Mark, Set, Golang

Benchmarking Concurrency in the Go Runtime

Audrey Dutcher & Caitlin Scarberry

What is the Go language?

- Statically-typed open-source language
- Does **not** use a virtual machine
  - Runtime features (e.g. garbage collection) are part of a Go library
- Strong built-in support for concurrency
- We'll use 'Go language' to refer to the Go language specification **and** its official implementation

What is the basis of Go’s concurrency model?

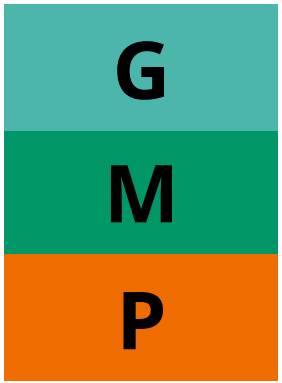
goroutines

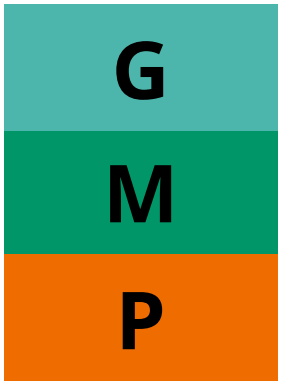
(lightweight threads)

channels

(locked queues)

The Go Scheduler





Goroutine

OS thread

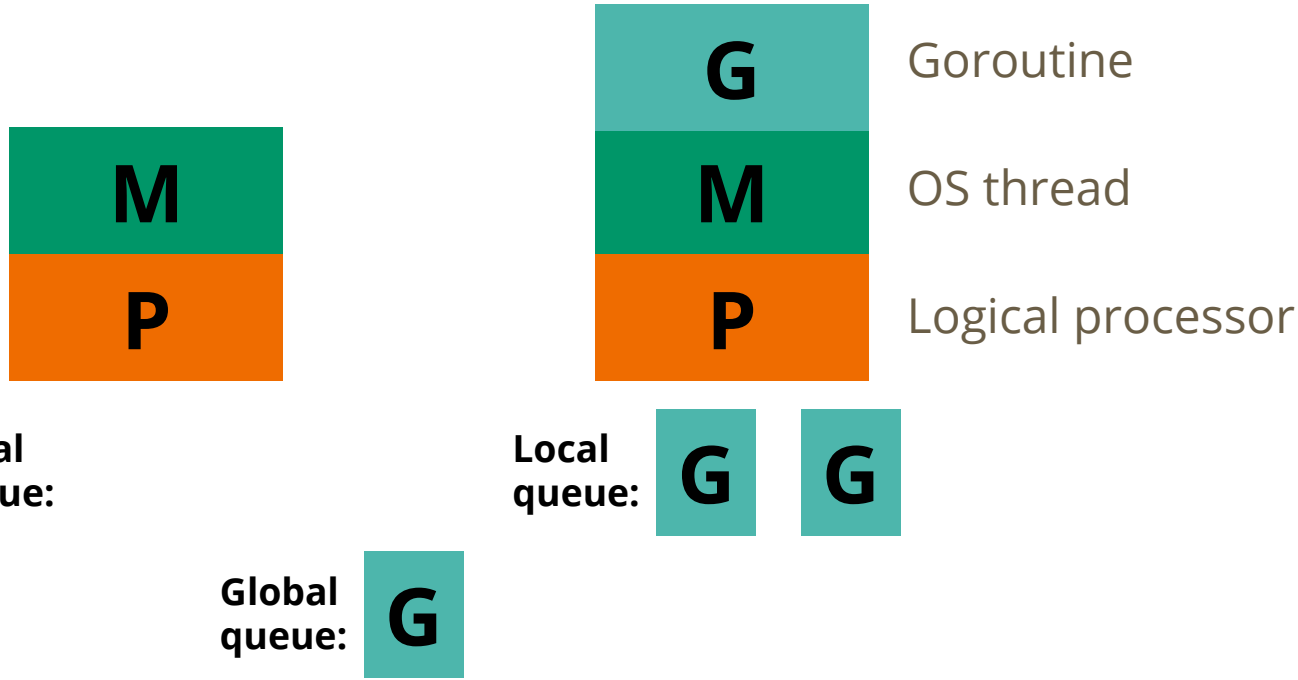
Logical processor

Local queue:

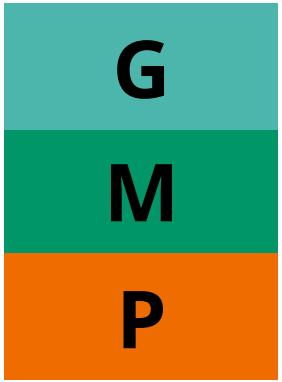
Local queue: G G

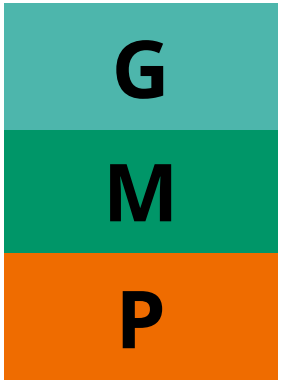
Global queue: G

The Go Scheduler



The Go Scheduler





Goroutine

OS thread

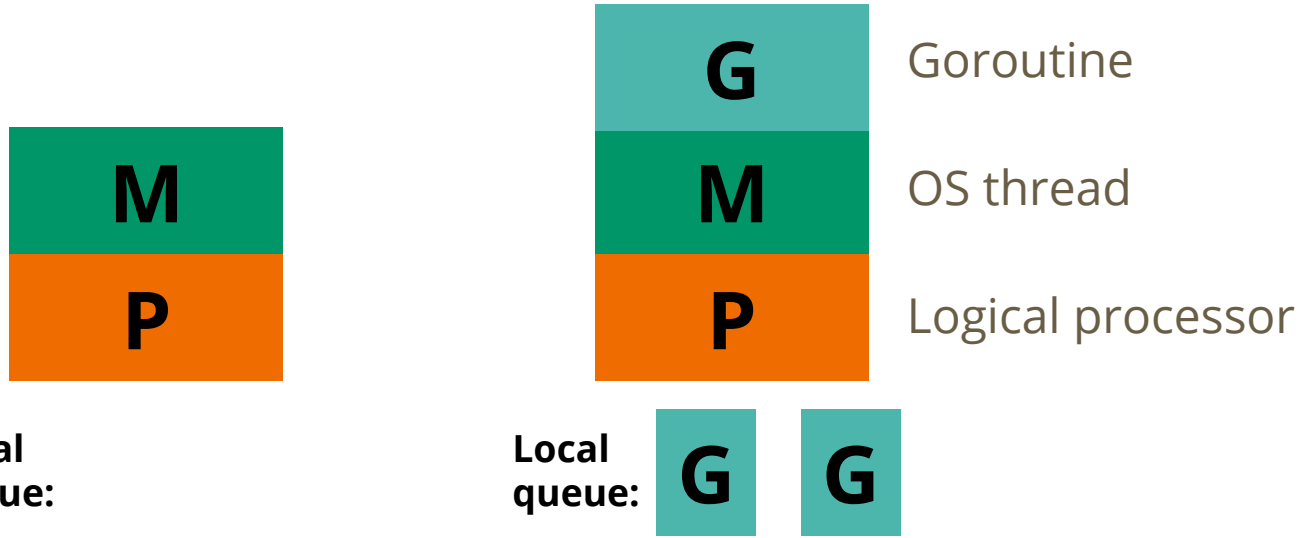
Logical processor

Local queue:

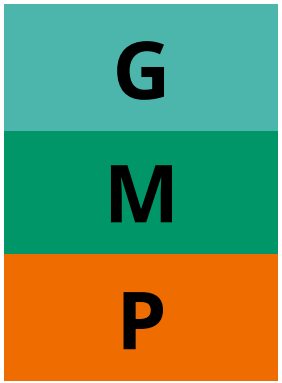
Local queue: G G

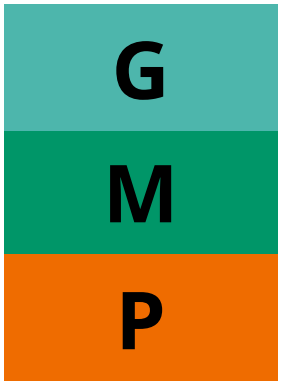
Global queue:

The Go Scheduler



The Go Scheduler





Goroutine

OS thread

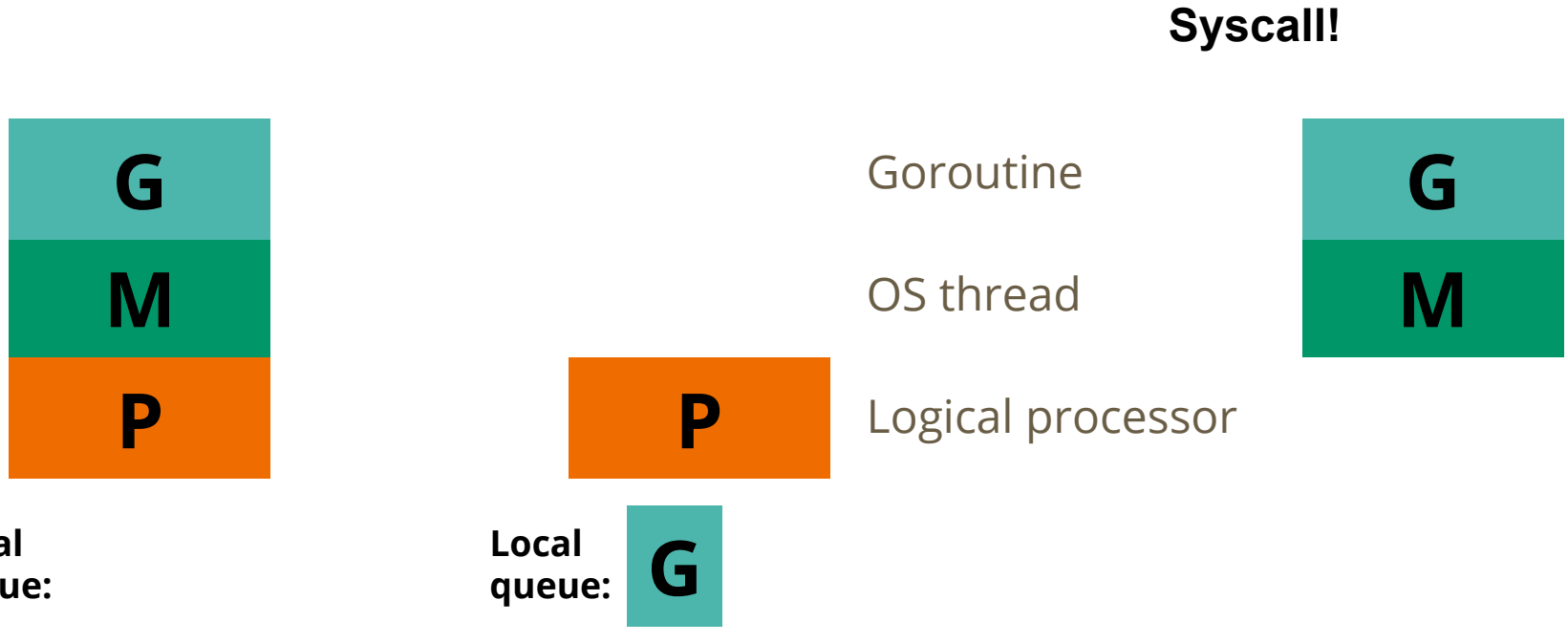
Logical processor

Local queue:

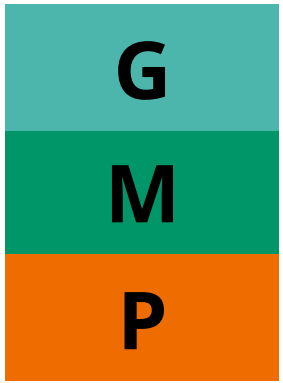
Local queue: G

Global queue:

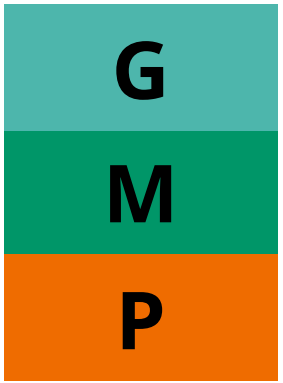
The Go Scheduler



The Go Scheduler



New thread!



Goroutine

OS thread

Logical processor

Local queue:

Local queue:

Global queue:

Go uses its support for concurrency as a selling point.

“I said Go supports concurrency. I mean it *really* supports concurrency.”

- Rob Pike

Our problem: How do we evaluate these claims?

Our solution: Benchmark the **runtime implementation!**

Choosing Benchmarks

- Good benchmarks reflect how the language is used in practice
- We analyzed popular open-source Go projects to determine their most common concurrency use-cases
- We preferred **relative** benchmarks to absolute benchmarks

Our Benchmarks

- Scheduler efficiency for large numbers of simultaneous **independent** goroutines
- The time for each step in a data pipeline
- The accuracy of timeouts when large numbers of simultaneous goroutines are running
- The cost of starting goroutines

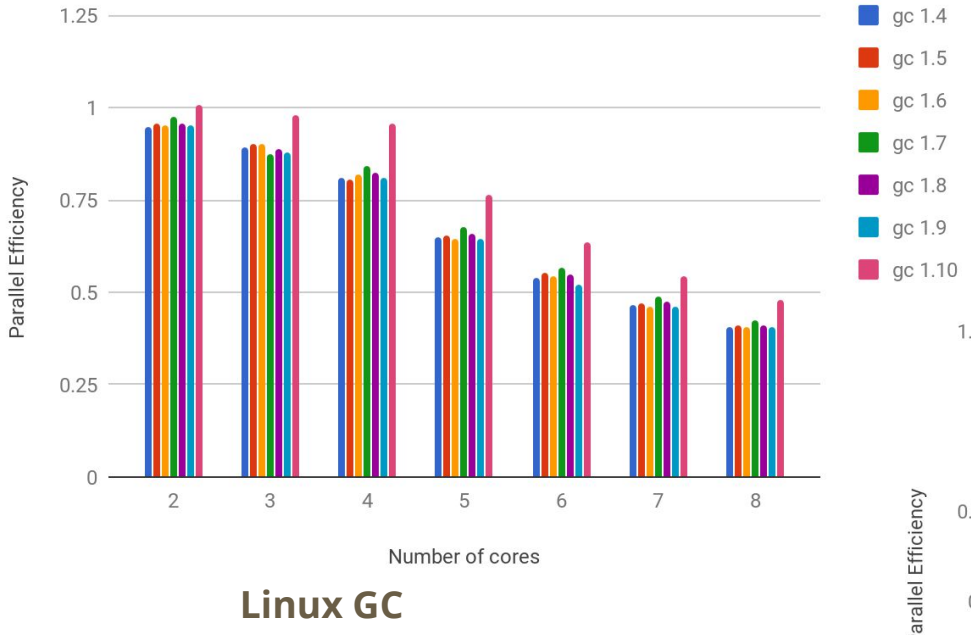
Test Environments

- Machines:
  - MacBook Pro, 2.7 GHz Intel Core i5, 4 cores
  - Dell, 2.9GHz Intel Core i7, 8 cores (Linux and Windows)
- Compilers:
  - gc
  - gccgo
- Versions:
  - 1.4
  - ...
  - 1.10

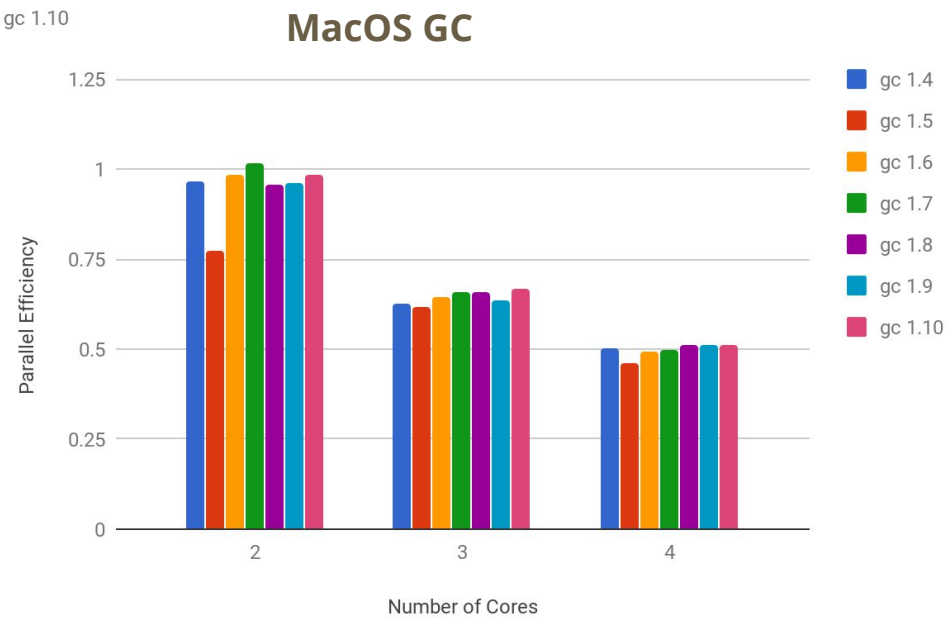
Implementation

- Implemented in Go
- Used Go’s runtime library to
  - set runtime parameters, e.g. number of logical processors
  - do CPU profiling

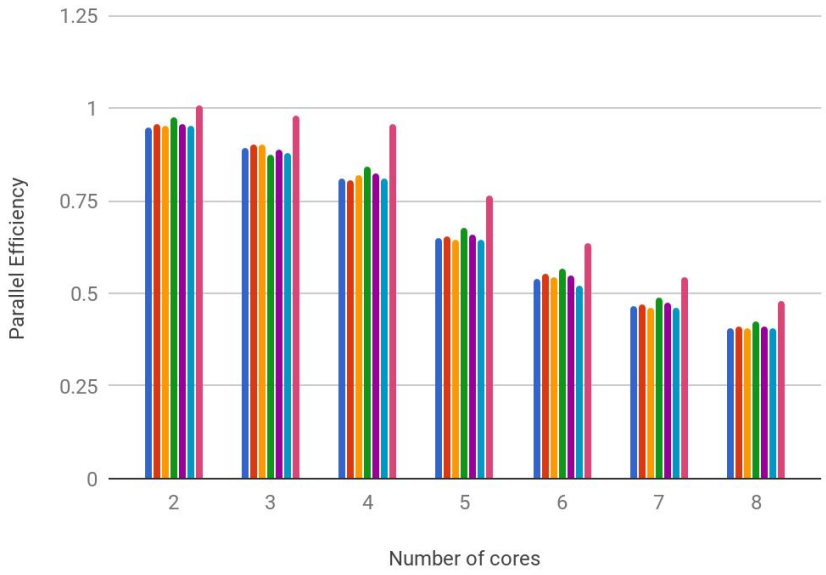
Results: Efficiency of Scheduler



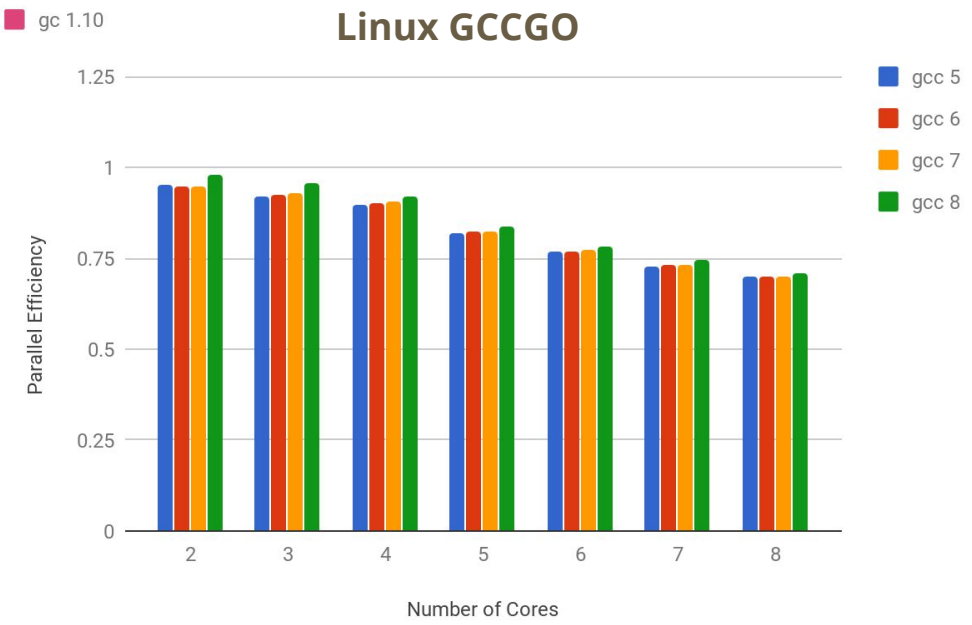
Linux GC



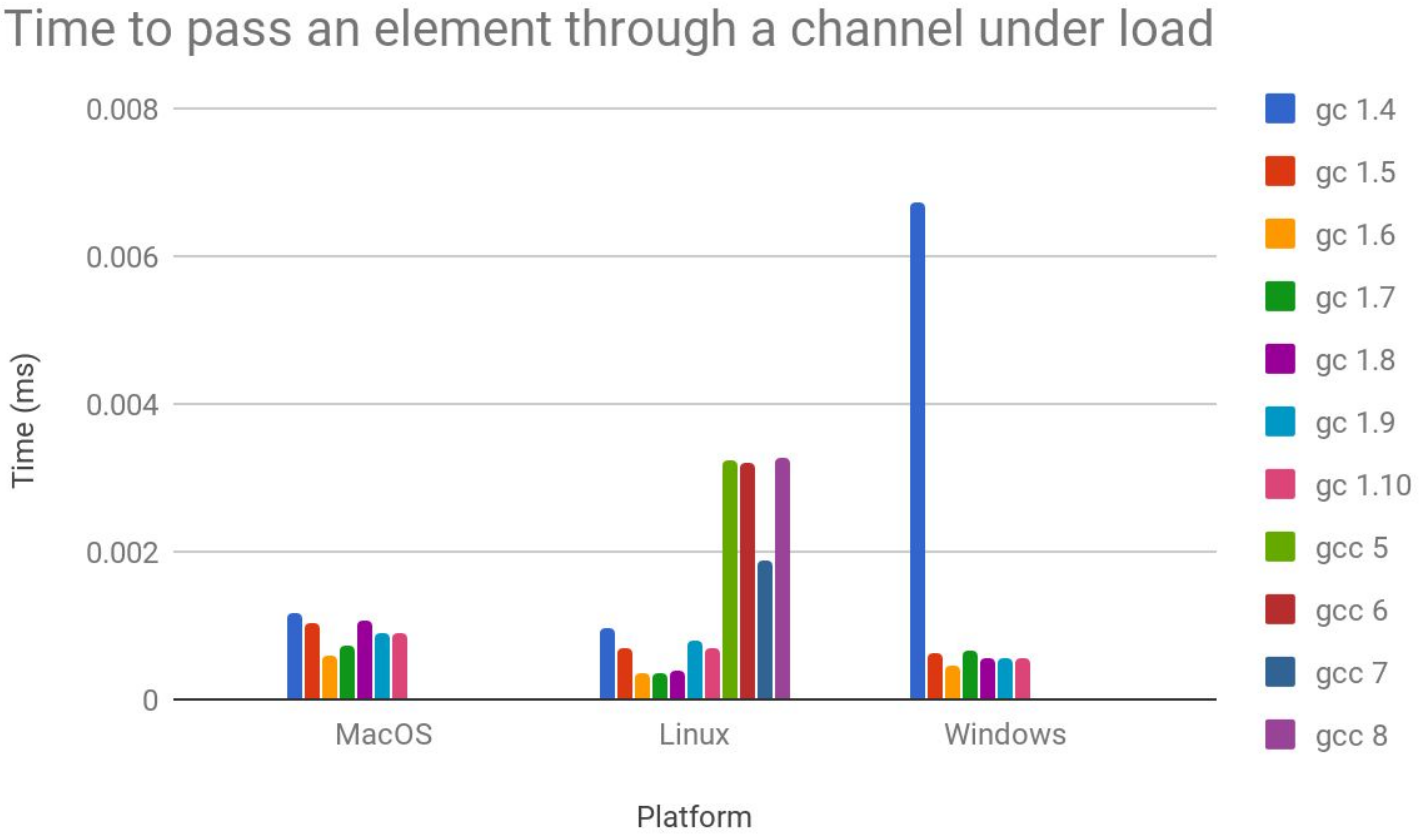
Results: Efficiency of Scheduler



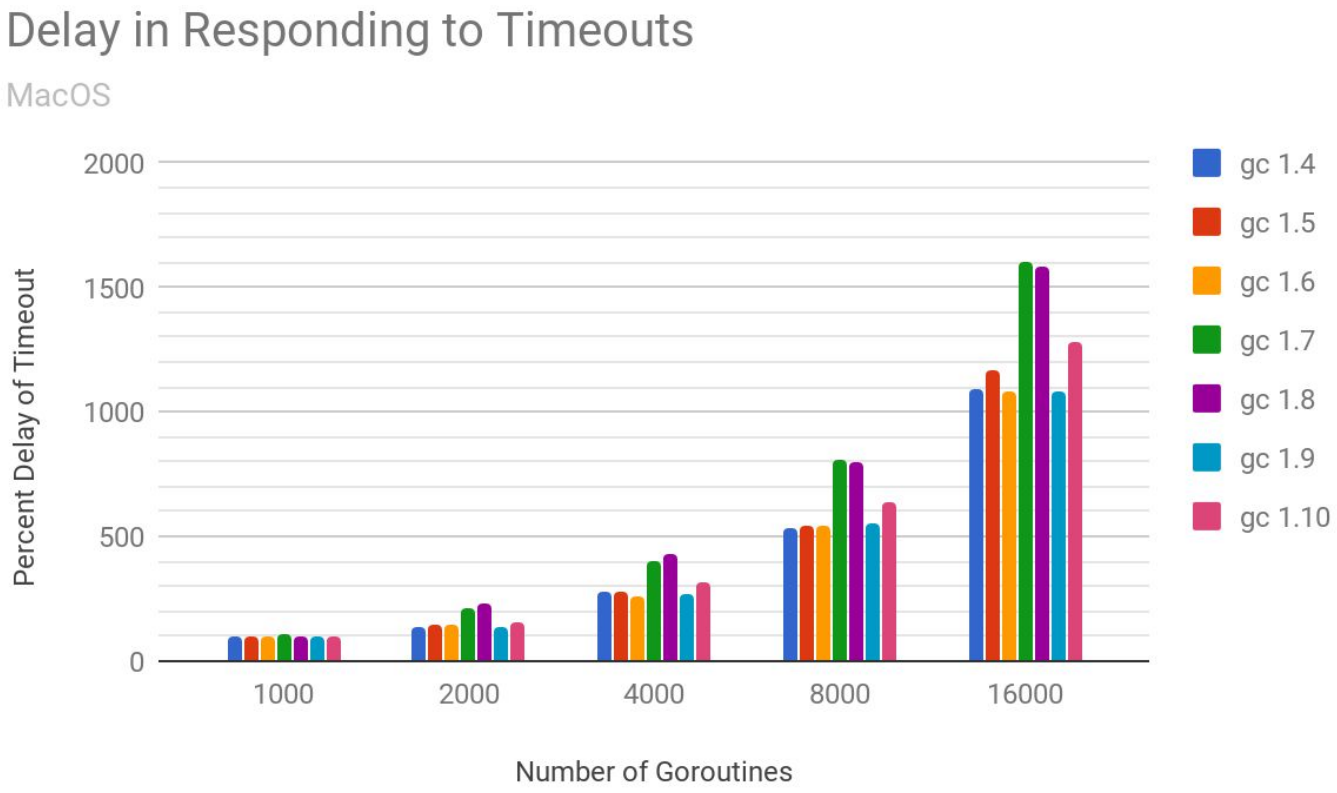
Linux GC



Results: Data Pipeline



Results: Timeout Accuracy Under Load

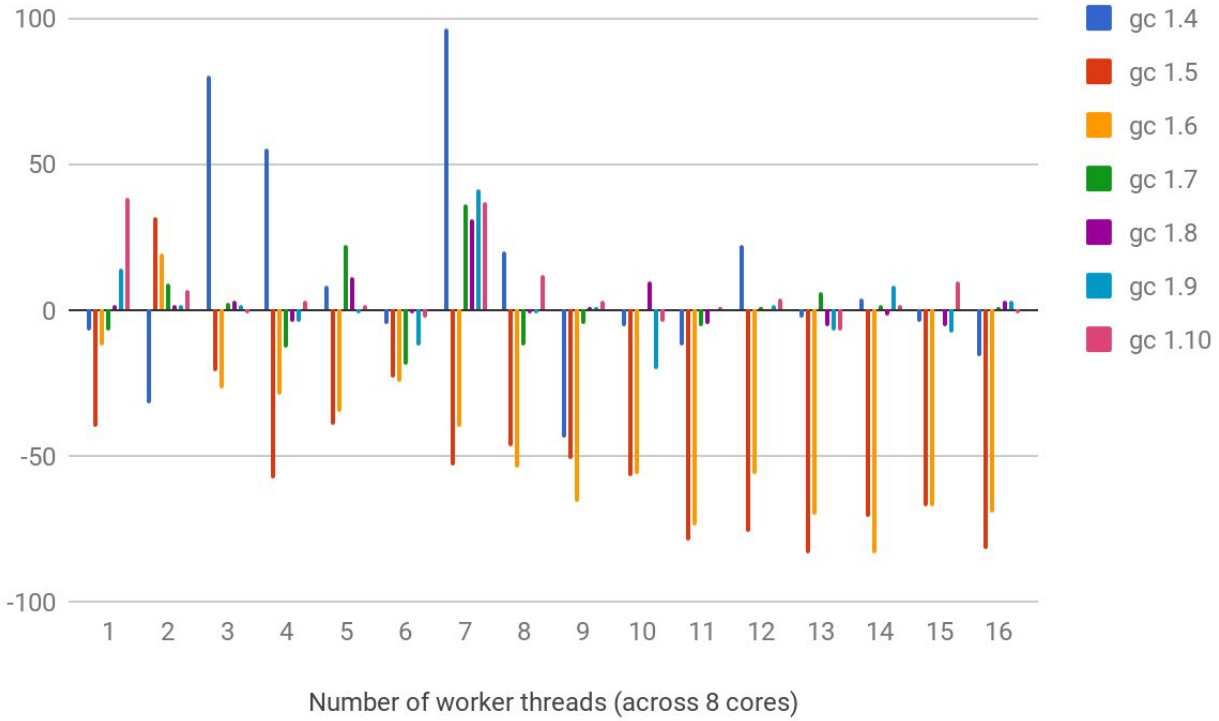


Results: Overhead of Starting Goroutines

Difference in time (ms) between starting many short-lived goroutines and having a small number of persistent worker goroutines for a second-long task.

Positive numbers mean many-short-lived-goroutines took longer.

On modern runtimes, if you're utilizing all your cores, there's no real difference!



Conclusions

- Benchmarking the language implementation gives valuable information to application developers and runtime developers
- Go does in fact have strong support for efficient concurrency
- The basic Go primitives are pretty much all you need - no fancy tricks required to get extra speed

Questions?