ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Scala Parallel Collections

Parallel Programming and Data Analysis

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

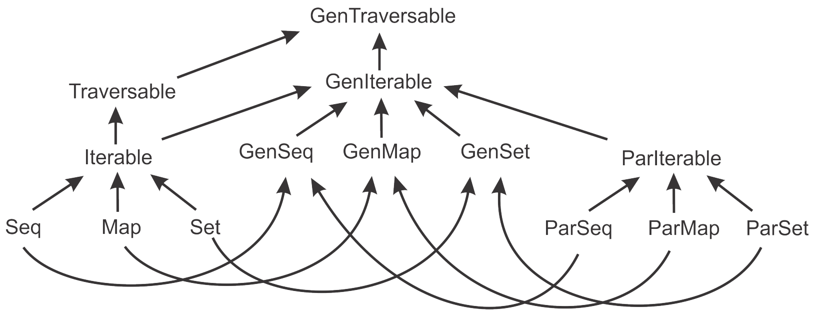
Traits `ParIterable[T]`, `ParSeq[T]`, `ParSet[T]` and `ParMap[K, V]` are the parallel counterparts of different sequential traits.

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

E.g. 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

E.g. find the largest palindrome in the sequence:

```scala
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

E.g. 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. Let's the performance difference:

```
val array = Array.fill(10000000)("")
val list = array.toList
```

Non-Parallelizable Collections

A sequential collection can be converted into a parallel one by calling `par`. Let's the performance difference:

```
val array = Array.fill(10000000)("")
val list = array.toList
```

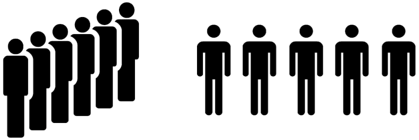
Converting an array is $65\times$ faster. Why is that?

Non-Parallelizable Collections

A sequential collection can be converted into a parallel one by calling `par`. Let's the performance difference:

```
val array = Array.fill(10000000)("")
val list = array.toList
```

Converting an array is $65\times$ faster. Why is that?



- It's hard to hand out all the tasks simultaneously to every worker in a long queue.
- In a factory, it's easy to simultaneously give everybody some work.

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`
- `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`
- `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`
- `mutable.ParHashSet[T]` – counterpart of `mutable.HashSet`
- `mutable.PasHashMap[K, V]` – counterpart of `mutable.HashMap`
- `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`
- `mutable.ParHashSet[T]` – counterpart of `mutable.HashSet`
- `mutable.PasHashMap[K, V]` – counterpart of `mutable.HashMap`
- `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`
- `mutable.ParHashSet[T]` – counterpart of `mutable.HashSet`
- `mutable.PasHashMap[K, V]` – counterpart of `mutable.HashMap`
- `immutable.ParHashSet[T]` – counterpart of `immutable.HashSet`
- `immutable.ParHashMap[K, V]` – counterpart of `immutable.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`
- `mutable.ParHashSet[T]` – counterpart of `mutable.HashSet`
- `mutable.PasHashMap[K, V]` – counterpart of `mutable.HashMap`
- `immutable.ParHashSet[T]` – counterpart of `immutable.HashSet`
- `immutable.ParHashMap[K, V]` – counterpart of `immutable.HashMap`
- `ParTrieMap[K, V]` – thread-safe parallel map with atomic snapshots, counterpart of `TrieMap`
- for other collections, `par` creates the most similar parallel collection – e.g. a `List` is converted to a `ParVector`

Side-Effecting Operations

Rule 1: Avoid mutations to the same memory locations without proper synchronization.

```
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)
```

Side-Effecting Operations

Rule 1: Avoid mutations to the same memory locations without proper synchronization.

```
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 code correct?

- Yes
- No

Avoiding Side-Effects

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

```scala
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 2: Never modify a parallel collection on which a data-parallel operation is in progress.

```
val array = Array.fill(10000000)("")
val (result, _) = common.parallel(
  array.par.count(_ == ""),
  for (i <- (0 until 10000000).par) array(i) = "modified"
)
println(s"result: $result")
```

Concurrent Modifications During Traversals

Rule 2: Never modify a parallel collection on which a data-parallel operation is in progress.

```
val array = Array.fill(10000000)("")
val (result, _) = common.parallel(
  array.par.count(_ == ""),
  for (i <- (0 until 10000000).par) array(i) = "modified"
)
println(s"result: $result")
```

- We read from a collection that is concurrently modified.
- We write to a collection that is concurrently traversed.

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

The TrieMap Collection

TrieMap is an exception to the previous rule.

Consider the Game of Life simulation:

```
val cells = TrieMap[(Int, Int), Cell]()
def step() {
  for ((xy, cell) <- cells.par) cells(xy) = update(cell)
}
```

We can traverse and modify the trie at the same time.

Game of Life Demo

Game of Life using `TrieMap` demo!