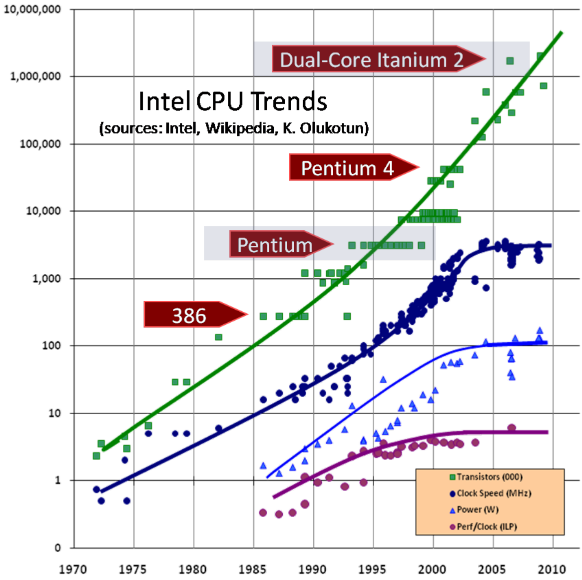
Software in HEP:
Parallelism strikes back

Sébastien Binet

LAL/IN2P3

2014-11-20

Parallelism: why?



A bit of history: 1950-2000

- 1954: computers were valve-based
- 1956: first magnetic disk system sold (IBM RAMAC)
- ∼ 1956: `FORTRAN` under development
- 1959: IBM-1401 shipped. Transistorised. Punched card input.
- 1960: PDP-1 launched (18-bit words)
- 1964: PDP-8 launched (12-bit words)
- 1964: System/360 launched (4*8-bit byte words, 8-64-256 *kB* of RAM)

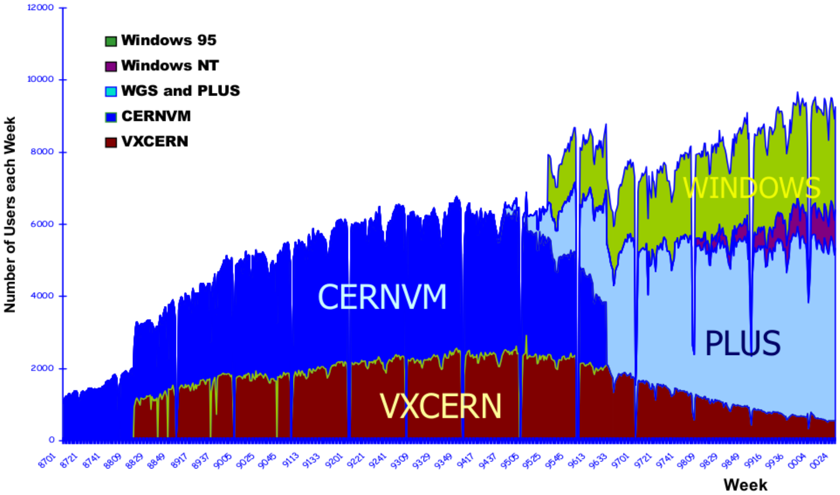
1950	1960	1970	1980	1990	2000
					
- Ferranti Mecury (1958) - Vacuum tube	- IBM mainframe (1960) - 32 kilobyte - record photographic film.	- CDC7600 & IBM mainframe - To serve the interests of different user groups.	- Cray vector supercomputer (1988) - Processing power of about 50 'CERN' units.	- RISC technology (RISC : Reduced Instruction Set Chip)	- Farm of 3000 PCs (2005) - Total of 30000 CPUs (2008)

A bit of history: 1950-2000 (at CERN)

- 1963: IBM-7090 ($\times$4 CERN total computing capacity at the time)
- 1965: CDC-6600 (1 *MFLOPs*, $\times$15 CERN's capacity)
- 1972-1984: CDC-7600, IBM-370/168
- 1982: VAX 750s,780s,8600s
- 1988-1993: Cray
- 1996: *mainframe rundown* completed. (mainframes replaced by UNIX and PC servers.)

1950	1960	1970	1980	1990	2000
					
- Ferranti Mecury (1958) - Vacuum tube	- IBM mainframe (1960) - 32 kilobyte - record photographic film.	- CDC7600 & IBM mainframe - To serve the interests of different user groups.	- Cray vector supercomputer (1988) - Processing power of about 50 'CERN' units.	- RISC technology (RISC : Reduced Instruction Set Chip)	- Farm of 3000 PCs (2005) - Total of 30000 CPUs (2008)

Weekly interactive users 1987-2000



A bit of history: 1950-2000 in (offline) software

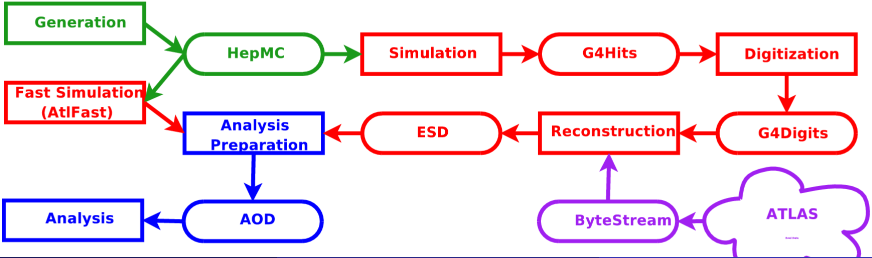
- 60's-00's: `FORTRAN` is king
- 1964: `CERN` Programme Library
- `REAP` (paper tape measurements), `THRESH` (geometry reconstruction), `GRIND` (kinematic analysis), `SUMX`, `HBOOK` (statistical analysis chain),
- `PATCHY` (source code management),
- `ZEBRA` (memory management, I/O, ...),
- `GEANT3`, `PAW`

- mid-90's-...: `C++` takes roots in HEP
- Object Oriented programming is the cool kid on the block
- `Geant4`, `ROOT`, `POOL`, `LHC++`, `AIDA`

- 00's-...: `Python` becomes the *de facto* scripting language in HEP
- framework data cards
- analysis glue, analyses in `python`
  - `PyROOT`, `rootpy`,
  - `numpy`, `scipy`, `IPython`, `matplotlib`

Current software in a nutshell (*e.g.:* ATLAS)

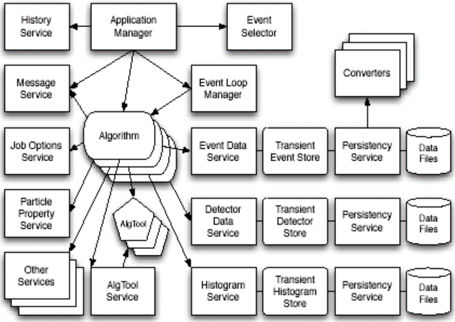
- Generators: generation of true particles from fondamental physics first principles,
  - not easy, but no software challenge either
- Full Simulation: tracking of all stable particles in magnetic field through the detector simulating interaction, recording energy deposition (CPU intensive),
- Reconstruction: from real data as it comes out of the detector, 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)

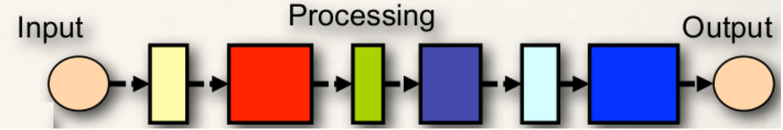


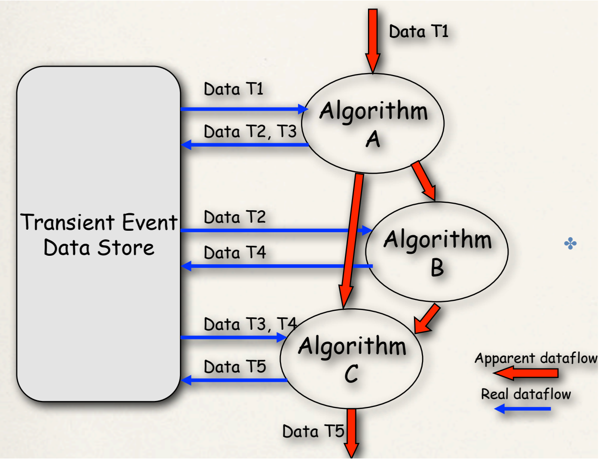
Offline frameworks (ATLAS, LHCB)

Gaudi is:

- a component object model (COM) based framework
- mainly written in C++
- with bits and pieces written in python for steering
- (although more and more pieces in python for analysis)
- most of the code written under a *single thread* assumption
  - most of the code is not thread safe




Input
Processing
Output


Data T1
Data T1
Algorithm A
Data T2, T3
Transient Event Data Store
Data T2
Algorithm B
Data T4
Data T3, T4
Algorithm C
Data T5
Apparent dataflow
Real dataflow
Data T5

Offline framework architecture: components & black-board

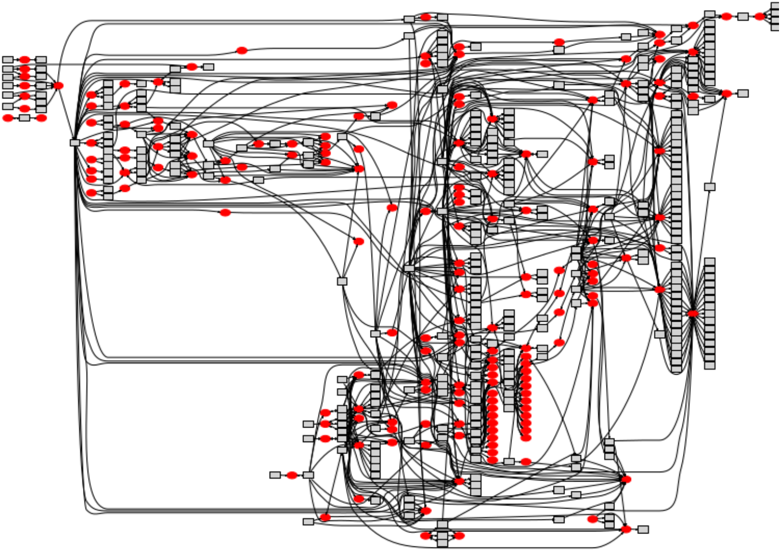


Figure: Directed acyclic graph of algorithms from a reconstruction job

Software in HEP: some numbers

- Reconstruction frameworks grew from ~3M SLOC to ~ 5M
- Summing over all HEP software stack for e.g. ATLAS:
  - event generators: ~ 1.4*M* SLOC (C++, FORTRAN-77)
  - I/O libraries ~ 1.7*M* SLOC (C++)
  - simulation libraries ~ 1.2*M* SLOC (C++)
  - reconstruction framework ~ 5*M* SLOC
  - reconstruction steering/configuration (~ 1*M* SLOC python)
- GCC: 7*M* SLOC
- Linux kernel 3.6: 15.9*M* SLOC

Software development cycle

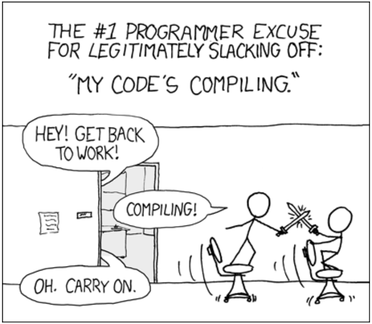
- VCS (CVS, then SVN. GIT: not yet, at least not for `LHC` experiments)
- Nightlies (Jenkins or homegrown solution)
  - need a sizeable cluster of build machines (`distcc`, `ccache`, ...)
  - builds the framework stack in ∼8h
  - produces ∼ 2000 shared libraries
  - installs them on `AFS` (also creates `RPM`s 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, . . .)

Big science, big data, big software, big numbers

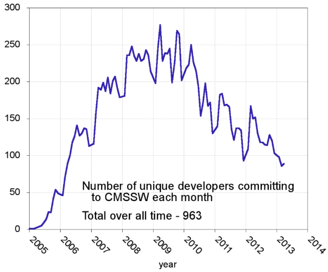
- $\sim$ 1min to initialize the application
- loading >500 shared libraries
- connecting to databases (detector description, geometry, ...)
- instantiating $\sim$2000 `C++` components
- $2Gb/4Gb$ memory footprint per process



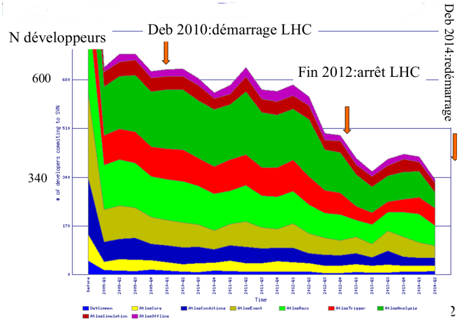
(obligatory `xkcd` reference)

Software in HEP: sustainable development ?

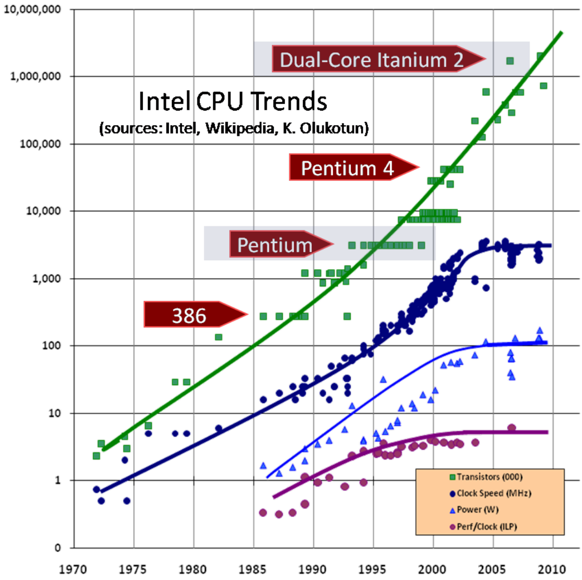
- People committing code to VCS per month
  - Wide variety of skill level
  - Large amount of churn
  - Once the physics data is pouring, people go and do physics instead of software



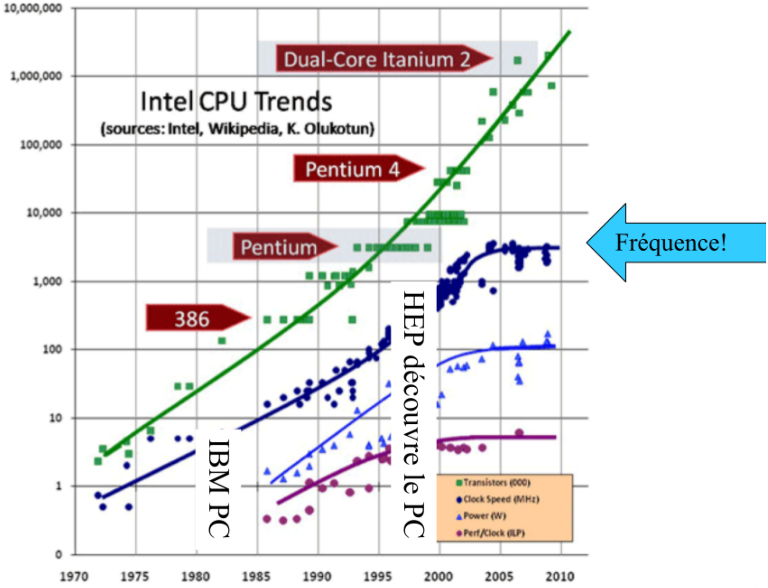
See also "The Life Cycle of HEP Offline Software", P.Elmer, L. Sexton-Kennedy, C.Jones, CHEP 2007



Moore's law



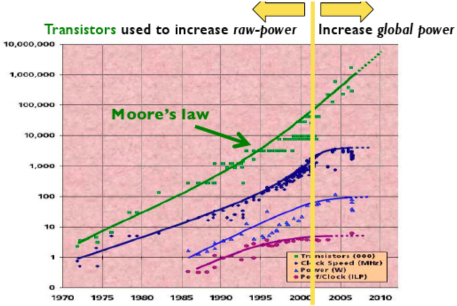
Intel CPU Trends
(sources: Intel, Wikipedia, K. Olukotun)
Dual-Core Itanium 2
Pentium 4
Pentium
386
10,000,000
1,000,000
100,000
10,000
1,000
100
10
1
0
1970
1975
1980
1985
1990
1995
2000
2005
2010
Transistors (000)
Clock Speed (MHz)
Power (W)
Perf/Clock (ILP)


Intel CPU Trends
(sources: Intel, Wikipedia, K. Olukotun)
Dual-Core Itanium 2
Pentium 4
Pentium
386
IBM PC
HEP découvre le PC
Fréquence!
Transistors (000)
Clock Speed (MHz)
Power (W)
Perf/Clock (ILP)
10,000,000
1,000,000
100,000
10,000
1,000
100
10
1
0
1970
1975
1980
1985
1990
1995
2000
2005
2010

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

Confronted with 3 walls:

- *power wall*
- *memory wall*
- *instruction level parallelism (ILP) wall*



Power wall

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

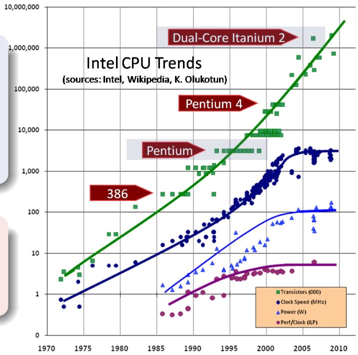
- Moore's law translated into doubling compute capacity every $\simeq$ 18 months (clock frequency)

But issues with power dissipation

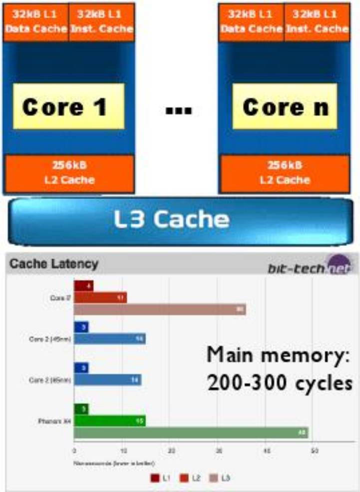
Moore's law still observed at the hardware level:

- $\uparrow$ transitors $\Rightarrow$ $\uparrow$ number of cores
- keep clock frequency constant to limit energy consumption

Concurrency & Parallelism are necessary to efficiently harness the compute power of our new multi-cores CPUs architectures.

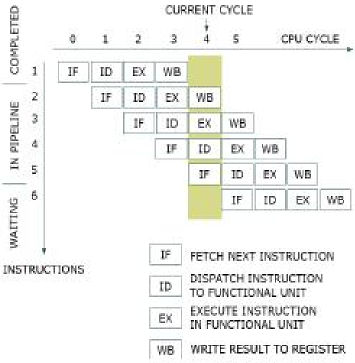


- clock frequency increases faster than memory
- bigger and faster caches somewhat mitigated impact (for now)
- memory access latency: bottleneck
- introduce multi-level (hierarchical) memory caches
  - for Intel Ivy Bridge (@3.4 *GHz*)
  - *L1*: ~ 4*cycles*
  - *L2*: ~ 12*cycles*
  - *L3*: ~ 30*cycles*
  - *RAM*: ~ 30*cycles*$+ \sim 50$*ns*



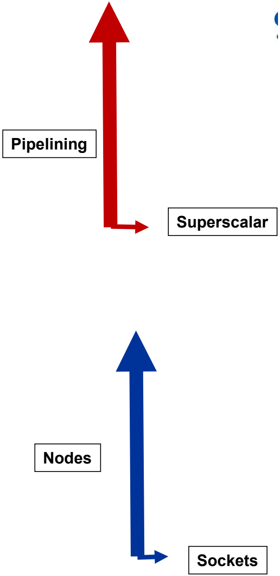
Architecture wall

- pipelines deep and large
- various techniques to improve *Instruction Level Parallelism* (`ILP`):
  - hardware branch prediction,
  - hardware speculative execution,
  - instruction re-ordering,
  - *Just-In-Time* (JIT) compilation,
  - hardware threading, . . .
- in practice: inter-dependence issues between instructions limit application of `ILP`



3 walls: *the free lunch is over*

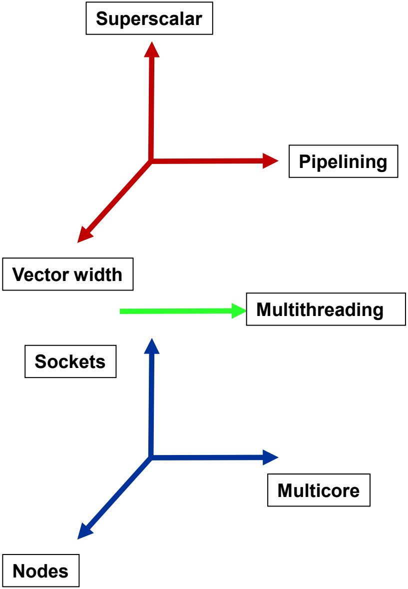
Pentium days



- 2 dimensions:
  - *pipeline* frequency
  - number of nodes
- semiconductor vendors were increasing frequency
- users were buying adequate number of nodes

Now: 7 dimensions

- first 3 dimensions:
  - vector units/`SIMD` (Single Instruction, Multiple Data)
  - *pipeline*
  - *superscalar*
- “pseudo” dimension:
  - hardware *multithreading*
- last 3 dimensions:
  - multi-cores
  - multi-sockets
  - multi-nodes



Now: 7 *multiplicative* dimensions

- 3 first dimensions:
  - vector units/`SIMD`
  - *pipeline*
  - *superscalar*
- "pseudo" dimension:
  - *multithreading* hardware
- 3 last dimensions:
  - multi-cores
  - multi-sockets
  - multi-nodes

Data-parallel

vectors/matrices

Task-parallel

- events
- tracks

Tasks/process-parallel

Where are we now ?

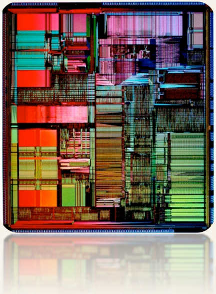
	SIMD	ILP	HW THREADS	CORES	SOCKETS
MAX	4	4	1.35	8	4
TYPICAL	2.5	1.43	1.25	8	2
HEP	1	0.80	1	6	2

	SIMD	ILP	HW THREADS	CORES	SOCKETS
MAX	4	16	21.6	172.8	691.2
TYPICAL	2.5	3.57	4.46	35.71	71.43
HEP	1	0.80	0.80	4.80	9.60

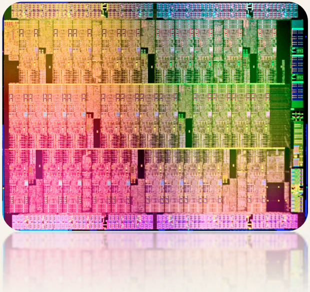
A. Nowak (CERN/OpenLab)

Where do we go ? (dunno!)

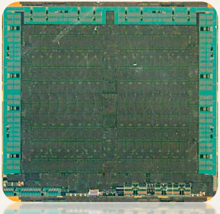
Pentium 4 (2000)



Xeon Phi (2013)



AMD GPU Radeon HD 7900 (2013)



2020

Impact on HEP *software*

CPU $\Rightarrow$ multi-cores

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

$\uparrow$ *number of CPU* cores $\Rightarrow \uparrow$ 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: naive implementation

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

Problem(s)

- resources requirements increase with the number of processes
- ↑ memory footprint
- other OS (limited) resources: *file descriptors*, *network sockets*, . . .
- share resources (+optimisation) - *eg* on DAQ clusters
  - manage number of running applications
  - nbr of network connections towards *readout* system
  - transfer exact same configuration data *n* times to same node
  - recompute *n* times exact same configuration data
  - *CPU* optimisation: interleave *CPU* for event data handling and *I/O-wait*

Multiprocessing and memory sharing: `fork+COW`

Principles

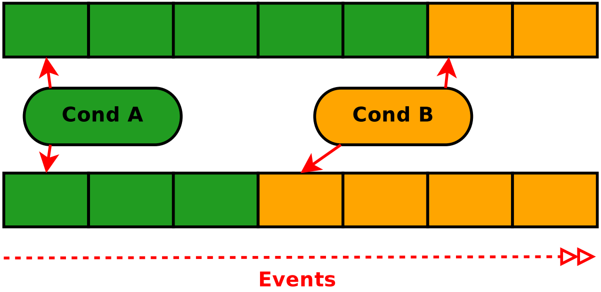
- launch many similar jobs, sharing as much memory as possible
- minimize code modifications
  - let the OS perform most of the work for us
- use the `fork()` system call

`fork()`

- `fork()` clones a process, including its entire address space
- `fork()`, on modern OSes, is implemented via Copy On Write (COW)
  - all the memory pages are shared until a process writes on them
  - these memory pages are then copied and become un-shared
- $\Rightarrow$ `fork()` as late as possible but before disk I/O
- optimal and AUTOMATED sharing of the memory between sub-processes
  - coupled to an OS + kernel version ...
  - isolation between sub-processes
  - *Chromium and Firefox* use this technique (`Zygote process`)

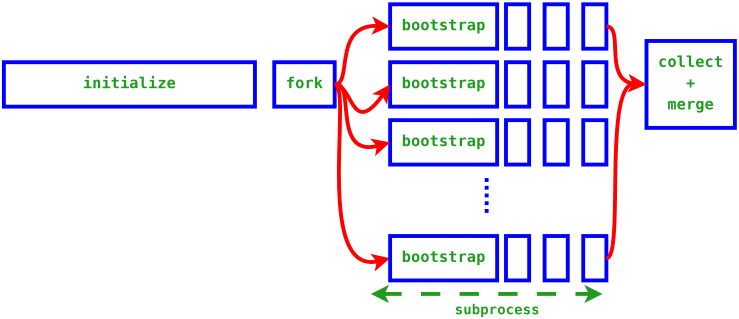
- pros:
  - ALL the memory that can be share will be shared
  - modifications restricted to a few *core framework packages*
  - no need for locks/mutexes

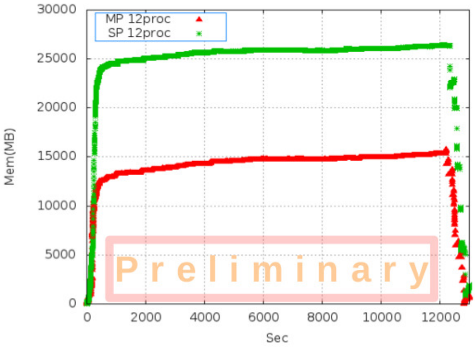
- cons
  - once *un-shared*, memory can not be re-shared
    - issue for *conditions data*

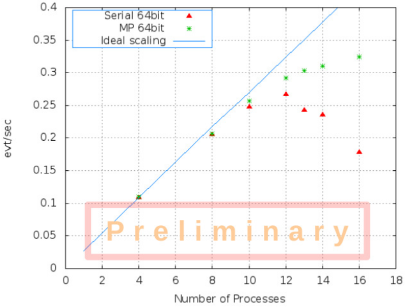


Example: AthenaMP (ATLAS reconstruction)

- minimize impact on client/physicist code
- use a python module multiprocessing for process' management
  - now re-written in C++ (AthenaMP-2)
- encapsulate modifications related to parallelism inside a new event loop scheduler.
- modifications of I/O-related components







- SP: *n* `Athena` in parallel ($\Rightarrow \sim$ 2Gb per proc.)
- MP: `AthenaMP` ($\Rightarrow \sim$ 1.2Gb per process)
  - allow to do more physics with the same h/w

Experiences with multi-processes

- limited long range impact
- modifications applied to control framework and I/O-related components
- easier to develop with
  - no implicit sharing
  - no lock, races, . . .

Problems

- random numbers, seeds and reproducibility
- I/O
  - need to chase people directly `open()`ing files, by-passing framework hooks
  - merging output files is tedious (but needed for production)
- GRID
  - submission of MP-jobs (overbooking computing nodes)
  - `vmem` accounting
    - most of grid resource monitoring tools will double-count the memory shared by `fork()`ed subprocesses

(first) Conclusions

It is possible to refactor an already existing `FORTRAN/C/C++` application, written in a *single-threaded* fashion (like `Gaudi`) with minimal modifications (or at least localised) to better leverage the *new* multicore architectures.

Automa(g)ic *scaling* with the number of cores ?

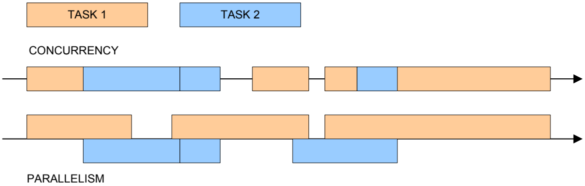
- unlikely if $N_{\texttt{cores}} \geq 1024$ (memory resources)
- unlikely at the `I/O` level
- *mapping* 1 processus per core not fine-grained enough

- concurrency at the event level
- concurrency at the algorithm level
- concurrency within the algorithms

$\Rightarrow$ multithreading !

Interlude: Parallelism & Concurrency

- 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 provides a way to structure a solution to solve a problem that may (but not necessarily) be parallelizable.



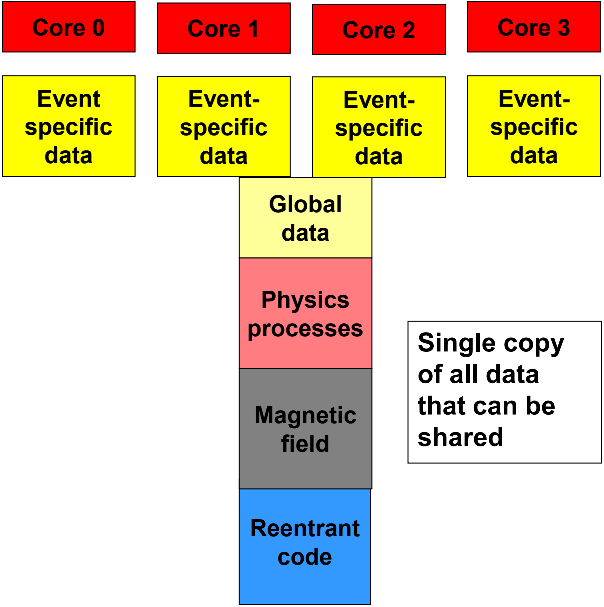
Concurrency plus communication

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.

New developments and/or adiabatic evolution of `Gaudi` need to:

- prepare for further gains by exploiting features of today's CPUs' μ-architecture
  - vector registers, instruction pipelining, multiple instructions per cycle (see Sverre Jarp presentations at CHEP)
  - improve data and code locality, hardware threading
  - (also relevant for non-fwk code)
- prepare for, or at least don't prevent use of, off-loading large computations to accelerators (GPGPUS, Xeon Phi)
- prepare for increased exposed concurrency
  - a means for better memory usage and improved throughput

Concurrency in HEP frameworks - II



Single copy of all data that can be shared

Concurrency in HEP frameworks - III

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, . . .

Event-level concurrency is achieved by:

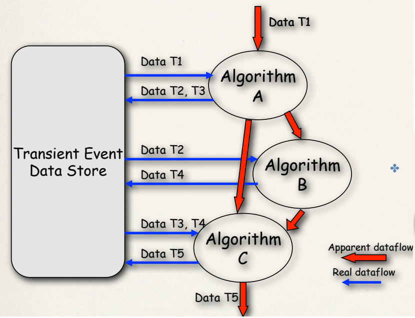
- modifying the event loop to hand over multiple events
- put these multiple events into multiple event stores
- have algorithms and sequence of algorithms work on these stores

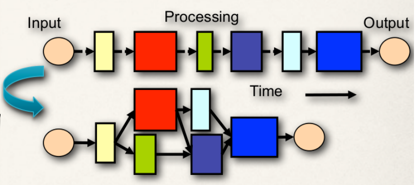
REQUIRES that at least the core components are race-free and thread-safe

Concurrency in HEP frameworks - V

Alg-level concurrency is achieved by:

- modifying the algorithm manager to execute multiple algorithms concurrently
- need new information to properly schedule these algorithms in the correct order: data dependency graph (hopefully acyclic!)
  - either extracted at runtime during a warm-up phase or explicitly at configuration-time





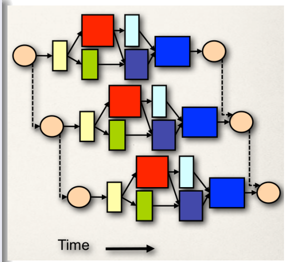
Concurrency in HEP frameworks - VI

SubAlg-level concurrency is achieved by:

- providing tools or libraries to expose concurrency
- making the framework aware of the available resources
- making the framework aware of the different tools/libraries for an efficient scheduling

Many concurrent events

- Need to deal with the tails of sequential processing
  - there is always an `Algorithm` taking very long producing data needed by many others
- introducing `pipeling` processing
  - exclusive access to resources
  - non-reentrant algorithms
  - *e.g.* file writing, DB access, ...
- Current frameworks handle a single event at the time. They need to be evolved
  - design a powerful and flexible `Algorithm` scheduler
  - need to define the concept of an `event context`



- 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*

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

In a C++ world…

Summing vector elements in C using OpenMP – openmp.org

```
#pragma omp parallel for reduction(+: s)
for (int i = 0; i < n; i++) {
  s += x[i];
}
```

Per element multiply in C++ using Intel® Array Building Blocks – intel.com/go/arbb

```
void products( const arbb::dense<arbb::f32>& a,
               const arbb::dense<arbb::f32>& b,
               arbb::dense<arbb::f32>& c) {
  c = a * b;
}
```

Dot product in Fortran using OpenMP – openmp.org

```
!$omp parallel do reduction ( + : adotb )
  do j = 1, n
    adotb = adotb + a(j) * b(j)
  end do
!$omp end parallel do
```

Sum in Fortran, using co-array feature – intel.com/software/products

```
REAL SUM[*]
CALL SYNC_ALL( WAIT=1 )
DO IMG= 2,NUM_IMAGES()
  IF (IMG==THIS_IMAGE()) THEN
    SUM = SUM + SUM[IMG-1]
  ENDIF
  CALL SYNC_ALL( WAIT=IMG )
ENDDO
```

Parallel function invocation in C using Intel® Cilk™ Plus – cilk.org

```
cilk_for (int i=0; i<n; ++i) {
  Foo(a[i]);
}
```

Parallel function invocation in C++ using Intel® Threading Building Blocks – threadingbuildingblocks.org

```
parallel_for (0, n,
  [=](int i) { Foo(a[i]); }
);
```

In a C++ world. . .

MPI code in C for clusters – intel.com/go/mpi

```
for (d=1; d<ntasks; d++) {
  rows = (d <= extra) ? avrow+1 : avrow;
  printf(" sending %d rows to task %d\n", rows, dest);
  MPI_Send(&offset, 1, MPI_INT, d, mtype, MPI_COMM_WORLD);
  MPI_Send(&rows, 1, MPI_INT, d, mtype, MPI_COMM_WORLD);
  MPI_Send(&a[offset][0], rows*NCA, MPI_DOUBLE, d, mtype, MPI_COMM_WORLD);
  MPI_Send(&b, NCA*NCB, MPI_DOUBLE, d, mtype, MPI_COMM_WORLD);
  offset = offset + rows;
}
```

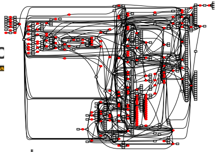
Per element multiply in C using OpenCL – intel.com/go/opencl

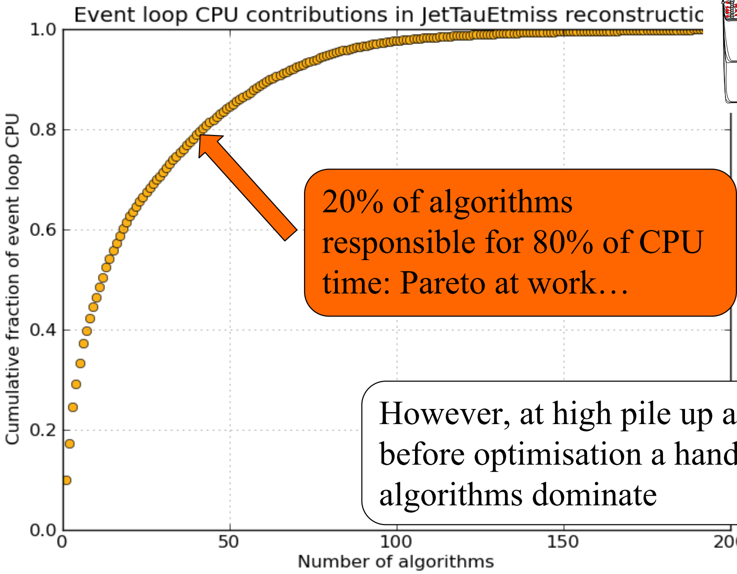
```
kernel void
  dotprod( global const float *a,
           global const float *b,
           global float *c) {
    int myid = get_global_id(0);
    c[myid] = a[myid] * b[myid];
}
```

Matrix Multiply in Fortran using Intel® Math Kernel Library – intel.com/software/products

```
    call DGEMM(transa,transb,m,n,k,alpha,a,lda,b,ldb,beta,c,ldc)
```

Jury is still out on which tool is the solution. . .

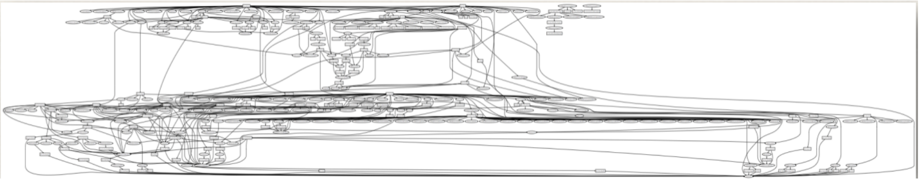




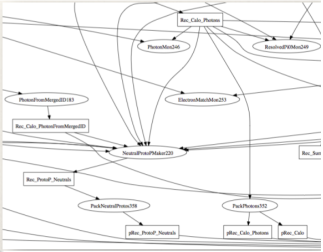
20% of algorithms responsible for 80% of CPU time: Pareto at work…

However, at high pile up and before optimisation a handful of algorithms dominate

Example: LHCb reconstruction



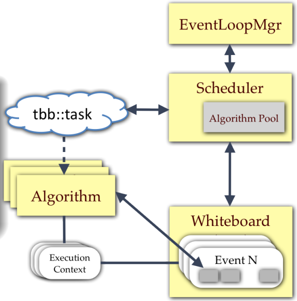
- DAG of Brunel (214 Algorithms)
  - obtained by instrumenting the existing sequential code
  - probably still missing 'hidden or indirect' dependencies
- this can give us an estimate of the potential for *concurrency*
  - assuming no changes in current reconstruction algorithms



Implementation and PoC

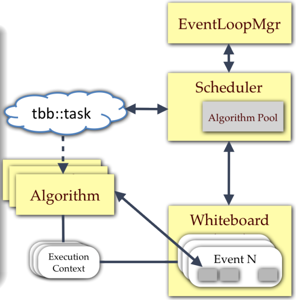
Testbed for these developments: GaudiHive

- a LCG/LHCb/ATLAS project
- toy framework evolving into a real one
  - No real algorithms but CPU crunchers
  - timing of real workflow reproduced
- Schedule algorithm when its inputs are available



Implementation and PoC - II

- Multiple events managed simultaneously
  - bigger probability to schedule an alg
  - whiteboard integrated in the `DataSvc`
  - `DataSvc` made thread-safe
- Several copies of the same algorithms can coexist
  - running on different events
  - responsibility of `AlgoPool`
- Data specific to execution stored in the *execution context*

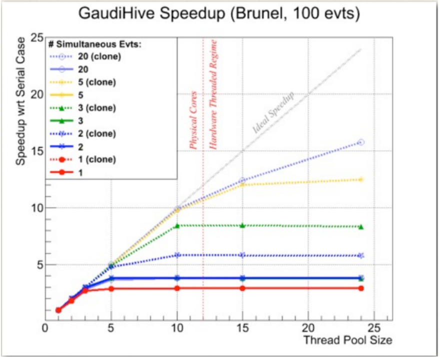


Implementation notes

- task-oriented concurrency handled via TBB
  - **de facto** standard among experiments (ATLAS, CMS and LHCb)
  - useful concurrent containers and algorithms (`parallel_for`,...)
- leverage `C++11` constructs and memory model
  - atomics
  - confortable syntax (range-based `for` loops, `auto`, `tuples`)
- new algorithm steering to handle dependency graph
  - opportunity to streamline/harmonize with `trigger steering` solution
- thread-safe message service (`TBBMessageService`)
- work on a thread-safe `ToolSvc` and `ServiceMgr`
- auditors ? incidents ?

Test on `Brunel` workflow

- 214 algorithms, real data dependencies, (average) real timing
  - maximum speedup depends stringly on the workflow chosen
- adding more simultaneous events moves the maximum concurrency from 3 to 4 with single `Algorithm` instances
- increased parallelism when cloning algorithms
  - even with a moderate number of events in flight



Test system with 12 physical cores x2 hyperthreads (HT)

Concurrent Gaudi: status

- a prototype of a concurrent Gaudi (GaudiHive) has been developed as an evolution (new branch in the Gaudi repository)
  - able to schedule and run algorithms concurrently
  - able to run multiple events simultaneously
  - friendly with sub-event parallelism if using TBB
- so far, tested with Brunel reconstruction workflow:
  - important speedup already been obtained, but no *'perfect'* scaling achieved yet
  - Algorithm cloning increases parallelism, keeps *latency* under control
- test bench to exercize timings and dependencies for other applications:
  - CMSSW reconstruction workflow
  - ATLAS calo-reconstruction

In a C++11 world...

C++11/C++14 is definitively an improvement,
but the old issues are still with us...
(one needs an adequate understanding of the 1300 pages of the C++ standard)

- build scalability
  - templates
  - headers
  - still no modules/packages
    - maybe in the next Technical Report ? (2017?)
- code distribution
  - no CPAN- nor PyPI-like 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 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

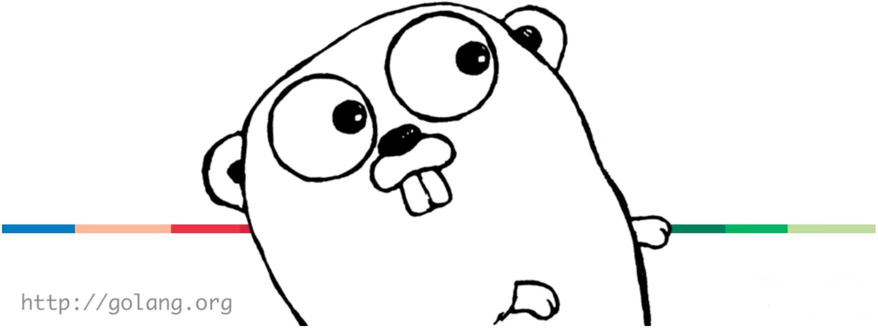
- `python/pypy`
- `FORTRAN-2008`
- `Vala`
- `Swift`
- `Rust`
- `Go`
- `Chapel`
- `Scala`
- `Haskell`
- `Clojure`

Why not Go ?
golang.org

Elements of go

- obligatory `hello world` example...

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



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
  - 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-hep/fads: real world use case

- translated C++ Delphes into Go
- go-hep/fads: Fast Detector Simulation for HEP
- installation:

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

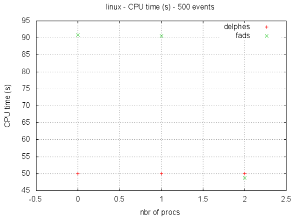
- a HepMC converter,
- particle propagator,
- calorimeter simulator,
- energy rescaler, momentum smearer,
- isolation,
- b-tagging, tau-tagging,
- jet-finder (reimplementation of FastJet in Go)
- histogram service

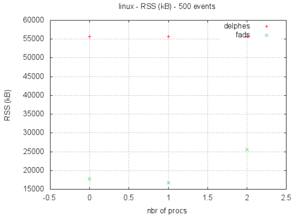
Caveats:

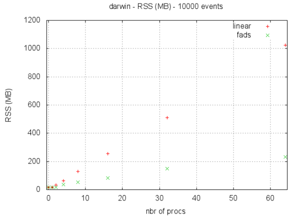
- no real persistency to speak of (JSON, ASCII, Gob)
- jet clustering limited to N^3 (slowest and dumbest scheme of C++-FastJet)

Performances

- good memory footprint scaling (*wrt* `Delphes` and multi-process)
- good `CPU` scaling (*wrt* multi-process)
- OK-ish `CPU` performances *wrt* `Delphes`







Conclusions

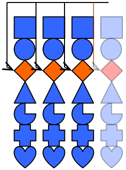
Conclusions

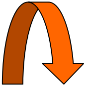
- Moore's law still observed at the *hardware* level
  - *power wall*
  - *memory wall*
  - *ILP wall*
- concurrent and parallel programming required to efficiently and fully leverage today (and tomorrow)'s CPU architectures
  - SIMD (SSE, AVX), (auto-)vectorization
  - *data-parallel*
  - *task-parallel*
  - "think parallel"
  - re-design code and algorithms in a multi-threaded context
- ARM based servers on the verge of being deployed
  - AMD: 2015
  - Boston Viridis: *now*
  - better (nimbler) energy consumption (*wrt* traditional x86)
- Go ?
  - tour.golang.org
  - tailored for concurrent programming
  - tailored for (easy) *cloud* deployment
  - language and *runtime* still relatively young but already quite robust and performant

Bonus

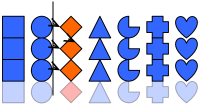
Note on data organisation

Array of objects





Struct of arrays



More suitable for vectorisation

Ex: un objet=une trace avec des variables, des pointeurs vers des informations géométriques, une liste (de longueur variable) de points

Data organisation often need to be completely revisited prior to algorithm vectorisation

(may improve performance even without vectorisation due to better locality (less cache misses))

HEP sw foundation

Motivation

- Pour exploiter efficacement les nouveaux matériels, et garder le contact avec les autres communautés scientifiques, notre patrimoine logiciel vieillissant a besoin d'une refonte profonde (C++11, parallélisme sous toute ses formes).

Proposition

- Créer une collaboration formelle mondiale, afin d'apporter plus de reconnaissance aux contributeurs, de solliciter des fonds auprès de H2020 et NSF/DOE, d'être plus attractif auprès de l'industrie.

Work Packages

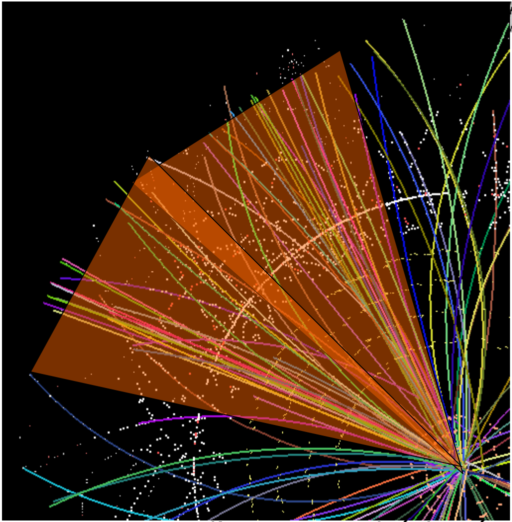
- Etudes R&D courtes sur les alternatives matérielles et logicielles.
- Remaniement des bibliothèques et boites à outils existantes, maintenance à long terme
- Développement de nouveaux composants logiciels d'intérêt général.
- Constitution d'une infrastructure d'essai matérielle (Xeon/Phi, AMD, NVidia, ARM, ...) et logicielle (compilateurs, déboggeurs, profileurs,...).
- Déploiement d'outils et processus communs (dépôts, système d'intégration continue, ...). o Expertise, consultance et accompagnement auprès des expériences.

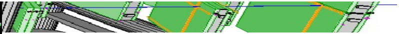
Réunion de lancement au CERN 3-4 avril 2014

Collecte de « white papers » en mai

Accord violent, d'où une organisation légère devrait émerger

Exemple : reco traces





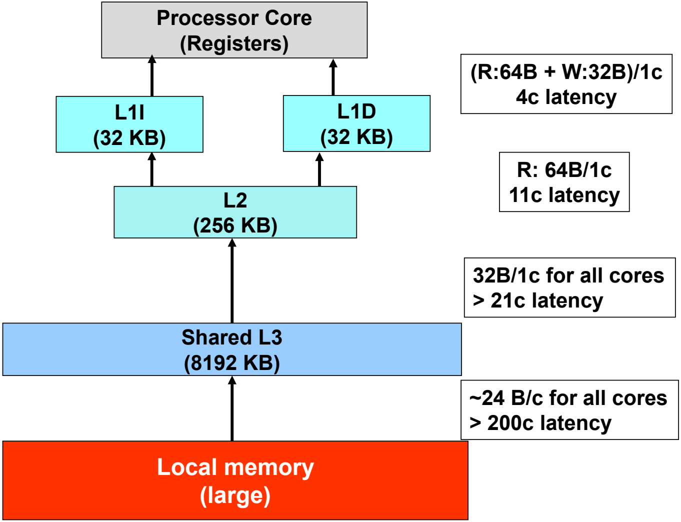
- Option 1: chaque trace reconstruite indépendamment
  - ➔OK mais points partagés?
- Option 2: chaque secteur reconstruit indépendamment
  - ➔OK mais traitement des bords ?

Note : niveau d'exigence *qualité des résultats* vs *rapidité* différent suivant le contexte. On peut être moins précis mais plus rapide au niveau du déclenchement qu'hors-ligne

30

Cache Hierarchy

- **From CPU to main memory on a Haswell processor**
  - With multicore, memory bandwidth is shared between cores in the same processor (socket)



c = cycle