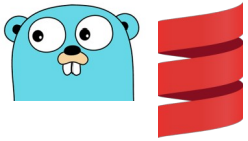
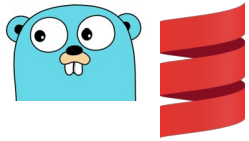


A Comparison of Parallel Programming in Go And Scala

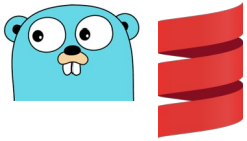
Marinus Gläßer

Outline



- What is Parallel Programming?
- Parallel Programming in Go
- Parallel Programming in Scala
- Summary and comparison

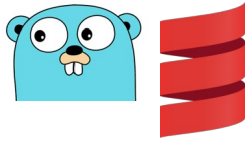
Importance of Parallel Programming



- CPUs don’t get (much) faster, but have more cores instead

- less then 1% of all Steam users have just 1 core in their CPU
- Octa-core CPU in almost every flagship smartphone of 2018

What is Parallel Programming?



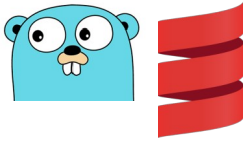
- Multiple tasks are executed on multiple CPU cores at *the same time*.

- Main Difference from Concurrent Programming:
  → Is for better responsiveness of an application
  → Tasks **must not** start at the same time
  → Concurrency enables parallelism

- True Parallelism:
  → Multiple execution starts at the **same time**
  → Mostly concerns about speed in algorithmic problems or Big Data processing
  → If you have only one processor, your program can still be concurrent but it cannot be parallel.

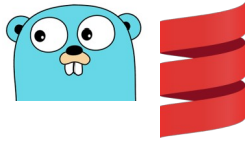
- Not every problem can be solved in parallel
  (Reading a text vs. counting the words)

Parallel Programming in Go



- Supports concurrency
- Automatic garbage collection
- The *GOMAXPROCS* environment variable defines the maximum number of OS threads goroutines may execute
  → the number of available CPU cores determine the default value.

Goroutines

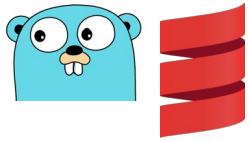


- Goroutines are like lightweight threads, but they are extremely cheap in terms of resources compared to threads
- Easy to use: special keyword “go”

```
package main

import (
  "fmt"
  "time"
)

func sayHello() {
   fmt.Println("Hello from a goroutine")
}

func main() {
  go sayHello()
  time.Sleep(1 * time.Millisecond)
  fmt.Println("main()-function call")
}
```

- main()-function is in it’s own goroutine, called “main goroutine”

Communication with channels

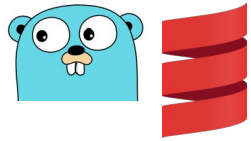


- Goroutines share the same address space, so their access to shared memory must be synchronized
  → channels provide this functionality

```
package main

import "fmt"

func sayHello(ch chan bool) {
  fmt.Println("Hello from a goroutine")
  ch <- true
}

func main() {
  ch := make(chan bool)
  go sayHello(ch)
  <-ch
  fmt.Println("main()-function call")
}
```

- Channels block until they receive the value

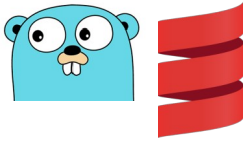
Closing and buffering channels



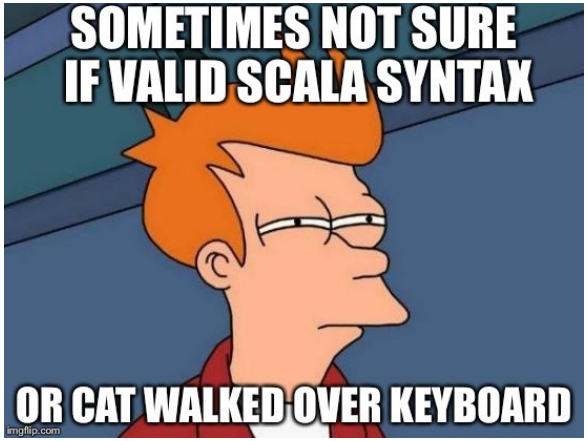
- Channels are unbuffered by default
- Closing tells the receiver, that there are no more values coming

```
package main

import "fmt"

func main() {
  ch := make(chan string, 2)
  ch <- "Hello"
  ch <- "World"
  hello := <-ch
  fmt.Println(hello)
  fmt.Println(<-ch)

  close(ch)
  _, ok := <-ch
  fmt.Println(ok)
}
```

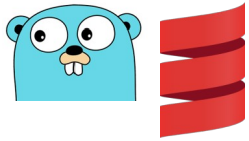
Parallel Programming in Scala



- Supports concurrency
- Automatic garbage collection
- Runs on the Java Virtual Machine
- A bit strange syntax



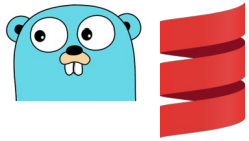
Threads in Scala



- Classic form of concurrency
- Scala can use any Java library

```
package de.thro.sinfmaglae

import com.typesafe.scalalogging.LazyLogging

object Threads extends App with LazyLogging {

  val thread1 = new Thread(() =>
    logger.debug(s"Hello World from ${Thread.currentThread()}")
  )
  thread1.start()

  val thread2 = new Thread(new Runnable {
    override def run(): Unit = {
      logger.debug(s"Hello World from ${Thread.currentThread()}")
    }
  })
  thread2.start()

  logger.debug("Hello from Main-Thread")
}
```

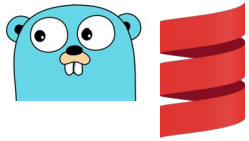
Parallel Collections Library



- Provides parallelism to Lists, Maps etc. just by adding par
- As usual in Scala, does not change element
  → returning a new parallel implementation of it instead

```
package de.thro.sinfmaglae

object ParallelCollections extends App {

  val list = List(1, 2, 3, 4, 5)
  list.foreach(print)
  println()
  list.par.foreach(print)

  val zipCodes = Map("83059" -> "Kolbermoor" /*, ...*/).par
  zipCodes.filterKeys(_.startsWith("83")).foreach(println)

}
```

Parallelism is easy in Scala, but...



- premature optimization is the root of all evil

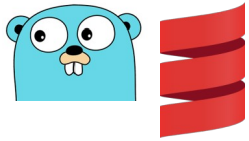
```
package de.thro.sinfmaglae

object ParallelCollectionsBenchmark extends App {

  var time = System.currentTimeMillis()
  val list = (1 to 250000).toList
  list.sum
  println(s"Time taken: ${System.currentTimeMillis() - time}")
  time = System.currentTimeMillis()
  list.par.sum
  println(s"Time taken: ${System.currentTimeMillis() - time}")
}
```

- Takes about 150ms each for my machine with 4 cores
  (Intel Core i5-2500k @ 3.3 GHz)

- 250.000 approx. the size when parallel and non-parallel calculations take the same amount of time
  (more elements: parallel execution is faster and vice versa with less elements)

- Always carefully select if parallelism is worth the overhead – depends on the problem's size!

Summary



- Go code compiles to into a single native binary
  → high performance system applications
- Scala runs on the JVM and can use the existing libraries
- Scala has Parallel Collections library and Go easy access to Goroutines
- Scala is easy for Java Developers to learn and to integrate in existing Java programs

Go for small high performance system applications and Scala for enterprise multi-platform applications