Informe Proyecto de Grado
2015

Andrés González

September 9, 2015

Informe Proyecto de Grado

Contents

Chapter 1

Introducción

INTRODUCCIÓN
En este informe se detalla el trabajo realizado para el Proyecto de Grado de la Carrera de Ingeniería en Computación. El objetivo de este trabajo es documentar todos los pasos seguidos en este proyecto y la investigación que se hizo del tema propuesto. La investigación consistió en estudiar cuáles son las posibilidades existentes actualmente para desarrollar programas que puedan ser ejecutados en paralelo utilizando lenguajes funcionales, en particular el lenguaje Haskell. Los programas paralelos pueden ser ejecutados por más de un procesador (CPU) a la vez y pueden tener importantes mejoras en cuanto a su performance. Además se debe tener en cuenta que la mayoría de las computadoras que se venden hoy en día, disponen de más de un núcleo y están disponibles para el público en general. Estos aspectos hacen que el estudio del tema planteado sea relevante.

Chapter 2

Primeros Conceptos

PARALELISMO/CONCURRENCIA

Un programa paralelo es aquel que es capaz de ser ejecutado en un sistema que dispone de varias unidades de procesamiento (cores o núcleos) en forma tal que el trabajo a realizar se reparte entre los procesadores disponibles para obtener los resultados del cómputo más rápidamente.

La concurrencia es una técnica para estructurar programas y que permite disponer de más de un hilo de ejecución corriendo en forma "simultánea". Cuando haya algún procesador disponible y algún hilo esté listo, el procesador podrá ejecutar ese hilo. De esta forma lo que logramos es sincronizar y comunicar procesos entre si, pero no es el objetivo de la concurrencia el lograr aumentos de la performance.

El objetivo del paralelismo es lograr un aumento en la eficiencia, el de la concurrencia es lograr una estructura en un programa tal que permita la interacción con diferentes usuarios (como p.ej. entre un servidor web y sus clientes).

En nuestro caso, existen dos modelos de programación a considerar: determinístico y no determinístico. Un modelo de programación determinístico es aquel en el cual siempre se obtiene el mismo resultado. En uno no determinístico, los resultados dependerán del orden de ejecución de cada caso particular. Los modelos de programación concurrentes son necesariamente no determinísticos porque dependen de la interacción de varios agentes y del orden en el que lo hacen. Para el caso de paralelismo, lo ideal es utilizar los modelos determinísticos, siempre que sea posible.

Chapter 3

Por qué usar Paralelismo y Haskell

MOTIVACIÓN
Desde hace ya varios años los esfuerzos de la industria de fabricación de computadoras por producir procesadores más rápidos y eficaces han variado su punto de enfoque. Lo que en un comienzo fue un intento de lograr cada vez mayor integración a menor escala, se transformó en el hecho de intentar (y lograr) ubicar en un solo chip más de un procesador. Hoy en día, el estandar del mercado es ofrecer computadoras con 2 núcleos de procesamiento por chip, y en muchos casos este número puede llegar a ser 4 u 8, sin que esto implique un aumento significativo de los costos. Si consideramos además, las enormes capacidades de cómputo que ofrecen los procesadores gráficos actuales (GPU y tarjetas gráficas) vemos que tenemos a nuestra disposición una gran cantidad de recursos computacionales muy poderosos, que en principio, deberían ayudarnos como usuarios a aumentar la performance de nuestros programas y a obtener mayores rendimientos de una computadora. Sin embargo, por regla general, estos recursos permanecen muchas veces desaprovechados o el rendimiento que se obtiene de ellos es inferior al que cabría esperar. El motivo principal para este despericio de la capacidad de cómputo, es que el desarrollo de software que haga uso real y efectivo de más de una unidad de procesamiento en forma simultánea es una tarea muy ardua y se requiere que el programador tenga mucha capacitación y experiencia en el tema. Con el fin de proveer a los desarrolladores de programas para este tipo de sistemas se han desarrollado numerosas herramientas de software. El objetivo para este trabajo es explorar las posibilidades de la programación funcional.

HASKELL

El lenguaje de programación Haskell es un lenguaje que sigue el paradigma de la programación funcional. En este tipo de lenguajes el principal concepto que se maneja es el de función, en un sentido puramente matemático. Es decir, al momento de evaluar una función lo que importa son los parámetros de entrada de esa función para la obtención de un resultado. Y ese resultado, en todos los casos, siempre será el mismo para los mismos parámetros. Un programa funcional se compone, normalmente, de una función principal y de varias funciones auxiliares. Cuando se ejecuta el programa, la función principal se evalúa lo que provoca a su vez nuevas llamadas y evaluaciones de las funciones auxiliares. De esta forma y utilizando el concepto matemático de composición de funciones se pueden desarrollar programas grandes y complejos y que además, se ajustan a cualquier tipo de necesidades.
Características de Haskell:

- Lenguaje funcional
- No posee efectos laterales
- Sistema de tipos estático
- Evaluación perezosa
- Uso de mónadas para la E/S

MOTIVOS PARA UTILIZAR HASKELL PARALELO

Haskell es un lenguaje que posee transparencia referencial. Esto es, podemos sustituir una expresión cualquiera por otra de igual valor, y mantener la semántica del programa. Esta propiedad que no la encontramos en los lenguajes imperativos, implica que el lenguaje Haskell no posee efectos laterales y por lo tanto Haskell una excelente opción si buscamos obtener mejoras en la performance. Al no existir efectos laterales, no importa el orden de ejecución, y el resultado obtenido siempre será el mismo. Un programa paralelo en Haskell podrá ser ejecutado en uno o en varios hilos, pero siempre vamos a obtener el mismo resultado. Como contrapartida también deberemos tomar en consideración que en Haskell la evaluación es perezosa (*lazy evaluation*). Esto quiere decir que una expresión cualquiera no va a ser evaluada a menos que sea estrictamente necesario. Deberemos tener mucho cuidado si intentamos evaluar, en forma paralela, expresiones que contengan a su vez subexpresiones sin evaluar. Necesitamos asegurarnos de obtener todos los resultados intermedios requeridos (es decir, que todas las computaciones necesarias se llevaron a cabo y se finalizaron). De otra forma, y dado que esos procesamientos intermedios no se completaron se producirán resultados incorrectos al final. La programaci´on paralela en Haskell es determinística, ya que siempre se obtiene el mismo resultado sin importar la cantidad de procesadores utilizados. Se facilita la tarea de debug ya que es posible dejar de lado el paralelismo y usar un solo procesador, dado que no habrá cambio alguno en los resultados. Un programa paralelo en Haskell es declarativo y no maneja explícitamente conceptos como sincronización o comunicación. El programador indica dónde está el paralelismo y el runtime se encarga de los detalles de la ejecución. El principal concepto en el que debe concentrarse el

programador Haskell para obtener paralelismo, es el particionamiento: el problema deber ser dividido en varias partes que se procesarán en paralelo. El ideal sería lograr una división tal que podamos tener a todos los procesadores ocupados la mayor parte del tiempo. Este problema puede ser complejo de resolver y puede verse limitado por:

- Granularidad: el tamaño de cada tarea no puede ser muy pequeño porque tendremos mucho overhead en la creación y el manejo de todas ellas. Pero tampoco puede ser tan grande que el paralelismo obtenido no sea significativo.
- Dependencias de datos: se produce cuando una tarea depende de los resultados de otra. En este caso la ejecución debe ser necesariamente secuencial.

Chapter 4

Paralelismo en Haskell: Técnicas

MÉTODOS PARA LOGRAR PARALELISMO EN HASKELL

Haskell provee al programador de varias bibliotecas para desarrollar programas paralelos. Estas bibliotecas se deben importar en los módulos que las quieran utilizar y son las que implementan las diferentes funciones que permiten crear paralelismo dependiendo de cada enfoque particular. Tenemos a nuestra disposición las siguientes bibliotecas:

1. Eval Monad
2. Strategy
3. Par Monad
4. Repa
5. Accelerate (para GPU's)

Para este proyecto se tomaron en cuenta Eval Monad, Strategy y Par Monad.
1) Eval Monad El módulo *Control.Parallel.Strategies* nos provee de dos operaciones para lograr paralelismo : *rpar* y *rseq*. La semántica de rpar es: "mi argumento puede ser evaluado en paralelo". La de rseq es: "evaluar mi argumento y esperar por el resultado". La función runEval es la encargada de hacer el procesamiento y de retornar el resultado:

```
data Eval a
instance Monad Eval

runEval :: Eval a -> a
rpar :: a -> Eval a
rseq :: a -> Eval a
```

Para ejemplificar las diferentes formas para su uso supondremos que deseamos ejecutar dos funciones f y g en forma paralela.
Caso 1: rpar/rpar

```
runEval $ do
a <- rpar (f x)
b <- rpar (g y)
return (a,b)
```

En este caso, la ejecución de f y g se realiza en forma paralela y el retorno es inmediato.
Caso 2: rpar/rseq

```
runEval $ do
a <- rpar (f x)
b <- rseq (g y)
return (a,b)
```

La ejecución de f y g se realiza también en forma paralela, pero el retorno se produce luego de que finaliza la ejecución de g.
Caso 3: rpar/rseq/rseq

```
runEval $ do
a <- rpar (f x)
b <- rseq (g y)
rseq a
return (a,b)
```

Ejecución en paralelo de f y g, pero en este caso el retorno se da luego de que f y g sean evaluadas.
Caso 4: rpar/rpar/rseq/rseq

```
runEval $ do
a <- rpar (f x)
b <- rpar (g y)
rseq a
rseq b
return (a,b)
```

Este caso es similar al caso anterior, se espera a la finalización de f y g antes del retorno.

Un concepto relevante en el contexto de Eval Monad es el de *spark*. Se denomina *spark* al argumento pasado a la función rpar. Los sparks se colocan en un pool de tamaño fijo, por el runtime de Haskell. Son retirados de ese pool cuando hay un procesador disponible, utilizando una técnica llamada *work stealing*. Los sparks pueden ser ejecutados en algún momento futuro, si es que hay algún procesador libre. Pero, también podría suceder que no lleguen a ejecutarse nunca o que queden fuera del pool por falta de espacio. La creación y manejo de sparks es muy eficiente ya que alcanza con tener un puntero a su ubicación en un array.

2) Evaluation Strategies

Es un medio para lograr modularizar código paralelo que separa el algoritmo del paralelismo. Permite lograr paralelismo de diferentes formas, cambiando la

estrategia (*Strategy*) utilizada. Es una función de la mónada Eval que toma un valor de tipo a y retorna el mismo valor. Para utilizarlo deberemos impotar el módulo *Control.Parallel.Strategies*.

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

La idea base para el uso de *Stategy* es que se le pase una estructura de datos como entrada, la recorra creando paralelismo con rpar y rseq y se retorne el mismo valor.

Como ejemplo del uso de Strategy consideramos el caso de la evaluación en paralelo de un par.

```
parPair :: Strategy (a,b)
parPair (a,b) = do
a' <- rpar a
b' <- rpar b
return (a',b')
```

Vemos que este método es análogo al caso 1 del punto anterior (*rpar/rpar*), la diferencia es que ahora lo tenemos empaquetado en una Strategy. A partir de un par de entrada, creamos el paralelismo con rpar y luego devolvemos la misma estructura de datos (par). Para su uso podemos utilizar *runEval* como anteriormente:

```
runEval (parPair (f, g))
```

Por comodidad y facilidad de uso, normalmente se utiliza una función llamada *using* para ejecutar la Strategy, de este modo:

```
using :: a -> Strategy a -> a
x `using` s = runEval (s x)
```

La función *using* toma un valor de tipo a y una *Strategy* para *a* como entrada, y aplica esa estrategia al valor. Lo común es que se utilice en forma infija:

```
(f, g) `using` parPair
```

El motivo para esto es que una Strategy devuelve el mismo valor que le fue pasado, y por lo tanto el código anterior es equivalente a:

```
(f, g)
```

De esta forma hemos desacoplado lo que el código hace (el par), del código que agrega paralelismo *(`using` parPair)*.

3) Par Monad

En este modelo el programador tiene la responsabilidad de ser más explicíto al momento de decidir el tipo de granularidad y las dependencias de datos, pero en cambio logra tener un mayor control. A diferencia de los casos anteriores, *Par Monad* no se basa en disponer de una estructura de datos y de las dependencias que genera la lazy evaluation. Esto facilita el debug y la evaluación de nuestro código, ya que la evaluación perezosa puede hacer dificultosa la tarea de encontrar el origen de posibles errores y de fallas de performance. De cualquier manera, con *Par Monad* se mantiene el determinismo que necesitamos para lograr paralelismo. *Par Monad* está basado en una mónada, llamada Par:

```
newtype Par a
instance Applicative Par
instance Monad Par
runPar :: Par a -> a
```

Para que Par pueda crear paralelismo necesitamos crear tareas paralelas. Para esto usamos:

```
fork :: Par () -> Par ()
```

El argumento pasado al *fork* (el hijo) se ejecuta en paralelo con el llamador del *fork* (el padre). Como esta función no retorna ningún valor al padre, se utilizan variables de un tipo especial para lograr la devolver los resultados. Estas variables son de tipo *IVar*.

```
data IVar a
-- instance Eq
new :: Par (IVar a)
put :: NFData a => IVar a -> a -> Par ()
get :: IVar a -> Par a
```

Con la función put podemos guardar un valor en una *IVar*, y con get podemos recuperarlo. Si al momento de ejecutar el get, la *IVar* está vacía, la tarea espera hasta que se haga un llamado a put. Una vez que se coloca un valor en una *IVar*, ese valor se mantiene y es un error intentar cambiarlo. El get es de solo lectura y no modifica el valor guardado ni lo remueve.

A modo de ejemplo veremos el uso de *Par Monad* utilizando la función *Fibonacci*:

```
runPar $ do
i <- new
j <- new
fork (put i (fib n))
fork (put j (fib m))
a <- get i
b <- get j
return (a+b)
```

En primer lugar declaramos a i y j de tipo *IVar*. Luego creamos dos tareas paralelas que calculen Fibonacci y almacenamos los resultados de esas computaciones en las *IVars* ya definidas. El padre espera por los resultados llamando a get. Una vez obtenidos los resultados, se suman y se devuelve esa suma.

Chapter 5

Inicio del Proyecto: Primeros Pasos

INICIOS del PROYECTO
(Repaso de conceptos de programación funcional e introducción al paralelismo)

La primera actividad desarrollada para este proyecto fue estudiar y repasar conocimientos de programación funcional. Conjuntamente con este repaso, se comenzó con el estudio en particular del paralelismo en Haskell. Luego de la etapa de estudio, y como forma de comenzar con el tema del proyecto, utilizamos el producto de matrices para investigar las posibilidades de paralelismo que nos ofrece Haskell. Las operaciones con matrices de grandes dimensiones consumen gran cantidad de recursos (procesador y memoria) y pueden demorar bastante tiempo en completarse. La idea inicial de que estas operaciones podrían computarse utilizando más de un procesador a la vez para mejorar la performance, fue la motivación para su elección.

OPERACIONES CON MATRICES
El producto de dos matrices A y B se define para el caso en que

$$A = (a_{ij})_{m \times n} \tag{5.1}$$

y

$$B = (b_{ij})_{n \times p} \tag{5.2}$$

como:

$$C = AB = (c_{ij})_{m \times p} = \sum_{k=1}^{n} a_{ik} \times b_{kj} \tag{5.3}$$

Vemos que para obtener cada posicion de la matriz C tenemos que calcular la suma de n productos, y en total son necesarias $m \times p$ sumas. Ya que se requieren tantas operaciones, pensamos en investigar si era posible mejorar la performance utilizando más de un procesador para realizar los cálculos. La implementación en Haskell de una matriz se hizo como una lista de vectores, y una lista de vectores se representó como una lista de enteros. Es decir, en nuestra implementación, una matriz es una lista de listas de enteros. Esta

representación es tal vez, la más simple posible, pero nos permite concentrarnos en el paralelismo de las operaciones con matrices sin tener que diseñar algoritmos muy complejos. En una primera instancia se debió tomar la decisión acerca de qué representaba cada lista de vectores, si era una fila o una columna de una matriz. Se optó por considerar que cada vector era una fila y se implementaron diferentes técnicas de paralelismo a partir de esta elección. Luego, y para tener algunos elementos más para realizar comparaciones entre los diferentes métodos implementados, también se consideró el caso de que un vector representa una columna de una matriz. Para cada uno de estos casos, se implementó el producto de matrices sin ningún tipo de paralelismo, para tener un punto de referencia con el cual hacer comparaciones. Luego se usaron algunas de las técnicas ya mencionadas anteriormente para añadir paralelismo y realizar las pruebas.
En primer lugar se hizo la definición del tipo Vector como una lista de enteros:

```
-- tipo para los vectores
type VectorD = [Int]
```

A partir de esta definición se implementaron operaciones auxiliares para el cálculo de matrices.

```
-- calcula el producto de dos vectores densos SIN paralelismo
-- pre: vectores del mismo tamaño
prod_VxV :: VectorD -> VectorD -> Int
prod_VxV v1 v2 = sum(zipWith (*) v1 v2)

-- calcula el producto vectorial de 2 vectores densos utilizando Eval
-- se calcula el prod vectorial para la primera y para la segunda mitad de los
-- dos vectores dados y luego se suman los resultados parciales
-- pre: los vectores son del mismo tamaño y además ese valor es un numero par
prodPar_VxV_div2 :: VectorD -> VectorD -> Eval Int
prodPar_VxV_div2 v1 v2 = do
a <- rpar(prod_VxV (take ((largoV v1) `div` 2) v1) (take ((largoV v2) `div` 2) v2))
b <- rpar(prod_VxV (drop ((largoV v1) `div` 2) v1) (drop ((largoV v2) `div` 2) v2))
rseq a
rseq b
return (a + b)

-- calcula el producto vectorial de dos vectores densos utilizando Eval
-- se asume que se pasan como parámetros dos vectores partidos por la mitad
-- v = v1 ++ v2 y w = w1 ++ w2
prodPar_VxV :: VectorD -> VectorD -> VectorD -> VectorD -> Eval Int
prodPar_VxV v1 v2 w1 w2 = do
a <- rpar (prod_VxV v1 w1)
b <- rpar (prod_VxV v2 w2)
rseq a
rseq b
return (a + b)

-- versión paralela de zipWith
parZipWith :: (a -> b-> c) -> [a] -> [b] -> Eval [c]
parZipWith f [] _ = return []
```

```
parZipWith f _ [] = return []
parZipWith f (x:xs) (y:ys) = do
w <- rpar (f x y)
ws <- parZipWith f xs ys
return (w:ws)

-- versión de zipWith con Strategy
-- parList :: Strategy a -> Strategy [a]
parStratZipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
parStratZipWith f xs ys = zipWith f xs ys `using` parList rpar

-- producto de 2 vectores usando la versión de zipWith con Strategy
prod_VxV_Strat :: VectorD -> VectorD -> Int
prod_VxV_Strat v1 v2 = sum $ parStratZipWith (*) v1 v2

-- version de map con Strategy
stratMap :: (a -> b) -> [a] -> [b]
stratMap f xs = map f xs `using` parList rseq
```

A partir del tipo Vector se construyó el tipo Matriz como una lista de Vectores, es decir, una lista de listas de enteros:

```
-- tipo para las matrices
-- cant filas: largo de [VectorD]
-- cant columnas: largo de los VectorD
type MatrizD = [VectorD]
```

Para las primeras pruebas se consideró que una Matriz venía dada por filas, esto es, la primera de las listas de la Matriz corresponde a su primera fila, y así sucesivamente. En un segundo intento grupo de pruebas se consideró que la Matriz venía dada por columnas.

```
-- tipo para las matrices
-- cant filas: largo de [VectorD]
-- cant columnas: largo de los VectorD
type MatrizD = [VectorD]
```

Basados en las operaciones sobre Vectores se implementaron las operaciones sobre Matrices. A partir del producto de dos Vectores se implementó el producto Matriz por Vector, y basado en este producto se implementó el producto Matriz por Matriz.

```
-- producto de dos matrices SIN paralelismo
-- pre: m1 es (m x p) y m2 es (p x n)
-- el producto resultante es (m x n)
prod_MxM :: MatrizD -> MatrizD -> MatrizD
prod_MxM m1 m2 = transpose $ map (prod_MxV m1) (transpose m2)

-- producto paralelo de 2 matrices
prodPar_MxM :: MatrizD -> MatrizD -> [[Eval Int]]
prodPar_MxM m1 m2 = do
```

```
map (prodPar_MxV m1) (transpose m2)

-- version paralela de map
paraMap :: (a -> b) -> [a] -> Eval [b]
paraMap f [] = return []
paraMap f (x:xs) = do
y <- rpar (f x)
ys <- paraMap f xs
return (y:ys)

-- producto matriz * vector usando map paralelo
prodPar_MxV_paraMap :: MatrizD -> VectorD -> Eval [Int]
prodPar_MxV_paraMap m v = do
paraMap (prod_VxV v) m

-- producto matriz * matriz usando map paralelo
prodPar_MxM_paraMap :: MatrizD -> MatrizD -> Eval [VectorD]
prodPar_MxM_paraMap m1 m2 = do
paraMap (prod_MxV m1) (transpose m2)

prodStrat_MxM :: MatrizD -> MatrizD -> MatrizD
prodStrat_MxM m1 m2 = transpose $ stratMap (prod_MxV m1) (transpose m2)
```