A Comparison of Parallel Programming in Go And Scala

Seminar Paper

Marinus Gläßer

02.01.2019 (WS 18/19)

Table of Contents

1 Importance of parallelizing programs

To parallel source code gets more and more important. The limit of what is possible with one traditional, serial processing CPU has been reached. For instance, almost every (over 99% (!)) user of the gaming platform “Steam” has at least two CPU cores in their computer. *Over 67% have four or more CPU cores* [1]. So as you can see from this simple statistic from August 2018, source code of software must be parallelized, otherwise it would not take any speed advantage from all the cores a modern CPU has.

There are multiple programming languages to code programs, which are then executed in parallel. With the focus on parallel programming Go and Scala will be compared with each other in this article. The similarities of both languages, the differences and the pros and cons of each language will be revealed and compared with each other. After reading this article the reader should have a feeling if weather Go or Scala is more suitable for his project with parallel programming involved. This article will not cover all possible features for parallel programming but rather the important aspects and features of both languages.

2 Parallel programming explained

But first of all let’s explain what is meant by parallel programming and what is the difference between concurrent programming which sounds quite similar. The traditional software programming schema is based on a single processor. Every instruction is executed after another. It’s like reading a text. This is a serial process where every word is processed one after another.

Counting the words can be executed in parallel. You have n number of persons available you would split the text in n parts. So each person just counts the number of the words in its part simultaneously. In the end you sum up the word counts from every person to get the final result.

Listing 1 shows a pseudo code example. For the sake of brevity only the important parts are in the code.

Listing 1. Parallel word count in pseudocode

```
# array to store textparts in
textparts = text.split(n) ①

# array to store word count result
wordCounts[n] ②

# parallel word count
Each processor i performs:
    wordCounts[i] = calculateWordCount(textparts[i]) ③

# sum all the wordcounts to get the final result
wordCounts.sum() ④
```

① First of all the text needs to be split in n (number of available processors) equal parts.

② To store the result of each word count we need an array of n fields.

③ The actual parallel part is where every processor calculates the number of words in its text part and stores the result in a field in the “wordCounts”-variable. The variable “i” is the index of the processor.

④ In the end the results from all the text parts have to be added together to get the final result.

The disadvantage of parallelizing is that you get some overhead. The text needs to be split in n parts and in the end all the word counts must be added together to get the final result. It is therefore more challenging for the programmer because he has to add extra code and has to take care that there are no race conditions or deadlocks. So it must always be carefully decided if the problem is big enough to be processed in parallel. If the problem is too small it might take even longer then processing it with just one task.

The main difference from concurrent programming is that *parallel programming mostly concerns about speed in algorithmic problems or Big Data processing*, whereas concurrent programming is for better responsiveness of an application. In concurrent programming multiple execution must not start at the same time. With just one CPU core it is still possible to have concurrency, because the concurrent tasks never need to run at the same time. Parallel programming is when *multiple tasks are executed on multiple CPU cores at the same time*.

3 Parallel programming in Go

Go is a free and open source programming language from Google which supports parallel programming. It is a compiled language which supports concurrency and has an automatic garbage collection. In this chapter the relevant aspects of the language for parallel programming are shown and described. The binary distributions can be downloaded for various system architectures and operating systems under https://golang.org/. A brief introduction to Go is also available online https://tour.golang.org/welcome/1.

3.1 Goroutines instead of threads in Go

Instead of threads there are “goroutines” in Go. They are extremely cheap in terms of resources compared to threads. Because they share the same address space, their access to shared memory must be synchronized [2]. Channels are a safe way for goroutines to communicate. By design channels prevent race conditions in Go. The `GOMAXPROCS` environment variable defines the maximum number of OS threads goroutines may execute [3]. It can be defined at compile time. Starting from Go 1.5 the *number of available CPU cores determine the default value*. Concurrency is achieved by using goroutines, but true parallel execution can only be done if there is more than one processor core.

They are started automatically with the simple keyword `go`. Listing 2 shows an example of how to use goroutines.

Listing 2. goroutines.go

```
1 package main ①
2
3 import "fmt"
4
5 func sayHello() {
6     fmt.Println("Hello from a goroutine")
7 }
8
9 func main() { ②
10     go sayHello() ③
11     fmt.Println("main()-function call") ④
12 }
```

① Every Go source file is in a package. The last element of the import path (folder) is the package name. The `package main` statement is special, because it says that this program is an executable command.

② The main()-function is the entry point of every go program.

③ This line starts a goroutine calling the sayHello()-function with the `go` keyword.

④ Program ends here. It does *not* wait for the other goroutine to terminate.

This program only prints “main()-function call”. Although the goroutine had been started and the sayHello()-function was executed parallel to the main()-function. The main()-function is also by default in its own goroutine called “main goroutine”, but when the main goroutine terminates the program will end all other goroutines, too. So there is no time left to execute the sayHello()-function.

By adding just a bit of waiting time the output from the goroutine with the sayHello()-function will be printed to the screen. Listing 3 shows the result:

Listing 3. goroutines2.go

```
1 package main
2
3 import ( ①
4     "fmt"
5     "time"
6 )
7
8 func sayHello() {
9     fmt.Println("Hello from a goroutine")
10 }
11
12 func main() {
13     go sayHello()
14     time.Sleep(1 * time.Millisecond) ②
15     fmt.Println("main()-function call")
16 }
```

① Instead of writing import in front of every package this is a very concise syntax to archive the same result. Writing `import` before every package works, too, but this is Go’s preferred way of importing multiple packages.

② After the statement with the goroutine some waiting time was added to the program with the Sleep()-function from the `time` package. Now the string from the sayHello()-function will be printed to the screen.

3.2 Communication with channels in goroutines

Goroutines use channels for communication. They connect goroutines, so they can communicate and must be created before use. To create a channel to store an integer value use `ch := make(chan int)` (make is a build-in function that creates an object). This creates a channel for an integer value and assigns it to variable `ch`. Channels block until the receiver or sender is ready, so deadlocks can be avoided even without explicit locks [4].

Listing 4 shows an example to archive the same result as listing 3 with channels.

Listing 4. channels.go

```
1 package main
2
3 import "fmt"
4
5 func sayHello(ch chan bool) {
6     fmt.Println("Hello from a goroutine")
7     ch <- true ③
8 }
9
10 func main() {
11     ch := make(chan bool) ①
12     go sayHello(ch) ②
13     <-ch ④
14     fmt.Println("main()-function call")
15 }
```

① A channel variable is created with the build-in function make. It can store a boolean value.

② This statement calls the sayHello-goroutine passing the channel variable `ch` as an argument to the function.

③ The boolean value `true` is written to the channel. It does not matter here if it is true or false because we do not do anything with the value except reading it.

④ The main goroutine is blocked until it receives the value from the channel `ch`. So “Hello from a goroutine” is printed to the screen, before the program exits. We have now the same result as in listing 3, but without the arbitrary waiting time.

If the statement in 3 is missing the go program will quit with a fatal error (`fatal error: all goroutines are asleep - deadlock!`).

3.3 Closing and buffering channels

Channels can be closed, too. After closing it, it is not allowed to write into them. Doing so will raise a panic (`panic: send on closed channel`). But because of the garbage collection in Go it is fine to keep the channel open and never close it. Go's garbage collection mechanism will take care of it.

Channels can be buffered with a second argument to the make()-function, so only when the buffer is full it will block [5]. By default, channels are unbuffered.

Listing 5 shows an example of how-to buffer and close a channel.

Listing 5. channels-buffered.go

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     ch := make(chan string, 2) ①
7     ch <- "Hello"
8     ch <- "World" ②
9     hello := <-ch
10    fmt.Println(hello)
11    fmt.Println(<-ch)
12
13    close(ch) ③
14    _, ok := <-ch ④
15    fmt.Println(ok)
16 }
```

① This value assigns to the variable `ch` a channel of strings buffering up to two values. It will now even accept sending to the channel (`chan ←`) if there is no corresponding receive (`← chan`).

② Accepts now a second string because the buffer length is 2. If the buffer is not big enough the program will quit with a fatal error (`fatal error: all goroutines are asleep - deadlock!`).

③ The channel is closed with the function close and the channel as argument.

④ Go is capable of assigning and returning multiple values at once. Reading from a channel can return two values: the value inside the channel and if this value was successfully read. The value `false` of the variable `ok` tells us that the channel is already closed. So closing can be useful when the receiver must be told that there are no more values coming.

4 Parallel programming in Scala

Scala is a functional and object-orientated programming language. The author assumes that there is a basic knowledge of Java and Go to understand the comparisons. Scala has been created to solve some criticisms that Java has. Designed to be compiled to Java bytecode it runs on the Java virtual machine. It is fully object-orientated so every value is an object, whereas in Java or Go there are primitive values (e.g. `int` in Java or `int32` in Go), which are not objects. Functions are objects in Scala, too.

4.1 Threads in Scala

In Listing 6 you see two threads running in parallel. They both print a string saying “Hello” from the currently running thread.

The execution order is completely arbitrary. So either the first or the second defined thread will be started first and it’s result printed on the standard output.

Listing 6. Threads.scala

```
1 package de.thro.sinfmaglae ①
2
3 import com.typesafe.scalalogging.LazyLogging
4
5 object Threads extends App with LazyLogging { ②
6
7   val thread1 = new Thread(() => ③
8     logger.debug(s"Hello World from ${Thread.currentThread()}")
9   )
10  thread1.start() ④
11
12  val thread2 = new Thread(new Runnable {
13    override def run(): Unit = {
14      logger.debug(s"Hello World from ${Thread.currentThread()}")
15    }
16  })
17  thread2.start()
18
19  logger.debug("Hello from Main-Thread") ⑤
20 }
```

① Like Java, every Scala source file is in a package and the package declaration is in the first line of the source file. If there is no package specified, it is the default package.

② A Scala program starts within a main method, which is in an object. When the `object` key word is used a singleton object is created. Here the main method is inside the `App` trait. That means that all the body’s source code is automatically run. The code `with LazyLogging` adds the LazyLogging-trait to the Scala object. This trait provides a slf4j logger instance via a logger field, so logging is easily possible without the need to declare an extra logger. By default, the logger shows us the thread from which the logger’s output comes from. Swapping `App` and `LazyLogging`

does not make any difference here.

③ The program uses the `Thread` class from `java.lang.*` and because it is in `java.lang.*` it must not be imported. *All existing Java libraries and frameworks can be imported and used in Scala projects.* The `Thread` class takes a `Runnable` as constructor parameter, which is provided here via a lambda expression. Lambda expressions in Scala work like lambda expressions in newer versions of Java. They use a different arrow (⇒) in Scala, because the → arrow is used in Scala for the map initialization syntax. This is the short form of the initialization of `thread2`. The `val` keyword defines a fixed value with the specified name. It is like `final` in Java or `const` in Go. Like Go the type of the variable or constant does not need to be specified because it can be inferred from the initialization.

④ The start()-method starts the thread.

⑤ This statement runs inside the main thread. The main thread is like the main goroutine in Go. But instead of terminating all other goroutines Scala programs wait until the other threads are finished. The main thread is finished when there is nothing more to execute inside the main thread and when there are no other threads running.

After executing the program, a possible result could look like this:

```
14:00:05.988 [main] DEBUG de.thro.sinfmaglae.Threads$ - Hello form Main-Thread
14:00:05.988 [Thread-0] DEBUG de.thro.sinfmaglae.Threads$ - Hello World from
Thread[Thread-0,5,main]
14:00:05.988 [Thread-1] DEBUG de.thro.sinfmaglae.Threads$ - Hello World from
Thread[Thread-1,5,main]
```

Or it could look like this.

```
14:01:49.878 [Thread-1] DEBUG de.thro.sinfmaglae.Threads$ - Hello World from
Thread[Thread-0,5,main]
14:01:49.878 [Thread-0] DEBUG de.thro.sinfmaglae.Threads$ - Hello World from
Thread[Thread-1,5,main]
14:01:49.878 [main] DEBUG de.thro.sinfmaglae.Threads$ - Hello from Main-Thread
```

4.2 Parallel collections

There are some libraries in Scala for easier development of parallel programs. Collections like maps, lists, etc. are usually being used in many programs. The “Parallel collections” library provides an easy, developer friendly access to implement parallelism in Scala programs which make use of collections. By providing a simple and high level abstraction it sparse the developer from the low level details in parallelization [6]. Listing 7 shows an example.

Listing 7. ParallelCollections.scala

```
1 package de.thro.sinfmaglae
2
3 object ParallelCollections extends App {
4
5   val list = List(1, 2, 3, 4, 5)
6   list.foreach(print) ①
7   println()
8   list.par.foreach(print) ②
9
10  val zipCodes = Map("83059" -> "Kolbermoor" /*, ...*/).par ③
11  zipCodes.filterKeys(_.startsWith("83")).foreach(println)
12
13 }
```

① This line prints the values of the list in the same order as they are in the list (12345), because the values are processed without any parallelism.

② Simply by adding `.par` to the collection's variable, the collections is processed in parallel. It is easily recognizable here, because the ordering of the list when they are printed out to the screen is now completely random.

③ Not only lists but all collections in Scala can be parallelized with `.par`. For most collections it takes linear time. Like in Scala common it will not alter the underlying collection but instead returning a new, parallel implementation of that collection.

Although "Parallel collections" provides a very nice and comfortable way to introduce parallelism into an application it must always be carefully selected if it is worth the overhead. Parallelism can also slow down a program if the effort of splitting the problem into chunks is bigger then the problem itself.

Listing 8 shows a benchmark of summing up a list of integer values. Here you can easily see how many elements it takes on a standard 4-core desktop CPU to perform as fast as the serial implementation without the overhead.

Listing 8. ParallelCollectionsBenchmark.scala

```
1 package de.thro.sinfmaglae
2
3 object ParallelCollectionsBenchmark extends App {
4
5   var time = System.currentTimeMillis()
6   val list = (1 to 250000).toList ①
7   list.sum
8   println(s"Time taken: ${System.currentTimeMillis() - time}") // Time taken: 145
9   time = System.currentTimeMillis()
10  list.par.sum
11  println(s"Time taken: ${System.currentTimeMillis() - time}") // Time taken: 146
12 }
```

① **250,000** is roughly the size of the list *on the testing machine* when the parallel sum of the elements takes about the same amount of time then the non-parallel sum. Measured on an Intel Core i5-2500k @ 3.3 GHz with Scala 2.12.8 and Java 1.8.0_181 it takes about 145 ms for each run to compute the sum. When the *number is higher the parallel sum is faster then the non-parallel sum*. When the number is lower the parallel sum is slower then the non-parallel sum.

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

The goroutines in Go have some advantages over the threads in Scala. They have a faster startup time, and they can safely communicate between themselves with channels. Their seamless integration into the language with the keyword `go` makes them really handy and easy to use for the developer. Compared to threads goroutines use very few system resources, so it is possible to have more of them running at the same time. This and the fact that Go code compiles into a single native binary with dependencies makes Go a nice language for programming high performance system level applications.

Because Scala can be compiled into standard bytecode it can be expected to run on any device equipped with a Java virtual machine. It has the advantage of the “Write once, run anywhere” possibility, which makes it unnecessary to compile it for multiple architectures or operating systems [7]. For Java developers Scala is also quite easy to learn because you can use any Java library in Scala. So you still have access to all the libraries from the most used programming language of the world [8]. In addition, you have access to the Scala libraries like “Parallel collections” which makes parallelizing of collections just a matter of adding an extra word. This makes Scala interesting for companies which use Java in their daily business. The available libraries provide a lot more comfort for programming an extensive application then Go. In Go you do not have access to such a big ecosystem of libraries.

To sum up, it all depends on the project which language is more suitable. But the tendency is that Go is more for specialized system level applications which do one small job at a high performance level. Scala instead is a nice language for writing critical parts of existing Java applications or to write platform independent desktop applications. Both provide features which allows the programmers more easily to introduce parallelism into their applications then Java, C or C++.

References

- [statista] https://de.statista.com/statistik/daten/studie/38755/umfrage/nutzungsanteil-von-pcs-auf-der-plattform-steam-nach-anzahl-der-cpus/ - Online 22. Dec. 2018
- [go-goroutines] https://tour.golang.org/concurrency/1 - Online 24. Dec 2018
- [go-GOMAXPROCS] https://golang.org/pkg/runtime/ - Online 24. Dec 2018
- [go-channels] https://tour.golang.org/concurrency/2 - Online 24. Dec 2018
- [go-channels-buffer] https://tour.golang.org/concurrency/3 - Online 24. Dec 2018

- [scala-parallel-collections] https://docs.scala-lang.org/overviews/parallel-collections/overview.html - Online 25. Dec. 2018
- [wiki-write-once] https://en.wikipedia.org/wiki/Write_once,_run_anywhere - Online 27. Dec 2018
- [tiobe-index] https://www.tiobe.com/tiobe-index/ - Online 27. Dec 2018