# Parallel Programming

# Different levels of parallelism

- Parallelism manifests itself at different granularity levels
  - bit-level parallelism
    - processing multiple bits of data in parallel
  - instruction-level parallelism
    - executing different instructions from the same instruction stream in parallel
  - task-level parallelism
    - executing separate instruction streams in parallel
- We will focus on programming techniques for task-level parallelism

# Classes of Parallel Computers

- Many different forms of parallel hardware
  - multi-core processors
  - symmetric multiprocessors
  - general purpose graphics processing unit
  - field-programmable gate arrays
  - computer clusters
- We will focus (in this section) on programming techniques for multi-cores and symmetric multiprocessors
  - GPUs and FPGAs have their own programming frameworks (eg. CUDA, OpenCL) and languages (eg. HDLs like Verilog)
- We will consider cluster computing in the last part of our course

# JVM parallelism

- JVM is based on thread-based parallelism
  - Each process can contain multiple independent concurrency units called threads
  - Threads can be started from within the same program, and they share the same memory address space
  - Each thread has a program counter and a program stack
  - JVM threads cannot modify each other's stack memory

# Creating, starting and waiting for threads

- Each JVM process starts with a main thread
- To start additional threads:
  - 1. Define a Thread subclass
  - 2. Instantiate a new Thread object
  - 3. Call start on the Thread object
- The Thread subclass defines, in a run method, the code that the thread will execute

```
class HelloThread extends Thread {
  override def run() = {
    println("Hello world!")
  }
}
val t1 = new HelloThread; val t2 = new HelloThread
t1.start(); t2.start()
t1.join(); t2.join()
```

# Memory model

- Memory model is a set of rules that describes how threads interact when accessing shared memory
- The memory model for the JVM:
  - Two threads writing to separate locations in memory do not need synchronization
  - In order for a thread X to observe all the writes by thread Y, it is necessary to add synchronization points (e.g. X calls join on Y before reading)

```scala
val vector = Array.fill(10000)(0)
class th extends Thread {
  override def run() = {
    for (i <- 0 until 10000) {vector(i)=i;Thread.sleep(1)}
  }
}
val thread = new th; thread.start; Thread.sleep(5000)
vector.max
```

# Parallel computation

- Assume to have a parallel function (implemented using concurrency libraries out of the scope of this course):

```
def parallel[A, B](taskA: => A, taskB: => B): (A, B)
```

  - that evaluates two expressions in parallel, and returns the pair composed of the two results
- Assume you have to compute the p-norm of a vector:

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

  - It is possible to compute the summation in parallel on distinct segments of the vector

# Parallel computation of the norm

```
def sumSegment(a: Array[Double], p: Double, s: Int, t: Int):
Double = {
  var i = s; var sum: Double = 0
  while (i < t) {
    sum = sum + Math.pow(a(i), p)
    i = i + 1
  }
  sum
}

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

# Unbounded parallel computation

```
def segmentRec(a: Array[Double], p: Double, s: Int, t: Int):
Double = {
  if (t - s < threshold)
    sumSegment(a, p, s, t) // small segment done sequentially
  else {
    val m = s + (t - s) / 2
    val (sum1, sum2) = parallel(
      segmentRec(a, p, s, m),
      segmentRec(a, p, m, t))
    sum1 + sum2
  }
}

def pNormRec(a: Array[Double]): Double =
  Math.pow (segmentRec(a, a.length, 0, a.length), 1 / a.length)
```

# Parallel computation: what happens?

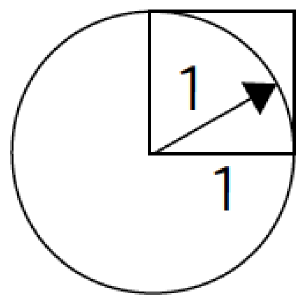
- Efficient parallelism requires support from
  - hardware
  - operating system
  - virtual machine (as in the case of JVM)
  - language
- Given sufficient resources, computation is parallelised and the parallel program can run faster

# Hardware architecture matters

- Assume we consider the same pattern of computation, but only summing without applying power
- The speed-up does not always increases, why?
  - The computation requires more frequent access to the shared memory
  - Memory, in this case, becomes a bottleneck

# Example: estimation of pi

- Consider a square of edge one and a circle of radius one with center in the bottom-left corner of the square:




  - The ratio between the surface of ¼ of the circle and the square:

$$\pi / 4$$

  - Estimation of pi:
    - random sample points inside the square
    - count the frequency of those that fall inside the circle
    - multiply it by 4

# Sequential implementation

```scala
import scala.util.Random

def mcCount(iter: Int): Int = {
  val randomX = new Random
  val randomY = new Random
  var hits = 0
  for (i <- 0 until iter) {
    val x = randomX.nextDouble  // in [0,1]
    val y = randomY.nextDouble  // in [0,1]
    if (x * x + y * y < 1) hits = hits + 1
  }
  hits
}

def monteCarloPiSeq(iter: Int): Double =
  4.0 * mcCount(iter) / iter
```

# Parallel implementation

- We can split the computation in parallel tasks:
  - Here is a parallel implementation with four parallel tasks

```
def monteCarloPiPar(iter: Int): Double = {
  val (pi1, pi2, pi3, pi4) = parallel(
      mcCount(iter / 4),
      mcCount(iter / 4),
      mcCount(iter / 4),
      mcCount(iter / 4)
    )
  4.0 * (pi1 + pi2 + pi3 + pi4) / iter
}
```

# Exercise

- Write a parallel implementation of MergeSort

# Exercise

- Write a parallel implementation of MergeSort

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

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

  def sort(from: Int, until: Int, depth: Int): Unit = {
    if (depth == maxDepth) {
      quickS(xs, from, until)
    } 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)
    }
  }
 ...
}
```

# How fast are parallel programs?

- Parallel versions of algorithms can be faster:
  - Can we perform an analytical analisis of the gain, following an approach like asymptotic complexity for sequential algorithms?
- Time complexity T(n) of sequential algorithms quantifies the expected number of elementary operations to execute for an input of size n
  - Such operations are executed is sequence (one at a time)
- For parallel algorithms, we use two measures:
  - Work complexity: W(n) (the same as the above T(n))
    Expected number of elementary operations for an input of size n
  - Depth complexity: D(n)
    Expected length of the longest sequence of elementary operations that must be executed in sequence  for an input of size n

# Work and depth complexity

- We consider the sumSegment example seen before:

```
def sumSegment(a: Array[Double], p: Double, s: Int, t: Int):
Double = {
  var i = s; var sum: Double = 0
  while (i < t) {
    sum = sum + Math.pow(a(i), p)
    i = i + 1
  }
  sum
}
```

Time complexity in the sequential case: $T(t-s) = c_1(t-s)+c_2$

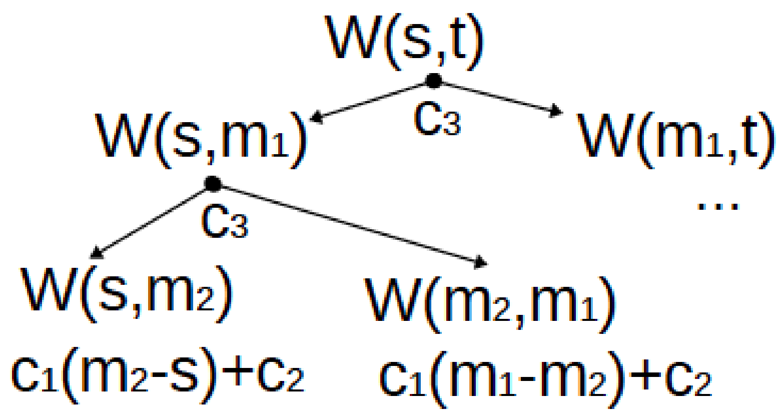
# Work and depth complexity

- Work complexity of the parallel recursive version:

```
def segmentRec(a: Array[Double], p: Double, s: Int, t: Int):
Double = {
  if (t - s < threshold)
    sumSegment(a, p, s, t)
  else {
    val m = s + (t - s) / 2
    val (sum1, sum2) = parallel(
      segmentRec(a, p, s, m),
      segmentRec(a, p, m, t))
    sum1 + sum2
  }
}
```

$$W(s,t) = \begin{cases} c_1(t-s) + c_2, & \text{if } t - s < \text{threshold} \\ W(s,m) + W(m,t) + c_3 & \text{otherwise, for } m = \lfloor (s+t)/2 \rfloor \end{cases}$$

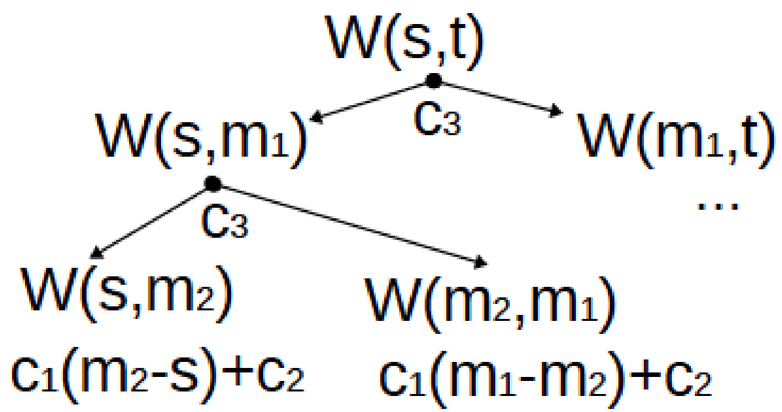
# Work and depth complexity




- Assume $s$, $t$ for which exists $N$ s.t. $t-s = 2^N(\text{threshold}-1)$
  - $N$ depth of the tree, having $2^N$ leaves and $2^N-1$ internal nodes
  - $W(s,t) = 2^N(c_1(\text{threshold}-1) + c_2) + (2^N-1)\, c_3 = 2^N c_4 + c_5$

$$W(s,t) = \begin{cases} c_1(t-s) + c_2, & \text{if } t-s < \text{threshold} \\ W(s,m) + W(m,t) + c_3 & \text{otherwise, for } m = \lfloor (s+t)/2 \rfloor \end{cases}$$

# Work and depth complexity




- Given any $s$, $t$, take $N$ s.t.
  $2^{N-1}(\text{threshold}-1) < (t-s) \le 2^{N}(\text{threshold}-1)$
  - As complexity increases, taking $s',t'$ s.t. $t'-s' = 2^{N}(\text{threshold}-1)$ we have $W(s,t) \le W(s',t') = 2^{N} c_4 + c_5$
  - From $2^{N-1}(\text{threshold}-1) < (t-s)$ we have $2^{N} < 2\,(t-s) / (\text{threshold}-1)$, hence we have $W(s,t) < 2\,(t-s) / (\text{threshold}-1)\, c_4 + c_5$
  - This means that $W(s,t)$ is in $O(t-s)$

# Work and depth complexity

- Depth complexity of the parallel recursive version:

```
def segmentRec(a: Array[Double], p: Double, s: Int, t: Int):
Double = {
  if (t - s < threshold)
    sumSegment(a, p, s, t)
  else {
    val m = s + (t - s) / 2
    val (sum1, sum2) = parallel(
      segmentRec(a, p, s, m),
      segmentRec(a, p, m, t))
    sum1 + sum2
  }
}
```

$$D(s,t) = \begin{cases} c_1(t-s) + c_2, & \text{if } t - s < \text{threshold} \\ \max(D(s,m), D(m,t)) + c_3 & \text{otherwise, for } m = \lfloor (s+t)/2 \rfloor \end{cases}$$

# Work and depth complexity

- Assume s, t for which exists N s.t. t-s = $2^N$(threshold-1)
  - N depth of the tree, having $2^N$ leaves and $2^N$-1 internal nodes
  - The value of D(s,t) in leaves is $c_1$(threshold-1) + $c_2$
  - One level above is: $c_1$(threshold-1) + $c_2$ + $c_3$
  - In the root, N levels above: $c_1$(threshold-1) + $c_2$ + N $c_3$ = N $c_3$ + $c_4$
- Assume N s.t. $2^{N-1}$(threshold-1) < (t-s) ≤ $2^N$(threshold-1)
  - As complexity increases, we have D(s,t) ≤ N $c_3$ + $c_4$
  - From $2^{N-1}$(threshold-1) < (t-s) we have
    N-1 < log( (t-s)/(threshold-1) ), hence N < log( (t-s) / $c_5$) + 1
  - Finally D(s,t) < (log( (t-s) / $c_5$) + 1) $c_3$ + $c_4$ which is in O(log(t-s))

$$D(s,t) = \begin{cases} c_1(t-s) + c_2, & \text{if } t-s < \text{threshold} \\ \max(D(s,m), D(m,t)) + c_3 & \text{otherwise, for } m = \lfloor (s+t)/2 \rfloor \end{cases}$$

# Number of phisically parallel threads

- Let P be the number of maximal parallel threads
- Can we bound running time as a function of P?

$$D(e) + \frac{W(e)}{P}$$

  - given the amount of work W(e), at least W(e)/P time is needed
  - if P goes to infinity, the depth D(e) any way remains
- Observations:
  - if P is fixed the complexity grows as the complexity of the sequential solution (it is fine to assume D(e) bound by W(e))
  - the existence of limits to the gain of parallelization (due to the presence of D(e)) is known as Amdahl's Law

# Amdahl's law

- Suppose a task is divided in two parts:
  - part1: a fraction f that cannot be speed-up
  - part2: the remaining 1-f that can be speed-up
- Speed-up: ratio between the sequential and the parallel execution time
- If we make part2 P times faster the speed-up is:

$$1/\left(f+\frac{1-f}{P}\right)$$

  - For example, if P=100 and f=0.4 we obtain 2.46
    - even if we speed-up the second part infinitely, we can obtain at most a global speed-up of 1/0.4 = 2.5

# Higher-order functions and parallelism

- Example: sequential and parallel version of map on Array

```scala
def mapArr[A,B](inp: Array[A], left: Int, right: Int,
                f : A => B, out: Array[B]): Unit = {
  var i= left
  while (i < right) {
    out(i)= f(inp(i))
    i= i+1
  }
}

def mapArrPar[A,B](inp: Array[A], left: Int, right: Int,
                   f : A => B, out: Array[B]): Unit = {
  if (right – left < threshold)
    mapArr(inp, left, right, f, out)
  else {
    val mid = left + (right – left)/2
    parallel(mapArrPar(inp, left, mid, f, out),
      mapArrPar(inp, mid, right, f, out))
  }
}
```

# Parallel map

- Parallelization on lists is inconvenient:
  - Split takes linear time (depth complexity is already linear!)
- Definition of a parallel map for the following trees
  - where the arrays in the leafs represent the partitions of the data in a collection
  - and the tree structure represents the order of split/combination of these partitions

```
trait 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
}
```

# Parallel map on trees

- Definition of a parallel map for the following trees

```
def mapTreePar[A,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) }
  }
```

  - Question: what is the depth complexity?

# Parallel map on trees

- Definition of a parallel map for the following trees

```scala
def mapTreePar[A,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) }
  }
```

  - Question: what is the depth complexity?
    - Answer: the height of the tree (hence logarithmic)

# Parallel Fold / Reduce

- We now move to fold/reduce higher order operations that apply a given operation to the elements in a 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
```

# Towards a parallel reduce

- We consider a parallel implementation of reduce on the previous trees

```
def reduceTreePar[A](t: ArrayTree[A], f: (A,A) => A) : A =
  t match {
    case ArrayLeaf(a) => {
      a.reduce(f)
    }
    case ArrayNode(l,r) => {
      val (lr,rr) = parallel(reduceTreePar(l,f),
                             reduceTreePar(r,f))
      f(lr, rr) }
  }
```

# Does the structure of the tree matters?

- Consider two trees with the same elements (in the same order) but with different structure:

```
def t1 =
  ArrayNode[Int](
    ArrayLeaf(Array(1)),
    ArrayNode(ArrayLeaf(Array(3)),ArrayLeaf(Array(8))))

def t2 =
  ArrayNode[Int](
    ArrayNode(ArrayLeaf(Array(1)),ArrayLeaf(Array(3))),
    ArrayLeaf(Array(8)))
```

# Does the structure of the tree matters?

- What are the values of the following expressions?

```
reduceTreePar(t1,(x:Int,y:Int) => x + y)
reduceTreePar(t2,(x:Int,y:Int) => x + y)

reduceTreePar(t1,(x:Int,y:Int) => x - y)
reduceTreePar(t2,(x:Int,y:Int) => x - y)
```

  - Why the structure matters only in the first case?

# Does the structure of the tree matters?

- What are the values of the following expressions?

```
reduceTreePar(t1,(x:Int,y:Int) => x + y)
reduceTreePar(t2,(x:Int,y:Int) => x + y)

reduceTreePar(t1,(x:Int,y:Int) => x - y)
reduceTreePar(t2,(x:Int,y:Int) => x - y)
```

  - Why the structure matters only in the first case?
    - Because minus is not associative, while plus is

# Associativity

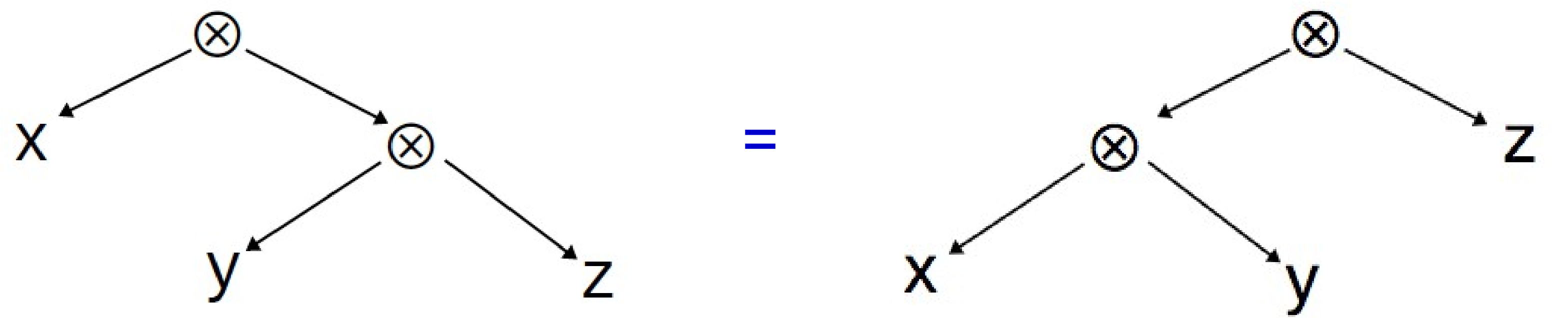
- Associativity:
  - an operation $f: (A,A) \Rightarrow A$ is associative iff for every $x$, $y$, $z$:

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

  - If we write $f(a, b)$ in infix form as $a \otimes$ .., associativity becomes

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

  - Graphically, using a tree representation:



# Property of associative operators

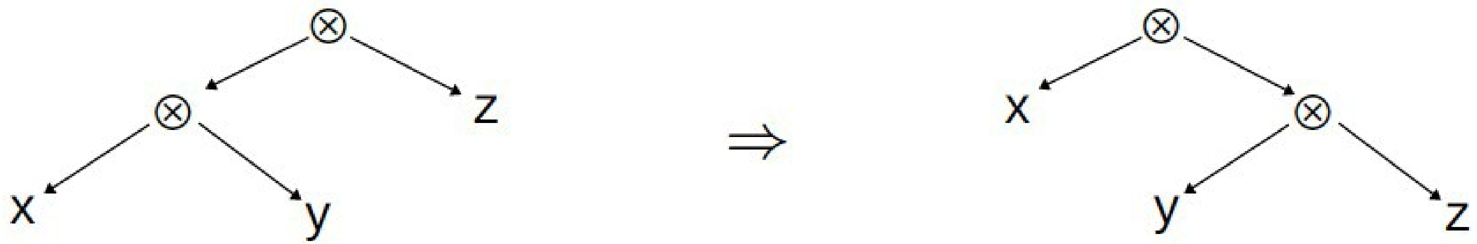
- Consider the following tree visit:

```
def toList[A](t: Tree[A]): List[A] = t match {
  case Leaf(v) => List(v)
  case Node(l, r) => toList(l) ++ toList(r) }
```

- Property of associativity:

  given f(A,A) => A associative and t1:Tree[A] and t2:Tree[A] such that toList(t1)==toList(t2), then reduce(t1,f) == reduce(t2,f)

  – Proof: applying rotation, trees can be put in list-like normal form




    • Note: rotation preserves toList and reduce (for associativity)

# Attention to associativity: floating point arithmetic

- Consider floating points values
  - Addition is not associative!

```
val e = 1e-200
val x = 1e200
val mx = -x

(x + mx) + e == x + (mx + e)
```

  - Multiplication is not associative!

```
val y = 1e-200
val z = 1e200

(y*z)*z == y*(z*z)
```

# 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)) == (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 also construct associative operations for n-tuples

# Example: average computation

- 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?
  - Solution: use pairs that compute sum and length at once

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

f is associative because addition is associative

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

# Reduce on Arrays

- How to implement a parallel reduce higher-order function on Arrays?
  - convert the Array into a balanced tree
    (to reduce depth complexity)
  - do tree reduction
- Attention:
  - works only for associative operations, for which we can choose any tree representation that preserves the order of elements
- Optimization:
  - It is not necessary to actually construct the tree
  - It is sufficient to apply directly the operator instead of the Node constructor

# Reduce on Arrays

- How to implement a parallel reduce higher-order function on Arrays?

```
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)
```

# Exercise

- Express the core of the computation of norm in terms of some fold/reduce higher-order function

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

# Exercise

- Express the core of the computation of norm in terms of some fold/reduce higher-order function

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

```
foldleft(0)((s,x)=>s+pow(abs(x),p))
```

  – Question: is it reasonable to expect a parallel implementation of `foldLeft`, like that of `reduce` ?

# Exercise

- Express the core of the computation of norm in terms of some fold/reduce higher-order function

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

`foldleft(0)((s,x)=>s+pow(abs(x),p))`

  - Question: is it reasonable to expect a parallel implementation of `foldLeft`, like that of `reduce` ?
    - NO: `foldLeft` is intrinsically sequential, due to the use of the accumulator

# Aggregate

- For parallel higher-order programming, fold-like operations are replaced by alternative functions like aggregate
  - On collections containing A objects, it is declared as:

```
def aggregate[B](z: =>B)(seqop: (B, A) => B, combop: (B, B) => B): B
```

  - How to implement it?
    - See next slide

# Aggregate on Arrays

```
def aggregateSeg[A,B](inp: Array[A], left: Int, right: Int,
a:B, f: (B,A) => B, g: (B,B) => B): B = {
  if (right - left < threshold) {
    var res = a; var i= left
    while (i < right) { res= f(res, inp(i)); i= i+1 }
    res
  } else {
    val mid = left + (right - left)/2
    val (a1,a2) = parallel(
      aggregateSeg(inp, left, mid, a, f, g),
      aggregateSeg(inp, mid, right, a, f, g))
    g(a1,a2)
  }
}

def aggregatePar[A,B](inp: Array[A],
                      a:B, f: (B,A) => B, g: (B,B) => B): B =
  aggregateSeg(inp, 0, inp.length, a, f, g)
```

# Exercise

- Express the core of the computation of norm in terms of the `aggregate` higher-order function

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

# Exercise

- Express the core of the computation of norm in terms of the `aggregate` higher-order function

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

```
aggregate(0)(_+pow(abs(_),p), _+_)
```

# Towards a parallel scanLeft

- Higher-order functions include scanLeft and scanRight

```
val l = List(1,3,8)
l.scanLeft(100)(_ + _)  //List(100, 101, 104, 112)
l.scanRight(100)(_ + _) //List(112, 111, 108, 100)
```

- We study the problem of implementing in parallel scanLeft
  - scanRight can be implemented symmetrically
  - let's start with a sequential implementation (on arrays)

```
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
  }
}
```

# Towards a parallel scanLeft

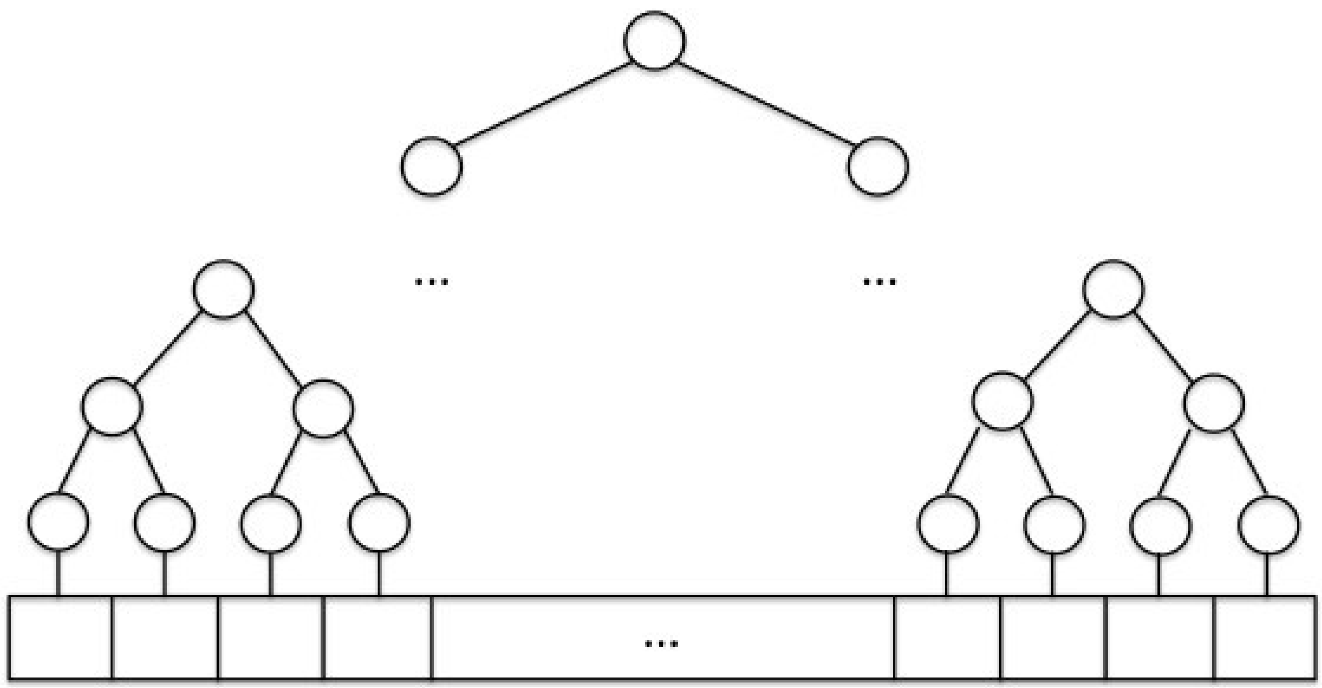
- In principle, we can use parallel map and reduce to obtain a parallel implementation with logarithmic depth

```scala
def mapSegS[A,B](inp: Array[A], left: Int, right: Int,
                 fi: (Int) => B, out: Array[B]): Unit = {
  if (right - left < threshold) {
    var i=left
    while (i < right) {
      out(i)=fi(i); i+=1
    }
  } else {
    val mid = left + (right - left)/2
    val (a1,a2) = parallel(mapSegS(inp, left, mid, fi, out),
      mapSegS(inp, mid, right, fi, out))
  }
}

def scanLeftMR[A](inp: Array[A], a0: A, f: (A,A) => A, out: Array[A]) = {
  val fi = { (i:Int) => f(a0,reduceSeg(inp, 0, i, f)) }
  mapSegS(inp, 1, inp.length+1, fi, out)
  out(0) = a0
}
```

# Towards a parallel scanLeft

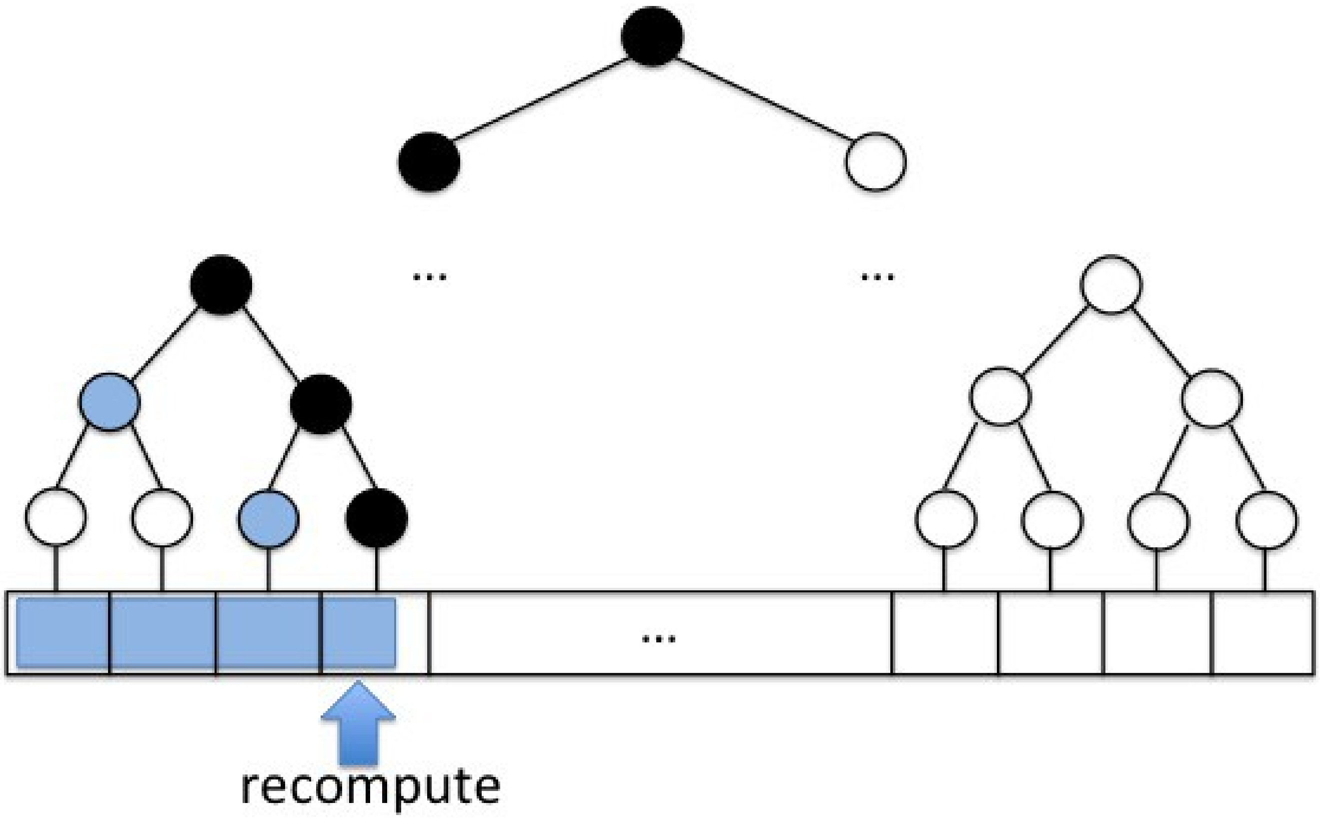
- Why is the map-reduce implementation not satisfactory?
  - Because the amount of work is quadratic!
  - Can we avoid such complexity? Yes, storing intermediary results




  - With linear work store the reduce on array segments in a tree

# Towards a parallel scanLeft

- Given the tree in the previous slide compute the reduce for each prefix of the array as follows:




  - along the path from the root to the segment, combine values in the left-child of nodes when move on the right
  - recompute on the remaining values in the segment

# Towards a parallel scanLeft

- First phase: compute the tree with the intermediary results

```
trait TreeRes[A] { val res: A }
case class Leaf[A](from: Int, to: Int, override val res: A)
                   extends TreeRes[A]
case class Node[A](l: TreeRes[A], override val res: A, r: TreeRes[A])
                   extends TreeRes[A]

def upsweep[A](inp: Array[A], from: Int, to: Int,
               f: (A,A) => A): TreeRes[A] = {
  if (to - from < threshold)
    Leaf(from, to, reduceOnSeg(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)
  }
}
```

# Towards a parallel scanLeft

- First phase: compute the tree with the intermediary results

```
def reduceOnSeg[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
}
```

# Towards a parallel scanLeft

- Second phase: go down the tree to compute scanLeft

```
def downsweep[A](inp: Array[A], a0: A, f: (A,A) => A, t: TreeRes[A],
                 out: Array[A]): Unit = t match {
  case Leaf(from, to, res) =>
    scanLeftOnSeg(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))
  }
}
def scanLeftOnSeg[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
    }
  }
}
```

# Towards a parallel scanLeft

- Third phase: put everything together

```
def scanLeftP[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
}
```

# Data parallelism

- So far, we have considered task-parallelism:
  - The `parallel(e1,e2)` primitive distributes tasks (i.e. the evaluation of its parameters) on parallel computing nodes
- We now move to data-parallelism:
  - Data are distributed on parallel computing nodes, and on each datum a (parameterized) parallel task is executed

```
def initializeArrayPar(xs: Array[Int])(v: Int): Unit = {
  for (i <- (0 until xs.length).par) {
    xs(i) = v
  }
}
```

- The data in the parallel collection are “split” in independent parts
- Parallel tasks act on the splitted parts of the collection
- A dedicated scheduler manages this parallelism

# Side effects

- The parallel tasks are not "functional"
  - they affect the program through side-effects
- As long as the data-parallel tasks write to separate memory locations, the program is correct
  - parallel write to the same location(s) needs synchronization to avoid data corruption

```scala
val mutual = new AnyRef()
def parMultiIncrement(n: Int) = {
  var v:Long = 0
  for (i <- (0 until n).par)
    mutual.synchronized{v=v+1}
  v
}
```

# Example

- Depict the Mandelbrot set

```
def run(n:Int, level:Int) : Unit = {
  val out = new FileOutputStream(fileName)
  out.write(("P5\n"+n+" "+n+"\n255\n").getBytes())
  for (j <- (0 until n*n))
  { val x = -2.0 + (j%n)*3.0/n
    val y = -1.5 + (j/n)*3.0/n
    var z = new Complex(0,0)
    var c = new Complex(x,y)
    var i = 0
    while (z.abs < 2 && i < level)
      {z = z*z + c; i=i+1}
    out.write(255*(level-i)/level) }
  out.close()
}
```

# Example

- Depict the Mandelbrot set (parallel version)

```
def runPar(n:Int, level:Int) : Unit = {
  val out = new FileOutputStream(fileName)
  out.write(("P5\n"+n+" "+n+"\n255\n").getBytes())
  var a=new Array[Int](n * n)
  for (j <- (0 until n*n).par)
  { val x = -2.0 + (j%n)*3.0/n
    val y = -1.5 + (j/n)*3.0/n
    var z = new Complex(0,0)
    var c = new Complex(x,y)
    var i = 0
    while (z.abs < 2 && i < level)
      {z = z*z + c; i=i+1}
    a(j) = 255*(level-i)/level }
  for{k <- 0 until n*n} out.write(a(k))
  out.close()
}
```

# Mandelbrot: evaluation

- Data-parallelism is appropriate in this case because:
  - Data-parallel tasks have different workloads
    - given the parallel task for datum i, let w(i) be its workload, i.e. the amount of work required to process it
  - For Mandelbrot we have w(i) = #iterations
    - Hence w(i) is not known a-priori, hence the programmer cannot appropriately manage the parallel executions (with primitives like parallel)
    - In fact, the complexity of parallel is the max of the two activities, and in case of unbalanced workload it is not efficient
  - Goal of the data-parallel scheduler:
    - Dynamically balance the workload across processors without any a-priori knowledge about the w(i)

# Parallelizable collections

- Sequential collections can be parallelized using .par
  - Notice that some collections do not have a parallel counterpart

```
val vector = Vector.fill(10000000)("")
val list = vector.toList
vector.par // creates a ParVector[String]
list.par   // also creates a ParVector[String]
```

  - The reason is that their representation does not naturally support data split
  - Array, Range and Vector are parallelizable (`ParArray[T]`, `ParRange[T]` and `ParVector[T]`)
  - Other collections, like Lists, are not directly parallelizable; `.par` returns the closest parallelizable supertype in the hierarchy

# Exercise

- Compute set intersection:

```
def intersection(a: Set[Int], b: Set[Int]):
Set[Int] = {
  val result = mutable.Set[Int]()
  for (x <- a) if (b contains x) result += x
  result
}
intersection((0 until 10000).toSet,
             (0 until 10000 by 4).toSet)
```

  - Can we parallelize this function?

# Exercise

- Compute set intersection:

```
def intersection(a: Set[Int], b: Set[Int]):
Set[Int] = {
  val result = mutable.Set[Int]()
  for (x <- a.par) if (b contains x) result += x
  result
}
intersection((0 until 10000).toSet,
             (0 until 10000 by 4).toSet)
```

  - Can we parallelize this function?
    - Yes, BUT the parallel version is not correct!
    - PROBLEM: concurrent modification on the same (non thread-safe) data structure (`result`)

# Exercise

- Compute set intersection:

```
def intersection(a: Set[Int], b: Set[Int]):
ParSet[Int] = {
  if (a.size < b.size) a.par filter (b contains _)
  else b.par filter (a contains _)
}
intersection((0 until 10000).toSet,
             (0 until 10000 by 4).toSet)
```

  - Here  is a correct parallel version:
    - Exploits a (parallel) method directly provided by the parallel collection

# Concurrent read/writes

- Consider the following example:

```
val graph = mutable.Map[Int, Int]() ++
   (0 until 100000).map(i => (i, i + 1))
graph += (graph.size - 1) -> 0
for ((k, v) <- graph.par) graph += k -> graph(v)
graph.count( p =>
              (p._2 != ((p._1 + 2) % graph.size)) )
```

- It contains two errors!
  - Concurrent tasks modify the non thread-safe Map collection
  - Concurrent tasks read from the collection while they modify it

# Concurrent read/writes

- Consider the following example:

```
val graph =
  concurrent.TrieMap[Int, Int]() ++
      (0 until 100000).map(i => (i, i + 1))
graph += (graph.size - 1) -> 0
val previous = graph.snapshot()
for ((k, v) <- graph.par) graph += k -> previous(v)
graph.count(p => (p._2 != ((p._1 + 2) % graph.size)))
```

- Here is a correct version:
  - TrieMap is a thread-safe collection (it supports concurrent modifications)
  - The method snapshot efficiently grab the current state