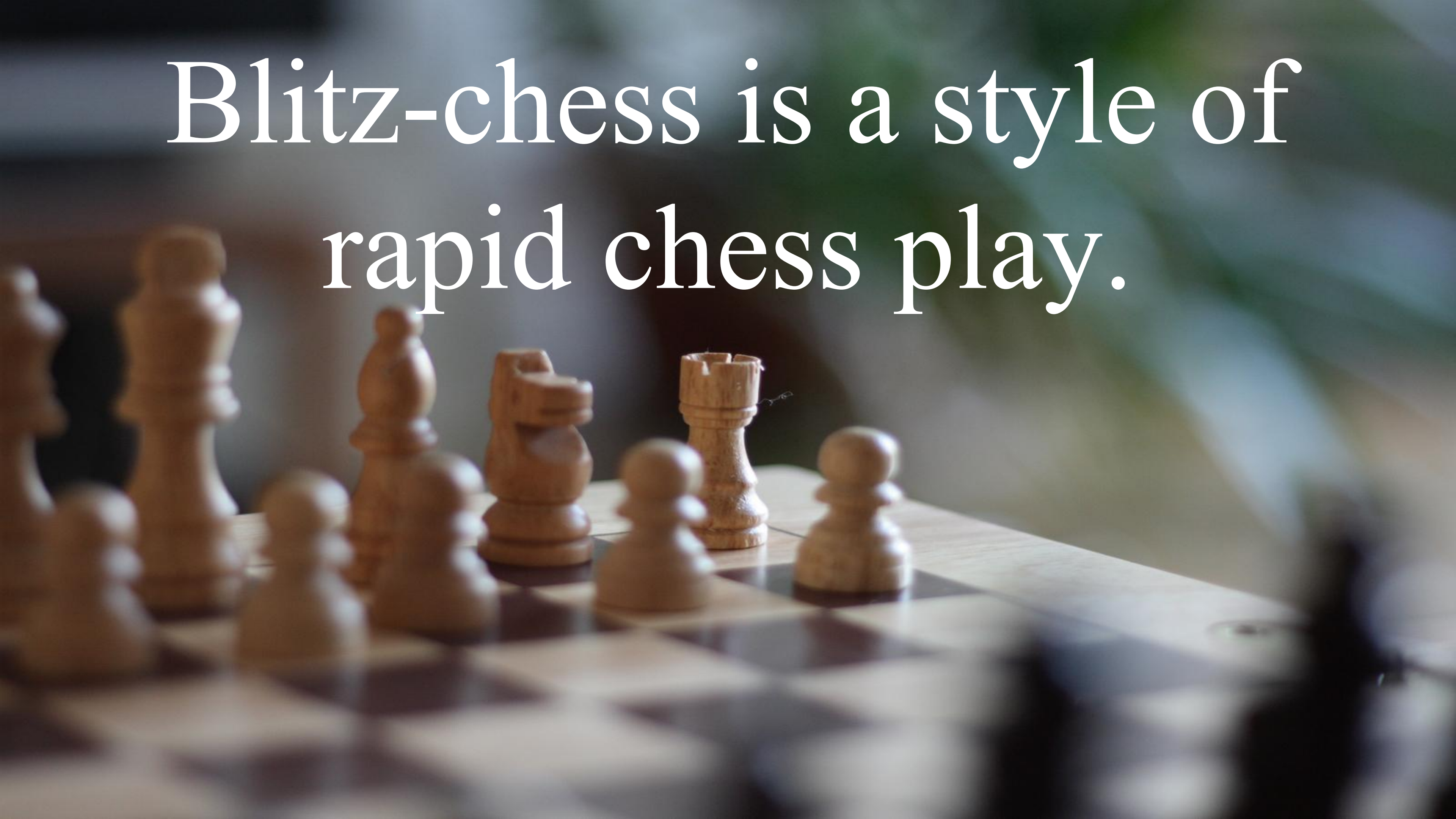
ScalaBlitz

Efficient Collections Framework

What’s a Blitz?



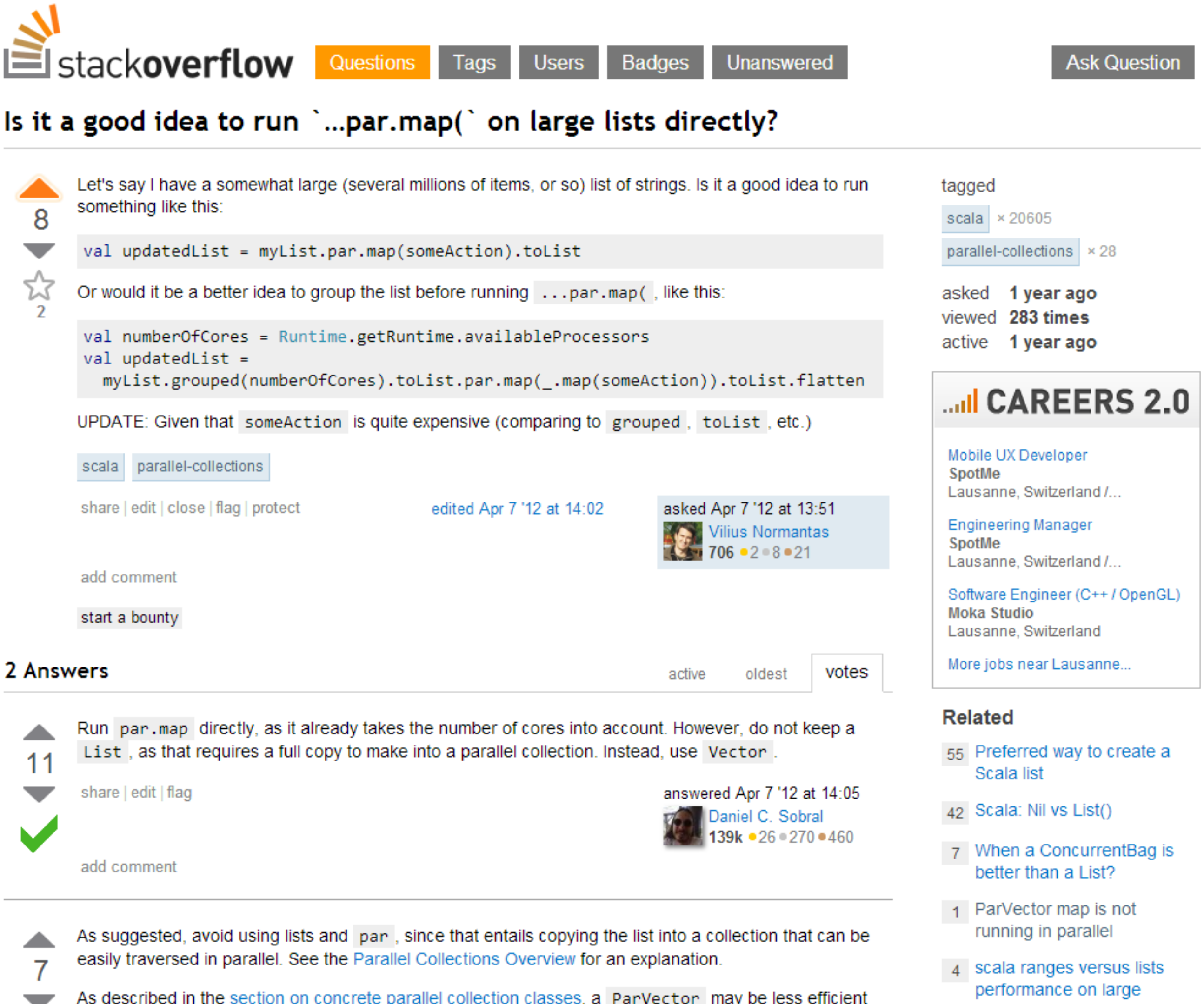
Blitz-chess is a style of rapid chess play.

Blitz-chess is a style of rapid chess play.

Knights have horses.

Horses run fast.

```
def mean(xs: Array[Float]): Float =
  xs.par.reduce(_ + _) / xs.length
```



Is it a good idea to run `...par.map(` on large lists directly?

Let's say I have a somewhat large (several millions of items, or so) list of strings. Is it a good idea to run something like this:

```
val updatedList = myList.par.map(someAction).toList
```

Or would it be a better idea to group the list before running `...par.map(`, like this:

```
val numberOfCores = Runtime.getRuntime.availableProcessors
val updatedList =
  myList.grouped(numberOfCores).toList.par.map(_.map(someAction)).toList.flatten
```

UPDATE: Given that `someAction` is quite expensive (comparing to `grouped`, `toList`, etc.)

edited Apr 7 '12 at 14:02

asked Apr 7 '12 at 13:51
Vilius Normantas
706 ●2 ●8 ●21

tagged
scala × 20605
parallel-collections × 28

asked 1 year ago
viewed 283 times
active 1 year ago

2 Answers

Run `par.map` directly, as it already takes the number of cores into account. However, do not keep a `List`, as that requires a full copy to make into a parallel collection. Instead, use `Vector`.

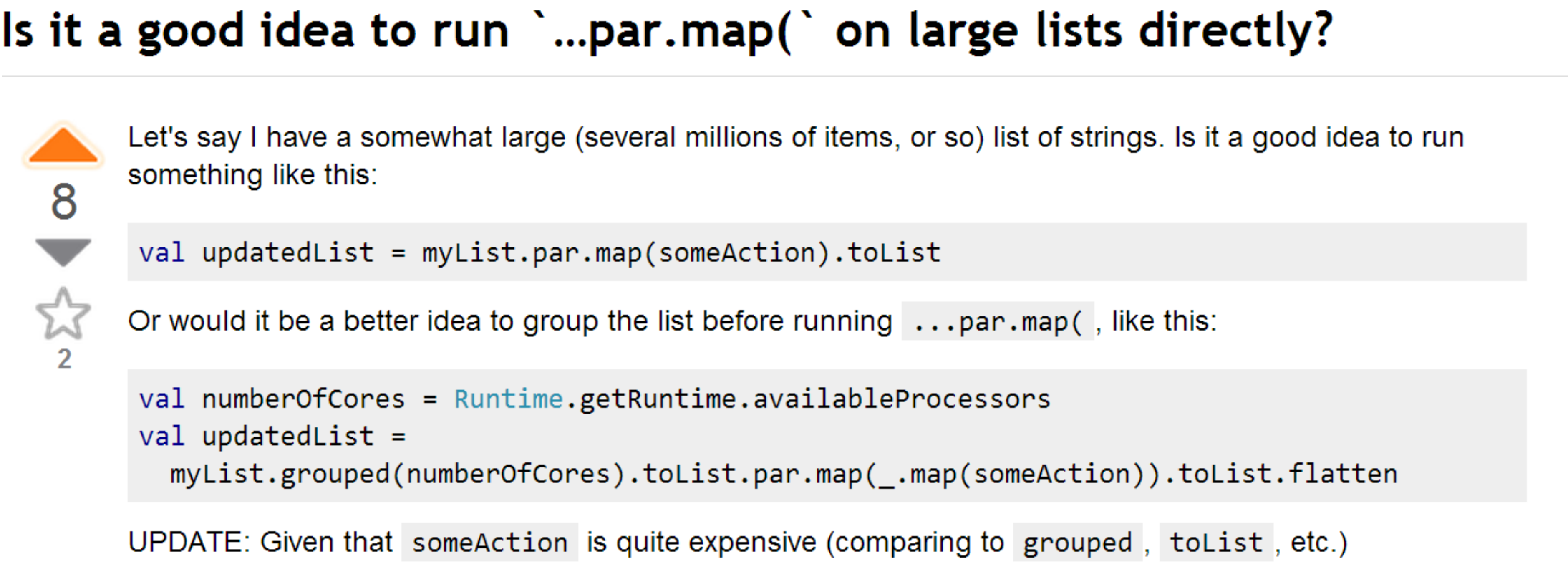
answered Apr 7 '12 at 14:05
Daniel C. Sobral
139k ●26 ●270 ●460

As suggested, avoid using lists and `par`, since that entails copying the list into a collection that can be easily traversed in parallel. See the Parallel Collections Overview for an explanation.

As described in the section on concrete parallel collection classes, a `ParVector` may be less efficient

Related

55	Preferred way to create a Scala list
42	Scala: Nil vs List()
7	When a ConcurrentBag is better than a List?
1	ParVector map is not running in parallel
4	scala ranges versus lists performance on large



Is it a good idea to run `...par.map(` on large lists directly?

Let's say I have a somewhat large (several millions of items, or so) list of strings. Is it a good idea to run something like this:

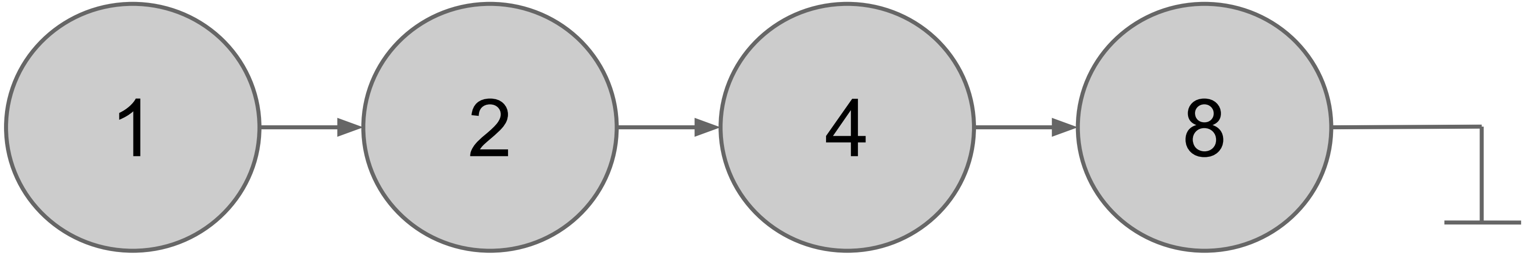
```
val updatedList = myList.par.map(someAction).toList
```

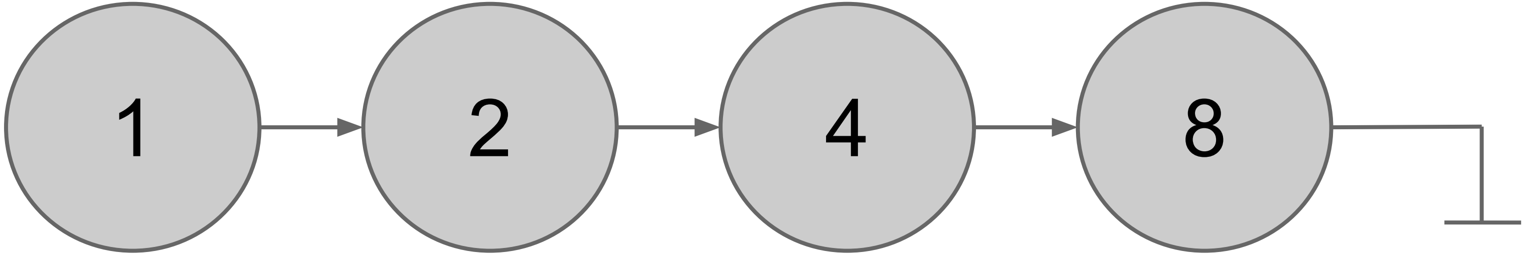
Or would it be a better idea to group the list before running `...par.map(`, like this:

```
val numberOfCores = Runtime.getRuntime.availableProcessors
val updatedList =
  myList.grouped(numberOfCores).toList.par.map(_.map(someAction)).toList.flatten
```

UPDATE: Given that `someAction` is quite expensive (comparing to `grouped`, `toList`, etc.)

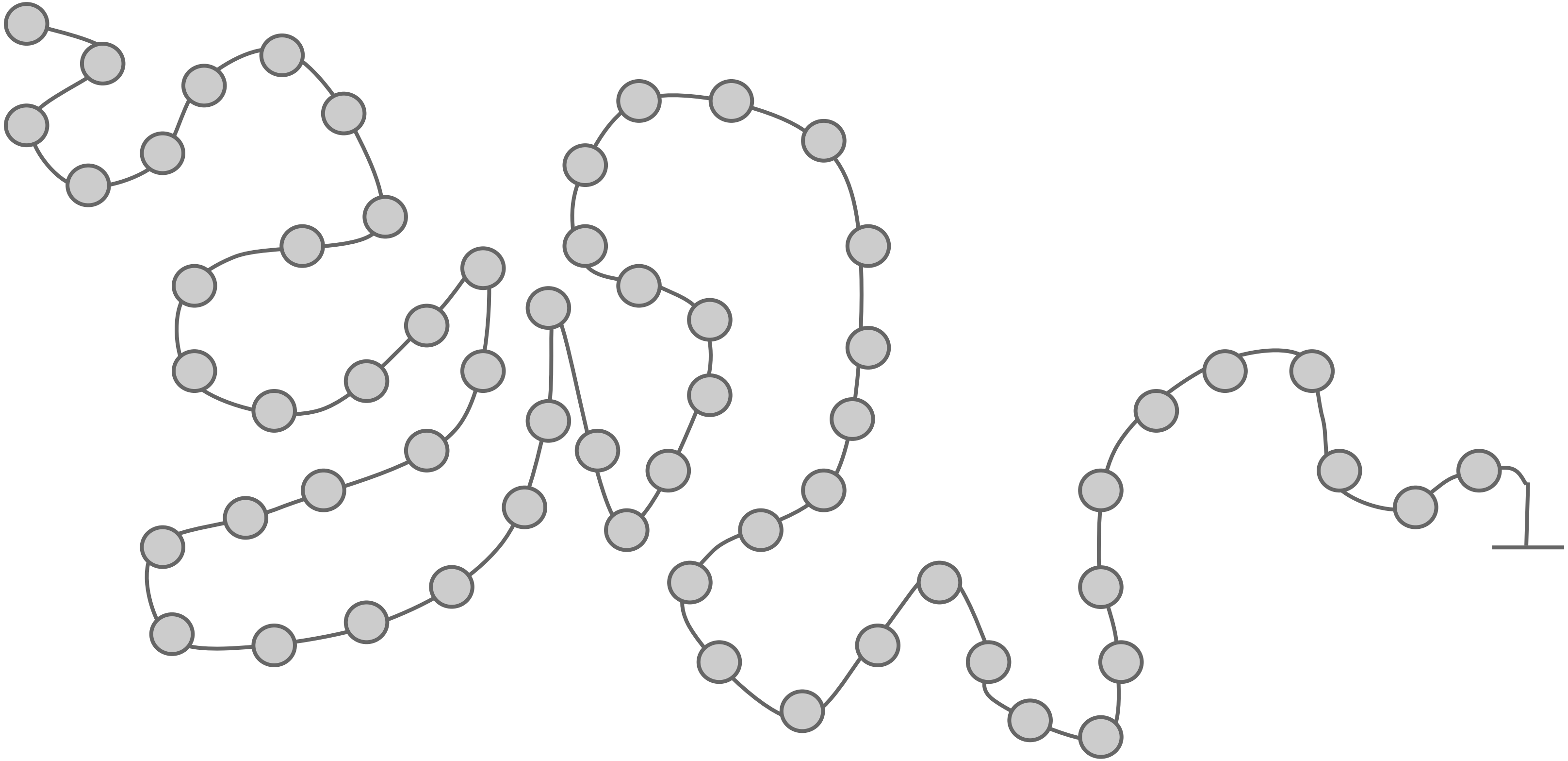
With `Lists`, operations can only be executed from left to right

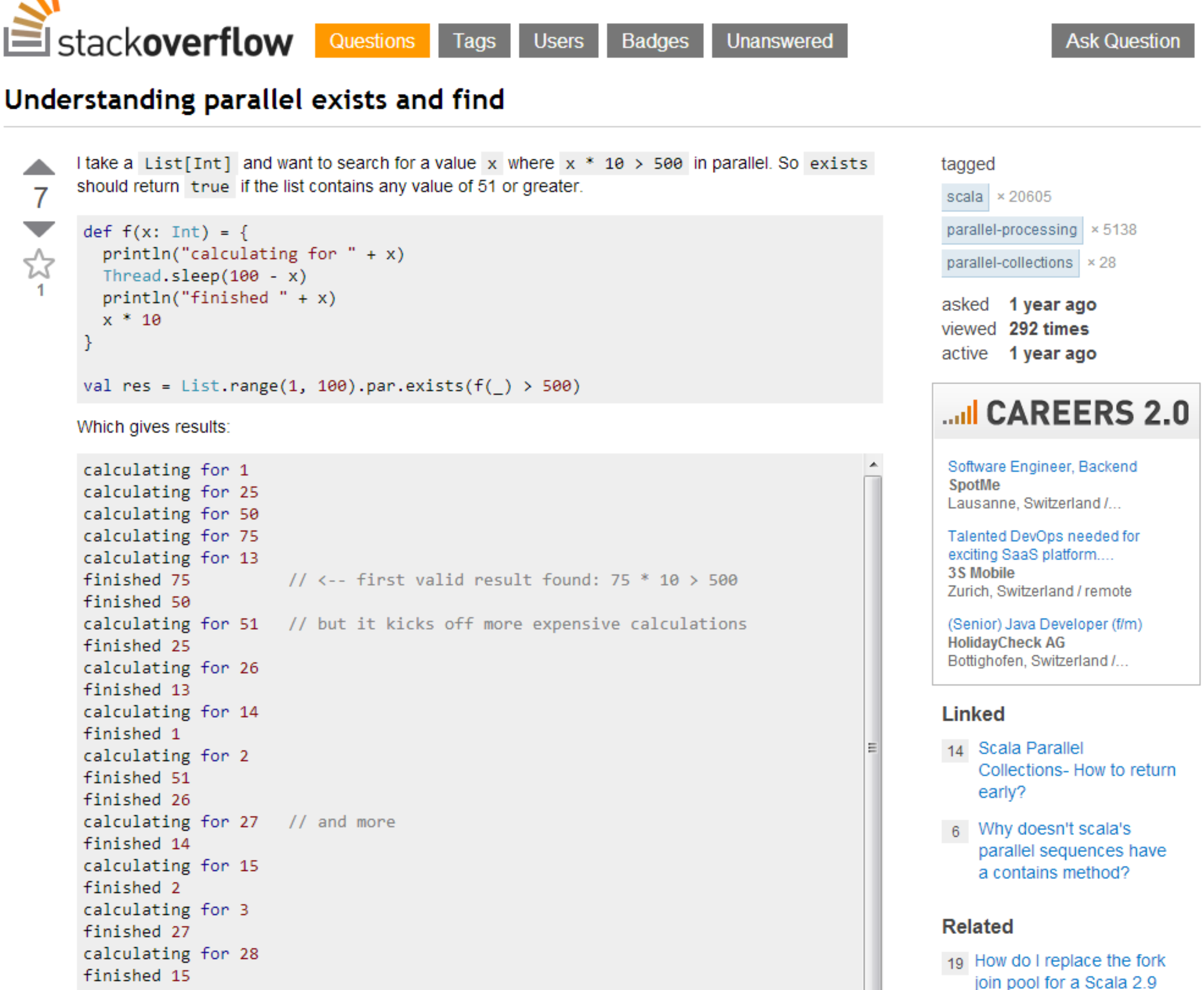

1
2
4
8



Not your typical list.

Bon app.





Understanding parallel exists and find

7

1

I take a `List[Int]` and want to search for a value `x` where `x * 10 > 500` in parallel. So `exists` should return `true` if the list contains any value of 51 or greater.

```
def f(x: Int) = {
  println("calculating for " + x)
  Thread.sleep(100 - x)
  println("finished " + x)
  x * 10
}

val res = List.range(1, 100).par.exists(f(_) > 500)
```

Which gives results:

```
calculating for 1
calculating for 25
calculating for 50
calculating for 75
calculating for 13
finished 75          // <-- first valid result found: 75 * 10 > 500
finished 50
calculating for 51   // but it kicks off more expensive calculations
finished 25
calculating for 26
finished 13
calculating for 14
finished 1
calculating for 2
finished 51
finished 26
calculating for 27   // and more
finished 14
calculating for 15
finished 2
calculating for 3
finished 27
calculating for 28
finished 15
```

tagged

scala × 20605

parallel-processing × 5138

parallel-collections × 28

asked **1 year ago**
viewed **292 times**
active **1 year ago**

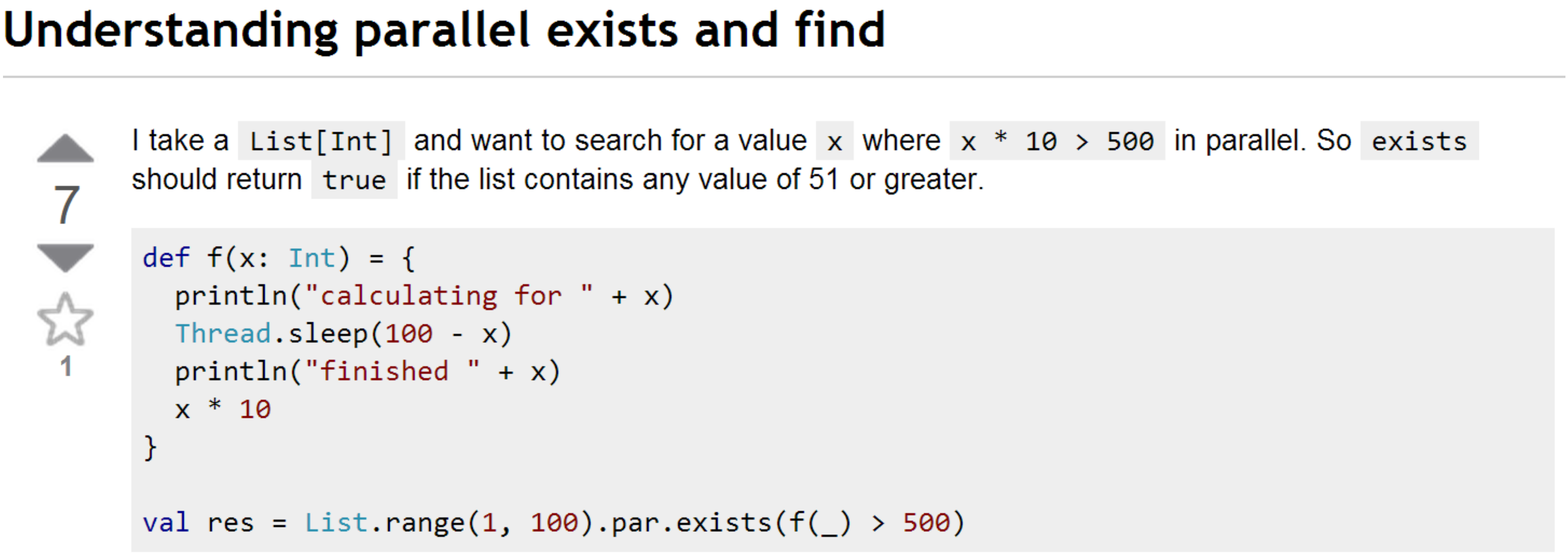
Linked

14 Scala Parallel Collections- How to return early?

6 Why doesn't scala's parallel sequences have a contains method?

Related

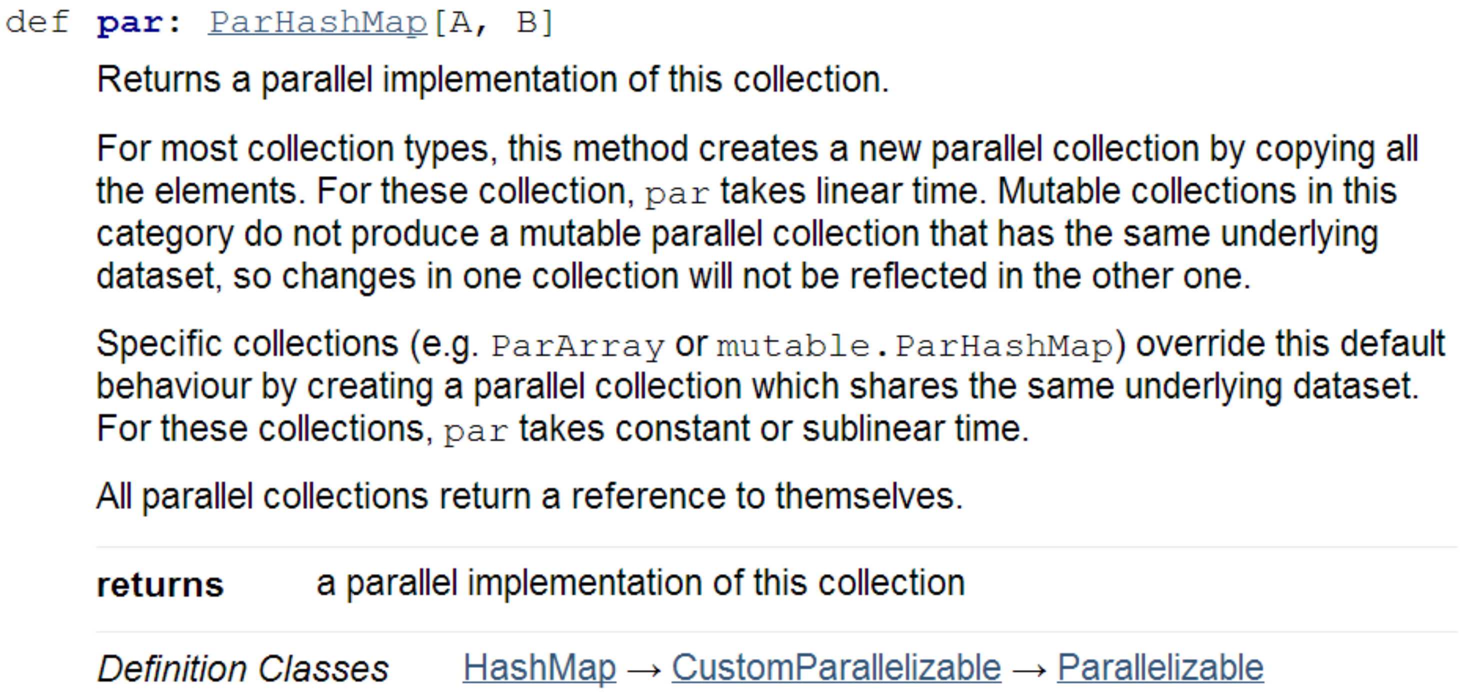
19 How do I replace the fork join pool for a Scala 2.9



Understanding parallel exists and find

I take a `List[Int]` and want to search for a value `x` where `x * 10 > 500` in parallel. So `exists` should return `true` if the list contains any value of 51 or greater.

```
def f(x: Int) = {
  println("calculating for " + x)
  Thread.sleep(100 - x)
  println("finished " + x)
  x * 10
}

val res = List.range(1, 100).par.exists(f(_) > 500)
```



```
def par: ParHashMap[A, B]
```

Returns a parallel implementation of this collection.

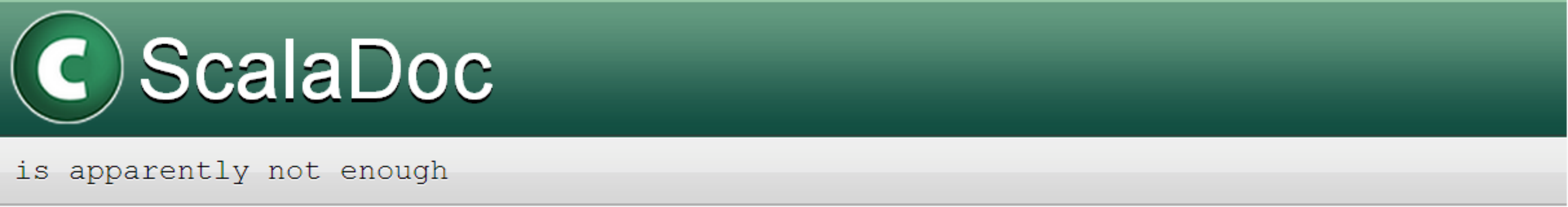
For most collection types, this method creates a new parallel collection by copying all the elements. For these collection, `par` takes linear time. Mutable collections in this category do not produce a mutable parallel collection that has the same underlying dataset, so changes in one collection will not be reflected in the other one.

Specific collections (e.g. `ParArray` or `mutable.ParHashMap`) override this default behaviour by creating a parallel collection which shares the same underlying dataset. For these collections, `par` takes constant or sublinear time.

All parallel collections return a reference to themselves.

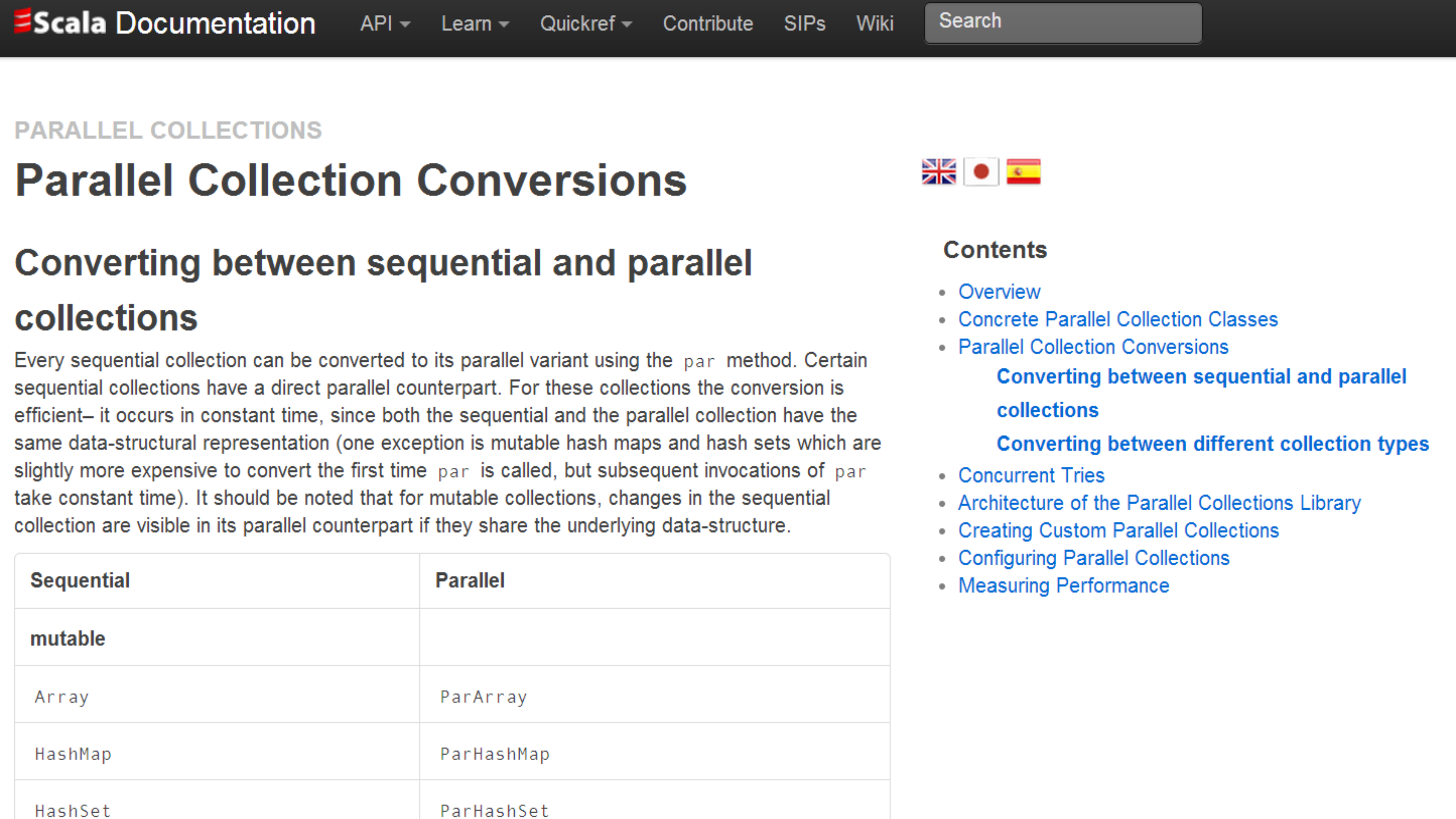
returns a parallel implementation of this collection

Definition Classes HashMap → CustomParallelizable → Parallelizable



```
is apparently not enough
```

Apparently not enough



PARALLEL COLLECTIONS

Parallel Collection Conversions

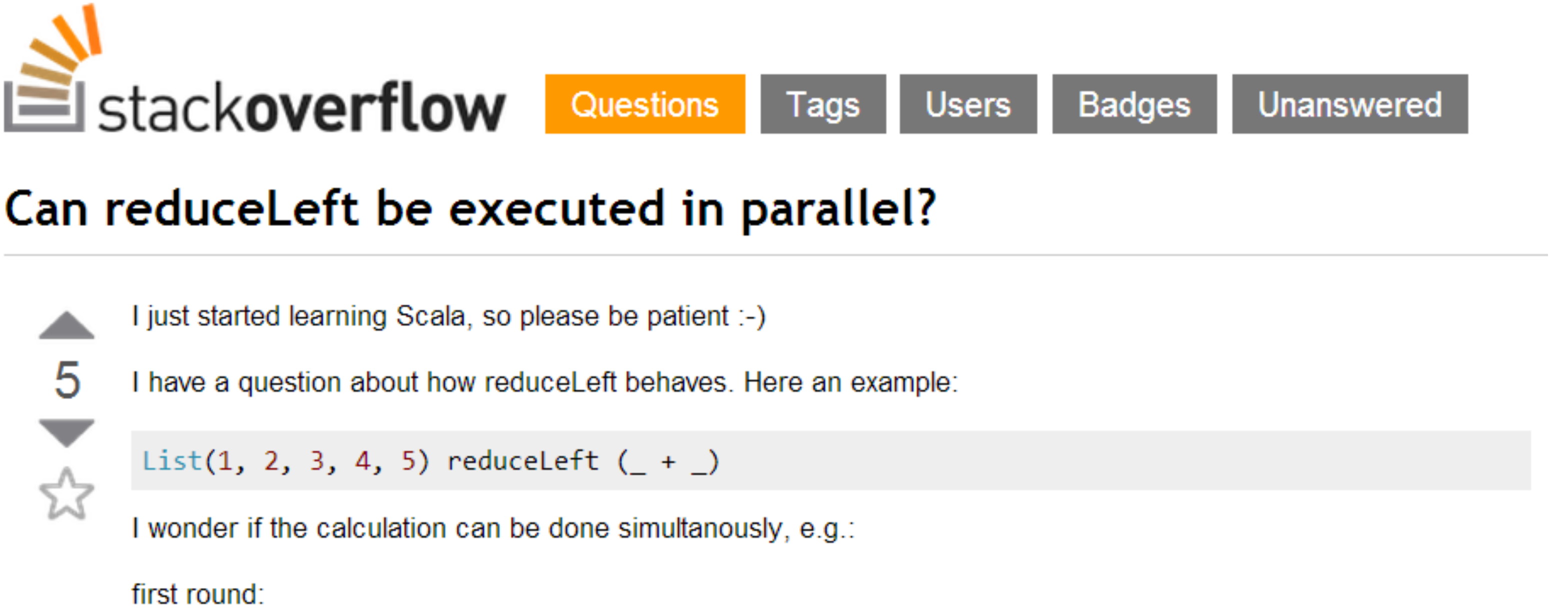
Contents

Converting between sequential and parallel collections

Every sequential collection can be converted to its parallel variant using the `par` method. Certain sequential collections have a direct parallel counterpart. For these collections the conversion is efficient– it occurs in constant time, since both the sequential and the parallel collection have the same data-structural representation (one exception is mutable hash maps and hash sets which are slightly more expensive to convert the first time `par` is called, but subsequent invocations of `par` take constant time). It should be noted that for mutable collections, changes in the sequential collection are visible in its parallel counterpart if they share the underlying data-structure.

Sequential	Parallel
mutable	
`Array`	`ParArray`
`HashMap`	`ParHashMap`
`HashSet`	`ParHashSet`

No amount of documentation is apparently enough



Can reduceLeft be executed in parallel?

I just started learning Scala, so please be patient :-)

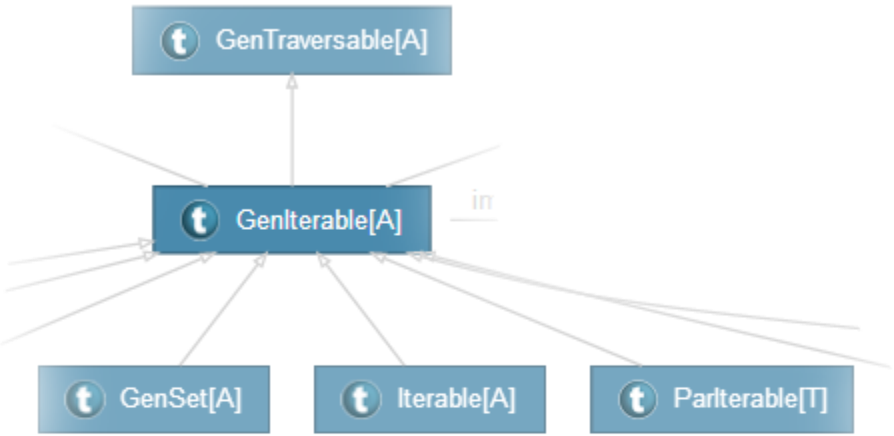
I have a question about how reduceLeft behaves. Here an example:

```
List(1, 2, 3, 4, 5) reduceLeft (_ + _)
```

I wonder if the calculation can be done simultanously, e.g.:

first round:

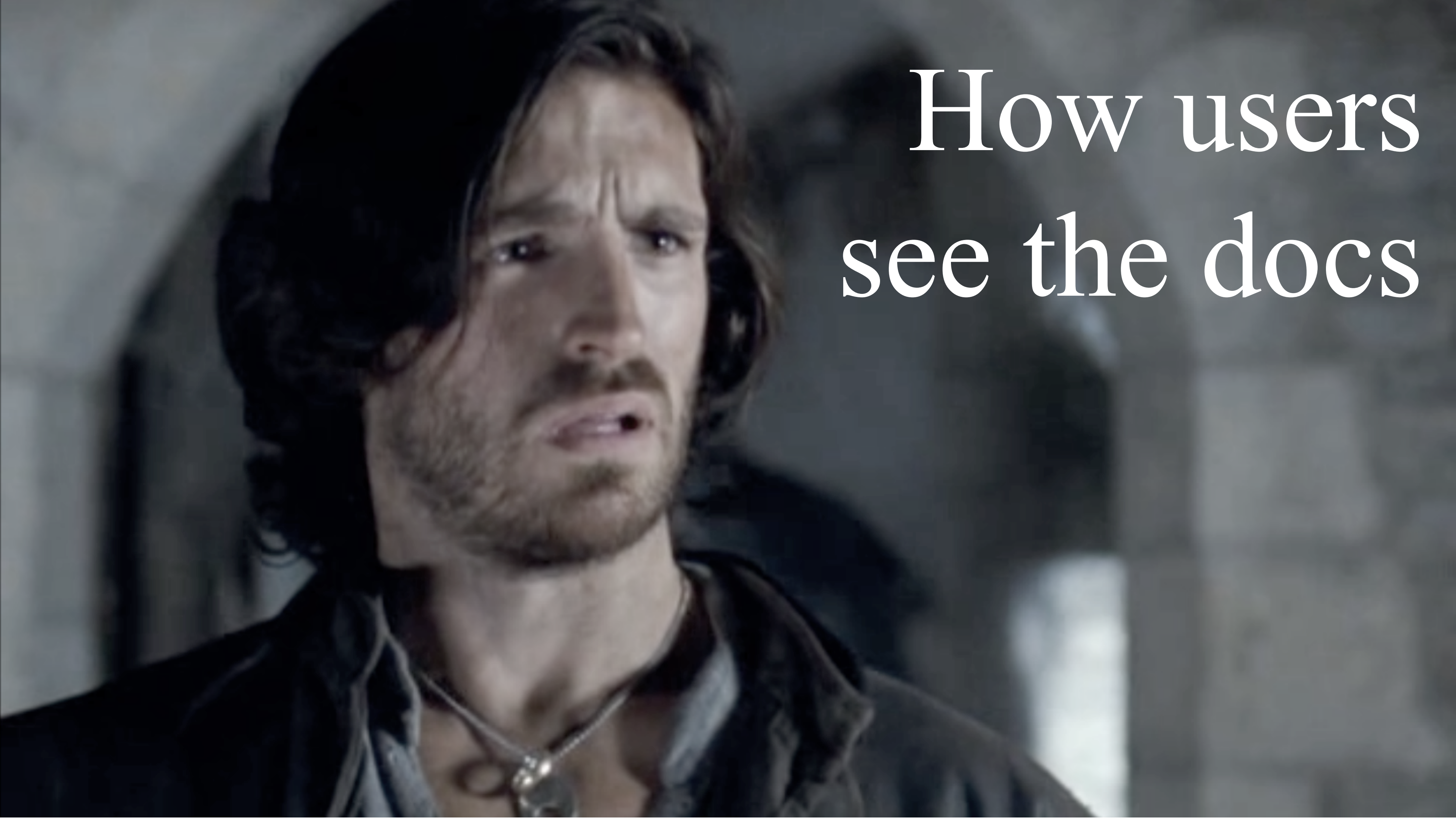
The `reduceLeft` guarantees operations are executed from left to right



Parallel and sequential collections sharing operations

There are several problems here

How we see users



How users see the docs

Bending the truth.

And sometimes we were just slow

So, we have a new API now

```
def findDoe(names: Array[String]): Option[String] =
{
  names.toPar.find(_.endsWith(“Doe”))
}
```

Wait, you renamed a method?

```
def findDoe(names: Array[String]): Option[String] =
{
  names.toPar.find(_.endsWith(“Doe”))
}
```

Yeah, `par` already exists. But, `toPar` is different.

```
def findDoe(names: Array[String]): Option[String] =
{
  names.toPar.find(_.endsWith(“Doe”))
}
```

```
def findDoe(names: Array[String]): Option[String] = {
  names.toPar.find(_.endsWith(“Doe”))
}

implicit class ParOps[Repr](val r: Repr) extends AnyVal {
  def toPar = new Par(r)
}
```

```
def findDoe(names: Array[String]): Option[String] = {
  ParOps(names).toPar.find(_.endsWith(“Doe”))
}

implicit class ParOps[Repr](val r: Repr) extends AnyVal {
  def toPar = new Par(r)
}
```

```
def findDoe(names: Array[String]): Option[String] = {
  ParOps(names).toPar.find(_.endsWith(“Doe”))
}

implicit class ParOps[Repr](val r: Repr) extends AnyVal {
  def toPar = new Par(r)
}

class Par[Repr](r: Repr)
```

```
def findDoe(names: Array[String]): Option[String] = {
  (new Par(names)).find(_.endsWith(“Doe”))
}

implicit class ParOps[Repr](val r: Repr) extends AnyVal {
  def toPar = new Par(r)
}

class Par[Repr](r: Repr)
```

```
def findDoe(names: Array[String]): Option[String] = {
  (new Par(names)).find(_.endsWith(“Doe”))
}

class Par[Repr](r: Repr)
```

But, `Par[Repr]` does not have the `find` method!

```
def findDoe(names: Array[String]): Option[String] = {
  (new Par(names)).find(_.endsWith(“Doe”))
}

class Par[Repr](r: Repr)
```

True, but `Par[Array[String]]` does have a `find` method.

```
def findDoe(names: Array[String]): Option[String] = {
  (new Par(names)).find(_.endsWith(“Doe”))
}

class Par[Repr](r: Repr)

implicit class ParArrayOps[T](pa: Par[Array[T]]) {
  ...
  def find(p: T => Boolean): Option[T]
  ...
}
```

More flexible!

More flexible!

- does not have to implement methods that make no sense in parallel

More flexible!

- does not have to implement methods that make no sense in parallel
- slow conversions explicit

No standard library collections were hurt doing this.

More flexible!

- does not have to implement methods that make no sense in parallel
- slow conversions explicit
- non-intrusive addition to standard library

More flexible!

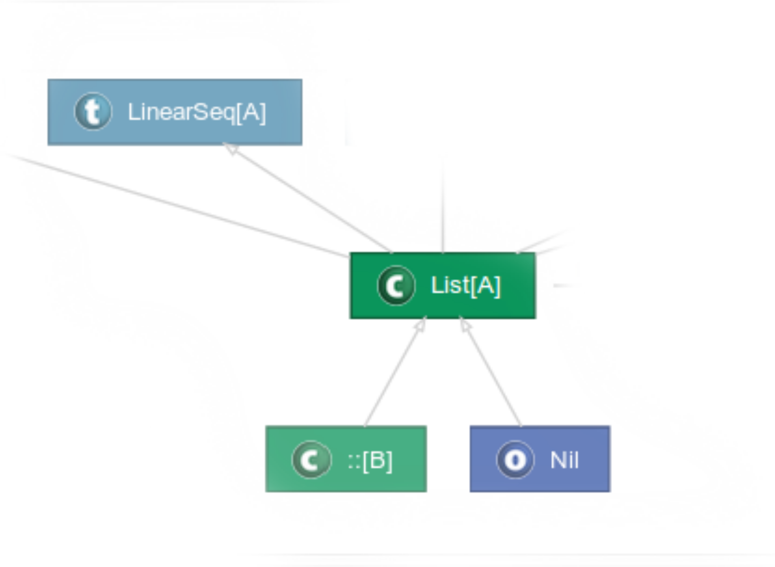
- does not have to implement methods that make no sense in parallel
- slow conversions explicit
- non-intrusive addition to standard library
- easy to add new methods and collections

More flexible!

- does not have to implement methods that make no sense in parallel
- slow conversions explicit
- non-intrusive addition to standard library
- easy to add new methods and collections
- `import` switches between implementations

```
def findDoe(names: Seq[String]): Option[String] = {
  names.toPar.find(_.endsWith(“Doe”))
}
```

```
def findDoe(names: Seq[String]): Option[String] = {
  names.toPar.find(_.endsWith(“Doe”))
}
```



```
def findDoe(names: Seq[String]): Option[String] = {
  names.toPar.find(_.endsWith(“Doe”))
}
```

But how do I write generic code?

```
def findDoe[Repr[_]](names: Par[Repr[String]]) = {
  names.toPar.find(_.endsWith(“Doe”))
}
```

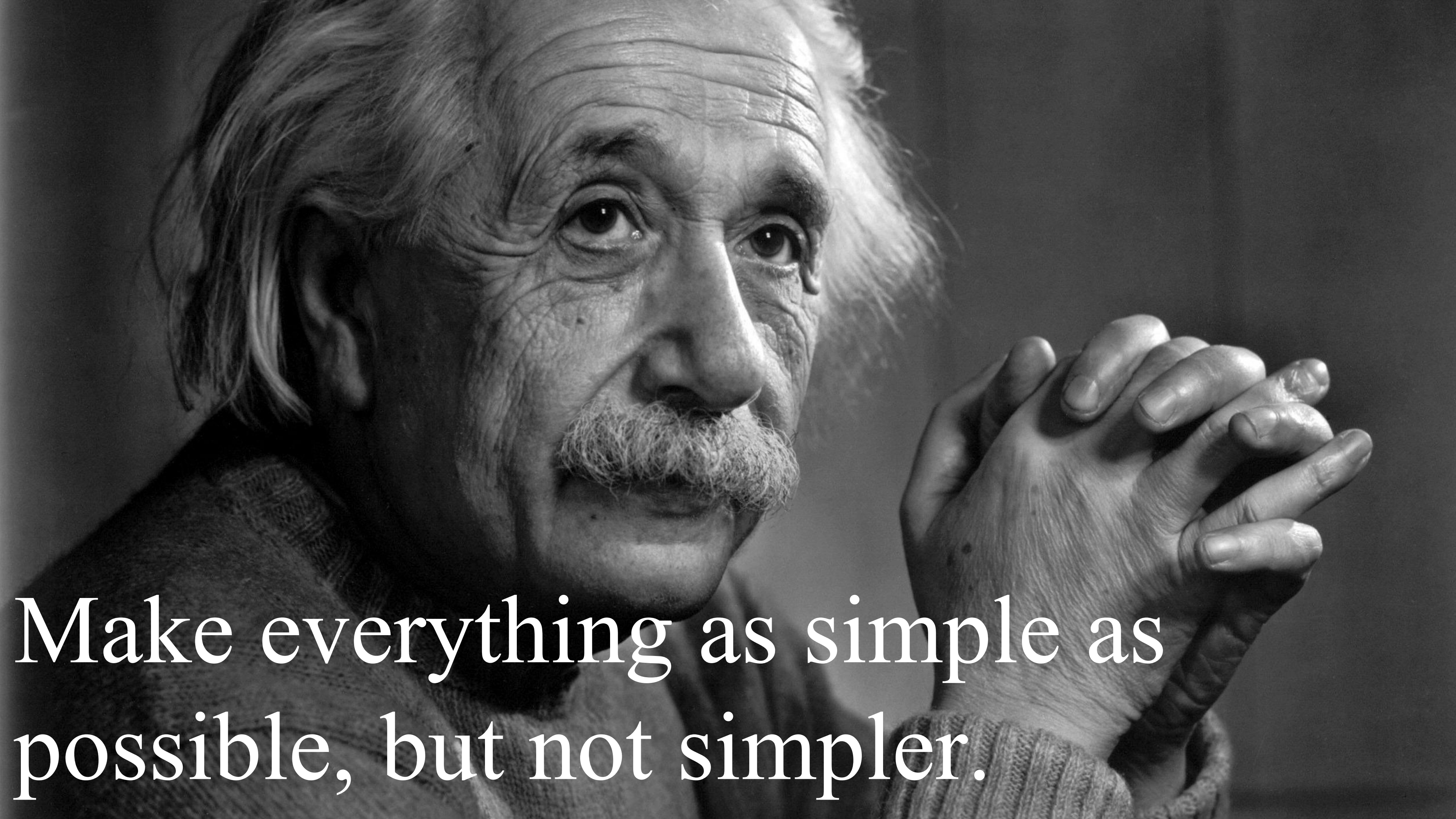
```
def findDoe[Repr[_]](names: Par[Repr[String]]) = {
  names.toPar.find(_.endsWith(“Doe”))
}
```

`Par[Repr[String]]` does not have a `find`

```
def findDoe[Repr[_]: Ops](names: Par[Repr[String]]) = {
  names.toPar.find(_.endsWith(“Doe”))
}
```

```
def findDoe[Repr[_]: Ops](names: Par[Repr[String]]) = {
  names.toPar.find(_.endsWith(“Doe”))
}
```

We don’t do this.



Make everything as simple as possible, but not simpler.

```
def findDoe(names: Reducable[String])= {
  names.find(_.endsWith(“Doe”))
}
```

```
def findDoe(names: Reducable[String])= {
  names.find(_.endsWith(“Doe”))
}
```

```
findDoe(Array(1, 2, 3).toPar)
```

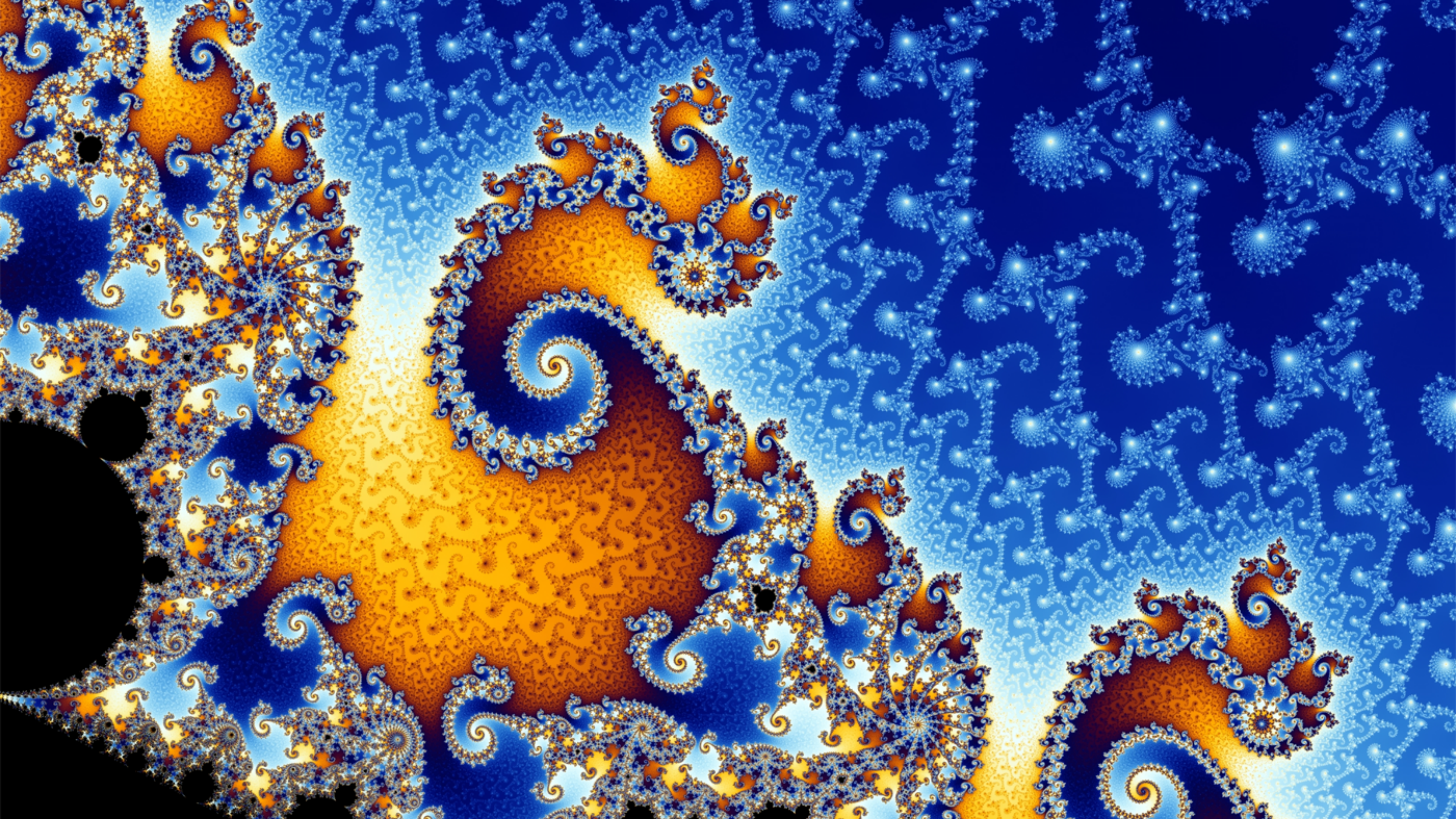
```
def findDoe(names: Reducable[String])= {
  names.find(_.endsWith(“Doe”))
}



findDoe(toReducable(Array(1, 2, 3).toPar))
```

```
def findDoe(names: Reducable[String])= {
  names.find(_.endsWith(“Doe”))
}
```

```
findDoe(toReducable(Array(1, 2, 3).toPar))
```

```
def arrayIsReducable[T]: IsReducable[T] = { … }
```

So let’s write a program!



```scala
import scala.collection.par._


val pixels = new Array[Int](wdt * hgt)
for (idx <- (0 until (wdt * hgt)).toPar) {



}
```

```
import scala.collection.par._


val pixels = new Array[Int](wdt * hgt)
for (idx <- (0 until (wdt * hgt)).toPar) {
  val x = idx % wdt
  val y = idx / wdt

}
```

```scala
import scala.collection.par._


val pixels = new Array[Int](wdt * hgt)
for (idx <- (0 until (wdt * hgt)).toPar) {
  val x = idx % wdt
  val y = idx / wdt
  pixels(idx) = computeColor(x, y)
}
```

```scala
import scala.collection.par._


val pixels = new Array[Int](wdt * hgt)
for (idx <- (0 until (wdt * hgt)).toPar) {
  val x = idx % wdt
  val y = idx / wdt
  pixels(idx) = computeColor(x, y)
}
```

`Scheduler` not found!

```scala
import scala.collection.par._
import Scheduler.Implicits.global

val pixels = new Array[Int](wdt * hgt)
for (idx <- (0 until (wdt * hgt)).toPar) {
  val x = idx % wdt
  val y = idx / wdt
  pixels(idx) = computeColor(x, y)
}
```

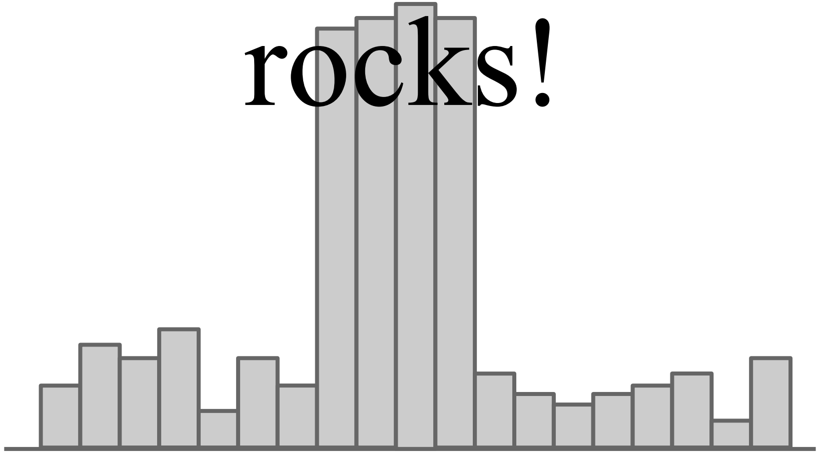
```scala
import scala.collection.par._
import Scheduler.Implicits.global

val pixels = new Array[Int](wdt * hgt)
for (idx <- (0 until (wdt * hgt)).toPar) {
  val x = idx % wdt
  val y = idx / wdt
  pixels(idx) = computeColor(x, y)
}
```

New parallel collections

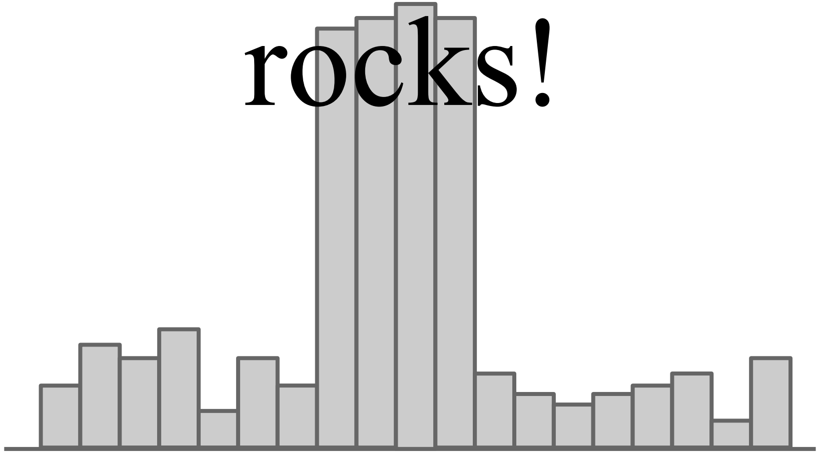
33% faster!

Now	Previously
103 ms	148 ms

Workstealing tree scheduler rocks!

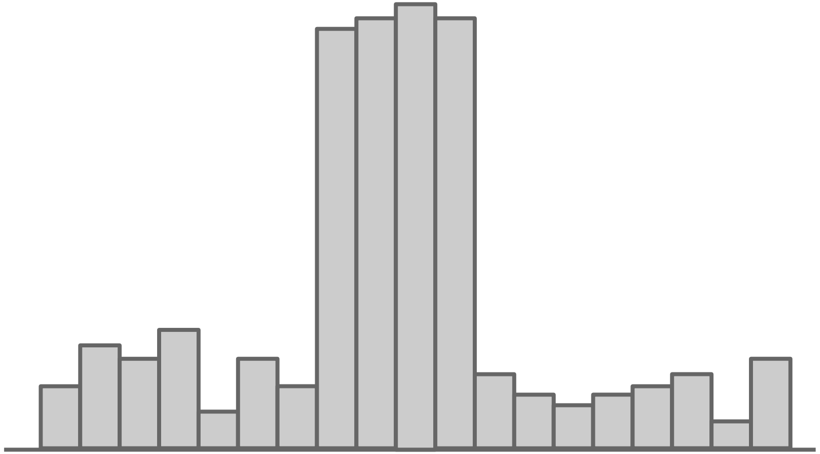


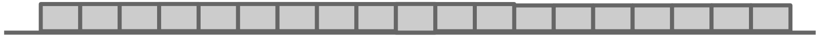
Workstealing tree scheduler rocks!



But, are there other interesting

Fine-grained uniform workloads are on the opposite side of the spectrum.





```
def mean(xs: Array[Float]): Float = {
  val sum = xs.toPar.fold(0)(_ + _)
  sum / xs.length
}
```

```
def mean(xs: Array[Float]): Float = {
  val sum = xs.toPar.fold(0)(_ + _)
  sum / xs.length
}
```

Now	Previously
15 ms	565 ms

But how?

```
def fold[T](a: Iterable[T])(z:T)(op: (T, T) => T) = {
  var it = a.iterator
  var acc = z
  while (it.hasNext) {
    acc = op(acc, it.next)
  }
  acc
}
```

```
def fold[T](a: Iterable[T])(z:T)(op: (T, T) => T) = {
  var it = a.iterator
  var acc = z
  while (it.hasNext) {
    acc = box(op(acc, it.next))
  }
  acc
}
```

```
def fold[T](a: Iterable[T])(z:T)(op: (T, T) => T) = {
  var it = a.iterator
  var acc = z
  while (it.hasNext) {
    acc = box(op(acc, it.next))
  }
  acc
}
```

Generic methods cause boxing of primitives

```
def mean(xs: Array[Float]): Float = {
  val sum = xs.toPar.fold(0)(_ + _)
  sum / xs.length
}
```

```
def mean(xs: Array[Float]): Float = {
  val sum = xs.toPar.fold(0)(_ + _)
  sum / xs.length
}
```

Generic methods hurt performance
What can we do instead?

```
def mean(xs: Array[Float]): Float = {
  val sum = xs.toPar.fold(0)(_ + _)
  sum / xs.length
}
```

Generic methods hurt performance
What can we do instead?

Inline method body!

```scala
def mean(xs: Array[Float]): Float = {
  val sum = {
    var it = xs.iterator
    var acc = 0
    while (it.hasNext) {
      acc = acc + it.next
    }
    acc
   }
  sum / xs.length
}
```

```
def mean(xs: Array[Float]): Float = {
  val sum = {
    var it = xs.iterator
    var acc = 0
    while (it.hasNext) {
      acc = acc + it.next
    }
    acc
   }
  sum / xs.length
}
```

Specific type

No boxing!

No memory allocation!

```
def mean(xs: Array[Float]): Float = {
  val sum = {
    var it = xs.iterator
    var acc = 0
    while (it.hasNext) {
      acc = acc + it.next
    }
    acc
   }
  sum / xs.length
}
```

Specific type
No boxing!
No memory allocation!

565 ms → 281 ms

2X speedup

```
def mean(xs: Array[Float]): Float = {
  val sum = {
    var it = xs.iterator
    var acc = 0
    while (it.hasNext) {
      acc = acc + it.next
    }
    acc
  }
  sum / xs.length
}
```

```
def mean(xs: Array[Float]): Float = {
  val sum = {
    var it = xs.iterator
    var acc = 0
    while (it.hasNext) {
      acc = acc + it.next
    }
    acc
   }
  sum / xs.length
}
```

Iterators? For Array?
We don’t need them!

```
def mean(xs: Array[Float]): Float = {
  val sum = {
    var i = 0
    val until = xs.size
    var acc = 0
    while (i < until) {
      acc = acc + a(i)
      i = i + 1
    }
    acc
   }
  sum / xs.length
}
```

Use index-based access!

```
def mean(xs: Array[Float]): Float = {
  val sum = {
    var i = 0
    val until = xs.size
    var acc = 0
    while (i < until) {
      acc = acc + a(i)
      i = i + 1
    }
    acc
  }
  sum / xs.length
}
```

Use index-based access!

281 ms → 15 ms

19x speedup

Are those optimizations parallel-collections specific?

Are those optimizations parallel-collections specific?

No

Are those optimizations parallel-collections specific?

No

You can use them on sequential collections

```scala
def mean(xs: Array[Float]): Float = {
  val sum = xs.fold(0)(_ + _)
  sum / xs.length
}
```

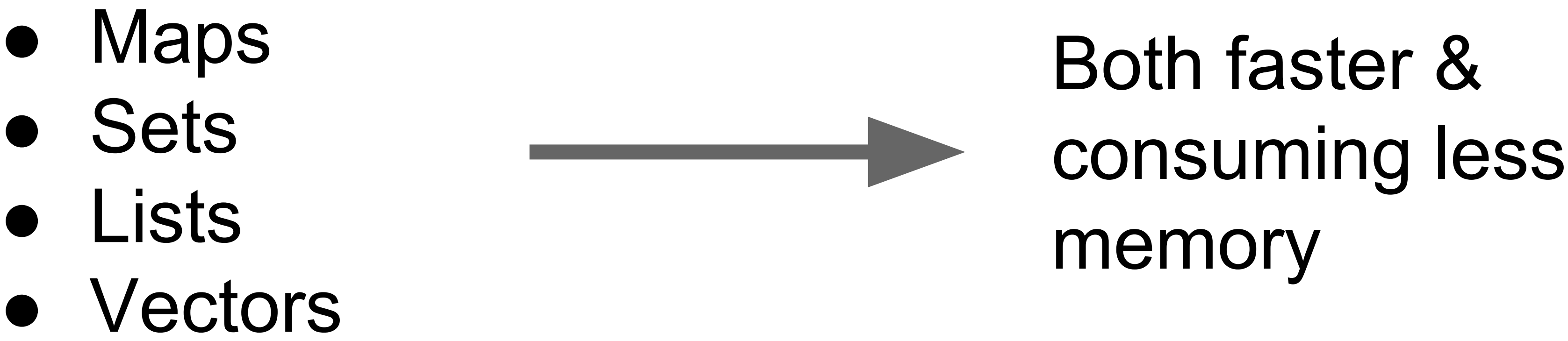
```scala
import scala.collections.optimizer._
def mean(xs: Array[Float]): Float = optimize{
  val sum = xs.fold(0)(_ + _)
  sum / xs.length
}
```

```
import scala.collections.optimizer._
def mean(xs: Array[Float]): Float = optimize{
  val sum = xs.fold(0)(_ + _)
  sum / xs.length
}
```

You get 38 times speedup!

Future work

@specialized collections



- Maps
- Sets
- Lists
- Vectors

Both faster & consuming less memory

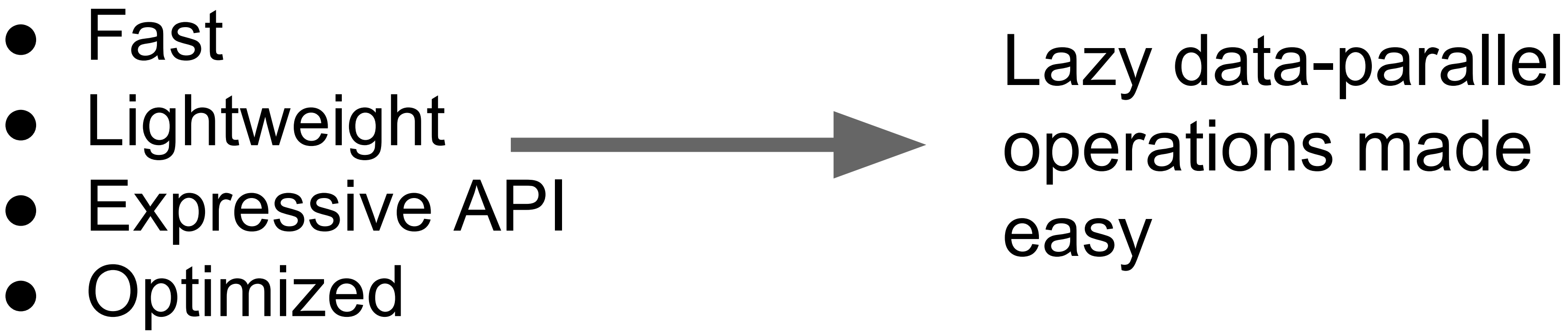
@specialized collections

- Maps
- Sets
- Lists
- Vectors

Both faster & consuming less memory

Expect to get this for free inside optimize{} block

jdk8-style streams(parallel views)



- Fast
- Lightweight
- Expressive API
- Optimized

Lazy data-parallel operations made easy

Future’s based asynchronous API

```
val sum = future{ xs.sum }
val normalized = sum.andThen(sum => sum/xs.size)
```

Boilerplate code, ugly

Future’s based asynchronous API

```
val sum = xs.toFuture.sum
val scaled = xs.map(_ / sum)
```

- Simple to use
- Lightweight
- Expressive API
- Optimized

Asynchronous dat
parallel operations
made easy

Current research: operation fusion

```
val minMaleAge = people.filter(_.isMale)
                    .map(_.age).min
val minFemaleAge = people.filter(_.isFemale)
                    .map(_.age).min
```

Current research: operation fusion

```
val minMaleAge = people.filter(_.isMale)
                    .map(_.age).min
val minFemaleAge = people.filter(_.isFemale)
                    .map(_.age).min
```

- Requires up to 3 times more memory than original collection
- Requires 6 traversals of collections

Current research: operation fusion

```
val minMaleAge   = people.filter(_.isMale)
                          .map(_.age).min
val minFemaleAge = people.filter(_.isFemale)
                          .map(_.age).min
```

- Requires up to 3 times more memory than original collection
- Requires 6 traversals of collections

We aim to reduce this to single traversal with no additional memory.
Without you changing your code