Programación Funcional Avanzada

Haskell Paralelo

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"

UNIVERSIDAD SIMÓN BOLÍVAR

Concurrencia vs. Paralelismo

. . . otra vez

- Dificultades de la programación concurrente
  - ¿Cuántos *cores* usar?
  - ¿Cuáles hilos se comunican entre si?
  - ¿Cómo dividir el trabajo entre hilos?
  - ¿Cuáles datos son compartidos y cuáles privados?
  - ¿Cómo determinar si todos los hilos terminaron?
  - ¿Cómo *probarlo* de forma exhaustiva?

UNIVERSIDAD SIMÓN BOLÍVAR

Concurrencia vs. Paralelismo

. . . otra vez

- Dificultades de la programación concurrente
  - ¿Cuántos *cores* usar?
  - ¿Cuáles hilos se comunican entre si?
  - ¿Cómo dividir el trabajo entre hilos?
  - ¿Cuáles datos son compartidos y cuáles privados?
  - ¿Cómo determinar si todos los hilos terminaron?
  - ¿Cómo *probarlo* de forma exhaustiva?
- Con lenguajes y métodos tradicionales, *todas* las actividades son responsabilidad exclusiva del programador.
- El problema de fondo: **no-determinismo**.

UNIVERSIDAD SIMÓN BOLÍVAR

Concurrencia vs. Paralelismo

. . . otra vez

- Dificultades de la programación concurrente
  - ¿Cuántos *cores* usar?
  - ¿Cuáles hilos se comunican entre si?
  - ¿Cómo dividir el trabajo entre hilos?
  - ¿Cuáles datos son compartidos y cuáles privados?
  - ¿Cómo determinar si todos los hilos terminaron?
  - ¿Cómo *probarlo* de forma exhaustiva?
- Con lenguajes y métodos tradicionales, *todas* las actividades son responsabilidad exclusiva del programador.
- El problema de fondo: **no-determinismo**.

Haskell provee mecanismos de paralelismo implícito
para reducir (¿eliminar?) estos problemas

UNIVERSIDAD SIMÓN BOLÍVAR

Paralelismo en Haskell

Can I haz parallelz pleez?

- "Paralelismo semi-implícito" – evaluación paralela
- Evaluar **todas** las subexpresiones en paralelo no es práctico.
- "Anotaciones de paralelismo" – **sugerir** fragmentos paralelizables.
  - El *Runtime System* los activa concurrentemente.
  - Barato para el programador.
  - Preserva la correctitud del programa.
- El programa sigue siendo determinístico.
- Mejora notable sin mucho esfuerzo – dominar la flojera.

UNIVERSIDAD SIMÓN BOLÍVAR

Paralelismo en Haskell

`Control.Parallel` – anotaciones paralelas

```
par :: a -> b -> b
```

- Crea una chispa (*spark*) para la primera expresión.

UNIVERSIDAD SIMÓN BOLÍVAR

Paralelismo en Haskell

`Control.Parallel` – anotaciones paralelas

```
par :: a -> b -> b
```

- Crea una chispa (*spark*) para la primera expresión.
- *Runtime system* busca la oportunidad de convertirla en hilo...
  - Según la carga de la máquina – ¿*cores* libres?
  - Según el costo de la evaluación – ¿vale la pena un hilo?
  - ...y si eso ocurre, correrá en paralelo.

UNIVERSIDAD SIMÓN BOLÍVAR

Paralelismo en Haskell

`Control.Parallel` – anotaciones paralelas

```
par :: a -> b -> b
```

- Crea una chispa (*spark*) para la primera expresión.
- *Runtime system* busca la oportunidad de convertirla en hilo...
  - Según la carga de la máquina – ¿*cores* libres?
  - Según el costo de la evaluación – ¿vale la pena un hilo?
  - ...y si eso ocurre, correrá en paralelo.
- Se retorna el valor de la segunda expresión.

UNIVERSIDAD SIMÓN BOLÍVAR

Paralelismo en Haskell

`Control.Parallel` – anotaciones paralelas

```
par :: a -> b -> b
```

- Crea una chispa (*spark*) para la primera expresión.
- *Runtime system* busca la oportunidad de convertirla en hilo...
  - Según la carga de la máquina – ¿*cores* libres?
  - Según el costo de la evaluación – ¿vale la pena un hilo?
  - ...y si eso ocurre, correrá en paralelo.
- Se retorna el valor de la segunda expresión.
- `par` **no** garantiza un nuevo hilo a cada rato.
  - Es una "sugerencia" del programador.
  - Es barata – en todo sentido.
  - Facilita "insistir" en el paralelismo.

Si `b` *depende* de `a`
podría valer la pena evaluarlas en paralelo.

UNIVERSIDAD SIMÓN BOLÍVAR

Paralelismo en Haskell

`Control.Parallel` – anotaciones anti-paralelas

```
pseq :: a -> b -> b
```

- **Impide** crear una chispa para cada expresión – todo en un sólo hilo.
- Evalúa la primera expresión y luego retorna la segunda expresión.
- Garantiza que el cómputo se hace en el orden indicado.

UNIVERSIDAD SIMÓN BOLÍVAR

Haskell Paralelo

`par` y `pseq` en acción

```
f `par` e `pseq` (f + e)
```

- Una chispa creada para `f`.
- La chispa se convierte en hilo para calcular `f`.
- `e` se evalúa en el hilo inicial, en paralelo con `f`.
- Se retorna `f+e` – en forma de *thunk*.

UNIVERSIDAD SIMÓN BOLÍVAR

Haskell Paralelo

par y pseq en acción

```
f `par` e `pseq` (f + e)
```

- Una chispa creada para `f`.
- La chispa se convierte en hilo para calcular `f`.
- `e` se evalúa en el hilo inicial, en paralelo con `f`.
- Se retorna `f+e` – en forma de *thunk*.

```
main = let p = primes 5000
           q = nqueens 12
       in
          p `par` q `pseq` $ print (p,q)
```

Uno o más *cores* mejoran el desempeño

UNIVERSIDAD SIMÓN BOLÍVAR

¡Se preserva la correctitud!

```
par a b == b
```

- Reemplazar `par a b` con `b` no cambia la semántica del programa
  - Solamente su velocidad y uso de memoria.
  - `par` no puede "hacer equivocar" al programa.
  - No hay *deadlocks* ni condiciones de carrera – ¡ga-ran-ti-za-do!
- Usar `par` es muy barato – agrega chispas a un *buffer* circular
  - Crear tantas chispas como se pueda.
  - "Exceso" de paralelismo en lo posible.
  - Permite escalar sin cambiar el programa.

UNIVERSIDAD SIMÓN BOLÍVAR

Modelo de operación

¿Cómo opera el *runtime*?

- `-N4` genera 4 hilos de sistema operativo.
- *RTS* multiplexa los hilos Haskell entre los hilos del sistema operativo.
- Hilos Haskell se generan con `forkIO` o `par` "promovido".
- Se asocia cada hilo de sistema operativo (*worker*) con un núcleo de cómputo particular – explotar afinidad (*affinity*).
- Cada *worker* tiene un *pool* de chispas – `par` agrega *thunks* a las listas.
- Un *worker* ocioso convierte alguna chispa en hilo.
- ¿Y el recolector de basura?
  - Es multihilo.
  - No interrumpe a los hilos que están ejecutando otras cosas.

UNIVERSIDAD SIMÓN BOLÍVAR

Nuestro primer intento paralelo

```
import Control.Monad
import Control.Parallel
import System.Environment

parfib 0 = 0
parfib 1 = 1
parfib n = n1 `par` n2 `pseq` (n1+n2)
  where n1 = parfib $ n-1
        n2 = parfib $ n-2

main = print . parfib . read . head =<< getArgs
```

Compilar, correr y analizar

- Compilamos para aprovechar hilos

```
$ ghc -O2 -o pfib --make -threaded pfib.hs
```

Compilar, correr y analizar

- Compilamos para aprovechar hilos

```
$ ghc -O2 -o pfib --make -threaded pfib.hs
```

- Lo corremos con un sólo núcleo

```
$ time ./pfib +RTS -N1 -RTS 42
267914296
real 1m47.902s
```

UNIVERSIDAD SIMÓN BOLÍVAR

Compilar, correr y analizar

- Compilamos para aprovechar hilos

```
$ ghc -O2 -o pfib --make -threaded pfib.hs
```

- Lo corremos con un sólo núcleo

```
$ time ./pfib +RTS -N1 -RTS 42
267914296
real 1m47.902s
```

- Lo corremos con cuatro núcleos

```
$ time ./pfib +RTS -N4 -RTS 42
267914296
real 0m46.219s
```

Bastante más rápido considerando el esfuerzo,
pero 175 % aún está lejos del ansiado 400 % teórico.

UNIVERSIDAD SIMÓN BOLÍVAR

¿Qué nos dice el *runtime*?

- Compilamos de nuevo con la opción `-rtsopts`
- Puede ejecutarse el programa solicitando estadísticas adicionales sobre las operaciones del *runtime*.

```
$ time ./pfib +RTS -N4 -sstderr -RTS 42
...
SPARKS: 433509910
   (252 converted, 430988296 GC'd, 2521362 fizzled)
...
```

  - Sólo 252 se convirtieron en hilos efectivamente.
  - El resto no tuvo oportunidad de convertirse (GC) o bien otro hilo evaluó la expresión antes (fizzled).
- Las chispas son baratas – ¡pero no son gratis!
- La unidad de procesamiento por hilo es demasiado pequeña.

UNIVERSIDAD SIMÓN BOLÍVAR

Combinar paralelo y secuencial

Umbrales para granularidad

```
parfib n t
  | n <= t     = fib n
  | otherwise = n1 `par` n2 `pseq` (n1+n2)
                where n1 = parfib (n-1) t
                      n2 = parfib (n-2) t

fib 0 = 0
fib 1 = 1
fib n = fib (n-2) + fib (n-1)

main = do [n,t] <- getArgs
          print $ parfib (read n) (read t)
```

UNIVERSIDAD SIMÓN BOLÍVAR

Un poco mejor . . .

OMFG!

```
$ time ./ppfib +RTS -N4 -RTS 42 17
267914296

real 0m10.909s
```

Moraleja

- `par` es barato – sea liberal al usarlo.
- Refine la granularidad del paralelismo para reducir el costo acumulado de chispas – más trabajo por chispa, que chispas por trabajo.
- Anote, mida, revise – enguaje y repita.

`par` es de muy “bajo nivel” – ¿cómo abstraernos?

UNIVERSIDAD SIMÓN BOLÍVAR

Aumentando la abstracción

Paralelismo sobre listas

¿Cómo aplicar una función sobre todos los elementos de una lista, pero explotando el paralelismo?

```
parMap :: (a -> b) -> [a] -> [b]
parMap f (x:xs) = let r = f x
                  in r `par` r : parMap f xs
parMap _ []     = []
```

- Haskell es perezoso – `f x` no evalúa "hasta el final"
- *Normal Form* – evaluación completa (ambiciosa).
- *Weak Head Normal Form* – evaluación perezosa mínima.
- El paralelismo se aprovecha mejor o peor dependiendo del tipo de valor calculado por `f`.

UNIVERSIDAD SIMÓN BOLÍVAR

Estrategias de Evaluación Paralela

`Control.Parallel.Strategies`

```
type Strategy a = a -> Eval a
```

- Una **estrategia**
  - Política para evaluar una expresión, posiblemente en paralelo.
  - Mantiene el resultado del programa – paralelismo determinístico.
  - Asegura que el valor es evaluado hasta cierto punto.
  - Estrategias complejas en base a estrategias simples.

UNIVERSIDAD SIMÓN BOLÍVAR

Estrategias de Evaluación Paralela

`Control.Parallel.Strategies`

```
type Strategy a = a -> Eval a
```

- Una **estrategia**
  - Política para evaluar una expresión, posiblemente en paralelo.
  - Mantiene el resultado del programa – paralelismo determinístico.
  - Asegura que el valor es evaluado hasta cierto punto.
  - Estrategias complejas en base a estrategias simples.
- Permite separar preocupaciones – como siempre
  - ¿Cómo evaluar el *conjunto*? – Algoritmo
  - ¿Cómo evaluar cada componente? – Estrategia
  - Definir una estructura perezosa que describa el cómputo – establecer una estrategia para evaluarla en paralelo.

UNIVERSIDAD SIMÓN BOLÍVAR

Estrategias de evaluación

`Eval` es un monad

```
type Strategy a = a -> Eval a
```

- `Eval a` es un monad `Identity` de evaluación estricta – `m >>= f` *garantiza* que `m` se evalúa antes de conectar con `f`.
- Instancias convenientes de `Functor` y `Applicative` – simplifica crear estrategias para estructuras de recorrido regular.
- . . . y es posible escaparse del Monad

```
runEval :: Eval a -> a
```

UNIVERSIDAD SIMÓN BOLÍVAR

Intuición – esta no es la implantación exacta

- Estrategia trivial – no evalúa nada

```
r0 :: Strategy a
r0 _ = ()
```

Intuición – esta no es la implantación exacta

- Estrategia trivial – no evalúa nada

```
r0 :: Strategy a
r0 _ = ()
```

- Evaluar hasta WHNF

```
rseq :: Strategy a
rseq x = x `seq` ()
```

UNIVERSIDAD SIMÓN BOLÍVAR

Intuición – esta no es la implantación exacta

- Estrategia trivial – no evalúa nada

```
r0 :: Strategy a
r0 _ = ()
```

- Evaluar hasta WHNF

```
rseq :: Strategy a
rseq x = x `seq` ()
```

- Evaluar un par en secuencia o en paralelo

```
seqPair :: Strategy a -> Strategy b
        -> Strategy (a,b)
seqPair a b = a `seq` b `seq` (a,b)

parPair :: Strategy a -> Strategy b
        -> Strategy (a,b)
parPair a b = a `par` b `seq` (a,b)
```

UNIVERSIDAD SIMÓN BOLÍVAR

Implantación

El API del Monad `Eval`

- Se definen como un Monad Aplicativo

```
data Strategy a = a -> Eval a
using :: a -> Strategy a -> a
```

- Diferentes profundidades de evaluación

```
r0        :: Strategy a   -- Lazy
rseq      :: Strategy a   -- Weak Head Normal Form
rpar      :: Strategy a   -- ...en paralelo
rdeepseq  :: Strategy a   -- Normal Form
```

- Para los contenedores comunes

```
parList :: Strategy a -> Strategy [a]
parMap  :: Strategy b -> (a -> b) -> [a] -> [b]
```

UNIVERSIDAD SIMÓN BOLÍVAR

Definibles por el usuario

```
data Tree a = Leaf a | Node [Tree a]

parTree :: Int -> Strategy (Tree a)
parTree 0 tree       = r0 tree
parTree n (Leaf a)   = return $ Leaf a
parTree n (Node ts) = do
  us <- parList (parTree (n-1)) ts
  return $ Node us
```

- Evalúar el árbol en paralelo hasta una profundidad particular.
- La estrategia es un Monad Aplicativo – puedo aplicar cualquier función (o secuencia de funciones) al contenedor.
- "¿Qué evaluar?" separado de "¿Cómo evaluar?"

UNIVERSIDAD SIMÓN BOLÍVAR

Evaluando en paralelo

La técnica general

- Se tiene una estructura de datos que se pretende evaluar en paralelo.
- Necesitamos una estrategia

```
type Strategy a = a -> Eval a
```

  - `Strategy` – función que expresa el paralelismo.
  - La estrategia puede evaluar parte o toda la estructura – eso depende de la complejidad de la estructura y el cómputo.
  - La estrategia *nunca* cambia el valor – sólo lo evalúa parcial o totalmente dentro del Monad `Eval`.
- Monad `Eval` establece el orden de aplicación de las estrategias
  - `rseq a` – evaluarlo de manera *estricta* inmediatamente.
  - `rpar a` – sugerir la evaluación en paralelo.

UNIVERSIDAD SIMÓN BOLÍVAR

Resolviendo Sudoku

- Algoritmo de Propagación de Restricciones (Norvig) implantado en Haskell (Manu & Fischer)
- Resolver *un* Sudoku es fundamentalmente secuencial – queremos resolver *miles* de lo más rápido posible.

UNIVERSIDAD SIMÓN BOLÍVAR

Resolviendo Sudoku

Línea base sin paralelismo

```
import Sudoku
import Control.Exception
import System.Environment

main = do
  [f] <- getArgs
  grids <- fmap lines $ readFile f
  mapM_ (evaluate . solve) grids
```

- `solve :: String -> Maybe Grid` – resuelve *un* Sudoku.
- `evaluate :: a -> IO a` – obliga a evaluar hasta WHNF.

UNIVERSIDAD SIMÓN BOLÍVAR

Resolviendo Sudoku

Línea base sin paralelismo

- Compilamos para aprovechar hilos

```
$ ghc --make -threaded sudoku0.hs
```

- Lo aplicamos a resolver 1000 Sudokus con un sólo hilo

```
$ sudoku0 1000.txt +RTS -s -RTS
...
SPARKS: 0 (0 converted, 0 pruned)
...
Total time    9.43s  (  9.54s elapsed)
```

UNIVERSIDAD SIMÓN BOLÍVAR

Resolviendo Sudoku

¡Pero tenemos dos núcleos!

```
let (as,bs) = splitAt (length grids `div` 2) grids
evaluate $ unEval $ do
  a <- rpar (deep (map solve as))
  b <- rpar (deep (map solve bs))
  rseq a
  rseq b
  return ()
```

- Dividimos la lista de problemas en dos.
- `deep` (basado en `deepseq`) – evaluar hasta Forma Normal.
- `rpar` – estrategia paralela para cada mitad.
- `rseq` – asegurar que ambos valores se calculan.
- Sólo nos importa *calcular*.

UNIVERSIDAD SIMÓN BOLÍVAR

Resolviendo Sudoku

Aprovechando los dos núcleos

- Compilamos para aprovechar hilos

```
$ ghc --make -threaded sudoku1.hs
```

- Lo aplicamos a resolver 1000 Sudokus con dos hilos

```
$ sudoku1 1000.txt +RTS -N2 -s -RTS
...
SPARKS: 2 (1 converted, 0 pruned)
...
Total time    11.48s  (  7.48s elapsed)
```

- ¡Mejoramos! – *speedup* $= 9{,}54/7{,}48 = 1{,}27$
- Sólo 1 chispa se promovió a hilo
  - Paralelismo, pero desbalanceado – un hilo ocupado, otro ocioso.
  - ¡Resolver cada Sudoku toma tiempo diferente!

UNIVERSIDAD SIMÓN BOLÍVAR

Mejorando las Particiones

- **Particionamiento Estático**
  - Dividir en partes según criterio particular *antes* de ejecutar el programa.
  - Lo que acabamos de hacer – "mitad y mitad".
  - Sólo sirve si el criterio es acertado.
- **Particionamiento Dinámico**
  - Distribuir el trabajo en partes más pequeñas.
  - Asignarlo a procesadores que estén ociosos.
  - Programador divide en partes suficientemente pequeñas, RTS se encarga de la asignación

UNIVERSIDAD SIMÓN BOLÍVAR

Resolviendo Sudoku

Hacia partes más pequeñas – primera aproximación

```
parMapper :: (a -> b) -> [a] -> Eval [b]
parMapper f []     = return []
parMapper f (a:as) = do
  b  <- rpar (f a)
  bs <- parMapper f as
  return (b:bs)
```

- Al usar `rpar` estamos *indicando* evaluar en paralelo – no es una "sugerencia" como se haría con `par`.
- La *cantidad* de evaluación efectuada por `(f a)` podría regularse vía un `Strategy a` – esta solución no deja espacio para esa posibilidad.

UNIVERSIDAD SIMÓN BOLÍVAR

Resolviendo Sudoku

Aprovechando los dos núcleos dinámicamente

```
evaluate $ deep $ unEval $ parMapper solve grids
```

- Cada Sudoku será un problema paralelizable – `map` paralelo.
- `parMapper` creará una chispa por cada problema –
  las chispas son baratas de crear así que no nos lastima.
- Problemas cortos liberan su CPU para que otro trabaje.
- Mantener todos los CPUs ocupados la mayor parte del tiempo.

UNIVERSIDAD SIMÓN BOLÍVAR

Resolviendo Sudoku

Aprovechando los dos núcleos dinámicamente

- Compilamos para aprovechar hilos

```
$ ghc --make -threaded sudoku2.hs
```

- Lo aplicamos a resolver 1000 Sudokus con dos hilos

```
$ sudoku2 1000.txt +RTS -N2 -s -RTS
...
SPARKS: 1000 (1000 converted, 0 pruned)
...
Total time    10.47s  (  5.40s elapsed)
```

- ¡Mejoramos! – *speedup* $= 9{,}54/5{,}40 = 1{,}77$

Work stealing like a boss!

UNIVERSIDAD SIMÓN BOLÍVAR

Resolviendo Sudoku

Particionamiento dinámico

- No usen `parMapper` – `Control.Parallel.Strategies` provee

```
parList :: Strategy a -> Strategy [a]
```

  - Estrategia para la lista, tomando en cuenta la estrategia por elemento – procesamiento paralelo generalizado para listas polimórficas.
  - No es "evaluación final" – combinable con más estrategias, `Par` y `Seq`.

- La manera idiomática de escribir el programa es

```
evaluate $ deep $ map solve grids
                  `using`
                  parList Seq
```

  - Estrategia `parList rseq` establece cómo evaluar elementos – en paralelo (un *spark* cada uno), cada evaluación estricta.
  - Evalúa la lista resultante del `map` "usando" esa estrategia.

UNIVERSIDAD SIMÓN BOLÍVAR

Quiero saber más...

- Documentación de `Control.DeepSeq`
- Documentación de `Control.Parallel`
- Documentación de `Control.Parallel.Strategies`
- Runtime Support for Multicore Haskell
  (detalles de implantación del RTS para soportar paralelismo)

`Control.Parallel` cambió en GHC 7.x –
mis ejemplos *no* funcionan en 6.12

UNIVERSIDAD SIMÓN BOLÍVAR