

Go tour

Sébastien Binet



2012-05-31

Outline

- Motivations
- Bootstrap, setup environment
- Full speed

Motivations

- Moore's law ceased to provide the traditional single-threaded performance increases
 - ▶ clock-frequency wall of 2003
 - ▶ still deliver increases in **transistor density**
- multicore systems become the norm
- need to “go parallel” to get scalability

In a C++ world...

- parallel programming in C++ is **doable**:
 - ▶ C/C++ “locking + threads” (pthreads, WinThreads)
 - ★ excellent performance
 - ★ good generality
 - ★ relatively **low productivity**
 - ▶ multi-threaded applications...
 - ★ hard to get right
 - ★ hard to **keep** right
 - ★ hard to **keep** efficient and optimized across releases
 - ▶ multi-process applications...
 - ★ leverage fork+COW on GNU/Linux

Parallel programming in C++ is **doable**,
but ***no panacea***

In a C++ world...

- in C++03, we have libraries to help with parallel programming
 - ▶ `boost::lambda`
 - ▶ `boost::MPL`
 - ▶ `boost::thread`
 - ▶ Threading Building Blocks (TBB)
 - ▶ Concurrent Collections (CnC)
 - ▶ OpenMP
 - ▶ ...

In a C++11 world...

- in C++11, we get:
 - ▶ λ functions (and a new syntax to define them)
 - ▶ `std::thread`,
 - ▶ `std::future`,
 - ▶ `std::promise`

Helps taming the beast
... at the price of sprinkling templates everywhere...
... and complicating further a not so simple language...

In a C++11 world...

yay! for C++11, but old problems are **still there...**

- **build scalability**

- ▶ templates
- ▶ headers system
- ▶ still no module system (WG21 - N2073)
 - ★ maybe in the next Technical Report ?

- **code distribution**

- ▶ no CPAN like readily available infrastructure (and cross-platform) for C++

Time for a new language ?

“Successful new languages build on existing languages and where possible, support legacy software. C++ grew out of C. java grew out of C++. To the programmer, they are all one continuous family of C languages.” (T. Mattson)

- notable exception (which confirms the rule): **python**

Can we have a language:

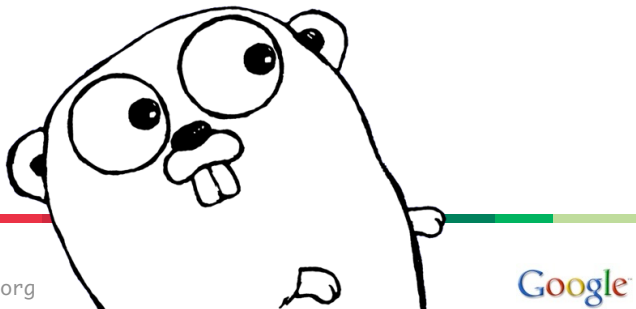
- as easy as **python**,
- as fast (or nearly as fast) as C/C++/FORTRAN,
- with none of the deficiencies of C++,
- and is multicore/manycore friendly ?

Why not Go ?
golang.org

Elements of go

- obligatory hello world example...

```
package main
import "fmt"
func main() {
    fmt.Println("Hello, World")
}
```



<http://golang.org>



Elements of go - II

- founding fathers:
 - ▶ Russ Cox, Robert Griesemer, Ian Lance Taylor
 - ▶ Rob Pike, Ken Thompson
- concurrent, compiled
- garbage collected
- an open-source general programming language
- best of both 'worlds':
 - ▶ feel of a dynamic language
 - ★ limited verbosity thanks to **type inference system**, map, slices
 - ▶ safety of a static type system
 - ▶ compiled down to machine language (so it is fast)
 - ★ goal is within 10% of C
- object-oriented (but w/o classes), builtin reflection
- first-class functions with closures
- duck-typing à la python

goroutines

- a function executing concurrently as other goroutines **in the same address space**
- starting a goroutine is done with the `go` keyword
 - ▶ `go myfct(arg1, arg2)`
- growable stack
 - ▶ **lightweight threads**
 - ▶ starts with a few kB, grows (and shrinks) as needed
 - ★ now, also available in GCC 4.6 (thanks to the GCC-Go front-end)
 - ▶ no stack overflow

channels

- provide (type safe) communication and synchronization

```
// create a channel of mytype
```

```
my_chan := make(chan mytype)
```

```
my_chan <- some_data    // sending data
```

```
some_data = <- my_chan  // receiving data
```

- send and receive are atomic

*"Do not communicate by sharing memory; instead,
share memory by communicating"*

Non-elements of Go

- **no** dynamic libraries (frown upon)
- **no** dynamic loading (yet)
 - ▶ but can either rely on separate processes
 - ★ IPC is made easy *via* the `netchan` package
 - ★ many RPC substrates too (JSON, XML, `protobuf`, ...)
 - ▶ or rebuild executables on the fly
 - ★ **compilation** of Go code is **fast**
 - ★ even faster than FORTRAN and/or C
- **no** templates/generics
 - ▶ still open issue
 - ▶ looking for the proper Go -friendly design
- **no** operator overloading

Go from anywhere to everywhere

- code compilation and distribution are (*de facto*) standardized
- put your code on some repository
 - ▶ bitbucket, launchpad, googlecode, github, ...
- check out, compile and install in one go with **go get**:
 - ▶ `go get bitbucket.org/binet/igo`
 - ▶ no root access required
 - ▶ automatically handle **dependencies**
- `go get` -able packages are listed on the dashboard:
 - ▶ godashboard.appspot.com

Bootstrap

- install Go from <http://golang.org/doc/install>
- check the environment. e.g.:

```
$ go version && go env
go version go1.0.1
GOROOT="/usr/lib/go"
GOBIN=""
GOARCH="amd64"
GOCHAR="6"
GOOS="linux"
GOEXE=""
GOHOSTARCH="amd64"
GOHOSTOS="linux"
GOTOOLDIR="/usr/lib/go/pkg/tool/linux_amd64"
GOGCCFLAGS="-g -O2 -fPIC -m64 -pthread"
CGO_ENABLED="1"
```


Bootstrap - II

```
$ cd dev/go
$ mkdir -p repos/{src,pkg,bin} && cd repos
$ tree
.
|-- bin
|-- pkg
+-- src

$ pwd
~/dev/go/repos

$ export GOPATH='pwd'
$ export PATH=${GOPATH}/bin:${PATH}
```

Discovering the environment

- getting documentation:

```
$ godoc fmt | less
```

```
$ godoc fmt Printf
```

- or using the html frontend:

```
$ godoc -http=:8080 &
```

```
$ open http://localhost:8080
```

Quick & dirty development: go run

```
$ cat ./hello.go
```

```
package main
import "fmt"
func main() {
    fmt.Println("Hello, World")
}
```

```
$ go run ./hello.go
Hello, World
```

“Serious” development

```
$ cd $GOPATH/src  
$ mkdir -p greet  
$ cat > src/greet/foo.go
```

```
package greet  
import "fmt"  
func Greet(name string) {  
    fmt.Println("Hello", name)  
}
```

```
# compiles and installs the 'greet' package  
$ go install greet
```

“Serious” development - II

The greet package can now be used by a command:

- via `go run`:
 - ▶ modify the previous `hello.go` file to use `greet.Greet`
- or with a proper command:

compiles and installs the 'greeter' command

```
$ mkdir -p $GOPATH/src/greeter
```

```
$ cp hello.go $GOPATH/src/greeter/main.go
```

```
$ go install greeter
```

```
$ greeter
```

```
"Hello World"
```

- modify `greet/foo.go` and recompile+rerun greeter:

```
$ go install greeter && greeter
```

```
"Hello [World]"
```

The Go tour

```
$ go get code.google.com/p/go-tour/gotour
```

```
$ gotour
```

Serving content from ~/dev/go/repos/src/code.google.com/p.

Open your web browser and visit <http://127.0.0.1:3999/>

A simple load balancer

A unit of work:

```
type Work struct {  
    x, y, z int  
}
```

A simple load balancer - II

A worker task:

```
func worker(in <-chan *Work, out chan<- *Work) {  
    for w := range in {  
        w.z = w.x * w.y  
        Sleep(w.z)  
        out <- w  
    }  
}
```

Must make sure other workers can run when one blocks.

A simple load balancer - III

The runner:

```
func Run() {  
    in, out := make(chan *Work), make(chan *Work)  
    for i := 0; i < NumWorkers; i++ {  
        go worker(in, out)  
    }  
    go sendLotsOfWork(in)  
    receiveLotsOfResults(out)  
}
```

Easy problem but also hard to solve concisely without concurrency.

Concurrency enables parallelism

The load balancer is **implicitly parallel** and **scalable**.

`NumWorkers` could be huge.

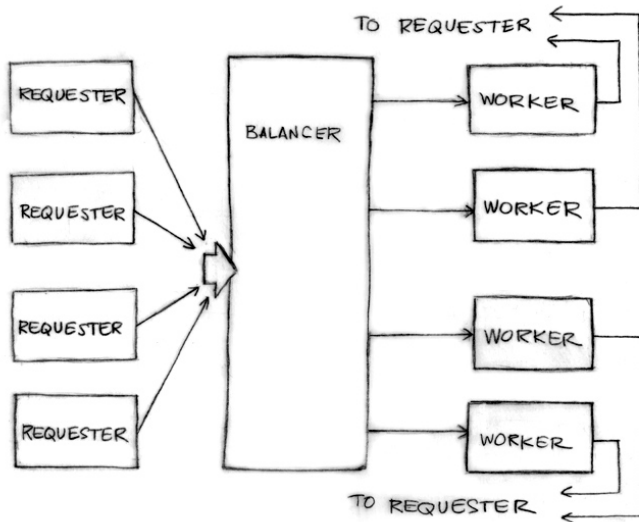
The tools of concurrency make it almost trivial to build a safe, working, scalable, parallel design.

Concurrency simplifies synchronization

No **explicit** synchronization needed.

The structure of the program is **implicitly synchronized**.

A more complex 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`.

- Heap interface: <http://golang.org/pkg/container/heap/>

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)  
}
```

- 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.

GO



<http://golang.org>

Bibliography

- <http://golang.org>
- <http://tour.golang.org>
- <http://concur.rspace.googlecode.com/hg/talk/concur.html>