Concurrency Let's GO

An Incomplete Guide to Concurrency and How to Concurrency in the Go Programming Language

Tommy TIAN

tommy.tian@credit-suisse.com

Personal email: txaty@proton.me

GitHub: txaty

A Tale of Two Laws

Moore’s Law

- The number of transistors in an integrated circuit (IC) doubles about every two years.
- For most of its life, Moore’s Law and single processor performance correlated well.
- An empirical study from industrial experience. NOT a physical law.

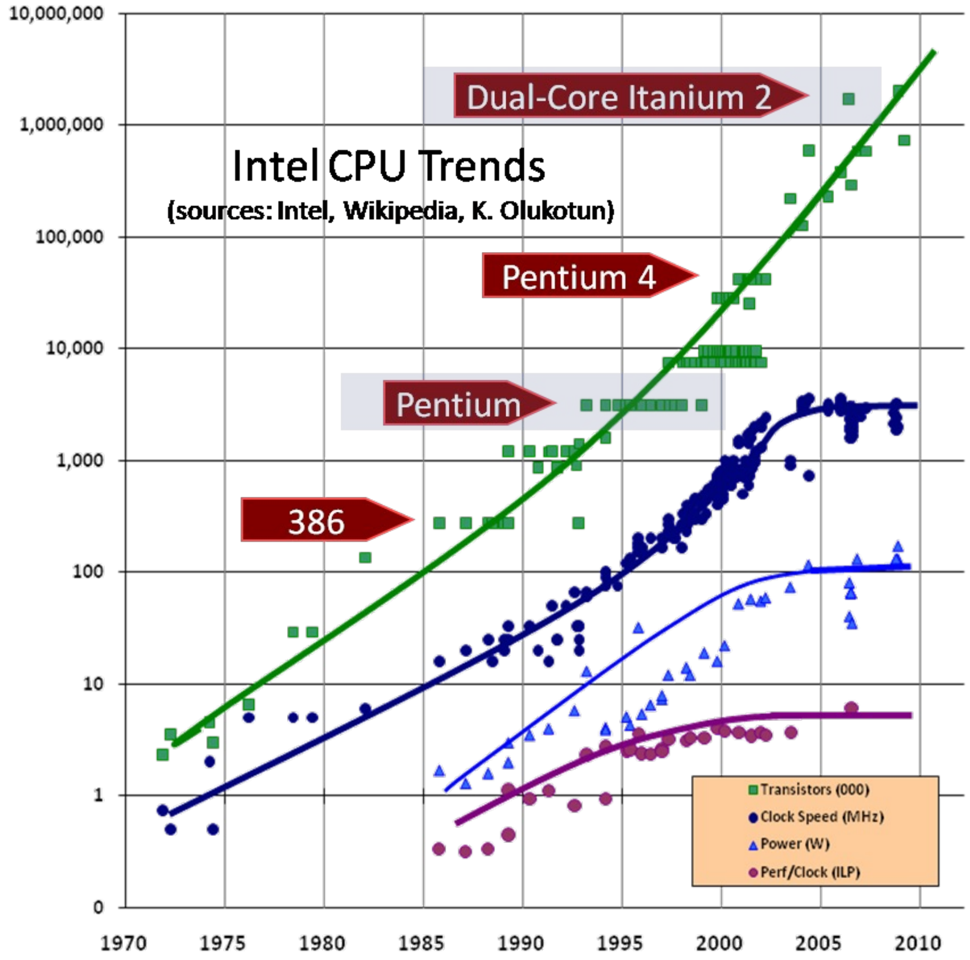


Chart credit: The free Lunch is Over: A Fundamental Turn Toward Concurrency in Software

In parallelization:

- P: proportion of a system or program that can be made parallel;
- $1\text{-}P$: the proportion that remains serial;
- The maximum speedup $S(N)$ achieved using N processors:

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

\- Amdahl's Law

Horizontal Scaling

Evolution: sequential computing ⇒ single-core concurrency

⇒ multi-threading and multi-core processors

⇒ distributed and cloud computing

⇒ GPU Computing and Heterogeneous Computing

Context: processor, operating system, application, machines, etc.

Embarrassingly Parallel Problems: Monte Carlo analysis, distributed relational DB query using distributed set processing, brute force searching in cryptography, serving static files on a webserver to multiple user at once, etc.

Concurrency and Parallelism?!

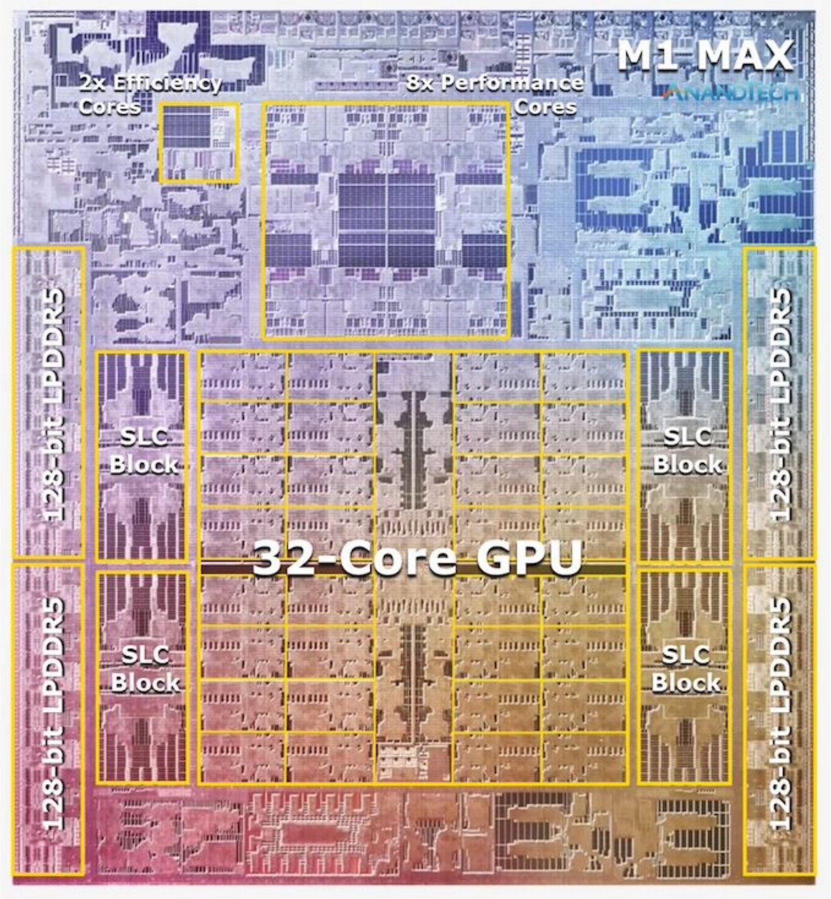


Image credit: Apple Announces M1 Pro & M1 Max: Giant New Arm SoCs with All-Out Performance from AnnadTech

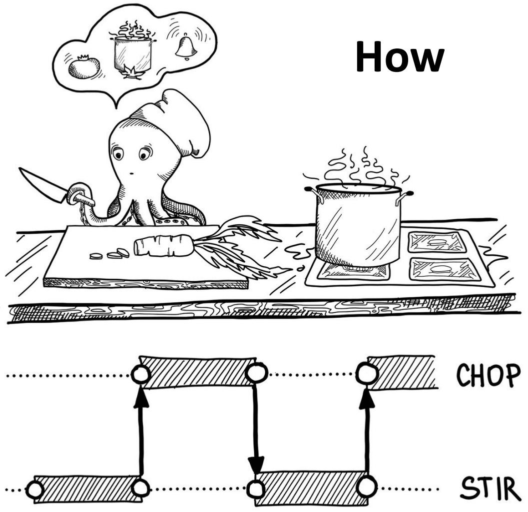
Concurrency is about dealing with lots of things at once.

Parallelism is about doing lots of things at once.

- Rob Pike

Concurrency vs. Parallelism

- Not the same, but related.
- Concurrency is about structure; parallelism is about execution.
- Concurrency provides a way to structure a solution to solve a problem that may (but not necessarily) by parallelizable.
- Concurrency is a property of the code; parallelism is a property of the running program.



How

Concurrency vs. **Parallelism**

What

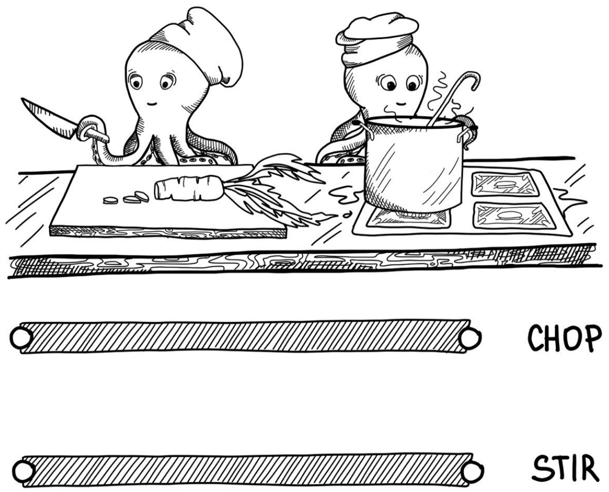


Image credit: Concurrency vs Parallelism by Manning Publication

A computer is like air conditioning – it becomes useless when you open Windows.

- Linus Torvalds

Process, Thread, and Coroutine (Linux)

Process

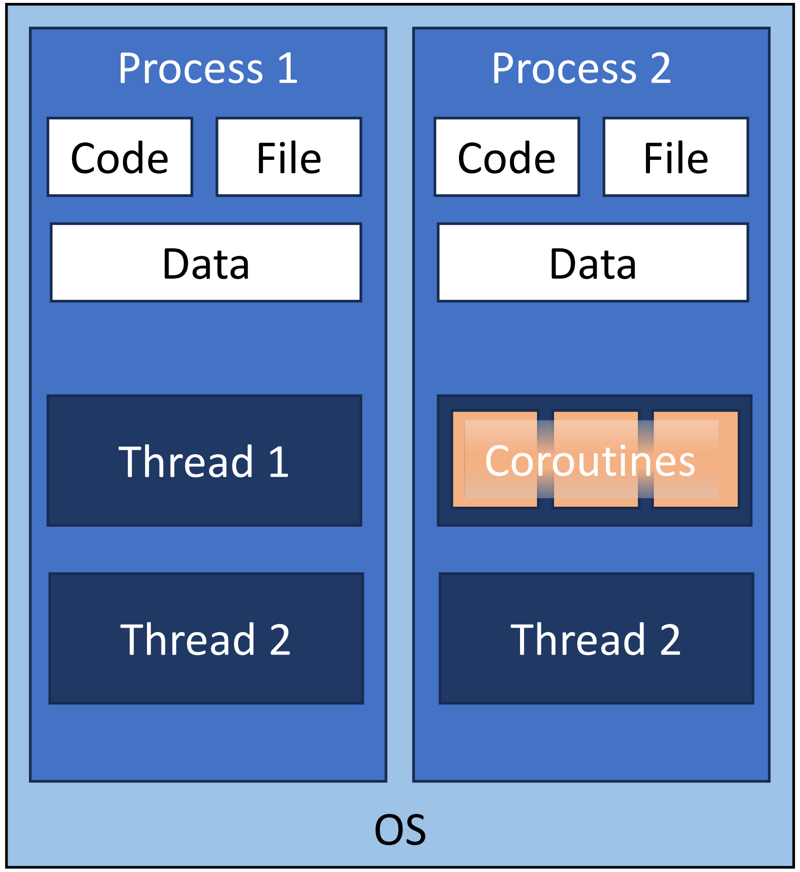
- Unit of resource allocation and scheduling, containing single or multiple threads.
- Context switching handled by OS, with lower efficiency.

Threads

- Basic unit of execution within a process, can contain multiple coroutines (fibers).
- Context switching handled by OS, with moderate efficiency.
- User threads and kernel threads.

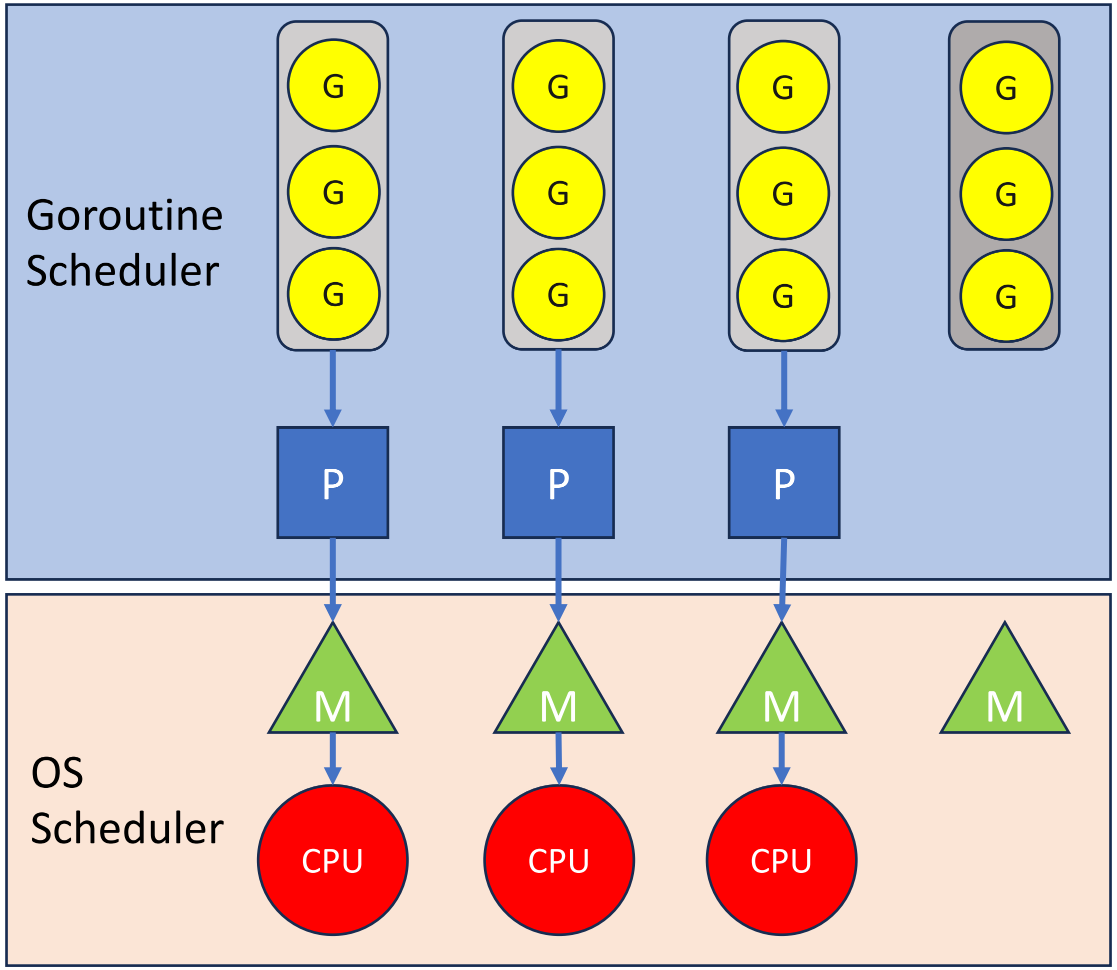
Coroutines

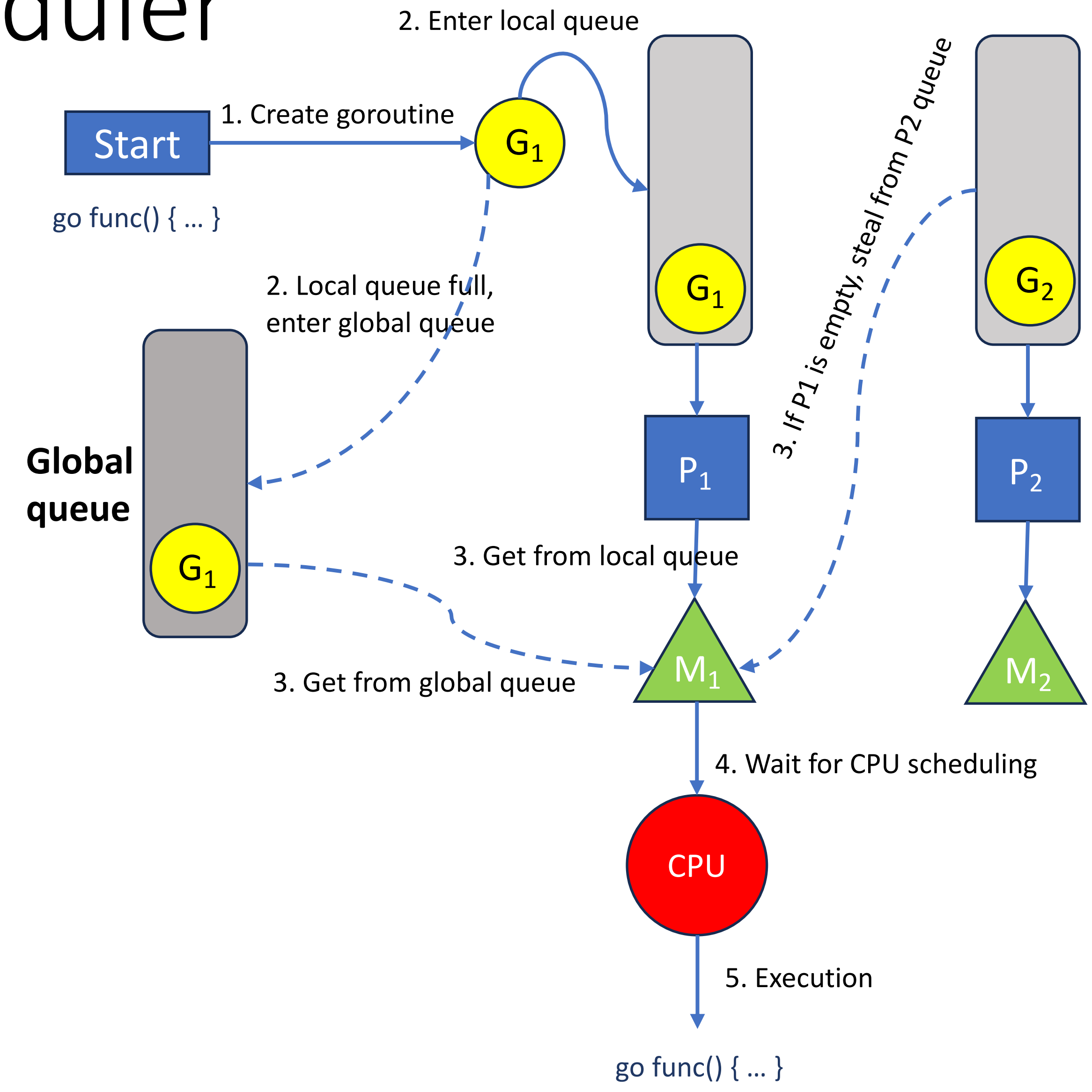
- Non-preemptive functions within a thread, executed sequentially.
- Context switching handled by the user-defined program (in user level only), with high efficiency.



GMP Model for Go Scheduler

G: goroutine, **M**: machine, kernel-level threads, **P**: processor

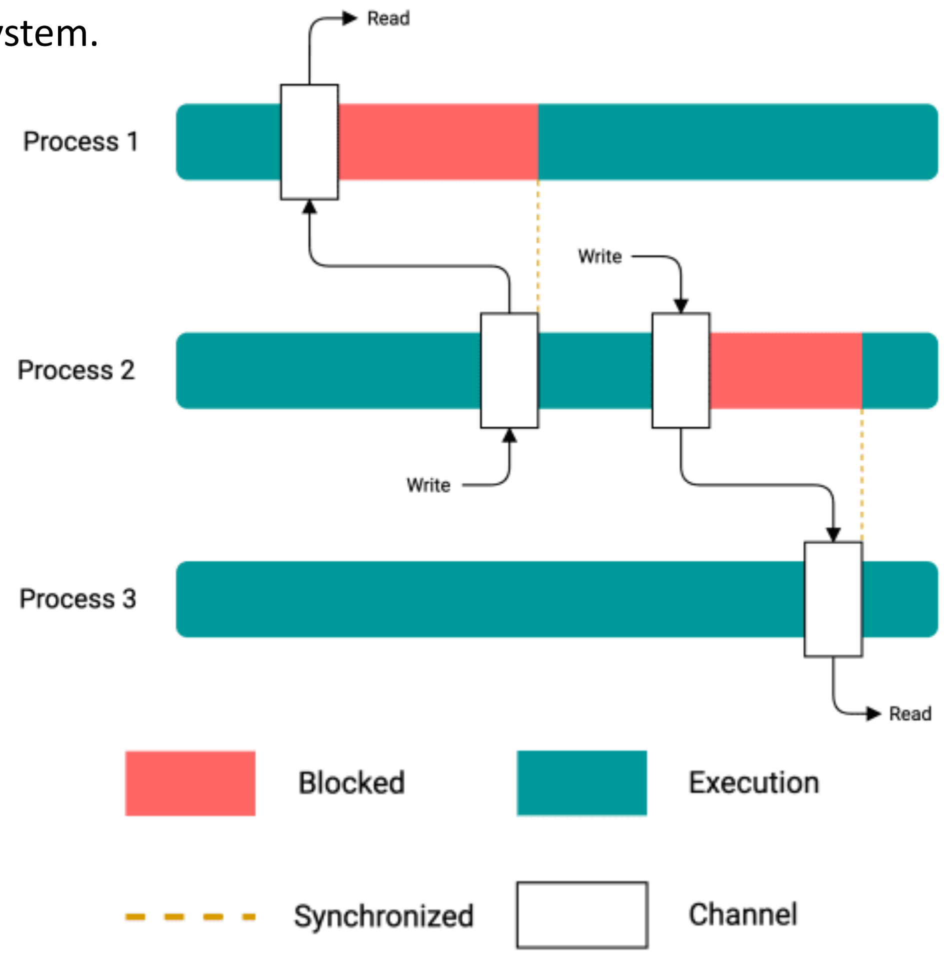




Communicating Sequential Process (CSP)

A formal language for describing patterns of interaction in concurrent system.

- **Processes**: They serve as the basic computational entities, capable of executing both independently and concurrently with other processes.
- **Communication**: Processes engage in dialogue through the exchange of messages via channels, sidestepping the need for shared memory.
- **Synchronization**: Built-in synchronization is triggered during process communication. A process sending a message will pause until the recipient process acknowledges receipt, and vice versa.
- **Channels**: Channels act as conduits facilitating communication between processes, akin to interconnecting pipes.



Concurrency in GO

Goroutines: Lightweight threads managed by the Go runtime

- Inexpensive to create and destroy
- Efficiently managed by the Go scheduler
- Easy to use with simple go keyword

Context Package: Carrying deadlines, cancellation signals, and other request-scoped values across API boundaries and between processes

- *context.Background* and *context.TODO:* base contexts
- *context.WithCancel:* cancelable contexts
- *context.WithDeadline* and *context.WithTimeout:* contexts with time limits

Channels: Main method of communication between concurrently executing goroutines

- Synchronous and asynchronous modes
- Safe, built-in way to pass data between goroutines

Sync Package: Additional primitives for concurrency control

- *sync.WaitGroup:* waiting for a collection of goroutines to finish
- *sync.Mutex* and *sync.RWMutex:* protecting shared resources
- *sync.Once:* one-time initialization
- *sync.Cond:* signaling between goroutines
- *sync.Map:* a safe, concurrent map

Share memory by communicating,

don't communicate by sharing memory.

- A motto in Go

CSP vs. Shared Memory

```go
package main

import (
    "fmt"
    "time"
)

func producer(data chan<- int) {
    for i := 0; i < 10; i++ {
        fmt.Println("Produced:", i)
        data <- i
        time.Sleep(time.Second)
    }
    close(data)
}

func consumer(data <-chan int) {
    for i := range data {
        fmt.Println("Consumed:", i)
    }
}

func main() {
    data := make(chan int)
    go producer(data)
    consumer(data)
}
```

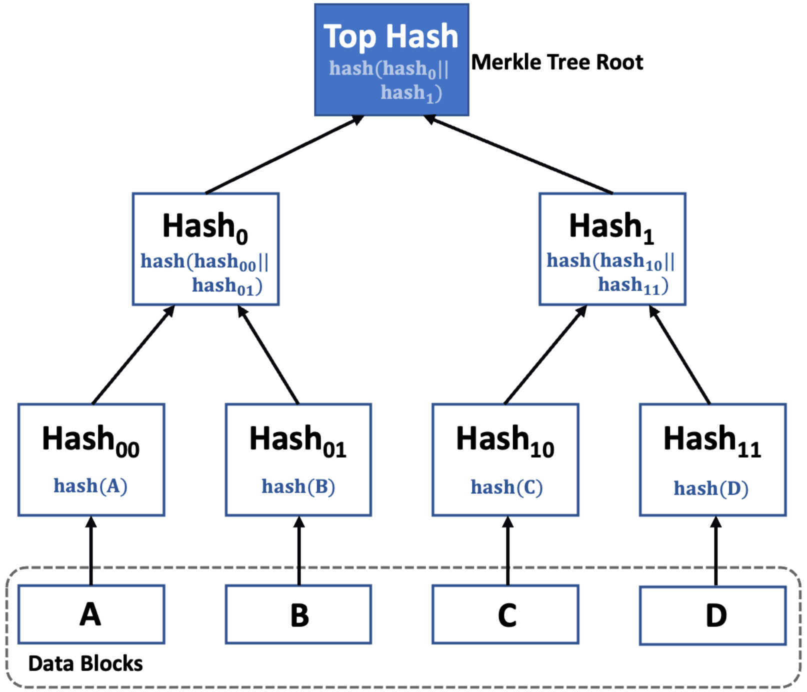
CSP Model
Using channels only

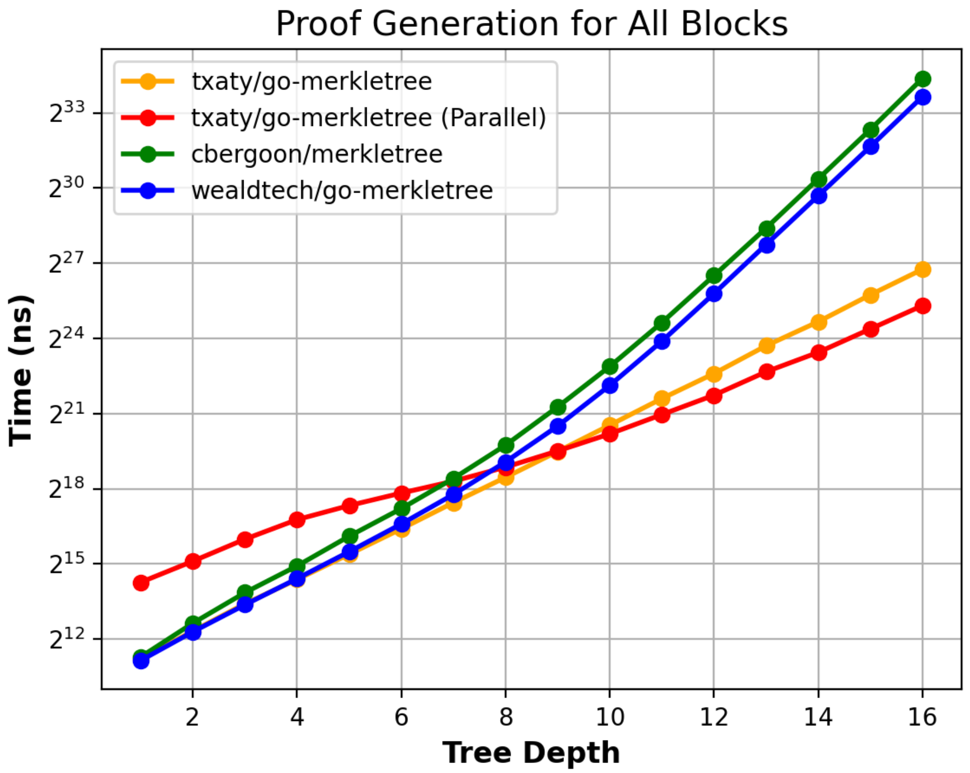
Shared Memory Model
Using mutex, wait group

```go
package main

import (
    "fmt"
    "sync"
)

var data []int
var cond = sync.NewCond(&sync.Mutex{})

func producer() {
    for i := 0; i < 10; i++ {
        cond.L.Lock()
        data = append(data, i)
        fmt.Println("Produced:", i)
        cond.Signal()
        cond.L.Unlock()
    }
}

func consumer() {
    cond.L.Lock()
    for len(data) == 0 {
        cond.Wait()
    }
    d := data[0]
    data = data[1:]
    fmt.Println("Consumed:", d)
    cond.L.Unlock()
}

func main() {
    go producer()
    for i := 0; i < 10; i++ {
        consumer()
    }
}
```

Example

Merkle Tree Parallel Computation (txaty/go-merkletree) and Goroutine Pool (txaty/gool)





Q & A