COMSW4995 Parallel Functional Programming
Proposal
Galaxy Simulator
Minhe Zhang (mz2864)
Instructor: Stephen A. Edwards

1 Introduction

Galaxy Simulator (GS) is a Haskell program which simulates celestial movement and visualizes celestial bodies using Gloss. The visualization of the galaxy should be dynamic which represent the the whole program assuming the universe is a 2-D plane.

2 Model

2.1 Simplification
1. GS assumes an isolated system which is not affected by any other system.
2. Instead of 3-D which is the real world situation, GS simulates 2-D world.
3. There are only two kinds of celestial body: star and planet.
4. All celestial bodies are considered as mass points. GS doesn’t worry about collision between celestial bodies.

2.2 Celestial Body
Celestial body defined as algebraic data type Planet in Haskell has the following properties:
1. Coordinate: float and float
2. Mass: float
3. Velocity: [float]
2.3 Gravity

The equation of gravity is

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

The sign of force \( F_x \) and \( F_y \) should be the same as \( x_2 - x_1 \) and \( y_2 - y_1 \).

2.4 Acceleration

We use Newton’s second law to calculate acceleration:

\[ F = ma \]

Then we can have acceleration in different dimension:

\[ a_x = \frac{F_x}{m} \]
\[ a_y = \frac{F_y}{m} \]

Because acceleration is a vector, we define that \( a_x \) has the same sign as \( F_x \).

2.5 Velocity

Let \( \Delta t \) denote the smallest time interval defined by user or default. The velocity of body in galaxy should change as

\[ v'_x = v_x + a_x \Delta t \]
\[ v'_y = v_y + a_y \Delta t \]

2.6 In Haskell

In the source code, celestial body is define as algebraic data type Planet in the Planet.hs module.

3 Algorithm

The algorithm of GS is pretty straight forward. Let \( s_i \) denote the \( i \)-th state of the system. \( s_i \) can be determined if we know \( s_{i-1} \). To identify different \( s_i \), we only need status of all bodies.

GS computes \( m \) states of \( n \) celestial bodies. Let \( p_{ij} \) denote the \( j \)-th body in \( i \)-th state. \( p_{ij} \) is determined by \( p_{(i-1)k}, k \in \{1, \ldots , n\} \). After compute the compound force from other bodies, GS accelerates \( p_{ij} \) and make an approximate move. After all \( p_{ik}, k \in \{1, \ldots , n\} \) are computed, we say GS finished computation of \( s_i \). The time complexity of computing each state is \( O(n^2) \). The time complexity of the whole program is \( O(mn^2) \).

GS generate \( s_0 \) randomly and starts the simulation. When the \( s_m \) is computed, GS terminates.
Algorithm 1 GS
\[
\begin{align*}
n, m, \text{interval} & \leftarrow \text{input} \\
s_0 & \leftarrow \text{randomly generate } n \text{ bodies} \\
\text{for } i \text{ in } [1, m] & \text{ do} \\
\quad s_i & \leftarrow \text{move}(s_{i-1}, \text{interval}) \\
\text{end for} \\
\text{return } s_m
\end{align*}
\]

Algorithm 2 move
\[
\begin{align*}
[p_{i1}, p_{i2}, \ldots, p_{in}], \text{interval} & \leftarrow \text{input} \\
\text{for } j \text{ in } [1, n] & \text{ do} \\
\quad \text{force} & \leftarrow [0, 0] \\
\quad \text{for } k \text{ in } [1, n] & \text{ do} \\
\quad\quad \text{if } & \text{ then } j \neq k \\
\quad\quad\quad \text{force} & \leftarrow \text{force} + \text{Gravity}(p_{ij}, p_{ik}) \\
\quad\text{end if} \\
\quad \text{end for} \\
\quad p_{(i+1)j} & \leftarrow \text{Adjust}(p_{ij}, \text{force}, \text{interval}) \\
\text{end for} \\
\text{return } [p_{(i+1)1}, p_{(i+1)2}, \ldots, p_{(i+1)n}]
\end{align*}
\]

4 Strategy and Performance

There are several strategies were evaluated and some of which the performance is very inefficient (static partition, for example). In this report, there are three strategies worth mentioning. parList, parListChunk, and parListChunk with depth limitation. Also, in this chapter, how \(m\) and \(n\) changed performance will be discussed.

4.1 parList

parList is the first reasonable efficient strategy used in this project. It has the finest granularity to evaluate \(\text{Planet}\). It is applied to the core function move. It generates large amount of sparks. Most of these sparks are overflowed. When set \(n = 50, m = 10000\), it has a poor performance.

4.1.1 Speedup

This is the speedup figure of parList strategy with finest granularity.
The speedup is less than 1.25.

### 4.1.2 Spark Statistic

This is the spark statistic of parList strategy with finest granularity. It is also stored in `parList-50-10000.csv`.

<table>
<thead>
<tr>
<th>thread</th>
<th>total time</th>
<th>total elapsed</th>
<th>total spark</th>
<th>converted</th>
<th>overflowed</th>
<th>dud</th>
<th>GC</th>
<th>fizzled</th>
</tr>
</thead>
<tbody>
<tr>
<td>-N1</td>
<td>4.290</td>
<td>4.811</td>
<td>500100</td>
<td>0</td>
<td>152703</td>
<td>0</td>
<td>58</td>
<td>3275</td>
</tr>
<tr>
<td>-N2</td>
<td>11.961</td>
<td>6.627</td>
<td>500100</td>
<td>3431</td>
<td>152535</td>
<td>0</td>
<td>0</td>
<td>70</td>
</tr>
<tr>
<td>-N3</td>
<td>12.664</td>
<td>5.014</td>
<td>500100</td>
<td>3026</td>
<td>152949</td>
<td>0</td>
<td>0</td>
<td>61</td>
</tr>
<tr>
<td>-N4</td>
<td>12.953</td>
<td>3.988</td>
<td>500100</td>
<td>3040</td>
<td>152935</td>
<td>0</td>
<td>0</td>
<td>61</td>
</tr>
<tr>
<td>-N5</td>
<td>15.588</td>
<td>4.087</td>
<td>500100</td>
<td>3111</td>
<td>152864</td>
<td>0</td>
<td>0</td>
<td>61</td>
</tr>
<tr>
<td>-N6</td>
<td>18.152</td>
<td>4.160</td>
<td>500100</td>
<td>3169</td>
<td>169189</td>
<td>0</td>
<td>0</td>
<td>62</td>
</tr>
<tr>
<td>-N7</td>
<td>19.801</td>
<td>3.954</td>
<td>500100</td>
<td>4559</td>
<td>167772</td>
<td>0</td>
<td>0</td>
<td>89</td>
</tr>
<tr>
<td>-N8</td>
<td>21.338</td>
<td>3.933</td>
<td>500100</td>
<td>7707</td>
<td>172754</td>
<td>0</td>
<td>0</td>
<td>151</td>
</tr>
</tbody>
</table>
4.2 parListChunk

Rather than sparking every element in list, parListChunk sparks chunks of elements in list. The performance of parListChunk is not only related to the input, but also related to the size of chunk. parListChunk is so far the best strategy in this project.

4.2.1 Speedup

This is the speedup figure of parListChunk strategy with coarse granularity.

![Speedup of chunk(5) (n=50, m=10000)](image)

The speedup starts to exceed 1.5. This is a better result than parList but still far from theoretical limitation.

4.2.2 Spark Statistic

This is the spark statistic of parListChunk strategy with coarse granularity. It is also stored in chunk-50-10000-size5.csv
4.3 Depth Limitation

After a few attempts, we can conclude that depth has nothing to do with speedup and running time.

4.3.1 Speedup

This is the speedup figure of speed up when we simulate 50 bodies in 10000 steps. The strategy it uses is parListChunk and the size of chunk is 5. The x axis is the number of threads and the depth is set to 1000.

4.4 How m and n changes performance

The speedup relies on the input parameters heavily. These are two diagram from threadscope.
4.4.1 n=50, m=10000

At the first glance, the first diagram is better than the latter one. However, the speedup of the first one is 1.5 the second one is 2.5. This is the detail after zooming in.

4.4.2 n=1500, m=10
50 bodies and 10000 steps:

1500 bodies and 10 steps:

We can tell from the figures above that when the number of body is larger, the better usage will be achieved.
Also, we can take a look at what happened when serial waiting: 50 bodies and 10000 steps:

1500 bodies and 10 steps:

Although it costs more at a single GC or waiting if we set $n$ large, it’s less frequent. The parallel proportion is larger when we set $n$ larger. This is the 2.5 speed up when we set $n = 1500$ and $m = 10$: 
5 Visualization

The result of visualization should be dynamic. After a certain time interval, the graph of current state of galaxy should update. It should show every existing celestial body as dot.

6 What’s Next

1. Barnes-Hut algorithm. This will improve the time complexity from $n^2$ to $n\log(n)$.
2. Interactive Simulation.
7 Reference


8 Source File

{-
  This is the entry file which starts the whole program.
-}

import Control.Parallel(par, pseq)
-- import Control.DeepSeq(deepseq)
import System.Environment
import System.Random
import Control.Parallel.Strategies
import qualified Planet as P
import qualified Laws as L
import qualified Visualize as V

chunkSize :: Int
chunkSize = 5

depth :: Int
depth = 2000

getSeeds :: Int -> IO [Float]
getSeeds n = sequence $ replicate n $ randomRIO (0,1::Float)

genPlanets :: [Float] -> [Float] -> [P.Planet]
genPlanets [] [] = []
genPlanets [s1] [s2] = [P.genPlanet s1 s2]
genPlanets l1 l2 = (P.genPlanet h1 h2) : genPlanets t1 t2
  where h1 = head l1
        h2 = head l2
\[ t_1 = \text{tail } l_1 \]
\[ t_2 = \text{tail } l_2 \]

-- Trivial approach, doesn't work.
\[
\text{trivial} :: \text{Int} \to \text{Float} \to [\text{P.Planet}] \to [\text{P.Planet}]
\]
\[
\text{trivial } 0 \_ \ _ = \_ \\
\text{trivial } 1 \_ \ _ = \text{[L.move p ps i| p <- ps]} \\
\text{trivial } s \_ \ _ = \text{trivial } (s - 1) \_ \ _ \_ \\
\text{where } \_ \_ = \text{[L.move p ps i| p <- ps]}
\]

-- Static partitioning
\[
\text{staticPart} :: \text{Int} \to \text{Float} \to [\text{P.Planet}] \to [\text{P.Planet}]
\]
\[
\text{staticPart } 0 \_ \ _ = \_ \\
\text{staticPart } 1 \_ \ _ = \text{[L.move p ps i| p <- ps]} \\
\text{staticPart } s \_ \ _ = \text{staticPart } (s - 1) \_ \ _ \_ \\
\text{where } \_ \_ = \text{[L.move p ps i| p <- ps]} \\
\]
\[
\text{ps}' = \text{[L.move p ps i| p <- ps]} \\
\text{(ps1, ps2)} = \text{splitAt } (\text{length } ps \div 2) \_ \_ \\
\]

-- Finest granularity
\[
\text{finePart} :: \text{Int} \to \text{Float} \to [\text{P.Planet}] \to [\text{P.Planet}]
\]
\[
\text{finePart } 0 \_ \ _ = \_ \\
\text{finePart } 1 \_ \ _ = \text{[L.move p ps i| p <- ps] `using` parList rseq} \\
\text{finePart } s \_ \ _ = \text{finePart } (s - 1) \_ \ _ \_ \\
\text{where } \_ \_ = \text{[L.move p ps i| p <- ps] `using` parList rseq}
\]

-- parListChunk
\[
\text{chunkPart} :: \text{Int} \to \text{Float} \to [\text{P.Planet}] \to [\text{P.Planet}]
\]
\[
\text{chunkPart } 0 \_ \ _ = \_ \\
\text{chunkPart } 1 \_ \ _ = \text{[L.move p ps i| p <- ps] `using` parListChunk} \\
\text{chunkPart } s \_ \ _ = \text{chunkPart } (s - 1) \_ \ _ \_ \\
\text{where } \_ \_ = \text{[L.move p ps i| p <- ps] `using` parListChunk}
\]

-- depth limited
\[
\text{depthPart} :: \text{Int} \to \text{Int} \to \text{Float} \to [\text{P.Planet}] \to [\text{P.Planet}]
\]
\[
\text{depthPart } 0 \_ \ _ = \_ \\
\text{depthPart } s \_ \ _ = \text{chunkPart } s \_ \ _ \_ \\
-- depthPart } s 0 \_ \ _ = \text{trivial } s \_ \ _
\]
depthPart s d i ps = depthPart (s - 1) (d - 1) i ps'
where ps' = [L.move p ps i | p <- ps] `using` parListChunk chunkSize rdeepseq

forceEval :: [P.Planet] -> IO ()
forceEval ps = do
  print $ length $ filter ((==) (P.Planet [0, 0] 0 0 0)) ps

main :: IO ()
main = do
  args <- getArgs
  let [n, s, i, mode] = args
  let planetNum = read n :: Int
  let steps = read s :: Int
  let interval = read i :: Float
  seedList1 <- getSeeds planetNum
  seedList2 <- getSeeds planetNum
  let planets = (genPlanets (seedList1) (seedList2)) `using` parList rseq
  case mode of
    "v" -> do
      let star = P.Planet [0, 0] (1e5 * 7) 0 0
      V.runSimulation (star : planets)
    "trivial" -> do
      let state = (trivial steps interval planets)
      forceEval state
    "static" -> do
      let state = (staticPart steps interval planets)
      forceEval state
    "parList" -> do
      let state = (finePart steps interval planets) `using` parList rseq
      forceEval state
    "chunk" -> do
      let state = (chunkPart steps interval planets) `using` parList rseq
      forceEval state
    "depth" -> do
      let state = (depthPart steps depth interval planets)
      forceEval state
_ -> do
print $ "Usage: ./Galaxy <Number of Bodies> <Number of Steps> <Time Inteval> <mode> +RTS -N<Number of Threads> -s"

{-
   This module defines the basic laws of physics.
-}

module Laws (  
    move,  
    nextState  
) where

import Planet
import Control.Parallel.Strategies
import Control.Parallel(par, pseq)

type Force = Float

type Acc = Float

type Vel = Float

type Time = Float

{-
   Gravitational constant.
-}
g :: Float
g = 10

{-
   This function takes two Planets as input, computes the gravity from p1 to p2, and returns a list of float value which represents Fx and Fy. Gravity formula F = g * m1 * m2 / r^2 
-}
gallery :: Planet -> Planet -> [Force]
gallery p1 p2 = [if x2 > x1 then fx else -fx, if y2 > y1 then fy else -fy]
  where  
    fx = if x1 == x2 then 0  
      else g * m1 * m2 / rSqr  
    fy = if y1 == y2 then 0  
      else g * m1 * m2 / rSqr  
    x1 = posX p1  
    x2 = posX p2  
    y1 = posY p1  
    y2 = posY p2  
    m1 = mass p1
m2 = mass p2
rSqr = (fSqr (x1 - x2)) + (fSqr (y1 - y2))

{-
Takes a planet and a pair of forces.
Returns a pair of velocities.
-}
acceleration :: Planet -> [Force] -> [Acc]
acceleration p fxy = map acc fxy
  where acc f = f / (mass p)

{-
Change velocities.
-}
accelerate :: [Vel] -> [Acc] -> Time -> [Vel]
accelerate vs as t = zipWith (\v a -> v + a * t) vs as

{-
Used for visualization.
-}
nextState :: [Planet] -> Double -> [Planet]
nextState ps _ = [move p ps (1/100) | p <- ps] `using` parList
  rseq
move :: Planet -> [Planet] -> Time -> Planet
-- move p ps t = moveHelper 100 p ps (t / 100)
move p ps t = moveHelper 1 p ps t

moveHelper :: Int -> Planet -> [Planet] -> Time -> Planet
moveHelper 0 p ps t = p
moveHelper n p ps t = (moveHelper (n - 1) p' ps t)
  where p' = Planet vs m x y
    vs = accelerate (velocity p) as t
    as = acceleration p fs
    -- fs = foldr (zipWith (+)) [0, 0] ((map
    <- (gravity p) ps) `using` parList rseq)
    fs = foldr (zipWith (+)) [0, 0] (map (gravity p)
    <- ps)
    [x, y] = zipWith (\pos v -> pos + v * t) [posX p,
    <- posY p] vs
    m = mass p

fSqr :: Float -> Float
fSqr x = x * x

{-
This module defines and exports data type Planet.

module Planet (  
    Planet(..),  
    genPlanet  
) where

import Control.DeepSeq

{-
Units:
  velocity m * s^-1
  mass  kg^-1
  posX  m
  posY  m
-
} data Planet = Planet {  
    velocity :: ![Float],  
    mass :: !Float,  
    posX :: !Float,  
    posY :: !Float  
} deriving (Show, Eq)

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

{-
Generate a planet from two seed which are randomly generated.
-
genPlanet :: Float -> Float -> Planet
genPlanet s1 s2 = Planet [vx, vy] m x y
    where vx = if s1 > 0.5 then (-absVx) else absVx
          vy = -(x * vx / y)
          m = 1e2 + d * 1e2 + s1 * 1e1
          x = -(720 / 4) + s1 * (720 / 2)
          y = -(720 / 4) + s2 * (720 / 2)
              -- x = if s1 > 0.5 then 20 + s1 * 200 else -(20 + s1 * 200)
              -- y = if s2 > 0.5 then 20 + s2 * 200 else -(20 + s2 * 200)
          absVx = 1 * 1e2 + s1 * 1e1 + (1 - d) * 1e1
          d = ((abs x) + (abs y)) / 720 / 2

{-
module Visualize (runSimulation, windowSize) where

import GHC.Float
import Graphics.Gloss
import Control.Parallel.Strategies
import qualified Planet as P
import qualified Laws as L

WindowSize :: Int
windowSize = 720

drawPlanet :: P.Planet -> Picture
drawPlanet p = Color white $ Translate x y (circleSolid
   (realToFrac $ 0.5 * log (P.mass p)))
   where x = realToFrac $ P.posX p
         y = realToFrac $ P.posY p

drawPlanets :: [P.Planet] -> [Picture]
drawPlanets ps = map drawPlanet ps

runSimulation :: [P.Planet] -> IO ()
r
runSimulation ps = simulate (InWindow "Galaxy Simulation"
   (windowSize, windowSize) (100, 100))
   black 60
   ps
   (\ps' -> pictures $ drawPlanets ps')
   (\_ dt ps' -> L.nextState ps' (float2Double
      \_ dt))