Concurrency programming with Go

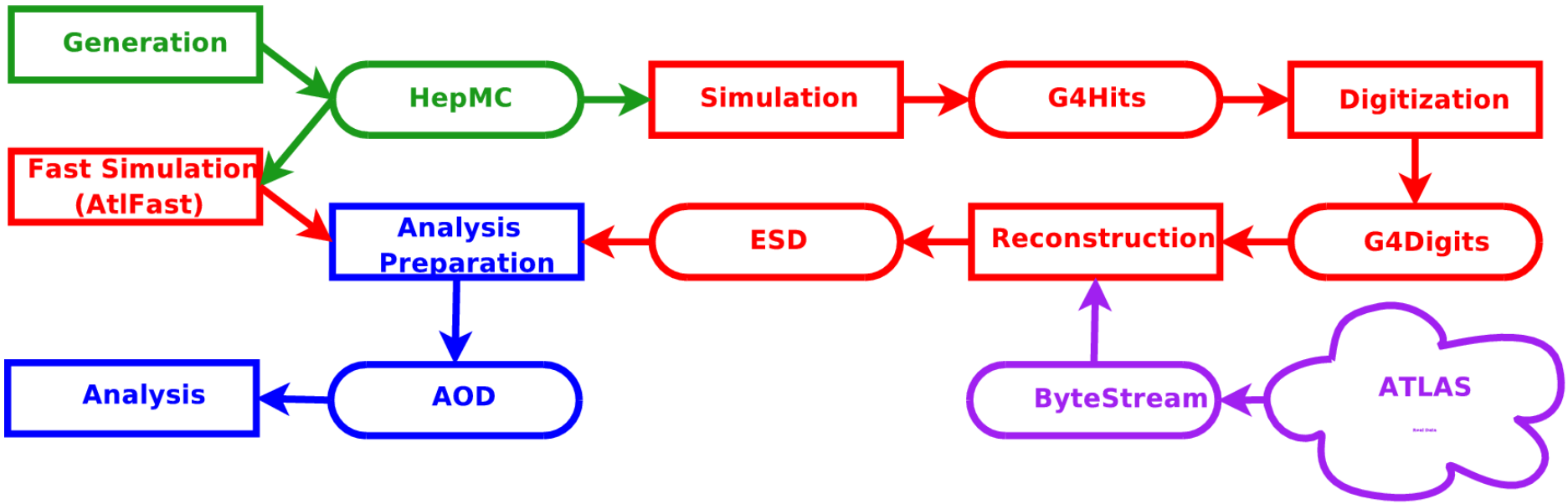
AUDACEs, 2017-06-01

Sebastien Binet
CNRS/IN2P3/LPC-Clermont

Software in High Energy Physics (HEP)

- 50's-90's: FORTRAN77
- 90's-...: C++
- 00's-...: Python

Software in HEP is mostly C++/Python (with pockets of Java and Fortran.)



Software in HEP is painful

Painful to develop:

- deep and complex software stacks
- huge dependencies issues (install, support)
- compilation time
- complex deployment of multi-GB stacks (shared libraries, configuration, DBs, ...)
- C++ is a complex language to learn, read, write and maintain
- unpleasant edit-compile-run development cycle

Software in HEP is painful

Painful to use:

- overly complicated Object Oriented systems
- overly complicated inheritance hierarchies
- overly complicated meta-template programming
- installation of dependencies
- granularity of dependencies
- no simple, nor standard, way to handle dependencies across OSes, experiments, groups, ...
- documentation

End-users tend to prefer `Python` because of its nicer development cycle, despite its runtime performances (or lack thereof.)

Software in HEP: optimization and performances

Software is painful and does not perform well:

- most of our stack is not optimized (OO anti-patterns, code-bloat from C++ templates)
- memory leaks, slow to initialize (loading `.so`/`.dll`), slow to run
- resources hungry to run and develop (CPU, VMem, people)
- most of our stack has to be re-written: support of multi-cores machine, parallelism/concurrency

Parallelism and **concurrency** need to be exposed and leveraged, but the language (C++14, C++17, ...) is **ill equiped** for these tasks.

And C++ is not well adapted for large, distributed development teams (of varying programming skills.)

Time for something new?

Are those our only options ?

Enter... Go

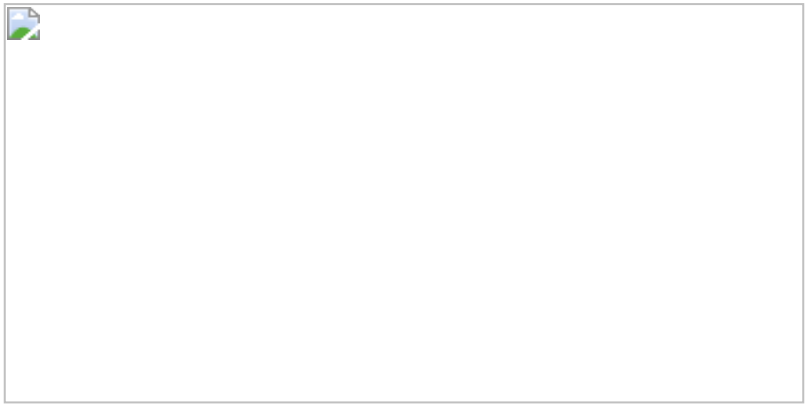
What is Go ?

```
package main

import "fmt"

func main() {
    lang := "Go"
    fmt.Printf("Hello from %s\n", lang)
}
```

Run

```
$ go run hello.go
Hello from Go
```

A nice language with a nice mascot.



History

- Project starts at Google in 2007 (by Griesemer, Pike, Thompson)
- Open source release in November 2009
- More than 700 contributors have joined the project
- Version 1.0 release in March 2012
- Version 1.1 release in May 2013
- Version 1.2 release in December 2013
- *[...]*
- Version 1.6 release in February 2016
- Version 1.7 release in August 2016
- Version 1.8 release in February 2017

golang.org (https://golang.org)

Elements of Go

- Russ Cox, Robert Griesemer, Ian Lance Taylor, Rob Pike, Ken Thompson
- **Concurrent**, **garbage-collected**
- An Open-source general progamming language (BSD-3)
- feel of a **dynamic language**: limited verbosity thanks to the *type inference system*, map, slices
- safety of a **static type system**
- compiled down to machine language (so it is fast, goal is ~10% of C)
- **object-oriented** but w/o classes, **builtin reflection**
- first-class functions with **closures**
- implicitly satisfied **interfaces**

Available on all major platforms (`Linux`, `Windows`, `macOS`, `Android`, `iOS`, ...) and for many architectures (`amd64`, `arm`, `arm64`, `i386`, `s390x`, `mips64`, ...)

Concurrency: basic examples

A boring function

We need an example to show the interesting properties of the concurrency primitives.
To avoid distraction, we make it a boring example.

```
func boring(msg string) {
    for i := 0; ; i++ {
        fmt.Println(msg, i)
        time.Sleep(time.Second)
    }
}
```

Run

Slightly less boring

Make the intervals between messages unpredictable (still under a second).

```
func boring(msg string) {
    for i := 0; ; i++ {
        fmt.Println(msg, i)
        time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)
    }
}
```

Run

Running it

The boring function runs on forever, like a boring party guest.

```
func main() {
    boring("boring!")
}

func boring(msg string) {
    for i := 0; ; i++ {
        fmt.Println(msg, i)
        time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)
    }
}
```

Run

Ignoring it

The go statement runs the function as usual, but doesn't make the caller wait.

It launches a goroutine.

The functionality is analogous to the & on the end of a shell command.

```
package main

import (
    "fmt"
    "math/rand"
    "time"
)

func main() {
    go boring("boring!")
}
```

Run

Ignoring it a little less

When `main` returns, the program exits and takes the boring function down with it.

We can hang around a little, and on the way show that both main and the launched goroutine are running.

```
func main() {
    go boring("boring!")
    fmt.Println("I'm listening.")
    time.Sleep(2 * time.Second)
    fmt.Println("You're boring; I'm leaving.")
}
```

Run

Goroutines

What is a goroutine? It's an independently executing function, launched by a go statement.

It has its own call stack, which grows and shrinks as required.

It's very cheap. It's practical to have thousands, even hundreds of thousands of goroutines.

It's not a thread.

There might be only one thread in a program with thousands of goroutines.

Instead, goroutines are multiplexed dynamically onto threads as needed to keep all the goroutines running.

But if you think of it as a very cheap thread, you won't be far off.

Communication

Our boring examples cheated: the main function couldn't see the output from the other goroutine.

It was just printed to the screen, where we pretended we saw a conversation.

Real conversations require communication.

Channels

A channel in Go provides a connection between two goroutines, allowing them to communicate.

```
// Declaring and initializing.
var c chan int
c = make(chan int)
// or
c := make(chan int)
```

```
// Sending on a channel.
c <- 1
```

```
// Receiving from a channel.
// The "arrow" indicates the direction of data flow.
value = <-c
```

Using channels

A channel connects the main and boring goroutines so they can communicate.

```
func main() {
    c := make(chan string)
    go boring("boring!", c)
    for i := 0; i < 5; i++ {
        fmt.Printf("You say: %q\n", <-c) // Receive expression is just a value.
    }
    fmt.Println("You're boring; I'm leaving.")
}
```

Run

```
func boring(msg string, c chan string) {
    for i := 0; ; i++ {
        c <- fmt.Sprintf("%s %d", msg, i) // Expression to be sent can be any suitable value.
        time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)
    }
}
```

Synchronization

When the main function executes <–c, it will wait for a value to be sent.

Similarly, when the boring function executes c <– value, it waits for a receiver to be ready.

A sender and receiver must both be ready to play their part in the communication. Otherwise we wait until they are.

Thus channels both communicate and synchronize.

The Go approach

Don't communicate by sharing memory, share memory by communicating.

"Patterns"

Generator: function that returns a channel

Channels are first-class values, just like strings or integers.

```
    c := boring("boring!") // Function returning a channel.
    for i := 0; i < 5; i++ {
        fmt.Printf("You say: %q\n", <-c)
    }
    fmt.Println("You're boring; I'm leaving.")
```

Run

```
func boring(msg string) <-chan string { // Returns receive-only channel of strings.
    c := make(chan string)
    go func() { // We launch the goroutine from inside the function.
        for i := 0; ; i++ {
            c <- fmt.Sprintf("%s %d", msg, i)
            time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)
        }
    }()
    return c // Return the channel to the caller.
}
```

Channels as a handle on a service

Our boring function returns a channel that lets us communicate with the boring service it provides.

We can have more instances of the service.

```
func main() {
    joe := boring("Joe")
    ann := boring("Ann")
    for i := 0; i < 5; i++ {
        fmt.Println(<-joe)
        fmt.Println(<-ann)
    }
    fmt.Println("You're both boring; I'm leaving.")
}
```

Run

Multiplexing

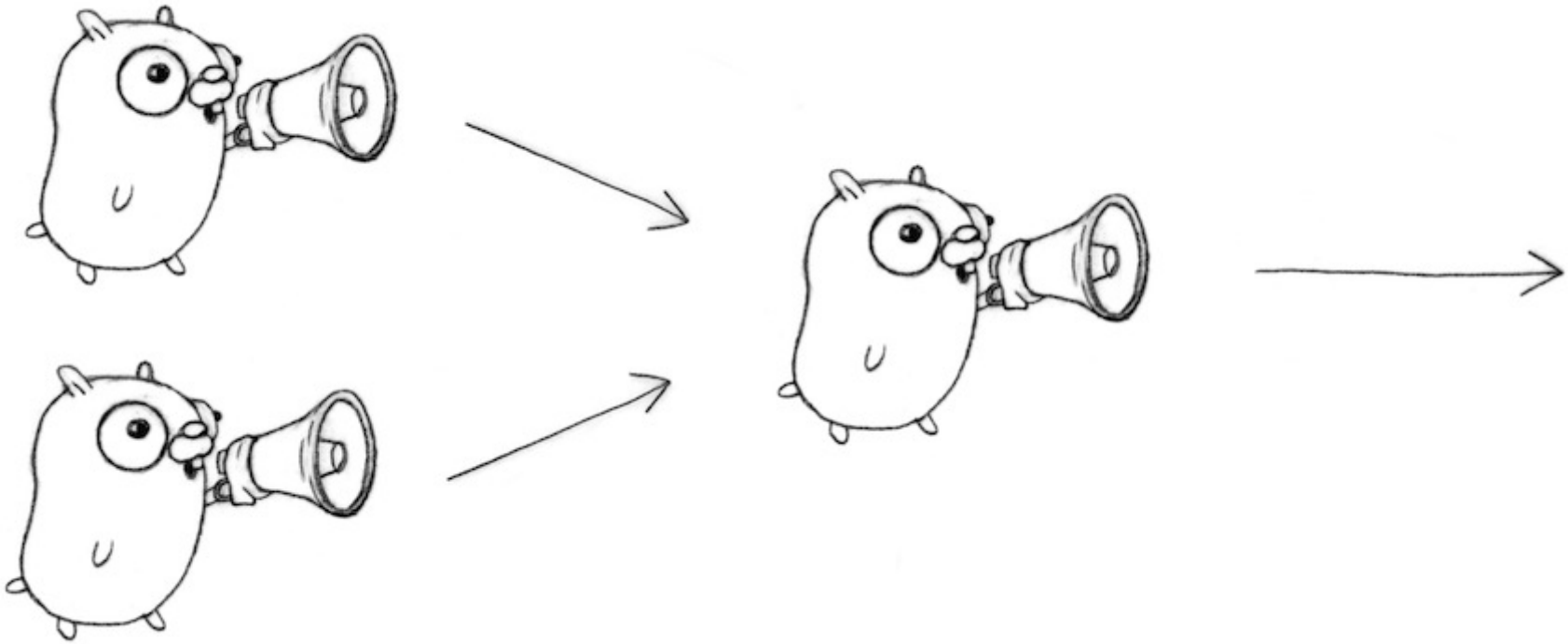
These programs make Joe and Ann count in lockstep.
We can instead use a fan-in function to let whosoever is ready talk.

```
func fanIn(input1, input2 <-chan string) <-chan string {
    c := make(chan string)
    go func() { for { c <- <-input1 } }()
    go func() { for { c <- <-input2 } }()
    return c
}
```

```
func main() {
    c := fanIn(boring("Joe"), boring("Ann"))
    for i := 0; i < 10; i++ {
        fmt.Println(<-c)
    }
    fmt.Println("You're both boring; I'm leaving.")
}
```

Run

Fan-in



Restoring sequencing

Send a channel on a channel, making goroutine wait its turn.

Receive all messages, then enable them again by sending on a private channel.

First we define a message type that contains a channel for the reply.

```
type Message struct {
    str  string
    wait chan bool
}
```

Restoring sequencing.

Each speaker must wait for a go-ahead.

```
    for i := 0; i < 5; i++ {
        msg1 := <-c
        fmt.Println(msg1.str)
        msg2 := <-c
        fmt.Println(msg2.str)
        msg1.wait <- true
        msg2.wait <- true
    }
```

```
    waitForIt := make(chan bool) // Shared between all messages.
```

```
            c <- Message{fmt.Sprintf("%s: %d", msg, i), waitForIt}
            time.Sleep(time.Duration(rand.Intn(2e3)) * time.Millisecond)
            <-waitForIt
```

Run

Select

A control structure unique to concurrency.

The reason channels and goroutines are built into the language.

Select

The select statement provides another way to handle multiple channels.
It's like a switch, but each case is a communication:

- All channels are evaluated.
- Selection blocks until one communication can proceed, which then does.
- If multiple can proceed, select chooses pseudo-randomly.
- A default clause, if present, executes immediately if no channel is ready.

```
select {
case v1 := <-c1:
    fmt.Printf("received %v from c1\n", v1)
case v2 := <-c2:
    fmt.Printf("received %v from c2\n", v1)
case c3 <- 23:
    fmt.Printf("sent %v to c3\n", 23)
default:
    fmt.Printf("no one was ready to communicate\n")
}
```

Fan-in again

Rewrite our original fanIn function. Only one goroutine is needed. Old:

```
func fanIn(input1, input2 <-chan string) <-chan string {
    c := make(chan string)
    go func() { for { c <- <-input1 } }()
    go func() { for { c <- <-input2 } }()
    return c
}
```

Fan-in using select

Rewrite our original fanIn function. Only one goroutine is needed. New:

```
func fanIn(input1, input2 <-chan string) <-chan string {
    c := make(chan string)
    go func() {
        for {
            select {
            case s := <-input1:  c <- s
            case s := <-input2:  c <- s
            }
        }
    }()
    return c
}
```

Run

Timeout using select

The time.After function returns a channel that blocks for the specified duration. After the interval, the channel delivers the current time, once.

```
func main() {
    c := boring("Joe")
    for {
        select {
        case s := <-c:
            fmt.Println(s)
        case <-time.After(1 * time.Second):
            fmt.Println("You're too slow.")
            return
        }
    }
}
```

Run

Timeout for whole conversation using select

Create the timer once, outside the loop, to time out the entire conversation.
(In the previous program, we had a timeout for each message.)

```
func main() {
    c := boring("Joe")
    timeout := time.After(5 * time.Second)
    for {
        select {
        case s := <-c:
            fmt.Println(s)
        case <-timeout:
            fmt.Println("You talk too much.")
            return
        }
    }
}
```

Run

Quit channel

We can turn this around and tell Joe to stop when we're tired of listening to him.

```
quit := make(chan bool)
c := boring("Joe", quit)
for i := rand.Intn(10); i >= 0; i-- {
    fmt.Println(<-c)
}
quit <- true
```

```
go func() {
    for i := 0; ; i++ {
        time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)
        select {
        case c <- fmt.Sprintf("%s: %d", msg, i):
            // do nothing
        case <-quit:
            return
        }
    }
}()
```

Run

Receive on quit channel

How do we know it's finished? Wait for it to tell us it's done: receive on the quit channel

```
quit := make(chan string)
c := boring("Joe", quit)
for i := rand.Intn(10); i >= 0; i-- {
    fmt.Println(<-c)
}
quit <- "Bye!"
fmt.Printf("Joe says: %q\n", <-quit)
```

```
        select {
        case c <- fmt.Sprintf("%s: %d", msg, i):
            // do nothing
        case <-quit:
            cleanup()
            quit <- "See you!"
            return
        }
```

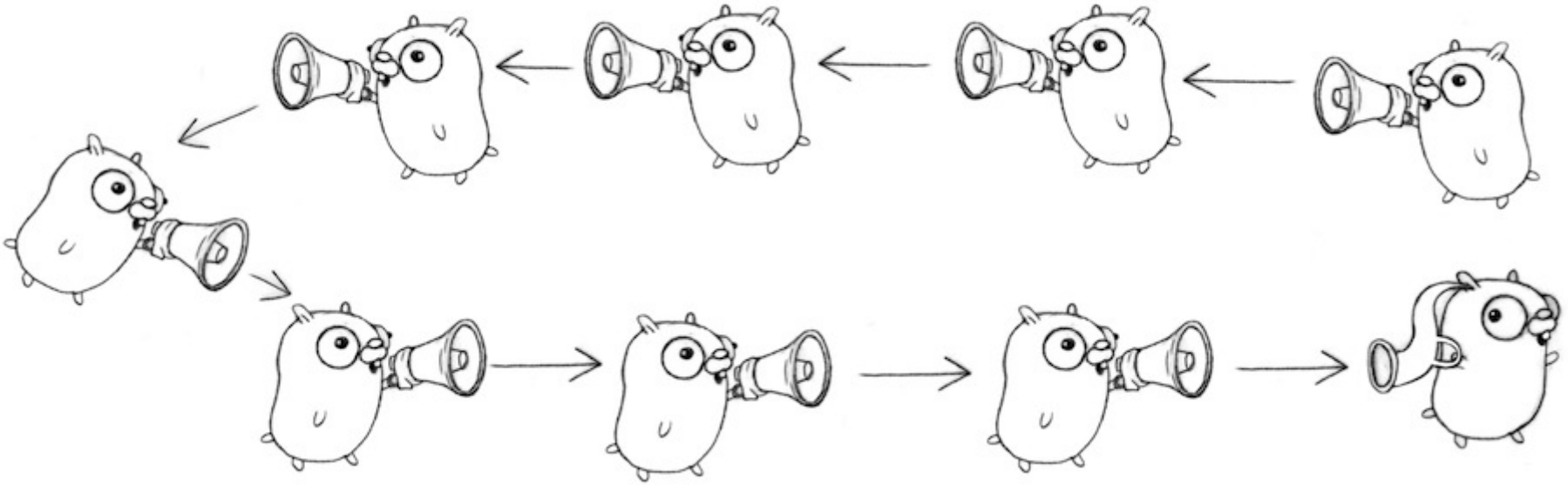
Run

Daisy-chain

```
func f(left, right chan int) {
    left <- 1 + <-right
}

func main() {
    const n = 10000
    leftmost := make(chan int)
    right := leftmost
    left := leftmost
    for i := 0; i < n; i++ {
        right = make(chan int)
        go f(left, right)
        left = right
    }
    go func(c chan int) { c <- 1 }(right)
    fmt.Println(<-leftmost)
}
```

Run

Chinese whispers, gopher style



Conclusions

Goroutines and channels make it easy to express complex operations dealing with:

- multiple inputs
- multiple outputs
- timeouts
- failure

And they're fun to use.

Conclusions - II

Go (https://golang.org) improves on C/C++/Java/... and addresses C/C++ and python deficiencies:

- code distribution
- code installation
- compilation/development speed
- runtime speed
- simple language

and:

- serviceable standard library (stdlib doc (https://golang.org/pkg))
- builtin facilities to tackle concurrency programming

Conclusions - III

Don't communicate by sharing memory, share memory by communicating.

Go is great at writing small and large (concurrent) programs.
Also true for **science-y** programs, even if the amount of libraries can still be improved.

Write your next tool/analysis/simulation/software in Go (https://golang.org/) ?

Acknowledgements / resources

tour.golang.org (https://tour.golang.org)

talks.golang.org/2012/splash.slide (https://talks.golang.org/2012/splash.slide)

talks.golang.org/2012/goforc.slide (https://talks.golang.org/2012/goforc.slide)

talks.golang.org/2012/waza.slide (https://talks.golang.org/2012/waza.slide)

talks.golang.org/2012/concurrency.slide (https://talks.golang.org/2012/concurrency.slide)

talks.golang.org/2013/advconc.slide (https://talks.golang.org/2013/advconc.slide)

talks.golang.org/2014/gocon-tokyo.slide (https://talks.golang.org/2014/gocon-tokyo.slide)

talks.golang.org/2015/simplicity-is-complicated.slide (https://talks.golang.org/2015/simplicity-is-complicated.slide)

talks.golang.org/2016/applicative.slide (https://talks.golang.org/2016/applicative.slide)

agenda.infn.it/getFile.py/access?contribId=24&sessionId=3&resId=0&materialId=slides&confId=11680 (https://agenda.infn.it/getFile.py/access?

contribId=24&sessionId=3&resId=0&materialId=slides&confId=11680)

Backup

Interlude: concurrency & parallelism

Interlude: Sequential, Concurrent & Parallel pizzas

Pizza recipe

(**Disclaimer:** don't ever eat any pizza prepared or cooked by me.)

How to prepare a (sequential) pizza?

```
      program main

      call pizza()

      stop
      end

      subroutine pizza()
c ... the special sauce ...
          write(*,*) 'making a pizza...'
      return
      end
```

Estimated time (1 chef, 1 pizza):

```
xx-oooo-xxx-oo-###
```

How to make this faster?

(Sequential) Pizza recipe

Tasks:

- wash tomatoes and onions
- cut tomatoes, onions
- prepare pizza dough
- put tomato sauce on top of pizza dough
- toppings: put tomatoes, onions, ham and mozarella
- (pre-)heat oven, bake
- (cut, then eat)

Estimated time (1 chef, 1 pizza):

```
xx-oooo-xxx-oo-###
```

Concurrent pizzas - Parallel pizzas

Estimated time (1 chef, 1 kitchen, 2 pizzas):

```
xx-oooo-xxx-oo-###-xx-oooo-xxx-oo-###
```

Estimated time (1 chef, 2 kitchens, 2 pizzas):

```
xx-oooo-xxx-oo+###
              +xx-oooo-xxx-oo-###
```

Estimated time (2 chefs, 1 kitchen, 2 pizzas):

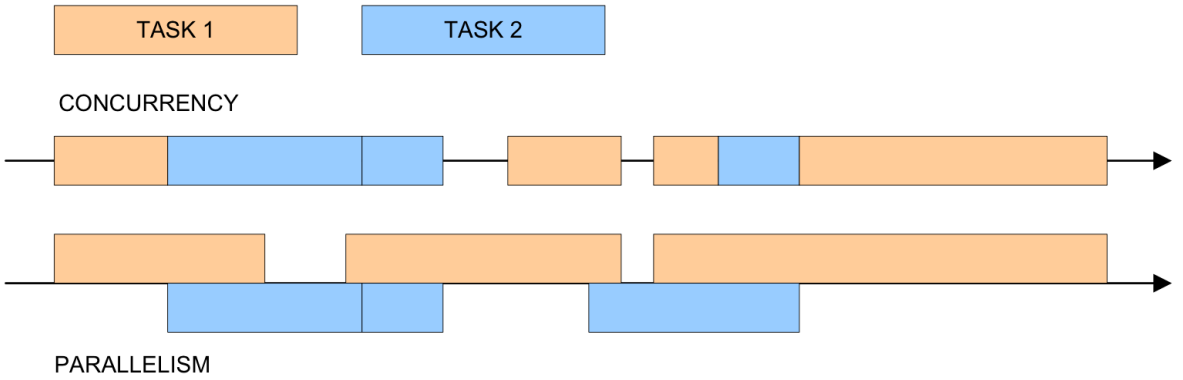
```
xx-xxx-+-xx-xxx-+
       +###     +###
oooo-oo+-oooo-oo+
```

Estimated time (2 chefs, 2 kitchens, 2 pizzas):

```
xx-oooo-xxx-oo-###
xx-oooo-xxx-oo-###
```

Interlude: concurrency & 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*.

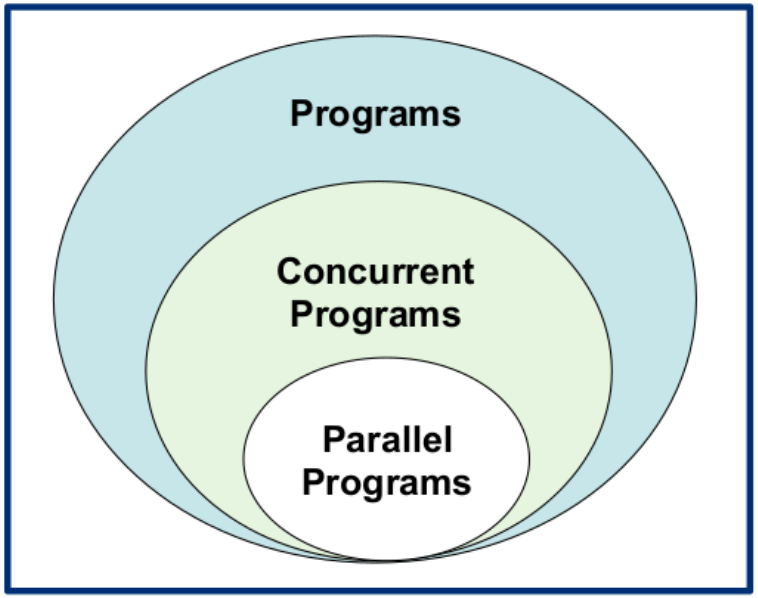


Concurrency is a way to structure a program by breaking it into pieces that can be executed independently.
Communication is the means to coordinate the independent executions.

Concurrency vs Parallelism

Concurrency is about dealing with lots of things at once.
Parallelism is about doing lots of things at once.

Concurrency is about (program) **structure**.
Parallelism is about (program) **execution**.



Concurrency is **not** parallelism, it's better :)

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)

Concurrency strategies

Multi-processing

Launch *N* instances of an application on a node with *N* cores

- re-use pre-existing code
- *a priori* no required modification of pre-existing code
- satisfactory *scalability* with the number of cores

But:

- resource requirements increase with the number of processes
- memory footprint **increases**
- as do other O/S (limited) resources (file descriptors, network sockets, ...)
- scalability of **I/O** debatable when number of cores > ~100

Multi-threading

C++11/14 libraries do help a bit:

- `std::lambda`, `std::thread`, `std::promise`
- (Intel) Threading Building Blocks
- ...

Thank you

Sebastien Binet
CNRS/IN2P3/LPC-Clermont
https://github.com/sbinet (https://github.com/sbinet)
@0xbins (http://twitter.com/0xbins)
sebastien.binet@clermont.in2p3.fr (mailto:sebastien.binet@clermont.in2p3.fr)