Data-Parallel Operations

Parallel Programming and Data Analysis

Aleksandar Prokopec
Scala Collection Recap

Scala Collections already come with a data-parallel programming model.
Scala Collection Recap

Scala Collections already come with a data-parallel programming model. Easy way to manipulate data – a large number of combinator methods.

(1 until 1000).filter(n => n % 3 == 0).foldLeft(0)(_ + _)

**Question:**

What does this expression evaluate to?
Scala Collection Recap

Scala Collections already come with a data-parallel programming model.
Easy way to manipulate data – a large number of combinator methods.

(1 until 1000).filter(n => n % 3 == 0).foldLeft(0)(_ + _)

Question:
What does this expression evaluate to?

- numbers smaller than 1000 that are divisible by 3
- a sum of numbers smaller than 1000 that are divisible by 3
- a sum of numbers smaller than 1000 modulo 3
(100 to 999).flatMap(i => (i to 999).map(i * _))
 .filter(n => n.toString == n.toString.reverse).max

*Question:*

What does this expression evaluate to?
Scala Collection Recap

\[(100 \text{ to } 999).flatMap(i => (i \text{ to } 999).map(i * _))\]
\[.filter(n => n.toString == n.toString.reverse).max\]

**Question:**
What does this expression evaluate to?

- the largest 3-digit palindrome
- the largest product of 3-digit palindromes
- the largest palindrome that is a product of 3-digit numbers
The `par` method transforms a sequential Scala collection into a parallel one.

```
(100 to 999).par.flatMap(i => (i to 999).map(i * _))
  .filter(n => n.toString == n.toString.reverse).max
```
The `par` method transforms a sequential Scala collection into a parallel one.

```
(100 to 999).par.flatMap(i => (i to 999).map(i * _))
  .filter(n => n.toString == n.toString.reverse).max
```

- the same operations as sequential collections, but most execute in parallel
The `par` method transforms a sequential Scala collection into a parallel one.

```
(100 to 999).par.flatMap(i => (i to 999).map(i * _))
  .filter(n => n.toString == n.toString.reverse).max
```

- the same operations as sequential collections, but most execute in parallel
- *deterministic* as long as they are used functionally
Non-Parallelizable Operations

Task: implement the method `sum` using the `foldLeft` method.

```scala
def sum(xs: Array[Int]): Int
```
Non-Parallelizable Operations

Task: implement the method `sum` using the `foldLeft` method.

```scala
def sum(xs: Array[Int]): Int = {
  xs.par.foldLeft(0)(_ + _)
}
```

Does this implementation execute in parallel?
Non-Parallelizable Operations

Task: implement the method `sum` using the `foldLeft` method.

```scala
def sum(xs: Array[Int]): Int = {
    xs.par.foldLeft(0)(_ + _)
}
```

Does this implementation execute in parallel?

Why not?
Non-Parallelizable Operations

Let's examine the foldLeft signature:

```scala
def foldLeft[B](z: B)(f: (B, A) => B): B
```
Non-Parallelizable Operations

Let's examine the foldLeft signature:

```scala
def foldLeft[B](z: B)(f: (B, A) => B): B
```
Non-Parallelizable Operations

Let's examine the `foldLeft` signature:

```scala
def foldLeft[B](z: B)(f: (B, A) => B): B
```
Non-Parallelizable Operations

Let's examine the `foldLeft` signature:

```scala
def foldLeft[B](z: B)(f: (B, A) => B): B
```
Non-Parallelizable Operations

Let's examine the foldLeft signature:

```scala
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:

```python
def fold(z: A)(f: (A, A) => A): A
```
The fold Operation

Next, let's examine the fold signature:

```python
def fold(z: A)(f: (A, A) => A): A
```
Next, let’s examine the fold signature:

```python
def fold(z: A)(f: (A, A) => A): A
```
The **fold** Operation

Next, let’s examine the **fold** signature:

```python
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.
Use-cases of the fold Operation

Implement the sum method:

```scala
def sum(xs: Array[Int]): Int
```
Use-cases of the fold Operation

Implement the sum method:

```scala
def sum(xs: Array[Int]): Int = {
  xs.par.fold(0)(_ + _)
}
```
Use-cases of the fold Operation

Implement the sum method:

```scala
def sum(xs: Array[Int]): Int = {
  xs.par.fold(0)(_ + _)
}
```

Implement the max method:

```scala
def max(xs: Array[Int]): Int
```
**Use-cases of the fold Operation**

Implement the sum method:

```scala
def sum(xs: Array[Int]): Int = {
  xs.par.fold(0)(_ + _)
}
```

Implement the max method:

```scala
def max(xs: Array[Int]): Int = {
  xs.par.fold(Int.MinValue)(math.max)
}```
Preconditions of the fold Operation

Given a list of "paper", "rock" and "scissors" strings, find out who won:

Array("paper", "rock", "paper", "scissors")
Preconditions of the fold Operation

Given a list of "paper", "rock" and "scissors" strings, find out who won:

Array("paper", "rock", "paper", "scissors")
.par.fold(""")(play)

```scala
def play(a: String, b: String): String = List(a, b).sorted match {
  case List("paper", "scissors") => "scissors"
  case List("paper", "rock") => "paper"
  case List("rock", "scissors") => "rock"
  case List(a, b) if a == b => a
  case List("", b) => b
}
```
Preconditions of the fold Operation

Given a list of "paper", "rock" and "scissors" strings, find out who won:

Array("paper", "rock", "paper", "scissors")
.par.fold("")(play)
Preconditions of the fold Operation

Given a list of "paper", "rock" and "scissors" strings, find out who won:

Array("paper", "rock", "paper", "scissors")
    .par.fold(""")(play)

play(play("paper", "rock"), play("paper", "scissors")) == "scissors"
Preconditions of the fold Operation

Given a list of "paper", "rock" and "scissors" strings, find out who won:

Array("paper", "rock", "paper", "scissors")
  .par.fold(""")(play)

play(play("paper", "rock"), play("paper", "scissors")) == "scissors"

play("paper", play("rock", play("paper", "scissors"))) == "paper"

Why does this happen?
Preconditions of the fold Operation

Given a list of "paper", "rock" and "scissors" strings, find out who won:

Array("paper", "rock", "paper", "scissors")
  .par.fold(""")(play)

play(play("paper", "rock"), play("paper", "scissors")) == "scissors"

play("paper", play("rock", play("paper", "scissors"))) == "paper"

Why does this happen?

The play operator is *commutative*, but not *associative*. 
In order for the fold operation to work correctly, the following relations must hold:

\[
\begin{align*}
  f(a, f(b, c)) &= f(f(a, b), c) \\
  f(z, a) &= f(a, z) = a
\end{align*}
\]

We say that the neutral element \( z \) and the binary operator \( f \) must form a monoid.
In order for the fold operation to work correctly, the following relations must hold:

\[ f(a, f(b, c)) = f(f(a, b), c) \]
\[ f(z, a) = f(a, z) = a \]

We say that the neutral element \( z \) and the binary operator \( f \) must form a monoid.

Commutativity does not matter for fold – the following relation is not necessary:

\[ f(a, b) = f(b, a) \]
Limitations of the fold Operation

Given an array of characters, use fold to return the vowel count:
Limitations of the fold Operation

Given an array of characters, use fold to return the vowel count:

Array('E', 'P', 'F', 'L').par
  .fold(0)((count, c) => if (isVowel(c)) count + 1 else count)
**Limitations of the fold Operation**

Given an array of characters, use fold to return the vowel count:

```haskell
Array('E', 'P', 'F', 'L').par
  .fold(0)((count, c) => if (isVowel(c)) count + 1 else count)

// does not compile -- 0 is not a Char
```

The fold operation can only produce values of the same type as the collection that it is called on.
Limitations of the fold Operation

Given an array of characters, use fold to return the vowel count:

```
Array('E', 'P', 'F', 'L').par
  .fold(0)((count, c) => if (isVowel(c)) count + 1 else count)
```

// does not compile -- 0 is not a Char

The fold operation can only produce values of the same type as the collection that it is called on.

The foldLeft operation is *more expressive* than fold. Sanity check:

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

Implementing foldLeft using fold is not so straightforward.
The aggregate Operation

Let's examine the aggregate signature:

```python
def aggregate[B](z: B)(f: (B, A) => B, g: (B, B) => B): B
```
The aggregate Operation

Let's examine the aggregate signature:

```scala
def aggregate[B](z: B)(f: (B, A) => B, g: (B, B) => B): B
```

A combination of `foldLeft` and `fold`.
Count the number of vowels in a character array:
Using the aggregate Operation

Count the number of vowels in a character array:

Array('E', 'P', 'F', 'L').aggregate(0)(
  (count, c) => if (isVowel(c)) count + 1 else count,
  _ + _
)
Parallelizable Operations

Alternatively, consider the `mapFold` method. Can you implement `mapFold` using only `foldLeft`?

```scala
def mapFold[B](f: A => B)(z: B)(g: (B, B) => B): B
```
Parallelizable Operations

Consider the `mapFold` method. Can you implement `mapFold` using only `foldLeft`?

```scala
def mapFold[B](f: A => B)(z: B)(g: (B, B) => B): B = 
  foldLeft(z)((b, a) => g(b, f(a))
```

This particular implementation of `mapFold`, however, does not run in parallel.
Parallelizable Operations

For those who like challenges, implement:

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

using only the `mapFold` operation.

Can this method execute in parallel?
The Transformer Operations

So far, we saw the accessor combinators.

*Transformer* combinators, such as map, filter, flatMap and groupBy, do not return a single value, but instead return new collections as results.
The Transformer Operations

So far, we saw the accessor combinators.

Transformer combinators, such as map, filter, flatMap and groupBy, do not return a single value, but instead return new collections as results. They should be familiar from sequential programming.