Novel functional and distributed approaches to data analysis available in ROOT

G. Amadio[1], J. Blomer[1], P. Canal[2], G. Ganis[1], E. Guiraud[1,3], P. Mato Vila[1] L. Moneta[1], D. Piparo[1], E. Tejedor[1], X. Valls Pla[1,4],

[1]CERN, [2]FNAL, [3]University of Oldenburg, [4]University Jaume I

E-mail: root-dev@cern.ch

Abstract. The bright future of particle physics at the Energy and Intensity frontiers poses exciting challenges to the scientific software community. The traditional strategies for processing and analysing data are evolving in order to (i) offer higher-level programming model approaches and (ii) exploit parallelism to cope with the ever increasing complexity and size of the datasets. This contribution describes how the ROOT framework, a cornerstone of software stacks dedicated to particle physics, is preparing to provide adequate solutions for the analysis of large amount of scientific data on parallel architectures.

The functional approach to parallel data analysis provided with the ROOT TDataFrame interface is then characterised. The design choices behind this new interface are described also comparing with other widely adopted tools such as Pandas and Apache Spark. The programming model is illustrated highlighting the reduction of boilerplate code, composability of the actions and data transformations as well as the capabilities of dealing with different data sources such as ROOT, JSON, CSV or databases. Details are given about how the functional approach allows transparent implicit parallelisation of the chain of operations specified by the user.

The progress done in the field of distributed analysis is examined. In particular, the power of the integration of ROOT with Apache Spark via the PyROOT interface is shown.

In addition, the building blocks for the expression of parallelism in ROOT are briefly characterised together with the structural changes applied in the building and testing infrastructure which were necessary to put them in production.

1. Future challenges for analysis at the Energy and Intensity frontiers

The full exploitation of the LHC is the highest priority in the European Strategy for Particle Physics adopted as part of the ESFRI Roadmap [1]. A major upgrade of the LHC will take place in about three years from now. The luminosity delivered by the machine will be about ten times higher than the present, nominal one.

Such an evolution poses a series of challenges to us for what concerns all steps of the HEP data processing chain: triggering, reconstruction, simulation, digitisation and analysis.

Assuming the performance of the software presently in use for HEP data processing and a reasonable evolution of hardware technologies, the amount of resources the LHC community will need to cope with the aforementioned computation requirements will have to be nearly about ten times bigger than what it is today [2]. This is clearly not realistic.

In addition, a sizable contribution to the need of computing resources is to be accounted for the High Intensity Frontier program: about 10% of the budget presently required by a LHC experiment today [3].

From the point of view of the software, a better exploitation of resources goes hand in hand with parallelisation. However, for this approach to succeed, the intrinsic complexity of software parallelisation needs to be hidden from the user.

This paper proposes a new technique to optimise the usage of resources of the last step of the HEP data processing: analysis. A declarative approach offered by the ROOT [4] framework is discussed: TDataFrame [5]. Its cornerstones are the exploitation of parallel architectures, including accelerators, as well as a straightforward interface for scientists: the effort expended on achieving high parallel processing rates is wasted unless accompanied by outstanding achievements in the simplification of the programming model.

2. A declarative approach to HEP data analysis: TDataFrame

Starting from release 6.10, ROOT provides a tool to interact with its columnar datasets in a declarative way: TDataFrame.

The approach draws inspiration from other widely adopted data analysis frameworks such as Pandas [6] or Spark [7]. In the context of TDataFrame, an analysis is made of two main kinds of operations: *transformations* and *actions*. Examples of *transformations* are the application of a filter to select entries, the creation of a new column, also based on the content of other existing columns, the caching of the dataset in memory or its writing on disk. Examples of *actions* are the creation of a histogram, counting entries and extracting the content of columns. Transformations and actions are chained to achieve the desired output results. The approach is similar to the one of the more traditional data processing frameworks such as Gaudi [8]: the event loop is not controlled by the user but the data processing units are orchestrated by the framework to manipulate the data (see listings 1).

```
// Traditional approach, explicit loop
TTreeReader data(tree);
TTreeReaderValue<A> x(data, "x");
TTreeReaderValue<A> y(data, "y");
TTreeReaderValue<A> z(data, "z");
while(data.Next()) if(IsGoodEvent(*x, *y, *z)) h.Fill(*x);
h.Draw();

// Hidden event loop, declarative style
TDataFrame d(tree);
auto h = d.Filter(IsGoodEvent, {"x","y","z"}).Histo1D("x");
h.Draw();
```

Listing 1. A TDataFrame usage example comparing to the previously existing handle to access columnar datasets: TTreeReader. The code illustrates how straightforward it is to access the data, to apply a filter and to create a histogram.

Historically, TDataFrame was born as a tool to implement *functional chains*. It was soon realised that the tool could have been much more powerful if it had been able to represent "functional graphs". For example, the code snippet 2 can be graphically represented as figure 1:

(i) an interface is created to the dataset: the TDataFrame,

(ii) a filter, expressed in the form of a lambda, is applied to it,

(iii) a new column, z is added for the entries passing the filter,

(iv) two histograms are created depending on this last node.

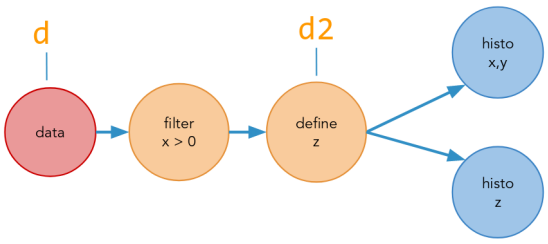

Figure 1. Complex control flows, which for example incarnates a series of cuts on collision events, can be concisely and expressively represented with TDataFrame.

```
// d2 is a new data-frame, a transformed version of the d
auto d2 = d.Filter([](const A& x){return x > 0;}, {"x"})
           .Define("z", "x*x + y*y");

// make multiple histograms out of it
auto hz = d2.Histo1D("z");
auto hxy = d2.Histo2D("x","y");
```

Listing 2. Intermediate nodes can be saved into temporary variables for the comfort of the user. Predicates associated to a transformation such as a filter can be expressed as a *callable* or a C++ string which is jitted by the ROOT interpreter and associated to the necessary columns on behalf of the user provided that the variables have the same name in the dataset.

It should be noted that all transformations and actions are executed in the same loop over all entries: it is therefore possible to create an arbitrary number of histograms or other quantities in a single event loop, in contrast with more traditional approaches such as the one offered by ROOT `TTree::Draw`.

One of the ideas behind TDataFrame is therefore to express an analysis as a "pipeline" of steps to be performed on the input data: the user chooses *what* has to be run while ROOT takes care of *how* the calculation is structured, for example for what concerns optimisations. For instance, the parallel runtime of ROOT is exploited so to run in parallel the analysis on behalf of the user transparently: the same code written for the sequential case is executed on all the cores available by the ROOT parallel runtime.

A serious investment was made in the programming model offered to the scientists by the TDataFrame interface. The APIs were designed respecting modern C++ best practices. In addition, the Python ROOT interface, PyROOT [9], was also considered during every stage of the design in order to provide seamless access to the functionality offered in the two languages by construction. In order to guarantee top performance, template metaprogramming techniques were aggressively adopted in the implementation. As a result, desirable behaviours such as absence of virtual function calls as well as type-safe access to the datasets are guaranteed.

3. Benchmarks

Besides usability, the principal added value of an approach like the one discussed in this contribution is performance and scaling on many cores architectures.

Figure 2 shows the behaviour of TDataFrame when dealing with a real-life study of the effect of parton distribution functions adopted by the Monte Carlo generator on the measured mass of the W boson. In this particular case, eight kinematic and quality cuts were applied and more than a thousand tridimensional histograms (70x10x10 bins) were filled. The input dataset was a flat n-tuple contained in a compressed 2.5 GB ROOT file. The times measured to calculate the scaling are *real times* needed by the program to execute, therefore including startup time, histogram merging as well as tear-down of the application: in other words, the time a user would have to wait to see the result of the study.

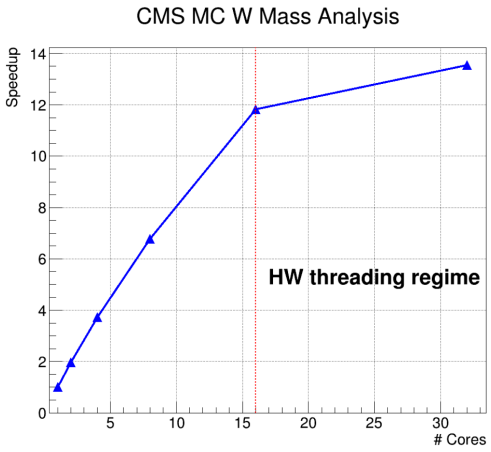

Figure 2. Scaling of a real-life application based on TDataFrame, running on an a machine equipped with Intel Xeon CPU E5-2650 v2 @ 2.60GHz equipped with 32 logical cores. Despite the presence of NUMA domains an encouraging scaling is achieved. Thanks to M. Dunser for providing the original code of this benchmark.

A comparison of different approaches to analysis of HEP data stored in ROOT columnar format has been performed. A sample study on a LHCb open dataset was examined, considering three different ways of accessing the data, compressed and uncompressed, namely: the usage of the bare TTree interface, the adoption of the TTreeReader tool, the exploitation of TDataFrame, in sequential and parallel mode. The results are shown in figure 3.

The TDataFrame approach is slightly slower than the TTreeReader one. The former, in its implementation, relies on the latter and adds some overhead (which is planned to be removed in the 6.12 ROOT release). The parallelised approach, possible only with TDataFrame, shows that with no additional effort required by the user, the full exploitation of the resources of the machine can be achieved.

Parallelisation can be achieved also with multiple processes, for instance running on a multicore computer or a batch farm, at the price of imposing a final, sequential, merging of the results. The kind of multithreading made accessible by TDataFrame avoids this last onerous step.

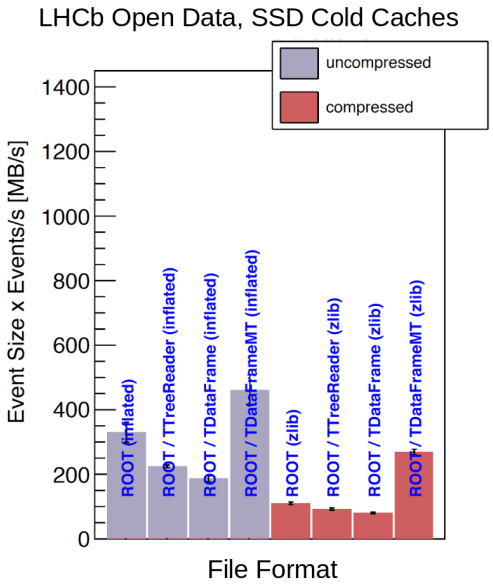

Figure 3. Amount of collision data processed for different file formats on a commodity laptop featuring an Intel i7 CPU with eight logical cores. The machine is equipped with an SSD. Considering the simplifications introduced by the programming model of TDataFrame and the possibility to parallelise an existing analysis on all cores available with no effort, the advantage over more traditional methods is clear. The overhead imposed by TDataFrame and TTreeReader over the bare TTree usage is understood and planned to be removed in a forthcoming ROOT releases.

4. The PyROOTSpark Project

The declarative approach described above is not specific to TDataFrame. The PyROOTSpark R&D project [10] was initiated in order to be able to exploit Spark [7] clusters via a slim Python interface. The underlying principle is shown in listing 3, i.e. to allow the design of a ROOT based data analysis designed around the map-reduce [11] pattern.

```
import ROOT
from DistROOT import DistTree
dTree = DistTree(filelist = ["myFile1", "myFile2"],
                 treename = "myTree", npartitions = 8)
myHistos = dTree.ProcessAndMerge(fillHistos, mergeHistos)
```

Listing 3. The map and reduce functions are executed by Spark on ROOT datasets. The Spark engine is used as a scheduler for the work. PyROOT guarantees the communication between C++ and Python, whereas PySpark the connection between Python and the Spark interface.

With HEP workflows running on Spark clusters in mind, an advanced and elegant monitoring system has also been developed to inspect the status of the jobs [12].

5. Conclusions and Outlook

A novel, declarative approach to ROOT based data analysis has been described: TDataFrame. The conjunction of TDataFrame's programming model and the possibility to run the same analysis in sequential and parallel mode without any effort required by the user makes the approach particularly powerful. With a few lines of code a scientist can apply filters to a columnar dataset, create histograms, create new columns, cache in memory or save the dataset on disk. A well behaved scaling of real-life usecases has been demonstrated on many cores servers as well as commodity laptops. The declarative approach is also being pursued in the context of PyROOTSpark, which offers a minimal interface to run ROOT based analyses orchestrating the simultaneous usage of ROOT, Python and Spark thanks to PyROOT and PySpark.

In the future, TDataFrame will evolve in order to offer an even better integration with PyROOT, giving for example the possibility to specify cuts as Python functions, and more handles to orchestrate parallelism such as an efficient *task local* storage. PyROOTSpark will be made available to the community as part of the CERN notebook service [13] in order to allow CERN scientists to transparently take advantage of Spark resources with limited knowledge of the technology and a minimal interface.

References

[1] Commission E 2017 European strategy forum on research infrastructures URL `https://ec.europa.eu/research/infrastructures/index_en.cfm`

[2] Bird I 2016 Wlcg and experiment computing for run ii URL `https://indico.cern.ch/event/563488`

[3] Fuess S, Gutsche O, Kirby M, Kutschke R, Lyon A, Norman A, Perdue G and Sexton-Kennedy E 2015 *Journal of Physics: Conference Series* **664** 032012 URL `http://stacks.iop.org/1742-6596/664/i=3/a=032012`

[4] Brun R and Rademakers F 1997 *Proceedings AIHENP'96 Workshop* vol A389 ed in Phys N I M pp 81–86 URL `http://root.cern`

[5] Guiraud E, Naumann A and Piparo D 2017 TDataFrame: functional chains for ROOT data analyses URL `https://doi.org/10.5281/zenodo.260230`

[6] McKinney W 2010 *Proceedings of the 9th Python in Science Conference* ed van der Walt S and Millman J pp 51 – 56

[7] 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 *Commun. ACM* **59** 56–65 ISSN 0001-0782 URL `http://doi.acm.org/10.1145/2934664`

[8] Barrand G, Belyaev I, Binko P, Cattaneo M, Chytracek R, Corti G, Frank M, Gracia G, Harvey J, Herwijnen E, Maley P, Mato P, Probst S and Ranjard F 2001 **140** 45–55

[9] Generowicz J, T L P Lavrijsen W, Marino M and Mato P 2005

[10] Tejedor E and Piparo D 2017 Pyrootspark Accessible at: https://github.com/etejedor/root- spark URL https://doi.org/10.5281/zenodo.1010129
[11] Dean J and Ghemawat S 2008 *Commun. ACM* **51** 107–113 ISSN 0001-0782 URL http://doi.acm.org/10.1145/1327452.1327492
[12] R K, Tejedor E and Piparo D 2017 Spark monitor 0.0.8 URL https://doi.org/10.5281/zenodo.1012585
[13] Piparo D, Tejedor E, Mato P, Mascetti L, Moscicki J and Lamanna M 2018 *Future Generation Computer Systems* **78** 1071 – 1078 ISSN 0167-739X