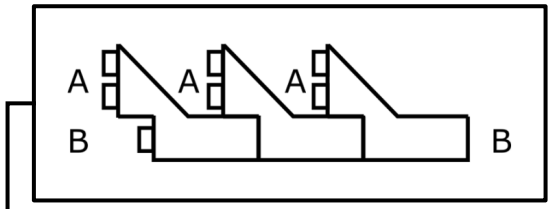
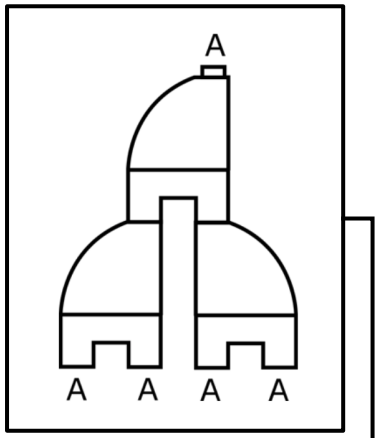
Scala Left Fold Parallelisation

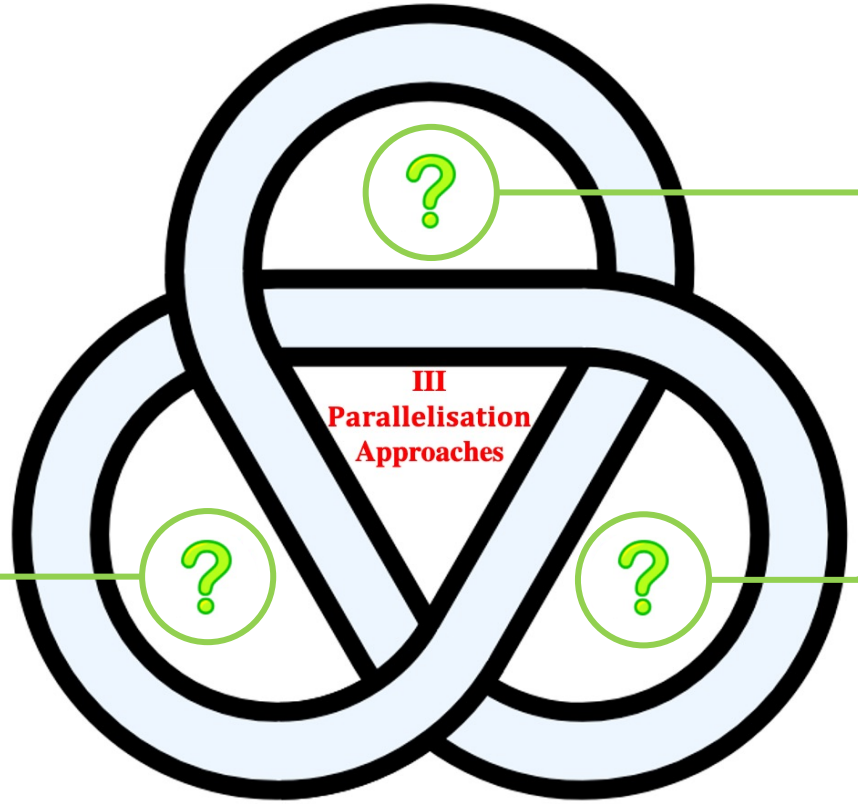
Three Approaches

`foldLeft`



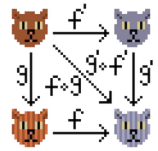
`fold`





Standard Library

Parallel Collections Library



Cats

+

Cats Effect

Adam Rosien

@arosien

slides by

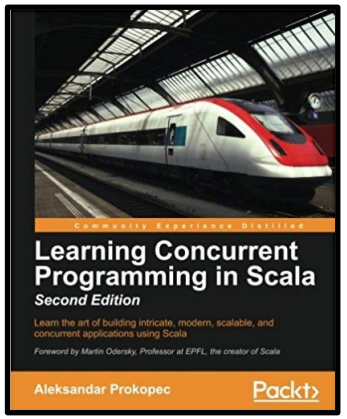
@philip_schwarz

FP Illuminated

http://fpilluminated.com/

Aleksandar Prokopec

@alexprokopec



@philip_schwarz

Let's begin by looking at a contrived example of a **left fold** over a relatively large collection.

It is an adaptation of an example from the following book by **Aleksandar Prokopec**: **Learning Concurrent Programming in Scala**.

The original example downloaded a text file containing the whole **HTML specification**, searched its lines for the keyword 'TEXTAREA', and then printed the lines containing the keyword.

We are going to search for a word supplied by the user, and the text that we are going to search is going to be that of a relatively large book downloaded from https://gutenberg.org/.

Initially I picked **War and Peace**, which is **66,036** lines long, but for reasons that will become clear later, I then decided to look for a book of about **100,000** lines, and the closest that I could find was **The King James Version of the Bible**, which is only **25** lines short of the desired number.

```
case class Book(name: String, numberOfLines: Int, numberOfBytes: Int, url: URL)

val theBible = Book(
  name = "The King James Version of the Bible",
  numberOfLines = 99_975,
  numberOfBytes = 4_456_041,
  url = URL("https://gutenberg.org/cache/epub/10/pg10.txt")
)
```

Here is a method that tries to get hold of the **lines of text** of a book…

```
def getText(book: Book): Try[Vector[String]] =
   Using(Source.fromURL(book.url)): source =>
     source.getLines.toVector
```

…and here is the first part of a program which, given a **search word**, uses the above method to find occurrences of the word in the **lines** of our chosen book.

```
@main def run(word: String): Unit =
  getText(book = theBible)
    .fold(
      error => handleErrorGettingText(error),
      lines =>
        announceSuccessGettingText(lines)
        val matches = find(word, lines)
        announceMatchingLines(matches))
```

If getting the **text lines** fails then we handle that, otherwise we announce that getting the text was successful, invoke a function to **find** occurrences of the **search word**, and then announce the results.

By the way, to simplify exposition, the error handling that you see in the run method is the only one in the whole slide deck. This is obviously not production-grade code!

As you can see below, the way that we search the book's **text lines** for the **search word** is by doing a **left fold** of function **accumulateLinesContaining** over the **lines**, so that the **fold** returns a single string with all the **lines** containing the **search word**, separated by newlines.

```
def find(word: String, lines: Vector[String]): String =
  lines.foldLeft("")(accumulateLinesContaining(word))
```

```
def accumulateLinesContaining(word: String): (String, String) => String =
  (acc, line) => if line.matches(s".*$word.*") then s"$acc\n'$line'" else acc
```

Before we run the program, here are its remaining methods

```
def handleErrorGettingText[A](error: Throwable): A =
  throw IllegalStateException(s"Failed to obtain the text lines to be searched.", error)

def announceSuccessGettingText(lines: Vector[String]): Unit =
  println(f"Successfully obtained ${lines.length}%,d lines of text to search.")

def announceMatchingLines(lines: String): Unit =
  println(f"Found the word in the following ${lines.count(_ == '\n')}%,d lines of text: $lines")
```

Let's search for the word '**joyous**'

```
$ sbt "run joyous"
…
[info] running run joyous
Successfully obtained 99,975 lines of text to search.
Found the word in the following 4 lines of text:
'stirs, a tumultuous city, joyous city: thy slain men are not slain'
'23:7 Is this your joyous city, whose antiquity is of ancient days? her'
'upon all the houses of joy in the joyous city: 32:14 Because the'
'12:11 Now no chastening for the present seemeth to be joyous, but'
[success] Total time: 4 s, completed 5 Nov 2023, 07:54:12
```

That took four seconds, with much of the time taken up by downloading the book (between 1.5s and 2.5s).

Let's see **how long** the program takes to **execute** if we **increase** the **number of lines** to be **searched**.

Once the **getText** function has downloaded the book and obtained its **lines of text**, it now makes as many **copies** of the **lines** as required.

`def getText(book: Book): Try[Vector[String]] =`	`<>`	`def getText(book: Book, copies: Int = 1): Try[Vector[String]] =`
`  Using(Source.fromURL(book.url)): source =>`	`=`	`  Using(Source.fromURL(book.url)): source =>`
`    source.getLines.toVector`	`<>`	`    val lines = source.getLines.toVector` `    Vector.fill(copies)(lines).flatten`

Let's make a **thousand copies** of the **lines**.

`  getText(book = theBible)`	`<>`	`  getText(book = theBible, copies = 1_000)`

`    .fold(`
`      error => handleErrorGettingText(error),`
`      lines =>`
`        announceSuccessGettingText(lines)`
`        val matches = find(word, lines)`
`        announceMatchingLines(matches)`
`    )`
	`=`	`    .fold(`
		`      error => handleErrorGettingText(error),`
		`      lines =>`
		`        announceSuccessGettingText(lines)`
		`        val matches = find(word, lines)`
		`        announceMatchingLines(matches)`
		`    )`

Since the book is about **100,000 lines** long, we'll now be searching about **1,000** x **100,000 lines**, i.e about **100 million lines**.

While it makes little sense to search **multiple copies** of the book, we are doing this purely to set the scene for the subject of this slide deck.

@philip_schwarz

Searching through a **thousand copies** of the book takes a little bit over **one minute**.

When we searched **one copy** of the book, we found **four matching lines**, so it makes sense that now that we are searching a **thousand copies**, we are finding **4,000 matching lines**.

```
$ sbt "run joyous"
…
[info] running run joyous
Successfully obtained 99,975,000 lines of text to search.
Found the word in the following 4,000 lines of text:
'stirs, a tumultuous city, joyous city: thy slain men are not slain'
'23:7 Is this your joyous city, whose antiquity is of ancient days? her'
'upon all the houses of joy in the joyous city: 32:14 Because the'
'12:11 Now no chastening for the present seemeth to be joyous, but'
'stirs, a tumultuous city, joyous city: thy slain men are not slain'
'23:7 Is this your joyous city, whose antiquity is of ancient days? her'
'upon all the houses of joy in the joyous city: 32:14 Because the'
'12:11 Now no chastening for the present seemeth to be joyous, but'
'stirs, a tumultuous city, joyous city: thy slain men are not slain'
'23:7 Is this your joyous city, whose antiquity is of ancient days? her'
'upon all the houses of joy in the joyous city: 32:14 Because the'
'12:11 Now no chastening for the present seemeth to be joyous, but'
…
'stirs, a tumultuous city, joyous city: thy slain men are not slain'
'23:7 Is this your joyous city, whose antiquity is of ancient days? her'
'upon all the houses of joy in the joyous city: 32:14 Because the'
'12:11 Now no chastening for the present seemeth to be joyous, but'
'stirs, a tumultuous city, joyous city: thy slain men are not slain'
'23:7 Is this your joyous city, whose antiquity is of ancient days? her'
'upon all the houses of joy in the joyous city: 32:14 Because the'
'12:11 Now no chastening for the present seemeth to be joyous, but'
[success] Total time: 66 s (01:06), completed 5 Nov 2023, 11:11:14
```

I ran the program **four times**, and its execution times were **66s**, **65s**, **65s** and **66s**.

By the way, when I first tried to run the program, I got some warnings suggesting that I increase the **heap space**, so I added the following to file **.sbtopt**s: **-J-Xmx5G**

In this deck we are going to look at three ways of **parallelising** the program's **search** for **matching lines**, which is carried out by the following **left fold**

```
def find(word: String, lines: Vector[String]): String =
  lines.foldLeft("")(accumulateLinesContaining(word))
```

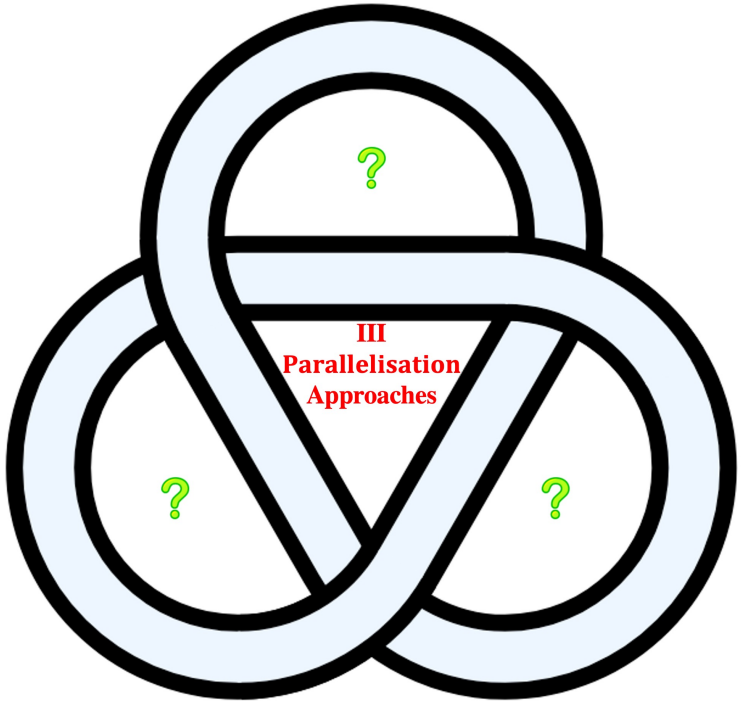
```
def accumulateLinesContaining(word: String): (String, String) => String =
  (acc, line) => if line.matches(s".*$word.*") then s"$acc\n'$line'" else acc
```

The **fold** is working its way **sequentially** through **100 million lines** of text.

Instead of processing all of the lines **sequentially**, can we get the program to **partition** the lines into a number of **batches**, search the batches in **parallel**, and then **combine** the results of all the searches?

Trick question: will the **foldLeft** function automatically do that for us, behind the scenes, if instead of invoking the function on a **sequential** collection, we first convert it to a **parallel** collection?

I ask because, as you can see on the next slide, there is a **Scala parallel collections library** that can be used to convert a **sequential** collection to a **parallel** one.



https://docs.scala-lang.org/overviews/parallel-collections/overview.html

Scala

docs

PARALLEL COLLECTIONS
OVERVIEW

Aleksandar Prokopec, Heather Miller

If you're using Scala 2.13+ and want to use Scala's parallel collections, you'll have to import a separate module, as described here.

Motivation

Amidst the shift in recent years by processor manufacturers from single to multicore architectures, academia and industry alike have conceded that *Popular Parallel Programming* remains a formidable challenge.

Parallel collections were included in the Scala standard library in an effort to facilitate parallel programming by sparing users from low-level parallelization details, meanwhile providing them with a familiar and simple high-level abstraction. The hope was, and still is, that implicit parallelism behind a collections abstraction will bring reliable parallel execution one step closer to the workflow of mainstream developers.

The idea is simple– collections are a well-understood and frequently-used programming abstraction. And given their regularity, they're able to be efficiently parallelized, transparently. By allowing a user to "swap out" sequential collections for ones that are operated on in parallel, Scala's parallel collections take a large step forward in enabling parallelism to be easily brought into more code.

Take the following, sequential example, where we perform a monadic operation on some large collection:

```
val list = (1 to 10000).toList
list.map(_ + 42)
```

To perform the same operation in parallel, one must simply invoke the `par` method on the sequential collection, `list`. After that, one can use a parallel collection in the same way one would normally use a sequential collection. The above example can be parallelized by simply doing the following:

```
list.par.map(_ + 42)
```

https://github.com/scala/scala-parallel-collections

scala / **scala-parallel-collections**

scala-parallel-collections

Scala parallel collections

This Scala standard module contains the package `scala.collection.parallel`, with all of the parallel collections that used to be part of the Scala standard library (in Scala 2.10 through 2.12).

For Scala 3 and Scala 2.13, this module is a separate JAR that can be omitted from projects that do not use parallel collections.

Usage

To depend on scala-parallel-collections in sbt, add this to your `build.sbt`:

```
libraryDependencies +=
  "org.scala-lang.modules" %% "scala-parallel-collections" % "<version>"
```

In your code, adding this import:

```
import scala.collection.parallel.CollectionConverters._
```

will enable use of the `.par` method as in earlier Scala versions.

Let's add the **Scala parallel collections library** to the build…

```
libraryDependencies += "org.scala-lang.modules" %% "scala-parallel-collections" % "1.0.4"
```

…and get the **find** function to convert the **sequential** collection of text lines to a **parallel** one…

	<>	`import scala.collection.parallel.CollectionConverters.*`
`def find(word: String, lines: Vector[String]): String =`	=	`def find(word: String, lines: Vector[String]): String =`
`  lines.foldLeft("")(accumulateLinesContaining(word))`	<>	`  lines.par.foldLeft("")(accumulateLinesContaining(word))`

Now let’s run the program again and see if converting the **sequential collection** of lines into a **parallel collection** has any effect on the program’s **execution time**.

```
$ sbt "run joyous"
…
[info] running run joyous
Successfully obtained 99,975,000 lines of text to search.
Found the word in the following 4,000 lines of text:
'stirs, a tumultuous city, joyous city: thy slain men are not slain'
'23:7 Is this your joyous city, whose antiquity is of ancient days? her'
'upon all the houses of joy in the joyous city: 32:14 Because the'
'12:11 Now no chastening for the present seemeth to be joyous, but'
'stirs, a tumultuous city, joyous city: thy slain men are not slain'
'23:7 Is this your joyous city, whose antiquity is of ancient days? her'
'upon all the houses of joy in the joyous city: 32:14 Because the'
'12:11 Now no chastening for the present seemeth to be joyous, but'
'stirs, a tumultuous city, joyous city: thy slain men are not slain'
'23:7 Is this your joyous city, whose antiquity is of ancient days? her'
'upon all the houses of joy in the joyous city: 32:14 Because the'
'12:11 Now no chastening for the present seemeth to be joyous, but’
…
'stirs, a tumultuous city, joyous city: thy slain men are not slain'
'23:7 Is this your joyous city, whose antiquity is of ancient days? her'
'upon all the houses of joy in the joyous city: 32:14 Because the'
'12:11 Now no chastening for the present seemeth to be joyous, but’
'stirs, a tumultuous city, joyous city: thy slain men are not slain'
'23:7 Is this your joyous city, whose antiquity is of ancient days? her'
'upon all the houses of joy in the joyous city: 32:14 Because the'
'12:11 Now no chastening for the present seemeth to be joyous, but'
[success] Total time: 68 s (01:08), completed 5 Nov 2023, 16:44:21
```

No difference: the **execution time** is pretty much the same as before.

@philip_schwarz

Earlier, when I asked the following, I did mention that is was a **trick question**:

> will the **foldLeft** function automatically do that for us, behind the scenes, if instead of invoking the function on a **sequential** collection, we first convert it to a **parallel** collection?

To see why it is a **trick question**, consider the signature of the **foldLeft** function:

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

The **fodLeft** function cannot avoid processing a collection's elements **sequentially**: even if it did break the collection of **A**s down into multiple **smaller collections** of **A**s, and then **sequentially folded** (using the **op** function) each of those **smaller collections** at the same time, in **parallel**, it would not know what to do with the resulting **B**s, because it doesn't have a function that it can use to **combine** two **B** results, and so it is unable to **combine** all the **B** results into a single overall **B** result.

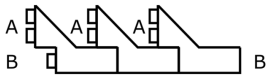
As we can see in the next slide, in **EPFL**'s **Scala Parallel Programming** course, **Aleksandar Prokopec** uses some really effective **Lego diagrams** to help visualise the situation.

EPFL: parprog1
Parallel programming

Non-Parallelizable Operations

Let's examine the `foldLeft` signature:

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



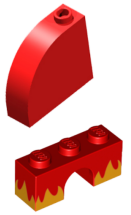
Operations `foldRight`, `reduceLeft`, `reduceRight`, `scanLeft` and `scanRight` similarly must process elements sequentially.

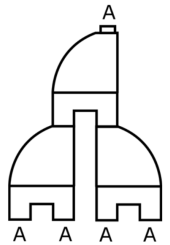
Aleksandar Prokopec
@alexprokopec

The `fold` Operation

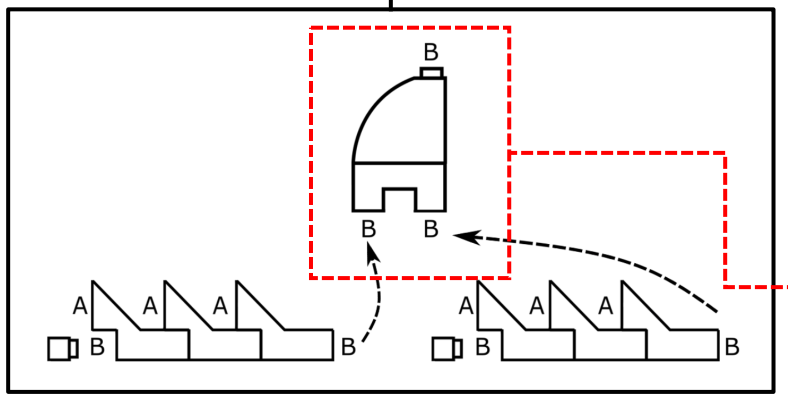
Next, let's examine the `fold` signature:

```
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.



Even if **foldLeft** could break a collection of **A**s down into multiple **smaller collections**, and **fold** each of those collections into a **B**, **in parallel**, it doesn't have a function for **combining** the resulting **B**s into a single overall **B**.

The **Scala parallel collections** library *does* have a solution for this problem though, and we'll come back to it later.

Since converting the **sequential vector** of lines to a **parallel collection** doesn't have any effect, let's **revert** our last change, and **rename** the **main method** to **runWithoutParallelism**.

```
import scala.collection.parallel.CollectionConverters.*     +-
  def find(word: String, lines: Vector[String]): String =    =
    lines.par.foldLeft("")(accumulateLinesContaining(word))  <>
```

```
  def find(word: String, lines: Vector[String]): String =
    lines.foldLeft("")(accumulateLinesContaining(word))
```

```
@main def run(word: String): Unit =                          <>
  getText(book = theBible, copies = 1_000)                   =
    .fold(
      error => handleErrorGettingText(error),
      lines =>
        announceSuccessGettingText(lines)
        val matches = find(word, lines)
        announceMatchingLines(matches)
    )
```

```
@main def runWithoutParallelism(word: String): Unit =
  getText(book = theBible, copies = 1_000)
    .fold(
      error => handleErrorGettingText(error),
      lines =>
        announceSuccessGettingText(lines)
        val matches = find(word, lines)
        announceMatchingLines(matches)
    )
```

As a recap, before moving on, the next slide shows the whole code for the current program.

@philip_schwarz

```scala
@main def runWithoutParallelism(word: String): Unit =
  getText(book = theBible, copies = 1_000)
    .fold(
      error => handleErrorGettingText(error),
      lines =>
        announceSuccessGettingText(lines)
        val matches = find(word, lines)
        announceMatchingLines(matches))
```

```scala
def find(word: String, lines: Vector[String]): String =
  lines.foldLeft("")(accumulateLinesContaining(word))
```

```scala
def accumulateLinesContaining(word: String): (String, String) => String =
  (acc, line) => if line.matches(s".*$word.*") then s"$acc\n'$line'" else acc
```

```scala
def getText(book: Book, copies: Int = 1): Try[Vector[String]] =
  Using(Source.fromURL(book.url)): source =>
    val lines = source.getLines.toVector
    Vector.fill(copies)(lines).flatten
```

```scala
def handleErrorGettingText[A](error: Throwable): A =
  throw IllegalStateException(s"Failed to obtain the text lines to be searched.", error)

def announceSuccessGettingText(lines: Vector[String]): Unit =
  println(f"Successfully obtained ${lines.length}%,d lines of text to search.")

def announceMatchingLines(lines: String): Unit
  println(f"Found the word in the following ${lines.count(_ == '\n')}%,d lines of text: $lines")
```

```scala
case class Book(
  name: String,
  numberOfLines: Int,
  numberOfBytes: Int,
  url: URL
)
```

```scala
import java.net.URL
import scala.io.Source
import scala.util.{Try, Using}
```

```scala
val theBible = Book(
  name = "The King James Version of the Bible",
  numberOfLines = 99_975,
  numberOfBytes = 4_456_041,
  url = URL("https://gutenberg.org/cache/epub/10/pg10.txt")
)
```

If we want to **parallelise** the **left fold**, but all we can use is **Scala**'s **standard library**, how can we do it?

One way is to use the **Future monad** and its `traverse` function.

Let's write a new **main method** called **runUsingFutureTraverse**. While its body is identical to that of **runWithoutParallelism**…

```
@main def runWithoutParallelism(word: String): Unit =
  getText(book = theBible, copies = 1_000)
    .fold(
      error => handleErrorGettingText(error),
      lines =>
        announceSuccessGettingText(lines)
        val matches = find(word, lines)
        announceMatchingLines(matches)
    )
```

<>

```
@main def runUsingFutureTraverse(word: String): Unit =
  getText(book = theBible, copies = 1_000)
    .fold(
      error => handleErrorGettingText(error),
      lines =>
        announceSuccessGettingText(lines)
        val matches = find(word, lines)
        announceMatchingLines(matches)
    )
```

…the **find** function that it invokes cannot be the one invoked by **runWithoutParallelism** …

```
def find(word: String, lines: Vector[String]): String =
  lines.foldLeft("")(accumulateLinesContaining(word))
```

…it needs to be rewritten, which we do on the next slide.

Here are some imports that we are going to need

```
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.duration.Duration
import scala.concurrent.{Await, Future}
```

If you could do with an introduction to the traverse function, one option you have is taking a look at part 1 of the collection shown below.

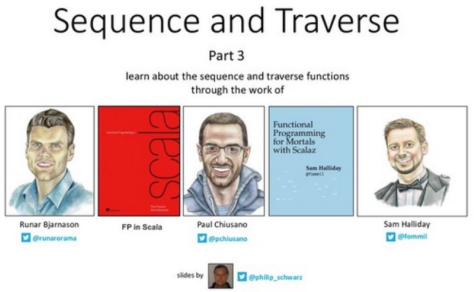
If you would like an introduction to the function in the context of Scala's Future type, then part 2 will help with that.

Philip Schwarz

PRO philipschwarz

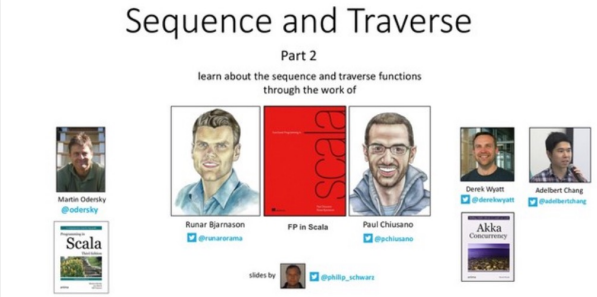
Speaker Deck

Sequence and Traverse



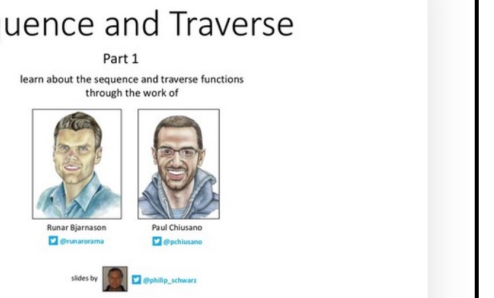
Sequence and Traverse - Part 3

philipschwarz PRO ☆ 1 👁 69



Sequence and Traverse - Part 2

philipschwarz PRO ☆ 0 👁 37



Sequence and Traverse - Part 1

philipschwarz PRO ☆ 1 👁 39

https://speakerdeck.com/philipschwarz/collections/sequence-and-traverse

```
def find(word: String, lines: Vector[String]): String =
  val batchSize = lines.size / (numberOfCores / 2)
  val groupsOfLines = lines.grouped(batchSize).toVector
  Await
    .result(
      Future.traverse(groupsOfLines)(searchFor(word))
            .map(_.foldLeft("")(_++_)),
      Duration.Inf
    )
```

We use the **grouped** function to break the collection of **text lines** into **multiple smaller collections**, one for each of the number of **CPU cores** that we want to use for **parallelising** the search.

We have decided to use half of the available **cores** for this purpose.

We then **traverse** the **smaller collections** of lines with a **searchFor** function that is used to **fold** each collection, and which is essentially the **find** function that we have been using up to now (on the right), except that it does the **folding** in a **Future**.

Compare the new **find** function (above on the left), with the one which we have been using up to now (below).

```
def searchFor(word: String)(lines: Vector[String]): Future[String] =
  Future(lines.foldLeft("")(accumulateLinesContaining(word)))
```

```
def find(word: String, lines: Vector[String]): String =
  lines.foldLeft("")(accumulateLinesContaining(word))
```

```
def searchFor(word: String)(lines: Vector[String]): Future[String] =
  Future(lines.foldLeft("")(accumulateLinesContaining(word)))
```

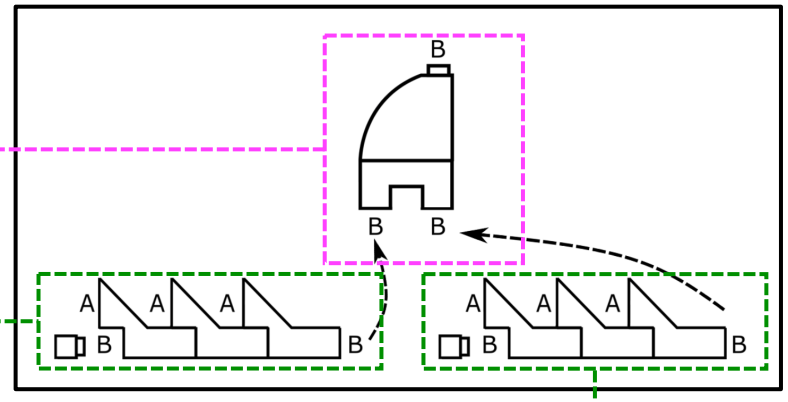
<>

```
def find(word: String, lines: Vector[String]): String =
  lines.foldLeft("")(accumulateLinesContaining(word))
```

traverse first creates a collection of **futures**, each of which does a **left fold** of the **A**s in a **smaller collection**, and then turns that collection **inside out**, i.e. it turns the collection of **future B**s into a **future** collection of **B**s.

The **futures** execute in **parallel**, each one on a **separate core**, and when they **complete**, each of them yields the result of the **left fold** of a **smaller collection**.

When the **future** collection returned by **traverse completes**, the **find** function has a collection of **B**s, which it then **folds** into a single overall **B**.




```
def accumulateLinesContaining(word: String): (String, String) => String =
  (acc, line) => if line.matches(s".*$word.*") then s"$acc\n'$line'" else acc
```

Now that the program **partitions** the collection of **text lines** into **multiple smaller collections**, and **folds** each of those **smaller collections** on a **separate core**, let's get the program to **print** on the **console** the **name** of the **thread** that does the **folding** on each such **core**.

To do that, let's first **extend Future** with the following method that turns a **Future** into a **Future** which, the last thing it does as part of its execution, is print the **thread name** on which it is being executed

```
extension [A](fa: Future[A])
  def printThreadName(): Future[A] =
    for
      a <- fa
      _ = println(s"[${Thread.currentThread().getName}]")
    yield a
```

Now all we have to do is invoke the new method.

`def searchFor(word: String)(lines: Vector[String]): Future[String] =`	=	`def searchFor(word: String)(lines: Vector[String]): Future[String] =`
`  Future(lines.foldLeft("")(accumulateLinesContaining(word)))`	<>	`  Future(lines.foldLeft("")(accumulateLinesContaining(word))).printThreadName`

Let's run the new program and search for the word '**joyous**' again.

That worked: the collection of lines was split into **six smaller collections** which got **folded in parallel**, each in a **separate thread**, with the names of the threads visible in the **console output**.

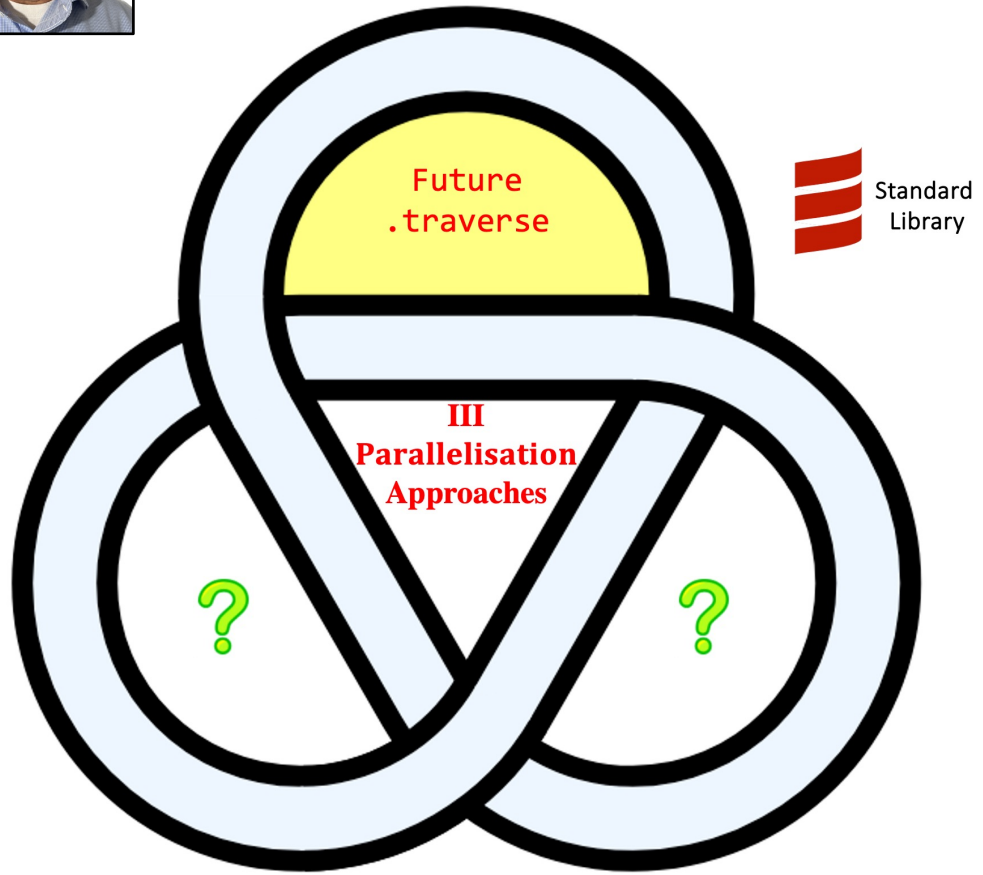
When the whole collection was processed **sequentially**, the **processing** took a bit over **one minute**, but now that different parts of the collection are being processed in **parallel**, the **processing** took **25 seconds**, almost **a third of the time**.

I ran the program another **three times**, and its execution times were **28s**, **28s** and **26s**.

@philip_schwarz

```
$ sbt "run joyous"
…
Multiple main classes detected. Select one to run:
 [1] runUsingFutureTraverse
 [2] runWithoutParallelism

Enter number: 1
[info] running runUsingFutureTraverse joyous
Successfully obtained 99,975,000 lines of text to search.
[scala-execution-context-global-167]
[scala-execution-context-global-169]
[scala-execution-context-global-170]
[scala-execution-context-global-165]
[scala-execution-context-global-168]
[scala-execution-context-global-166]
Found the word in the following 4,000 lines of text:
'stirs, a tumultuous city, joyous city: thy slain men are not slain'
'23:7 Is this your joyous city, whose antiquity is of ancient days? her'
'upon all the houses of joy in the joyous city: 32:14 Because the'
'12:11 Now no chastening for the present seemeth to be joyous, but'
'stirs, a tumultuous city, joyous city: thy slain men are not slain'
'23:7 Is this your joyous city, whose antiquity is of ancient days? her'
'upon all the houses of joy in the joyous city: 32:14 Because the'
'12:11 Now no chastening for the present seemeth to be joyous, but'
'stirs, a tumultuous city, joyous city: thy slain men are not slain'
'23:7 Is this your joyous city, whose antiquity is of ancient days? her'
'upon all the houses of joy in the joyous city: 32:14 Because the'
'12:11 Now no chastening for the present seemeth to be joyous, but'
…
'stirs, a tumultuous city, joyous city: thy slain men are not slain'
'23:7 Is this your joyous city, whose antiquity is of ancient days? her'
'upon all the houses of joy in the joyous city: 32:14 Because the'
'12:11 Now no chastening for the present seemeth to be joyous, but'
'stirs, a tumultuous city, joyous city: thy slain men are not slain'
'23:7 Is this your joyous city, whose antiquity is of ancient days? her'
'upon all the houses of joy in the joyous city: 32:14 Because the'
'12:11 Now no chastening for the present seemeth to be joyous, but'
[success] Total time: 25 s, completed 12 Nov 2023, 11:56:55
```

That was the **first** of the **three approaches** that we are going to explore for **parallelising** our **left fold**.



As a recap, before moving on, the next slide shows the whole code for the new program.

New/changed code is highlighted with a yellow background.

```scala
@main def runUsingFutureTraverse(word: String): Unit =
  getText(book = theBible, copies = 1_000)
    .fold(
      error => handleErrorGettingText(error),
      lines =>
        announceSuccessGettingText(lines)
        val matches = find(word, lines)
        announceMatchingLines(matches))
```

```scala
def find(word: String, lines: Vector[String]): String =
  val batchSize = lines.size / (numberOfCores / 2)
  val groupsOfLines = lines.grouped(batchSize).toVector
  Await.result(
    Future.traverse(groupsOfLines)(searchFor(word))
          .map(_.foldLeft("")(_++_)),
    Duration.Inf)
```

```scala
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.duration.Duration
import scala.concurrent.{Await, Future}
```

```scala
val numberOfCores = Runtime.getRuntime().availableProcessors()
```

```scala
def searchFor(word: String)(lines: Vector[String]): Future[String] =
  Future(lines.foldLeft("")(accumulateLinesContaining(word))).printThreadName()
```

```scala
extension [A](fa: Future[A])
  def printThreadName(): Future[A] =
    for
      a <- fa
      _ = println(s"[${Thread.currentThread().getName}]")
    yield a
```

```scala
def accumulateLinesContaining(word: String): (String, String) => String =
  (acc, line) => if line.matches(s".*$word.*") then s"$acc\n'$line'" else acc
```

```scala
def getText(book: Book, copies: Int = 1): Try[Vector[String]] =
  Using(Source.fromURL(book.url)): source =>
    val lines = source.getLines.toVector
    Vector.fill(copies)(lines).flatten
```

```scala
case class Book(
  name: String,
  numberOfLines: Int,
  numberOfBytes: Int,
  url: URL
)
```

```scala
import java.net.URL
import scala.io.Source
import scala.util.{Try, Using}
```

```scala
def handleErrorGettingText[A](error: Throwable): A =
  throw IllegalStateException(s"Failed to obtain the text lines to be searched.", error)

def announceSuccessGettingText(lines: Vector[String]): Unit =
  println(f"Successfully obtained ${lines.length}%,d lines of text to search.")

def announceMatchingLines(lines: String): Unit
  println(f"Found the word in the following ${lines.count(_ == '\n')}%,d lines of text: $lines")
```

```scala
val theBible = Book(
  name = "The King James Version of the Bible",
  numberOfLines = 99_975,
  numberOfBytes = 4_456_041,
  url = URL("https://gutenberg.org/cache/epub/10/pg10.txt")
)
```

If we want to **parallelise** the **left fold**, and we *are* allowed to use **external libraries**, how can we do it?

One way is to do something similar to what we did using the **Future monad** and its **traverse** function, but using the **Cats Effect IO monad** and **Cats Core**'s **parTraverse** function.

Because this approach is very similar to the previous one, I am going to explain it by revisiting the explanation of the latter, so if you get a sense of déjà vu, that's intentional, because I reckon it will make things easier to digest.

Let's write a new **method** called **runUsingCatsParTraverse**. While its body is very similar to that of **runWithoutParallelism**…

`@main def runWithoutParallelism(word: String): Unit =`
`  getText(book = theBible, copies = 1_000)`
 	
`    .fold(`
`      error => handleErrorGettingText(error),`
	<>	`def runUsingCatsParTraverse(word: String): IO[Unit] =`
		`  IO.blocking(getText(book = theBible,copies = 1_000))`
		`    .flatMap(`
		`      _.fold(`
		`        error => IO(handleErrorGettingText(error)),`

`      lines =>`	=	`        lines =>`

`        announceSuccessGettingText(lines)`
`        val matches = find(word, lines)`
`        announceMatchingLines(matches)`
	<>	`          IO(announceSuccessGettingText(lines)) *>`
		`            find(word, lines)`
		`              .map(matches => announceMatchingLines(matches))`

`    )`	=	`      )`
	-+	`    )`

…the **find** function that it invokes cannot be the one invoked by **runWithoutParallelism** …

```
def find(word: String, lines: Vector[String]): String =
  lines.foldLeft("")(accumulateLinesContaining(word))
```

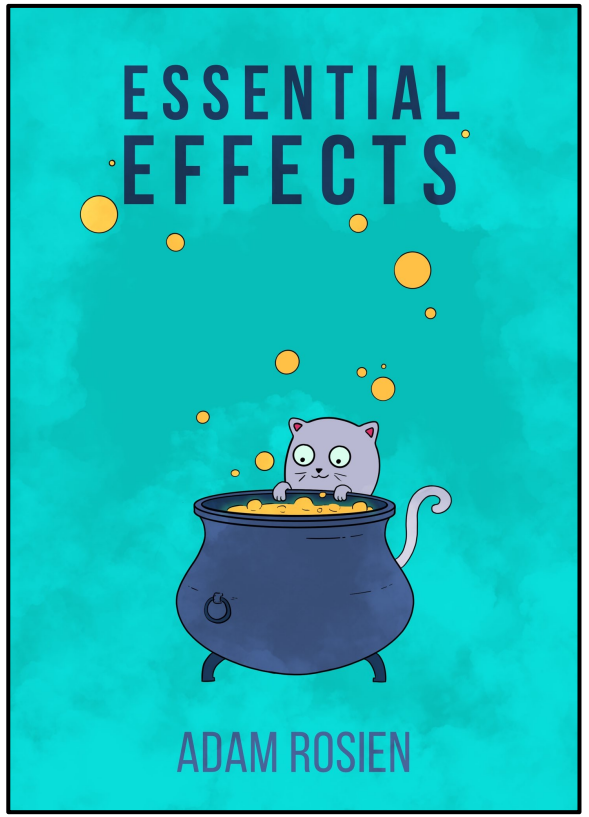
…it needs to be rewritten, which we'll be doing next.

If you are familiar with **Cats**' **parTraverse** then please skip the next two slides.

If you are not, there is an amazingly helpful, clear, detailed, and hands-on explanation of **parMapN** and **parTraverse** (and much more) in **Adam Rosien's** great book: **Essential Effects**.

While there is no substitute for reading Chapter 3, **Parallel execution**, the following two slides are my humble attempt to capture some of the information imparted by that chapter, by cherry picking some of its sentences, passages, and diagrams, and stitching them together in an order of my own devising, in the hope that it serves as a very brief, high-level introduction to concepts that are fully explained in the book.

Adam Rosien
@arosien

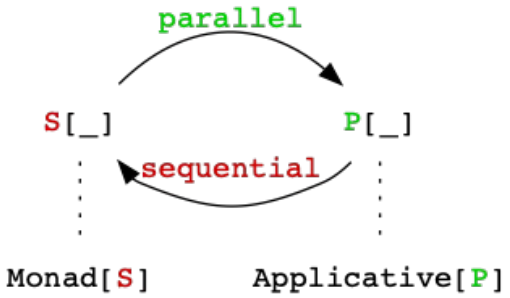


IO does not support parallel operations itself, because it is a Monad.

The Parallel typeclass specifies the translation between a pair of effect types: one that is a Monad and the other that is “only” an Applicative.

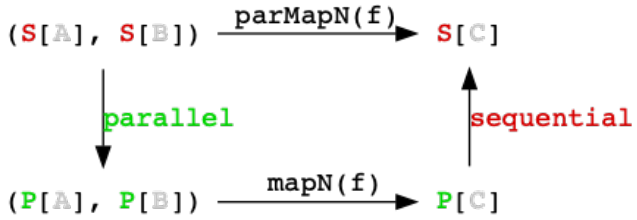
The Parallel typeclass encodes transformations between a sequential type S and a parallel type P.

```
Parallel[S[_]] { type P[_] }
```



Parallel[IO] connects the IO effect to its parallel counterpart, IO.Par.

parMapN is the parallel version of the applicative mapN method. It lets us combine multiple effects into one, in parallel, by specifying how to combine the outputs of the effects



The parMapN extension method is implemented as (1) translating the sequential effect types into parallel representations, (2) performing the alternative mapN, and (3) translating the parallel representation back to the sequential form.

Adam Rosien
@arosien

3.5. parTraverse

parTraverse is the **parallel** version of traverse; both have the type signature:

```
F[A] => (A => G[B]) => G[F[B]]
```

For example, if F is List and G is IO, then (par)traverse would be a function from a List[A] to an IO[List[B]] when given a function A ⇒ IO[B].

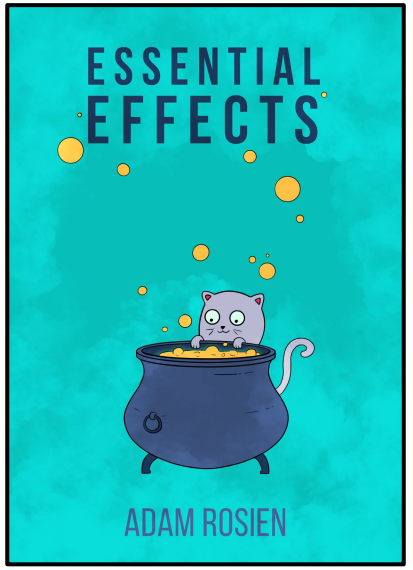
```
List[A] => (A => IO[B]) => IO[List[B]]
```

The most common use case of (par)traverse is when you have a collection of work to be done, and a function which handles one unit of work. Then you get a collection of results **combined** into one **effect**:

```
val work: List[WorkUnit] = ???
def doWork(workUnit: WorkUnit): IO[Result] = ??? ①
val results: IO[List[Result]] = work.parTraverse(doWork)
```

① Note that processing one unit of work is an **effect**, in this case, IO.

Adam Rosien
@arosien



Let's add **Cats Core** and **Cats Effect** to the build…

```
libraryDependencies += "org.typelevel" %% "cats-core" % "2.9.0"
libraryDependencies += "org.typelevel" %% "cats-effect" % "3.5.2"
```

and let's import the following…

```
import cats.effect.{ExitCode, IO, IOApp}
import cats.syntax.foldable.*
import cats.syntax.parallel.*
```

```
def find(word: String, lines: Vector[String]): IO[String] =
  val batchSize = lines.size / (numberOfCores / 2)
  val groupsOfLines = lines.grouped(batchSize).toVector
  groupsOfLines
    .parTraverse(searchFor(word))
    .map(_.combineAll)
```

We use the **grouped** function to break the collection of **text lines** into **multiple smaller collections**, one for each of the number of **CPU cores** that we want to use for **parallelising** the search.

We have decided to use half of the available **cores** for this purpose.

We then **parTaverse** the **smaller collections** of lines with a **searchFor** function that is used to **fold** each collection, and which is essentially the **find** function that we have been using up to now (on the right), except that it does the **folding** in an **IO**.

Compare the new **find** function (above on the left), with the one used by **runWithoutParallelism** (below).

```
def searchFor(word: String)(lines: Vector[String]): IO[String] =
  IO(lines.foldLeft("")(accumulateLinesContaining(word)))
```

```
def find(word: String, lines: Vector[String]): String =
  lines.foldLeft("")(accumulateLinesContaining(word))
```

```
def searchFor(word: String)(lines: Vector[String]): IO[String] =
  IO(lines.foldLeft("")(accumulateLinesContaining(word)))
```

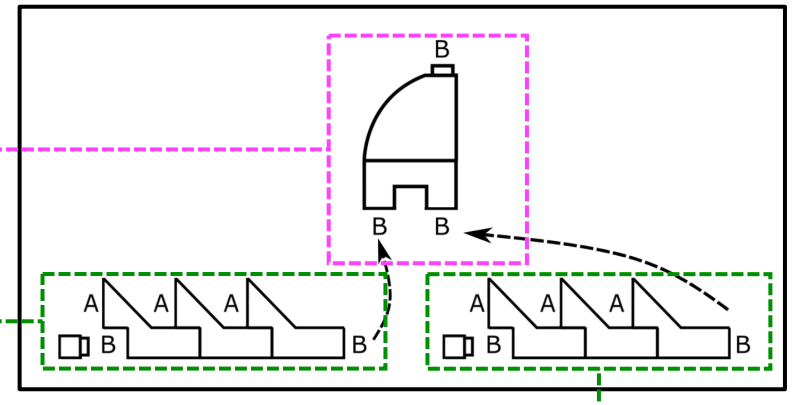
<>

```
def find(word: String, lines: Vector[String]): String =
  lines.foldLeft("")(accumulateLinesContaining(word))
```

parTaverse first creates a collection of **IO**s, each of which does a **left fold** of the **A**s in a **smaller collection**, and then turns that collection **inside out**, i.e. it turns the collection of **IO**s of **B** into an **IO** of a collection of **B**s.

The **IO**s execute in **parallel**, each one on a **separate core**, and when they **complete**, each of them yields the result of the **left fold** of a **smaller collection**.

When the **IO** of a collection of **B**s returned by **parTaverse** **completes**, the **find** function has a collection of **B**s, which it then **folds** into a single overall **B**.




```
def accumulateLinesContaining(word: String): (String, String) => String =
  (acc, line) => if line.matches(s".*$word.*") then s"$acc\n'$line'" else acc
```

Now that the program **partitions** the collection of **text lines** into **multiple smaller collections**, and **folds** each of those **smaller collections** on a **separate core**, let's get the program to **print** on the **console** the **name** of the **thread** that does the **folding** on each such **core**.

To do that, let's adapt the **printThreadName** extension method that we wrote earlier, so that it also works for **IO**

```
extension [A](fa: Future[A])
  def printThreadName(): Future[A] =
    for
      a <- fa
      _ = println(s"[${Thread.currentThread().getName}]")
    yield a
```

<>

```
import cats.syntax.functor.*
extension [A, F[_] : Functor](fa: F[A])
  def printThreadName(): F[A] =
    for
      a <- fa
      _ = println(s"[${Thread.currentThread().getName}]")
    yield a
```

Now all we have to do is invoke the new method.

```
def searchFor(word: String)(lines: Vector[String]): IO[String] =
  IO(lines.foldLeft("")(accumulateLinesContaining(word)))
```

=

```
def searchFor(word: String)(lines: Vector[String]): IO[String] =
  IO(lines.foldLeft("")(accumulateLinesContaining(word))).printThreadName()
```

The next slide shows the whole code for the new program.

New/changed code is again highlighted with a yellow background.

@philip_schwarz

```scala
object CatsParTraverse extends IOApp:
  override def run(args: List[String]): IO[ExitCode] =
    val word = args.headOption.getOrElse("joyous")
    runUsingCatsParTraverse(word).as(ExitCode.Success)
```

```scala
def runUsingCatsParTraverse(word: String): Unit =
  getText(book = theBible, copies = 1_000)
    .fold(
      error => handleErrorGettingText(error),
      lines =>
        announceSuccessGettingText(lines)
        val matches = find(word, lines)
        announceMatchingLines(matches))
```

```scala
def find(word: String, lines: Vector[String]): String =
  val batchSize = lines.size / (numberOfCores / 2)
  val groupsOfLines = lines.grouped(batchSize).toVector
  groupsOfLines
    .parTraverse(searchFor(word))
    .map(_.combineAll)
```

```scala
import cats.effect.{ExitCode, IO, IOApp}
import cats.syntax.foldable.*
import cats.syntax.parallel.*
```

```scala
val numberOfCores = Runtime.getRuntime().availableProcessors()
```

```scala
def searchFor(word: String)(lines: Vector[String]): IO[String] =
  IO(lines.foldLeft("")(accumulateLinesContaining(word))).printThreadName()
```

```scala
import cats.syntax.functor.*
extension [A, F[_]: Functor](fa: F[A])
  def printThreadName(): F[A] =
    for
      a <- fa
      _ = println(s"[${Thread.currentThread().getName}]")
    yield a
```

```scala
def accumulateLinesContaining(word: String): (String, String) => String =
  (acc, line) => if line.matches(s".*$word.*") then s"$acc\n'$line'" else acc
```

```scala
def getText(book: Book, copies: Int = 1): Try[Vector[String]] =
  Using(Source.fromURL(book.url)): source =>
    val lines = source.getLines.toVector
    Vector.fill(copies)(lines).flatten
```

```scala
case class Book(
  name: String,
  numberOfLines: Int,
  numberOfBytes: Int,
  url: URL
)
```

```scala
import java.net.URL
import scala.io.Source
import scala.util.{Try, Using}
```

```scala
def handleErrorGettingText[A](error: Throwable): A =
  throw IllegalStateException(s"Failed to obtain the text lines to be searched.", error)

def announceSuccessGettingText(lines: Vector[String]): Unit =
  println(f"Successfully obtained ${lines.length}%,d lines of text to search.")

def announceMatchingLines(lines: String): Unit
  println(f"Found the word in the following ${lines.count(_ == '\n')}%,d lines of text: $lines")
```

```scala
val theBible = Book(
  name = "The King James Version of the Bible",
  numberOfLines = 99_975,
  numberOfBytes = 4_456_041,
  url = URL("https://gutenberg.org/cache/epub/10/pg10.txt")
)
```

Note the following similarities and differences between the code for the **parallelisation approach** using **Future** + **traverse** and that for the **approach** using **IO** + **parTraverse**.

```
@main def runUsingFutureTraverse(word: String): Unit =
  getText(book = theBible, copies = 1_000)
    .fold(
      error => handleErrorGettingText(error),
      lines =>
        announceSuccessGettingText(lines)
        val matches = find(word, lines)
        announceMatchingLines(matches))
```

```
def runUsingCatsParTraverse(word: String): Unit =
  getText(book = theBible, copies = 1_000)
    .fold(
      error => handleErrorGettingText(error),
      lines =>
        announceSuccessGettingText(lines)
        val matches = find(word, lines)
        announceMatchingLines(matches))
```

```
def find(word: String, lines: Vector[String]): String =
  val batchSize = lines.size / (numberOfCores / 2)
  val groupsOfLines = lines.grouped(batchSize).toVector
  Await.result(
    Future.traverse(groupsOfLines)(searchFor(word))
          .map(_.foldLeft("")(_++_)),
    Duration.Inf)
```

```
def find(word: String, lines: Vector[String]): String =
  val batchSize = lines.size / (numberOfCores / 2)
  val groupsOfLines = lines.grouped(batchSize).toVector
  groupsOfLines
    .parTraverse(searchFor(word))
    .map(_.combineAll)
```

```
def searchFor(word: String)(lines: Vector[String]): Future[String] =
  Future(lines.foldLeft("")(accumulateLinesContaining(word))).printThreadName()
```

```
def searchFor(word: String)(lines: Vector[String]): IO[String] =
  IO(lines.foldLeft("")(accumulateLinesContaining(word))).printThreadName()
```

Let's run the new program and search for the word '**joyous**' again.

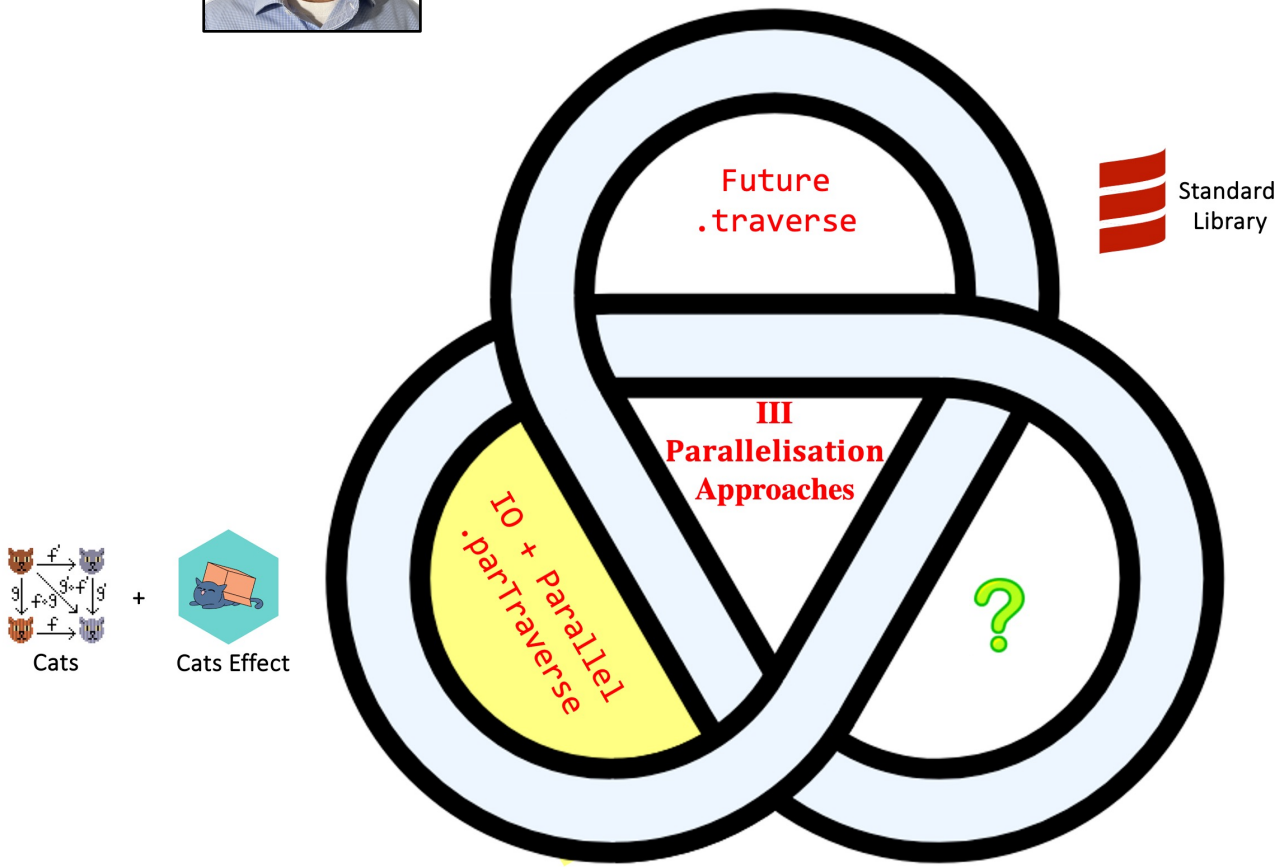
That worked: the collection of lines was split into **six smaller collections** which got **folded in parallel**, each in a **separate thread**, with the names of the threads visible in the **console output**.

When the whole collection was processed **sequentially**, the **processing** took a bit over **one minute**, but now that different parts of the collection are being processed in **parallel**, the **processing** took **28 seconds**, almost **a third of the time**.

I ran the program another **three times**, and its execution times were **30s**, **37s** and **28s**.

```
$ sbt "run joyous"
…
Multiple main classes detected. Select one to run:
 [1] CatsParTraverse
 [2] runUsingFutureTraverse
 [3] runWithoutParallelism

Enter number: 1
[info] running CatsParTraverse joyous
Successfully obtained 99,975,000 lines of text to search.
[io-compute-3]
[io-compute-5]
[io-compute-11]
[io-compute-6]
[io-compute-10]
[io-compute-8]
Found the word in the following 4,000 lines of text:
'stirs, a tumultuous city, joyous city: thy slain men are not slain'
'23:7 Is this your joyous city, whose antiquity is of ancient days? her'
'upon all the houses of joy in the joyous city: 32:14 Because the'
'12:11 Now no chastening for the present seemeth to be joyous, but'
'stirs, a tumultuous city, joyous city: thy slain men are not slain'
'23:7 Is this your joyous city, whose antiquity is of ancient days? her'
'upon all the houses of joy in the joyous city: 32:14 Because the'
'12:11 Now no chastening for the present seemeth to be joyous, but'
'stirs, a tumultuous city, joyous city: thy slain men are not slain'
'23:7 Is this your joyous city, whose antiquity is of ancient days? her'
'upon all the houses of joy in the joyous city: 32:14 Because the'
'12:11 Now no chastening for the present seemeth to be joyous, but'
…
'stirs, a tumultuous city, joyous city: thy slain men are not slain'
'23:7 Is this your joyous city, whose antiquity is of ancient days? her'
'upon all the houses of joy in the joyous city: 32:14 Because the'
'12:11 Now no chastening for the present seemeth to be joyous, but'
'stirs, a tumultuous city, joyous city: thy slain men are not slain'
'23:7 Is this your joyous city, whose antiquity is of ancient days? her'
'upon all the houses of joy in the joyous city: 32:14 Because the'
'12:11 Now no chastening for the present seemeth to be joyous, but'
[success] Total time: 28 s, completed 12 Nov 2023, 11:56:55
```

That was the **second** of the **three approaches** that we are going to explore for **parallelising** our **left fold**.



For our third and final approach to **parallelising** the **left fold**, let's go back to using the **scala parallel collections library**.

What we are going to do is use the library's **aggregate function**.

Let's write a new **method** called **runUsingParallelAggregation**, whose body is identical to that of **runWithoutParallelism**.

```
@main def runWithoutParallelism(word: String = "joyous"): Unit =
  getText(book = theBible, copies = 1_000)
    .fold(
      error => handleErrorGettingText(error),
      lines =>
        announceSuccessGettingText(lines)
        val matches = find(word, lines)
        announceMatchingLines(matches)
    )
```

<>

```
@main def runUsingParallelAggregation(word: String = "joyous"): Unit =
  getText(book = theBible, copies = 1_000)
    .fold(
      error => handleErrorGettingText(error),
      lines =>
        announceSuccessGettingText(lines)
        val matches = find(word, lines)
        announceMatchingLines(matches)
    )
```

As for the **find** function that it invokes, here is how it needs to change

```
def find(word: String, lines: Vector[String]): String =
  lines.foldLeft("")(accumulateLinesContaining(word))
```

<>

```
def find(word: String, lines: Vector[String]): String =
  lines.par.aggregate("")(seqop = accumulateLinesContaining(word), combop = _++_)
```

The function first converts the **sequential** vector of lines to a **parallel collection**, and then invokes the **aggregate** function on the latter.

```
def find(word: String, lines: Vector[String]): String =
  lines.par.aggregate("")(seqop = accumulateLinesContaining(word), combop = _++_)
```

For an explanation of the **aggregate** function, on the next two slides we are going to turn to **Aleksandar Prokopec**'s book: **Learning Concurrent Programming in Scala**.

On the first slide, as a recap, is his explanation of why **foldLeft** cannot be **parallelised**, and on the second slide, his explanation of how the **aggregate** function allows a **left fold** to be **parallelised**.

We are also going to throw in his diagrams from **EPFL**'s **Scala Parallel Programming** course.

Non-parallelizable operations

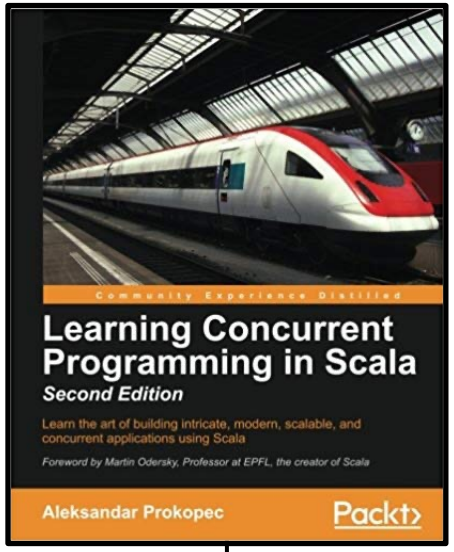
While most parallel collection operations achieve superior performance by executing on several processors, some operations are inherently sequential, and their semantics do not allow them to execute in parallel. Consider the foldLeft method from the Scala collections API:

```
def foldLeft[S](z: S)(f: (S, T) => S): S
```

This method visits elements of the collection going from left to right ...

The crucial property of the foldLeft operation is that it traverses the elements of the list by going from left to right. This is reflected in the type of the function f; it accepts an accumulator of type S and a list value of type T. The function f cannot take two values of the accumulator type S and merge them into a new accumulator of type S. As a consequence, computing the accumulator cannot be implemented in parallel; the foldLeft method cannot merge two accumulators from two different processors.

...

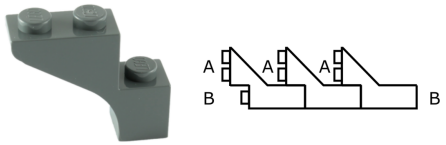


EPFL: parprog1
Parallel programming

Non-Parallelizable Operations

Let's examine the `foldLeft` signature:

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

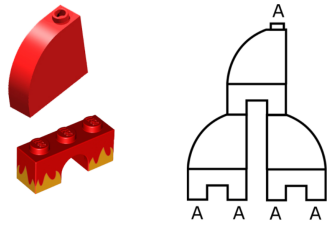


Operations `foldRight`, `reduceLeft`, `reduceRight`, `scanLeft` and `scanRight` similarly must process elements sequentially.

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.

Aleksandar Prokopec
@alexprokopec

To specify how the accumulators produced by different processors should be merged together, we need to use the aggregate method.

The aggregate method is similar to the foldLeft operation, but it does not specify that the elements are traversed from left to right. Instead, it only specifies that subsets of elements are visited going from left to right; each of these subsets can produce a separate accumulator. The aggregate method takes an additional function of type (S, S) => S, which is used to merge multiple accumulators.

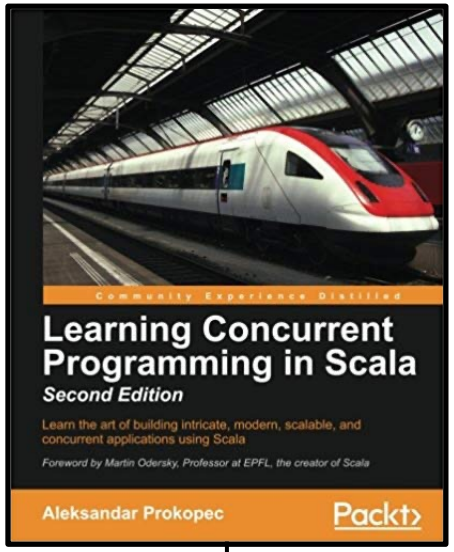
```
def aggregate[S](z: => S)(seqop: (S, T) => S, combop: (S, S) => S): S
```

```
d.aggregate("")
          ((acc, line) =>   if (line.matches(".*TEXTAREA.*")) s"$acc\n$line" else acc,
           (acc1, acc2) => acc1 + acc2 )
```

...

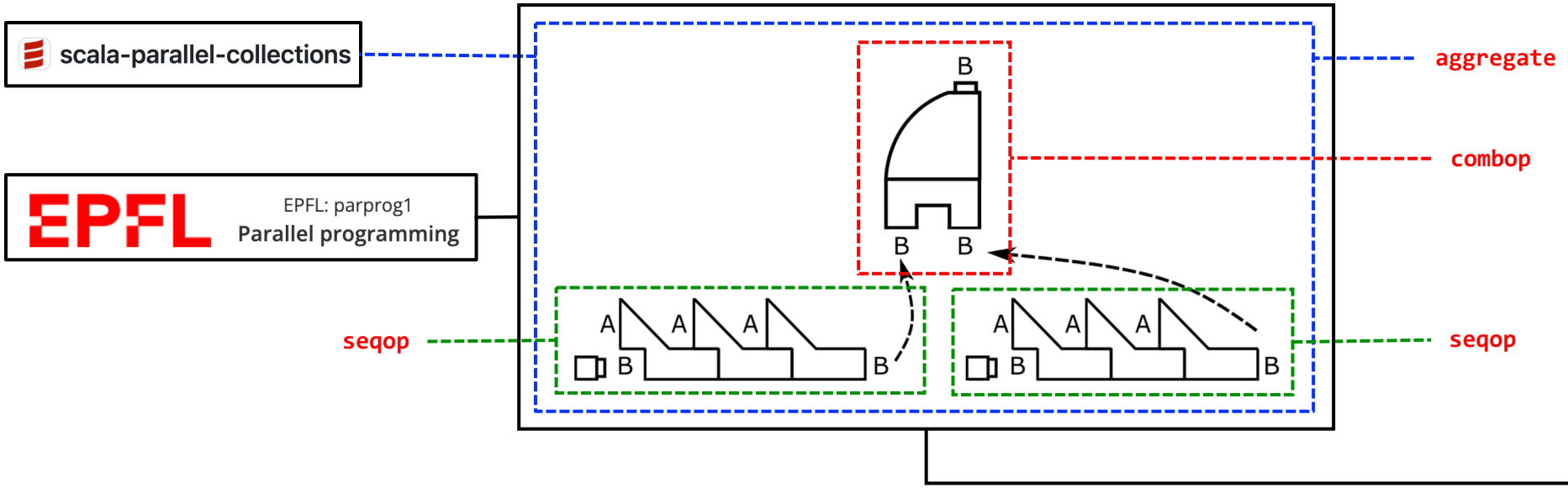
When doing these kinds of reduction operation in parallel, we can alternatively use the reduce or fold methods, which do not guarantee going from left to right. The aggregate method is more expressive, as it allows the accumulator type to be different from the type of the elements in the collection.

...



Aleksandar Prokopec

@alexprokopec



Let’s run the new program and search for the word ‘**joyous**’ again.

That worked: the collection of lines was split into **six smaller collections** which got **folded in parallel**, each in a **separate thread**, with the names of the threads visible in the **console output**.

When the whole collection was processed **sequentially**, the **processing** took a bit over **one minute**, but now that different parts of the collection are being processed in **parallel**, the **processing** took **24 seconds**, almost **a third of the time**.

I ran the program another **three times**, and its execution times were **28s**, **25s** and **26s**.

@philip_schwarz

```
$ sbt "run joyous"
…
Multiple main classes detected. Select one to run:
 [1] CatsParTraverse
 [2] runUsingFutureTraverse
 [3] runUsingParallelAggregation
 [4] runWithoutParallelism

Enter number: 3
[info] running runUsingParallelAggregation joyous
Successfully obtained 99,975,000 lines of text to search.
Found the word in the following 4,000 lines of text:
'stirs, a tumultuous city, joyous city: thy slain men are not slain'
'23:7 Is this your joyous city, whose antiquity is of ancient days? her'
'upon all the houses of joy in the joyous city: 32:14 Because the'
'12:11 Now no chastening for the present seemeth to be joyous, but'
'stirs, a tumultuous city, joyous city: thy slain men are not slain'
'23:7 Is this your joyous city, whose antiquity is of ancient days? her'
'upon all the houses of joy in the joyous city: 32:14 Because the'
'12:11 Now no chastening for the present seemeth to be joyous, but'
'stirs, a tumultuous city, joyous city: thy slain men are not slain'
'23:7 Is this your joyous city, whose antiquity is of ancient days? her'
'upon all the houses of joy in the joyous city: 32:14 Because the'
'12:11 Now no chastening for the present seemeth to be joyous, but'
…
'stirs, a tumultuous city, joyous city: thy slain men are not slain'
'23:7 Is this your joyous city, whose antiquity is of ancient days? her'
'upon all the houses of joy in the joyous city: 32:14 Because the'
'12:11 Now no chastening for the present seemeth to be joyous, but'
'stirs, a tumultuous city, joyous city: thy slain men are not slain'
'23:7 Is this your joyous city, whose antiquity is of ancient days? her'
'upon all the houses of joy in the joyous city: 32:14 Because the'
'12:11 Now no chastening for the present seemeth to be joyous, but'
[success] Total time: 24 s, completed 12 Nov 2023, 11:56:55
```

```scala
@main def runUsingParallelAggregation(word: String): Unit =
  getText(book = theBible, copies = 1_000)
    .fold(
      error => handleErrorGettingText(error),
      lines =>
        announceSuccessGettingText(lines)
        val matches = find(word, lines)
        announceMatchingLines(matches))
```

```scala
def find(word: String, lines: Vector[String]): String =
  lines.par.aggregate("")(seqop = accumulateLinesContaining(word), combop = _++_)
```

```scala
import scala.collection.parallel.CollectionConverters.*
```

```scala
def accumulateLinesContaining(word: String): (String, String) => String =
  (acc, line) => if line.matches(s".*$word.*") then s"$acc\n'$line'" else acc
```

```scala
def getText(book: Book, copies: Int = 1): Try[Vector[String]] =
  Using(Source.fromURL(book.url)): source =>
    val lines = source.getLines.toVector
    Vector.fill(copies)(lines).flatten
```

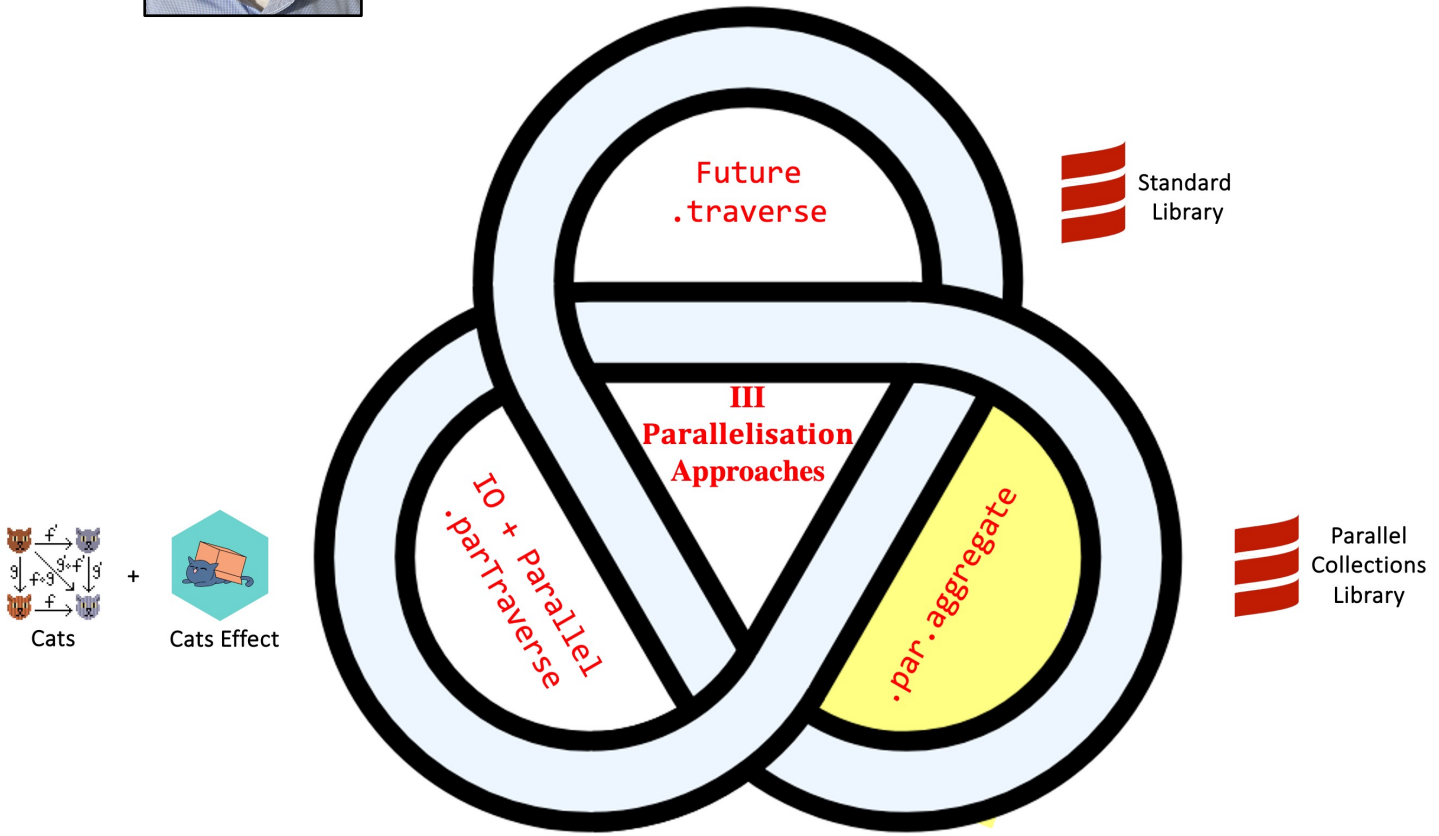
```scala
case class Book(
  name: String,
  numberOfLines: Int,
  numberOfBytes: Int,
  url: URL
)
```

```scala
import java.net.URL
import scala.io.Source
import scala.util.{Try, Using}
```

```scala
def handleErrorGettingText[A](error: Throwable): A =
  throw IllegalStateException(s"Failed to obtain the text lines to be searched.", error)

def announceSuccessGettingText(lines: Vector[String]): Unit =
  println(f"Successfully obtained ${lines.length}%,d lines of text to search.")

def announceMatchingLines(lines: String): Unit
  println(f"Found the word in the following ${lines.count(_ == '\n')}%,d lines of text: $lines")
```

```scala
val theBible = Book(
  name = "The King James Version of the Bible",
  numberOfLines = 99_975,
  numberOfBytes = 4_456_041,
  url = URL("https://gutenberg.org/cache/epub/10/pg10.txt")
)
```

That was the **third** and final of the **three approaches** that we explored for **parallelising** our **left fold**.



In conclusion, the next slide compares and contrasts four versions of the **find** function, the one in the original **sequential** code, and the ones in the three different approaches to **parallelisation** that we have explored.

The slide after that is the same but without any highlighting

No Parallelism

```
def find(word: String, lines: Vector[String]): String =
  lines.foldLeft("")(accumulateLinesContaining(word))
```

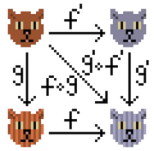
Future .traverse

```
def find(word: String, lines: Vector[String]): String =
  val batchSize = lines.size / (numberOfCores / 2)
  val groupsOfLines = lines.grouped(batchSize).toVector
  Await.result(
    Future.traverse(groupsOfLines)(searchFor(word))
          .map(_.foldLeft("")(_++_)),
    Duration.Inf)

def searchFor(word: String)(lines: Vector[String]): Future[String] =
  Future(lines.foldLeft("")(accumulateLinesContaining(word)))
```

Standard Library

Parallel .parTraverse

```
def find(word: String, lines: Vector[String]): IO[String] =
  val batchSize = lines.size / (numberOfCores / 2)
  val groupsOfLines = lines.grouped(batchSize).toVector
  groupsOfLines
    .parTraverse(searchFor(word))
    .map(_.combineAll)

def searchFor(word: String)(lines: Vector[String]): IO[String] =
  IO(lines.foldLeft("")(accumulateLinesContaining(word)))
```



Cats

+

Cats Effect

.par.aggregate

```
import scala.collection.parallel.CollectionConverters.*

def find(word: String, lines: Vector[String]): String =
  lines.par.aggregate("")(seqop = accumulateLinesContaining(word), combop = _++_)
```

Parallel Collections Library

No Parallelism

```scala
def find(word: String, lines: Vector[String]): String =
  lines.foldLeft("")(accumulateLinesContaining(word))
```

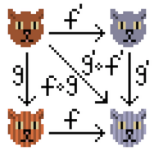
Future .traverse

```scala
def find(word: String, lines: Vector[String]): String =
  val batchSize = lines.size / (numberOfCores / 2)
  val groupsOfLines = lines.grouped(batchSize).toVector
  Await.result(
    Future.traverse(groupsOfLines)(searchFor(word))
          .map(_.foldLeft("")(_++_)),
    Duration.Inf)

def searchFor(word: String)(lines: Vector[String]): Future[String] =
  Future(lines.foldLeft("")(accumulateLinesContaining(word)))
```

Standard Library

Parallel .parTraverse

```scala
def find(word: String, lines: Vector[String]): IO[String] =
  val batchSize = lines.size / (numberOfCores / 2)
  val groupsOfLines = lines.grouped(batchSize).toVector
  groupsOfLines
    .parTraverse(searchFor(word))
    .map(_.combineAll)

def searchFor(word: String)(lines: Vector[String]): IO[String] =
  IO(lines.foldLeft("")(accumulateLinesContaining(word)))
```



Cats

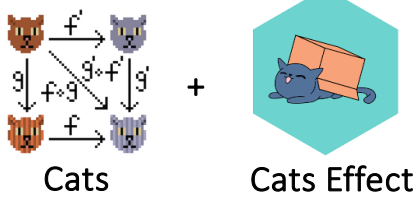
+

Cats Effect

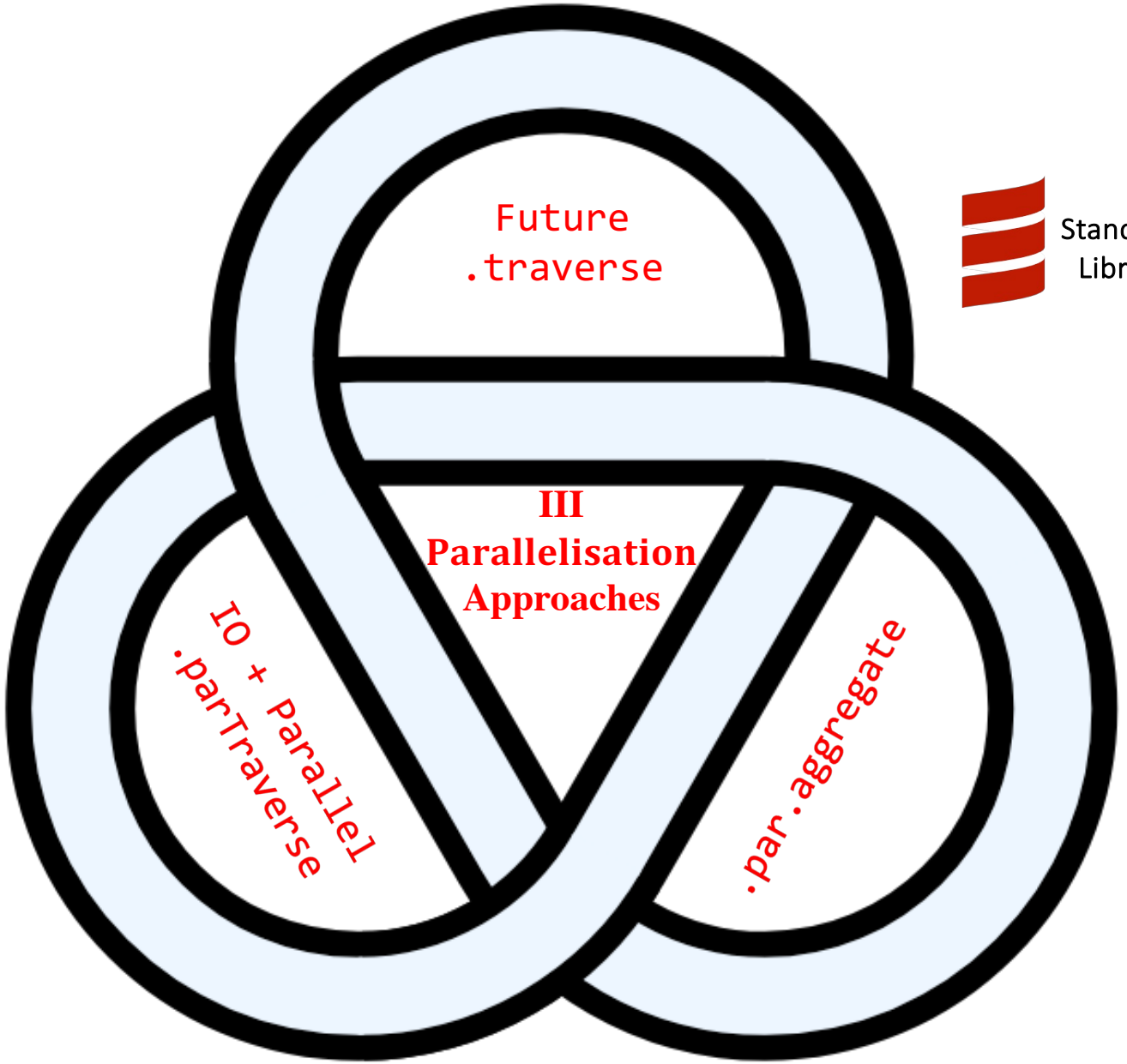
.par.aggregate

```scala
import scala.collection.parallel.CollectionConverters.*

def find(word: String, lines: Vector[String]): String =
  lines.par.aggregate("")(seqop = accumulateLinesContaining(word), combop = _++_)
```

Parallel Collections Library



+



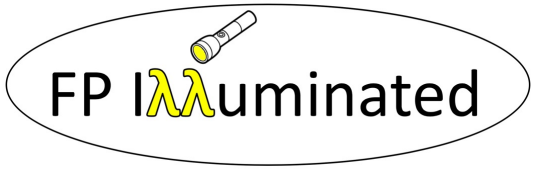
Standard
Library

Parallel
Collections
Library

@philip_schwarz

That's all.

I hope you found it useful.



If you ejoyed it, you can find more like it at http://fpilluminated.com/