Fold (Reduce) Operations

Parallel Programming in Scala

Viktor Kuncak

Functional programming and collections

We have seen operation:

map: apply function to each element

- `List(1,3,8).map(x => x*x) == List(1, 9, 64)`

We now consider:

fold: combine elements with a given operation

- `List(1,3,8).fold(100)((s,x) => s + x) == 112`

Fold: meaning and properties

Fold takes among others a binary operation, but variants differ:

- whether they take an initial element or assume non-empty list
- in which order they combine operations of collection

```
List(1,3,8).foldLeft(100)((s,x) => s - x) == ((100 - 1) - 3) - 8 == 88

List(1,3,8).foldRight(100)((s,x) => s - x) == 1 - (3 - (8-100)) == -94

List(1,3,8).reduceLeft((s,x) => s - x) == (1 - 3) - 8 == -10

List(1,3,8).reduceRight((s,x) => s - x) == 1 - (3 - 8) == 6
```

To enable parallel operations, we look at **associative** operations

- addition, string concatenation (but not minus)

Associative operation

Operation `f: (A,A) => A` is associative iff for every x, y, z:

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

If we write $f(a, b)$ in infix form as $a \otimes b$, associativity becomes

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

Consequence: consider two expressions with same list of operands connected with $\otimes$, but different parentheses. Then these expressions evaluate to the same result, for example:

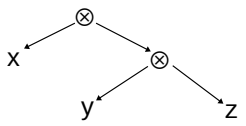
$$(x \otimes y) \otimes (z \otimes w) = (x \otimes (y \otimes z)) \otimes w = ((x \otimes y) \otimes z) \otimes w$$

Trees for expressions

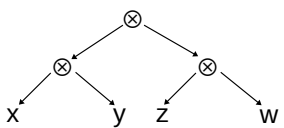
Each expression built from values connected with $\otimes$ can be represented as a tree

- leaves are the values
- nodes are $\otimes$

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



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



Folding (reducing) trees

How do we compute the value of such an expression tree?

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

Result of evaluating the expression is given by a **reduce** of this tree.

What is its (sequential) definition?

Folding (reducing) trees

How do we compute the value of such an expression tree?

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

Result of evaluating the expression is given by a **reduce** of this tree.

What is its (sequential) definition?

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

We can think of `reduce` as replacing the constructor `Node` with given `f`

Running reduce

For non-associative operation, the result depends on structure of the tree:

```
def tree = Node(Leaf(1), Node(Leaf(3), Leaf(8)))
def fMinus = (x:Int,y:Int) => x - y
def res = reduce[Int](tree, fMinus)     // 6
```

Parallel reduce of a tree

How to make that tree reduce parallel?

Parallel reduce of a tree

How to make that tree reduce parallel?

```
def reduce[A](t: Tree[A], f : (A,A) => A): A = t match {
  case Leaf(v) => v
  case Node(l, r) => {
    val (lV, rV) = parallel(reduce[A](l, f), reduce[A](r, f))
    f(lV, rV)
  }
}
```

Parallel reduce of a tree

How to make that tree reduce parallel?

```
def reduce[A](t: Tree[A], f : (A,A) => A): A = t match {
  case Leaf(v) => v
  case Node(l, r) => {
    val (lV, rV) = parallel(reduce[A](l, f), reduce[A](r, f))
    f(lV, rV)
  }
}
```

What is the depth complexity of such reduce?

Parallel reduce of a tree

How to make that tree reduce parallel?

```
def reduce[A](t: Tree[A], f : (A,A) => A): A = t match {
  case Leaf(v) => v
  case Node(l, r) => {
    val (lV, rV) = parallel(reduce[A](l, f), reduce[A](r, f))
    f(lV, rV)
  }
}
```

What is the depth complexity of such reduce?

Answer: height of the tree

Associativity stated as tree reduction

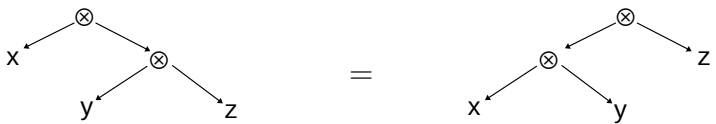
How can we restate associativity of such trees?

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

Associativity stated as tree reduction

How can we restate associativity of such trees?

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



If f denotes $\oplus$, in Scala we can write this also as:

```
reduce(Node(Leaf(x), Node(Leaf(y), Leaf(z))), f) ==
reduce(Node(Node(Leaf(x), Leaf(y)), Leaf(z)), f)
```

Order of elements in a tree

Observe: can use a list to describe the ordering of elements of a tree

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

Order of elements in a tree

Observe: can use a list to describe the ordering of elements of a tree

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

Suppose we also have tree map:

```
def map[A,B](t: Tree[A], f : A => B): Tree[B] = t match {
  case Leaf(v) => Leaf(f(v))
  case Node(l, r) => Node(map[A,B](l, f), map[A,B](r, f))  }
```

Can you express `toList` using `map` and `reduce`?

Order of elements in a tree

Observe: can use a list to describe the ordering of elements of a tree

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

Suppose we also have tree map:

```
def map[A,B](t: Tree[A], f : A => B): Tree[B] = t match {
  case Leaf(v) => Leaf(f(v))
  case Node(l, r) => Node(map[A,B](l, f), map[A,B](r, f))  }
```

Can you express `toList` using `map` and `reduce`?

```
toList(t) == reduce(map(t, List(_)), _ ++ _)
```

Consequence stated as tree reduction

Consequence of associativity: consider two expressions with same list of operands connected with $\otimes$, but different parentheses. Then these expressions evaluate to the same result.

Express this consequence in Scala using functions we have defined so far.

Consequence stated as tree reduction

Consequence of associativity: consider two expressions with same list of operands connected with $\otimes$, but different parentheses. Then these expressions evaluate to the same result.

Express this consequence in Scala using functions we have defined so far.

Consequence (Scala): if `f : (A,A)=>A` is associative, `t1:Tree[A]` and `t2:Tree[A]` and if `toList(t1)==toList(t2)`, then:

```
reduce(t1, f)==reduce(t2, f)
```

Consequence stated as tree reduction

Consequence of associativity: consider two expressions with same list of operands connected with $\otimes$, but different parentheses. Then these expressions evaluate to the same result.

Express this consequence in Scala using functions we have defined so far.

Consequence (Scala): if `f : (A,A)=>A` is associative, `t1:Tree[A]` and `t2:Tree[A]` and if `toList(t1)==toList(t2)`, then:

```
reduce(t1, f)==reduce(t2, f)
```

Can we prove that this fact follows from associativity?

Explanation of the consequence

Intuition: given a tree, use tree rotation until it becomes list-like.

Associativity law says tree rotation preserves the result:

Example use:

Applying rotation to tree preserves `toList` as well as the value of `reduce`.
`toList(t1)==toList(t2)` $\Rightarrow$ rotations can bring `t1`,`t2` to same tree

Towards a reduction for arrays

We have seen reduction on trees.

Often we work with collections where we only know the ordering and not the tree structure.

How can we do reduction in case of, e.g., arrays?

- convert it into a balanced tree
- do tree reduction

Because of associativity, we can choose any tree that preserves the order of elements of the original collection

Tree reduction replaces `Node` constructor with `f`, so we can just use `f` directly instead of building tree nodes.

When the segment is small, it is faster to process it sequentially

Parallel array reduce

```
def reduceSeg[A](inp: Array[A], left: Int, right: Int, f: (A,A) => A): A = {
  if (right - left < threshold) {
    var res= inp(left); var i= left+1
    while (i < right) { res= f(res, inp(i)); i= i+1 }
    res
  } else {
    val mid = left + (right - left)/2
    val (a1,a2) = parallel(reduceSeg(inp, left, mid, f),
                           reduceSeg(inp, mid, right, f))
    f(a1,a2)
  }
}
def reduce[A](inp: Array[A], f: (A,A) => A): A =
  reduceSeg(inp, 0, inp.length, f)
```

End of Slide Deck