Week 6: Principles of Parallelism

CS-214 Software Construction

Outline

- What is parallelism and why we need it
- Maps on parallel collections. Parallel matrix multiply
- Divide and conquer using “parallel”. Array norm.
- Reduce. Associativity

What is Parallelism and Why We Need It

Parallelism

Parallelism = multiple computations happen *at once* (*simultaneously*) in physically different (parts of) devices.

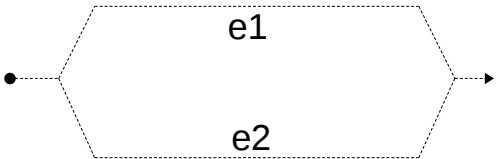
Examples of physical parallelism:

- several connected computers (cluster, cloud)
- multiple cores in one chip (most important for this lecture!)
  - 4 cores: i7-1165G7
  - 20 cores: Intel® Xeon® E5-2680
  - 96 cores: AMD Ryzen® Threadripper® Pro 7000 WX
- Thousands of threads on a GPU (used for graphics, ML)
- Intel® AVX-512 Vector instructions that work on 512 bits at a time
  - Scala/JVM Int is 32 bit, Long is 64 bit number
- FPGA (e.g. AMD® Alveo U200, over 800k LUTs)
  - could put 200+ simple softcore CPUs (like LEON3)

Parallel vs Sequential Programming

Sequential computation: split task into two steps: e1; e2

Parallelism: split into *independent* tasks (fork), solve in parallel, combine (join):



Parallel programming is difficult:

- It subsumes sequential programming
- Need to think which parts of computation are independent
- Different parts of hardware need to communicate
- Its efficiency depends on hardware details more

Historical Context of Parallel Programming

First established theoretical models were sequential:

- Mathematical computations explained by humans, step by step, as humans have only one mouth, one verbalized train of thoughts
- *Turing machine* is well-accepted but sequential model of algorithms:
  - read a letter on a tape, write a letter, change the state, repeat
- our substitution model as a sequential sequence of steps

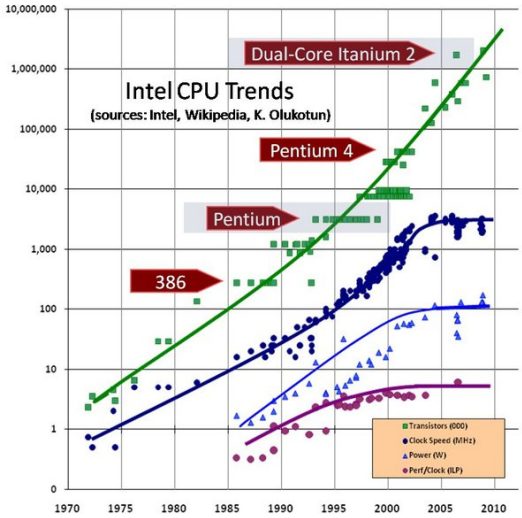
First physical computers were sequential:

- It was hard enough to build a sequential computer with vaccum tubes

In commercial industry, CPUs were getting faster regularly:

- Moore's law (Intel): transistors can be made smaller, more fit in a given area
- Dennard (DRAM) scaling: smaller transistor needs lower voltage, less energy; can switch faster (GHz)

End of Dennard Scaling in Early 2000s



More transistors, but not faster.

Multicore Era

We no longer know how to run CPUs at higher clock frequency

- they would melt due to generated heat

Instead, we develop hardware with:

- more complex cores, which try to automatically parallelize a few consecutive instructions (pipelined superscalars)
- multiple cores that all run in parallel $\Longleftarrow$

Energy use became even more important: mobile, sustainability

If the same problem can be split into two and solved with cores half the speed, this is more energy efficient!

- modern mobile phones: many cores, lower frequency (battery!)
- training a LLM: "up to 10 gigawatt-hour (GWh)''
  `https://www.washington.edu/news/2023/07/27/how-much-energy-does-chatgpt-use/`

Operating Systems (OS) Runs Threads on Cores

OS sits between programs and machine. A key role: handle *processes*

- run user computation (move a boid, play a game, run web browser)
- handle mouse and keyboard, draw on screen
- read/write files, send/receive network packets

How does OS do it?

- preemptive multi-tasking (time-slicing): OS gives a slice of time to each process then interrupts it to give chance to others
- when there are multiple cores: each core executes a different process at the same time $\rightarrow$ parallelism

Java Virtual Machine (JVM) can start *threads*, which run on multiple OS processes.

When OS schedules those processes on multiple cores, we get parallelism!

Example with Threads

```
class MyThread(val k: Int) extends Thread:
  override def run: Unit =
    var i: Int = 0
    while i < 6 do
      println(f"${getName} has counter ${i}")
      i += 1
      Thread.sleep(3)

def testThread: Unit =
  val t1 = MyThread(0)
  val t2 = MyThread(1)
  t1.start; t2.start
  Thread.sleep(4)
  t1.join; t2.join
```

Race Condition: Loser Writes the Final Result

```
var result = 0
class MyThread(val k: Int) extends Thread:
  override def run: Unit =
    var i: Int = 0
    while i < 6 do
      println(f"${getName} has counter ${i}"); i += 1
      Thread.sleep(3)
    result = k

def testThread: Unit =
  val t1 = MyThread(0); val t2 = MyThread(1)
  t1.start; t2.start
  Thread.sleep(4)
  t1.join; t2.join
  println(f"result = ${result}")
```

Imperative vs Functional Parallel Programs

Race condition is when a thread writes to a variable while another thread reads or writes to it.

- result changes from run to run
- result can even be ill-defined (one thread reads half-old half-new value)

To avoid race conditions, threads often use synchronization constructs (waiting on a lock), but this can lead to deadlocks, where program freezes.

Functional programming as a solution:

- functions compute values instead of writing to variables
- given h(f(x), g(x)) we can compute f(x) and g(x) in parallel (in different threads)
  - this would not be true if f writes to a var that g reads

Implicit vs Explicit Parallelism

Implicit parallelism:

Programming language compiler and runtime decide what to run in parallel.

- example: parallel Haskell (pH), various past research projects
- it would be wonderful, but so far not efficient

Explicit parallelism: programmer indicates which parts to run in parallel

Two important requirements for parallel programs make it hard:

- result should be correct: e.g. same as some sequential program
- computation should be faster (otherwise we worked for nothing)

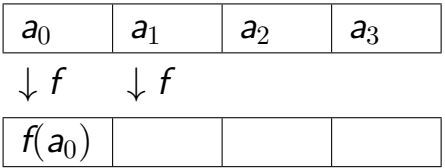
Our approach is explicit but high-level: parallel collections

- manipulate high-level collections (e.g. vectors)
- write program using higher order operations (map, reduce)
- to make them parallel, request a collection to be parallel (using `.par`)

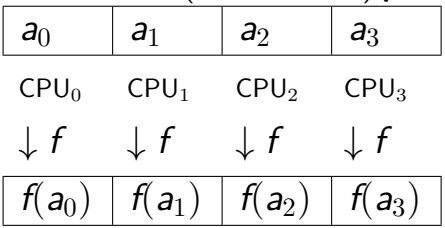
Maps on Parallel Collections

Map on a Collection Can Run in Parallel

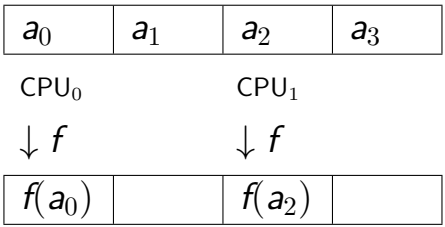
Sequential: Vector(a0, a1, a2, a3).map(f)



Parallel: Vector(a0, a1, a2, a3)**.par**.map(f)



or



Scala Parallel Collections Library

Hosted at

`https://github.com/scala/scala-parallel-collections`

To use it, in your `build.sbt` file add:

```
libraryDependencies ++=
  Seq("org.scala-lang.modules" %% "scala-parallel-collections" % "1.2.0")
```

To enable easy conversion of collections to parallel collections, use

```
import scala.collection.parallel.CollectionConverters.*
```

Parallel collections support the same operations as usual collections

But they try to run these operations in parallel, using multiple CPU cores.

Selected Parallel Collections with Examples

```
val r = (1 until 10) : Range                    | r.par : ParRange
val a = Array.tabulate(3)(i => i.toDouble*0.5)  | a.par : ParArray[Double]
=== Array(0.0, 0.5, 1.0)                        | === ParArray(0.0,0.5,1.0)
val v = a.toVector: Vector[Double]              | v.par : ParVector[Double]
```

Array of arrays (matrix):

```
val m = Array.tabulate(5)(i => Array.tabulate(3)(j => i + j))
val m: Array[Array[Int]] = Array(Array(0, 1, 2),
                                 Array(1, 2, 3),
                                 Array(2, 3, 4),
                                 Array(3, 4, 5),
                                 Array(4, 5, 6))
```

Example: Matrix Multiply

```
type Matrix[A] = Array[Array[A]]
def prod(m1: Matrix[T], m2: Matrix[T])(i: Int, j: Int): T =
```

$$\sum_k m_1(i)(k) * m_2(k)(j)$$

Sequential matrix multiply:

```
def seqMultiply(m1: Matrix[T], m2: Matrix[T]): Matrix[T] =
  (0 until m1.length).map(i =>
    Array.tabulate(m2(0).length)(j => prod(m1, m2)(i, j))).toArray
```

Parallel matrix multiply (spot the difference!):

```
def parMultiply(m1: Matrix[T], m2: Matrix[T]): Matrix[T] =
  (0 until m1.length).par.map(i =>
    Array.tabulate(m2(0).length)(j => prod(m1, m2)(i, j))).toArray
```

Parallel Matrix Multiply Performance

Now, spot the difference in running time. Which one is which?

On a Xeon server with 20 cores:

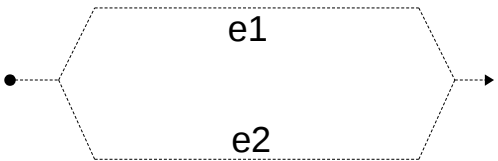
```
First version:
Time: 27.30 ms average.

Second version:
Time: 3.90 ms average.
```

Divide and conquer using “parallel”. Array norm

Simplest construct for parallel computation

Given expressions `e1` and `e2`, compute them in parallel and return the pair of results

```
parallel(e1, e2)
```



Implementation of parallel using collections

```
def parallel[A](e1: => A, e2: => A): (A,A) =
  val both = ParArray(1,2).map(i => if i == 1 then e1 else e2)
  (both(0), both(1))
```

Alternative:

```
def parallel1[A](e1: => A, e2: => A): (A,A) =
  val both = ParArray(() => e1, () => e2).map(f => f())
  (both(0), both(1))
```

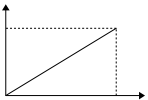
We can also implement `parallel` using (virtual) threads; we do not discuss that.

Example: computing p-norm

Given a vector as an array (of integer type `T`), compute its p-norm

A *p-norm* is a generalization of the notion of *length* from geometry

2-norm (p=2) of a two-dimentional vector (a_1, a_2) is $\left(a_1^2 + a_2^2\right)^{1/2}$



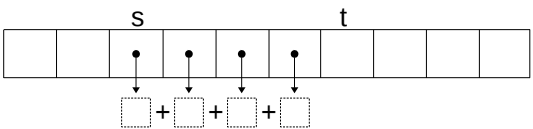
The p-norm of a vector $(a_1, \ldots, a_n)$ is $\left(\sum_{i=1}^{n} |a_i|^p \right)^{1/p}$

Main step: sum of powers of array segment

First, solve *sequentially* the following `sumSegment` problem: given

- an integer array `a`, representing our vector
- a positive double floating point number `p`
- two valid indices `s <= t` into the array `a`

compute $\sum_{i=s}^{t-1} \lfloor |a_i|^p \rfloor$ where $\lfloor y \rfloor$ rounds down to an integer



Sum of powers of array segment: solution

The main function is

```
def sumSegment(a: Array[T], 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
```

Here power computes $\lfloor |x|^p \rfloor$:

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

Given `sumSegment(a,p,s,t)`, how to compute p-norm?

$$||a||_p := \left(\sum_{i=0}^{N-1} \lfloor |a_i|^p \rfloor \right)^{1/p}$$

where `N = a.length`

Given `sumSegment(a,p,s,t)`, how to compute p-norm?

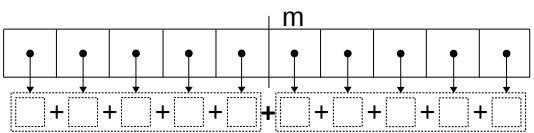
$$||a||_p := \left(\sum_{i=0}^{N-1} \lfloor |a_i|^p \rfloor \right)^{1/p}$$

where `N = a.length`

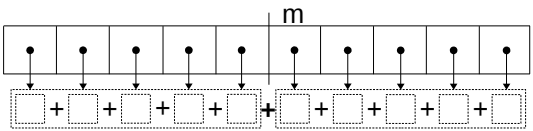
```
def pNorm(a: Array[T], p: Double): Int =
  power(sumSegment(a, p, 0, a.length), 1/p)
```

Observe that we can split this sum into two

$$||a||_p := \left(\sum_{i=0}^{N-1} \lfloor |a_i|^p \rfloor \right)^{1/p} = \left(\sum_{i=0}^{m-1} \lfloor |a_i|^p \rfloor + \sum_{i=m}^{N-1} \lfloor |a_i|^p \rfloor \right)^{1/p}$$



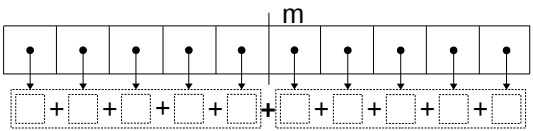
Using sumSegment twice



The resulting function is:

```
def pNormTwoPart(a: Array[T], p: Double): Int =
  val m = a.length / 2
  val (sum1, sum2) =          (sumSegment(a, p, 0, m),
                               sumSegment(a, p, m, a.length))
  power(sum1 + sum2, 1/p)
```

Making two sumSegment invocations parallel

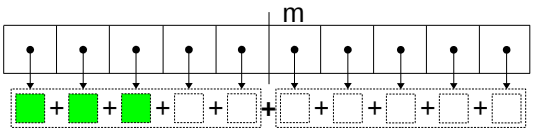


The resulting function is:

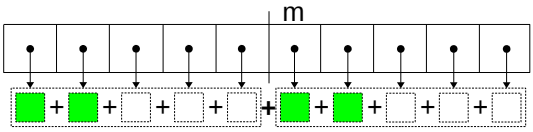
```
def pNormTwoPart(a: Array[T], p: Double): Int = {
  val m = a.length / 2
  val (sum1, sum2) = parallel(sumSegment(a, p, 0, m),
                              sumSegment(a, p, m, a.length))
  power(sum1 + sum2, 1/p) }
```

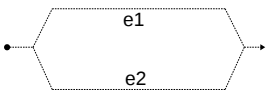
Comparing execution of two versions

```
val (sum1, sum2) = (sumSegment(a, p, 0, m),
                    sumSegment(a, p, m, a.length))
```

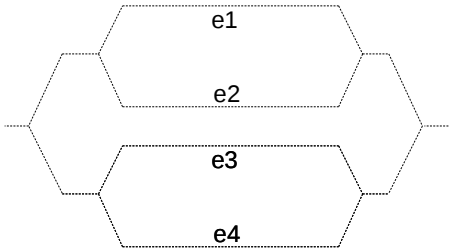



```
val (sum1, sum2) = parallel(sumSegment(a, p, 0, m),
                            sumSegment(a, p, m, a.length))
```

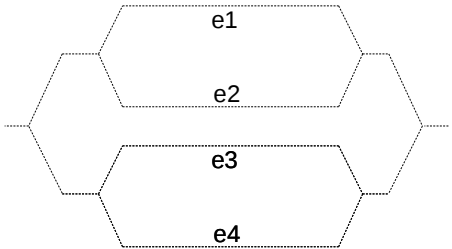




How to process four array segments in parallel?



How to process four array segments in parallel?




```
val m1 = a.length/4; val m2 = a.length/2; val m3 = 3*a.length/4
val ((sum1, sum2),(sum3,sum4)) =
  parallel(parallel(sumSegment(a, p, 0, m1), sumSegment(a, p, m1, m2)),
           parallel(sumSegment(a, p, m2, m3), sumSegment(a, p, m3, a.length)))
```

Is there a recursive algorithm for an unbounded number of threads?

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

// like sumSegment but parallel
def segmentPar(a: Array[T], p: Double, s: Int, t: Int) =
  if (t - s < threshold)
    sumSegment(a, p, s, t)  // small segment: do it sequentially
  else
    val m = s + (t - s)/2
    val (sumLeft, sumRight) = parallel(segmentPar(a, p, s, m),
                                       segmentPar(a, p, m, t))
    sumLeft + sumRight
```

Reduce. Associativity

Map and Fold

We have seen operation:

map: apply function to each element

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

We now consider:

fold: combine elements with a given operation

- `Array(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

```
Array(1,3,8).foldLeft(100)((s,x) => s - x) == ((100 - 1) - 3) - 8 == 88
```

```
Array(1,3,8).foldRight(100)((s,x) => s - x) == 1 - (3 - (8-100)) == -94
```

```
Array(1,3,8).reduceLeft((s,x) => s - x) == (1 - 3) - 8 == -10
```

```
Array(1,3,8).reduceRight((s,x) => s - x) == 1 - (3 - 8) == 6
```

To enable parallel operations, today we look at **associative** operations

- addition, string concatenation (but not minus)
- instead of `reduceLeft` or `reduceRight` we will have simply `reduce`

Idea of parallel reduce

We would like to obtain the following:

- allow parallel sequence implementation to decide how to split the sequence into parts that it will do in parallel
- ensure that no matter how this splitting happens, the result is the same

We will

- define associativity
- reduce for trees: here, the result is obviously unique
- how associativity allows us to imagine any tree within a sequence

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:

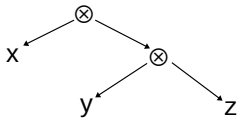
$$(x \otimes y) \otimes (z \otimes w) = (x \otimes (y \otimes z)) \otimes w = ((x \otimes y) \otimes z) \otimes w$$

Trees for expressions

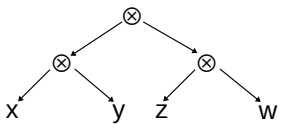
Each expression built from values connected with $\otimes$ can be represented as a tree

- 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?

```
def reduce[A](t: Tree[A], f: (A,A) => A): A =
  t match
    case Leaf(v) => v
    case Node(l, r) => f(reduce(l, f), reduce(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 tree1 = Node(Leaf(1), Node(Leaf(3), Leaf(8)))
def fMinus = (x:Int,y:Int) => x - y
val res1 = reduce(tree1, fMinus)
```

What is `res1`? 6

On the other hand:

```
def tree2 = Node(Node(Leaf(1), Leaf(3)), Leaf(8)))
def res = reduce(tree2, fMinus)
```

What is `res2`? -10

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)
```

Associativity stated as tree reduction

How can we restate associativity of such trees?

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

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: we 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: if `f : (A,A)=>A` is associative, `t1:Tree[A]` and `t2:Tree[A]` and if `toList(t1)==toList(t2)`, then:

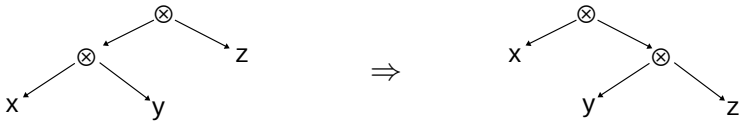
```
reduce(t1, f)==reduce(t2, f)
```

Proof would be by induction. Some intuition follows.

Explanation of the consequence

Intuition: given a tree, use tree rotation until it becomes list-like.

Associativity law says tree rotation preserves the result:



Example use:

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?

- treat array as a balanced tree (left half and right half)
- 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: divide and conquer (compare to array norm)

```
def reduceSeg[A](inp: Array[A], left: Int, right: Int, f: (A,A) => A): A =
  if right - left < threshold then
    var res= inp(left); var i= left+1
    while i < right do
      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)
```

Conclusion: why associativity

Parallel computation splits collection into pieces

Depending on the split, we obtain different computation trees

- depending on parallel collection implementation
- depending on the available parallel cores

To ensure that the result is predictable, we want all these trees to give the same result.

Associativity ensures that all these trees give the same result.

In practice, no need to implement your own parallel reduce.

Use parallel collections:

- you can do `Array(1,3,8).par.reduce(_ + _)`
- please do not do `Array(1,3,8).par.reduce(_ - _)`

Associativity vs Commutativity

Array Norm Using Parallel Collections

Our array norm example computes first:

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

How can we compute this using parallel collections?

Suppose `a` is an ordinary array.

```
a.par.map(power(_, p)).reduce(_ + _)
```

Here $+$ is the operation given to `reduce`

- operation given to reduce of a parallel collection must be **associative**

Associative operation

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

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

that is, if we write f(u,v) as $u \oplus v$,

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

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

A **different** algebraic property: commutativity

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

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

that is, if we write f(u,v) as $u \oplus v$,

$$x \otimes y = y \otimes x$$

There are operations that are associative but not commutative

There are operations that are commutative but not associative

For correctness of `reduce` in Scala parallel collection, we need only 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 `BigInt`s)
- addition and multiplication modulo a positive integer (e.g. 2^{32}), including the usual arithmetic on 32-bit `Int` or 64-bit `Long` values in Scala
- union, intersection, and symmetric difference of sets ($\cup, \cap, \Delta$)
- union of bags (multisets) that preserves duplicate elements
- Boolean operations $\&\&, ||$, exclusive or (`!=` on Boolean)
- addition of vectors (of `Int`)
- addition and multiplication of polynomials (with e.g. `Int` coefficients)
- addition of matrices of fixed dimension (e.g. all 2x3 matrices of `Int`)

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 sequences of `Char`)
- matrix multiplication AB for square matrices A and B of fixed 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.

Example: matrix multiplication is not commutative

```
val ma = Array(Array(1L,1L),
               Array(0L,0L))

val mb = Array(Array(1L,0L),
               Array(1L,0L))

val mab = seqMultiply(ma, mb) // Array(Array(2, 0), Array(0, 0))
val mba = seqMultiply(mb, ma) // Array(Array(1, 1), Array(1, 1))
```

(to be continued on Wednesday)