

Scala - Parallel Collections

Parallele Collection in Scala - Wie, Warum und
was bringt.

Andreas Neumann -
a.neumann@smarchive.de

Warum Parallel Collections

- Parallelisierung ist schwer
- Viele Probleme werden durch Fehler in der Implementierung von low-level Tasks (Locking usw.) verursacht
- Verwendung von Collections ist weithin bekannt
- --> **Collection als high Level Abstraction für Parallelisierung**

Verwendung paralleler Collections

- Funktionieren wie ihre sequentiellen Gegenstücke
- Austauschbar sequentiell / parallel durch geänderten Import
- Konvertierung möglich **.par <-> .seq**

```
scala> List("Yakko", "Wakko", "Dot")
res0: List[String] = List(Yakko, Wakko, Dot)

scala> List("Yakko", "Wakko", "Dot").par
res1: scala.collection.parallel.immutable.ParSeq[String] = ParVector(Yakko,
Wakko, Dot)

scala> List("Yakko", "Wakko", "Dot").par.seq
res2: scala.collection.immutable.Seq[String] = Vector(Yakko, Wakko, Dot)

scala> List("Yakko", "Wakko", "Dot").par map(_.toLowerCase)
res7: scala.collection.parallel.immutable.ParSeq[String] = ParVector(yakko,
wakko, dot)
```

Vorhandene parallele Collections

- ParArray
- ParVector
- mutable.ParHashMap
- mutable.ParHashSet
- immutable.ParHashMap
- immutable.ParHashSet
- ParRange
- ParTrieMap
(collection.concurrent.TrieMaps seit Scala 2.10)

prinzipielle Funktionsweise

- **Splitters:** Collection in Teile für parallele Abarbeitung zerlegen

```
trait Splitter[T] extends Iterator[T] {  
    def split: Seq[Splitter[T]]  
}
```

- **Combiners:** Ergebnisse der parallelen Abarbeitung wieder kombinieren

```
trait Combiner[Elem, To] extends Builder[Elem, To] {  
    def combine(other: Combiner[Elem, To]):  
        Combiner[Elem, To]  
}
```

Talk 1 -

Aleksandar Prokopec: Scala Parallel Collections, Scaladays 2010

- Allgemeiner Überblick über verwendete Verfahren und Implementierung der parallelen Standardcollections
- <http://days2010.scala-lang.org/node/138/140>

Fallstricke 1 : Seiteneffekte

- Operationen mit Seiteneffekten

```
var sum = 0
val list = (1 to 1000).toList.parallel
list.foreach(sum += _); sum
sum = 0
list.foreach(sum += _); sum
sum = 0
list.foreach(sum += _); sum
```

```
scala> list.foreach(sum += _); sum
res21: Int = 360849
scala> sum = 0
sum: Int = 0
scala> list.foreach(sum += _); sum
res22: Int = 364636
scala> sum = 0
sum: Int = 0
scala> list.foreach(sum += _); sum
res23: Int = 325810
```

```
scala> list.foreach(sum += _); sum
res0: Int = 254374
scala> sum = 0
sum: Int = 0
scala> list.foreach(sum += _); sum
res1: Int = 365032
scala> sum = 0
sum: Int = 0
scala> list.foreach(sum += _); sum
res2: Int = 398012
```

Fallstricke 2: nicht assoziative Operationen

- assoziativ : Reihenfolge der Ausführung spielt keine Rolle
- Anders gesagt: Die Klammerung mehrerer assoziativer Verknüpfungen ist beliebig
- Assoziativ: $+$, $*$, $++$
- nicht assoziativ: $-$, $/$, $--$

Fallstricke 2: nicht assoziative Operationen Beispiel

```
val list = (1 to 1000).toList.par  
list.reduce(_-_)  
list.reduce(_-_)  
list.reduce(_-_)  
...  
...
```

```
scala> list.reduce(_-_)  
res6: Int = -43900  
  
scala> list.reduce(_-_)  
res7: Int = 80260  
  
scala> list.reduce(_-_)  
res8: Int = -92480  
  
scala> list.reduce(_-_)  
res9: Int = 27210  
  
scala> list.reduce(_-_)  
res10: Int = -101188  
  
scala> list.reduce(_-_)  
res11: Int = 66440
```

Fehlannahmen: nicht kommunitative Operationen

- Nicht kommunitative Operationen funktionieren problemlos, weil die Reihenfolge erhalten bleibt

```
List("Yakko", "Wakko", "Dot", "Dr. Otto Scratchansniff", "Hello Nurse")
```

```
List("Yakko", "Wakko", "Dot", "Dr. Otto Scratchansniff", "Hello Nurse").  
par.reduce(_ + " <-> " + _)
```

```
res16: String = Yakko <-> Wakko <-> Dot <-> Dr. Otto Scratchansniff <-> Hello Nurse
```

Fehlannahmen: nicht kommutative Operationen 2

```
scala> (1 to 500).sliding(3,3).toIndexedSeq
```

```
res35: scala.collection.immutable.IndexedSeq[scala.collection.immutable.IndexedSeq[Int]] = Vector(Vector(1, 2, 3), Vector(4, 5, 6), Vector(7, 8, 9), Vector(10, 11, 12), Vector(13, 14, 15), Vector(16, 17, 18), Vector(19, 20, 21), Vector(22, 23, 24), Vector(25, 26, 27), Vector(28, 29, 30), Vector(31, 32, 33), Vector(34, 35, 36), Vector(37, 38, 39), Vector(40, 41, 42), Vector(43, 44, 45), Vector(46, 47, 48), Vector(49, 50, 51), Vector(52, 53, 54), Vector(55, 56, 57), Vector(58, 59, 60), Vector(61, 62, 63), Vector(64, 65, 66), Vector(67, 68, 69), Vector(70, 71, 72), Vector(73, 74, 75), Vector(76, 77, 78), Vector(79, 80, 81), Vector(82, 83, 84), Vector(85, 86, 87), Vector(88, 89, 90), Vector(91, 92, 93), Vector(94, 95, 96), Vector(97, 98, 99), Vector(100, 101, 102), Vector(103, 104, 105), Ve...
```

```
(1 to 500).sliding(3,3).toIndexedSeq.par.reduce(_ ++ _)
```

```
res36: scala.collection.immutable.IndexedSeq[Int] = Vector(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 1...
```

Konvertierungen

- Es ist abhängig von der Ausgangs- und Zielcollection wie teuer die Konvertierungen von/ bzw. zu einer parallelen Collection ist
- Bei rein Sequentiellen Collections, z.B. **List** muss jedes Element kopiert werden
- Sehr gut geeignet ist der **Vector** als Standardimplementierung für **IndexedSeq**

Threadpool konfigurieren - Ab Scala 2.10

- Konfiguration des verwendeten Threadpools möglich

```
scala> val numbers = (1 to 1000).par
numbers: scala.collection.parallel.immutable.ParRange = ParRange(1, 2, 3, ... )

scala> numbers.tasksupport =
  new ForkJoinTaskSupport(
    new scala.concurrent.forkjoin.ForkJoinPool( 3 )
  )
numbers.tasksupport: scala.collection.parallel.TaskSupport =
scala.collection.parallel.ForkJoinTaskSupport@2b45f918

scala> numbers filter ( i => i % 3 == 0 && i % 4 == 0)
res1: scala.collection.parallel.immutable.ParSeq[Int] = ParVector(12, 24, 36, 48, 60, 72,
84, 96, 108, 120, 132, 144, 156, 168, 180, 192, 204, 216, 228, 240, 252, 264, 276, 288,
300, 312, 324, 336, 348, 360, 372, 384, 396, 408, 420, 432, 444, 456, 468, 480, 492, 504,
516, 528, 540, 552, 564, 576, 588, 600, 612, 624, 636, 648, 660, 672, 684, 696, 708, 720,
732, 744, 756, 768, 780, 792, 804, 816, 828, 840, 852, 864, 876, 888, 900, 912, 924, 936,
948, 960, 972, 984, 996)
```

Talk 2 - Aleksandar Prokopec: Scala Parallel Collections, Scaladays 2011

- Überblick über neuer Collections: parallel HashMaps and HashTries
- Anwendungsbeispiel: WordCount
- <http://days2011.scala-lang.org/node/138/272>

Kennziffern, ab wann es sich lohnt

- keine Allgemeingültige Aussage möglich
- Abhängig von:
 - *Prozessorarchitektur / Anzahl*
 - *Speicher*
 - *JVM - Implementierung*
 - *Workload pro Element*
 - *Verwendete Collection*
 - *Split / Combine Performance in Collection*
 - *Memory Management*

<http://docs.scala-lang.org/overviews/parallel-collections/performance.html>

Ausblick

- Scala Futures / Promises
- Actors

Quellen

- Talk 1: <http://days2010.scala-lang.org/node/138/140>
- Talk 2: <http://days2011.scala-lang.org/node/138/272>
- Scala-Doku: <http://docs.scala-lang.org/overviews/parallel-collections/overview.html>
- Paper: <http://infoscience.epfl.ch/record/150220/files/pc.pdf>