Go 1.5 GOMAXPROCS Default

(discussion on golang-dev)

Russ Cox
May 2015

Abstract

To date, the default setting of GOMAXPROCS in Go releases has been 1.
For Go 1.5, we propose to change the default to the number of CPUs available.

Background

The GOMAXPROCS setting controls how many operating systems threads attempt to execute code simultaneously. For example, if GOMAXPROCS is 4, then the program will only execute code on 4 operating system threads at once, even if there are 1000 goroutines. The limit does not count threads blocked in system calls such as I/O.

GOMAXPROCS can be set explicitly using the GOMAXPROCS environment variable or by calling runtime.GOMAXPROCS from within a program.

The default setting of GOMAXPROCS in all extant Go releases is 1, because programs with frequent goroutine switches ran much slower when using multiple threads. It is much cheaper to switch between two goroutines in the same thread than to switch between two goroutines in different threads.

Goroutine scheduling affinity and other improvements to the scheduler have largely addressed the problem, by keeping goroutines that are concurrent but not parallel in the same thread.

At the same time, as core count continues to increase, not using additional cores by default puts Go at a disadvantage compared to other languages, especially in programs without high-frequency goroutine switching.

Proposal

For Go 1.5, we propose to change the default setting of GOMAXPROCS to the number of CPUs available, as determined by runtime.NumCPU.

We believe this will improve performance for the vast majority of Go programs, and it is a better long-term default than 1.

Impact

We expect the impact of this change to be largely positive. The performance of programs with many goroutines but no real parallelism is (in contrast to Go 1.0) no longer hurt significantly by raising GOMAXPROCS. The performance of single-goroutine programs can improve by raising GOMAXPROCS due to parallelism of the runtime, especially the garbage collector. And the performance of multi-goroutine programs with real parallelism can scale linearly with GOMAXPROCS.

Worst cases. The first concern is the kinds of goroutine switch-heavy programs that motivated the default of 1, in particular programs that pass data along a chain of goroutines. It is important that they run reasonably with the new default. To that end, we wrote simple benchmark versions of a basic goroutine chain and a prime sieve and tested them with different GOMAXPROCS settings and different versions of Go. The results show that while earlier versions did not run these programs well, Go 1.5 does.

The benchmarks here are presented as pairs of sets. The first set is taken from 20 runs on a 2012 MacBook Pro Core i5 with 4 CPUs (2 cores x 2 hyperthreads) running OS X. The second set is taken from a server-class Intel Xeon E5520 V2 with 16 CPUs (2 sockets x 4 cores x 2 hyperthreads) running Linux. See the appendix for information about running benchmarks on your own system.

The first benchmark creates a chain of 100 goroutines connected by channels and times how long it takes an integer message to propagate from one end to the other:

```
benchmark     Go 1.0                 Go 1.4                 “Go 1.5” [May 20 2015]
Chain         22.1µs × (0.98,1.06)   28.8µs × (0.99,1.03)   27.1µs × (0.99,1.02)
Chain-2        299µs × (0.99,1.02)     57µs × (0.97,1.03)     29µs × (0.98,1.02)
Chain-4        300µs × (0.98,1.03)     66µs × (0.98,1.04)     29µs × (0.99,1.01)

ChainBuf      24.2µs × (1.00,1.01)   32.4µs × (0.99,1.05)   30.2µs × (0.99,1.01)
ChainBuf-2     290µs × (0.99,1.03)     60µs × (0.97,1.04)     33µs × (0.99,1.01)
ChainBuf-4     289µs × (0.98,1.01)     69µs × (0.98,1.05)     33µs × (0.99,1.01)
```

```
benchmark     Go 1.0                 Go 1.4                 “Go 1.5” [May 20 2015]
Chain         27.7µs × (0.99,1.01)   38.8µs × (0.99,1.02)   36.1µs × (0.98,1.02)
Chain-2        341µs × (0.98,1.06)    110µs × (0.94,1.08)     37µs × (0.99,1.02)
Chain-4        359µs × (0.98,1.03)    119µs × (0.97,1.05)     37µs × (0.99,1.02)
Chain-8        369µs × (0.95,1.09)    120µs × (0.97,1.04)     38µs × (0.98,1.06)
Chain-12       379µs × (0.93,1.12)    111µs × (0.98,1.03)     37µs × (0.98,1.02)
Chain-16       373µs × (0.94,1.10)    106µs × (0.97,1.05)     37µs × (0.98,1.02)
```

```
ChainBuf      31.0µs × (0.99,1.01)   40.8µs × (0.99,1.01)   39.7µs × (0.99,1.01)
```

```
ChainBuf-2    343µs × (0.98,1.03)   116µs × (0.98,1.03)    41µs × (0.98,1.02)
ChainBuf-4    365µs × (0.97,1.03)   121µs × (0.96,1.02)    41µs × (0.99,1.02)
ChainBuf-8    370µs × (0.96,1.03)   125µs × (0.97,1.06)    41µs × (0.99,1.02)
ChainBuf-12   376µs × (0.95,1.16)   115µs × (0.97,1.07)    41µs × (0.99,1.02)
ChainBuf-16   370µs × (0.97,1.09)   109µs × (0.95,1.04)    41µs × (0.98,1.02)
```

The Chain benchmark uses unbuffered channels; ChainBuf uses buffered channels. The numeric suffix indicates the GOMAXPROCS setting. In Go 1.0 the slowdown of GOMAXPROCS=2 vs GOMAXPROCS=1 was over 10x; by Go 1.1 (not shown) it had been reduced to about 2x, where it has stayed through Go 1.4. Changes made in the Go 1.5 cycle have dropped the slowdown to under 1.1x (10%).

As another test, the prime sieve also uses many goroutines but has more opportunity for parallelism:

```
benchmark     Go 1.0                Go 1.4                "Go 1.5" [May 20 2015]
Sieve         29.8s × (0.94,1.07)   21.8s × (0.98,1.04)   21.1s × (0.98,1.03)
Sieve-2       23.4s × (0.97,1.03)   18.9s × (0.97,1.04)   17.0s × (0.79,1.31)
Sieve-4       24.0s × (0.99,1.02)   12.7s × (0.95,1.08)   10.6s × (0.80,1.28)
```

```
benchmark     Go 1.0                Go 1.4                "Go 1.5" [May 20 2015]
Sieve         27.3s × (0.99,1.01)   27.8s × (0.99,1.02)   27.3s × (0.99,1.02)
Sieve-2       31.3s × (0.81,1.30)   18.9s × (0.92,1.09)   15.3s × (0.80,1.36)
Sieve-4       37.0s × (0.91,1.06)   10.2s × (0.95,1.07)    8.1s × (0.79,1.30)
Sieve-8       34.1s × (0.99,1.02)    5.4s × (0.84,1.09)    4.7s × (0.90,1.22)
Sieve-12      35.6s × (0.95,1.13)    4.5s × (0.95,1.03)    3.9s × (0.91,1.08)
Sieve-16      58.1s × (0.95,1.05)    3.6s × (0.92,1.05)    3.7s × (0.94,1.10)
```

In Go 1.0 the GOMAXPROCS overhead mostly negated any benefit from the additional parallelism; in the current Go 1.5 tree, Sieve is able to use the additional cores to reduce the runtime significantly. On the Mac laptop, adding 4x the cores roughly halves the time; on the Linux server, adding only 2x the cores roughly halves the time.

There are still programs that do not run as nicely. For example, the Go port of Doug McIlroy's power series program is almost a worst case for GOMAXPROCS > 1: it creates many ephemeral goroutines, arranges the communication pattern in a tree, and has no use for parallelism:

```
benchmark     Go 1.0                Go 1.4                "Go 1.5" [May 20 2015]
Powser        2.54s × (1.00,1.00)   3.34s × (0.98,1.09)   3.10s × (0.99,1.03)
Powser-2      24.1s × (0.94,1.13)    8.1s × (0.96,1.06)    3.3s × (0.99,1.01)
Powser-4      32.8s × (0.95,1.13)    8.3s × (0.96,1.09)    3.5s × (0.99,1.02)
```

```
benchmark     Go 1.0                Go 1.4                "Go 1.5" [May 20 2015]
```

```
Powser        2.93s × (1.00,1.00)   4.18s × (1.00,1.01)   4.05s × (0.99,1.02)
Powser-2      34.4s × (0.98,1.02)   11.2s × (0.99,1.01)    4.2s × (0.99,1.02)
Powser-4      35.9s × (0.98,1.02)   13.5s × (0.98,1.02)    4.7s × (0.99,1.01)
Powser-8      36.1s × (0.99,1.02)   13.2s × (0.99,1.01)    4.7s × (0.97,1.02)
Powser-12     35.9s × (0.99,1.01)   12.3s × (0.99,1.01)    4.7s × (0.98,1.02)
Powser-16     35.8s × (0.99,1.01)   11.6s × (0.98,1.02)    4.7s × (0.97,1.04)
```

In Go 1.0 the slowdown of GOMAXPROCS=2 vs GOMAXPROCS=1 was over 10x, like Chain above. Go 1.1 through Go 1.4 reduced that overhead to about 2.5x. Changes made in the Go 1.5 cycle have dropped the slowdown to under 1.2x (20%).

Single-threaded programs. The above has focused on worst cases, which are the reason GOMAXPROCS defaults to 1 today. The other half of the performance story is programs with real parallelism are helped significantly by higher GOMAXPROCS values. This will vary by program, but even ostensibly single-threaded programs get faster, due to running garbage collection and other runtime work on the additional cores. For example, here are the times for various development operations on the same pair of servers used in the benchmarks above.

GOMAXPROCS	make.bash	go list -json std	go test -c html/template
1	51.4s	0.21s	1.9s
4	48.3s	0.21s	1.6s

GOMAXPROCS	make.bash	go list -json std	go test -c html/template
1	39.5s	0.27s	2.5s
4	34.5s	0.22s	1.9s
16	34.1s	0.22s	1.8s

The script make.bash is helped least, because it is already using process-level parallelism during the build of many Go packages to mostly saturate the machine with work. In contrast, an incremental build, like when testing changes to a package, runs the compiler and then the linker. There is no process-level parallelism. In this case the time taken improves by 15-30%.
The go list command is primarily I/O bound, but on the Intel server additional CPUs can speed up the runtime.

Parallel programs. Of course, programs with inherent parallelism (unlike the go commands above) improve even more. Using golang.org/x/benchmarks/bench:

GOMAXPROCS	garbage	http	json
1	31461838	146737	145014842
4	10427806	32853	44788720
16	5754040	12652	21817571

Correctness. There is one non-performance concern. Increased parallelism could make bugs in racy programs more likely to cause crashes or other problems. The setting of GOMAXPROCS=1 may have thus far let those bugs go undetected. Raising it may therefore make buggy programs less reliable. We hope that both the race detector and the introduction of goroutine preemption and interleaving in Go 1.1 has already helped programmers identify and eliminate many such bugs. In any event, we can't reasonably hobble the performance of working Go programs for fear of breaking buggy ones. If such buggy programs do turn up, the authors can set GOMAXPROCS=1 explicitly to keep them working until the bugs can be fixed.

Deployment

We propose to make this change soon, so that it will be in the Go 1.5 betas and release candidates.

If end users run into any kind of problems with the new default, there is a trivial workaround; set GOMAXPROCS=1 in the environment before invoking the problematic program.

We will decide whether to keep the new default setting based on problems reported (or not reported) during the testing period.

The relevant entries in the FAQ will need to be reworded.

Appendix: Running Benchmarks

The worst case benchmarks above can be run on a Unix system by executing:

```
go get -u rsc.io/tmp/gomaxprocs/sieve rsc.io/benchstat
cd $GOPATH/src/rsc.io/tmp/gomaxprocs
./runall
./showall # -html
```

The script assumes you have commands "go1.0", "go1.1", "go1.2", "go1.3", and "go1.4" installed, and that the development version of Go is installed as just "go". On my system, I have each version checked out and built in $HOME/go1.0, $HOME/go1.1, and so on, and the commands invoke those tools after setting GOROOT appropriately:

```
$ cat $(which go1.2)
#!/bin/bash
export GOROOT=$HOME/go1.2
export PATH=$HOME/go1.2/bin:$PATH
exec $GOROOT/bin/go "$@"
$
```

The build times were gathered with commands like "GOMAXPROCS=1 time ./make.bash."

The golang.org/x/benchmarks/bench benchmarks were run with:

```
go get -u golang.org/x/benchmarks/bench
for i in 1 4 16
do
    for j in garbage http json
    do
        echo GOMAXPROCS=$i $j \
            $(GOMAXPROCS=$i bench -bench $j 2>&1|grep GOPERF-METRIC:time)
    done
done
```