EPFL

Week 6: Wednesday.par

CS-214 Software Construction

Outline

- Associativity vs Commutativity
- Work and Depth Analysis of Parallel Programs

Associativity vs Commutativity - Continued

CS-214 Software Construction

Example: function composition is not commutative

```scala
val f = (x:Int) => x*x

val g = (x:Int) => x + 1

val fg = f andThen g
val gf = g andThen f

val L = fg(5) // 26
val R = gf(5) // 36
```

Example: Ben Bitdiddle wants to optimize the sum of squares

```
a.par.map(x => x*x).reduce(_ + _)
```

To avoid using both `map` and `reduce`, he introduces

```
def f(x:T, y:T) = x*x + y*y
```

and computes this sum as:

```
def sumsq(a: Array[T]): T = a.par.reduce(f)
```

Is his result correct?

```
sumsq(Array(1))      // 1
sumsq(Array(1,2))    // 5
sumsq(Array(1,2,3))  // 170, but could also be e.g. 34
```

Many operations are commutative but not associative

This function is 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 always the same on a parallel collection.

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 also 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) then 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 general trick for associativity

Associative operations on tuples

Theorem.

Suppose that `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.

Proof:

```
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)))
```

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 a of integers, compute the average of its values

```
def average(a: List[Int]) =
    val sum = a.par.reduce(_ + _)
    val length = a.par.map(x => 1).reduce(_ + _)
    sum/length
```

This uses two reductions: one for sum, one for length.

Is there a solution using a single map and a single reduce?

- also, avoid using fractions in intermediate steps

Average using only one reduction

Use pairs that compute sum and length at once

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

Function `f` is associative by the theorem, because + is associative.

A solution is then:

```
def average2(a: List[Int]) =
  def f(x: (Int,Int), y: (Int,Int)) = (x._1 + y._1, x._2 + y._2)
  val (sum,length) = a.map(x => (x,1)).reduce(f)
  sum/length
```

Exercise: express average2 using only one call to `aggregate` method. Prove that the arguments to `aggregate` satisfy the necessary algebraic properties (see the exercise session for properties of `aggregate`).

Another design of collections library may need different laws

Consider the following implementation of sort-reduce:

```
extension[E] (arr: Array[E])
  def sreduce(less: (E,E) => Boolean)(op: (E,E) => E) =
    val shuffled = arr.sortWith(less) // may use another way to reorder
    shuffled.par.reduce(op)
```

What laws does `op` need to satisfy?

```
val arr1 = Array(10,3,100).sreduce(_ <= _)(_ + _)  // 113
```

In general, is associativity enough to get same result as `reduce` ?

Take a collection of integers with a lexicographic order

```
def lexLess(l1: List[Int], l2: List[Int]): Boolean = (l1, l2) match
    case (Nil, _) => true
    case (_, Nil) => false
    case (h1::t1, h2::t2) =>
      if h1 < h2 then true
      else if h1 > h2 then false
      else lexLess(t1, t2)

val arrOfLists = Array(List(1,2), List(30,20,10), List(7))
val arr2 = arrOfLists.sreduce(lexLess)(_ ++ _)

// === arrOfLists.sortWith(lexLess).reduce(_ ++ _)
// === Array(List(1,2), List(7), List(30,20,10)).reduce(_ ++ _)
// === List(1, 2, 7, 30, 20, 10)
```

Different from: `arrOfList.reduce == Array(1, 2, 30, 20, 10, 7)`

Operations that reorder elements arbitrarily vs Scala parallel sequences

Consider reduce that may change the order of elements, e.g. `sreduce`

```
val shuffled = arr.sortWith(less) // may use another way to reorder
shuffled.par.reduce(op)
```

including changing the order in response to when tasks complete.

Such operations require both:

- associativity
- commutativity

`reduce` in Scala's parallel sequences (ParArray, ParVector) do **not** shuffle arbitrarily; only vary the way they split sequences during reduction – only need associativity.

- we can use them to `.par.reduce` with operations such as matrix multiplication

How Fast are (Parallel) Programs?

CS-214 Software Construction

How long does our computation take?

Performance: a key motivation for parallelism

How to estimate it?

- empirical measurement:
  - can use `System.currentTimeMillis`, repeat and find average
  - impact of JIT, GC, class loading, thread scheduling, caches, frequency scaling
  - for caches, see `https://gist.github.com/jboner/2841832`
- asymptotic analysis

Asymptotic analysis is important to understand how algorithms scale when:

- inputs get larger
- we have more hardware parallelism available

Asymptotic analysis of sequential running time

You may have previously learned how to concisely characterize behavior of functions and programs using the number of operations they perform as a function of arguments.

- inserting into an integer into a sorted linear list takes time $O(n)$, for list storing n integers
- inserting into an integer into a balanced binary tree of n integers takes time $O(\log n)$, for tree storing n integers

More information: Kenneth H. Rosen, Discrete Mathematics And Its Applications, 8th ed., 2019

- 3.2 Growth of Functions
- 3.3 Complexity of Algorithms

and you will expand on this in CS-250 (Algorithms I).

Let us review worst-case complexity on the sum-segment example

Asymptotic analysis of sequential running time

Find time bound on sequential `sumSegment` as a function of `s` and `t`

```
def sumSegment(a: Array[Int], p: Double, s: Int, t: Int): Int =
  var i= s; var sum: Int = 0
  while i < t do
    sum= sum + power(a(i), p)
    i= i + 1
  sum
```

The answer is: $O(t-s)$, bounded by a function of the form: $c_1(t-s)+c_2$ for some constants c_1, c_2

- $t-s$ loop iterations
- a constant amount of work in each iteration

Big O notation

Definition: We say that a function $p(n)$ is $O(g(n))$ if there is a constant M and some strating point n_0 such that

$$p(n) \leq M \cdot g(n)$$

for all $n \geq n_0$.

For example, $100n$ is $O(n^2)$ because $100n \leq n^2$ for $n \geq 10$.

$100n$ is also $O(n)$ because $100n \leq 100 \cdot n$ (so, $M = 100$) for all $n \geq 0$.

- we often state the best $O(f(n))$ we know, but that is not part of the definition

Functions are ordered hierarchically from slower growing ones to faster growing ones:

$\log(n)$, n, $n\log(n)$, n^2, n^3, 2^n

Work and depth

We would like to speak about the asymptotic complexity of parallel code

- but this depends on available parallel resources (e.g. number of CPU cores)
- we introduce *two measures* for a program

Work $W(e)$: number of steps e would take if there was no parallelism

- this is simply the sequential execution time
- treat all `parallel(e1,e2)` as `(e1,e2)` because all work still needs to be done

Depth $D(e)$: number of steps if we had unbounded parallelism (infinitely many cores)

- we take **maximum** of running times for arguments of `parallel`
- if we split work into two, we may be done twice as soon

Key insight: if the depth is large, no amount of parallel processors will make code fast

Rules for depth and work

Key rules are:

- $W(parallel(e_1, e_2)) = W(e_1) + W(e_2) + c_2$
- $D(parallel(e_1, e_2)) = \max(D(e_1), D(e_2)) + c_1$

If we divide work in equal parts, for depth it counts only once!

For parts of code where we do not use `parallel` explicitly, we must add up costs. For function call or operation $f(e_1, ..., e_n)$:

- $W(f(e_1, ..., e_n)) = W(e_1) + ... + W(e_n) + W(f)(v_1, ..., v_n)$
- $D(f(e_1, ..., e_n)) = D(e_1) + ... + D(e_n) + D(f)(v_1, ..., v_n)$

Here v_i denotes values of e_i. If f is primitive operation on integers, then $W(f)$ and $D(f)$ are constant functions, regardless of v_i.

Note: we assume (reasonably) that constants are such that $D \leq W$

Analysis of work and depth for segmentPar

We used the following recursive function to illustrate divide and conquer parallelism.

In this version, I replace `until` parameter with `len` (just for the analysis)

```
def segmentPar(xs: Array[T], p: Double, from: Int, len: Int): Int =
  if len < threshold then sumSegment(xs, p, from, from + len)
  else val (l, r) = parallel(segmentPar(xs, p, from,         len/2),
                             segmentPar(xs, p, from + len/2, len - len/2))
       l + r
```

Computation Tree: work (W) is its size, depth (D) is its height

```
def segmentPar(xs: Array[T], p: Double, from: Int, len: Int): Int =
  if len < threshold then sumSegment(xs, p, from, from + len)
  else val (l, r) = parallel(segmentPar(xs, p, from,          len/2),
                             segmentPar(xs, p, from + len/2, len - len/2))
       l + r
```

Analyzing **work** of segmentPar

Work only depends on *len*, denote it by L (assume L is a power of two):

$$W(L) = \begin{cases} O(L), & \text{if } L < \text{threshold} \\ 2W(L/2) + c & \text{otherwise} \end{cases}$$

For $L = 2^N$:

$$\begin{aligned} W(2^N) &= c + 2W(2^{N-1}) = c + 2(c + 2W(2^{N-2})) = \\ &= c(1 + 2^1 + 2^2 + \ldots + 2^{N-k-1}) + 2^{N-k}O(2^k) = \\ &\quad \text{(work of recursive calls)} \quad + \text{(work in leaves)} \\ &= c(2^{N-k} - 1) + O(2^N) \\ &= O(2^N) = O(L) \end{aligned}$$

We split the work, but both parts need to be done, so work remains linear.

Analyzing **depth** of segmentPar

```
def segmentPar(xs: Array[T], p: Double, from: Int, len: Int): Int =
  if len < threshold then sumSegment(xs, p, from, from + len)
  else val (l, r) = parallel(segmentPar(xs, p, from,          len/2),
                             segmentPar(xs, p, from + len/2, len - len/2))
       l + r
```

$$D(L) = \begin{cases} O(L), & \text{if } L < \text{threshold} \\ \max(D(L/2), D(L/2)) + c & \text{otherwise} \end{cases}$$

$$D(L) = \begin{cases} O(L), & \text{if } L < \text{threshold} \\ D(L/2) + c & \text{otherwise} \end{cases}$$

$$D(2^N) = c + D(2^{N-1}) = c + c + D(2^{N-2}) = \ldots = c(N - k - 1) + O(\mathit{threshold}) = O(N)$$

Bounding depth of segmentPar

Since $L = 2^N$, we have $N = \log(L)$

The depth is only logarithmic, $O(\log(L))$

Note: if the input is given to one processor, we cannot transform, e.g., a linear-time algorithm into a constant time one using the `parallel` construct.

- you need to spend time dividing the work and combining the results
- divide and conquer algorithms are a way to do it

Another Example

Suppose the input to sumAll is an N times N matrix, represented as a list of lists:

```
def sumAll(m: List[List[Int]]): Int =
  m match
    case Nil => 0
    case x :: xs =>
      val (k, ks) = parallel(x.sum, sumAll(xs))
      k + ks
```

What is the work and depth of sumAll as a function of N ?

Assume x.sum is sequential (does not use `parallel`)