Leveraging HPC resources with distributed RDataFrame

V. E. Padulano[1,2], I. D. Kabadzhov[1,3], E. T. Saavedra[1] and E. Guiraud[1]

[1] ROOT team, EP-SFT, CERN
[2] Departamento de Sistemas Informáticos y Computación, Universitat Politècnica de València
[3] Faculty of Engineering, Albert Ludwig University of Freiburg

E-mail: `vincenzo.eduardo.padulano@cern.ch`, `ivan.donchev.kabadzhov@cern.ch`, `enric.tejedor.saavedra@cern.ch`, `enrico.guiraud@cern.ch`

Abstract. The declarative approach to data analysis provides high-level abstractions for users to operate on their datasets in a much more ergonomic fashion compared to imperative interfaces. ROOT offers such a tool with RDataFrame, which has been tested in production environments and used in real-world analyses with optimal results. Its programming model acts by creating a computation graph with the operations issued by the user and executing it lazily only when the final results are queried. It has always been oriented towards parallelisation, with native support for multi-thread execution on a single machine. Recently, RDataFrame has been extended with a Python layer that is capable of steering and executing the RDataFrame computation graph over a set of distributed resources. In addition, such a layer requires minimal code changes for an RDataFrame application to run distributedly. The new tool effectively allows running a C++ event loop based on RDataFrame while leveraging common industry tools like Dask to schedule the usage of resources. This work presents results and insights gathered through the distributed RDataFrame tool running a physics analysis connecting multiple nodes with a Dask scheduler that requests resources from a Slurm cluster.

1. Introduction

The Large Hadron Collider (LHC) at CERN has generated an unprecedented amount of data over the course of its first two active periods (also called "runs"). The next active period, Run 3, is about to begin and it will be immediately followed by a major hardware upgrade (named HL-LHC[1]) that will in turn start generating data in 2027. With each run, the collider is fine-tuned and more events are generated. HL-LHC is foreseen to generate roughly thirty times more data than the LHC has produced so far. Given the available future budget estimations and expected technological evolution [2], collaborations in the HEP community would benefit from improvements on the software.

The information of the physics events happening in the accelerator is usually kept in large storage facilities and follows a specific skimming pipeline, so that the groups of researchers can access datasets in a well-defined format. This common format is implemented within the ROOT [3] software framework. It is a columnar layout that allows writing to disk any kind of structure, from scalar values to arbitrarily complex objects. ROOT has become the de facto standard for data I/O, processing and visualisation in this field.

The high amount of data collected by the LHC experiments has made distributed computing a staple in HEP data processing workflows for a long time, with the WLCG [4] being the prime example of efforts in that direction. With the challenges ahead, it will be crucial to make the most out of current and future infrastructure. In this regard, distributed computing will need to be further explored with new approaches that allow making best use of available computing resources while providing an ergonomic interface for final users [5].

In this context, ROOT RDataFrame [6] provides a high-level programming model to define analyses in terms of a computation graph that can be parallelised to run both on a multi-core machine and a set of distributed resources. This second option enables for example submitting RDataFrame applications through multiple tasks in a distributed execution engine [7].

In this work, an RDataFrame application is distributed over a cluster of HPC resources at CERN. The specifics of connecting the interactive execution engine to the batch system that manages cluster resources are described. A physics analysis benchmark is parallelised on more than one thousand cores, in order to evaluate the performance and scalability of the tool.

2. Methodology

The user workflow in a distributed RDataFrame application is not different from its local counterpart and usually follows three steps:

(i) The application begins by constructing a distributed RDataFrame from a dataset (e.g. the path to the dataset file(s)) and some object that represents the connection to a computing cluster. The type of the latter changes depending on the distributed backend, when using Dask this is called "client".

(ii) The dataset can be then transformed, e.g. by applying filters for unimportant events or creating new columns needed for further operations. When the relevant information is present in the data frame, it can be queried to retrieve important statistics and results of the analysis (the most common kind of result produced by HEP analyses is a histogram of one or more column values). All these calls to the RDataFrame API are lazy, both locally and in distributed mode.

(iii) The execution of the analysis is triggered the first time the user requests any of the results (e.g. when they plot a histogram for the first time). In distributed mode, this triggers calls to the underlying distributed scheduler that send tasks to the nodes of the cluster. When the distributed computations are finished, the final merged result is sent back to the user. Merging all partial results coming from the different tasks is done transparently by the tool, as opposed to being a responsibility of the user like in the past.

At the time of writing, this tool supports two distributed execution engines, namely Spark [8] and Dask [9], and its design allows to accommodate even more engines in the future. For the purposes of the tests run in this work, the Dask backend was used to connect to a job queuing system acting as resource manager of a cluster at CERN. In general, distributed Dask library allows to coordinate multiple bare-metal machines either manually, by starting the scheduler service on one node and the worker services on all other nodes, or automatically, by means of spawning the mentioned services through, for example, an SSH connection between the nodes. For the specific use case of HEP distributed computing, resources are most often managed by batch systems; for this situation, the `dask-jobqueue` library [10] offers a series of plugins to interface the scheduling capabilities of Dask with job queuing systems. In particular, this work has made use of the `SLURMCluster` class to spawn a Dask cluster on a set of nodes reserved through Slurm [11]. This is shown in Listing 1. For any given test run, the analysis is repeated three times (see line 25 of the listing).

```python
from dask_jobqueue import SLURMCluster
# This declares the connection to the Slurm cluster,
# No jobs are launched yet
cluster = SLURMCluster(memory=f'{4*NCORES}g',
                       processes=NCORES,
                       cores=NCORES,
                       queue='photon')

# Use the scale method to send as many jobs as needed
cluster.scale(NJOBS)

from dask.distributed import Client
# Create Dask client from cluster to establish connection between
# the cluster and the python application
client = Client(cluster)

# Wait for the workers before starting the analysis
client.wait_for_workers(NJOBS)

# Provide Dask client to RDF constructor
import ROOT
RDataFrame = ROOT.RDF.Experimental.Distributed.Dask.RDataFrame
if __name__ == '__main__':
    df = RDataFrame('treename', 'file.root',
                    daskclient = client, npartitions = NPARTITIONS)
    for i in range(NRUNS):
        run_analysis(df)
```

Listing 1. Example of distributed RDataFrame analysis using Dask to connect to a Slurm cluster. Specific information like number of cores or nodes used in the benchmark is omitted since it changes depending on the test run.

3. Experiments

The benchmark described in Section 2 is run through the Dask interface connecting to an HPC cluster at CERN. Each node has a 2x AMD EPYC 7302 16-Core Processor (total of 32 physical cores, no hyper-threading), 512GB DDR4 3200Mhz memory, and Infiniband 100Gbps network. The test is performed with a varying amount of nodes of the cluster, from one to thirty-two (from 32 to 1024 cores). The number of tasks sent to the distributed scheduler is fixed to 4096, that corresponds to 4 tasks per core in the benchmark run with the highest core count. Each task will be assigned its own chunk of the original dataset, so that two different tasks always process different parts of the dataset. The original dataset used in this analysis is made of one file that contains physics events recorded by the CMS experiment at CERN in 2012, currently published as open data [12]. For our purposes, in order to simulate a large-scale workload, the original dataset was replicated four thousand times to reach more than 246 billion events and almost 9 TB of data.

At runtime, the connection to the Dask cluster is initialised as shown in Listing 1. Before starting the analysis, the application waits for all the jobs to start so the requested cluster resources are ready for processing. The time between the actual trigger of the computations and the moment the client application receives back the final results from the distributed resources is often called “time to plot”. This is what is being measured and reported in what follows. While running the analysis, the workers read data from local storage. Code for the benchmarks is available in a public repository [13].

A first notable discovery is that for lower amount of cores, the benchmark fills up the memory of the nodes completely and cannot run until completion. This is due to a overhead present in each task, that needs to just-in-time (JIT) compile the RDataFrame C++ computation graph that needs to be executed on the assigned portion of the input dataset. Each task thus has a memory footprint and since there are always 4096 tasks, this becomes too great of a burden when too few nodes are available. This is the reason why Figure 1 presents the increase in processing throughput of the benchmark starting at 128 cores. By looking at the same image, a few more interesting insights can be obtained. For example, two lines can be seen in the plot: a red one, consistently on top, labeled “hot run”; and a blue one, consistently below the red one, labeled “cold run”. The blue line represents the first analysis run of each test, that always reports a higher runtime overall. It has been discovered that this happens because the newly created Dask workers incur an initialization cost in importing the ROOT module and initialising the JIT compiler engine. Consecutive analysis runs that happen within the same Dask cluster do not suffer from this, thus they run faster and they are represented by the red line in the plot. Irrespective of this difference, both lines show that the distributed RDataFrame tool is able to properly scale with an increasing amount of cores available, reaching a peak processing throughput of 52 GB/s.

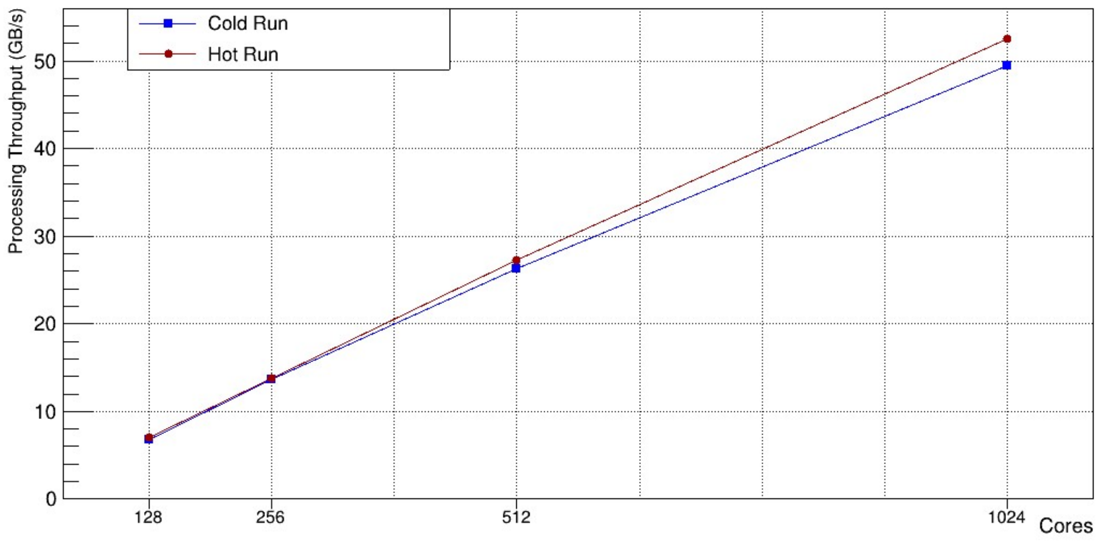

Figure 1. Processing throughput (GB/s) per core of a hot and a cold run of DistRDF, using the Dask backend, on an HPC cluster

4. Conclusions

The future challenges in HEP analysis require solutions that are scalable both in terms of programming model and resource usage. This work has shown how the distributed RDataFrame extension leverages an established modern analysis interface to steer computations to more than a thousand cores, obtaining a very good speedup with a peak of more than 50 GB/s. A few key insights have defined the next steps in further improving the performance of the tool, that will benefit in the future from a decreased JIT overhead in the distributed tasks.

References

[1] Apollinari G, Béjar Alonso I, Brüning O, Fessia P, Lamont M, Rossi L and Tavian L 2017 High-Luminosity Large Hadron Collider (HL-LHC): Technical Design Report V. 0.1 Tech. rep.

[2] Elsen E 2019 *Comput Softw Big Sci* **16**

[3] Brun R and Rademakers F 1997 *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* **389** 81–86 ISSN 0168-9002 New Computing Techniques in Physics Research V

[4] Bird I 2011 *Annual Review of Nuclear and Particle Science* **61** 99–118

[5] ROOT Team, Brann K A, Amadio G, An S, Bellenot B, Blomer J, Canal P, Couet O, Galli M, Guiraud E, Hageboeck S, Linev S, Vila P M, Moneta L, Naumann A, Tadel A M, Padulano V E, Rademakers F, Shadura O, Tadel M, Saavedra E T, Pla X V, Vassilev V and Wunsch S 2020 Software challenges for hl-lhc data analysis (*Preprint* 2004.07675)

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

[7] Padulano V E, Cervantes Villanueva J, Guiraud E and Tejedor Saavedra E 2020 *EPJ Web Conf.* **245** 03009

[8] 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 https://doi.org/10.1145/2934664

[9] Rocklin M 2015 *Proceedings of the 14th Python in Science Conference* ed Huff K and Bergstra J pp 130 – 136

[10] Dask Team `dask-jobqueue` library documentation Accessed on: 2022-02-12 URL http://jobqueue.dask.org/en/latest/

[11] Jette M, Dunlap C, Garlick J and Grondona M 2002 URL https://www.osti.gov/biblio/15002962

[12] Wunsch S 2019 Analysis of the di-muon spectrum using data from the CMS detector taken in 2012 URL http://doi.org/10.7483/OPENDATA.CMS.AAR1.4NZQ

[13] Ivan Donchev Kabadzhov 2022 Repository of benchmarks with distributed RDataFrame, Dask and Slurm URL https://github.com/ikabadzhov/DistRDF_benchmarks/tree/Slurm_Setup