Morgan Navarro (msn2139)
COMS 4995
Topics in Computer Science - Parallel Functional Programming
Professor Edwards

Final Project Report

Team
Morgan Navarro (msn2139)

Project Idea
Physics Particle Simulation

Project Description

I have built a physics particle simulation that supports parallel rendering by using parallel programming in computer graphics. My physics particle simulator shoots particles/spheres up into the air and will be shaded and rendered onto the program screen. The simulator will allow the user to alter the number of particles and the runtime of the program. The base run of the program will fire the 3 particles at a time until the HSV value of the particles is equal to 40 (a basic way to make the program execute the same number of tasks). The particles will fade, change colors, and simulate a physics environment with gravity. It fires particles from the bottom of the screen to the left and right at random angles with random speeds.

In regards to the specific algorithms being used, it is just a simple N-Body simulation that uses particles with properties of color (hue, saturation, and value), size, position (x, y, z), velocity (x, y, z), and time to live. Most of these values are set randomly when the particle is created but then are updated as time goes on, creating the movement of the particles and the simulation of gravity. By using parallel programming within the updates of the particles (and other methods), I was able to make the program run faster and have a more balanced workload.

Project Dependencies

My project uses a few different graphics libraries such as OpenGL, GLUT, GLUtil, and Data.Colour. Most of these libraries come standard with the Haskell Platform, but for me I had to manually install each one using:

“cabal install OpenGL”
“cabal install GLUT”
“cabal install GLUtil”
“cabal install colour”

Project Methods

The particle simulation program is a simple step-wise N-Body simulation that uses particles with properties of color (hue, saturation, and value), size, position (x, y, z), velocity (x, y, z), and time to live. I created a data type “Particle” that has fields for the HSV values (hue, saturation, and value), the size of the particle, the location of the particle (x, y, z coordinates), the velocity of the particle (x, y, z velocity), and the particle’s time to live. Then I used an array/list to store the particles and ultimately display them on the screen. The value for gravity is set to 0.0015 in order to keep the particles from flying off the screen and the screen size was set to 900 by 900. A few of the methods in the program are standard methods that are used within any OpenGL program that deals with screen displays, textures, and keyboard input.

Main Method: The main method handles the program arguments, creates the simulation window, handles keyboard/mouse input, and loads the correct picture file depending on whether the user runs the program with “particle” or “stephen” as a command line argument. If the user runs the program with “particle” as the picture name, then a simple sphere image is used for each particle; however, if the user runs the program with “stephen” then a picture of Professor Stephen Edwards is used. The first command line argument is the picture type (“particle” or “stephen”) and the second command line argument is run time of program. This run time number is not an exact run time value, since I was not sure how to exactly do a timer in Haskell that would stop the program after a specified time value. Instead, this number represents the HSV value that should be used to stop the program, basically new particles are given slightly increasing HSV values, so once the value reaches the run time argument value then the program is stopped. Finally, the last command line argument is the number of particles that should be created at each time step. For my tests I used 3, so 3 particles are created at a time in the “idleProcess” function. The main method uses OpenGL “displayCallback”, “reshapeCallback”, and “idleCallback” in order to display the particles, reshape the window, and call upon other methods for moving the particles. Finally, the main method uses the OpenGL mainLoop function in order to continuously run the necessary functions for the particle simulation.

Reshape Method: This method is a standard method that was taken from the OpenGL examples that is used when the simulation window is resized.

KeyboardMouse Method: This method is another standard method that handles keyboard input to the program. If an escape or backspace key is pressed then the program automatically exits; otherwise, the key pressed is echoed to the terminal.

Extract Method: This is another standard method that is used to extract texture info from a picture and allow it to be used with OpenGL to display an image.

DisplayParticle Method: This method does exactly what it says, it displays a particle to the simulation window by first getting the color of the particle and then creating texture coordinates based on the size and (x, y, z) coordinates of the particle.

FindColor Method: This method returns a Color3 value based on the hue, saturation, and value for an HSV color. This is used in the displayParticle method that renders the particles on the simulation screen.

NewColor Method: This method makes sure that the new color for a particle does not exceed the maximum value for colors, and if so, it subtracts the maximum color so that the color value can start again from 0.

GetTexture Method: This method is another pretty standard OpenGL method that is used to load textures from a picture file.

DisplaySimulation Method: This method takes in the array of particles and clears the buffer of colors so that only the colors of a particle at its current location are shown at any given time. It then calls upon displayParticle for each particle in the array and then reloads the screen.

IdleProcess Method: This method is the method that is constantly called by the program to update particles and create new particles. It serves as the main loop of the program and is also responsible for ending the program after a certain amount of time (color changes) has elapsed. It takes in the expected run time, the number of particles to create each time step, the list of particles, and the current color value being used and then updates the particles using the moveParticle method. It then creates new particles using the initParticles method and creates a list of the new particles, and the old particles, that is filtered so that the time to live is greater than 0 for all of the particles. Finally, it increases the color value for the next loop and checks to see if that color value is above the run time threshold. If the color value exceeds the run time value then the program exits successfully; otherwise, it prints out the current color value before moving forward with the program.

MoveParticle Method: This method contains the logic for updating each particle and changing the x, y, and z coordinates based on the current coordinates and velocities/gravity. This method also decreases the time to live for each particle and updates the saturation, color value, and particle size. The majority of the parallelization of the code is found within this method.

InitParticle Method: This method is responsible for creating new particles and it selects the hue, saturation, x velocity, y velocity, and time to live at random so that the particles fly off in different directions. It has set values for the starting x, y, and z coordinates along with the starting size, color value, and z velocity.

Project Video Demo

YouTube Link: https://youtu.be/YhpHvLsB0M0

As you can see in this video demo, the simulation on my computer has noticeable lags in rendering that occur due to my computers low performance GPU and issues with I/O Bottlenecking. The tests of my program with parallelization and a higher number of threads decreases this lag, but it still exists even when fully parallelized.

Project Issues

I ran into a lot of issues with installing some of the graphics libraries (specifically GLUtil) and installing/running threadscope. On my Windows desktop computer (Lenovo ideacentre AIO Intel(R) Core(™) i3-6100T CPU @ 3.20GHz), I was able to download GLUtil and fully run the program; however, I was not able to download Threadscope. Additionally, on my Macbook, I was not able to install GLUtil or threadscope. On my dual-booted usb version of Ubuntu Linux, I was able to download threadscope and get it to work, but I was not able to run the program since I had issues with downloading GLUtil again. This meant that I had to run the program on my Windows desktop and generate the eventlog, and then switch over to Ubuntu and load the eventlogs into threadscope. This made the process of debugging/profiling a lot more difficult. In addition, my desktop computer is a dual-core machine that is an All-In-One, therefore it does not have the best CPU or GPU that is needed for parallel graphics rendering. The Intel(R) HD Graphics 530 does not support hardware acceleration or disabling GPU Rendering and the 2 cores makes it difficult to run the program on a much larger number of threads. Without the ability to disable GPU Rendering (due to my graphics card being a low performance, basic model), I was not able to get cleaner benchmarks that did not depend on CPU/GPU bottlenecks. With more computing power, a CPU with a larger number of cores, and a more performance enhanced GPU, I would estimate much better results and greater improvements from parallelization.

Performance

In order to gather data on the performance of my CPU and GPU while running the code without parallelization compared to with parallelization on a various number of cores, I utilized the MSI Afterburner program. These were the results of CPU and GPU usage percentage for different runs of the program:

Non-Parallel:

Simulation Particle 40 3 +RTS -ls 81% GPU 100% CPU
Simulation Particle 40 3 +RTS -ls -N 80% GPU 89% CPU
Simulation Particle 40 3 +RTS -ls -N2 80% GPU 78% CPU
Simulation Particle 40 3 +RTS -ls -N4 79% GPU 78% CPU

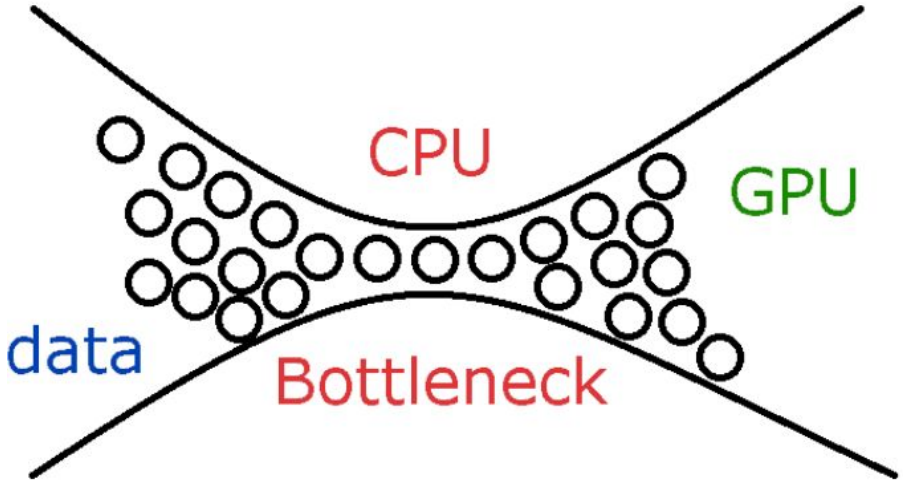
Parallel:

Simulation Particle 40 3 +RTS -ls 62% GPU 74% CPU
Simulation Particle 40 3 +RTS -ls -N 52% GPU 91% CPU
Simulation Particle 40 3 +RTS -ls -N2 53% GPU 84% CPU
Simulation Particle 40 3 +RTS -ls -N4 57% GPU 100% CPU

As you can see, the CPU usage tends to max out and be close to 90-100% while the GPU is much lower in the 50-60% range. This points to a bottleneck at the CPU level that occurs when the processor does not have the right speed to process and transfer data. Additionally, for certain runs (especially runs with a higher amount of particles being created per time step), I noticed the GPU was maxing out at 90% with the CPU usage at around 70%, this means that for a higher amount of particles generated, the program's parallelization is helping with CPU processing but there is a GPU bottleneck that is

causing the simulation to have noticeable lag and performance issues. Unfortunately, due to my low performance GPU, I was not able to disable GPU rendering or do anything that would help to mitigate the bottleneck occuring at the I/O stage. This caused issues with measuring performance in terms of run time, but I was still able to get solid measurements by just measuring each run multiple times.

Representation of Bottleneck:



Since this program does not have a specific “end” that would allow me to compare runtime, I changed the code in order to stop after a specified amount of time (based on the HSV value of the current particle being generated). Due to this, the runtime of the program does have some variation from run to run which makes performance comparison slightly difficult. A run using 2 cores will most likely have a slight time difference from another identical run using 2 cores. For this reason, I did not weight the elapsed time comparison between runs with different numbers of threads and instead I focused more on workload balance, parallel GC work balance, sparks converted, and productivity.

After testing a non-parallelized version of my code on one core against a parallelized version of the code on 2 and 4 threads, I was able to see a slight improvement in total run time and obviously an improvement in the balance of the workload. In terms of rendering causing an issue with bottlenecks, I did see some issues when I ran the program with a large number of particles being created each second; however, at a reasonable rate, there does not seem to be any serious bottleneck at the I/O or rendering levels.

Run Time Performance:

I tested the program for run time performance by running the program without parallelized code and 1 core, parallel code with 2 cores, parallel code with 4 cores, parallel code with 2 cores and specified allocation area size (A20M), and parallel code with 4 cores and specified allocation area size. I decided to try some runs with a specified allocation area size using the -A20M flag in execution. This sets the allocation area size that is used in garbage collection and can help to improve performance. In some cases, a larger allocation area can lead to negative effects in the cache, but it can also lead to less

garbage collection and overall system performance. In my case, using this flag helped to improve the runtime of the 2 core parallelized code and also significantly improved the Parallel GC Work Balance.

Non-Parallel 1 Core = 2.325s Elapsed
Parallel 2 Cores = 2.042s Elapsed
Parallel 4 Cores = 2.082s Elapsed
Parallel 2 Cores -A20M = 1.992s Elapsed
Parallel 4 Cores -A20M = 2.145s Elapsed

Based on these results, you can see that running the parallel code on 2 cores had the best results and the -A20M flag on 2 cores had even better results. There was not an extremely significant difference between results, as even the best and worst results only differed by about 0.3 seconds. But when it comes to larger scale performance, a small difference would be much more important. Additionally, because of the CPU/GPU bottlenecking and rendering issues, some of the times for each run varied and that made it more difficult to get accurate benchmarks. If I was able to test my program on a computer with a much more upgraded GPU/CPU then I would expect to see better results. Finally, I believe the program ran better on 2 cores instead of 4, because my machine is a 2 core machine so that would be the maximum number of threads that you could run the program on without causing issues.

Parallel GC Work Balance Performance:

Another interesting performance benchmark is the parallel GC work balance which basically represents the performance of the parallelization when it comes to garbage collection threads. As you can see, the parallel GC work balance was not very good for the parallelized 2 core and 4 core runs (with 2.35% and 1.7% respectively); however, after using the -A20M option and increasing the garbage collection allocation area size, the GC work balance significantly increased to 31.15% on the 2 core run and 39.83% on the 4 core run.

Sparks Converted:

The sparks converted statistic helps to see if the parallel processing was successful, because more sparks converted means a more balanced workload and better parallelization. The 2 core parallel code had the largest number of sparks converted with 557 and the 4 core parallel code was not as successful with only 242 sparks converted. With the -A20M option used, the number of sparks converted decreased for the 2 core run with 459 sparks, while the 4 core run had an increase in converted sparks with 350. Overall the 2 core run was the most successful in terms of sparks being used properly and contributing to better performance in regards to parallelization.

Simulation - Non-Parallel

```
TASKS: 3 (1 bound, 2 peak workers (2 total), using -N1)

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

INIT    time    0.000s  (  0.000s elapsed)
MUT     time    1.234s  (  2.159s elapsed)
GC      time    0.078s  (  0.164s elapsed)
EXIT    time    0.000s  (  0.000s elapsed)
Total   time    1.312s  (  2.325s elapsed)

Alloc rate    290,031,350 bytes per MUT second

Productivity  94.0% of total user, 92.9% of total elapsed
```

Simulation - Parallel 2 Cores

```
Parallel GC work balance: 2.35% (serial 0%, perfect 100%)

TASKS: 4 (1 bound, 3 peak workers (3 total), using -N2)

SPARKS: 337347(557 converted, 0 overflowed, 28789 dud, 2436 GC'd, 305565 fizzled)

INIT    time    0.000s  (  0.000s elapsed)
MUT     time    1.703s  (  1.956s elapsed)
GC      time    0.156s  (  0.083s elapsed)
EXIT    time    0.000s  (  0.002s elapsed)
Total   time    1.859s  (  2.042s elapsed)

Alloc rate    126,386,791 bytes per MUT second

Productivity  91.6% of total user, 95.8% of total elapsed
```

Simulation - Parallel 4 Cores

```
Parallel GC work balance: 1.70% (serial 0%, perfect 100%)

TASKS: 6 (1 bound, 5 peak workers (5 total), using -N4)

SPARKS: 337312(242 converted, 0 overflowed, 28770 dud, 2924 GC'd, 305376 fizzled)

INIT    time    0.000s  (  0.001s elapsed)
MUT     time    1.812s  (  1.976s elapsed)
GC      time    0.172s  (  0.105s elapsed)
EXIT    time    0.000s  (  0.000s elapsed)
Total   time    1.984s  (  2.082s elapsed)

Alloc rate    122,930,546 bytes per MUT second

Productivity  91.3% of total user, 94.9% of total elapsed
```

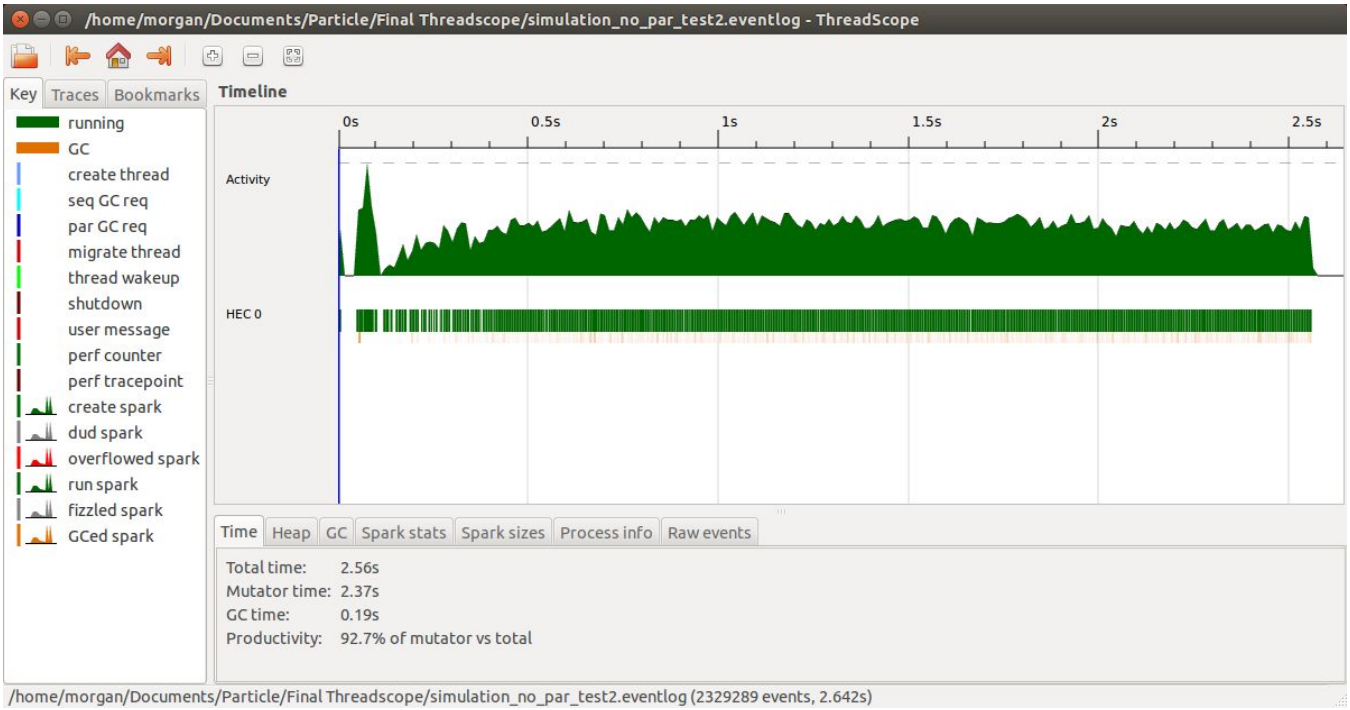
Simulation - Parallel 2 Cores -A20M Sets the minimum allocation area size

```
Parallel GC work balance: 31.15% (serial 0%, perfect 100%)

TASKS: 4 (1 bound, 3 peak workers (3 total), using -N2)

SPARKS: 337336(459 converted, 0 overflowed, 927 dud, 40 GC'd, 335910 fizzled)

INIT    time    0.000s  (  0.001s elapsed)
MUT     time    1.641s  (  1.979s elapsed)
GC      time    0.000s  (  0.012s elapsed)
EXIT    time    0.000s  (  0.000s elapsed)
Total   time    1.641s  (  1.992s elapsed)

Alloc rate    128,763,206 bytes per MUT second

Productivity 100.0% of total user, 99.4% of total elapsed
```

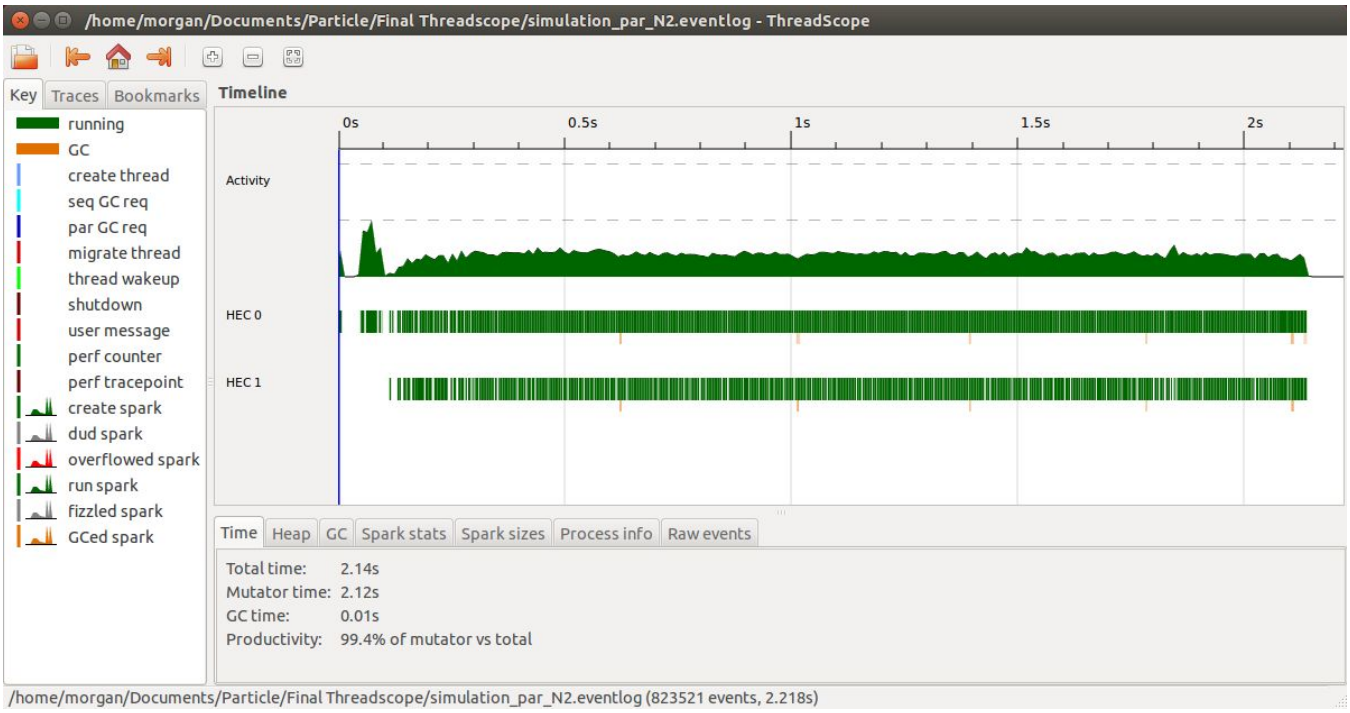
Simulation - Parallel 4 Cores -A20M

```
Parallel GC work balance: 39.83% (serial 0%, perfect 100%)

TASKS: 6 (1 bound, 5 peak workers (5 total), using -N4)

SPARKS: 337370(350 converted, 0 overflowed, 397 dud, 23 GC'd, 336600 fizzled)

INIT    time    0.000s  (  0.002s elapsed)
MUT     time    1.797s  (  2.125s elapsed)
GC      time    0.000s  (  0.016s elapsed)
EXIT    time    0.000s  (  0.002s elapsed)
Total   time    1.797s  (  2.145s elapsed)

Alloc rate    124,182,087 bytes per MUT second

Productivity 100.0% of total user, 99.1% of total elapsed
```

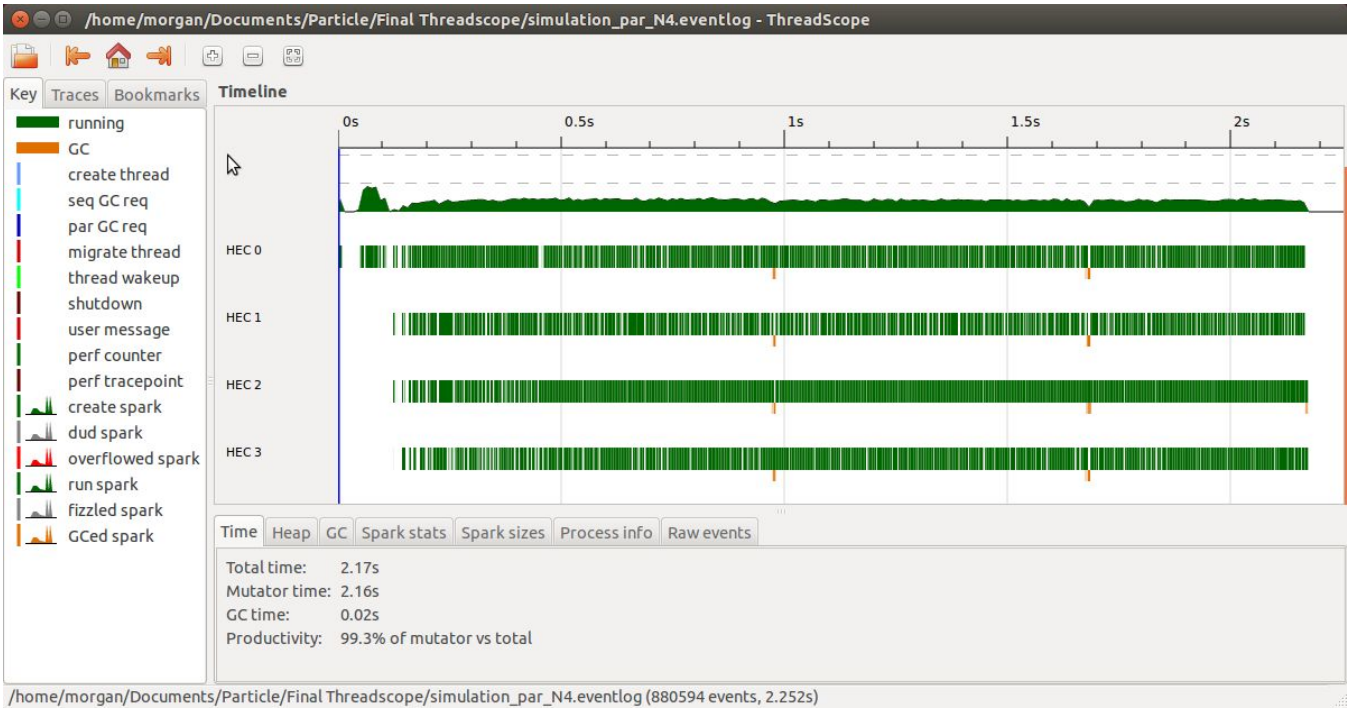
Simulation - No Parallel (Threadscope)



Simulation - Parallel 2 Cores (Threadscope)



Simulation - Parallel 4 Cores (Threadscope)



Possible Performance Improvements

Since this is a simple step-wise particle simulation, parallelization does not drastically improve the performance unless there are a lot of particles being created at each time step. Due to the limitations of my computer and CPU/GPU, some bottlenecking does occur in the I/O stage when the number of particles is increased to be larger than 10 per time step. At this rate and above, parallelization would result in the best performance improvements; however, due to the bottlenecking, the program does have noticeable lag in the screen rendering stage which results in slower total elapsed times for the program. If I were able to disable GPU rendering or if I had a system with a larger number of cores, I believe the parallelization would reap much greater benefits for the total runtime of the system and the overall balance of the workload. Additionally, on a system with more cores, there would not be any CPU bottlenecking which would dramatically increase the performance of the system. Finally, I am sure that there are better ways to parallelize the system, but with my changes I did see a slight improvement and I did see the benefits of parallel programming; however, I do recognize that for certain programs (such as my own), parallelizing the code can actually decrease the performance and make the program run slower.

Code Listing

All of the code for the program resides in the simulation.hs file which can easily be compiled using “ghc -threaded -eventlog -rtsopts --make simulation.hs” and run using “simulation <pic type> <run time> <number of particles>”.

Simulation.hs

```
import Control.Parallel(par)
import Data.Colour.SRGB.Linear hiding (blend)
import Data.Colour hiding (blend)
import Data.Colour.RGBSpace
import Data.Colour.RGBSpace.HSV
import Data.IORef
import Graphics.GLUtil
import Graphics.GLUtil.JuicyTextures
import Graphics.Rendering.OpenGL
import Graphics.UI.GLUT
import System.Environment
import System.Random
import System.Exit

data Particle = Particle { colorHue :: GLfloat
                         , colorSaturation :: GLfloat
                         , colorValue :: GLfloat
                         , size :: GLfloat
                         , x :: GLfloat
                         , y :: GLfloat
```

```
                    , z :: GLfloat
                    , xVel :: GLfloat
                    , yVel :: GLfloat
                    , zVel :: GLfloat
                    , time :: Int
                    } deriving (Eq, Show, Read)

type ParticlesArr = [Particle]

grav = 0.0015 :: GLfloat
simulationSize = Size 900 900

main :: IO ()
main = do
  args <- getArgs
  case args of
    [pic,time,num_p] -> do
       let run_time = read time
       let num_particles = read num_p
       (_progName, _args) <- getArgsAndInitialize
       particles <- newIORef ([] :: ParticlesArr)
       color <- newIORef (0.0 :: GLfloat)
       window <- createWindow "Parallel Functional Programming Final Project"
       windowSize $= simulationSize
       displayCallback $= (displaySimulation particles)
       reshapeCallback $= Just reshape
       keyboardMouseCallback $= Just keyboardMouse
       idleCallback $= Just (idleProcess run_time num_particles particles color)
       case pic of
          "stephen" -> do
             let picture = "stephenpic.png"
             spritetex <- getTexture picture
             mainLoop
          "particle" -> do
             let picture = "particle.png"
             spritetex <- getTexture picture
             mainLoop
          _ -> error "Picture type must be either 'particle' or 'stephen'"
    _ -> error "Usage:  simulation <pic type> <run time> <number of particles>"

reshape s@(Size w h) = do
  viewport $= (Position 0 0, s)
  postRedisplay Nothing

keyboardMouse (Char '\ESC') Down _ _ = exitSuccess
```

```
keyboardMouse (Char '\BS') Down _ _ = exitSuccess
keyboardMouse k s m p = putStrLn $ show k

extract :: (IO (Either String a)) -> IO a
extract action = do
  ioAction <- action
  case ioAction of
    Left a -> error a
    Right b -> return b

displayParticle :: Particle -> IO ()
displayParticle p = do
  color $ findColor (colorHue p) (colorSaturation p) (colorValue p)
  texCoord $ TexCoord2 zero zero
  vertex $ Vertex3 (posx - psize) (posy - psize) (posz)
  texCoord $ TexCoord2 zero one
  vertex $ Vertex3 (posx - psize) (posy + psize) (posz)
  texCoord $ TexCoord2 one one
  vertex $ Vertex3 (posx + psize) (posy + psize) (posz)
  texCoord $ TexCoord2 one zero
  vertex $ Vertex3 (posx + psize) (posy - psize) (posz)
  where
    posx = x p
    posy = y p
    posz = z p
    psize = size p
    zero = 0 :: GLfloat
    one = 1 :: GLfloat

findColor :: GLfloat -> GLfloat -> GLfloat -> Color3 GLfloat
findColor hue sat val = Color3 (par chanr (chanr)) (par chang (chang)) (par chanb (chanb))
  where
    rgbchannel = (hsv hue sat val)
    chanr = (channelRed rgbchannel)
    chang = (channelGreen rgbchannel)
    chanb = (channelBlue rgbchannel)

newColor val
  | val > 360 = par newVal (newVal)
  | otherwise = val
    where
      newVal = newColor (val - 360)

getTexture pic = do
  picTexture <- extract (readTexInfo pic loadTexture)
```

```
  texture Texture2D $= Enabled
  activeTexture $= TextureUnit 0
  textureBinding Texture2D $= Just picTexture
  textureFilter Texture2D $= ((Linear', Just Nearest), Linear')
  textureWrapMode Texture2D S $= (Mirrored, ClampToEdge)
  textureWrapMode Texture2D T $= (Mirrored, ClampToEdge)
  blend $= Enabled
  blendFunc $= (SrcAlpha, OneMinusSrcAlpha)
  generateMipmap' Texture2D
  return picTexture

displaySimulation :: IORef ParticlesArr -> IO ()
displaySimulation particleArr = do
  clear [ColorBuffer]
  showParticles <- get particleArr
  mapM_ (renderPrimitive Quads . displayParticle) showParticles
  flush
  postRedisplay Nothing

idleProcess :: GLfloat -> Int -> IORef ParticlesArr -> IORef GLfloat -> IO ()
idleProcess run_time num_particles particles curr_color = do
        curr_particles <- get particles
        let moveParticles = map moveParticle curr_particles
        initParticles <- sequence (replicate num_particles (initParticle curr_color))
        let totalParticles = (initParticles ++ moveParticles)
        particles $= filter (\particle -> time particle >= 0)  totalParticles
        colorVal <- get curr_color
        curr_color $= newColor (colorVal + 0.2)
        if colorVal > run_time
           then do exitSuccess
           else do putStrLn (show colorVal)

moveParticle :: Particle -> Particle
moveParticle p =
  p { yVel = par vel (vel - grav)
    , x = par posx (posx + xVel p)
    , y = par posy (posy + yVel p)
    , z = par posz (posz + zVel p)
    , time = par ttl (ttl - 1)
    , colorSaturation = par sat (sat * 1.3)
    , colorValue = par val (val - 0.015)
    , size = par psize (psize * 1.02)
    }
        where
           vel = yVel p
```

```
        posx = x p
        posy = y p
        posz = z p
        ttl = time p
        sat = colorSaturation p
        val = colorValue p
        psize = size p

initParticle :: IORef GLfloat -> IO Particle
initParticle huebase = do
  base <- get huebase
  phue <- randomRIO (base, base + 20)
  psaturation <- randomRIO (0.0001, 0.15 :: GLfloat)
  xv <- randomRIO (-0.04, 0.04)
  yv <- randomRIO (0.03, 0.065)
  return $ Particle { x = 0
                    , y = -1
                    , z = 1
                    , colorHue = newColor phue
                    , colorSaturation = psaturation
                    , colorValue = 1
                    , size = 0.04
                    , xVel = xv
                    , yVel = yv
                    , zVel = 0
                    , time = 60
                    }
```

References

https://stackoverflow.com/questions/14221761/haskell-opengl-transparency-not-working
https://stackoverflow.com/questions/5557082/what-does-the-dollar-equals-operator-do-in-haskell-glut-library
https://hackage.haskell.org/package/base-4.12.0.0/docs/Data-Either.html
https://hackage.haskell.org/package/GLUtil-0.3.0/docs/Graphics-GLUtil-Textures.html
https://hackage.haskell.org/package/GLUtil-0.8.8/docs/Graphics-GLUtil-JuicyTextures.html