Parallel Scan Left

Parallel Programming in Scala

Viktor Kuncak

Parallel scan

Having seen parallel map and parallel fold

map: apply function to each element

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

fold: combine elements with a given operation

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

we now examine parallel scanLeft:

scanLeft: list of the folds of all list prefixes

`List(1,3,8).scanLeft(100)((s,x) => s + x) == List(100, 101, 104, 112)`

scanLeft: meaning and properties

```
List(1,3,8).scanLeft(100)(_ + _) == List(100, 101, 104, 112)
```

```
List(a1, a2, a3).scanLeft(f)(a0) = List(b0, b1, b2, b3)
```

where

- `b0 = a0`
- `b1 = f(b0, a1)`
- `b2 = f(b1, a2)`
- `b3 = f(b2, a3)`

We assume that `f` is assocative, throughout this segment.

scanLeft: meaning and properties

```
List(1,3,8).scanLeft(100)(_ + _) == List(100, 101, 104, 112)
```

```
List(a1, a2, a3).scanLeft(f)(a0) = List(b0, b1, b2, b3)
```

where

- `b0 = a0`
- `b1 = f(b0, a1)`
- `b2 = f(b1, a2)`
- `b3 = f(b2, a3)`

We assume that `f` is assocative, throughout this segment.

`scanRight` is different from `scanLeft`, even if `f` is associative

```
List(1,3,8).scanRight(100)(_ + _) == List(112, 111, 108, 100)
```

We consider only `scanLeft`, but `scanRight` is dual.

Sequential Scan

$$List(a_1, a_2, ..., a_N).scanLeft(f)(a_0) = List(b_0, b_1, b_2, ..., b_N)$$

where $b_0 = a_0$ and $b_i = f(b_{i-1}, a_i)$ for $1 \leq i \leq N$.

Sequential Scan

$$List(a_1, a_2, ..., a_N).scanLeft(f)(a_0) = List(b_0, b_1, b_2, ..., b_N)$$

where $b_0 = a_0$ and $b_i = f(b_{i-1}, a_i)$ for $1 \leq i \leq N$.

Give a sequential definition of `scanLeft`:

- take an array `inp`, an element `a0`, and binary operation `f`
- write the output to array `out`, assuming `out.length >= inp.length + 1`

```
def scanLeft[A](inp: Array[A],
                a0: A, f: (A,A) => A,
                out: Array[A]): Unit
```

Sequential Scan Solution

```
def scanLeft[A](inp: Array[A],
                a0: A, f: (A,A) => A,
                out: Array[A]): Unit = {
  out(0)= a0
  var a= a0
  var i= 0
  while (i < inp.length) {
    a= f(a,inp(i))
    i= i + 1
    out(i)= a
  }
}
```

Making scan parallel

Can `scanLeft` be made parallel? Assume that `f` is associative.

Goal: an algorithm that runs in $O(\log n)$ given infinite parallelism

Making scan parallel

Can `scanLeft` be made parallel? Assume that `f` is associative.

Goal: an algorithm that runs in $O(\log n)$ given infinite parallelism

At first, the task seems impossible; it seems that:

- the value of the last element in sequence depends on all previous ones
- need to wait on all previous partial results to be computed first
- such approach gives $O(n)$ even with infinite parallelism

Making scan parallel

Can `scanLeft` be made parallel? Assume that `f` is associative.

Goal: an algorithm that runs in $O(\log n)$ given infinite parallelism

At first, the task seems impossible; it seems that:

- the value of the last element in sequence depends on all previous ones
- need to wait on all previous partial results to be computed first
- such approach gives $O(n)$ even with infinite parallelism

Idea: give up on reusing all intermediate results

- do more work (more `f` applications)
- improve parallelism, more than compensate for recomputation

High-level approach: express scan using map and reduce

Can you define result of `scanLeft` using `map` and `reduce`?

High-level approach: express scan using map and reduce

Can you define result of `scanLeft` using `map` and `reduce`?

Assume input is given in array `inp` and that you have `reduceSeg1` and `mapSeg` functions on array segments:

```
def reduceSeg1[A](inp: Array[A], left: Int, right: Int,
                  a0: Int, f: (A,A) => A): A

def mapSeg[A,B](inp: Array[A], left: Int, right: Int,
                fi : (Int,A) => B,
                out: Array[B]): Unit
```

High-Level Solution

According to definition, element on position *i* is the reduce of the previous elements.

We thus map the array with a function defined using reduce:

```
def scanLeft[A](inp: Array[A], a0: A, f: (A,A) => A, out: Array[A]) = {
  val fi = { (i:Int,v:A) => reduceSeg1(inp, 0, i, a0, f) }
  mapSeg(inp, 0, inp.length, fi, out)
  val last = inp.length - 1
  out(last + 1) = f(out(last), inp(last))
}
```

Map always gives as many elements as the input, so we additionally compute the last element.

Reusing intermediate results of reduce

In the previous solution we do not reuse any computation.

Can we reuse some of it?

Recall that `reduce` proceeds by applying the operations in a tree

Idea: save the intermediate results of this parallel computation.

Reusing intermediate results of reduce

In the previous solution we do not reuse any computation.

Can we reuse some of it?

Recall that `reduce` proceeds by applying the operations in a tree

Idea: save the intermediate results of this parallel computation.

We first assume that input collectio is also (another) tree.

Tree definitions

Trees storing our input collection only have values in leaves:

```scala
sealed abstract class Tree[A]
case class Leaf[A](a: A) extends Tree[A]
case class Node[A](l: Tree[A], r: Tree[A]) extends Tree[A]
```

Trees storing intermediate values also have (`res`) values in nodes:

```scala
sealed abstract class TreeRes[A] { val res: A }
case class LeafRes[A](override val res: A) extends TreeRes[A]
case class NodeRes[A](l: TreeRes[A],
                      override val res: A,
                      r: TreeRes[A]) extends TreeRes[A]
```

Tree definitions

Trees storing our input collection only have values in leaves:

```
sealed abstract class Tree[A]
case class Leaf[A](a: A) extends Tree[A]
case class Node[A](l: Tree[A], r: Tree[A]) extends Tree[A]
```

Trees storing intermediate values also have (`res`) values in nodes:

```
sealed abstract class TreeRes[A] { val res: A }
case class LeafRes[A](override val res: A) extends TreeRes[A]
case class NodeRes[A](l: TreeRes[A],
                      override val res: A,
                      r: TreeRes[A]) extends TreeRes[A]
```

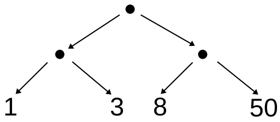
Can you define `reduceRes` function that transforms `Tree` into `TreeRes`?

Reduce that preserves the computation tree

```
def reduceRes[A](t: Tree[A], f: (A,A) => A): TreeRes[A]
```

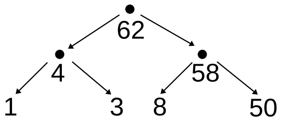
Reduce that preserves the computation tree

```
def reduceRes[A](t: Tree[A], f: (A,A) => A): TreeRes[A] = t match {
  case Leaf(v) => LeafRes(v)
  case Node(l, r) => {
    val (tL, tR) = (reduceRes(l, f), reduceRes(r, f))
    NodeRes(tL, f(tL.res, tR.res), tR)
  }
}
```



Reduce that preserves the computation tree

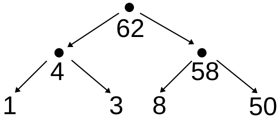
```scala
def reduceRes[A](t: Tree[A], f: (A,A) => A): TreeRes[A] = t match {
  case Leaf(v) => LeafRes(v)
  case Node(l, r) => {
    val (tL, tR) = (reduceRes(l, f), reduceRes(r, f))
    NodeRes(tL, f(tL.res, tR.res), tR)
  }
}
val t1 = Node(Node(Leaf(1), Leaf(3)), Node(Leaf(8), Leaf(50)))
val plus = (x:Int,y:Int) => x+y
scala> reduceRes(t1, plus)
res0: TreeRes[Int] = NodeRes(NodeRes(LeafRes(1),4,LeafRes(3)),
                             62,
                             NodeRes(LeafRes(8),58,LeafRes(50)))
```



Parallel reduce that preserves the computation tree (upsweep)

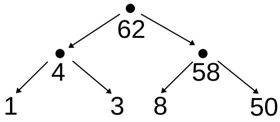
```
def upsweep[A](t: Tree[A], f: (A,A) => A): TreeRes[A] = t match {
  case Leaf(v) => LeafRes(v)
  case Node(l, r) => {
    val (tL, tR) = parallel(upsweep(l, f), upsweep(r, f))
    NodeRes(tL, f(tL.res, tR.res), tR)
  }
}
```

Using tree with results to create the final collection



Next: a tree for 100, 101, 104, 112, 162

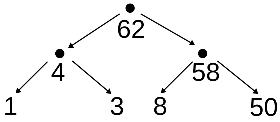
Using tree with results to create the final collection



Next: a tree for 100, 101, 104, 112, 162

```scala
// 'a0' is reduce of all elements left of the tree 't'
def downsweep[A](t: TreeRes[A], a0: A, f : (A,A) => A): Tree[A] = t match {
  case LeafRes(a) => Leaf(f(a0, a))
  case NodeRes(l, _, r) => {
    val (tL, tR) = parallel(downsweep[A](l, a0, f),
                            downsweep[A](r, f(a0, l.res), f))
    Node(tL, tR) } }
```

Using tree with results to create the final collection



Next: a tree for 100, 101, 104, 112, 162

```
// 'a0' is reduce of all elements left of the tree 't'
def downsweep[A](t: TreeRes[A], a0: A, f : (A,A) => A): Tree[A] = t match {
  case LeafRes(a) => Leaf(f(a0, a))
  case NodeRes(l, _, r) => {
    val (tL, tR) = parallel(downsweep[A](l, a0, f),
                            downsweep[A](r, f(a0, l.res), f))
    Node(tL, tR) } }

scala> downsweep(res0, 100, plus)
res1: Tree[Int] = Node(Node(Leaf(101),Leaf(104)),Node(Leaf(112),Leaf(162)))
```

scanLeft on trees

```
def scanLeft[A](t: Tree[A], a0: A, f: (A,A) => A): Tree[A] = {
  val tRes = upsweep(t, f)
  val scan1 = downsweep(tRes, a0, f)
  prepend(a0, scan1)
}
```

scanLeft on trees

```
def scanLeft[A](t: Tree[A], a0: A, f: (A,A) => A): Tree[A] = {
  val tRes = upsweep(t, f)
  val scan1 = downsweep(tRes, a0, f)
  prepend(a0, scan1)
}
```

Define `prepend`.

scanLeft on trees

```scala
def scanLeft[A](t: Tree[A], a0: A, f: (A,A) => A): Tree[A] = {
  val tRes = upsweep(t, f)
  val scan1 = downsweep(tRes, a0, f)
  prepend(a0, scan1)
}
```

Define `prepend`.

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

scanLeft and arrays

Previous definition on trees is good for understanding

As with map and reduce, to make it more efficient, we use trees that have arrays in leaves instead of individual elements.

scanLeft and arrays

Previous definition on trees is good for understanding

As with map and reduce, to make it more efficient, we use trees that have arrays in leaves instead of individual elements.

Exercise: define `scanLeft` on trees with such large leaves, using sequential scan left in the leaves.

scanLeft and arrays

Previous definition on trees is good for understanding

As with map and reduce, to make it more efficient, we use trees that have arrays in leaves instead of individual elements.

Exercise: define `scanLeft` on trees with such large leaves, using sequential scan left in the leaves.

Next step: parallel scan when the entire collection is an array

- we will still need to construct the intermediate tree

Intermediate tree for array reduce

```
sealed abstract class TreeResA[A] { val res: A }
case class Leaf[A](from: Int, to: Int,
                   override val res: A) extends TreeResA[A]
case class Node[A](l: TreeResA[A],
                   override val res: A,
                   r: TreeResA[A]) extends TreeResA[A]
```

The only difference compared to previous TreeRes: each Leaf now keeps track of the array segment range (from, to) from which `res` is computed.

We do not keep track of the array elements in the Leaf itself; we instead pass around a reference to the input array.

Upsweep on array

Starts from an array, produces a tree

```
def upsweep[A](inp: Array[A], from: Int, to: Int,
               f: (A,A) => A): TreeResA[A] = {
  if (to - from < threshold)
    Leaf(from, to, reduceSeg1(inp, from + 1, to, inp(from), f))
  else {
    val mid = from + (to - from)/2
    val (tL,tR) = parallel(upsweep(inp, from, mid, f),
                           upsweep(inp, mid, to, f))
    Node(tL, f(tL.res,tR.res), tR)
  }
}
```

Sequential reduce for segment

```scala
def reduceSeg1[A](inp: Array[A], left: Int, right: Int,
                  a0: A, f: (A,A) => A): A = {
  var a= a0
  var i= left
  while (i < right) {
    a= f(a, inp(i))
    i= i+1
  }
  a
}
```

Downsweep on array

```
def downsweep[A](inp: Array[A],
                 a0: A, f: (A,A) => A,
                 t: TreeResA[A],
                 out: Array[A]): Unit = t match {
    case Leaf(from, to, res) =>
      scanLeftSeg(inp, from, to, a0, f, out)
    case Node(l, _, r) => {
      val (_,_) = parallel(
        downsweep(inp, a0, f, l, out),
        downsweep(inp, f(a0,l.res), f, r, out))
    }
  }
```

Sequential scan left on segment

Writes to output shifted by one.

```
def scanLeftSeg[A](inp: Array[A], left: Int, right: Int,
                   a0: A, f: (A,A) => A,
                   out: Array[A]) = {
  if (left < right) {
    var i= left
    var a= a0
    while (i < right) {
      a= f(a,inp(i))
      i= i+1
      out(i)=a
    }
  }
}
```

Finally: parallel scan on the array

```scala
def scanLeft[A](inp: Array[A],
                a0: A, f: (A,A) => A,
                out: Array[A]) = {
  val t = upsweep(inp, 0, inp.length, f)
  downsweep(inp, a0, f, t, out) // fills out[1..inp.length]
  out(0)= a0 // prepends a0
}
```

End of Slide Deck