



 @AndrzejWasowski

**Andrzej Wąsowski**

# Advanced Programming

---

## Purely Functional Parallelism [Par]

---

IT UNIVERSITY OF COPENHAGEN

**S** SOFTWARE  
**Q** QUALITY  
**R** RESEARCH

# Parallel Data Structures (Motivation)

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

- If  $f$  slow but referentially transparent, parallelizing is beneficial!
- In Scala standard library we can: `list.par.map (f)`
- `list.par: collection.parallel.immutable.ParSeq[Int] = ParVector(1, ...`
- Scala has `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#

# Parallel Collections in Spark

- Spark has similar facilities:

```
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

# 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 / Pimp Your Library Pattern (A general Pattern)

# map2 for Option and Par

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

```
1 def sum (ints: IndexedSeq[Int]): Par[Int] =                               // Returns immediately  
2   if (ints.size <= 1)  
3     Par.unit (ints.headOption.getOrElse (0))  
4   else {  
5     val (l,r) = ints.splitAt (ints.length/2)  
6     Par.map2 (Par.fork (sum (l)), Par.fork (sum(r))) (_+_ ) // After forking don't wait explicitly!  
7   }                                                                    // 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 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[B], pb: Par[C]) (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 can 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 of `As`
  - 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))
```



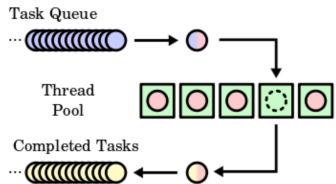
# Continuation Passing Style

```
1  def f (x: X): Y                                // Consider two functions
2  def g (y: Y): Z
3  // Normally this is how we compose them:
4  g (f (x)): Z                                    // f (x) first returns, then we call 'g' on the outcome
5
6  def f (x: X) (cont: Y => Unit): Unit = {         // Rewrite f to call a *continuation* instead of returning
7    val y_result = ... // the original body of f
8    cont (y_result)
9  }
10 def g (y: Y) (cont: Z => Unit): Unit = {         // Do the same to 'g';
11   val z_result = ... // the original body of g
12   cont (z_result)
13 }
14 // Let 'consumer: Z => Unit' execute what we shall do with g's result
15 f (x) (y => g (y) (z => consumer (z)))          // 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](https://www.slideshare.net/afkham_azeez/java-colombo-developing-highly-scalable-apps)

```
public interface ExecutorService  
extends Executor
```

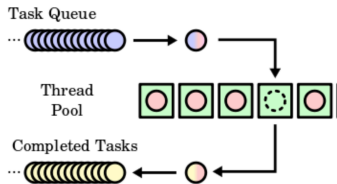
An **Executor** that provides methods to manage termination and methods that can produce a **Future** for tracking progress of one or more asynchronous tasks.

The thread pool execution uses a blocking queue. It keeps storing all the tasks that you have submitted to the **executor service**, and all threads (workers) are always running and performing the same steps:

- Take the Task from the queue
- Execute it
- Take the next or wait until a task will be added to the queue

# Typical usage of 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)
```

Typical usage2

```
val es=Executors.newWorkStealingPool()  
val fut=es.submit(new Callable[Int] {  
  override def call(): Int = (1 to 10).sum  
})  
val res=fut.get
```

# The Implementation of Par [1/3]

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

```
1 // We use futures to represent an asynchronous calculation of a value.
2
3 trait Future[+A] {                                     // Java futures deadlock, own design
4   private def apply (k: A => Unit): Unit              // The future calls 'k' when A ready
5 }
6
7 // Futures (also in Java) don't have a way to continue computation without waiting for A. Par has.
8
9 type Par[A] = ExecutorService => Future[A]            // Just an alias, using Java's Executors
10
11 // Normally we do not execute Par, but compose it with new calculations (using map,map2,chooser,etc.)
12 // Once we have a representation of the whole thing we can run it:
13
14 def run[A] (es: ExecutorService) (p: Par[A]): A = {
15   val ref = new java.util.concurrent.atomic.AtomicReference[A] // Mutable threadsafe cell (local!)
16   val latch = new CountDownLatch(1)                             // Create a lock
17   p (es) { a => ref.set(a); latch.countDown }                  // Continuation sets ref and unlocks
18   latch.await                                                    // Wait for unlock (never if p crashes)
19   ref.get                                                         // Return value of p set by continuation
20 }
```

# The Implementation of Par [2/3]

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

```
1 def unit[A] (a: A): Par[A] =                                // A strict unit
2   es => new Future[A] { def apply (cb: A => Unit) = cb(a) } // Return 'a' immediately to continuation
3
4 def eval (es: ExecutorService) (r: => Unit): Unit =         // A helper function
5   es.submit (new Callable[Unit] { def call = r })           // Submit a unit computation to an executor
6
7 def fork[A] (a: => Par[A]): Par[A] =                          // Marks 'a' for parallel execution
8   es => new Future[A] {                                     // Do not evaluate, but delay in a Future
9     def apply (cb: A => Unit) =                             // (like unit)
10       eval (es) (a (es) (cb))                             // Eval 'a' and 'cb' via 'es' (parallel)
11   }
12
13 def lazyUnit[A] (a: =>A) : Par[A] =                          // A lazy (by-name) version of unit
14   fork (unit (a))                                           // Mark 'a' for parallel, wrap in unit
15                                                         // NB. fork is by-name in the first arg.
```

# The Implementation of Par [3/3]

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

```
1 def map2[A,B,C] (p: Par[A], p2: Par[B]) (f: (A,B) => C): Par[C] =
2   es => new Future[C] {                                     // 1. Create a Par
3     def apply (cb: C => Unit): Unit = {
4       var ar: Option[A] = None                             // 2. A mutable cell to store result of p
5       var br: Option[B] = None                             // 3. A mutable cell to store result of p2
6       val combiner = Actor[Either[A,B]] (es) {             // 4. An actor to receive results from p,p2
7         case Left (a) =>                                    // 7. When p is done
8           if (br.isDefined)                                 // 8. If p2 has already finished (br is set)
9             eval (es) (cb (f (a, br.get)))                 // 9. Eval the callback with the merged result
10          else ar = Some (a)                                // 10. Otherwise store and finish
11                                                         // NB. No race, an actor handles 1 message at a time
12          case Right (b) =>                                  // 11. When p2 is done
13            if (ar.isDefined)                                // 12. If p has already finished (ar is set)
14              eval(es) (cb (f (ar.get, b)))                 // 13. Eval the callback with the merged result
15            else br = Some(b)                                // 14. Otherwise store and finish
16          }
17       p (es) (a => combiner ! Left (a))                     // 5. Spawn 'p', notify 'combiner' with result when done
18       p2 (es) (b => combiner ! Right (b))                  // 6. Spawn 'p2', notify 'combiner' with result on done
19     }                                                         // NB. If 'p' has a fork, then line 17 will exit
20   }                                                         // immediately and continue with 'p2' in parallel
```

**Other operators are derived  
(see exercises)**

# Extension Methods (C# vs Scala)

```
1 namespace ExtensionMethods {  
2     public static class MyExtensions {  
3         public static int WordCount(this String str)  
4         {  
5             return str.Split(  
6                 new char[] { ' ', '.', '?' },  
7                 StringSplitOptions.RemoveEmptyEntries  
8             ).Length;  
9         }  
10    }  
11 }  
12 using ExtensionMethods;  
13 "Hello Extension Methods".WordCount();
```

```
1 case class MyStringOps (val str: String) {  
2     def wordCount = str.split (" .?".toArray)  
3                             .filter { !_isEmpty }  
4                             .length  
5 }  
6  
7 implicit def myStringOps (s: String) =  
8     MyStringOps (s)  
9  
10 ...  
11 ...  
12 "Hello Extension Methods".wordCount  
13 ...
```

- **Extension methods** C#, F#, Xtend, Kotlin: define static methods, call like instance method
- **Pimp my library pattern** Scala: define an **auxiliary class** with **new methods and fields** and an **implicit conversion** to this class
- 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)



# Extension Methods and Pimp My Library Pattern

- Two mechanisms 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>
- **Exercises:** use this pattern to add methods to Par which is a function type alias! Not even an explicitly a class!