Go in High Energy Physics & Cosmology: computing, monitoring and concurrency

Sébastien Binet
CNRS/IN2P3/LPC
binet@clermont.in2p3.fr
@0xb1ns

Séminaire Aristote

2016-06-30

HEP (High Energy Physics)

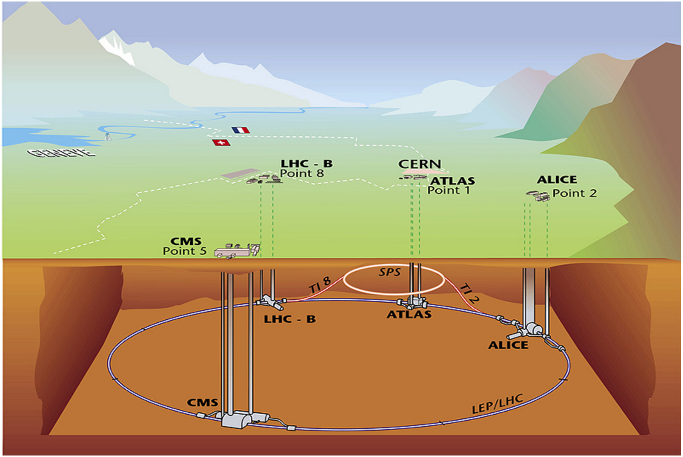
Field of physics which studies the fundamental laws of Nature and the properties of the constituents of matter.

Many labs working on HEP around the world. But, perhaps one of the most famous ones is `CERN`.

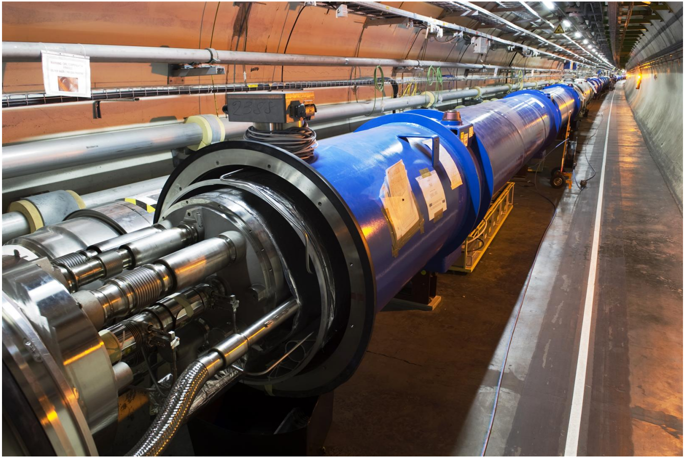
CERN

CERN-LHC

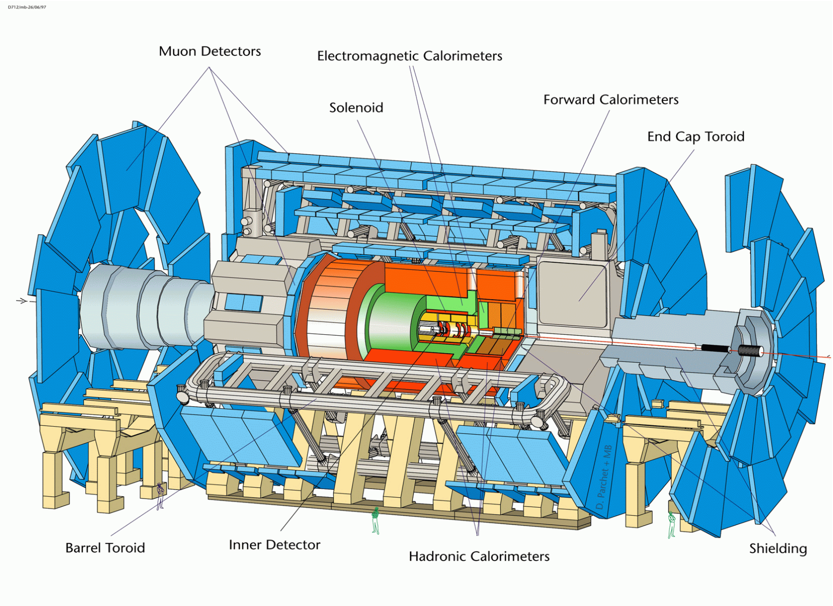
LHC: Large Hadron Collider. A proton-proton collider of 27km of circumference.



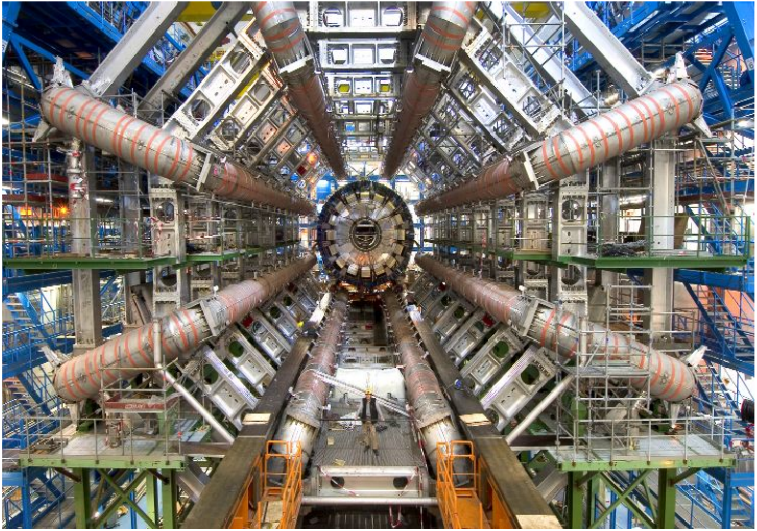
LHC tunnel and one of the ~1200 dipole magnets



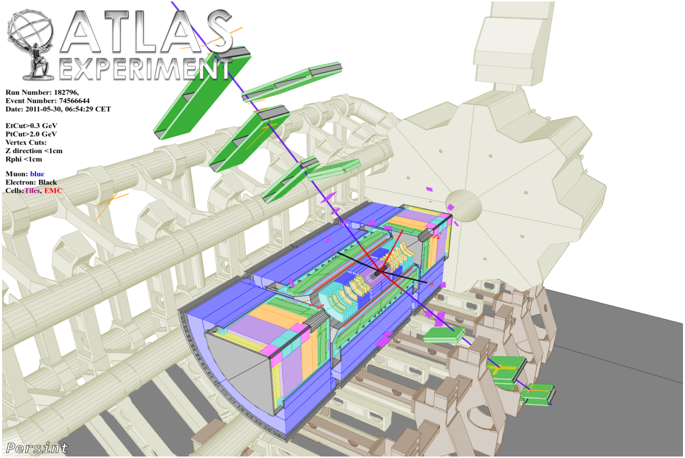
ATLAS detector (44m x 25m)



ATLAS installation



Results


ATLAS
EXPERIMENT
Run Number: 182796,
Event Number: 74566644
Date: 2011-05-30, 06:54:29 CET
EtCut>0.3 GeV
PtCut>2.0 GeV
Vertex Cuts:
Z direction <1cm
Rphi <1cm
Muon: blue
Electron: Black
Cells:Tiles, EMC
Persint

High level view of the data pipeline

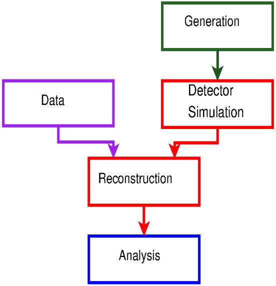
Data is collected at the 4 interaction points

- collisions every 25 ns
- ˜1 Mb (compressed) / collision
- ˜10 Pb/year of raw data

Raw data is then filtered to only keep "interesting" events (collisions of protons)

Main steps of data analysis/massaging

- **Generation:** production of a single physics event (*e.g.:* a collision and its decay products)
- **Simulation:** modelling interactions between particles and detector material
- **Reconstruction:** building physics objects (electrons, photons, ...) from the detector signals (energy deposits)
- **Analysis:** testing hypotheses against the reconstruction output



Software in HEP

Historically, software in HEP has been written in **FORTRAN-77**.

- HEP people even wrote compilers
- HEP community even defined a few extensions (**MORTRAN**)

Mid-90's: migration to **C++**
Mid-2000's: **Python** gained tremendous mindshare

- first thru the steering of **C++** binaries
- then as complete analyses gluing **C++** libraries together

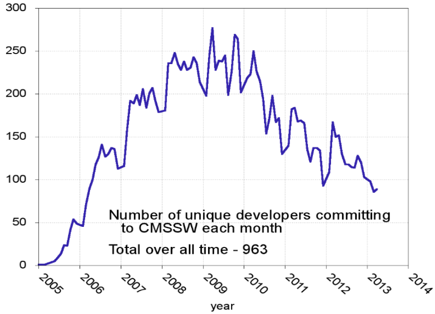
An LHC experiment (*e.g.* ATLAS, CMS) is ~3000 physicists but only a fraction of those is developing code.
Reconstruction frameworks grew from ~3M SLOC to ~5M
Summing over all HEP software stack for e.g. ATLAS:

- event generators: ~1.4M SLOC (C++, FORTRAN-77)
- I/O libraries ~1.7M SLOC (C++)
- simulation libraries ~1.2M SLOC (C++)
- reconstruction framework ~5M SLOC (C++) + steering/configuration (~1M SLOC python) (want to have a look at the `ATLAS code`? `CMS code`?)

GCC: ~7M SLOC
Linux kernel 3.6: 15.9M SLOC

People committing code to VCS per month

Wide variety of skill level Large amount of churn Once the physics data is pouring, people go and do physics instead of software



See also "The Life Cycle of HEP Offline Software", P.Elmer, L. Sexton-Kennedy, C.Jones, CHEP 2007

Software developers

~300 active developers (per experiment)
~1000 different developers integrated over the lifetime of a single LHC experiment.

- few "real" s/w experts
- some physicists with strong skill set in s/w
- many with some experience in s/w development
- some with **no** experience in s/w development

A multi-timezone environment

- Europe, North-America, Japan, Russia

Many communities (core s/w people, generators, simulation, ...)
Development and infrastructures usually CERN-centric
Heavily Linux based (`Scientific Linux CERN`)

Software development cycle

VCS (CVS, then SVN. GIT: almost there)
Nightlies (Jenkins or homegrown solution)

- need a sizeable cluster of build machines (distcc, ccache, ...)
- builds the framework stack in ~8h
- produces ~2000 shared libraries
- installs them on AFS (also creates RPMs and tarballs)

Devs can then test and develop off the nightly *via* AFS
Every 6 months or so a new production release is cut, validated (then patched) and deployed on the World Wide LHC Computing Grid (WLCG).
Release size: **~5Gb**

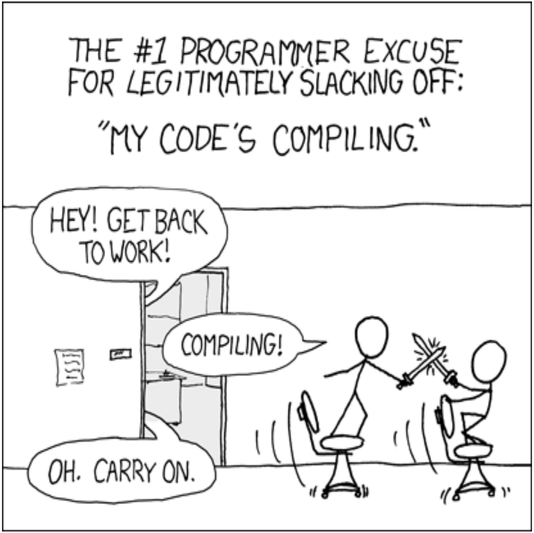
- binaries, libraries (externals+framework stack)
- extra data (sqlite files, physics processes' modelisation data, ...)

Software runtime ?

Big science, big data, big software, big numbers

- ~1min to initialize the application
- loading >500 shared libraries
- connecting to databases (detector description, geometry, ...)
- instantiating ~2000 C++ components
- 2Gb/4Gb memory footprint per process

(obligatory xkcd reference)



We learn to love hating our framework. (every step of the way)
And even more so in the future:

- work to make our software stack thread-safe
- or at least parts of it multithread friendly to harness multicore machines
- quite a lot of fun ahead

Remember Go ?

- compiles quickly (no warnings, imports)
- enforces coherent coding rules (across projects)
- builtin test/benchmark/documentation facilities
- deploys easily, cross-compiles easily
- installs easily (also 3rd-party packages: *"go get"*)
- fast to pick up, not as complicated as **C++**
- builtin reflection system
- builtin (de)serialization capabilities
- concurrency support
- garbage collected

Perfect match for many HEP use cases.

Migrating to Go ? (evil plan for (HEP) world domination)

Migrating ~5M SLOC of C++ code to Go, during data taking, **unfortunately**, won't fly.
Creating new applications for data massaging or post-processing **might**.
Creating a new concurrent and parallel framework for the next accelerator **might**.
Need to build a critical mass of Go HEP enthusiasts
So far:

- building the packages to read/write data in HEP formats (see under `go-hep`)
- built a proof of concept of a concurrent framework: `go-hep/gaudi-fwk`
- now building the real thing `go-hep/fwk`
- building a physics simulation detector app on top of go-hep/fwk: `go-hep/fads`
- building a package of data analysis facilities

Experience so far

Wrapping `C++` libraries (via a C-shim) with cgo is ˜OK

- time consuming (no surprise)
- as with `Python`, you don't want to cross language boundaries too frequently (perfs!)
- using the `SWIG` support wasn't possible (because of some `C++` constructs not supported by `SWIG` 's parser)

Experience so far

Experience pre-Go-1.5: No shared libraries.

- plugins system (heavily used in our reconstruction frameworks) use shared libraries
- no, funneling data through some IPC won't fly (for ATLAS use case)
- actually not a limitation: re-compile on the fly, fork-exec (and still faster than booting `Python` VM + loading shared libraries, plus you get a static binary)

now, with `Go` $>$ `1.5` (August-2015): shared libraries are here.

Experience so far

Building small (and not so small) command-line utilities (a la `git`) is fun.

- Wrote a build tool that way (`hwaf`)

Building a concurrent framework is also fun and surprisingly easy (thanks to `goroutines` and `channels`.)

- see `go-fwk` and `go-hep@ACAT-2011` for more details

Experience so far

No generics/templates, no operator overloading.
In my experience, this hasn't been a limitation.
Operator overloading isn't a panacea.
Generics can easily be implemented with:

```
$ gofmt -r 'T -> MyType' tmpl.go > mytype.go
```

(without the build-time cost that C++ templates impose) (w/o all the benefits of C++ templates. More of a pain point for framework implementers than users, though.)

What about number crunching ?

The gc compiler is improving, especially the (yet to be released (in August-2016)) `go-1.7` version:

- brings a SSA backend for `amd64`
- (more) SSE+AVX instructions

Number crunching `Go` programs can outperform `C++` programs though:

- `3photons`: a toy Monte-Carlo simulation program of `e+e-`⇒`3photons` collisions

```
## C++ (serial)
$ time ./mc
real   0m36.733s
user   0m36.710s
sys    0m0.000s
```

```
## Go-1.6.2 (serial)
$ time ./3photons
real   0m30.075s
user   0m30.210s
sys    0m0.020s
```

```
## Go-1.7b2 (serial)
$ time./3photons-1.7b2
real   0m23.832s
user   0m23.793s
sys    0m0.037s
```

Application to HEP

Using `go-hep/fads` as a guinea pig and poster child...

fads

`fads` is a "FAst Detector Simulation" toolkit.

- morally a translation of `C++-Delphes` into Go
- uses `go-hep/fwk` to expose, manage and harness concurrency into the usual `HEP` event loop (`initialize` — `process-events` — `finalize`)
- uses `go-hep/hbook` for histogramming, `go-hep/hepmc` for `HepMC` input/output

Code is on github (BSD-3):
`github.com/go-hep/fwk`
`github.com/go-hep/fads`
Documentation is served by `godoc.org`:
`godoc.org/github.com/go-hep/fwk`
`godoc.org/github.com/go-hep/fads`

go-hep/fads - Installation

As easy as:

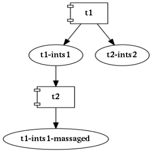
```
$ go get github.com/go-hep/fads/...
```

Yes, with the ellipsis at the end, to also install sub-packages.

- go get will recursively download and install all the packages that `go-hep/fads` depends on. (no `Makefile` needed)

```
$ fwk-ex-tuto-1 -help
Usage: fwk-ex-tuto1 [options]

ex:
 $ fwk-ex-tuto-1 -l=INFO -evtmax=-1

options:
  -evtmax=10: number of events to process
  -l="INFO": message level (DEBUG|INFO|WARN|ERROR)
  -nprocs=0: number of events to process concurrently
```

Runs 2 tasks.



go-hep/fwk - Examples

```
$ fwk-ex-tuto-1
::: fwk-ex-tuto-1...
t2                     INFO configure...
t2                     INFO configure... [done]
t1                     INFO configure ...
t1                     INFO configure ... [done]
t2                     INFO start...
t1                     INFO start...
app                    INFO >>> running evt=0...
t1                     INFO proc... (id=0|0) => [10, 20]
t2                     INFO proc... (id=0|0) => [10 -> 100]
[...]
app                    INFO >>> running evt=9...
t1                     INFO proc... (id=9|0) => [10, 20]
t2                     INFO proc... (id=9|0) => [10 -> 100]
t2                     INFO stop...
t1                     INFO stop...
app                    INFO cpu: 654.064us
app                    INFO mem: alloc:             62 kB
app                    INFO mem: tot-alloc:         74 kB
app                    INFO mem: n-mallocs:        407
```

`fwk` enables: - event-level concurrency - tasks-level concurrency
`fwk` relies on `Go`'s runtime to properly schedule *goroutines*.
For sub-task concurrency, users are by construction required to use `Go`'s constructs (*goroutines* and *channels*) so everything is consistent **and** the *runtime* has the **complete picture**.

- **Note:** `Go`'s runtime isn't yet *NUMA-aware*. A proposal *(June-2015)* is in the `works`.

go-hep/fads - real world use case

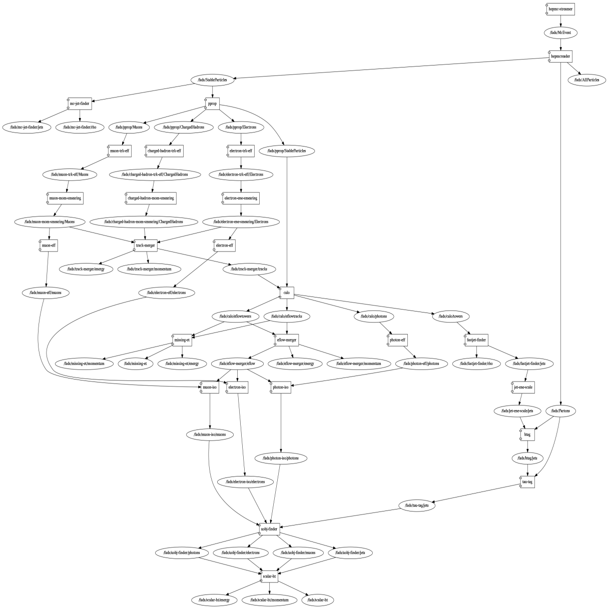
- translated C++-Delphes' ATLAS data-card into Go
- go-hep/fads-app
- installation:

```
$ go get github.com/go-hep/fads/cmd/fads-app
$ fads-app -help
Usage: fads-app [options] <hepmc-input-file>

ex:
 $ fads-app -l=INFO -evtmax=-1 ./testdata/hepmc.data

options:
  -cpu-prof=false: enable CPU profiling
  -evtmax=-1: number of events to process
  -l="INFO": log level (DEBUG|INFO|WARN|ERROR)
  -nprocs=0: number of concurrent events to process
```

- a HepMC converter
- particle propagator
- calorimeter simulator
- energy rescaler, momentum smearer
- isolation
- b-tagging, tau-tagging
- jet-finder (reimplementation of FastJet in Go: go-hep/fastjet)
- histogram service (from go-hep/fwk)

Caveats:

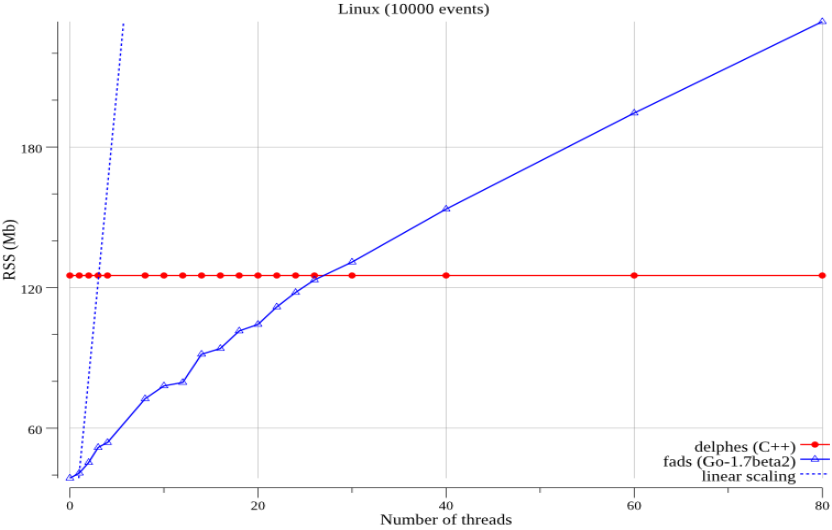
- no battle-tested persistency (JSON, ASCII, Gob, rio)
- jet clustering limited to N^3 (slowest and dumbest scheme of C++-FastJet)


hepmc-streamer
/fads/McEvent
hepmcreader
/fads/StableParticles
/fads/AllParticles
mc-jet-finder
pprop
/fads/mc-jet-finder/jets
/fads/mc-jet-finder/rho
/fads/pprop/Muons
/fads/pprop/ChargedHadrons
/fads/pprop/Electrons
muon-trk-eff
charged-hadron-trk-eff
electron-trk-eff
/fads/pprop/StableParticles
/fads/muon-trk-eff/Muons
/fads/charged-hadron-trk-eff/ChargedHadrons
/fads/electron-trk-eff/Electrons
muon-mom-smearing
charged-hadron-mom-smearing
electron-ene-smearing
/fads/muon-mom-smearing/Muons
/fads/charged-hadron-mom-smearing/ChargedHadrons
/fads/electron-ene-smearing/Electrons
muon-eff
track-merger
electron-eff
/fads/track-merger/energy
/fads/track-merger/momentum
/fads/track-merger/tracks
/fads/muon-eff/muons
/fads/electron-eff/electrons
calo
/fads/calo/eflowtowers
/fads/calo/eflowtracks
/fads/calo/photons
/fads/calo/towers
missing-et
eflow-merger
photon-eff
fastjet-finder
/fads/missing-et/momentum
/fads/missing-et
/fads/missing-et/energy
/fads/eflow-merger/eflow
/fads/eflow-merger/energy
/fads/eflow-merger/momentum
/fads/photon-eff/photons
/fads/fastjet-finder/rho
/fads/fastjet-finder/jets
muon-iso
electron-iso
photon-iso
jet-ene-scale
/fads/jet-ene-scale/jets
/fads/Partons
/fads/muon-iso/muons
btag
/fads/photon-iso/photons
/fads/btag/jets
/fads/electron-iso/electrons
tau-tag
/fads/tau-tag/jets
uobj-finder
/fads/uobj-finder/photons
/fads/uobj-finder/electrons
/fads/uobj-finder/muons
/fads/uobj-finder/jets
scalar-ht
/fads/scalar-ht/energy
/fads/scalar-ht/momentum
/fads/scalar-ht

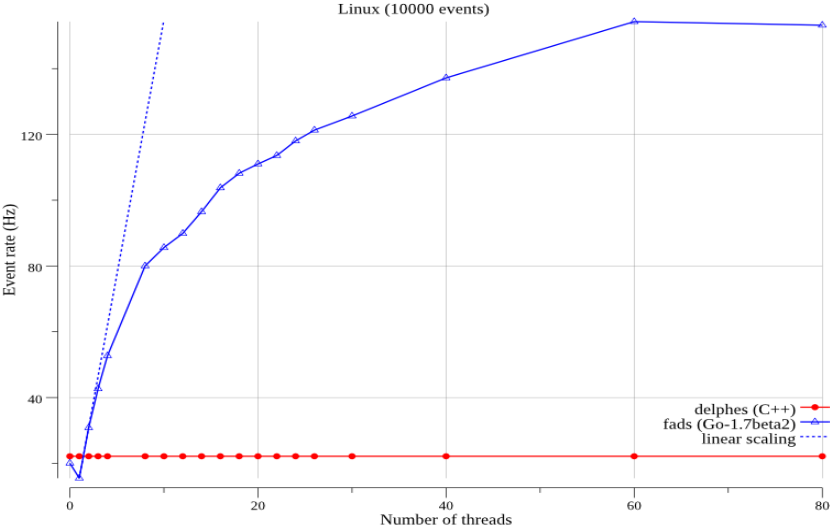
Results - testbenches

- Linux: Intel(R) Core(TM)2 Duo CPU @ 2.53GHz, 4GB RAM, 2 cores
- MacOSX-10.6: Intel(R) Xeon(R) CPU @ 2.27GHz, 172GB RAM, 16 cores
- Linux: Intel(R) Xeon(R) CPU E5-2660 v2 @ 2.20GHz, 40 cores
- Linux: Westmere E56xx/L56xx/X56xx (Nehalem-C) (3066.774 MHz), 20-cores, 40Gb RAM

Linux (20 cores) testbench: memory (smaller==better)



Linux (20 cores) testbench: event throughput (higher==better)



- good RSS scaling
- good CPU scaling

- bit-by-bit matching physics results wrt `Delphes` (up to calorimetry)
- no need to merge output files, less chaotic I/O, less I/O wait

Also addresses `C++` and `python` deficiencies:

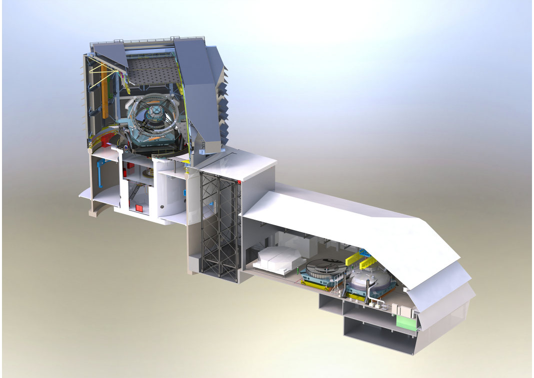
- code distribution
- code installation
- compilation/development speed
- runtime speed
- simple language

Go in Cosmology

Go in Cosmology

Relatively new activity at LPC-Clermont-Ferrand: Large Synoptic Survey Telescope (LSST)

- hardware development activities
- software (analysis, simulation, db) development activities



Go in Cosmology (and medical applications)

- developed a supernovae fusion simulation (replacing an Excel-based one)
- developed a control command application (+GUI) to steer a testbench (replacing a Java-based one)
- developed (not by me, actually) a data acquisition (+GUI) for a medical detector (replacing a `C++03-pthreads` one)

See bonus slides for more informations.

Experience so far

Even if Go is relatively new, support for general purpose scientific libraries is there and growing, thanks to the gonum community:

- gonum/blas, a go based implementation of Basic Linear Algebra Subprograms
- gonum/matrix, to work with matrices
- gonum/graph, to work with graphs
- gonum/optimize, for finding the optimum value of functions
- gonum/integrate, provides routines for numerical integration
- gonum/stat, for statistics and distributions
- ...

Plotting data is also rather easy:

- gonum/plot (most of the plots seen here were made w/ gonum/plot)
- go-gnuplot

What's missing ?

The `Go` scientific-oriented ecosystem is slowly bootstrapping itself. Other "communities" are gathering too:

- Biology: `biogo`
- Chemistry: `gochem`

So, what's missing ? (IMHO) not that much.

- a dash of performance (but we are still light years ahead of `CPython`)
- critical mass

the rest is/will-be history :)

What's missing ? - II

- A (native) GUI toolkit?

Bindings to `Qt`, `GTK`, etc... exist but they break the nice "go-get" install experience.
A (native) GUI toolkit is being built: `golang.org/x/exp/shiny`
Right now, workaround is to create a `Go` web server and serve `JavaScript+HTML`.
Nice to have to help spreading `Go`:

- a robust way to write *e.g.* `Python` extension modules in `Go` (see `go-python/gopy`)
- a go interpreter ? (see `igo`, `go-interpreter` and/or `Jupyter+Go`)

Thank you

Sébastien Binet CNRS/IN2P3/LPC

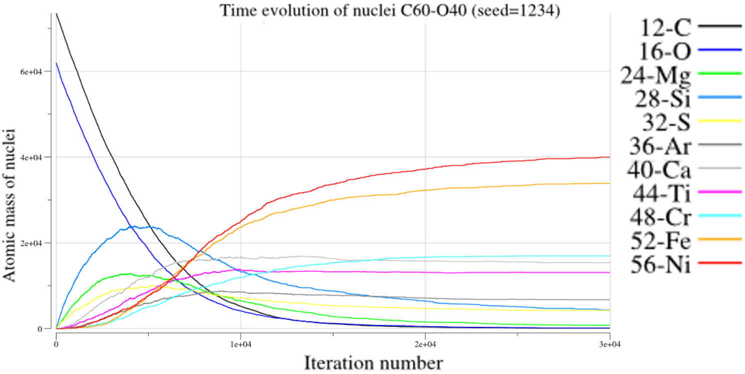
Bonus

Go @LSST: Fusil

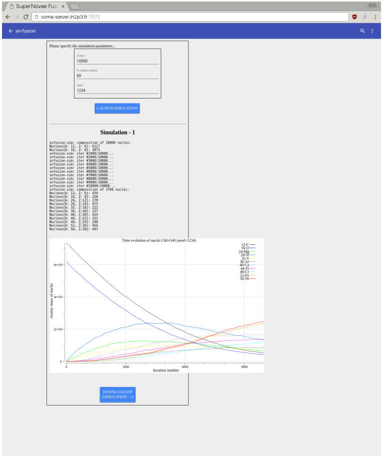
- replaced an Excel-based (!) supernovae fusion simulation
- astrogo/snfusion (aka FuSil)
- Split into 2 commands: `snfusion-gen` and `snfusion-plot`

```
$ snfusion-gen -n 30000
snfusion-gen: processing...
snfusion-gen: composition of 10000 nuclei:
Nucleus{A: 12, Z: 6}: 6127
Nucleus{A: 16, Z: 8}: 3873
snfusion-gen: iter #3000/30000...
[...]
snfusion-gen: iter #30000/30000...
snfusion-gen: composition of 3066 nuclei:
Nucleus{A: 12, Z: 6}: 71
Nucleus{A: 16, Z: 8}: 63
[...]
Nucleus{A: 56, Z:28}: 639
snfusion-gen: processing... [done]: 10.52320492s
```

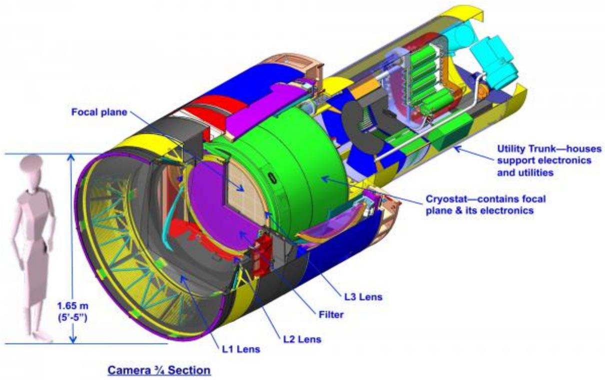
```
$ snfusion-plot -f output.csv -o output.png
snfusion-plot: plotting...
snfusion-plot: NumIters:   30000
snfusion-plot: NumCarbons: 60
snfusion-plot: Seed:       1234
snfusion-plot: Nuclei:     [Nucleus{A: 12, Z: 6} [...] Nucleus{A: 52, Z:26}
```



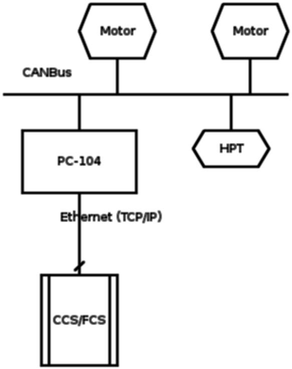
For ease of use, added a simulation web portal `snfusion-web`



Replaced a `Java` based application to control a set of motors to rotate a (dummy for now) telescope apparatus:



Replaced a `Java` based application to control a set of motors to rotate a (dummy for now) telescope apparatus:



Go @LSST: LSST testbench - III

- a web server written in Go (with `net/http`), serves as the GUI (`WebSocket` + `Polymer`)
- handles authentication, authorization
- commands relayed to the motors over `Modbus`
- displays webcam stream, stores motors' status in a database (`BoltDB`)

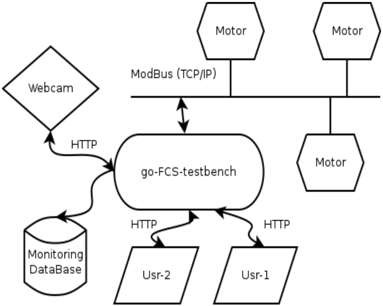
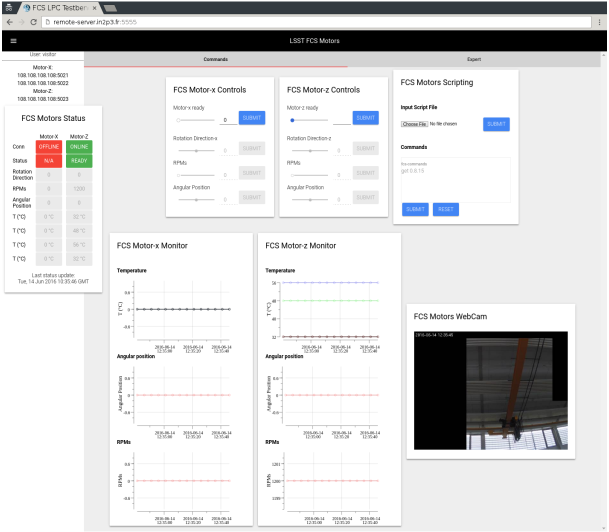
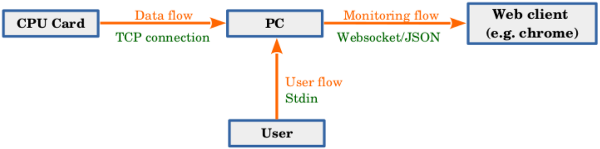


Figure: The `fcs-lpc-motor-ctl` architecture.

Go @LSST: LSST testbench - IV

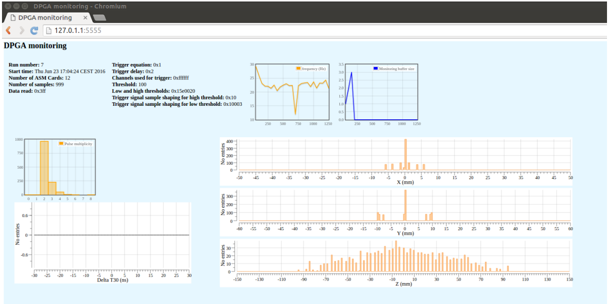


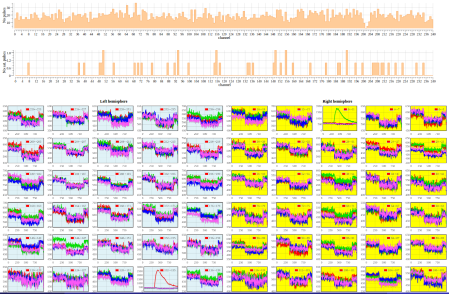
Replaced (not by me) a `C++-03/pthreads` application for data acquisition, with a much improved feature-wise `Go` version:



- receives data flow from socket (@ 20-100 Hz, limited by VME dead-time)
- checks binary data integrity (`0xCAFEDECA` control words)
- writes data to disk
- launches/stops/pauses monitoring
- listens for instructions from user

Available at `gitlab.in2p3.fr/avirm/analysis-go`


DPGA monitoring - Chromium
DPGA monitoring
127.0.1.1:5555
DPGA monitoring
Run number: 7
Start time: Thu Jun 23 17:04:24 CEST 2016
Number of ASM Cards: 12
Number of samples: 999
Data read: 0x3ff
Trigger equation: 0x1
Trigger delay: 0x2
Channels used for trigger: 0xffffff
Threshold: 100
Low and high thresholds: 0x15e0020
Trigger signal sample shaping for high threshold: 0x10
Trigger signal sample shaping for low threshold: 0x10003
Pulse multiplicity
No entries
Delta T30 (ns)
X (mm)
Y (mm)
Z (mm)


No pulses
channel
No sat. pulses
channel
Left hemisphere
Right hemisphere