High Performance Analysis, Today and Tomorrow

Josh Bendavid

CERN, Esplanade des Particules 1, P.O. Box, 1211 Geneva 23, Switzerland

E-mail: Josh.Bendavid@cern.ch

Abstract. The unprecedented volume of data and Monte Carlo simulations at the HL-LHC will pose increasing challenges for data analysis both in terms of computing resource requirements as well as "time to insight". Discussed are the evolution and current state of analysis data formats, software, infrastructure and workflows at the LHC, and the directions being taken towards fast, efficient, and effective physics analysis at the HL-LHC.

1. Introduction

The Large Hadron Collider at CERN has already produced a large amount of proton collision data, with on the order of 10^{16} proton-proton collisions in each of the general purpose detector experiments in LHC run 2. The High Luminosity LHC program will represent a huge increase in the amount of data collected, representing a twenty-fold increase with respect to today [1]. While the exact impact on data volume depends on the interplay between integrated luminosity, physics priorities, and trigger strategy, it is clear that searches and measurement across the full range of final states and phase space regions will have significantly larger sets of data and Monte Carlo simulation to analyze.

For some Standard Model processes with large cross sections, measurements using LHC run 2 data already face significant technical challenges. As an example, inclusive $W \to \ell\nu$ production accounts for more than 3×10^9 events per lepton flavour per experiment for the full run 2 dataset, implying billions of data and Monte Carlo events. For measurements which cover an inclusive phase space, there is relatively little scope for skimming, and the technical challenge of analyzing these datasets is a prelude to those which will be faced by a wider range of measurements and searches in the HL-LHC era.

2. Analysis workflows and requirements

At a very high level, two critical properties are needed from the corresponding software, hardware and workflows in order to do effective physics analysis:

(i) **Fast turnaround:** Fast iteration time is essential for the debugging of experimental, theoretical or technical issues, and for developing and improving the analysis. What is relevant in this context is iteration time "as the physicist waits" (as opposed to in terms of computing performance metrics.)

(ii) **Flexibility:** Flexible capabilities are needed for defining the analysis logic, selection, binning, categorization, systematic uncertainties and statistical interpretation. The ideal is

Content from this work may be used under the terms of the Creative Commons Attribution 3.0 licence. Any further distribution of this work must maintain attribution to the author(s) and the title of the work, journal citation and DOI.
Published under licence by IOP Publishing Ltd

to avoid limitations related to technical feasibility when it comes to optimizing the analysis, or improving the description of the underlying physics or related uncertainties.

A representative workflow for an analysis of LHC Run 2 data at the CMS experiment might contain the following steps:

(i) Reconstruction of data and generation, detector simulation and reconstruction of Monte Carlo simulation events on the grid, centrally managed by the CMS offline and computing project, producing files, in ROOT [7] format, containing CMS reconstruction software objects in a "MINIAOD" format which is approximately 30kB/event [3].
(ii) Production on the grid, centrally or privately managed, of files in ROOT format, containing a list of basic types and arrays for each collision event, in the "NANOAOD" format or a customized variation thereof, which is approximately 1kB/event [4].
(iii) Processing NANOAOD or similar in order to to produce a "final" set of histograms for statistical interpretation and plotting, or a very condensed dataset for unbinned maximum likelihood fits.
(iv) Statistical interpretation and visualization, including likelihood fits, statistical limits/confidence intervals, plotting, etc.

In addition to the above, there may also be various auxiliary workflows or steps which might be needed for object/detector calibrations/corrections/etc. Much of the subsequent discussion will focus on step (iii) above, where the compact NANOAOD or similar analysis data-format is reduced to histograms or very small datasets for statistical interpretation and visualization.

The appropriate and feasible contents of the analysis data-format, given the stringent size constraints plays a key role in the design of the subsequent reduction step. In particular this determines in large part which classes of operations can be performed "on the fly" at this stage, as opposed to being better suited to pre-computation in one of the previous steps on the grid. Typical contents included variable length arrays for 4-vectors and summarized properties of high level objects, such as electrons, muons, photons, taus, jets, as well as event level properties or summary variables. For the high-level objects, aside from 4-vectors, summary properties such as isolation sums for leptons and photons, or substructure variables for jets might be included, as well as the output of one or more multivariate discriminator for suppression of misidentified objects, tagging of jets for boosted hadronic decays of heavy objects, etc. Among information which is **not** feasible to include in a NANOAOD or similar data-format of this size are things like detailed information on isolation sum constituents, detailed information on jet constituents (calorimeter clusters, tracks, particle flow candidates) or similar detailed global information about the event.

As a consequence of the above, the types of operations which can be performed "on the fly" when analyzing this data-format include object selection (lepton identification, isolation, kinematics cuts, etc), composite object combinatorics (forming $Z \to \mu\mu$, $H \to \gamma\gamma$, $t\bar{t} \to \ell jjb\bar{b}\not{E}_T$ candidates and the like), energy scale and resolution corrections for high level objects like leptons, photons, jets, plus corresponding systematic variations. Machine learning inference on high level object and event quantities, for example classifiers which are used for signal vs background discrimination based on lepton and jet kinematics, or high level quantities can also be computed at this stage. Operations which typically cannot be performed at this stage of the analysis and must be pre-computed at earlier steps include re-computation of isolation sums, re-clustering of jets, re-calculation of jet substructure variables, or machine learning inference on low-level detector information (individual hits, clusters, tracks, etc).

A heuristic goal for analysis turn-around time is $\mathcal{O}(1 \text{ hour})$ between making high-level changes to the analysis selection or other details, and being able to produce a new set of results. For $\mathcal{O}(10^9)$ events this implies event throughput rates in the MHz for the NANOAOD$\rightarrow$ histograms or similar reduction step. Achieving this while also realizing the corresponding flexibility goals requires modern, easy to learn and use software frameworks for the analysis step, where a consensus is building around python for the user-facing interface. Aside from this, “smart” parallelism is needed to achieve quick turnaround time “as the physicist waits”. In this context, latency is just as important as throughput, and it is critical to avoid IO bottlenecks, serial processing bottlenecks, and long tails in processing or error recovery. Typical batch-queue based solutions involve submitting a large number ($\mathcal{O}(1000)$) single core jobs, reading inputs from mass storage, and writing 10’s or 100’s of Mbytes of histograms to a shared filesystem. This potentially suffers from long tails in job scheduling, processing, and failure recovery, as well as additional overhead, bottlenecks and delays from merging the resulting outputs, which can be be 10’s or 100’s of GBytes in aggregate, and be stored on mass storage systems poorly suited for operations on large numbers of files, or shared filesystems with poor throughput characteristics and/or limited capacity.

One alternative parallelization model is simply to use multi-thread or multi-process parallelism on a single node, with shared memory or inter-process communication used to merge results. Recent substantial increases in the number of CPU cores available per-socket/node, as well as radical improvements in SSD performance from the transition to NVMe greatly extend the limits of single node scaling. For this approach to be effective, care needs to be taken to avoid lock contention and serial processing bottlenecks (Amdahl’s law). Where python is used in a multi-threading context, particular care must be taken with respect to the Global Interpreter Lock. Parallelization on a single node is ultimately limited by the maximum number of CPU cores per socket (with commodity servers having at most two CPU sockets). In order to scale up to multiple nodes, while avoiding the drawbacks of the batch-queue solution, newer task-based scheduling solutions can be used, such as Spark [5], or Dask [6]. These can maintain a more ”interactive-like“ user-facing behaviour via the scheduling of short tasks on persistent or longer-provisioned resources, as well as incremental and/or parallelized merging of results which can be transmitted between nodes directly over the network. These solutions have much more flexible scaling and provisioning, and more efficient utilization of shared resources compared to a single node, but additional challenges for maintaining robustness and avoiding IO bottlenecks.

3. Recent advances in analysis software for HEP

Until relatively recently, the vast majority of physics analysis at the LHC was carried out by means of ”legacy“ functionality in ROOT, including TTree::Draw(), TSelector, or hand-coded C++ event loops, with parallelization usually realized via batch queue systems, such as Condor [8]. There have been major recent developments in ROOT, as well as new python-based tools which enable additional paradigms and approaches for high performance analysis.

3.1. ROOT

Modern versions of ROOT include major developments such as Cling [9], which provides LLVM/Clang based just-in-time compilation of C++, fundamentally improving the robustness and feature-set of ”interpreted“ code. The PyROOT [10] interface leverages this to provide a much friendlier interface to ROOT via python on the one hand, and opens many possibilities for interoperability between python tools, ROOT, and other C++ libraries via the automatic python bindings and comprehensive C++ language support. Very recent developments in ROOT 6.26 enable optimization of just-in-time compiled code, and largely close the remaining

Journal of Physics: Conference Series **2438** (2023) 012002 doi:10.1088/1742-6596/2438/1/012002

performance gap between just-in-time and pre-compiled code. Another major advancement is RDataFrame [11], which allows the use of high performance computation graphs for data analysis. Combined with PyROOT, this provides a relatively user-friendly interface to implement complex analysis logic while maintaining high performance. Multi-threaded parallelization on a single node is $\sim$ fully transparent to the user, and fully addresses common issues related to parallelization discussed previously, as long as the runtime for the analysis in question is short enough given the number of CPU cores available in the available nodes. There are recent and ongoing developments to enable parallelization on Spark, Dask, or similar systems via distributed RDF functionality [12]. There are also significant ongoing developments to improve the functionality, convenience and performance of RDF in general [13]. Taken together, the combination of Cling, PyROOT and RDataFrame enable ROOT to provide a flexible and user-friendly interface to high performance C++.

3.2. Python Ecosystem

Beyond high energy physics, many industry-standard tools are based on the python ecosystem, including libraries such as Numpy and Scipy. In recent years there has been a major effort to leverage this ecosystem for physics analysis. A key requirement for this, and a critical limitation of numpy for HEP usage is the need to deal with "jagged" multi-dimensional arrays, where the number of elements in an array varies from one collision event to the next, as typical for HEP collision data. This has been addressed by the Awkard Array [14] library, which allows numpy-like syntax and operations to be applied to batches of events. Efficient reading of batches of collision events into Awkard arrays is facilitated by the Uproot library [15], which provides a python-based re-implementation of a subset of ROOT I/O, sufficient to analyze NANOAOD and similar data-formats by populating appropriate sets of awkward arrays, a paradigm often referred to as "columnar" analysis. Histogram functionality for python-ecosystem-based analyses are typically provided by the C++ Boost Histogram library [16], and the corresponding python bindings and extended functionality provided by the boost-histogram [17] and Scikit-HEP Hist [18] libraries. Additional high level analysis functionality and tools, building on the above are provided by the Coffea library [19], including multi-process parallelization on a single node, as well as efficient scaling to multiple nodes with Spark, Dask and similar systems.

3.3. Performance implications of event-loop vs batch processing

Physics analysis has traditionally been carried out using event event loops, where processing and aggregation of data is carried out one event at a time. The RDataFame interface in ROOT is a recent example of this. Columnar analysis may also be used, where operations and aggregation are applied to batches of events, up to hundred of thousands at a time using the python ecosystem. There are important technical and conceptual differences between these two paradigms. For tools where python is used beyond the initial set up, this type of batching is actually essential to maintain good performance, because of significant call overhead for each operation, given the need to pass through the python interpreter. With sufficiently large batches, the impact of this overhead can be reduced to negligible levels. Though typically much smaller, C++ is not immune from call overheads, through things like vtable lookups, which can negatively impact branch prediction, or dynamic memory allocation from std::vector or similar objects. Operating on batches of events also makes it easier to exploit data level parallelism (vectorization), though this might also be possible with event loops given sufficient levels of inlining of the computation graph. The per-event logic of operations in event-loop based tools is more in line with how HEP analysis has been done up to this point, and columnar analysis on batches of events often requires re-thinking or re-implementation of existing algorithms, but

may also in some cases allow more intuitive or performant solutions. Benchmarks provided by the Boost Histogram project, shown in Fig. 1 are an interesting example where compile-time specification of the number and types of axes and the corresponding optimizations this enables provides a large performance benefit when filling histograms per-event, but becomes largely irrelevant for large batches.

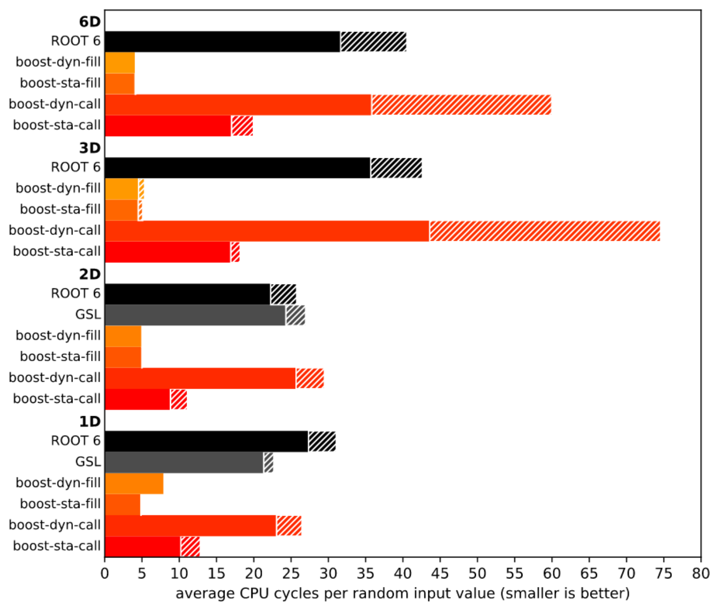

Figure 1. Number of cpu cycles per input value (lower is better) to fill Boost histograms in C++ of various dimensions, comparing compile-time ("sta") vs run-time ("dyn") specification for the number and types of axes, as well as per-event("call") vs batched ("fill") filling [16]. Compile-time specification of axes incurs less call overhead when filling the histogram, but the impact of this becomes small or negligible when using batched filling.

3.4. Multi-threading vs Multi-processing

When comparing multi-threading based parallelization, where multiple threads are spawned from a single process, vs multi-process based parallelization, where multiple processes are used, an important distinction which is sometimes overlooked are the benefits of shared memory, most easily exploited in the multi-threading case. When filling histograms as part of a physics analysis, the most common strategy is to keep one copy of the histogram per thread, which are then merged by adding them together. This has the advantage of avoiding any contention issues between threads, and also incurs relatively modest overhead from multi-process based parallelism, since histograms can be serialized for inter-process communication or for temporary storage on disk. The main drawback is potential memory limitations on the total number of histogram bins. A typical server or batch slot in the LHC computing infrastructure provides 2Gbytes of memory per thread. If the total size of the histograms for a given analysis exceeds this (with each histogram bin typically consisting of two double-precision floating point values, one for the sum of weights, and one for the sum of weights squared), then it can be difficult to efficiently use resources given only multi-process parallelism. In the multi-threading case this can be mitigated by using atomic operations for the aggregation. In particular this is possible using C++ Boost histograms together with RDataFrame, allowing only a single copy of the histogram to be used for a process containing many threads, and with minimal performance overhead for typical cases where the number of bins greatly exceeds the number of threads [20]. The boost-histogram python bindings provide some support for atomic histograms, but the python global interpreter lock can make it difficult to efficiently exploit this at the analysis level. Other possibilities to avoid memory limitations include sparse histograms, buffers and/or locks for filling of shared histograms, or distributing histograms across multiple processes or nodes.

3.5. Interoperability
While there is currently a reasonable level of interoperability between ROOT and python ecosystem tools, there is some room for improvement. Uproot is already able to read and write ROOT histograms from ROOT files, as well as providing limited conversion facilities between ROOT and python boost-histograms. Additional desirable functionality could include the use of ROOT and/or RDataFrame to read data into Awkard Arrays [21]. One particular issue is that the use of different C++ python bindings in ROOT vs the boost-histogram and hist libraries (PyROOT/cppyy vs pybind11) makes it difficult to use Boost histograms with RDataFrame while preserving all of the functionality of the python bindings in the boost-histogram and hist libraries. There are ongoing efforts to improve this [20].

4. Outlook
Both RDataFrame in ROOT as well as python ecosystem tools are being used for high performance analysis of LHC Run 2 today already today, with excellent performance and scaling both on large servers with multi-processing/multi-threading, and distributed across multiple nodes with Spark, Dask or similar [20,22–24]. Recent developments in analysis software therefore already going a long way towards addressing the challenges to be faced in the HL-LHC era, and are enabling increasingly innovative or complex analyses even of existing LHC data. Some challenges to address include the optimal leveraging of thread-level and multi-node parallelism in combination, as well as optimising storage and IO patterns and infrastructure to avoid bottlenecks as analyses scale in size and complexity, and many high-performance analyses begin making concurrent use of computing resources. Recent developments and progress are extremely promising, with improvements to functionality, ease-of-use and performance expected to continue over the coming years, in order to meet the challenges and requirements for flexible high performance analysis which are needed to fully exploit the physics potential of the HL-LHC.

References
[1] Aberle O *et al.* 2020 *High-Luminosity Large Hadron Collider (HL-LHC): Technical design report* CERN Yellow Reports: Monographs (Geneva: CERN) URL `https://cds.cern.ch/record/2749422`
[2] CMS Collaboration 2021 The Phase-2 Upgrade of the CMS Data Acquisition and High Level Trigger Tech. rep. CERN Geneva this is the final version of the document, approved by the LHCC URL `https://cds.cern.ch/record/2759072`
[3] Petrucciani G, Rizzi A and Vuosalo C (CMS) 2015 *J. Phys. Conf. Ser.* **664** 7 (*Preprint* `1702.04685`)
[4] Rizzi A, Petrucciani G and Peruzzi M (CMS) 2019 *EPJ Web Conf.* **214** 06021
[5] Zaharia M, Xin R S, Wendell P, Das T, Armbrust M, Dave A, Meng X, Rosen J, Venkataraman S, Franklin M J, Ghodsi A, Gonzalez J, Shenker S and Stoica I 2016 *Communications of the ACM* **59** 56–65 ISSN 0001-0782
[6] Rocklin M 2015 *Proceedings of the 14th python in science conference* 130-136 (Citeseer)
[7] Brun R *et al.* 2019 root-project/root URL `https://doi.org/10.5281/zenodo.3895860`
[8] Thain D, Tannenbaum T and Livny M 2005 *Concurrency - Practice and Experience* **17** 323–356
[9] Vasilev V, Canal P, Naumann A and Russo P 2012 *Journal of Physics: Conference Series* **396** 052071 URL `https://doi.org/10.1088/1742-6596/396/5/052071`
[10] Galli, Massimiliano, Tejedor, Enric and Wunsch, Stefan 2020 *EPJ Web Conf.* **245** 06004 URL `https://doi.org/10.1051/epjconf/202024506004`
[11] Piparo, Danilo, Canal, Philippe, Guiraud, Enrico, Pla, Xavier Valls, Ganis, Gerardo, Amadio, Guilherme, Naumann, Axel and Tejedor, Enric 2019 *EPJ Web Conf.* **214** 06029 URL `https://doi.org/10.1051/epjconf/201921406029`
[12] Padulano, Vincenzo Eduardo, Cervantes Villanueva, Javier, Guiraud, Enrico and Tejedor Saavedra, Enric 2020 *EPJ Web Conf.* **245** 03009 URL `https://doi.org/10.1051/epjconf/202024503009`
[13] Guiraud, E 2021 Rdataframe enhancements for hep analyses: systematics, metadata and collection operations `https://indico.cern.ch/event/855454/contributions/4596507/` 20th International Workshop on Advanced Computing and Analysis Techniques in Physics Research
[14] Pivarski J, Osborne I, Ifrim I, Schreiner H, Hollands A, Biswas A, Das P, Roy Choudhury S

and Smith N 2018 Awkward array if you use this software, please cite it as below. URL https://doi.org/10.5281/zenodo.6321292

[15] Pivarski J *et al.* 2022 scikit-hep/uproot4: 4.2.2 URL https://doi.org/10.5281/zenodo.6354509

[16] Dembinski H 2021 Boost.histogram: Version 1.78.0 URL https://www.boost.org/doc/libs/1_78_0/libs/histogram/doc/html/index.html

[17] Schreiner H *et al.* 2022 scikit-hep/boost-histogram: Version 1.3.1 URL https://doi.org/10.5281/zenodo.6107728

[18] Schreiner H *et al.* 2022 scikit-hep/hist: Version 2.6.1 URL https://doi.org/10.5281/zenodo.6345348

[19] Smith N and Gray L 2019 COFFEA - Columnar Object Framework For Effective Analysis URL https://doi.org/10.5281/zenodo.3598789

[20] Bendavid J 2022 Rdf interoperability with boost histograms and eigen for high performance analysis https://indico.cern.ch/event/1127096/contributions/4730609/ 120th ROOT Parallelism, Performance and Programming Model Meeting

[21] Pivarski J 2020 Rdataframe integration https://github.com/scikit-hep/awkward-1.0/issues/588 awkward Array

[22] Manca E and Guiraud E 2019 Using rdataframe, root's declarative analysis tool, in a cms physics study https://indico.cern.ch/event/849610/ cERN EP Software Seminar

[23] Padulano, V E, Kabadzhov, I, Guiraud, E and Tejedor, E 2021 Distributed rdataframe: leveraging dask and latest optimisations https://indico.cern.ch/event/855454/contributions/4596502/ 20th International Workshop on Advanced Computing and Analysis Techniques in Physics Research

[24] Smith N and Gray L 2021 Evaluating awkward arrays, uproot, and coffea as a query platform for high energy physics data https://indico.cern.ch/event/855454/contributions/4605033/ 20th International Workshop on Advanced Computing and Analysis Techniques in Physics Research