fads: a FAst Detector Simulation toolkit

GdR Terascale, 2014/12/13

Sebastien Binet
CNRS/IN2P3

(a brief) History of software in HEP

50's-90's: FORTRAN77

```
c     == hello.f ==
      program main
      implicit none
      write ( *, '(a)' ) 'Hello from FORTRAN'
      stop
      end
```

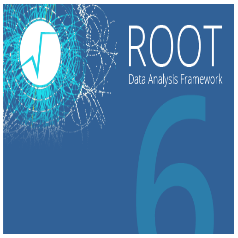
```
$ gfortran -c hello.f && gfortran -o hello hello.o
$ ./hello
Hello from FORTRAN
```

- FORTRAN77 is the **king**
- 1964: **CERNLIB**
- REAP (paper tape measurements), THRESH (geometry reconstruction)
- SUMX, **HBOOK** (statistical analysis chain)
- ZEBRA (memory management, I/O, ...)
- GEANT3, **PAW**

90's-...: C++

```
#include <iostream>
int main(int, char **) {
  std::cout << "Hello from C++" << std::endl;
  return EXIT_SUCCESS;
}
```

```
$ c++ -o hello hello.cxx && ./hello
Hello from C++
```



- object-oriented programming (OOP) is the cool kid on the block
- **ROOT**, POOL, LHC++, AIDA, **Geant4**
- C++ takes roots in HEP

00's-...: python

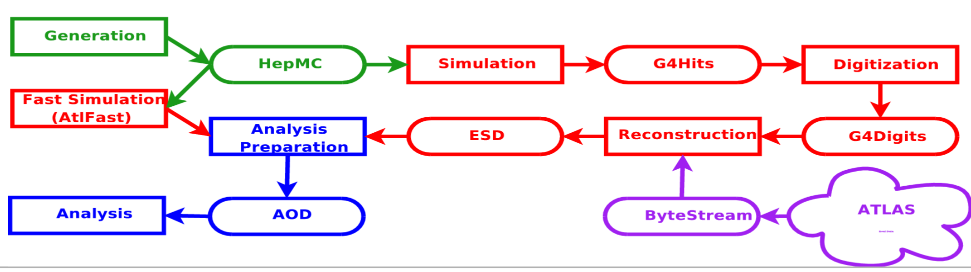
```
print "Hello from python"
```

```
$ python ./hello.py
Hello from python
```

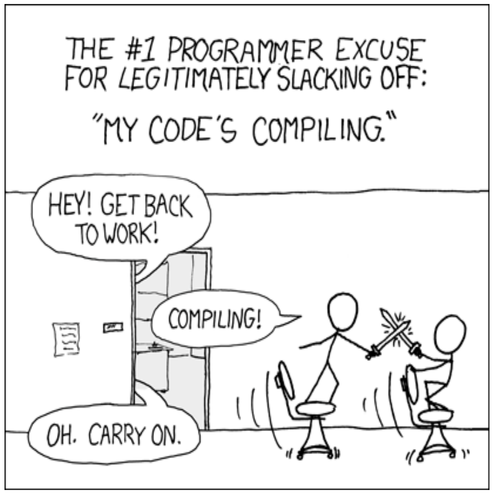
- python becomes the *de facto* scripting language in HEP
- framework data-cards
- analysis glue, (whole) analyses in python
- **PyROOT**, rootpy
- numpy, scipy, **IPython**, matplotlib

Current software in a nutshell

- **Generators**: generation of true particles from fondamental physics first principles
- **Full Simulation**: tracking of all stable particles in magnetic field through the detector simulating interaction, recording energy deposition (**CPU intensive**)
- **Reconstruction**: from real data, or from `Monte-Carlo` simulation data as above
- **Fast Simulation**: parametric simulation, faster, coarser
- **Analysis**: daily work of physicists, running on output of reconstruction to derive analysis specific information (**I/O intensive**)
- everything in the same `C++` offline control framework (except analysis)

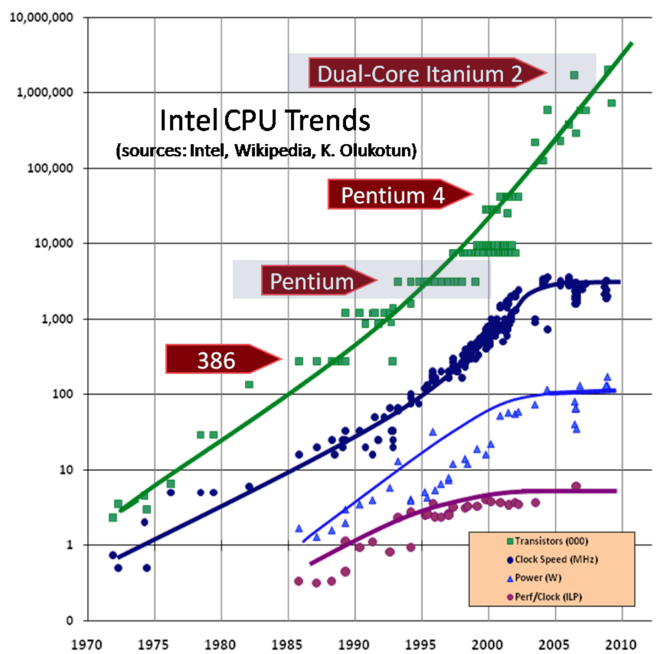


- C++: **slow** (very slow?) to compile/develop, **fast** to execute
- `python`: **fast** development cycle (no compilation), **slow** to execute



Are those our only options ?

Moore's law



Moore's law

- Moore's law still observed at the hardware level
- **However** the *effective* perceived computing power is mitigated

"Easy life" during the last 20-30 years:

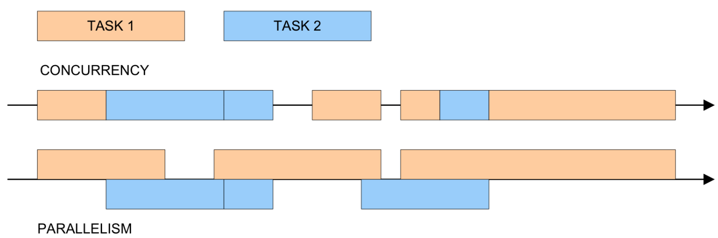
- Moore's law transleted into **doubling** compute capacity every ~18 months (*via* clock frequency)
- **Concurrency** and **parallelism** necessary to efficiently harness the compute power of our new multi-core CPU architectures.

But our current software isn't prepared for parallel/concurrent environments.

Interlude: concurrency & parallelism

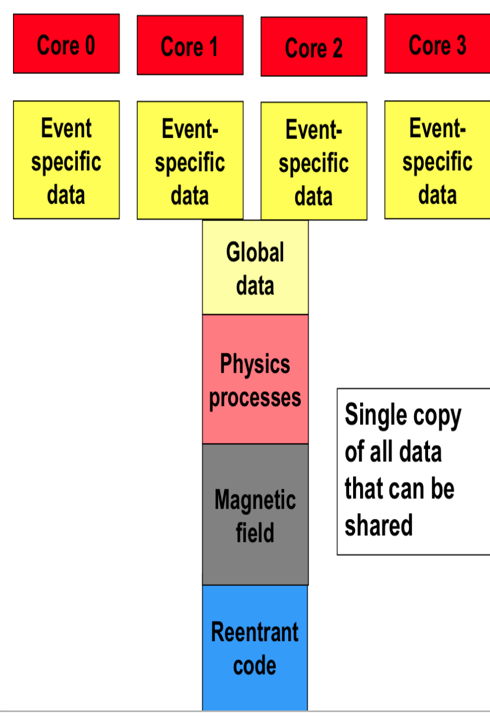
Interlude: concurrency & parallelism

- **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.

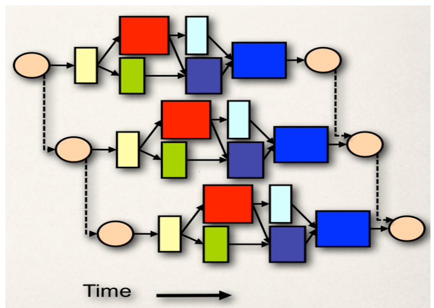
Concurrency in HEP software



Concurrency in HEP software - II

Various levels of concurrency can be exposed in current HEP applications:

- event-level concurrency
  - the framework allows to properly and safely process multiple events at a given time
- algorithm-level concurrency, task- and/or data- oriented concurrency
  - the framework allows to partition the processing of an event into various sub-tasks (calorimetry, tracking, RoIs, . . .)
  - task/functional oriented concurrency: split according to "logical" tasks
  - data oriented concurrency: partition the data domain
- subalgorithm-level concurrency
  - each algorithm can itself exposes concurrent sub-sub-tasks
  - leverage co-processors, vector units, . . .



CPU ⇒ multi-cores

- each *CPU* may hold multiple (2 →∼ 64) cores
- each core is individually slower than the "old" *CPUs*
- available memory per core decreases

↑ *number of CPU* cores ⇒ ↑ concurrency + parallelism

- analysis & reconstruction applications:
  - parallelism at event level
  - *embarassingly parallel*
- parallelism at algorithm level
  - potentially more scalable
  - more difficult too (code *redesign/rewrite*)

Amdahl's law: $R_{speedup} = \frac{1}{(1-S)+\frac{S}{N_{CPU}}}$

harness parallelism *via*:

- multi-processing (*eg:* AthenaMP, GaudiMP, CMSSW, ...)
- multi-threading (*eg:* AthenaMT, GaudiHive, Geant4-MT, CMSSW, ...)

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

- parallel programming in `C++` is doable:
  - `C/C++ locks + threads` (`pthread`, `WinThreads`)
    - great performances
    - good generality
    - rather low productivity
  - multithreaded applications
    - *hard to get right*
    - *hard to keep right*
    - *hard to keep efficient and optimized across releases*

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

C++11/14 libraries do help a bit:
- `std::lambda`, `std::thread`, `std::promise`
- (Intel) Threading Building Blocks
- ...

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

Candidates

- python/pypy
- FORTRAN-2008
- Vala
- Swift
- Rust
- Go
- Chapel
- Scala
- Haskell
- Clojure

Why not Go ?

```
package main

import "fmt"

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

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

A nice language with a nice mascot.

Go in a nutshell

Go (https://golang.org) is a new, general-purpose programming language.

- Compiled
- Statically typed
- Concurrent
- Simple
- Productive

"Go is a wise, clean, insightful, fresh thinking approach to the greatest-hits subset of the well understood."
- Michael T. Jones

History

- Project starts at Google in 2007 (by Griesemer, Pike, Thompson)
- Open source release in November 2009
- More than 250 contributors join the project
- Version 1.0 release in March 2012
- Version 1.1 release in May 2013
- Version 1.2 release in December 2013
- Version 1.3 release in June 2014
- Version 1.4 release in December 2014 (last Thursday)

Elements of Go

- Founding fathers: 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**

Elements of Go - II

- available on MacOSX, Linux, Windows,... x86, x64, ARM.
- available on *lxplus*:

```
$ ssh lxplus
[...]
* LXPLUS Public Login Service
* 2014-09-23 - expect installed
* 2014-10-02 - golang (Go Language) installed
* **********************************************************************

$ /usr/bin/go version
go version go1.2.2 linux/amd64

$ . /afs/cern.ch/sw/lcg/contrib/go/1.3/linux_amd64/setup.sh
$ go version
go version go1.3 linux/amd64
```

Concurrency

Goroutines

- The *go* statement launches a function call as a goroutine

```
go f()
go f(x, y, ...)
```

- A goroutine runs concurrently (but not necessarily in parallel)
- A goroutine has its own (growable/shrinkable) stack

A simple example

```
func f(msg string, delay time.Duration) {
    for {
        fmt.Println(msg)
        time.Sleep(delay)
    }
}
```

Function f is launched as 3 different goroutines, all running concurrently:

```
func main() {
    go f("A--", 300*time.Millisecond)
    go f("-B-", 500*time.Millisecond)
    go f("--C", 1100*time.Millisecond)
    time.Sleep(20 * time.Second)
}
```

Run

Communication via channels

A channel type specifies a channel value type (and possibly a communication direction):

```
chan int
chan<- string  // send-only channel
<-chan T       // receive-only channel
```

A channel is a variable of channel type:

```
var ch chan int
ch := make(chan int)  // declare and initialize with newly made channel
```

A channel permits *sending* and *receiving* values:

```
ch <- 1   // send value 1 on channel ch
x = <-ch  // receive a value from channel ch (and assign to x)
```

Channel operations synchronize the communicating goroutines.

Communicating goroutines

Each goroutine sends its results via channel ch:

```
func f(msg string, delay time.Duration, ch chan string) {
    for {
        ch <- msg
        time.Sleep(delay)
    }
}
```

The main goroutine receives (and prints) all results from the same channel:

```
func main() {
    ch := make(chan string)
    go f("A--", 300*time.Millisecond, ch)
    go f("-B-", 500*time.Millisecond, ch)
    go f("--C", 1100*time.Millisecond, ch)

    for i := 0; i < 100; i++ {
        fmt.Println(i, <-ch)
    }
}
```

Run

fads

fads

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

- morally a translation of C++-Delphes (https://cp3.irmp.ucl.ac.be/projects/delphes) into Go
- uses go-hep/fwk (https://github.com/go-hep/fwk) to expose, manage and harness concurrency into the usual HEP event loop (`initialize` | `process-events` | `finalize`)
- uses go-hep/hbook (https://github.com/go-hep/hbook) for histogramming, go-hep/hepmc (htpps://github.com/go-hep/hepmc) for HepMC input/output

Code is on github (BSD-3):

github.com/go-hep/fwk (https://github.com/go-hep/fwk)

github.com/go-hep/fads (https://github.com/go-hep/fads)

Documentation is served by godoc.org (https://godoc.org):

godoc.org/github.com/go-hep/fwk (https://godoc.org/github.com/go-hep/fwk)

go-hep/fads - Installation

As easy as:

```
$ export GOPATH=$HOME/dev/gocode
$ export PATH=$GOPATH/bin:$PATH

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

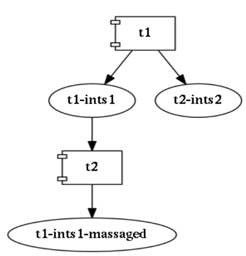
Yes, with the ellipsis at the end, to also install sub-packages.

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

go-hep/fwk - Examples

```
$ 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
app                  INFO mem: n-frees:           60
app                  INFO mem: gc-pauses:          0 ms
::: fwk-ex-tuto-1... [done] (cpu=788.578us)
```

go-hep/fwk - Concurrency

fwk (https://github.com/go-hep/fwk) enables:

- event-level concurrency
- tasks-level concurrency

fwk (https://github.com/go-hep/fwk) relies on Go (https://golang.org) 's runtime to properly schedule *goroutines*.

For sub-task concurrency, users are by construction required to use Go (https://golang.org) 's constructs (*goroutines* and *channels*) so everything is consistent **and** the *runtime* has the **complete picture**.

- **Note:** Go (https://golang.org) 's runtime isn't yet *NUMA-aware*. A proposal for *Go-1.5 (June-2015)* is in the works (https://docs.google.com/document/d/1d3iI2QWURgDIsSR6G2275vMeQ_X7w-qxM2Vp7iGwwuM/pub) .

go-hep/fads - real world use case

- translated C++-Delphes (https://cp3.irmp.ucl.ac.be/projects/delphes)' ATLAS data-card into Go
- go-hep/fads-app (https://github.com/go-hep/fads/blob/master/cmd/fads-app/main.go)
- installation:

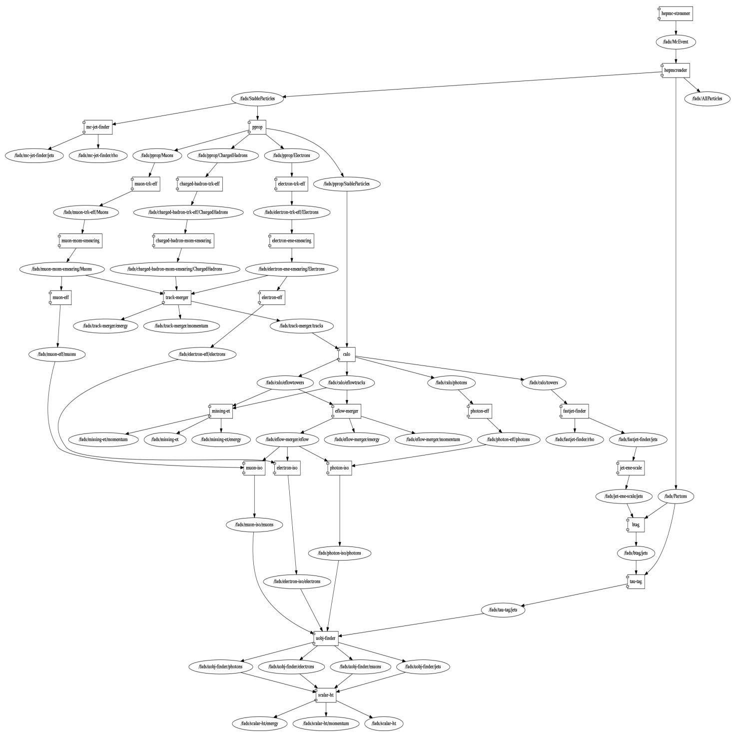
```
$ go get github.com/go-hep/fads/cmd/fads-app
$ fads-app -help
Usage: fads-app [options] <hepmc-input-file>

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
```

go-hep/fads - components

- 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 (https://github.com/go-hep/fastjet))
- histogram service (from go-hep/fwk (https://github.com/go-hep/fwk))

Caveats:

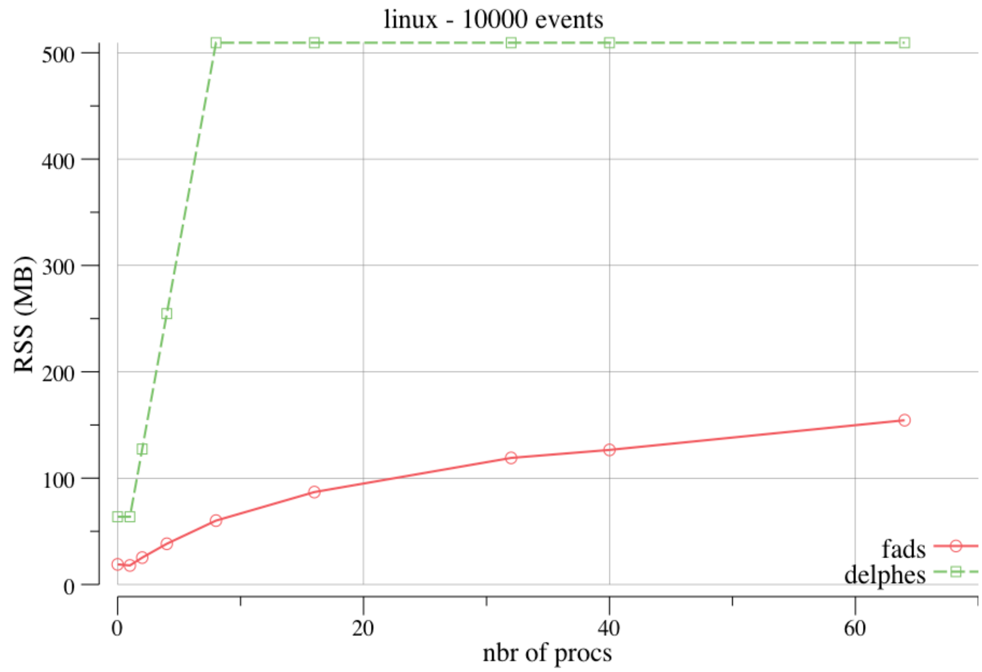
- no real persistency to speak of (*i.e.:* `JSON`, `ASCII` and `Gob`)
- jet clustering limited to N^3 (slowest and dumbest scheme of `C++-FastJet`)



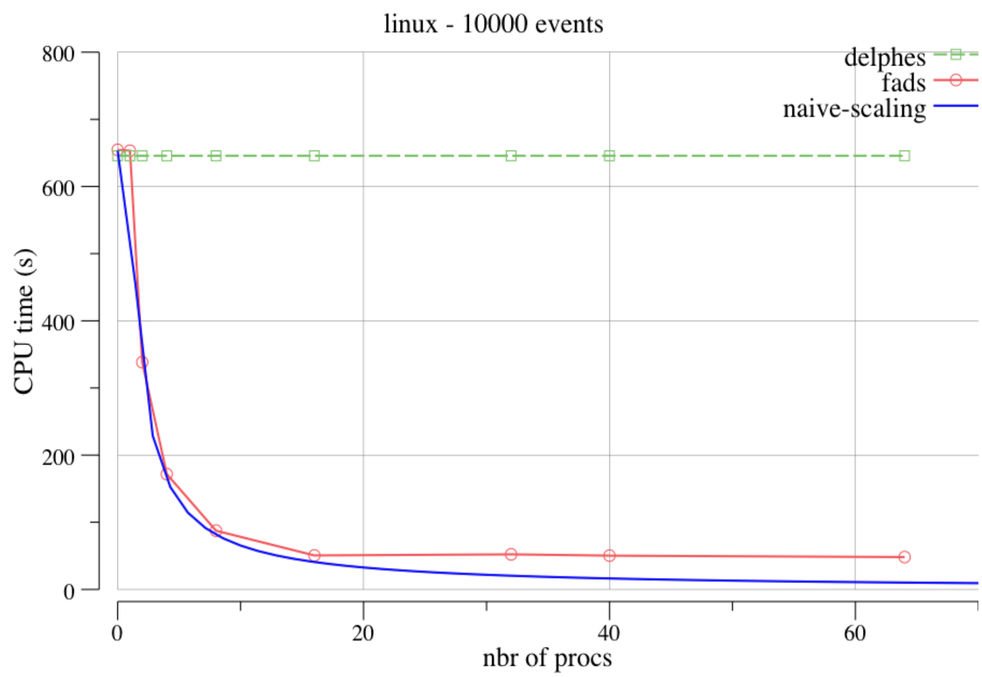
Results - testbenches

- 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

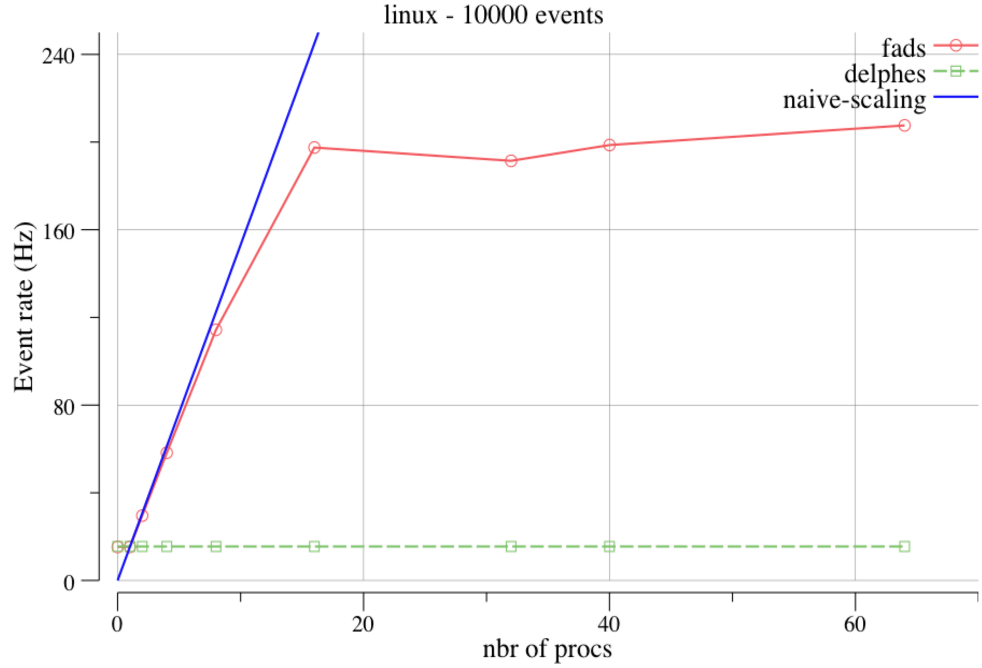
Linux (40 cores) testbench: memory



Linux (40 cores) testbench: CPU



Linux (40 cores) testbench: event throughput



Results & Conclusions

- good RSS scaling
- good CPU scaling
- bit-by-bit matching physics results wrt `Delphes` (up to calorimetry)

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

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

Prospects

- proper persistency package (in the works: go-hep/rio (https://github.com/go-hep/rio))
- histograms (persistency) + n-tuples (interactivity): go-hep/hbook (https://github.com/go-hep/hbook)
- performance improvements (cpu-profiling via `go tool pprof`)
- implement more of `go-fastjet` combination schemes and strategies
- more end-user oriented documentation

Join the fun: go-hep forum (https://groups.google.com/d/forum/go-hep)

Acknowledgements / resources

talks.golang.org/2012/tutorial.slide (http://talks.golang.org/2012/tutorial.slide)

talks.golang.org/2014/taste.slide (http://talks.golang.org/2014/taste.slide)

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

That's all !

Backup

go-hep/fwk - configuration & steering

- use regular Go (https://golang.org) to configure and steer.
- still on the fence on a DSL-based configuration language (`YAML`, `HCL`, `Toml`, ...)
- probably **not** `Python` though

```
// job is the scripting interface to 'fwk'
import "github.com/go-hep/fwk/job"

func main() {
    // create a default fwk application, with some properties
    app := job.New(job.P{
        "EvtMax":   10,
        "NProcs":   2,
    })

    // ... cont'd on next page...
```

go-hep/fwk - configuration & steering

```
// create a task that reads integers from some location
// and publish the square of these integers under some other location
app.Create(job.C{
    Type: "github.com/go-hep/fwk/testdata.task2",
    Name: "t2",
    Props: job.P{
        "Input":  "t1-ints1",
        "Output": "t1-ints1-massaged",
    },
})
// create a task that publish integers to some location(s)
// create after the consummer task to exercize the automatic data-flow scheduling.
app.Create(job.C{
    Type: "github.com/go-hep/fwk/testdata.task1",
    Name: "t1",
    Props: job.P{
        "Ints1": "t1-ints1",
        "Ints2": "t2-ints2",
        "Int1":  int64(10), // initial value for the Ints1
        "Int2":  int64(20), // initial value for the Ints2
    },
})
app.Run()
```

Thank you

Sebastien Binet
CNRS/IN2P3