Fundamentos de Programación Funcional y Concurrente

Paralelismo de Datos

Juan Francisco Díaz Frias

Profesor Titular (1993-hoy)
juanfco.diaz@correounivalle.edu.co
Edif. B13 - 4009

Octubre 2023

Plan

Plan

Plan

Paralelismo de datos vs Paralelismo de tareas

- Previamente, se trabajó el paralelismo de tareas:

  Una forma de paralelización que distribuye la ejecución de los procesos a través de nodos de computación.

  Y usamos las abstracciones *parallel* y *task* para escribir los programas paralelos.
  Por ejemplo, cocinar un almuerzo con entrada, plato fuerte y postre.
- Ahora, se trabajará el paralelismo de datos:

  Una forma de paralelización que distribuye los datos a través de nodos de computación.

  Por ejemplo, cocinar arroz para muchos comensales.

Idea básica

- La idea básica es muy sencilla: si tiene una tarea a realizar sobre un conjunto muy grande de datos, y la quiere hacer en paralelo:
  1. divida ese conjunto en subconjuntos,
  2. haga la misma tarea en paralelo para los subconjuntos,
  3. y combine los resultados.

  O sea, lo que se paraleliza no es la tarea, sino los datos sobre los que se hace la tarea.
- Por ejemplo, se desea inicializar un arreglo, con un valor v.

```
0  def initializeArray(xs: Array[Int])(v: Int): Unit = {
1    for (i <- (0 until xs.length).par) {
2      xs(i) = v
3    }
4  }
```

  Nótese la estructura que se usa: $(0\ until\ xs.length).par$.
- El paralelismo de datos significa que la operación *for*:
  - Parte el rango en varios subrangos disyuntos,
  - realiza la tarea para cada uno de los subrangos, y
  - una vez cada tarea haya terminado, considera terminada la tarea original

Limitaciones de la idea básica

- ¿Es correcto este *for*? ¿Es decir, el arreglo queda bien inicializado?

```
0  def initializeArray(xs: Array[Int])(v: Int): Unit = {
1    for (i <- (0 until xs.length).par) {
2      xs(i) = v
3    }
4  }
```

Si, porque cada tarea *for* paralela, escribe a diferentes posiciones de memoria

- Considere el siguiente ejemplo:

```
0  def lleneArrayPar(xs:Array[Int])(ini:Int): Unit ={
1    xs(0)=ini
2    for (i <- (1 until xs.length).par) {
3      xs(i) = xs(i - 1) + 1
4    }
5  }
```

¿Es correcto este *for*? ¿Es decir, el arreglo queda bien inicializado? ¿Es decir, si yo ejecuto el mismo código sobre un rango normal, el resultado es el mismo?

for paralelo vs *for* secuencial

- Comparemos las dos versiones de *for* corriendo en este ejemplo:

```
0  def lleneArray(xs:Array[Int])
1                (ini:Int): Unit ={
2    xs(0)=ini
3    for (i <- (1 until xs.length)) {
4      xs(i) = xs(i - 1) + 1
5    }
6  }
7  def lleneArrayPar(xs:Array[Int])
8                   (ini:Int): Unit ={
9    xs(0)=ini
10   for (i <- (1 until xs.length).par) {
11     xs(i) = xs(i - 1) + 1
12   }
13 }
14 val b = new Array[Int](10)
15 val c = new Array[Int](10)
16 lleneArray(b)(3)
17 b
18 lleneArrayPar(c)(3)
19 c
```

```
0  def lleneArray(xs: Array[Int])(ini: Int): Unit
1
2
3
4
5
6
7  def lleneArrayPar(xs: Array[Int])(ini: Int): Unit
8
9
10
11
12
13
14 val b: Array[Int] = Array(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
15 val c: Array[Int] = Array(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
16
17 val res4: Array[Int] = Array(3, 4, 5, 6, 7, 8, 9, 10, 11, 12)
18
19 val res6: Array[Int] = Array(3, 4, 5, 6, 7, 1, 1, 2, 3, 1)
```

¿Por qué ocurre ese error en el cálculo en paralelo?

Beneficios de la idea básica (1)

- Considere el siguiente cálculo del máximo de un vector:

```
0  object ProbandocoleccionesPar extends App {
1    import LogThread.log
2    import LogThread.thread
3    import scala.collection._
4    import scala.util.Random
5    import scala.collection.parallel.CollectionConverters._
6
7
8    @volatile var dummy: Any = _
9    def timed[T](body: =>T): Double = {
10     val start = System.nanoTime
11     dummy = body
12     val end = System.nanoTime
13     ((end - start) / 1000) / 1000.0
14   }
15   def warmedTimed[T](n: Int = 20)(body: =>T): Double = {
16     for (_ <- 0 until n) body
17     timed(body)
18   }
19    // Mediciones de tiempos
20     val numbers = Random.shuffle(Vector.tabulate(5000000)(i => i))
21     val seqtime = warmedTimed() { numbers.max }
22     log(s"Tiempo secuencial seqtime ms")
23     val partime = warmedTimed() { numbers.par.max }
24     log(s"Tiempo en paralelo partime ms")
25 }
```

En secuencial, 303,685*ms* y en paralelo 104,84*ms*

Beneficios de la idea básica (2)

- Calcular los números palíndromes entre 0 y 100.000:

```
println(timed( (0 until 100000).filter(x => x.toString == x.toString.reverse)))
println(timed( (0 until 100000).par.filter(x => x.toString == x.toString.reverse)))
```

En secuencial, 27,496*ms* y en paralelo 16,269*ms*

- Combinación de paralelismo con programación funcional para resolver los problemas que generan las variables mutables:

```
def intersectionSize(a: Set[Int],
                     b: Set[Int]): Int = {
  var total = 0
  for (x <- a) if (b contains x) total += 1
  total
}
def intersectionSizePar(a: ParSet[Int],
                        b: ParSet[Int]): Int = {
  var total = 0
  for (x <- a) if (b contains x) total += 1
  total
}
val a = (0 until 1000).toSet
val b = (0 until 1000 by 4).toSet
val seqres = intersectionSize(a, b)
val parres = intersectionSizePar(a.par, b.par)
log(s"Sequential result - seqres")
log(s"Parallel result - parres")
```

```
def intersectionSize(a: Set[Int],
                     b: Set[Int]): Int = {
  a.count(x => b contains x)
}
def intersectionSizePar(a: ParSet[Int],
                        b: ParSet[Int]): Int = {
  a.count(x => b contains x)
}
val a = (0 until 1000).toSet
val b = (0 until 1000 by 4).toSet
val seqres = intersectionSize(a, b)
val parres = intersectionSizePar(a.par, b.par)
log(s"Sequential result - seqres")
log(s"Parallel result - parres")
```

Ejemplo de uso de operaciones sobre datos paralelos

- Suponga que queremos contar cuántos números palíndromes hay entre 1 y un valor *tam*:

Versión secuencial

```
0  val tam=1000000
1  val palSec = (1 until tam)
2    .filter(n => n % 3 == 0)
3    .count(n => n.toString == n.toString.reverse)
4  println(palSec)
5  println(warmedTimed()((1 until tam)
6    .filter(n => n % 3 == 0)
7    .count(n => n.toString == n.toString.reverse)))
```

Versión paralela

```
0  val tam=1000000
1  val palPar = (1 until tam).par
2    .filter(n => n % 3 == 0)
3    .count(n => n.toString == n.toString.reverse)
4  println(palPar)
5  println(warmedTimed()((1 until tam).par
6    .filter(n => n % 3 == 0)
7    .count(n => n.toString == n.toString.reverse)))
```

- Desempeño:

Tam	*palSec*(*ms*)	*palPar*(*ms*)
1000	0.211	2.806
10000	0.925	2.802
100000	10.623	8.667
1000000	36.994	10.837

- ¿Todas las operaciones son paralelizables?

Plan

La suma de los elementos de un arreglo

- Suponga que tenemos un arreglo de enteros y queremos hacer la suma de sus elementos: usaremos *foldLeft* como lo hemos visto:

Versión secuencial

```
0  val tam=100
1  val random = new Random()
2  val a = Array.fill(tam){ random.nextInt(1000) }
3  def sumSec(xs: Array[Int]): Int = {
4    xs.foldLeft(0)(_ + _)
5  }
6
7  val sSec = sumSec(a)
8  println(sSec)
9  println(warmedTimed()(sumSec(a)))
```

Versión paralela

```
0  val tam=100
1  val random = new Random()
2  val a = Array.fill(tam){ random.nextInt(1000) }
3  def sumPar(xs: Array[Int]): Int = {
4    xs.par.foldLeft(0)(_ + _)
5  }
6
7  val sPar = sumPar(a)
8  println(sPar)
9  println(warmedTimed()(sumPar(a)))
```

- Desempeño:

Tam	*palSec*(*ms*)	*palPar*(*ms*)
100	0.027	0.035
1000	0.393	0.195
10000	0.334	0.386
100000	2.226	2.549
1000000	20.065	19.088
10000000	83.75	382.766

- ¿*foldLeft* es paralelizable?

¿Por qué *foldLeft* no es paralelizable?

- Revisemos el tipo de *foldLeft*:

$$foldLeft : B \to ((B, A) \to B) \to (Seq[A] \to B)$$

- Revisemos la semántica (escribiendo f de forma infija):

$$foldLeft(b)(f)(List(a_1, a_2, \ldots, a_n)) = \underbrace{(\ldots(\underbrace{\underbrace{(b\ f\ \overbrace{a_1}^{:A})}_{b_1:B}\ f\ \overbrace{a_2}^{:A})}_{b_2:B}\ldots\ f\ \overbrace{a_n}^{:A})}_{b_n:B}$$

  - De entrada conocemos b y $(a_1, a_2, \ldots, a_n)$. ¿Qué cálculo podríamos hacer en paralelo?
  - Ninguno: no se puede calcular b_n sin b_{n-1}, ni este sin b_{n-2}, ni ..., ni este sin b_1. Secuencialidad obligada

- Lo mismo sucede con *foldRight*, *reduceLeft*, *reduceRight*, *scanLeft* y *scanRight*. ¿Qué podemos hacer? Variar levemente las operaciones

Plan

Operación *fold*

- Revisemos el tipo de la operación *fold*:

$$fold : A \to ((A, A) \to A) \to (Seq[A] \to A)$$

- Revisemos la semántica (escribiendo f de forma infija):

$$fold(a)(f)(List(a_1, a_2, \ldots, a_n)) = (\ldots(\underbrace{\underbrace{(a\ f\ \overbrace{a_1}^{:A})}_{b_1:A}\ f\ \overbrace{a_2}^{:A})}_{b_2:A}\ldots\ f\ \overbrace{a_n}^{:A})}_{b_n:A}$$

  - ¿Se podrían reorganizar los paréntesis para el cálculo?

$$(\ldots((a\ f\ a_1)\ f\ a_2)\ldots\ f\ a_n) \stackrel{?}{=}$$

$$\underbrace{(\ldots((a\ f\ a_1)\ f\ a_2)\ldots\ f\ a_{n/2})}_{b_1:A}\ f\ \underbrace{(\ldots((a\ f\ a_{n/2+1})\ f\ a_{n/2+2})\ldots\ f\ a_n)}_{b_2:A}$$

  - Se podría armar un árbol de reducción y paralelizar

Casos de usos de *fold*

- Sumar los elementos de un arreglo usando *fold*:

Versión secuencial

```
0  val tam=100
1  val random = new Random()
2  val a = Array.fill(tam){ random.nextInt(1000) }
3  def sumSec(xs: Array[Int]): Int = {
4    xs.fold(0)(_ + _)
5  }
6  val sSec = sumSec(a)
7  println(sSec)
8  println(warmedTimed()(sumSec(a)))
```

Versión paralela

```
0  val tam=100
1  val random = new Random()
2  val a = Array.fill(tam){ random.nextInt(1000) }
3  def sumPar(xs: Array[Int]): Int = {
4    xs.par.fold(0)(_ + _)
5  }
6  val sPar = sumPar(a)
7  println(sPar)
8  println(warmedTimed()(sumPar(a)))
```

- Hallar el máximo de un arreglo usando *fold*:

Versión secuencial

```
0  val tam=100
1  val random = new Random()
2  val a = Array.fill(tam){ random.nextInt(1000) }
3  def maxSec(xs: Array[Int]): Int = {
4    xs.fold(math.max)(Int.MinValue)
5  }
6  val mSec = maxSec(a)
7  println(mSec)
8  println(warmedTimed()(maxSec(a)))
```

Versión paralela

```
0  val tam=100
1  val random = new Random()
2  val a = Array.fill(tam){ random.nextInt(1000) }
3  def maxPar(xs: Array[Int]): Int = {
4    xs.par.fold(math.max)(Int.MinValue)
5  }
6  val mPar = maxPar(a)
7  println(mPar)
8  println(warmedTimed()(maxPar(a)))
```

- Haga evaluaciones comparativas de desempeño. ¿Conclusiones?

Precondición para la corrección de *fold*

- Para que *fold* funcione correctamente, se necesita que:
  1. $f(a, f(b, c)) = f(f(a, b), c)$ (asociatividad)
  2. $f(a, z) == f(z, a) == a$ (elemento neutro)

  Formalmente se dice que el neutro y la operación binaria forman un *monoide*.
- La conmutatividad, en cambio, no es una propiedad importante para la corrección del *fold*. $f(a, b) == f(b, a)$ no es necesaria

Limitaciones de *fold*

- Dado un arreglo de caracteres, se desea contar cuántas vocales hay en él.
  - Una solución con *foldLeft* es:

```
0 Array("E","P","A","L").par .foldLeft(0)
1                 ((count, c) => if ((c=="A")||(c=="E")) count + 1 else count)
```

  - Si en lugar de *foldLeft* se usa *fold*, funciona?

```
0 Array("E","P","A","L").par .fold(0)
1                 ((count, c) => if ((c=="A")||(c=="E")) count + 1 else count)
```

- El tipo de *f* no es compatible con el tipo que espera *fold*, pero sí con el tipo de *foldLeft*. ¿Por qué?
- En conclusión, *foldLeft* es más expresiva que *fold*.

La operación *aggregate*

- Para resolver esta limitación se crea la operación *aggregate*:

$$def\ aggregate[B](z : B)(f : (B, A) => B, g : (B, B) => B) : B$$

  - Se distinguen la operación binaria de análisis de los elementos de la secuencia (f) de la operación de acumulación (g).
  - El ejercicio anterior se resolvería así:

```
0 Array("E","P","A","L").par.aggregate(0)
1                ((count, c) => if ((c=="A")||(c=="E")) count + 1 else count, _ + _)
```

- *aggregate* es una combinación, paralelizable, de *foldLeft* y *fold*.

Sobre la paralelización de operaciones

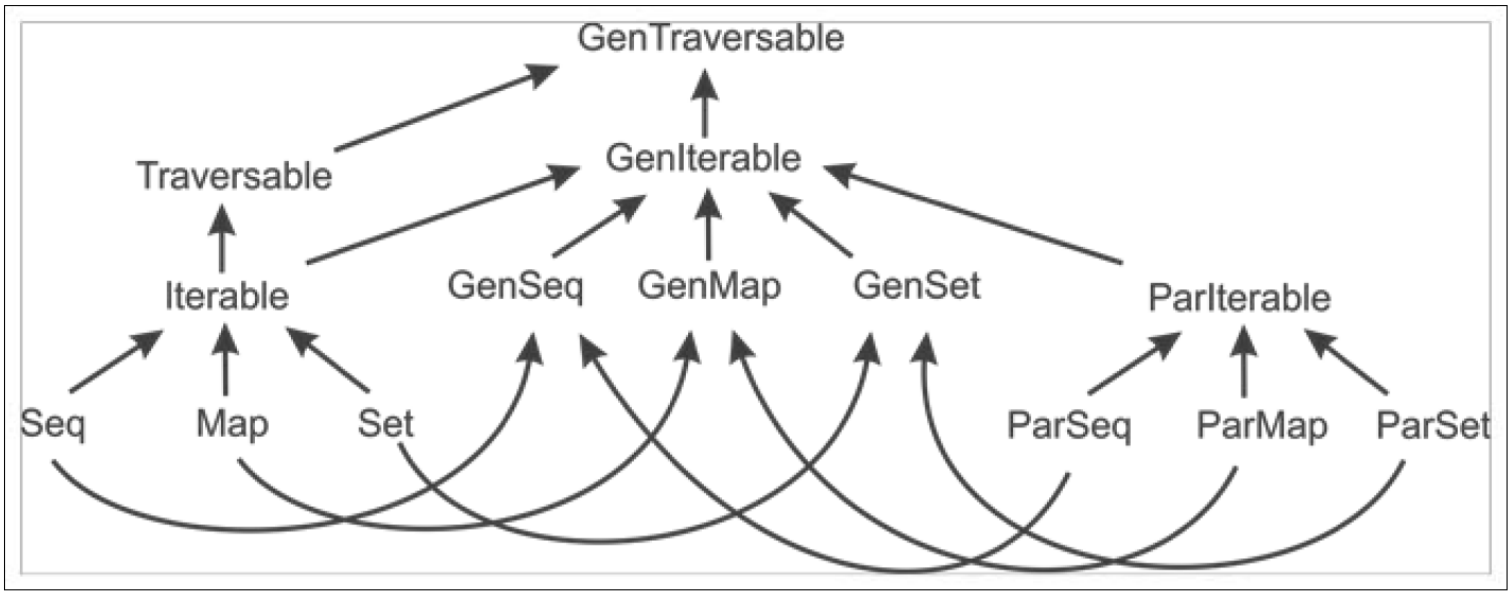
- Hasta acá, hemos analizado las funciones denominadas combinadores de acceso, que procesan la colección produciendo un valor.
- Otro tipo de funciones como *map*, *filter*, *flatMap* y *groupBy* se denominan combinadores de transformación porque procesan la colección calculando nuevas colecciones.
- Su paralelización automática necesitará de otras abstracciones más sofisticadas.

Las colecciones de Scala

- Tres tipos básicos de colecciones: secuencias (Seq), asociaciones (Map) y conjuntos (Set).
  - Las secuencias son ordenadas, y pueden ser consultadas usando un índice entero.
  - Las asociaciones almacenan parejas (*llave*, *valor*) y pueden ser consultadas a través de las llaves.
  - Los conjuntos no tienen orden y se puede consultar si un elemento hace parte o no del conjunto.
- Las colecciones pueden ser mutables o inmutables.
  - Inmutables: no pueden ser modificadas una vez han sido creadas (*List*, *Vector*, *HashTrie*)
  - Mutables: pueden ser actualizadas luego de ser creadas (*ArrayBuffer*, *HashMap*, *HashSet*)
- Las colecciones de Scala se transforman en colecciones paralelas usando el método *par*. Las operaciones sobre estas colecciones se aceleran cuando se usan múltiples procesadores simultáneamente.
  - La colección paralela comparte el mismo *dataset* que la original
  - No se copian elementos y la conversión es rápida

Plan

Jerarquía de colecciones en Scala



Scala collection hierarchy

Vale la pena leer el capítulo 24 de la referencia [2] para comprender el espíritu de las colecciones en Scala

Plan

Abstracciones para paralelismo de datos: Divisores y Combinadores

- Las colecciones secuenciales en Scala se implementan con base en las abstracciones siguientes:
  - iteradores (*iterators*)
  - constructores (*builders*)
- Las colecciones paralelas en Scala se implementan con base en las abstracciones análogas:
  - divisores (*splitters*)
  - combinadores (*combiners*)

Iteradores (*Iterators*)

- El *Iterator trait* simplificado es el siguiente:

```
0  trait Iterator[A] {
1  def next(): A
2  def hasNext: Boolean
3  }
4  def iterator: Iterator[A] // sobre cada coleccion
```

- La especificación del *iterator* es:
  - *next* se puede invocar sólo si *hasNext* devuelve *true*
  - Una vez *hasNext* devuelve *false*, en adelante sigue devolviendo *false*
- ¿Cómo se implementa *foldLeft* sobre un iterador?

```
0  def foldLeft[B](z: B)(f: (B, A) => B): B = {
1  var s = z
2  while (hasNext) s = f(s, next())
3  s
4  }
```

Constructores (*Builders*)

- El *Builder trait* simplificado es el siguiente:

```
0  trait Builder[A, Repr] {
1  def +=(elem: A): Builder[A, Repr]
2  def result: Repr
3  }
4  def newBuilder: Builder[A, Repr] // sobre cada coleccion
```

- La especificación del *Builder* es:
  - Al invocar *result* se devuelve una colección de tipo *Repr* que contiene los elementos que han sido añadidos previamente con $+=$
  - Al invocar *result* el *Builder* queda en estado *indefinido* y no se puede volver a usar
- ¿Cómo se implementa *filter* usando *newBuilder*?

```
0  def filter(p: T => Boolean): Repr = {
1  val b = newBuilder
2  for (x <- this) if (p(x)) b += x
3  b.result
4  }
```

Divisores (*Splitters*)

- El *Splitter trait* simplificado es el siguiente:

```
0  trait Splitter[A] extends Iterator[A] {
1  def split: Seq[Splitter[A]]
2  def remaining: Int
3  }
4  def splitter: Splitter[A] // sobre cada coleccion paralela
```

- La especificación del *splitter* es:
  - Una vez se invoca *split*, el divisor original queda en estado indefinido
  - Los divisores resultantes permiten recorrer subconjuntos disyuntos de la colección original
  - *remaining* devuelve el número de elementos restantes en la colección original
  - *split* debe ser un método eficiente ($\mathcal{O}(\log n)$ o mejor)
- ¿Cómo se implementa *fold* sobre un divisor?

```
0  def fold(z: A)(f: (A, A) => A): A = {
1     if (remaining < umbral) foldLeft(z)(f)
2     else {
3       val divisiones = for (division <- split) yield task { division.fold(z)(f) }
4       divisiones.map(_.join()).foldLeft(z)(f)
5     }
6  }
```

Combinadores (*Combiners*)

- El *Combiner trait* simplificado es el siguiente:

```
0  trait Combiner[A, Repr] extends Builder[A, Repr] {
1  def combine(that: Combiner[A, Repr]): Combiner[A, Repr]
2  }
3  def newCombiner: Combiner[T, Repr] // sobre cada coleccion
```

- La especificación del *Combiner* es:
  - Al invocar *combine* se crea un nuevo combinador que contiene los elementos de los dos combinadores originales
  - Al invocar *combine* los *Combiner* originales quedan en estado *indefinido* y no se pueden volver a usar
  - *combine* debe ser un método eficiente ($\mathcal{O}(\log n)$ o mejor)
- ¿Cómo se implementa *filter* en paralelo usando *splitter* y *newCombiner*?