Common Software Tools

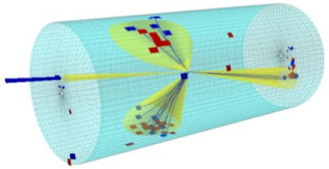
Graeme A Stewart, EP-SFT

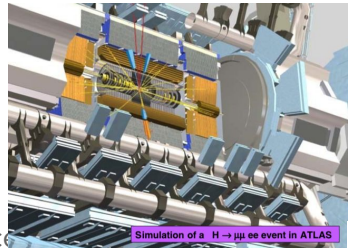
Future Collider Software Workshop, 2019-06-12

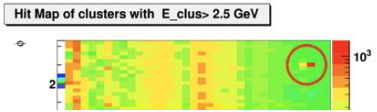
Disclaimer

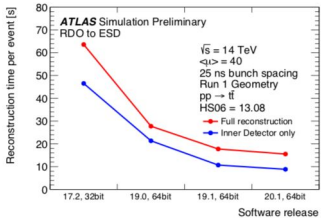
- In this talk I will not mention many specific pieces of software
- Why?
  - Because I think we all know many of them - Geant4, ROOT, DD4hep, PODIO, LCIO, Gaudi, Marlin, CMSSW, acts, PandoraPFA, Trick-Track, ...
    - These will be great to cover in the discussion
  - Because I think the larger questions are
    - How does the lifecycle of software track the lifecycle of an experiment?
    - How should software interact?
    - What programming languages should we use?
    - What is technology going to force us to adapt to?
    - Why do we keep rewriting tools?
    - Best practices for development?
    - How can we cooperate better in the future?
- After that, we can discuss all of the details and enjoy all of the devils within!

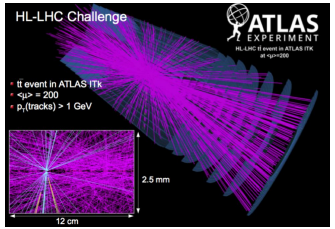
Experiment Software Lifecycle











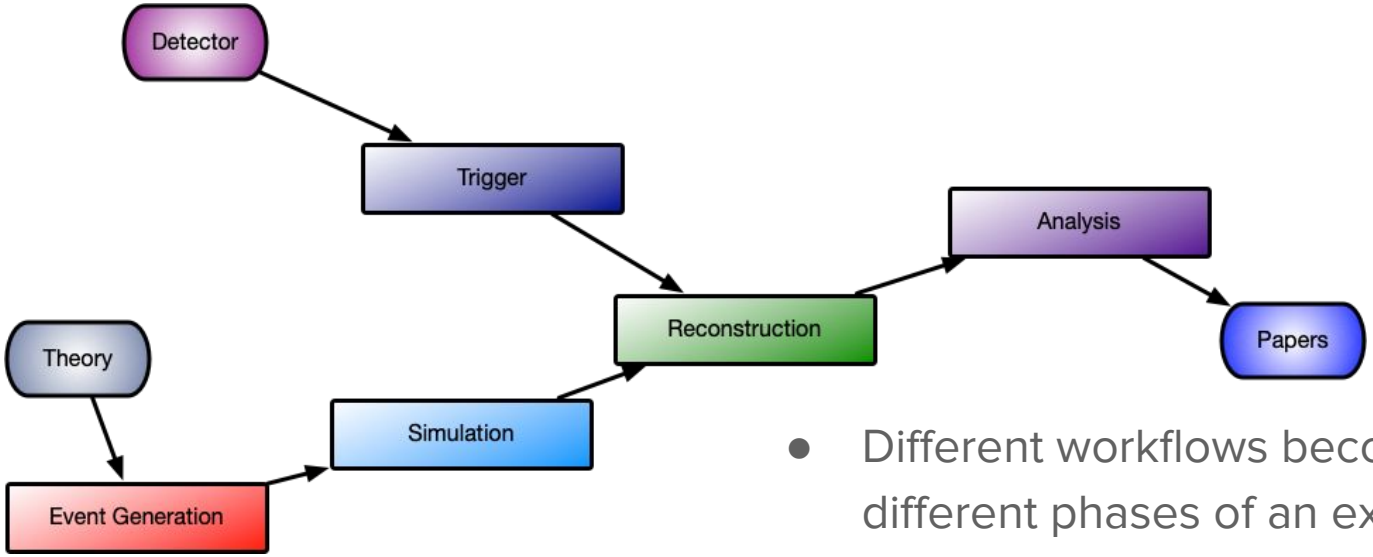
- First ideas and inspiration…
- Concepts
  - Very fast approximate methods, e.g. Delphes (smeared tracks, etc.)
- Design
  - Still need to be flexible to decide between alternatives
  - Ultimately need to pay a lot of attention to details for accurate performance evaluation
    - Accurate geometry, full simulation, realistic digitisation, …
- Production
  - Dealing with the real world - calibration, alignments, dead and noisy elements
  - Learn about the detector, need stability but also continual improvements
- Upgrade
  - Design better sub-detectors for the next version
- Preservation
  - How can I make sure we can look at the data in the future?

Not everything needs to, or should be, solved up-front, but forgetting about the next step entirely will cause problems down the line (**technical debt!**)

```
TOFGN=TOFG*1.E+9
WRITE(CHMAIL,1000)ITRA,ISTAK,NTMULT,(NAPART(I),I=1,5),TOFGN
CALL GMAIL(0,0)
WRITE(CHMAIL,1100)
CALL GMAIL(0,0)
IEVOLD=IEVENT
NTMOLD=NTMULT
```

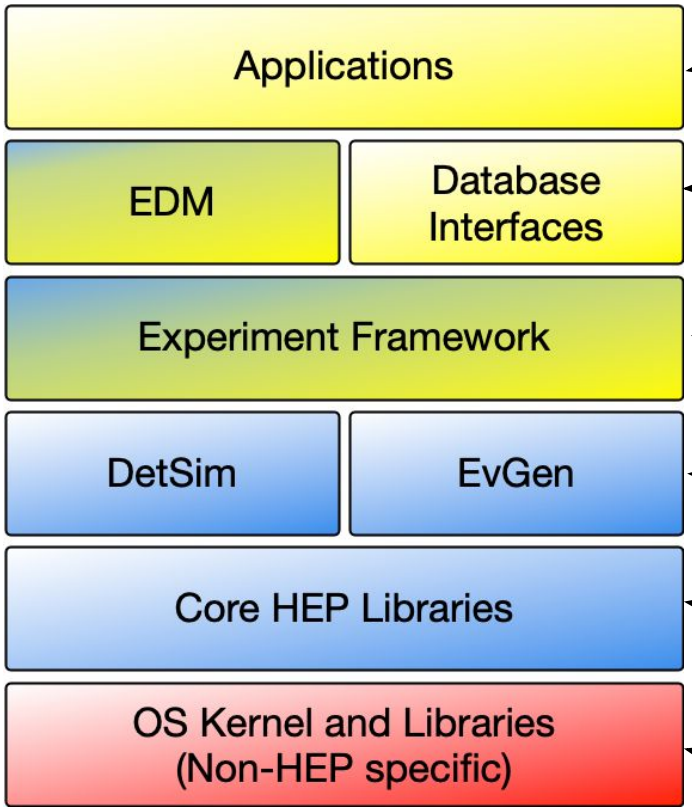
Snippet from CERNLIB

Software Workflows for HEP



- Different workflows become dominant at different phases of an experiment
- In the phases of *conception*, simulation is to relied on
  - This becomes more detailed as the concept develops towards TDRs and beyond

HEP Application Software



Most General → Most Specific

Application layer of modules/algorithms/processors that perform physics tasks (some generic examples like FastJet and PandoraPFA)

Usually experiment specific libraries for data representation and access: xAOD (but LCIO!); also detector specific conditions data

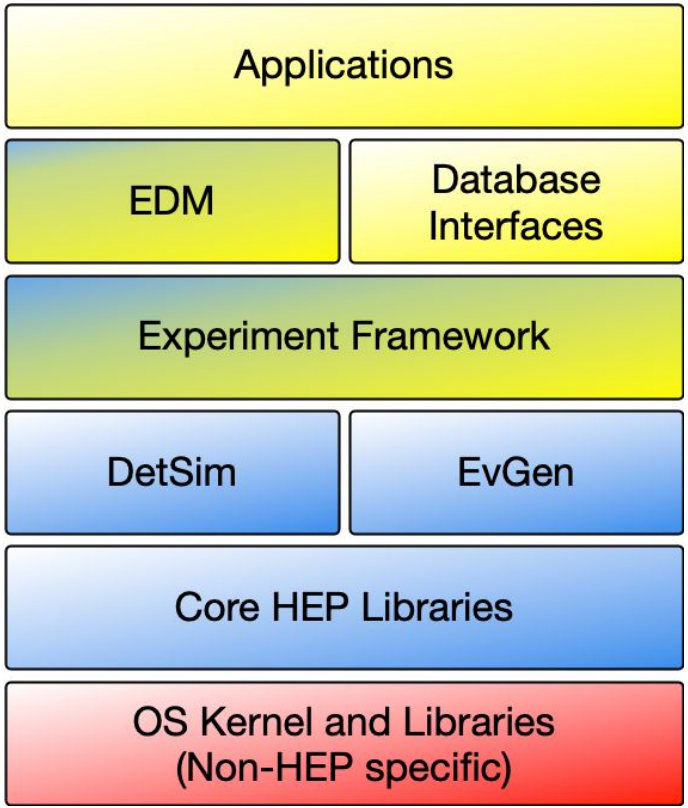
Experiment core orchestration layer, where everything else plugs in: Gaudi, CMSSW, Marlin

Specific components used by many experiments: Geant4, DELPHES, Pythia, ...

Provide core functionality widely used: ROOT, HepMC, HepPDT, DD4hep, ...

Many widely used non-HEP libraries: Boost, Python, Zlib, CMake, ...

Building HEP Applications



- Each piece of software does not live in isolation
- There is an ecosystem of interacting pieces
- Compatibility between the elements doesn't usually come for free
  - Common standards do help a lot
- Building a consistent set of software for an experiment is a task in itself
  - But the software used to do it benefits from commonality
  - LCGCMake, Spack, etc.

Constituent Components

- Foundation Libraries
  - Basic types
  - Utility libraries
  - System isolation libraries
- Mathematical Libraries
  - Special functions
  - Minimization, Random Numbers
- Data Organization
  - Event Data
  - Event Metadata (Event collections)
  - Detector Description
  - Detector Conditions Data
- Data Management Tools
  - Object Persistency
  - Data Distribution and Replication
- Simulation Toolkits
  - Event generators
  - Detector simulation
- Statistical Analysis Tools
  - Histograms, N-tuples
  - Fitting
- Interactivity and User Interfaces
  - GUI
  - Scripting
  - Interactive analysis
- Data Visualization and Graphics
  - Event and Geometry displays
- Distributed Applications
  - Parallel processing (concurrency)
  - Distributed computing (grid/cloud/...)

Interoperability I

- Level 0 - Common Data Formats
  - Allows interoperability between different programs, even running on different hardware
  - E.g., HepMC event records, LCIO, GDML, ALFA messages
- Level 1 - Callable Interfaces
  - Basic calling interfaces defined by the programming language
    - Cross language calls are, of course, possible
  - Can be dependent on the compiler and language version (C++ in particular)
  - Details are important
    - how to handle errors and exceptions, is it thread safe, are objects const, dependent libraries and runtime setup
  - E.g., FastJet, Eigen, Boost

Interoperability II

- Level 2 - Introspection Capabilities
  - Software elements to facilitate the interaction of objects in a generic manner such as Dictionaries and Scripting interfaces
  - Example: PyROOT, which is a Python extension module that allows the user to interact with any ROOT (C++) class from the Python interpreter
- Level 3 - Component Model
  - Software components of a “common framework” offers maximum re-use
  - ‘standard’ way to configure its parameters, to log and report errors, manage object lifetime and ownership rules, ‘standard’ plug-in management, etc.
  - Unfortunately, no single Framework has been generally adopted

The right interoperability point between packages varies, but fixing it correctly eases life a lot for other developers and users

Programming Languages

- Community has standardised on two core languages: C++ and Python
- Other languages do circulate, but rarely seem to offer sufficient advantage over the two current dominant languages to really motivate a change
- C++
  - Extremely high performance when used correctly
  - Very large language that can do anything, certainly more than we need...
  - Training in correct modern use and good framework support essential
- Python
  - Fast to develop in, easy for prototyping
  - Good glue language to express concepts (configuration) or shim different pieces
  - Backed by an increasingly large ecosystem of high performance code (data analytics, machine learning)

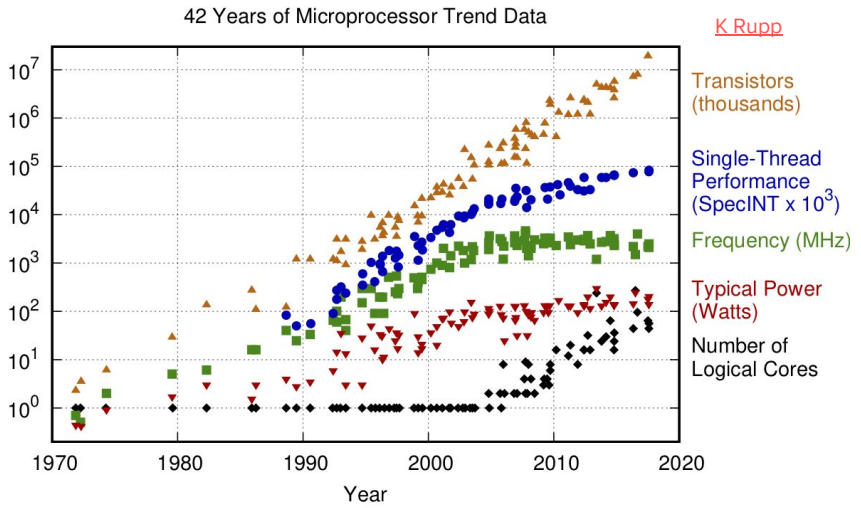
Interactive Access

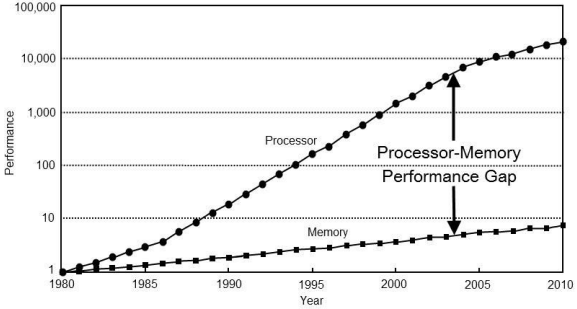
- Prototyping requires rapid iterations, ideally interactivity as well
- There has been a huge advance in this area in recent years
  - Jupyter notebooks brought a new level of interactivity and sharing to Python
  - ROOTv6 revolutionised ROOT internals, replacing the ageing CINT with Cling (llvm plugin)
  - Cling is usable more generally to power interactive C++
  - SWAN service brings all of this to bear in an interactive service, supporting C++, Python, ROOT
- Interactive services also able to now steer cluster level resources (e.g., Python's Dask)

ROOT
Data Analysis Framework

Technology Evolution

- Moore's Law continues to deliver increases in transistor density
  - But, doubling time is lengthening
- Clock speed scaling failed around 2006
  - No longer possible to ramp the clock speed as process size shrinks
- Basically stuck at ~3GHz clocks from the underlying Wm^{-2} limit
  - Limits the capabilities of serial processing
- Memory access times are now ~100s of clock cycles
  - Poor data layouts are catastrophic for software performance
- Conclusion is that diversity of new architectures will only grow
  - Best known example is of GPUs



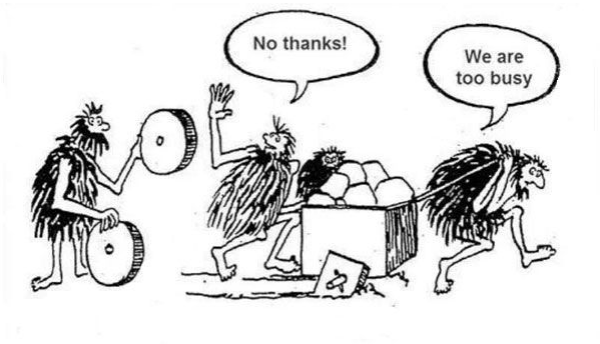


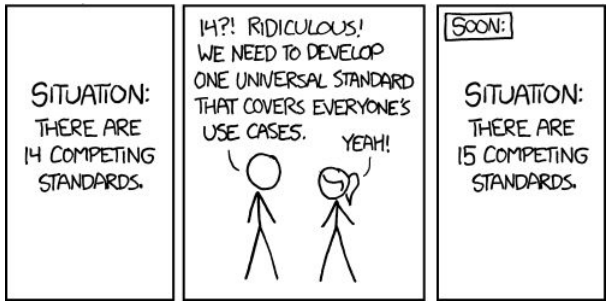
Concurrency and Heterogeneity

- The one overriding characteristic of modern processor hardware is concurrency
  - SIMD - Single Instruction Multiple Data (a.k.a. vectorisation)
    - Doing exactly the same operation on multiple data objects
  - MIMD - Multiple Instruction Multiple Data (a.k.a. multi-theading or multi-processing)
    - Performing different operations on different data objects, but at the same time
- Because of the inherently parallel nature of HEP processing a lot of concurrency can be exploited at rough granularity
  - However, the push to highly parallel processing (1000s of GPU cores) requires **parallel algorithms**
- Developments to exploit heterogeneous architectures are happening now in the current experiment's frameworks
  - Gaudi, CMSSW, ALFA/O2 in particular
  - This requires advanced designs to hide latency and also careful study of appropriate data layouts
    - Data layouts for the LHC experiments are being optimised - it's clear that some level of abstraction for clients is important to ease that process

But wait, all I wanted to do was...

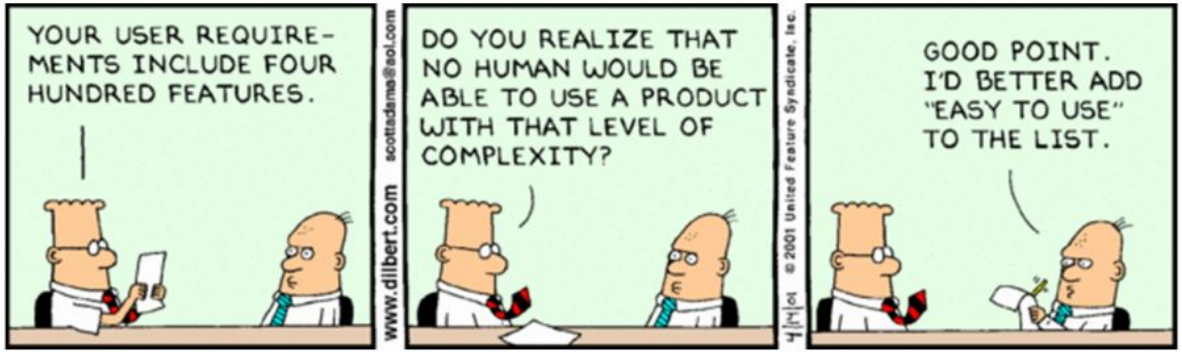
- Fatal thinking behind all attempts to reinvent software in general is to think too narrowly about the problem to be solved
  - Mature solutions can seem overly complex and difficult
  - This situation is not helped if the documentation for current projects is poor
- If approached the the wrong way then complexity gets re-added in an ad hoc manner
  - Leading to a much more complicated piece of software than the original version it was designed to replace
- This does not mean that we should not rewrite code
  - But when we do we should have a clear *architecture* that learned from past attempts





All Reasonable Requests...

- Edge cases can be a curse
  - Having to do something that was not in the original vision of the software
- Dialogue between the users and the developers is important to consider alternative ways to solve a problem
  - E.g., Not reasonable to store changing conditions in the database in millisecond intervals, but some detectors need this
  - Solution: use a compound data object that is stored per lumi-block



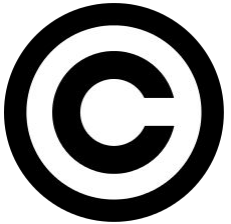
Open Source Development

- A more modular development practice is very much enabled by modern open source tools
  - Our students grow up with this as *the way* to develop software
- git is the common standard for source code
- Social coding sites, like GitHub and GitLab, amplify the git advantage enormously
  - Pull request workflow
  - Code discussion and review in advance of acceptance
- Standard build systems, like CMake, help a great deal with integration of dependencies as well
  - Especially new CMake3 concepts
  - HSF project template script will set this up correctly

Build, Test, Deploy, Run

- Continuous integration of code helps the review process a lot
- This of course relies on a robust test suite
  - Even more important when a wide group of users adopt a package to ensure that functionality doesn’t break
- Ease of deployment is also important
  - CVMFS for production
  - Containers for easy runtimes
    - Can be lightweight and leverage CVMFS
  - Local installations great for developers, though runtime can be tricky

Copyright and Licenses



- This is far from the most exciting topic for physicists or scientific software developers, but…
- It is actually very important to establish who owns pieces of code and how other people can contribute to their development
  - N.B. this contribution extends to other sciences and academic departments and to industry
- Copyright
  - The legal owner of the code, can be a lab, cannot be an experiment
    - © CERN for the benefit of the FOO experiment
  - Keep this as small as practical - good idea to enshrine it in a collaboration agreement
- License
  - Grant the ability to use the code to other people
  - Open Source almost goes without saying
  - Merits of GPL vs. permissive are a loooong debate
    - However, for library software strong advice to avoid GPL

Distributed Computing

- Sooner or later computing problem becomes too large for a single workstation or a small cluster
- Then having access to distributed resources is essential
  - Single cluster support from a lab may be limited (a la lxbatch)
  - Endeavours are more international and cooperative anyway
- Significant user communities have emerged around two key components that allow you go distributed
  - Rucio - for advanced data management
  - DIRAC - for workload management (and some basic data management features)

HEP Software Foundation

- Formed in 2015 to encourage common approaches to software in HEP
- Published the Roadmap for Software and Computing in 2020s in CSBS [https://doi.org/10.1007/s41781-018-0018-8; arXiv:1712.06982]
- Now has active working groups in many important areas
  - Event Generation
  - Detector Simulation
  - Reconstruction and Software Triggers
  - Data Analysis
  - Software Tools and Packaging
  - Training (and careers)
- Overlap between current and next experiments is huge
  - Plenty of opportunity for interaction

Conclusion

- HEP has a rich collection of software already and a lot of experience
- We face a lot of challenges in the future and we need to be aware of these when solving today's problems
- Common development helps develop good software and to sustain it through the lifetime of our experiments
- Good practice, flexibility and communication are key for successful interoperability and reuse of tools and projects

This is the key work for the HEP Software Foundation