First implementation and results of the Analysis Grand Challenge with a fully Pythonic RDataFrame

Vincenzo Eduardo Padulano[1,*], *Enrico* Guiraud[1,2,**], *Andrii* Falko[3,***], *Elena* Gazzarrini[1,****], *Enrique* Garcia Garcia[1,†], and *Domenic* Gosein[1,4,‡]

[1]CERN, Esplanade des Particules 1, 1211 Geneva 23, Switzerland
[2]Princeton University, Princeton, NJ 08544, USA
[3]Taras Shevchenko National University of Kyiv, Kyiv, Ukraine
[4]Mannheim University of Applied Sciences, Mannheim, Germany

Abstract. The growing amount of data generated by the LHC requires a shift in how HEP analysis tasks are approached. Efforts to address this computational challenge have led to the rise of a middle-man software layer, a mixture of simple, effective APIs and fast execution engines underneath. Having common, open and reproducible analysis benchmarks proves beneficial in the development of these modern tools. One such benchmark is provided by the Analysis Grand Challenge (AGC), which represents a specification for realistic analysis pipelines. This contribution presents the first AGC implementation that leverages ROOT RDataFrame, a powerful, modern and scalable execution engine for the HENP use cases. The different steps of the benchmarks are written with a composable, flexible and fully Pythonic API. RDataFrame can then transparently run the computations on all the cores of a machine or on multiple nodes thanks to automatic dataset splitting and transparent workload distribution. The portability of this implementation is shown by running on various resources, from managed facilities to open cloud platforms for research, showing usage of interactive and distributed environments.

1 Introduction

Research in High Energy and Nuclear Physics (HENP) has always been entangled with large computational challenges, which are addressed by a well-structured data lifecycle. This defines the various processes involved in the management of collision data coming from the most important technical instrument for the physics programme of the field, the Large Hadron Collider (LHC) at CERN. The last steps of this lifecycle involve the operations performed on fixed datasets that LHC experiments store in their data centers. Data management systems, data transfer (including remote or local I/O), execution engines (single or multi node), computing resource managers, analysis interfaces, processing kernels and visualization routines can all fit in this context, which can be broadly described as the HEP data analysis landscape.

*e-mail: vincenzo.eduardo.padulano@cern.ch
**e-mail: enrico.guiraud@pm.me
***e-mail: andrii.falko@cern.ch
****e-mail: elena.gazzarrini@cern.ch
†e-mail: enrique.garcia.garcia@cern.ch
‡e-mail: domenic.gosein@cern.ch

The tools used by different LHC experiments, university groups or single physicists are varied and depend on the different use cases at hand. Some experiment collaborations such as ATLAS or CMS provide centralised software frameworks to serve common cases which are shared and used by all collaborators in their analyses [1, 2]. Similarly, research groups may have their own specialised analysis software that caters to the particular needs of their analysis jobs.

In this field, a common building block for higher-level software packages is ROOT [3], a de facto standard toolkit for storage, processing and visualization of HEP data. It is a C++-based project with full Python compatibility thanks to dynamic bindings, featuring a multitude of components. In particular, it offers a modern data analysis interface called RDataFrame [4], featuring a high-level programming model and lazy execution of the computations. This interface allows users to define the operations of their analysis and then execute them sequentially or in parallel in a transparent way. It supports native parallelisation on a single machine via implicit multithreading but also on multiple nodes by using a distributed execution layer that interacts with cluster resources [5].

Considering the plethora of workflows that can be found in physics analyses, ROOT and other software packages providing analysis interfaces such as coffea [6], nanoAOD-tools [7], or ADL [8] must strive to provide generic APIs that can cater to as many use cases as possible. In particular, this involves the implementation of specialised computation kernels that may appear often in HEP computations (e.g. computing the mass of a system of particles). At the same time, this means that the software tools tremendously benefit from having as many example workflows as possible, to be able to optimise for the various cases.

The full-scale analyses run in production by LHC collaborations are often not available as target benchmarks for software interfaces, due to access restriction to experiment data. Knowledge coming from the community regarding what usually happens in physics applications is thus distilled in smaller examples or tutorials that serve as a starting point for new users of the software. This can be seen for example in the official RDataFrame tutorials webpage [9]. In the last few years, with more advances in the analysis interfaces themselves and more community awareness, the available analysis examples have grown in number and complexity, especially helped by the increase in data availability thanks to the CERN Open Data Portal. Among these examples we can find some analyses using open data from the CMS experiment that provide more realistic workflows than available before [10].

More recent community efforts furthered the same goals of providing a common ground for understanding and optimising different analysis workflows. In particular, the Analysis Grand Challenge project (AGC) [11] represents a specification for more realistic HEP analyses. It aims at combining a series of analysis tutorials and the tools that enable the end-to-end processing pipeline. The project provides a reference implementation of the analysis benchmarks, which can be used as a starting point to showcase different frameworks. In fact, this paper demonstrates the first implementation of one reference physics analysis using RDataFrame. In the following sections, the steps required to implement the application will be described in detail. In particular, this is developed in the Python language, aligned with the reference implementation. The differences in user experience are discussed, with a focus on making the RDataFrame workflow as Python-idiomatic as possible. Finally, the execution of the application is demonstrated running in parallel on multiple computing nodes, thanks to the distributed capabilities of RDataFrame.

2 Translation of the reference analysis

The reference implementation of the AGC is available in a public repository [12]. For the purposes of this work, the task taken in consideration is a $t\bar{t}$ analysis performed on open data

from the CMS experiment, originally collected in 2015. The analysis task includes event selection, weighting, definition of new observables, application of systematic variations and aggregation into histograms for plotting. Being a project in continuous evolution, the AGC also defines a set of reference versions for the presented analysis [13]. At the beginning of the development of this work, the version in place was 0.1.0, so that will be taken as reference for the rest of the article.

The reference implementation features a set of different tools used for the separate parts of the analysis pipeline. The main interface for data analysis used is based on the coffea package. Listing 1 shows a summarised snippet of code representing the steps needed to run the analysis. The user needs to implement a subclass of the `processorABC` abstract class that contains all the details of the analysis logic (lines 3-5). An `executor` instance is responsible of executing the computations on the input dataset. Executors in coffea can run the application in parallel, both on the same machine and on multiple nodes. For example, the `FuturesExecutor` shown in line 3 runs on multiple local cores using Python futures. A `Runner` object takes the input dataset (in the form of a dataset name and a set of files), the executor and the processor instances to steer the whole execution.

```python
from coffea import processor

class TtbarAnalysis(processor.ProcessorABC):
    def __init__(self, *args): ...
    def process(self, events): ...

ex = processor.FuturesExecutor(workers=N_CORES)

run = processor.Runner(executor=ex, ...)

run(fileset, dataset_name, processor_instance=TtbarAnalysis())
```

Listing 1: Summary of the main ingredients involved in the data analysis step of the reference implementation.

Listing 2 shows the same key ingredients as needed by the implementation of the task using RDataFrame. Users can define the analysis logic starting by the input dataframe and then appending any amount of operations needed, following the high-level API with a lazy-execution model (line 5). The input dataset specification (same as the one used in the reference implementation) is used as input argument to the RDataFrame instance (line 7). Parallelisation on a single machine with multithreading is activated via a call to `EnableImplicitMT` (line 3).

The task at hand contains multiple steps of analysis logic and different computations. Clearly, the objective of new implementations is to replicate the same logic as the reference.The selection step of the analysis task is taken into consideration to draw a comparison between the two implementations for what concerns specifically the API. Listings 3 and 4 show respectively the reference implementation and the RDataFrame one. On the one hand, the reference API shows usage of a particular type of Pythonic arrays that can be adapted to the jagged arrays typical of HEP use cases, defined in the awkward array library [14]. The library offers both the implementation of the arrays and utility functions for their manipulation

```python
import ROOT

ROOT.EnableImplicitMT()

def run_analysis(dataframe): ...

df = ROOT.RDataFrame(dataset_name, fileset)

run_analysis(df)
```

Listing 2: Summary of the main ingredients involved in the RDataFrame implementation of the task.

(such as seen in `ak.count`). On the other hand, RDataFrame allows users to pass functions that manipulate the underlying data to the operations in the API, such as `Define` or `Filter`. When writing a C++ program making use of the RDataFrame interface, user functions can be passed to RDataFrame methods as any type of C++ callable object. When writing Python code, as in the current example, Python strings containing valid C++ code should be passed, as demonstrated in Listing 4. Such strings are just-in-time (JIT) compiled into performant C++ code via the ROOT C++ interpreter cling [15].

```python
selected_electrons = events.electron[events.electron.pt > 25]
selected_muons = events.muon[events.muon.pt > 25]
event_filters = (
  (ak.count(selected_electrons.pt, axis=1) +
   ak.count(selected_muons.pt, axis=1)) == 1)
```

Listing 3: Event selection steps in the reference application.

```python
selected_events = (
  df.Define('electron_pt_mask', 'electron_pt>25')
    .Define('muon_pt_mask', 'muon_pt>25')
    .Filter('Sum(electron_pt_mask) + Sum(muon_pt_mask) == 1')
)
```

Listing 4: Event selection steps in the RDataFrame application.

The listings shown reflect different approaches of the APIs. As many operations needed by physicists revolve around intra-event array manipulation, this is also where the differences in the APIs are more visible. The reference implementation makes use of a Python-only API, which adapts to the most used HENP computations by employing numpy-array-like ergonomics. This also results in executing user-defined workflows on the entire arrays, which hides the extra overhead that could be expected when running pure Python for-loops by delegating the execution to optimised C code. The RDataFrame implementation instead runs

user functions event-by-event, supporting a more well-known approach that can be more familiar to physicists. The current status of the API integrates Python bindings with the need to define C++ code in strings, which results in less Python-idiomatic code that is mitigated by the direct access to C++ performance. Nonetheless, this limitation is known and currently part of further research, as discussed in Section 3. Overall, the new implementation clearly presents differences in the API and different usage of typical HEP computation kernels. At the same time, it preserves the analysis logic of the reference application and produces exactly the same final histograms. The translation of the analysis to RDataFrame is available in a public repository [16].

3 Enhancing user experience with a Pythonic API

Having the full power of a C++ interpreter comes in handy in many occasions, as it allows not only to define functions on-the-fly but also to load externally-compiled fully-optimised C++ libraries directly in the Python script. Nonetheless, it is not always ideal to mix interfaces and syntaxes of two different languages in the same application. The ROOT project invests into bridging this language gap. In particular, for the RDataFrame API, this for example includes allowing users to pass Python callables directly instead of defining C++ code in strings. The general idea is having a way to extract a C++ compatible function from a Python callable. The approach that was adopted is that of compiling the Python code into a C function, the memory address of which can be then given to cling. The C++ interpreter is then able to run the C function as part of the RDataFrame operation execution. The translation of Python code into C is made possible by the Numba library [17]. As an example, Listing 5 shows how the event selection step could be rewritten by substituting the string expressions seen in Listing 4 with simple and idiomatic Python lambda functions, which in turn can also make use of popular Python libraries such as Numpy [18] (line 5). The whole analysis example is thus re-written to use this new interface, the full application is available in a public repository [19].

```
1 selected_events = (
2   df.Define('electron_pt_mask', lambda electron_pt: electron_pt > 25)
3     .Define('muon_pt_mask', lambda muon_pt: muon_pt > 25)
4     .Filter(lambda electron_pt_mask, muon_pt_mask:
5       numpy.sum(electron_pt_mask) + numpy.sum(muon_pt_mask) == 1)
6 )
```

Listing 5: Examples of Python lambdas passed to the RDataFrame API.

4 Distributed execution tests

The RDataFrame version of the AGC benchmark was continuously tested locally during development, to ensure full compatibility between the new results and their reference. One of the goals of the project is also to showcase how this type of interactive analysis workflows, in particular brought forward by the popularisation of the Jupyter notebook, fits well also in distributed infrastructures. To this end, the distributed RDataFrame layer made the transition from one machine to multiple nodes seamless, allowing testing the analysis different distributed environments using the popular Dask Python library as execution engine.

4.1 Test setup

A first round of tests was performed on an HPC cluster at CERN. Resource management is done via Slurm. Each computing node of the cluster has the following characteristics: 2x AMD EPYC 7302 16-Core CPU; 512GB DDR4 3200Mhz RAM; 10Gbit ethernet connection.

For the scaling tests, up to 8 computing nodes were requested from the cluster, to run the analysis in parallel up to a maximum of 256 concurrent cores. Every node is requested with exclusive access, so that while the jobs are running no other process can interfere with the measurements. One separate node is also requested to act as the Dask scheduler.

Another set of tests was carried out on the Virtual Research Environment (VRE) [20], which is a fully integrated data analysis environment built around JupyterHub [1]. The VRE relies on a microservices-oriented and cloud-agnostic design, allowing its architecture to be portable while easily integrating with other industry-standard technologies. The VRE exposes a data catalogue managed and orchestrated by the Rucio software package [21] and gives users the possibility of processing data by exploiting the power of workflow managers such as Reana [22] and Dask. The current VRE cluster runs on the CERN OpenStack cloud and uses Magnum Kubernetes as a compute backend for job distribution across 23 worker nodes made up mostly of 2x Intel(R) Xeon(R) Silver 4216 2.10GHz CPUs, 192GB DDR4 2933Mhz RAM and connected to a 10Gbit ethernet network, for an overall count of 184 vCPUs and 335.8 GB RAM. The analysis is run on the VRE by sending a declarative 'reana.yaml' file to the Kubernetes Reana cluster by specifying the IP address of the Dask scheduler; in this way, the analysis is distributed across all the cloud nodes.

The input dataset total size is 3.6 TB, of which roughly 5% is actually read by the analysis task. It is stored at CERN using EOS, the CERN storage system.

4.2 Results

This section reports the benchmark results using the CERN HPC cluster. The Python script is executed on the full dataset, choosing an increasing number of cores to parallelise the computations. For each core, one task is created that will run on a different subrange of entries from the total input dataset (different subranges do not overlap). Figure 1 shows the results of the benchmarks. The left image represents the end-to-end runtime, also called time-to-plot, which includes every step from the creation of the RDataFrame object to the retrieval of all the histograms. The total runtime decreases steeply with the number of cores used, crossing two orders of magnitude from a maximum of $5679 \pm 147\ s$ to a minimum of $50 \pm 3\ s$. The plot also presents error bars (vertically, in red, at any core count), but they are not big enough to represent any unexpected variance. The right image shows the same results expressed in terms of speedup with respect to the one core count. Being a log-log plot, the trend is clearly ideal at least until only one machine is used, i.e. the 32 core mark. The last three data points present some non-idealities in the speedup, closing at the 256 core mark with a speedup of 115. These may include network latency, poor scheduling which could be improved by increasing the number of tasks or implementing an asynchronous prefetching of the dataset entries, or also not having enough data since the total runtime at 256 cores is below one minute.

5 Conclusions

This work demonstrated a first implementation of the Analysis Grand Challenge $t\bar{t}$ tutorial using the ROOT RDataFrame interface. The new implementation demonstrates usage of a

[1] https://jhub-vre.cern.ch/

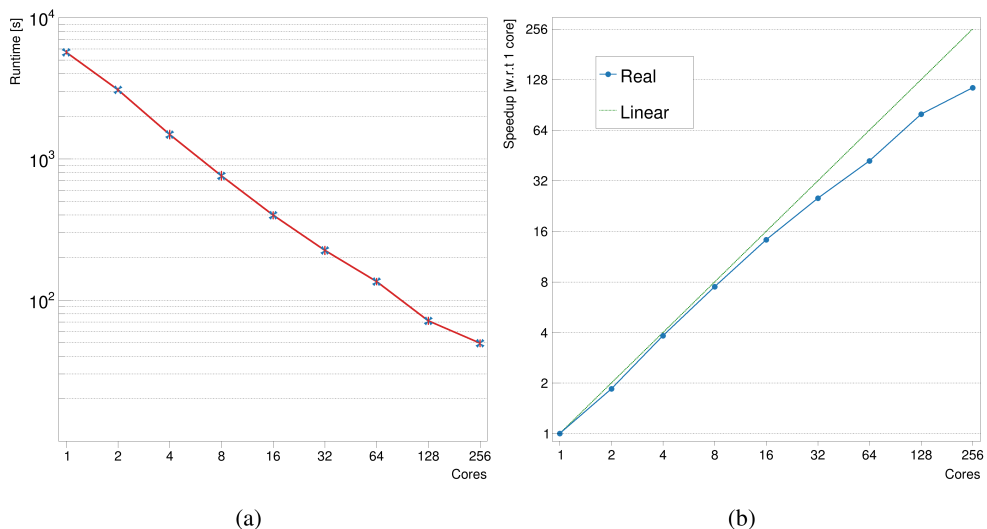

Figure 1: Results of running the AGC example with RDataFrame on the Slurm cluster at CERN. On the x axis of both figures, until the 32 cores data point the parallelisation happens on the same computing node. Consecutive core counts involve true distribution of computations over multiple nodes, up to a total of 8 nodes at the maximum of 256 cores. a: end-to-end runtime of the analysis. b: speedup with respect to the one core data point.

high-level programming model that can seamlessly allow parallelisation on one or multiple computing nodes. Differences in the code development experience were highlighted in Section 2, which do not alter the analysis logic but may present a different approach needed when thinking about function definitions and jagged array manipulations. A fully Python-idiomatic version of the RDataFrame AGC benchmark was shown in Section 3, which makes use of the Numba library. Although it already provides a way to greatly shorten the C++-Python language gap, it is still limited to mathematical operations on one-dimensional arrays and in general needs to go hand-in-hand with improvements in Numba to make full use of the power of C++ classes. Finally, Section 4 has shown the potential speedup improvements brought by running the interactive analysis tutorial on distributed execution environments.

References

[1] G. Benelli, B. Bozsogi, A. Pfeiffer, D. Piparo, V. Zemleris, *Measuring CMS software performance in the first years of LHC collisions*, in *2011 IEEE Nuclear Science Symposium Conference Record* (2011), pp. 108–112

[2] ATLAS Collaboration, *Athena* (2019), `https://doi.org/10.5281/zenodo.2641997`

[3] R. Brun, F. Rademakers, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment **389**, 81 (1997), New Computing Techniques in Physics Research V

[4] D. Piparo, P. Canal, E. Guiraud, X. Valls Pla, G. Ganis, G. Amadio, A. Naumann, E. Tejedor Saavedra, EPJ Web Conf. **214**, 06029 (2019)

[5] V.E. Padulano, I.D. Kabadzhov, E. Tejedor Saavedra, E. Guiraud, P. Alonso-Jordá, Journal of Grid Computing **21**, 9 (2023)

[6] L. Gray, N. Smith, B. Tovar, Y.M.E. Chen, A. Novak, J. Chakraborty, P. Fackeldey, N. Hartmann, G. Watts, D. Thain et al., *CoffeaTeam/coffea: v2023.6.0.rc2* (2023), https://doi.org/10.5281/zenodo.8147186

[7] CMS, *Tools for working with NanoAOD*, https://github.com/cms-nanoAOD/nanoAOD-tools (2022), accessed on 2023-09-08

[8] H.B. Prosper, S. Sekmen, G. Unel, *Analysis Description Language: A DSL for HEP Analysis*, https://arxiv.org/abs/2203.09886 (2022)

[9] ROOT Team, *RDataFrame tutorials*, https://root.cern.ch/doc/master/group__tutorial__dataframe.html (2023), accessed on 2023-09-08

[10] Various Authors, *CMS Open Data analysis examples and tools.*, https://github.com/cms-opendata-analyses (2023), accessed on 2023-09-08

[11] A. Held, O. Shadura, PoS **ICHEP2022**, 235 (2022)

[12] A. Held, O. Shadura, M. Feickert, J. Chakraborty, M. Proffitt, K. Choi, A. Novak, D. Koch, M. Adamec, S. Chopra et al., *iris-hep/analysis-grand-challenge: v0.1.0* (2022), https://doi.org/10.5281/zenodo.7274937

[13] Analysis Grand Challenge Team, *Versions description*, https://agc.readthedocs.io/en/latest/versionsdescription.html (2023), accessed on 2023-09-08

[14] J. Pivarski, I. Osborne, I. Ifrim, H. Schreiner, A. Hollands, A. Biswas, P. Das, S. Roy Choudhury, N. Smith, M. Goyal, *Awkward array* (2023), https://doi.org/10.5281/zenodo.8317185

[15] V. Vasilev, P. Canal, A. Naumann, P. Russo, Journal of Physics: Conference Series **396**, 052071 (2012)

[16] A. Falko, *Analysis Grand Challenge task implementation with RDataFrame*, https://github.com/andriiknu/RDF (2023), accessed on 2023-09-08

[17] S.K. Lam, A. Pitrou, S. Seibert, *Numba: A LLVM-Based Python JIT Compiler*, in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC* (Association for Computing Machinery, New York, NY, USA, 2015), LLVM '15, ISBN 9781450340052, https://doi.org/10.1145/2833157.2833162

[18] C.R. Harris, K.J. Millman, S.J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N.J. Smith et al., Nature **585**, 357 (2020)

[19] V.E. Padulano, *Demonstration of the Analysis Grand Challenge task with a Pythonic RDataFrame API*, https://github.com/vepadulano/analysis-grand-challenge/tree/rdf-agc-chep-2023 (2023), accessed on 2023-09-08

[20] E. Gazzarrini, E. Garcia, D. Gosein, A.V. Moya, A. Kounelis, X. Espinal, *The virtual research environment: towards a comprehensive analysis platform*, https://arxiv.org/abs/2305.10166 (2023), 2305.10166

[21] M. Barisits, T. Beermann, F. Berghaus, B. Bockelman, J. Bogado, D. Cameron, D. Christidis, D. Ciangottini, G. Dimitrov, M. Elsing et al., Computing and Software for Big Science **3**, 11 (2019)

[22] T. Šimko, L. Heinrich, H. Hirvonsalo, D. Kousidis, D. Rodríguez, *REANA: A system for reusable research data analyses*, in *EPJ web of conferences* (EDP Sciences, 2019), Vol. 214, p. 06034