Parallel Writing of Nested Data in Columnar Formats

Jonas Hahnfeld[1,2] Jakob Blomer[1] Thorsten Kollegger[2]
jonas.hahnfeld@cern.ch

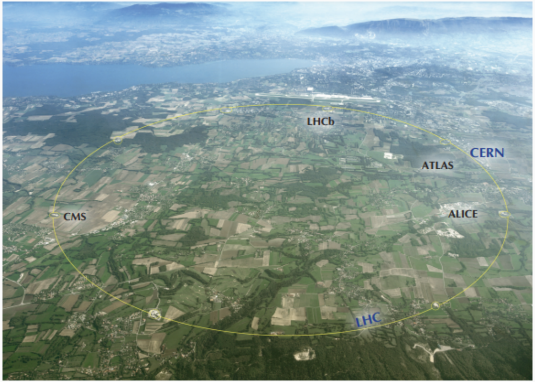
[1] CERN, Geneva, Switzerland

[2] Goethe University Frankfurt, Institute of Computer Science, Frankfurt, Germany

Euro-Par 2024 – August 29, 2024

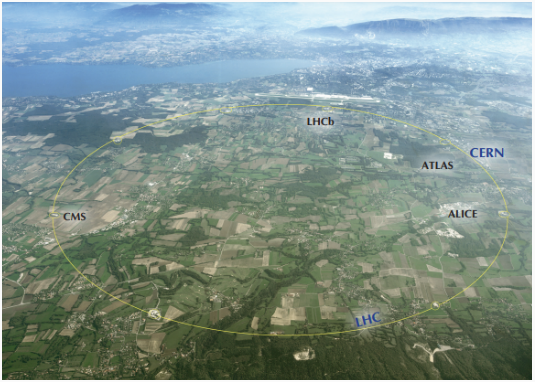
Motivation

- Large Hadron Collider at CERN
  - More than 2 exabytes since 2010
  - Stored in binary columnar format



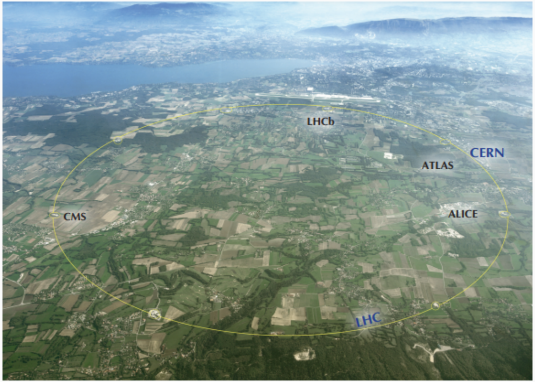
Motivation

- Large Hadron Collider at CERN
  - More than 2 exabytes since 2010
  - Stored in binary columnar format
- High-Luminosity LHC (from 2029)
  - Further increase of data rate



Motivation

- Large Hadron Collider at CERN
  - More than 2 exabytes since 2010
  - Stored in binary columnar format
- High-Luminosity LHC (from 2029)
  - Further increase of data rate
- RNTuple: evolution of the currently used TTree columnar format
  - Opportunity to include parallel writing from the start



Outline

RNTuple Overview

Concepts for Parallel Writing of Columnar Data

Evaluation of Parallel RNTuple Writing

Conclusions and Future Work

RNTuple Overview

RNTuple Overview

- Serialize acyclic C++ data structures into a columnar format
  - Important to support nested collections for HEP use cases

```
struct Event {
  int fId;
  std::vector<Track> fTracks;
};
```

```
struct Track {
  float fEnergy;
  std::vector<int> fIds;
};
```

Figure 1: Simplified example of nested data structures. Real-world HEP data models often have thousands of fields.

RNTuple Overview

- Recursively decompose data structures into *fields*
  - *Columns* at leafs of field tree: primitive, fixed-size types

RNTuple Overview

- Recursively decompose data structures into *fields*
  - *Columns* at leafs of field tree: primitive, fixed-size types

- Columns partitioned into *pages*
  - Transparently compressed (default for RNTuple: Zstandard)

RNTuple Overview

- Recursively decompose data structures into *fields*
  - *Columns* at leafs of field tree: primitive, fixed-size types
- Columns partitioned into *pages*
  - Transparently compressed (default for RNTuple: Zstandard)
- *Cluster:* all pages of a consecutive range of rows, or *entries*

RNTuple Overview

Table 1: Example of columnar representation for the nested data structure shown in Figure 1.

fId	fTracks	fTracks._0.fEnergy	fTracks._0.fIds	fTracks._0.fIds._0
6873	2	25.4f	2	42
				27
		32.8f	3	16
6874	3	14.7f	5	21
				8
⋮	⋮	⋮	⋮	⋮

RNTuple Overview

Table 1: Example of columnar representation for the nested data structure shown in Figure 1.

`fId`	`fTracks`	`fTracks._0.fEnergy`	`fTracks._0.fIds`	`fTracks._0.fIds._0`
6873	2	25.4f	2	42
				27
		32.8f	3	16
6874	3	14.7f	5	21
				8
⋮	⋮	⋮	⋮	⋮

entry

column

RNTuple Overview

Table 1: Example of columnar representation for the nested data structure shown in Figure 1.

fId	fTracks	fTracks._0.fEnergy	fTracks._0.fIds	fTracks._0.fIds._0
6873	2	25.4f	2	42
				27
		32.8f	3	16
6874	3	14.7f	5	21
				8
⋮	⋮	⋮	⋮	⋮

page

entry

column

RNTuple Overview

Table 1: Example of columnar representation for the nested data structure shown in Figure 1.

fId	fTracks	fTracks._0.fEnergy	fTracks._0.fIds	fTracks._0.fIds._0
6873	2	25.4f	2	42
				27
		32.8f	3	16
6874	3	14.7f	5	21
				8
⋮	⋮	⋮	⋮	⋮

page

entry

cluster

column

Concepts for Parallel Writing of Columnar Data

Concepts for Parallel Writing of Columnar Data

- Support for nested data implies variable row sizes
  - Transparent compression leads to variable page sizes

  $\rightarrow$ Data cannot be organized in a regular grid

Concepts for Parallel Writing of Columnar Data

- Support for nested data implies variable row sizes
  - Transparent compression leads to variable page sizes

  $\rightarrow$ Data cannot be organized in a regular grid

- Clusters are *relocatable*
  - Serialization and compression without synchronization
  - Then: final size is known, can be written into binary file format

Concepts for Parallel Writing of Columnar Data

- Support for nested data implies variable row sizes
  - Transparent compression leads to variable page sizes

  $\rightarrow$ Data cannot be organized in a regular grid

- Clusters are *relocatable*
  - Serialization and compression without synchronization
  - Then: final size is known, can be written into binary file format

- Metadata updated in critical section

Concepts for Parallel Writing of Columnar Data

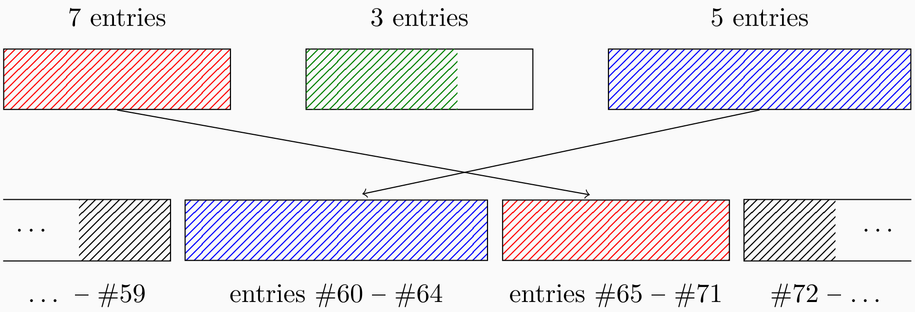


Figure 2: Illustration of filling three RNTuple clusters in parallel. After buffering in memory (top), entries are appended into a sequential file (bottom).

Evaluation of Parallel RNTuple Writing

Writing to SSD

- Evaluate weak scaling behavior for a synthetic benchmark
  - Fixed number of 20 million entries per thread
  - Two top-level fields: "event ID" and a vector of `floats`

Writing to SSD

- Evaluate weak scaling behavior for a synthetic benchmark
  - Fixed number of 20 million entries per thread
  - Two top-level fields: “event ID” and a vector of `floats`
- Write data on Samsung PM1733 NVMe SSD formatted with `ext4`
  - Measured bandwidth with Flexible I/O Tester (`fio`): 768 MB/s

Writing to SSD

- Evaluate weak scaling behavior for a synthetic benchmark
  - Fixed number of 20 million entries per thread
  - Two top-level fields: "event ID" and a vector of `floats`
- Write data on Samsung PM1733 NVMe SSD formatted with `ext4`
  - Measured bandwidth with Flexible I/O Tester (`fio`): 768 MB/s
  - Possible optimization: pre-allocate file with `fallocate`
  - Increases bandwidth to 1062 MB/s

Writing to SSD

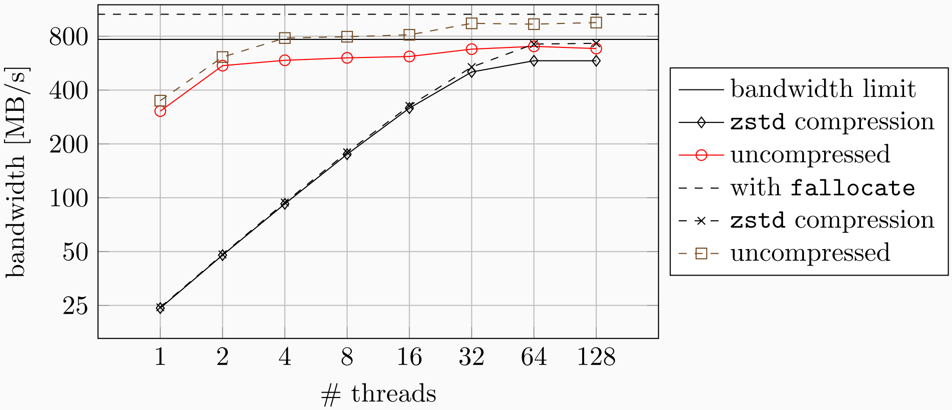


Figure 4: Bandwidth measured with the synthetic benchmark writing on a server SSD.[1]

[1] Figure adapted from the paper, see Backup Slide 3 for the original plot.

Dataset Skimming of the Analysis Grand Challenge

- Analysis Grand Challenge: test workflows at scales required for HL-LHC
  - "ttbar" analysis, input dataset derived from 2015 Open Data of the CMS experiment
  - Conversion to RNTuple: 969 GB across 787 files, divided into nine partitions

Dataset Skimming of the Analysis Grand Challenge

- Analysis Grand Challenge: test workflows at scales required for HL-LHC
  - "ttbar" analysis, input dataset derived from 2015 Open Data of the CMS experiment
  - Conversion to RNTuple: 969 GB across 787 files, divided into nine partitions

- Evaluation benchmark: reduce size of the dataset by *skimming*
  - Retain only fields used by analysis, filter events (entries) based on coarse cuts
  - Output: nine files, total size of 19 GB

Dataset Skimming of the Analysis Grand Challenge

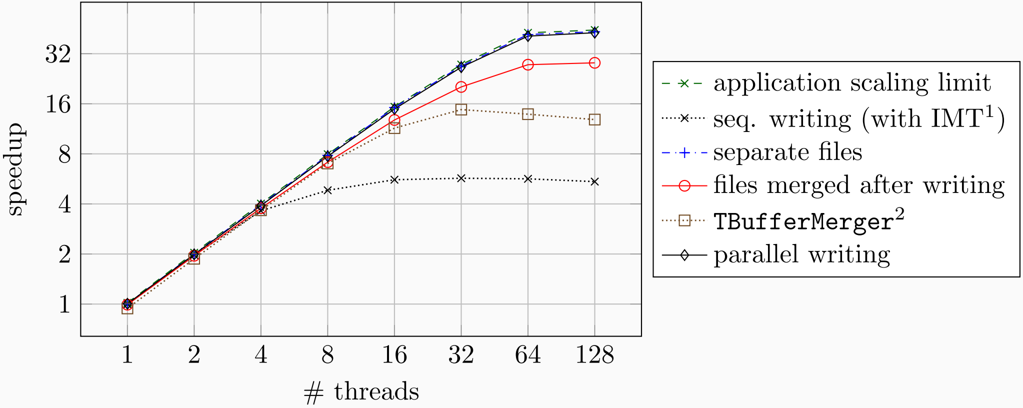


Figure 6: Speedup of the AGC dataset skimming benchmark compared to a full sequential run.

[1] ROOT's *implicit multithreading* (IMT) used to parallelize compression of pages.
[2] `TBufferMerger` merges files in memory. It is the current way of parallel writing in ROOT.

Conclusions and Future Work

Conclusions and Future Work

- Presented concept for parallel writing of nested data in columnar formats
- Implemented for RNTuple: https://github.com/root-project/root/
- Evaluation with synthetic benchmark and application of dataset skimming

Conclusions and Future Work

- Presented concept for parallel writing of nested data in columnar formats
- Implemented for RNTuple: `https://github.com/root-project/root/`
- Evaluation with synthetic benchmark and application of dataset skimming

- Synthetic benchmark scales up to storage bandwidth limit
  - Plan to investigate usage of Direct I/O

  $\rightarrow$ Presentation at CHEP 2024 in October

Conclusions and Future Work

- Presented concept for parallel writing of nested data in columnar formats
- Implemented for RNTuple: `https://github.com/root-project/root/`
- Evaluation with synthetic benchmark and application of dataset skimming

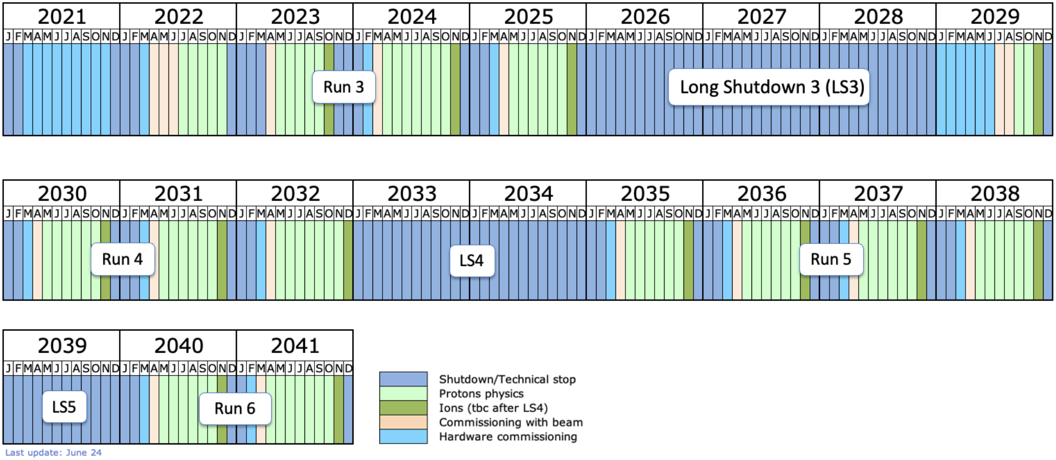
- Synthetic benchmark scales up to storage bandwidth limit
  - Plan to investigate usage of Direct I/O

  $\rightarrow$ Presentation at CHEP 2024 in October

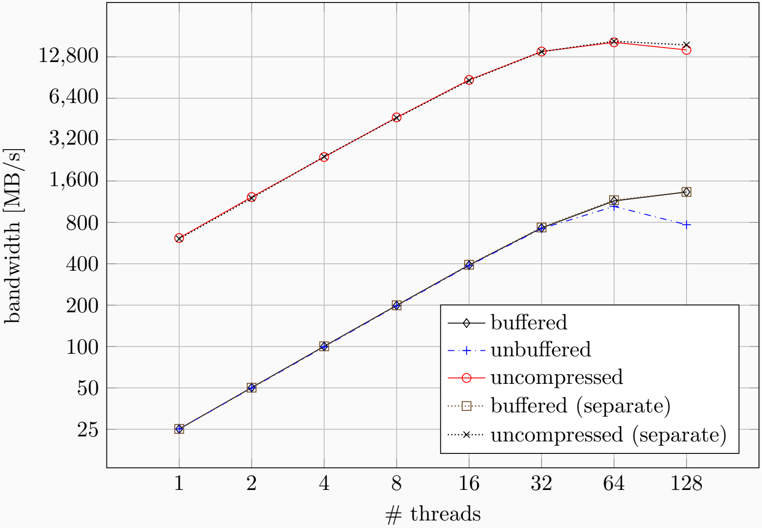
- Extend parallel writing to process-level parallelism
- Distributed parallelism with MPI and writing to cluster filesystems

Backup Slides

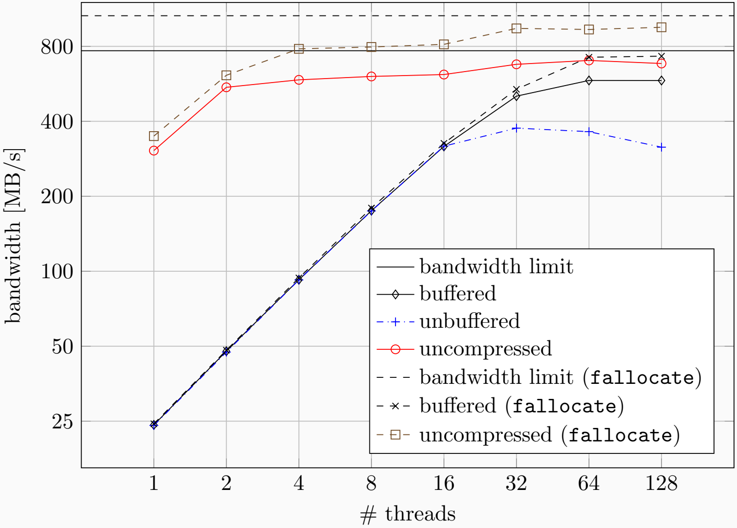
LHC Long Term Schedule



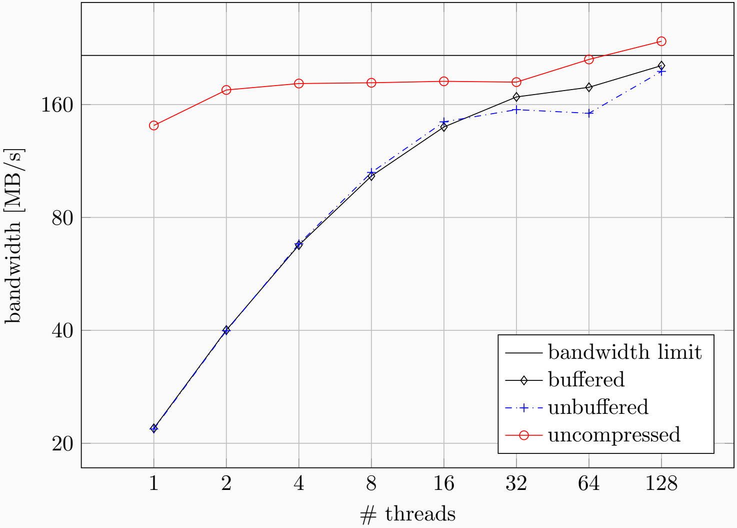
Writing to /dev/null



Writing to SSD (original plot from the paper)



Writing to HDD



Direct I/O – Writing to SSD, no compression – preliminary