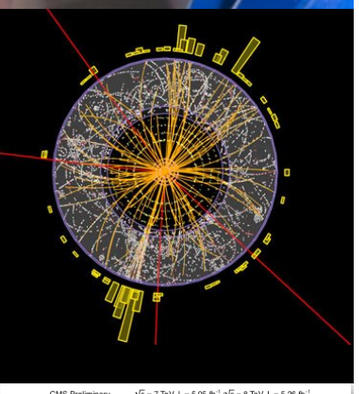
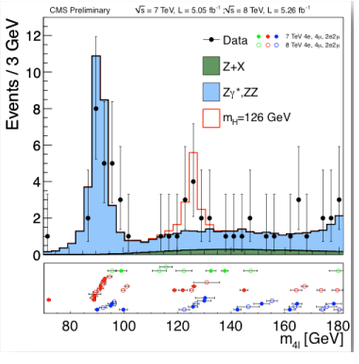
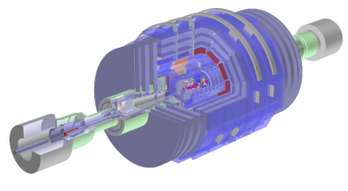


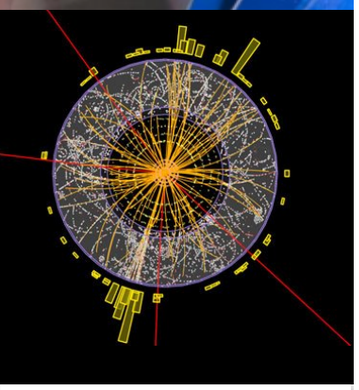


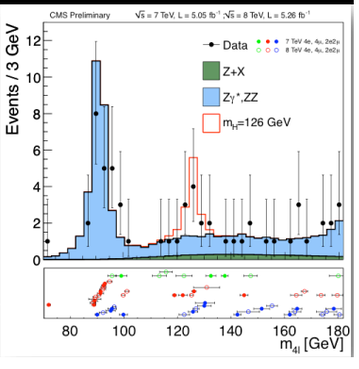


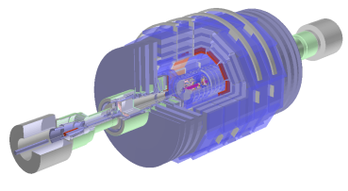
ROOT and Multi/Many Streams Of Execution

Philippe Canal, Fermilab
For the ROOT Team.

Overview







- Current Situation
- Challenges
- Future directions and outlook

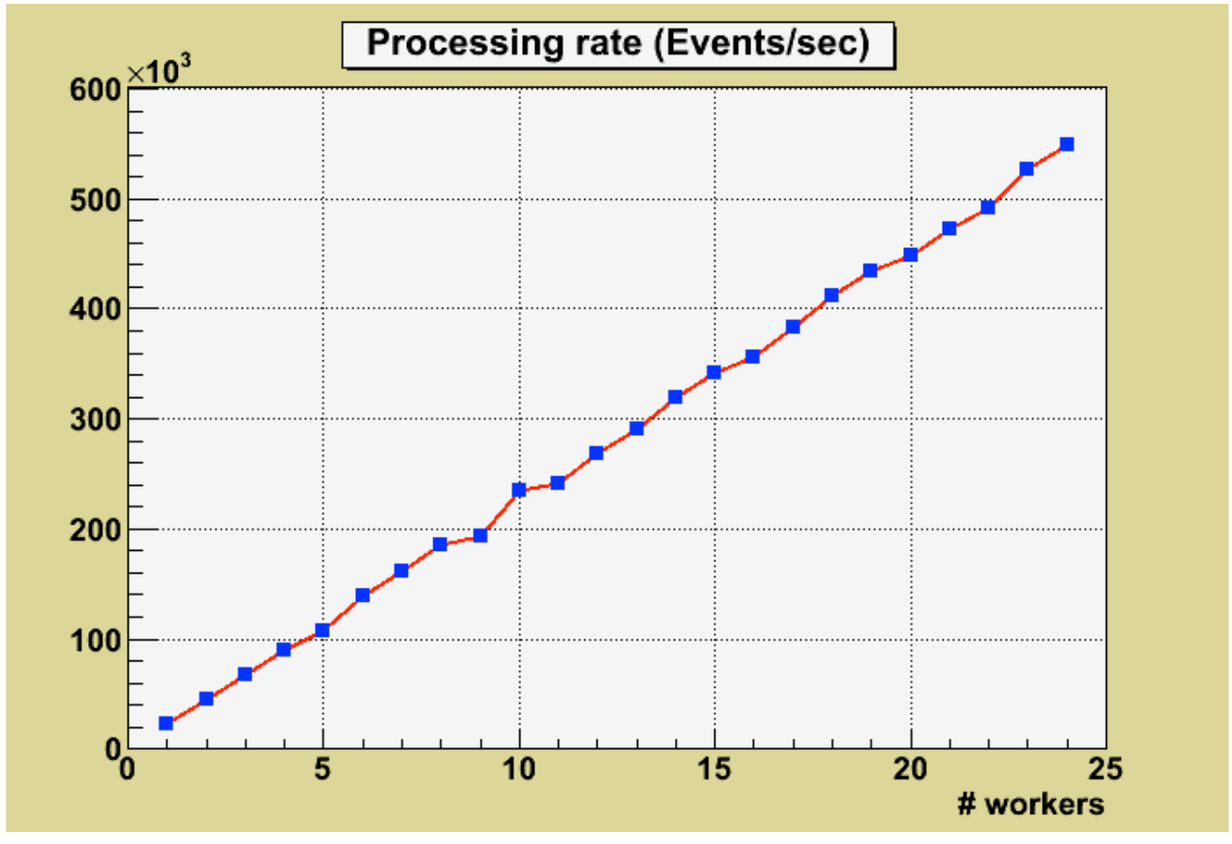
Current Situation

- Used both as:
  - Set of libraries within experiment framework
  - End-user tool via compiled C++, interpreted C++ or python.
- Parallelism via ***PROOF*** and ***PROOF-Lite***
  - Working model
  - Scales from multi cores to many nodes (including grid/ clouds)
  - Continues to improve (See Gerri's talk)
- Parallel file merge available via a separate process
- Threads used in a few places
  - eg. Read ahead for ***TTreeCache***
- User code
  - Could use TThread and ROOT I/O in limited/simple cases.

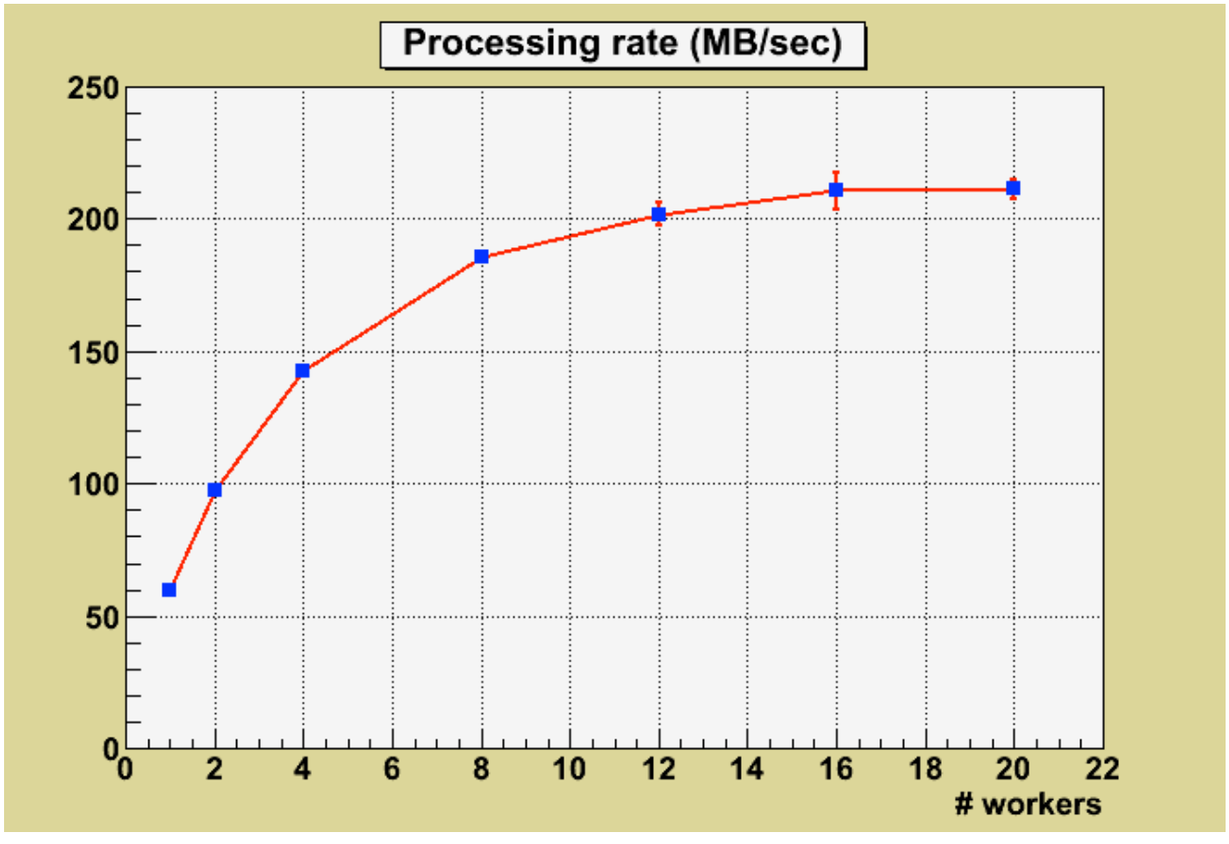
PROOF-Lite Performance

“I couldn't resist trying this. I just got myself the ROOT trunk, compiled it and tried your .C file. Indeed, there is zero configuration on my part and it ran on our 8-core mac pro (photo included)... Very impressive.” -- Akira Shibata, email.

"BTW, we just tried PROOF-lite on my 8-core Mac Pro with 14GB RAM and a fast RAID-0. We processed 3 million events in ~6 seconds, setting the highest processing rate of ATLAS analysis data I've seen..." -- Kyle Cranmer, email.



Perfect linear scaling processing events from cache (S. Panitkin, BNL).



Getting full performance from SSD (S. Panitkin, BNL).

Current Situations – Math Libraries

- Vectorization
  - ***Vc*** and ***VDT*** are now available in ***ROOT*** in both master and 5.34
  - ***Vc*** vector types useable with ***SMatrix*** and ***GenVector***
- ***RooFit/RooStats*** support multi-process.
  - Works with ***PROOF*** lite for toy/pseudo-experiments generation in ***RooStats***.
- ***OpenMP*** support in ***Minuit2***
- Fitting prototype (OpenLab / Vincenzo)
  - What matters is memory
  - Hardware layout should always be taken into account
  - Dynamic Task scheduling (of a DAG) is the most efficient solution for concurrency

Concurrent ROOT I/O Support

- Supported only in ***C++11*** in both v5.34/19+ and v6
  - Relies on ***std::atomics***
  - Cost of atomics (and ***thread_locals***) about 5% of streaming time (mitigated by ***C++11*** being 2% faster).
- In v5, limited to non-interactive sessions
  - High deadlock risk when starting the command line.
  - Fundamental limitation due to overlap of execution engine and database in ***CINT***

Concurrent ROOT I/O Support

- Multiple threads each accessing their own ***TFile*** and ***TTree***.
  - Creation and update of ***TClass*** and ***TStreamerInfo*** is protected.

- Access to same ***TFile/TTree*** object from multiple thread requires explicit lock.

- ***gDirectory/gFile/gPad*** are thread local
  - Does not match with the task models (eg. ***TBB***)
    - i.e. user need to explicitly set them at every task (re)start or avoid any code that relies on those globals

What did it take?

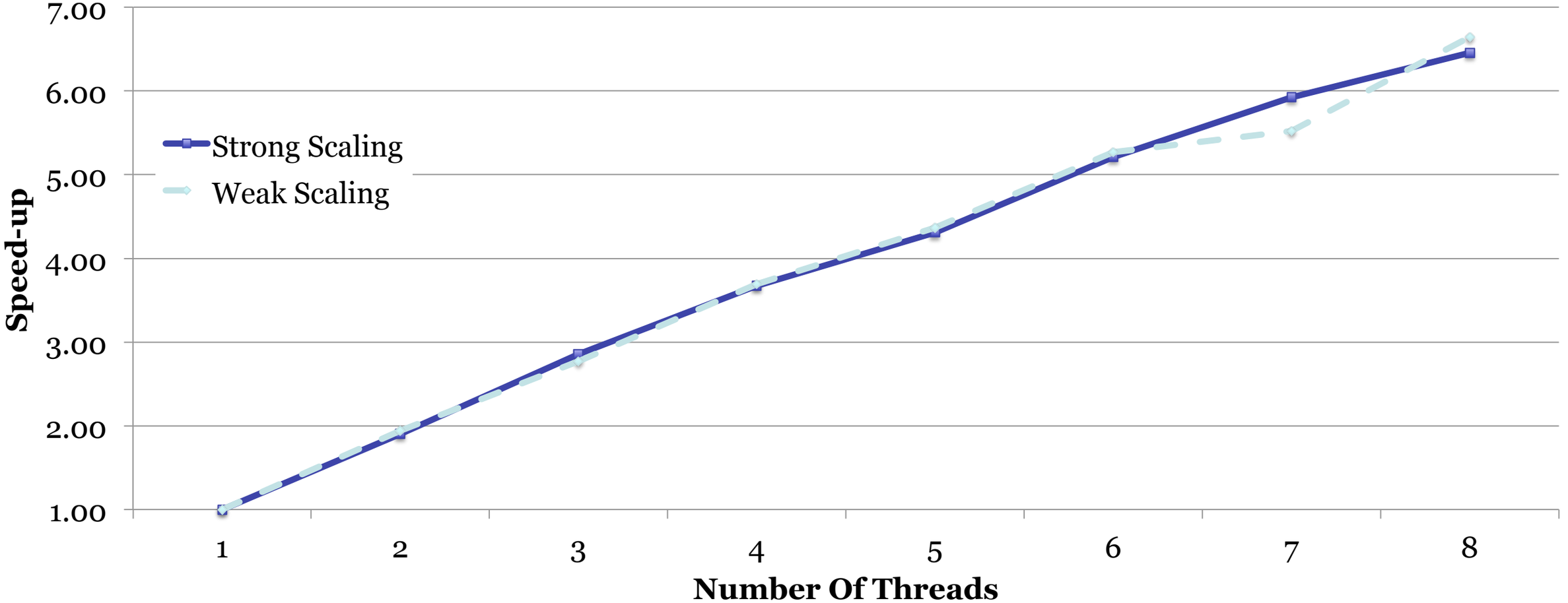
- Algorithmic changes
  - ***TObject*** recording if instance on stack or heap
  - Avoid rebuilding ***StreamerInfos***
  - Avoid resetting values
- ***C++11 thread_local***
  - Used for globals in ROOT which hold temporary state for a callstack
- ***C++11 std::atomic<>***
  - Used for global variables used to assign unique IDs
  - Used for member data which are caches
- Adding more mutex locks
  - Originally incomplete coverage of mutex in ***ROOT*** code
- Avoid deadlock
  - Merge ***gROOT*** and ***CINT*** mutex

Reduction Of Sequential Parts

- Update ***TClassRef***
  - From a linked list of ref per TClass object update at each creation/deletion of ***TClassRef***
  - To a single long lived pointer per ***TClass*** object shared by the ***TClassRef*** objects.
- ***TClass::GetClass***
  - Move code around to reduce length the lock is held.
- ***TClass::Get/FindStreamerInfo***
  - Remove use of lock in the common case by caching in an atomic the most recently found for each ***TClass***
- ***TThread::Self***
  - Remove linear search doing string comparison by using thread local storage.
- Remove locks in ***TBaseClass*** by caching information.

CMS Event

- ***CMS*** Event file 133MB, 100 entries.
- One ***TFile*** and ***TTree*** per thread.
- Use ***TTreeCache*** and slightly modified ***MakeProject*** lib.
- Less than 5% sequential

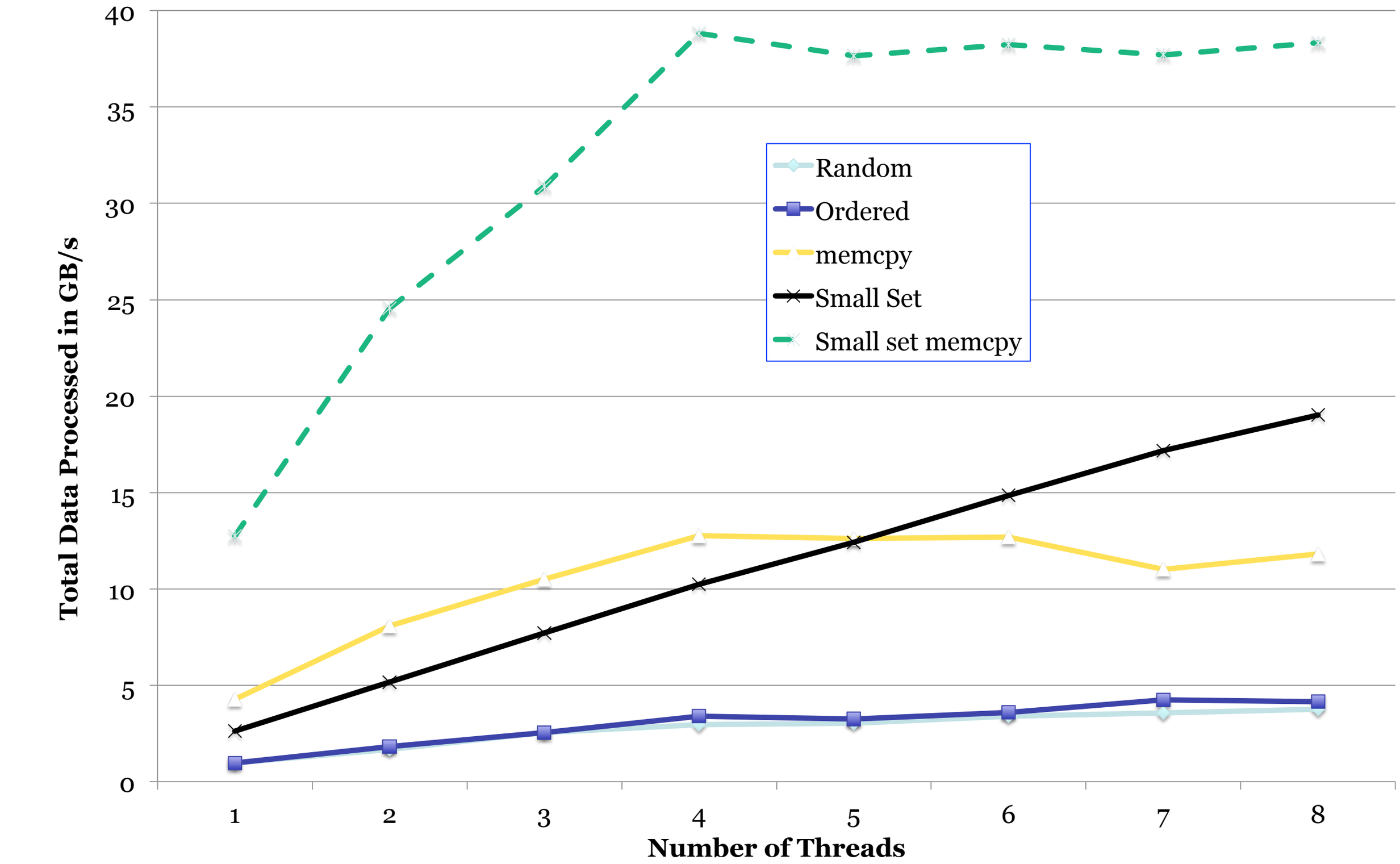


CMS Condition Test Description

- 226 distincts ***CMS*** condition database objects
- 233MB of data

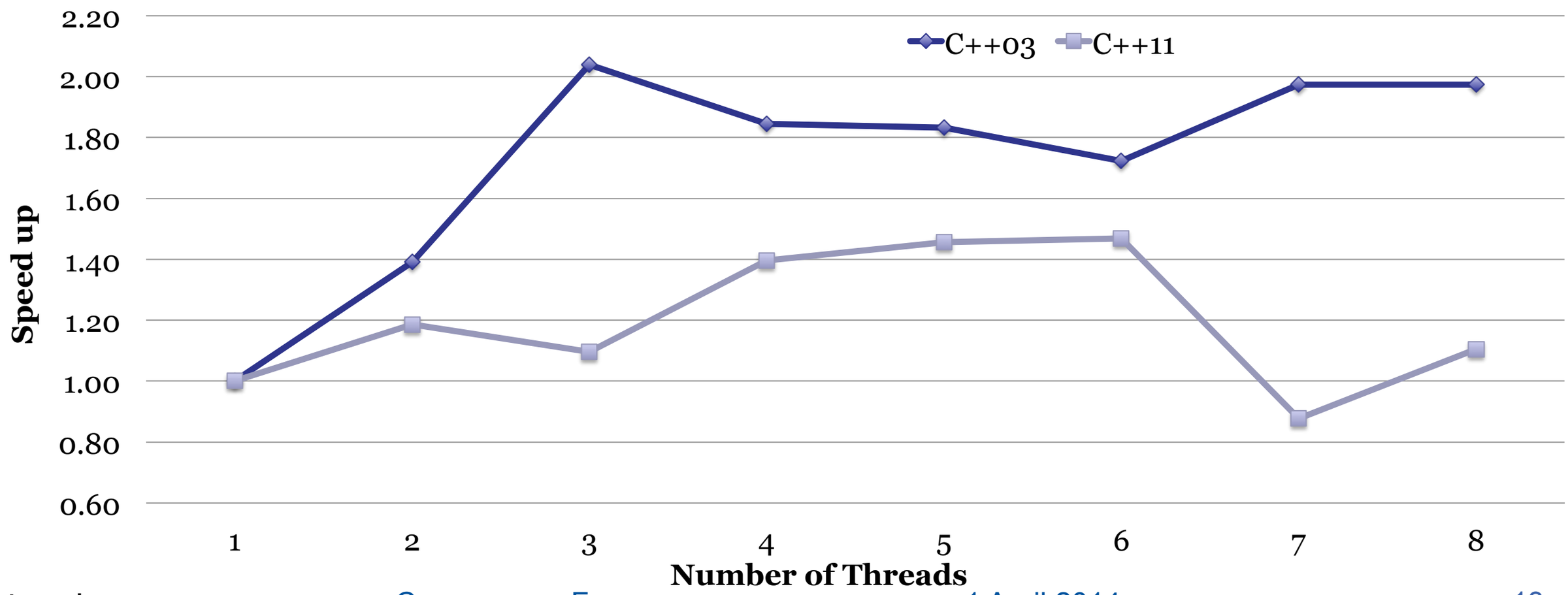
- Run example with no thread enabled then 1 through 8 threads.
- Load the data (amount varies) into 1 ***TBufferFile*** per thread.
  - Small Set: first 3 objects for 393KB.
- Each thread deserialize the content multiple times

Data Bandwidth



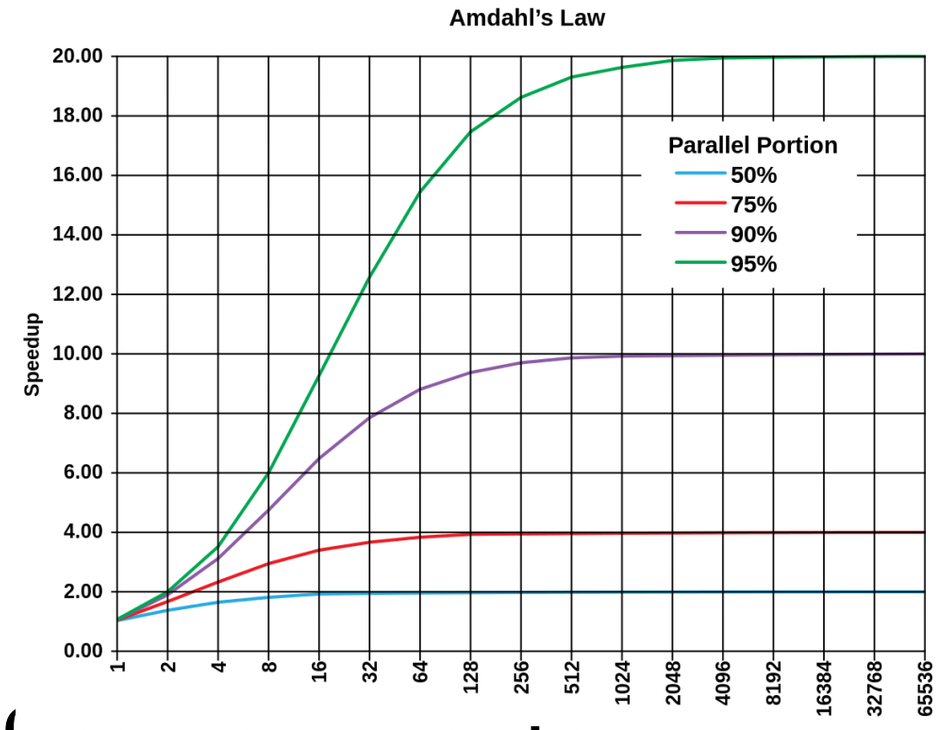
Worst case.

- Class ***L1RPCConfig***
  - 22% of time, 3% of space
- Most time consuming object.
- Contains ***vector*** of 93160 objects
  - which contains an array of 6 objects.
  - Each of those contains 2 bytes!
- Large Sequential Part
  - Atomics and lock play a role but not enough to explain behavior
- But no clear explanation
  - Maybe try running in ***VTune***



ROOT I/O

- ROOT I/O is now thread friendly
  - It works!
  - Less than 5% sequential exec when reading CMS Events TTree.
  - Less than 15% sequential exec when reading all CMS cond db objects.



- Some object layout lead to poor performance and poor scalability.
- More has to be done to optimize
  - Reduce number of 'class/version/checksum' searches.
    - To reduce the number of atomic and thread local uses.
  - Change byte swap order (increase memcpy case)
  - Continue refactoring of the I/O internals
    - Increase vectorization, reduce branches, etc.

Concurrency Challenges

- Global Database
  - ***TClass***, Interpreter (at least part of)
- Global State
  - ***gDirectory***, ***gFile***, ***gPad***, ***gRandom***, etc.
  - List of Files and other resources
    - Use as global registry for cleanup, shortcuts and for GUI
  - List of Cleanups
    - Shared ownership mechanism for both ***C++*** code, GUI and interpreter.
  - Made (in most cases) thread local
    - Not adapted for use in Task based model
  - Integral part of some interfaces
    - ***TObject::Write*** ; Object auto-addition to a ***TFile***.
  - Some behavior not yet optional
    - ***TTree*** auto-addition
- Other challenges
  - Access to hardware resources, merging (histo, tree), caching

Path Forward

- Enabling threads in ***V6***

- Benefiting from threads

- Brainstorming Future Interfaces

Enabling threads in *V6*

- Port latest updates to ***Version 6***
- Reduce lock contention thanks to clear separation of execution from database access
  - Eliminate dead lock risk when starting the command line
- Implement ***TThread*** using ***C++11*** threads
- Clarify which interfaces are ‘thread-safe’, side effects and reliance on global state
- Updates For Random Numbers:
  - Add ***Random123***
    - Fast initialization and small state
  - Add ***MixMax***
    - Mathematical proof randomness
    - Guaranteed different sequences if input seeds are different

Benefiting From Threads

- Design/Decide on multi level interface(s)
  - Supporting multi-process, multi-thread and task concurrency as seamlessly as possible.
  - For Example:
    - How to schedule in coordination with user framework
    - Resolve current asymmetry between typical ***TSelector*** and those useable with ***PROOF***
- Parallelism in ***TTree*** filling and reading
  - Allow multiple threads to use the same ***Ttree***
  - Unstreaming objects in parallel
    - Could be parallelized at both ***TTree*** and ***TDirectory*** level
    - Also for writing
  - Unzipping
    - See proposals by *Andreas Peters* and *Håkan Johansson*
  - Need to fit with task based paradigms
- Parallelism for Histograms
  - Balance memory increase with throughput increase
  - 'Live' merging

Benefiting From parallelism

- Random Number, ***Math***, ***RooFit***, ***TMVA***, etc.
  - eg. decide on how to handle ***gRandom***.
  - Explore and decide on where (and how) to use threads.

- Vectorization in ***Math***, ***I/O*** and ***TTree***
  - Eg. ***TTree::Draw*** execute formula on more than one element at a time
  - Vectorization and parallelization for fitting in ***ROOT***.
    - See results of prototype presented in previous meetings

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

Brainstorming Future Interfaces

- Many interfaces can be improved in ***C++11***
  - Ownership, type safe containers, string options
  - Resulting in improved user productivity
    - Dramatically reduce memory errors, wrong results, etc.

- Conflicting goals
  - Significantly improve interface
  - Keep backward compatibility for existing code

- Large existing code base relied upon in production across sciences and continents
  - **Must be backward compatible and reuse code base**
  - **Must evolve the current interfaces**
  - **Both can be done in a backward compatible way**

"Things alter for the worse spontaneously,
if they be not altered for the better designedly."
– *Francis Bacon*

Brainstorming Future Interfaces

- Lesson learned in industry:
  - deprecation does not work (***Google, Apple, etc.***)
  - but interface versioning *does* work: ***Windows***, ***Javascript***, ***libc++***,…
- Challenge
  - reduce duplication by making old interfaces use new implementations
- One example of a possible solution

```
namespace ROOT {
  namespace v6 {
    class TFile { current interface };  // ROOT::v6::TFile
  }
  inline namespace v7 {
    class TFile { better interface };   // ROOT::TFile
  }
}
// If backward compatibility is needed/wanted
using namespace ROOT::v6;                //  TFile <==> ROOT::v6::TFile
```

Brainstorming Future Interfaces

- So Yes, we can reinvigorate ***ROOT***'s core
- Main Goals:
  - Simplicity
  - Robustness
  - Performance
    - Embrace multi-tasking and vectorization
    - Pay for only what is used
  - Provide best/better features

Brainstorming Future Interfaces

- Some possible examples:
  - Type safe interfaces: no more casting
  - No globals, minimal static caching, const == thread safe
  - From:

```
OwnOrNot(TWhatever* arg);
```

  - To:

```
OwnOrNot(std::unique_ptr<TWhatever> arg);
OwnOrNot(&myWhatever); // Compilation error!
```

  - Conscious inlining e.g. for vectorization
  - Improve data structure for vectorization
  - Revisit/Redesign all functions in ***ROOT/Meta*** in view of ***cling***
  - Further simplify and reduce dictionaries

Conclusion

- Leveraging multi cores (since a long time) via ***PROOF/PROOF-Lite***

- and now ***ROOT I/O*** and core/meta fully multi-thread
  - (but not lock free)

- Ambitious path to update ***ROOT*** for tomorrow's need
  - Update interfaces reflecting/solving usage problems
  - Use current ***C++***, code style and patterns
  - Reduce need for locks/atomics etc
  - **Improve performance**
  - Extend use of vectorization