t1', t2', t3', t4'] / totalMass
18   newX <- rpar $ foldr (\t acc -> acc + getCenterY t * getMass t) 0 [t1', t2', t3', t4'] / totalMass
19   rdeepseq newX
20   rdeepseq newY
21
22   return $ Tree t1' t2' t3' t4' s {center = Vec newX newY, mass = totalMass}
```
Both addParticle and calcSquad 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>
<thead>
<tr>
<th>Table 1: simpleRpar: 1,000 iter, 500 particles</th>
</tr>
</thead>
<tbody>
<tr>
<td>core #</td>
</tr>
<tr>
<td>Time (sec)</td>
</tr>
<tr>
<td>Converted</td>
</tr>
<tr>
<td>Overflows</td>
</tr>
<tr>
<td>GC’d</td>
</tr>
<tr>
<td>Fizzled</td>
</tr>
</tbody>
</table>

3.2 parListChunk

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.
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.
A Code Listing

Github repo: [https://github.com/adobemomo/ptcls-barnes-hut-haskell](https://github.com/adobemomo/ptcls-barnes-hut-haskell)

```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
```

Figure 3: parListChunk threadscope: 1,000 iter, 500 particles, performance.
scalar \( v = \text{dist} \ v \ \text{zeroVec} \)

```haskell
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)
```

```haskell
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---**

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

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

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

```haskell
defaultParticle = Particle zeroVec zeroVec 1
```

---

**---Squard---**

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

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

```haskell
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 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

g centerX :: Tree -> Double
g centerX tree = x $ getCenter tree

g centerY :: Tree -> Double
g centerY tree = y $ getCenter tree

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

emptyTree :: Vec -> Vec -> Tree
emptyTree topleft bottomright =
  Tree
  (emptyLeaf topleft (Vec xmid ymid))
  (emptyLeaf (Vec xmid ymid) bottomright)
  (emptyLeaf (Vec xmid ymid) (Vec xmax ymid))
  (emptyLeaf (Vec xmid ymid) (Vec xmax 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

module BarnesHut where

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

thetaThreshold :: Double
thetaThreshold = 1

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

fromList :: [Particle] -> Vec -> Vec -> Tree
fromList [] t1 br = emptyTree t1 br
fromList (p : ps) t1 br = foldl (flip addParticle) (fromList ps t1 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'
```
-- 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
defines
  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}
```
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 p3' + v p4'}

-- 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'
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 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

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
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) $ bhstepParPar tl br g dt particles
    return ()