J Grid Computing (2023) 21:9
https://doi.org/10.1007/s10723-023-09645-2

Leveraging State-of-the-Art Engines for Large-Scale Data Analysis in High Energy Physics

Vincenzo Eduardo Padulano · Ivan Donchev Kabadzhov · Enric Tejedor Saavedra · Enrico Guiraud · Pedro Alonso-Jordá

Received: 9 August 2022 / Accepted: 5 January 2023
© The Author(s) 2023

Abstract The Large Hadron Collider (LHC) at CERN has generated a vast amount of information from physics events, reaching peaks of TB of data per day which are then sent to large storage facilities. Traditionally, data processing workflows in the High Energy Physics (HEP) field have leveraged grid computing resources. In this context, users have been responsible for manually parallelising the analysis, sending tasks to computing nodes and aggregating the partial results. Analysis environments in this field have had a common building block in the ROOT software framework. This is the de facto standard tool for storing, processing and visualising HEP data. ROOT offers a modern analysis tool called RDataFrame, which can parallelise computations from a single machine to a distributed cluster while hiding most of the scheduling and result aggregation complexity from users. This is currently done by leveraging Apache Spark as the distributed execution engine, but other alternatives are being explored by HEP research groups. Notably, Dask has rapidly gained popularity thanks to its ability to interface with batch queuing systems, widespread in HEP grid computing facilities. Furthermore, future upgrades of the LHC are expected to bring a dramatic increase in data volumes. This paper presents a novel implementation of the Dask backend for the distributed RDataFrame tool in order to address the aforementioned future trends. The scalability of the tool with both the new backend and the already available Spark backend is demonstrated for the first time on more than two thousand cores, testing a real HEP analysis.

V. E. Padulano (✉) · I. D. Kabadzhov ·
E. Tejedor Saavedra · E. Guiraud
EP-SFT, CERN, Meyrin, 1211, Geneva, Switzerland
e-mail: vincenzo.eduardo.padulano@cern.ch

I. D. Kabadzhov
e-mail: ivan.donchev.kabadzhov@cern.ch

E. Tejedor Saavedra
e-mail: enric.tejedor.saavedra@cern.ch

E. Guiraud
e-mail: enrico.guiraud@cern.ch

V. E. Padulano · P. Alonso-Jordá
Department of Computation Systems and Computation,
Universitat Politècnica de València,
Valencia, 46022, Valencia, Spain

P. Alonso-Jordá
e-mail: palonso@upv.es

I. D. Kabadzhov
Department of Computer Science, Albert Ludwig
University of Freiburg, Freiburg, 79098, Freiburg, Germany

Keywords Root · High energy physics · Distributed computing · Dask · Spark

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, has recently begun and it will be immediately followed by a major hardware upgrade (named HL-LHC [1]) that will in turn start generating data in 2029. 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], software tools will play a crucial role to cover the performance gap and be able to process the foreseen data volumes.

Already with its current configuration, the accelerator produces information from physics events at rates of multiple GB/s. The information is initially filtered to remove non-interesting events that would only bring noise, then is sent from the experiments present around the accelerator itself to large storage facilities both at CERN and at collaborating institutes. These datasets follow a specific pipeline where they are further filtered and structured into a well-defined format that serves as the standard for different groups of physicists. This format uses a columnar layout that allows writing to disk any kind of structure, from scalar values to arbitrarily complex objects. Data are saved according to the different columns but also in groups of rows, every time there is more than a certain size threshold to be written (by default every 30 MB). Thus, it allows reading parts of the dataset independently, with a granularity that can vary from different columns in their entirety to a certain group of rows from a certain column.

This common format is implemented within the ROOT [3] software framework and it has been demonstrated that this custom data format brings tangible I/O performance gains when dealing with the specific characteristics of HEP data [4, 5]. ROOT has become the de facto standard for data I/O, processing and visualisation in the High Energy Physics (HEP) field. This framework is mainly implemented in C++ but also offers Python bindings. It is made of different components, among which RDataFrame [6] is the main interface for data analysis.

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. The traditional approach in this field is taking advantage of computing power granted by the different institutions that collaborate with CERN, coordinating it through a common grid. In particular, this is called the Worldwide LHC Computing Grid (WLCG) [7]. Historically, a researcher wanting to run a physics analysis on the grid would have had to perform various steps. First, the core application logic would have had to be thought and implemented. Usually, this would have involved thinking about how to process each row of the dataset (one row equals to one physics event), then iterating the logic on all the rows. This application would have expected as input one part of the total dataset. Then the user had to submit multiple jobs to the grid, each applying the code to a different piece of data. This would result in many partial results, that would usually be saved into the remote storage of the grid. The user would then have to code a separate program and submit it as another job on the grid, in order to take care of merging all the different partial results into the final result. In this context, one of the most common statistics to obtain is a histogram of one or more columns in the dataset. It is quite common to see analyses where thousands of partial histograms are generated in each job and then the merging step has to produce a list of histograms resulting from correctly merging the partial histograms that referenced the same physical dimensions in the various jobs. This approach was not user-friendly and it was often error-prone, even though many HEP collaborations developed their own distributed computing frameworks to mitigate potential issues.

Considering the challenges that HL-LHC will bring, it will be crucial to ensure that physicists only have to think about the algorithm involved to process an event and not about all the scheduling and implementation details. 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 [8].

While HEP has relied mostly on grid computing and batch queueing technologies such as HTCondor [9] or Slurm [10], distributed computing in other industries has also seen other approaches being used at scale. Notably, interactive execution engines from Hadoop such as Apache Spark [11] or from the Python ecosystem such as Dask [12] have seen more and more usage in many data science applications over a wide spectrum of use cases. One of the driving factors for such tools is exactly that they do not require the user to think about scheduling the distributed computations, instead offering APIs that can be called interactively

(meaning inside the same user application) to steer the computations to remote machines and directly get back the results.

This work presents a novel extension of the ROOT RDataFrame analysis interface, allowing to steer a physics analysis application to a distributed cluster of nodes with the Dask Python library. This enables large scale parallelisation of physics applications written in Python while still executing C++ code inside a distributed task on a computing node. Through Dask, RDataFrame can leverage widely used batch systems like HTCondor as resource managers of HEP hardware infrastructure. The new backend is compared against the already existing Spark backend using a concrete physics analysis as a benchmark, for the first time showing scalability of the tool with more than two thousand cores.

2 Related Work

Large-scale distributed computing has become more and more relevant in recent times with the huge increase in data volumes that characterises many use cases in different industries and research fields.

Examples of using Dask to distribute computations can be found in Earth and Climate sciences [13, 14]. Also in those fields datasets contain many years of data and can reach sizes of multiple TBs, with non-trivial multidimensional schemas similar to what can be found in HEP. In one of the cited works, data processing through Dask allows parallelising part of the data analysis workflow, the scalability results are shown up to 32 cores with a speedup value of around 9 in the best cases [13]. The same community made an effort to develop an ecosystem for distributed data analysis, recognising that previous approaches led to fragmentation and unproductivity [15]. In another work, molecular dynamics simulations were processed on a computing cluster using Dask as execution engine [16]. The split-apply-combine approach to distribute tasks shown in that work is similar to what will be discussed in this paper, but applied to a different field with different user workflows in mind. The Dask library itself provides a `dataframe` interface which can be directly used as an entrypoint for generic data analysis [17].

Apache Spark has been used as building block to scale analysis computations in many use cases. It can be considered as a more mature framework since it has been developed and explored for longer than Dask [18] and can be employed at least on the same set of workflows. Examples include multi-stage deep learning approaches [19], generic feature selection frameworks [20], streaming data analytics for IoT devices [21] and smart grid systems [22]. Usage extends as well to academia, for example in a geoscience effort to compare different storage systems with the objective of getting the best performance from a Spark-based analysis framework [23].

Although many tools exist to address data analysis needs of industries and academia, it should not be taken for granted that they can all work just as well in any other field. Particularly, it has been shown that for the HEP data analysis requirements a tailor-made tool like ROOT with its RDataFrame data analysis interface still has a major advantage over other industry frameworks [24]. The HEP field is not new to the investigation of large-scale distributed execution engines. The ROOT framework itself offered the Parallel ROOT Facility (PROOF), a tool to automatically parallelise HEP applications [25]. This provided a way to avoid all the manual submission work required by traditional batch systems, but it could only work with ROOT services that needed to be launched on the cluster resources (no other resource manager was supported). In 2017 two similar works presented a distributed data analysis system of the CMS experiment [26] using Spark, but encountered limitations in having to convert data from the standard HEP ROOT format to formats that Spark could understand natively, incurring in major performance bottlenecks [27, 28]. A later study overcame this issue, but did not achieve higher scaling when using available CERN storage facilities [29]. An example of good scalability was provided by researchers of the TOTEM experiment at CERN, with a first approach at distributing a ROOT application over Spark resources in a cloud [30]. The presence of Spark in the HEP community has become relevant enough that CERN has invested in specific infrastructure to support Spark analysis workflows [31].

No other large-scale execution engine has been explored as much as Spark in this field. Usage of Dask has begun only recently, also brought by the increased popularity in HEP of Python-based interfaces. In particular, it is being explored in the context of the so-called analysis facilities, where different tools are

unified in a coherent software stack that can fulfill all of physicists' analysis needs [32]. In this regard, a key feature of Dask is provided by its interfaces with batch computing systems, in particular HTCondor, widely used in HEP computing clusters.

No scalability test was presented with more than two thousand cores in the literature found for the HEP field. Also, while there are efforts to steer distributed computations to computing clusters through large-scale engines, none of them combines the possibility of using a user-friendly interface language like Python to actually distribute C++ computations. This is actually made possible by ROOT RDataFrame, that is already capable of automatically steering user code to a Spark cluster. This work extends that capability by adding a new Dask backend, which fulfills a very strong need to make use of available HEP hardware resources which are tightly integrated with batch resource managers. Furthermore, it lets C++ computations run along other Python-only functions called by internal Dask mechanisms.

3 Engines for Large-Scale Data Analysis

It has been established that HEP data analysis needs require distributed computing resources, which are mostly available through the grid. Accessing these resources requires a lot of effort from the users and is usually not an interactive workflow, rather a series of job submissions to the grid. HEP research groups have been exploring more interactive and user-friendly ways to steer the computations to the remote resources, which rely on execution engines that provide both a solid framework to access and utilise the distributed resources as well as more modern APIs that can be used interactively in the user application. These often share a common interface language in Python, that is used as a glue language to allow for an easier programming experience. Two such engines are briefly described in this section, namely Spark and Dask, which are widely used in various data science communities and are notably important for the future of distributed computing in HEP.

3.1 Apache Spark

Spark is an Apache project aimed at cluster computing and based on Hadoop MapReduce [33], extending it to more types of computations with a higher efficiency. The main feature of Spark is the ability to perform in-memory cluster computing to increase the processing speed of an application. Spark implements its own cluster management logic, separate from Hadoop, providing a faster and more general data processing platform.

Writing a Spark program involves creating an object called `SparkContext` that is able to connect and send computations to a remote cluster. Various functions in the Spark API enable parallelisation of user code through the `SparkContext`. On the cluster side, one Spark scheduler is responsible to send computations to one or more Spark workers and retrieve the results before sending them again to the user. A few cluster managers are available in Spark, namely:

- Spark standalone: manually starting the scheduler and worker services on the available nodes of the cluster.
- YARN [34]: the default Hadoop resource manager.
- Kubernetes [35]: a deployment system to scale containerised applications.

3.2 Dask

Dask is a Python library that allows to easily parallelise existing Python workflows. It is mainly targeted at supporting other common Python analysis tools like Numpy [36] or Pandas [37], but it is flexible enough to accomodate any type of computation. Thus, it offers many interfaces for data processing, including machine learning and real-time analysis. In the context of this work, Dask is employed as a distributed scheduler, offering a wide set of configurations thanks to which an application can be scaled to different cluster setups like:

1. Start all the remote nodes from a single machine through SSH.
2. Leverage existing cluster deployments with Kubernetes or YARN.
3. Connect to high performance computing resource managers that implement batch submission systems, like HTCondor, Slurm or PBS [38].

Two ingredients are necessary in order to distribute computations in a Dask application. The first is the

object representing the remote cluster itself, including how many resources will be assigned to it for the duration of the application. The second is an object representing the connection between the local machine and the remote cluster. This is simply called `Client` and can be used with any of the different implementations of resource managers available in Dask described above. The `Client` API allows users to asynchronously launch tasks to the remote cluster.

4 A Dask Backend for Distributed RDataFrame

As highlighted in Sections 1 and 2, distributed computing plays a key role in fulfilling data analysis needs of large HEP collaborations. In recent years, more attention has been put towards user-friendliness and interactivity aspects of distributed analysis frameworks, shifting away from the traditional batch queue systems. Nonetheless, these still provide the backbone of HEP computing facilities. In this regard, Dask has recently started to be explored by researchers in the field since it allows connecting arbitrary user code to an interactive scheduler that can also interface to the traditional batch systems. This section describes the development of a new backend for distributed RDataFrame based on Dask. Section 4.1 gives more detail about the RDataFrame tool in the ROOT framework, its programming model and the extension that was developed to distribute physics analysis which currently supports scheduling through Spark. Section 4.2 describes the implementation of the new Dask backend. Section 4.3 highlights the developments brought to the distributed RDataFrame tool that allow executing C++ computations from the Python processes spawned by Dask without overhead. Section 4.4 describes the automatic splitting of the user application into multiple tasks that are sent to the execution engine. Finally, Section 4.5 explores some of the nuances that may impact the user workflow when employing Spark or Dask as the execution engine.

4.1 ROOT RDataFrame

RDataFrame is the high level interface to data analysis offered by ROOT. It features a declarative API with lazy evaluation [39] of the functions called by the user. In fact, the tool effectively builds a computation graph that is only triggered when the results are actually requested in the application. Through the Python bindings offered by ROOT, RDataFrame allows physicists to write their code with the user friendliness and flexibility offered by the Python language, while the underlying tool runs computations in C++. It also implements specific HEP features like support for systematic variations, nested collections, producing histograms with associated statistics.

The declarative approach of RDataFrame also allows to better manage low-level optimisations and I/O scheduling with the ROOT format, the standard format in which all HEP data is stored. For example, it is able to parallelise the execution on different ranges of entries, aligned with respect to the parts of the dataset that can be read independently from disk. The operation of splitting the input dataset into various smaller ranges of entries and assigning one range to each task is also called “dataset chunking”, and it is commonly found in distributed computing use cases [40, 41].

Parallelisation has been a key feature of RDataFrame since its inception. The native C++ implementation allows to use all the cores in a single machine through implicit multithreading, which can be activated by a single function call at the beginning of the user application. More recently, the tool has been extended with a Python module that is able to take an already existing RDataFrame application and distribute its computations to a cluster of nodes.

The core idea of the extension is to wrap the computation graph defined in user code and execute it on each node, on a different portion of the original dataset. A HEP dataset can be split along its rows, each one usually representing a collision event. Since different physics events are statistically independent, parallelising the workflow on different data chunks is a valid approach. This also makes HEP data analysis an embarassingly parallel problem. The ROOT data format allows not only reading different columns independently, but it also defines a minimum amount of rows that can be read independently. In ROOT, the minimum amount of entries that can be retrieved independently from a file is called “entry cluster” or just “cluster”. Thus, distributed RDataFrame computing can be achieved by executing the same computation graph on different ranges of entries of the original

dataset. Aligning a task with respect to the entry clusters of the dataset ensures that no two tasks repeat the computations on the same entries while at the same time minimising the I/O transactions needed. The specific algorithm for task creation developed together with the new backend is better described in Section 4.4.

Once a list of tasks is generated on the client side, this should be sent to a scheduler for distributed execution. An important design choice in this regard was to make the extension modular, so that the tasks can be sent to potentially many different execution engines. The first prototype of this tool used the Apache Spark framework to distribute the physics analysis. This was the only available distributed RDataFrame backend in ROOT before this work, which thus builds upon that previous effort [42].

4.2 Executing the Computation Graph with Dask

The main object of this work is thus to enable RDataFrame workflows for users who want to exploit the flexibility offered by Dask scheduling. This goal is also completely in line with the design goal of the distributed RDataFrame tool, to support as many execution engines as the physicists may need.

In order to achieve it, the following must happen inside a distributed RDataFrame application:

1. The list of operations that make up the computation graph defined by the user is structured in a way that is serialisable.
2. Similarly, the input dataset is split in logical partitions, giving the user the option to specify the number of partitions.
3. A Dask client is created and connected to a cluster.
4. The computation graph, together with all the partitions defined in step 2, are registered with the Dask client.
5. Finally, the application triggers the Dask client, that will in turn communicate with the Dask scheduler in order to run the distributed computations in the cluster.

The first two items in the list are in principle handled by the already available implementation of the distributed RDataFrame tool: a mapper and a reducer function are already defined to take care of transforming the user-defined operations in actual RDataFrame function calls that will be executed on different chunks of the input dataset. Although Sections 4.4 and 4.3 will describe the improvements that went into the task creation and execution thanks to this work.

What is needed then is finding a way of submitting these functions defined locally to Dask. Dask offers many interfaces for data analysis, but the most interesting for the purposes of this work is called `dask.delayed` [43]. This is a Python decorator that effectively allows to run custom workflows of any type, by wrapping the user provided functions in objects that will delay the computations until actually requested by the application. In particular, calling a function that was previously decorated with `delayed` returns a `Delayed` object, which will wait to start the computations until the user calls its `compute` method. Both the mapper and reducer functions created in distributed RDataFrame are decorated with the `delayed` function, thus when they will be called on any given data chunk, they will not be executed right away, but accumulated by Dask. Effectively, a parallel computation graph is built at the Dask level, that will then execute the RDataFrame computation graph when triggered. From the point of view of the execution engine, this is still a MapReduce graph (like it is done in the previously available Spark backend), with the reduce phase done in a tree-like pattern. This is depicted in Fig. 1. In the figure, each square represents a node of the Dask computation graph. Looking at each vertical set of nodes, the first one includes all mapper tasks, that is all the different applications of the RDataFrame computation graph to a separate range of entries of the original dataset. All other tasks are reducing partial results of the mappers, two at a time until only the single, final result is left to be sent to the user.

Implementing this pattern on the backend side is shown in Listing 1. This shows the function that is responsible to take the computation graphs and the data chunks from the framework and connect it to Dask. In particular:

- Lines 4, 5: calls to the `delayed` interface. This makes the execution of the mapper and reducer functions deferrable. When they are called, they are registered in a computation graph internal to Dask.
- Lines 7-9: creation of the tasks by calling the delayed mapper function on each chunk of the

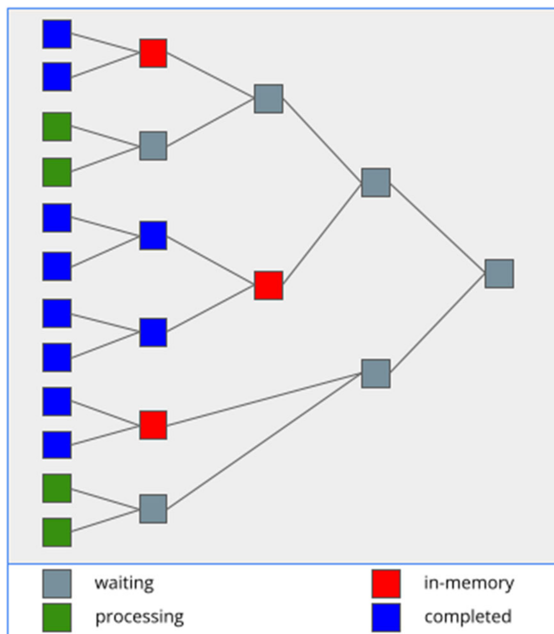


Fig. 1 Visualisation of the Dask computation graph generated by calling the `delayed` mapper and reducers in distributed RDataFrame. The colours of the nodes represent their status: nodes that are waiting for results from others are in grey; those that are currently processing a task are in green; those that have completed their task but need to wait for another task before sending their result to the next reducer keep the result in memory and are shown in red; finally, the nodes that have completed their task and do not need to wait for others are shown in blue

dataset. This is represented by the first vertical line of nodes described in Fig. 1.

- Lines 11-13: implementation of the reduce phase. The first two elements of the list of partial results are removed from the list and given to the reduce function; the result of the reduction is then appended to the same list; this is repeated until there is only one element in the list, meaning that all the partial results have been merged into the final result.
- Line 15: trigger the start of the computations by the Dask scheduler, by calling the `compute` method on the only remaining value of the described list.

4.3 Efficient Execution of C++ Code in Dask Python Processes with RDataFrame

The details given so far already show the core ingredient needed to enable the new backend. But another

```
1  def process_dask(
2    mapper, reducer, chunks):
3
4    dmapper = dask.delayed(mapper)
5    dreducer = dask.delayed(reducer)
6
7    futures = []
8    for chunk in chunks:
9      futures.append(dmapper(chunk))
10
11   while len(futures) > 1:
12     futures.append(dreducer(
13       futures.pop(0), futures.pop(0)))
14
15   return futures[0].compute()
```

Listing 1 Implementation of the MapReduce pattern in the Dask distributed RDataFrame backend

important part of this work is taking care that parallelisation of RDataFrame computations is properly handled by Dask on the computing nodes. At the node level, the parallelisation is done through Python multiprocessing. Each process is a separate Dask worker that will receive one or more tasks to run. Inside each Python process, Dask will spawn multiple Python threads. The main thread is responsible for running the user-provided function, which in the context of distributed RDataFrame is a mapper (or reducer) task. Other threads involve Dask internal mechanisms such as inter-task and inter-node communication.

A distributed RDataFrame application is written in Python and also the functions that are serialised and sent to the Dask workers are in Python. But when a worker is executing a task and calls into the RDataFrame API, it is going to run C++ code because the actual implementation of RDataFrame is in C++. This is made possible thanks to the dynamic Python bindings available in ROOT, named PyROOT, which can load a C++ library at runtime and call into it right away, thanks to the ROOT C++ interpreter. Thanks to PyROOT, there is no need to generate or send static Python-C++ bindings to the computing nodes, Dask will see only the Python layer of the tool and the implementation of distributed RDataFrame will transparently and automatically call functions implemented in C++.

Since Python is constrained by the Global Interpreter Lock (GIL), the different Python threads actually need to wait on each other before doing their job. Crucially, the main thread that is running the computations should never acquire the GIL for too long

to avoid blocking resources for the others. Instead, by default, the thread that runs the task code through PyROOT holds the GIL and does not release it when calling into C++ functions. Thus, the threads that are responsible for communication starve, leading to major slowdowns and timeouts when running the tasks on the Dask workers.

In order to overcome this issue, this work changes the implementation of the distributed RDataFrame Python package to ensure that the Python GIL is released when calling into C++ RDataFrame code to run a task. This is done by exploiting a feature of the Python bindings in ROOT, that is the possibility to unlock the GIL for the duration of a specific function. This is done at the beginning of the mapper function in each task, before executing the computation graph. From the point of view of a Python process running in one of the computing nodes, All Dask mechanisms can now work freely while the C++ computations are running. Communication between different Dask workers or with the Dask scheduler is ensured and the main Python thread does not block them anymore, thus bringing a tangible performance increase. Without this change, the Dask backend would just not be convenient for users and would not be competitive with the already existing Spark backend.

4.4 Client-Side Generic Task Creation for Distributed Backends

Automatic creation of tasks for distributed execution is a core feature of the distributed layer for RDataFrame. The main goal is ensuring that two distinct tasks will operate on two distinct parts of the dataset specified by the user. This requires exploiting as much as possible the I/O granularity offered by the data format when reading a ROOT file. As a simple example, let us take a ROOT file containing a dataset with two columns and two entry clusters. Two tasks can be possibly defined for this example, the first task operates on the first cluster, the second task on the second cluster. Both tasks need to also receive the information about the file(s) that contain the entries they were assigned. In general, the following rules can be derived about task creation:

- A task should be assigned with a range of entries to process from a particular set of files. This range should be aligned with respect to the cluster boundaries, to avoid reading a whole group of rows in memory and then just processing a few of them.
- For any given application, the maximum amount of tasks that should be created is equal to the total amount of clusters in the dataset. Creating more tasks would mean triggering unnecessary I/O requests to (potentially remote) ROOT files, adding a significant overhead to the analysis.

Creating the list of tasks to be passed to the execution engine has proven to be a great bottleneck in distributed RDataFrame. This issue arises from the fact that the information regarding clusters in a file can only be queried after the file has been opened. Before this work, all files of the dataset had to be opened on the client side to query the relevant metadata. Tests done at CERN have shown that a single open operation of a remote file stored within the CERN network from a client machine within the same network can take a few seconds to complete. A real HEP analysis can process O(1000) files, thus bringing a startup time cost of tens of minutes just to create the tasks. In this work, a new algorithm for task creation has been developed to completely remove the need to open remote files in the client, thus bringing the startup time close to zero. This is implemented via creating the task in two steps, one on the client side and the other on a distributed worker.

The generic idea of the algorithm is as follows. On the client side, the only information that is readily available because it is provided by the user is some specification of the input dataset, namely the list of files to process. This list is split into a series of tasks with length equal to the number of chunks specified by the user. On the local machine, no file is opened. Instead, splitting the input files into tasks is done by considering each file as an entity that can be divided according to percentages. For example, an application that processes two files in two tasks would have one task processing 100% of the first file and the other task processing 100% of the other file. The granularity can go even further: a task can be assigned with a range of percentage of a single file (e.g. [33, 66)). This is done because such a generic task will then be sent to some distributed worker which will need to convert this percentages into actual cluster boundaries. The difference

```
def create_generic_tasks(
  filenames, n_partitions):

  all_files = compute_files_in_tasks()
  percs_f, percs_l = (
    compute_percentages()
  )

  res = []
  for files, perc_f, perc_l in zip(
    all_files, percs_f, percs_l):

    res.append(
      (files, perc_f, perc_l)
    )

  return res
```

Listing 2 Approximate implementation of the creation of tasks on the client side

now is that files are only opened on the computing nodes that need to process them.

Listing 2 shows more details about the client side. In particular:

- Lines 1, 2: the function expects in input only the list of files that must be processed (i.e. the dataset) and a number of partitions in which they should be split.
- Line 4: gather a list of the files that should be processed in each task. This involves a few extra steps which are omitted from the listing:
  - First, the function creates a list of percentages according to how many partitions are required. For example, 5 files and 3 partition would give a list such as `[0, 1.66, 3.33, 5]`.
  - Then it retrieves the corresponding list of file boundaries as integers: `[0, 1, 3, 5]`.
  - Then it computes the difference element by element, to get the corresponding portion of the file for each percentage of the first list: `[0, 0.66, 0.33, 0]`.
  - From the list of file boundaries, the begin and end index (end exclusive) of the files in each task can be computed. Using this information, the `all_files` variable is a list where each element is another list containing the files that a task should process (a subset of the list of files in input to the function).
- Lines 5-7: gather two lists with the percentages of the first and last files, respectively, in each task where the processing should begin or end. Using the same example as above, `percs_f` is `[0, 0.66, 0.33]` and `percs_l` is `[0.66, 0.33, 1]`. Taking the first task for example, it will read files 0 and 1, file 0 will be read starting from percentage 0 (i.e. from its beginning) and file 1 will be read until percentage 0.66 (i.e. 66% of the entries in that file).
- Lines 9-15: construct the list of tasks by storing together the file indexes, the percentage of the first file and the percentage of the last file in a single object (i.e. the payload) for each task.

At the end of this function, each created task contains: a list of files (subset of the list of total files of the dataset), the percentage from which the task should start processing the first file, and the percentage until which the task should start processing the last file. Files in between the first and the last will be fully processed.

The payload obtained from the function described above will be sent to the remote workers. There, the approximate task needs to be concretised, i.e. the percentages need to be converted to actual entry numbers in the files. It must be noted again that this is deferred until the task is processed on the remote workers to avoid opening the files on the client side (a costly operation). Listing 3 shows an approximate implementation of this conversion from the payload task to an actual task. In particular:

- Lines 3-7: gather information from the payload. `first_file_idx` and `last_file_idx` represent the index of the first and last file that should be processed from the list of files received in the payload. This are shown here to help readability, as they will be used in later parts of the function. `perc_f` and `perc_l` represent the percentages of the first and last file obtained from the function in Listing 2.
- Lines 9-11: retrieve, for each file in the payload, a list of the entry clusters and the total number of entries in the file. The concept of entry cluster is specific to the data format provided by ROOT. It is the minimum amount of entries that can be read independently from a file, thus represents the smallest I/O transaction that can be performed. It is thus important that concrete tasks are aligned

with respect to cluster boundaries, so that I/O is minimised.

- Lines 13-16: convert the percentage of the first file from the payload to a real beginning entry of the first file. This is computed in the following steps:
  - The percentage is multiplied by the number of entries in the file, to get a candidate beginning entry.
  - This candidate is compared against the list of cluster boundaries for that file. The clusters are considered as bins and the corresponding cluster index is computed for the candidate entry.
  - The beginning entry of that cluster is taken as the beginning entry for the task.
  - The algorithm establishes whether a generic task should actually process a cluster or not based on whether the candidate entry coincides with the beginning entry of the cluster. For example, suppose a cluster spans entries from 10 (inclusive) to 20 (exclusive). If the candidate entry is equal to 10, the task will start processing from that cluster, otherwise not. This ensures that only one task takes that particular cluster of entries, so that there cannot be two tasks processing the same entries.

```
def convert_task(task):

  files = task.files
  first_file_idx = 0
  last_file_idx = len(files) - 1
  perc_f = task.perc_f
  perc_l = task.perc_l

  clusters, entries = (
    get_clusters_entries(files)
  )

  begin_entry_f = get_begin_entry(
    perc_f, entries[first_file_idx],
    clusters[first_file_idx]
  )

  end_entry_l = get_end_entry(
    perc_l, entries[last_file_idx],
    clusters[last_file_idx]
  )

  return (
    files, begin_entry_f, end_entry_l
  )
```

Listing 3 Approximate implementation of the conversion of a generic task into an actual task in a distributed worker

- Lines 18-21: convert the percentage of the last file from the payload to a real ending entry of the last file. This follows the same algorithm as in the previous item.
- Lines 23-25: return the concrete task. This includes the list of files received in the payload, the beginning entry where the task should start processing the first file, the ending entry where the task should stop processing the last file. These entries are aligned with respect to cluster boundaries. Files in between the first and the last are processed fully.

4.5 Impact of the Two Execution Engines on End User Workflows

As stated in Section 4.1, one of the main design choices for distributed RDataFrame is to make the extension modular so that it can fit with different execution backends. Thus, for the final user, using the Spark backend or the Dask backend should make no difference when running a distributed RDataFrame application. In fact, they write exactly the same analysis code, with the only difference being in the setup of the connection to the cluster.

The two engines also have many similarities. Notably, they both support expressing custom computations using the MapReduce paradigm (although Dask extends the support to any arbitrary execution pattern). Furthermore, they will both present the user with graphical dashboards showing real-time task execution and status of the nodes [44, 45]. Nonetheless, the two execution engines have different approaches which may result in nuanced differences regarding their usability.

One notable example is the cluster setup that is implicitly expected by the two different engines. Factoring out the standalone setups (i.e. manually launching either Spark or Dask services on various machines), Dask has the clear, crucial advantage of being able to interface to submission systems that are ubiquitous in HEP computing infrastructures. This means that setting up a Spark cluster will require either manual user intervention, or new facilities built ad-hoc for the purpose of interactive distributed analysis.

With enough user demand this may become worth its cost, as mentioned in Section 2. But being able to use pre-existing computing resources with no added engineering, logistic or maintenance cost is undoubtedly a valuable feature.

This advantage is not only visible on the infrastructure side, but also on the end user side. Not all users have direct access to the Spark cluster at CERN, so their only alternative, if they would like to use the Spark functionalities, is to launch the services on the traditional distributed computing resources. This involves manually launching jobs on the cluster, then launching the Spark services when the job is ready. Furthermore, although this work, and the distributed RDataFrame tool in general, are framed in the context of Python-based analysis workflows, the Spark library depends on Java in order to work. On most systems this is installable through the system package manager, but it is not guaranteed that users will have permission to do that. Dask on its side is a pure Python library, and any user can create a Python virtual environment in their home directory without additional permissions. There is thus a larger overhead with choosing Spark over Dask for a user that wants to start their analysis workflow for the first time, and it will be shown with more details in Section 5.3.2.

5 Experiments

In this section, the performance of the newly presented Dask backend is tested on a computing cluster, running a physics analysis example with different configurations. First, the analysis is run on a single node with varying number of cores. The processing throughput per core is compared against the processing throughput of running the same analysis with RDataFrame in sequential execution. Later, the Dask backend is compared against the previously present Spark backend running the same analysis on a larger dataset with both backends. By fixing the number of tasks executed by the two backends and the granularity of those tasks, we can compare how much they scale and if they introduce any noticeable overhead.

5.1 Application

The physics analysis used in this work processes data from events recorded by the CMS experiment at CERN in 2012. The analysis extracts the di-muon mass spectrum by computing the invariant mass of muon particles with opposite charge in the dataset. The result of the analysis is a histogram of the mass spectrum, showing peaks highlighting the presence of different particles in the physics events. This application is available through the CERN open data portal [46].

The original application was implemented using ROOT RDataFrame on a single machine. In this work, it was adapted to run with distributed RDataFrame. This practically involves only a few lines of setup code which are needed to connect to the cluster resources, depending on the desidered backend. The calls to the RDataFrame API are completely unmodified.

In the two different test configurations mentioned above, the original dataset is replicated in order to reach a higher size providing a more realistic computational workload. In the first configuration that runs the application on one node only, the dataset is replicated fifty times, whereas on the second configuration it is replicated four thousand times. The final dataset sizes for both configurations are reported in Table 1. Values in the table may present small rounding adjustments, the original dataset contains exactly 61 540 413 entries and its size is 2 244 449 133 bytes.

Replicating a ROOT dataset can be easily done by providing multiple times the path to a ROOT file to RDataFrame. Internally, the entries of the various files will be chained together and RDataFrame will be able to process them as a single coherent entity. This practice is valid for benchmarking purposes, since the physics events are statistically independent. In a first round of tests, the dataset is made available locally on each computing node, in order to factor out I/O performance from the results and read directly from filesystem cache. Subsequently, the dataset will be read remotely from its storage location at the CERN data center, providing a more concrete example of what physicists may experience.

5.2 Testbed

Resources from an HPC cluster at CERN are used for these tests. The computing nodes have the following characteristics:

1. 2x AMD EPYC 7302 16-Core Processor (total of 32 physical cores, no hyper-threading).

Table 1 Dataset sizes in the proposed experiments

Dataset	# Files	# Entries [K]	Size [MB]
Original	1	61 540	2 224
1st configuration	50	3 077 020	111 222
2nd configuration	4 000	246 161 652	8 897 796

First row: original dataset
Second row: dataset used when testing the Dask backend on a single node. Third row: dataset used when comparing the Dask backend against the Spark backend on many nodes of the cluster

2. 512GB DDR4 3200Mhz memory.
3. Infiniband 100 Gbps network card.
4. Samsung PCI-e NVME SSD.

This cluster relies on the Slurm framework to manage hardware resources. It is a batch queuing system, which offers a perfect example of where the new backend may shine also in terms of usability.

5.3 Methodology

When running a HEP analysis, one the most interesting measurements for the final user is the so-called "time to plot". That is the time between triggering the computations and receiving the final result that can be then plotted. In the following sections, the time to plot is measured in each test between the beginning and the end of the RDataFrame computations, irrespective of whether they are run locally or distributedly. The processing throughput of an application is then computed by dividing the size of the data that is actually read and processed in the benchmark by the corresponding time to plot. The considered analysis processes all columns and all entries of the input dataset.

5.3.1 Single Node Test with Dask

The original application is run on one of the nodes of the computing cluster, to get a first baseline measurement of the processing throughput on a single core. Then, it is converted to run with Dask, using a single computing node and increasing the number of cores used in that node. Details of the connection to Dask are specified in the next section. Furthermore, the number of chunks in which the dataset is split is also increased. Namely, 1, 2 and 4 chunks per core are tested.

5.3.2 Tests Comparing Dask and Spark Backends

Before actually running the tests with Spark, the needed resources must be requested to Slurm. When they are granted, the Spark scheduler and all the worker services on the computing nodes are started with a bash script that was provided in the Slurm request. Only after the Spark cluster is available, the Python application with distributed RDataFrame code is started. Thus, there is no direct way to start the Spark cluster by connecting to the batch system in the user application. The extra code needed by the Spark version of the benchmark is shown in Listing 4. In this case, the object that represents the connection to the cluster is called `SparkContext`. Its configuration options can be defined in the `SparkConf`, as shown in lines 10-21 in the listing. In this case, they just mirror the same resource configuration that was used to launch the services through the bash script mentioned above.

A different approach is shown in Listing 5 with the Dask setup. In this case, the object representing the connection to the cluster is a Dask `Client`. The cluster resources can be programmatically defined in the application itself, creating a cluster object from the types supported by Dask. In this work, the `SLURMCluster` class was chosen to connect to the cluster. It allows to automatically submit request for resources to Slurm, without the need to invoke a separate bash script, and shows the more user-friendly approach that Dask enables. This enables a direct interface with the resource manager, providing a different, more ergonomic approach for users with respect to what was described for Spark. The relevant steps needed for this setup are highlighted in Listing 5:

```
from pyspark import SparkConf
from pyspark import SparkContext
def main_spark(
  master, n_nodes, cores_per_node, dataset):

  total_cores = (
    n_nodes * cores_per_node
  )

  sconf = SparkConf().setAll([
    ("spark.master",
     f"spark://{master}:7077"),
    ("spark.executorEnv.PYTHONPATH",
     os.environ["PYTHONPATH"]),
    ("spark.executor.instances",
     n_nodes),
    ("spark.executor.cores",
     cores_per_node),
    ("spark.cores.max",
     total_cores)
  ])

  scontext = SparkContext(conf=sconf)

  run_analysis(scontext, dataset)
```

Listing 4 Setup function of a Spark benchmark. The analysis receives the created `SparkContext` object to distribute the application on the cluster

- Lines 6-14: The class `SLURMCluster` expects the information needed to create a job, i.e. the resources that it should request to the Slurm manager. These include: the amount of cores per job, which corresponds to the cores of one node in these benchmarks; the amount of Dask processes, which is set to the amount of cores per node in order to have no Python multithreading; the name of the Slurm queue where the job should be submitted; a few extra options that make sure the job will gain exclusive access to the node.
- Line 15: Calling the `scale` method on the created object will launch as many jobs as the number provided as argument. For these benchmarks, one job per node is requested.
- Line 18: Before starting the RDataFrame analysis, the Dask client waits for all the Slurm jobs to be started and the Dask workers to be ready. This is done to have consistent time to plot measurements for the purposes of the benchmarks. In a real scenario, the application would start as soon as at least one job is ready, in order to minimise the waiting time for the user.

```
from dask.distributed import Client
from dask_jobqueue import SLURMCluster
def main_dask(
  n_nodes, cores_per_node, dataset):

  cluster = SLURMCluster(
    cores=cores_per_node,
    processes=cores_per_node,
    queue=QUEUENAME,
    job_extra_directives=[
      "--exclusive",
      "--ntasks-per-node=1"
    ]
  )
  cluster.scale(n_nodes)

  daskclient = Client(cluster)
  daskclient.wait_for_workers(n_nodes)

  run_analysis(daskclient, dataset)
```

Listing 5 Setup function of a Dask benchmark. The analysis receives the created `Client` object to distribute the application on the cluster

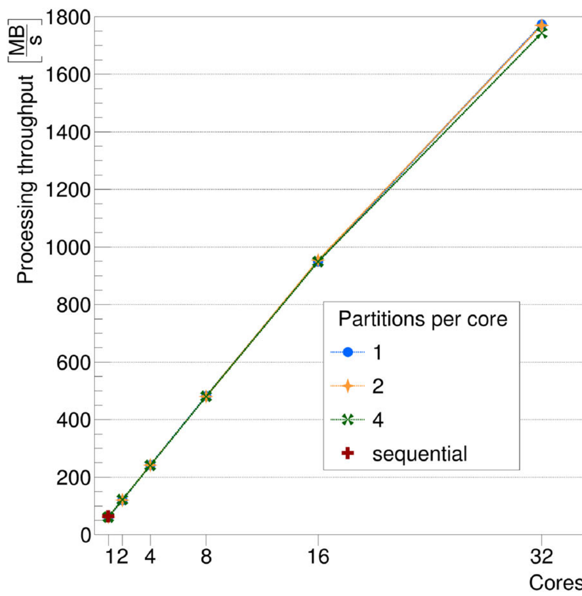
The benchmarks are executed both with the new Dask backend and the Spark backend of distributed RDataFrame. Each benchmark is repeated ten times, using from one to 64 computing nodes to distribute the computations. 32 distinct processes are run concurrently on each node (one process per core). In Spark, this is handled through Java Virtual Machine (JVM), whereas Dask spawns a different Python process for each core. For each test, the number of chunks in which the dataset is split is four times the number of cores used for that particular test.

Source code of the tests discussed in this section is publicly available on GitHub [47].

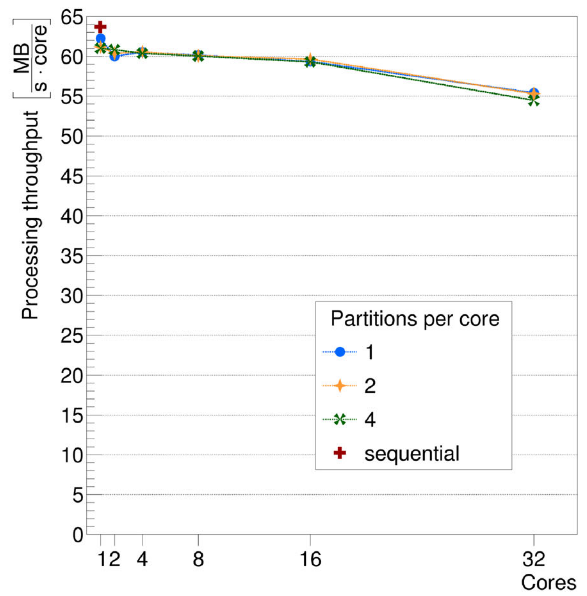
5.4 Results and Discussion

The results of the tests done on one computing node are shown in Fig. 2. The lines in the figure refer to benchmarks of distributed RDataFrame with the Dask backend, using a variable number of cores of the node and also an increasing number of tasks per core. Each task processes a separate chunk of the original dataset. The figures also report the result of the original application, which uses the traditional RDataFrame version processing the dataset sequentially.

In Fig. 2a, the processing throughput is reported in terms of Megabytes of data processed per second. Generally, changing the number of partitions per core



(a)



(b)

Fig. 2 Processing throughput achieved on a single computing node, with increasing number of cores and partitions per core. In each plot, three lines indicate the results of increasing the number of cores used in the benchmark. Each line corresponds to a different number of partitions per core. The result of running the original analysis sequentially is also indicated at the 1 core data point of the x axis. a: Processing throughput expressed in Megabytes per second. b: Processing throughput normalised by the number of cores

doesn't affect the overall throughput of the benchmark, with the same number of cores. This is a positive result because it means that the Dask backend is able to properly process multiple tasks assigned to the same core without creating imbalance.

In Fig. 2a, the throughput shown in the previous figure is normalised by the number of cores used in each benchmark run. Overall, the highest throughput per core is achieved by the benchmark running the original analysis with RDataFrame sequentially. This can be explained easily since there is no extra scheduling of tasks and remote communications involved, so it runs slightly faster. It should be noted that the benchmark using the Dask backend and running with one core and one partition has a slightly higher throughput than the benchmarks using one core and two or four partitions. Since there is only one processing core, it can be expected that having more than one task brings some overhead. When more cores are used and there is more than one task per core, different tasks can be sent to those cores that are free. Potentially, this can be very beneficial when the dataset is imbalanced (different files having very different number of entries) or when it is read remotely and network I/O becomes an issue. The main objective of the distributed mode is still to use the full potential of the node. Varying the number of partitions per core leads to the same throughput when considering the same amount of cores. The throughput per core shows an almost flat line with respect to the number of cores. The lowest point is reached when all 32 cores of the node are used at the same time. Since the node is being fully utilised, the processing tasks are influenced by other processes happening like I/O, Dask internal communication threads. Nonetheless, the drop in normalised throughput between using one core and 32 cores is around 10%, while achieving a 28 times higher nominal throughput.

The results obtained in the single node scenario also had some influence on the methodology of the benchmarks with multiple nodes. For example, setting the number of chunks to four times the number of cores can be safely done knowing that it will not decrease the throughput notably with any number of cores. At the same time, it allows for an overall better scheduling due to finer task granularity.

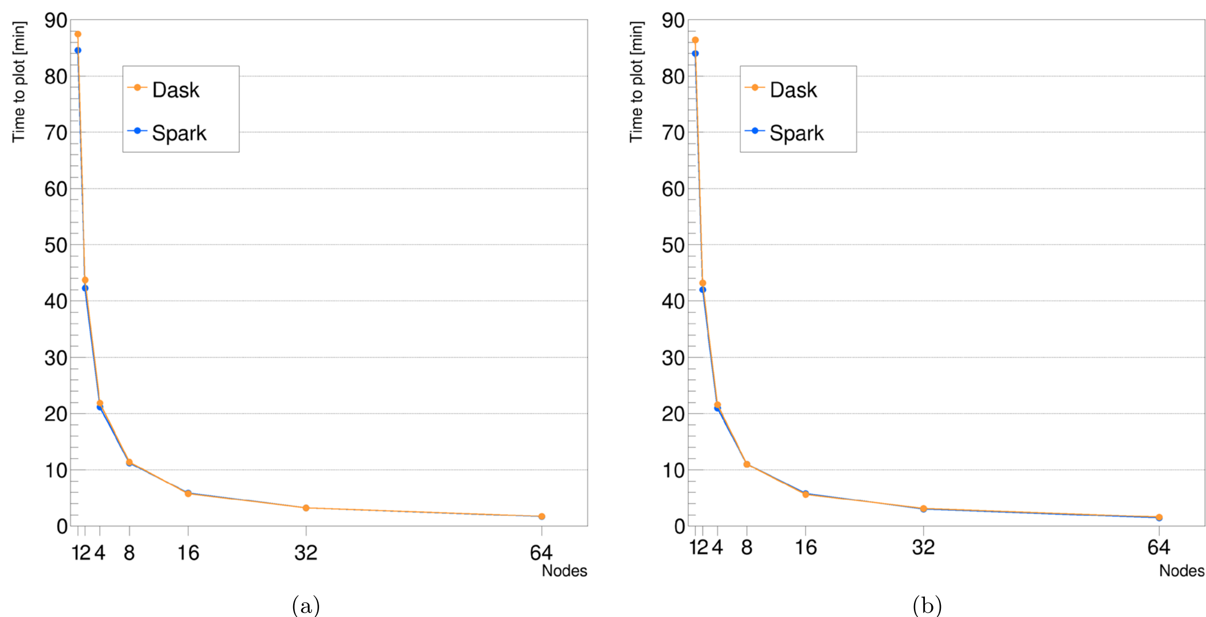


Fig. 3 Time to plot (in minutes) achieved with an increasing number of nodes. 32 concurrent processes are run on each node. a: Time to plot of the first run of the benchmark. b: Average time to plot for consequent runs of the benchmark

Figure 3 shows the time to plot of the benchmark scaling to multiple nodes. For this figure and following similar ones, it must be noted that: the image on the left shows the time to plot of the first run of the benchmark at each node count; the image on the right shows the average time to plot of the following runs at each node count; error bars for the average are not visible in the plot because they are too small with respect to the y-axis scale. In this configuration, the time to plot experienced by the user continuously decreases with the increased number of nodes, from slightly less than 90 minutes to less than 2 minutes. As far as the time to plot is concerned, there is no appreciable overhead in the first run of the benchmark with respect to following runs.

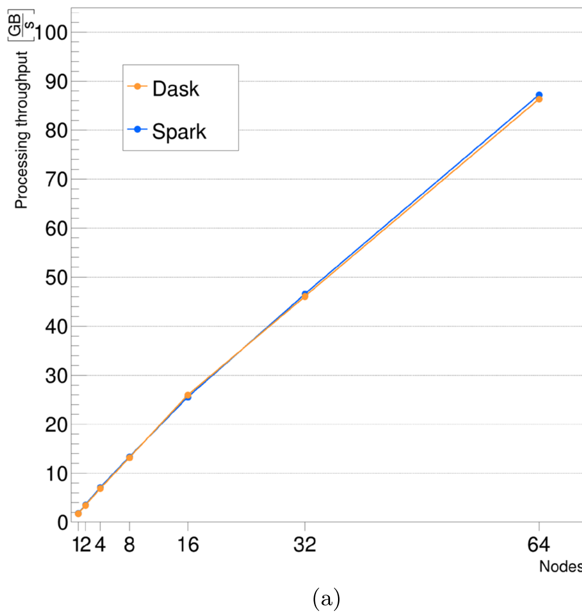
Figure 4 shows the same results expressed in processing throughput (in Gigabytes per second). Here, the overhead of the first run can be better appreciated. The peak processing throughput achieved in the first run is 87 GB/s, whereas in following runs a peak of 102 GB/s is reached when using the Spark backend. The Dask backend can still achieve a very high throughput, although slightly lower than the Spark backend.

Figure 5 shows the speedup achieved when running on an increasing number of nodes. The speedup in the first run shows a change of slope after the 16 nodes count and stays further away from the linear line. Consecutive runs show a better trend for both backends, closer to a linear behaviour with respect to the number of nodes used.

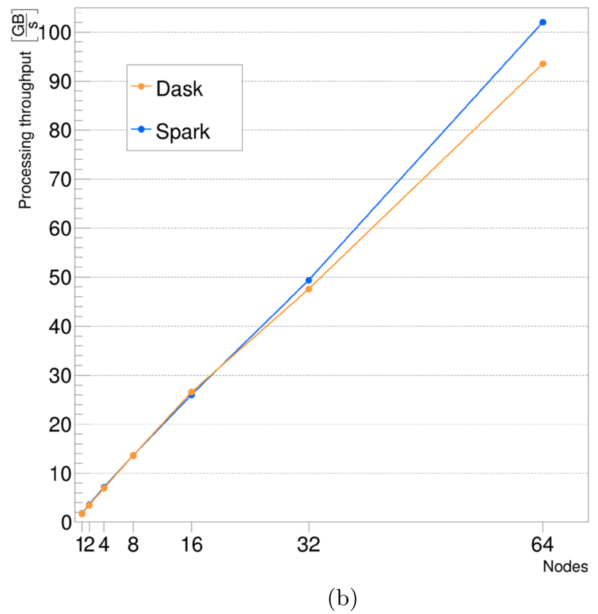
All these results show that the new Dask backend performs on par with the previously existing Spark backend. In general, the distributed RDataFrame tool can parallelise well even when thousands of computing cores are used.

It should be noted that, from the comparison of the time to plot measurements with the processing throughput of Figs. 3 and 4, the experience for the user does not change significantly between the first run and consecutive runs. The highest difference in time to plot is present when using only one node and it is just less than two minutes with respect to an overall runtime of almost ninety minutes. The overhead present in the first run becomes more evident only when discussing the throughput and trying to lower it becomes important in the effort to best utilise cluster resources.

Figure 4 clearly shows that the first run of the benchmark has a lower throughput than consecutive runs. This is mainly due to the necessary startup routines performed by ROOT and its C++ interpreter. Furthermore, when using RDataFrame with the Python bindings, users still run their C++ functions by passing them as Python strings to the RDataFrame API



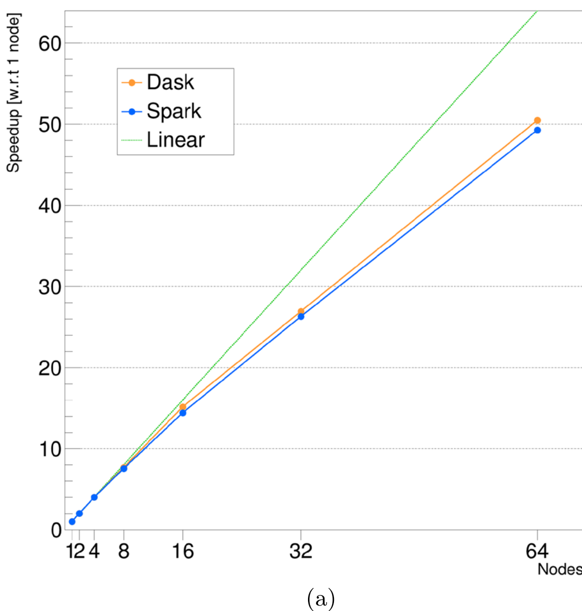
(a)



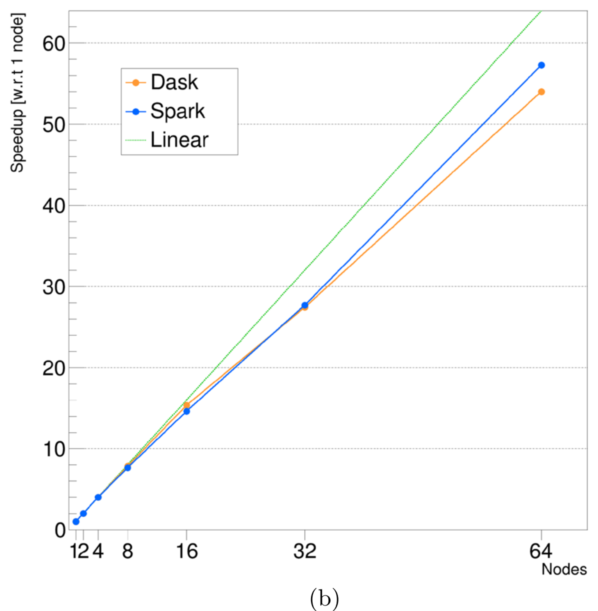
(b)

Fig. 4 Processing throughput (in Gigabytes per second) achieved with an increasing number of nodes. 32 concurrent processes are run on each node. a: Throughput achieved by the first run of the benchmark. b: Average throughput for consequent runs of the benchmark

functions. These strings then need to be just-in-time (JIT) compiled by the ROOT interpreter. This operation, also called JITting, incurs in initialisation cost the first time it is done in a certain Python process and will cache some information internally for later use. Notably, most of the functions, C++ template



(a)



(b)

Fig. 5 Speedup achieved with an increasing number of nodes. 32 concurrent processes are run on each node. Speedup is relative to the result obtained when using one node. a: Speedup obtained with the first run of the benchmark. b: Speedup obtained with consequent runs of the benchmark

instantiations, constant structures that belong to the computation graph. Since the execution of the distributed application is done with multiprocessing on each computing node, this means that the initialisation cost is incurred once per core used in the benchmark. Potentially, this can be improved with a hybrid parallelism approach. That is, if distributed RDataFrame could communicate with the execution engines in a way that the given resources can be completely controlled by the analysis tool, then instead of relying on multiprocessing and the scheduling of the various engines the parallelism could be achieved by the implicit multithreading capabilities already available within ROOT. From the point of view of a single node, the tool would start a single Python process which would then run RDataFrame in multi-threading with as many threads as available cores on the node. This would thus decrease startup costs and intranode communications between different Python processes happening on the same node. With the resource configuration used in this work, this improvement would mean running the ROOT startup routines only once instead of 32 times per computing node.

The benchmark runs following the first run, presented in Figs. 4 and 5, show a better throughput and speedup behaviour overall. Although the throughput achieved in this case is very high at around 100 GB/s with both backends, some non-idealities are still present and become more evident after the 32 nodes mark. This is due to the fact that even though the interpreter has been already initialised at this point, some JITting is still involved in each task. A clear improvement on this side could be brought by creating and compiling the RDataFrame workflow in a certain process when the first task starts, then caching it in memory and reusing it in subsequent task happening in the same process.

The results for the next configuration of this benchmark, reading data from remote CERN storage facilities rather than locally on the nodes, are shown in Fig. 6. In this case, only the runs with the Dask backend are shown, so that the comparison with Spark is not influenced by variability due to the remote I/O. In fact, this image shows the distributions of the time to plot measurements with each different number of nodes. The distributions are much wider than in the previous cases, sometimes also showing outliers (as indicated by the "x" marks at 1 and 64 nodes). This large variance could be helped by an even finer

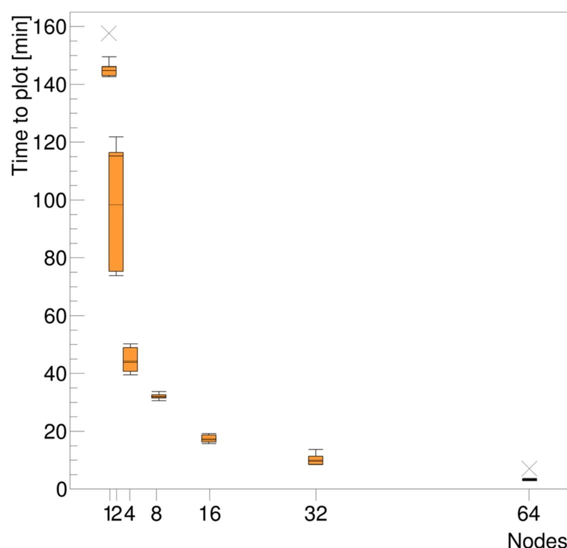


Fig. 6 Time to plot (in minutes) achieved with an increasing number of nodes. 32 concurrent processes are run on each node. Data processed in the experiments are read remotely from the storage facilities at CERN. A box plot of the distribution of results is shown for each point on the x axis, ten runs per node count. The box spans from the first to the third quartile. Inside the box, a dashed line represents the mean, a solid line represents the median. "x" marks at 1 and 64 nodes represent outliers, those points that are outside the range of the whiskers. This range is defined as the distance between the first quartile and the third (also called interquartile range) multiplied by 1.5

grained task distribution on the application side, but it is also influenced by the status of the network and possibly the implementation of the remote I/O. It is thus left for future studies directed at this specific issue.

Both in the first run results discussed above and in consecutive runs, the amount of JITting is proportional to the number of tasks. This also raises the question of finding a good balance in parallelising the processing of the input dataset. On the one hand, creating more tasks means splitting the dataset in smaller chunks, thus leaving more room for the scheduler to assign work to the computing nodes, avoiding possible imbalances due to some nodes being slightly slower than others or some parts of the dataset requiring higher computational load than others. On the other hand, creating more tasks leads to more overhead in spawning them and recreating the computation graph on-the-fly.

The slopes of the speedup lines shown with both backends do not diverge greatly from the linear behaviour shown for reference in Fig. 5. Although it might be possible in principle to reach this behaviour

when using multiple nodes, the non-idealities discussed in scaling the computation graph justify the missing performance gains. It should be especially noted that the overhead discussed is per-task, not per-core or per-node. This means that the more tasks, the more overhead on the whole runtime of the distributed execution. This fact clashes with the intuition that splitting the input dataset in more chunks should allow for a more fine-grained scheduling by the execution engine, thus avoiding potential slowdowns due to some tasks or nodes taking longer than others. In general, optimising dataset splitting is a non-trivial problem of distributed execution engines, with sparse literature suggesting different ways to choose the splitting value. For example, the Spark documentation suggests setting it to 2 or 3 times the amount of available CPUs in the cluster [48]. But other strategies exist depending on the number of files and size of the input dataset and the available resources [49] or even on the choice of the number of chunks being done statically or dynamically [50]. The results discussed previously refer to benchmarks with four tasks per computing core, which, for the purposes of this work, seems to strike a good balance between making the load even on the nodes and avoiding too much overhead.

6 Conclusions

This paper has presented a novel backend for the distributed ROOT RDataFrame tool, which relies on the popular Dask framework to steer the computations to a distributed cluster. The backend implementation involved different efforts, such as: a plugin to interface the existing RDataFrame code with Dask primitives; a redesign of the computation graph trigger on the computing nodes, so that the C++ event loop does not interfere with Dask Python threads; a more generic implementation of task creation on the client side, so that no remote I/O is involved locally but only on the distributed nodes. These changes have been committed to the upstream ROOT repository and are available to all users.

A HEP analysis example has been used to test the scalability of distributed RDataFrame. Initially, the analysis was run on a single node, to compare the throughput obtained running sequentially against the throughput obtained using multiple cores of the node. This was also normalised to the number of cores used in each run, showing that the extra work in scheduling the tasks does not account for a high drop in throughput. Also, it was demonstrated that the dataset can be split in many partitions without loss in performance, opening the door to heavier workloads involving remote I/O where the finer granularity may lead to improved balancing. At a second stage, the analysis was scaled on multiple nodes, comparing the already existing Spark backend with the newly developed Dask backend presented in this paper on an HPC cluster at CERN. The test was run using up to a total of 64 nodes (2048 cores). Results show very high raw processing throughput values (more than 100 GB/s with the highest core count) and good scaling, with non-idealities showing up after the 512 cores mark. Both backends perform similarly, with Dask having a slight disadvantage when more than 1000 cores are used, which can be due to being a less mature framework than Spark in the data science ecosystem.

The overhead brought by JITting found in the benchmark runs with very high core counts is understood and potential improvements have been described. These will help in further addressing very large scale requirements of HEP distributed computing.

Acknowledgments The hardware used to perform the experimental evaluation of the software was made available by CERN.

Author Contributions The following statement is based on CRediT taxonomy, giving contributions of each author in the work:

- Vincenzo Eduardo Padulano: Conceptualization, Investigation, Writing - original draft, Software, Methodology
- Ivan Donchev Kabazhov - Conceptualization, Investigation
- Enric Tejedor Saavedra: Conceptualization, Supervision, Investigation, Writing - review and editing, Validation
- Enrico Guiraud: Conceptualization, Supervision, Investigation, Writing - review and editing, Validation
- Pedro Alonso-Jordá: Supervision, Writing - review and editing, Validation

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature. This work benefited from the support of grant PID2020-113656RB-C22 funded by Ministerio de Ciencia e Innovación (Spain) MCIN/AEI/10.13039/501100011033.

Data Availability All benchmarks performed in this work and the information about the datasets used are available in a public repository [47].

Compliance with Ethical Standards This work benefited from the support of grant PID2020-113656RB-C22 funded by Ministerio de Ciencia e Innovación (Spain) MCIN/AEI/10.13039/501100011033.

The hardware used to perform the experimental evaluation of the software was made available by CERN.

Ethics approval and consent to participate Not applicable.

Consent for Publication All authors consent to the publication of this article.

Competing interests Financial interests are disclosed in Section 7.5. The authors declare no other competing interest for this work.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visithttp://creativecommons.org/licenses/by/4.0/.

References

1. Apollinari, G., Béjar Alonso, I., Brüning, O., Fessia, P., Lamont, M., Rossi, L., Tavian, L.: High-luminosity large hadron collider (HL-LHC): technical design report V. 0.1. Technical report CERN. https://doi.org/10.23731/CYRM-2017-004 (2017)
2. Elsen, E.: A roadmap for HEP software and computing R&D for the 2020s. Comput Softw Big Sci, vol 16(3). https://doi.org/10.1007/s41781-019-0031-6 (2019)
3. Brun, R., Rademakers, F.: ROOT — an object oriented data analysis framework. Nuclear Instr. Methods Phys. Res. Section A Accelerators, Spectrometers, Detectors Assoc. Equip. **389**(1), 81–86 (1997). https://doi.org/10.1016/S0168-9002(97)00048-X. New computing techniques in physics research V
4. Blomer, J., Canal, P., Naumann, A., Piparo, D.: Evolution of the ROOT tree I/O. EPJ Web Conf. **245**, 02030 (2020). https://doi.org/10.1051/epjconf/202024502030
5. Lopez-Gomez, J., Blomer, J.: RNTUple performance: status and outlook. arXiv:2022.09043. https://doi.org/10.48550
6. Piparo, D., Canal, P., Guiraud, E., Valls Pla, X., Ganis, G., Amadio, G., Naumann, A., Tejedor Saavedra, E.: RDAtaframe: easy parallel ROOT analysis at 100 threads. EPJ Web Conf. **214**, 06029 (2019). https://doi.org/10.1051/epjconf/201921406029
7. Bird, I.: Computing for the large hadron collider. Annu. Rev. Nucl. Part. Sci. **61**(1), 99–118 (2011). https://doi.org/10.1146/annurev-nucl-102010-130059
8. Team, R., 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., Wunsch, S.: Software challenges for HL-LHC data analysis. arXiv:2004.07675. 10.48550 (2020)
9. Tannenbaum, T., Wright, D., Miller, K., Livny, M.: Condor – a Distributed Job Scheduler. In: Sterling, T. (ed.) Beowulf Cluster Computing with Linux. MIT Press (2001)
10. Jette, M., Dunlap, C., Garlick, J., Grondona, M.: Slurm: simple linux utility for resource management. Technical report, LLNL. https://www.osti.gov/biblio/15002962 (2002)
11. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster computing with working sets. In: Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing. HotCloud'10, p. 10. USENIX association. https://www.usenix.org/conference/hotcloud-10/spark-cluster-computing-working-sets (2010)
12. Rocklin, M.: Dask: parallel computation with blocked algorithms and task scheduling. In: Huff, K., Bergstra, J. (eds.) Proceedings of the 14th Python in Science Conference, pp. 130–136. SciPy (2015)
13. Rilee, M., Griessbaum, N., Kuo, K.-S., Frew, J., Wolfe, R.: STARE-based integrative analysis of diverse data using dask parallel programming demo paper. In: Proceedings of the 28th International Conference on Advances in Geographic Information Systems. SIGSPATIAL '20, pp. 417–420. Association for computing machinery. https://doi.org/10.1145/3397536.3422346 (2020)
14. Gharat, J., Kumar, B., Ragha, L., Barve, A., Jeelani, S.M., Clyne, J.: Development of NCL equivalent serial and parallel python routines for meteorological data analysis. Int. J. High Performance Comput. Appl., https://doi.org/10.1177/10943420221077110 (2022)
15. Hamman, J.J., Rocklin, M., Abernathy, R.M.: Pangeo: a big-data ecosystem for scalable earth system science. In: 20th EGU General Assembly, EGU2018, p. 12146. The SAO/NASA astrophysics data system (ADS) (2018)
16. Fan, S., Linke, M., Paraskevakos, I., Gowers, R.J., Gecht, M., Beckstein, O.: PMDA - Parallel molecular dynamics analysis. In: Calloway, C., Lippa, D., Niederhut, D., Shupe, D. (eds.) Proceedings of the 18th Python in Science Conference, pp. 134–142. SciPy. https://doi.org/10.25080/Majora-7ddc1dd1-013 (2019)
17. Dask: dask.dataframe documentation. https://docs.dask.org/en/stable/dataframe.html . Accessed 25 Nov 2022 (2022)
18. Salloum, S., Dautov, R., Chen, X., Peng, P.X., Huang, J.Z.: Big data analytics on Apache Spark. Int. J. Data Sci. Anal. **1**, 145–164 (2016). https://doi.org/10.1007/s41060-016-0027-9
19. Khan, M.A., Karim, M.R., Kim, Y.: A two-stage big data analytics framework with real world applications using spark machine learning and long Short-Term memory network. Symmetry, vol. 10(10). https://doi.org/10.3390/sym10100485 (2018)

20. Ramírez-Gallego, S., Mouriño-Talín, H., Martínez-Rego, D., Bolón-Canedo, V., Benítez, J.M., Alonso-Betanzos, A., Herrera, F.: An information theory-based feature selection framework for big data under apache spark. IEEE Trans. Syst. Man Cybern. Syst. **48**(9), 1441–1453 (2018). https://doi.org/10.1109/TSMC.2017.2670926
21. Chaudhari, A.A., Mulay, P.: SCSI: real-time data analysis with cassandra and spark, pp. 237–264. Springer. https://doi.org/10.1007/978-981-13-0550-4_11 (2019)
22. Shyam, R., Bharathi Ganesh, H.B., Sachin Kumar, S., Poornachandran, P., Soman, K.P.: Apache spark a big data analytics platform for smart grid. Proced. Technol. **21**, 171–178 (2015). https://doi.org/10.1016/j.protcy.2015.10.085
23. Shin, H., Lee, K., Kwon, H.: A comparative experimental study of distributed storage engines for big spatial data processing using GeoSpark. J. Supercomput. **78**, 2556–2579 (2022). https://doi.org/10.1007/s11227-021-03946-7
24. Graur, D., Müller, I., Proffitt, M., Fourny, G., Watts, G.T., Alonso, G.: Evaluating query languages and systems for high-energy physics data. Proc. VLDB Endow. **15**(2), 154–168 (2021). https://doi.org/10.14778/3489496.3489498
25. Feichtinger, D., Canal, P., Reed, C., Loizides, C., Ballintijn, M., Rademakers, F., Peters, A.J., Kickinger, G., Iwaszkiewicz, J., Ganis, G., Brun, R., Bellenot, B., Feichtinger, D., Canal, P., Reed, C., Loizides, C., Ballintijn, M., Rademakers, F., Peters, A.J., Kickinger, G., Iwaszkiewicz, J., Ganis, G., Brun, R., Bellenot, B.: PROOF - the parallel ROOT facility. In: 2006 15th IEEE International Conference on High Performance Distributed Computing, pp. 379–380. EDP sciences. https://doi.org/10.1109/HPDC.2006.1652193 (2006)
26. Chatrchyan, S., et al.: The CMS experiment at the CERN LHC. JINST **3**, 08004 (2008). https://doi.org/10.1088/1748-0221/3/08/S08004
27. Sehrish, S., Kowalkowski, J., Paterno, M.: Spark and HPC for high energy physics data analyses. In: 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 1048–1057. IEEE, Lake Buena Vista, FL, USA. https://doi.org/10.1109/IPDPSW.2017.112 (2017)
28. Gutsche, O., Cremonesi, M., Elmer, P., Jayatilaka, B., Kowalkowski, J., Pivarski, J., Sehrish, S., Surez, C.M., Svyatkovskiy, A., Tran, N.: Big data in HEP: a comprehensive use case study. J. Phys. Conf. Ser. **898**, 072012 (2017). https://doi.org/10.1088/1742-6596/898/7/072012
29. Gutsche, O., Canali, L., Cremer, I., Cremonesi, M., Elmer, P., Fisk, I., Girone, M., Jayatilaka, B., Kowalkowski, J., Khristenko, V., Motesnitsalis, E., Pivarski, J., Sehrish, S., Surdy, K., Svyatkovskiy, A.: CMS analysis and data reduction with apache spark. J. Phys. Conf. Ser. **1085**, 042030 (2018). https://doi.org/10.1088/1742-6596/1085/4/042030
30. Avati, V., Blaszkiewicz, M., Bocchi, E., Canali, L., Castro, D., Cervantes, J., Grzanka, L., Guiraud, E., Kaspar, J., Kothuri, P., Lamanna, M., Malawski, M., Mnich, A., Moscicki, J., Murali, S., Piparo, D., Tejedor, E.: Declarative big data analysis for high-energy physics: TOTEM use case. In: Yahyapour, R. (ed.) Euro-par 2019: Parallel Processing, pp. 241–255. Springer (2019)
31. Baranowski, Z., Kleszcz, E., Kothuri, P., Canali, L., Castellotti, R., Marquez, M.M., De Barros, N.G.M., Motesnitsalis, E., Mrowczynski, P., Duran, J.C.L.: Evolution of the hadoop platform and ecosystem for high energy physics. EPJ Web Conf. **214**, 04058 (2019). https://doi.org/10.1051/epjconf/201921404058
32. Adamec, M., Attebury, G., Bloom, K., Bockelman, B., Lundstedt, C., Shadura, O., Thiltges, J.: Coffea-casa: an analysis facility prototype. EPJ Web Conf. **251**, 02061 (2021). https://doi.org/10.1051/epjconf/202125102061
33. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008). https://doi.org/10.1145/1327452.1327492
34. Vavilapalli, V.K., Murthy, A.C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B., Baldeschwieler, E.: Apache hadoop YARN: yet another resource negotiator. In: Proceedings of the 4th Annual Symposium on Cloud Computing. SOCC '13. Association for computing machinery. https://doi.org/10.1145/2523616.2523633 (2013)
35. Kubernetes: homepage. https://kubernetes.io/. Accessed 25 Nov 2022 (2022)
36. NumPy: homepage. https://numpy.org/. Accessed 25 Nov 2022 (2022)
37. Pandas: homepage. https://pandas.pydata.org/. Accessed 25 Nov 2022 (2022)
38. Nitzberg, B., Schopf, J.M., Jones, J.P.: PBS pro: grid computing and scheduling attributes, pp. 183–190. Kluwer academic publishers, USA (2004)
39. Hudak, P.: Conception, evolution, and application of functional programming languages. ACM Comput. Surv. **21**(3), 359–411 (1989). https://doi.org/10.1145/72551.72554
40. Dozza, M., Bärgman, J., Lee, J.D.: Chunking: a procedure to improve naturalistic data analysis. Accident Anal. Prevention **58**, 309–317 (2013). https://doi.org/10.1016/j.aap.2012.03.020
41. Rew, R.: Chunking data: why it matters. https://www.unidata.ucar.edu/blogs/developer/en/entry/chunking_data_why_it_matters (2013)
42. Padulano, V.E., Villanueva, J.C., Guiraud, E., Saavedra, E.T.: Distributed data analysis with ROOT RDataframe. EPJ Web Conf. **245**, 03009 (2020). https://doi.org/10.1051/epjconf/202024503009
43. Dask: dask.delayed documentation. https://docs.dask.org/en/stable/delayed.html. Accessed 25 Nov 2022 (2022)
44. Spark: web UI. Accessed 25 NOv 2022. https://spark.apache.org/docs/latest/web-ui.html (2022)
45. Dask: dashboard diagnostics. Accessed 25 Nov 2022. https://docs.dask.org/en/stable/dashboard.html (2022)
46. Wunsch, S.: Analysis of the di-muon spectrum using data from the CMS detector taken in 2012. https://doi.org/10.7483/OPENDATA.CMS.AAR1.4NZQ (2019)
47. Padulano, V.E.: Test suite repository. Accessed 25 Nov 2022. https://github.com/vepadulano/distRDF_benchmarks (2022)
48. Spark: tuning guide. Accessed 25 Nov 2022. https://spark.apache.org/docs/latest/tuning.html#level-of-parallelism (2022)

49. Gupta, A.: Building partitions for processing data files in apache spark. Accessed 25 Nov 2022. https://medium.com/swlh/building-partitions-for-processing-data-files-in-apache-spark-2ca40209c9b7 (2020)
50. Bertolucci, M., Carlini, E., Dazzi, P., Lulli, A., Ricci, L.: Static and dynamic big data partitioning on apache spark, vol. 27, pp. 489–498. IOS Press. https://doi.org/10.3233/978-1-61499-621-7-489 (2016)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.