Week 1: Parallel Programming

Intro

Parallel computing: many calculations performed at the same time. Idea: divide computation into smaller subproblems each of which can be solved simultaneously using parallel hardware.

Interest: processor frequency scaling hit the *power wall*, so processor vendors decided to provide multiple CPU cores on the same chip, each capable of executing separate instructions streams.

Parallelism and concurrency are close concepts:

- **Parallel** program: uses parallel hardware to execute computation more quickly: EFFICIENCY
- **Concurrent** program: *may or may not* execute multiple executions at the same time. MODULARITY, RESPONSIVENESS, MAINTAINABILITY.

We focus on **task-level parallelism**: executing separate instruction streams in parallel. The hardware we will target is **multi-core processors** and **symmetric multiprocessors**.

Parallelism on the JVM

Operating system, processes and multitasking

- *Operating system* : software that manages hardware and software resources, and schedules program execution.
- *Process* : an instance of a program that is executing in the OS.

The operating system multiplexes many different processes and a limited number of CPUs, so that each process gets a *time slice* of execution: this is **multitasking**. Two different processes cannot access each other's memory directly, they are **isolated**.

Threads

Each process can contain multiple *independent concurrenccy units* called **threads**. They can be started from the same program and **share the same memory address space**. **Each thread has a program counter and a program stack**. In the JVM threads cannot modify each other's stack memory, only the heap memory.more here

Each JVM process starts with a **main thread**, to add additional threads

- Define a `Thread` subclass containing a `run` method
- Instantiate a new object
- Call `start`
- When done call `join()`

Atomicity Statements in different threads can overlap. Sometimes we want to ensure that a sequence of statements in specific thread executes at once.

An operation is ***atomic*** if it appears as if it occurred instantaneously from the point of view of other threads.

To achieve atomicity we use the `synchronized` block: code executed in the block is never executed by two threads at the same time.

Example:

```
private val x = new AnyRef{}
private var uidCount = 0L
def getUniqueId(): Long = x.syncronized {
   uidCount += 1
   uidCount
}
```

Different threads use the block to *agree on unique values*. It is an example of a *syncronization primitive*. Invocations of the `syncronized` block can nest.

Deadlocks Deadlocks can be caused by nested `synchronized` blocks A *Deadlock* occurs when multiple threads compete for resources and wait for each other to finish without releasing the already acquired resources.

One solution is to always acquire resources in the same order.

Memory Model Memory model is a set of rules that describes how threads interact when accessing shared memory.

In the JVM we have the *Java Memory Model*:

1. Two threads writing to two separate locations in memory do not need synchronization
2. A thread X that calls `join` on another thread Y is guaranteed to observe all the writes by thread Y after `join` returns

Running Computations in Parallel

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

```
val (res1, res2) = parallel(e1, e2)
val ((res1, res2), (res3, res4)) = parallel(parallel(e1, e2), parallel(e3, e4))

// The parallel function's signature
def parallel[A, B](taskA: => A, taskB: => B): (A, B) = { ... }
// NOTE: call by name to pass unevaluated computations or we wouldn't have parallel computations
```

Under the hood

Efficient parallelism require language and libraries, virtual machine, operating system and hardware support.

`parallel` uses JVM threads which *typically* map to OS threads which can be scheduled on different cores.

Given sufficient resources, a parallel program can run faster. Note that thanks to the different layers of abstractions a program written for parallel hardware can run even when there is only one processor core available (without speedup).

Hardware is important: **memory (RAM) is bottleneck** that can void all the effort necessary to write parallel software.

The running time of `parallel(e1, e2)` is the maximum of the two running times.

First-Class Tasks

As we have just seen, the running time of `parallel(e1, e2)` is the maximum of the two running times. When need a more flexible construct, that does not wait for the end of both threads to return. We use the `task` construct:

```
val t1 = task(e1)
val t2 = task(e2)
val v1 = t1.join //blocks and waits until the result is computed
val v2 = t2.join //obtain the result of e1
```

`task` start computation in *the background*. Subsequent calls to `join`return the same result

```
def task(c: => A): Task[A]

trait Task[A] {
  def join: A
}
```

We could omit `.join` by defining an implicit conversion: `implicit def getJoin[T](x: Task[T]): T = x.join`, then `task` will return the result.

`task` can be used to define `parallel`:

```
def parallel[A, B](cA: => A, cB: => B): (A, B) = {
   val tB: Task[B] = task { cB  } // if we called .join here the two threads would be executed sequentially and
   val tA: A = cA
   (tA, tB.join)
}
```

How Fast are Parallel Programs?

Benchmarking Parallel Programs

Week 2: Basic Task Parallel Algorithms

Parallel Merge-Sort:

Use divide and conquer:

1. Recursively sort the two halves of the array in parallel

```
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)
```

2. Sequentially merge the two halves by using a temporary array
3. Copy the temporary array into the original one

```
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)
```

Data Operations and Parallel Mapping

We need to process collections in parallel but this can be done only if some properties of collections and of the operation we want to apply are satisfied.

Functional programming and collections

Operations such as `map`, `fold` and `scan` are key to functional programming but become even more important for parallel collections: they allow write simpler parallel programs.

As always, the choice of the data structure (collection) to use is key: **Lists** are not good for parallel implementations:

- Splitting is slow (linear search for the middle)
- Concatenation is slow (linear search for the end)

Therefore we use:

- **Arrays**, which are imperative
- **Trees**, which can be implemented functionally

Parallelizing Map on arrays

We need to parallelize both the computations of the function and the access to the elements to apply the function.

```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))
    }
```

Remarks:

- **writes need to be disjoint**, or unpredictable output (nondeterministic behavior)
- **thresholds need to be large enough** or we lose efficiency

Parallelizing Map on immutable trees

Assume:

```scala
sealed abstract class Tree[A] { val size: Int }
// Leaves store array segments
case class Leaf[A](a: Array[A]) extends Tree[A] {
    override val size = a.size
}
// Non-leaf node stores two subtrees
case class Node[A](l: Tree[A], r: Tree[A]) extends Tree[A] {
    override val size = l.size + r.size
}
```

If trees are balanced, we can explore branches in parallel efficiently:

```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)
        }
}
```

Which will run in O(height of tree)

Arrays or immutable trees?

Arrays	Immutable Trees
- Imperative: must ensure parallel task write to disjoint paths	+ Purely functional: no need to worry about disjointness of
+ Good memory locality	- Bad locality
- Expensive to concatenate	+ efficient combination of two trees
+ random access to elements, on shared memory can share same array	- High memory allocation overhead

Fold (Reduce) Operations

Seen `map`, we focus on `fold`: combining elements with a given operation

`fold` takes a binary operations, but variants differ whether they take an initial element or assume non-empty list, in which order to combine operations of collection.

To parallelize we focus on **associative operations** such as addition and string concatenation (not subtraction)

Associative operations

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

Then if we have two expressions with same list of operands but different parentheses, these expression **evaluate to the same result**.

We can represent expression with trees: each leaf is a value, each node a computation of the operation.

Folding (reducing) trees in parallel

```
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]

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)) // remove parallel word to have the sequentia
        f(lV, rV)
    }
}
```

The complexity is again the height of the tree. If the operation is not associative, the result depends on the structure of the tree.

Folding (reducing) arrays

To reduce an array we can convert it into a balanced tree and then do tree reduction. If the algorithm is "simple enough" we do not need a formal definition of the tree, just divide it into halves:

```
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)
```

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

Associative Operation

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

Consequence (the two are equivalent):

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

False friends: commutativity

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

- **Associativity does not imply commutativity**
- **Commutativity does not imply associativity**

for correctness of `reduce` associativity is sufficient

Examples of operations that are bot associative and commutative

- addition and multiplication
- addition and multiplication modulo a positive integer
- union, intersection, symmetric difference
- union of multisets preserving duplicate elements
- boolean || && xor
- addition and multiplication of polynoials
- addition of vectors
- addition of matrices of fixed dimension

Examples of operations that are associative but not commutative

- list concatenation
- string concatenation
- matrix multiplication of compatible dimensions
- composition of relations
- composition of functions

Many operations are commutative but not associative : *f(x,y) = x^2 + y^2*

Mapping does not preserve associativity

if *f* is commutative and associative and *g*,*h* are arbitrary functions then

i(x,y) = h(f(g(x), h(y))) = h(f(g(y), h(x))) = i(y,x)

is commutative but if often loses *f*'s associativity.

floating point addition is commutative but not associative

Making an operation commutative is easy!

Suppose binary operation `g` and a strict total ordering `less`. Then

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

is commutative. **There is no such trick for associativity**.

Example: average

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

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

Associativity thanks to symmetry and commutativity

If

- Commutativity
- *f(f(x,y), z) = f(f(y,z), x)*

are satisfied then *f* is also **associative**.

Parallel Scan Left

Now we turn our attention to **scanLeft** which produces a list of the folds of all list prefixes: `List(1,3,8).scanLeft(100)((s,x) => s + x) == List(100, 101, 104, 112)` We assume that the operation is **associative**.

Sequential definition

```
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 parallelization using trees

We have to give up on reusing all intermediate results:

- do more `f` applications
- improve parallelism, to compensate the additional `f` applications

To reuse some of the intermediate results, we remember that `reduce` proceeds by applying the operations in a tree, so let's assume that the input is also a tree.

```
// input tree class
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]

//result tree class
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]
```

We now need to transform the input tree into the output tree:

```
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)
    }
}
```

And to use this intermediate tree to produce a `Tree` whose leaves are the result

```
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)
        }
}
```

and `scanLeft` becomes easy to define

```
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)
}

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)
}
```

Using arrays

To make it more efficient, we use trees that have arrays in leaves instead of individual elements.

Week 3: Data-Parallelism

Data-Parallel Programming

We now turn our attetion to **data-parallel** programming: *a form of parallelization that distributes data across computing nodes.*

The simplest form of data-parallel programming is the parallel `for` loop:

```
for (i <- (0 until array.length).par) do stuff
```

This loop **is not** functional: it only affects the program through side effects but **as long as iterations write to *separate* memory location, the program is correct**.

Workload

Different data-parallel programs have different workloads: *Workload* is a function that maps each input element to the amount of work required to process it. It may be uniform (therefore easy to parallelize) or irregular **depending on the problem instance**.

A *data-parallel scheduler* is therefore needed to efficiently balance the workload across computing units without any knowledge of the workload of each element.

Data-Parallel Operations I

In scala the `.par` converts a sequential collection to a parallel collection.

However **some operations are not parallelizable**.

Non-Parallelizable Operations

The function

```
def sum(xs: Array[Int]): Int = xs.par.foldLeft(0)(_ + _)
```

cannot be executed in parallel.

SEQUENTIAL :`foldRight, foldLeft, reduceRight, scanLeft, scanRight` must process the elements sequentially

PARALLEL : `fold` can process elements in a reduction tree and can therefore execute in parallel.

Some use-cases for `fold`

```
def sum(xs: Array[Int]): Int = xs.par.fold(0)(_ + _)

def max(xs: Array[Int]): Int = xs.par.fold(Int.MinValue)(math.max)
```

Prerequisite for `fold`

For `fold` to give us the expected result, we need to provide it with an **associative function**. Commutativity is not sufficient.

In particular the following conditions on `f` must be satisfied:

1. `f(a, f(b, c) == f(f(a, b), c)`
2. `f(z, a) == f(a, z) == a` (`z` being the neutral element)

We say that **the neutral element `z` and the binary operation `f` form a *monoid***.

We remark that **commutativity (`f(a,b) == f(b,a)`) does not matter for `fold`**

Limitations of `fold`

`fold` can only produce values of the same type as the collection that it is called on! Counting elements of a collection is impossible.

We need a more general (*expressive*) operations

The `aggregate` operation

```
def aggregate[B](z: B)(f: (B, A) => B, g: (B, B) => B): B
```

`f` produces the intermediary collections which will be combined by `g` to produce the final result. To count vowels now:

```
text.par.aggregate(0)( (count, chara) => if (isVowel(c)) count + 1 else count, _ + _ )
```

So far we studied *accessor combinators*

Operations like `map`, `filter`, `flatMap`, `groupBy` do not return a single value but instead return new collections as results. They are called *Transformer combinators*. `f`.

Scala Parallel Collections

Here are the main types of scala collections and their parallel counterparts:

Description	Sequential	Parallel	Agnostic
operations implemeted using `foreach`	`Traversable[T]`		
operations implemented using `iterator`	`Iterable[T]`	`ParIterable[T]`	`GenIterable[T]`
ordered sequence	`Seq[T]`	`ParSeq[T]`	`GenSeq[T]`
unordered collection, no duplicates	`Set[T]`	`ParSet[T]`	`GenSet[T]`
keys -> values, no duplicate keys	`Map[K ,V]`	`ParMap[K ,V]`	`GenMap[K ,V]`

`Gen*` allows us to write code that is **agnostic** about parallelism: code that is *unaware* of parallelism. `.par` converts a sequential collection into a parallel one.

Sequential Collection	Corresponding Parallel Collection
`ParArray[T]`	`Array[T]`and `ArrayBuffer[T]`
`Range`	`ParRange`
`Vector[T]`	`ParVector[T]`
`immutable.HashSet[T]`	`immutable.ParHashSet[T]`
`immutable.HashMap`	`immutable.ParHashMap[K, V]`
`mutable.HashSet[T]`	`mutable.ParHashSet[T]`
`mutable.HashMap[T]`	`mutable.ParHashMap[K, V]`
`TrieMap[K, V]`	`ParTrieMap[K, V]`

Another interesting parallel data structures is `ParTrieMap[K, V]`: a **thread-safe** parallel map with **atomic snapshots** (counterparto of `TrieMap`).

For other collections, `.par` creates the closest parallel collection: `List` is converted to `ParVector`.

Side effecting operations

```
def intersection(a: GenSet[Int], b: GenSet[Int]): Set[Int] = {
  val result = mutable.Set[Int]()
  for (x <- a) if (b contains x) result += x
  result
}
```

This code is not safe for parallelism! **Avoid mutations to the same memory locations without proper synchronization**. However synchronization might not be the solution as it has its own side effects.

Side effects can be avoided by using the correct *combinators* (`filter....`):

```
def intersection(a: GenSet[Int], b: GenSet[Int]): Set[Int] = {
  if (a.size < b.size) a.filter(b(_))
  else b.filter(a(_))
}
```

Concurrent Modifications during Traversals

Rule: **Never modify a parallel collection on which a data-parallel operation is in progress**

- Never write to a collection that is concurrently traversed
- Never read from a collection that is concurrently modified

In either case, propgram **non-deterministically** prints different results or crashes.

The `TrieMap` collection

`TrieMap` is an exception to these rules as its method `snapshot` can be used to efficiently grab the current state:

```
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(k)
```

This code works as expected.

Splitters and Combiners

Let's now focus on the following abstractions:

- Iterators
- Splitters
- Builders
- Combiners

Iterator

The simplified trait is

```
trait Iterator[A] {
   def next(): A
   def hasNext(): Boolean
}

def iterator: Iterator[A] // on every collection
```

The *iterator contract*: - `next` can be called only if `hasNext` returns `true` - after `hasNext` returns `false`, it will always return `false`

use:

```
def foldLeft[B](z: B)(f: (B, A) => B): B = {
  var s = z
  while (hasNext) s = f(s, next())
}
```

Splitter

The `Splitter` trait:

```
trait Splitter[A] extends Iterator[A] {
  def split: Seq[Splitter[A]]
  def remaining: Int
}

def splitter: Splitter[A] // on every parallel collection
```

the contract:

- After calling `split` the original splitter is left in an *undefined state*
- the resulting splitters traverse *disjoint subsets* of the original splitter
- `remaining` is an *estimate* on the number of remaining elements
- `split` is *efficient* : O(log n) or better

use:

```
def fold(z: A)(f: (A, A) => A): A = {
  if (remaining < threshold) foldLeft(z)(f)
  else {
    val children = for (child <- split) yield task {child.fold(z)(f)}
    children.map(_.join()).foldLeft(z)(f)
  }
}
```

Builder

trait:

```
trait Builder[A, Repr] {
  def +=(elem: A): Builder[A, Repr]
  def result: Repr
}

def newBuilder: Builder[A, Repr] // on every collection
```

Contract:

- calling `result` returns a collection of type `Repr`, containing the elements that were previously added with `+=`
- calling `result` leaves the `Builder` in an *undefined state*

use:

```
def filter(p: T=> Boolean): Repr = {
  val b = newBuilder
  for (x <- this) if (p(x)) b += x
  b.result
}
```

Combiner

trait:

```
trait Combiner[A, Repr] extends Builder[A, Repr] {
  def combine(that: Combiner[A, Repr]): Combiner[A, Repr]
}

def newCombiner: Combiner[T, Repr] //on every parallel collection
```

contract:

- calling `combine` returns a *new combiner* that contains elements of the input combiners
- calling `combine` leaves both original `Combiners` in an *undefined state*
- `combine` is efficient: O(log n) or better

use:

Example during class

Week 4:

Implementing Combiners

- `conbine` represents union for a set or a map and concatenation for a sequence.
- `combine` must be **efficient** i.e. $O(\log n + \log m)$ (n,m being the size of the 2 input collections)

Arrays

cannot be efficiently concatenated.

Sets

Sets have have efficient lookup, insertion and deletion. The running time depends on the implementation:

- Hash tables, *expected* O(1)
- Balanced trees: O(log n)
- Linked lists: O(n)

However, **most implementations do not have efficient union operation**.

Sequences

Complexities depend on the implementation:

- Mutable linked lists: O(1) append and prepend (and concatenaton), O(n) insertion
- functional (cons) lists: O(1) prepend, everything else (concatenaton) O(n)
- Array Lists: amortized O(1) append, O(1) random acces, everything else (concatenaton) O(n)

Parallel Two-Phase Construction

To avoid these unefficient methods, most data structures can be constructed using ***two-phase construction***.

This technique relies on an *intermediate data structures* that:

- has an efficient ($O(\log n + \log m)$ or better) `combine`
- has an efficient `+=`
- can be converted to the resulting data structures in $O(n/P)$

Two-phase construction for Arrays

The two phases are implemented in `result` and they are:

1. partition the indices into subintervals
2. initialize array in **parallel**

```
class ArrayCombiner[T <: AnyRef: ClassTag](val parallelism: Int) {
  private var numElems = 0
  private val buffers = new ArrayBuffer[ArrayBuffer[T]]
  buffers += new ArrayBuffer[T]

  // O(1) amortized, low constat factors ~ arraybuffer
  def +=(x: T) {
    buffers.last += x
    numElems += 1
    this
  }

  //O(P) if buffers contains less than O(P) nested arraybuffers
  def combine(that: ArrayCombiner[T]) = {
    buffers ++= that.buffers
    numElems += that.numElems
    this
  }

  def result: Array[T] = {
    val array = new Array[T](numElems)
    val step = math.max(1, numElems / parallelism)
    val starts = (0 until numElems by step) :+ numElems
    val chunks = starts.zip(starts.tail)
    val tasks = for ((from, end) <- chunks) yield task {
      copyTo(array, from, end)
    }
    task.foreach(_.join())
    array
  }
}
```

Two-phase Construction for Hash Tables

1. Partition the hash codes into buckets
2. allocate the table, map hash codes from different buckets into different regions

Two-phase Construction for Search Trees

1. Partition the elements **into non-overlapping intervals** according to their ordering
2. Construct search trees in **parallel** and link non-overlapping trees.

Two-phase Construct for Spatial Data Structures

1. Spatially partition the elements
2. Construct non **non-overlapping** subsets and link them

Implementing combiners

Two phase construction is one of the three main strategies:

1. Two-phase construction: The combiner uses an intermediate data structure with an efficient `combine` method to partition the elements. Then `result` is called to construct the final data structure **in parallel** from the intermediate data structure.
2. An efficient concatenation or union operation (*algorithm*): the preferred way **when the resulting data structure allows it**
3. *Concurrent data structure*: different combiners share the same underlying data structure and rely on **synchronization** to correctly update the data structure when `+=` is called.

Week 5: Introduction to Concurrent Programming

In concurrent programming a program is:

- A set of **concurrent** computations
- That execute during **overlapping time intervals**
- That **need to coordinate** in some way.

Concurrency is hard: RISKS = non determinism + race conditions + deadlocks

Advantages:

1. Performances (thanks to parallelism in hardware)
2. Responsiveness (faster I/O **without blocking** or polling)
3. Distribution (programs can be spread over multiple nodes)

Terminology

The operating systems schedules *cores* to run *threads*. Threads in the same process *share memory*.

Week1 -> how to start threads in the JVM

In scala a helper method is provided to create threads:

```scala
def thread(b: => Unit) = {
  val t = new Thread {
    override def run() = b
  }
  t.start()
  t
}

val t = thread { println("New thread running")}
t.join()
```

Threads on the JVM

A thread image in memory contains *copies of the processor registers* and the *call stack (~2MB)*.

The os has a *scheduler* that runs all active threads on all cores. If there are more threads than cores, they are *time sliced*.

Switching from a thread to another (***context switching***) is a complex and expensive operation (~1000ns = 1 us per switch) but is less expensive that blocking io.

Threads can be paused **from inside** with `Thread.sleep(time)`.

Interleaving and locking

Threads operations are executed concurrently or interleaved: there could be a context switch (ordered by the processor scheduler for example) in the middle of the execution of a thread. This could cause non-determinism.

Atomicity and synchronize were covered in week1.

```
def synchronized[T](block: => T): T
```

```
obj.syncronized { block }
```

Here `obj` acts as a ***lock***: it makes sure that no other thread executes inside a syncronized method ***of the same object***.

- `block` can be executed only by the thread holding the lock
- **only one thread can hold a lock at any one time**.

Every object can serve as a lock, which is expensive when not on the JVM (e.g. Scala.js) therefore in the future `synchronized` will become a method of the trait `Monitor` whose instances only will be able to serve as locks.

A Memory Model

Sequential Consistency Model:

The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Forget it

However **modern multicore processors do not implement the sequential consistency model**. The problem is caused by **caches**:

- Each core might have a different copy of shared memory in its private caches
- Write-backs of caches to main-memory happens at *unpredictable* times.

Also compilers: they are often allowed to reorder instructions.

As a general rule: ***Never write to a variable being read by another thread && Never try to read a variable being written by another thread***.

`synchronized` ensures atomic execution and that **writes are visible**: - after `obj.synchronized { ... }` was executed by thread `t`, all writes of `t` up to that point are visible to every thread that subsequently executes a `obj.synchronized {... }`

Thread coordination

Monitors can do more than just locking with `syncronized`, they offer the methods:

- `wait()`: suspends the current thread. it *releases the lock* so other threads can enter the monitor.
- `notify()` wakes up **one thread** waiting on the current object
- `notifyAll()` wakes up **all threads** waiting on the current object.

They **should only be called from within a synchronized on `this`**. `notify()` and `notifyAll` schedule other threads for execution after the calling thread has released the lock (left the monitor).

See assignment 5 for an example.

Signals

The original model of *Monitors* by Per Brinch Hansen has synchronized but not `wait, notify, notifyAll`. Instead it uses ***signals***:

```
class Signal {
  def wait(): Unit // wait for someone
  def send(): Unit // wakes up first waiting thread
}

// Example
class OnePlaceBuffer[Elem] extends Monitor {
  var elem: Elem = _
  var full = false
  val isEmpty, isFull = new Signal
  def put(e: Elem): Unit = synchronized {
    while (full) isEmpty.wait()
    isFull.send()
    full = true; elem = e
  }
  def get(): Elem = synchronized {
    while (!full) wait()
    isEmpty.send()
    full = false; elem
  }
}
```

Stopping Thread

A method `stop()` exists in `Thread` but is **deprecated and should not be used**:

The thread will be killed **at any arbitrary time (when the message is received)**. It may happen during a lock, leaving the system in a completely undetermined state.

Volatile Variables

Sometimes we only need ***safe publication*** (instead of atomic publication): in these case there is a **cheaper solution** than using a synchronized: ***volatile fields***:

```
@volatile var found = _
val t1 = thread {... ; found = true}
val t2 = thread {while(!found) ...}
```

- Assignments to the volatile variable **will not be reordered** with respect to other statements in the thread
- Assignments to the volatile variable are **visible immediately to all other threads**.

Memory Models in general

A memory model is an abstraction of the hardware capabilities of different computer systems. It abstracts over the underlying system's *cache coherence protocol*. Memory models are **non-deterministic** to allow some freedom of implementation in hardware or compiler. Every model is a compromise:

- More guarantees => easier to write concurrent programs
- Fewer guarantees => more possibilities for optimization.

The Java Memory Models

- **Program order**: each action in a thread happens before every subsequent action in the same thread.
- **Monitor locking**: unlocking a monitor happens before every subsequent locking of that monitor
- **Volatile fields**: a write to a volatile field happens before every subsequent read of that field
- **Thread start**: a call to `start()` on a thread happens before all actions of that thread.
- **Thread termination**: an action in a thread happens before another thread completes a `join` on that thread
- **Transivity**: If A happens before B and B happens before C, then A happens before C

Week 6: Concurrency Building Blocks

Executors

Thread creation is expensive. Therefore, one often *multiplexes threads to perform different task.* The JVM offers abstractions for that `ThreadPools` and `Executors`.

- Threads become in essence the workhorses that perform various tasks presented to them.
- The number of available threads in a pool is typically some polynomial of the number of cores N (e.g. N*^2*).
- That number is *independent of the number of concurrent activities to perform.*

Runnables

A task presented to an executor is encapsulated in a `Runnable` object:

```
trait Runnable {
  def run(): Unit = {// actions to be performed by the task }
}
```

Runnables can be passed to the `ForkJoinPool` executor. This is a **system provided** thread-pool which also handles tasks spawned by Scala's implementation of parallel collections and Java 8's implementation of streams.

```
import java.util.concurrent.ForkJoinPool
object ExecutorsCreate extends App {
  val executor = new ForkJoinPool
  executor.execute(new Runnable {
    def run() = log("This task is run asynchronously")
    })
    Thread.sleep(500)
}
```

1. Task are run by passing `Runnable` objects to an executor.
2. There is **no way to await the end of a task** (no .join()). Instead we pause the main thread to give the executor threads time to finish.

In alternative `executor.shutdown()` and `executor.awaitTermination(60, TimeUnit.SECONDS)` (defined by Java's interface `ExecutorService` implemented by `ForkJoinPool`) could be used.

Execution Contexts

The `scala.concurrent` package defines the `ExecutionContext` trait and object which is similar to `Executor` but more specific to scala. Execution contexts are passed as implicit parameters to many of Scala's concurrency abstractions. Using the default:

```
import scala.concurrent
object ExecutionContextCreate extends App {
  val ectx = ExecutionContext.global
  ectx.execute( new Runnable ).... // as in previous example.
}

// UTILITY METHOD TO CUT THE BOILERPLATE
def execute(body: => Unit) = scala.concurrent.ExecutionContext.global.execute(
  new Runnable { def run () = body})

// use:
execute { log("This task is run asynchronously")
  Thread.sleep(500)}
```

Atomic Primitives

We now look at the primitives used to realize the higher level operations `wait()`, `notify` and `notifyAll`.

On the JVM they are based on the notion of an ***atomic variable***.

- An ** *atomic variable* is a memory location that supports *linearizable* operations**
- A ***linearizable*** operation is one that **appears instantaneously with the rest of the system**.

We also say that the operation is performed **atomically**. In Java, classes that create atomic variables are defined in the package `java.util.concurrent.atomic` and include `AtomicInteger, AtomicLong, AtomicBoolean, AtomicReference`.

Using Atomic Variables

Atomic classes have atomic methods such as `incrementAndGet, getAndSet`: they are complex and linearizable operations during which **no intermediate result can be observed by other threads**.

Compare and Swap

Atomic operations are usually based on the **compare-and-swap (CAS)** primitive: this operation is available as `compareAndSet` on atomic variables. It is usually *implemented by the underlying hardware as a machine instruction.*

We can imagine they are implemented like this:

```
def compareAndSet(expect: T, update: T): Boolean = this.synchronized {
  if (this.get == expect) { this.set(update); true}
  else false
}
```

Programming Without Locks

`synchronized` locks are convenient but they bring about the possibility of deadlocks and they can be arbitrarily delayed by the OS or other threads.

Using atomic variables and their **lock-free operations** we can avoid `synchronized` and its problems: a thread executing a lock-free operation cannot be pre-empted by the OS so **it cannot temporarily block other threads**.

Simulating Locks

We can implement `synchronized` only from atomic operations:

```scala
private val lock = new AtomicBoolean(false)
def mySynchronized(body: => Unit): Unit = {
  while (!lock.compareAndSet(false, true)) {}
  try body
  finally lock.set(false)
}
```

Lock-Free operations

Definition of lock-freedom without atomic variables or other primitives:

An operation op is lock-free if whenever there is a set of threads executing op at least one thread completes the operation after a finite number of steps, regardless of the speed in which the different threads progress.

Lock-freedom is difficult to reason about!!

Lazy Values

```scala
@volatile private var x_defined = false
private var x_cached: T = _
def x: T = {
  if (!x_defined) this.synchronized {
    if (!x_defined) { x_cached = E ; x_cached = true}
  }
  x_cached
}
```

This pattern is called *double locking* (double checking would more appropriate) This is the actual implementation of lazy vals in scalac. However:

- `synchronized` -> not lock-free
- if this is already used as a monitor there is the risk of a deadlock

  ```scala
  object A { lazy val x = B.y }
  object B { lazy val y = A.x }
  ```

  Sequential execution -> Infinite loop & stack overflow Concurrent execution -> A waits for B, B waits for A. . . . Either stack overflow or deadly (not only it does not do what expected, it is **nodeterministc**).

An alternative (used in `dotty`, new scala compiler)

```scala
private var x_evaluating = false
private var x_defined = false
private var x_cached = _

def x: T = {
  if (!x_defined) {
    this.synchronized {
      if (x_evaluating) wait() else x_evaluating = true
    }
    if (!x_defined) {
      x_cached = E
      this.synchronized {
        x_evaluating = false
```

```
        x_defined = true
        notifyAll()
      }
    }
  }
  x_cached
}
```

The synchronized blocks are very short, they will not cause deadlocks and they do not use `this` as a monitor inside. Having 2 `synchronized` does not worsen performances (two small instead of one big). Other variants could use different locks but would require allocating other object(s), which would too expensive.

Some perks: - The evaluation of E happens outside a monitor (synchronized) -> no arbitrary slowdowns - No interference with user defined locks - **Deadlocks are still possible, but only in cases where sequential execution would give an infinite loop**

Using Collections Concurrently

Operations on **mutable** collections are usually not thread safe.

One solution to deal with this is to use `synchronized`

```
object CollectionBad extends App {
  val buffer = mutable.ArrayBuffer[Int]()
  def add(numbers: Seq[Int]) = execute {
    buffer ++= numbers
    log("buffer = "+ buffer.toString)
  }
}
// add with synchronized becomes
def add(numbers: Seq[Int]) = execute { buffer.synchronized {
  buffer ++= numbers ; log( ... )}}
```

However `synchronized` often leads to too much blocking because of *coarse-grained locking*.

Concurrent Collections

To gain speed we can use or implement *concurrent collection*.

Example: concurrent queue Make `head` and `last` atomic. We use an atomic CAS for (`last`) and fix the assignment of `prev` when CAS is successful. Fix `prev.next == null` instead of `prev.next == last1` in `remove`:

```
object ConcQueue {
  private class Node[T](@volatile var next: Node[T]) {
    var elem: T = _
  }
}

class ConcQueue[T] {
  import ConcQueue._
  private var last = new AtomicReference(new Node[T](null))
  private var head = new AtomicReference(last.get)

  @tailrec final def append(elem: T): Unit = {
    val last1 = new Node[T](null)
    last1.elem = elem //create new node
    val prev = last.get
    if (last.compareAndSet(prev, last1)) prev.next = last1 // if one modification succeds, do the other
    else append(elem) // start over if fail <-> someone modified prev during lines
  }

  @tailrec final def remove(): Option[T] =
    if (head eq last) None
    else {
      val hd = head.get
      val first = hd.next
      if (first != null && head.compareAndSet(hd, first)) Some(first.elem)
      else remove()
    }
}
```

In the library In `java.util.concurrent` there are some multiple implementations of the interface `BlockingQueue` (blocking => `wait()` such as Assignment 5).

The Scala library also provides implementations of concurrent sets and maps.

Week 7: Try and Future

	One	Many
Synchronous	`T/Try[T]`	`Iterable[T]`
Asynchronous	`Future[T]`	`Observable[T]`

Try[T], a monad that handles exceptions

We use this monad to **encapsulate exceptions inside objects**: normally they work only in one thread, encapsulation allows us to move them between threads.

Let's consider the simple adventure game:

```
trait Adventure {
  def collectCoins(): List[Coin]
  def buyTreasure(coins: List[Coin]): Treasure
}

val adventure = Adventure()
val coins = adventure.collectCoins()
val treasure = adventure.buyTreasure(coins)
```

We might need to throw an exception in `collectCoins`:

```
def collectCoins(): List[Coin] = {
  if (eatenByMonster(this)) throw new GameOverException("oops")
  List(Gold, Gold, Silver)}
```

But then the return type would be wrong! Accept that failure may happen:

```
abstract class Try[T]
case class Success[T] (elem: T) extends Try[T]
case class Failure(t: Throwable) extends Try[Nothing]

// Then our adventure:
trait Adventure {
  def collectCoins(): Try[List[Coin]]
  def buyTreasure(coins: List[Coin]): Try[Treasure]
}
```

Higher-order functions to manipulate Try[T]

- `def flatMap[S](f: T=>Try[S]): Try[S]`
- `def flatten[U <: Try[T]]: Try[U]`
- `def map[S](f: T=>S): Try[T]`
- `def filter(p: T=>Boolean): Try[T]`
- `def recoverWith(f: PartialFunction[Throwable, Try[T]]): Try[T]`

They allow us `val treasure: Try[Treasure] = adventure.collectCoins().flatMap(coins => adventure.buyTreasure(coins)`
We can also use for-comprehension:

```
val treasure: Try[Treasure] = for {
  coins     <- adventure.collectCoins()
  treasure  <- buyTreasure(coins)
} yield treasure
```

Under the hood: map

```
def map[S](f: T=>S): Try[S] = this match {
  case Success(value)     => Try(f(value))
  case failure@Failure(t) => failure
}

object Try {
  def apply[T](r: => T): Try[T] = {
    try { Success(r) }
    catch { case t => Failure(t) }
  }
}
```

Future[T], a monad that handles exceptions and latency

In the future, not now, the computation will be done. You will either have a result or an exception. Consider:

```
trait Socket {
  def readFromMemory(): Array[Byte]
  def sendToEurope(packet: Array[Byte]): Array[Byte]
}
```

It looks a lot like the previous game. However here some `readFromMemory` and `sendToEurope` are operations that ***take time*** (50 000 and 150 000 000 ns). It would be a huge waste to have the computantion waiting for these actions to complete before proceeding. We could even wait all that time to discover that the action *failed* resulting in an exception.

Therefore we use `Future[T]`:

```
trait Future[T] {
  // we will ignore the implicit execution context
  def onComplete(callback: Try[T] => Unit)(implicit executor: ExecutionContext): Unit
}
```

We can just pass a block of code: `def/va name = Future {action returning T}`

The method `onComplete` is called when the Future is ready. Future is ***asynchronous***. `callback` needs to use pattern matching to distinguish if the Try is a success or a failure. This however introduces a lot of boilerplate code.

Thus we consider some alternative designs:

```
def onComplete(success: T => Unit, failed: Throwable => Unit): Unit
```

Which is what happens with javascript promises: we pass to the function the actions to perform both in case of success and of failure. In javascript these actions are *closures* simple function. In scala they are objects: these are just two sides of the same coin:

- An *object* is a closure with multiple methods
- A *closure* is an object with a single method

We can adapt our program:

```scala
trait Socket {
  def readFromMemory(): Future[Array[Byte]]
  def sendToEurope(packet: Array[Byte]): Future[Array[Byte]]
}
```

Using `onComplete` as defined can be cumbersome:

```scala
val confirmation: Future[Array[ByteA]] = packet.onComplete {
  case Success(p) => ... // continue
  case Failure(t) => ...
}
```

The type do not match: `onComplete` returns `Unit`. We could

```scala
packer.onComplete {
  case Success(p) => {
    val confirmation = .... // The rest of the application
  }
}
```

all our program would have to move inside this case. UGLY (javascript)

The solution: flatMap!

```scala
val packet: Future[Array[Byte]] = socket.readFromMemory
val confirmation: Future[Array[Byte]] = packet.flatMap(p => socket.sendToEurope(p))
```

Robust solution for packet sending

```scala
//This can have Failures inside
def sendTo(url: URL, packet: Array[Byte]): Future[Array[Byte]] =
  Http(url, Request(packet))
    .filter(response => response.isOK)
    .map(response => response.toByteArray)

//use recoverWith to handle exceptions if they are there
def sendToSafe(packet: Array[Byte]): Future[Array[Byte]] =
  sendTo(mailServer.europe, packet) recoverWith {
    case europeError => sendTo(mailServer.usa, packet) recover {
      case usaError => usaError.getMessage.toByteArray
    }
  }
```

An auxiliary function to reduce boilerplate

"'scala def fallbackTo(that: =>Future[T]): Future[T] = { //if this future fails take the successful result of that future //if that future fails too, take the error of this future this recoverWith { case _ => that recoverWith (case _ => this) } } | | V def sendToSafe(packet: Array[Byte]): Future[Array[Byte]] = sendTo(mailServer.europe, packet) fallbackTo { sendTo(mailServer.usa, packet) } recover { case europeError => europeError.getMessage.toByteArray }

Creating futures

```scala
object Future {
  def apply(body: => T)(implicit ....) : Future[T]
}
```

Future can block, if necessary

```scala
Trait Awaitable[T] extends AnyRef {
  abstract def ready(atMost: Duration): Unit
  abstract def result(atMost: Duration): T
}

trait Future[T] extends Awaitable[T] ....

// Then
val confirmation: Future[Array[Byte]] = packet.flatMap(socket.sendToSafe(_))
val c = Await.result(confirmation, 2 seconds)
```

`Future` can also block our program until the computation is finished. However this is to be avoided in a good reactive program.

Because you can, in Scala (almost an anecdote) The Scala library provides a duration object:

```scala
import scala.language.postfixOps

object Duration {
  def apply(length: Long, unit: TimeUnits): Duration
}
val fiveYears = 1826 minutes
```