THE WAY TO GO

A Thorough Introduction to the Go Programming Language

IVO BALBAERT

Chapter 14—Goroutines and Channels

As expected of a 21st century programming language, Go comes with built-in support for *communication* between applications (networking, client-server, distributed computing, see chapter 15) and support for *concurrent* applications. These are programs that execute different pieces of code simultaneously, possibly on different processors or computers. The basic building blocks Go proposes for structuring concurrent programs are **goroutines** and **channels**. Their implementation requires support from the language, the compiler and the runtime. The garbage collection which Go provides is also essential for easy concurrent programming.

Do not communicate by sharing memory. Instead, share memory by communicating.

Communication forces coordination.

14.1 Concurrency, parallelism and goroutines

14.1.1 What are goroutines?

An application is a *process* running on a machine; a process is an independently executing entity that runs in its own address space in memory. A process is composed of one or more operating system *threads* which are simultaneously executing entities that share the same address space. Almost all real programs are *multithreaded*, so as not to introduce wait times for the user or the computer, or to be able to service many requests simultaneously (like web servers), or to increase performance and throughput (e.g. by executing code in parallel on different datasets). Such a *concurrent* application can execute on 1 processor or core using a number of threads, but it is only when the same application process executes at the same point in time on a number of cores or processors that it is truly called *parallelized*.

Parallelism is the ability to make things run quickly by using multiple processors. So concurrent programs may or may not be parallel.

Multithreaded applications are notoriously difficult to get right, the main problem is the shared data in memory, which can be manipulated by the different threads in a non-predictable manner, thereby delivering sometimes irreproducible and random results (called *racing conditions)*.

!! Do not use global variables or shared memory, they make your code unsafe for running concurrently !!

The solution lies in *synchronizing* the different threads, and *locking* the data, so that only one thread at a time can change data. Go has facilities for locking in its standard library in the package sync for when they're needed in lower level code; we have discussed them in § 9.3. But the past experience in software engineering has shown that this leads to complex, error-prone programming and diminishing performance, so this classic approach is clearly not the way to go for modern multicore and multiprocessor programming: the 'thread-per-connection'- model is not nearly efficient enough.

Go adheres to another, in many cases better suited paradigm, which is known as Communicating Sequential Processes (CSP, invented by C. Hoare) or also called the message passing-model (as applied in other languages such as Erlang).

The parts of an application that run concurrently are called *goroutines* in Go, they are in effect concurrently executing computations. There is no one-to-one correspondence between a goroutine and an operating system thread: a goroutine is mapped onto (multiplexed, executed by) one or more threads, according to their availability; this is accomplished by the goroutine-scheduler in the Go runtime.

Goroutines run in the same address space, so access to shared memory must be synchronized; this could be done via the sync package (see § 9.3), but this is highly discouraged: Go use *channels* to synchronize goroutines (see § 14.2 etc.)

When a goroutine is blocked by a system call (e.g. waiting for I/O), other goroutines continue to run on other threads. The design of goroutines hides many of the complexities of thread creation and management.

Goroutines are lightweight, much lighter than a thread. They have a very small footprint (use little memory and resources): they are created with a 4K memory stack-space on the heap. Because they are cheap to create, a great number of them can be started on the fly if necessary (in the order of 100 thousands in the same address space). Furthermore they use a segmented stack for dynamically growing (or shrinking) their memory-usage; stack management is automatic. The stacks are not managed by the garbage collector, instead they are freed directly when the goroutine exits.

Goroutines can run across multiple operating system threads, but crucially, they can also run *within* threads, letting you handle myriad tasks with a relatively small memory footprint. Goroutines time-slice on OS threads as it were, so you can have any number of goroutines being serviced by a smaller number of OS threads, and the Go runtime is smart enough to realize which of those goroutines is blocking something and go off and do something else.

Two styles of concurrency exist: deterministic (well-defined ordering) and non-deterministic (locking/mutual exclusion but order undefined). Go's goroutines and channels promote deterministic concurrency (e.g. channels with one sender, one receiver), which is easier to reason about. We will compare both approaches in a commonly occurring algorithm (the Worker-problem) in § 14.7

A goroutine is implemented as a function or method (this can also be an anonymous or lambda function) and called (invoked) with the keyword **go**. This starts the function running in parallel with the current computation but in the same address space and with its own stack, for example:

```
go sum(bigArray) // calculate sum in the background
```

The stack of a goroutine grows and shrinks as needed, there is no possibility for stack overflow; the programmer needn't be concerned about stack size. When the goroutine finishes it exits silently: nothing is returned to the function which started it.

The `main()` function which every Go program must have can also be seen as a goroutine, although it is not started with go. Goroutines may be run during program initialization (in the `init()` function).

When 1 goroutine is e.g. very processor-intensive you can call `runtime.Gosched()` periodically in your computation loops: this yields the processor, allowing other goroutines to run; it does not suspend the current goroutine, so execution resumes automatically. Using Gosched() computations are more evenly distributed and communication is not starved.

14.1.2 The difference between concurrency and parallelism

Go's concurrency primitives provide the basis for a good concurrency program design: expressing program structure so as to represent independently executing actions; so Go's emphasis is not in the 1st place on parallelism: concurrent programs may or may not be parallel. Parallelism is the ability to make things run quickly by using multiple processors. But it turns out most often that a well designed concurrent program also has excellent performing parallel capabilities.

In the current implementation of the runtime (Jan 2012) Go ~~does not~~ parallelize code by default, ~~only a single core or processor is dedicated to a Go program~~, regardless of how many goroutines

As of Go 1.5, the default value of GOMAXPROCS is the number of CPUs (whatever your operating system considers to be a CPU) visible to the program at startup.

are started in it; so these goroutines are running *concurrent*, they are not running in parallel: only one goroutine is running at a time.

This will probably change, but until then in order to let your program execute simultaneously by more cores, that is so that goroutines are really running in parallel, you have to use the variable GOMAXPROCS .

This tells the run-time how many goroutines shall execute simultaneously.

Also only the gc-compilers have a true implementation of goroutines, mapping them onto OS threads as appropriate. With the gccgo compiler, an OS thread will be created for each goroutine.

14.1.3 Using GOMAXPROCS