Expressing Parallelism with ROOT

https://root.cern

D. Piparo (CERN) for the ROOT team

CHEP 2016

This Talk

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 !

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

Explicit parallelism and protection of resources

Protection of Resources

`ROOT::EnableThreadSafety()`

A single directive for internal thread safety

- 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::MakeThreaded<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

```
ROOT::EnableThreadSafety();
auto ts_h = ROOT::MakeThreaded<TH1F>("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));
   }
};

auto seeds = ROOT::TSeqI(1, 5);
std::vector<std::thread> pool;

for (auto s : ROOT::TSeqI(seeds)) 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

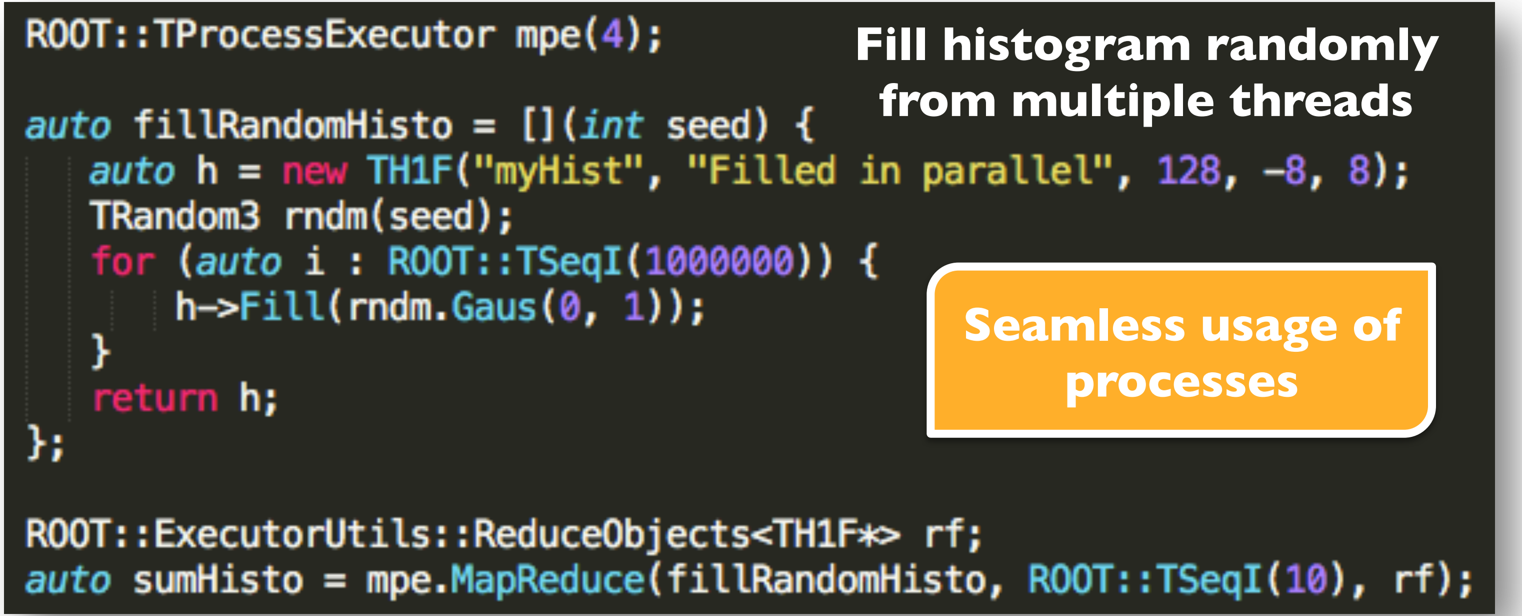
General Purpose Parallel Executors

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**: additional methods to process trees!
  - Interplay with TTreeReader

- Adopted threading library: **TBB**
  - Not visible to the user, share pool with experiments' frameworks

Programming Model



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

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;
};

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

Fill histogram randomly from multiple threads

Seamless usage of processes

Return type inferred from work-item signature

Implicit Parallelism in ROOT

Implicit Parallelism

- Cover common use cases: focus on dataset processing (TTree's)
- Two scenarios:
  - Sequential processing of entries: parallel branches' reading, decompression and deserialization (independent from analysis/reconstruction code)
  - Parallel processing of entries (needs thread-safe analysis code)

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

-Dimt=ON for configuring ROOT with CMake!

Parallel Treatment of Branches

- Two modes for reading from trees:
  - Access individual branches - TBranch::GetEntry()
  - (De)activate some branches, access entire entries - TTree::GetEntry
- Immediately useful with sequential (and thus possibly not thread-safe) analysis code

Example: PyROOT uses TTree::GetEntry

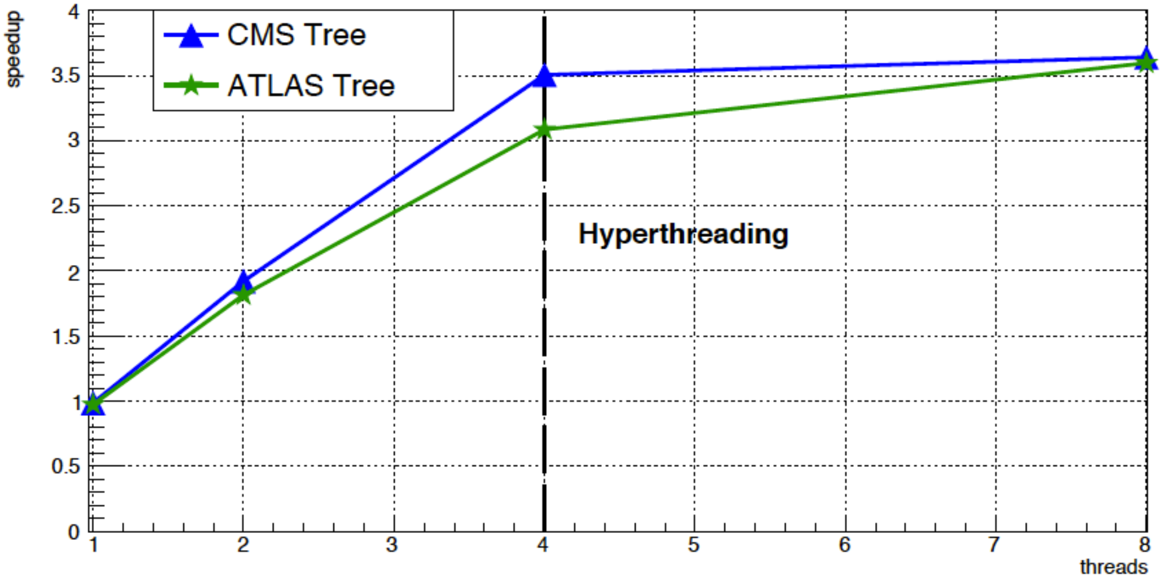
```
ROOT::EnableImplicitMT();

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);  // parallel read
}
```

No change in user code required

A Performance Figure



- Intel i7-3770
- 4, 8 HT
- Read, decompress, deserialize entire dataset

- CMS: ~70 branches, GenSim data
- ATLAS: subset of ~200 branches, xAOD

Multiple Entries In Parallel

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

- One task per *cluster* scheduled: No duplication of decompression + deserialisation

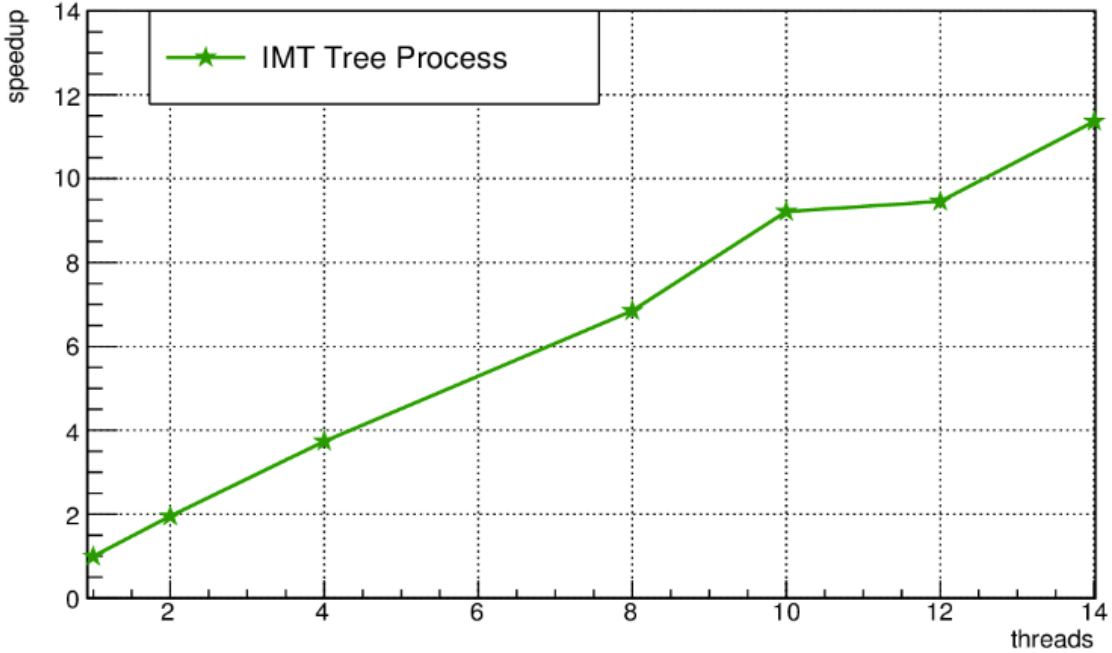
```
auto ptHist = ROOT::MakeThreaded<TH1F>("pt_dist", "pt_dist", 64, 0, 4);
ROOT::TTreeProcessor tp("tp_process_imt.root", "events");

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

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

Manage TTree processing scheduling tasks on N threads

A Performance Figure



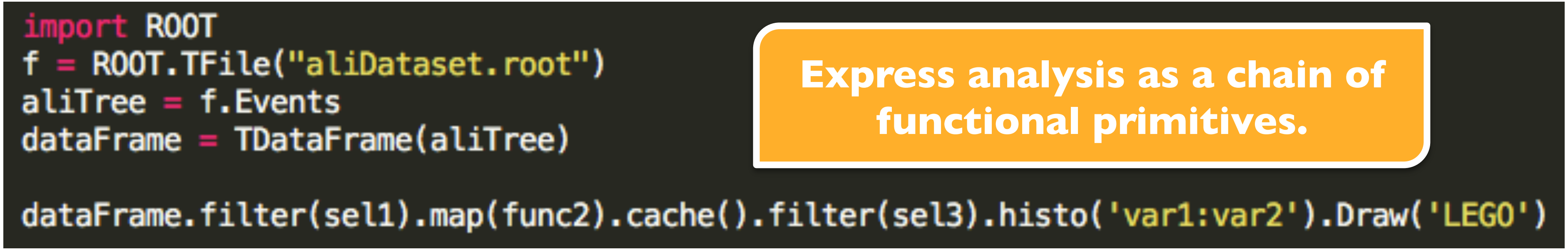
- Dual Intel 5-2683V3
- 14 cores, 28 HT per CPU
- Basic analysis of MC tracks

- 50 clusters in the dataset: unbalanced execution after 10 threads

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
- Goal: express selections on datasets via concatenation of transformations
  - Alternative to traditional imperative approach
  - Gives room for optimising operations internally



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

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

Express analysis as a chain of functional primitives.

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: general engine for large-scale data processing
  - Cluster management tool widely adopted in data-science community
  - Scala, Java, R and Python support

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

Our idea:

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

A First Test

- CMS Opendata http://opendata.cern.ch/record/1640 dataset
  - Transverse momentum spectrum of AK5 generated jets
- Read ROOT files natively with PyROOT
- IT managed Spark cluster at CERN, CVMFS and EOS available on the nodes
- Driver is LXPLUS node, identical software setup via CVMFS

PROOF OF CONCEPT

SCALING?

Spectrum

Bottomline and Outline

- ROOT evolves: new utilities for expressing parallelism, a modern approach
  - E.g. native interoperability with the STL, focus on the programming model
- General purpose MT and MP executors (e.g. map, mapReduce patterns)
- Utilities to facilitate explicit parallelism, complement STL
  - ROOT is 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

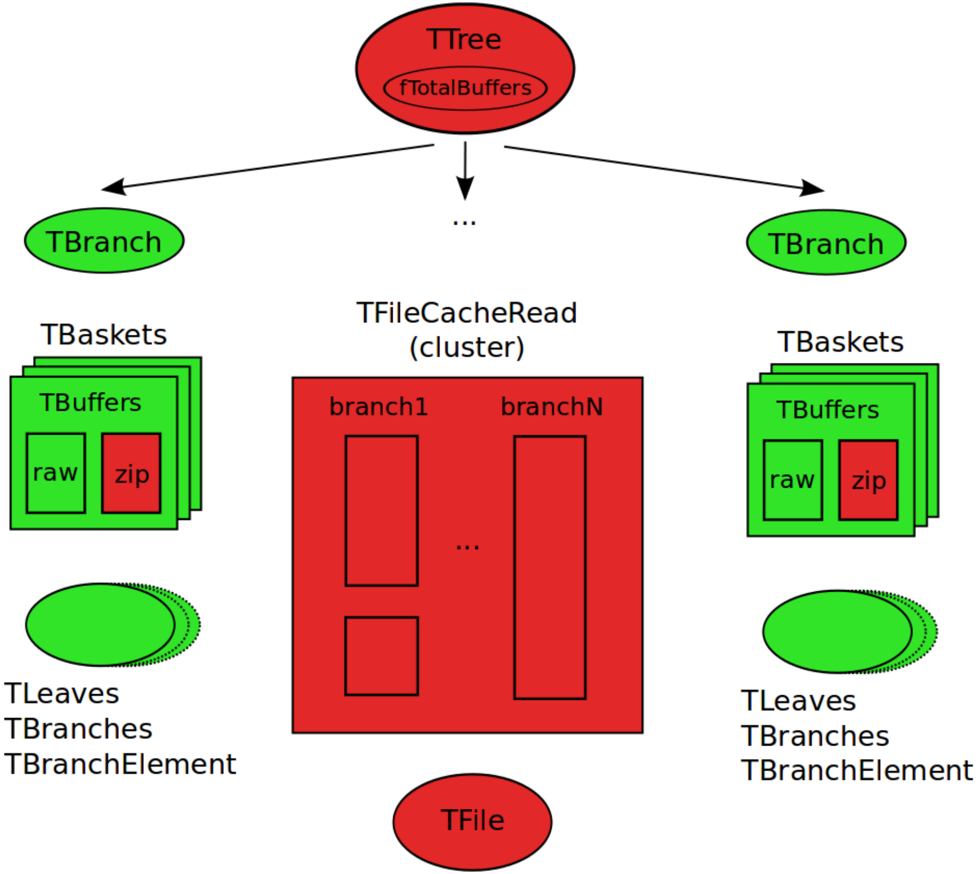
- Find new opportunities for implicit parallelism, e.g. functional chains
- Continue exploring new technologies, e.g. Apache Spark and other runtimes

Backup

TTree I/O Objects

🟢 Per-branch data

🔴 Per-tree data



TTree
fTotalBuffers

TBranch
...
TBranch

TBaskets
TBuffers
raw
zip

TFileCacheRead
(cluster)
branch1
...
branchN

TBaskets
TBuffers
raw
zip

TLeaves
TBranches
TBranchElement

TLeaves
TBranches
TBranchElement

TFile