ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Data-Parallel Programming

Parallel Programming in Scala

Aleksandar Prokopec

Data-Parallelism

Previously, we learned about task-parallel programming.

> *A form of parallelization that distributes execution processes across computing nodes.*

We know how to express parallel programs with `task` and `parallel` constructs.

Data-Parallelism

Previously, we learned about task-parallel programming.

> *A form of parallelization that distributes execution processes across computing nodes.*

We know how to express parallel programs with `task` and `parallel` constructs.

Next, we learn about the data-parallel programming.

> *A form of parallelization that distributes data across computing nodes.*

Data-Parallel Programming Model

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

Example: initializing the array values.

Data-Parallel Programming Model

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

Example: initializing the array values.

```
def initializeArray(xs: Array[Int])(v: Int): Unit
```

Data-Parallel Programming Model

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

Example: initializing the array values.

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

  }
}
```

Data-Parallel Programming Model

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

Example: initializing the array values.

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

Data-Parallel Programming Model

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

Example: initializing the array values.

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

The parallel `for` loop is not functional – it can only affect the program through side-effects.

Data-Parallel Programming Model

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

Example: initializing the array values.

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

The parallel `for` loop is not functional – it can only affect the program through side-effects.

As long as iterations of the parallel loop write to separate memory locations, the program is correct.

Example: Mandelbrot Set

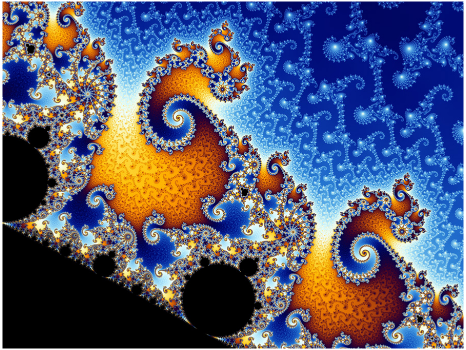
Although simple, parallel `for` loop allows writing interesting programs.

Render a set of complex numbers in the plane for which the sequence $z_{n+1} = z_n^2 + c$ does not approach infinity.

Example: Mandelbrot Set

Although simple, parallel `for` loop allows writing interesting programs.

Render a set of complex numbers in the plane for which the sequence $z_{n+1} = z_n^2 + c$ does not approach infinity.



Example: Mandelbrot Set

We approximate the definition of the Mandelbrot set – as long as the absolute value of z_n is less than 2, we compute z_{n+1} until we do `maxIterations`.

```
private def computePixel(xc: Double, yc: Double, maxIterations: Int): Int = {
  var i = 0
  var x, y = 0.0
  while (x * x + y * y < 4 && i < maxIterations) {
    val xt = x * x - y * y + xc
    val yt = 2 * x * y + yc
    x = xt; y = yt
    i += 1
  }
  color(i)
}
```

Example: Mandelbrot Set (Data-Parallel)

How do we render the set using data-parallel programming?

```
def parRender(): Unit = {
  for (idx <- (0 until image.length).par) {
    val (xc, yc) = coordinatesFor(idx)
    image(idx) = computePixel(xc, yc, maxIterations)
  }
}
```

Rendering the Mandelbrot Set: Demo

Time for a demo!

Rendering the Mandelbrot Set: Demo

Time for a demo!

Summary:

- task-parallel implementation – the slowest.
- data-parallel implementation – about $2\times$ faster.

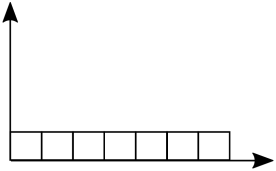
Workload

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.

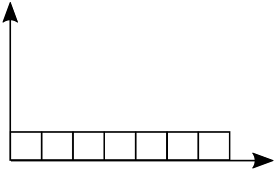
Uniform Workload

Defined by a constant function: $w(i) = const$



Uniform Workload

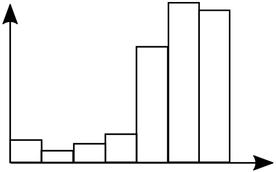
Defined by a constant function: $w(i) = const$



Easy to parallelize.

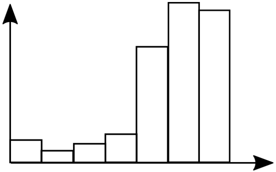
Irregular Workload

Defined by an arbitrary function: $w(i) = f(i)$



Irregular Workload

Defined by an arbitrary function: $w(i) = f(i)$

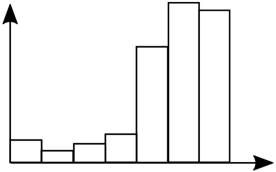


In the Mandelbrot case: $w(i) = \#iterations$

The workload depends on the problem instance.

Irregular Workload

Defined by an arbitrary function: $w(i) = f(i)$



In the Mandelbrot case: $w(i) = \#iterations$

The workload depends on the problem instance.

Goal of the *data-parallel scheduler*: efficiently balance the workload across processors without any knowledge about the $w(i)$.

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Data-Parallel Operations I

Parallel Programming in Scala

Aleksandar Prokopec

Parallel Collections

In Scala, most collection operations can become data-parallel.

The `.par` call converts a sequential collection to a parallel collection.

```
(1 until 1000).par
  .filter(n => n % 3 == 0)
  .count(n => n.toString == n.toString.reverse)
```

Parallel Collections

In Scala, most collection operations can become data-parallel.

The `.par` call converts a sequential collection to a parallel collection.

```
(1 until 1000).par
  .filter(n => n % 3 == 0)
  .count(n => n.toString == n.toString.reverse)
```

However, some operations are not parallelizable.

Non-Parallelizable Operations

Task: implement the method `sum` using the `foldLeft` method.

```
def sum(xs: Array[Int]): Int
```

Non-Parallelizable Operations

Task: implement the method `sum` using the `foldLeft` method.

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

Does this implementation execute in parallel?

Non-Parallelizable Operations

Task: implement the method `sum` using the `foldLeft` method.

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

Does this implementation execute in parallel?

Why not?

Non-Parallelizable Operations

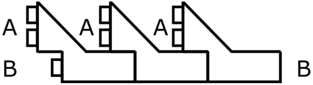
Let's examine the `foldLeft` signature:

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

Non-Parallelizable Operations

Let's examine the `foldLeft` signature:

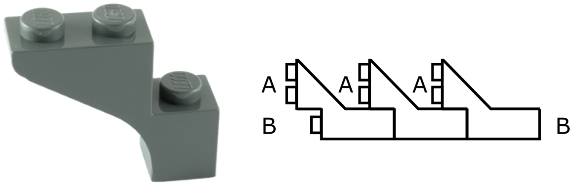
```
def foldLeft[B](z: B)(f: (B, A) => B): B
```



Non-Parallelizable Operations

Let's examine the foldLeft signature:

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



Operations foldRight, reduceLeft, reduceRight, scanLeft and scanRight similarly must process the elements sequentially.

The `fold` Operation

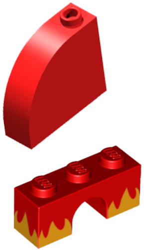
Next, let's examine the `fold` signature:

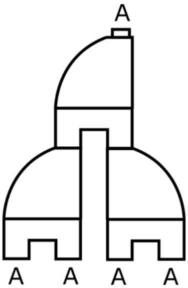
```
def fold(z: A)(f: (A, A) => A): A
```

The fold Operation

Next, let's examine the `fold` signature:

```
def fold(z: A)(f: (A, A) => A): A
```





The `fold` operation can process the elements in a reduction tree, so it can execute in parallel.

Data-Parallel Operations II

Parallel Programming in Scala

Aleksandar Prokopec

Use-cases of the fold Operation

Implement the sum method:

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

Use-cases of the fold Operation

Implement the sum method:

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

Implement the max method:

```
def max(xs: Array[Int]): Int
```

Use-cases of the fold Operation

Implement the sum method:

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

Implement the max method:

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

Preconditions of the `fold` Operation

Given a list of `"paper"`, `"rock"` and `"scissors"` strings, find out who won:

```
Array("paper", "rock", "paper", "scissors")
```

Preconditions of the fold Operation

Given a list of "paper", "rock" and "scissors" strings, find out who won:

```
Array("paper", "rock", "paper", "scissors")
  .par.fold("")(play)

def play(a: String, b: String): String = List(a, b).sorted match {
  case List("paper", "scissors") => "scissors"
  case List("paper", "rock")     => "paper"
  case List("rock", "scissors")  => "rock"
  case List(a, b) if a == b      => a
  case List("", b)               => b
}
```

Preconditions of the fold Operation

Given a list of ”paper”, ”rock” and ”scissors” strings, find out who won:

```
Array(”paper”, ”rock”, ”paper”, ”scissors”)
  .par.fold(””)(play)
```

Preconditions of the `fold` Operation

Given a list of `"paper"`, `"rock"` and `"scissors"` strings, find out who won:

```
Array("paper", "rock", "paper", "scissors")
  .par.fold("")(play)
```

```
play(play("paper", "rock"), play("paper", "scissors")) == "scissors"
```

Preconditions of the fold Operation

Given a list of ”paper”, ”rock” and ”scissors” strings, find out who won:

```
Array(”paper”, ”rock”, ”paper”, ”scissors”)
  .par.fold(””)(play)
```

```
play(play(”paper”, ”rock”), play(”paper”, ”scissors”)) == ”scissors”
```

```
play(”paper”, play(”rock”, play(”paper”, ”scissors”))) == ”paper”
```

Why does this happen?

Preconditions of the `fold` Operation

Given a list of "paper", "rock" and "scissors" strings, find out who won:

```
Array("paper", "rock", "paper", "scissors")
  .par.fold("")(play)
```

```
play(play("paper", "rock"), play("paper", "scissors")) == "scissors"
```

```
play("paper", play("rock", play("paper", "scissors"))) == "paper"
```

Why does this happen?

The `play` operator is *commutative*, but not *associative*.

Preconditions of the `fold` Operation

In order for the `fold` operation to work correctly, the following relations must hold:

```
f(a, f(b, c)) == f(f(a, b), c)
f(z, a) == f(a, z) == a
```

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

Preconditions of the `fold` Operation

In order for the `fold` operation to work correctly, the following relations must hold:

```
f(a, f(b, c)) == f(f(a, b), c)
f(z, a) == f(a, z) == a
```

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

Commutativity does not matter for `fold` – the following relation is not necessary:

```
f(a, b) == f(b, a)
```

Limitations of the `fold` Operation

Given an array of characters, use `fold` to return the vowel count:

Limitations of the fold Operation

Given an array of characters, use `fold` to return the vowel count:

```
Array('E', 'P', 'F', 'L').par
  .fold(0)((count, c) => if (isVowel(c)) count + 1 else count)
```

Limitations of the fold Operation

Given an array of characters, use fold to return the vowel count:

```
Array(‘E‘, ‘P‘, ‘F‘, ‘L‘).par
  .fold(0)((count, c) => if (isVowel(c)) count + 1 else count)
```

Question:

What does this snippet do?

- The program runs and returns the correct vowel count.
- The program is non-deterministic.
- The program returns incorrect vowel count.
- The program does not compile.

Limitations of the `fold` Operation

Given an array of characters, use `fold` to return the vowel count:

```
Array(‘E‘, ‘P‘, ‘F‘, ‘L‘).par
  .fold(0)((count, c) => if (isVowel(c)) count + 1 else count)

// does not compile -- 0 is not a Char
```

The `fold` operation can only produce values of the same type as the collection that it is called on.

Limitations of the fold Operation

Given an array of characters, use `fold` to return the vowel count:

```
Array(‘E‘, ‘P‘, ‘F‘, ‘L‘).par
  .fold(0)((count, c) => if (isVowel(c)) count + 1 else count)

// does not compile -- 0 is not a Char
```

The `fold` operation can only produce values of the same type as the collection that it is called on.

The `foldLeft` operation is *more expressive* than `fold`. Sanity check:

```
def fold(z: A)(op: (A, A) => A): A = foldLeft[A](z)(op)
```

The aggregate Operation

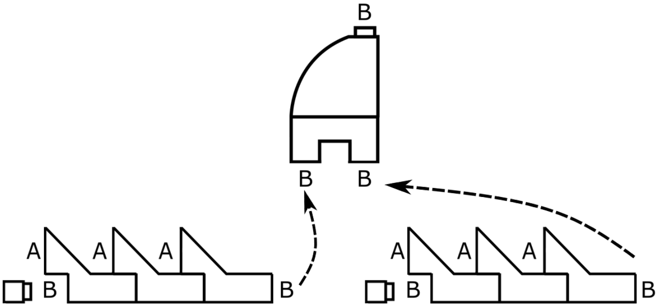
Let's examine the aggregate signature:

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

The aggregate Operation

Let's examine the aggregate signature:

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



A combination of foldLeft and fold.

Using the aggregate Operation

Count the number of vowels in a character array:

Using the aggregate Operation

Count the number of vowels in a character array:

```
Array('E', 'P', 'F', 'L').par.aggregate(0)(
  (count, c) => if (isVowel(c)) count + 1 else count,
  _ + _
)
```

The Transformer Operations

So far, we saw the *accessor* combinators.

Transformer combinators, such as `map`, `filter`, `flatMap` and `groupBy`, do not return a single value, but instead return new collections as results.

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Scala Parallel Collections

Parallel Programming in Scala

Aleksandar Prokopec

Scala Collections Hierarchy

- `Traversable[T]` – collection of elements with type `T`, with operations implemented using `foreach`

Scala Collections Hierarchy

- `Traversable[T]` – collection of elements with type `T`, with operations implemented using `foreach`
- `Iterable[T]` – collection of elements with type `T`, with operations implemented using `iterator`

Scala Collections Hierarchy

- `Traversable[T]` – collection of elements with type `T`, with operations implemented using `foreach`
- `Iterable[T]` – collection of elements with type `T`, with operations implemented using `iterator`
- `Seq[T]` – an ordered sequence of elements with type `T`

Scala Collections Hierarchy

- `Traversable[T]` – collection of elements with type `T`, with operations implemented using `foreach`
- `Iterable[T]` – collection of elements with type `T`, with operations implemented using `iterator`
- `Seq[T]` – an ordered sequence of elements with type `T`
- `Set[T]` – a set of elements with type `T` (no duplicates)

Scala Collections Hierarchy

- `Traversable[T]` – collection of elements with type `T`, with operations implemented using `foreach`
- `Iterable[T]` – collection of elements with type `T`, with operations implemented using `iterator`
- `Seq[T]` – an ordered sequence of elements with type `T`
- `Set[T]` – a set of elements with type `T` (no duplicates)
- `Map[K, V]` – a map of keys with type `K` associated with values of type `V` (no duplicate keys)

Parallel Collection Hierarchy

Traits `ParIterable[T]`, `ParSeq[T]`, `ParSet[T]` and `ParMap[K, V]` are the parallel counterparts of different sequential traits.

Parallel Collection Hierarchy

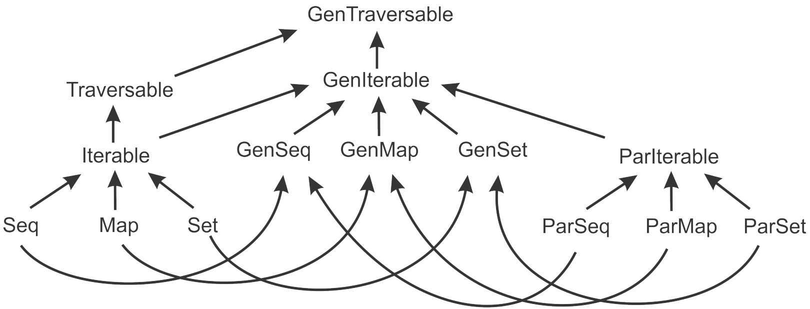
Traits `ParIterable[T]`, `ParSeq[T]`, `ParSet[T]` and `ParMap[K, V]` are the parallel counterparts of different sequential traits.

For code that is *agnostic* about parallelism, there exists a separate hierarchy of *generic* collection traits `GenIterable[T]`, `GenSeq[T]`, `GenSet[T]` and `GenMap[K, V]`.

Parallel Collection Hierarchy

Traits `ParIterable[T]`, `ParSeq[T]`, `ParSet[T]` and `ParMap[K, V]` are the parallel counterparts of different sequential traits.

For code that is *agnostic* about parallelism, there exists a separate hierarchy of *generic* collection traits `GenIterable[T]`, `GenSeq[T]`, `GenSet[T]` and `GenMap[K, V]`.



Writing Parallelism-Agnostic Code

Generic collection traits allow us to write code that is unaware of parallelism.

Example – find the largest palindrome in the sequence:

```
def largestPalindrome(xs: GenSeq[Int]): Int = {
  xs.aggregate(Int.MinValue)(
    (largest, n) =>
    if (n > largest && n.toString == n.toString.reverse) n else largest,
    math.max
  )
}
val array = (0 until 1000000).toArray
```

Writing Parallelism-Agnostic Code

Generic collection traits allow us to write code that is unaware of parallelism.

Example – find the largest palindrome in the sequence:

```
def largestPalindrome(xs: GenSeq[Int]): Int = {
  xs.aggregate(Int.MinValue)(
    (largest, n) =>
    if (n > largest && n.toString == n.toString.reverse) n else largest,
    math.max
  )
}
val array = (0 until 1000000).toArray

largestPalindrome(array)
```

Writing Parallelism-Agnostic Code

Generic collection traits allow us to write code that is unaware of parallelism.

Example – find the largest palindrome in the sequence:

```
def largestPalindrome(xs: GenSeq[Int]): Int = {
  xs.aggregate(Int.MinValue)(
    (largest, n) =>
    if (n > largest && n.toString == n.toString.reverse) n else largest,
    math.max
  )
}
val array = (0 until 1000000).toArray

largestPalindrome(array)

largestPalindrome(array.par)
```

Non-Parallelizable Collections

A sequential collection can be converted into a parallel one by calling `par`.

```
val vector = Vector.fill(10000000)("")
val list = vector.toList
```

Non-Parallelizable Collections

A sequential collection can be converted into a parallel one by calling `par`.

```
val vector = Vector.fill(10000000)("")
val list = vector.toList
```

```
vector.par // creates a ParVector[String]
list.par // also creates a ParVector[String]
```

Parallelizable Collections

- `ParArray[T]` – parallel array of objects, counterpart of `Array` and `ArrayBuffer`

Parallelizable Collections

- `ParArray[T]` – parallel array of objects, counterpart of `Array` and `ArrayBuffer`
- `ParRange` – parallel range of integers, counterpart of `Range`

Parallelizable Collections

- `ParArray[T]` – parallel array of objects, counterpart of `Array` and `ArrayBuffer`
- `ParRange` – parallel range of integers, counterpart of `Range`
- `ParVector[T]` – parallel vector, counterpart of `Vector`

Parallelizable Collections

- `ParArray[T]` – parallel array of objects, counterpart of `Array` and `ArrayBuffer`
- `ParRange` – parallel range of integers, counterpart of `Range`
- `ParVector[T]` – parallel vector, counterpart of `Vector`
- `immutable.ParHashSet[T]` – counterpart of `immutable.HashSet`

Parallelizable Collections

- ParArray[T] – parallel array of objects, counterpart of Array and ArrayBuffer
- ParRange – parallel range of integers, counterpart of Range
- ParVector[T] – parallel vector, counterpart of Vector
- immutable.ParHashSet[T] – counterpart of immutable.HashSet
- immutable.ParHashMap[K, V] – counterpart of immutable.HashMap

Parallelizable Collections

- ParArray[T] – parallel array of objects, counterpart of Array and ArrayBuffer
- ParRange – parallel range of integers, counterpart of Range
- ParVector[T] – parallel vector, counterpart of Vector
- immutable.ParHashSet[T] – counterpart of immutable.HashSet
- immutable.ParHashMap[K, V] – counterpart of immutable.HashMap
- mutable.ParHashSet[T] – counterpart of mutable.HashSet

Parallelizable Collections

- `ParArray[T]` – parallel array of objects, counterpart of `Array` and `ArrayBuffer`
- `ParRange` – parallel range of integers, counterpart of `Range`
- `ParVector[T]` – parallel vector, counterpart of `Vector`
- `immutable.ParHashSet[T]` – counterpart of `immutable.HashSet`
- `immutable.ParHashMap[K, V]` – counterpart of `immutable.HashMap`
- `mutable.ParHashSet[T]` – counterpart of `mutable.HashSet`
- `mutable.PasHashMap[K, V]` – counterpart of `mutable.HashMap`

Parallelizable Collections

- ParArray[T] – parallel array of objects, counterpart of Array and ArrayBuffer
- ParRange – parallel range of integers, counterpart of Range
- ParVector[T] – parallel vector, counterpart of Vector
- immutable.ParHashSet[T] – counterpart of immutable.HashSet
- immutable.ParHashMap[K, V] – counterpart of immutable.HashMap
- mutable.ParHashSet[T] – counterpart of mutable.HashSet
- mutable.PasHashMap[K, V] – counterpart of mutable.HashMap
- ParTrieMap[K, V] – thread-safe parallel map with atomic snapshots, counterpart of TrieMap

Parallelizable Collections

- `ParArray[T]` – parallel array of objects, counterpart of `Array` and `ArrayBuffer`
- `ParRange` – parallel range of integers, counterpart of `Range`
- `ParVector[T]` – parallel vector, counterpart of `Vector`
- `immutable.ParHashSet[T]` – counterpart of `immutable.HashSet`
- `immutable.ParHashMap[K, V]` – counterpart of `immutable.HashMap`
- `mutable.ParHashSet[T]` – counterpart of `mutable.HashSet`
- `mutable.PasHashMap[K, V]` – counterpart of `mutable.HashMap`
- `ParTrieMap[K, V]` – thread-safe parallel map with atomic snapshots, counterpart of `TrieMap`
- for other collections, `par` creates the closest parallel collection – e.g. a `List` is converted to a `ParVector`

Computing Set Intersection

```
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
}
intersection((0 until 1000).toSet, (0 until 1000 by 4).toSet)
intersection((0 until 1000).par.toSet, (0 until 1000 by 4).par.toSet)
```

Computing Set Intersection

```
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
}
intersection((0 until 1000).toSet, (0 until 1000 by 4).toSet)
intersection((0 until 1000).par.toSet, (0 until 1000 by 4).par.toSet)
```

Question: Is this program correct?

- Yes.
- No.

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
}
intersection((0 until 1000).toSet, (0 until 1000 by 4).toSet)
intersection((0 until 1000).par.toSet, (0 until 1000 by 4).par.toSet)
```

Rule: Avoid mutations to the same memory locations without proper synchronization.

Synchronizing Side-Effects

Solution – use a concurrent collection, which can be mutated by multiple threads:

```
import java.util.concurrent._
def intersection(a: GenSet[Int], b: GenSet[Int]) = {
  val result = new ConcurrentSkipListSet[Int]()
  for (x <- a) if (b contains x) result += x
  result
}
intersection((0 until 1000).toSet, (0 until 1000 by 4).toSet)
intersection((0 until 1000).par.toSet, (0 until 1000 by 4).par.toSet)
```

Avoiding Side-Effects

Side-effects can be avoided by using the correct combinators. For example, we can use `filter` to compute the intersection:

```
def intersection(a: GenSet[Int], b: GenSet[Int]): GenSet[Int] = {
  if (a.size < b.size) a.filter(b(_))
  else b.filter(a(_))
}
intersection((0 until 1000).toSet, (0 until 1000 by 4).toSet)
intersection((0 until 1000).par.toSet, (0 until 1000 by 4).par.toSet)
```

Concurrent Modifications During Traversals

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

```
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)
val violation = graph.find({ case (i, v) => v != (i + 2) % graph.size })
println(s"violation: $violation")
```

Concurrent Modifications During Traversals

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

```
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)
val violation = graph.find({ case (i, v) => v != (i + 2) % graph.size })
println(s"violation: $violation")
```

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

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

The TrieMap Collection

TrieMap is an exception to these rules.

The snapshot method 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(v)
val violation = graph.find({ case (i, v) => v != (i + 2) % graph.size })
println(s"violation: $violation")
```

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Splitters and Combiners

Parallel Programming in Scala

Aleksandar Prokopec

Data-Parallel Abstractions

We will study the following abstractions:

- iterators
- splitters
- builders
- combiners

Iterator

The simplified `Iterator` trait is as follows:

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

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

Iterator

The simplified `Iterator` trait is as follows:

```
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`

Using Iterators

Question: How would you implement `foldLeft` on an iterator?

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

Using Iterators

Question: How would you implement `foldLeft` on an iterator?

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

Splitter

The simplified `Splitter` trait is as follows:

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

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

Splitter

The simplified `Splitter` trait is as follows:

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

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

The *splitter 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 an efficient method – $O(\log n)$ or better

Using Splitters

Question: How would you implement `fold` on a splitter?

```
def fold(z: A)(f: (A, A) => A): A
```

Using Splitters

Question: How would you implement `fold` on a splitter?

```
def fold(z: A)(f: (A, A) => A): A = {
  if (remaining < threshold) foldLeft(z)(f)
```

Using Splitters

Question: How would you implement `fold` on a splitter?

```
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

The simplified `Builder` trait is as follows:

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

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

The *builder 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

Using Builders

Question: How would you implement the `filter` method using `newBuilder`?

```
def filter(p: T => Boolean): Repr
```

Using Builders

Question: How would you implement the `filter` method using `newBuilder`?

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

Combiner

The simplified `Combiner` trait is as follows:

```
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
```

The *combiner contract*:

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

Using Combiners

Question: How would you implement a parallel `filter` method using `splitter` and `newCombiner`?