

Fundamentos de Programación Funcional y Concurrente

Paralelismo de Tareas

Juan Francisco Díaz Frias

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



Universidad del Valle

Octubre 2023

Plan

1 Ordenamiento paralelo

2 Operaciones sobre datos

- Map
- Reduce
- Operaciones asociativas
- Scan

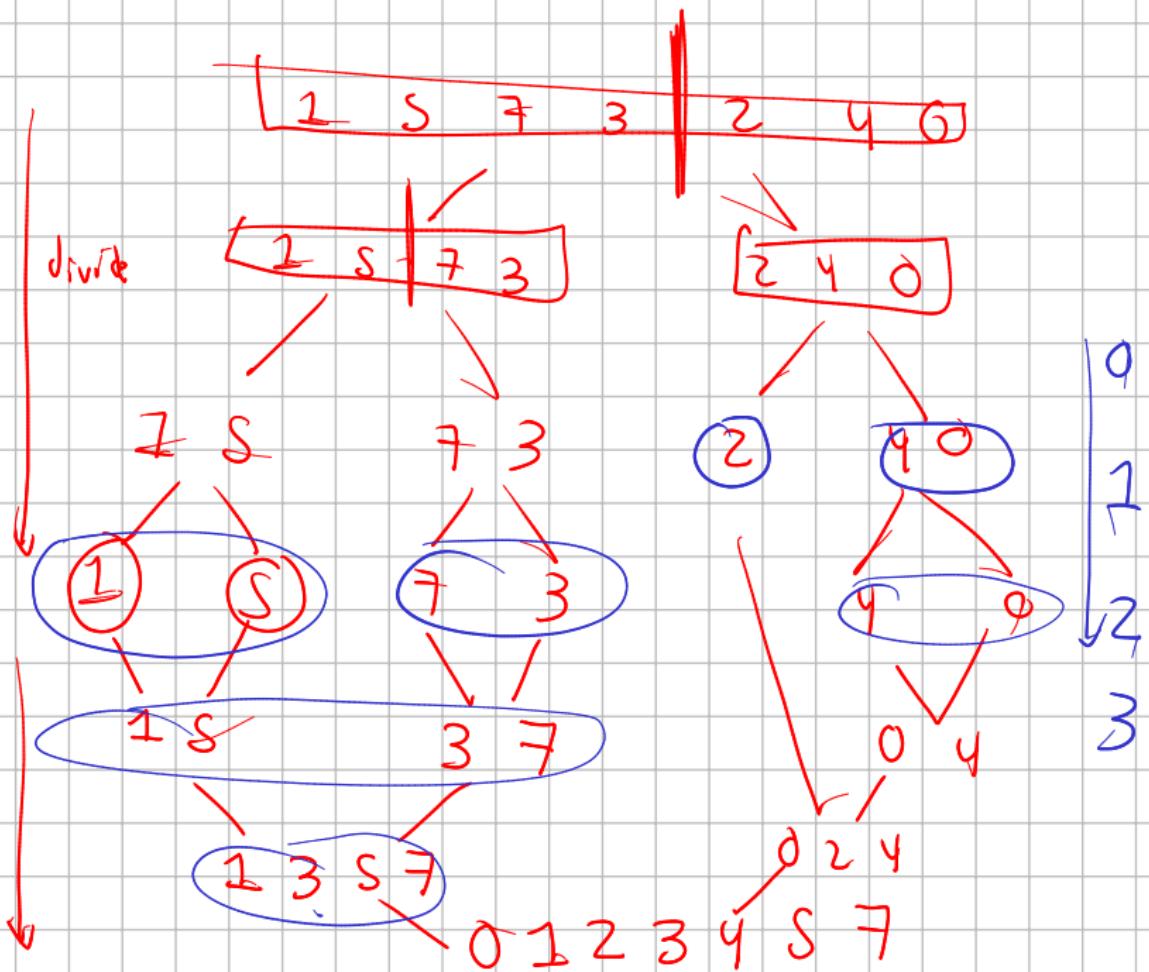
Plan

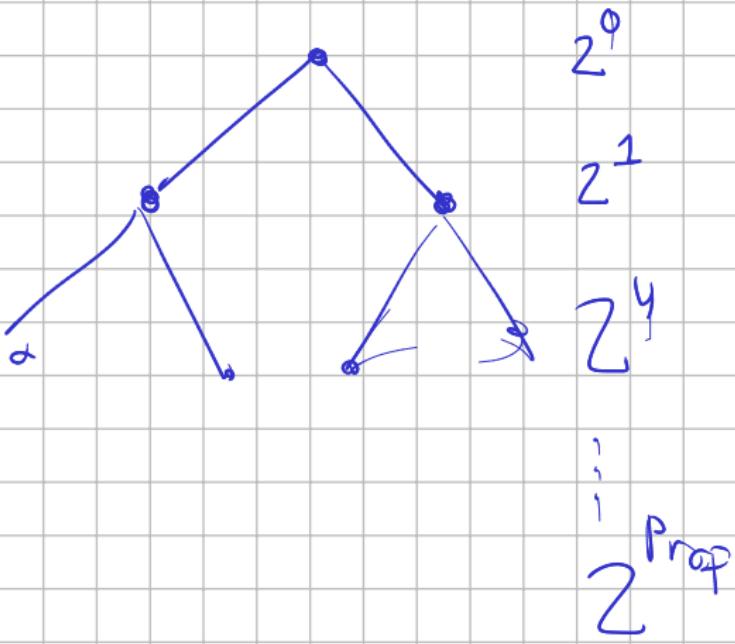
1 Ordenamiento paralelo

2 Operaciones sobre datos

- Map
- Reduce
- Operaciones asociativas
- Scan

.



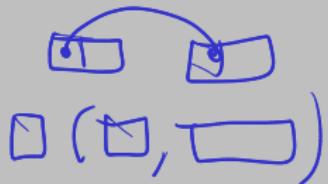


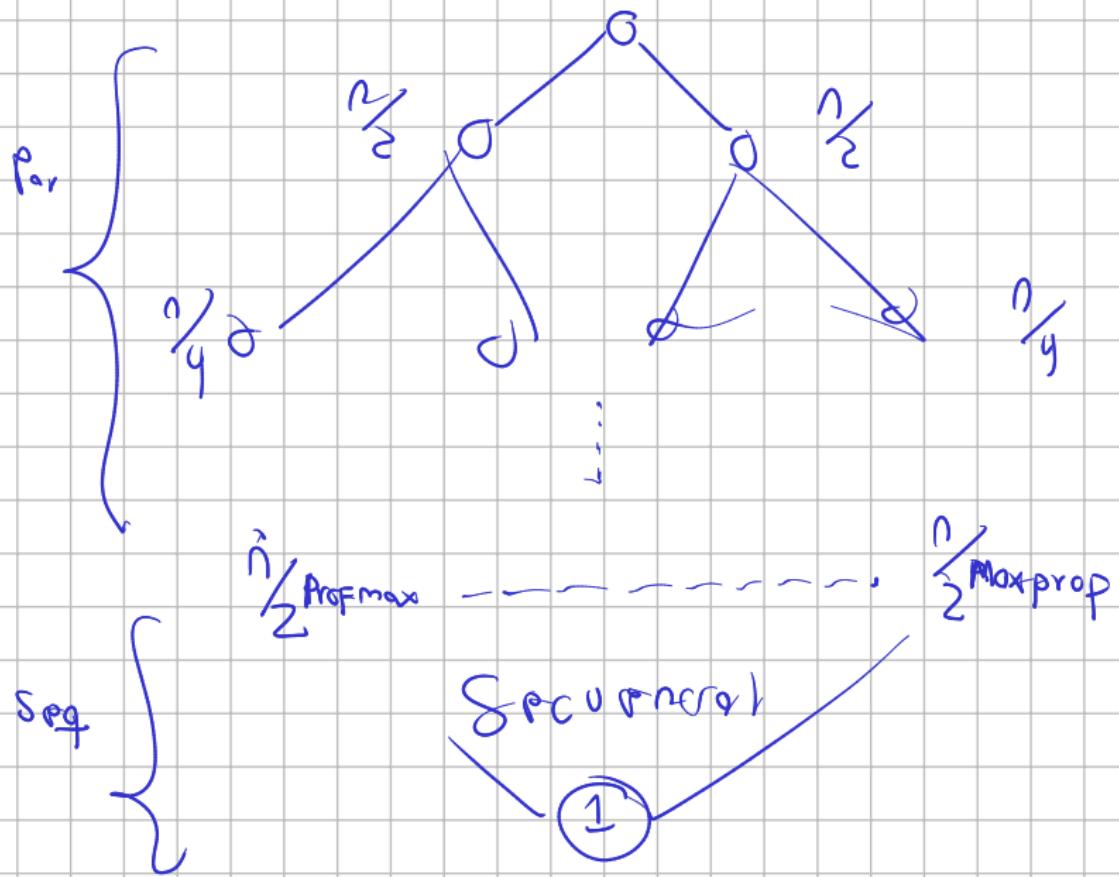
MergeSort paralelo

- Se desea implementar una versión paralela del *MergeSort*:
 - Ordenar recursivamente y en paralelo las dos mitades del arreglo
 - Mezclar secuencialmente las dos mitades del arreglo devolviendo la respuesta en un nuevo arreglo
- El método *mergeSortPar* toma un arreglo v y un nivel de profundidad del paralelismo, $maxProf$, y finaliza devolviendo un arreglo ordenado con los mismos elementos de v .
$$\text{def mergeSortPar}(\text{maxProf} : \text{Int})(\text{prof} : \text{Int})(v : \text{Vector[Int]}) : \text{Vector[Int]} = \{\dots\}$$

Una versión de *MergeSort* paralelo (1)

```
0 import common._  
1 import scala.util.Random  
2 import org.scalameter._  
3  
4 def merge(v1: Vector[Int], v2: Vector[Int]): Vector[Int] = {  
5   val n1=v1.length  
6   val n2=v2.length  
7   if (n1==0) v2 else if (n2==0) v1  
8   else if (v1(0) < v2(0)) v1(0) +: merge(v1.slice(1,n1),v2)  
9   else v2(0) +: merge(v1,v2.slice(1,n2))  
10 }  
11  
12 def mergeSortSeq(v: Vector[Int]): Vector[Int] = {  
13   val n=v.length  
14   val m=n/2  
15   if (n <= 1) v else merge(mergeSortSeq(v.slice(0,m)), mergeSortSeq(v.slice(m,n)))  
16 }  
17  
18 def mergeSortPar(maxProf: Int)(prof: Int)(v: Vector[Int]): Vector[Int] = {  
19   val n=v.length  
20   val m=n/2  
21   if (prof > maxProf) mergeSortSeq(v) else {  
22     val (v1,v2) = parallel(mergeSortPar(maxProf)(prof+1)(v.slice(0,m)), mergeSortPar(maxProf)(prof+1)(v.  
23     merge(v1,v2)  
24   }  
25 }
```





Una versión de *MergeSort* paralelo (2)

```
0 ...
1
2 val random = new Random()
3 val v = Vector.fill(100){random.nextInt(1000)}
4
5 val seq = withWarmer(new Warmer.Default) measure {
6   mergeSortSeq(v)
7 }
8
9 val par = withWarmer(new Warmer.Default) measure {
10   mergeSortPar(6)(0)(v)
11 }
12
13 (seq, par, seq.value / par.value)
```

Comparando las dos versiones, midiendo la aceleración...

Paralelismo y colecciones

- El procesamiento paralelo de colecciones es importante, pues es una de las **principales aplicaciones del paralelismo hoy**
- Vamos a estudiar las condiciones bajo las cuáles ésto se puede hacer:
 - Propiedades de las colecciones: la facilidad para dividirlas y combinarlas
 - Propiedades de las operaciones: la asociatividad y la independencia

Programación funcional y colecciones

- Las operaciones disponibles sobre las colecciones son esenciales para la programación funcional
- **map**: aplicar una función a cada elemento de la colección
 - $List(1, 3, 8).map(x \Rightarrow x * x) == List(1, 9, 64)$
- **fold**: combina elementos con una operación dada
 - $List(1, 3, 8).fold(100)((s, x) \Rightarrow s + x) == 112$
- **scan**: devuelve la lista de los resultados parciales de aplicar *fold*
 - $List(1, 3, 8).scan(100)((s, x) \Rightarrow s + x) == List(100, 101, 104, 112)$
- Estas operaciones son aún más importantes para las colecciones paralelas que para las secuenciales, gracias a que ellas encapsulan algoritmos más complejos

(100) 101, 104 (112)

Escogencia de estructuras de datos

- Nótese que se ha usado la colección **List** para especificar el resultado de las operaciones
- Las listas no son la mejor opción para implementar algoritmos paralelos debido a que es ineficiente:
 - Dividirlas en dos mitades, por ejemplo
 - Combinarlas (la concatenación necesita tiempo lineal)
- Por ello, son mejores alternativas:
 - **arreglos**: imperativo
 - **árboles**: se pueden implementar funcionalmente
- Scala ofrece una buena biblioteca de colecciones para la computación paralela: que incluye otras estructuras de datos implementadas eficientemente.

Plan

1 Ordenamiento paralelo

2 Operaciones sobre datos

- Map
- Reduce
- Operaciones asociativas
- Scan

Map: significado y propiedades

- *Map* recibe una lista y una función, y devuelve la lista de los resultados de aplicar esa función a cada elemento de la lista
 - $List(1, 3, 8).map(x \Rightarrow x * x) == List(1, 9, 64)$
 - $List(a_1, a_2, \dots, a_n).map(f) == List(f(a_1), f(a_2), \dots, f(a_n))$
- Propiedades para recordar:
 - $list.map(x \Rightarrow x) == list$
 - $list.map(f.compose(g)) == list.map(g).map(f)$
Recordar que $(f.compose(g))(x) = f(g(x))$

La implementación de *Map* sobre listas

- La definición secuencial de *Map* actual es:

```
0 def mapSeq[A,B](lst: List[A], f : A => B): List[B] = lst match {  
1   case Nil => Nil  
2   case h :: t => f(h) :: mapSeq(t,f)  
3 }
```



- La versión paralela sería:

```
0 def mapPar[A,B](lst: List[A], f : A => B): List[B] = lst match {  
1   case Nil => Nil  
2   case h :: t => {  
3     val (fh, lh) = parallel(f(h),mapPar(t,f))  
4     fh :: lh  
5   }  
6 }
```

¿Qué tan eficiente es esta versión? (ver en funcionamiento)

- Se quiere una versión paralela de *Map* tal que:
 - Los diferentes cálculos de $f(h)$ se hagan en paralelo
 - Se puedan encontrar los elementos mismos rápidamente (las listas no son una buena alternativa)
 - Funcione sobre arreglos

Map secuencial de un arreglo, devuelve un nuevo arreglo

```

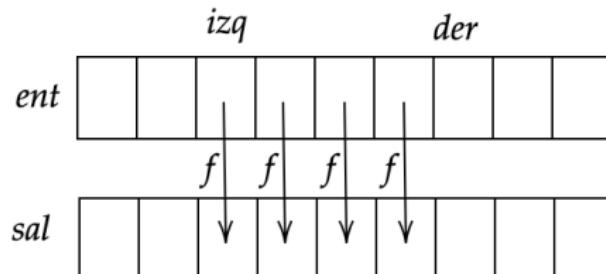
0 def mapASegSeq[A,B](ent: Array[A],
1                     izq: Int, der: Int,
2                     f : A => B,
3                     sal: Array[B]) = {
4   // Escribe en sal(i) para izq <= i <= der-1
5   var i = izq
6   while (i < der) {
7     sal(i) = f(ent(i))
8     i = i+1
9   }
10  val e = Array(2,3,4,5,6)
11  val s = Array(0,0,0,0,0)
12  val f= (x:Int) => x*x
13  mapASegSeq(e, 1, 3, f, s)
14  s

```

```

0 def mapASegSeq[A, B](ent: Array[A],
1                      izq: Int, der: Int,
2                      f: A => B,
3                      sal: Array[B]): Unit
4
5
6
7
8
9
10 val e: Array[Int] = Array(2, 3, 4, 5, 6)
11 val s: Array[Int] = Array(0, 0, 0, 0, 0)
12 val f: Int => Int = <function>
13
14 val res1: Array[Int] = Array(0, 9, 16, 0, 0)

```



Map paralelo de un arreglo, devuelve un nuevo arreglo

Map

```

0  val umbral=10000
1  def mapASegPar[A,B](ent: Array[A],
2                      izq: Int, der: Int,
3                      f : A => B,
4                      sal: Array[B]): Unit = {
5    // Escribe en sal(i) para izq <= i <= der-1
6    if (der - izq < umbral)
7      mapASegSeq(ent, izq, der, f, sal)
8    else {
9      val mid = izq + (der - izq)/2
10     parallel(mapASegPar(ent, izq, mid, f, sal),
11               mapASegPar(ent, mid, der, f, sal))
12   }
13 }
14 val e= Array(2,3,4,5,6)
15 val s= Array(0,0,0,0,0)
16 val f= (x:Int) => x*x
17 mapASegSeq(e, 1, 3, f, s)
18 s
19
20 val p: Double = 1.5
21 val e= Array(2,3,4,5,6)
22 val s1= Array(0.0,0.0,0.0,0.0,0.0)
23 val s2= Array(0.0,0.0,0.0,0.0,0.0)
24 def f(x: Int): Double = power(x, p)
25 mapASegSeq(e, 0, e.length, f, s1) // secuencial
26 s1
27 mapASegPar(e, 0, e.length, f, s2) // paralelo
28 s2

```

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

```

val umbral: Int = 10000
def mapASegPar[A, B](ent: Array[A],
                     izq: Int, der: Int,
                     f: A => B,
                     sal: Array[B]): Unit
  val e: Array[Int] = Array(2, 3, 4, 5, 6)
  val s: Array[Int] = Array(0, 0, 0, 0, 0)
  val f: Int => Int =<function>
  val res3: Array[Int] = Array(0, 9, 16, 0, 0)
  val p: Double = 1.5
  val e: Array[Int] = Array(2, 3, 4, 5, 6)
  val s1: Array[Double] = Array(0.0, 0.0, 0.0, 0.0, 0.0)
  val s2: Array[Double] = Array(0.0, 0.0, 0.0, 0.0, 0.0)
  def f(x: Int): Double
    val res5: Array[Double] = Array(2.8284271247461903,
                                    5.196152422706632, 8.0, 11.180339887498949,
                                    14.696938456699069)
    val res7: Array[Double] = Array(2.8284271247461903,
                                    5.196152422706632, 8.0, 11.180339887498949,
                                    14.696938456699069)

```

Map paralelo: evaluación comparativa con arreglos grandes

```
0 val random = new Random()
1 val e = Array.fill(1000000){ random.nextInt(1000) }
2 val s1= Array.fill(1000000){0.0}
3 val s2= Array.fill(1000000){0.0}
4 val par = withWarmer(new Warmer.Default) measure {mapASegPar(e, 0, e.length, f, s2)}
5 val seq = withWarmer(new Warmer.Default) measure {mapASegSeq(e, 0, e.length, f, s1)}
6 ...
```

```
0 val random: scala.util.Random = scala.util.Random@252f3e26
1 val e: Array[Int] = Array(958, 475, 930, 418, 38, 437, 684, 904, 195, 603, 416, 179, 119, 353, 694, 794,
2 val s1: Array[Double] = Array(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
3 val s2: Array[Double] = Array(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
4 val par: org.scalameter.Quantity[Double] = 5.931639 ms
5 val seq: org.scalameter.Quantity[Double] = 35.984994 ms
6 paralelo: 5.931639 ms
7 secuencial: 35.984994 ms
8 aceleracion: 6.066619023848215
```

Nótese que:

- Se necesita que cada hilo escriba una porción diferente del arreglo... sino ... comportamientos indeseados
- El umbral debe ser suficientemente grande para buenos desempeños...
- **La paralelización vale la pena**

Map paralelo con árboles inmutables

Considere árboles tal que:

- Las hojas almacenan segmentos de arreglos
- Los nodos que no son hojas almacenan dos subárboles

```
0 sealed abstract class Tree[A] { val size: Int }
1 case class Leaf[A](a: Array[A]) extends Tree[A] {
2   override val size = a.size
3 }
4 case class Node[A](l: Tree[A], r: Tree[A]) extends Tree[A] {
5   override val size = l.size + r.size
6 }
```

Suponemos árboles balanceados: podemos explorar en paralelo

```
0 def mapTreeSeq[A:Manifest,B:Manifest](t: Tree[A],
1   f: A => B) : Tree[B] =
2   t match {
3     case Leaf(a) => {
4       val len = a.length; val b = new Array[B](len)
5       var i=0
6       while (i < len) { b(i)= f(a(i)); i= i + 1 }
7       Leaf(b)
8     }
9     case Node(l,r) => {
10       val (lb,rb) = (mapTreeSeq(l,f),
11                      mapTreeSeq(r,f))
12       Node(lb, rb)
13     }
14 }
```

```
0 def mapTreePar[A:Manifest,B:Manifest](t: Tree[A],
1   f: A => B) : Tree[B] =
2   t match {
3     case Leaf(a) => {
4       val len = a.length; val b = new Array[B](len)
5       var i=0
6       while (i < len) { b(i)= f(a(i)); i= i + 1 }
7       Leaf(b)
8     }
9     case Node(l,r) => {
10       val (lb,rb) = parallel(mapTreePar(l,f),
11                             mapTreePar(r,f))
12       Node(lb, rb)
13     }
14 }
```

Map paralelo árboles: evaluación comparativa

```
0 val umbral=10000
1 val p: Double = 1.5
2 val random = new Random()
3 val e = Array.fill(10000000){ random.nextInt(1000) }
4 def f(x: Int): Double = power(x, p)
5
6 val par = withWarmer(new Warmer.Default) measure {
7   mapTreePar(t, f)
8 }
9
10 val seq = withWarmer(new Warmer.Default) measure {
11   mapTreeSeq(t, f)
12 }
13 ...
```

Comparación entre usar arreglos y árboles

- Usando arreglos:

- Acceso rápido a los elementos, útil con memoria compartida
- Buena "localidad" de la memoria
- Es imperativo: debe asegurarse que no se escribe la misma posición de memoria desde dos hilos distintos
- Es costoso de concatenar

- Usando árboles:

- Funcional puro, produce nuevos árboles, se guardan los antiguos.
- No toca preocuparse por escribir en la misma posición de memoria
- Es eficiente combinar dos árboles (concatenar)
- Sobrecosto alto de memoria
- Mala "localidad" de la memoria

Plan

1 Ordenamiento paralelo

2 Operaciones sobre datos

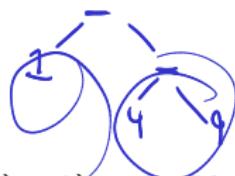
- Map
- **Reduce**
- Operaciones asociativas
- Scan

fold y *reduce*: significado

- Recordemos las funciones *fold* y *reduce*: que definimos sobre colecciones. Ellas reciben una colección y una operación binaria sobre elementos de la colección. Lo que devuelven depende de:
 - Si además reciben un elemento inicial o si sólo funcionan para colecciones no vacías
 - En qué orden se combinan las operaciones de la colección
- Por ejemplo, con la colección *List*, tenemos:
 - $(List(x_1, \dots, x_n) \text{ foldLeft } z)(op) = (\dots((z \text{ op } x_1) \text{ op } x_2) \text{ op } \dots) \text{ op } x_n$
 $(List() \text{ foldLeft } z)(op) = z$
 - $List(x_1, \dots, x_n) \text{ reduceLeft } op = (\dots((x_1 \text{ op } x_2) \text{ op } x_3) \text{ op } \dots) \text{ op } x_n$
(la lista no puede ser vacía)
 - $(List(x_1, \dots, x_n) \text{ foldRight } z)(op) = x_1 \text{ op } (\dots(x_{n-1} \text{ op } (x_n \text{ op } z)) \dots)$
 $(List() \text{ foldRight } z)(op) = z$
 - $List(x_1, \dots, x_n) \text{ reduceRight } op = x_1 \text{ op } (\dots(x_{n-2} \text{ op } (x_{n-1} \text{ op } x_n)) \dots)$

Ejemplos de aplicación de *fold* y *reduce*

- Ejemplo con listas de enteros y la operación binaria +
 - $List(1, 4, 9).foldLeft(100)((s, x) \Rightarrow s + x) == (((100 + 1) + 4) + 9) == 114$
 - $List(1, 4, 9).foldRight(100)((s, x) \Rightarrow s + x) == (1 + (4 + (9 + 100))) == 114$
 - $List(1, 4, 9).reduceLeft((s, x) \Rightarrow s + x) == ((1 + 4) + 9) == 14$
 - $List(1, 4, 9).reduceRight((s, x) \Rightarrow s + x) == 1 + (4 + 9) == 14$
- Ejemplo con listas de enteros y la operación binaria -
 - $List(1, 4, 9).foldLeft(100)((s, x) \Rightarrow s - x) == (((100 - 1) - 4) - 9) == 86$
 - $List(1, 4, 9).foldRight(100)((s, x) \Rightarrow s - x) == (1 - (4 - (9 - 100))) == -94$
 - $List(1, 4, 9).reduceLeft((s, x) \Rightarrow s - x) == ((1 - 4) - 9) == -12$
 - $List(1, 4, 9).reduceRight((s, x) \Rightarrow s - x) == 1 - (4 - 9) == 6$
- ¿Cuál es la propiedad que tiene + que no tiene - que garantiza el mismo resultado cuando se aplica *foldLeft* (*reduceLeft*) que *foldRight* (*reduceRight*): **la asociatividad**



Operaciones binarias asociativas

- Una operación binaria $f : (A, A) \Rightarrow A$ es asociativa si para cada $x, y, z \in A$:

$$f(x, f(y, z)) = f(f(x, y), z)$$

- Si escribimos $f(x, y)$ de forma infija como $x \otimes y$, la asociatividad se escribe:

$$x \otimes (y \otimes z) = (x \otimes y) \otimes z$$

- Considere dos expresiones con la misma lista de operandos conectados por \otimes pero con diferente parentización. Se puede concluir que **ambas expresiones evalúan al mismo resultado**.

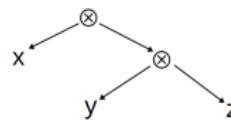
$$(x \otimes y) \otimes (z \otimes w) = x \otimes (y \otimes (z \otimes w)) = ((x \otimes y) \otimes z) \otimes w$$

Expresiones como árboles

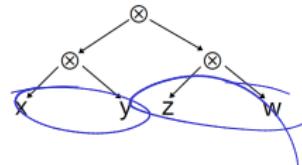
Cada expresión construida a partir de valores conectados por \otimes puede ser representada como un árbol:

- Las hojas guardan los valores
- Los nodos son el operador \otimes

$x \otimes (y \otimes z)$:



$(x \otimes y) \otimes (z \otimes w)$:



reduce en árboles de expresiones

- Creamos los árboles de valores:

```
0 sealed abstract class Tree[A]
1 case class Leaf[A](value: A) extends Tree[A]
2 case class Node[A](left: Tree[A], right: Tree[A]) extends Tree[A]
```

- Evaluar una expresión consiste en hacer *reduce* del árbol. La versión secuencial es:

```
0 def reduce[A](t: Tree[A], f : (A,A) => A): A = t match {
1   case Leaf(v) => v
2   case Node(l, r) => f(reduce[A](l, f), reduce[A](r, f)) // Node -> f
3 }
```

- Y la versión paralela sería:

```
0 def reducePar[A](t: Tree[A], f : (A,A) => A): A = t match {
1   case Leaf(v) => v
2   case Node(l, r) => {
3     val (lV, rV) = parallel(reduce[A](l, f), reduce[A](r, f))
4     f(lV, rV)
5   }
6 }
```

El *map – reduce* como mecanismo de cálculo

- Así como definimos el *reduce*, podemos definir el *map*:

```
0 def reducePar[A](t: Tree[A], f : (A,A) => A): A = t match {
1   case Leaf(v) => v
2   case Node(l, r) => {
3     val (lV, rV) = parallel(reduce[A](l, f), reduce[A](r, f))
4     f(lV, rV)
5   }
6 }
7 def map[A,B](t: Tree[A], f : A => B): Tree[B] = t match {
8   case Leaf(v) => Leaf(f(v))
9   case Node(l, r) => Node(map[A,B](l, f), map[A,B](r, f)) }
```

- Si queremos conocer la lista de valores en las hojas del árbol, en el orden en que se encuentran, ¿cómo podríamos calcularlos? Una opción sería:

```
0 def toList[A](t: Tree[A]): List[A] = t match {
1   case Leaf(v) => List(v)
2   case Node(l, r) => toList[A](l) ++ toList[A](r)
3 }
```

- Pero componiendo *map* y *reduce* quedaría:

```
0 def toList[A](t: Tree[A]): List[A] = reduce(map(t, (v:A) => List(v)),
1                                              (l1: List[A], l2: List[A]) => l1 ++ l2)
```

El *reduce* sobre arreglos

- ¿Qué hacer cuando trabajamos con colecciones donde conocemos el orden pero no la estructura de árbol?
- Por ejemplo, ¿cómo podríamos reducir arreglos?
 - Convertirlos en un árbol balanceado
 - Aplicar la reducción sobre árboles
- Gracias a la asociatividad, podríamos escoger cualquier árbol que preserve el orden de los elementos de la colección original para hacer la reducción.
- Cuando el arreglo sea suficientemente pequeño hacemos el proceso secuencialmente.

reduce paralelo sobre arreglos

```
0 def reduceSeg[A](ent: Array[A], izq: Int, der: Int, f: (A,A) => A): A = {
1   if (der - izq < limite) {
2     var res= ent(izq); var i= izq+1
3     while (i < der) { res= f(res, ent(i)); i= i+1 }
4     res
5   } else {
6     val mid = izq + (der - izq)/2
7     val (a1,a2) = parallel(reduceSeg(ent, izq, mid, f),
8                           reduceSeg(ent, mid, der, f))
9     f(a1,a2)
10    }
11  }
12 def reduceArrayPar[A](ent: Array[A], f: (A,A) => A): A =
13   reduceSeg(ent, 0, ent.length , f)
```

Plan

1 Ordenamiento paralelo

2 Operaciones sobre datos

- Map
- Reduce
- **Operaciones asociativas**
- Scan

Propiedades de operaciones binarias y corrección del *reduce*

- Una operación binaria $f : (A, A) \Rightarrow A$ es asociativa si para cada $x, y, z \in A$:

$$f(x, f(y, z)) = f(f(x, y), z)$$

- Una operación binaria $f : (A, A) \Rightarrow A$ es conmutativa si para cada $x, y \in A$:

$$f(x, y) = f(y, x)$$

- Si escribimos $f(x, y)$ de forma infija como $x \otimes y$,
 - la asociatividad se escribe:

$$x \otimes (y \otimes z) = (x \otimes y) \otimes z$$

- la conmutatividad se escribe:

$$x \otimes y = y \otimes x$$

- Existen operaciones asociativas pero no conmutativas, y conmutativas pero no asociativas.

La corrección de *reduce* depende de que la operación sea asociativa.

Operaciones asociativas y conmutativas

Muchas de las operaciones matemáticas típicas:

- Suma y multiplicación de enteros (BigInt) y de números racionales (expresados como e.g., pares de BigInts)
- Suma y multiplicación módulo un entero positivo (e.g. 232) incluyendo la aritmética usual sobre 32 y 64 bits.
- Unión, intersección y diferencia simétrica sobre conjuntos.
- Unión de *bolsas* (multiconjuntos)
- Operaciones booleanas \wedge , \vee
- Suma y multiplicación de polinomios
- Suma de vectores
- Suma de matrices de dimensión fija

Operaciones asociativas pero no conmutativas

Existen operaciones asociativas que no son conmutativas:

- Concatenación de listas (`++`)
- Concatenación de *Strings*
- Multiplicación de matrices
- Composición de relaciones
- Composición de funciones

Por ser asociativas, *reduce* funciona correctamente

Operaciones conmutativas pero no asociativas

Existen operaciones conmutativas que no son asociativas:

- La suma de punto flotante

```
0  scala> val e = 1e-200
1  val e: Double = 1.0E-200
2  scala> val x = 1e200
3  val x: Double = 1.0E200
4  scala> val mx = -x
5  val mx: Double = -1.0E200
6  scala> (x + mx) + e
7  val res0: Double = 1.0E-200
8  scala> x + (mx + e)
9  val res1: Double = 0.0
10 scala> (x + mx) + e == x + (mx + e)
11 val res2: Boolean = false
```

- La multiplicación de punto flotante

```
0  scala> e
1  val res3: Double = 1.0E-200
2  scala> x
3  val res4: Double = 1.0E200
4  scala> (e*x)*x
5  val res5: Double = 1.0E200
6  scala> e*(x*x)
7  val res6: Double = Infinity
8  scala> (e*x)*x == e*(x*x)
9  val res7: Boolean = false
```

Plan

1 Ordenamiento paralelo

2 Operaciones sobre datos

- Map
- Reduce
- Operaciones asociativas
- Scan

scanLeft: significado

- La función *scanLeft* se define igual que *map* y *reduce* sobre colecciones. Ella recibe una colección, una operación binaria asociativa sobre elementos de la colección y un elemento inicial y devuelve la lista de los *reduceLeft* de todos los prefijos de la lista original.
- Formalmente, con la colección *List*, tenemos:
$$\text{List}(x_1, \dots, x_n).\text{scanLeft}(z)(op) = \text{List}(y_0, y_1, \dots, y_n)$$
 donde:
 - $y_0 = z$
 - $y_i = f(y_{i-1}, x_i)$ para $1 \leq i \leq n$
- $\text{List}(1, 4, 9).\text{scanLeft}(100)((s, x) \Rightarrow s + x) ==$
$$\text{List}(100, 101, 105, 114)$$
- La función *scanRight* es dual y su resultado no es el mismo:
$$\text{List}(1, 4, 9).\text{scanRight}(100)((s, x) \Rightarrow s + x) ==$$

$$\text{List}(114, 113, 109, 100)$$

scanLeft secuencial sobre arreglos

- Programar *scanLeft* secuencial tal que:
 - Reciba un arreglo *ent*, un elemento *a0*, una operación binaria asociativa *f* y un arreglo *sal* para la respuesta
 - Rellenar *sal* con la respuesta, suponiendo que $sal.length = ent.length + 1$

```
0 // scanLeft secuencial
1 def scanLeft(ent: Array[Int],
2             a0: Int, f: (Int, Int) => Int,
3             sal: Array[Int]): Unit = {
4   sal(0)= a0
5   var a= a0
6   var i= 0
7   while (i < ent.length) {
8     a= f(a,ent(i))
9     i= i + 1
10    sal(i)= a
11  }
12 }
```

Paralelizando *scanLeft*

- ¿Se puede paralelizar *scanLeft*, suponiendo que f es asociativa, al punto de lograr un algoritmo mucho más rápido que el secuencial?
- A primera vista la tarea parece imposible:
 - El valor de cada elemento en la secuencia, depende del valor previo.
 - Parece necesario esperar los resultados previos, para calcular el actual.
 - Parecen necesarios n pasos antes de poder dar una respuesta, sin importar el paralelismo con que contemos.
- Idea: reutilizar resultados intermedios
 - Se hace más trabajo (más aplicaciones de f)
 - Se puede paralelizar, lo cual compensa los cálculos de más

Aproximación: expresar *scan* usando *map* y *reduce*

- Ya hemos paralelizado *map* y *reduce*. Si logramos expresar *scan* usando *map* y *reduce* tendríamos una primera versión paralela. Recordemos las interfaces de *map* y *reduce* sobre segmentos de arreglos:

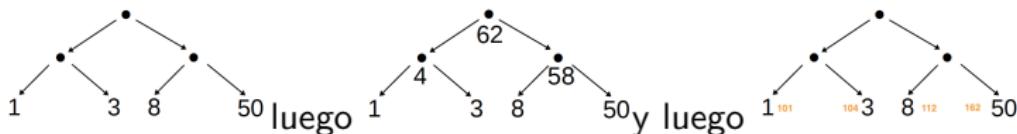
```
0 def reduceSeg1[A](ent: Array[A], izq:Int, der:Int,
1     a0: A, f: (A,A) => A,
2     ): A = { ... }
3
4 def mapSeg[A](ent: Array[A], izq: Int, der: Int, f : A => A,
5     sal: Array[A]):Unit = { ... }
```

- *scan* se podría realizar así:

```
0 def scanLeft(ent: Array[Int], a0: Int, f: (Int,Int) => Int, sal: Array[Int]) = {
1     val fi = { (i:Int) => reduceSeg1(ent, 0, i, a0, f) }
2     mapSeg(ent, 0, ent.length, fi, sal)
3     val ult = ent.length - 1
4     sal(ult + 1) = f(sal(ult), ent(ult))
5 }
```

Idea: Reutilizar resultados intermedios

- En la versión anterior no se reutilizan resultados intermedios del cálculo.
- Hay que aprovechar que *reduce* funciona aplicando las operaciones en un árbol (*¿recuerdan?*)
- Idea: guardar los resultados intermedios de esta computación paralela



Definiciones de árboles

- Los árboles que almacenan la colección sólo tienen valores en las hojas:

```
0 // scanLeft paralelo con arboles
1 sealed abstract class Tree[A]
2 case class Leaf[A](a: A) extends Tree[A]
3 case class Node[A](l: Tree[A], r: Tree[A]) extends Tree[A]
```

- Los árboles intermedios almacenan valores en las hojas y en los nodos:

```
0 // Arboles para guardar resultados intermedios
1 sealed abstract class TreeRes[A] { val res: A }
2 case class LeafRes[A](override val res: A) extends TreeRes[A]
3 case class NodeRes[A](l: TreeRes[A],
4                         override val res: A,
5                         r: TreeRes[A]) extends TreeRes[A]
```

La reducción que preserva resultados intermedios

- Versión secuencial: *reduceRes* recibe un árbol de valores y devuelve un árbol con resultados intermedios:

```
0 // Version secuencial
1 def reduceRes[A](t: Tree[A], f: (A,A) => A): TreeRes[A] = t match {
2   case Leaf(v) => LeafRes(v)
3   case Node(l, r) => {
4     val (tL, tR) = (reduceRes(l, f), reduceRes(r, f))
5     NodeRes(tL, f(tL.res, tR.res), tR)
6   }
7 }
8 val t1 = Node(Node(Leaf(1), Leaf(3)), Node(Leaf(8), Leaf(50)))
9 val suma = (x:Int, y:Int) => x+y
10 reduceRes(t1, suma)
```

- Versión paralela: *llenarSubiendo*:

```
0 // Version paralela
1 def llenarSubiendo[A](t: Tree[A], f: (A,A) => A): TreeRes[A] = t match {
2   case Leaf(v) => LeafRes(v)
3   case Node(l, r) => {
4     val (tL, tR) = parallel(llenarSubiendo(l, f), llenarSubiendo(r, f))
5     NodeRes(tL, f(tL.res, tR.res), tR)
6   }
7 }
8 val t2= llenarSubiendo(t1, suma)
```

Creando la colección final

- *llenarBajando* recorre el árbol de resultados intermedios hacia abajo, creando la colección con casi todos los resultados finales

```
0 def llenarBajando[A](t: TreeRes[A], a0: A, f : (A,A) => A): Tree[A] = t match {
1   case LeafRes(a) => Leaf(f(a0, a))
2   case NodeRes(l, _, r) => {
3     val (tL, tR) = parallel(llenarBajando[A](l, a0, f),
4       llenarBajando[A](r, f(a0, l.res), f))
5     Node(tL, tR) }
6   llenarBajando(t2, 100, suma)
```

- Añadiendo el primer elemento de la colección: *prepend*:

```
0 def prepend[A](x: A, t: Tree[A]): Tree[A] = t match {
1   case Leaf(v) => Node(Leaf(x), Leaf(v))
2   case Node(l, r) => Node(prepend(x, l), r)
3 }
```

- Poniendo todo junto: *scanLeft*

```
0 def scanLeft[A](t: Tree[A], a0: A, f: (A,A) => A): Tree[A] = {
1   val tRes = llenarSubiendo(t, f)
2   val scan1 = llenarBajando(tRes, a0, f)
3   prepend(a0, scan1)
4 }
5 scanLeft(t1, 100, suma)
```

scanLeft sobre arreglos

- La anterior definición permite entender la idea de la paralelización.
¿Podemos hacer lo mismo si la colección viene en un arreglo?
- Idea: armar un árbol cuyas hojas son segmentos disyuntos del arreglo original. En las hojas se aplica un *scanLeft* secuencial.
- La estructura de los árboles intermedios será ligeramente diferente a la estructura *TreeRes[A]* definida antes:

```
0 // Arboles para guardar resultados intermedios
1 sealed abstract class TreeResA[A] { val res: A }
2 case class Leaf[A](desde: Int, hasta: Int,
3                     override val res: A) extends TreeResA[A]
4 case class Node[A](l: TreeResA[A],
5                     override val res: A,
6                     r: TreeResA[A]) extends TreeResA[A]
```

- Ahora, *TreeResA[A]* guarda en las hojas los límites del segmento de arreglo que corresponde a ese nodo (rango: *[desde, hasta]*), así como el resultado de aplicar *reduce* en ese segmento.
- Y se tiene una referencia al arreglo de entrada *ent* original.

llenarSubiendo sobre arreglos

- *llenarSubiendo* produce el árbol de resultados intermedios de *reduce*:

```
0 def llenarSubiendo[A](ent: Array[A], desde: Int, hasta: Int,
1   f: (A,A) => A): TreeResA[A] = {
2   if (hasta - desde < limite)
3     Leaf(desde, hasta, reduceSeg1(ent, desde + 1, hasta, ent(desde), f))
4   else {
5     val mid = desde + (hasta - desde)/2
6     val (tL,tR) = parallel(llenarSubiendo(ent, desde, mid, f),
7       llenarSubiendo(ent, mid, hasta, f))
8     Node(tL, f(tL.res,tR.res), tR)
9   }
10 }
```

- Cuando el tamaño del segmento es menor que un *limite*, se usa *reduceSeg1*, una reducción secuencial.

```
0 def reduceSeg1[A](ent: Array[A], izq:Int, der:Int,
1   a0: A, f: (A,A) => A,
2   ): A = {
3   var a= a0
4   var i= izq
5   while (i < der) {
6     a= f(a,ent(i))
7     i= i + 1
8   }
9   a
10 }
```

llenarBajando sobre arreglos

- *llenarBajando* produce el árbol de resultados finales del *scan* en cada hoja.
- En el caso de una hoja, se usa *scanLeftSeg*, un *scan* secuencial del arreglo.
- Cuando es un nodo, se realizan en paralelo escaneos sobre los árboles izquierdo y derecho, con los acumulados adecuados.

```
0 def llenarBajando[A](ent: Array[A],  
1      a0: A, f: (A,A) => A,  
2      t: TreeResA[A],  
3      sal: Array[A]): Unit = t match {  
4      case Leaf(desde, hasta, res) =>  
5          scanLeftSeg(ent, desde, hasta, a0, f, sal)  
6      case Node(l, _, r) => {  
7          val (_,_) = parallel(  
8              llenarBajando(ent, a0, f, l, sal),  
9              llenarBajando(ent, f(a0,l.res), f, r, sal))  
10     }  
11 }
```

scanLeft sobre arreglos

- Sobre las hojas se aplica *scanLeftSeg*:

```
0 def scanLeftSeg[A](inp: Array[A], left: Int, right: Int,
1                     a0: A, f: (A,A) => A,
2                     out: Array[A]) = {
3   if (left < right) {
4     var i = left
5     var a = a0
6     while (i < right) {
7       a = f(a, inp(i))
8       i = i + 1
9       out(i) = a
10    }
11  }
12 }
```

- El proceso completo se define por medio de *scanLeft*:

```
0 def scanLeft[A](ent: Array[A],
1                  a0: A, f: (A,A) => A,
2                  sal: Array[A]) = {
3   val t = llenarSubiendo(ent, 0, ent.length, f)
4   llenarBajando(ent, a0, f, t, sal) // fills out[1..inp.length]
5   sal(0) = a0 // prepends a0
6 }
```

Análisis comparativo

```
0 // Pruebas del scanLeft secuencial versus el paralelo
1 // sobre un arreglo grande
2 val random = new Random()
3 val e = Array.fill(10000000){ random.nextInt(1000) }
4 val s1= Array.fill(10000001){0}
5 val s2= Array.fill(10000001){0}
6 val suma = (x:Int,y:Int) => x+y
7 val a0=100
8 s1(0) = a0
9 val seq = withWarmer(new Warmer.Default) measure {
10   scanLeftSeg(e, 0, e.length, a0, suma, s1)
11 }
12 s1
13 val par = withWarmer(new Warmer.Default) measure {
14   scanLeft(e,a0,suma,s2)
15 }
16 s2
17 println(s"paralelo:-par")
18 println(s"secuencial:-seq")
19 println(s"aceleracion:-{seq.value--par.value}")
```

```
0 // Pruebas del scanLeft secuencial versus el paralelo
1 // sobre un arreglo grande
2 val random: scala.util.Random = scala.util.Random@175b
3 val e: Array[Int] = Array(228, 734, 325, 762, 641, 646,
4 val s1: Array[Int] = Array(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
5 val s2: Array[Int] = Array(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
6 val suma: (Int, Int) => Int = <function>
7 val a0: Int = 100
8
9 val seq: org.scalameter.Quantity[Double] = 94.183197 ms
10
11
12 val res1: Array[Int] = Array(100, 328, 1062, 1387, 214
13 val par: org.scalameter.Quantity[Double] = 87.595821 ms
14
15
16 val res2: Array[Int] = Array(100, 328, 1062, 1387, 214
17 paralelo: 87.595821 ms
18 secuencial: 94.183197 ms
19 aceleracion: 1.075201943709164
```

Pruebas sobre un arreglo al azar, de 10 millones de números, con límite de 10.000. Este límite hay que mejorarla experimentalmente. Por ejemplo, cuando se puso en 1000 no hubo aceleración sino desaceleración.