Golang Multithreading

Mykhaylo Bavdys
dept. of Electronics and Computer Technologies
Ivan Franko National university of Lviv
Lviv, Ukraine
bavdysmyh@ukr.net

Abstract—**In the thesis the issues of development of microservices on the basis of object-oriented design are considered. The peculiarities of creating the layers of logic were described. The methods of research and implementation of the information system for the development of software products for the system on the basis of microservices in the style of object-oriented design have been analyzed, namely: the requirements to the information system; the database structure; the layer structure; architecture of microservice; the structure of the administrative part of the system; the description of the restriction of access to data. The work is based on the Golang language. The coursework contains the following sections: basic concepts of development tools, description of the project, project testing. In the course of the work, conclusions and suggestions were made on improving the development of information systems and the use of micro-service architecture development tools in the style of object-oriented design to achieve the set goals.**

Index Terms—**software framework, database, main tool, programming, microservice, architecture, Go, web server, Nginx, analysis, abstract layer.**

I. INTRODUCTION

Currently research on the Go language and the multi-threaded Go-language experiments are currently relevant. The main purpose of the study of multi-threaded Go language is carrying information and ensuring the relevance and usefulness of this information.

II. MULTITHREADING IN GO

When initially launched, Golang uses one thread, using its own scheduler and asynchronous calls. (The programmer acquires a sense of multithreading and parallelism.) In this case, the channels run very fast. But if you specify Go to use 2 or more threads then Go starts using lockdowns and channel performance may drop.

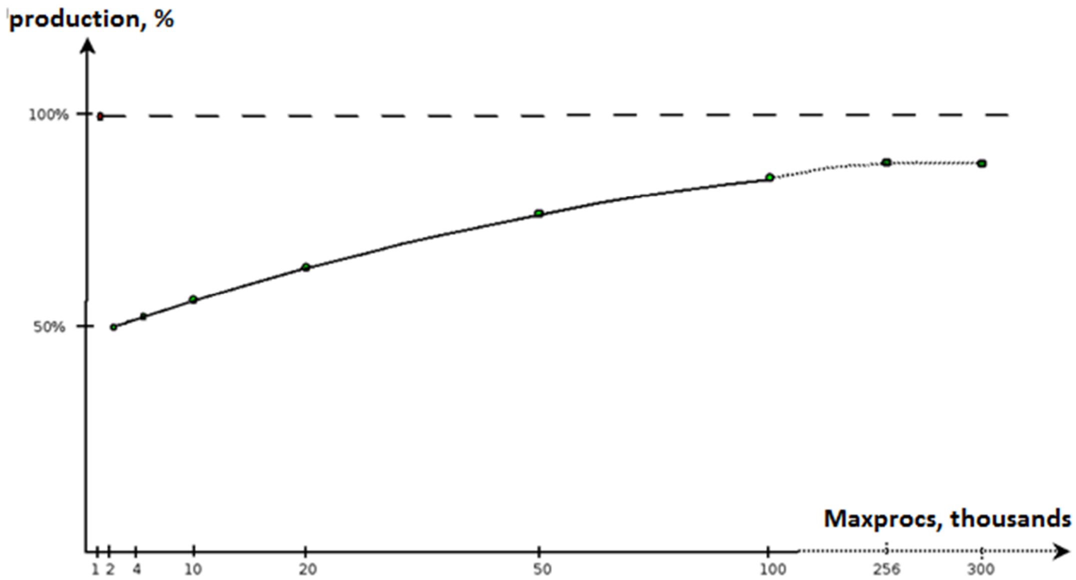
This is a big disadvantage. Moreover, most third-party libraries use channels any time it’s convenient. Therefore, it is often effective to launch Go with one channel, as it is done by

```
default:
package main
import "fmt"
import "time"
import "runtime"

func main () {
      num = = runtime.NumCPU ()
   fmt.Println (NumCPU, numb)
   //runtime.GOMAXPROCS(numcpu)
   runtime.GOMAXPROCS (1)

   ch1: = make (chan int)
   ch2: = make (chan float64)

   go func () {
      for i: = 0; i <1000000; i ++ {
         ch1 <- i
      }
      ch1 <- -1
      ch2 <- 0.0
   } ()
   go func () {
       total: = 0.0
      for {
         t1: time.Now (). UnixNano ()
         for i: = 0; i <100000; i ++ {
            m: = <-ch1
            if m == -1 {
               ch2 <- total
            }
         }
         t2: = time.Now (). UnixNano ()
         dt: = float64 (t2 - t1) / 1000000.0
         total + = dt
         fmt. Println (dt)
      }
   } ()
   fmt.Println ("Total:", <-ch2, <-ch2)
}
```

(pic.1)

```
users-iMac: channel user $ go run channel01.go
```

NumCPU 4
23.901
24.189
23.957
24.072
24.001
23.807
24.039
23.854
23.798
24.1
Total: 239.718 0



Picture 1. The ratio of productivity to the number of processes

Now, with the active use of all the cores we'll add a comment to the following lines:

runtime.GOMAXPROCS (numcpu)
//runtime.GOMAXPROCS(1)
users-iMac: channel user $ go run channel01.go
NumCPU 4
543.092
534.985
535.799
533.039
538.806
533.315
536.501
533.261
537.73
532.585
Total: 5359.113 0

20 times slower? What is the reason? The default channel size is 1.

ch1: = make (chan int)

Let's set it to 100.

ch1: = make (chan int, 100)

result with 1 thread

users-iMac: channel user $ go run channel01.go
NumCPU 4
9.704
9.618
9.178
9.84
9.869
9.461
9.802
9.743
9.877
9.756
Total: 0 96.848

result with 4 threads

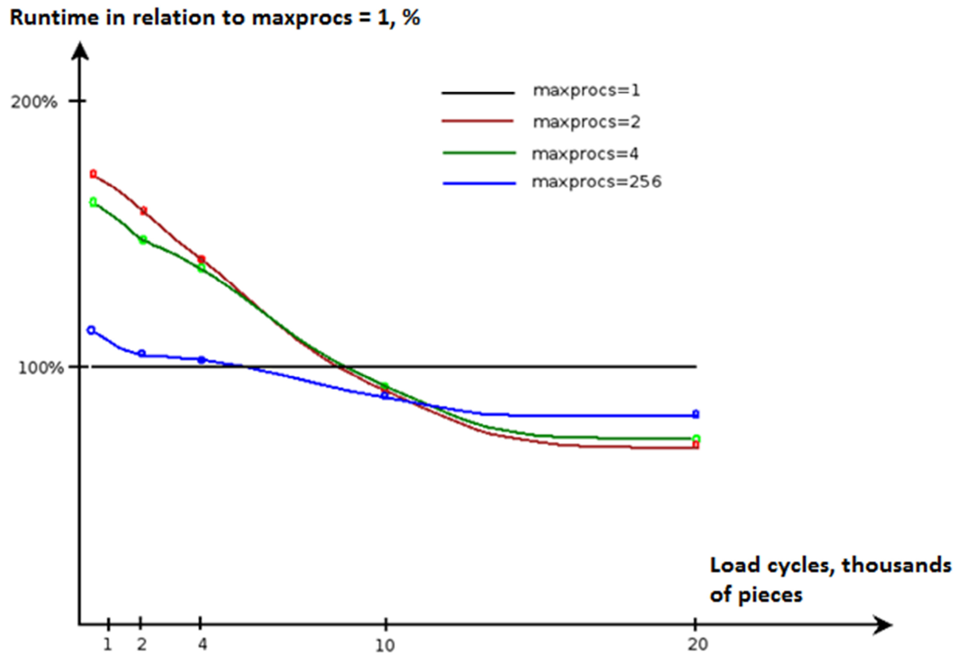
users-iMac: channel user $ go run channel01.go
NumCPU 4
17.046
17.046
16.71
16.315
16.542
16.643
17.69
16.387
17.162
15.232
Total: 0 166.77300000000002

Example "Thread of threads"

```
package main
import "fmt"
import "time"
import "runtime"
func main () {
   num = = runtime.NumCPU ()
   fmt.Println (NumCPU, numb)
   //runtime.GOMAXPROCS(numcpu)
   runtime.GOMAXPROCS (1)

   ch1: = make (chan chan int, 100)
   ch2: = make (chan float64, 1)

   go func () {
      t1: = time.Now () .UnixNano ()
      for i: = 0; i <1000000; i ++ {
          ch: = make (chan int, 100)
         ch1 <- ch
         <- ch
      }
      t2: = time.Now (). UnixNano ()
      dt: = float64 (t2 - t1) / 1000000.0
      fmt.Println (dt)
      ch2 <- 0.0
   } ()
   go func () {
      for i: = 0; i <1000000; i ++ {
         ch: = <-ch1
         ch <- i
      }
      ch2 <- 0.0
   } ()

   <-ch2
   <-ch2
}
```

result with 1 thread

users-iMac: channel user $ go run channel03.go
NumCPU 4
1041.489
Result with 4 threads
(pic.2)
users- iMac: channel user $ go run channel03.go
NumCPU 4
11170.616



Picture 2. The ratio of runtime in relation to maxprocs to load cycles

CONCLUSIONS.

So if you have 8 cores and you write a server using Go, you do not have to rely entirely on parallelizing the program. It is better to run 8 single-threaded processes, with a balancer in front of them.

What do these figures mean? The task was to handle 3000 requests per second in the same context (for example, to issue for each query sequentially the following numbers: 1, 2, 3, 4, 5 ...) and the productivity of 3000 requests per second is limited primarily to channels. With the addition of threads and cores, productivity is growing not as quickly as it would be desirable.

REFERENCES

[1] The Go Programming Language. [Electronical resource]. – Access mode: https://golang.org/doc/
[2] BSD Licenses. [Electronical resource]. – Access mode: ftp://ftp.cs.berkeley.edu/pub/4bsd/README.Impt.License.Change
[3] Go at Google: Language Design in the Service of Software Engineering. [Electronical resource]. – Access mode: https://talks.golang.org/2012/splash.article