Aalto University
School of Science

Scala Concurrency and Parallel Collections

CS-E4110 Concurrent Programming

Keijo Heljanko

Department of Computer Science
Aalto University School of Science

November 23rd, 2016

Slides by Keijo Heljanko

Scala

Scala

- Originally an Acronym Scala - Scalable Language
- A general purpose programming language
- Implemented on top of the Java Virtual Machine (JVM)
- Object oriented
- Functional programming can be done in Scala, prefers the use of immutable data
- Also imperative programs with mutable data can be coded in Scala
- Can use all Java libraries
- The Akka Actor model is integrated into Scala
- Has also other interesting parallel features - **Scala Parallel Collections**

Learning Scala

Learning Scala

- In this course we will not teach the Scala basics, just the concurrency features
- For learning Scala, see the course: ICS-A1120 Ohjelmointi 2
- We will not go deep into Scala, examples should be understandable with Java programming background
- Getting started with Scala: `http://www.scala-lang.org/documentation/getting-started.html`
- Main Scala tutorials are at: `http://docs.scala-lang.org/tutorials/`
- Scala Tutorial for Java programmers: `http://docs.scala-lang.org/tutorials/scala-for-java-programmers.html`

Scala Parallel Collections Tutorial

Parallel Collections Tutorial

- New feature of Scala since version 2.9 (version 2.11 is current the most recent)
- We are using materials from the Scala Parallel Collections tutorial at: `http://docs.scala-lang.org/overviews/parallel-collections/overview.html`
- For design of the internals, see:
  Aleksandar Prokopec, Phil Bagwell, Tiark Rompf, Martin Odersky: A Generic Parallel Collection Framework. Euro-Par (2) 2011: 136-147

Scala Collections

Scala Collections

- Consider the following piece of Scala code using collections:

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

- The code adds 42 to each member of the collection, using a single thread to do so
- When run through the Scala interpreter, we get:

```
scala> val list = (1 to 10000).toList
list: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, ...
scala> list.map(_ + 42)
res0: List[Int] = List(43, 44, 45, 46, 47, 48, 49, 50, 51, 52, ...
```

Scala Parallel Collections

Scala Parallel Collections

- To make this code parallel, we can just use the `par` method on the list to generate a ParVector, a parallel vector datatype

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

- The code adds 42 to each member of the collection, using several threads running in parallel, we get:

```
scala> val list = (1 to 10000).toList
list: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, ...
scala> list.par.map(_ + 42)
res0: scala.collection.parallel.immutable.ParSeq[Int] =
ParVector(43, 44, 45, 46, 47, 48, 49, 50, 51, 52, ...
```

Scala Parallel Collections

Scala Parallel Collections

- One can generate parallel collection types from sequential collections
- Operations on parallel collection types can use all the threads available to the Scala runtime to process the collections in parallel
- The load balancing and scheduling of the parallel collections is done by the Scala runtime
- Due to the overhead of creating new threads, better performance is only obtained for operations which are CPU heavy per item in the collection, or when the collections are quite large
- The parallelization uses the functional programming nature Scala collections - The map operations performed should not have side effects (if possible)

Available Parallel Collections

Available Parallel Collections

- ParArray
- ParVector
- mutable.ParHashMap
- mutable.ParHashSet
- immutable.ParHashMap
- immutable.ParHashSet
- ParRange
- ParTrieMap

Example: Using Parallel Map

Parallel Map

- Consider the following piece of Scala code using a parallel map:

```
val lastNames = List("Smith","Jones","Frankenstein","Bach","Jackson","Rodin").par
lastNames.map(_.toUpperCase)
```

Example: Using Parallel Map (cnt.)

Parallel Map (cnt.)

- The code converts all elements of the map in parallel to upper case

```
scala> val lastNames = List("Smith","Jones","Frankenstein",
"Bach","Jackson","Rodin").par
lastNames: scala.collection.parallel.immutable.ParSeq[String] =
ParVector(Smith, Jones, Frankenstein, Bach, Jackson, Rodin)

scala> lastNames.map(_.toUpperCase)
res0: scala.collection.parallel.immutable.ParSeq[String] =
ParVector(SMITH, JONES, FRANKENSTEIN, BACH, JACKSON, RODIN)
```

Example: Using Fold

Parallel Fold

- Consider the following piece of Scala code summing up all integers in a list using `fold`, which applies an associative operation to all elements of the collection:

```scala
val parArray = (1 to 1000000).toArray.par
parArray.fold(0)(_ + _)
```

Example: Using Fold (cnt.)
Parallel Fold

- The output of the operation is well defined as the addition method given to `fold` is an associative operation, and the parameter of fold `0` is the zero element of addition

```
scala> val parArray = (1 to 1000000).toArray.par
parArray: scala.collection.parallel.mutable.ParArray[Int] =
ParArray(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, ...
scala> parArray.fold(0)(_ + _)
res0: Int = 1784293664
```

- Note: In Scala the operation passed to `fold` does not have to be commutative, only associative!
- Note: Many other frameworks, such as Apache Spark require also operator commutativity, be careful when porting from Scala parallel collections to Spark!

Example: Using Reduce

Parallel Reduce

- The `reduce` operation is like `fold` except that because you do not give the zero element, it can not be applied to empty collections (it will throw an exception in that case). As fold, it also requires the applied operation to be associative:

```scala
val parArray = (1 to 1000000).toArray.par
parArray.reduce(_ + _)
```

```
scala> val parArray = (1 to 1000000).toArray.par
parArray: scala.collection.parallel.mutable.ParArray[Int] =
ParArray(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, ...
scala> parArray.reduce(_ + _)
res0: Int = 1784293664
```

Example: Using Filter

Parallel Filter

- Consider the following piece of Scala code filtering last names starting with letters larger than 'J':

```
val lastNames = List("Smith","Jones","Frankenstein","Bach","Jackson","Rodin").par
lastNames.filter(_.head >= 'J')
```

Example: Using Filter (cnt.)

Parallel Filter

- Notice that in Scala the filtered collection still preserves order of the original collection:

```
scala> val lastNames = List("Smith","Jones","Frankenstein",
"Bach","Jackson","Rodin").par
lastNames: scala.collection.parallel.immutable.ParSeq[String] =
ParVector(Smith, Jones, Frankenstein, Bach, Jackson, Rodin)

scala> lastNames.filter(_.head >= 'J')
res0: scala.collection.parallel.immutable.ParSeq[String] =
ParVector(Smith, Jones, Jackson, Rodin)
```

Creating Parallel Collections

Creating Parallel Collections

- By using the `new` keyword after importing the right package

```
import scala.collection.parallel.immutable.ParVector
val pv = new ParVector[Int]
```

```
scala> import scala.collection.parallel.immutable.ParVector
import scala.collection.parallel.immutable.ParVector

scala> val pv = new ParVector[Int]
pv: scala.collection.parallel.immutable.ParVector[Int] =
ParVector()
```

Creating Parallel Collections (cnt.)

Creating Parallel Collections

- By constructing a parallel collection from an existing sequential collection using the `par` method of the sequential collection:

```scala
val pv = Vector(1,2,3,4,5,6,7,8,9).par
```

```
scala> val pv = Vector(1,2,3,4,5,6,7,8,9).par
pv: scala.collection.parallel.immutable.ParVector[Int] =
ParVector(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Semantics of Parallel Collections

Semantics

- Code with side-effects will result in non-deterministic behaviour. Proper locking needs to be taken if operations on parallel collections manipulate shared state
- Using operations that are not associative will result in non-deterministic behaviour as evaluation order is based on scheduling of concurrently executing threads

Buggy! Summation using Side-effects

Buggy code due to side effects!

- The following code uses the variable sum in a racy manner, the outcome of the code depends on the interleaving:

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

var sum = 0;
list.foreach(sum += _); sum

var sum = 0;
list.foreach(sum += _); sum

var sum = 0;
list.foreach(sum += _); sum
```

Buggy! Summation using Side-effects (cnt.)

Different results on different runs!

```
scala> val list = (1 to 1000).toList.par
list: scala.collection.parallel.immutable.ParSeq[Int] =
ParVector(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, ...
scala> var sum=0
sum: Int = 0

scala> list.foreach(sum += _); sum
res0: Int = 481682

scala> var sum=0
sum: Int = 0

scala> list.foreach(sum += _); sum
res1: Int = 486426

scala> var sum=0
sum: Int = 0

scala> list.foreach(sum += _); sum
res2: Int = 500500
```

Buggy! Code due to Non-Associativity

Non-Associative Operations are Non-Deterministic

- The substraction operator is not associative (e.g, $(1-2)-3 \neq 1-(2-3)$). Thus the order of scheduling of operations affects the outcome of the reduce:

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

Buggy! Code due to Non-Associativity (cnt.)

Different results on different runs!

```
scala> val list = (1 to 1000).toList.par
list: scala.collection.parallel.immutable.ParSeq[Int] =
ParVector(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, ...
scala> list.reduce(_-_)
res0: Int = 169316

scala> list.reduce(_-_)
res1: Int = 497564

scala> list.reduce(_-_)
res2: Int = -331818
```

Correct Associative but Non-Commutative Operators

In Scala Parallel Collections Commutativity is not needed!

- The following code is correct in Scala parallel collections, as String concatenation is associative (even if it is not commutative!)

```
val strings = List("abc","def","ghi","jk","lmnop","qrs","tuv","wx","yz").par
val alphabet = strings.reduce(_++_)
```

Correct Associative but Non-Commutative Operators

In Scala Parallel Collections Commutativity is not needed!

- The outcome is the same regardless of thread scheduling:

```
scala> val strings = List("abc","def","ghi","jk",
"lmnop","qrs","tuv","wx","yz").par
strings: scala.collection.parallel.immutable.ParSeq[String] =
ParVector(abc, def, ghi, jk, lmnop, qrs, tuv, wx, yz)

scala> val alphabet = strings.reduce(_++_)
alphabet: String = abcdefghijklmnopqrstuvwxyz
```

- Note: Other frameworks, such as Apache Spark, require the operator applied by reduce to also be commutative, so this code would be incorrect in Spark!

Architecture of Parallel Collections

Architecture is based on two core abstractions:

- Splitter: A way to split a parallel collection to disjoint subparts that can be operated on in parallel
- Combiner: A way to combine the results of subtasks done in parallel into a final output

Architecture of Parallel Collections (cnt.)

Splitters:

- The job of a splitter is to split the parallel array into a non-trivial partition of its elements, until the partitions are small enough to be operated on sequentially

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

- As splitters inherit iterators, they have methods such as `next` and `hasNext`
- Splitters are in the end used to iterate over the elements of the collection
- For each parallel collection class, the splitter tries to split the collection into subsets of roughly the same size

Architecture of Parallel Collections (cnt.)

Combiners:

- Combiners are based on Builder from the Scala sequential collections library

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

- The `combine` method of a Combiner takes another parallel collection as a parameter, and returns a parallel collection which has the union of the elements of the two parallel collections
- Each parallel collection class needs to implement its own Combiner

Work Stealing

Parallel collections use Work Stealing:

- Using Splitters allows one to divide-and-conquer the work of traversing the elements of a parallel collection
- Each worker thread maintains its own work queue
- The full collection is pushed to the head of the queue of one of the threads as the initial split
- If there is work in the queue: A split is popped from the head of the work queue
  - If the popped split is smaller than a threshold, items on it are operated on sequentially to produce a subresult
  - Otherwise, the split is divided into subsplits, and subsplits are pushed to the head of the work queue one at a time

Work Stealing (cnt.)

Parallel collections use Work Stealing:

- If a thread has an empty work queue, it tries to “steal work” (remove a split) from another thread with a non-empty work queue
- Syncronization is needed to guarantee that all threads work on disjoint parts of the collection
- To maximize the size of the stolen work (and thus minimize overhead of the costly work stealing synchronizations), the stealing is done from the tail of the work queue, which contains one of the largest splits to be still worked on
- When all threads have emptied their work queues, the processing has ended for the splitting
- After this the computed subresults have to still be combined together into a single parallel operation to generate the output parallel collection

Work Stealing

Work Stealing

- Work stealing is one of the most efficient dynamic load balancing techniques for divide-and-conquer type workloads
- For further info, see:
  `https://en.wikipedia.org/wiki/Work_stealing`
- For an example, see the Cilk programming framework
- A very efficient commercial framework for C / C++ is Intel Cilk plus:
  `https://software.intel.com/en-us/intel-cilk-plus`