Final Programación Funcional

Strubolini Diego Martín

14 de julio de 2016

1. Introducción

En el presente trabajo, se analizan distintas operaciones realizadas utilizando el modelo *MapReduce*. Dichas operaciones pertenecen al dominio de las películas y se utilizan datos reales obtenidos del sitio IMDb en formato *JSON*. El lenguaje de programación elegido es *Haskell* y la librería elegida para la implementación del mencionado modelo es *Haskell-MapReduce*.

Dicha implementación cuenta con la posibilidad de tener paralelismo al realizar las operaciones. Es por ello que se realizaron distintas pruebas con el objetivo de analizar el impacto de dicha característica.

2. MapReduce

2.1. Principio Básicos

El modelo de programación *MapReduce* fue presentado por trabajadores de la empresa Google en el año 2008 [1] como una solución sencilla a los procesamientos de gran escala dónde se requiere paralelismo y distribución.

El modelo comprende dos funciones claves:

La función *map* o *mapper* la cual procesa una entrada con clave/valor y produce una clave de salida y una valor intermedio:

```
map (in_key, in_value) -> list(out_key, intermediate_value)
```

La función *reduce* o *reducer* la cual combina todos los valores intermedios para una clave particular y produce un conjunto de valores agrupados:

```
reduce (out_key, list(intermediate_value)) -> list(out_value)
```

Dichas funciones están inspiradas en primitivas similares en los lenguajes de programación de tipo LISP.

2.2. Haskell MapReduce

Este será el módulo a utilizar para el presente trabajo, presentado por Julian Porter en el año 2011 [2] dónde propuso una mónada para *MapReduce*.

Dicho módulo provee una interfaz de abstracción por sobre dicha mónada lo que permite al usuario utilizar *MapReduce* sin necesidad de conocer la implementación interna.

3. Módulos

3.1. IMDbMovie

En este módulo se incluyen los tipos y los métodos que representan el dominio de una película. Para representar dicho tipo, se utilizó una estructura de registro con el objetivo de

poder relacionar fácilmente los datos de los archivos *JSON* con un tipo definido en Haskell de la siguiente manera:

```
data Movie = Movie { title       :: String
                   , year        :: String
                   , director    :: String
                   , genre       :: String
                   , imdbRating  :: String
                   , imdbVotes   :: String
                   , product     :: String
                   , poster      :: String
                   , actors      :: String
                   } deriving (Generic)
```

Se puede ver que todas los campos son de tipo *String*, cuándo, por ejemplo, el año de una película tendría sentido que sea de algún tipo numérico. Es por esto, que se agregaron métodos en este módulo con el objetivo de obtener los datos de una película en un formato más conveniente. Por ejemplo la función para obtener la cantidad de votos de una película:

```
imdbVotesNum :: Movie -> Int
imdbVotesNum movie = if (imdbVotes movie) == "N/A"
                     then 0
                     else read (replace "," "" (imdbVotes movie)) :: Int
```

También en este módulo se encuentran algunas operaciones simples sobre listas de películas, dónde se aprovechan varias funciones del lenguaje de manera que se expresen de forma sencilla. Un ejemplo de esto, es la función obtener la película más votada de una lista de ellas.

```
topRated :: [Movie] -> [Movie]
topRated xs = filter (\m-> imdbVotesNum m == topRatings) xs
              where topRatings = (maximum (map imdbVotesNum xs))
```

3.2. JSONMovie

El objetivo de este módulo es el de usar la librería `Data.Aeson` para poder leer los archivos en formato *JSON* obtenidos del sitio `IMDb`. Para poder realizar esto, se tuvieron que preprocesar los archivos con los datos de las películas por algunas restricciones del lenguaje.

El principal problema fue que los campos dentro del tipo `Movie` (mostrado en la sección anterior), no pueden empezar con letra mayúscula, ya que sino es tomado como otro tipo. Es por esto que a todos los campos utilizados del archivo se les pasó la primer letra a minúscula.

El segundo problema fue el tipo de cada elemento de la lista en el archivo. Dentro de los archivos hay información no sólo de películas, sino también de series, documentales, etc. Dicho campo se llamaba *type*, la cuál es otra es de las palabras reservadas de el lenguaje, por lo tanto se decidió cambiar su nombre a *product*.

3.3. MapReduceOperations

Una vez conseguida la manera de representar la estructura de datos de una película y de cargarla con datos a partir de un archivo, ya se puede empezar a realizar operaciones sobre ella utilizando el modelo de MapReduce. Dentro de este módulo, se encuentran todas las operaciones que se implementaron sobre una lista de películas utilizando el modelo previamente mencionado.

Para que la creación de dichos métodos sea más sencilla, se creó una función auxiliar llamada `MapReduce` la cual utilizarán todos los métodos como interfaz para acceder a los métodos provistos por la librería `Parallel.MapReduce.Simple`.

```
mapReduce :: (Eq c, Binary a, NFData b, NFData c, NFData d1, NFData d2)
                  => Int -> [a] -> ([a] -> [(b, c)]) -> ([b] -> [(d1, d2)])
                     -> [(d1,d2)]
mapReduce n state mapper reducer = run mr state
                                   where mr = distribute n >>=
                                              lift mapper >>=
                                              lift reducer
```

Este método provee una capa de abstracción sobre el funcionamiento de dicha librería ya que logra que lo único que el desarrollador debe implementar es la función *mapper* y *reducer*, además de especificar la cantidad de mappers paralelos a utilizar.

Se puede destacar que la mayoría de los parámetros de dicha función son de tipo Binary o NFData, los cuales son necesarios ya que se cuenta con paralelismo. Dichos parámetros deben ser serializables con el objetivo de poder ser enviados entre distintos hilos de ejecución. También es entendible que el parámetro `c` tenga una función de igualdad ya que es necesaria para poder agrupar los resultados de los mappers en base a una misma clave.

Una vez definida dicha función, se implementaron las siguientes operaciones:

- **topRatedMovie:** Películas más votadas agrupadas por el año de estreno.
- **popularDirectors:** Directores cuya popularidad proviene de la suma de los votos de sus películas.
- **actorsCouple:** Películas en las cuales trabajaron juntos dos actores.
- **feticheActors:** Actores que trabajaron más veces junto a un director.

Algunas de dichas operaciones son más complejas que otras, pero tienen en común que, al utilizar el modelo MapReduce, son paralelizables.

3.4. Main

Módulo principal encargado de obtener los comandos ingresados por el usuario, llamar a las operaciones correspondientes e imprimir su resultado por consola. De acuerdo a la operación elegida, puede recibir parámetros adicionales con el objetivo de brindar una mayor flexibilidad al usuario para realizar operaciones en base a sus necesidades. Por ejemplo para las siguientes operaciones se puede especificar:

- **topRatedMovie:** Películas mejor calificadas mayores a un año
- **popularDirectors:** Los directores cuya popularidad supere determinado valor
- **actorsCouple:** Actores que trabajaron juntos en N películas o más

3.4.1. Compilación y Ejecución

Para compilar el proyecto se deben compilar todos los módulos de la siguiente manera:

```
$ ghc Main.hs IMDbMovie.hs JSONMovie.hs MapReduceOperations.hs -XDeriveGeneric
```

El `-XDeriveGeneric` es necesario para poder derivar tipos genéricos. Una vez generado el ejecutable se lo puede utilizar de la siguiente manera:

```
$ ./Main <json_path> <operation_num> [operation_param] [num_mappers]
```

- **json_path:** Path al archivo JSON.

- **operation_num:** Número correspondiente a la operación que se desea realizar.
- **operation_param:** Parámetro adicional para las operaciones
- **num_mappers:** Cantidad de mappers a utilizar

4. Paralelismo

Para permitir que un programa escrito en haskell genere threads de ejecución se lo debe compilar con el flag `-threaded`. De esta manera, se habilita la generación de threads cuándo se utilizan estructuras de datos como *parMap* del módulo `Control.Parallel.Strategies`. En el presente trabajo, el paralelismo es utilizado con el objetivo de distribuir las películas en distintos mappers con el objetivo de que actúen sobre distintos núcleos en la mayor medida posible.

Una vez logrado eso, se pueden ver los siguientes resultados obtenidos del tiempo de ejecución de algunas de las operaciones implementadas para distinta cantidad de núcleos y distinta cantidad de películas.

4.1. Resultados

Para cada operación se realizaron 5 corridas y se realizó el promedio del tiempo que demoró desde que se lo corre hasta que imprime el resultado. Para obtener dicho valor se utilizó el comando `time` nativo de los sistemas *BSD*. Las pruebas se realizaron utilizando de 1 a 4 cores o núcleos con el objetivo de comparar la performance para cada caso.

A continuación se muestran los resultados de las pruebas realizadas:

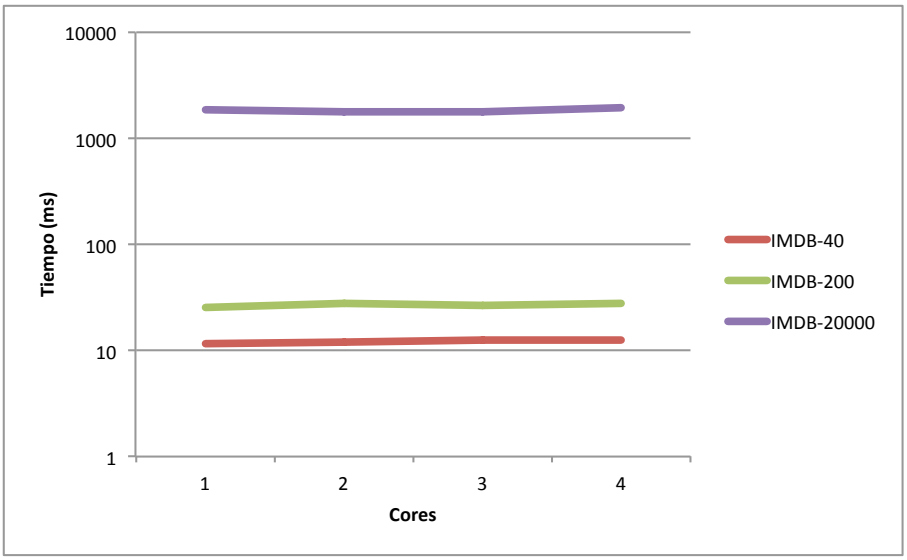

Figura 1: Tiempo de ejecución del programa para la operación `topRated` para archivos de 40, 200 y 20000 películas

En la Figura 1 se puede ver dicha operación no fue afectada en gran medida por la utilización de varios cores. Esto se debe a que la operación `topRated` requiere la menor cantidad de procesamiento de las operaciones implementadas. Por lo tanto, al agregar más núcleos, se indica que la información debe ser distribuida entre ellos y esto también afecta la performance. Al ser la operación que requiere menos procesamiento con respecto a las demás, es posible que no fuese necesario distribuir la información para la cantidad de registros de películas utilizada.

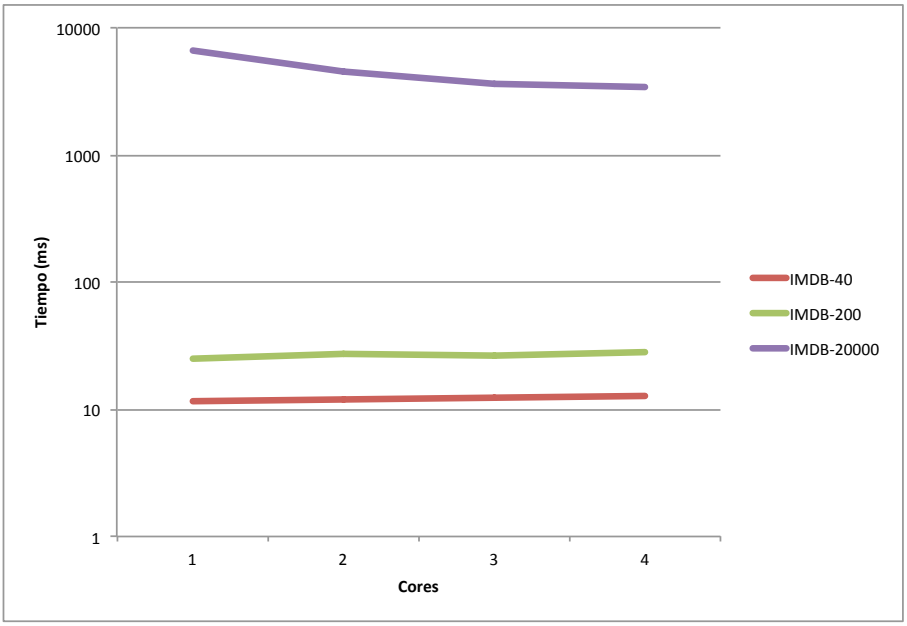

Figura 2: Tiempo de ejecución del programa para la operación `popularDirectors` para archivos de 40, 200 y 20000 películas

En la Figura 2 se puede ver un ejemplo donde para una gran cantidad de datos (20000 películas) se mejora la performance al distribuir el procesamiento entre distintos cores. Se puede ver que a medida que aumentan la cantidad de cores utilizados, se mejora el tiempo que demora el programa en devolver un resultado. Para los archivos con pocas películas, sucede lo mismo que en la operación anterior, es decir, no vale la pena distribuir dicho procesamiento si no se cuenta con una gran cantidad de datos.

5. Conclusión

El modelo *MapReduce* resulta de gran utilidad cuándo se requiere analizar una gran cantidad de datos. Con solo aportar dos funciones, se pueden realizar operaciones que, aprovechando la notación del lenguaje, logran escribirse de manera sencilla y además obtener resultados relevantes del dominio que se desea analizar.

Si se cuenta con un sistema distribuido, se puede ver una mejora en el tiempo de procesamiento cuándo se utilizan gran cantidad de datos a los cuales se les quiere realizar operaciones complejas.

6. Código Fuente

6.1. IMDbMovie.hs

```
module IMDbMovie where

import Data.Binary
import Data.String.Utils
import Data.List
import Control.DeepSeq
import GHC.Generics

data Movie =
  Movie { title   :: String
        , year    :: String
        , director    :: String
        , genre    :: String
        , imdbRating    :: String
        , imdbVotes    :: String
        , product    :: String
        , poster    :: String
        , actors    :: String
        } deriving (Generic)

type Actor = String
type Director = String

instance Binary Movie
instance NFData Movie

instance Show Movie where
  show movie = show (title movie)

-- Operations over a movie

imdbVotesNum :: Movie -> Int
imdbVotesNum movie = if (imdbVotes movie) == "N/A" then 0 else read (replace "," "" (imdbVotes
    movie)) :: Int

yearNum :: Movie -> Int
yearNum movie = if (year movie) == "N/A" then 0 else read (year movie) :: Int

actorsList :: Movie -> [Actor]
actorsList movie = if (actors movie) == "N/A" then [] else split ", " (actors movie)

directorsList :: Movie -> [Director]
directorsList movie = if (director movie) == "N/A" then [] else split ", " (director movie)

isMovie :: Movie -> Bool
isMovie movie = (IMDbMovie.product movie) == "movie"

actorsCombination :: Movie -> [(Actor, Actor)]
actorsCombination movie = map orderActor [(x,y) | (x:ys) <- tails (actorsList movie), y <- ys]

orderActor :: (Actor, Actor) -> (Actor, Actor)
orderActor (a1, a2) = if a1 < a2 then (a1,a2) else (a2,a1)

-- Operations over movie list

topRated :: [Movie] -> [Movie]
topRated xs = filter (\m-> imdbVotesNum m == topRatings) xs where topRatings = (maximum (map
    imdbVotesNum xs))

totalIMDbVotes :: [Movie] -> Int
totalIMDbVotes xs = sum (map imdbVotesNum xs)
```

6.2. MapReduceOperations.hs

```
module MapReduceOperations where

import Parallel.MapReduce.Simple (run,distribute,lift,(>>=))
import Data.Binary
import Data.Map (insertWith', empty, filter, elems, keys)
import Control.DeepSeq
import Prelude hiding ((>>=))
import IMDbMovie


mapReduce :: (Eq c, Binary a, NFData b, NFData c, NFData d1, NFData d2)
              => Int -> [a] -> ([a] -> [(b, c)]) -> ([b] -> [(d1, d2)]) -> [(d1,d2)]
mapReduce n state mapper reducer = run mr state
        where
        mr = distribute n >>= lift mapper >>= lift reducer

-- Top Rated Movies by Year (The movies with best imdbRating for each Year)

topRatedMovie :: Int -> Int -> [Movie] -> [(String, [Movie])]
topRatedMovie n year movies = mapReduce n (Prelude.filter (\m -> yearNum m >= year) movies)
    topRatedMapper topRatedReducer

topRatedMapper :: [Movie] -> [(Movie, String)]
topRatedMapper [] = []
topRatedMapper (x:xs) = [(x, year x)] ++ topRatedMapper xs

topRatedReducer :: [Movie] -> [(String, [Movie])]
topRatedReducer [] = []
topRatedReducer xs = [(year (head xs), topRated xs)]

-- Popular Directors (Directors whose total number of votes for every movie reaches a specific
    number)

popularDirectors:: Int -> Int -> [Movie] -> [(String, Int)]
popularDirectors n votes movies = Prelude.filter(\(d,v) -> v >= votes) (mapReduce n movies
    popularDirectorsMapper popularDirectorsReducer)

popularDirectorsMapper :: [Movie] -> [((Movie,Director), Director)]
popularDirectorsMapper [] = []
popularDirectorsMapper (x:xs) = map (\d -> ((x, d), d)) (directorsList x) ++ popularDirectorsMapper
     xs

popularDirectorsReducer :: [(Movie, Director)] -> [(String, Int)]
popularDirectorsReducer [] = []
popularDirectorsReducer xs = [((snd (head xs)), totalIMDbVotes (map fst xs))]

-- Actors Couples (All the movies every couple of actors has worked in)

actorsCouple:: Int -> Int -> [Movie] -> [((Actor, Actor), [Movie])]
actorsCouple n m movies =  Prelude.filter (\((a1,a2), ms) -> length ms >= m) (mapReduce n movies
    actorsCoupleMapper actorsCoupleReducer)

actorsCoupleMapper :: [Movie] -> [((Movie, (Actor, Actor)), (Actor, Actor))]
actorsCoupleMapper [] = []
actorsCoupleMapper (x:xs) = map (\ac -> ((x, ac), ac)) (actorsCombination x) ++ actorsCoupleMapper
    xs

actorsCoupleReducer :: [(Movie, (Actor, Actor))] -> [((Actor, Actor), [Movie])]
actorsCoupleReducer [] = []
actorsCoupleReducer xs = [(snd (head xs), map fst xs)]

-- Fetiche Actors (Actors that worked the highest number of times with each director)

feticheActors:: Int -> [Movie] -> [(Director, [Actor])]
feticheActors n movies =  mapReduce n movies feticheActorsMapper feticheActorsReducer

feticheActorsMapper :: [Movie] -> [((Director, [Actor]), Director)]
feticheActorsMapper [] = []
feticheActorsMapper (x:xs) = map (\d->((d, actorsList x),d)) (directorsList x) ++
    feticheActorsMapper xs

feticheActorsReducer :: [(Director, [Actor])] -> [(Director, [Actor])]
feticheActorsReducer [] = []
feticheActorsReducer xs = [(fst (head xs), mode (concat (map snd xs)))]


-- Code from https://rosettacode.org/wiki/Averages/Mode#Haskell

mode :: (Ord a) => [a] -> [a]
mode xs = keys (Data.Map.filter (== maximum (elems counts)) counts)
  where counts = foldr (\x -> insertWith' (+) x 1) empty xs
```

6.3. JSONMovie.hs

```
module JSONMovie where

import Data.Aeson
import Data.Aeson.Encode.Pretty

import qualified Data.ByteString.Lazy.Char8 as B
import IMDbMovie

instance ToJSON Movie
instance FromJSON Movie

getJSON :: FilePath -> IO B.ByteString
getJSON jsonFile = B.readFile jsonFile

readMoviesFromJSON :: FilePath -> IO (Maybe [Movie])
readMoviesFromJSON filePath = (decode <$> getJSON filePath)
```

6.4. Main.hs

```
module Main where


import System.IO
import System.Environment (getArgs)
import IMDbMovie
import JSONMovie
import MapReduceOperations

main::IO()
main = do
        args <- getArgs
        out <- case length args of
                0 -> error "Usage: Main [filename] ([num mappers])"
                _ -> do
                        let option = case length args of
                                1 -> 1
                                _ -> read $ args!!1
                        let param = case length args of
                                1 -> -1
                                2 -> -1
                                _ -> read $ args!!2
                        let nMap = case length args of
                                1 -> 16
                                2 -> 16
                                3 -> 16
                                _ -> read $ args!!3
                        d <- readMoviesFromJSON (head args)
                        let movies = case d of
                                Nothing -> []
                                Just ms -> filter isMovie ms
                        let res = case option of
                                1 -> printTopRated (case param of
                                        -1 -> topRatedMovie nMap 0 movies
                                        _ -> topRatedMovie nMap param movies)
                                2 -> printPopularDirectors (case param of
                                        -1 -> popularDirectors nMap 0 movies
                                        _ -> popularDirectors nMap param movies)
                                3 -> printActorsCouple (case param of
                                        -1 -> actorsCouple nMap 1 movies
                                        _ -> actorsCouple nMap param movies)
                                4 -> printFeticheActors (feticheActors nMap movies)
                                _ -> []
                        return res
        putStrLn "=============================="
        putStrLn "___________Results____________"
        putStrLn "=============================="
        putStrLn out
        putStrLn "=============================="

printTopRated:: [(String, [Movie])] -> String
printTopRated = foldr (\m ms -> "[" ++ (fst m) ++ "]" ++ "\n" ++  printMovies (snd m) ++ ms) ""

printPopularDirectors:: [(String, Int)] -> String
printPopularDirectors = foldr (\(n,v) ds ->  n ++ "_:_" ++ show v  ++ "\n"  ++ ds) ""

printActorsCouple:: [((Actor, Actor), [Movie])] -> String
printActorsCouple = foldr (\((a1,a2), ms) as ->  "[" ++ a1 ++ "|" ++ a2 ++ "]" ++ "\n" ++
     printMovies ms ++ as) ""

printFeticheActors:: [(Director, [Actor])] -> String
printFeticheActors = foldr (\(d,as) fs ->  "[" ++ d ++ "]_:_" ++ printActors as ++ "\n"  ++ fs) ""

printMovies:: [Movie] -> String
printMovies =  foldr (\m ms -> show m ++ "\n" ++ ms) ""

printActors:: [Actor] -> String
printActors [] = ""
printActors [a] = a
printActors (a:as) =  a ++ "_|_" ++ printActors as
```

Referencias

[1] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[2] Julian Porter, Tomas Petricek, and Douglas M Auclair. The monad. reader issue 18. 2011.