Scala Collections
How many ways are there to say "Multiple Things"

Julian Bieber

December 13, 2018

Collections Trivia

Sequence

```
val sequence = Seq(1, 2, 3, 4, 5)
println(sequence)
```

Sequence

```
val sequence = Seq(1, 2, 3, 4, 5)
println(sequence)
```

```
List(1, 2, 3, 4, 5)
```

Stream

```
val stream = Stream(1, 2, 3, 4, 5)
println(stream)
val seq: Seq[Int] = stream.toSeq
seq.map(println)
```

Stream

```
val stream = Stream(1, 2, 3, 4, 5)
println(stream)
val seq: Seq[Int] = stream.toSeq
seq.map(println)
```

```
Stream(1, ?)
1
```

Stream Consumption

```
val stream = Stream(1, 2, 3)
stream.foreach(println)
val streamPlusOne = stream.map(_ + 1)
println(stream.size)
streamPlusOne.foreach(println)
```

Stream Consumption

```
val stream = Stream(1, 2, 3)
stream.foreach(println)
val streamPlusOne = stream.map(_ + 1)
println(stream.size)
streamPlusOne.foreach(println)
```

```
1
2
3
3
2
3
4
```

Sequence Order of Execution

```
Seq(1, 2, 3).map{ i =>
    println("method1", i)
    i
}.map{ i =>
    println("method2", i)
    i
}
```

Sequence Order of Execution

```
Seq(1, 2, 3).map{ i =>
    println("method1", i)
    i
}.map{ i =>
    println("method2", i)
    i
}
```

```
(method1,1)
(method1,2)
(method1,3)
(method2,1)
(method2,2)
(method2,3)
```

Stream Order of Execution

```
Stream(1, 2, 3).map{ i =>
  println("method1", i)
  i
}.map{ i =>
  println("method2", i)
  i
}
```

Stream Order of Execution

```
Stream(1, 2, 3).map{ i =>
  println("method1", i)
  i
}.map{ i =>
  println("method2", i)
  i
}
```

```
(method1,1)
(method2,1)
```

Stream.force?

```
Stream(1, 2, 3).map{ i =>
  println("method1", i)
  i
}.map{ i =>
  println("method2", i)
  i
}.force
```

Stream.force?

```
Stream(1, 2, 3).map{ i =>
  println("method1", i)
  i
}.map{ i =>
  println("method2", i)
  i
}.force
```

```
(method1,1)
(method2,1)
(method1,2)
(method2,2)
(method1,3)
(method2,3)
```

Infinite Streams

```
val stream = Stream.from(1).map{ i =>
  println("method1", i)
  i
}.map{ i =>
  println("method2", i)
  i
}

println(stream.sum)
```

Infinite Streams

```
val stream = Stream.from(1).map{ i =>
  println("method1", i)
  i
}.map{ i =>
  println("method2", i)
  i
}

println(stream.sum)
```

```
...
(method1,2517132)
(method2,2517132)

Exception: java.lang.OutOfMemoryError
```

Infinite Iterators

```
val iterator = Stream.from(1).map{ i =>
  println("method1", i)
  i
}.map{ i =>
  println("method2", i)
  i
}.toIterator

println(iterator.sum)
```

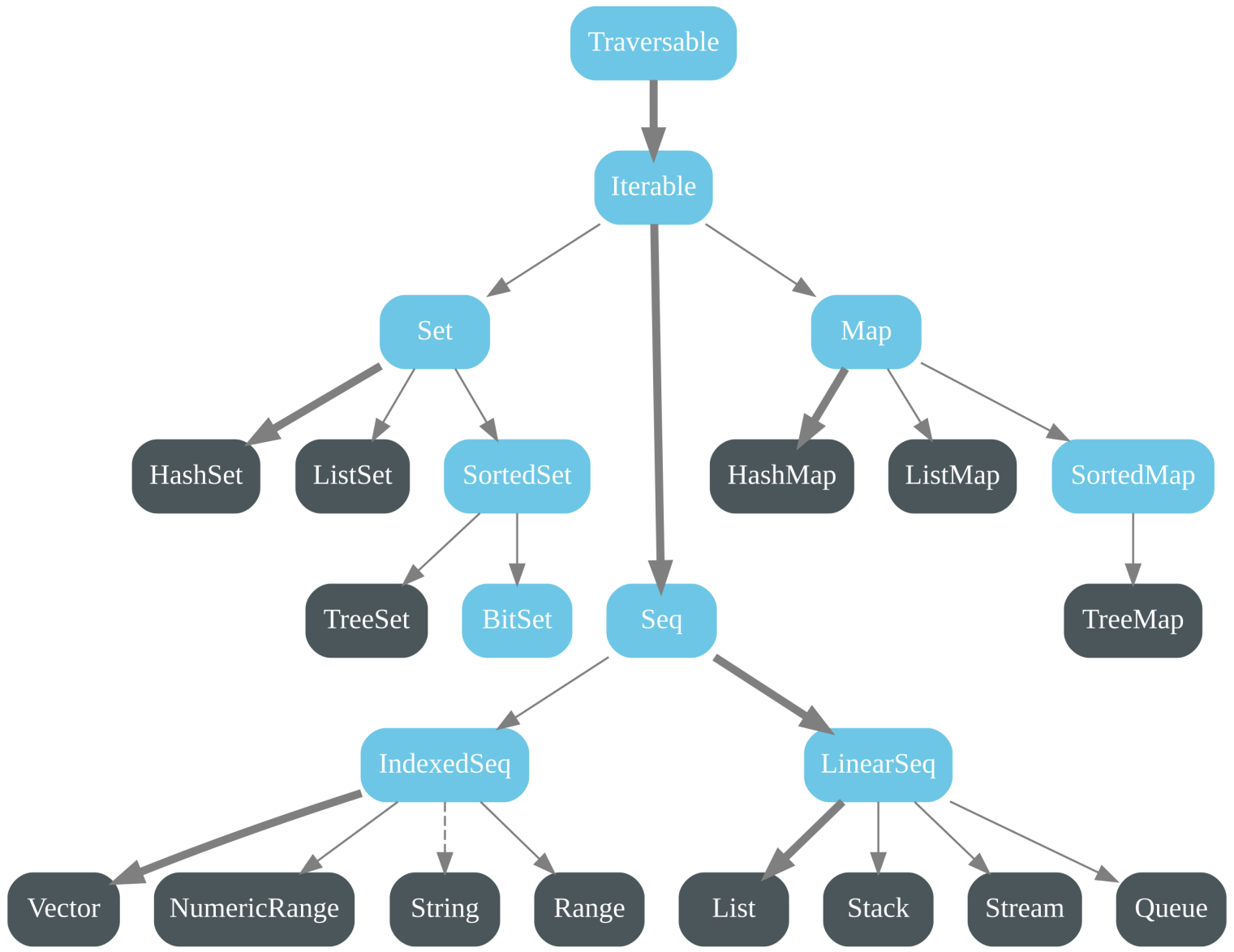
Infinite Iterators

```
val iterator = Stream.from(1).map{ i =>
  println("method1", i)
  i
}.map{ i =>
  println("method2", i)
  i
}.toIterator

println(iterator.sum)
```

```
...
(method1,2517132)
(method2,2517132)
...
```

Type Hierarchy

Immutabe


Traversable
Iterable
Set
Map
HashSet
ListSet
SortedSet
HashMap
ListMap
SortedMap
TreeSet
BitSet
Seq
TreeMap
IndexedSeq
LinearSeq
Vector
NumericRange
String
Range
List
Stack
Stream
Queue

Array

- slow prepend/append
- very fast random access ($O(1)$)
- contiguous locality
- strict

Vector

- fast random access ($O(c)$)
- fast append/prepend ($O(c)$)
- good but not contiguous locality
- strict

List

- very fast prepend ($O(1)$)
- very fast head access ($O(1)$)
- single linked $\Rightarrow$ slow random access ($O(n)$)
- bad locality
- strict

Stream

- List with lazy tail
- while something holds the head it can not be GC'ed
- will not be consumed by iterating (while something hold the head)
- not strict

Concept of a View

- calling view on a collection makes it non strict
- modifications can be applied as usual (map, filter, ...)
- however they are not evaluated yet

Concept of a View

- calling view on a collection makes it non strict
- modifications can be applied as usual (map, filter, . . .)
- however they are not evaluated yet
- force applies the changes
- better memory foodprint

Scala SeqView

- Signature: SeqView[+A, +Coll]
- Represents a view to Coll[+A]
- chaining maps without creating new collections
- $\Rightarrow$ less memory consumption

Scala SeqView

- Signature: SeqView[+A, +Coll]
- Represents a view to Coll[+A]
- chaining maps without creating new collections
- ⇒ less memory consumption
- ⇒ less GC activity

Scala SeqView

- Signature: SeqView[+A, +Coll]
- Represents a view to Coll[+A]
- chaining maps without creating new collections
- ⇒ less memory consumption
- ⇒ less GC activity
- ⇒ faster

SeqView

```
val seq: Seq[Int] = Seq(1, 2, 3).view.map{ i =>
  println("method1", i)
  i
}.map{ i =>
  println("method2", i)
  i
}.force
```

SeqView

```
val seq: Seq[Int] = Seq(1, 2, 3).view.map{ i =>
  println("method1", i)
  i
}.map{ i =>
  println("method2", i)
  i
}.force
```

```
(method1,1)
(method2,1)
(method1,2)
(method2,2)
(method1,3)
(method2,3)
```

SeqView

```
case class CaseClass(member: Int) {
  println("CaseClass", member)
}
println(function(Seq(1, 2, 3).view.map{ i =>
  println("method1", i)
  i
}.map{ i =>
  println("method2", i)
  i
}))

def function(seq: Seq[Int]): Seq[CaseClass] = {
  seq.map(CaseClass)
}
```

SeqView

SeqViewMMM(...)

SeqView

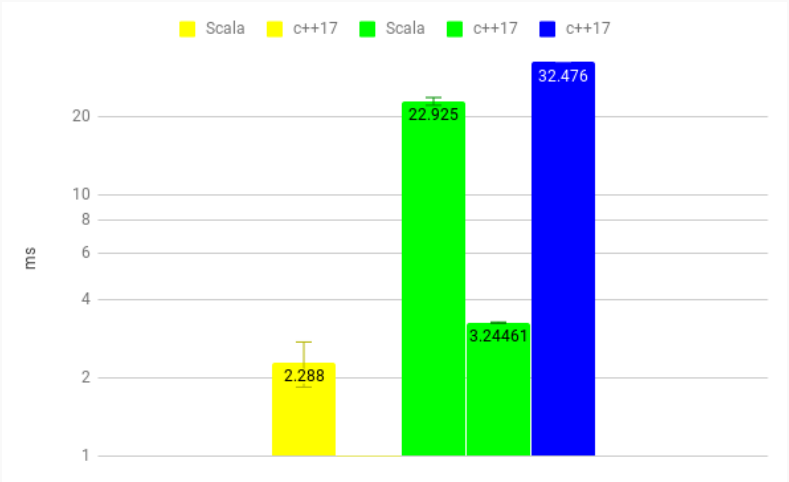
```
case class CaseClass(member: Int) {
  println("CaseClass", member)
}
println(function(Seq(1, 2, 3).view.map{ i =>
  println("method1", i)
  i
}.map{ i =>
  println("method2", i)
  i
}).view.force)

def function(seq: Seq[Int]): Seq[CaseClass] = {
seq.map(CaseClass)
}
```

SeqView

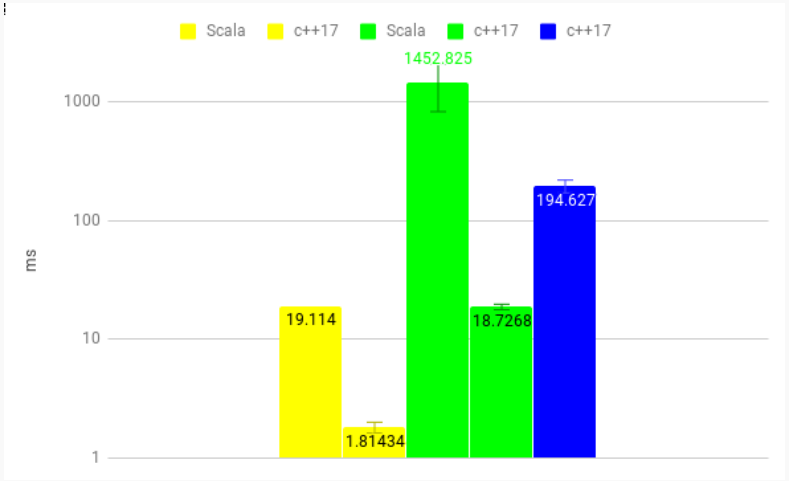
```
(method1,1)
(method2,1)
(CaseClass,1)
(method1,2)
(method2,2)
(CaseClass,2)
(method1,3)
(method2,3)
(CaseClass,3)
List(CaseClass(1), CaseClass(2), CaseClass(3))
```

Benchmarks

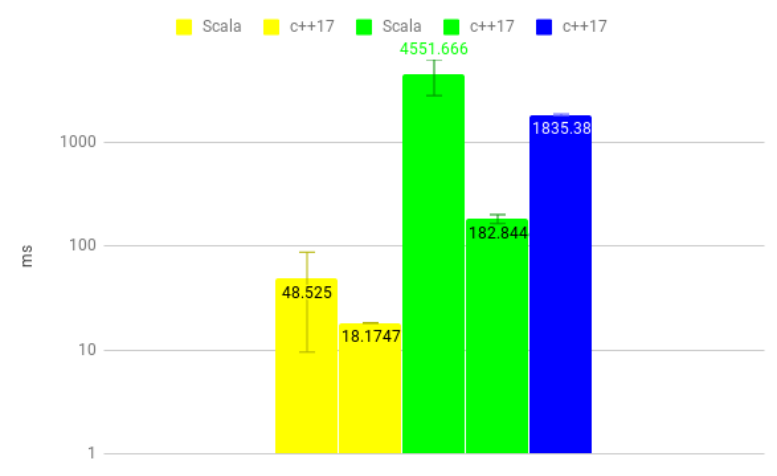
Forech with Baseline



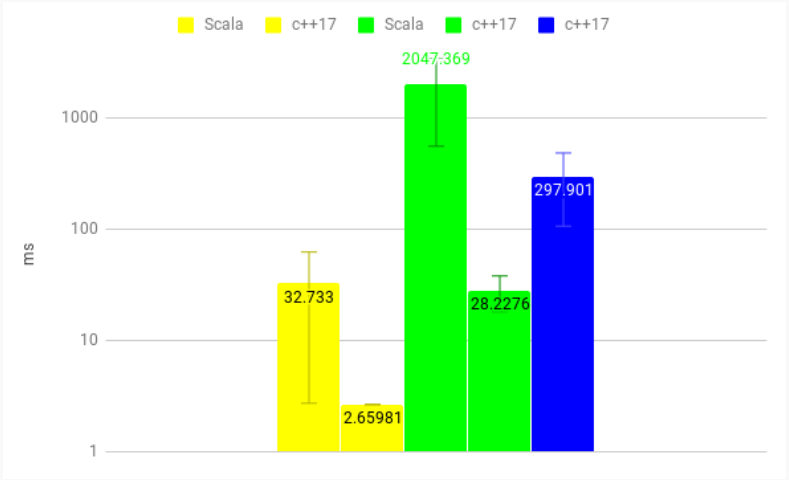
Map with Baseline



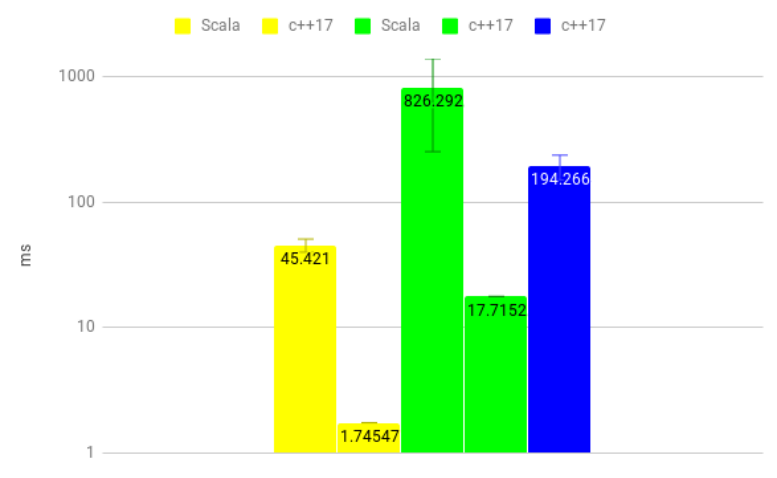
View map with Baseline



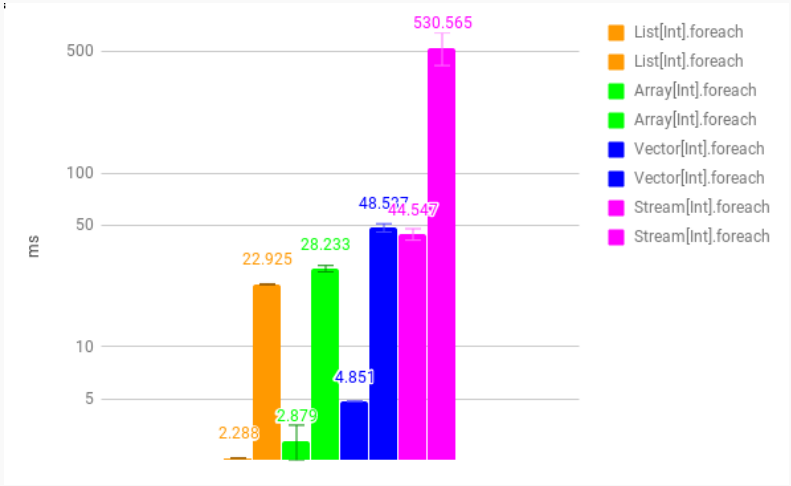
View map one tranfsormation with Baseline



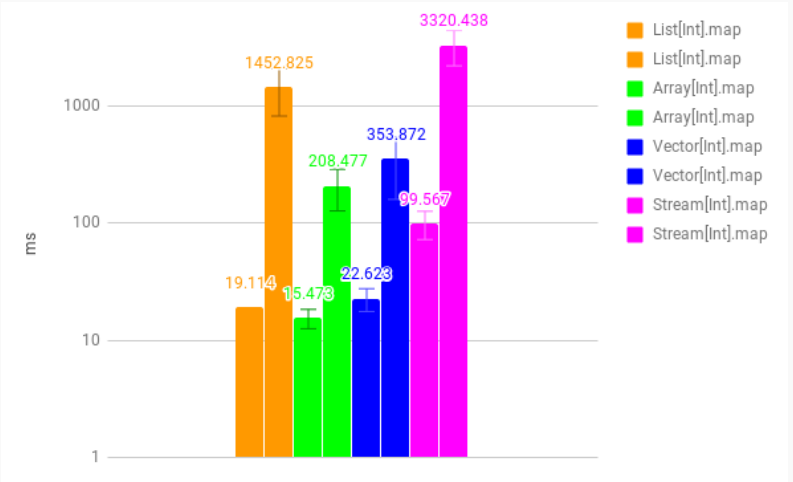
Parmap with Baseline



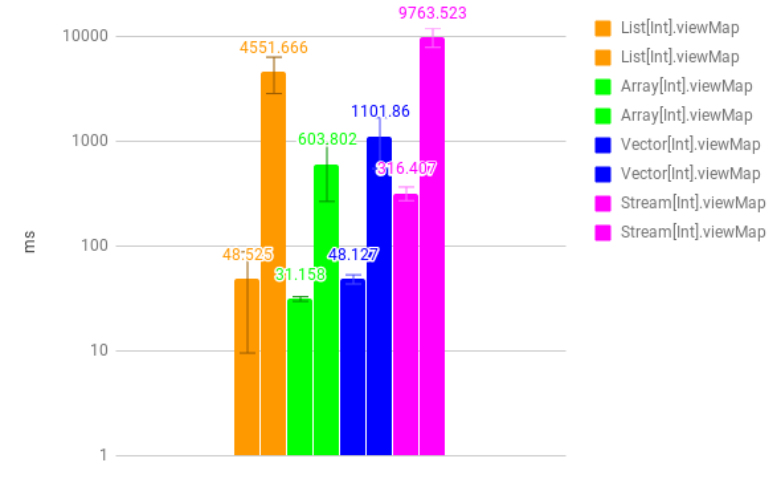
Foreach Scala collections



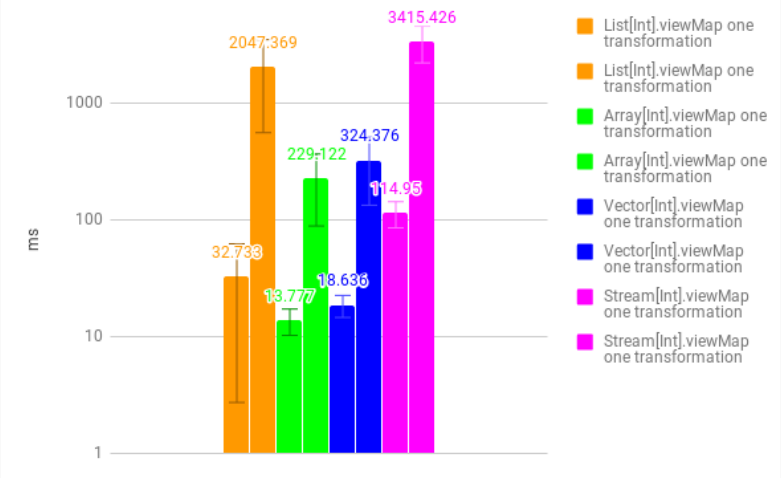
Map Scala collections



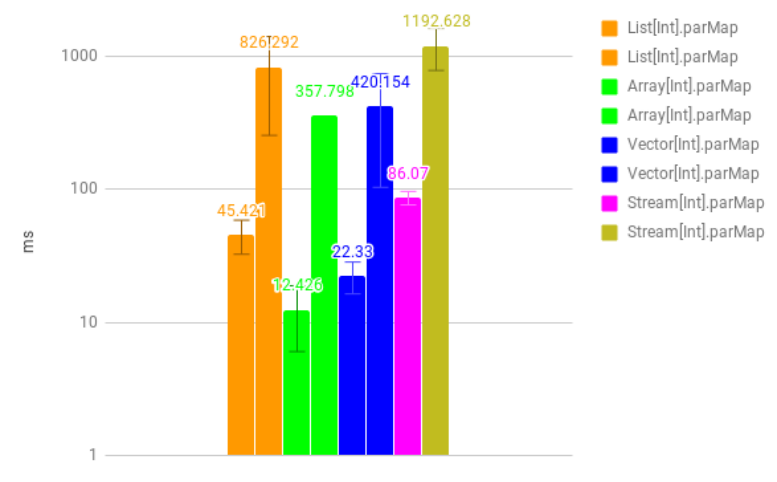
View map Scala collections



View map one transformation Scala collections



ParMap Scala collections



Further Reading

- http://www.lihaoyi.com/post/BenchmarkingScalaCollections
- https://github.com/julianbieber/ScalaCollectionsPresentation