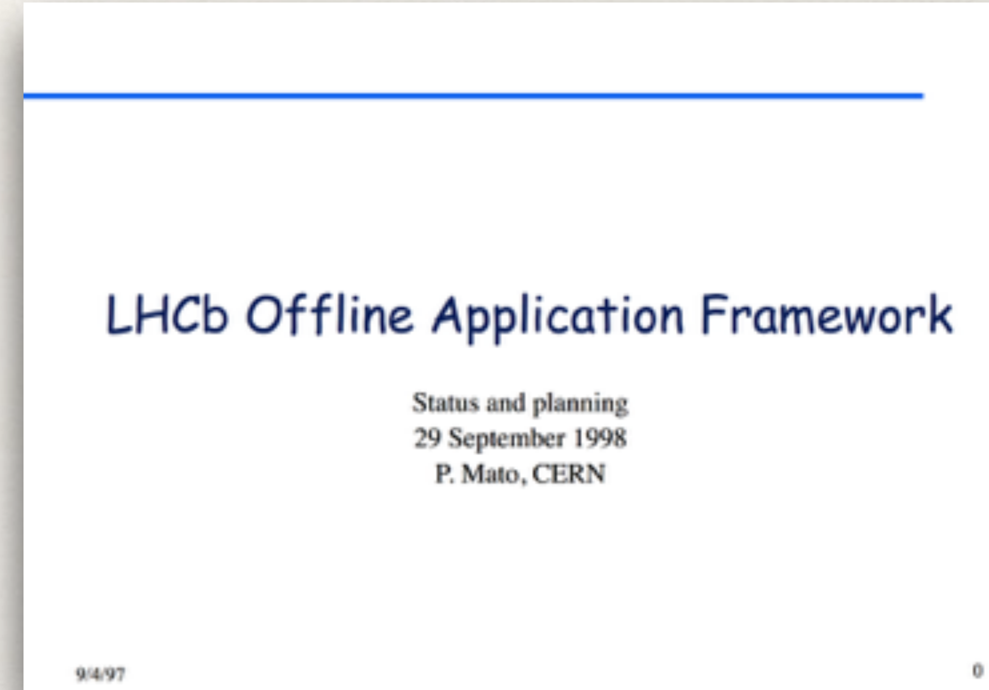
Gaudi History

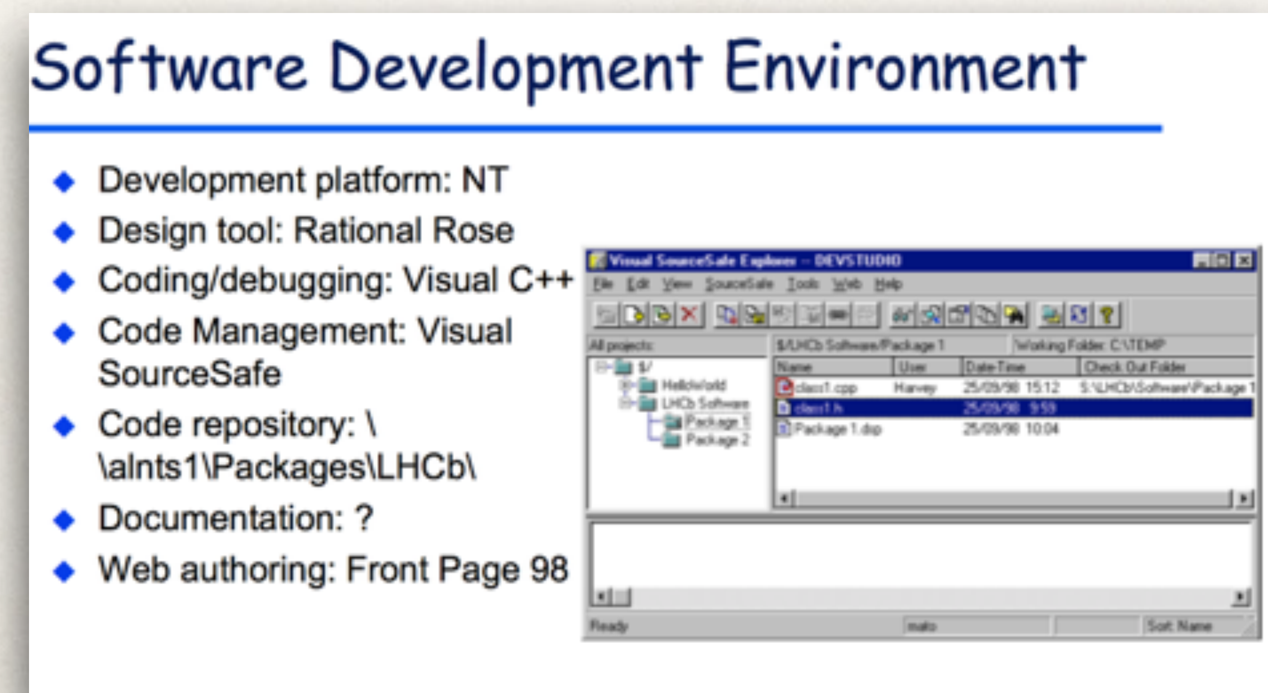
Gaudi Workshop, 21-23 September 2016

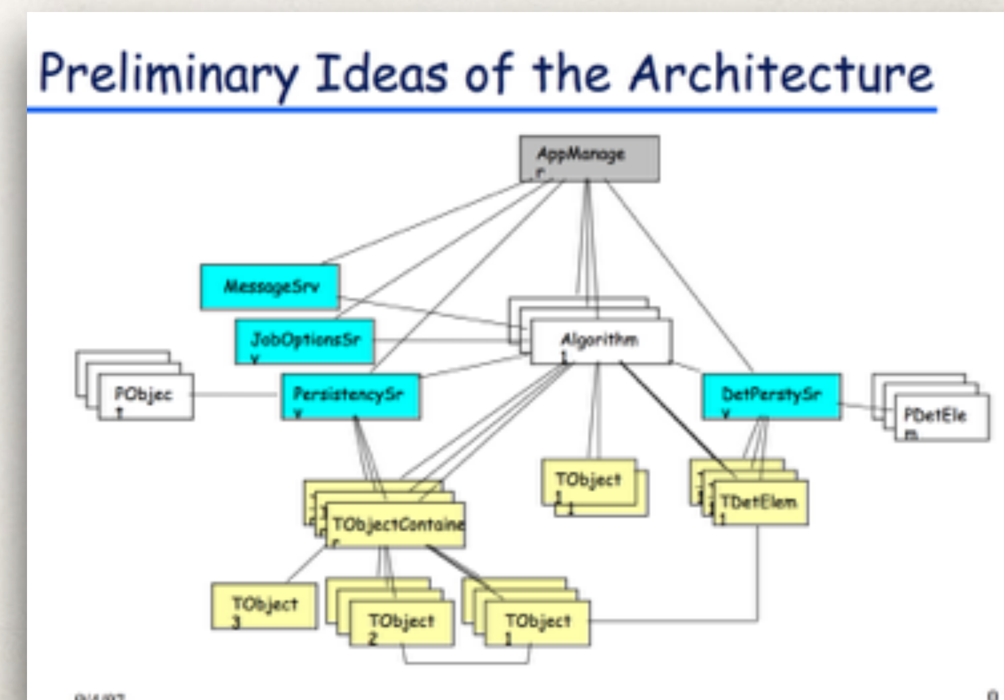
Pere Mato / CERN

Gaudi before called Gaudi

- Back in autumn 1998 first ideas being layered-down
  - Introduced new concepts ‘framework’, ‘architecture’, ‘components’, …
  - Most of them still valid…
- About the software development environment :-)







First Architecture Review

Major design criteria

- Clear separation between "data" and "algorithms"
- Three basic types of data:
  - **event data** (data obtained from the particle collisions)
  - **detector data** (structure, geometry, calibration, alignment, environmental parameters,..)
  - **statistical data**: (histograms, …)
- Clear separation between "persistent data" and "transient data".
  - Isolation of user's code.
  - Different/incompatible optimization criteria.
  - Transient as a bridge between various representations.

20/9/16 LHCb Computing 8

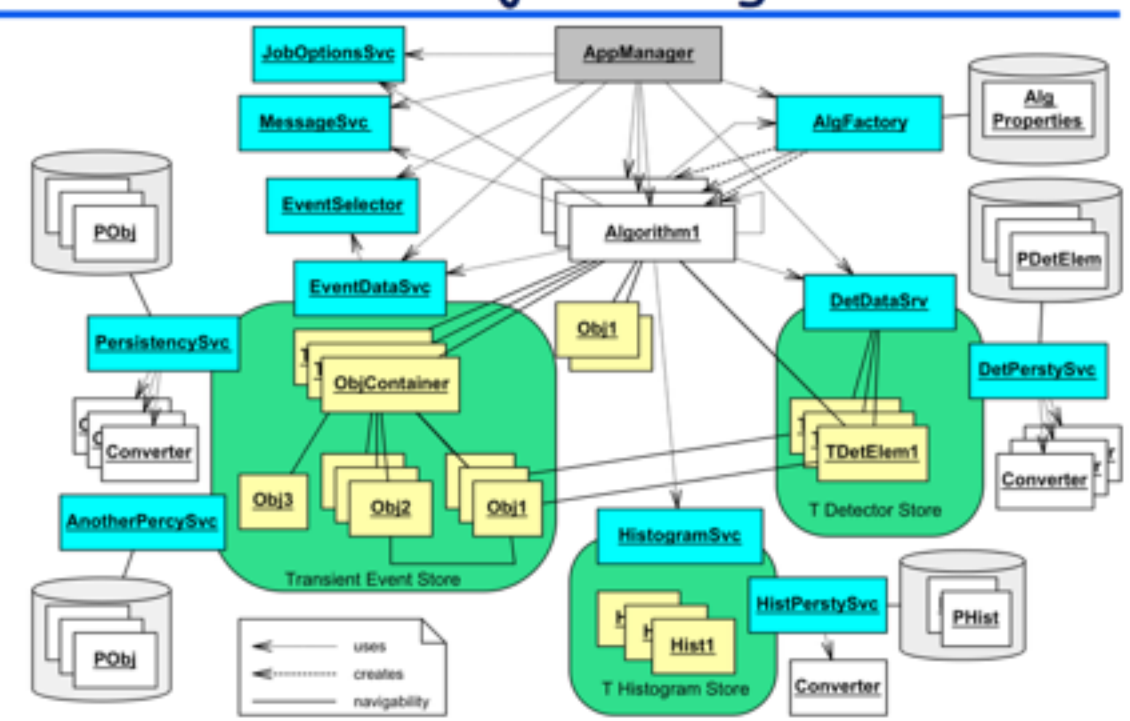
Major design criteria (2)

- Data centered architectural style.
  - Algorithms as data producers and consumers.
- *User code* encapsulated in few specific places:
  - "Algorithms": Physics code
  - "Converters": Converting data objects into other reprentations
- All components with well defined "interfaces" and as "generic" as possible.
- Re-use components where possible
- Integration technology standards

20/9/16 LHCb Computing 9

- November 1998 (almost 18 years ago!)
  - Use Cases
  - Name, logo, initial development team, **design criteria**, architecture, etc.
  - Initial implementations

Architecture: Object Diagram



9/4/97 LHCb Computing 0

Development Strategy

Followed strategy

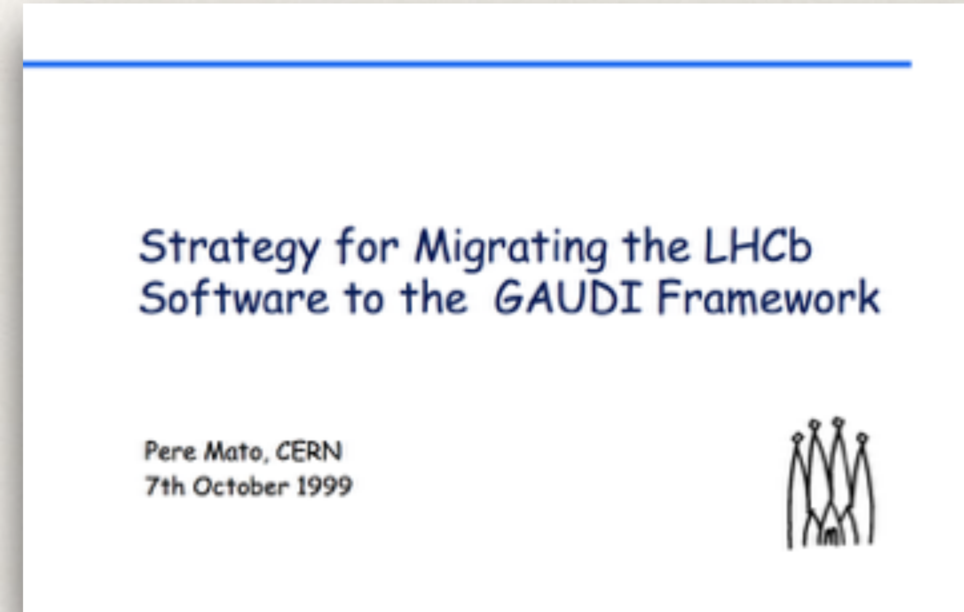
- Start with small design team of 6-8 people
  - architect, librarian, domain specialists with design/programming experience
- Collect User Requirements and use-cases
- Establish basic criteria for the overall design
- Make technology choices for implementation of initial prototypes
- Incremental approach to development.
  - Release every ~4 months.
  - Releases accompanied by complete documentation
  - Development cycle driven by the users: priorities, feedback, etc.
- Strategic decisions after thorough design review (~1/yea

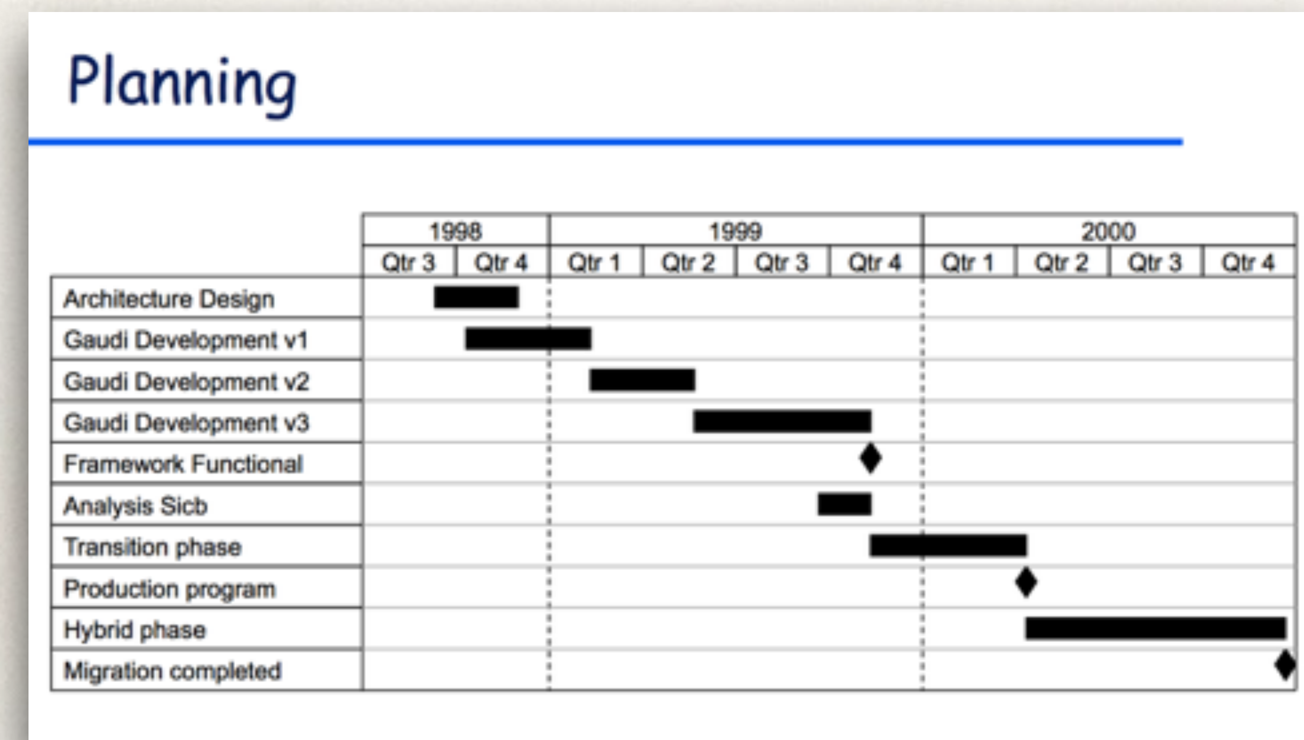
Project History

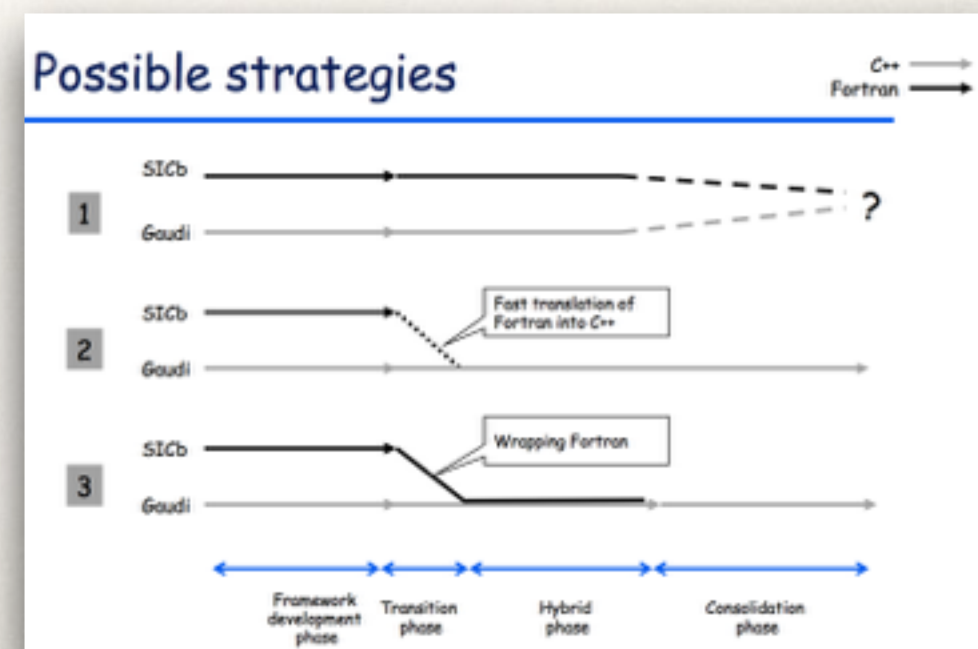
- Sep '98 - architect appointed, design team (6 people) constituted
- Nov 25 '98 - external architecture review
  - objectives, architecture design document, URD, scenarios
- Feb 8 '99 - first GAUDI release
  - first software week, presentations, tutorials
  - plan second release (together with users)
  - expand GAUDI team
- May 30 '99 - second GAUDI release
  - second software week, plan third release with users, expand team.
- Nov 23 '99 - third GAUDI release and software week
  - plan deployment for production applications
- Spring '00 - second external review

Ready for the LHCb Migration

- 1 year later the the framework had sufficient functionality to start the migration of the LHCb software
- Use the **same framework for ALL applications** (simu, reco, ana, trigger)







ATLAS interested in Gaudi

- February 2000, first presentation/discussion with ATLAS
- ATLAS decided to join efforts after some negotiations
  - ATLAS will call the framework "Athena"
  - Incorporation of specific services like "StoreGate"

Possible Collaboration

- Scope
  - Common foundation libraries
  - Common interface model
  - Common frameworks (interfaces + basic services)
  - Different *Event Model* and *Algorithms*
  - Different Applications
- Benefits
  - Better design
  - Sharing development of basic infrastructure services (higher quality)
  - CERN/IT efforts better focussed (single request may fulfill more than one experiment) (AIDA project)
  - Better communication (same vocabulary)

Possible Collaboration (2)

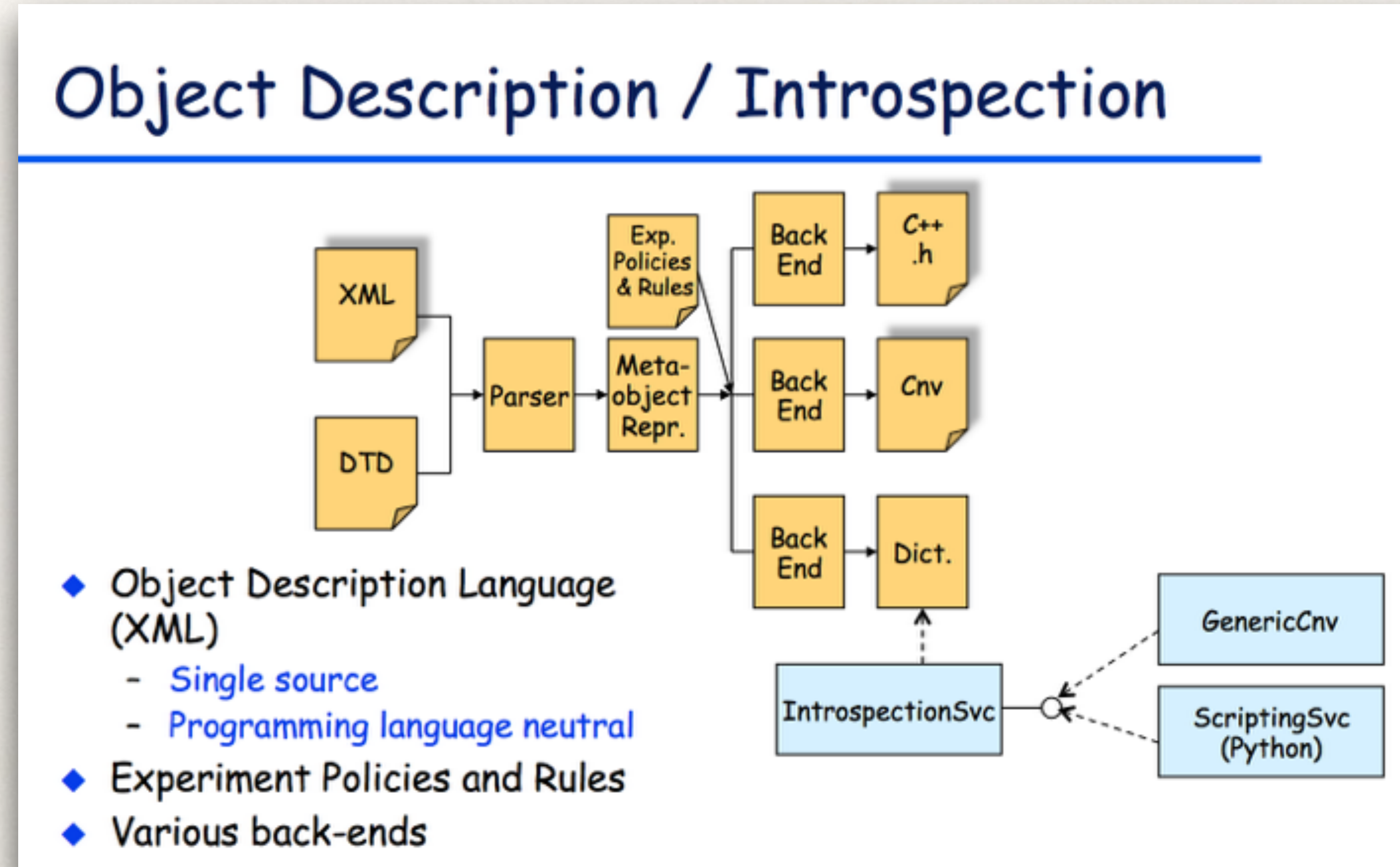
- Disadvantages
  - Less freedom
  - Needs more formality (change procedures, upgrades, etc.)
  - It may fail
- Practical aspects
  - Regular meetings, workshops, ...
  - Mailing lists and other collaborating tools
  - Common code repository ?

- December 2000: ATLAS Architecture Review

CERN

Major Additions/Evolutions

Data Definition (GOD) - Dec 2001



CERN

Gaudi Sequences - Mar 2003

Event Filtering Requirements

- The ability to vary the processing based on the physics signature
  - Different sequences of *Algorithms*
    - » Concept of processing **path**
  - Different parameters (properties) for *Algorithms*
    - » Concept of algorithm **instances**
- The ability to make event selections based o
  signature
  - Not all the events passes through all the trigger
    chain
  - Early termination of processing if event fails sele
    - » Concept of **filter**

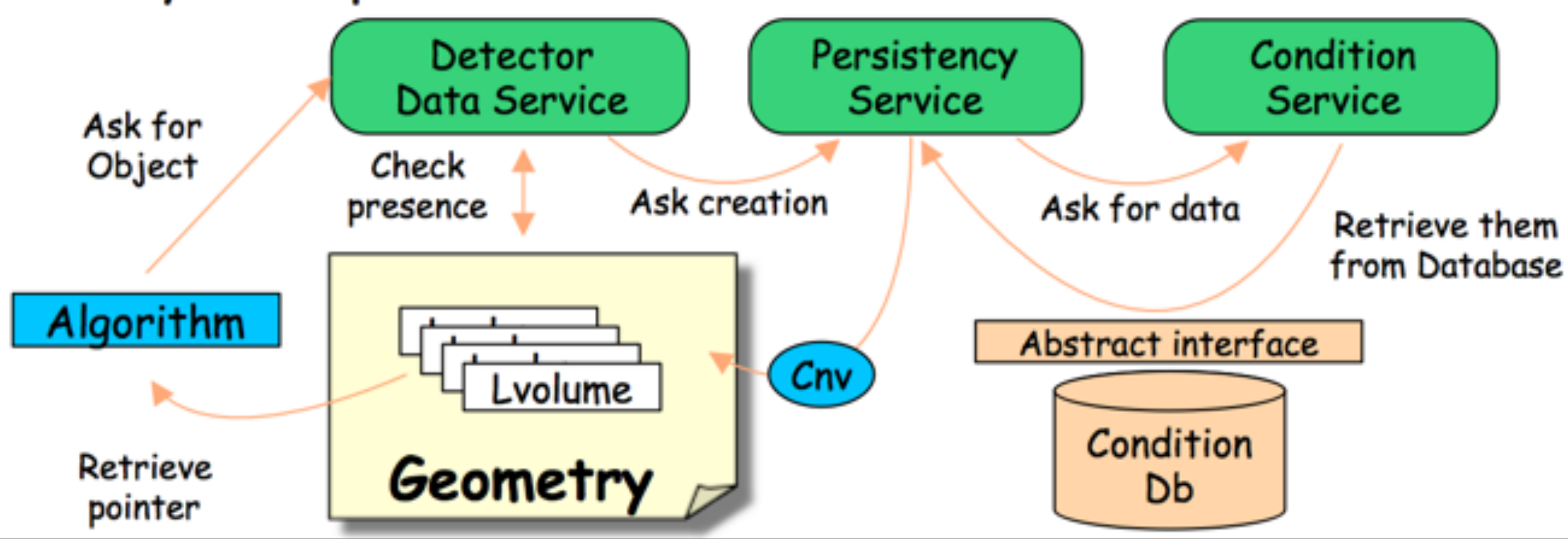
Implementation

- Sequencer Class
  - Subclass of *Algorithm*
  - Manages a set of members (other Algorithms or Sequencers).
  - Allows hierarchical sequencing.
  - Obeys filtering/enabled protocols (see later)
- Filter Handling
  - *Algorithms* can call setFilterPassed(true/false)
    - » Default is true
  - *Algorithms* downstream of the one that sets it's filter flag to false will not get executed
    - » This default behavior can be overridden by the "StopOverride" property of the *Sequencer*

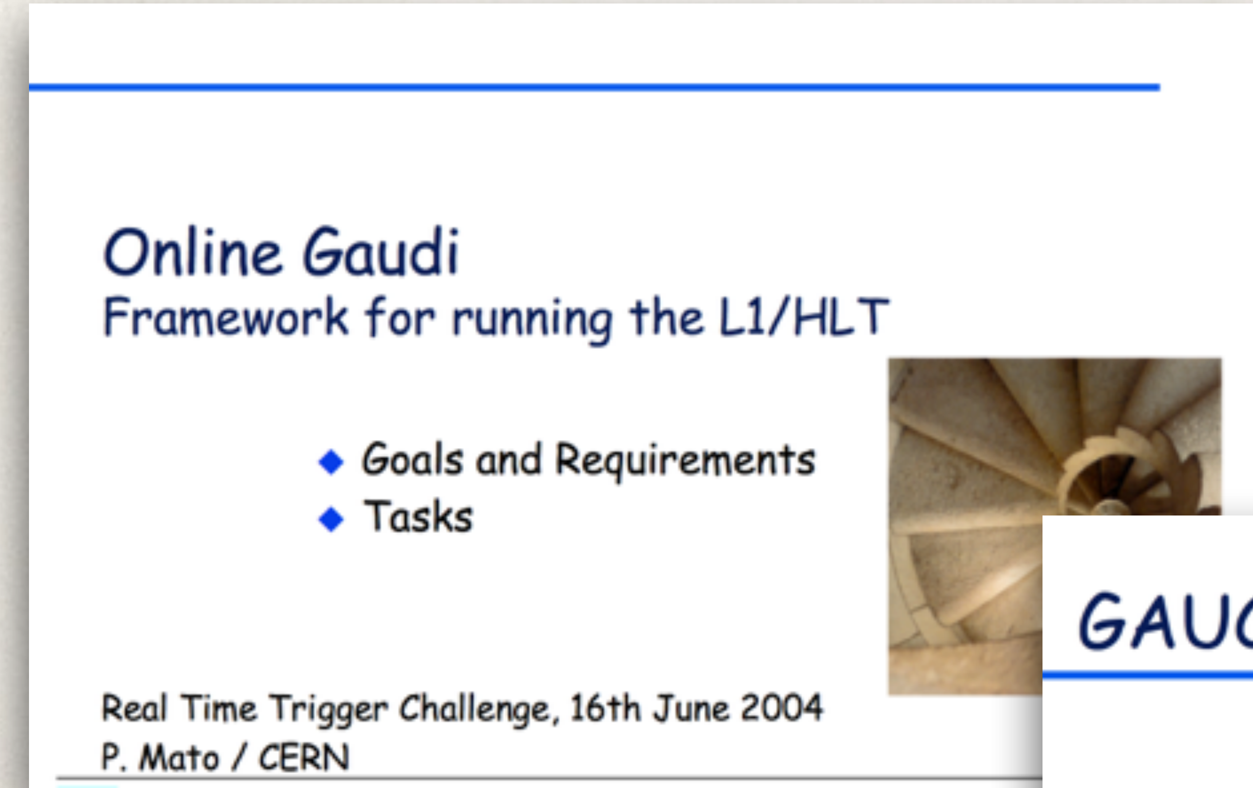
Conditions DB - Dec 2003

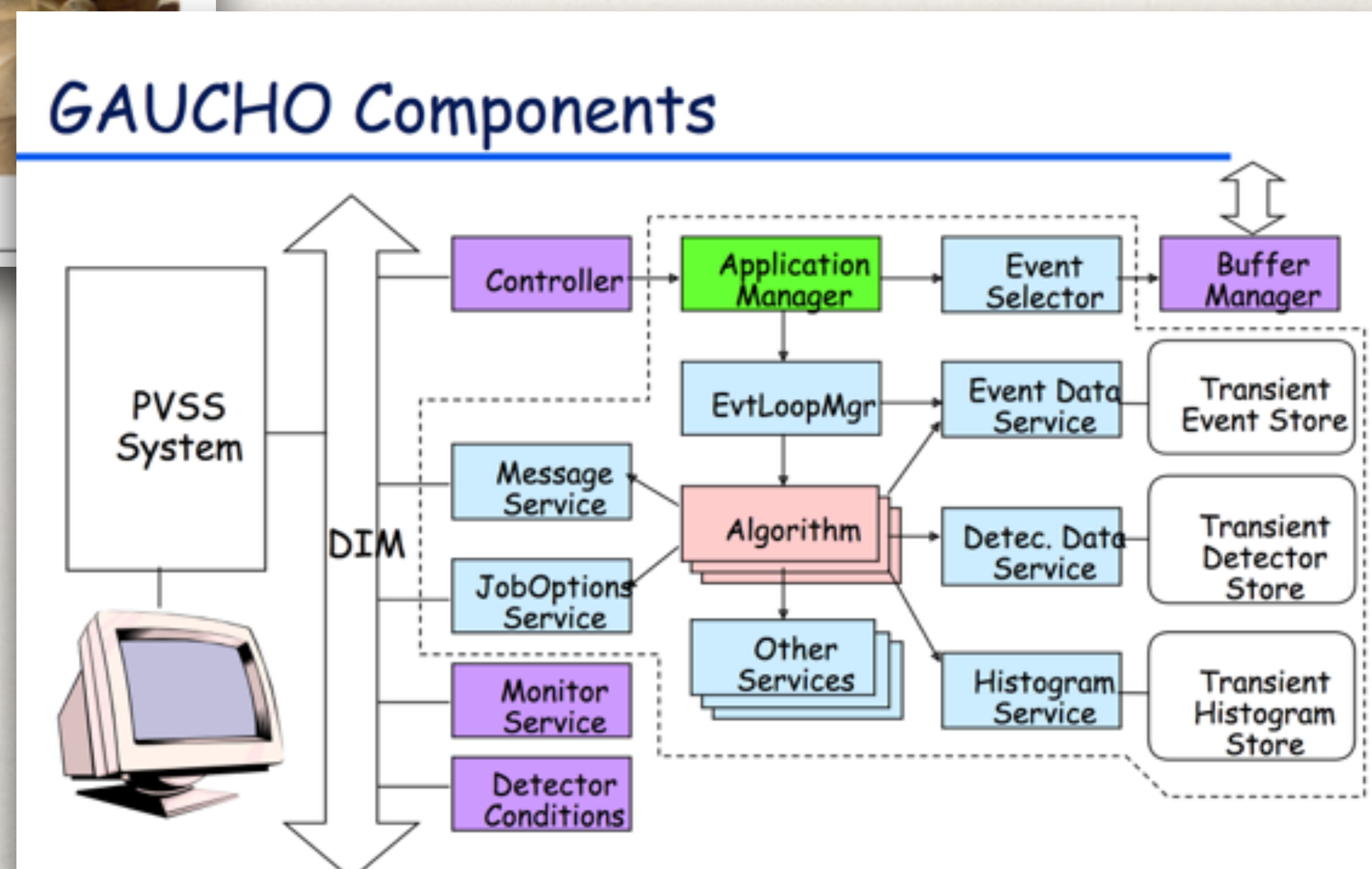
Gaudi Interface to Conditions Db

- Emphasis on the data retrieval functionality
- One new service was defined : *ConditionSvc*
- Independent from data content, only deals with data retrieval depending on time, version and/or tag
- Fully transparent for the user



Online Gaudi - June 2004





GaudiPython - June 2004

GaudiPython

- Enabling the interaction of Gaudi components from Python
  - Configuration, Interactivity, etc.
- Starting from Gaudi v14r1, GaudiPython has been re-implemented using PyLCGDict
  - Generated dictionaries for most common Gaudi "Interfaces" and "Base classes" (~80 classes)
  - Not need to generate dictionaries for all classes (in particular the implementations)
- The end-user module "gaudimodule.py" hides some of the technicalities and adds some handy functionality
  - Very easy to extern/modify/adapt since is written in Python
  - Basically backward compatible with previous version

PyLCGDict: Supported Features

- Conversion of C++ and Python primitive types
- C++ classes
  - Mapped to Python classes and loaded on demand. Templated classes supported.
- C++ namespaces
  - Mapped to python scopes. The "::" separator is replaced by the python "." separator
- Class methods
  - Static and non static class methods are supported. Default arguments.
  - Method arguments are passed by value or by reference
  - The return values are converted into python types and new python classes are created if required. Dynamic type returned if possible.
  - Method overloading works by dispatching sequentially to the available methods with the same name until a match with the provided arguments is successful.
- Class data members
  - Public data members are accessible as python properties
- Emulation of python containers
  - Container C++ classes (*std::vector*, *std::list*, *std::map* like) are given the behavior of the python collections to be use in iterations and slicing operations.
- Operator overloading
  - Standard C++ operators are mapped to the corresponding python overloading operators

New PluginMgr - Sept 2006

Using Plugin Manager

- Coding the plugin/component
  - No predefined model
  - Declaring factory with signature

```
class MyClass : public ICommon {
  MyClass(int, ISvc*);
  ...
};
```

MyClass.h

```
PLUGIN_FACTORY(MyClass,ICommon*(int,ISvc*));
/* implementation */
```

MyClass.cpp

- Creating the rootmap file
  - Text file listing all plugins and the associated dynamic library
  - The build system creates it

```
Library.MyClass:          MyLibrary.so
Library.AnotherClass:     MyLibrary.so
```

rootmap

- Instantiating the plugin
  - Library loaded if needed
  - Strong argument type checking

```
...
ISvc* svc = ...
ICommon* myc;
myc = PluginMgr::create<ICommon*>("MyClass",10, svc);
If ( myc ) {
  myc->doSomething();
}
```

Program.cpp

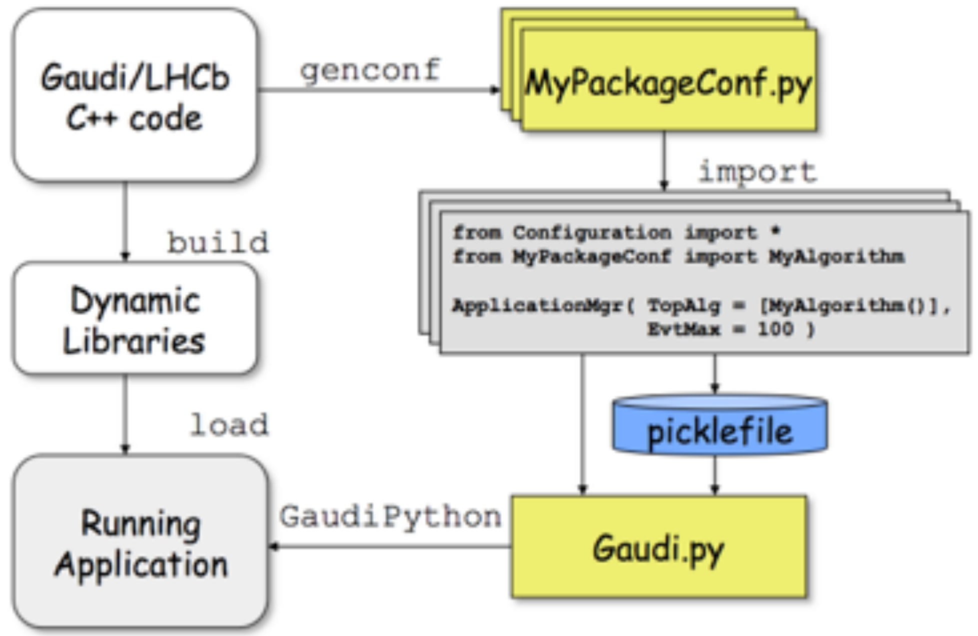
CERN

Configurables - Sept 2007

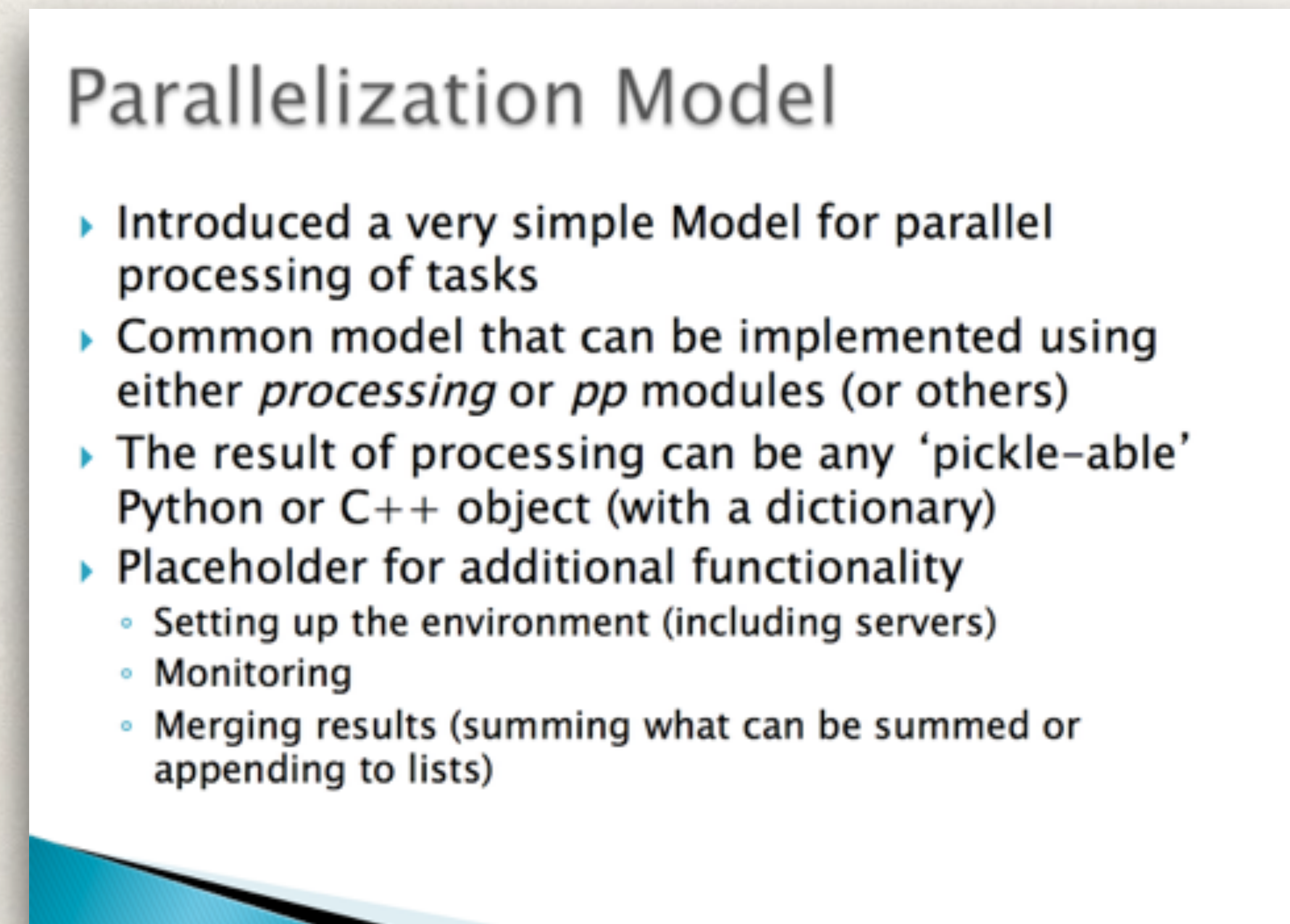
Python Configuration

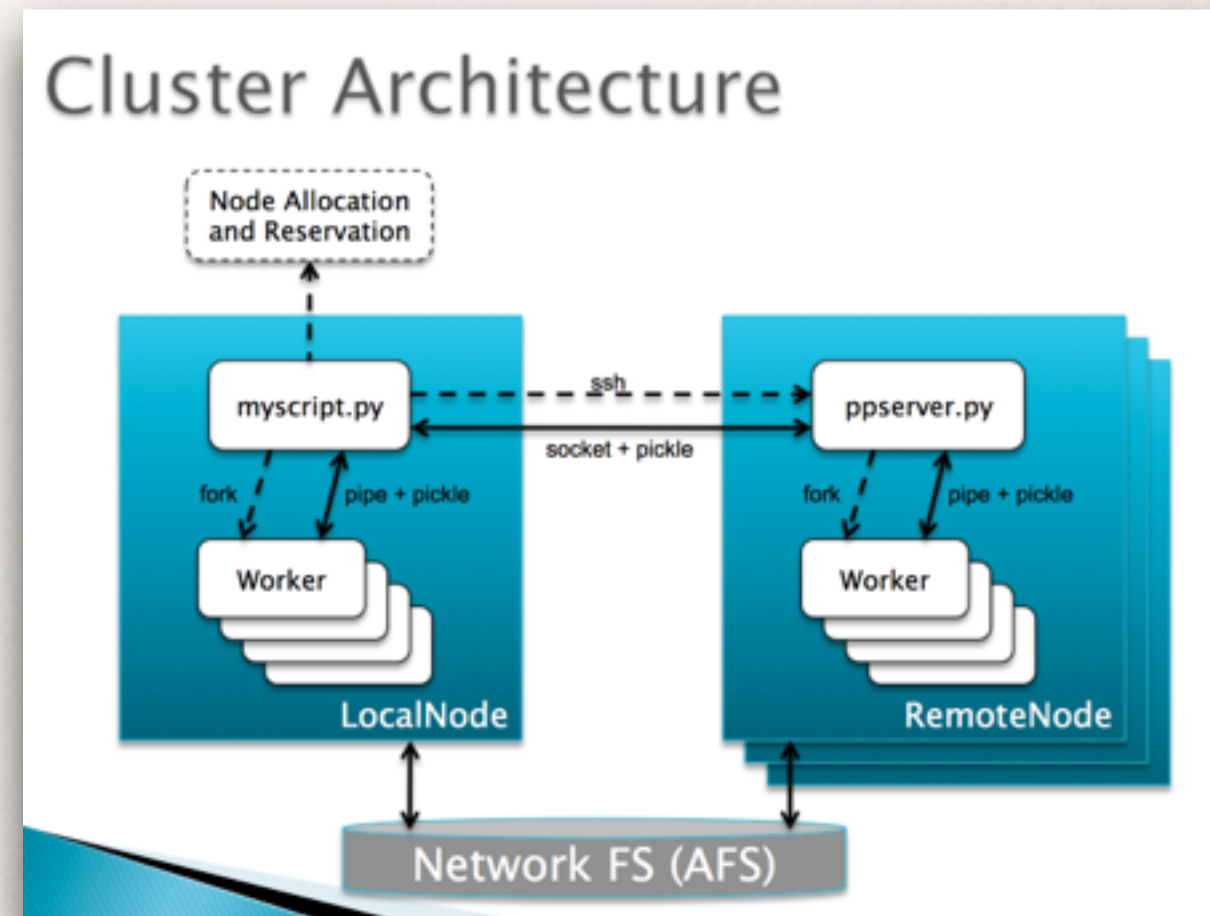
- Background
  - Proposed in Barcelona at collaboration meeting 2 years ago
  - What was available from ATLAS at that time was not usable
  - What is available from ATLAS is much better … but it maybe some tuning is still needed
- Goals
  - Use the power of a powerful and complete language to configure applications
    - Expressions, if-then-else logic, loops, modularization, etc.
  - Validate configuration earlier in the process
    - Incorrect type, non-existing property, non-existing component, etc.
  - Increase user friendly-ness
    - Less writing, avoid duplication of information, etc.
  - Smooth migration from JobOptions files to Python confguration files
    - The adoption/transition should happen when convenient and in steps

Configurables

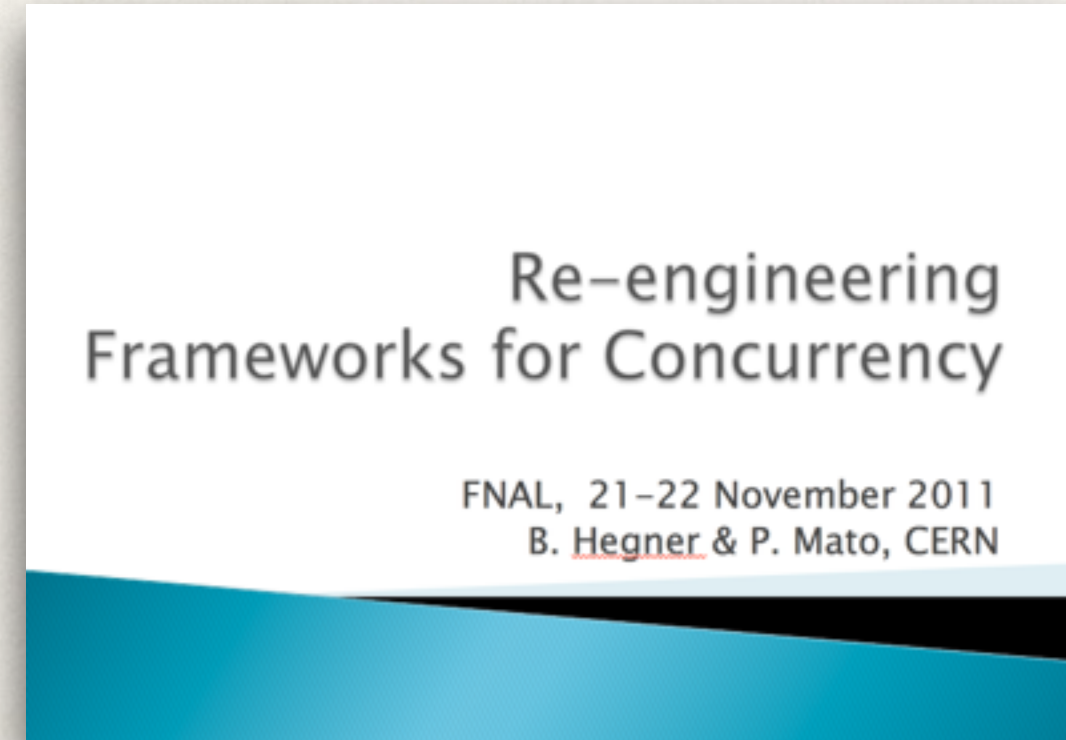


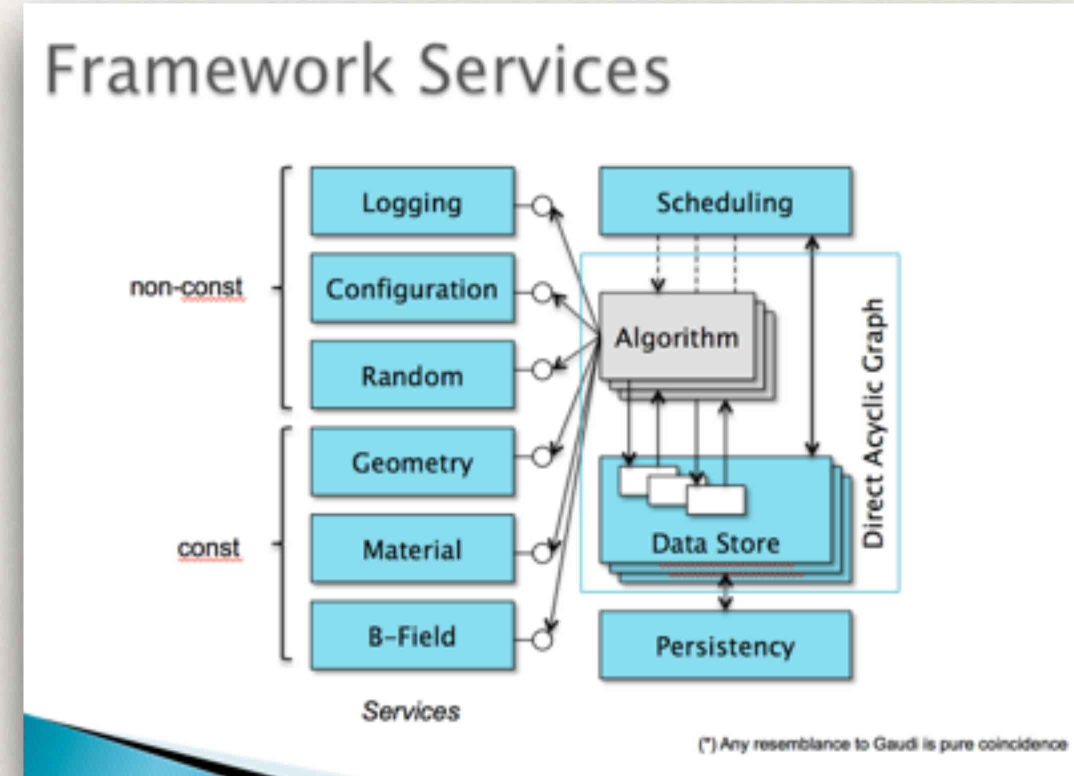
Parallel Gaudi - June 2008

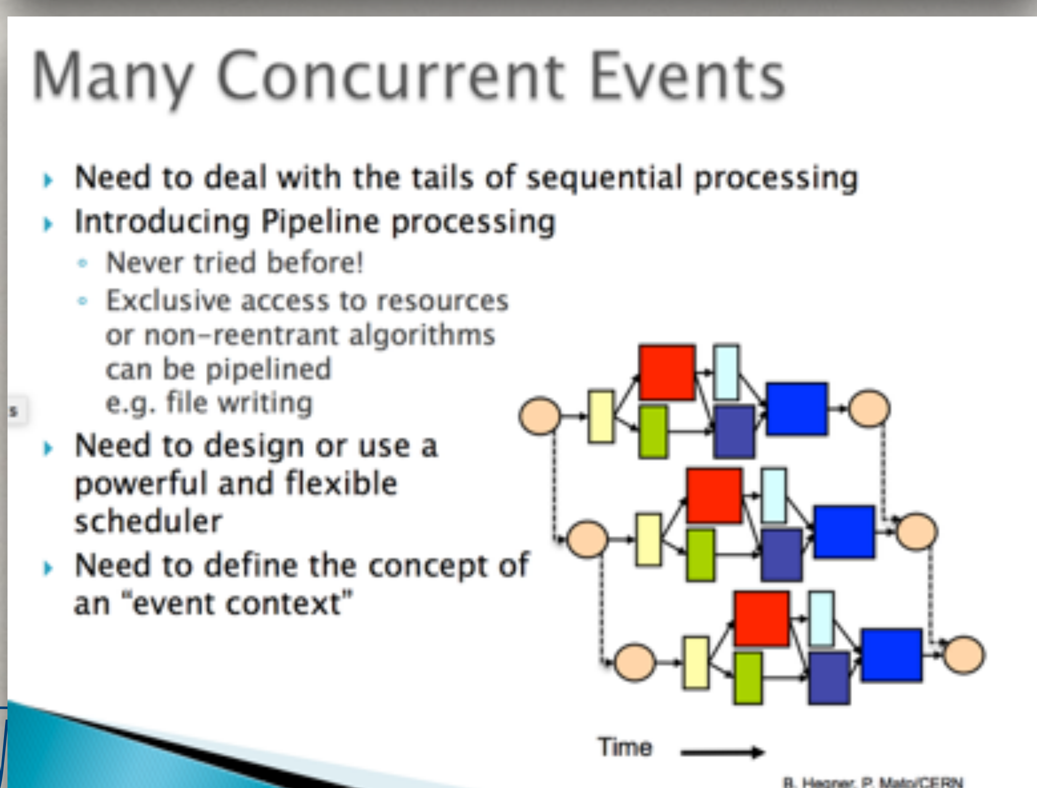


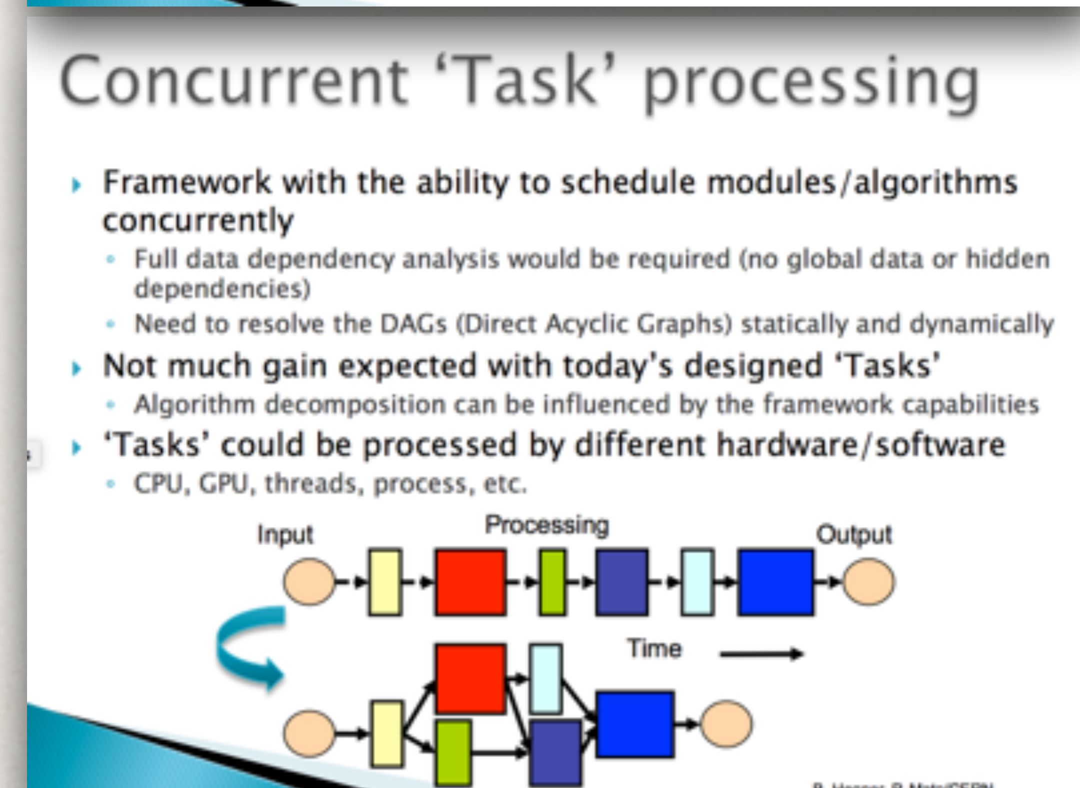


Concurrency - Nov 2011







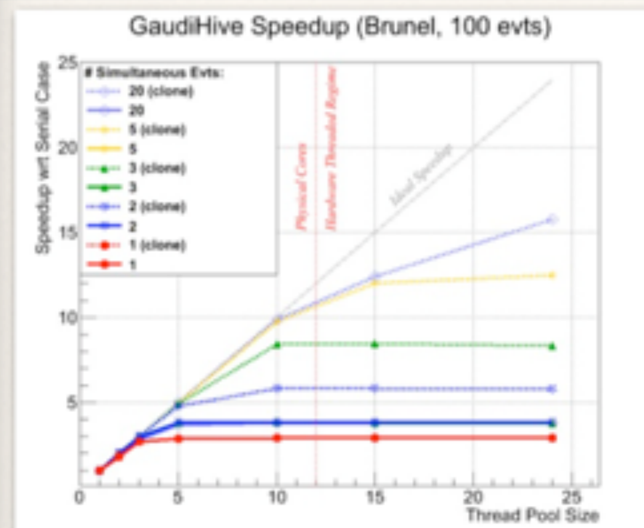


Hive First Results - Nov 2012

Prototype: GaudiHive

- So far a 'toy' Framework implemented using TBB
  - No real algorithms but CPU crunchers
  - Timing and data dependencies from real workflows
- Schedule an *Algorithm* when its inputs are available
  - Need to declare *Algorithms'* inputs
  - The tbb::task is the pair (Algorithm*, EventContext*)
- Multiple events managed simultaneously
  - Bigger probability to schedule an *Algorithm*
  - Whiteboard integrated in the Data Store
  - Which has been made thread safe
- Several copies of the same algorithm can coexist
  - Running on different events
  - Responsibility of AlgoPool to manage the copies
- Some services have been made thread-safe
  - E.g. TBBMessageService

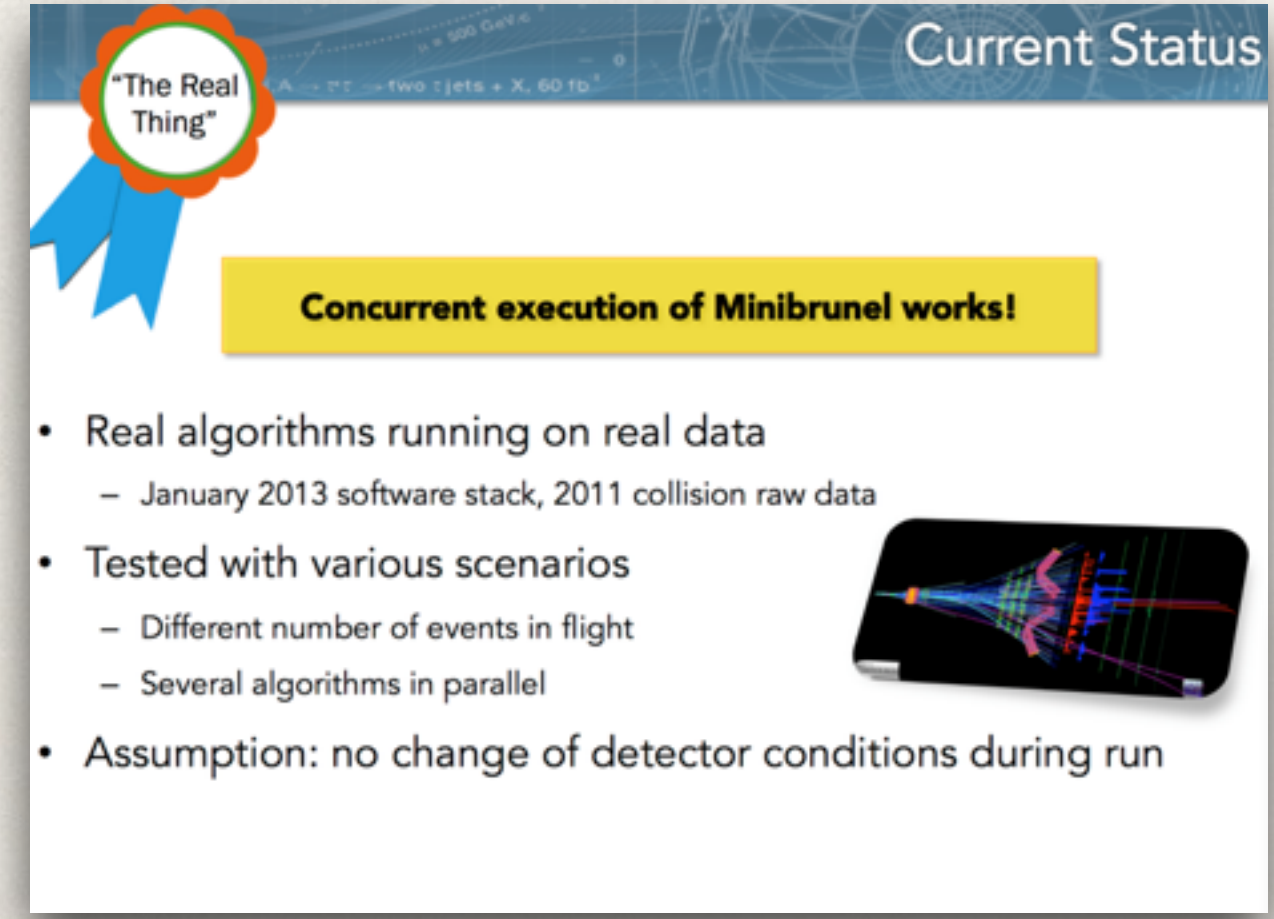
Test On Brunel Workflow

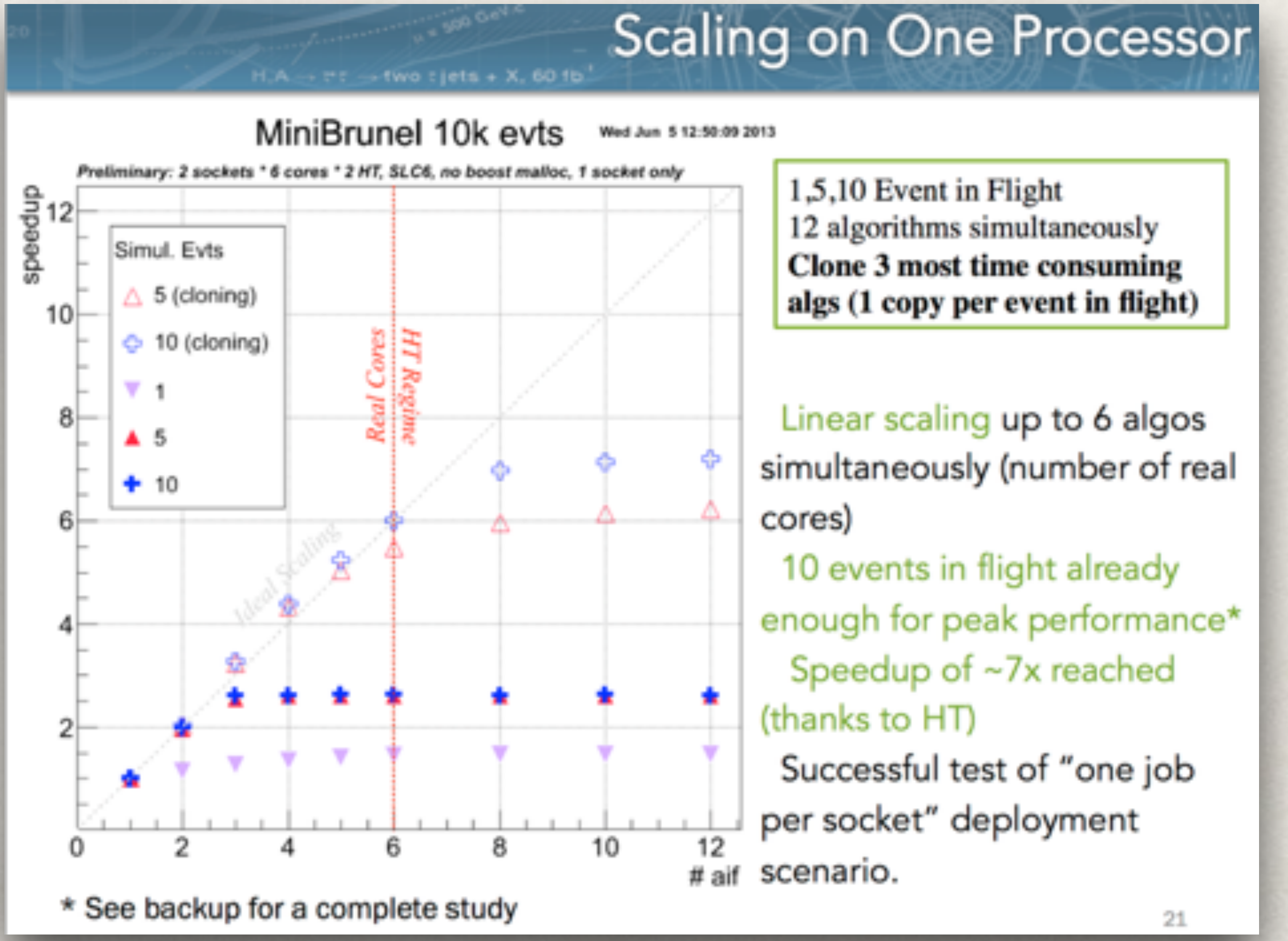


Test system with 12 physical cores x 2 hardware threads (HT)

- 214 Algorithms, real data dependencies, (average) real timing
  - Maximum speedup depends strongly 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

Mini-Brunel - May 2013





To be continued…

Main Messages

- The original design criteria still very valid
  - Data vs. Algorithms, different data representations, interfaces vs. implementations, types of data, etc.
- The original development strategy still valid
  - Use cases, architectural design, divide the work into components, etc.
  - Re-use existing libraries for implementations, release often, etc.
- Simple architecture with very simple design
  - Very few concepts that translates into few classes (components) have enabled us to add continuously new functionality that was not foreseen from the beginning (resilient design)
- Keep it simple (and as stupid as possible)
  - Hide complexity from 'physicists'
  - Not everybody masters all techniques (C++++, MT, etc.)

CERN