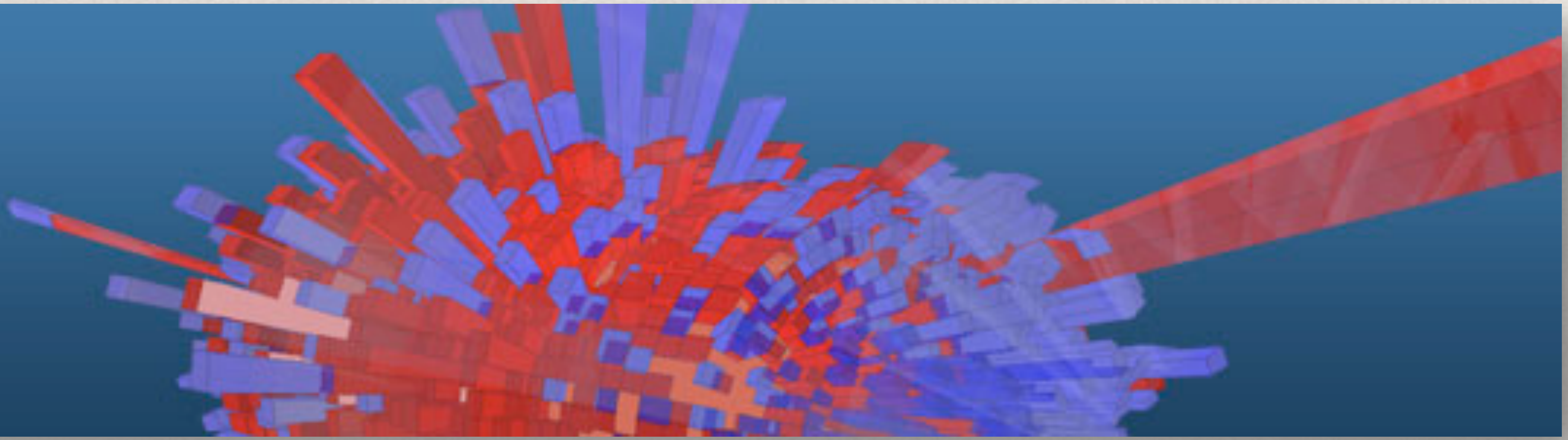
ROOT
Data Analysis Framework



Quest for Performance in ROOT

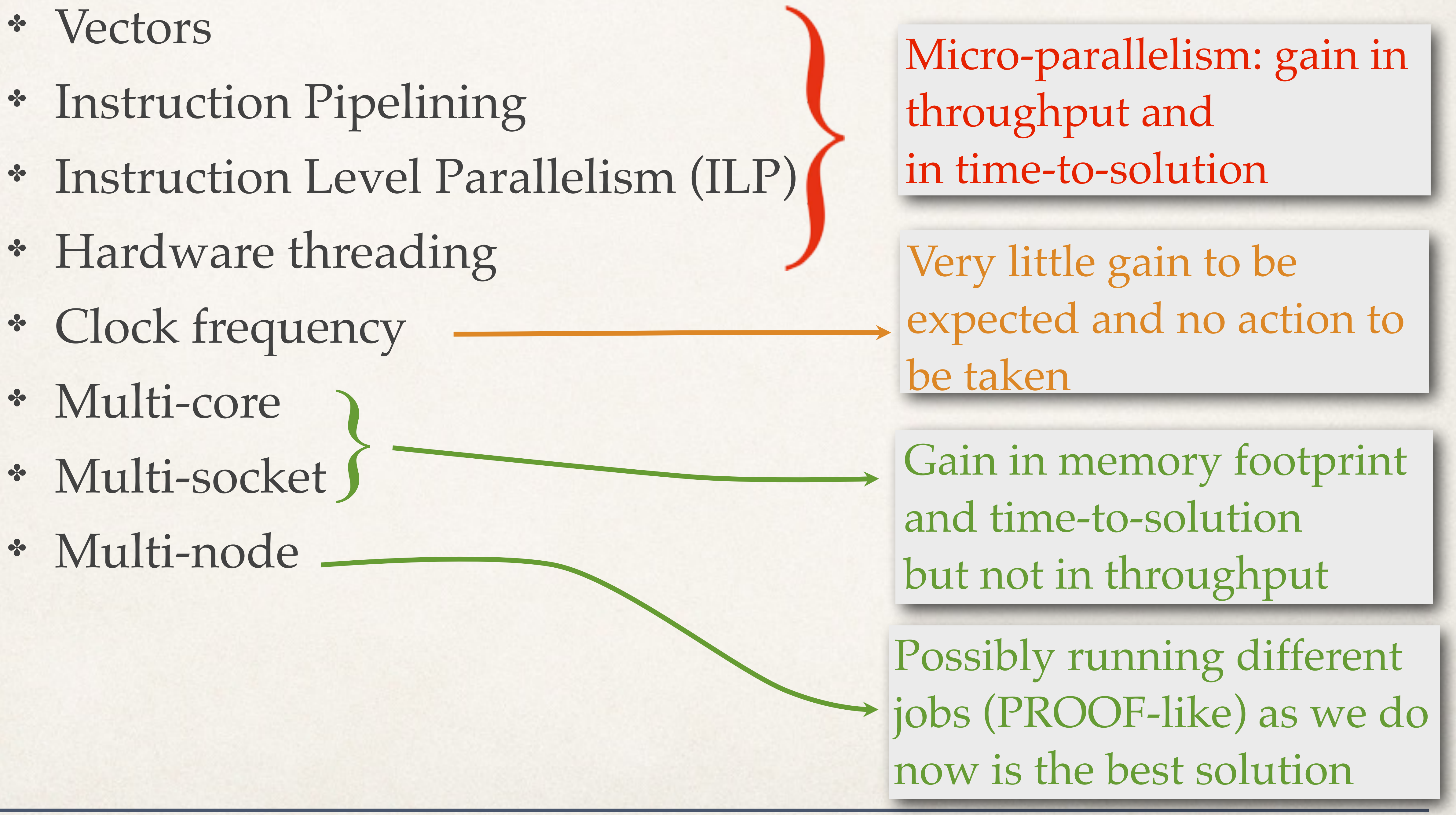
HSF Workshop, LAL, Orsay 2-4 May 2016
Pere Mato/CERN on behalf of the ROOT Team

Outline

- The 8 Dimensions of Performance
- Main Challenges
- Performance Monitoring
- Exploiting Vectorization
- Exploiting Multi-processing
- Exploiting Multi-Threading
- Exploiting Multi-Node
- Code optimizations: modules, modern C++, JIT, etc.
- Conclusions

The 8 Dimensions of Performance

- The “dimensions of performance” also for ROOT



  - Vectors
  - Instruction Pipelining
  - Instruction Level Parallelism (ILP)
  - Hardware threading
  - Clock frequency
  - Multi-core
  - Multi-socket
  - Multi-node

Micro-parallelism: gain in throughput and in time-to-solution

Very little gain to be expected and no action to be taken

Gain in memory footprint and time-to-solution but not in throughput

Possibly running different jobs (PROOF-like) as we do now is the best solution

ROOT Main Challenges

- ROOT is 20 years old, and some parts **require re-engineering and modernization**
    - Need to **exploit modern hardware** (many-core, GPU, etc.) to boost performance
    - **Modernize implementations** (C++11/14 constructs, use existing libraries, etc.)
    - **Modernize C++ API** to improve robustness eventually giving up on backward/forward compatibility
- Require the collaboration of the community to **ensure evolution and sustainability**
    - **Facilitate contributions** to ROOT without engaging our responsibility in the maintenance and user support
    - Layered software modules or plugins that can bring new functionality to the end-users

Improving Performance of ROOT

- ROOT will evolved and undertake the necessary changes to **improve performance in any area that will be possible**
- The strategy is to be transparent to the user whenever possible
  - Deliver more computation with the same or even simpler user interface
  - Hide complexity as much as possible
- Ambitious program for both making parallelism "endemic" within ROOT itself and for providing components to help users expressing parallelism
- The roadmap foresees the utilisation of different parallelisation strategies for different problems, qualified by their scale, amount of legacy code, affordable overhead

Measuring Performance

- Improvement is not possible without measurement
- ROOT could benefit from common profiling strategies and tools shared with the experiments.
  - The experience with IgProf has been shown to be a perfect example of the sharing of such strategies and tools achieving concrete and numerous results during the integration of ROOT6 with CMS and ATLAS.
- A continuous profiling effort of experiments' data processing “candles" is immediately beneficial for all building blocks of the experiments' software stacks:
  - Time and memory allocated per function
  - Possibility to inspect memory allocation patterns
  - Impact of serial code

Data Parallelism - Vectorization

- Exploit the SIMD instructions available in modern microprocessors
- Targeting mainly the Math Libraries in ROOT
  - Function evaluation with vector interface (Vc)
  - Likelihood calculations, fitting, etc.
  - Histogramming
- Use types in the *VecCore* library, which embeds low-level support for vectorization, in the ROOT Liner Algebra classes
- Re-implement the geometry package TGeom in terms of the *VecGeom* library developed in the context of GeantV

Multi-Processing

- Seamless parallelisation (no contention issues) of legacy code
  - Unique case in C++ landscape thanks to serialisation capabilities (even without dictionaries thanks to CLING)
- **PROOF** is playing an important role in speedup analysis. Several flavors:
  - PROOF-Lite (optimized for single many-core machines)
  - Dedicated PROOF Analysis Facilities (multi-user)
  - PROOF on Demand (single-user)
- New **TProcPool** utility, analogies with modern data science, map/reduce, integration with TTree and ROOT ecosystem.

MultiProc package

- Developed a new lightweight framework for multi-process applications
  - Inspired by the Python *multiprocessing* module
  - Idea to **re-implement Proof-Lite** using it
- Distribute work to a number of fork()'d *workers,* then collect results
  - Main advantage: workers have access to complete 'master' state

defaults to # of cores

C/C++ function
loaded macro
std::function
lambda

std::container
initializer list
TCollection&
unsigned N

std::vector or TObjArray

```
TProcPool pool(8)

auto result = pool.Map(fun, args);

auto result = pool.MapReduce(fun, args, redfun);
```

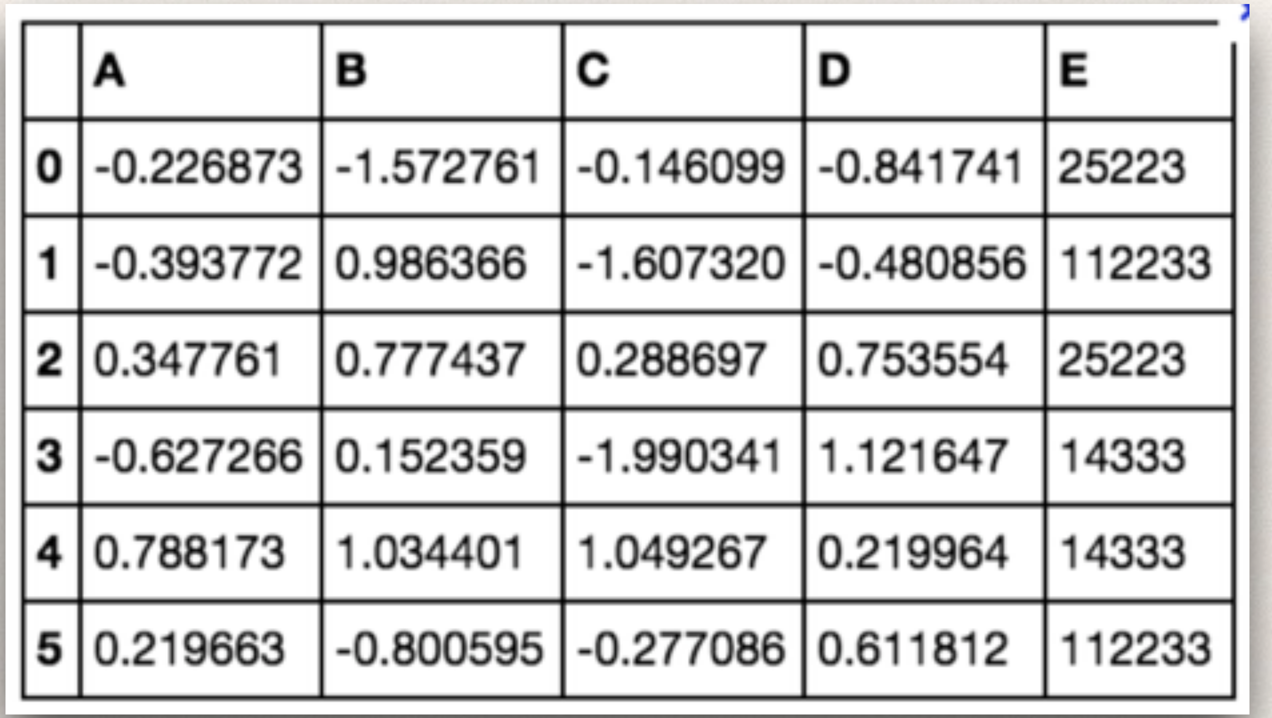
Reduce function

Multi-Threading

- Complete abstraction from the ROOT threading model (*TThread*) and ROOT seamlessly pluggable with arbitrary threading models (e.g. TBB, HPX, STL)
- New *ThreadPool* class offering same interface of *TProcPool*
- Solve problems for **merging efficiently the output objects** produced by the parallel tasks: (histograms, trees, etc….)
  - Entities to facilitate resource protection and handling of object merging (*TThreadedObject*)
- Introduce **thread-safety** where needed (e.g. I/O)
- **Implicit multithreading** for parallel TTree reading palette of examples illustrating new features, application for statistics, likelihood calculations, minimisations, generation of toy experiments.

Multi-Threading: TTree

- Every TTree is different, so it is every read operation:
  - E.g. trees with many branches, reading systematically all of them (exp. frameworks), or trees with many entries and reading only a subset of branches (analysis)
- No solution will fit all cases



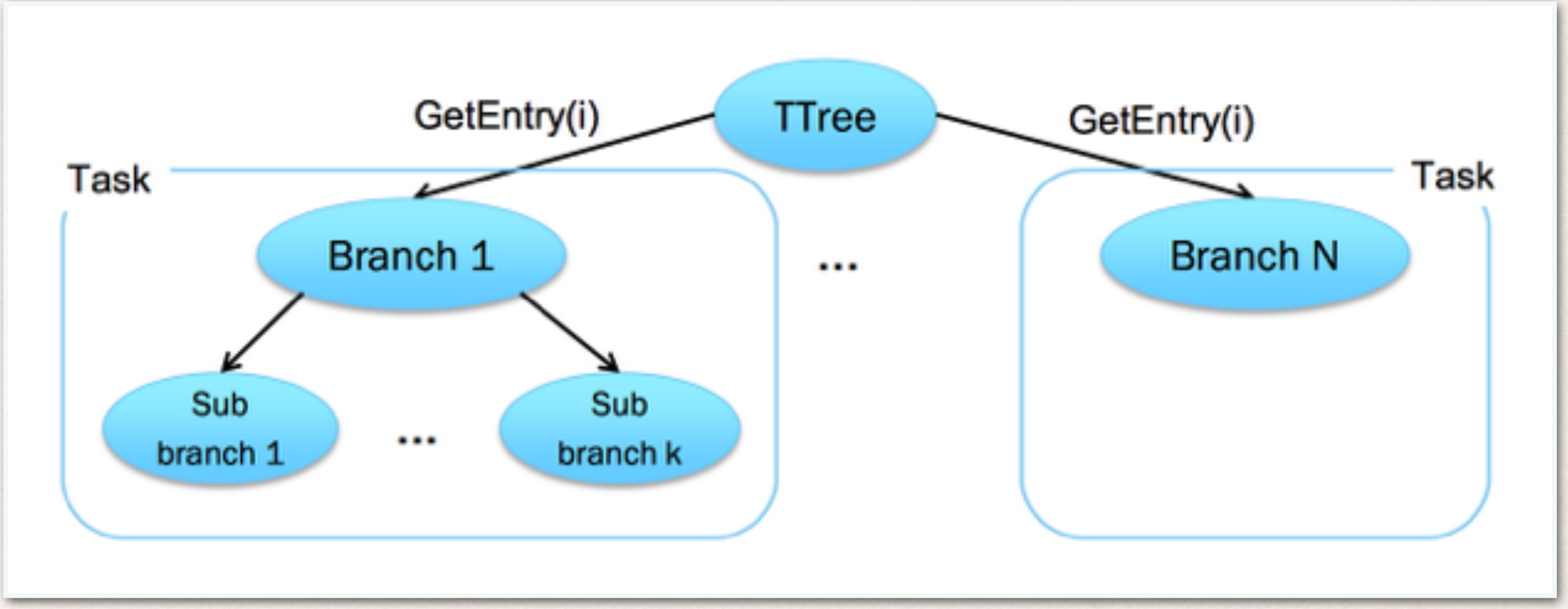
	A	B	C	D	E
0	-0.226873	-1.572761	-0.146099	-0.841741	25223
1	-0.393772	0.986366	-1.607320	-0.480856	112233
2	0.347761	0.777437	0.288697	0.753554	25223
3	-0.627266	0.152359	-1.990341	1.121647	14333
4	0.788173	1.034401	1.049267	0.219964	14333
5	0.219663	-0.800595	-0.277086	0.611812	112233

entry ranges

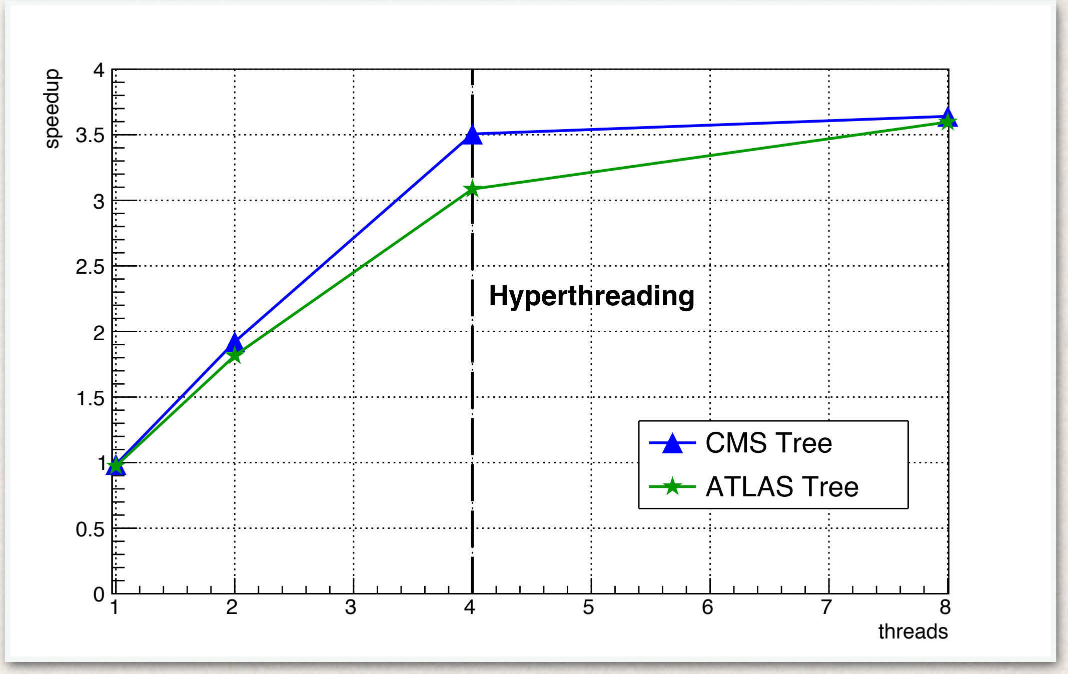
parallel_for

branches

Loop Over Branches

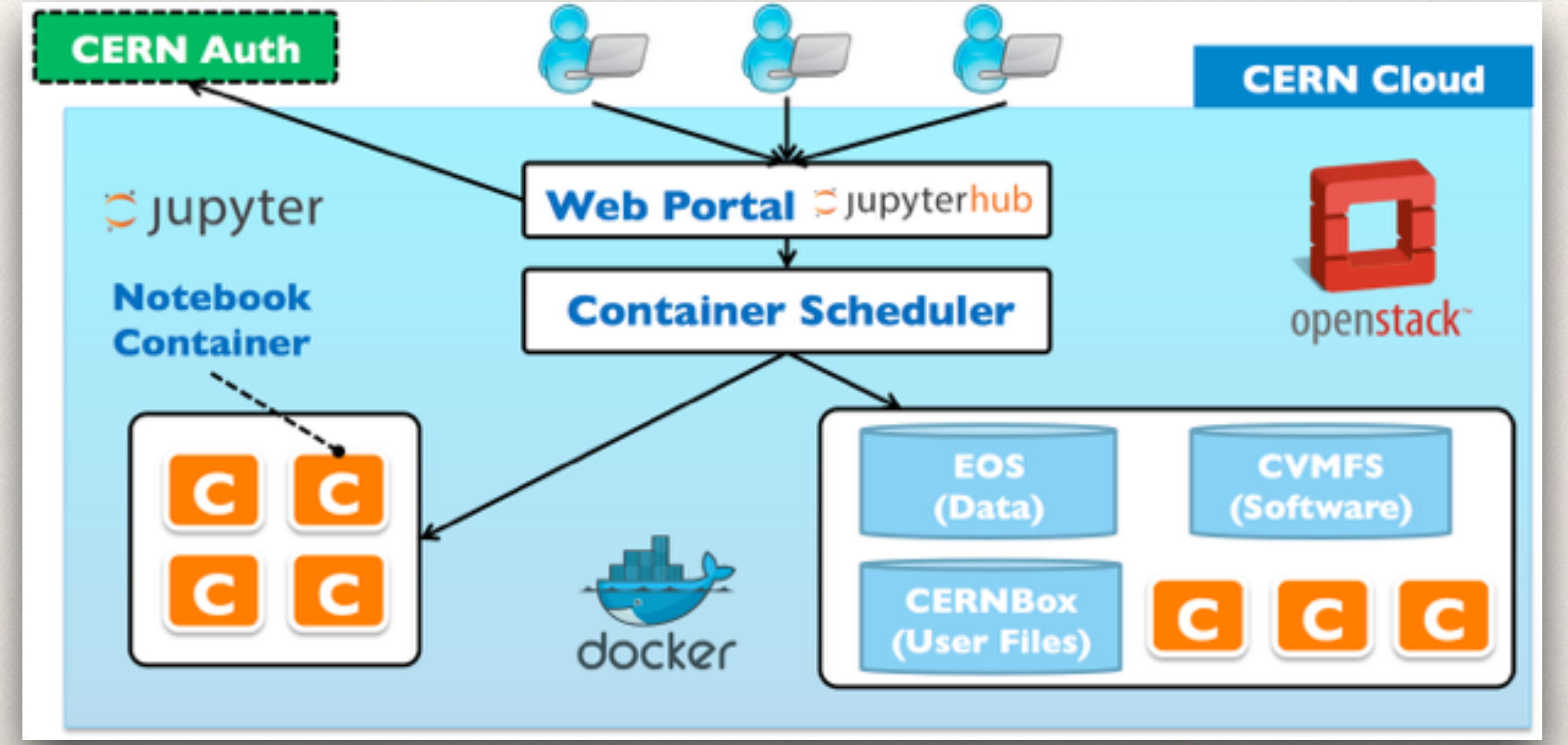


- Implemented parallel TTree reading using a “task programming model” (e.g. TBB)
  - speeding up the TTree:GetEntry(i)
- Concurrent de-serialization, decompression of each branch
- Tested with realistic ATLAS/CMS data



Multi-Node: SWAN

- A platform to perform interactive data analysis in the cloud (ROOT-as-a-Service)
  - Analyse data without the need to install any software
  - Front-end for on-demand/elastic cloud resources (e.g. Spark cluster, PROOF cluster, etc.)



Code Optimization

- Interpreter and *type system* optimization (CPU and memory)
  - Introducing CLANG modules to avoid header compilation
- I/O Performance
  - Code implementation optimization
  - Optimization via change in file format (endianness, memory layout, binary POD, etc.)
- New C++ Interfaces: an opportunity for improving performance
  - Modernize implementations with STL
  - Opportunity to replace 'virtual functions' with 'concrete' template instantiations and specializations
  - Prototype of ROOT 7 histograms with much better performance

Exploiting JIT

- Using the new JIT capability of LLVM/CLANG is a great opportunity for performance optimizations
  - Specific code can generated and executed at compiled (optimized) code speed
- Example: New *TFormula*
  - Pre-parsing of expressions (as before) but now using a real compiler
- Other examples: I/O proxies, TTree readers, etc.

```
auto f = new TFormula(“F”,”[0]+[1]*x”);

auto f = new TFormula("F","[](double *x, double *p)
                      {return p[0]+p[1]*x[0];}”,1,2);
```

ROOT
Data Analysis Framework

Functional Chains

- Prototyping some ideas of ‘declarative/functional’ chains of basic concepts such as map, filter, reduce, accumulate, etc.
  - Inspired from data analytic tools such as Spark
- The user specifies the **What** and system chooses **How**
  - Actions are only triggered at the end of the chain
  - Great opportunity for optimizations (partitioning, caching, re-ordering, etc.)

```
hist = ttree.filter(lambda event: event.Emiss > 40)
           .flatMap(lambda event: event.tracks)
           .map(lambda track: sqrt(px**2 + py**2))
           .histo(100, 0, 20)
```

The chain is only executed when is completed

Conclusions

- ROOT will evolved and undertake the necessary changes to improve performance in any area that will be possible
- Following several development lines
  - Multi-dimensional, Multi-domains (math, fitting, geometry, I/O, analysis, type system, etc.)
  - Not a single solution will fit all cases
- Enabling implicit multi-threading
  - Transparent to the user
- Prototyping new ideas with the goal to improve performance on making use of clusters, functional chains, ROOT-as-a-Service, etc.