Improving ROOT I/O Performance for Analysis

CHEP 2023, Norfolk, U.S.
May 8, 2023

Canal, Philippe (FNAL) Blomer, Jakob (CERN) Naumann, Axel (CERN)

ROOT
Data Analysis Framework
https://root.cern

Introduction

- Using **100s** of cores (and threads) brings a slew of new challenges.

- Will describe a series of improvements accelerating by order of magnitude
  - Reading via `RDataFrame`
  - Writing using `TBufferMerger`
- Including several re-usable lessons learned

Many of the improvements are thanks to
Josh BenDavid & Chris Jones

Amdahl's law is harsh at 256 threads

- Result on a smallish ***CMS*** analysis test with 256 threads:
  - **78x** speedup in elapsed time
  - Reduces wall time from 25 minutes to 19 seconds
  - Increases CPU usage from 400% to 4000%.

- Change in a single function (`TBufferFile::ReadClassBuffer`)
  - Use Read and Write part of global Read/Write lock
  - Reduce critical section (write lock) down to the (rare) one time initialization
  - Keep rest of the hotspot under 'only' the read lock.

Atomics are easy, right?

- Not so fast. Still need an (implicit) synchronization.
- 2.2x improvement (40M obj write on 32 threads: 322s down to 141s)
- Avoid multiple calls to std::atomic::load
- Use more relaxed memory order
  - Switched from memory_order_seq_cst (sequentially-consistent ordering, default) to std::memory_order_relaxed

```
auto value = fAtomic.load();
if (!value)
   … some initialization …
return value ? value : kDefault;
```

```
if( !fIsInit.load(std::memory_order_relaxed) )
{
  std::lock_guard(mutex);
  if (!fIsInit.load())
    // Actual one-time initialization
}
```

Relaxed operation: no synchronization or ordering constraints imposed on other reads or writes, only this operation's atomicity is guaranteed.

Works here as spurious execution of ”then” will be harmless thanks to lock and recheck.

Making multiple line static initialization thread safe

- Dangerous

```
static bool isinit = false;
static std::vector<size_t> lengths;
if (!isinit) {
  for (...) {
    lengths.push_back(...);
  }
  isinit = true;
}
return lengths;
```

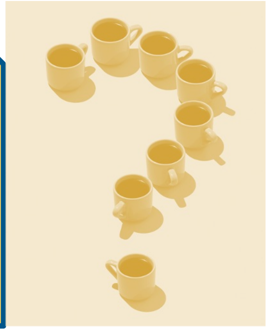
- Thread safe

```
static std::vector<size_t> lengths{ []()
{
  std::vector<size_t> create_lengths;
  for (...) {
    create_lengths.push_back(...);
  }
  return create_lengths;
}() };
return lengths;
```

Speeding-up TFile

- **8x** reduction in elapsed time in a `RDF` benchmark reading one column from 4000 files with 1M entries and using 256 threads
  - New `TFile::Open` option to skip global registration, `RDF` uses this option by default
  - `TFile::Open` no longer reprocess identical `TStreamerInfo`.
- Another **2x** by improving `TFile::Open's` plugin mechanism
  - Increase pre-calculation (pay upfront, avoid synchronization later)
  - Increase caching to avoid calls to locking checks
  - Use local mutex rather than global lock (required attention to avoid dead lock)
- Yet another **2x** Skip registration of `TFile's` UUIDs
  - Breaks the very rare case where a `TRef` points to the TFile object
  - Cpu usage from 1557% to **14271%**

2x in time
9x in CPU usage
Bottleneck switches from **mutex** to spin **locks**

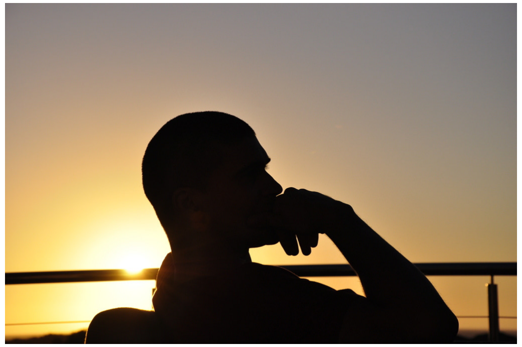


Additional speedups

- **9x** in a realistic `RDataFrame` ***CMS*** based example with many branches
  - Disable garbage collector for `TBranch` within RDF
- **7%** speedup in a medium sized test filling histograms from ***CMS NanoAOD*** with RDF and 256 threads.
  - *New (optional)* ***TBB***-based internal counter for ROOT main to Recursive Read/Write Lock

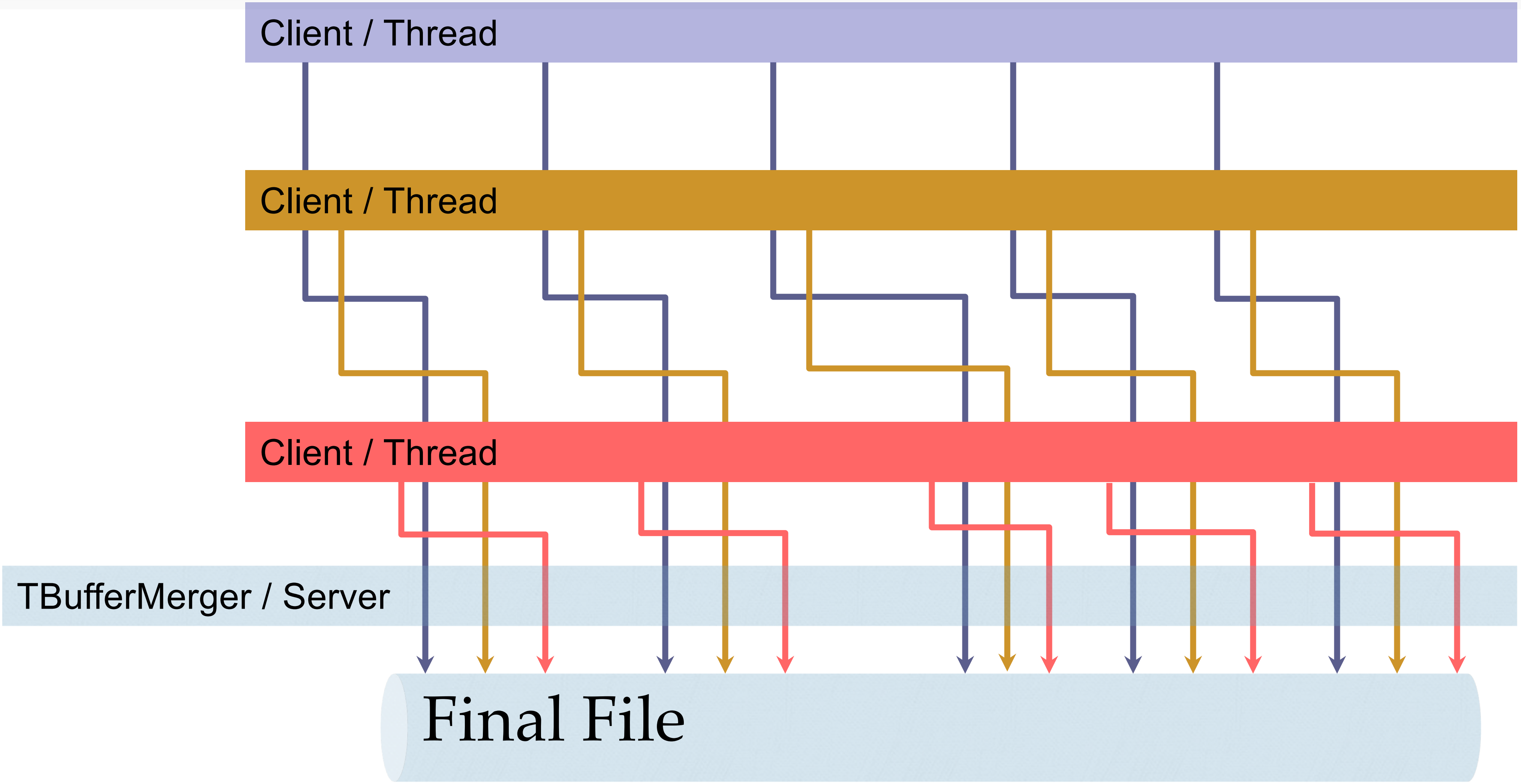
It gets complicated

- `TClassTable`: <u>fixed</u> data race between dlopen and other uses
  - Opening a library register what classes it contains
  - If done from multiple threads, it fills the same containers at the 'same time'
  - Required fine grained lock because:
    - `dlopen` itself take a lock so there is risk of dead locks
    - User might hold ROOT global lock
  - Required to only include elementary actions
    - Even simple 'error handling function' can both take the global lock and recursively call `TClassTable`.

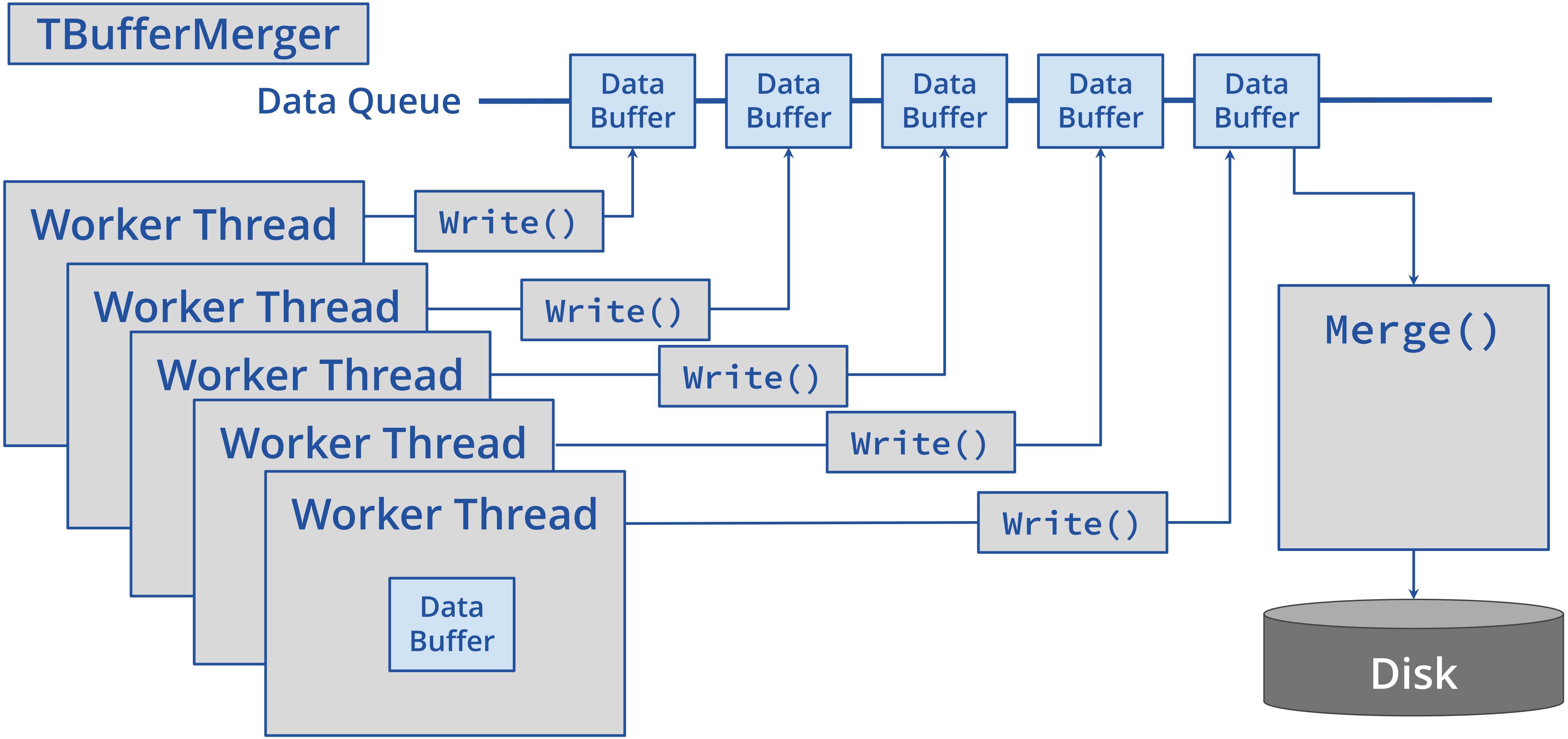


- `gDirectory` is a thread local variable that points to a `TFile` that can be deleted by another thread
  - Requires <u>extremely challenging</u> intercommunication between threads.

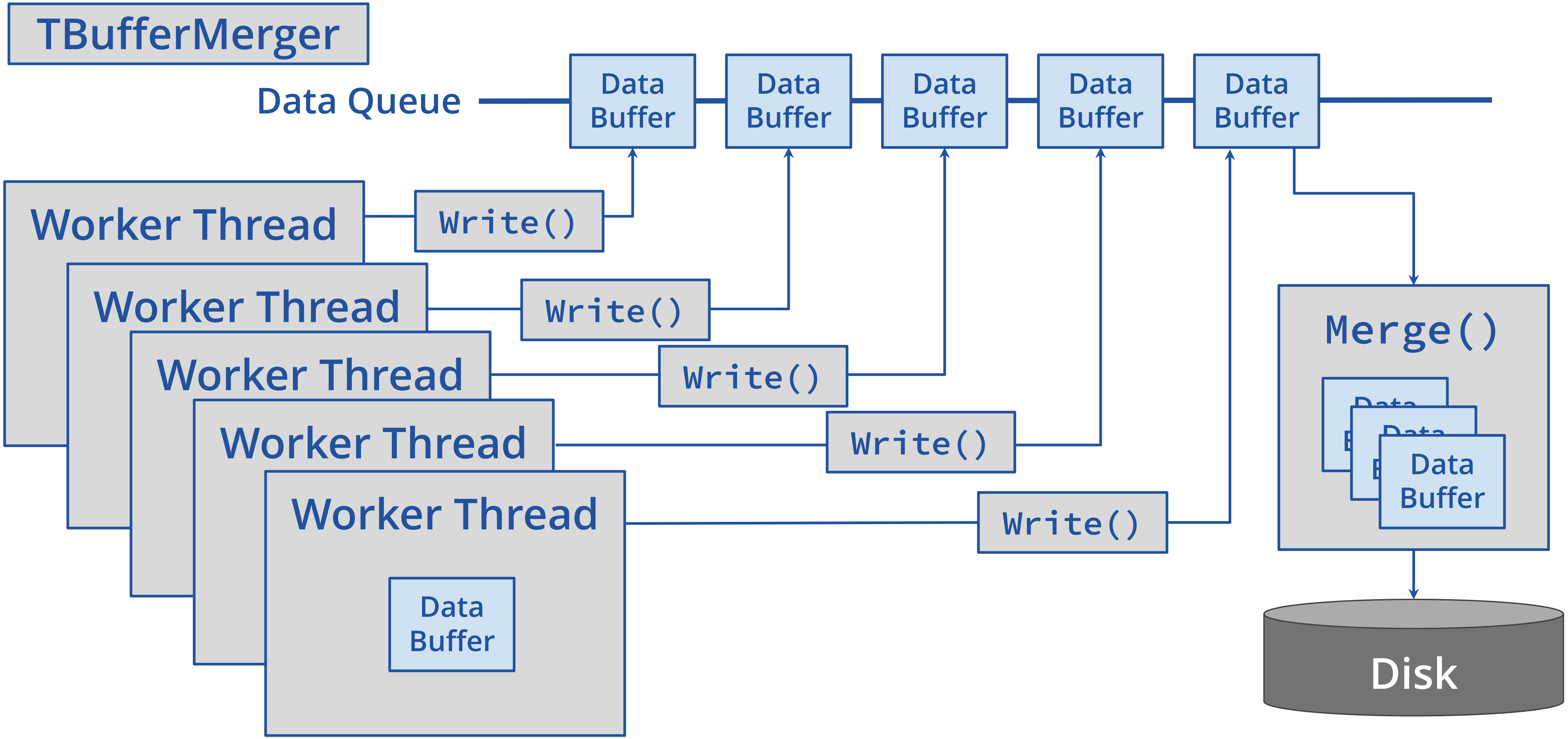
Multiple writer into a single file



TBufferMerger



TBufferMerger when per-thread-time < merging-time*threads



Taming the flood

- *Problem might appear in case of fast data producer and very high thread count (**and high branch count**)*
- Make the merging step as fast as possible:
  - Keep the in-memory `TTree` alive avoid compression and streaming back and forth
  - Optimize the main code paths
  - Trade off crash recovery safety for speed (no intermediate snapshot of meta data)
  - Reduce size of critical section
  - RNTuple merging will be even faster (scale with just number of clusters rather than also number of branches)
- Provide ways to monitor queue size to allow framework to suspend work
  - **Auto-backup will be incorporated in upcoming release.**
- Test:
  - Reading and writing 1000+ branches ran longer than user patience.
  - New version: 11s with 1 thread, 8s with 6 threads with 50 events per chunk (and 22s for 500 events)

Summary

- Improvements made thanks in **very large parts** to submission of running challenging examples **and even** to actual code contributions from user(s).

- Amdahl's law is very noticeable at 256 threads

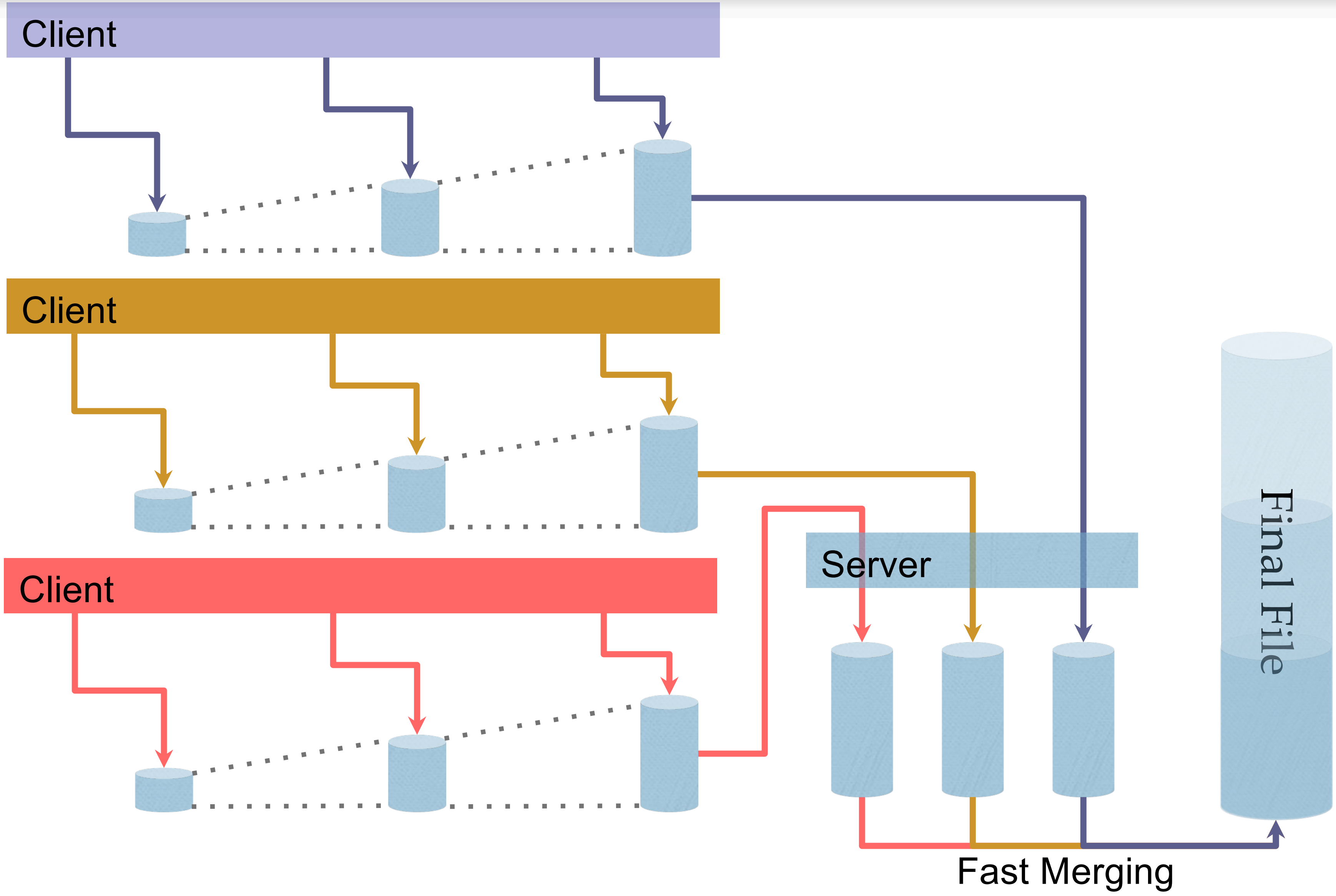
This Photo by Unknown Author is licensed under CC BY

- Broad-strokes enabling of thread safely can ***sometimes*** be enough but source of noticeable slowdown at high thread count.

- But still, existing code can be significantly improved with a few (some simple and some not so simple) techniques.
  - A `RDataFrame` scenario with 256 threads ran **`O(100x)`** faster

Backup Slides

Old Fashion Arrangement



Node used for many of the test

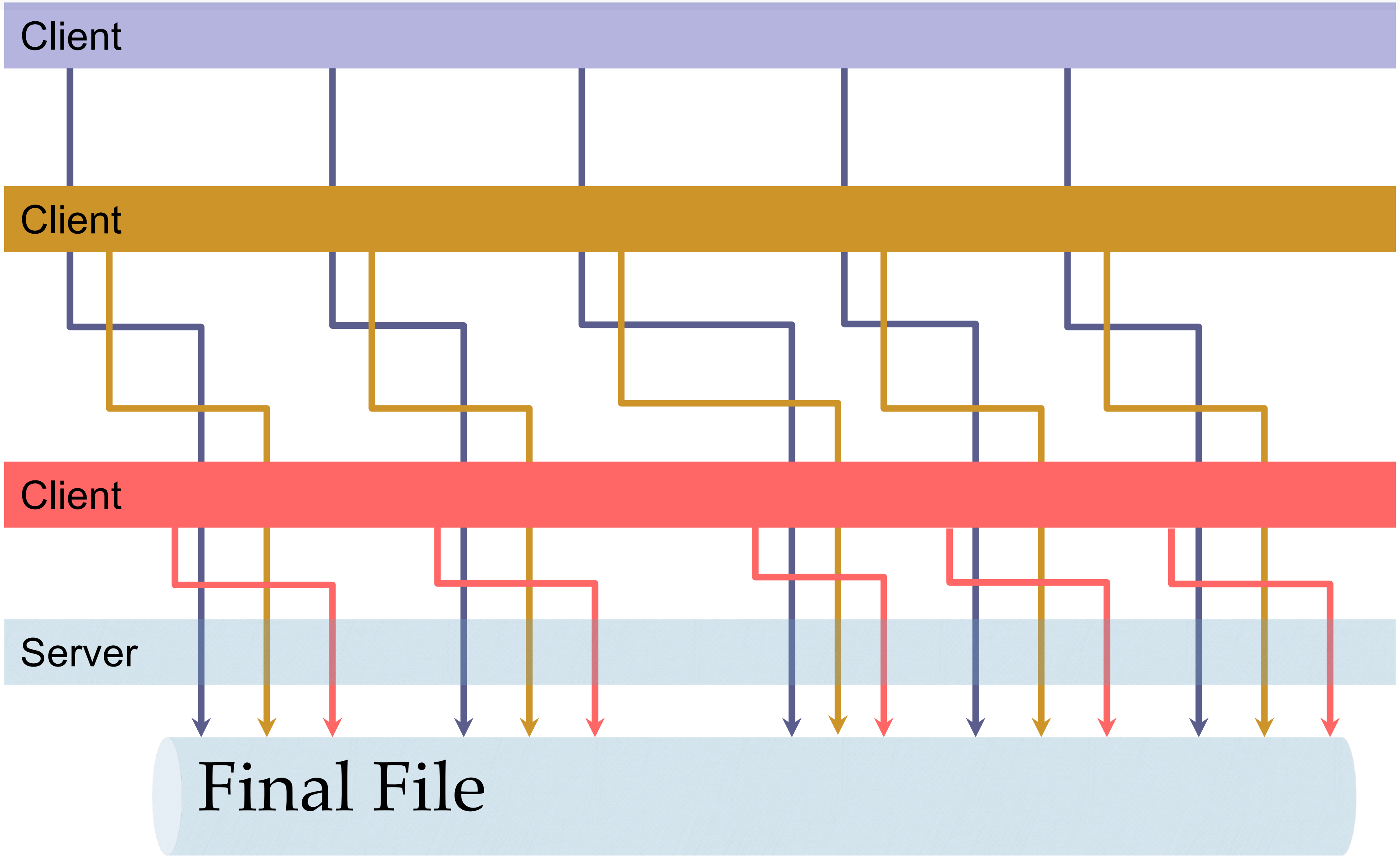
- 128core/256 thread cpu (dual EPYC 7702)
- 1TB RAM
- raid0 array of gen4 nvme ssd's (in synthetic benchmarks the array can push 100Gbytes/sec in sequential reads)
- NIC is 100gbps (relevant for the cases where we test network reads from eos/xrootd, ceph etc, though all of the comparisons are running from the local ssd array unless explicitly stated otherwise)

Fast Merging

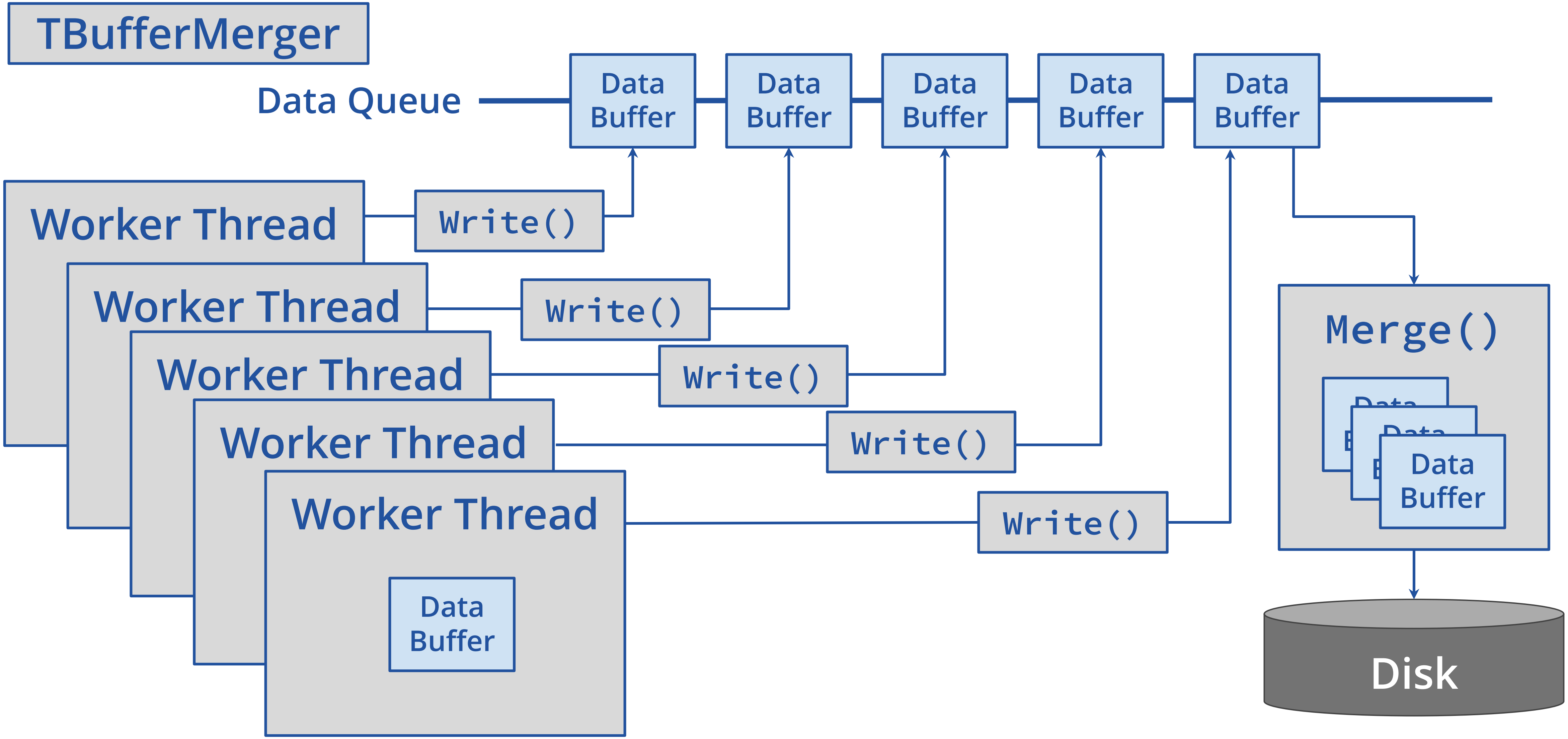
- ROOT Files can be ‘fast’ merged by ‘only’
  - Copying/appending the compressed data (baskets)
  - Updating the meta data (TTree object)
  - In first approximation we reach disk bandwith
    - Actually … half … since we read then write.

- Leverage this capability and use in-memory file to add support for multiple writers to the same file
  - Data just written once, hence reaching disk bandwith
  - Multi-thread in production
  - MPI in production

With Parallel Merging



TBufferMerger



One sort-of breaking change

- <u>Skip registration</u> of `TFile's` UUIDs
  - **2x** elapsed time reduction on a RDF benchmark reading one column from 4000 files with 1M entries and using 256 threads:
    - Baseline:
      Percent of CPU this job got: 1557%
      Elapsed time: 0:49.89
    - Improved
      Percent of CPU this job got: **14271%**
      Elapsed time: 0:21.11

2x in time
9x in CPU usage
Bottleneck switches from
mutex to spin **locks**

- Breaks the very rare case where a `TRef` points to the TFile object
  - which was already not properly supported in multi-thread

Additional Note

- TFile WriteCache
  - Allow delaying and coalescing the write at the cost of more memory
  - Not often used as gain is minimal on a single disk and memory often tight

- FastMerge additional features:
  - Reorganize how the baskets are laid out on the file

- And could be improve to:
  - Delay, coalesce or even distribute the actual writing