Productive Go

Three reasons why Go feels like a productive language. A personal review.

> Martin Czygan, martin.czygan@gmail.com, 2020-XX-XX, Developer Group Leipzig (online)

About You

- most informative, if you have little (or no) Go exposure
- if you have done a lot of Go, then maybe not too many surprises

---

About Me

- software developer at Leipzig University Library (Library of the Year 2017) and data engineer at the Internet Archive - check out Archive Scholar, a search engine for scholarly documents
- open source contributor, computer scientist, lecturer and author
- co-host of Leipzig Golang User Group, meetup.com/Leipzig-Golang

I started to use Go in 2013, that must have been Go 1.1 release. The first program was a replacement for a Java command line tool.

Random

In my spare time, I sometimes take part in hackathons (join me); last time I created slot machine animations with a numpy:

Overview

- all languages have (significant) tradeoffs
- in this talk I want to highlight a few positive aspects of the language; there are many more
- Go is not great because of a single killer feature; in fact none of the highlights is that extraordinary, but the it adds up
- I believe, Go will become more popular (slowly) because it does less (and less can be more)

Three reasons

- Performance
- Ergonomics
- Deployment

Reason 1: Performance

Go is fast.

Fast compilation

It starts with dependency management.

> In 1984, a compilation of ps.c, the source to the Unix ps command, was observed to #include <sys/stat.h> 37 times by the time all the preprocessing had been done. Even though the contents are discarded 36 times while doing so, most C implementations would open the file, read it, and scan it all 37 times. – https://talks.golang.org/2012/splash.article#TOC_5.

So, compile time reduction starts with less I/O.

Example: A 67529 LOC project, seaweedfs: with empty go build cache: 67s, subsequent builds: 8s.

Go blurs the line

- Go first appeared on November 10, 2009 (remember Google Tech Talks?)

A few years before, there seemingly was a cold war going, since Erik Meijer et al. published Static Typing Where Possible, Dynamic Typing When Needed:The End of the Cold War Between Programming Languages.

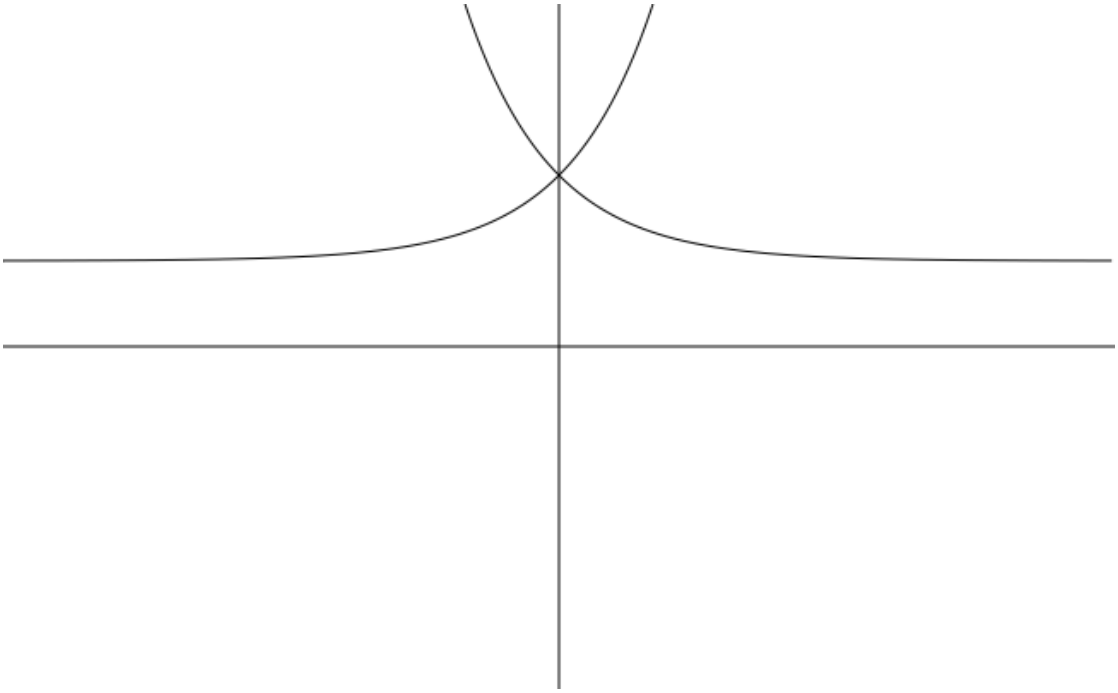
The paper goes into a “softer type system” direction, but Go also wanted to end this war. It wanted to be a safe language (static) that was fun to write (dynamic).

On part of that is: Can I run this instantly? And in Go, you can, with `go run prog.go` it sure feels fast.

Go is fast enough

- Go is not the fastest language, but fast enough
- there is a (assumed) optimum for a given problem, between how fast it is, and how quickly you can implement it

A tradeoff between time spent and runtime, e.g. as you **increase** the time spent on programming, the running time **comes down**.



Let's look at one example.

Fast enough Reservoir Sampling

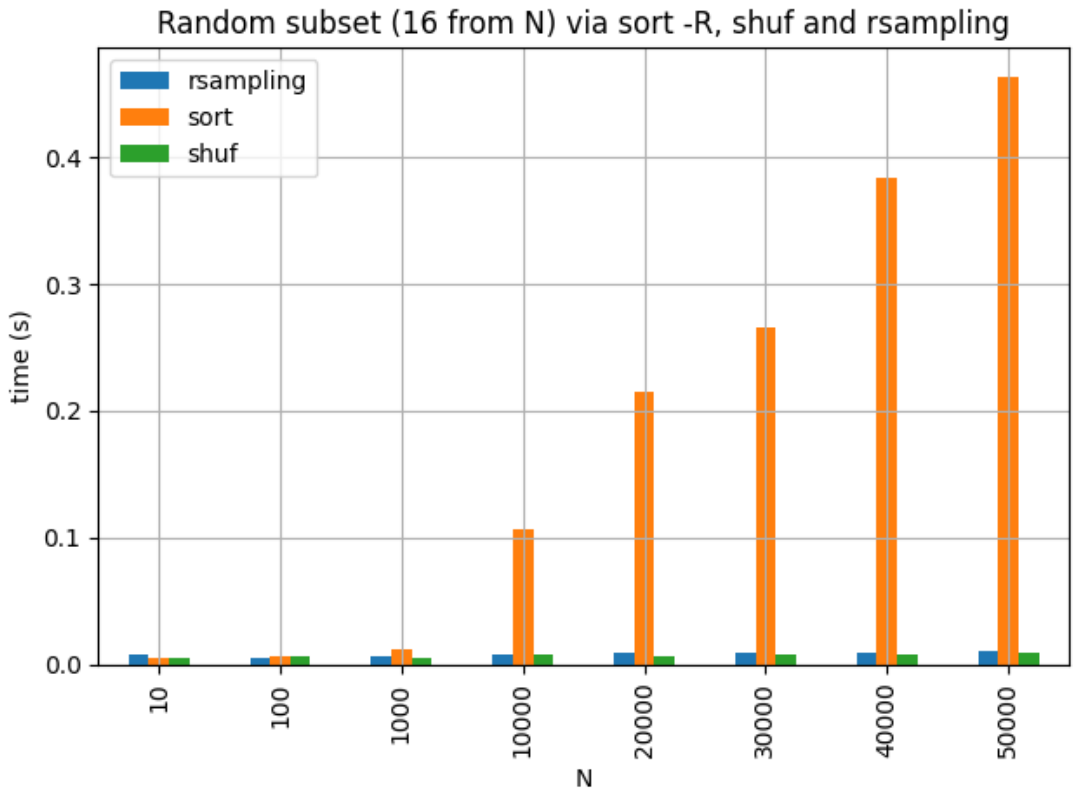
Reservoir sampling is a powerful technique to get a sample of a fixed size from a potentially infinite stream.

- Not POSIX, but included in GNU core utils is shuf, which uses reservior sampling (since 2013) - I use shuf regularly (and also once needed a variant to shuffle large files, and found terashuf - porting that C++ program to Go is still a TODO).

> Hi, I would like to know why shuf.c is using reservoir sampling + write_permuted_output_reservoir rather than just using an inside-out version Fisher-Yates shuffle. – https://lists.gnu.org/archive/html/coreutils/2013-12/msg00165.html | *Reservoir sampling is used to limit memory usage*

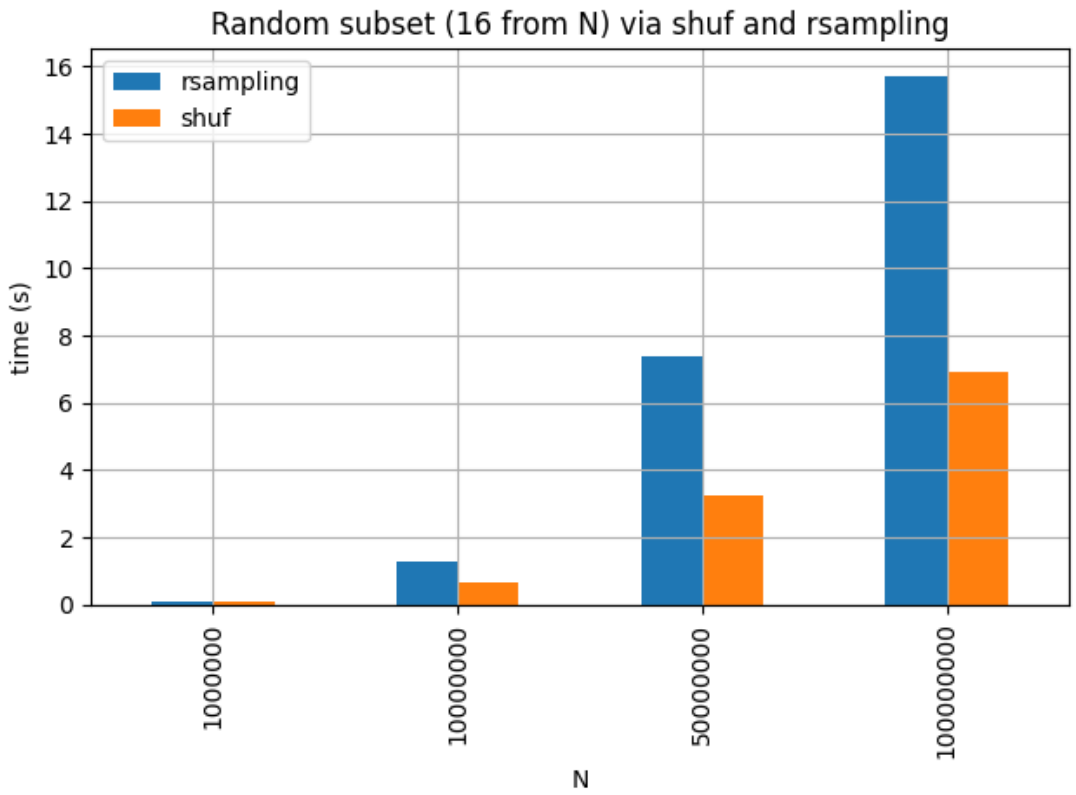
A Go version: rsampling

The wikipedia page on `shuf` mentions `sort -R`, so let's see:



Fast enough is enough

Let's zoom in.



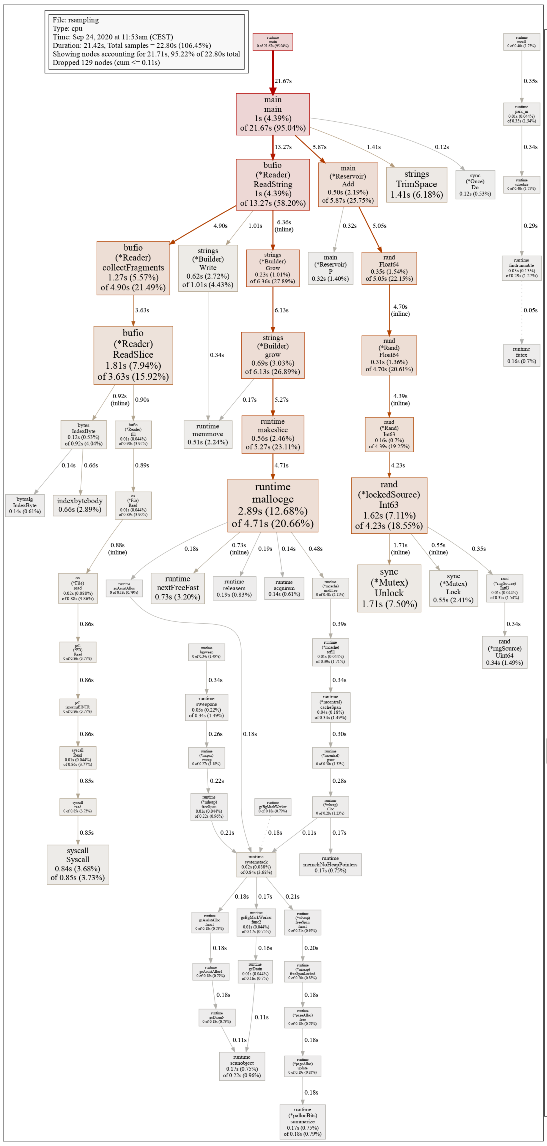
The Go project is 91 lines of code of which 12 are imports - standard library only - which my editor completes for me. Also 6 lines for a “version” flag. It responds to SIGINT, which is a nice-to-have and 12 more lines. Essentially around 60 lines of code.

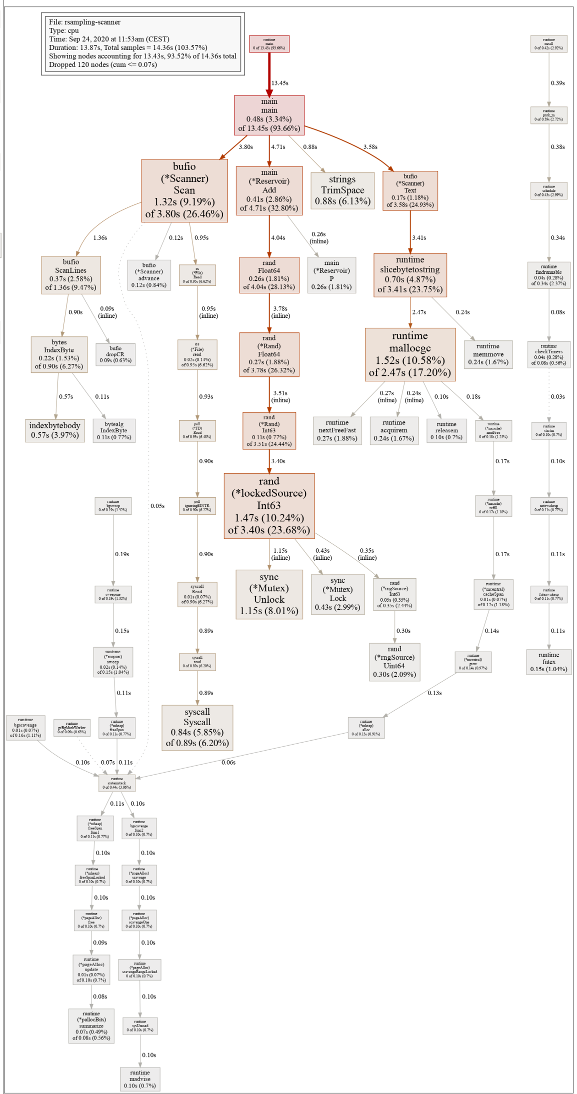
It did not took long to write the Go version, and the initial, unoptimized version was *fast enough*.

A bit of optimization

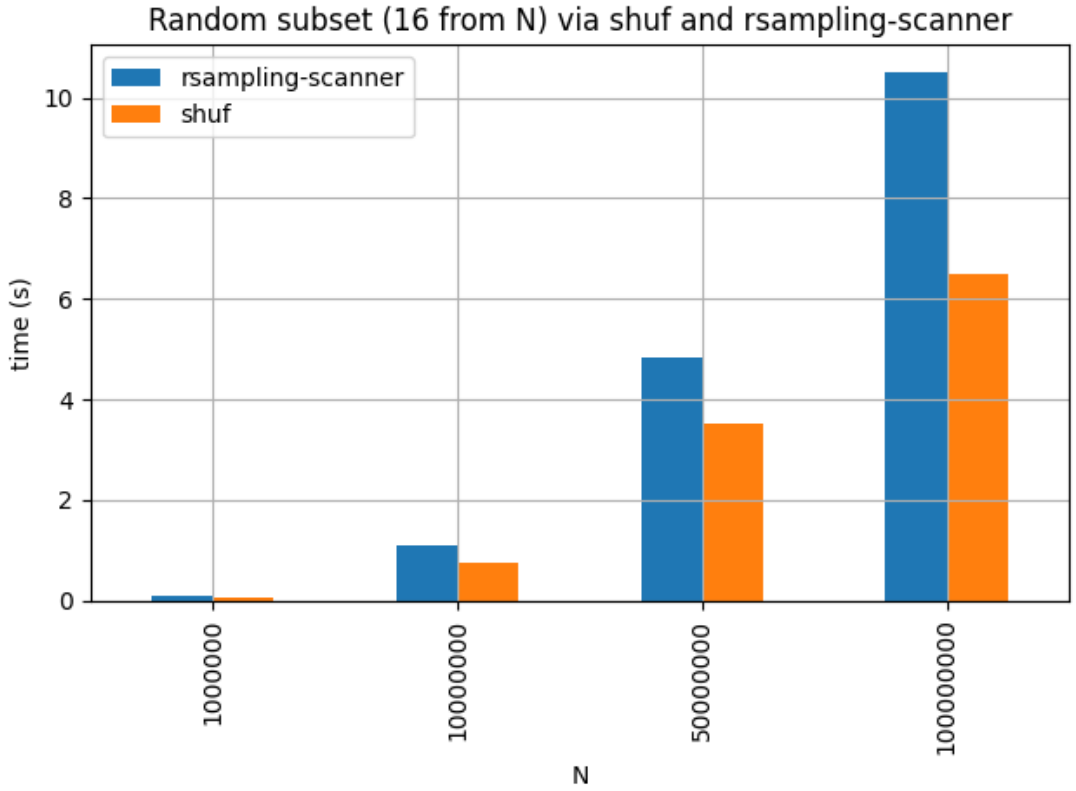
Go has been described as both high and low level language. One optimization for rsampling relates to memory allocation.

The following is an output of the builtin go profiler, left Reader, right Scanner. It is hard to see, but the Scanner is lighter on “malloc”.


File: rsampling
Type: cpu
Time: Sep 24, 2020 at 11:53am (CEST)
Duration: 21.42s, Total samples = 22.80s (106.45%)
Showing nodes accounting for 21.71s, 95.22% of 22.80s total
Dropped 129 nodes (cum <= 0.11s)


File: rsampling-scanner
Type: cpu
Time: Sep 24, 2020 at 11:53am (CEST)
Duration: 13.87s, Total samples = 14.36s (103.57%)
Showing nodes accounting for 13.43s, 93.52% of 14.36s total
Dropped 120 nodes (cum <= 0.07s)

Scanner is a bit faster



That is not too bad for a garbage collected, memory-safe language (even if the sample size is one).

The free lunch is over

- Free Lunch Is Over

Interestingly, Go has concurrency support built into the language. The keyword is go which starts a goroutine.

Concurrency

- a way to decompose a program first (see also: Concurrency is not parallelism)
- a concurrent program may run in parallel, when possible
- used in popular parts of the standard library, e.g. in `net/http`

Raw Primitives

- Go CSP concurrency primitives can feel raw

However, concurrency can be wrapped into a synchronous model. Example (parallel command line filter, error handling omitted):

```
parallel.NewProcessor(os.Stdin, os.Stdout, func(p []byte) ([]byte,
↪  error) {
    var data Data
    json.Unmarshal(p, &data)
    ...
}).Run()
```

No thread, goroutine, channel or select, yet it will use all cores and will use batching to keep balance communication overhead.

Further performance options

- SIMD

- go is fast
- fast compilation
- fast enough
- low level, high level

- concurrent programming; raw patterns
- implement JSON parser; SIMD
- show off cubietruck ARM, CRUD App, 500 r/s

Reason 2: Ergonomics

- emphasis on reading code
- gofmt
- can read code of key value stores, or more complex pieces of code
- regular language, tools on code
- high level, low level
- stdlib ftw
- writing complete crud apps with minimal dependencies

Reason 3: Deployment

- single binary
- what is in that binary; show off
- smaller linux images
- cross-compilation
- selective compilation