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)
}
```

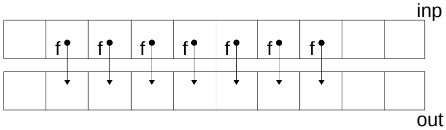
We would like a version that parallelizes

- 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

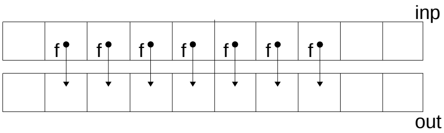
```
def mapASegSeq[A,B](inp: Array[A], left: Int, right: Int, f : A => B,
                    out: Array[B]) = {
  // Writes to out(i) for left <= i <= right-1
  var i= left
  while (i < right) {
    out(i)= f(inp(i))
    i= i+1
} }
val in= Array(2,3,4,5,6)
val out= Array(0,0,0,0,0)
val f= (x:Int) =>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

```
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

```scala
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

$\mathit{expression}$	$\mathit{time}(\mathit{ms})$
$\mathit{mapASegSeq}(\mathit{inp}, 0, \mathit{inp.length}, f, \mathit{out})$	174.17
$\mathit{mapASegPar}(\mathit{inp}, 0, \mathit{inp.length}, f, \mathit{out})$	28.93
$\mathit{normsOfSeq}(\mathit{inp}, p, 0, \mathit{inp.length}, \mathit{out})$	166.84
$\mathit{normsOfPar}(\mathit{inp}, p, 0, \mathit{inp.length}, \mathit{out})$	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

End of Slide Deck