@AndrzejWasowski
Andrzej Wąsowski

Advanced Programming

Purely Functional Parallelism [Par]

IT UNIVERSITY OF COPENHAGEN

SOFTWARE QUALITY RESEARCH

Parallel Data Structures (Motivation)

```scala
val list = (1 to 100).toList
val flist = list.map(f)
```

- If `f` slow but referentially transparent, parallelizing is beneficial!
- The parallel collections library for Scala allows: `list.par.map (f)`
- `list.par: collection.parallel.immutable.ParSeq[Int] =ParVector(1,...`
- More types supported: `ParArray`, `ParVector`, `mutable.ParHashMap`, `mutable.ParHashSet`, `immutable.ParHashMap`, `immutable.ParHashSet`, `ParRange`, `ParTrieMap`
- **Similarities** with `Par`: enable parallelism at the level of processing data structures without low level concurrency primitives (**parallel programming for the masses!**)
- **Differences** from `Par`: Scala's parallel collections are **eager**. We separate construction of the computation from execution. This gives more flexibility.
- Similar facilities exist in LINQ (C#) and in F#

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

Parallel Collections in Spark

- Spark has similar facilities:

```scala
val data =Array(1, 2, 3, 4, 5)
val distData =sc.parallelize(data)
```

- Constructs an `RDD` from a collection.
- RDD resembles a parallel collection, but it can **also** be distributed
- **RDD constructions are lazy**. As long as transformations are applied to an RDD, no computation is executed.
- Allows Spark schedulers to control the computation better
- This is more like `Par` than Scala's native parallel collections
- Today we look at a design of such general APIs

https://spark.apache.org/docs/latest/rdd-programming-guide.html#parallelized-collections

API for Functional Parallelism

What problem we are solving today?

The Problem:

- Design a general functional parallel programming API
- That allows to arbitrarily structure a parallel computation
- And then to execute it on a limited amount of resources (a bounded thread pool)

The Solution:

- **Separate** description of computation from execution
- Use **Continuation Passing Style** (CPS) to pass resources between threads without blocking
- Use of familiar combinators to **combine computations** (flatMap, map, map2, sequence, ...). The semantics of these combinators is much less trivial than for list, option, or state monads.

API for Functional Parallelism

How to read this chapter?

Chapter 7

- No right answers in design
- You will see a collection of design choices
- You are to understand their trade-offs, and think critically.
- The lecture explains the key design, but not all the meanders of the story — not the full learning experience!

Agenda for Today

- Motivation for Par [Done]
- Usage & Design of Par
- Continuation Passing Style (A general pattern)
- Implementation of Par
- Extension methods / Opaque types

map2 for Option and Par

```
def map2[A, B, C](oa: Option[A], ob: => Option[B])(f: (A, B) => C): Option[C] =
  oa.flatMap { a => ob.map { b => f(a, b) } }
```

- Why is oa by-value and ob by-name?
- Now a version for Par:

```
def map2[A, B, C](pa: => Par[A], pb: => Par[B])(f: (A, B) => C): Par[C]
```

- Why are both pa and pb by name?
- An example of use, parallel summation of list:

```
def sum(ints: IndexedSeq[Int]): Par[Int] =              // Returns immediately
  if ints.size <= 1 then
    Par.unit(ints.headOption.getOrElse(0))
  else
    val (l, r) = ints.splitAt(ints.length/2)
    Par.map2(Par.fork(sum(l)), Par.fork(sum(r)))(_ + _)  // After forking don't wait explicitly!
                                                         // We can apply _+_ to computations!
```

The Par Type

- `Par[A]`: a **pure data structure** describing a **parallel computation**
  - Some similarity to `Stream` which describes computations happening in sequence
  - `Par` is Java's `Callable` with a way better API
  - Allows expanding the computation, without waiting for results
  - Separates construction and parallelization of computation from scheduling and execution
  - First decide what runs in separate threads, then on what resource (Executor) to run it
- `unit[A](a: A): Par[A]` promote a constant to `Par` eagerly (trivial, return `a` immediately)
- `map2[A, B, C](pa: Par[A], pb: Par[B])(f: (A, B) => C): Par[C]` combines results of two computations with a binary function. Does not introduce new threads.
- `fork[A](a: => Par[A]): Par[A]` marks a computation for concurrent evaluation (separate thread). No evaluation until forced by `run`. Introduce a new thread.
- `lazyUnit[A](a: => A): Par[A]` wraps its unevaluated argument in a `Par` and marks it for concurrent evaluation (so it combines `unit` and `fork`).
- `run[A](es: ExecutorService)(p: Par[A]): A` extracts a value from a `Par` by actually performing the computation (the non-blocking version)

Other Combinators

- `def map[A, B](pa: Par[A])(f: A =>B): Par[B] =map2(pa, unit(())) { (a, _) =>f(a) }`
  - `map` **extends** the definition of a parallel computation with a **new step** (`f`)
  - Example: `def sortPar(parList: Par[List[Int]]) =map(parList) { _.sorted }`
  - Q. What does `sortPar` do?
  - It changes a parallel computation producing a list, into one whose resulting list is sorted. Nothing is run at this stage!
  - An example how we an build a computation without committing to where and how to execute
  - Choose to use a different number of threads or a different scheduling policy in different places
  - For example UI vs background batch processing
- `def asyncF[A, B](f: A =>B): A =>Par[B]` change `f` to **run in parallel** (exercise)
- `def sequence[A](ps: List[Par[A]]): Par[List[A]]`
  - **Recomposes** a list of parallel computations of `A` into a single parallel computation of a list
  - Example: schedule `n` downloads, get a list of downloaded files in parallel (exercise)
  - Familiar from `State`
- We can use it to implement `parMap` (that maps over list in parallel):

```
def parMap[A, B](as: List[A])(f: A =>B): Par[List[B]] =
  sequence(as.map(asyncF(f)))
```


Mentimeter

- `def map[A, B](pa: Par[A])(f: A =>B): Par[B] =map2(pa, unit(())) (a, _) =>f (a)`
- `def asyncF[A, B](f: A =>B): A =>Par[B]`
- `def sequence[A](ps: List[Par[A]) : Par[List[A]]`
- ```
  def parMap[A, B](as: List[A])(f: A =>B): Par[List[B]] =
    sequence(as.map(asyncF(f)))
  ```

Question. Why is `asyncF` called? What would happen if we did:

```
def parMap[A, B](as: List[A])(f: A =>B): Par[List[B]] =
  sequence(as.map(f))
```
```

Continuation Passing Style

```
def f(x: X): Y                          // Consider two functions
def g(y: Y): Z
// Normally this is how we compose them:
g(f(x)): Z                              // f (x) first returns, then we call ’g’ on the outcome

def f(x: X)(cont: Y => Unit): Unit =    // Rewrite f to call a *continuation* instead of returning
  val y_result = ...                    // the original body of f
  cont(y_result)                        // tail call, no new stack/thread use

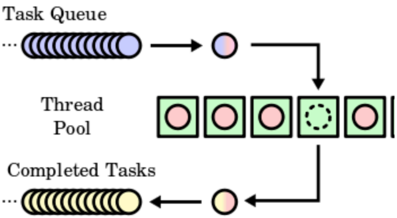
def g(y: Y)(cont: Z => Unit): Unit =    // Do the same to ’g;
  val z_result = ...                    // the original body of g
  cont(z_result)

// Let ’consumer: Z => Unit’ execute what we shall do with g’s result
f(x) { y => g(y)(consumer) }            // Compose ’f’ and ’g’ in the continuation passing style
```

- All returning via argument passing; `Unit` inessential, consumer could be pure and return value
- The last thing each function does is calling the next function
- A peculiar generalization of tail recursion and accumulator style
- Used to implement `Par` so that handing over from one to another can reuse same thread

Background: Java's ExecutorService and Future API

```
class ExecutorService {
  def submit[A](a: Callable[A]): Future[A]
}
trait Future[A] {
  def get: A
}
```



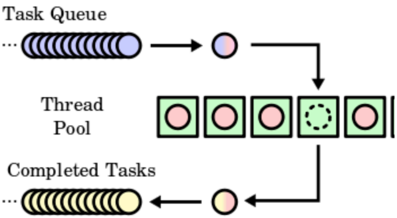
https://www.slideshare.net/afkham_azeez/java-colombo-developing-highly-scalable-apps

- An `ExecutorService` provides methods to **manage termination**, and
- Can produce a `Future` for **tracking progress of asynchronous tasks**

- A **thread pool** is a **blocking queue**
- A **worker thread** executes a task from the queue as long as non-empty

Typical use of an ExecutorService

```
class ExecutorService {
  def submit[A](a: Callable[A]): Future[A]
}
trait Future[A] {
  def get: A
}
```



Typical usage1

```
val service: ExecutorService =
Executors.newFixedThreadPool(2)
service.submit(t1)
service.submit(t2)
service.submit(t3)
```

The Implementation of Par [1/3]

The non-blocking CPS-based variant (Section 7.4.4)

```scala
// We use futures to represent an asynchronous calculation of a value.
// Java Futures don’t have a way to continue computation without waiting for A. Par has.
opaque type Future[+A] = (A => Unit) => Unit          // Java futures deadlock, new design
                                                      // The future calls the continuation when ready


opaque type Par[A] = ExecutorService => Future[A]     // Just an alias, using Java’s Executors
// opaque means it looks like a class outside this file (others cannot see it is a function)


// Normally we do not execute Par, but compose it with new calculations (using map,map2,chooser,etc.)
// Once we have a representation of the whole thing we can run it:
extension [A] (pa: Par[A])                                    // An extension method for Par[A] (only)
  def run[A](es: ExecutorService): A =
    val ref = java.util.concurrent.atomic.AtomicReference[A]() // Mutable threadsafe cell (local!)
    val latch = CountDownLatch(1)                             // Create a lock
    pa (es) { a => ref.set(a); latch.countDown }              // Continuation sets ref and unlocks
    latch.await                                               // Wait for unlock (never if p crashes)
    ref.get                                                   // Return ‘a‘ set by the continuation
```

The Implementation of Par [2/3]

The non-blocking CPS-based variant (Section 7.3.4)

```
def unit[A](a: A): Par[A] =                            // A strict unit
  es => k => k (a)

def eval(es: ExecutorService)(r: => Unit): Unit =     // A helper function
  es.submit(new Callable[Unit] { def call = r })      // Submit a unit computation to an executor

def fork[A](a: => Par[A]): Par[A] =                    // Marks 'a' for parallel execution
  es => k => eval(es)(a(es)(k))                        // Do not evaluate, but delay in a Future, and eval

def lazyUnit[A](a: => A): Par[A] =                     // A lazy (by-name) version of unit
  fork(unit(a))                                        // Mark 'a' for parallel, wrap in unit
                                                       // NB. fork is by-name
```

The Implementation of Par [3/3]

The non-blocking CPS-based variant (Section 7.4.4)

```
extension [A] (pa: Par[A])

  def map2[B, C](pb: Par[B])(f: (A, B) => C): Par[C] =
    es => k =>
      var ar: Option[A] = None
      var br: Option[B] = None
      val combiner = Actor[Either[A, B]] (es) {
        case Left (a) =>
          if br.isDefined then Par.eval(es)(k(f(a,br.get)))
          else ar = Some(a)
        case Right(b) =>
          if ar.isDefined then Par.eval(es)(k(f(ar.get,b)))
          else br = Some(b)
      }
      pa(es) { a => combiner ! Left (a)  }
      pb(es) { b => combiner ! Right (b) }
```

Other operators are derived
(see exercises)

Extension Methods (C# vs Scala)

```csharp
namespace ExtensionMethods {
  public static class MyExtensions {
    public static int WordCount(this String str)
    {
      return str.Split(
          new char[] { ' ', '.', '?' },
          StringSplitOptions.RemoveEmptyEntries
        ).Length;
    }
  }
}
using ExtensionMethods;
"Hello Extension Methods".WordCount();
```

```scala
extension (val str: String)
  def wordCount =
    str.split(" .?".toArray)
      .filter { ! _.isEmpty }
      .length



import wordCount // if not imported with package, vis
...
...
"Hello Extension Methods".wordCount
...
```

- **Extension methods** C#, F#, Xtend, Kotlin: define static methods, call like instance method
- That's why `String` in Scala has **more methods** than in Java, even though it is the **same class**!
- In fact, `split` is a method on `StringOps` not on `String` (see above)
- **Extensions work very well with opaque types**: limit an extension only to a named type

C# docs https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/extension-methods
Scala docs: https://docs.scala-lang.org/scala3/book/ca-extension-methods.html

Extension Methods

- A mechanism to **extend an existing library**
- When you **cannot change the source** code
- Add methods to classes **without recompiling** the source
- **Even to Java classes from 1995!**
- Add methods to classes at **call location**, not at class definition location
- Even **objects** produced by **old code** (factories) get the new methods
- When you **read** someone else's code you need to know that you have to search **not only for class methods** but also for extension methods
- In Scala, extensions are often placed in the `*Ops` classes, e.g. https://www.scala-lang.org/api/2.12.3/scala/collection/immutable/StringOps.html
- Warning: in older versions of Scala, implicts have been used to implement extensions. A more complex syntax and mechanism to explain. These are now deprecated. Beware Stack Overflow!
- **Exercises:** use this pattern to add methods to `Par` which is a function type alias! Not even explicitly a class!

source: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/extension-methods