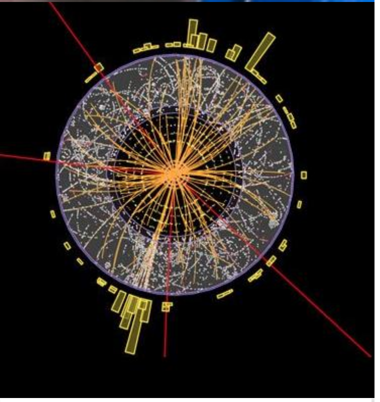
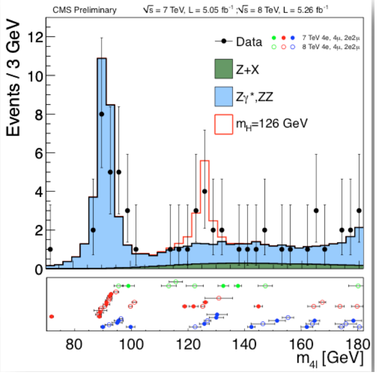
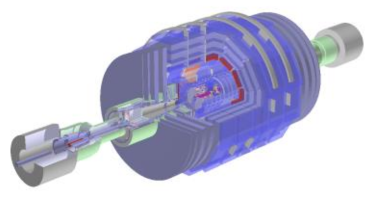


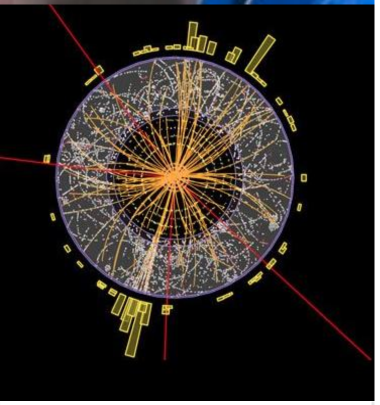


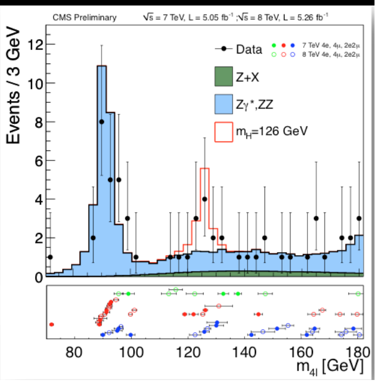


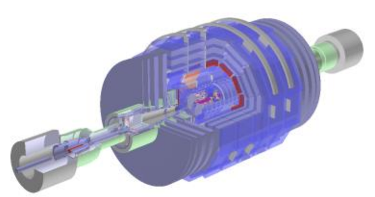
ROOT, I/O and Concurrency

Philippe Canal
Fermilab

Overview

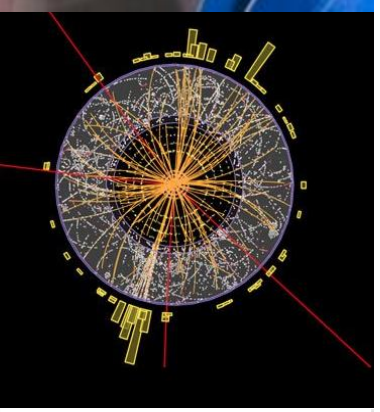


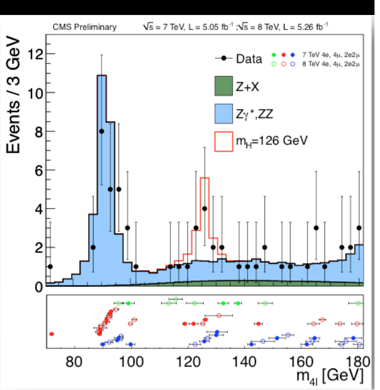


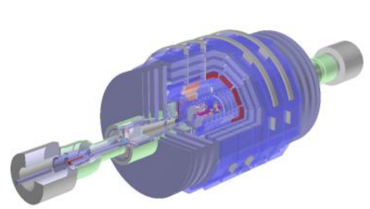


- ROOT
- I/O
- Concurrency

ROOT Guiding Principles

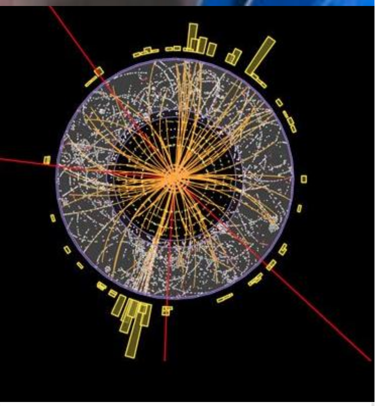


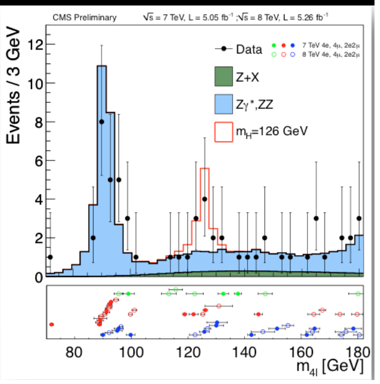


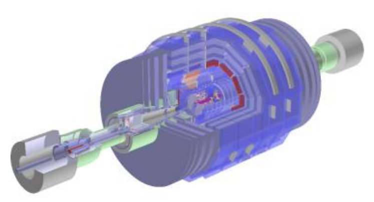


- User Oriented
  - Support and Feedback is ***essential***
  - HEP is main but not sole user/target
- C++: Interpreter and Reflection
- High performance (many dimensions)
- Release early and often
- Open-ended
  - Include *interfaces* to other languages
  - Help promote associated project (RooFit, etc.)

I/O Long Term Goals

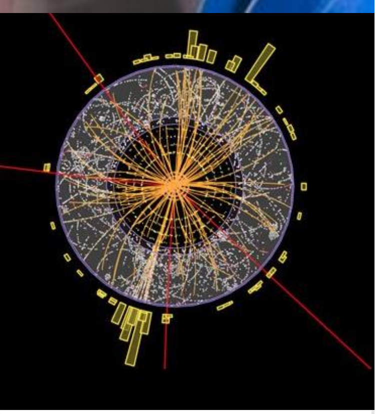


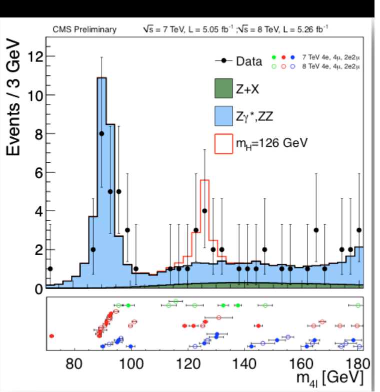


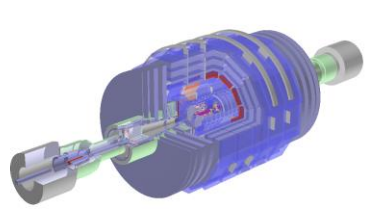


- Performance
  - Keep up / Pass competition
    - With StreamerInfo layer and JIT, large opportunities for optimizations
- Features
  - Maintain and enhance schema evolution support
  - Adapt to new hardware landscape
    - Today: multitask, vectorization ; Tomorrow: transactional memory.
- Interoperability
  - Open their ecosystem to using ROOT [I/O]
  - Open ROOT users to use the other ecosystem(s)

I/O Priorities







- Multi-processing / Multi-threading
- Performances improvements
  - Amdahl, File Format, Streaming, Vectorization
- Interface Simplification and Clarification
  - Leverage C++11 for ease of use/documentation
- Interoperability
  - HDF5, R, Python, Blaze, numpy, etc.
- Additional statistics and Feedback on I/O Perf.

Here comes cling

- ***Cling*** introduces binary compatible Just In Time compilation of script and code snippets.

- Will allow:
  - ***I/O*** for 'interpreted' classes
  - Runtime generation of ***CollectionProxy***
    - Dictionary ***no longer*** needed for collections! ***[Summer Student]***
  - Run-time compilation of ***I/O*** Customization rules
    - including those carried in ***ROOT*** file.
  - Derivation of 'interpreted' class from compiled class
    - In particular ***TObject***
  - Faster, smarter ***TTreeFormula***
  - Potential performance enhancement of ***I/O***
    - Optimize hotspot by generating/compiling new code on demand
  - Interface simplification thanks to full ***C++*** support
    - New, simpler TTree interface (***TTreeReader***) ***[Summer Contributor]***

Challenges

- Two distinct user bases
  - Individual Users
    - Want everything automatic / just works
  - Framework Developers
    - Want to control everything (want no surprise)
- Two distinct mode of operations
  - Thread based
  - Task based
- Must support all 4 combinations
- Concurrent access must not cost (too much) for non-threaded use.

User / Thread Example

- Simple merge histo interface.
  - User add ‘only’ the lines (*)

```
// Main Thread
TDirectory *merger = new THistoMerger(nthreads); // (*)
```

```
// Each thread init
merger->cd();  // (*)
TH1F* h = new THF1(“h”,…);

// Each thread event loop
h->Fill(value);
```

```
// Each thread init
merger->cd();  // (*)
TH1F* h = new THF1(“h”,…);

// Each thread event loop
h->Fill(value);
```

```
// Tear down or end
merger->Merge(); // (*)
ouputdir->cd();
merger->Write();
```

User / Task Example

- Simple merge histo interface.
  - User add 'only' the lines (*)

```
// Main initialization
TDirectory *merger = new THistoMerger(nthreads); // (*)
```

```
// Each task init
TH1F* h = new THF1("h",…);
h->SetDirectrory(merger);  // (*)

// Each task iteration
h->Fill(value);
```

```
// Each task init
TH1F* h = new THF1("h",…);
merger->Append(h);  // (*)

// Each task iteration
h->Fill(value);
```

```
// Final task
merger->Merge(); // (*)
ouputdir->cd();
merger->Write();
```

Framework example

- Framework want to owns histos.

```
// Each thread/task init
TH1F* h = new THF1(“h”,…);
fwk_ownlist->push_back(h);

// Each thread/task event loop
h->Fill(value);

// Tear down or end
foreach h in ownlist(s) (or equiv)
   TList *lst = complex_code_to_gather_histo(fwk_threadlist);
   merged = Merge(lst);
   outputdir->WriteObject(merged);
```

- But what about the case where there is 100,000 of histo?
  - Especially if filled rapidly (so need lock less Fill)

100,000 of histos on several threads

- Questions:
  - What is the use case really like?
    - What is the required performance ('where' can we put a lock)
  - When is the data merge done?
    - Every Fill
    - Every so many calls fills
      - One of the threads
      - A merger thread?
      - Why is it better that one histo per threads
  - Are TH* really heavy weight?
    - What is the real over-head?
    - When/how is the allocate-the-bins-only-when needed used?
  - Related concerns
    - Should variable size and fixed size bins histo be more clearly separated?
      - Improve performance, Reduce over head (of fixed size case)
      - Is it making the interface harder to use/explain?

Another interface idea.

-

```
// Main initialization
TH1F *mainh = new TH1F("h",….)
```

```
// Each task init
HistoTaskHandle h(mainh);
h->SetBufferSize(…);

// Each task iteration
h->Fill(value);
```

```
// Each task init
HistoTaskHandle h(mainh);
h->SetBufferSize(…);

// Each task iteration
h->Fill(value);
```

```
// Final task
foreach handle h:
   mainh->Add(h);
outputdir->Write(mainh);
```

Another interface idea.

- Spot the difference ☺

```
// Main initialization
TH1F *mainh = new TH1F("h",….)
```

```
// Each task init
TH1*h = mainh->Clone();
h->SetBufferSize(…);

// Each task iteration
h->Fill(value);
```

```
// Each task init
TH1*h = mainh->Clone();
h->SetBufferSize(…);

// Each task iteration
h->Fill(value);
```

```
// Final task
foreach handle h:
    lst->Add(h);
mainh->Merge(lst);
outputdir->Write(mainh);
```

Framework Requirements

- If ROOT uses threads that should be not computationally intensive
  - Example: prefetching thread

- If ROOT wants to run cpu-intensive tasks they must (be able to be forced to) request CPU time from the framework
  - For example a parallel unzipping would need to be a Task pushed on the TBB stack
  - Implies that when adding improvement that uses parallel execution we ought to follow a task model

Task vs. Thread

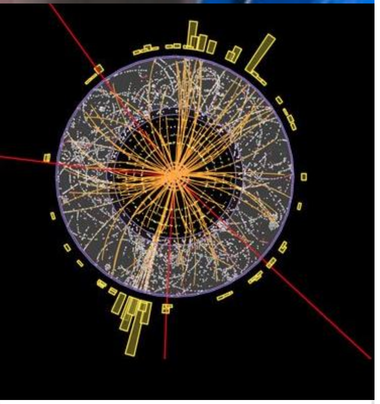
- Task model simpler to use
  - Delegate load balancing to ‘framework’
  - Similar to Proof
    - However task more ‘flexible’ control flow
    - Proof more extensive (over more hardware config)
  - Should we promote the task mode (tutorials, etc.) ?
- Requirements
  - Need (virtual) Interface
    - To allow replacing TBB with Apple GCD.
  - Need always available default implementation
    - Which is easily replaceable by the user controlled one.
  - Need to be pluggable/controllable by the user
- Other Utilities we could offer
  - Similar to boost::thread_specific smart pointer.

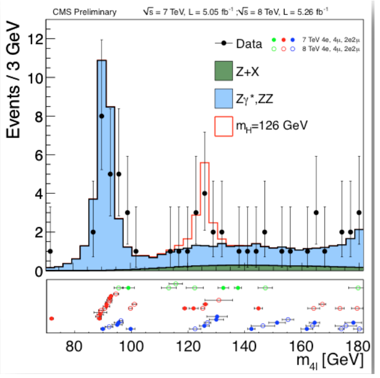
PROOF vs. Task Models

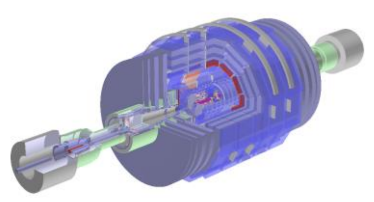
- What are the difference?
- i.e. can PROOF(lite-with-thread) be (extended to be) our interface?
- If not how do we provide a smooth experience
  - From single stream of operation
  - To many streams of operation
  - To many machine with many streams of operations
  - Without changing the code?

Vectorization

- Many alternative vectorization techniques
  - VC, VDT, Cuda, by hand, etc.
- GeantV uses template techniques and traits to steer the choice.
- Should we adopt the same techniques
  - and share/distribute the common parts?

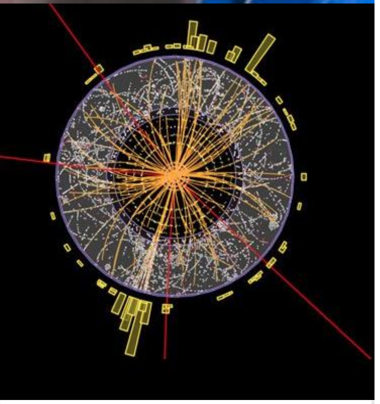
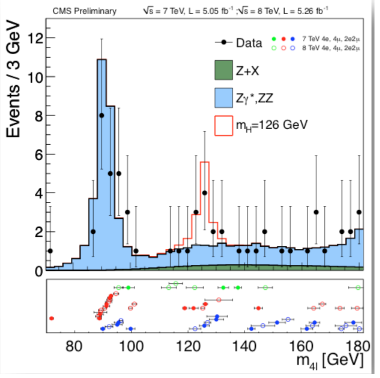


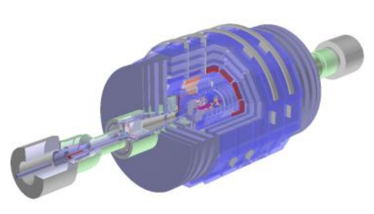




Backup slides

Priorities



- Multi-processing / Multi-threading

- Performances improvements
  - Amdahl, File Format, Streaming, Vectorization

- Interface Simplification and Clarification
  - Leverage ***C++11*** for ease of use/documentation

- Interoperability
  - ***HDF5, R, Python, Blaze, numpy***, etc.

- Additional statistics and Feedback on I/O Perf.

Multi-Processing

- Import Chris' changes to ***v5.34*** and port to ***v6.02***

- Extend the ability to disable auto-add
  - Limited to ***TH**** so far
  - Remove use of ***I/O*** in ***TH*::Clone***

- Resolve parallelism limitations
  - As shown in the **CMS** condition database example

Multi-Processing

- ***Histogram*** and multi-threading
  - Need to start prototyping & testing asap
  - New interface to incrementally merge histograms from multiple threads
- Read/Write ***TTree*** branches in multiple user thread
  - Need to start prototyping/testing asap
  - Do we need new/simpler interface?
  - Need to design the limit and semantics
  - Extra complexity/cost to conserve basket clustering
  - Require ***TFile*** synchronization

Thread Safety

- ***Cling*** enables support for robust multi-thread ***I/O***
  – ***Cling*** has clear separation of database engine and execution engine allowing to lock them independently

- Chris' changes allow multi-threaded ***I/O*** as long as
  – Each ***TFile*** and ***TTree*** objects are accessed by only one thread (or the user code is explicitly locking the access to them)
  – Interpreter is *not* the top level entry point.
  – ***Cling*** will allow to remove the second limitation.

- More has to be done to optimize
  – Some object layout leads to poor performance and poor scalability
  – Reduce number of 'class/version/checksum' searches
    - To reduce the number of atomic and thread local uses

Parallel Merge Challenges

- Need official daemon/thread ***parallelMergeServer***
  - Could use ***Zero MQ*** as underlying transport.
- Need to efficiently deal with many histograms
  - Each of them still need to be merged at the end
- Lack of ordering of the output of the workers
  - No enforcing of luminosity block boundaries for example
  - Support for ordering increases worker/server coupling
  - Space reservation is challenging (variable entry)

- Need a new concept (an ***Entry Block***)
  - 'Set of entries that are semantically related'
  - To be used to gather those entries together 'automatically'
  - Need flexible/customizable marker
  - Is it really worth the extra complexity?

Parallel merge

- Fully tested and performing version requires
  - Parallel Merge Thread
  - Parallel Merge Daemon (authorization, auto-start, error handling)
  - ***Parallel Merge for Histogram*** (proper set of benchmarks, performance improvement, etc.)

- Benchmarks
  - Still to be designed
  - Based on existing example (some multithread) and new example based of the ***Event*** test.
  - Based on experiment uses cases.

Other Possible Parallel Processing

- Read/Write branches using *internals* thread/tasks
  - Need to partially back out memory optimization
  - Require ***TFile*** synchronization
- Offload work (compression) to separate thread
  - Need to work well with task based scheduler
- Thread safe version of ***TFile***
  - Not quite sure of semantic
  - Need to be cost-neutral for traditional uses

- Support for ‘multiple’ interpreter state
  - Decide on need / interface / use limitations
  - shared libraries (their PCMs) shared between interpreters?

Vectorization

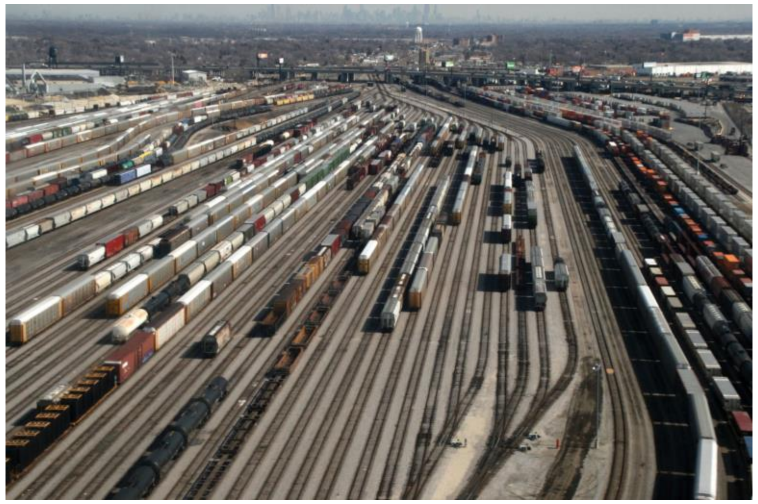
- In ***TTree***
  - Eg. ***TTree::Draw*** execute formula on more than one element at a time
  - New interface allowing retrieval of multiple entries at once.
- In Streaming
  - Changing endianess would also merging and vectorization of even more streaming actions.

Interoperability

- ***HDF5, R, Python, Blaze, numpy***, etc.
  - These ecosystems has their strengths and weaknesses as well some similarities and significant differences with ***ROOT***
  - What can we learn from them?
  - How can ***ROOT [I/O]*** can be leveraged to enhance them?
  - How could our workflows benefit from using directly or indirectly any part of these ecosystems?
  - Who can help?

Why one thread/schedule per TTree

- When reading TTree holds:
  – Static State:
    - List of branches, their types their data location on file.
  – Dynamic State:
    - Current entry number, ***TTreeCache*** buffer (per ***TTree***), User object ptr (one per (top level) branch), Decompressed basket (one per branch)
  – Separating both would decrease efficiency



- Advantages
  – Works now!
  – No need for locks or synchronization
  – Decoupling of the access patterns
- Disadvantages
  – Duplication of some data and some buffers.
    - However this is usually small compare to the dynamic state.
  – Duplication of work if access overlap