ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Parallel Sorting

Parallel Programming in Scala

Viktor Kuncak

Merge Sort

We will implement a parallel merge sort algorithm.

1. recursively sort the two halves of the array in parallel
2. sequentially merge the two array halves by copying into a temporary array
3. copy the temporary array back into the original array

The `parMergeSort` method takes an array, and a maximum depth:

Merge Sort

We will implement a parallel merge sort algorithm.

1. recursively sort the two halves of the array in parallel
2. sequentially merge the two array halves by copying into a temporary array
3. copy the temporary array back into the original array

The `parMergeSort` method takes an array, and a maximum depth:

```
def parMergeSort(xs: Array[Int], maxDepth: Int): Unit = {
```

Allocating an Intermediate Array

We start by allocating an intermediate array:

```
val ys = new Array[Int](xs.length)
```

At each level of the merge sort, we will alternate between the source array `xs` and the intermediate array `ys`.

Sorting the Array

```
def sort(from: Int, until: Int, depth: Int): Unit = {
  if (depth == maxDepth) {
    quickSort(xs, from, until - from)
  } else {
    val mid = (from + until) / 2
    parallel(sort(mid, until, depth + 1), sort(from, mid, depth + 1))
```

Sorting the Array

```
def sort(from: Int, until: Int, depth: Int): Unit = {
  if (depth == maxDepth) {
    quickSort(xs, from, until - from)
  } else {
    val mid = (from + until) / 2
    parallel(sort(mid, until, depth + 1), sort(from, mid, depth + 1))

    val flip = (maxDepth - depth) % 2 == 0
    val src = if (flip) ys else xs
    val dst = if (flip) xs else ys
    merge(src, dst, from, mid, until)
  }
}
sort(0, xs.length, 0)
```

Merging the Array

Given an array `src` consisting of two sorted intervals, merge those interval into the `dst` array:

```
def merge(src: Array[Int], dst: Array[Int],
  from: Int, mid: Int, until: Int): Unit
```

The `merge` implementation is sequential, so we will not go through it.

Merging the Array

Given an array `src` consisting of two sorted intervals, merge those interval into the `dst` array:

```
def merge(src: Array[Int], dst: Array[Int],
  from: Int, mid: Int, until: Int): Unit
```

The `merge` implementation is sequential, so we will not go through it.

How would you implement `merge` in parallel?

Copying the Array

```
def copy(src: Array[Int], target: Array[Int],
  from: Int, until: Int, depth: Int): Unit = {
  if (depth == maxDepth) {
    Array.copy(src, from, target, from, until - from)
  } else {
    val mid = (from + until) / 2
    val right = parallel(
      copy(src, target, mid, until, depth + 1),
      copy(src, target, from, mid, depth + 1)
    )
  }
}
if (maxDepth % 2 == 0) copy(ys, xs, 0, xs.length, 0)
```

Demo

Let's compare the performance of `parMergeSort` against the Scala `quicksort` implementation.

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Data Operations and Parallel Mapping

Parallel Programming in Scala

Viktor Kuncak

Parallelism and collections

Parallel processing of collections is important

- one the main applications of parallelism today

We examine conditions when this can be done

- properties of collections: ability to split, combine
- properties of operations: associativity, independence

Functional programming and collections

Operations on collections are key to functional programming

map: apply function to each element

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

Functional programming and collections

Operations on collections are key to functional programming

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`

Functional programming and collections

Operations on collections are key to functional programming

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`

scan: combine folds of all list prefixes

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

These operations are even more important for parallel than sequential collections: they encapsulate more complex algorithms

Choice of data structures

We use **List** to specify the results of operations

Lists are not good for parallel implementations because we cannot efficiently

- split them in half (need to search for the middle)
- combine them (concatenation needs linear time)

Choice of data structures

We use **List** to specify the results of operations

Lists are not good for parallel implementations because we cannot efficiently

- split them in half (need to search for the middle)
- combine them (concatenation needs linear time)

We use for now these alternatives

- **arrays**: imperative (recall array sum)
- **trees**: can be implemented functionally

Subsequent lectures examine Scala's parallel collection libraries

- includes many more data structures, implemented efficiently

Map: meaning and properties

Map applies a given function to each list element

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

List(a_1, a_2, ..., a_n).map(f) $==$ List(f(a_1), f(a_2), ..., f(a_n))

Properties to keep in mind:

- `list.map(x => x) == list`
- `list.map(f.compose(g)) == list.map(g).map(f)`

Recall that `(f.compose(g))(x) = f(g(x))`

Map as function on lists

Sequential definition:

```
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)
}
```

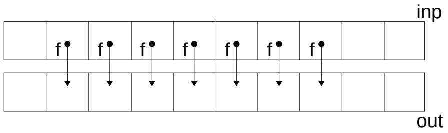
We would like a version that parallelizes

- computations of `f(h)` for different elements `h`
- finding the elements themselves (list is not a good choice)

Sequential map of an array producing an array

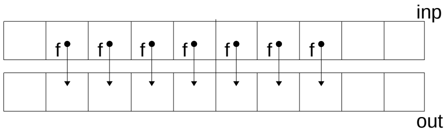
```
def mapASegSeq[A,B](inp: Array[A], left: Int, right: Int, f : A => B,
                    out: Array[B]) = {
  // Writes to out(i) for left <= i <= right-1
  var i= left
  while (i < right) {
    out(i)= f(inp(i))
    i= i+1
} }
val in= Array(2,3,4,5,6)
val out= Array(0,0,0,0,0)
val f= (x:Int) =>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

res1: Array[Int] = Array(0, 9, 16, 0, 0)
```



Parallel map of an array producing an array

```scala
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, left, right, f, out)
  else {
    val mid = left + (right - left)/2
    parallel(mapASegPar(inp, left, mid, f, out),
             mapASegPar(inp, mid, right, f, out))
  }
}
```



Note:

- writes need to be disjoint (otherwise: non-deterministic behavior)
- threshold needs to be large enough (otherwise we lose efficiency)

Example of using mapASegPar: pointwise exponent

Raise each array element to power p:

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

We can use previously defined higher-order functions:

```
val p: Double = 1.5
def f(x: Int): Double = power(x, p)

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

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

Example of using mapASegPar: pointwise exponent

Raise each array element to power p:

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

We can use previously defined higher-order functions:

```
val p: Double = 1.5
def f(x: Int): Double = power(x, p)

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

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

Questions on performance:

- are there performance gains from parallel execution
- performance of re-using higher-order functions vs re-implementing

Sequential pointwise exponent written from scratch

```
def normsOf(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
  }
}
```

Parallel pointwise exponent written from scratch

```
def normsOfPar(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
     parallel(normsOfPar(inp, p, left, mid, out),
              normsOfPar(inp, p, mid, right, out))
  }
}
```

Measured 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
$normsOfSeq(inp, p, 0, inp.length, out)$	166.84
$normsOfPar(inp, p, 0, inp.length, out)$	28.17

Measured 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
$normsOfSeq(inp, p, 0, inp.length, out)$	166.84
$normsOfPar(inp, p, 0, inp.length, out)$	28.17

Parallelization pays off

Manually removing higher-order functions does not pay off

Parallel map on immutable trees

Consider trees where

- leaves store array segments
- non-leaf node stores two subtrees

```
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 that our trees are balanced: we can explore branches in parallel

Parallel map on immutable trees

```scala
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 for the array

Give depth bound of mapTreePar

Give a correct but as tight as possible asymptotic parallel computation depth bound for `mapTreePar` applied to complete trees with height h and 2^h nodes, assuming the passed first-class function f executes in constant time.

1. 2^h
2. h
3. $\log h$
4. $h \log h$
5. $h2^h$

Give depth bound of mapTreePar

Give a correct but as tight as possible asymptotic parallel computation depth bound for `mapTreePar` applied to complete trees with height h and 2^h nodes, assuming the passed first-class function f executes in constant time.

1. 2^h
2. h
3. $\log h$
4. $h \log h$
5. $h2^h$

Answer: h. The computation depth equals the height of the tree.

Comparison of arrays and immutable trees

Arrays:

- (+) random access to elements, on shared memory can share array
- (+) good memory locality
- (-) imperative: must ensure parallel tasks write to disjoint parts
- (-) expensive to concatenate

Immutable trees:

- (+) purely functional, produce new trees, keep old ones
- (+) no need to worry about disjointness of writes by parallel tasks
- (+) efficient to combine two trees
- (-) high memory allocation overhead
- (-) bad locality

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

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:

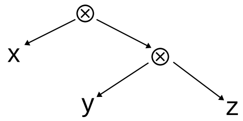
$$(x \otimes y) \otimes (z \otimes w) = (x \otimes (y \otimes z)) \otimes w = ((x \otimes y) \otimes z) \otimes w$$

Trees for expressions

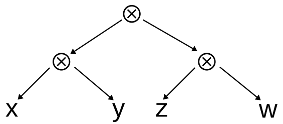
Each expression built from values connected with $\otimes$ can be represented as a tree

- 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?

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

$$\begin{tikzcd}
& \otimes \arrow[dl] \arrow[dr] & & & & & & \otimes \arrow[dl] \arrow[dr] & \\
x & & \otimes \arrow[dl] \arrow[dr] & & = & & \otimes \arrow[dl] \arrow[dr] & & z \\
& y & & z & & x & & y &
\end{tikzcd}$$

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:

$$\begin{tikzcd}
& & \otimes \arrow[dl] \arrow[dr] & \\
& \otimes \arrow[dl] \arrow[dr] & & z \\
x & & y &
\end{tikzcd}
\quad \Rightarrow \quad
\begin{tikzcd}
& \otimes \arrow[dl] \arrow[dr] & & \\
x & & \otimes \arrow[dl] \arrow[dr] & \\
& y & & z
\end{tikzcd}$$

Example use:

$$\begin{tikzcd}
& & & \otimes \arrow[dll] \arrow[drr] & & & \\
& \otimes \arrow[dl] \arrow[dr] & & & & \otimes \arrow[dl] \arrow[dr] & \\
x & & y & & p & & q
\end{tikzcd}
\quad \Rightarrow \quad
\begin{tikzcd}
& \otimes \arrow[dl] \arrow[dr] & & & \\
x & & \otimes \arrow[dl] \arrow[dr] & & \\
& y & & \otimes \arrow[dl] \arrow[dr] & \\
& & p & & q
\end{tikzcd}$$

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)
```

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Associative Operations

Parallel Programming in Scala

Viktor Kuncak

Associative operation

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

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

Consequence:

- two expressions with same list of operands connected with $\otimes$, but different parentheses evaluate to the same result
- reduce on any tree with this list of operands gives the same result

Associative operation

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

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

Consequence:

- two expressions with same list of operands connected with $\otimes$, but different parentheses evaluate to the same result
- reduce on any tree with this list of operands gives the same result

Which operations are associative?

A different property: commutativity

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

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

There are operations that are associative but not commutative

There are operations that are commutative but not associative

For correctness of `reduce`, we need (just) associativity

Examples of operations that are both associative and commutative

Many operations from math:

- addition and multiplication of mathematical integers (`BigInt`) and of exact rational numbers (given as, e.g., pairs of `BigInts`)
- addition and multiplication modulo a positive integer (e.g. 2^{32}), including the usual arithmetic on 32-bit `Int` or 64-bit `Long` values
- union, intersection, and symmetric difference of sets
- union of bags (multisets) that preserves duplicate elements
- boolean operations $\&\&, ||$, exclusive or
- addition and multiplication of polynomials
- addition of vectors
- addition of matrices of fixed dimension

Using sum: array norm

Our array norm example computes first:

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

Which combination of operations does sum of powers correspond to?

Using sum: array norm

Our array norm example computes first:

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

Which combination of operations does sum of powers correspond to?

```
reduce(map(a, power(abs(_), p)), _ + _)
```

Here $+$ is the associative operation of `reduce`

`map` can be combined with `reduce` to avoid intermediate collections

Examples of operations that are associative but not commutative

These examples illustrate that associativity does not imply commutativity:

- concatenation (append) of lists: `(x ++ y) ++ z == x ++ (y ++ z)`
- concatenation of `Strings` (which can be viewed as lists of `Char`)
- matrix multiplication AB for matrices A and B of compatible dimensions
- composition of relations $r \odot s = \{(a, c) \mid \exists b.(a, b) \in r \wedge (b, c) \in s\}$
- composition of functions $(f \circ g)(x) = f(g(x))$

Because they are associative, `reduce` still gives the same result.

Many operations are commutative but not associative

This function is also commutative:

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

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

$$\begin{aligned} f(f(x,y),z) &= (x^2+y^2)^2 + z^2 \\ f(x,f(y,z)) &= x^2 + (y^2+z^2)^2 \end{aligned}$$

These are polynomials of different growth rates with respect to different variables and are easily seen to be different for many x, y, z.

Proving commutativity alone does not prove associativity and does not guarantee that the result of `reduce` is the same as e.g. `reduceLeft` and `reduceRight`.

Associativity is not preserved by mapping

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

$$g(x, y) = h_2(f(h_1(x), h_1(y)))$$

is equal to $h_2(f(h_1(y), h_2(x))) = g(y, x)$, so it is commutative, but it often loses associativity even if f was associative to start with.

Previous example was an instance of this for $h_1(x) = h_2(x) = x^2$.

When combining and optimizing `reduce` and `map` invocations, we need to be careful that operations given to `reduce` remain associative.

Floating point addition is commutative but 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
```

Floating point multiplication is commutative but not associative

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

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

scala> (e*x)*x
res0: Double = 1.0E200

scala> e*(x*x)
res1: Double = Infinity

scala> (e*x)*x == e*(x*x)
res2: Boolean = false
```

Making an operation commutative is easy

Suppose we have a binary operation `g` and a strict total ordering `less` (e.g. lexicographical ordering of bit representations).

Then this operation is commutative:

```
def f(x: A, y: A) = if (less(y,x)) g(y,x) else g(x,y)
```

Indeed `f(x,y)==f(y,x)` because:

- if `x==y` then both sides equal `g(x,x)`
- if `less(y,x)` then left sides is `g(y,x)` and it is not `less(x,y)` so right side is also `g(y,x)`
- if `less(x,y)` then it is not `less(y,x)` so left sides is `g(x,y)` and right side is also `g(x,y)`

We know of no such efficient trick for associativity

Associative operations on tuples

Suppose `f1: (A1,A1) => A1` and `f2: (A2,A2) => A2` are associative

Then `f: ((A1,A2), (A1,A2)) => (A1,A2)` defined by

```
f((x1,x2), (y1,y2)) = (f1(x1,y1), f2(x2,y2))
```

is also associative:

```
f(f((x1,x2), (y1,y2)), (z1,z2)) ==
f((f1(x1,y1), f2(x2,y2)), (z1,z2)) ==
(f1(f1(x1,y1), z1), f2(f2(x2,y2), z2)) == (because f1, f2 are associative)
(f1(x1, f1(y1,z1)), f2(x2, f2(y2,z2))) ==
f((x1 x2), (f1(y1,z1), f2(y2,z2))) ==
f((x1 x2), f((y1,y2), (z1, z2)))
```

We can similarly construct associative operations on for n-tuples

Example: rational multiplication

Suppose we use 32-bit numbers to represent numerator and denominator of a rational number.

We can define multiplication working on pairs of numerator and denominator

```
times((x1,y1), (x2, y2)) = (x1*x2, y1*y2)
```

Because multiplication modulo 2^{32} is associative, so is `times`

Example: average

Given a collection of integers, compute the average

```
val sum = reduce(collection, _ + _)
val length = reduce(map(collection, (x:Int) => 1), _ + _)
sum/length
```

This includes two reductions. Is there a solution using a single reduce?

Example: average

Use pairs that compute sum and length at once

```
f((sum1,len1), (sum2, len2)) = (sum1 + sum1, len1 + len2)
```

Function `f` is associative because addition is associative.

Solution is then:

```
val (sum, length) = reduce(map(collection, (x:Int) => (x,1)), f)
sum/length
```

Associativity through symmetry and commutativity

Although commutativity of `f` alone does not imply associativity, it implies it if we have an additional property. Define:

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

We say arguments of `E` can rotate if `E(x,y,z) = E(y,z,x)`, that is:

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

Claim: if `f` is commutative and arguments of `E` can rotate then `f` is also associative.

Proof:

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

Example: addition of modular fractions

Define

```
plus((x1,y1), (x2, y2)) = (x1*y2 + x2*y1, y1*y2)
```

where $*$ and + are all modulo some base (e.g. 2^{32}).

We can have overflows in both numerator and denominator

Is such `plus` associative?

Example: addition of modular fractions

```
plus((x1,y1), (x2, y2)) = (x1*y2 + x2*y1, y1*y2)
```

Observe: `plus` is commutative. Moreover:

```
E((x1,y1), (x2,y2), (x3,y3)) ==
plus(plus((x1,y1), (x2,y2)), (x3,y3)) ==
plus((x1*y2 + x2*y1, y1*y2), (x3,y3)) ==
((x1*y2 + x2*y1)*y3 + x3*y1*y2, y1*y2*y3) ==
(x1*y2*y3 + x2*y1*y3 + x3*y1*y2, y1*y2*y3)
```

Therefore

```
E((x2,y2), (x3,y3), (x1,y1)) ==
(x2*y3*y1 + x3*y2*y1 + x1*y2*y3, y2*y3*y1)
```

which is the same. By previous claim, `plus` is associative.

Example: relativistic velocity addition

Let u, v range over rational numbers in the open interval $(-1, 1)$

Define f to add velicities according to special relativity

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

Clearly, f is commutative: $f(u, v) = f(v, u)$.

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

We can rotate arguments u, v, w

Example: relativistic velocity addition

Let u, v range over rational numbers in the open interval $(-1, 1)$

Define f to add velicities according to special relativity

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

Clearly, f is commutative: $f(u, v) = f(v, u)$.

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

We can rotate arguments u, v, w

f is commutative and we can rotate, so f is associative.

Consequences of non-associativity on floating points

If we implement f given by expression

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

using floating point numbers, then the operation is not associative.

Even though the difference between $f(x, f(y, z))$ and $f(f(x, y), z)$ is small in one step, over many steps it accumulates, so the result of the `reduceLeft` and a `reduce` may differ substantially.

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:

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

Example of $*$: convex hull, Kleene star in regular expressions

Claim: every such f is associative.

Proof: f is commutative. It remains to show

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

because from there it is easy to see that the arguments rotate.

First subset inclusion

We need to prove: $((A \cup B)^* \cup C)^* \subseteq (A \cup B \cup C)^*$.

Since $A \cup B \subseteq A \cup B \cup C$, by monotonicity:

$$(A \cup B)^* \subseteq (A \cup B \cup C)^*$$

Similarly

$$C \subseteq A \cup B \cup C \subseteq (A \cup B \cup C)^*$$

Thus $(A \cup B)^* \cup C \subseteq (A \cup B \cup C)^*$. By monotonicity and idempotence

$$((A \cup B)^* \cup C)^* \subseteq ((A \cup B \cup C)^*)^* = (A \cup B \cup C)^*$$

Second subset inclusion

We need to prove: $(A \cup B \cup C)^* \subseteq ((A \cup B)^* \cup C)^*$

From expansion we have $A \cup B \subseteq (A \cup B)^*$. Thus

$$A \cup B \cup C \subseteq (A \cup B)^* \cup C$$

The property then follows by monotonicity.

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

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`

```scala
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:

```
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]
```

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]
```

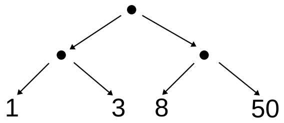
Can you define `reduceRes` function that transforms `Tree` into `TreeRes`?

Reduce that preserves the computation tree

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

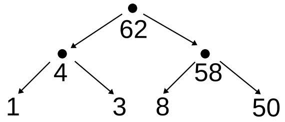
Reduce that preserves the computation tree

```
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

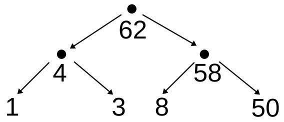
```
def reduceRes[A](t: Tree[A], f: (A,A) => A): TreeRes[A] = t match {
  case Leaf(v) => LeafRes(v)
  case Node(l, r) => {
    val (tL, tR) = (reduceRes(l, f), reduceRes(r, f))
    NodeRes(tL, f(tL.res, tR.res), tR)
  }
}
val t1 = Node(Node(Leaf(1), Leaf(3)), Node(Leaf(8), Leaf(50)))
val plus = (x:Int,y:Int)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

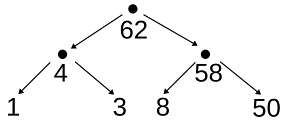
```
def upsweep[A](t: Tree[A], f: (A,A) => A): TreeRes[A] = t match {
  case Leaf(v) => LeafRes(v)
  case Node(l, r) => {
    val (tL, tR) = parallel(upsweep(l, f), upsweep(r, f))
    NodeRes(tL, f(tL.res, tR.res), tR)
  }
}
```

Using tree with results to create the final collection



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

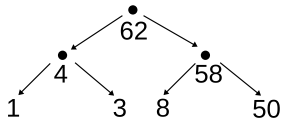
Using tree with results to create the final collection



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

```
// '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

```
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`.

```
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

```
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

```
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