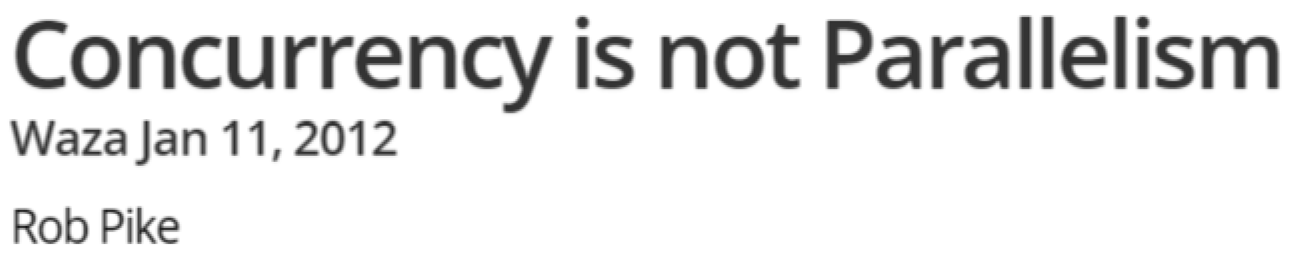


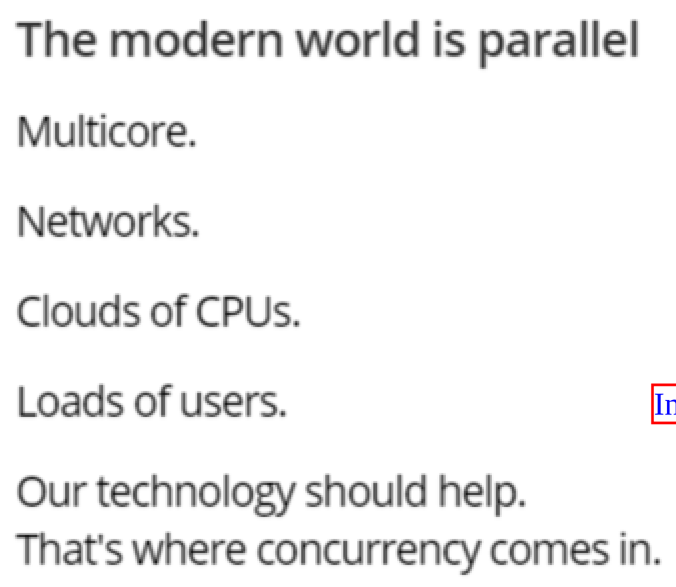
Concurrency is not Parallelism

Waza Jan 11, 2012

Rob Pike

https://talks.golang.org/2012/waza.slide#3

Rob Pike
r@golang.org



The modern world is parallel

Multicore.

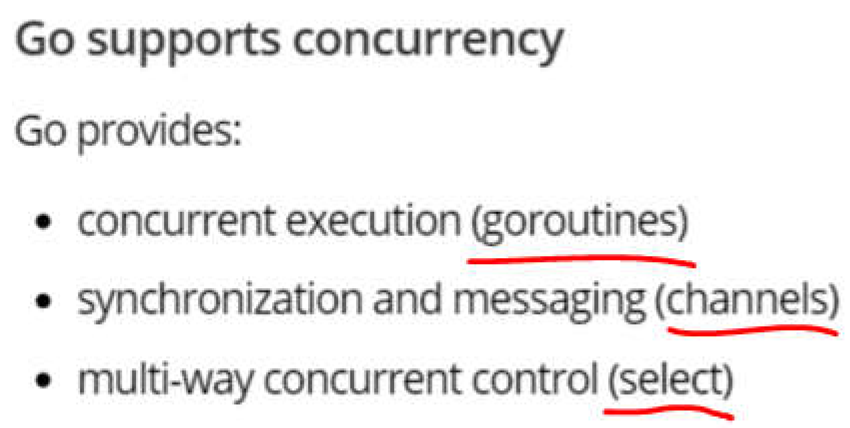
Networks.

Clouds of CPUs.

Loads of users.

Internet of Things

Our technology should help.
That's where concurrency comes in.



Go supports concurrency

Go provides:

- concurrent execution (goroutines)
- synchronization and messaging (channels)
- multi-way concurrent control (select)

Concurrency vs. parallelism

Concurrency is about dealing with lots of things at once.

Parallelism is about doing lots of things at once.

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) be parallelizable.

An analogy

Concurrent: Mouse, keyboard, display, and disk drivers.

Parallel: Vector dot product.

Concurrency plus communication

Concurrency is a way to structure a program by breaking it into pieces that can be executed independently.

Communication is the means to coordinate the independent executions.

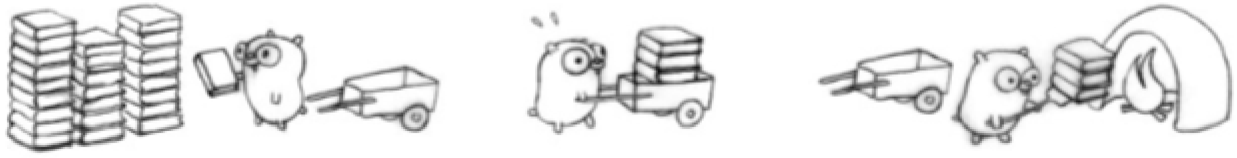
This is the Go model and (like Erlang and others) it's based on CSP:

C. A. R. Hoare: Communicating Sequential Processes (CACM 1978)

Concurrent composition

The concurrent composition of two gopher procedures.

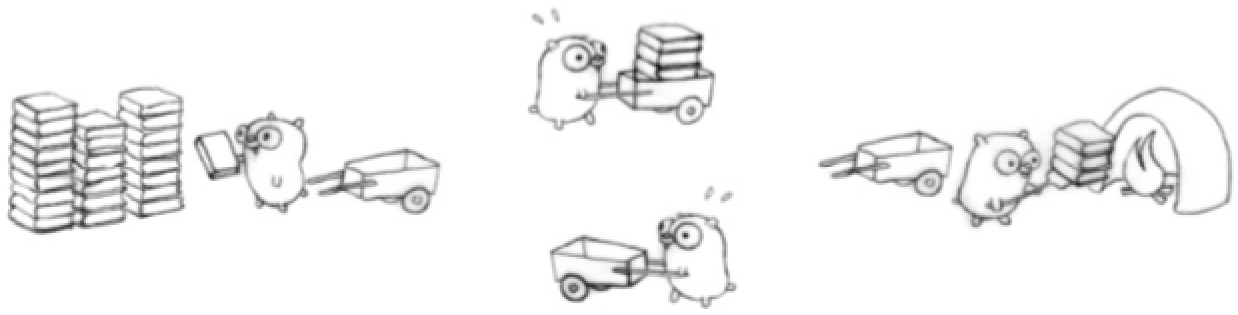
Another design



Three gophers in action, but with likely delays.
Each gopher is an independently executing procedure,
plus coordination (communication).

Finer-grained concurrency

Add another gopher procedure to return the empty carts.



Four gophers in action for better flow, each doing one simple task.

If we arrange everything right (implausible but not impossible), that's four times faster than our original one-gopher design.

Observation

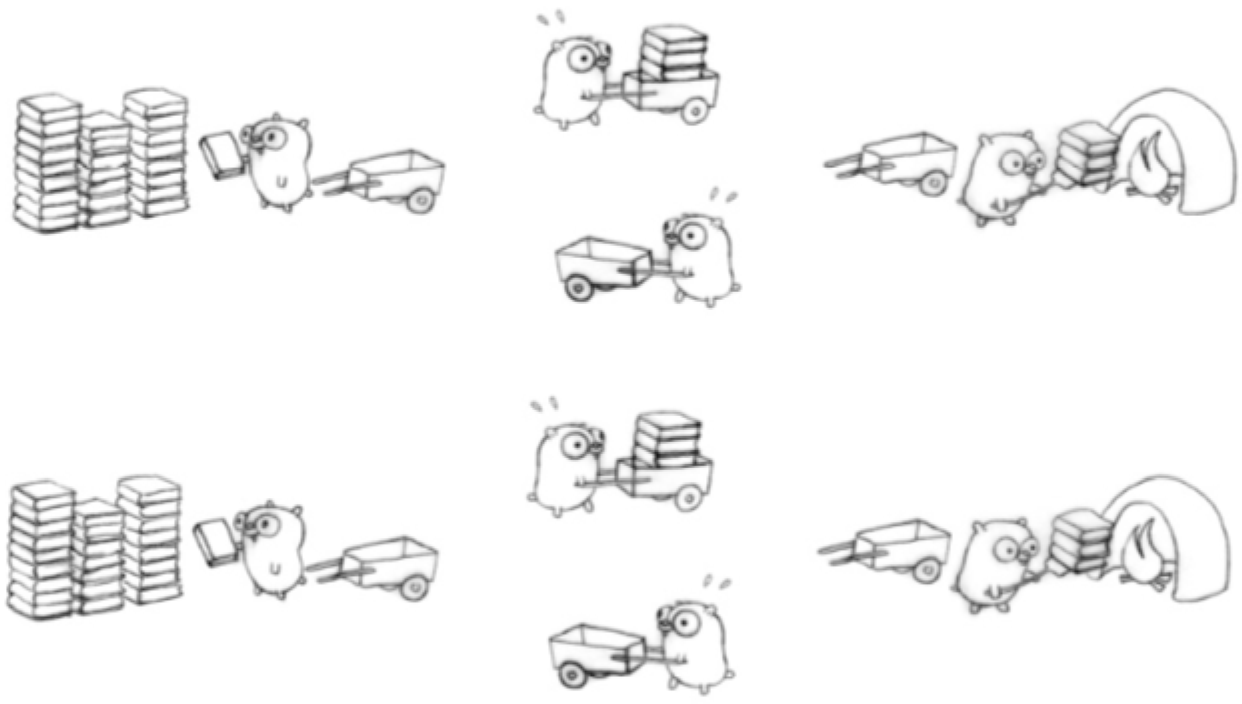
We improved performance by adding a concurrent procedure to the existing design.

More gophers doing more work; it runs better.

This is a deeper insight than mere parallelism.

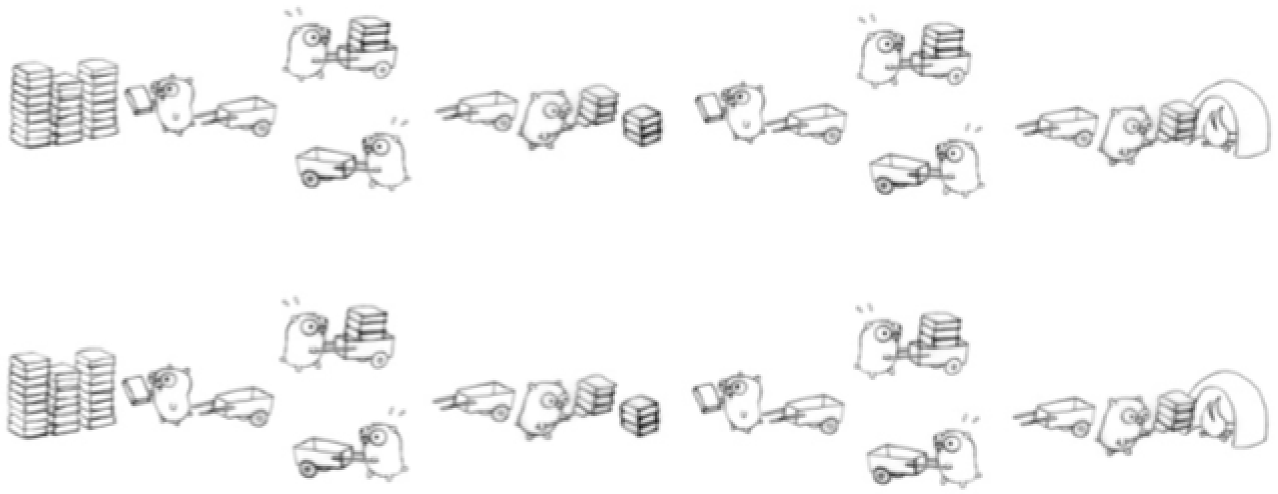
More parallelization!

We can now parallelize on the other axis; the concurrent design makes it easy. Eight gophers, all busy.



Full on optimization

Use all our techniques. Sixteen gophers hard at work!



Lesson

There are many ways to break the processing down.

That's concurrent design.

Once we have the breakdown, parallelization can fall out and correctness is easy.

Back to Computing

In our book transport problem, substitute:

- book pile => web content
- gopher => CPU
- cart => marshaling, rendering, or networking
- incinerator => proxy, browser, or other consumer

It becomes a concurrent design for a scalable web service.
Gophers serving web content.

Goroutines

A goroutine is a function running independently in the same address space as other goroutines

```
f("hello", "world") // f runs; we wait
```

```
go f("hello", "world") // f starts running
g() // does not wait for f to return
```

Like launching a function with shell's & notation.

Goroutines are not threads

(They're a bit like threads, but they're much cheaper.)

Goroutines are multiplexed onto OS threads as required.

When a goroutine blocks, that thread blocks but no other goroutine blocks.

Channels

Channels are typed values that allow goroutines to synchronize and exchange information.

```
timerChan := make(chan time.Time)
go func() {
    time.Sleep(deltaT)
    timerChan <- time.Now() // send time on timerChan
}()
// Do something else; when ready, receive.
// Receive will block until timerChan delivers.
// Value sent is other goroutine's completion time.
completedAt := <-timerChan
```

Select

The `select` statement is like a `switch`, but the decision is based on ability to communicate rather than equal values.

```
select {
case v := <-ch1:
    fmt.Println("channel 1 sends", v)
case v := <-ch2:
    fmt.Println("channel 2 sends", v)
default: // optional
    fmt.Println("neither channel was ready")
}
```

Go really supports concurrency

Really.

It's routine to create thousands of goroutines in one program.
(Once debugged a program after it had created 1.3 million.)

Stacks start small, but grow and shrink as required.

Goroutines aren't free, but they're very cheap.

Closures are also part of the story

Make some concurrent calculations easier to express.

They are just local functions.
Here's a non-concurrent example:

```
func Compose(f, g func(x float) float)
                  func(x float) float {
     return func(x float) float {
        return f(g(x))
    }
}

print(Compose(sin, cos)(0.5))
```

Concurrency enables parallelism

The load balancer is implicitly parallel and scalable.

`NumWorkers` could be huge.

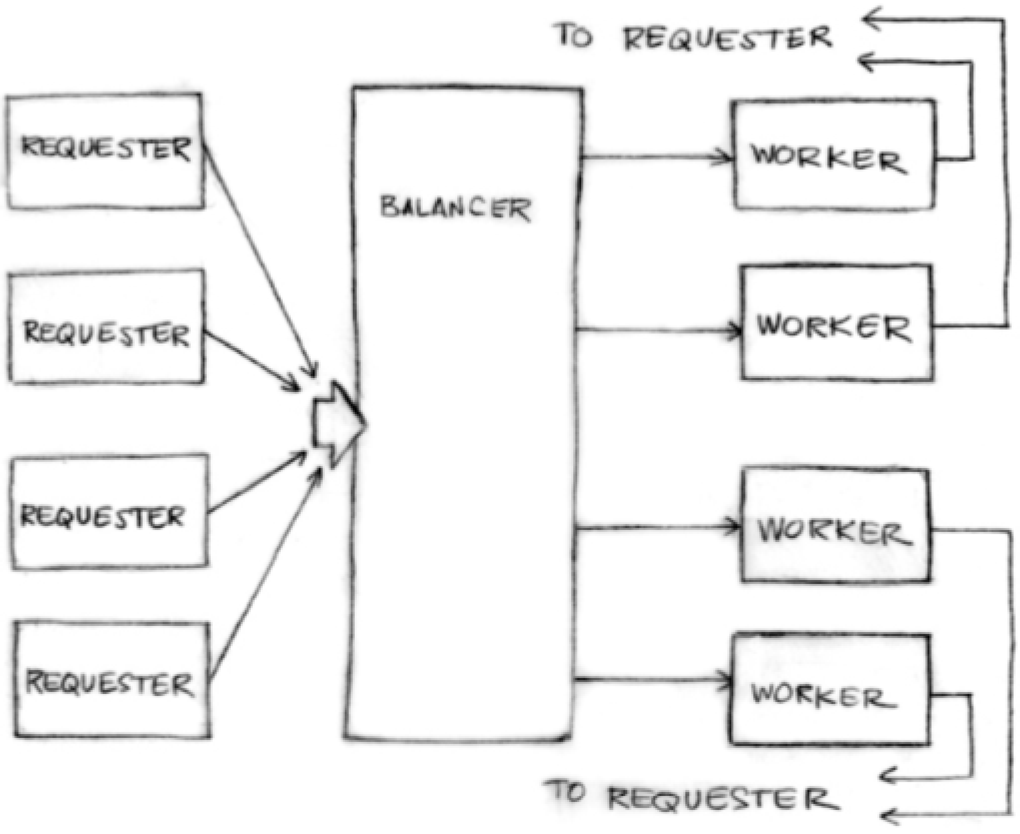
The tools of concurrency make it almost trivial to build a safe, working, scalable, parallel design.

Concurrency simplifies synchronization

No explicit synchronization needed.

The structure of the program is implicitly synchronized.

Load balancer



Request definition

The requester sends Requests to the balancer

```
type Request struct {
    fn func() int  // The operation to perform.
    c  chan int    // The channel to return the result.
}
```

Note the return channel inside the request.
Channels are first-class values.

Requester function

An artificial but illustrative simulation of a requester, a load generator.

```
func requester(work chan<- Request) {
    c := make(chan int)
    for {
        // Kill some time (fake load).
        Sleep(rand.Int63n(nWorker * 2 * Second))
        work <- Request{workFn, c} // send request
        result := <-c              // wait for answer
        furtherProcess(result)
    }
}
```

Worker definition

A channel of requests, plus some load tracking data.

```
type Worker struct {
    requests chan Request // work to do (buffered channel)
    pending  int          // count of pending tasks
    index     int         // index in the heap
}
```

Worker

Balancer sends request to most lightly loaded worker

```
func (w *Worker) work(done chan *Worker) {
    for {
        req := <-w.requests // get Request from balancer
        req.c <- req.fn()   // call fn and send result
        done <- w           // we've finished this request
    }
}
```

The channel of requests (`w.requests`) delivers requests to each worker. The balancer tracks the number of pending requests as a measure of load.
Each response goes directly to its requester.

Could run the loop body as a goroutine for parallelism.

Balancer definition

The load balancer needs a pool of workers and a single channel to which requesters can report task completion.

```
type Pool []*Worker

type Balancer struct {
    pool Pool
    done chan *Worker
}
```

Balancer function

Easy!

```
func (b *Balancer) balance(work chan Request) {
    for {
        select {
        case req := <-work: // received a Request...
            b.dispatch(req) // ...so send it to a Worker
        case w := <-b.done: // a worker has finished ...
            b.completed(w)  // ...so update its info
        }
    }
}
```

Just need to implement dispatch and completed.

A heap of channels

Make Pool an implementation of the `Heap` interface by providing a few methods such as:

```
func (p Pool) Less(i, j int) bool {
    return p[i].pending < p[j].pending
}
```

Now we balance by making the `Pool` a heap tracked by load.

Dispatch

All the pieces are in place.

```
// Send Request to worker
func (b *Balancer) dispatch(req Request) {
    // Grab the least loaded worker...
    w := heap.Pop(&b.pool).(*Worker)
    // ...send it the task.
    w.requests <- req
    // One more in its work queue.
    w.pending++
    // Put it into its place on the heap.
    heap.Push(&b.pool, w)
}
```

Completed

```
// Job is complete; update heap
func (b *Balancer) completed(w *Worker) {
    // One fewer in the queue.
    w.pending--
    // Remove it from heap.
    heap.Remove(&b.pool, w.index)
    // Put it into its place on the heap.
    heap.Push(&b.pool, w)
}
```

Lesson

A complex problem can be broken down into easy-to-understand components.

The pieces can be composed concurrently.

The result is easy to understand, efficient, scalable, and correct.

Maybe even parallel.

Conclusion

Concurrency is powerful.

Concurrency is not parallelism.

Concurrency enables parallelism.

Concurrency makes parallelism (and scaling and everything else) easy.