Lecture 2
Parallel Map, Fold, and Scan

Maps

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

```
scala> mapSeq(List(2,3,4), (x:Int) => x*x)
res1: List[Int] = List(4, 9, 16)
```

List traversal is not parallelizable: we need $O(n)$ span to reach n-th element (even with ∞ threads).

- makes sense to parallelize only for expensive operation f

Alternatives to Lists

Two simple alternatives:

- arrays
  - imperative: must be careful that parallel tasks write to disjoint parts of the array
  - on a shared memory machine, easy to dynamically partition (just pass around indices for array segments)
  - ideal memory locality
  - hard to construct from pieces
- immutable trees
  - purely functional, produce new trees
  - no need to worry about disjointness of writes by parallel tasks
  - easier to combine
  - can have high memory allocation overhead
  - can have bad locality

We start with arrays.

Map on Arrays: Sequential Map on Segment

```
def mapASegSeq[A,B](inp: Array[A], left: Int, right: Int,
                    f : A => B,
                    out: Array[B]) = {
  // Writes to out(i) for left <= i <= right-1
  var i= left
  while (i < right) {
    out(i)= f(inp(i))
    i= i+1
  }
}
val i= Array(2,3,4,5,6)
val o= Array(0,0,0,0,0)
val f=(x:Int) => x*x
mapASegSeq(i, f, 1, 3, o)
o
====>
res1: Array[Int] = Array(0, 9, 16, 0, 0)
```

Parallel Map on Arrays

```
def mapASegPar[A,B](inp: Array[A], left: Int, right: Int,
                    f : A => B,
                    out: Array[B]): Unit = {
  // Writes to out(i) for left <= i <= right-1
```

Parallel Map on Arrays

```
def mapASegPar[A,B](inp: Array[A], left: Int, right: Int,
                    f : A => B,
                    out: Array[B]): Unit = {
  // Writes to out(i) for left <= i <= right-1

  if (right - left < threshold)
    mapASegSeq(inp, f, left, right, out)
  else {
    val mid = left + (right - left)/2
    val _ = parallel(mapASegPar(inp, left, mid, f, out),
                     mapASegPar(inp, mid, right, f, out))
  }
}
```

Parallel Map on Arrays

```
def mapASegPar[A,B](inp: Array[A], left: Int, right: Int,
                    f : A => B,
                    out: Array[B]): Unit = {
  // Writes to out(i) for left <= i <= right-1

  if (right - left < threshold)
    mapASegSeq(inp, f, left, right, out)
  else {
    val mid = left + (right - left)/2
    val _ = parallel(mapASegPar(inp, left, mid, f, out),
                     mapASegPar(inp, mid, right, f, out))
  }
}
```

- threshold needs to be large enough to compensate expense of parallel task creation
- we must ensure that two arguments of parallel write to disjoint parts of out array; do they?

Example: Pointwise Exponent of an Array

$$Array(a_1, a_2, \ldots, a_n) \longrightarrow Array(|a_1|^p, |a_2|^p, \ldots, |a_n|^p)$$

We can use previously defined higher-order functions:

```
def power(x: Int, p: Double): Int =
  math.exp(p * math.log(math.abs(x))/logE).toInt
def f(x: Int): Double = power(x, p)

mapASegSeq(inp, 0, inp.length, f, out) // sequential

mapASegPar(inp, 0, inp.length, f, out) // parallel
```

Exponent of an Array, Inlined Manually - Sequential

```
def mapPowerSeq(inp: Array[Int], p: Double,
                left: Int, right: Int,
                out: Array[Double]): Unit = {
  var i= left
  while (i < right) {
    out(i)= power(inp(i),p)
    i= i+1
  }
}
```

Exponent of an Array, Inlined Manually - Parallel

```
def mapPowerPar(inp: Array[Int], p: Double,
                left: Int, right: Int,
                out: Array[Double]): Unit = {
  if (right − left < threshold) {
    var i= left
    while (i < right) {
      out(i)= power(inp(i),p)
      i= i+1
    }
  } else {
     val mid = left + (right − left)/2
     val _ = parallel(normsOfPar(inp, p, left, mid, out),
                      normsOfPar(inp, p, mid, right, out))
  }
}
```

Measuring Performance Using Scalameter

- inp.length $= 2000000$
- threshold $= 10000$
- Intel(R) Core(TM) i7-3770K CPU @ 3.50GHz (4-core, 8 HW threads), 16GB RAM

$expression$	$time(ms)$
$mapASegSeq(inp, 0, inp.length, f, out)$	174.17
$mapASegPar(inp, 0, inp.length, f, out)$	
$mapPowerSeq(inp, p, 0, inp.length, out)$	
$mapPowerPar(inp, p, 0, inp.length, out)$	

Measuring Performance Using Scalameter

- inp.length = 2000000
- threshold = 10000
- Intel(R) Core(TM) i7-3770K CPU @ 3.50GHz (4-core, 8 HW threads), 16GB RAM

expression	*time(ms)*
$mapASegSeq(inp, 0, inp.length, f, out)$	174.17
$mapASegPar(inp, 0, inp.length, f, out)$	28.93
$mapPowerSeq(inp, p, 0, inp.length, out)$	
$mapPowerPar(inp, p, 0, inp.length, out)$	

Measuring Performance Using Scalameter

- inp.length = 2000000
- threshold = 10000
- Intel(R) Core(TM) i7-3770K CPU @ 3.50GHz (4-core, 8 HW threads), 16GB RAM

expression	*time(ms)*
$mapASegSeq(inp, 0, inp.length, f, out)$	174.17
$mapASegPar(inp, 0, inp.length, f, out)$	28.93
$mapPowerSeq(inp, p, 0, inp.length, out)$	166.84
$mapPowerPar(inp, p, 0, inp.length, out)$	

Measuring Performance Using Scalameter

- inp.length $= 2000000$
- threshold $= 10000$
- Intel(R) Core(TM) i7-3770K CPU @ 3.50GHz (4-core, 8 HW threads), 16GB RAM

$expression$	$time(ms)$
$mapASegSeq(inp, 0, inp.length, f, out)$	174.17
$mapASegPar(inp, 0, inp.length, f, out)$	28.93
$mapPowerSeq(inp, p, 0, inp.length, out)$	166.84
$mapPowerPar(inp, p, 0, inp.length, out)$	28.17

Measuring Performance Using Scalameter

- inp.length = 2000000
- threshold = 10000
- Intel(R) Core(TM) i7-3770K CPU @ 3.50GHz
  (4-core, 8 HW threads), 16GB RAM

expression	*time(ms)*
$mapASegSeq(inp, 0, inp.length, f, out)$	174.17
$mapASegPar(inp, 0, inp.length, f, out)$	28.93
$mapPowerSeq(inp, p, 0, inp.length, out)$	166.84
$mapPowerPar(inp, p, 0, inp.length, out)$	28.17

- does parallelization pay off?

Measuring Performance Using Scalameter

- inp.length = 2000000
- threshold = 10000
- Intel(R) Core(TM) i7-3770K CPU @ 3.50GHz
  (4-core, 8 HW threads), 16GB RAM

expression	*time*(*ms*)
$mapASegSeq(inp, 0, inp.length, f, out)$	174.17
$mapASegPar(inp, 0, inp.length, f, out)$	28.93
$mapPowerSeq(inp, p, 0, inp.length, out)$	166.84
$mapPowerPar(inp, p, 0, inp.length, out)$	28.17

- does parallelization pay off?
- does manually removing higher-order functions pay off?

Trees

Consider trees where

- leaves store array segments
- non-leaf node stores number of elements in left subtree

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

Assume our trees are balanced: we can explore branches in parallel

Functional Parallel Map on Trees - Creates New Tree

```
def mapTreePar[A:Manifest,B:Manifest](t: Tree[A], f: A => B) : Tree[B] =
t match {
  case Leaf(a) => {
    val len = a.length
    val b = new Array[B](len)
    var i= 0
    while (i < len) {
      b(i)= f(a(i))
      i= i + 1
    }
    Leaf(b)
  }
  case Node(l,r) => {
    val (lb,rb) = parallel(mapTreePar(l,f),mapTreePar(r,f))
    Node(lb, rb)
  }
}
```

Speedup and performance similar as the array

Parallel Fold (Reduce) for Associative Operations

Fold / Reduce

If $f(x, y) = x + y$ then

$$List(2, 5, 20).foldLeft(100)(f) = ((100 + 2) + 5) + 20$$

Given $f : (B, A) => B$

$$List(a_1, a_2, a_3).foldLeft(b)(f) = f(f(f(b, a_1), a_2), a_3)$$

Given $g : (A, B) => B$

$$List(a_1, a_2, a_3).foldRight(b)(g) = g(a_1, g(a_2, g(a_3, b)))$$

$$List(a_1, a_2, a_3).foldRight1(g) = g(a_1, g(a_2, a_3))$$

Fold / Reduce

If $f(x, y) = x + y$ then

$$List(2, 5, 20).foldLeft(100)(f) = ((100 + 2) + 5) + 20$$

Given $f : (B, A) => B$

$$List(a_1, a_2, a_3).foldLeft(b)(f) = f(f(f(b, a_1), a_2), a_3)$$

Given $g : (A, B) => B$

$$List(a_1, a_2, a_3).foldRight(b)(g) = g(a_1, g(a_2, g(a_3, b)))$$

$$List(a_1, a_2, a_3).foldRight1(g) = g(a_1, g(a_2, a_3))$$

Difficult to parallelize if we know nothing about f:

- iterating arbitrary functions leads to arbitrary messy behavior

Fold / Reduce

If $f(x, y) = x + y$ then

$$List(2, 5, 20).foldLeft(100)(f) = ((100 + 2) + 5) + 20$$

Given $f : (B, A) => B$

$$List(a_1, a_2, a_3).foldLeft(b)(f) = f(f(f(b, a_1), a_2), a_3)$$

Given $g : (A, B) => B$

$$List(a_1, a_2, a_3).foldRight(b)(g) = g(a_1, g(a_2, g(a_3, b)))$$

$$List(a_1, a_2, a_3).foldRight1(g) = g(a_1, g(a_2, a_3))$$

Difficult to parallelize if we know nothing about f:

- iterating arbitrary functions leads to arbitrary messy behavior

We look at functions $f : (A, A) => A$ that are **associative** and try to fold over a data structure in parallel

Associative Operation

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

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

Associative 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 as infix operator $\otimes$, this becomes

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

Associative 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 as infix operator $\otimes$, this becomes

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

Consequence: consider any two expressions with same list of operands connected with $\otimes$, but different parentheses. Then these expressions are equal.

Associative 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 as infix operator $\otimes$, this becomes

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

Consequence: consider any two expressions with same list of operands connected with $\otimes$, but different parentheses. Then these expressions are equal.

How to prove this precisely?

Lemma about Combining foldRight1 Expressions

Lemma 1: If $\otimes$ is associative, then for non-empty lists *xs*, *ys*

$$(xs.foldRight1(\otimes)) \otimes (ys.foldRight1(\otimes)) == (xs ::: ys).foldRight1(\otimes)$$

Proof is by induction on the length of *xs*.

- Base case: $xs = List(x)$. Then $xs.foldRight1 = x$ and $xs ::: ys = x :: ys$.
  On the right side of equality we have $(x :: ys).foldRight1(\otimes) = x \otimes (ys.foldRight1(\otimes))$, which is the same as on the left side.
- Inductive case: $xs = x :: xs1$.
  $((x :: xs1).foldRight1(\otimes)) \otimes (ys.foldRight1(\otimes)) =$
  $(x \otimes (xs1.foldRight1(\otimes))) \otimes (ys.foldRight1(\otimes)) =$
  $(x \otimes (xs1.foldRight1(\otimes))) \otimes (ys.foldRight1(\otimes)) = (\oplus \text{ assoc})$
  $x \otimes (xs1.foldRight1(\otimes) \otimes (ys.foldRight1(\otimes))) = (\text{I.H.})$
  $x \otimes (xs1 ::: ys).foldRight1(\otimes) =$
  $(x :: xs1 ::: ys).foldRight1(\otimes) = (xs ::: ys).foldRight1(\otimes)$

Lemma saying every expression equals foldRight1

Lemma 2: If $\otimes$ is associative, then every expression containing only operator $\otimes$ and operands from list xs, in that order, regardless of parantheses, produces the same result for all values as $xs.foldRight1(\otimes)$.
Proof is by induction on the size of the expression, using Lemma 1.

- Base case: expression has no $\oplus$ so it is of the form x. Then it equals $List(x).foldRight1(\oplus)$
- Inductive case. Expression is of the form $x \oplus y$. Let xL be the list of operands in x and yL the list of operands in y. By I.H., the expression produces the same result as

  $$(xL.foldRight1(\oplus)) \oplus (yL.foldRight1(\oplus))$$

  By Lemma 1, this produces the same result as $(xL ::: yL).foldRight1(\oplus)$.

Sequential Fold on Array Segments

```
def foldASegSeq[A,B](inp: Array[A], b0: B,
                     left: Int, right: Int,
                     f: (B,A) => B): B = {
  var b= b0
  var i= left
  while (i < right) {
    b= f(b, inp(i))
    i= i+1
  }
  b
}
```

Sequential Fold on Array Segments

```
def foldASegSeq[A,B](inp: Array[A], b0: B,
                     left: Int, right: Int,
                     f: (B,A) => B): B = {
  var b= b0
  var i= left
  while (i < right) {
    b= f(b, inp(i))
    i= i+1
  }
  b
}
```

If f was not known to be associative, would this be foldRight or foldLeft ?

Sequential Fold on Array Segments

```
def foldASegSeq[A,B](inp: Array[A], b0: B,
                     left: Int, right: Int,
                     f: (B,A) => B): B = {
  var b= b0
  var i= left
  while (i < right) {
    b= f(b, inp(i))
    i= i+1
  }
  b
}
```

If *f* was not known to be associative, would this be foldRight or foldLeft ?

- In general, this is foldLeft
- Result is the same as, e.g., foldRight when *f* is associative

Parallel Fold on Array Segments

```
def foldASegPar[A](inp: Array[A], a0: A,
                   left: Int, right: Int,
                   f: (A,A) => A): A = {
  // requires f to be associative
  if (right - left < threshold)
    foldASegSeq(inp, a0, left, right, f)
  else {
    val mid = left + (right - left)/2
    val (a1,a2) = parallel(foldASegPar(inp, a0, left, mid, f),
                           foldASegPar(inp, a0, mid, right, f))
    f(a1,a2)
  }
}
```

- Sequential version computed *foldLeft*
- here we compute a more balanced expression tree, combining fold of two halves of array
- the result is the same when f is associative.

Important Associative Operations that Happen to be also Commutative

Examples

- addition and multiplication modulo a positive integer (e.g. 2^{32}), including the usual arithmetic on 32-bit or 64-bit integers
- addition and multiplication of BigInt-s (mathematical integers)
- union, intersection, and symmetric difference of sets
- boolean operations &&, ||, exclusive or

If operation $f(x, y)$ is associative, then $\bar{f}$ defined by

$$\bar{f}((x_1, ..., x_n), (y_1, ..., y_n)) = (f(x_1, y_1), ..., f(x_n, y_n))$$

is also associative.

Associativity Does Not Imply Commutativity

Associativity **does NOT imply** $x \otimes y = y \otimes x$ (commutativity)
Examples of *associative* and *not commutative* operations:

- concatenation (append) of lists $(x ::: y) ::: z == x ::: (y ::: z)$ and strings
- matrix multiplication AB
- composition of relations
  $r \odot s = \{(a, c) \mid \exists b.(a, b) \in r \wedge (b, c) \in s\}$
- composition of functions $(f \circ g)(x) = f(g(x))$

Similarly, Commutativity Does Not Imply Associativity

Example:

$$f(x, y) = x^2 + y^2 = f(y, x)$$

Then

$$f(f(x, y), z) = (x^2 + y^2)^2 + z^2$$

whereas

$$f(x, f(y, z)) = x^2 + (y^2 + z^2)^2$$

Similarly, Commutativity Does Not Imply Associativity

Example:

$$f(x, y) = x^2 + y^2 = f(y, x)$$

Then

$$f(f(x, y), z) = (x^2 + y^2)^2 + z^2$$

whereas

$$f(x, f(y, z)) = x^2 + (y^2 + z^2)^2$$

In general, if $p(x, y)$ is commutative and $h_1(z), h_2(z)$ are arbitrary, then any function defined by

$$q(x, y) = h_2(p(h_1(x), h_1(y)))$$

is equal to $h_2(p(h_1(y), h_2(x))) = q(y, x)$, so it is commutative, but can lose associativity even if q was associative.

Floating Point Addition is Not Associative

```
scala> val e = 1e-200
e: Double = 1.0E-200

scala> val x = 1e200
x: Double = 1.0E200

scala> val mx = -x
mx: Double = -1.0E200

scala> (x + mx) + e
res2: Double = 1.0E-200

scala> x + (mx + e)
res3: Double = 0.0

scala> (x + mx) + e == x + (mx + e)
res4: Boolean = false
```

Similarly for multiplication

Two Rules Implying Associativity

Suppose that $f(x, y)$ is commutative and if we define

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

then $E(x, y, z) = E(y, z, x)$. Show that f is then associative.

Two Rules Implying Associativity

Suppose that $f(x, y)$ is commutative and if we define

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

then $E(x, y, z) = E(y, z, x)$. Show that f is then associative.
Solution:

Two Rules Implying Associativity

Suppose that $f(x, y)$ is commutative and if we define

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

then $E(x, y, z) = E(y, z, x)$. Show that f is then associative.
Solution:

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

Is this Operation on Real Numbers Associative?

Let u, v range over the open interval of reals $(-1, 1)$

$$f(u, v) = \frac{u + v}{1 + uv}$$

Is this Operation on Real Numbers Associative?

Let u, v range over the open interval of reals $(-1, 1)$

$$f(u, v) = \frac{u + v}{1 + uv}$$

Clearly, $f(u, v) = f(v, u)$.

Is this Operation on Real Numbers Associative?

Let u, v range over the open interval of reals $(-1, 1)$

$$f(u, v) = \frac{u + v}{1 + uv}$$

Clearly, $f(u, v) = f(v, u)$.
Next

$$f(f(u, v), w) = \frac{\frac{u+v}{1+uv} + w}{1 + \frac{u+v}{1+uv} w} = \frac{u + v + w + uvw}{1 + uv + uw + vw}$$

From the above two we have with v, w, u playing the role of u, v, w:

$$f(u, f(v, w)) = f(f(v, w), u) = \frac{v + w + u + vwu}{1 + vw + vu + uv}$$

So two sides of associativity condition are equal.

Is this Operation on Real Numbers Associative?

Let u, v range over the open interval of reals $(-1, 1)$

$$f(u, v) = \frac{u + v}{1 + uv}$$

Clearly, $f(u, v) = f(v, u)$.
Next

$$f(f(u, v), w) = \frac{\frac{u+v}{1+uv} + w}{1 + \frac{u+v}{1+uv} w} = \frac{u + v + w + uvw}{1 + uv + uw + vw}$$

From the above two we have with v, w, u playing the role of u, v, w:

$$f(u, f(v, w)) = f(f(v, w), u) = \frac{v + w + u + vwu}{1 + vw + vu + uv}$$

So two sides of associativity condition are equal.
What is the motivation for this operation?

Is this Operation on Real Numbers Associative?

Let u, v range over the open interval of reals $(-1, 1)$

$$f(u, v) = \frac{u + v}{1 + uv}$$

Clearly, $f(u, v) = f(v, u)$.
Next

$$f(f(u, v), w) = \frac{\frac{u+v}{1+uv} + w}{1 + \frac{u+v}{1+uv} w} = \frac{u + v + w + uvw}{1 + uv + uw + vw}$$

From the above two we have with v, w, u playing the role of u, v, w:

$$f(u, f(v, w)) = f(f(v, w), u) = \frac{v + w + u + vwu}{1 + vw + vu + uv}$$

So two sides of associativity condition are equal.
What is the motivation for this operation?
Law of adding (normalized) velocities in special relativity theory

Velocity Addition as an Example for Fold

```
val c = 2.99792458e8
def assocOp(v1: Double, v2: Double): Double = {
  val u1 = v1/c
  val u2 = v2/c
  (v1 + v2)/(1 + u1*u2)
}
def addVelSeq(inp: Array[Double]): Double = {
  foldASegSeq(inp, 0.0, 0, inp.length, assocOp)
}
def addVelPar(inp: Array[Double]): Double = {
  foldASegPar(inp, 0.0, 0, inp.length, assocOp)
}
```

We obtain noticeable speedup (2-3 times).
Value computed differs slightly because of roundoff errors.

A Family of Associative Operations on Sets

Define binary operation on sets A, B by

$$f(A, B) = (A \cup B)^*$$

where $*$ is any operator on sets (closure) with these properties:

- $A \subseteq A^*$
- if $A \subseteq B$ then $A^* \subseteq B^*$
- $(A^*)^* = A^*$

Prove that f is associative.

A Family of Associative Operations on Sets

Define binary operation on sets A, B by

$$f(A, B) = (A \cup B)^*$$

where $*$ is any operator on sets (closure) with these properties:

- $A \subseteq A^*$
- if $A \subseteq B$ then $A^* \subseteq B^*$
- $(A^*)^* = A^*$

Prove that f is associative.

Hint:

A Family of Associative Operations on Sets

Define binary operation on sets A, B by

$$f(A, B) = (A \cup B)^*$$

where $*$ is any operator on sets (closure) with these properties:

- $A \subseteq A^*$
- if $A \subseteq B$ then $A^* \subseteq B^*$
- $(A^*)^* = A^*$

Prove that f is associative.

Hint:

- Observe that $P \subseteq Q^*$ implies $P^* \subseteq Q^*$

A Family of Associative Operations on Sets

Define binary operation on sets A, B by

$$f(A, B) = (A \cup B)^*$$

where $*$ is any operator on sets (closure) with these properties:

- $A \subseteq A^*$
- if $A \subseteq B$ then $A^* \subseteq B^*$
- $(A^*)^* = A^*$

Prove that f is associative.

Hint:

- Observe that $P \subseteq Q^*$ implies $P^* \subseteq Q^*$
- Keep in mind that $P \cup Q \subseteq R$ is equivalent to the conjunction of $P \subseteq R$ and $Q \subseteq R$.

A Family of Associative Operations on Sets

Define binary operation on sets A, B by

$$f(A, B) = (A \cup B)^*$$

where $*$ is any operator on sets (closure) with these properties:

- $A \subseteq A^*$
- if $A \subseteq B$ then $A^* \subseteq B^*$
- $(A^*)^* = A^*$

Prove that f is associative.

Hint:

- Observe that $P \subseteq Q^*$ implies $P^* \subseteq Q^*$
- Keep in mind that $P \cup Q \subseteq R$ is equivalent to the conjunction of $P \subseteq R$ and $Q \subseteq R$.
- Use previous to show $f(f(A, B), C) = (A \cup B \cup C)^*$

A Family of Associative Operations on Sets

Define binary operation on sets A, B by

$$f(A, B) = (A \cup B)^*$$

where $*$ is any operator on sets (closure) with these properties:

- $A \subseteq A^*$
- if $A \subseteq B$ then $A^* \subseteq B^*$
- $(A^*)^* = A^*$

Prove that f is associative.

Hint:

- Observe that $P \subseteq Q^*$ implies $P^* \subseteq Q^*$
- Keep in mind that $P \cup Q \subseteq R$ is equivalent to the conjunction of $P \subseteq R$ and $Q \subseteq R$.
- Use previous to show $f(f(A, B), C) = (A \cup B \cup C)^*$
- Observe that $f(f(A, B), C) = f(f(B, C), A)$ and $f(A, B) = f(B, A)$, then use the slide "two rules implying associativity".

Parallel Scan

Scan

If $f(x, y) = x + y$ then

$$List(2, 5, 20).scanLeft(100)(f) = List(100, 102, 107, 127)$$

$$List(a_1, a_2, a_3).scanLeft(f)(a_0) = List(b_0, b_1, b_2, b_3)$$

where

- $b_0 = a_0$
- $b_1 = f(b_0, a_1)$
- $b_2 = f(b_1, a_2)$
- $b_3 = f(b_2, a_3)$

Can scan be parallelized? Assume f associative.

Sequential Scan over a Segment

```
def scanASegSeq1[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))
      out(i+1)=a
      i= i+1
    }
  } }
```

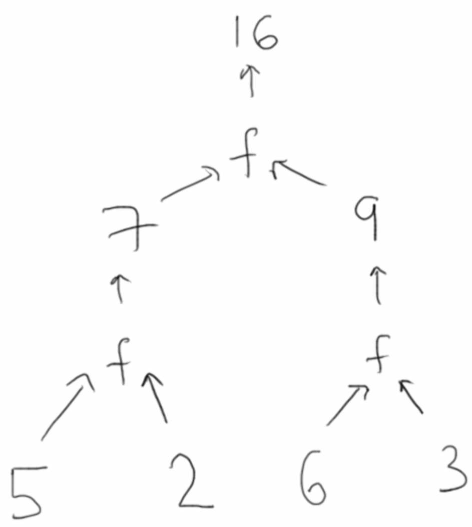
Scans array segment inp(left) to inp(right-1), storing results into out(left+1) to out(right). At the end, each out(i+1) stores fold of elements: [a0, in(left),... in(i)] for i from left to right-1. In particular, out(left+1) stores f(a0,inp(left)) and out(right) stores fold of [a0, in(left),... inp(right-1)]. The value a0 is not directly stored anywhere.

Computation Tree for Recording Results of Parallel Fold

```
sealed abstract class FoldTree[A] {
  val res: A // value of the tree, whether it is leaf or not
}
case class Leaf[A](from: Int, to: Int, resLeaf: A) extends FoldTree[A] {
  val res= resLeaf
}
case class Node[A](l: FoldTree[A], r: FoldTree[A],
                   resNode: A) extends FoldTree[A] {
  val res= resNode
}
```

upsweep: Parallel Fold that Records its Computation Tree

```
def upsweep[A](inp: Array[A], left: Int, right: Int,
                    a0: A, f: (A,A) => A): FoldTree[A] = {
  // requires f to be associative
  if (right - left < threshold)
    Leaf(left, right, foldASegSeq(inp, left, right, a0, f))
  else {
    val mid = left + (right - left)/2
    val (t1,t2) = parallel(upsweep(inp, left, mid, a0, f),
                           upsweep(inp, mid, right, a0, f))
    Node(t1, t2, f(t1.res,t2.res))
} }
```

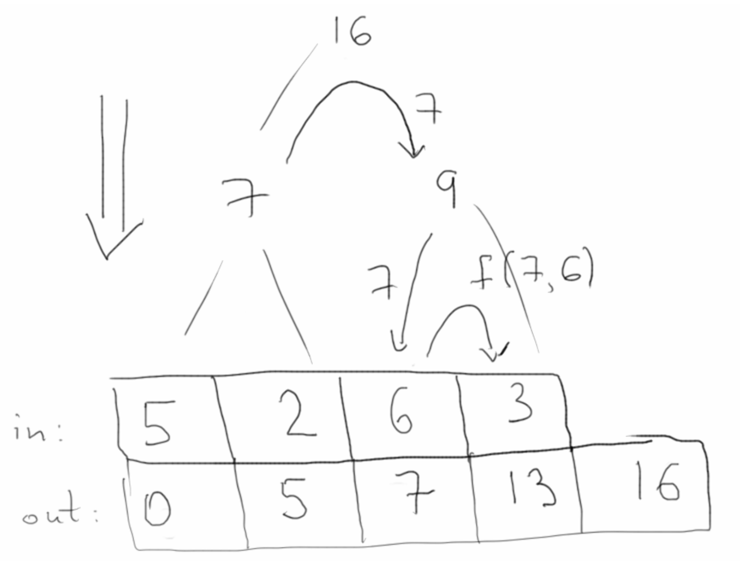


Folds array segment in parallel and record the intermediate computation results in a Tree[A].

- In the context of scan, this phase is called **upsweep**. For an intuition, picture the array to fold on the bottom, and the root of the tree at the top. Once the 'parallel' tasks are initiated, the results are combined in the 'up' direction, from array to the result of the fold.

Using the Tree to Compute Scan: Key Part

```
def downsweep[A](inp: Array[A],
                 a0: A, f: (A,A) => A,
                 t: FoldTree[A],
                 out: Array[A]): Unit = {
  t match {
    case Leaf(from, to, res) =>
      scanASegSeq1(inp, from, to, a0, f, out)
    case Node(l, r, res) => {
      parallel(downsweep(inp, a0, f, l, out),
               downsweep(inp, f(a0, l.res), f, r, out))
    }
} }
```



Scan of [5,2,6,3] with +

Scanning the Entire Array

```
def scanASegPar[A](inp: Array[A], from: Int, to: Int,
                   a0: A, f: (A,A) => A, out: Array[A]) = {
  val t = upsweep(inp, from, to, a0, f)
  downsweep(inp, a0, f, t, out) }

def scanAPar[A](inp: Array[A], a0: A, f: (A,A) => A,
                out: Array[A]) = {
  out(0)= a0
  scanASegPar(inp, 0, inp.length, a0, f, out) }
```

Example: producing all partial sums of velocities, relativistically:

```
val c = 2.99792458e8
def assocOp(v1: Double, v2: Double): Double = {
  val u1 = v1/c; val u2 = v2/c
  (v1 + v2)/(1 + u1*u2)
}
scanAPar(inp, 0.0, assocOp, outPar)
```

Combining and Fusing Operations

Combining Maps

If $f(x, y) = x + y$ then

$$List(2, 5, 20).map(f).map(g) == List(g(f(2)), g(f(5)), g(f(20)))$$

Instead of producing intermediate structure, we can apply both f and g in one pass.

- this idea applies to both sequential and parallel traversals

Array Norm

$$\sum_{i=s}^{t-1} \lfloor |a_i|^p \rfloor$$

Which combination of operations does sum of powers correspond to?

Array Norm

$$\sum_{i=s}^{t-1} \lfloor |a_i|^p \rfloor$$

Which combination of operations does sum of powers correspond to?

- first: $map(x => power(abs(x), p))$

Array Norm

$$\sum_{i=s}^{t-1} \lfloor |a_i|^p \rfloor$$

Which combination of operations does sum of powers correspond to?

- first: $map(x => power(abs(x), p))$
- second: *fold* with $+$

Note: folding with $f(x, y) = |x|^p + |y|^p$ gives

Array Norm

$$\sum_{i=s}^{t-1} \lfloor |a_i|^p \rfloor$$

Which combination of operations does sum of powers correspond to?

- first: $map(x => power(abs(x), p))$
- second: *fold* with $+$

Note: folding with $f(x, y) = |x|^p + |y|^p$ gives nonsense

Array Norm as Example of Fused Operations

The recursive case is not affected, as if we had just fold of +:

```
def pNormRec(a: Array[Int], p: Real): Int =
  power(segmentRec(a, p, 0, a.length), 1/p)

def segmentRec(a: Array[Int], p: Real, s: Int, t: Int) = {
  if (t - s < threshold)
    sumSegment(xs, p, s, t) // we read array content only here
  else {
    val mid = s + (t - s)/2
    val (leftSum, rightSum) =
      parallel(segmentRec(a, p, s, mid),
               segmentRec(a, p, mid, t))
    leftSum + rightSum
  }
}
```

Array Norm as Example of Fused Operations

fused map and sum happen below the cutoff:

```
def sumSegment(a: Array[Int], p: Double, s: Int, t: Int): Int = {
  var i= s; var sum: Int = 0
  while (i < t) {
    sum= sum + power(a(i), p) // fused map(power(_,p)) and fold(_+_)
    i= i + 1
  }
  sum
}
```

Histograms

Suppose elements of $inp : Array[Int]$ are between 0 and 99.

For each interval $[10k, 10k + 9]$, count how many array elements are in there, storing the result in array $hist : Array$ of size 10.

Express this task as a combination of map and fold.

What is the type on which fold works?

Running Average

Given an array *inp*, compute array *out* where $out(i)$ is the average of elements $inp(j)$ for $0 \leq j < i$.

Running Average

Given an array *inp*, compute array *out* where $out(i)$ is the average of elements $inp(j)$ for $0 \leq j < i$.

```
// code from parallel scan
def scanASegSeq1[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))
      out(i+1)=a // can we modify it to compute average?
      i= i+1
    }
  }
}
```