Writing ROOT Data in Parallel with TBufferMerger

Guilherme Amadio[1,*], *Philippe* Canal[2], *Enrico* Guiraud[1], and *Danilo* Piparo[1]

[1]CERN, Geneva, Switzerland
[2]Fermi National Accelerator Laboratory, Batavia, IL, USA

Abstract. Experiments at the Large Hadron Collider (LHC) produce tens of petabytes of new data in ROOT format per year that need to be processed and analysed. In the next decade, following the planned upgrades of the LHC and its detectors, this data production rate is expected to increase at least ten-fold. Therefore, optimizing the ROOT I/O subsystem is of critical importance to the success of the LHC physics programme. This contribution presents ROOT's approach for writing data from multiple threads to a single output file in an efficient way. Technical aspects of the implementation—the `TBufferMerger` class—and programming model examples are described. Measurements of runtime performance and the overall improvement relative to the case of serial data writing are also discussed.

1 Introduction

Most of the tens of petabytes of new data produced each year at the Large Hadron Collider (LHC) is stored in ROOT's file format [1]. At the current rate, this translates to about 30 petabytes of ROOT data per year [2] that needs to be stored by physicists for later analysis. In the next decade, a sudden increase of this rate to hundreds of petabytes is expected at the beginning of Run 3. Since the amount of computational resources required to analyse data is proportional to the total volume of data to be analysed, it has become clear that under the flat or mildly increasing budget prospects for the coming years we will suffer from a shortage of computational resources if no measures are taken to optimize all HEP software. ROOT has a central role in HEP data analysis and as such has undergone efforts in several fronts to increase data processing performance. Some of these efforts have been discussed before in [3]. Here we shall focus on the latest developments on parallel data writing with ROOT using the `TBufferMerger` class.

2 The `TBufferMerger` Class

The `TBufferMerger` class, introduced in ROOT 6.10, allows users to write ROOT data in parallel to a single output file. Two use cases inspired the development of `TBufferMerger`: parallel writing of detector simulation and track reconstruction data, and parallel processing of existing datasets with ROOT's `RDataFrame`, for example, when applying filters, creating derived quantities, and then saving the results to a new file for further processing.

[*]e-mail: amadio@cern.ch

Since users are already familiar with ROOT's interfaces for creating and filling ROOT datasets, the main concern when designing `TBufferMerger` was to not diverge too much from existing interfaces in order to minimize the amount of changes required to convert sequential code to write data in parallel using the new class. ROOT already has the `TMemFile` class that can be used to store files in memory, and the `TFileMerger` class to merge files on disk. In the initial implementation of `TBufferMerger`, we adapted `TMemFile` to write data into a queue that is then merged by `TFileMerger` in a background thread into the output file on disk. The diagram in Figure 1 shows how this works in practice. The red dot represents a sentinel indicating that the output file is complete and should be closed.

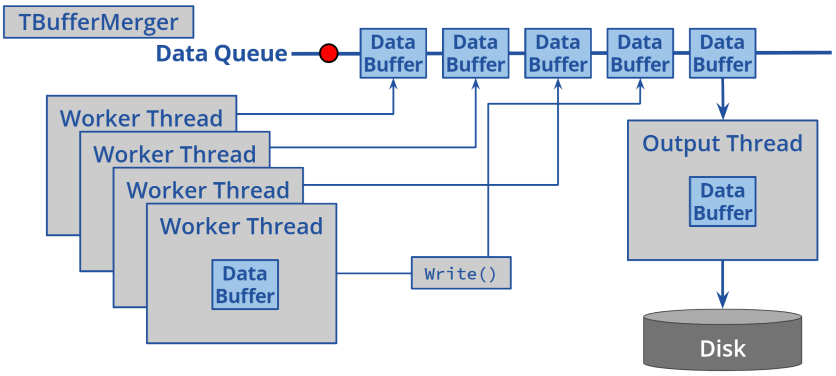

Figure 1: Initial implementation of `TBufferMerger`. Worker threads place data in the queue, while the output thread merges buffers one by one into the output file.

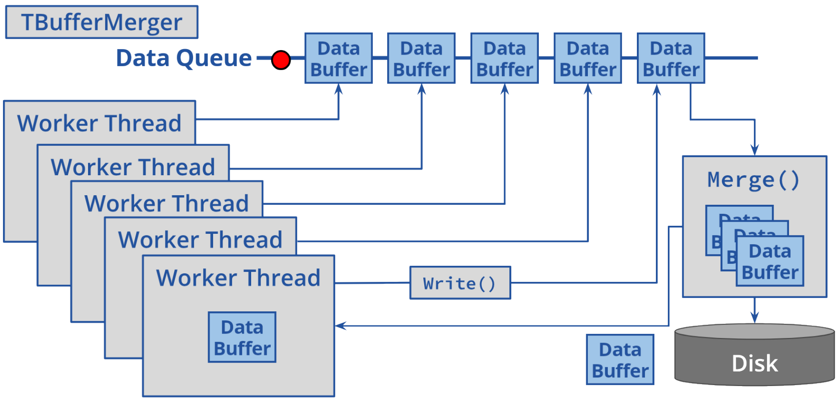

Figure 2: Current implementation of `TBufferMerger`. Worker threads/tasks place data in the queue. Upon writing its own data to the queue, each thread or task checks if a merge should be triggered and performs the merge if necessary, without spawning new threads or tasks in the background.

After being integrated into ROOT and tested by CMS for writing out reconstruction data in parallel [4], it became clear that despite the performance gains, the choice of using a background thread led to some inconveniences, like oversubscription of machines where reconstruction jobs were running within the CMSSW framework. Therefore, the `TBufferMerger` class has undergone some improvements to integrate better with experiment's frameworks by being agnostic about the user's choice for parallelism. In the current implementation, shown schematically in Figure 2, workers (threads or tasks) place data in the queue as before, but merging is no longer performed by a background thread. It is now performed by the workers themselves upon demand—that is, when a worker writes to the queue, it checks if a merge should be triggered, then performs the merge and returns to the caller if necessary. If a merge is already in progress, the worker appends its data to the queue and continues without blocking. Any data buffers remaining in the queue at the end (i.e. after all workers are done) are merged by the main thread before closing the output file.

3 Programming Model

To discuss the programming model for `TBufferMerger` we use a simple example for generating Pythia events in parallel and saving to a ROOT file. Assuming we have a function `InitPythia()` to initialize the Pythia generator, we can implement a function to generate and save events to a given ROOT file as shown on the left of Listing 1. An adaptation of this function to become a worker for use with the `TBufferMerger` class is then shown on the right of Listing 1 for comparison. Instead of a plain `TFile`, we call `GetFile()` on `TBufferMerger`, and the rest remains virtually the same as before. Listing 2 shows an example of how to create workers from the function shown in Listing 1 using standard C++ threads. ROOT's own `TTaskGroup` class could be used to create workers using tasks instead, with minor modifications.

```cpp
#include <Pythia8/Pythia.h>

#include <TFile.h>
#include <TTree.h>


void InitPythia(Pythia8::Pythia*);

void pythia_tfile(TFile* f,
                  size_t nEvents)
{
   Pythia8::Pythia pythia;
   InitPythia(&pythia);
   Pythia8::Event *e = &pythia.event;

   f->cd();
   TTree t("pythia", "pythia");

   t.Branch("event", &e);

   for (size_t n = 0; n < nEvents; ++n) {
      while (!pythia.next()); t.Fill();
   }

   f->Write();
};
```

```cpp
#include <Pythia8/Pythia.h>

#include <ROOT/TBufferMerger.hxx>
#include <TTree.h>

using namespace ROOT::Experimental;

void InitPythia(Pythia8::Pythia*);

void pythia_tbm(TBufferMerger* m,
                size_t nEvents)
{
   Pythia8::Pythia pythia;
   InitPythia(&pythia);
   Pythia8::Event *e = &pythia.event;

   auto f = m->GetFile();
   TTree t("pythia", "pythia");

   t.Branch("event", &e);

   for (size_t n = 0; n < nEvents; ++n) {
      while (!pythia.next()); t.Fill();
   }

   f->Write();
};
```

Listing 1: Pythia event generation using `TFile` (left) and `TBufferMerger` (right).

The performance of the Pythia example code just presented is shown in Figure 3 below. In this example, output is written to a solid-state disk (SSD), and the version of ROOT used is a preview of version 6.16. Scaling is nearly ideal up to the number of physical cores of the machine, but has only marginal gains with hyper-threading.

```
#include "ROOT/TBufferMerger.hxx"
#include "TROOT.h"

#include <thread>
#include <vector>

using namespace ROOT::Experimental;

void pythia_tbm(TBufferMerger *m, size_t nEvents);

int main(int argc, char **argv)
{
   size_t nEvents  = 8192;
   size_t nWorkers = std::thread::hardware_concurrency();

   std::string filename("pythia.root");

   if (argc >= 2) nWorkers = std::atoi(argv[1]);
   if (argc >= 3) nEvents  = std::atoi(argv[2]);
   if (argc >= 4) filename = argv[3];

   size_t nEventsPerWorker = nEvents/nWorkers;

   gROOT->SetBatch();

   if (nWorkers > 1)
      ROOT::EnableImplicitMT(nWorkers);

   TBufferMerger merger(filename.c_str(), "recreate");

   std::vector<std::thread> workers;
   for (size_t i = 0; i < nWorkers; ++i)
      workers.emplace_back(pythia_tbm, &merger, nEventsPerWorker);

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

   return 0;
}
```

Listing 2: Example of how to create workers for `TBufferMerger` using C++11 threads.

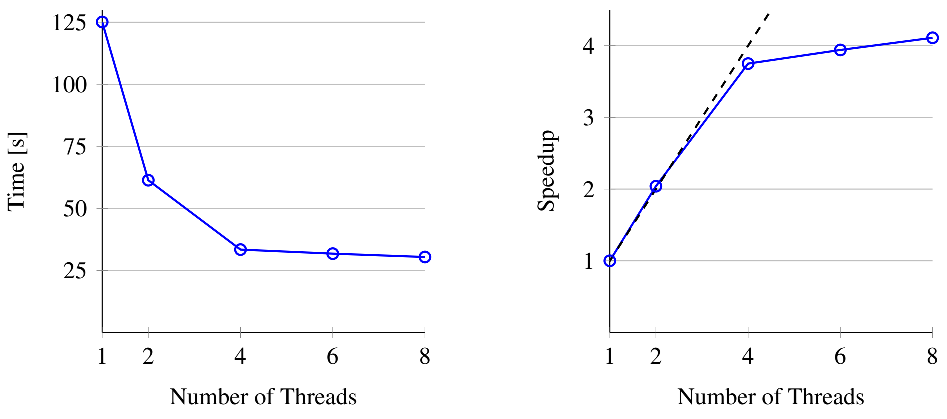

Figure 3: Performance of Pythia event generation example on a quad-core desktop machine (Intel Core i7 6700 CPU at 3.4GHz).

4 Parallel Data Writing Benchmarks

In this section we present performance benchmarks using `TBufferMerger` in various scenarios. All benchmarks have been run on a machine with an Intel® Core™ i7-7820X processor (8 cores, 11M Cache, up to 4.30 GHz) and 32GB of RAM, where a release build of ROOT has been compiled with GCC 8.1 using the native instruction set (Skylake AVX-512). The output disks used were a WD Black Hard Disk Drive (1TB), and a Samsung Evo 960 NVMe SSD (256GB). For comparison, we have also tested writing out to system memory (using an output file in tmpfs).

The first benchmark consists in creating a dataset of ≈1GB and saving it in parallel using different compression algorithms and output media. The dataset contains a single column of data with randomly generated floating point numbers in the range [0, 1]. The LZ4 compression algorithm is the fastest but produces larger files than ZLIB and LZMA. The ZLIB algorithm has a good balance between compression ratio and compression speed, and the LZMA algorithm has the best compression ratio but has also the slowest compression speed. Results are shown in Figure 4. When using LZMA compression, the benchmark is CPU-bound, and shows nearly ideal scaling. In other configurations, performance is limited by the output bandwidth.

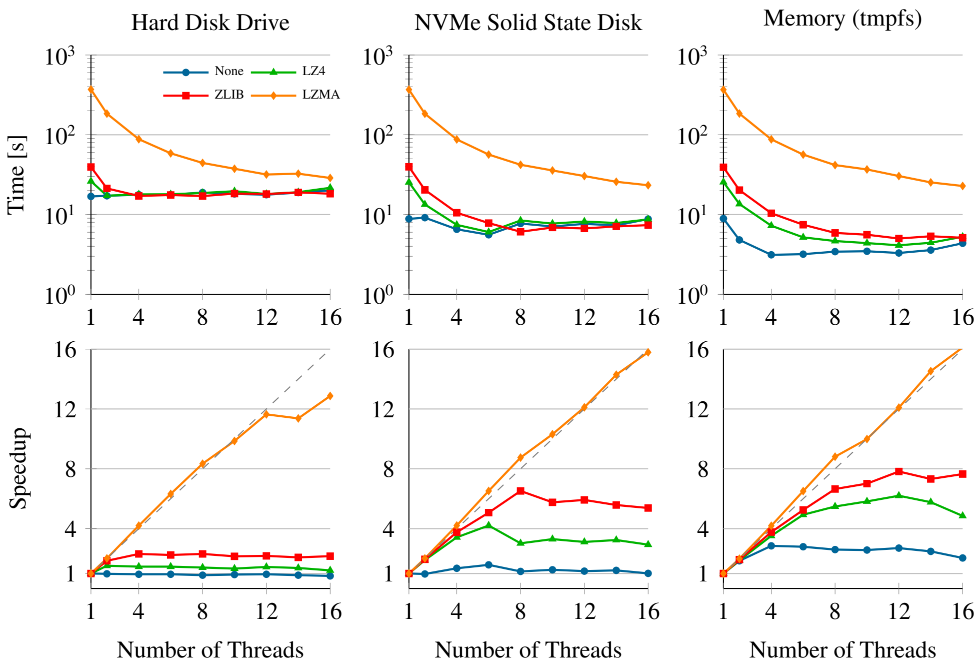

Figure 4: Run time and speedup for creating and saving ≈1GB of randomly generated floating point numbers in various configurations.

In the next benchmark we increase the role of serialisation in the I/O subsystem by generating a more complex dataset where each column consists of an `std::vector<Event>`, where `Event` is a class type that itself consists of 3 vectors (3 double precision numbers) and 6 other numbers (for a total of $3 \times 3 + 3 = 12$ doubles + 3 integers). In each run, the number of columns and the number of elements in each vector is varied. In Figure 5, we compare different compression algorithms and output media like in the previous benchmark, but this time using a dataset containing 10 columns where each entry is a vector with 10 randomly generated `Event`s as elements.

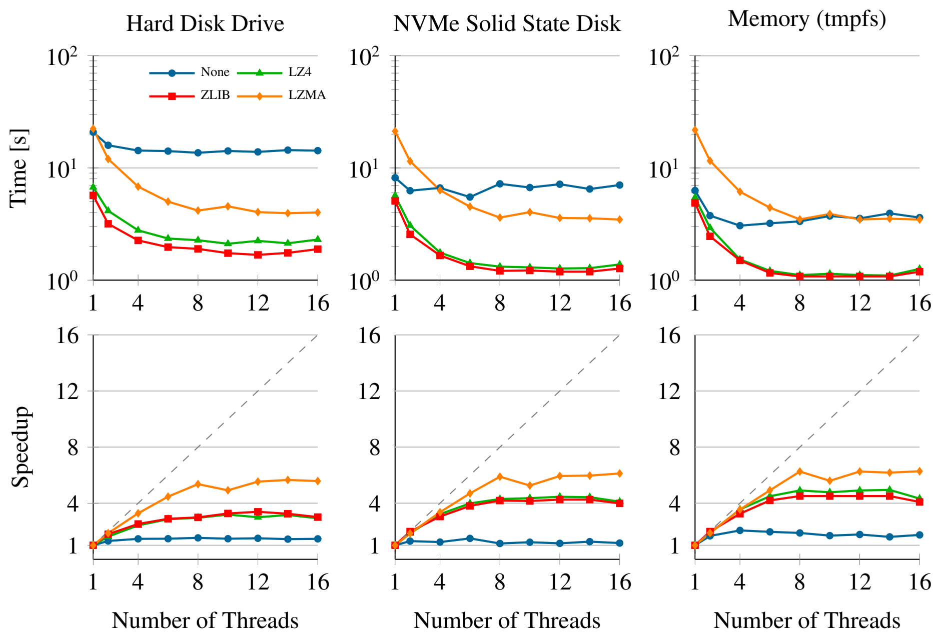

Figure 5: Run time and speedup for creating and saving ≈1GB of randomly generated `std::vector<Event>` in various configurations. Each dataset has 10 columns with vectors of 10 elements as entries.

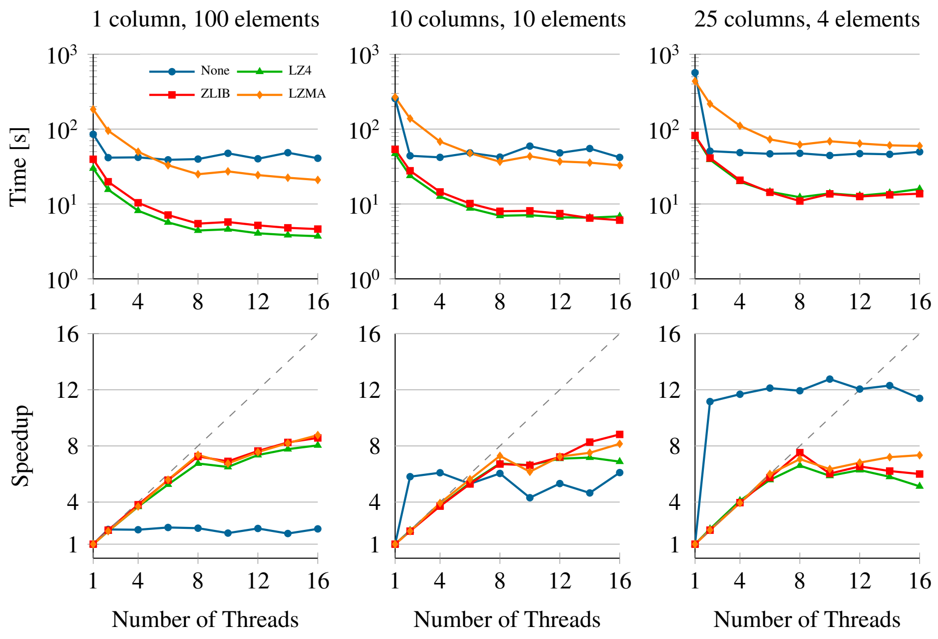

Figure 6: Run time and speedup for creating and saving ≈10GB of randomly generated `std::vector<Event>`. Datasets have different numbers of columns and entries, but the total amount of data is kept constant. Only the NVMe solid state disk has been used for data output.

In the benchmarks of Figure 5, implicit multi-threading (IMT) has been disabled in ROOT. Parallelism is achieved via multiple threads or tasks writing to `TBufferMergerFile`s, without noticeable differences when using standard threads or TBB tasks for the workers. The time necessary to flush disk buffers after each run is also negligible compared with the run time of each benchmark.

Figure 6 shows that the difference in run time is no longer limited by the bandwidth of the output media, even if it is visibly affected by it. A noticeable difference from the previous benchmark is that LZMA compressed data can be written out to disk faster than uncompressed data due to the higher compression ratio. In all benchmarks, no big difference in performance is observed between ZLIB and LZ4 compression algorithms. In Figure 6, output is written only into the Samsung Evo 960 NVMe SSD, and the flushing strategy is different than before. Per worker, each 1% of data generated causes a flush event (compression of memory buffers), and each 10% causes data to be written out from memory buffers to disk. Implicit multi-threading remains disabled, and the size of the generated dataset is increased ten-fold, to about 10GB. The number of data columns is varied along with the number of elements in each vector in such a way that the total amount of generated data remains constant. With the new approach for flushing and saving the data, scaling is improved, but only up to the number of physical cores available on the machine.

5 Conclusion

We presented the `TBufferMerger` class introduced in ROOT 6.10 and used it to characterize the performance of ROOT's I/O subsystem in various scenarios. We have shown that strong scaling can be nearly ideal when the workload is CPU-bound, but that the cost of serialization can become a bottleneck when writing out more complex datasets. In particular, writing out uncompressed datasets has poorer scaling in part due to the higher volume of data, but also due to the heavier relative weight of I/O operations given the lack of compression. In realistic workloads, such as simulation and reconstruction, ROOT's I/O subsystem is unlikely to become a performance bottleneck. In CMS reconstruction jobs, for instance, the event processing rate drops only by 1.3% when writing out results to disk compared with not writing anything (4.59 events/s vs 4.65 events/s) [4]. Performance issues identified by these benchmarks (lock contention during accesses to ROOT's type system) have been fixed and will be available in the upcoming release of ROOT, 6.16/00.

References

[1] R. Brun, F. Rademakers, *ROOT – An Object Oriented Data Analysis Framework (see also ROOT [software], Release v6.16/00)*, in *New computing techniques in physics research V. Proceedings, 5th International Workshop, AIHENP '96, Lausanne, Switzerland* (1996)

[2] *CERN Computing Page*, https://home.cern/about/computing

[3] G. Amadio, B.P. Bockelman, P. Canal, D. Piparo, E. Tejedor, Z. Zhang, Journal of Physics: Conference Series **1085**, 032014 (2018)

[4] D. Riley, *CMS and ROOT I/O*, in *ROOT I/O Workshop* (2018), `https://indico.cern.ch/event/715802/contributions/2942558`

[5] *CMSSW Framework*, `https://github.com/cms-sw/cmssw`

[6] J. Blomer, Journal of Physics: Conference Series **1085**, 032020 (2018)