CI4251 - Programación Funcional Avanzada
Entrega Tarea 4

Brian-Marcial Mendoza
07-41206
<07-41206@usb.ve>

Junio 07, 2015

Conjunto de Mandelbrot

Un Conjunto de Mandelbrot es un conjunto de puntos en el plano complejo que son cuasi-estables cuando se calculan iterando una función. Usualmente se emplea la función

$$z_{k+1} = z_k^2 + c$$

donde z_{k+1} es la $(k+1)-$ésima iteración del número complejo $z = a + bi$, z_k es la $k-$ésima iteración, y c es el número complejo que expresa la posición del punto en el plano complejo.
El valor inicial para z es cero, y las iteraciones deben repetirse hasta que la magnitud de z sea mayor que 2 (indicando que z tendría magnitud infinita eventualmente) o se llega a una cantidad arbitraria de iteraciones sin que esto ocurra.
La magnitud de un número complejo $z = a + bi$ se calcula como

$$|z| = \sqrt{a^2 + b^2}$$

Así mismo, calcular la función compleja $z_{k+1} = z_k^2 + c$ es muy simple si se simplifica la expresión hasta llegar

$$\begin{aligned} z'_{real} &= z^2_{real} - z^2_{imaginary} + c_{real} \\ z'_{imaginary} &= 2 \cdot z_{real} \cdot z_{imaginary} + c_{imaginary} \end{aligned}$$

Considerando que el punto con coordenadas (x, y) corresponde al número complejo $z = x + yi$, implante la función que itere la función compleja sobre el número complejo hasta converger o por un máximo de 255 iteraciones. La función debe retornar el número de iteraciones efectivamente completadas.

Para visualizar el Conjunto de Mandelbrot sobre un conjunto arbitrario en el plano complejo, presentaremos cada punto (x, y) con una tonalidad de gris proporcional a la cantidad de iteraciones completadas por la función anterior. Esto es, si `converge (x,y)` produce n como resultado, el pixel (x, y) tendrá el "color" `RGB n n n` que corresponde al $n-$ésimo gris.

Las partes interesantes del Conjunto de Mandelbrot están en un círculo de radio 2 alrededor del centro del Plano Complejo. Esto quiere decir que basta variar la parte real en el intervalo $[-2, 2]$ y simultáneamente se hace variar la parte imaginaria en el intervalo $[-2, 2]$, analizando los números complejos allí presentes.

El nivel de detalle observable dependerá del tamaño de la ventana con la cual se presente el conjunto, pues la cantidad de pixels horizontales y verticales establece cuántos puntos calcular. Para los que no han cursado Computación Gráfica, si se desea una ventana con w pixels de ancho y h pixels de alto, Ud. puede calcular

$$\begin{aligned} step_{real} &= 4{,}0/w \\ step_{imaginary} &= 4{,}0/h \end{aligned}$$

que le permitirán recorrer el intervalo real y el intervalo imaginario paso a paso, i.e. si $0 \leq x < w \wedge 0 \leq y < h$, entonces el pixel (x, y) correspondería al número complejo $(-2{,}0 + step_{real} * x, -2{,}0 + step_{imaginary} * y)$

Provea tres implantaciones del cálculo de la parte interesante del Conjunto de Mandelbrot sobre una "ventana" de visualización que aprovechen estrategias paralelas, el Monad `Par` y la librería de vectorización REPA, respectivamente.

Asegúrese que el desempeño de su implantación sea bueno en "ventanas" de hasta 1280x1024. No haga ninguna suposición sobre la cantidad de núcleos disponibles para el cómputo; si lo desea, utilice las funciones ofrecidas por GHC para determinar cuántos hay disponibles, pero intente encontrar un particionado dinámico razonable. Finalmente, escriba un programa principal usando `Criterion` que permita comparar la velocidad de ejecución de las tres implantaciones.

1. Librerías Utilizadas y Funciones Auxiliares

Para la resolución de este problema se utilizaron las siguientes librerías:

```
{-# LANGUAGE BangPatterns #-}

import Control.Concurrent
import Control.Monad.Identity
import Control.Monad.Par
import Control.Parallel
import Control.Parallel.Strategies hiding (parMap)
import Criterion
import Criterion.Main
import qualified Data.Array.Repa as AR
import Data.List.Split (chunksOf)
import Data.Word
import GHC.Conc (numCapabilities)
```

Junto a esto se usaron las siguientes funciones auxiliares:

```
cores :: Int
cores = numCapabilities
```

Esta función permite obtener cuantos cores están disponibles en la ejecución, para así tener una mejor de idea de como distribuir el trabajo a los distintos cores.

```
converge :: (Double,Double) -> Word8
converge (!x,!y) = go 0 0 x y 0
  where
    go _ _ _ _ 255 = 255
    go !zr !zi cr ci !n
      = if sqrt (zr^2 + zi^2) > 2
        then n
        else go newRealZ newImagZ cr ci (n+1)
      where
        newRealZ = zr^2 - zi^2 + cr
        newImagZ = 2*zr*zi + ci
```

Esta función es la que se aplicará sobre cada pixel (x, y) para obtener su color RGB. Se utilizaron hashbangs para evitar la creación de thunks muy

grandes y así acelerar la ejecución.

```
buildWindow :: Double -> Double -> [(Double,Double)]
buildWindow !w !h
  = [ (x,y) | x <- [0..(w-1)], y <- [0..(h-1)] ]
```

Esta función será utilizada por cada uno de los tres métodos para construir la "ventana" inicial de pixeles, que será una lista plana desde $(0, 0)$ hasta (w-1,h-1).

2. Solución con Estrategias Paralelas

```
mandelStrat :: Word32 -> Word32 -> [[Word8]]
mandelStrat w h
  = chunksOf height'
  ( map (converge . toComplex) window
    `using`
    parListChunk chunks rdeepseq)
  where
    width :: Double
    width = fromIntegral w

    height :: Double
    height = fromIntegral h

    height' :: Int
    height' = fromIntegral h

    window :: [(Double,Double)]
    window = buildWindow width height

    toComplex :: (Double,Double) -> (Double,Double)
    toComplex (!x,!y) = (-2 + stepReal*x, -2 + stepImaginary*y)

    stepReal :: Double
    stepReal = 4 / width

    stepImaginary :: Double
    stepImaginary = 4 / height

    chunks = height' `div` cores
```

Esta función se basa en paralelizar el trabajo usando una estrategia de hacer chunks de la lista a procesar y se pasa un chunk a cada core para que eje-

cute las cuentas necesarias. Se encontró que el número de chunks que daba mayores beneficios era la cantidad de cores disponibles. Es decir, dividir la lista en partes iguales para cada core y cada uno tendría un único workload de "$altura/n$" filas a calcular.

El flujo de ejecución de esta solución es el siguiente:

- Generar la "ventana" utilizando `buildWindow`.
- Dividir dicha "ventana" de forma tal que quedara una cantidad de "subventanas" igual al número de cores disponibles, con cada "subventana" con "$altura/n$" elementos.
- Cada core le aplica dos funciones a cada elemento de su "subventana" en el siguiente orden:
  - Transforma el "pixel" en su representación de número complejo en el rango especificado por el ejercicio utilizando la función `toComplex`.
  - Aplica la función de convergencia `converge` a dicho número complejo.
- Una vez todos los cores hayan terminado su trabajo sobre la "ventana", se ejecuta la función `chunksOf` a la lista resultante para tener así un resultado que concuerde con la firma del enunciado, una lista de listas de Word8.

3. Solución con el Monad Par

Esta función se basa en paralelizar el trabajo usando el Monad Par. La librería no provee una forma directa de paralelizar el trabajo de un map sobre una lista vía chunks, así que se utilizó una función auxiliar `parMapChunk` para obtener esta funcionalida para poder competir en tiempo en comparación con las otras soluciones. Esta función fue encontrada en una presentación de Simon Marlow titulada "Parallel & Concurrent Haskell 2:The Par Monad".

```
parMapChunk :: NFData b => Int -> (a -> b) -> [a] -> Par [b]
parMapChunk n f xs
  = do
    xss <- parMap (map f) (chunksOf n xs)
    return (concat xss)
```

```
mandelPar   :: Word32 -> Word32 -> [[Word8]]
mandelPar w h
  = chunksOf height'
  $ runPar
  $ parMapChunk chunks (converge . toComplex) window
  where
    width :: Double
    width = fromIntegral w

    height :: Double
    height = fromIntegral h

    height' :: Int
    height' = fromIntegral h

    window :: [(Double,Double)]
    window = buildWindow width height

    toComplex :: (Double,Double) -> (Double,Double)
    toComplex (!x,!y) = (-2 + stepReal*x, -2 + stepImaginary*y)

    stepReal :: Double
    stepReal = 4 / width

    stepImaginary :: Double
    stepImaginary = 4 / height

    chunks :: Int
    chunks = height' `div` cores
```

El flujo de ejecución de esta solución es el siguiente:

- Generar la "ventana" utilizando `buildWindow`.
- Dividir dicha "ventana" de forma tal que quedara una cantidad de "subventanas" igual al número de cores disponibles, con cada "subventana" con "$altura/n$" elementos.
- Cada core le aplica dos funciones a cada elemento de su "subventana" en el siguiente orden:
  - Transforma el "pixel" en su representación de número complejo en el rango especificado por el ejercicio utilizando la función `toComplex`.
  - Aplica la función de convergencia `converge` a dicho número complejo.
- Una vez todos los cores hayan terminado su trabajo sobre la "ventana", se ejecuta la función `chunksOf` a la lista resultante para tener así un resultado que concuerde con la firma del enunciado, una lista de listas de Word8.

4. Solución con Vectores REPA

```
mandelREPA  :: Word32 -> Word32 -> [[Word8]]
mandelREPA w h
  = do
    let window = AR.fromListUnboxed
                 (AR.Z AR.:. height' AR.:. width')
                 (buildWindow width height)
                  :: AR.Array AR.U AR.DIM2 (Double,Double)
    chunksOf height' $ AR.toList $ runIdentity $ go window
  where
    go window
      = do
        window' <- AR.computeP
                     $ AR.map (converge . toComplex) window
                       :: Identity (AR.Array AR.U AR.DIM2 Word8)
        return window'

    width :: Double
    width = fromIntegral w

    height :: Double
    height = fromIntegral h

    width' :: Int
    width' = fromIntegral w

    height' :: Int
    height' = fromIntegral h

    toComplex :: (Double,Double) -> (Double,Double)
    toComplex (!x,!y) = (-2 + stepReal*x, -2 + stepImaginary*y)

    stepReal :: Double
    stepReal = 4 / width

    stepImaginary :: Double
    stepImaginary = 4 / height
```

Esta función se basa en paralelizar el trabajo utilizando vectores de la librería REPA. El flujo de ejecución de esta solución es el siguiente:

- Generar la "ventana" utilizando `buildWindow`.
- Convertir dicha "ventana" en una vector REPA Unboxed de dos dimensiones que contiene tuplas de Doubles.

- Se utilizan en conjunto las funciones `map` y `computeP` de la librería REPA para efectuar en paralelo las siguientes dos funciones sobre cada elemento del vector en este orden:
    - Transforma el "pixel" en su representación de número complejo en el rango especificado por el ejercicio utilizando la función `toComplex`.
    - Aplica la función de convergencia `converge` a dicho número complejo.
- Una vez terminada la transformación se transforma de vector REPA a una lista, y se ejecuta la función `chunksOf` a la lista resultante para tener así un resultado que concuerde con la firma del enunciado, una lista de listas de Word8.

5. Suite de Pruebas y Resultados

A continuación se presenta el suite de pruebas de `Criterion` utilizado para comparar cada función, utilizando la "ventana" de mayor tamaño posible sugerida por el enunciado. Se utilizan pruebas nf (Normal Form) para asegurar que los cálculos se están realizando y la estructura sea evaluada a completitud para tener una comparación lo más "justa" posible.

```
main = defaultMain [
          bench "mandelStrat" $ nf (mandelStrat 1280) 1024,
          bench "mandelPar" $ nf (mandelPar 1280) 1024,
          bench "mandelREPA" $ nf (mandelREPA 1280) 1024
       ]
```

La ejecución devolvió estos resultados, primero utilizando un sólo core:

```
benchmarking mandelStrat
time                 921.7 ms   (907.5 ms .. 934.8 ms)
                     1.000 R^2   (1.000 R^2 .. 1.000 R^2)
mean                 957.5 ms   (949.1 ms .. 964.3 ms)
std dev              10.69 ms   (0.0 s .. 11.80 ms)
variance introduced by outliers: 19% (moderately inflated)

benchmarking mandelPar
time                 867.0 ms   (757.5 ms .. 1.083 s)
                     0.992 R^2   (0.988 R^2 .. 1.000 R^2)
mean                 882.2 ms   (852.3 ms .. 901.4 ms)
std dev              28.86 ms   (0.0 s .. 33.23 ms)
variance introduced by outliers: 19% (moderately inflated)

benchmarking mandelREPA
time                 755.9 ms   (671.4 ms .. 816.7 ms)
                     0.999 R^2   (0.995 R^2 .. 1.000 R^2)
mean                 764.9 ms   (752.9 ms .. 773.9 ms)
std dev              13.85 ms   (0.0 s .. 15.68 ms)
variance introduced by outliers: 19% (moderately inflated)
```

Luego utilizando dos cores:

```
benchmarking mandelStrat
time                 786.8 ms   (748.2 ms .. 835.5 ms)
                     1.000 R^2   (0.998 R^2 .. 1.000 R^2)
mean                 777.3 ms   (765.0 ms .. 785.2 ms)
std dev              11.78 ms   (0.0 s .. 13.56 ms)
variance introduced by outliers: 19% (moderately inflated)

benchmarking mandelPar
time                 675.8 ms   (620.1 ms .. 720.5 ms)
                     0.999 R^2   (0.997 R^2 .. 1.000 R^2)
mean                 670.3 ms   (659.9 ms .. 676.9 ms)
std dev              9.964 ms   (0.0 s .. 11.46 ms)
variance introduced by outliers: 19% (moderately inflated)

benchmarking mandelREPA
time                 747.2 ms   (729.5 ms .. 774.1 ms)
                     1.000 R^2   (1.000 R^2 .. 1.000 R^2)
mean                 744.7 ms   (740.1 ms .. 747.6 ms)
std dev              4.289 ms   (0.0 s .. 4.901 ms)
variance introduced by outliers: 19% (moderately inflated)
```

Y después utilizando todos los cores disponibles, en este caso 4:

```
benchmarking mandelStrat
time                 768.3 ms   (741.5 ms .. 796.7 ms)
                     1.000 R^2   (0.999 R^2 .. 1.000 R^2)
mean                 781.3 ms   (775.6 ms .. 786.9 ms)
std dev              9.849 ms   (0.0 s .. 9.850 ms)
variance introduced by outliers: 19% (moderately inflated)

benchmarking mandelPar
time                 599.4 ms   (512.7 ms .. 673.9 ms)
```

```
                    0.998 R^2   (0.991 R^2 .. 1.000 R^2)
mean                618.4 ms   (597.9 ms .. 632.6 ms)
std dev             21.33 ms   (0.0 s .. 24.58 ms)
variance introduced by outliers: 19% (moderately inflated)


benchmarking mandelREPA
time                801.5 ms   (704.0 ms .. 854.1 ms)
                    0.998 R^2   (0.996 R^2 .. 1.000 R^2)
mean                846.3 ms   (817.6 ms .. 862.3 ms)
std dev             25.32 ms   (0.0 s .. 27.74 ms)
variance introduced by outliers: 19% (moderately inflated)
```

Cada solución presenta tiempos de ejecución bastante aceptables considerando que se esta haciendo un cálculo sobre 1,310,720 elementos, y cada cálculo puede tener hasta 255 iteraciones cada uno. Sin embargo, la diferencia empleando varios cores no pareciera ser significativa con lo esperado, con una ganancia en velocidad respecto a el uso de un solo core de:

- 17 % (Estrategias), 28 % (Monad Par) y 1 % (REPA) utilizando dos cores.
- 19 % (Estrategias), 44 % (Monad Par) y -6 % (REPA) utilizando cuatro cores.

Se estima que utilizando la librería Accelerate en conjunto con una máquina con un GPU disponible los tiempos de REPA podrían mejorar, pero en conclusión pareciera que el beneficio de utilizar varios cores no es significativo, tanto en porcentaje de incremento de velocidad y en tiempo total de ejecución respecto a la cantidad de cores disponibles.