

Expressing Parallelism with ROOT

<https://root.cern>

D. Piparo (CERN) for the ROOT team

DIANA Meeting 14-11-2016





ROOT helps scientists to express parallelism

- Adopting **multi-threading** (MT) and **multi-processing** (MP) approaches
- Following explicit and implicit paradigms
 - **Explicit:** give users the control on the parallelism's expression
 - **Implicit:** offer users high level interfaces, deal with parallelism internally

All available in
ROOT 6.08 !

- General purpose parallel executors
- Implicit parallelism and processing of datasets
- Explicit parallelism and protection of resources
- R&Ds: functional chains and ROOT-Spark



General Purpose Parallel Executors



Parallel Executors

- **ROOT::TProcessExecutor** and **ROOT::TThreadExecutor**
 - Same interface: **ROOT::TExecutor**
 - Inspired by Python's *concurrent.futures.Executor*
- **Map, Reduce, MapReduce** patterns available



```
ROOT::TProcessExecutor pe(Nworkers);  
auto myNewColl = pe.Map(myLambda, myColl);
```



A Word about the Runtime

- Multiprocessing: ROOT provides its own utilities
- Threading: adopted **Threading Building Blocks (TBB)**
 - **Optional** installation, necessary for multithreading
 - Task based parallelism
 - **Build system builds and installs it** if requested and not available
 - **Interfaces not exposed directly**
 - Complement with other runtimes in the future






Implicit Parallelism in ROOT



Implicit Parallelism: Processing Trees

Case 1) Parallel treatment of branches - read, decompress, deserialise in parallel

- Immediately useful with sequential (and thus possibly not thread-safe) analysis code
- Example: PyROOT uses `TTree::GetEntry()` !



```
ROOT::EnableImplicitMT();
```

```
auto file = TFile::Open("http://root.cern.ch/files/h1/dstarmb.root");  
TTree *tree = nullptr; file->GetObject("h42", tree);
```

```
for (Long64_t i = 0; tree->LoadEntry(i) >= 0; ++i) tree->GetEntry(i);
```

No change in user code required

Case 2) Parallel treatment of entries

ROOT::TTreeProcessor class, relies on **TTreeReader**

- One task per *cluster* scheduled: No duplication of reading+decompression
- See later for programming model example



Implicit Parallelism: How To

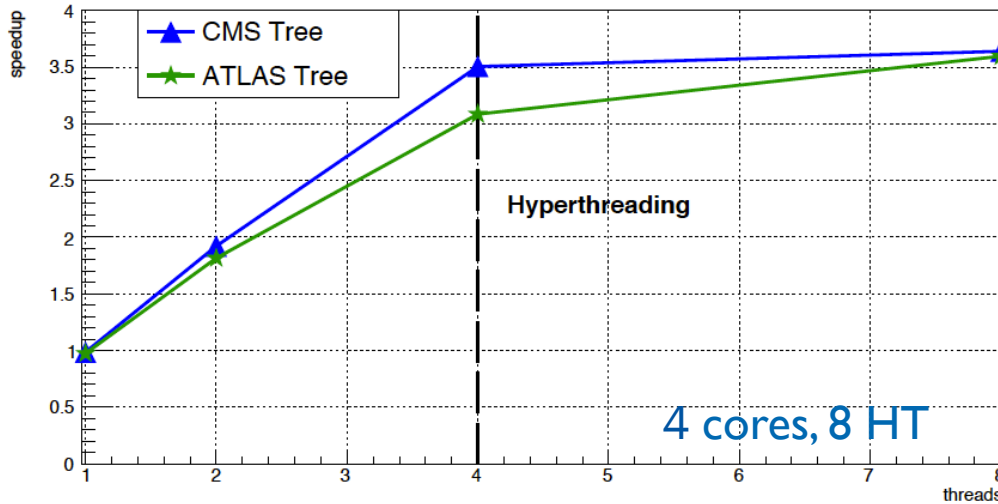
```
ROOT::EnableImplicitMT()
```

-Dimt=ON for
configuring ROOT with
CMake!

ROOT takes care of packetising work and submit it to the runtime

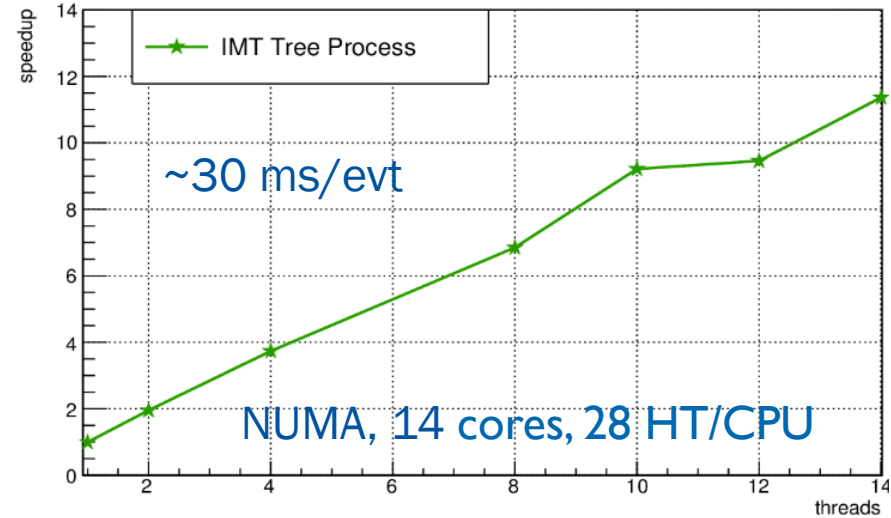


A Performance Figure



Parallel treatment of branches

- Only read, decompress, deserialize entire dataset
- **CMS:** ~70 branches, GenSim data
- **ATLAS:** ~200 branches, xAOD



Parallel treatment of entries

- Basic analysis of MC tracks
- 50 *clusters* in total (**cluster=task**)
- Unbalanced execution with more than 10 threads

Explicit parallelism and protection of resources





Protection of Resources

A single directive for internal thread safety

```
ROOT::EnableThreadSafety()
```

- Some of the code paths protected:
 - Interactions with type system and interpreter (e.g. interpreting code)
 - Opening of TFiles and contained objects (one file per thread)

New utilities, **none of which in the STL:**

- **ROOT::TThreadedObject<T>**
 - Separate objects in each thread, lazily created, manage merging
 - Create threaded objects with `ROOT::TThreadedObject<T>(c'tor params)`
- **ROOT::TSpinMutex**
 - STL interface: e.g. usable with `std::condition_variable`
- **ROOT::TRWSpinLock**
 - Fundamental to get rid of some bottlenecks

Usable with any
threading model



Programming Model

Manages one object per thread, transparently

```
ROOT::TThreadedObject<TH1F> ptHist("pt_dist", "pt_dist", 128, 0, 64);
ROOT::TTreeProcessor tp("tp_process_imt.root", "events");

auto myFunction = [&](TTreeReader &myReader) {
    TTreeReaderArray<ROOT::Math::PxPyPzEVector> trks(myReader, "tracks");
    while (myReader.Next()) {
        for (auto& trk : trks) myPtHist->Fill(track.Pt());
    }
};

tp.Process(myFunction);
auto ptHistMerged = ptHist.Merge();
```

“Work item”

**Mix ROOT, modern
C++ and STL**

More about the programming model in the backup slides!



Two R&D Lines



Functional Chains R&D

- We are constantly looking for opportunities to apply implicit parallelism in ROOT
- “Functional Chains” R&D being carried out
 - Functional programming principles: no global states, no for/if/else/break
 - Analogy with tools like ReactiveX*, R dataframe, Spark
 - Gives room for optimising operations internally

Can this be a successful model for our physicists?

```
import ROOT
f = ROOT.TFile("aliDataset.root")
aliTree = f.Events
dataFrame = TDataFrame(aliTree)
```

Express analysis as a chain of functional primitives.

```
dataFrame.filter(sel1).map(func2).cache().filter(sel3).histo('var1:var2').Draw('LEGO')
```

* <https://reactivex.io>



The ROOT-Spark R&D



- HEP data: statistically independent collisions
- Lots of success: PROOF, the LHC Computing Grid
 - Can we adapt this paradigm to modern technologies?
- **Apache Spark**: fault-tolerant, in-memory distributed computation framework
 - Written in Scala, bindings for Java, R and **Python** (our bridge to ROOT)
 - Widely adopted in data-science community

**In collaboration with CERN
IT-DB-SAS and IT-ST-AD**

Our idea:

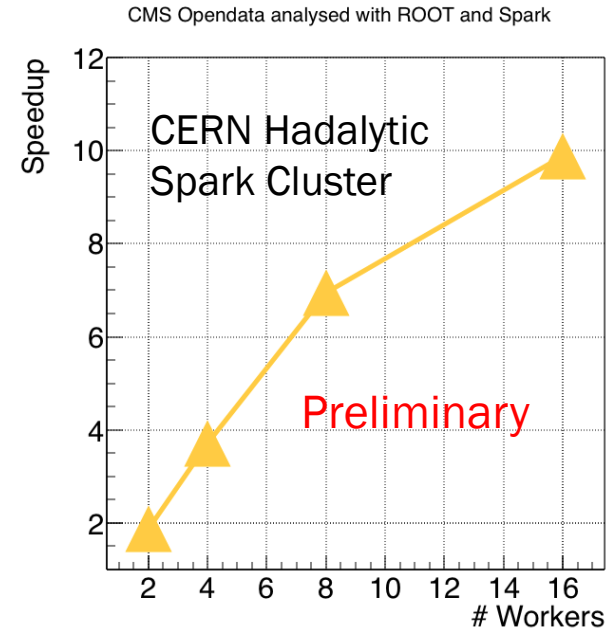
- 1) Use Spark to process with PyROOT + C++ libraries / C++ code JITted by ROOT
- 2) Cloud storage for software and data (CVMFS and EOS)
- 3) Identical environment on client and workers



Our First Test

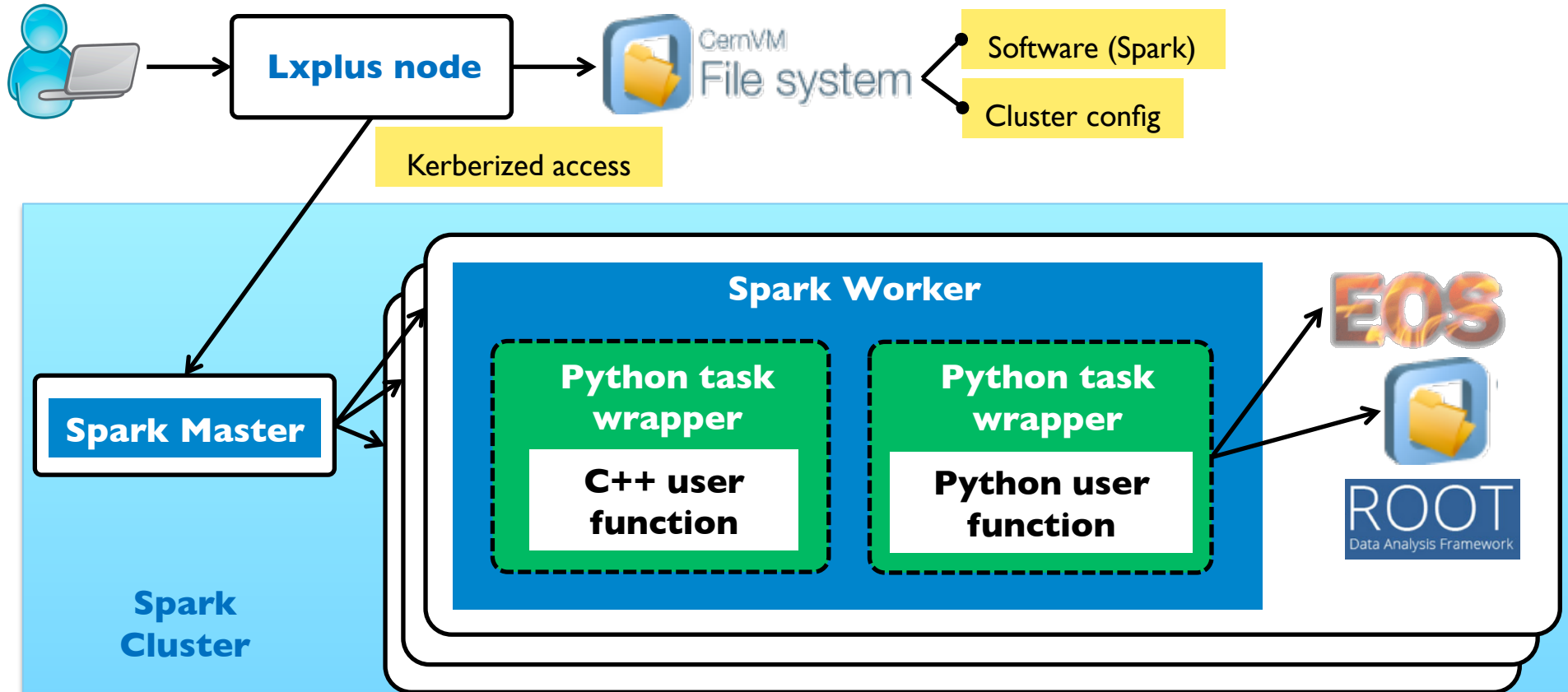
- CMS Opendata <http://opendata.cern.ch/record/1640>
 - Analyse kinematic properties of generated jets
- Read ROOT files natively with PyROOT
 - Get back merged histograms
- IT managed Spark cluster at CERN
 - Client is LXPLUS node
- Easy setup: source a script

Split by file



We can run on CMS Opendata with ROOT exploiting an already existing Spark cluster

Our First Test

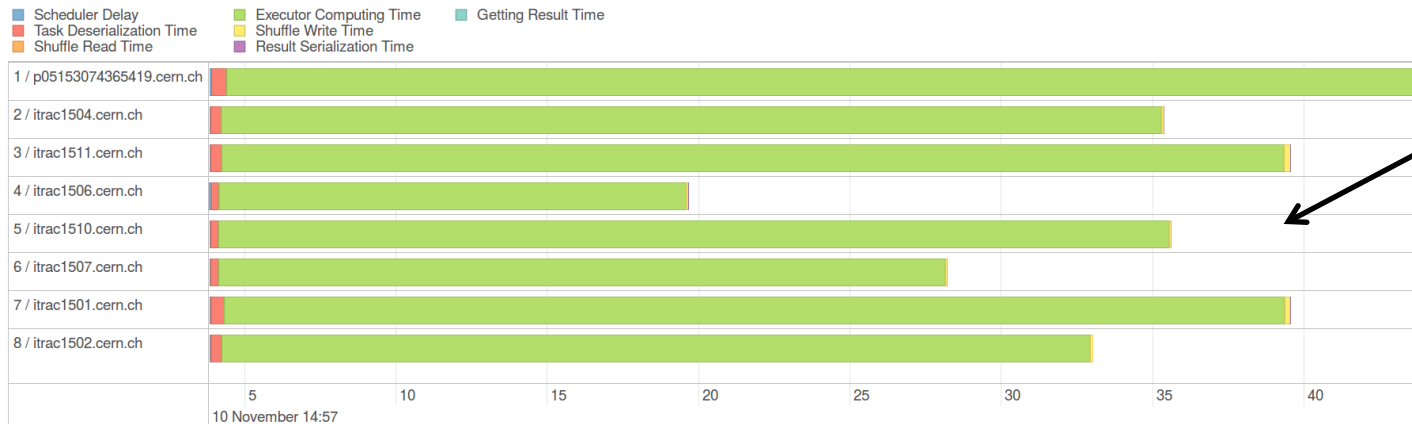




Reduce by Key + Collect

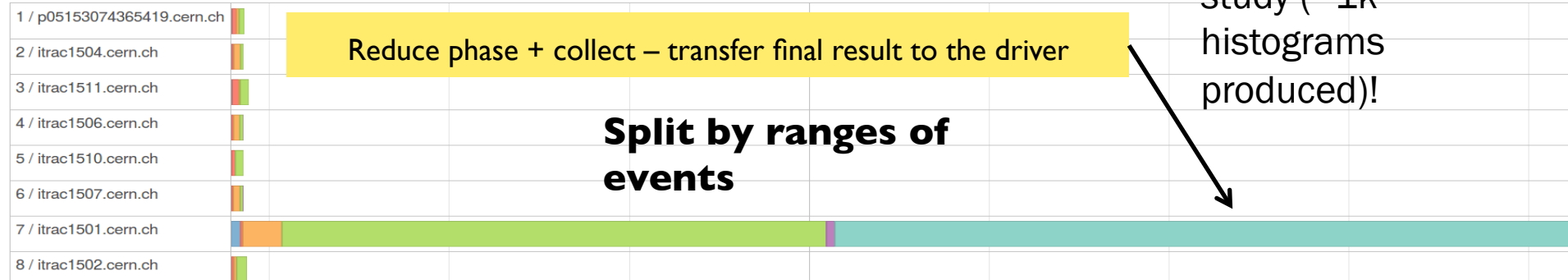
CERN groups involved: EP-SFT + IT-DB + IT-ST

- **8 executors**
- **8 partitions – 8 mapper tasks, 1 reducer task**



Map phase - imbalance

Full study on Hadalytic: reduce, reduceTree
Real CMS W mass study (~1k histograms produced)!





- ROOT is evolving and following a modern approach
 - ROOT namespace, new classes ...
- General purpose MT and MP executors (e.g. map, mapReduce patterns)
- Utilities to facilitate explicit parallelism, complement STL
 - ROOT as a “foundation library”
- Provide access to implicit parallelism
 - Formulate solution using certain interfaces, ROOT takes care of the rest

All this delivered in ROOT 6.08

The future:

- Find new opportunities for implicit parallelism, e.g. functional chains
- Continue exploring new technologies, e.g. Apache Spark and other runtimes



Backup



Programming Model

```
ROOT::EnableThreadSafety();
ROOT::TThreadedObject<TH1F> ts_h("myHist", "Filled in parallel", 128, -8, 8);

auto fillRandomHisto = [&](int seed = 0) {
    TRandom3 rndm(seed);
    auto histogram = ts_h.Get();
    for (auto i : ROOT::TSeqI(1000000)) {
        histogram->Fill(rndm.Gaus(0, 1));
    }
};

std::vector<std::thread> pool;

for (auto s : ROOT::TSeqI(1, 5)) pool.emplace_back(fillRandomHisto, s);
for (auto && t : pool) t.join();

auto sumRandomHisto = ts_h.Merge();
```

**Fill histogram randomly
from multiple threads**

**Mix ROOT,
modern C++ and
STL seamlessly**



Programming Model

```
ROOT::TProcessExecutor mpe(4);
```

**Fill histogram randomly
from multiple threads**

```
auto fillRandomHisto = [](int seed) {  
    auto h = new TH1F("myHist", "Filled in parallel", 128, -8, 8);  
    TRandom3 rndm(seed);  
    for (auto i : ROOT::TSeqI(1000000)) {  
        h->Fill(rndm.Gaus(0, 1));  
    }  
    return h;  
};
```

**Seamless usage of
processes**

```
ROOT::ExecutorUtils::ReduceObjects<TH1F*> rf;  
auto sumHisto = mpe.MapReduce(fillRandomHisto, ROOT::TSeqI(10), rf);
```

Return type inferred from work-item signature



TTree I/O Objects

- Per-branch data
- Per-tree data

