A performance study of HEP data processing tools

Dung Hoang^{1}, Adriano Di Florio^{2,4}, Alexis Pompili^{2,3}, Umit Sozbilir^{2,3}, Vincenzo Mastrapasqua^{2,3}

^{1}Department of Physics, Rhodes College, Memphis, TN 38112, USA
^{2}Istituto Nazionale di Fisica Nucleare - Sezione di Bari, Via E. Orabona 4, 70125 Bari, Italy
^{3}Department of Physics, University of Bari, Via E. Orabona 4, 70125 Bari, Italy
^{4}Department of Physics, Polytechnic University of Bari, Via Amendola 126/b, 70126 Bari, Italy

E-mail: `hoadh-23@rhodes.edu`, `adriano.diflorio@ba.infn.it`, `alexis.pompili@ba.infn.it`

Abstract. In recent years, there has been a significant increase in the use of the Python programming language in High Energy Physics (HEP), particularly in the area of data analysis. Several efforts have been made to facilitate HEP data processing in Python, with Scikit-HEP and PyROOT being prominent examples. With multiple software packages offering similar functionalities, it becomes essential to consider their pros and cons before selecting one for a project. This study aimed to compare the performance of two software tools that facilitate the conversion of HEP data from the standard ROOT format to Numpy arrays or pandas dataframes. Our analysis identified the strengths and weaknesses of each tool when it comes to performance and parallelizability, providing valuable insights for users in selecting an appropriate package for their project.

1. Introduction

The Python programming language is characterized by an enormous and rapidly growing ecosystem of software tools that support the analysis and visualization of Big Data, datasets that are too large or complex to be dealt with by traditional data-processing methods. Examples include Numpy (numerical computing), pandas (data manipulation), matplotlib (visualization), and tensorflow (machine learning). With the growing volume and complexity of data produced by HEP experiments, a complete analysis workflow in Python can provide great benefits. However, the standard ROOT format common to HEP needs to be first converted to Python-friendly formats. Two popular approaches to handle such conversion are `Uproot` [1], a library for reading and writing ROOT files in pure Python and `NumPy`, and `RDataFrame` [2], the modern ROOT's high-level interface for efficient data analysis. In both cases the workflows were executed in a Python script. With reference to a representative HEP workflow taken as use case, we evaluated the performance of these two software packages.

2. Methods

2.1. Analysis task

In order to properly assess the performance of `Uproot` and `RDataFrame` , we conducted our study using a representative HEP analysis workflow. The ROOT files used in our study, which have a total size of about 128GB, come from CMS Experiment [3] Open Data [4]. The data

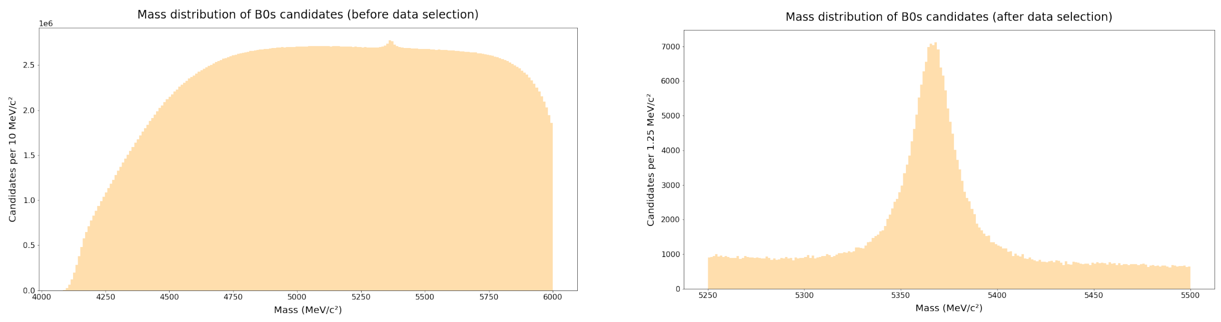

Figure 1. Mass distribution of B_s^0 candidates before and after data selection.

contains the reconstructed data corresponding to *pp* collisions from LHC Run1 from which the signal associated with a decay chain of a beauty meson ($B_s^0 \to J/\psi\phi$, $J/\psi \to \mu\mu$, $\phi \to KK$) is reconstructed. The goal of the workflow is to separate this signal from background noise, which made up most of the data.

Using `Uproot` and `RDataFrame`, we measured the total execution time of the following operations:

- accessing the data, stored as a ROOT `TTree` in the input ROOT files;
- applying specific filters (selection criteria) on the variables stored in the table in order to extract a known physical signal associated with the B_s^0 meson. Note that in both cases the filtering has been done within the `Uproot` or the `RDataFrame` framework, thus relying on their internal C++ backend implementation.
- converting the column (`TBranch`) containing the invariant mass of the particle to a `NumPy` array

No output file is written in the process. Figure 1 shows the meson mass distribution before and after the data selection procedure.

2.2. Parallel processing

The main goal of this study, is to test how the workflow mentioned above may be accelerated by splitting the workload on multiple processors, namely multiple CPUs on a single server, processing equal blocks of data concurrently. In the best case scenario, if all the process tasks may be parallelized, running in parallel would reduce the execution time by a factor of P, namely the number of parallel processes [5].

2.2.1. Parallel processing with `Uproot`

`Uproot` does not provide a built-in option for implicit parallel processing. To enable parallelism, therefore, we manually split the data into smaller chunks and then create subprocesses with Python's multiprocessing module to handle them. Since `Uproot` allows users to specify the number of rows to be processed in each table, data can be evenly distributed among all the subprocesses. Thus, the processing time function for `Uproot` may be computed as

$$T(N_r, P) = C + t_r \lceil \frac{N_r}{P} \rceil \tag{1}$$

where t_r is the time it takes to process one row, N_r is the total number of rows in the entire dataset, P is the number of processes, and C is the processing time of the not parallelizable parts of the workflow, which includes operations such as file opening and array concatenation.

Through testing, we found that C is constant with respect to the amount of data and the number of processes. The fraction $\frac{N_r}{P}$ is rounded up so that no row is lost when distributing data to each process.

2.2.2. Parallel processing with `RDataFrame`

`RDataFrame` , on the other hand, has a built-in function (`EnableImplicitMT()`) to enable implicit parallelism that allows to specify the number of threads on which the workload may be offloaded. Unlike `Uproot` , `RDataFrame` does not provide the option to specify the number of rows to be processed in each table, so a single file represents the smallest unit of data assignable to a single process. As a result, an even distribution is not always guaranteed given that, if the number of files is not divisible by the number of processes, some processes would have to handle one extra file. Then, an uneven distribution of file sizes would result in an unbalanced workload bottle-necking the entire workflow. To prevent this, therefore, we split the original dataset into 128 files of the same size. To ensure a fair comparison, we used these files as input for both `Uproot` and `RDataFrame` . Thus, the processing time function for `RDataFrame` may be formulated as

$$T(N_f, P) = C + t_f \lceil \frac{N_f}{P} \rceil \tag{2}$$

where t_f is the time it takes to process one file and N_r is the total number of files in the entire dataset.

2.3. Computing resources

Measurements were run on three different machines on the ReCaS-Bari computing center: wn-gpu-8-3-22 (256 CPUs, i.e. 2 AMD EPYC 7742 Processors), wn-1-8-9 (64 CPUs, i.e. 2 AMD EPYC 7281 Processors), and tesla04 (32 Intel(R) Xeon(R) Silver 4110 CPUs @ 2.10GHz). For the first two machines, data could only be stored remotely on the computing cluster, via the Lustre file-system [6], while on tesla04, it was possible to access the data on the local optical disks. Therefore, we used tesla04 to also study the potential difference in performance between local and remote data storage.

3. Result

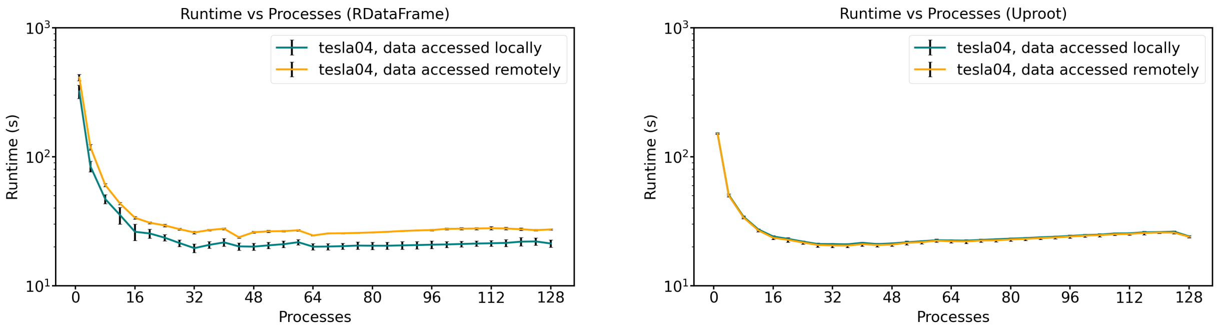

Figure 2. Processing time when accessing the data locally (teal) or remotely (orange) versus the number of processes on tesla04 machine. Left, using `RDataFrame` . Right, using `Uproot`.

As a first test, we checked the effect of accessing the data either locally or remotely, and we verified it is negligible. Figure 2 shows the processing time as a function of the number of processes for both `RDataFrame` and `Uproot`. In all other measurements, therefore, data were always accessed remotely.

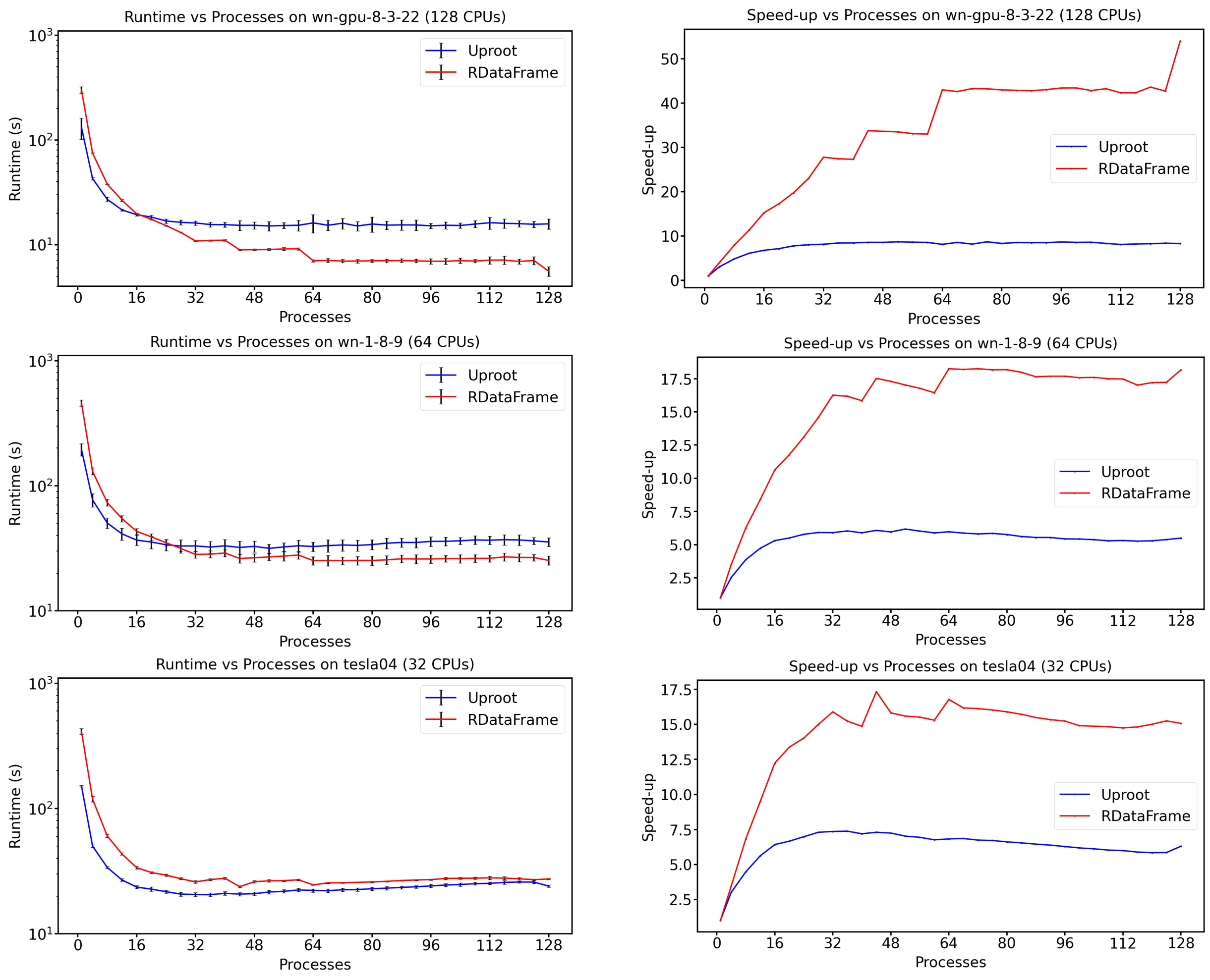

Figure 3. Processing time (left) and speed-up (right) for the whole dataset of `Uproot` (blue) and `RDataFrame` (red) as a function of the number of processes on different machines: wn-gpu-8-3-22 with 256 CPUs (top), wn-1-8-9 with 64 CPUs (middle), tesla04 with 32 CPUS (bottmom).

Secondly, we measured the processing time as a function of the number of processes, reported in Figure 3, and we found it consistent with the processing time functions in Equations (2) and (1). While the curves of `Uproot` are fairly smooth, we can see a discrete pattern in `RDataFrame` 's plots. This reflects the fact that `RDataFrame` 's smallest unit of data is a single file, resulting in a coarser granularity compared to `Uproot` , which allows the user to distribute the workload on a single entry basis. From Figure 3, it can also be observed that, in both configurations, we gained a significant performance boost when offloading the workload on multiple processors. On the machine wn-gpu-8-3-22, for instance, `Uproot` reached speed-ups up to 9 and `RDataFrame` , with 128 subprocesses, ran over 50 times faster than with a single process. As expected, we did not gain anything when we created more subprocesses than the number of CPUs on the machine. This is clearly visible on wn-1-8-9 and tesla04, which have 64 and 32 CPUs, respectively. With `Uproot` , however, the performance boost peaks at around 32 processes, no matter how many CPUs we use. This is because for `Uproot` , the unparalellizable time C is large, mostly due to the time needed to spawn each new process. As a result, it dominates the speed-up gained when the number of processes increases. In the case of `RDataFrame` , C is relatively small, so the performance improves until the maximum number of CPUs is reached.

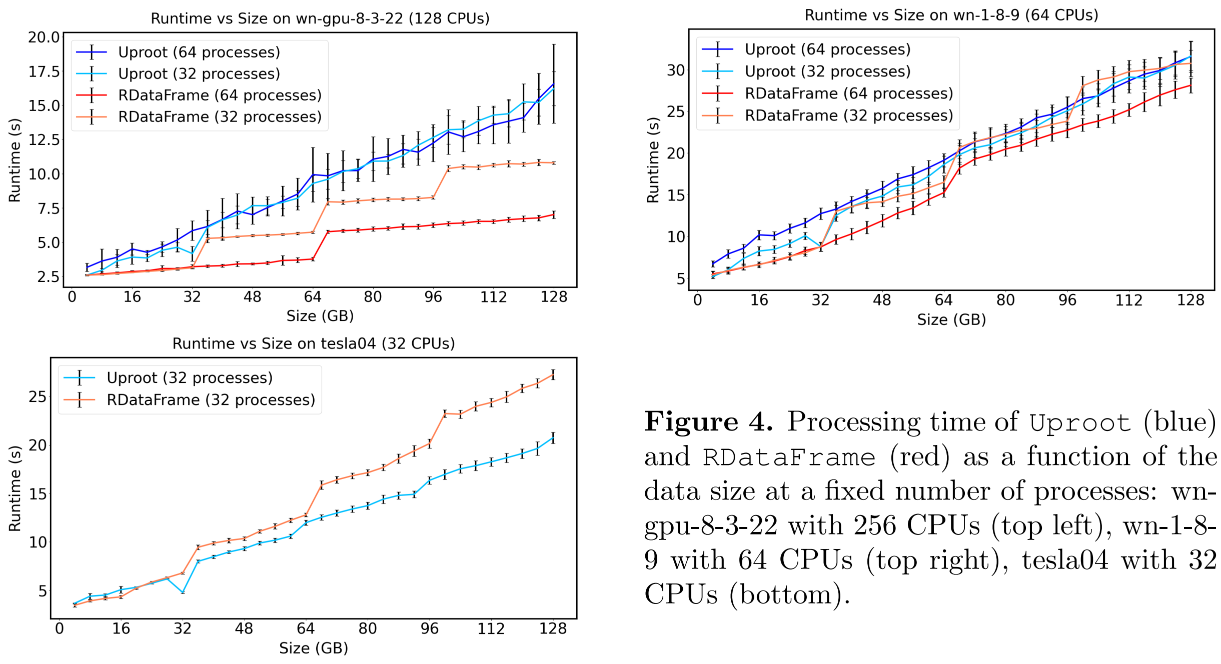

Figure 4. Processing time of `Uproot` (blue) and `RDataFrame` (red) as a function of the data size at a fixed number of processes: wn-gpu-8-3-22 with 256 CPUs (top left), wn-1-8-9 with 64 CPUs (top right), tesla04 with 32 CPUs (bottom).

Finally, Figure 4 shows that, as expected, the total processing time is linearly proportional to the size of data. In `RDataFrame` 's curves, a step-like pattern is observed, reflecting the file-based parallelism approach we mentioned above.

4. Conclusion

In this study, we compared the performance of `Uproot` and `RDataFrame` when offloading a simplified standard HEP workflow on multiple cores. While `Uproot` ran faster at fewer processes, `RDataFrame` performed better as the number of processes increase above 32, suggesting that on machines with an higher number of CPUs, it is more beneficial to use the latter. However, we must keep in mind that `RDataFrame` reached such a great performance because the original dataset had been previously split into smaller and evenly sized files. In a real-life data analysis scenarios, such condition may be not guaranteed, so `Uproot` may still be the better choice.

Acknowledgments

This project was supported by the DOE-INFN Research Fellowship, awarded by U.S. Department of Energy (DOE) and Italy's National Institute for Nuclear Physics (INFN) to promote the exchange of physics research between the United States and Italy.

References

[1] Jim Pivarski et al. *scikit-hep/uproot4: 4.0.4*. Version 4.0.4. Feb. 2021. DOI: `10.5281/zenodo.4543730`. URL: `https://doi.org/10.5281/zenodo.4543730`.

[2] Enrico Guiraud, Axel Naumann, and Danilo Piparo. *TDataFrame: functional chains for ROOT data analyses*. Version v1.0. Jan. 2017. DOI: `10.5281/zenodo.260230`. URL: `https://doi.org/10.5281/zenodo.260230`.

[3] The CMS Collaboration. "The CMS experiment at the CERN LHC". In: *Journal of Instrumentation* 3.08 (2008), S08004. DOI: `10.1088/1748-0221/3/08/S08004`. URL: `https://dx.doi.org/10.1088/1748-0221/3/08/S08004`.

[4] Harri Hirvonsalo. *Overview of research using CMS Open Data*. Aug. 2015. DOI: `10.5281/zenodo.33734`. URL: `https://doi.org/10.5281/zenodo.33734`.

[5] Gene M. Amdahl. “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities”. In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS ’67 (Spring). Atlantic City, New Jersey: Association for Computing Machinery, 1967, 483–485. ISBN: 9781450378956. DOI: `10.1145/1465482.1465560`. URL: `https://doi.org/10.1145/1465482.1465560`.

[6] Peter Braam. “The Lustre Storage Architecture”. In: *CoRR* abs/1903.01955 (2019). arXiv: `1903.01955`. URL: `http://arxiv.org/abs/1903.01955`.