Go-HEP

Sébastien Binet

LAL/IN2P3

2014-10-15

Introduction

- 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
    - ★ event-level based parallelism

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++
  - remember ROOT/BOOT ? (CHEP-06)

Time for a new language ?

> *"Successful new languages build on existing languages and where possible, support legacy software. C++ grew our 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 JI-2014")
}
```

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` (but better) thanks to its interfaces

Go concurrent

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

Go concurrent - II

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
  - 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 github.com/go-hep/fwk`
  - no `root` access required
  - automatically (and recursively) handle dependencies

Go and C/C++

Interfacing with `C`:

- done with the `CGo` foreign function interface
- `#include` the header file to the `C` library to be wrapped
- access the `C` types and functions under the artificial "C" package

```
package myclib
// #include <stdio.h>
// #include <stdlib.h>
import "C"
import "unsafe"

func foo(s string) {
  c_str := C.CString(s)  // create a C string from a G
  C.fputs(c_str, C.stdout)
  C.free(unsafe.Pointer(c_str))
}
```

Go and C/C++

Interfacing with `C++`:

- a bit more involved
- uses `SWIG`
  - you write the `SWIG` interface file for the library to be wrapped
  - `SWIG` will generate the `C` stub functions
  - which can then be called using the `CGo` machinery
  - the `Go` files doing so are automatically generated as well
- handles overloading, multiple inheritance
- allows to provide a `Go` implementation for a `C++` abstract class

Problem

`SWIG` doesn't understand all of `C++03`

- *e.g.* can't parse `TObject.h`

Conclusions

Can `Go` address the (non-) multicore problems of yesterday ?

- yes:
  - productivity (dev cycle of a scripting language)
  - build scalability (package system)
  - deployment (go-get, simple `scp`, ease of cross-compilation)
  - support for "legacy" `C/C++/Fortran` software (cgo+swig)

Can `Go` address the multicore issues of today/tomorrow ?

- yes:
  - easier to write concurrent code with the builtin abstractions (goroutines, channels)
  - easier to have efficient concurrent code (stack management)
  - still have to actually write efficient concurrent code, though...
    - ⋆ work partitioning, load balancing, ...
- **but:** no such thing as a magic wand for multicores/manycores

References

General Go pointers

- golang.org
- talks.golang.org
- blog.golang.org
- tour.golang.org
- dave.cheney.net/resources-for-new-go-programmers
- gobyexample.net
- godoc.org
- golang nuts mailing list

Science-related Go pointers

- gonum: BLAS/LAPACK,numerics,...packages
- go-hep: HEP related packages

Workshop

- simple exercizes (create a command, handle arguments)
- discover a bit of the surrounding tooling ecosystem:
  - doc system,
  - build system,
  - `CPAN/PyPI/...`-like market-store;
- discover a bit of the standard library (`json`, `io`, `os`, `...`)
- touch upon `interfaces` and concurrency:
  - `channels`
  - `goroutines`

`github.com/sbinet/ji-2014-go`