# Increasing Parallelism in the ROOT I/O subsystem

G. Amadio, B. Bockelman, P. Canal, D. Piparo, E. Tejedor, Z. Zhang
for the ROOT Team

ROOT
Data Analysis Framework
https://root.cern

# This Talk

- Introduction
  - Meaning of ROOT I/O: disk access, (de)compression, (de)serialisation
  - The “one file per thread” paradigm
- Parallel Reading
  - Reading columns (branches) in parallel
  - Decompressing data in parallel
- Parallel Writing
  - Writing columns in parallel
  - Writing data from multiple threads to the same file
- Bottomline and Outlook

# ROOT I/O at a Glance

- Much more than reading/writing data from/to disk
  - By the way, something that's hard to parallelise itself
- Read / inflate / deserialize ↔ serialize / deflate / write out
  - Granularity imposed onto files (e.g., clusters of entries)
- Interactions with other parts of ROOT
  - Dynamic library loading
  - Bulk reading of data (TTreeCache)
  - Queries to the type system to determine how objects are represented
- Objects can be stored column-wise or row-wise
  - Partial reads possible, data dependencies (e.g. pointers, array sizes)
- We cover mostly column-wise I/O in this talk

# Several Ways to Parallelise

- High-level parallelisation of I/O of columnar data
  - Process multiple columns in parallel
  - Process multiple entries (events) in parallel
- Not mutually exclusive
  - Nested parallelism is possible and recommended!
- ROOT I/O has multiple phases
  - Each phase can be parallelised independently
- Runtime support for nested parallelism is crucial
  - ROOT uses Intel® Threading Building Blocks (TBB)

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

# ROOT I/O:
# Parallel Reading

# Reading a Single File in Parallel

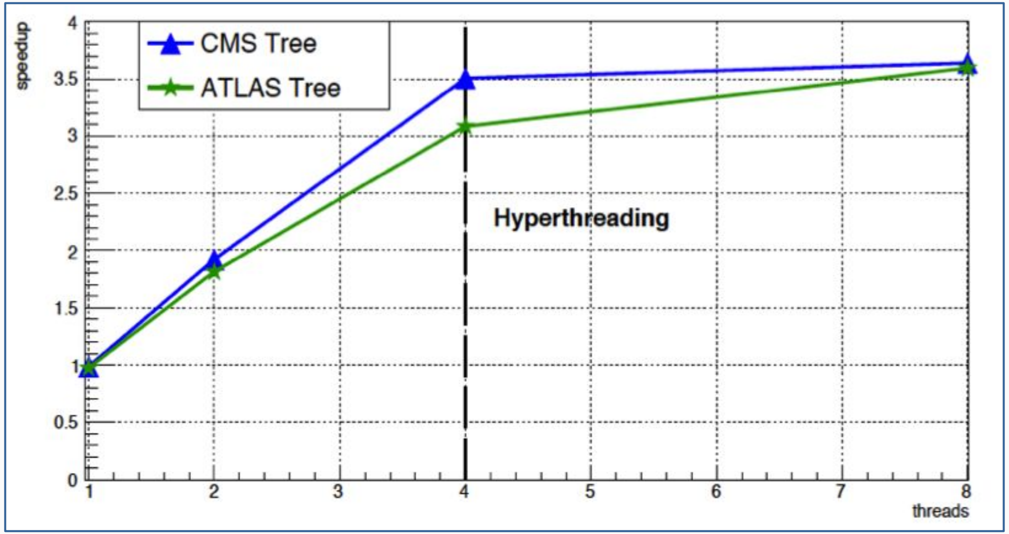
- ROOT can read the same file from multiple threads
  - Manages non-trivial interactions with type system
  - Automatic loading of dynamic libraries
- Implementation
  - One instance of a `TFile` per worker thread
  - Parallelise on entries: all independent
  - Transparent to the user

**Available since ROOT 6.06**

# Reading Data Columns in Parallel

- First example of parallelisation of ROOT I/O
- Concept: read multiple columns (branches) in parallel
  - Not a trivial “parallel for” loop due to data dependencies (e.g. references, array sizes)
- Benchmark: Read, decompress, deserialize two datasets
  - **CMS**
    - ~70 branches
    - GenSim data
  - **ATLAS**
    - ~200 branches
    - xAOD format

**Available since ROOT 6.08**




*Thanks to B. Bockelman*

# Decompressing Baskets in Parallel

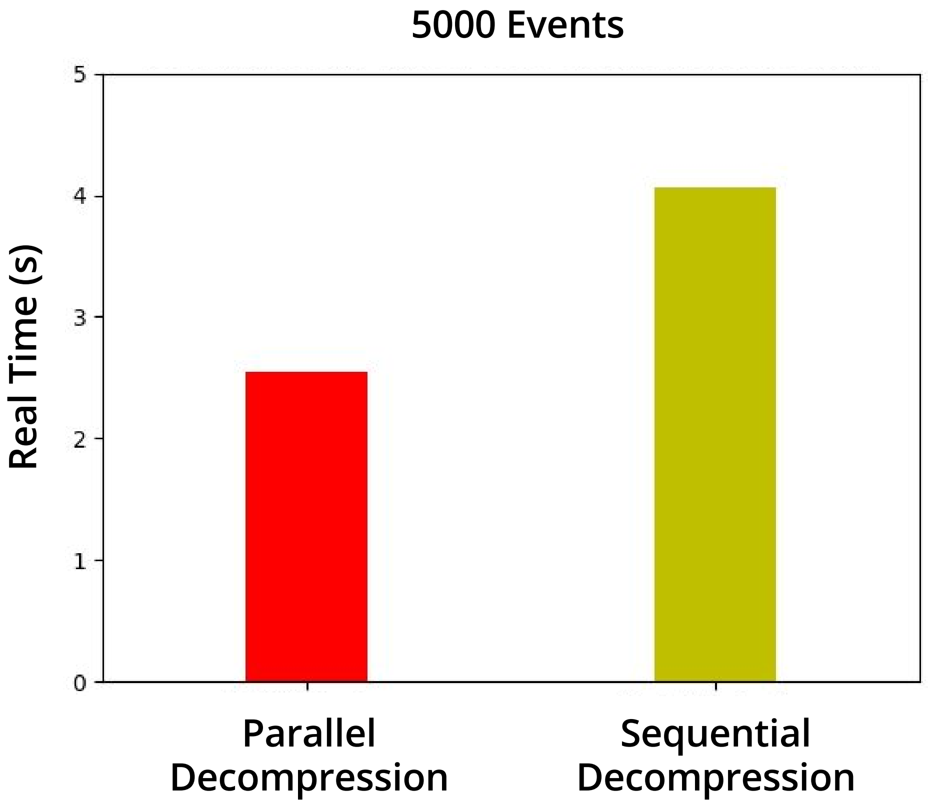
1. ROOT optimises reading by retrieving data in big chunks
   a. Mechanism referred to as `TTreeCache`
   b. Useful also with remote files (good bandwith, bad latency)
2. Concept: increase parallelism during decompression
   a. Start processing data in a `TTreeCache`
   b. Asynchronously fetch a new big chunk filling the `TTreeCache`
   c. Start inflating baskets of compressed data contained in it
   d. Go to a. and repeat
3. Interleave decompression with data processing

**Work in progress, targeting ROOT 6.12 release in November**

*See backup slides for more details about the federation of classes involved*

# Benchmark: Parallel Basked Decompression

- ROOT Event Data
- Fully split dataset
- Tested on an Intel® Core i5 3330 (6M cache, 3.00 GHz)




*Thanks to Z. Zhang*

# ROOT I/O:
# Parallel Writing

# Writing to Different Files in Parallel

- Analogous to the read case: write one file per thread
- Not transparent: data needs to be merged at the end
  - Parallelisation of merging tool **hadd**
- This situation needed an improvement

**Available since ROOT 6.06**

# Writing Data Columns in Parallel

- Writing counterpart of the parallel reading of columns
- Concept: flush contents of columns to disk in parallel
  - Serialisation, interaction with trees and files
  - Consequence: compression tackled in parallel
- As with parallel reading, not a trivial "parallel for"
  - Several optimisations in place
  - For example, sorting by size: long tasks start first → better balance

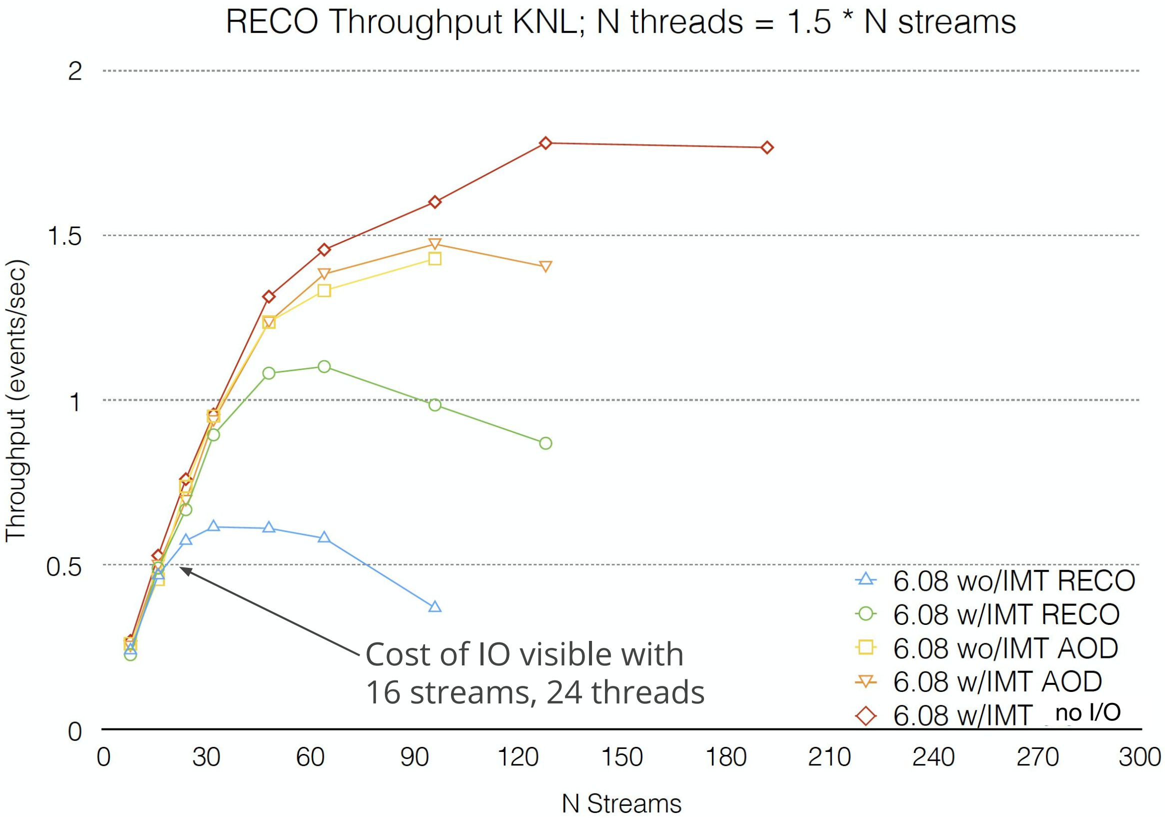
**Available since ROOT 6.10**

# Benchmark: Writing out CMS RECO and AOD

Software Release: CMSSW 8_1_X

- Experimental data from 2016
- Stream: “data processing lane”
- RECO: reconstruction format (“High I/O”)
- AOD: analysis format (“Low I/O”)
- IMT: Implicit Multi Threading

**Available since ROOT 6.10**




*Thanks to B. Bockelman*

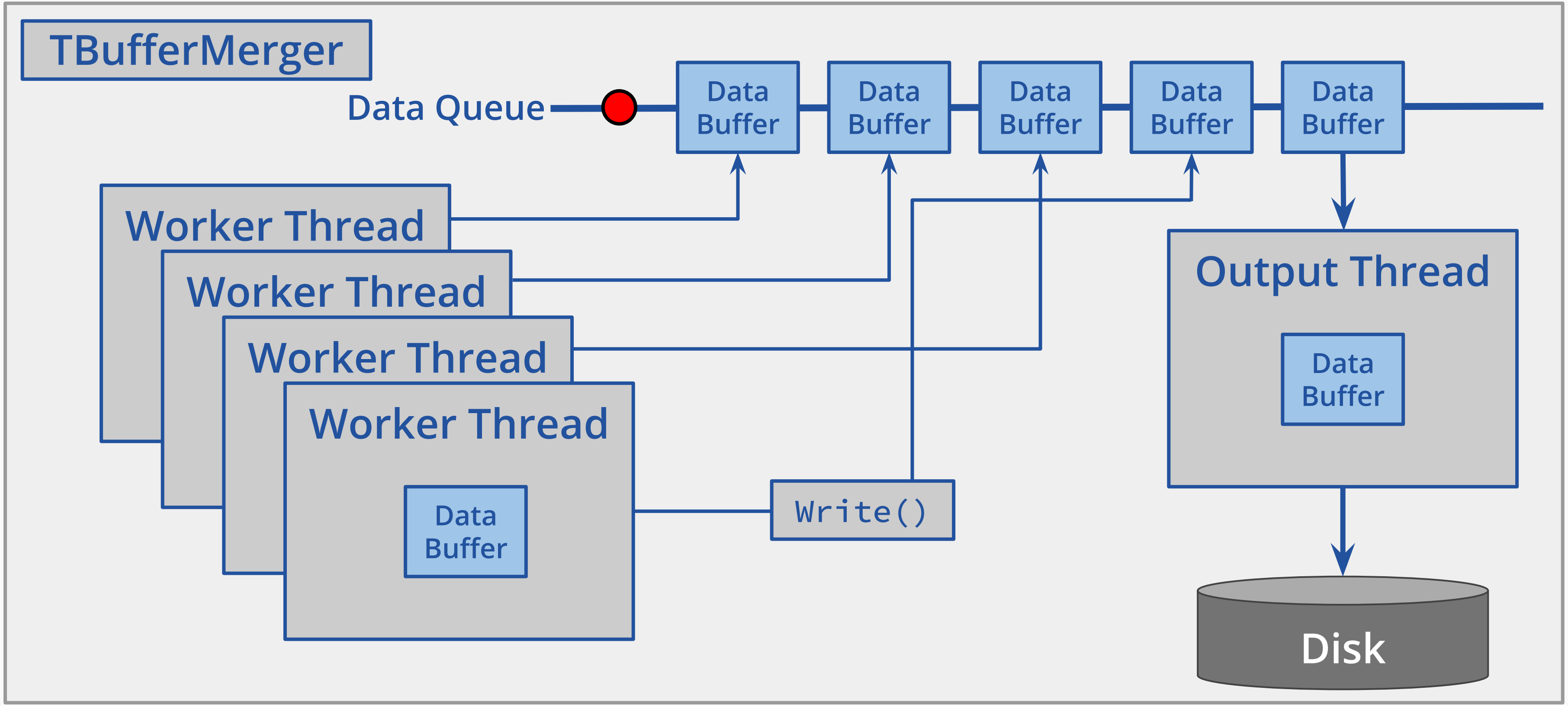
# Writing to a File from Multiple Threads

- Goal: overcoming current “one file per thread” limitation
- In-process handling of writing mergeable objects from different threads
  - Here we focus on TTrees for simplicity
- Created to support TDataFrame’s snapshot action
  - However, can be used in other cases, e.g. experiments’ frameworks
  - Discussing with CMS on how to improve/customize it
- Implementation: TBufferMerger class
  - Factory of in-memory files that send their buffers into a merging queue

**Available since ROOT 6.10**

*See slides for talk Novel functional and distributed approaches to data analysis available in ROOT for more information on snapshot action*

# TBufferMerger Class



# TBufferMerger Programming Model

## Sequential usage of TFile

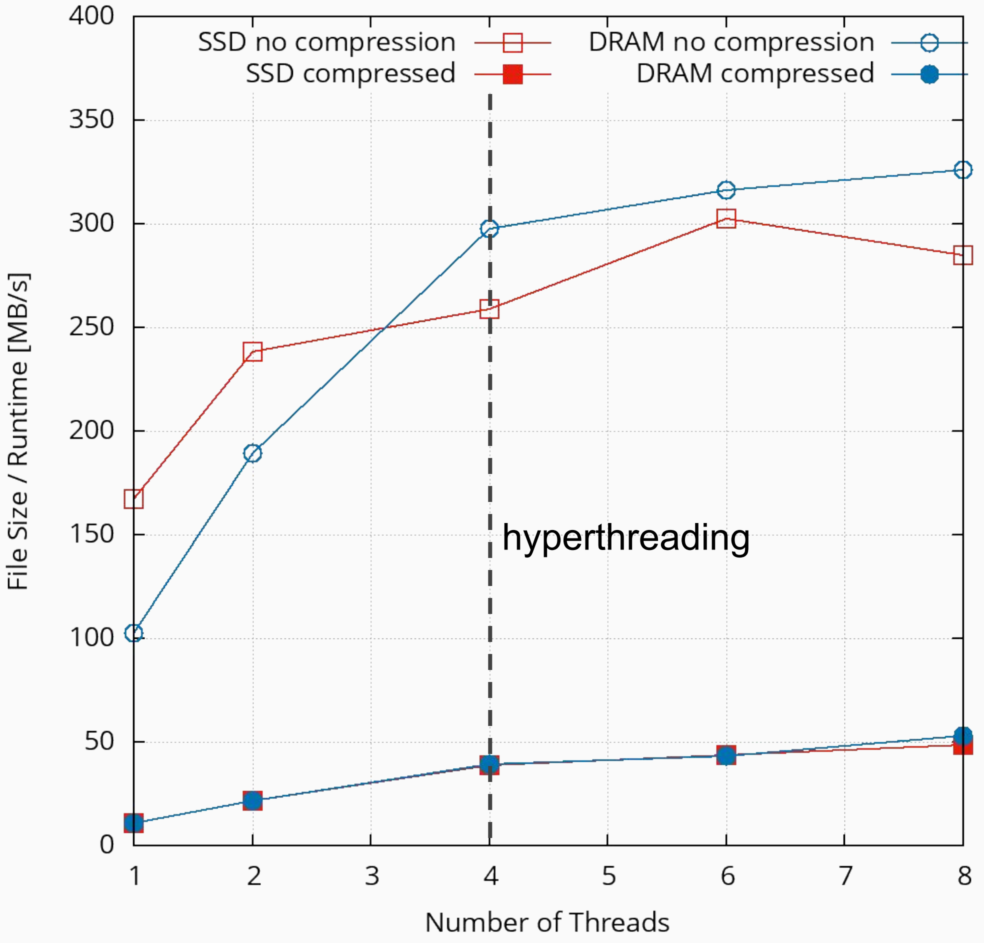
```
void Fill(TTree &tree, int init, int count)
{
   int n = 0;

   tree->Branch("n", &n, "n/I");

   for (int i = 0; i < count; ++i) {
      n = init + i;
      tree.Fill();
   }
}

int WriteTree(size_t nEntries)
{
   TFile f("myfile.root");
   TTree t("mytree","mytree");

   Fill(&t, 0, nEntries);

   t.Write();

   return 0;
}
```

## Parallel usage of TFile with TBufferMerger

```
void Fill(TTree *t, int init, int count); // same as on the left

int WriteTree(size_t nEntries, size_t nWorkers)
{
   size_t nEntriesPerWorker = nEntries/nWorkers;

   ROOT::EnableThreadSafety();
   ROOT::Experimental::TBufferMerger merger("myfile.root");

   std::vector<std::thread> workers;

   auto workItem = [&](int i) {
         auto f = merger.GetFile();
         TTree t("mytree", "mytree");
         Fill(t, i * nEntriesPerWorker, nEntriesPerWorker);
         f->Write(); // Send remaining content over the wire
      };

   for (size_t i = 0; i < nWorkers; ++i)
      workers.emplace_back(workItem,i);

   for (auto&& worker : workers) worker.join();

   return 0;
}
```

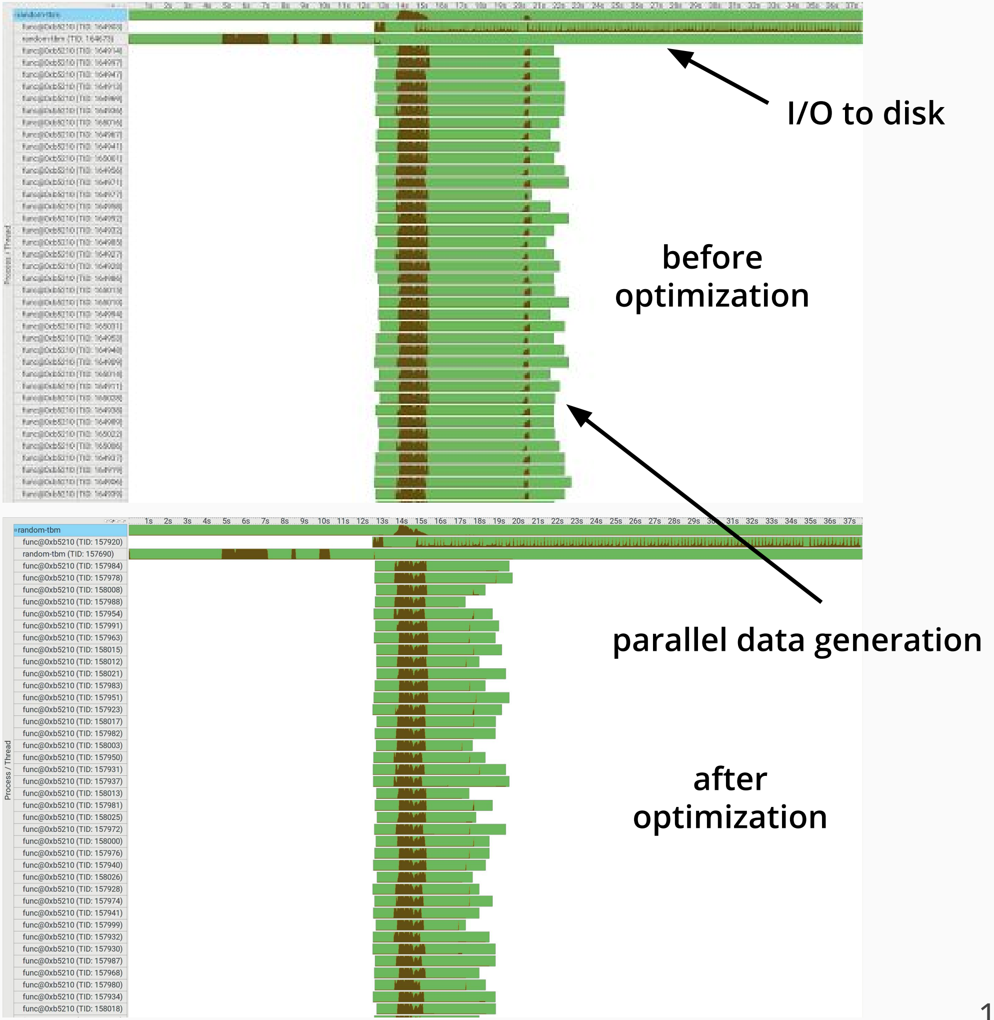
# Benchmark: TBufferMerger with Random Data

- Fill a tree with one branch with random numbers
- Synthetic benchmark that exacerbates the role of I/O by doing only lighweight computations
- Create ~1GB of data and write out to different media (SSD and DRAM)
- Quad core laptop Intel® Core i7 4710HQ (2.5GHz, 6M cache)



# Improving the Performance of ROOT I/O

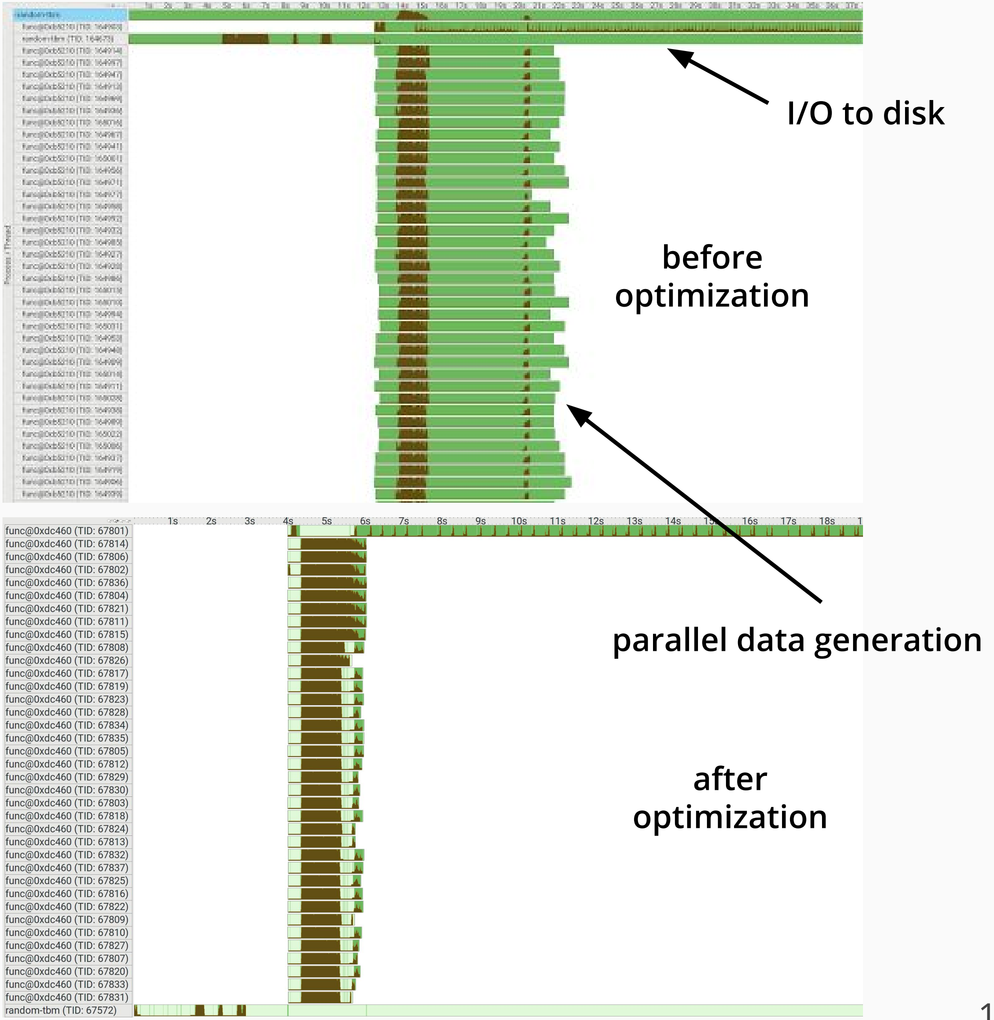
- Use simple case with TBufferMerger to optimize ROOT I/O
- Same random number generation from before
- Reduce number of mutex locks acquired when checking the type system
- Reduced from a few hundred locks to a single lock per thread



# Improving the Performance of ROOT I/O

- Use simple case with TBufferMerger to optimize ROOT I/O
- Same random number generation from before
- Reduce number of mutex locks acquired when checking the type system
- Reduced from a few hundred locks to a single lock per thread

**Targeting ROOT 6.12**



# Interlude: Good Old hadd

- Merging several files with identical / similar structure still needed
- Parallelism can be exploited in this case as well
  - **hadd** is now parallelised too

```
$ hadd -j N
```

- Works as before, but better
- Uses multiprocessing for parallelism with `TProcessExecutor`

**Available since ROOT 6.10**

# Bottomline and Outlook

- ROOT continues to parallelise its I/O subsystem
  - Focus not only on experiments' data processing, but also on analysis
- Reading/Writing branches in parallel
  - Factor of 2x on CMS RECO data
- Parallel writing to single output file via **`TBufferMerger`**
  - Leveraged by **`TDataFrame`** already
  - Good performance

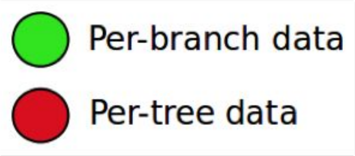
Challenges posed to us:

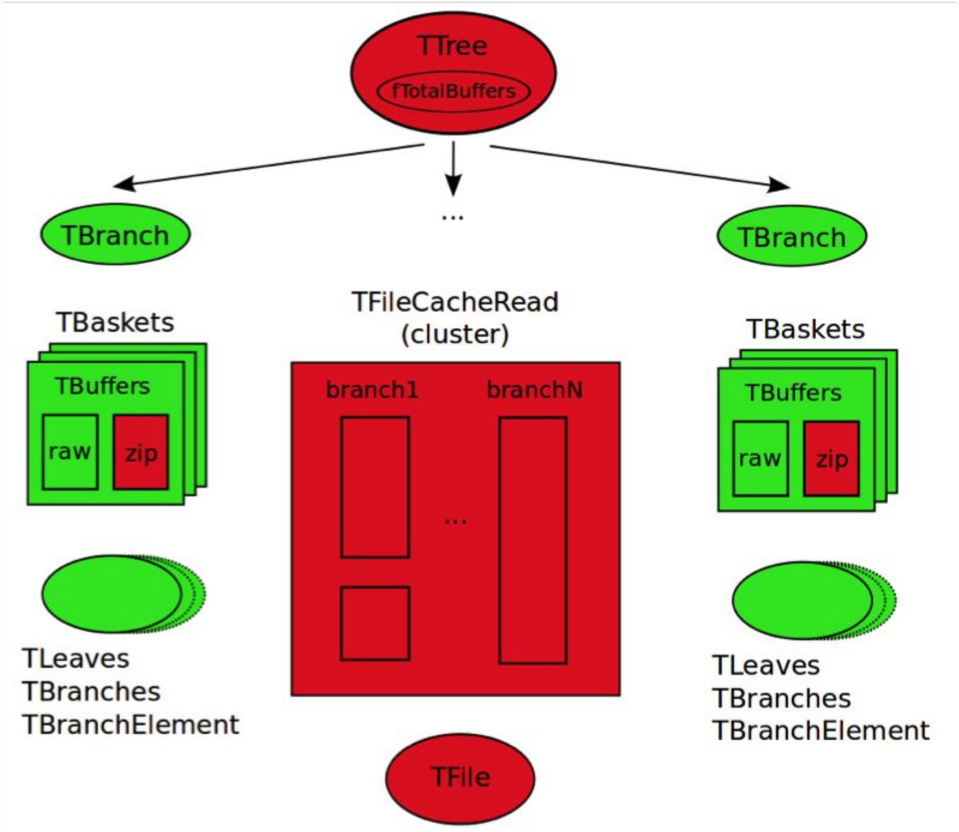
- Better exploit data parallelism (e.g. use vectorised zlib)
- Optimise parallel merging of trees with **`TBufferMerger`**

# Questions?

# Backup Slides

# The Cache Federation of Classes



# One File per Thread Paradigm

- ROOT supports since 6.08 reading and writing of one file per thread
  - Global states eliminated or made thread local
- Good solution but has shortcomings
  - Cannot write/read N files simultaneously with N arbitrarily large
  - Merging several files after they have been written has a cost!
- Does not fit “extreme” architectures, e.g. KNL
  - Need to find more opportunities for expressing parallelism
  - E.g. parallelism nested in all steps of ROOT I/O
- Delicate to match with task-based parallelism
  - Thread-local vs “task specific”