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

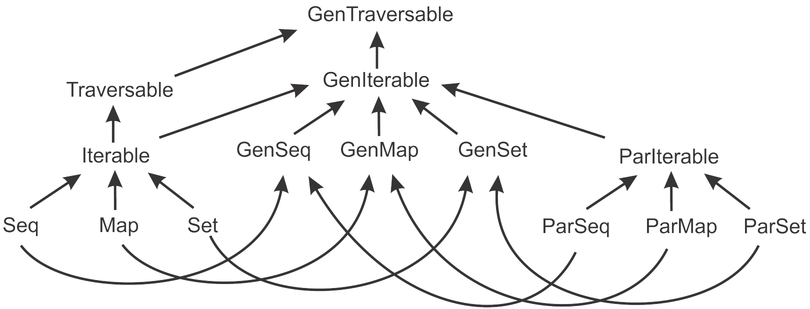
Traits `ParIterable[T]`, `ParSeq[T]`, `ParSet[T]` and `ParMap[K, V]` are the parallel counterparts of different sequential traits.

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

```scala
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")
```