RDataFrame Thread Scaling Optimizations etc

Josh Bendavid (CERN)
with input from
M. Cipriani, M. Dünser, K. Long (CERN)
E. Manca (INFN/SNS)

Mar. 18, 2021
ROOT PPP Meeting

Introduction

- Working on precision W measurements in CMS
- This analysis has some special considerations driven by the large inclusive W cross section:
  - O(1B) data and Monte Carlo events with little scope for skimming
- for this type of analysis **HL-LHC is now**
- Two somewhat different analysis frameworks in use, but both using RDataFrame underneath
- Today:
  - Optimizations to improve thread scaling of RDataFrame
  - Some benchmarks
  - Discussion on N-dimensional histograms

Typical Analysis and Computing Workflow for CMS analysis in CMG

- CMS Offline and Computing project centrally produces Monte Carlo and reconstructed data on the grid in two data-tiers typically used for analysis
  - MINIAOD: O(30kB)/event, Root files CMS reconstruction object structure
  - NANOAOD: O(1kB)/event, Root files with flat TTrees
- NANOAOD or other flat trees may also be produced privately on the grid
- NANOAOD or other flat trees are then used $\sim$ directly to fill histograms for plotting and/or statistical analysis in searches and measurements
  - Using RDataFrame in our case (but others are using custom C++ event loops/TSelector/Uproot+Awkward+Coffea/etc)
- NANOAOD produced on the grid tends to be split into 100s or thousands of small files (O(1GB))
  - Can be merged into larger ones, but this adds an extra step

Auxiliary Workflows

- Several other steps are needed for a complete analysis
  - Object corrections and calibrations, often involving complex maximum likelihood fits for derivation and/or validation
  - Statistical analysis with very large/complex maximum likelihood fits
  - Training and inference of machine learning models
  - Likelihood fits already using statistical tools with Tensorflow and/or Jax backends and are ready to run on GPUs
- Some of the corrections/calibration workflows are also using RDF

Computing Resources

- While some other steps are performed on the grid or on batch systems, RDF analysis of NANOAOD or custom trees in our case is done multithreaded on a single machine
- In the CERN CMS group we are using a 32 core/64 thread machine with fast local SSD (many SATA SSDs in raid0)
- Pisa CMS group has a 128 core/256 thread machine with several NVME SSD's
- **Motivation to ensure good thread scaling of RDF**

Thread Scaling

- Immediately saw serious issues scaling RDF up to 256 threads with a large number of input files
- Looking at stack traces revealed this to be mostly due to use of the global ROOT lock in operations associated with opening/closing files and creating the TChains for each of the tasks in TTreeProcessorMT
- Leading to "lock convoying" and extreme slowdown of processing

Thread Scaling Improvements: Task Splitting

- Task splitting in TTreeProcessorMT was not optimal for a large number of input files
- Previously could be tuned with TTreeProcessorMT::SetMaxTasksPerFilePerWorker (default 24), but even if set to 1, this would create a huge number of tasks in case of many input files and many thread $\rightarrow$ more frequent file open/close and TChain initialization
- Fixed by `https://github.com/root-project/root/pull/7106` which replaces this with SetMaxTasksPerWorkerHint, which takes into account the number of files in determining the task splitting (targets 24 tasks per thread in total by default)
- Tasks cannot currently group more than one file (though it's unclear this would really help since the subsequent files would still need to be opened)

Thread Scaling Improvements: Lock Contention

- Identified hotspot in TUrl::GetSpecialProtocols
  - Improved by Axel with `https://github.com/root-project/root/pull/6857`
  - Global lock was being used to protect some initialization on first call, replaced with an atomic_bool

Thread Scaling Improvements: Lock Contention

- Identified hotspot in TBufferFile::ReadClassBuffer
  - Improved with `https://github.com/root-project/root/pull/7105`
  - Replaces use of global lock with explicit use of global read and write locks
  - Write locks in this function are needed much less commonly than read locks
  - For a small test with 9.7M events in 19 input files on 256 threads, **runtime goes from 25 minutes to 19 seconds** (improvements are enough to avoid the severe lock convoying)
  - (Also fixes an uninitialized variable in global read write lock which could cause problems if not zero)

Thread Scaling Improvements: Lock Contention

- The default global read write lock in ROOT is currently

```
ROOT::gCoreMutex = new ROOT::TRWMutexImp<std::mutex,
                       ROOT::Internal::RecurseCounts>();
```

- In this implementation, the global std::mutex still needs to be locked (very briefly) when a thread takes the read lock
- There is another implementation using thread local storage which avoids this, but this was observed to cause deadlocks (see `https://github.com/root-project/root/commit/bd1894b2bc8306c789557ddf12266c9af6259046`)
- Implementation using Threaded Building Blocks thread local storage which should resolve this in `https://github.com/root-project/root/pull/7260` (still under discussion)
- This gives a further $\sim$10% improvement in the previous benchmark
- May be possible to also accomplish this without TBB with an appropriate use of C++11 thread_local keyword (being checked)

Thread Scaling Improvements: Lock Contention

- Overall situation is greatly improved
- Remaining use of global lock in RDF processing during file open/close, mainly related to global TList and other collection objects

https://github.com/root-project/root/blob/4d18ee14a6b91b1f01acbb3ef303f8a694f1bad1/io/io/src/TFile.cxx#L529
https://github.com/root-project/root/blob/4d18ee14a6b91b1f01acbb3ef303f8a694f1bad1/io/io/src/TFile.cxx#L814
https://github.com/root-project/root/blob/4d18ee14a6b91b1f01acbb3ef303f8a694f1bad1/io/io/src/TFile.cxx#L962
https://github.com/root-project/root/blob/4d18ee14a6b91b1f01acbb3ef303f8a694f1bad1/core/cont/src/TList.cxx#L158
https://github.com/root-project/root/blob/4d18ee14a6b91b1f01acbb3ef303f8a694f1bad1/core/cont/src/TList.cxx#L404

Thread Scaling Improvements: Lock Contention

- Overall situation is greatly improved
- Remaining use of global lock in RDF processing during file open/close, mainly related to global TList and other collection objects

https://github.com/root-project/root/blob/4d18ee14a6b91b1f01acbb3ef303f8a694f1bad1/core/cont/src/THashList.cxx#L86
https://github.com/root-project/root/blob/4d18ee14a6b91b1f01acbb3ef303f8a694f1bad1/core/cont/src/THashList.cxx#L97
https://github.com/root-project/root/blob/4d18ee14a6b91b1f01acbb3ef303f8a694f1bad1/core/cont/src/THashList.cxx#L191
https://github.com/root-project/root/blob/4d18ee14a6b91b1f01acbb3ef303f8a694f1bad1/core/cont/src/THashList.cxx#L209
https://github.com/root-project/root/blob/4d18ee14a6b91b1f01acbb3ef303f8a694f1bad1/core/cont/src/THashTable.cxx#L169

Benchmark: Fill Histograms from NANOAD

- "Small" benchmark, fill 9 TH3D's from 513M Monte Carlo events in NANOAOD format to compute angular coefficients for W production using **RDataFrame** with implicit multithreading (this is a small subset of the real analysis)
- 477G total, 63GB in read branches with LZMA (compression algo comparison in backup)
- Have been working with Root authors to improve thread scaling, so this includes recent and WIP developments as part of this effort
  - `https://github.com/root-project/root/pull/6857`
  - `https://github.com/root-project/root/pull/6919`

CPU	Storage	CPU Eff.	Time
2 x Xeon (32C/64T)	Local SSD (2xSATA)	0.92	8m45s
2 x Xeon (32C/64T)	eos/xrootd (10gbps)	0.67	10m36s
2 x Xeon (32C/64T)	CephFS HDD (10gbps)	0.83	9m27s
2 x Xeon (32C/64T)	CephFS SSD (10gbps)	0.88	9m07s
2 x EPYC (128C/256T)	Page Cache	0.77	2m58s

Further improvements to thread scaling and eos usage likely possible

Factor of 3 wall time improvement already possible with larger machine

Benchmark: Aggregate Gradients for Muon Calibrations

- Muon momentum scale calibrations are based on a high granularity correction for B-field, material and alignment residuals, approaching complexity of full tracker alignment
- In large debugging version currently being used for R&D, ∼130k parameters, 7TB of flat trees
- **Gradients must be aggregated into a single large** (130GB) **matrix in memory → parallelization on batch not straightforward**
- Using RDataFrame with multithreading and std::atomics for aggregation

CPU	Storage	Time	Avg. Rate (GBytes/sec)
2 x Xeon (32C/64T)	eos/xrootd (25gbps)	17m55s	1.3
2 x Xeon (32C/64T)	Local SSD (16xSATA)	14m35s	1.6

Numbers for a subset of input: 1.4TB (>98% in read branches)

- Need to set e.g. XRD_PARALLELEVTLOOP=16 to get good eos performance
- Thread scaling limited by spinlock contention, improvements possible

Benchmark: Gradient Aggregation: IO Limits

- Taking the same workflow and running with tight selection of tracks on top of the large trees → reduced CPU load → **workflow becomes largely IO limited**

CPU	Storage	Avg. Rate (GBytes/sec)
2 x Xeon (32C/64T)	eos/xrootd (eoscms) (25gbps)	1.64
2 x Xeon (32C/64T)	eos/xrootd (test inst.) (25gbps)	2.62
2 x Xeon (32C/64T)	CephFS HDD (25gbps)	2.60
2 x Xeon (32C/64T)	CephFS SSD (25gbps)	2.64
2 x Xeon (32C/64T)	Local SSD (16xSATA)	5.21

thanks to IT-ST group for help setting up some of these tests

- Need to set e.g. XRD_PARALLELEVTLOOP=16 to get good eos performance
- EOS+xrootd standard production instance not quite scaling up to network limits (possible xrootd client/ROOT bottlenecks?)
- Extremely good performance of EOS test instance, and CephFS (CentOS 8 kernel client), approaching limits of ethernet connection
- **Reach 5.2GBytes/sec from local SSDs, approaching limits of disk array)**

Aside: fetch_add emulation for atomic floating point

- STL only implements fetch_add for floating point atomic specializations starting from C++20
- This can implemented relatively efficiently (at least in low contention scenarios) using atomic compare-exchange and a spinlock

```
std::atomic<double>& ref = (*grad_)[idx];
const double& diff = vec[k];
double old = ref.load();
double desired = old + diff;
while (!ref.compare_exchange_weak(old, desired))
{
    desired = old + diff;
}
```

Filling Histograms from NANOAOD: Scaling with number of Histograms

- Filling histograms from 1.37B W Monte Carlo Events with RDataFrame
- 809GB stored on local SSD, divided into 4374 files
- Using CMS default LZMA compression for now, other options under study

Number of Histograms	Wall Time	CPU Efficiency (32C/64T)
1	7m04s	0.82
51	11m55s	0.92
81	16m18s	0.94
101	19m02s	0.94

- One additional Define per histogram here
- Reasonable scaling with number of histograms (optimizations possible)
- CPU efficiency increases with task complexity

EOS+xrootd vs local SSD: More Realistic Use Case

- Filling histograms from CMS NANOAOD with RDataFrame
- 800M data + 200M Monte Carlo Events

Storage	Wall Time	CPU Efficiency (32C/64T)
EOS+xrootd (25gbps)	31m39s	0.20
Local SSD (16xSATA)	9m41s	0.92

- Behaviour of eos vs local SSD somewhat workload dependent
- Bottlenecks and scaling to be investigated
- **This machine has enough storage and network to effectively copy $\sim$ all relevant input data to local disk**

Multidimensional Histograms and Systematic Uncertainties

- Systematic uncertainties can be propagated either by a variation of event weights for MC histograms (e.g. PDF variations) or by a variation of the variables being histogrammed (e.g. momentum scale uncertainties)
- Consider a concrete example: Muon p_T and η filled in a TH2D for the central value
- PDF/α_S uncertainties in the form of 103 alternate event weights (defined a single vector-valued column)
- Naive solution with 100 Define and Histo2D calls:

```
hists = []
for i in range(103):
    rdftmp = rdf.Define(f"weight_{i}", f"pdfWeights[{i}]")
    h = rdftmp.Histo2D(model, "leadPt", "leadEta", f"weight_{i}"
    hists.append(h)
```

Multidimensional Histograms and Systematic Uncertainties

- In order to avoid 103 extra Define and Histo2D calls:

```
gInterpreter.Declare("""
RVec<unsigned int> indices(const RVec<float>& v) {
  RVec<unsigned int> res(v.size());
  std::iota(std::begin(res), std::end(res), 0);
  return res;
}
""")

rdf = rdf.Define("leadPtvec", "RVec<float>(pdfWeights.size(),
                  leadPt)")
rdf = rdf.Define("leadEtavec", "RVec<float>(pdfWeights.size(),
                  leadEta)")
rdf = rdf.Define("pdfIdxs", "indices(pdfWeights)")
h = rdf.Histo3D(model, "leadPtvec", "leadEtavec", "pdfIdxs",
                  "pdfWeights")
```

- n.b in a simple test with 119M DY events, this is about 5x faster (1m38s vs 7m55s)

Multidimensional Histograms and Systematic Uncertainties

- The indexed filling used here is a very special case:
  - The bins which are filled are sequential with respect to the global indexing and fill an entire "row" along the last axis
  - The bin index can be looked up only once (and no lookup needed for the last axis)
- Specially optimized vector Fill in principle possible (would touch both TH1 and RDF)
- General vectorized Fill also interesting (planned for Root 7?)

Multidimensional Histograms

- This type of pattern for systematics extend by 1 the number of histogram dimensions, and quickly motivates >3D histograms
- Implemented HistoND() call for RDF to fill THnD
  - `https://github.com/root-project/root/pull/7499`
- Also implements more flexible filling capabilities for Fill, Histo, Profile, to support arbitrary number of dimensions, and arbitrary mix of scalar and vector columns
  - Avoids need to define extra vector columns for scalar values in previous example
- Some technical issues to discuss (see PR description)
- Some simplifications possible with C++14/17 and/or supporting only indexed access to containers
  - **Nominal (C++11 and generic containers with iterators)**
  - **C++17 version**
  - **C++17 version supporting only index container access**
- n.b. Elisabetta has been using N-dimensional histograms instead with Boost Histograms and the Book() method with custom helper classes

Conclusions

- Good experience with RDataFrame so far
- Some optimizations were needed to maintain scaling up to 256 threads
- Higher dimensional histogram support is useful for several cases
- Some scaling issues with EOS+xrootd, and with large number of Defines/histograms

Backup

Benchmark 1: Compression Algorithms and IO vs CPU Limits

- ROOT supports several compression algorithms with tradeoff between compression ratio and (de)compression speed, see **detailed comparisons**
- Different algorithms make IO vs CPU constraints more or less relevant

Algorithm (level)	Total Size (GB)	Size of read branches (GB)
LZMA (9)	477	63
ZSTD (5)	680	86
LZ4 (4)	967	116

Benchmark 1: Compression Algorithms and IO vs CPU Limits

CPU	Storage	Comp.	CPU Eff.	Time
2 x Xeon (32C/64T)	Local SSD (2xSATA)	LZMA	0.92	8m45s
2 x Xeon (32C/64T)	Local SSD (2xSATA)	ZSTD	0.75	7m11s
2 x Xeon (32C/64T)	Local SSD (2xSATA)	LZ4	0.70	7m17s
2 x Xeon (32C/64T)	eos/xrootd (10gbps)	LZMA	0.67	10m36s
2 x Xeon (32C/64T)	eos/xrootd (10gbps)	ZSTD	0.64	8m16s
2 x Xeon (32C/64T)	eos/xrootd (10gbps)	LZ4	0.34	13m42s
2 x Xeon (32C/64T)	CephFS HDD (10gbps)	LZMA	0.83	9m27s
2 x Xeon (32C/64T)	CephFS HDD (10gbps)	ZSTD	0.63	8m29s
2 x Xeon (32C/64T)	CephFS HDD (10gbps)	LZ4	0.39	11m50s
2 x Xeon (32C/64T)	CephFS SSD (10gbps)	LZMA	0.88	9m07s
2 x Xeon (32C/64T)	CephFS SSD (10gbps)	ZSTD	0.64	8m16s
2 x Xeon (32C/64T)	CephFS SSD (10gbps)	LZ4	0.40	11m37s
2 x EPYC (128C/256T)	Page Cache*	LZMA	0.77	2m58s
2 x EPYC (128C/256T)	Page Cache*	ZSTD	0.74	2m26s
2 x EPYC (128C/256T)	Page Cache*	LZ4	0.75	2m22s

Larger data-size and faster decompression emphasize IO limits