ペパボ研究所
Pepabo R&D Institute, GMO Pepabo, Inc.

Optimization for number of goroutines using feedback control

Yusuke MIYAKE / Pepabo R&D Institute, GMO Pepabo, Inc.
2019.07.25 GopherCon 2019

GMOペパボ

Yusuke MIYAKE **@monochromegane**

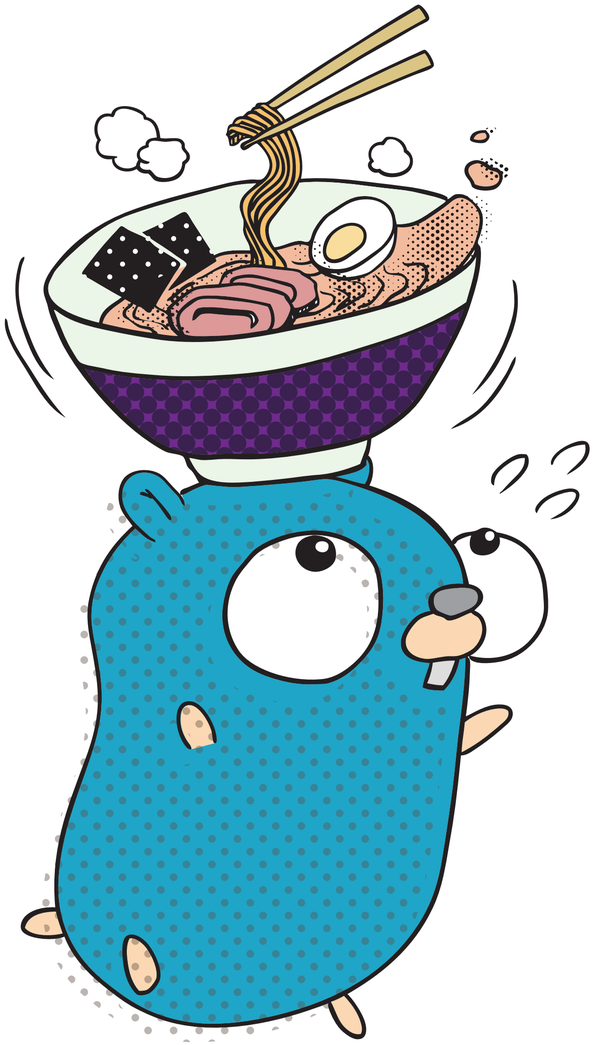
Principal engineer

Pepabo R&D Institute, GMO Pepabo, Inc.

https://blog.monochromegane.com/

Go and I

- **A Japanese gopher who loves writing OSS using Go.**
  - monochromegane/the_platinum_searcher
- **An organizer of the local Go community in Fukuoka, Japan.**
  - Fukuoka.go (https://fukuokago.dev)
- **Held Go Conference'19 summer in Fukuoka on 7/13.**
  - https://fukuoka.gocon.jp

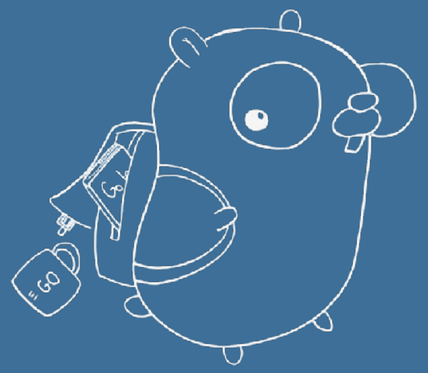


Agenda

1. Introduction
2. Background
3. Proposal
4. Evaluation
5. Conclusion

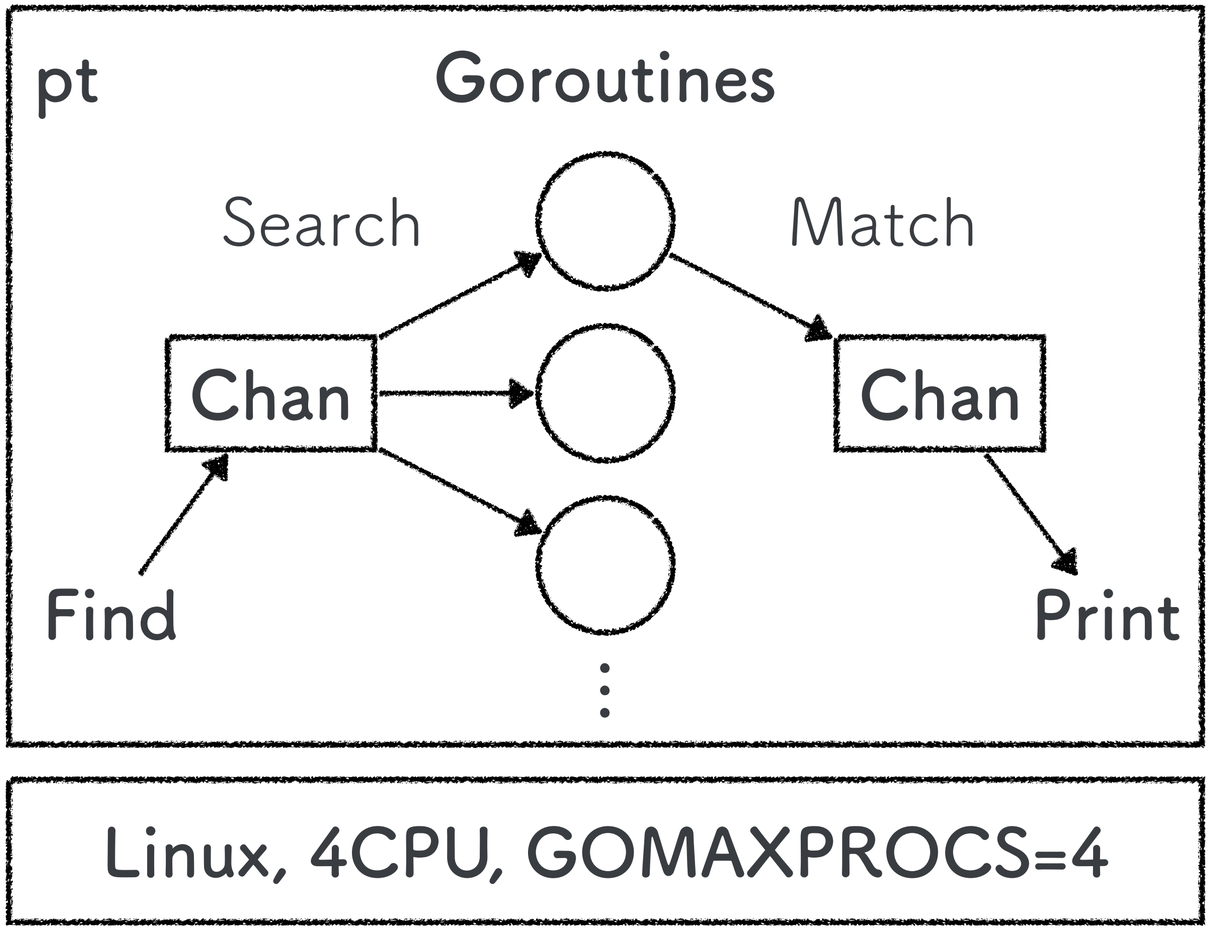
1.

Introduction

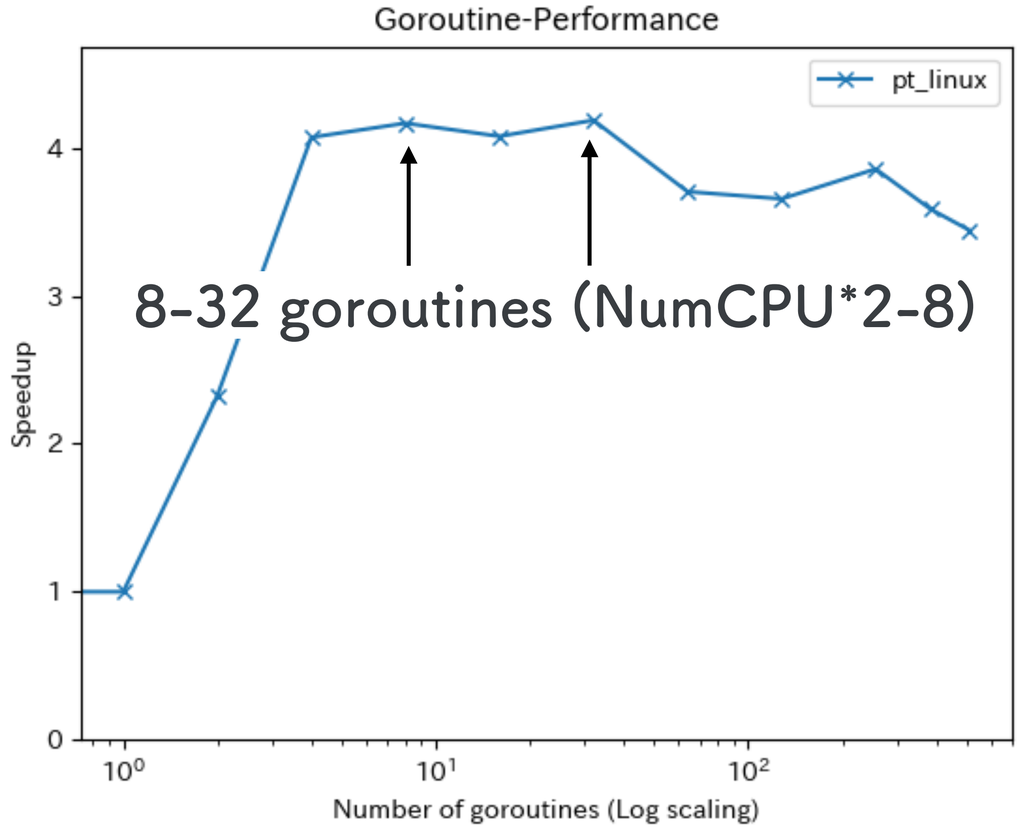


How many is the optimal number of goroutines?

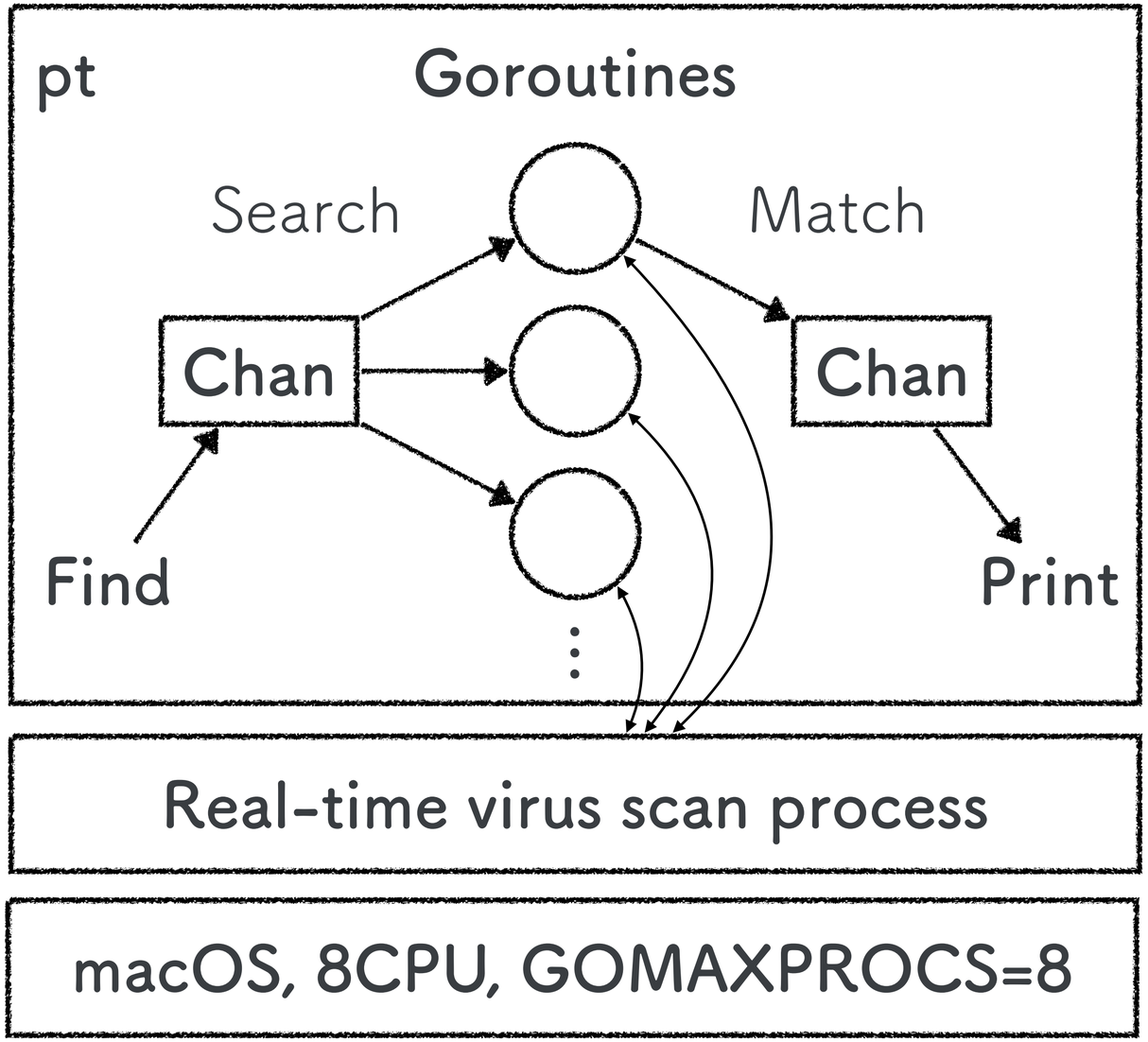
Performance tuning in case of pt

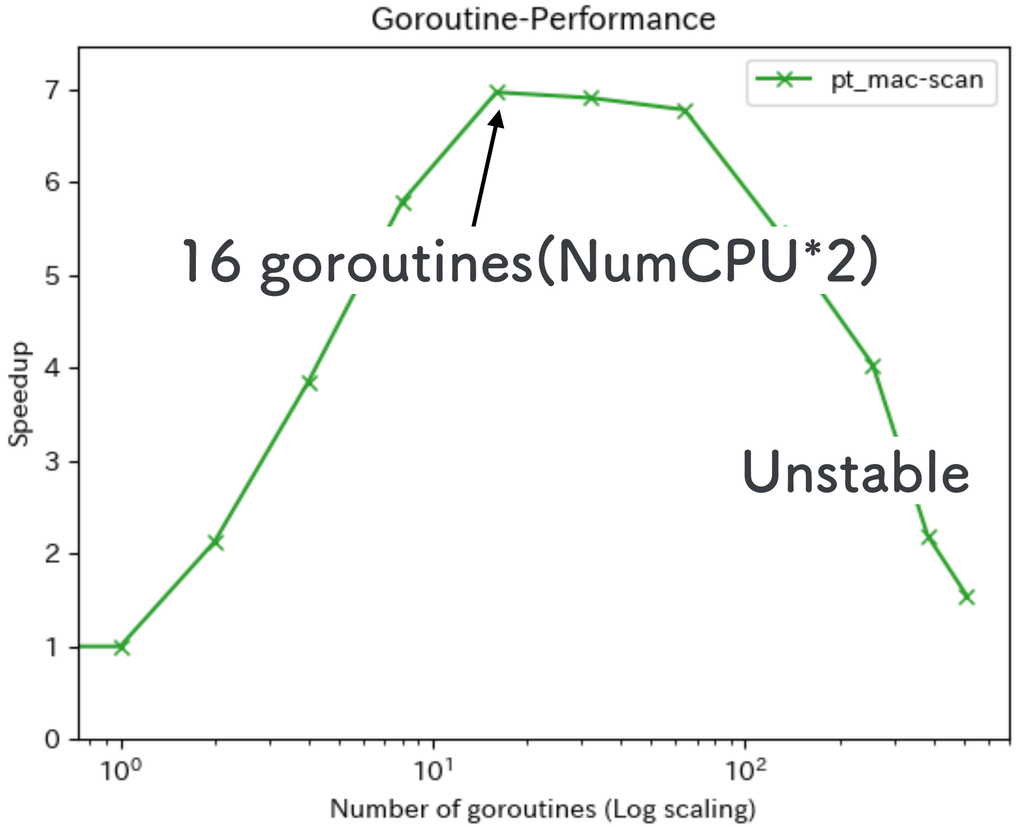


Linux, 4CPU, GOMAXPROCS=4

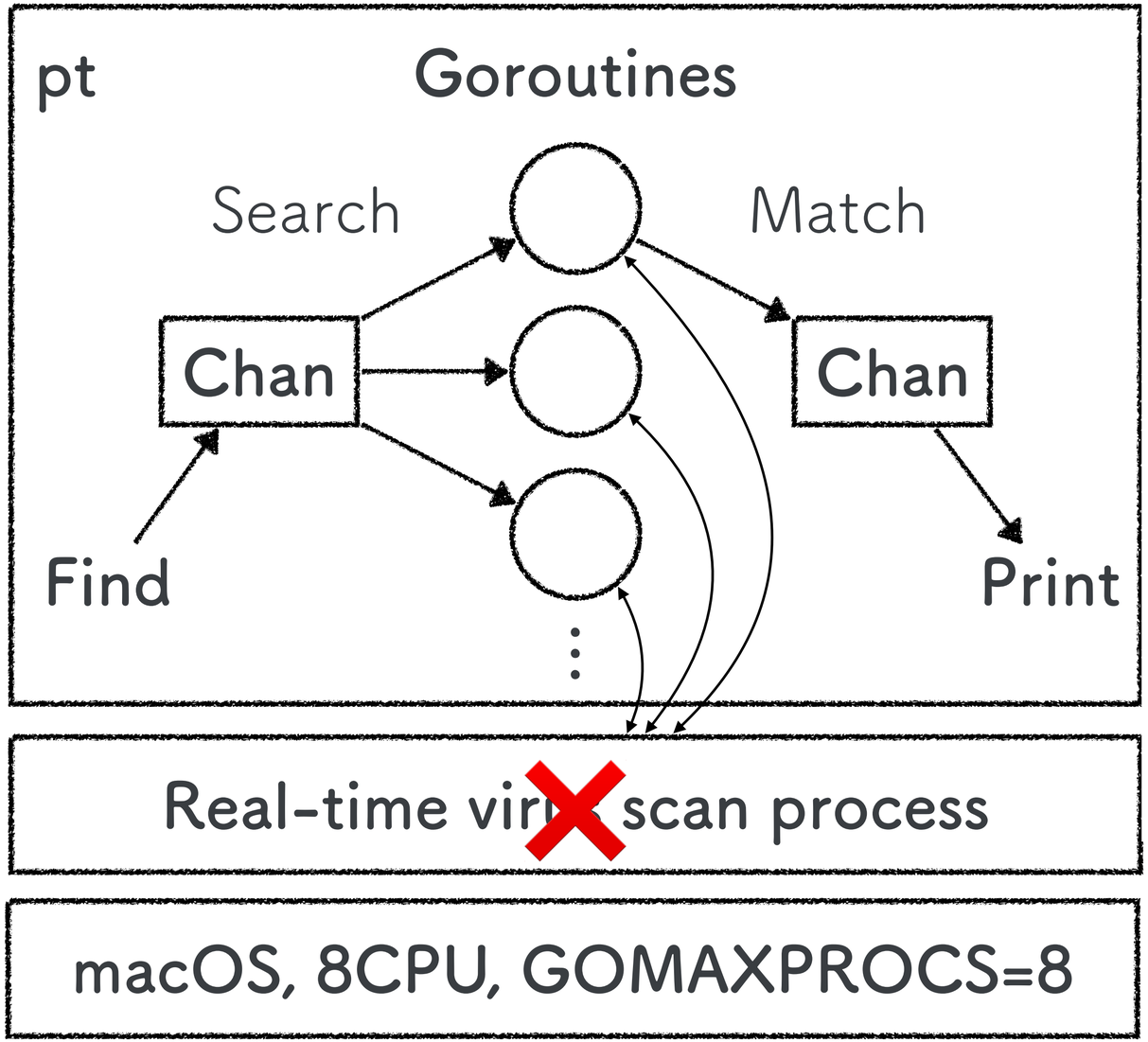


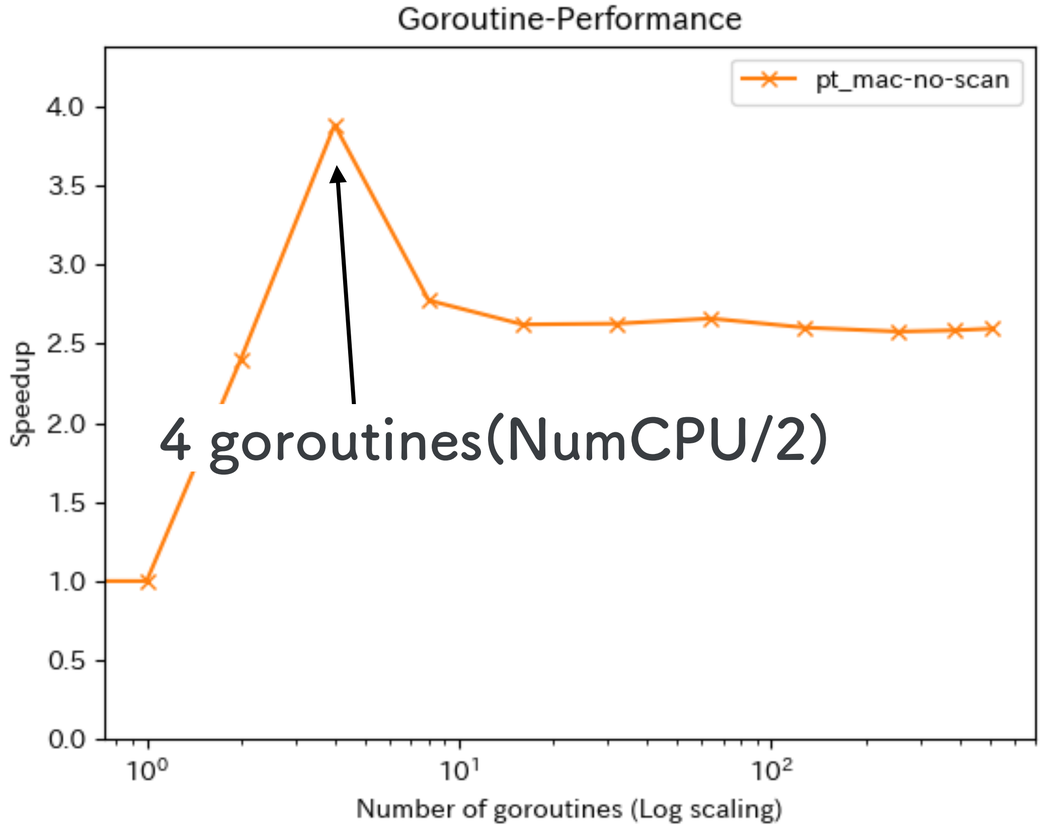
Performance tuning in case of pt





Performance tuning in case of pt





Performance tuning in case of pt

- I chose stability rather than speed…

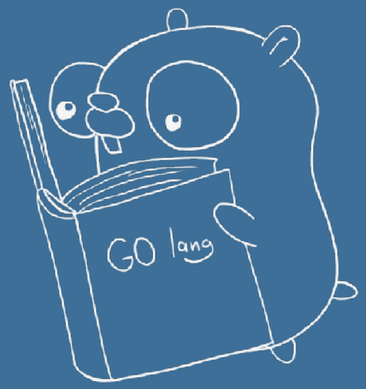
```
29   func (g grep) start() {
30 -       sem := make(chan struct{}, 208)
31         wg := &sync.WaitGroup{}
32 -
33 -       for path := range g.in {
34 -               sem <- struct{}{}
35
36 -               go func(path string) {
37 -                       defer wg.Done()
38 -                       defer func() { <-sem }()
39 -                       g.grepper.grep(path)
40 -               }(path)
41         }
```

```
33   func (g grep) start() {
34
35 +       worker := func() {
36 +               defer wg.Done()
37 +               buf := make([]byte, 16384)
38 +               for path := range g.in {
39 +                       g.grepper.grep(path, buf)
40 +               }
41 +       }
42 +       num := int(math.Max(float64(runtime.NumCPU()), 2.0))
43 +       for i := 0; i < num; i++ {
44                 wg.Add(1)
45 +               go worker()
46         }
```

How many is the optimal number of goroutines in each case?

2.

Background



1. **Concurrency and complexity**
2. Concurrency and Go
3. Concurrency and application

Concurrency and complexity

- Concurrency brings our application good performance and complexity.
- We design a task to have concurrency and request runtime to run them in parallel.

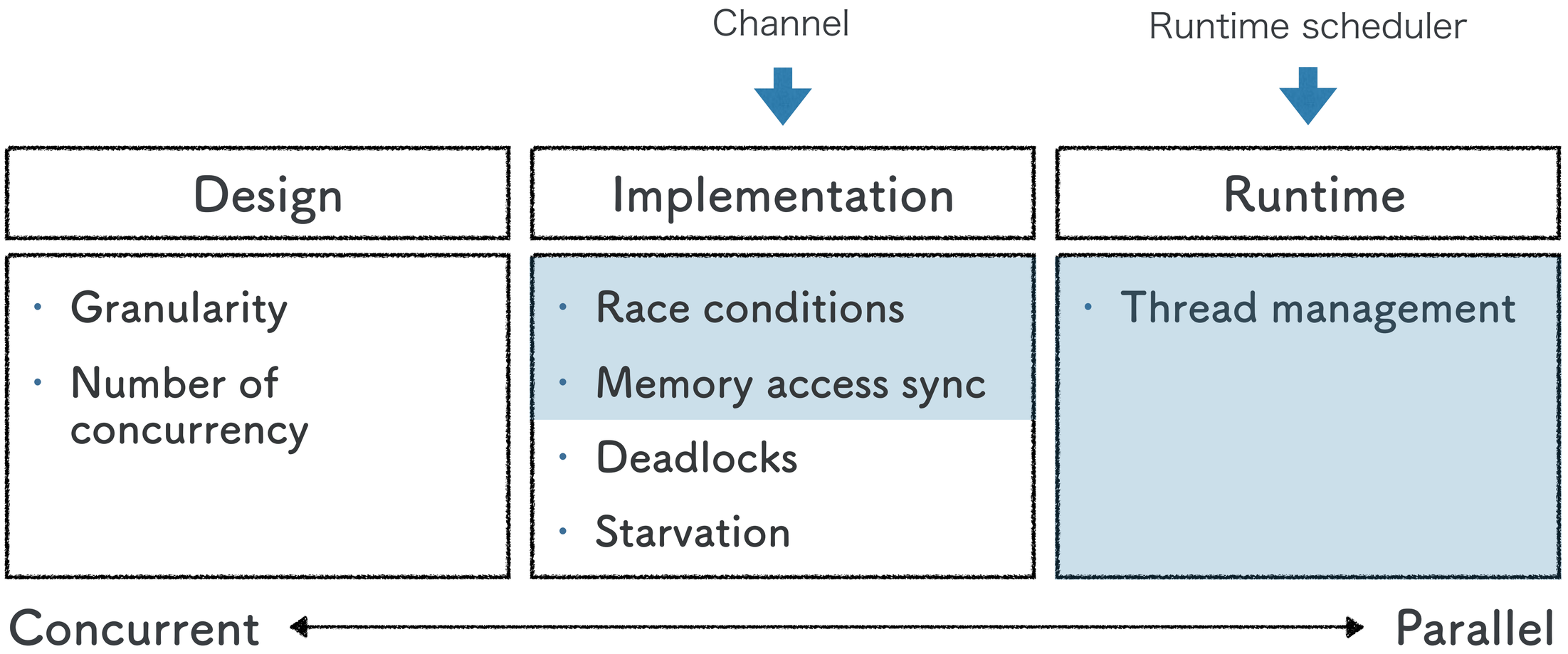
Design	Implementation	Runtime
• Granularity • Number of concurrency	• Race conditions • Memory access sync • Deadlocks • Starvation	• Thread management

Concurrent ⟷ Parallel

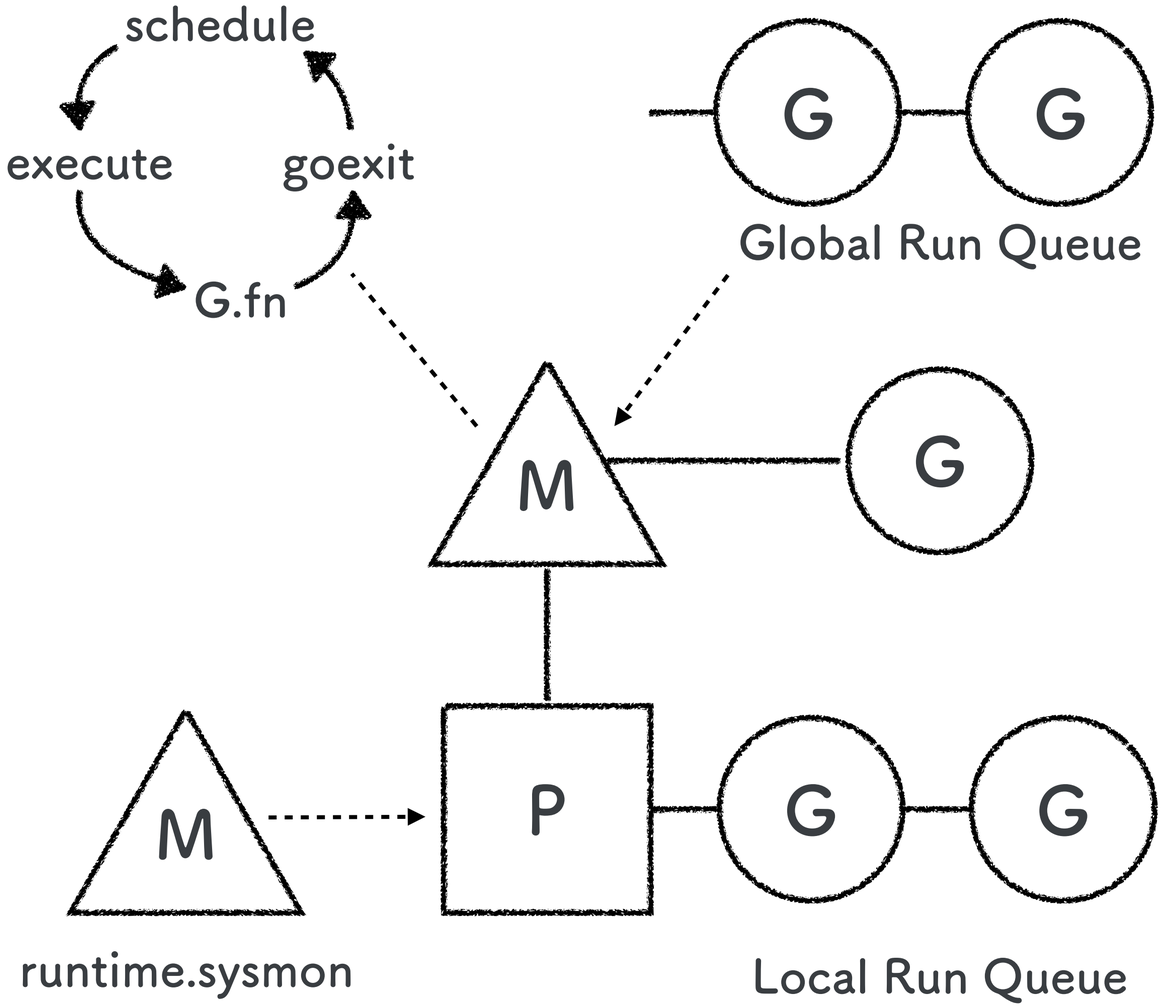
1. Concurrency and complexity
2. **Concurrency and Go**
3. Concurrency and application

Concurrency and Go

- **Go hides the complexity by smart runtime scheduler and features.**



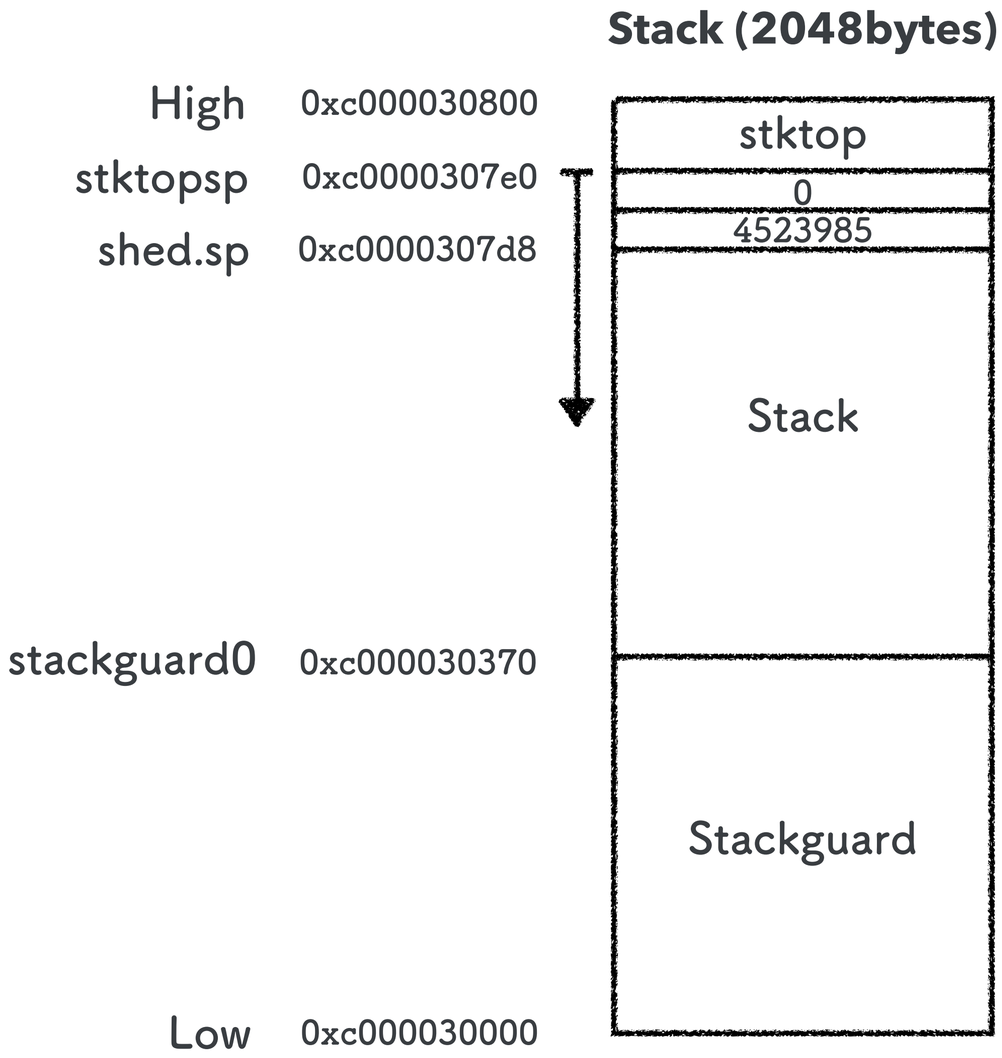
Runtime scheduler of Go



- G: Goroutine
- M: OS thread
- P: Processor (Scheduling context)
  - It provides M:N Scheduler (Some goroutines:Some threads).
  - It has local run queue.
- We can think of Goroutines as application-level threads.
  OS thread behaves like a worker for goroutine using a run queue.

Ref: https://speakerdeck.com/retervision/go-runtime-scheduler

New “g”



Sched (type gobuf)

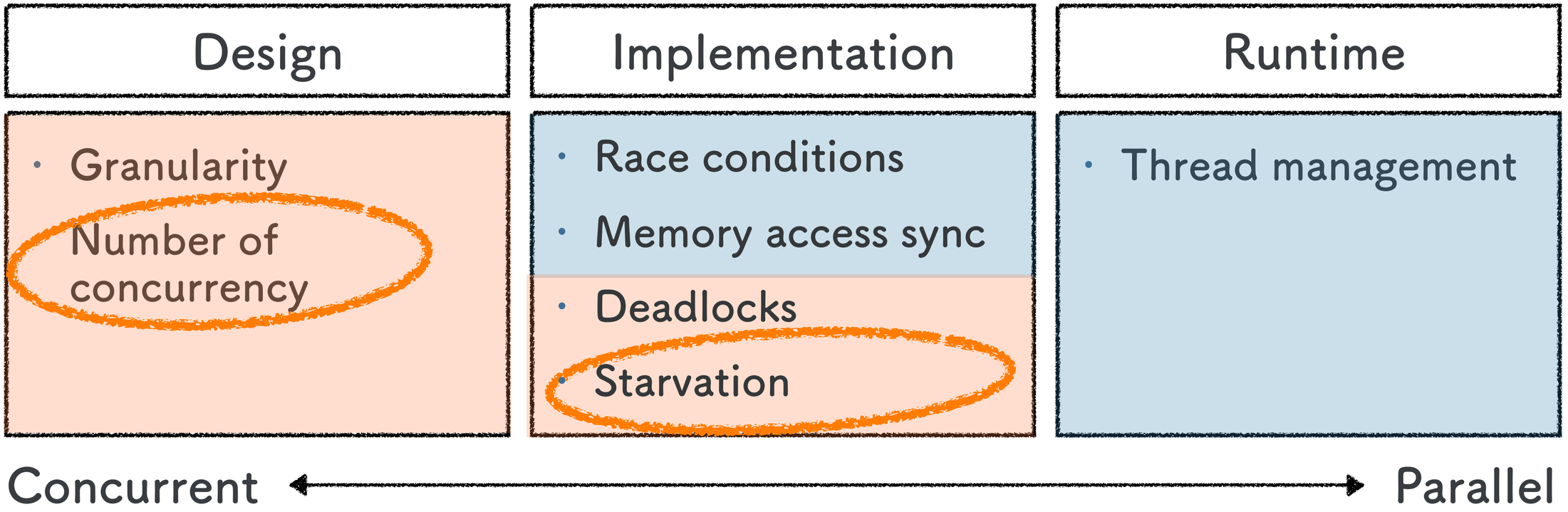
```
type gobuf struct {
    sp   uintptr         // 824633919448 -> 0xc0000307d8
    pc   uintptr         // 4359616 -> 0x4285c0 <runtime.main>
    g    guintptr        // newg
    ctxt unsafe.Pointer // 0x4c96c0 <runtime.mainPC>
    ret  sys.Uintreg.    // 0
    lr   uintptr         // 0
    bp   uintptr         // 0
}
```

- Runtime allocates 2kB stack to “g”.
- “g” has gobuf named “sched”.
- “sched” has goroutines state. (e.g.: stack pointer, program counter…)

1. Concurrency and complexity
2. Concurrency and Go
3. **Concurrency and application**

Concurrency and application

- The design for the number of concurrency is important to achieve both speed and stability.
  - Running many tasks tend to cause starvation.

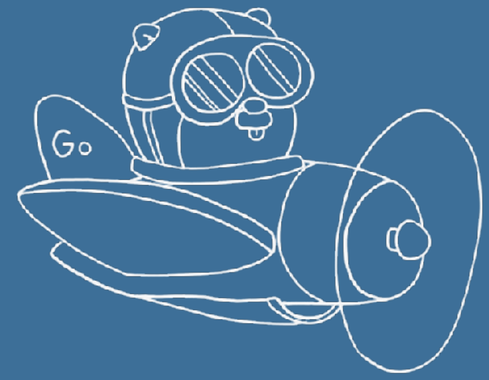


Concurrency and application

- The design for the number of concurrency is difficult.
  - The optimal number depends on the app, environments and load condition.
  - The environment in which the program is tuned and the environment in which it will be executed is different.

- It's desirable for the number of concurrency to be determined dynamically and be controlled rapidly by detection the bottleneck on running program.

3.



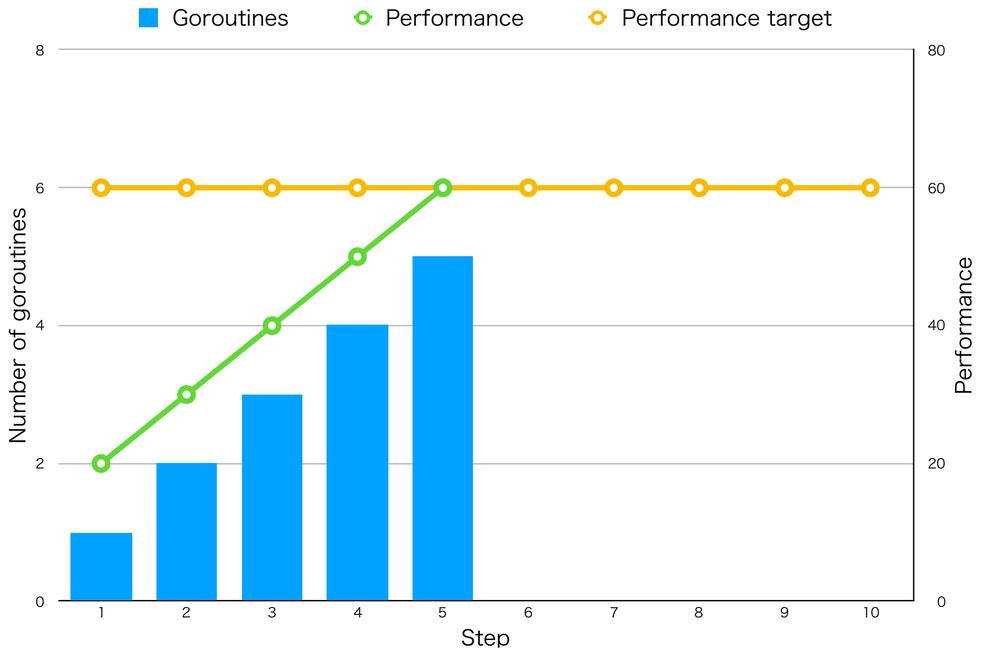
Proposal

Goal

- The optimal number of concurrency to be determined dynamically and be controlled rapidly.

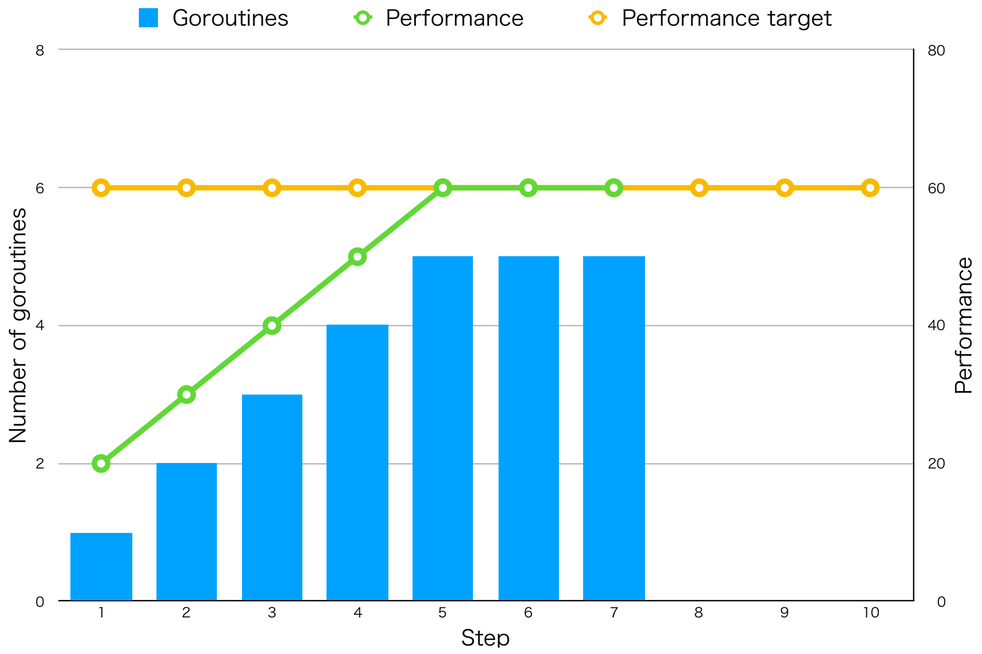
Basic idea

1. Increase the number of goroutines to performance target



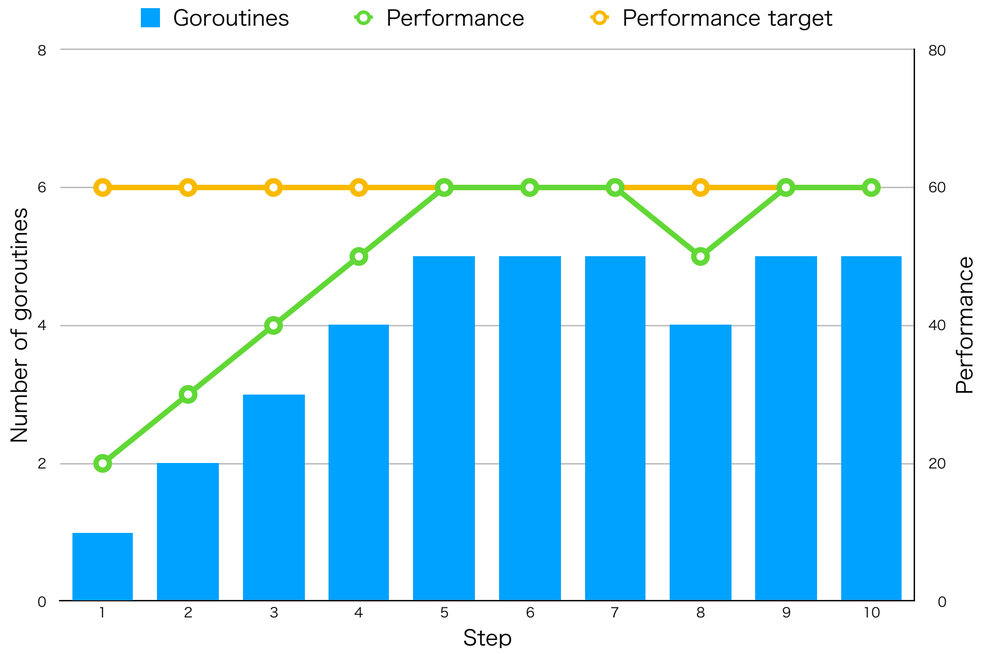
Basic idea

1. Increase the number of goroutines to performance target
2. Stop increasing the number of goroutines if it meets the performance target



Basic idea

1. Increase the number of goroutines to performance target
2. Stop increasing the number of goroutines if it meets the performance target
3. Attempt to decrease the number of goroutines



Issues to solve for the realization

1. Selection of performance metrics
2. Finding how to control rapidly and continuously

Performance metrics

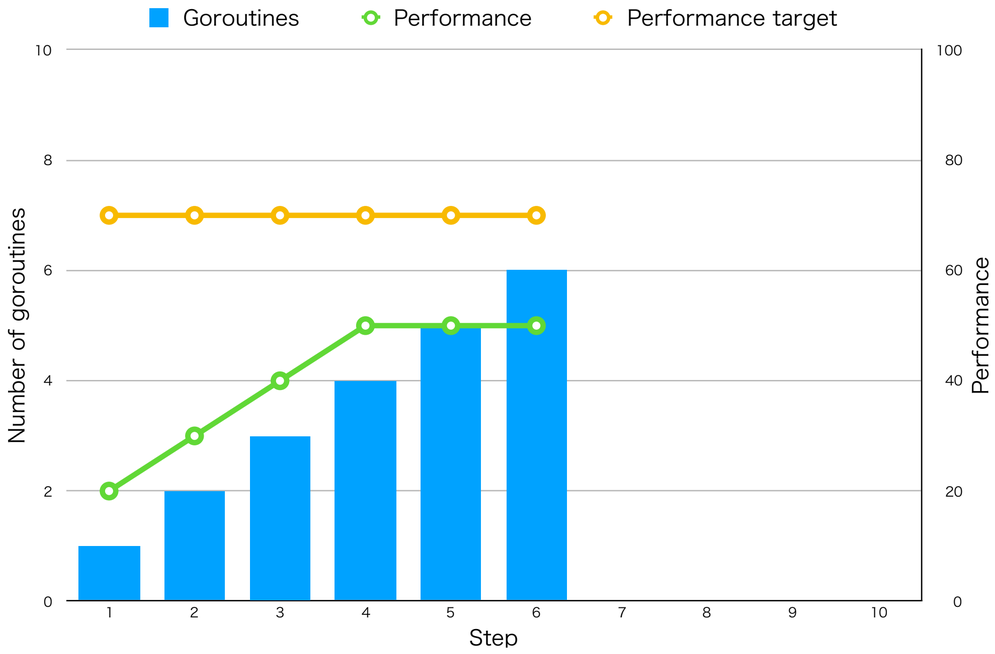
Performance metrics

- Independent resource type which task use.
- e.g.: CPU usage, Throughput.
  - Go's scheduler turns blocking tasks into as CPU bound as possible by switching tasks continuously.

Performance metrics

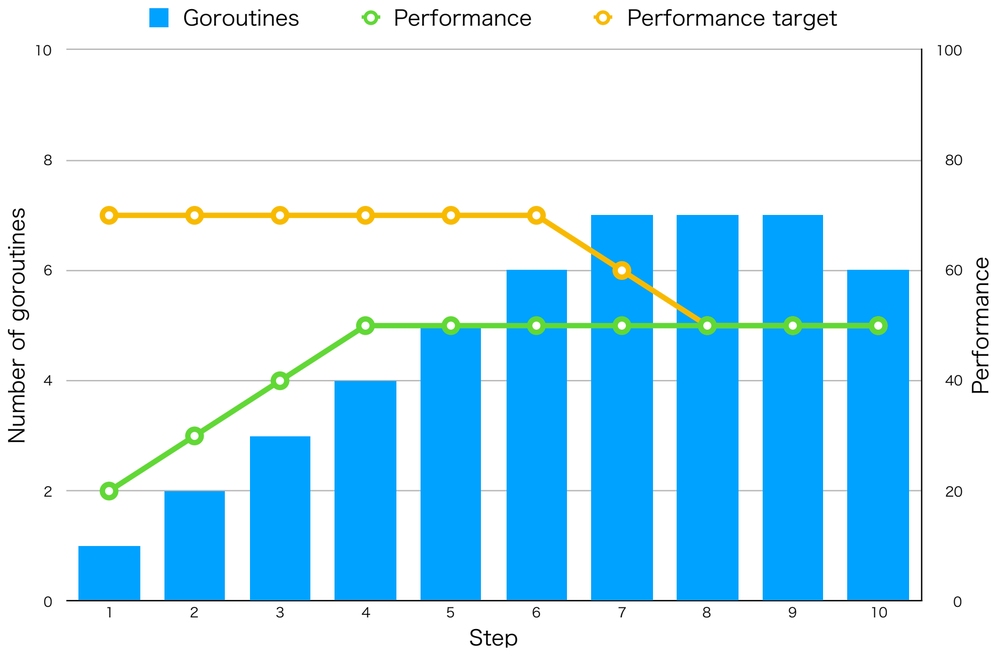
- The metrics upper limit is calculated on running.

  1. Set the target value high



Performance metrics

- The metrics upper limit is calculated on running.
  1. Set the target value high
  2. gradually adjust it to the upper limit value.



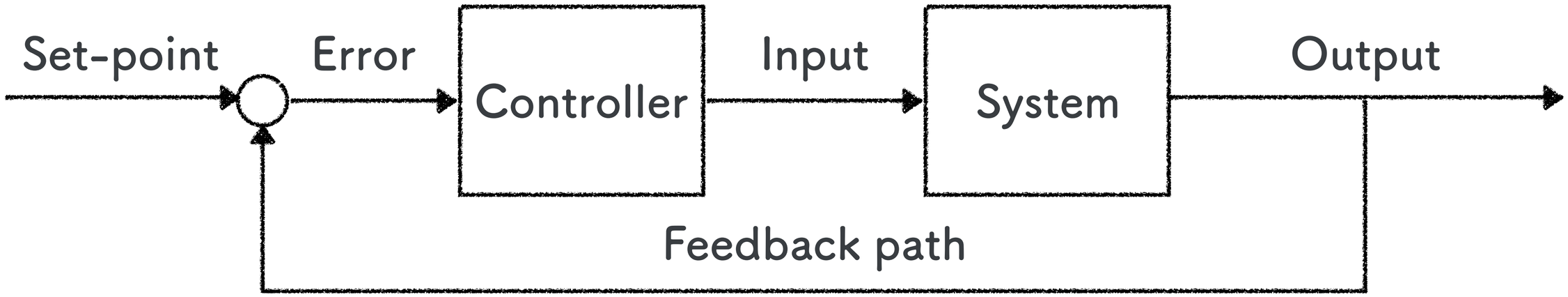
Determining

Determine the number of goroutines

- Determine the number of goroutines based on the metrics
  - Continuously
  - Rapidly
  - Accurately
- Feedback control meets these conditions.

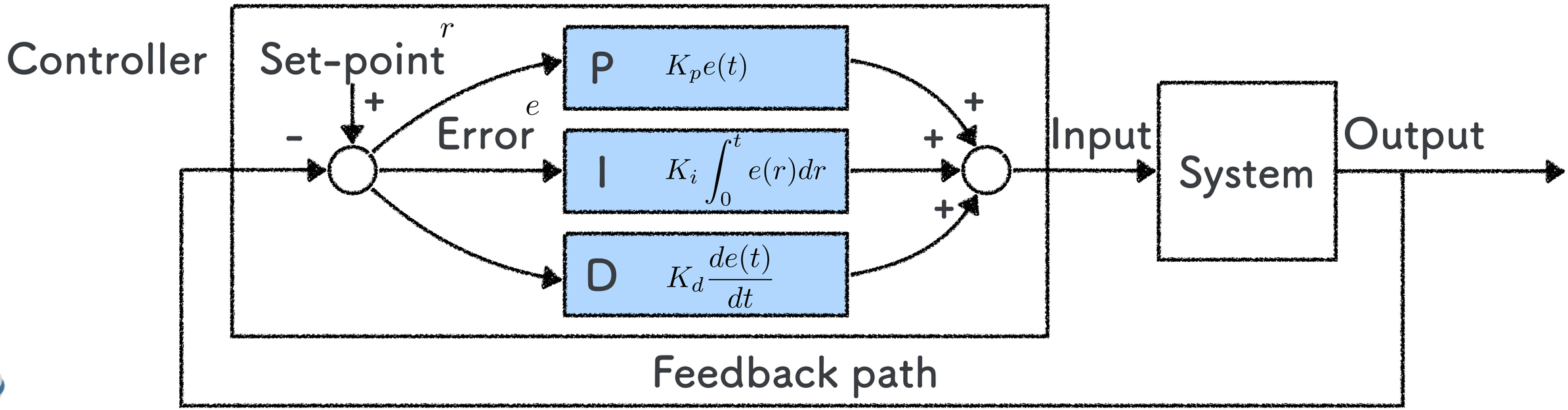
Feedback control

- Tracking a given set-point using errors from the set-point.
  - Applying an automatic correction continuously.
- Identifying suitable set-point and input/output is important.
  - Dealing with the system as a black box.



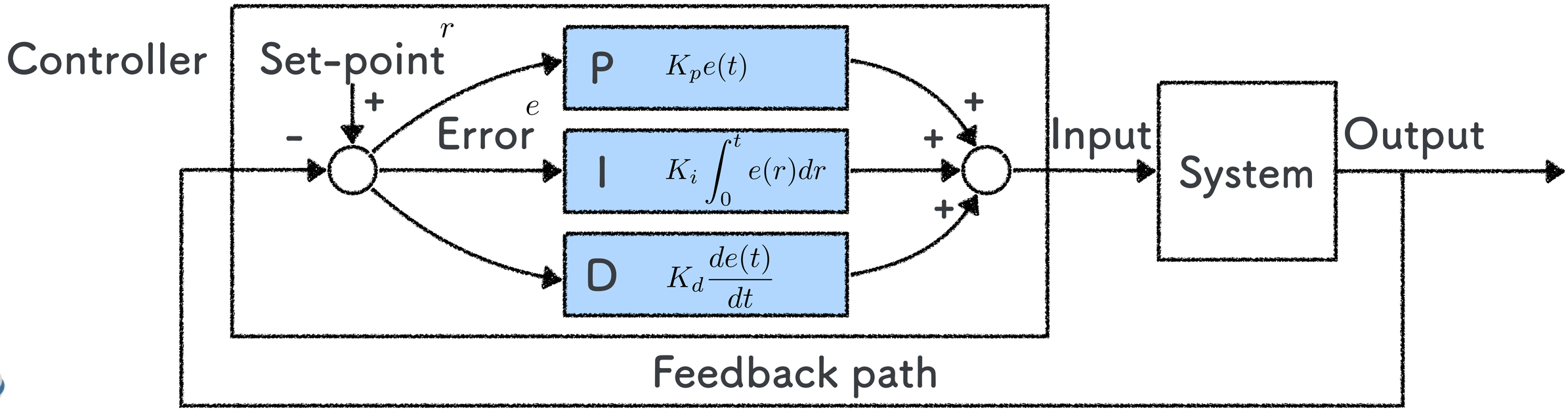
PID Controller (1/2)

- Applying a correction rapidly and accurately.
- The output is a combination of its {Proportional, Integral, Derivative} control.



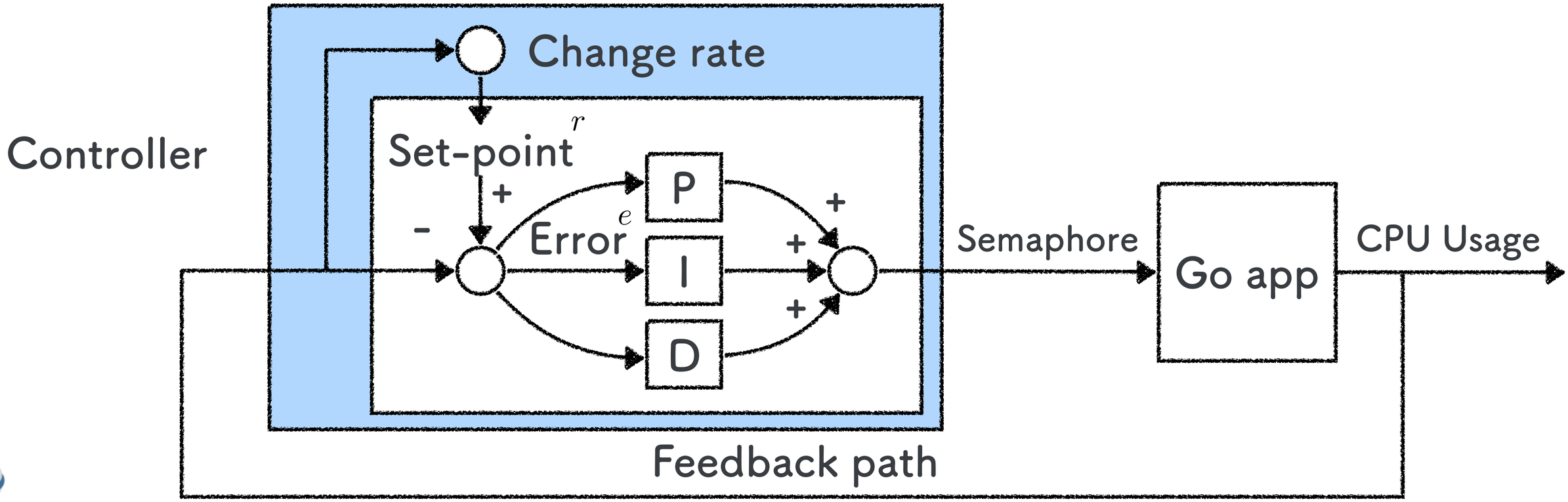
PID Controller (2/2)

- Proportional control(P) output is proportional to the error.
- Integral control(I) output is proportional to the integral of the error over time.
- Derivative control(D) output is proportional to the derivative of the error.



Dynamic Target Controller

- Having two nested control loops to change set-point on running.
- The outer loop sets new set-points of the inner loop when CPU usage reaches the upper limit.



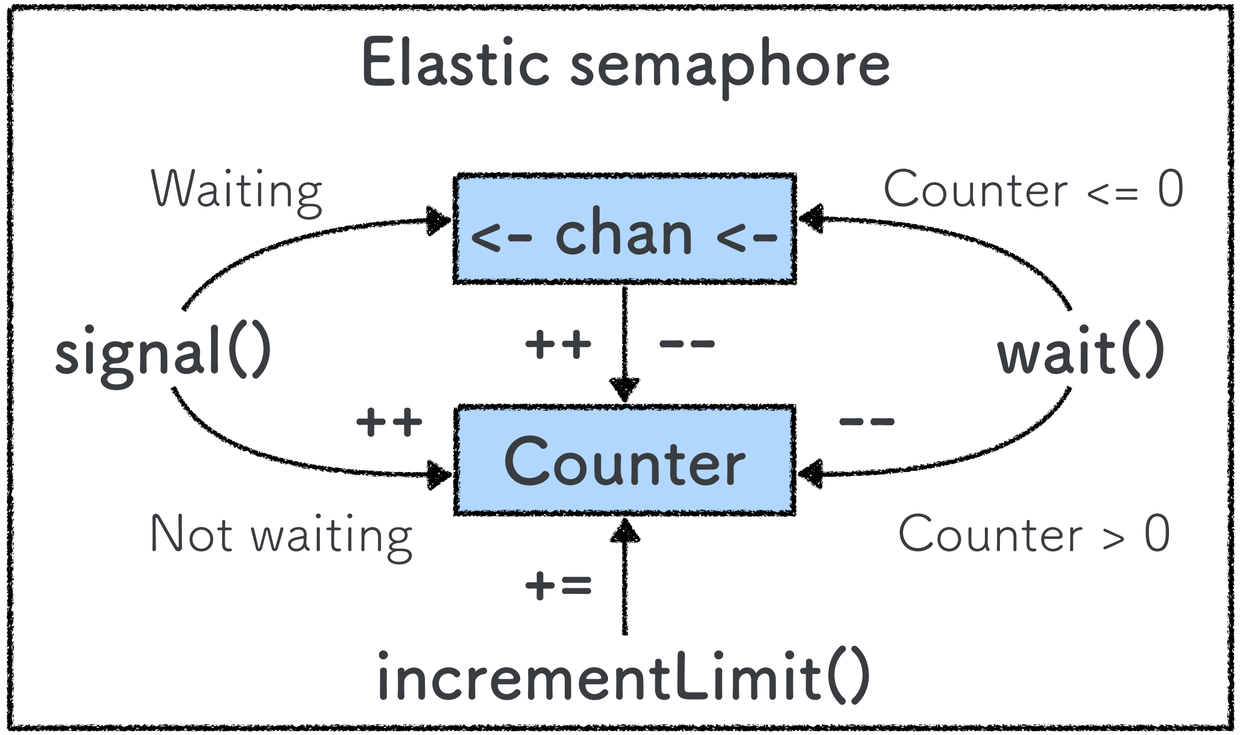
Bounding

To bound concurrency in Go

```
func main() {
  var wg sync.WaitGroup
  sem := make(chan struct{}, 3)
  for i := 0; i < 10; i++ {
    wg.Add(1)
    sem <- struct{}{}
    go func() {
      defer wg.Done()
      defer func() { <-sem }()
      task()
    }()
  }
  wg.Wait()
  close(sem)
}
```

Bounding concurrency using buffered channel as semaphore

Elastic semaphore



- wait() & signal() as same as semaphore
- incrementLimit() changes upper of semaphore variable
- Ensure atomicity of above operations

Kaburaya

Architecture of kaburaya

- The optimal number of concurrency to be determined dynamically and be controlled rapidly
  - Use CPU usage upper limit as performance metrics.
  - Use feedback control to determine the number of goroutines continuosly.
  - Use elastic semaphore to bound concurrency.
- Kaburaya is “鏑矢” which is a Japanese arrow with a whistle.
  - https://github.com/monochromegane/kaburaya

monochromegane/kaburaya

- Kaburaya optimizes the number of goroutines by feedback control.

```
func main() {
  sem := kaburaya.NewSem(100 * time.Millisecond)
  var wg sync.WaitGroup
  for i := 0; i < 10; i++ {
    wg.Add(1)
    sem.Wait()
    go func() {
      defer sem.Signal()
      defer wg.Done()
      task()
    }()
  }
  wg.Wait()
  sem.Stop()
}
```

Create semaphore and specify time span for update the semaphore and CPU target.

Wait for decrement operation.

Increment operation.

Stop update the semaphore and CPU target.

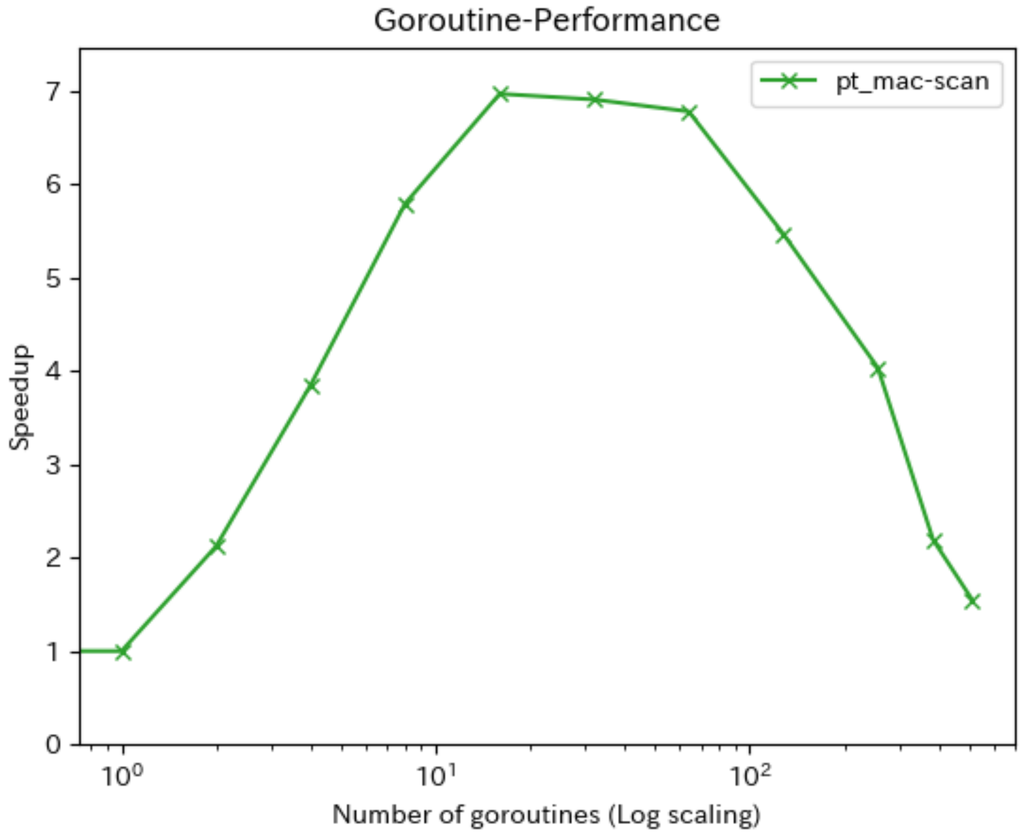
4.

Evaluation

Evaluation

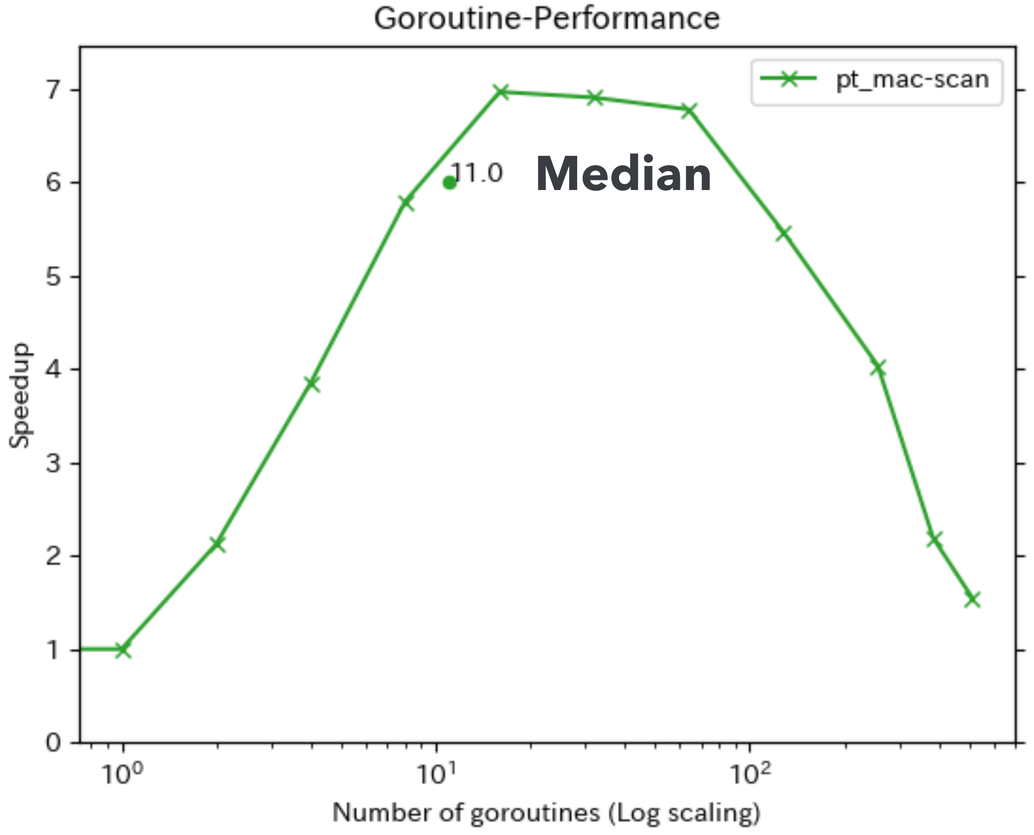
- Apply kaburaya to some tasks to evaluate efficacy.
- Measure speedup, CPU usage, and limit of semaphore.
- Find good and not so good cases for the kaburaya controller.

Task: pt_mac-scan

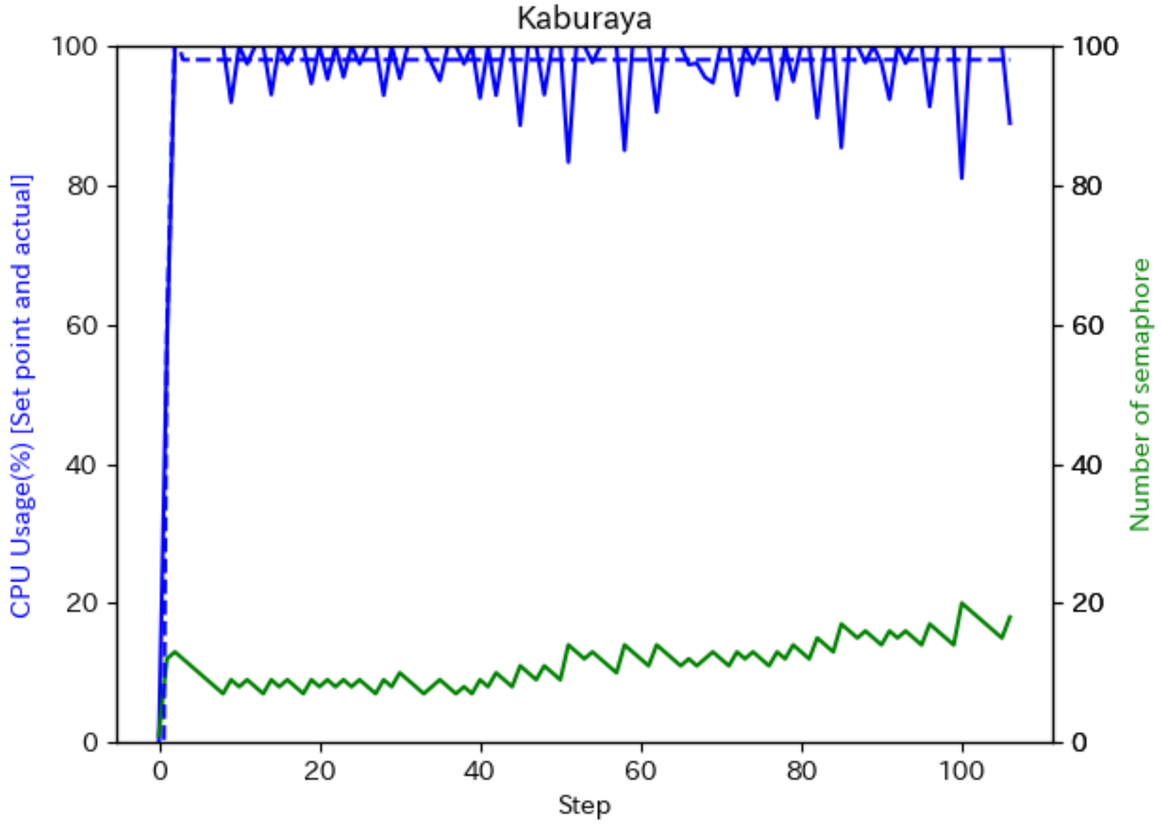


- pt runs on macOS with realtime virus scan
- CPU 8, Memory 16GB
- GOMAXPROCS=8
- Better concurrent are from 16 to 64.
- More degradation of performance due to increased goroutines.

Task: pt_mac-scan (Good)

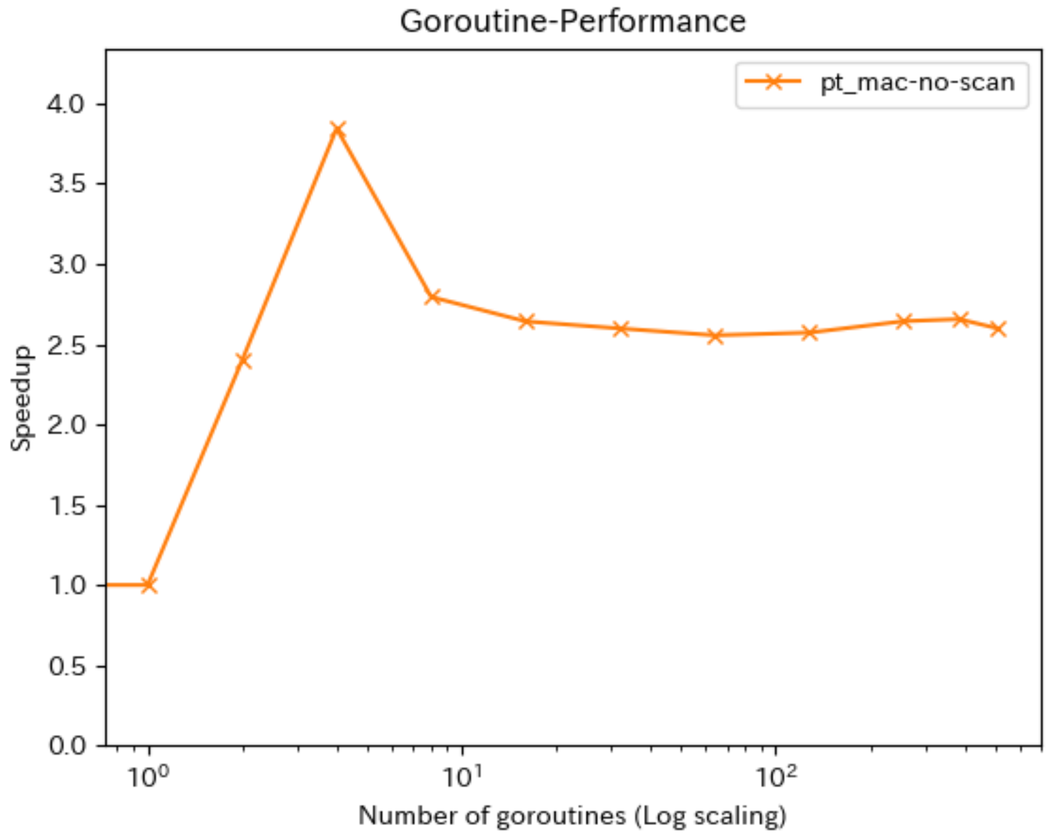


Duration: 50ms, Gain: 0.1, Change rate: ±0.3



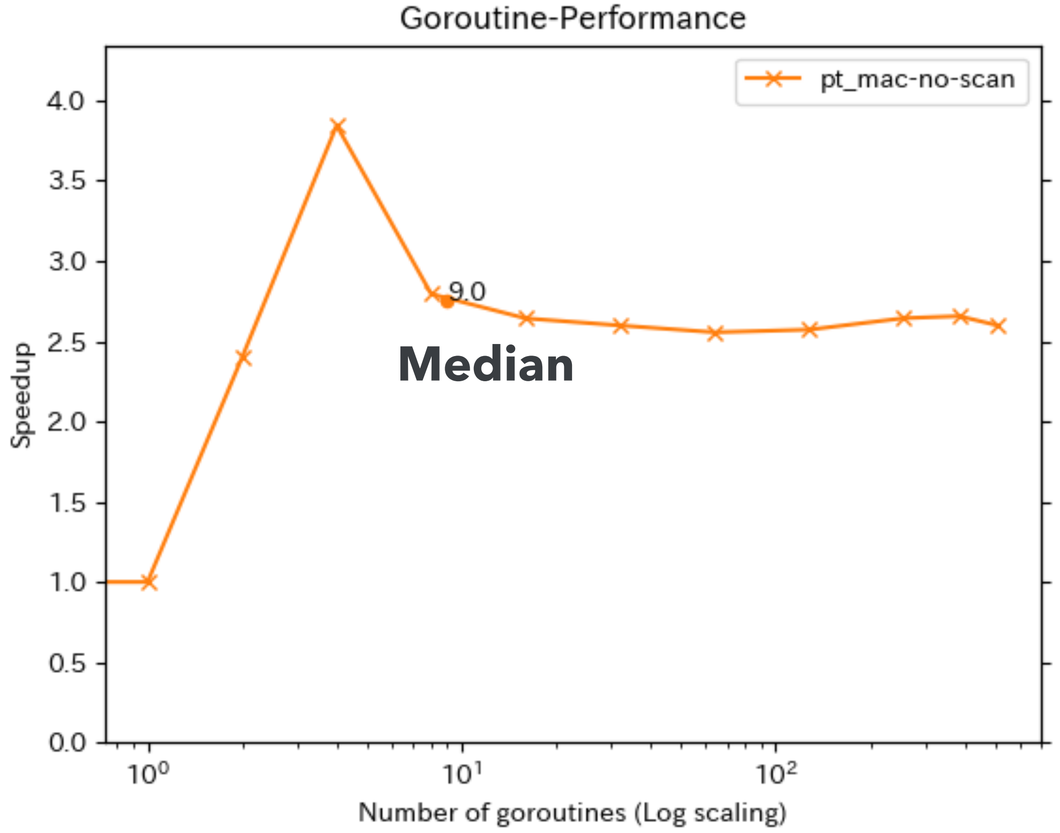
- Kaburaya found good set points and adjust numbers within ideal range.

Task: pt_mac-no-scan

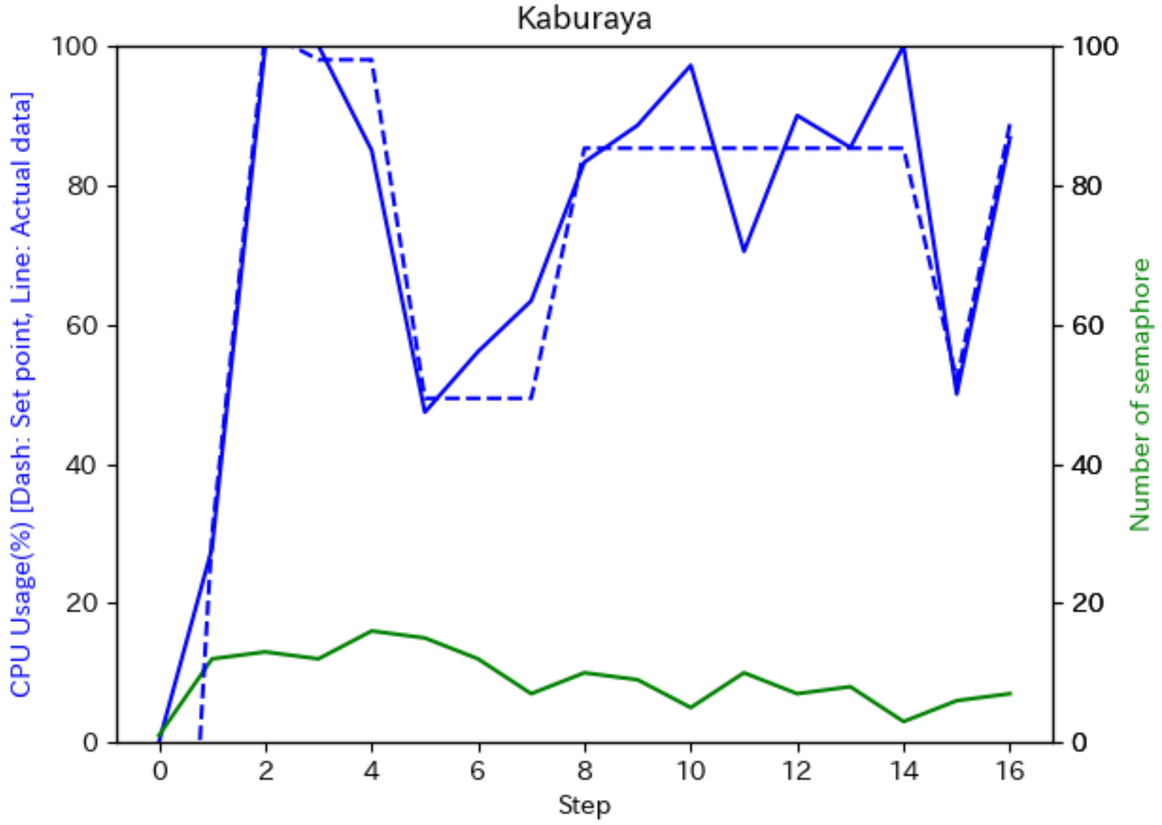


- Runs pt on Mac <u>without</u> realtime virus scan
- CPU 8, Memory 16GB
- GOMAXPROCS=8
- Better concurrent is 4
- Less degradation of performance due to increased goroutines.

Task: pt_mac-no-scan (Good)

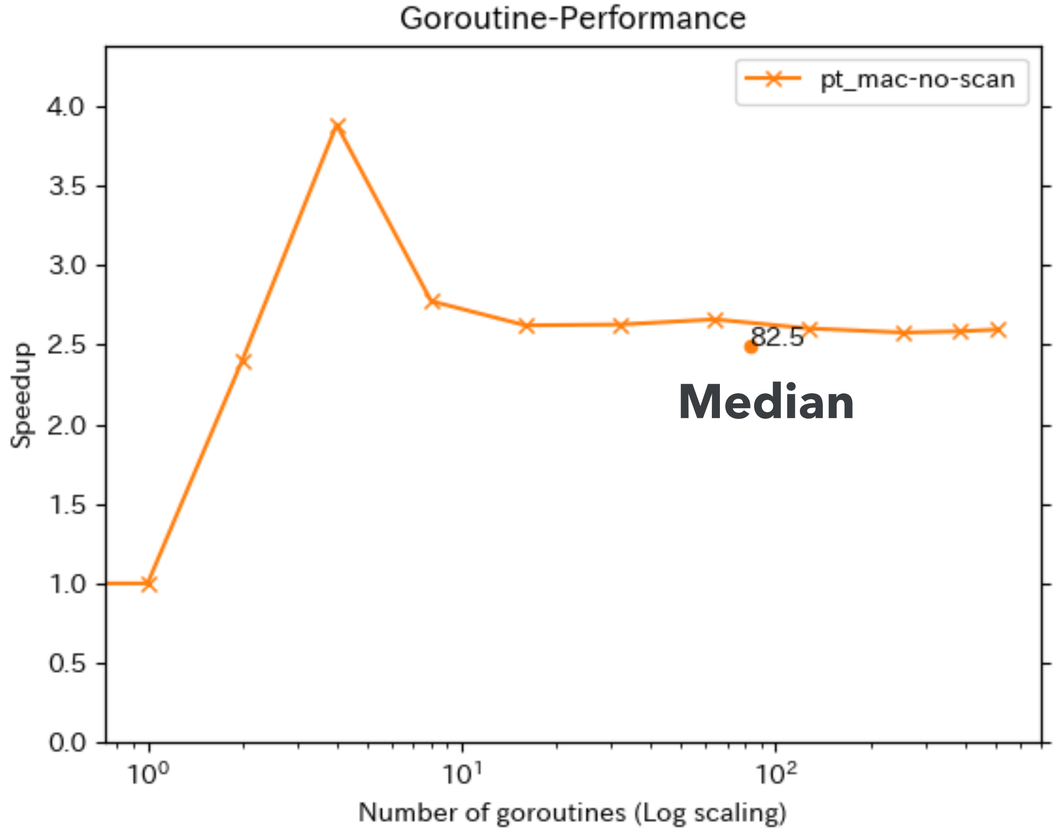


Duration: 50ms, Gain: 0.1, Change rate: ±0.3

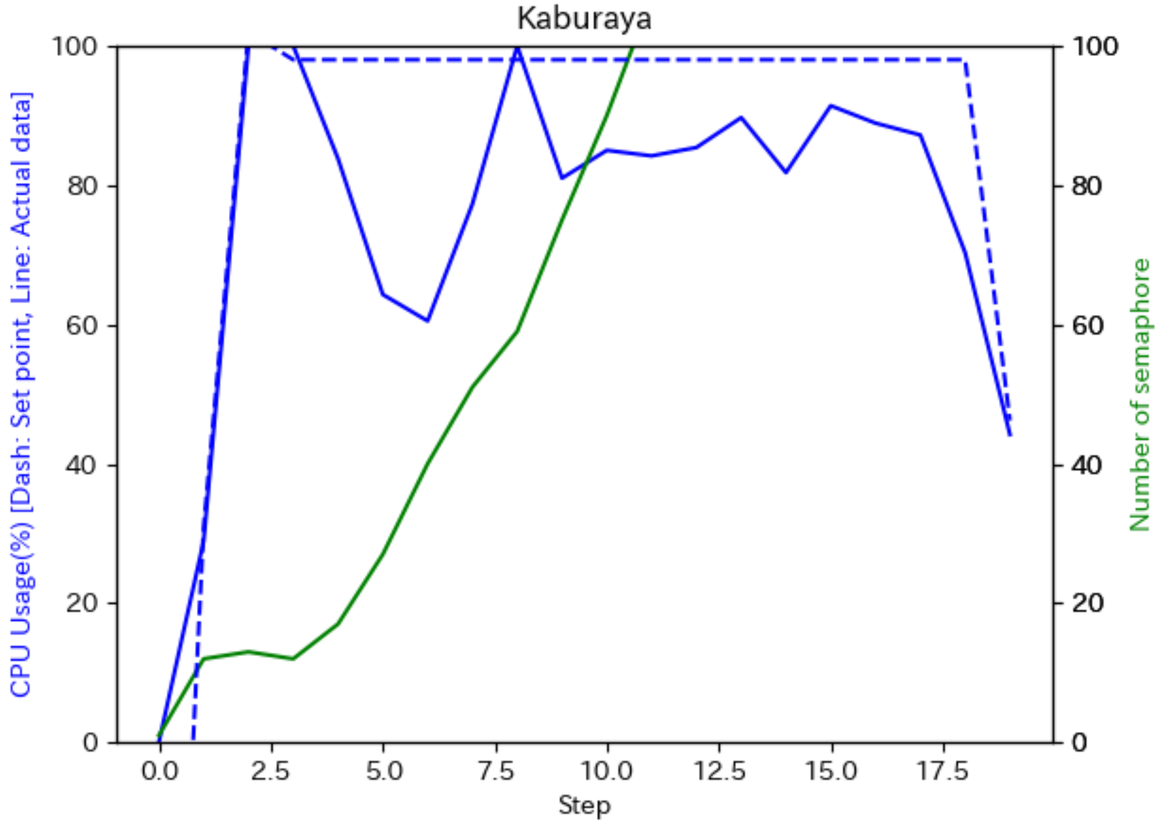


- Kaburaya found good set points and adjust numbers within ideal range.

Task: pt_mac-no-scan (Bad)

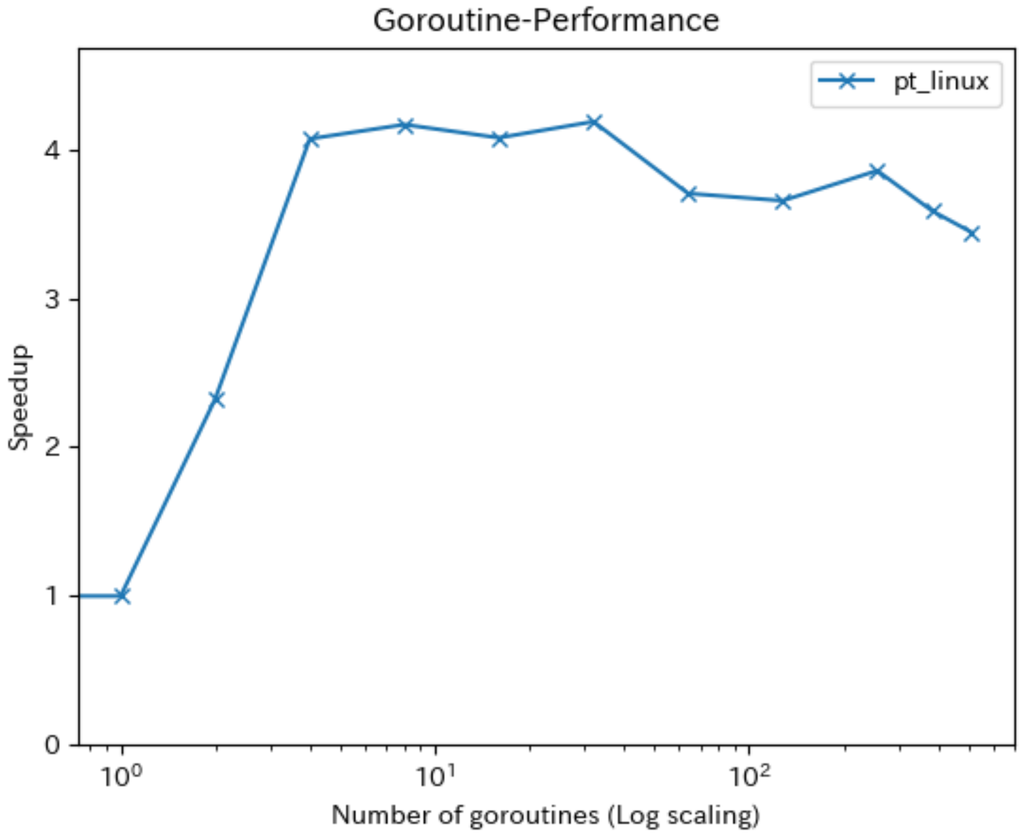


Duration: 50ms, Gain: 0.1, Change rate: ±0.3



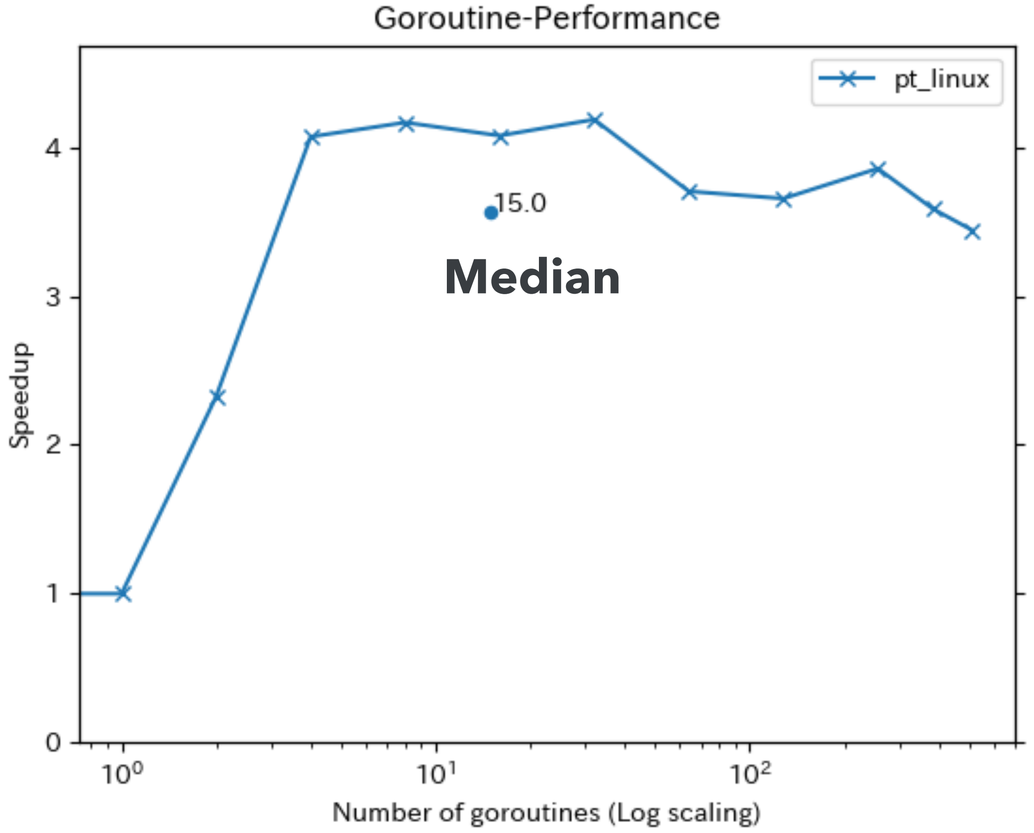
- Find set points and adjust numbers within ideal range.
- Increase the number due to failure to reset the set point at end of period.

Task: pt_linux

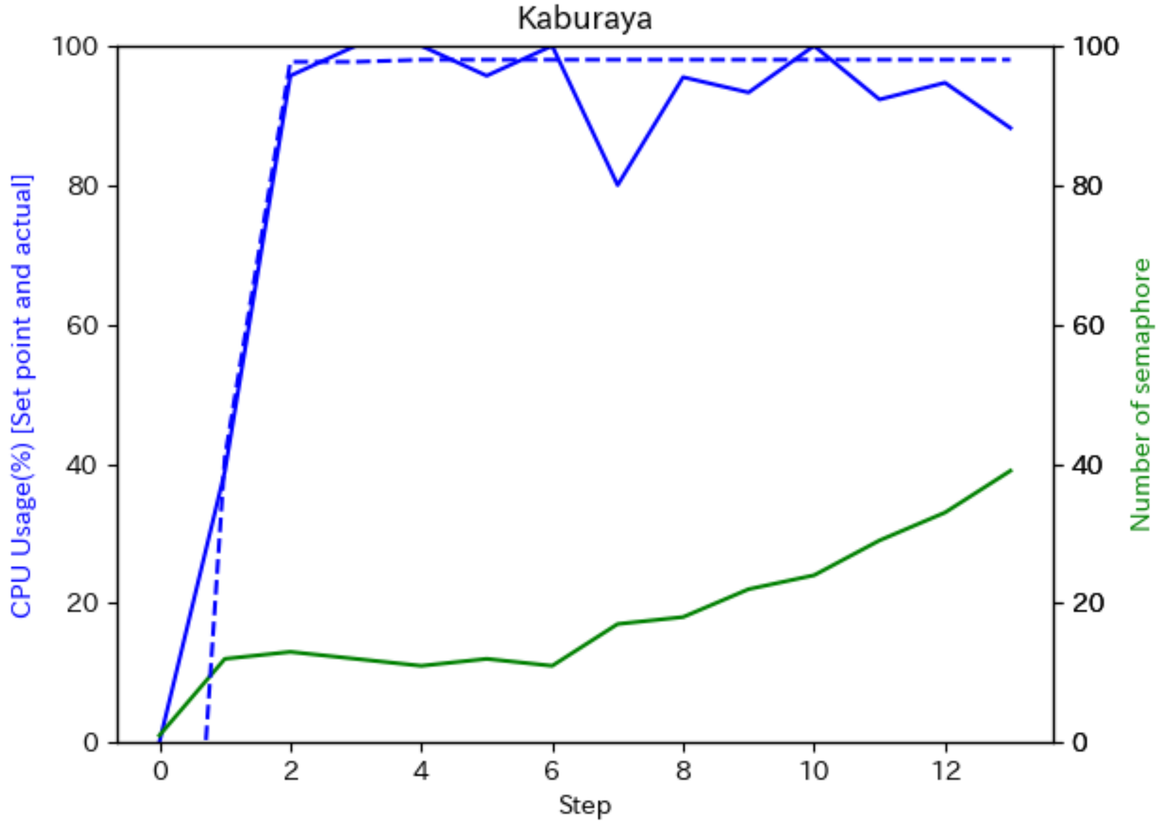


- pt runs on Linux
- CPU 4, Memory 4GB
- GOMAXPROCS=4
- Better concurrent are from 4 to 32.
- Less degradation of performance due to increased goroutines.

Task: pt_linux (Not so bad)

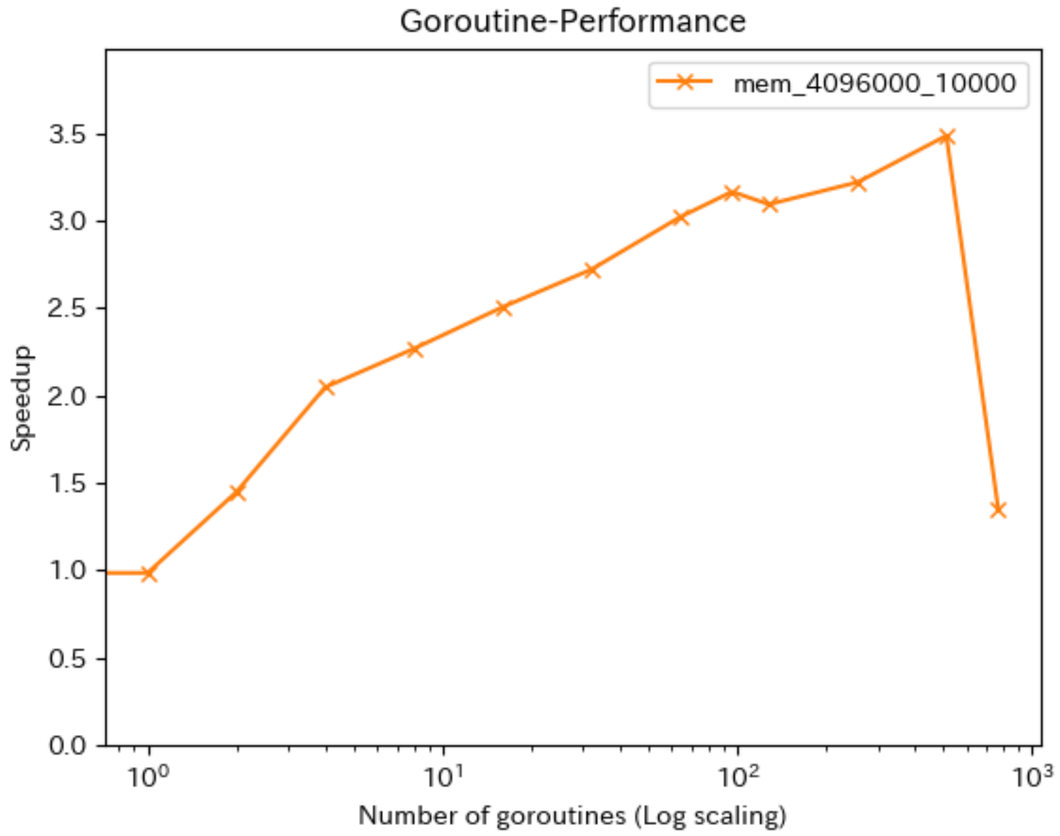


Duration: 50ms, Gain: 0.1, Change rate: ±0.3



- Kaburaya found good set points and adjust numbers within ideal range.
- Increase the number due to failure to reset the set point at end of period.

Task: mem_4096000_10000



Runs task uses memory.

- 4MB/task x 10,000

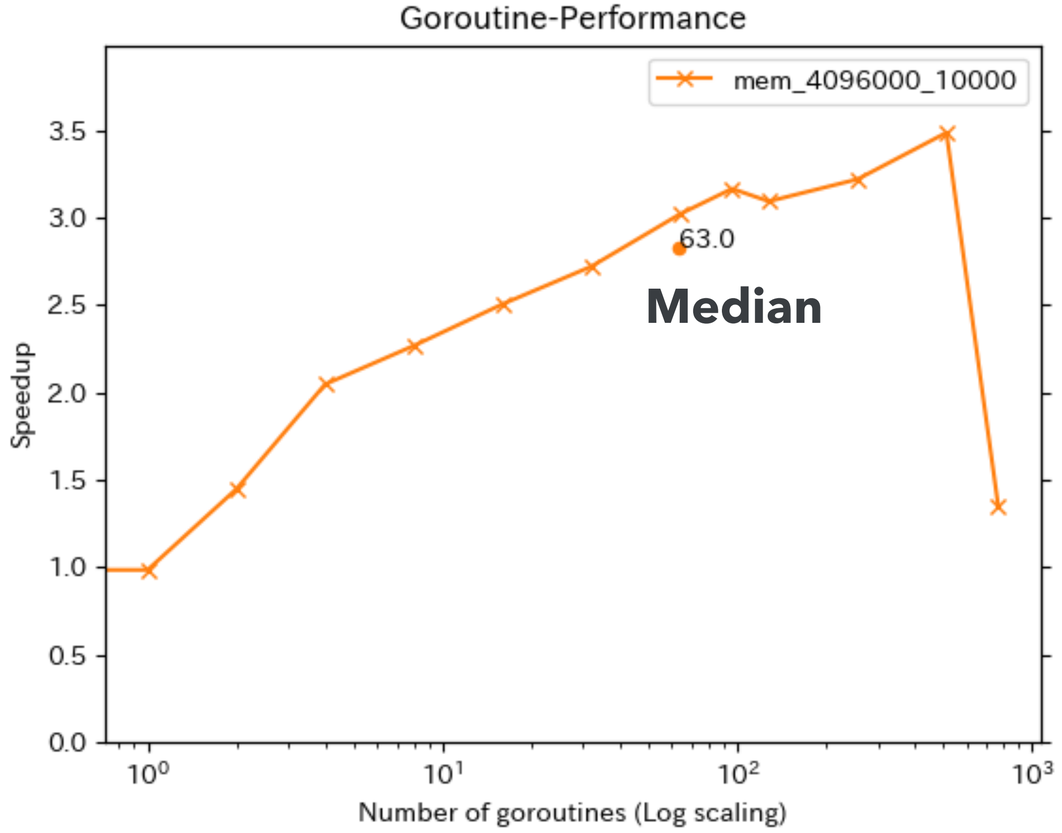
CPU 4, Memory 4GB

GOMAXPROCS=4

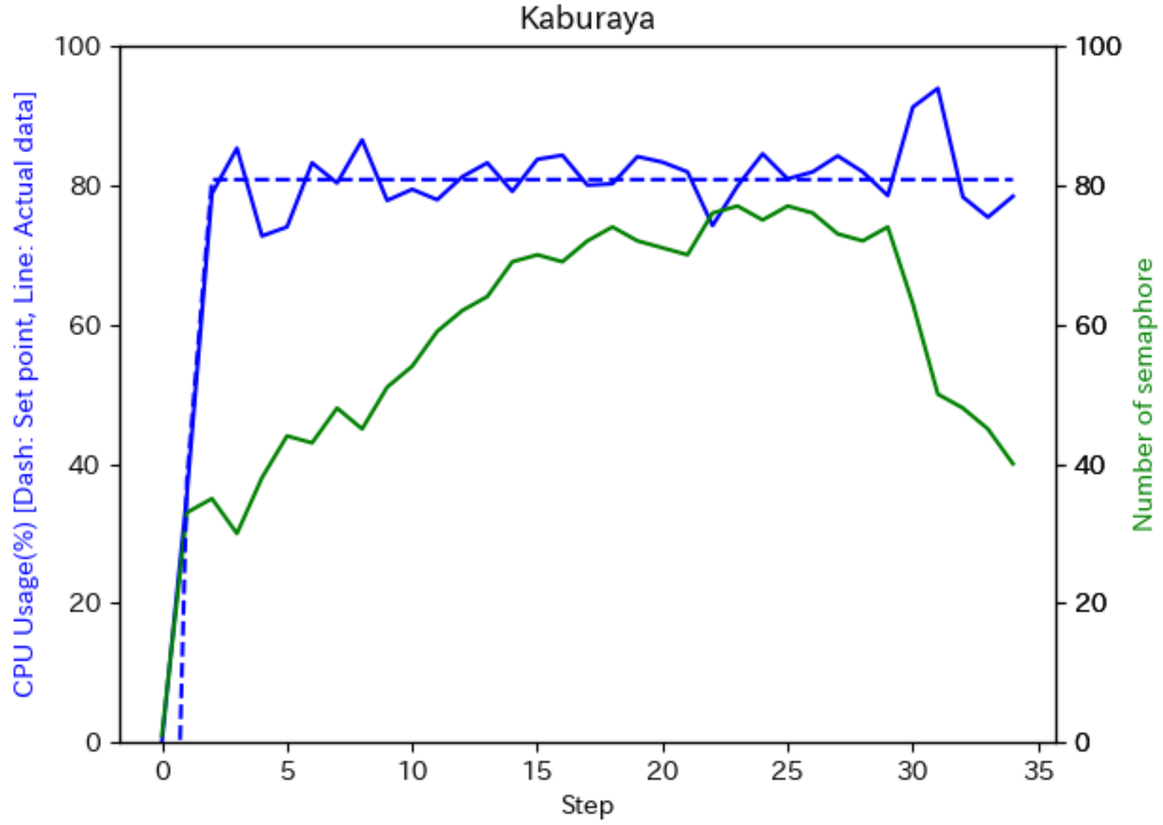
Better concurrent is 512.

More degradation of performance (Swap out) due to increased goroutines.

Task: mem_4096000_10000 (Good)

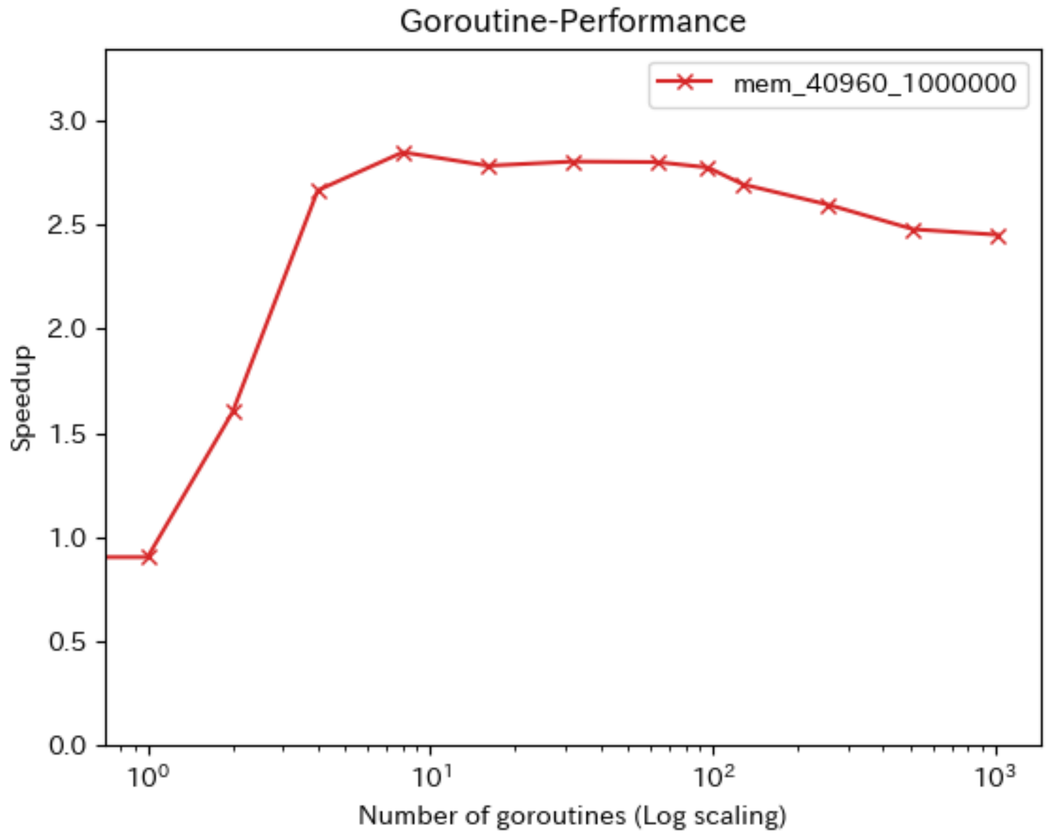


Duration: 500ms, Gain: 0.3, Change rate: ±0.3



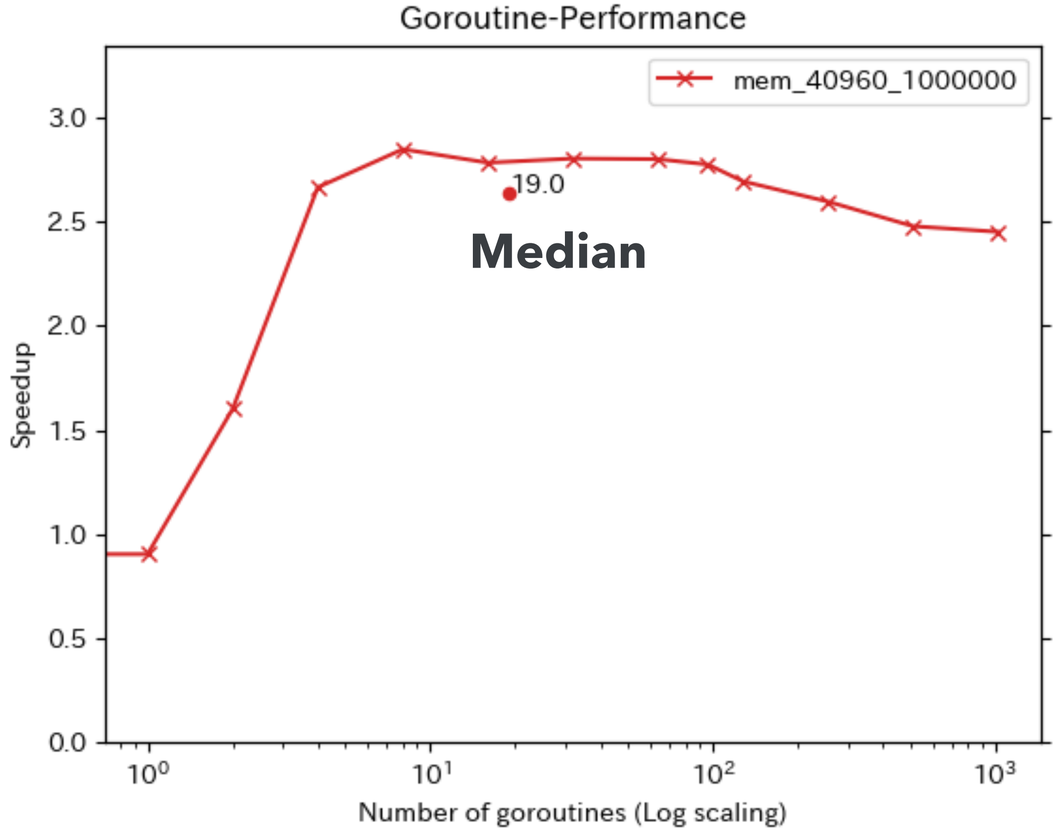
- Kaburaya found set points and adjust numbers within ideal range.
- Avoid starvation of resource

Task: mem_40960_1000000

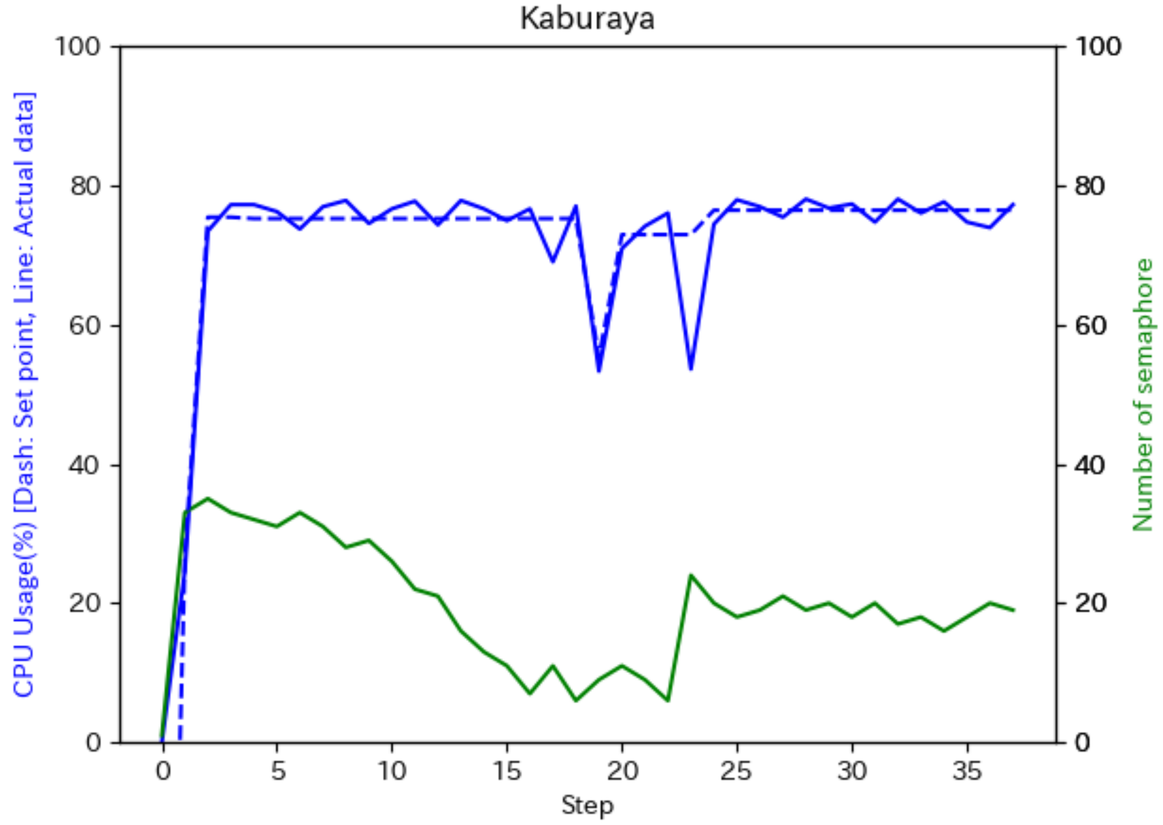


- Runs task uses memory.
  - 40kB/task x 1,000,000
- CPU 4, Memory 4GB
- GOMAXPROCS=4
- Better concurrent are from 4 to 64.
- Less degradation of performance due to increased goroutines.

Task: mem_40960_1000000 (Good)

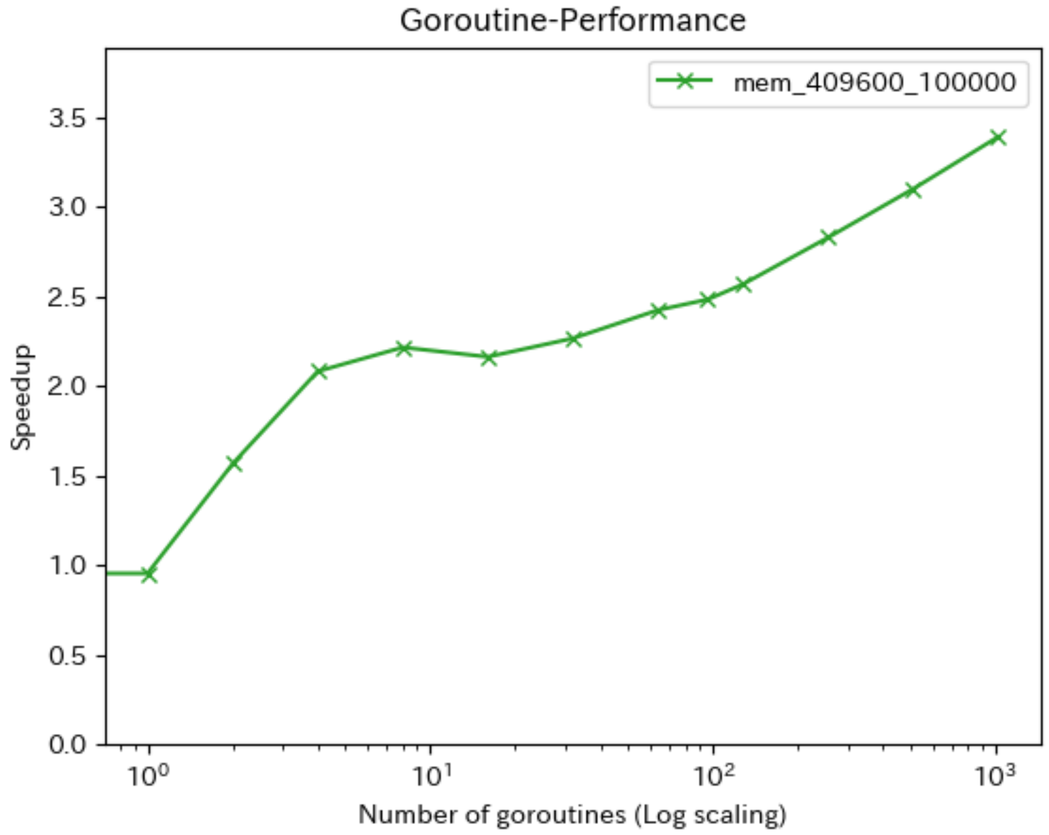


Duration: 500ms, Gain: 0.3, Change rate: ±0.3



- Kaburaya found set points and adjust numbers within ideal range.

Task: mem_409600_100000



Runs task uses memory.

- 400kB/task x 100,000

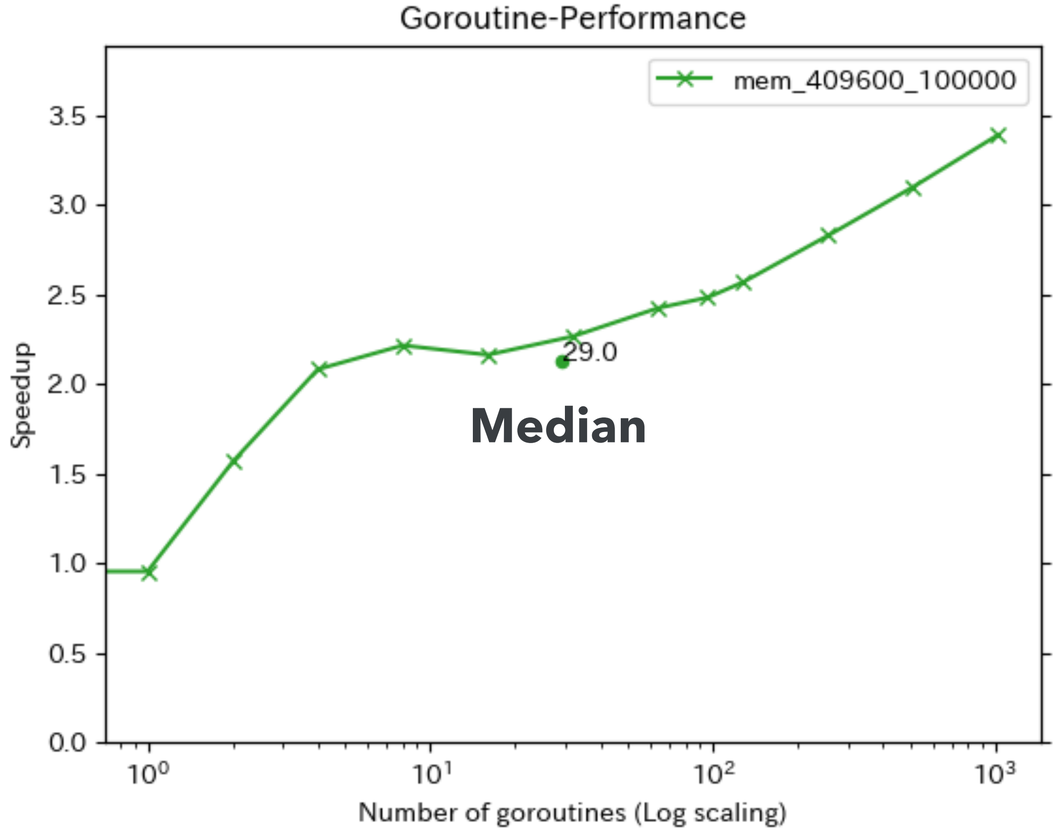
CPU 4, Memory 4GB

GOMAXPROCS=4

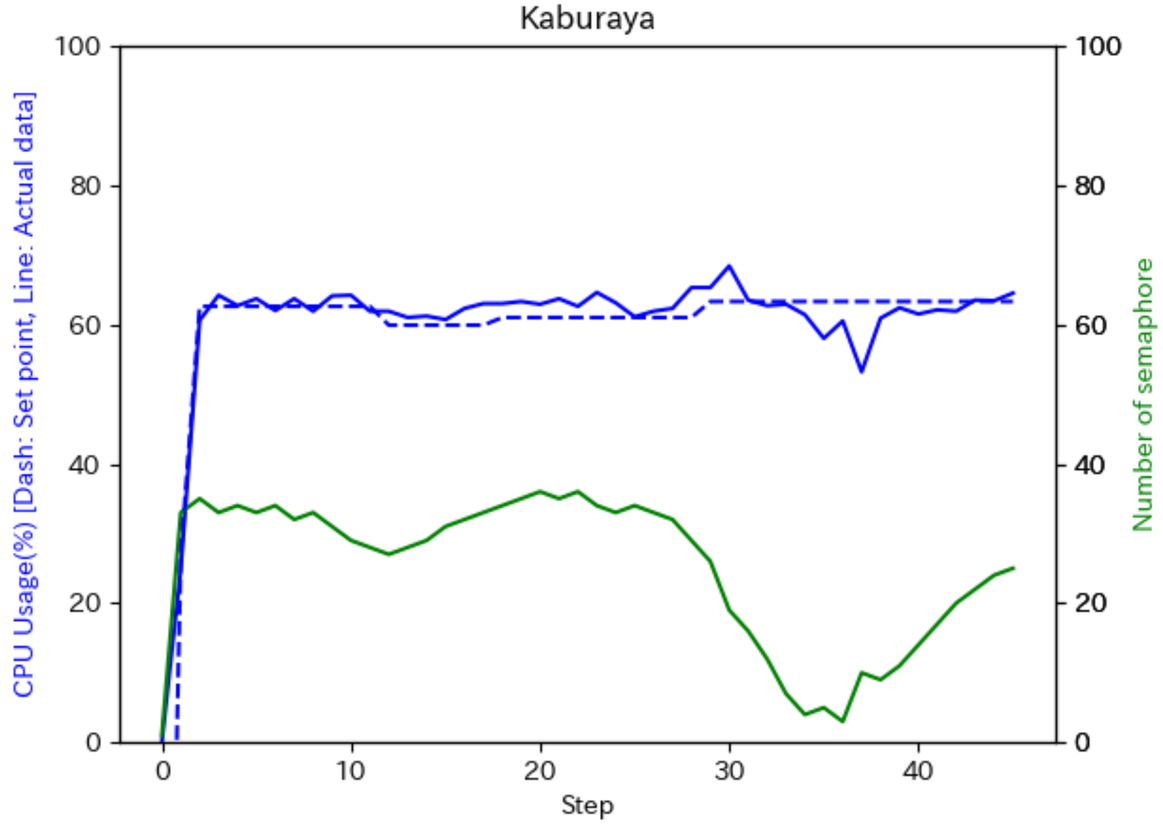
Better concurrent is over 1024?

Less degradation of performance due to increased goroutines.

Task: mem_409600_100000 (Not so good)



Duration: 500ms, Gain: 0.3, Change rate: ±0.3



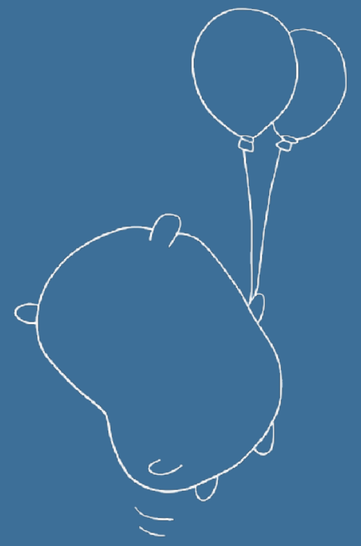
- Kaburaya found set points and adjust numbers.
- Does not reach the limit of performance metrics…

Experimental results

- Kaburaya finds set point and adjusts numbers within ideal range by feedback control and elastic semaphore.
- Towards to improve the controller
  - prediction accuracy of set point
  - Optimal parameters (K{p,i,d} and range of change rate)
- Solve them in the related researches about feedback control as the next step.

5.

Conclusion



Conclusion

- **I proposed kaburaya to control the number of goroutines without depending on platform, runtime and current load.**
- **Experimental results show possible and some issues of kaburaya.**
  - In particular improvement detecting performance upper limit of task is important.
- **In the future, I will also consider the application to auto-scaling of cloud computing.**
- ***ΞGo***

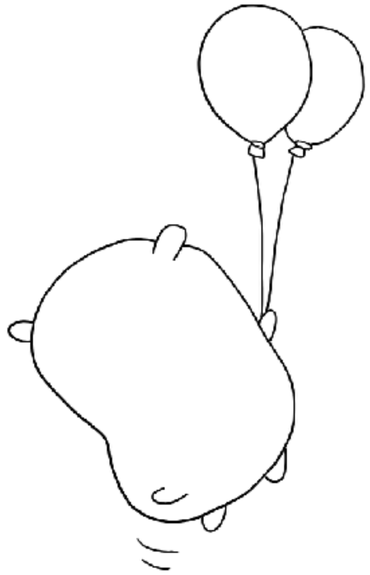
Appendix

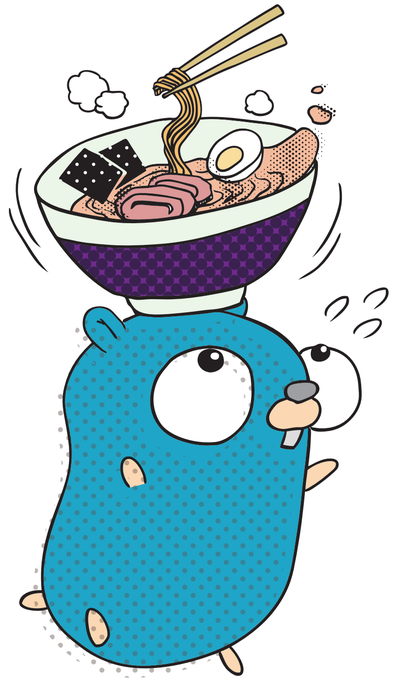
Reference

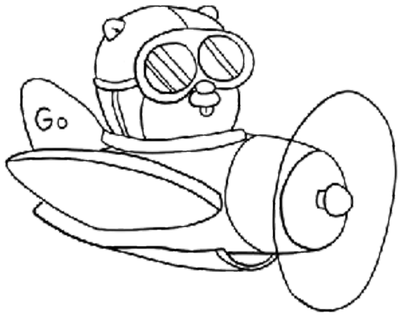
- **Implementation of Kaburaya**
  - https://github.com/monochromegane/kaburaya
- **Study of Kaburaya (In Japanese)**
  - https://blog.monochromegane.com/blog/2018/11/25/wsa_3_kaburaya/
  - https://gist.github.com/monochromegane/5e9fc94371a9b2385eedd38d98341e6b
  - https://blog.monochromegane.com/blog/2019/04/14/wsa_4_ebira/
- **Learn more goroutine**
  - https://speakerdeck.com/monochromegane/road-to-your-goroutines

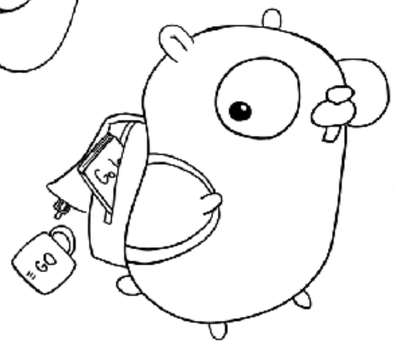
Go gopher

- The Go gopher was designed by Renee French.
- The gopher illustrations were drawn by Keita Kawamoto.
- Licensed under the Creative Commons 3.0 Attributions license.

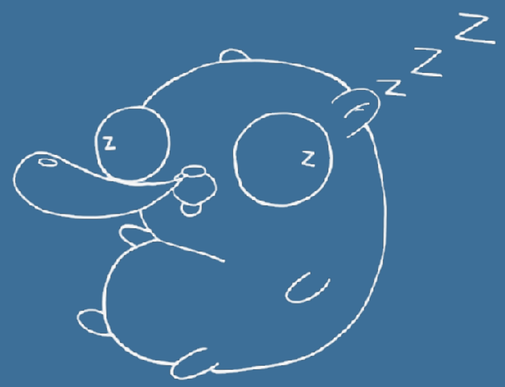








Thank you!



ペパボ研究所

Pepabo R&D Institute, GMO Pepabo, Inc.