The Journal of Supercomputing (2023) 79:8940–8965
https://doi.org/10.1007/s11227-022-05016-y

# Leveraging an open source serverless framework for high energy physics computing

**Vincenzo Eduardo Padulano[1,2] · Pablo Oliver Cortés[1] · Pedro Alonso-Jordá[1] · Enric Tejedor Saavedra[2] · Sebastián Risco[3] · Germán Moltó[3]**

Accepted: 16 December 2022 / Published online: 2 January 2023


## Abstract

CERN (Centre Europeen pour la Recherce Nucleaire) is the largest research centre for high energy physics (HEP). It offers unique computational challenges as a result of the large amount of data generated by the large hadron collider. CERN has developed and supports a software called *ROOT*, which is the de facto standard for HEP data analysis. This framework offers a high-level and easy-to-use interface called *RDataFrame*, which allows managing and processing large data sets. In recent years, its functionality has been extended to take advantage of distributed computing capabilities. Thanks to its declarative programming model, the user-facing API can be decoupled from the actual execution *backend*. This decoupling allows physical analysis to scale automatically to thousands of computational cores over various types of distributed resources. In fact, the distributed *RDataFrame* module already supports the use of established general industry engines such as Apache Spark or Dask. Notwithstanding the foregoing, these current solutions will not be sufficient to meet future requirements in terms of the amount of data that the new projected accelerators will generate. It is of interest, for this reason, to investigate a different approach, the one offered by serverless computing. Based on a first prototype using *AWS Lambda*, this work presents the creation of a new *backend* for *RDataFrame* distributed over the *OSCAR* tool, an open source framework that supports serverless computing. The implementation introduces new ways, relative to the *AWS Lambda*-based prototype, to synchronize the work of functions.

**Keywords** CERN · *ROOT* · *OSCAR* · Serverless computing · *AWS Lambda*

✉ Vincenzo Eduardo Padulano
vincenzo.eduardo.padulano@cern.ch

Extended author information available on the last page of the article

## 1 Introduction

One of the main instruments needed for research and progress in the field of High Energy Physics (HEP) is the particle accelerator, the largest of which is the Large Hadron Collider (LHC). This accelerator, built and operated by the European Organization for Nuclear Research (CERN), can generate up to one petabyte (PB) of data per second while operational.

The LHC does not work constantly, the generation of the physics events is organized in *runs*. The third run of the LHC began in July 2022 and is scheduled to last until 2025. In addition, the LHC is scheduled for an update, called High Luminosity Large Hadron Collider (HL-LHC) [7], which will start operations in 2029 and it is estimated that it will require between 50 and 100 times more computational resources than those currently used.

Figure 1 shows the computational requirements of CMS [42], one of the main experiments at the LHC. Its computational needs have been satisfied through progressive budget added to generational technological advances received year after year. However, as shown in the figure, once the HL-LHC is commissioned, these computing capabilities will fall far behind the necessary requirements [1]. Europeen pour la Recherce Nucleaire) is the largest research centre for High Energy Physics (HEP). It offers unique computational challenges as a result of the large amount of data generated by the Large Hadron Collider (LHC). CERN has developed and supports a software called ROOT, which is the de facto standard for HEP data analysis. This framework offers a high-level and

CERN and many other academic institutions share resources in order to satisfy the computing requirements of the LHC experiments. Collectively, these form a grid of computing clusters called Worldwide LHC Computing Grid (WLCG) [48]. Thus, distributed computing has been a popular topic of research in the HEP field for decades. In recent years, a few key research areas have been highlighted in an R &D project [1, 10, 43] in order to improve the entire HEP software landscape. These efforts range from the efficiency, scalability and performance of the

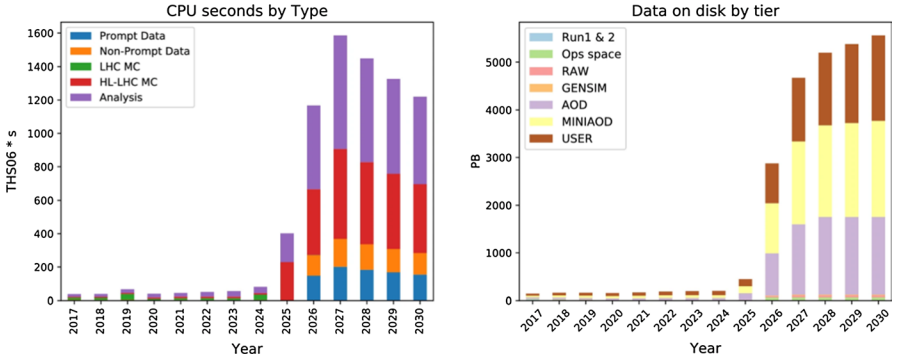


**Fig. 1** Estimation of computational and storage requirements of the CMS experiment for the HL-LHC [1]. (The HS06 unit refers to the score obtained by a system in the HEP-SPEC06 benchmark focused on analyzing the performance of systems for tasks similar to those performed by HEP analysis [23])

software currently used to take advantage of new architectures, to the adoption of new paradigms, such as the use of Artificial Intelligence (AI) in certain aspects of HEP or the use of *Big Data* models for data analysis.

In the context of data analysis, *ROOT* [13] is the de facto standard software framework used for storage, processing and visualization. It provides a high-level interface to data analysis, called *RDataFrame* [38]. This was initially developed for use on a single machine, either with one or multiple cores, but more recently it has been extended to automatically run workflows on distributed clusters by leveranging various *backends*, such as Apache Spark [50] or Dask [41]. The distributed extension of RDataFrame is also called *DistRDF* [36].

Traditional HEP distributed computing is based on batch computing systems on a managed cluster, where all the software and storage are provided by cluster administrators for the final user. Spark or Dask processing relies on the principle of infrastructure management and setting up resources on premises. On the contrary, in an unmanaged or serverless environment the users interface with abstract resources, which potentially allows even more massive parallelization across multiple machines compared to the managed approach. Furthermore, it may ease the development and deployment of parallel applications, an ongoing pain point for physics users. Thus, a serverless approach could provide a more scalable and more open alternative for HEP analysis workflows.

A first prototype of *serverless backend* for *RDataFrame* exists, that makes use of *AWS Lambda* [25]. Using that proposal as a starting point, this paper introduces a new serverless *backend* for *RDataFrame* based on *OSCAR* [40] (*Open Source Computing for Data-Processing Applications*). *OSCAR* is an *open source* solution that provides a self-scaling environment for *serverless* computing. *OSCAR* uses file-based events to run a *High Throughput Computing* model. The execution of the functions is done through the use of Docker [30] containers. This proposal provides various improvements to the model presented in the prototype backend with *AWS Lambda* among which we find the capability of *OSCAR* to be used in a cluster on-premises free of charge.

Summarizing all of the above, the novel contributions brought by this paper are:

- The implementation of a new *backend* for distributed processing of HEP data analysis in *ROOT* using *OSCAR*.
- A first study on the adaptation of the distributed computing model available so far in *DistRDF* to the file-based, event-driven model offered by *OSCAR*.
- The implementation of different strategies for serverless reduction algorithms, namely an uncoordinated strategy and a coordinated one. This is precisely described in Sect. 4.2.2.
- A performance evaluation of the backend implementation and the different reduction strategies, with a particular description of the different components of the workflow that may potentially add overhead to the runtime of the physics analysis.

It should be noted from this point going forward that, unlike *AWS Lambda*, *OSCAR* is an experimental tool. Thus, we state in advance in this introduction that some downsides were identified while using *OSCAR* that should be improved in future. At the same time, *OSCAR* is only an example of an open source serverless tool through which this work shows how *ROOT* itself can be adapted to run on various execution engines.

The rest of the paper is divided as follows. Section 3 describes the two main tools used in this work. Section 4 describes the whole implementation of the *backend* proposed in this work. Results are shown and discussed in Sect. 5. Finally, Sect. 6 concludes the paper with some considerations for future work.

## 2 State of the art

The *Function as a Service* computing model (*FaaS*), part of the larger serverless computing paradigm, has seen a steady increase in popularity ever since its first usages by big *cloud* providers, first being Amazon Web Services (AWS) in 2014 [3]. This model abstracts the user from a large part of the costs associated with infrastructure management, whether *on-premises* or in the *cloud*, and its configurations. The user only needs to provide the code of the functions to execute without the need to explicitly pre-provision any type of infrastructure. For this type of services, the scaling unit is the function.

Function executions in this model are *stateless* (although for a few years AWS has offered the possibility for *Lambda* functions to have a shared and persistent file system [8]). This means that during the execution of a function, they cannot depend internally on memory, mounted disks or similar elements that in other models provide information on the state of the distributed system. Clearly, this approach can be adapted quite well to the HEP data analysis requirements described in Sect. 1.

Another feature of this model is that the functions can be executed inside a container, providing the necessary flexibility to accommodate different workflows. For instance, the container may have an environment with a *ROOT* installation together with any other libraries that may be needed in the analysis.

The more generic *serverless* computing paradigm includes other components together with the functions. For example, streaming components that generate events that will be used to trigger function invocations. Or storage systems such as *Amazon S3* [5] or *MinIO* [31] to extract the data to be processed or store the results.

Every serverless execution engine relies on a type of abstraction layer for the infrastructure itself. This is usually provided by big vendors, such as Cloud Functions by Google [20], Lambda by Amazon [3], or open source solutions such as OpenWhisk [6] running on real or virtual machines or Knative [45] running ephemeral Docker containers directly on Kubernetes [46]. All the solutions offered by these players share many features. In some cases, communities compare the products in terms of cost efficiency and availability to make informed choices about the workflow they will propose to users. An example is offered by a recent overview of the serverless computing scenario in bioinformatics by Grzesik et al. [21].

Efficient orchestrating frameworks are useful in utilizing the power of serverless functions in data processing applications. One among these would be *PyWren* [24]. It allows to seamlessly distribute arbitrary Python code over multiple nodes with serverless functions. Needed objects and dependencies are serialized and sent to the Lambda execution environment in order to run the application on AWS resources natively. As of 2021, the original project is no longer maintained, but it was used as a basis for interesting extensions, including *NumPyWren* for numerical algebra [44]. This package provided support for distributed computation of *serverless* linear algebra problems on *AWS Lambda*, along with the development of a language, *LAmbdaPACK*, designed to implement highly parallel linear algebra algorithms in *serverless* environments. A recent alternative to *NumPyWren* is *Wukong* [16] a framework that allows the execution of jobs divided into tasks that can form complex directed acyclic graphs through a decentralized scheduler that greatly improves the features of *NumPyWren*.

The serverless research scenario is quite wide. Other works in this line of research are *MARLA* [19] (MApReduce on *AWS Lambda*) and *SCAR* [39] (Serverless Container-aware ARchitectures). *MARLA* is a framework that supports the MapReduce model in *AWS Lambda* for Python. One of the advantages of *MARLA* over other similar frameworks is that it is in charge of managing the entire MapReduce process, from the partitioning of the data to the generation of the final result, where the user only has to define the Map and Reduce functions of the process. *SCAR* is a framework that offers the possibility of executing any programming language in *AWS Lambda* through the use of containers, allowing the execution of any type of application in a *FaaS* environment, which supported this execution model before AWS itself. In particular, the use of containerized environments to run the serverless functions has become more and more popular thanks to a few key advantages it brings in terms of reproducibility and ease of deployment. But this also comes with issues to be addressed. Bila et al. [9] discuss how containers may hide vulnerabilities which can be exploited at runtime. Li et al. [28] present the benefits of reusing the same container for multiple serverless functions to avoid cold startups. Oakes et al. [34] go as far as implementing a new container runtime especially aimed at serverless workflows to obtain factors of speedup with respect to using Docker.

Serverless workflows are not a common topic in High Energy Physics literature. A recent review has highlighted a few motivations that would make the *FaaS* model applicable to the online trigger systems employed by LHC experiments [33]. This paper theorizes that inference using neural networks could be triggered by the events streaming from the accelerator. A more concrete example is provided by a paper related to *CernVM-FS* (*CVMFS*) [11], a file system that provides the software distribution backbone for collaborations in the field. The paper describes advances in the publishing system of the software distributions. The default model has a single server node responsible for the compilation of all the libraries and only upon a commit from this machine the distribution would be actually published and replicated. The envisioned changes would have any machine enabled with *CVMFS* publish the changes to cloud storage (e.g., Amazon S3) through a gateway serverless function [12]. Regarding the data analysis use case, a good example of serverless engine is provided by *Lambada* [32], that

adds facilities on top of *AWS Lambda* to steer the serverless functions and shows a cost-efficient usage of the resources.

The contribution of this work aims at furthering the knowledge of the FaaS paradigm applied to the HEP data analysis use case. With respect to the literature reviewed, the execution backend based on *OSCAR* can provide an open platform for research in this field and potentially offer physicists a simple to use and cost-effective solution.

## 3 Tools

The two main software tools used in this work are *ROOT* and *OSCAR*. The integration of a C++ framework, such as *ROOT* into a serverless ecosystem brings a few challenges like its packaging and the instrumentation of the C++ code within a serverless function invocation. This section highlights the different aspects that are relevant to this study when using these two key tools.

### 3.1 *ROOT*

*ROOT* is a software developed at CERN for the analysis of high energy physics data (HEP). This software has been updated since its initial conception in 1994, expanding its functionality and adapting to new technologies. It is made up of several object oriented components that allow efficient management and analysis of large volumes of data.

One of the most important features of ROOT is its solid and flexible I/O layer. Users can write to disk any type of C++ object, including custom classes, in a compressed binary format managed by the *TFile* class. It is common to find in a *TFile* all the most important objects needed for HEP statistics, such as histograms, which in ROOT are implemented with the *TH1* class. When mentioning physics datasets, the traditional *ROOT* implementation is done with the *TTree* class. This is a flexible container that can also store any type of C++ object, organizing its contents in columns (usually called branches). Different branches can be read independently, making *TTree* a truly columnar data format implementation. A *TTree* is stored on disk via the *TFile* class as well and multiple trees can be joined together either vertically or horizontally to create a larger dataset. Most notably, a *ROOT* file can be read remotely via protocols such as HTTP or XRootD [18].

Another relevant feature for this work is the presence of a C++ interpreter, called cling [47], and dynamic Python bindings based on the cppyy [26] library. These two together offer the possibility for higher-level tools like *RDataFrame* to provide users with modern, ergonomic interfaces that can interact with other common data science libraries, which most often offer Python APIs.

#### 3.1.1 *RDataFrame*

*RDataFrame* is the high-level data analysis interface offered by *ROOT*. Compared to the more traditional imperative programming model, *RDataFrame* offers a lazy

and declarative model inspired by other popular libraries such as Spark [50] or pandas [29]. It supports many types of input data formats, the most used one is *TTree* but for example also *CSV* files and *numpy* [22] arrays can be processed.

The declarative model has the clear advantage of taking away responsibility from the user, who only needs to declare the desired operations on the dataset, allowing for under-the-hood optimizations in the processing and I/O scheduling. The lazy approach used by the API means that the execution of the requested operations only starts when the final results are queried in the application, for example when a histogram is drawn for the first time.

The data generated by the physical events are statistically independent, so *RDataFrame* is able to abstract from the different underlying data structures in which these events are stored. This also means that parallelism can be achieved by splitting the input dataset in different groups of events and applying the computation graph to events in parallel. *RDataFrame* allows users to leverage all cores of their laptop with an implicit parallelism machinery based on Intel Threading Building Blocks (TBB) [37], but also to distribute the same workload to multiple machines transparently. The latter feature will be further discussed in the next section.

### 3.1.2 Distributed *RDataFrame*

One of the latest features introduced by the *ROOT* team for *RDataFrame* is the ability to distribute user applications to computing clusters without changing the analysis code. This is done via a thin Python layer that wraps the *RDataFrame* computation graph and sends it to the remote nodes leveraging popular execution engines like Spark and Dask. Such layer is called distributed *RDataFrame* or *DistRDF*. Listings 1 and 2 show how the user analysis can be run distributedly with minimal code changes.

```
import ROOT
if __name__ == "__main__":
    df = ROOT.RDataFrame("mydataset", "myfile.root")
    h = df.Define("x", "gRandom->Rndm()").Histo1D("x")
    h.Draw()
```

**Listing 1** A simple example with *RDataFrame*

```
from dask.distributed import LocalCluster, Client
import ROOT
# Point to the correct RDataFrame instance
# RDataFrame = ROOT...Distributed.RDataFrame
if __name__ == "__main__":
    # Configure access to remote resources.
    client = Client(LocalCluster(n_workers=4,
                                 threads_per_wroker=1,
                                 processes=true))

    # The keyword argument tells RDataFrame
    # to connect to the dask cluster.
    df = RDataFrame("mydataset", "myfile.root",
                    daskclient=client)

    # Continue working on the RDataFrame object as if
    # it was a local RDataFrame object.
    h = df.Define("x", "gRandom->Rndm()").Histo1D("x")
    h.Draw()
```

**Listing 2** A simple example with *DistRDF*

Workload distribution follows the MapReduce [17] scheme: a mapper task has to apply the full set of operations requested by the user on a certain range of entries of the dataset, a reducer task needs to combine the partial outputs of two mapper tasks. Once all the results have been combined into one final result, this is sent back from the computing cluster to the user. This saves a lot of time for the analyst, who does not need to take care of merging partial results of different mapper tasks on their own as it used to happen with the traditional batch computing approach.

A fundamental feature of the distributed RDataFrame tool is the automatic splitting of the input dataset in multiple logical partitions, so that each one can be processed independently in different tasks. This splitting is done on the client side, when the user asks for the results of their operations the first time. At this moment, the only information available is the list of files that the user wants to process. In general, the number of partitions can be either provided by the user or a default value is used. The default value is the total number of *cores* available in the cluster. As many "approximate tasks" are created as the number of partitions indicated, the tasks are approximate since the files are only accessed once a mapper task starts on a remote node, never by the client. This is done to avoid unnecessary remote I/O which is a notorious source of performance bottlenecks for a physics analysis.

*DistRDF* implements a few backends that have been used as a reference for the development of this work. The following are currently supported and featured in *ROOT* releases:

1. Apache Spark [50]. A framework whose goal is large-scale data processing using MapReduce. It can be mounted on top of existing Apache Hadoop clusters or on its own, and the main advantage over Hadoop is that intermediate data storage is done in memory instead of disk, providing much faster speed than Hadoop.
2. Dask [41]. A Python library for parallel computing, consisting of two main elements: a dynamic task scheduler optimized for interactive workloads and support

for *Big Data* data structures. In addition, it also offers capabilities for distributed computing in Python.

As already stated in Sect. 1, a previous study also developed a prototype implementation of a backend for *DistRDF* that makes use of *AWS Lambda*. The approach taken in that work was to invoke all the *Lambda* functions synchronously from the client application, using Python multithreading. This was done to be able to check the status of the invocations and retry their execution in case of error. Once all the functions had finished without errors, the reduction of the results was carried out locally on the user side.

This work provides various improvements with respect to the prototype just described. For example, the client side is completely decoupled from the rest of the architecture, avoiding the need for synchronous calls to functions. Furthermore, the reduction phase is completely carried out remotely in the computing cluster, thus avoiding possibly heavy network transactions sending the partial results from the remote nodes to the client. Although these improvements are developed for *OSCAR*, they could be implemented also in other *serverless* environments.

### 3.2 *OSCAR*

*OSCAR* focuses on file processing by supporting a *High Throughput Computing* model. The execution of the functions is done through the use of Docker containers. an *OSCAR* installation is based on a Kubernetes cluster that is deployed using the following tools: EC3 [14] to provide horizontal cluster elasticity; Infrastructure Manager, IM [15], to support multi-cloud deployment; and CLUES [2] to manage the elasticity of the cluster by taking care of the *scale in* and *scale out* of the nodes in the cluster, based on job demand. In Fig. 2 you can see how these services interact for the deployment and scaling of the cluster.

#### 3.2.1 Main components

The main components of *OSCAR* that affect the *backend* for *ROOT* are:

- *OSCAR Manager*. The core of *OSCAR*, in charge of managing the services and the interconnection with the rest of the components. It provides a REST API for service creation. This is characterized by stateless client–server communication through the HTTP protocol. A client with the necessary credentials may consult, create, modify or delete services via the API. *OSCAR* also provides other forms of interaction a web interface and a command line tool, named *OSCAR-CLI*.[1]
- *Serverless Backends*. *OSCAR* offers two possibilities for implementing the *FaaS* part of the Kubernetes cluster, namely, OpenFaaS [27] and Knative [45]. They support synchronous calls to services and invocation via calls to a REST API. For this work we will not use these functionalities and we will focus on the default functionality of uploading files that invoke the services asynchronously, thus generating Kubernetes jobs for the processing of files.

[1] https://github.com/grycap/oscar-cli.

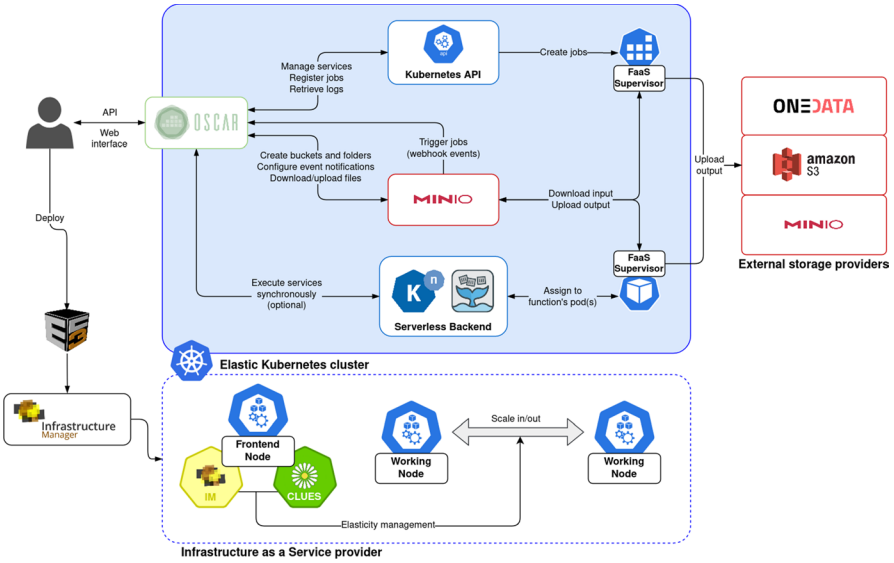


**Fig. 2** *OSCAR* architecture

- *Internal storage service*. The cluster uses an object storage system named *MinIO*. The operation of this type of system is based on the creation of *buckets*, or *repositories*, in which files of any type are uploaded. These files do not have directory hierarchies like traditional file systems. Often, prefixes are used to simulate a directory structure and to maintain a certain order within a *bucket*. The term directory or folder is used as a concept for practical purposes [4]. These *buckets* provide certain features such as access control, auditing or versioning, among others. Any modification of a bucket will be sent to *OSCAR* as an event. Depending on the configured rules, *OSCAR* will launch, or not, a service associated with that object. The results of the process can be stored directly within the service. Alternatively, they can be sent automatically to external providers such as *MinIO*, *Amazon S3* or *ONEDATA* [35].

### 3.2.2 Workflow

*OSCAR* 's basic units of work are functions, also called *services*. For each service it must be defined the required resources, i.e., CPU[2] and memory; the Docker image to use as the runtime environment; the script that will be executed inside the container as an entry point and, finally, which *bucket* should listen for the generation of events that are created when objects are uploaded, modified or deleted. For a given *bucket*

[2] CPU, in the context of resources allocated to a *OSCAR* service, refers to a logical CPU.

a prefix or suffix can be used to filter which objects will trigger the execution of the associated service. These prefixes or suffixes do not allow the option of using *wildcards* and must be specific.

When a service is defined, *OSCAR* automatically takes care of generating the corresponding *buckets* in *MinIO*, as well as configuring the types of events it will listen for. The results generated by the services can be stored back on the cluster's internal *MinIO* server, allowing multiple services to be chained together if desired. Special care must be taken not to generate recursive loops, since no tools are provided to detect them and, in the case of a public *cloud* deployment, it could generate considerable costs.

*OSCAR* focuses on a *file-driven* approach. When a user uploads a file, this generates an event that is picked up by *OSCAR*. *OSCAR* then takes care of launching a job for that file by generating a Kubernetes *job*, that is, launching a container based on the image provided by the client. In this *job* the file that has triggered the service will be injected in addition to executing a script that contains the operations to be carried out, also defined by the user.

## 4 Implementation of the *backend*

The implementation of the *backend* consists of two blocks. On the one hand, the interaction between *DistRDF* and *OSCAR* is defined. On the other hand, the implementation defines how the *backend* schedules resources, distributes the load and collects the final result. We begin with the part related to the *backend* (Sect. 4.1) to continue later with the description of *OSCAR* services (Sect. 4.2) necessary for the satisfactory execution of the analysis.

### 4.1 The ROOT *backend*

When an analysis is performed, all the actions defined in the *RDataFrame* are stored without being executed. When an operation that generates output is added, e.g., generating a histogram, *DistRDF* takes care of parsing all the code provided by the user and passing it to an internal function, called `ProcessAndMerge`.

The *backend* features the following three objects: Mapper:

Mapper: Function to be applied to each record in the specified dataset. It receives an object of type *range* as input and generates as output another object corresponding to the output generated by applying the *Mapper* function to that range, which will be referred to as a *partial result*.
Ranges: Is a list equal in size to the number of partitions specified for analysis. Each element of the list contains the information corresponding to the data to be processed, both the source of that data and the "approximate tasks" of each of the sources. These tasks are used to get the exact events to be processed.
Reducer: Is the aggregation function that will be used to reduce two by two the partial results generated by the *Mapper* in order to obtain the final result.

The implementation of the *backend* corresponding to the execution of the analysis must be carried out within function `ProcessAndMerge` since the value that this function must return is the result of adding all the partial results generated and it does not make sense to carry it out in another place. Somehow, all this information received by the `ProcessAndMerge` function has to be sent either to the *OSCAR* internal storage system or passed directly to the services so that not only one experiment can be run in parallel on multiple nodes but multiple users can be served concurrently by the cluster running *OSCAR*.

On this basis, for the integration with *OSCAR* an experiment can be carried out in a *bucket* or can be carried out in the folder[3] of an already existing *bucket* belonging to a specific user. The option in which a *bucket* represents an experiment has been chosen for simplicity, but the term *root directory* will be used as a generic concept to demonstrate that both implementations could be carried out, the root directory being, in the first case the name of the *bucket* itself and, in the second case, the root directory denoting a *bucket* folder. To identify an experiment, a universally unique identifier (*UUID*) is generated during the creation of the *RDataFrame* abd then used as the name of the root directory.

During preliminary tests performed in this evaluation, no performance difference was found between using a single *bucket* with a folder hierarchy and using multiple *buckets*. The difference relates to configuration issues, certain properties and permissions. Therefore, the final decision on the configuration should be relegated to the requirements of the specific cluster deployment.

In order to send these objects from the client to a remote host, they all need to be serialized. We used *cloudpickle*,[4] a library which extends the basic functionality of the standard Python serialization library (`pickle`), which allows to send defined functions during the *backend* execution, something that the `pickle` tool by itself does not support. Therefore, both the *Mapper* and the *Reducer* functions are serialized, and they all are written to a folder in the root directory of the Object Storage System (OSS) called *functions*. None of these scripts launch any service. With this information already in the cluster, the *backend* of *ROOT* only has to define a way to invoke the necessary services and wait for the result.

Once invoked these services, the function remains blocked and waiting for the final result. The final result can be written either to an external provider such as *MinIO*, *Amazon S3*, or *ONEDATA*, or to the cluster's own internal system. In our implementation, we have used the *OSCAR* internal system, i.e., *MinIO*, to store the final result. *MinIO* offers a function that allows to receive notifications of certain events, e.g, writing or modifying files in a specific location when they are generated. With this functionality, the client will stay on hold, avoiding a constant polling of the system.

[3] Object Storage Systems lack of a hierarchy of directories but, for the rest of the document, the term *folder* will be used for simplicity.

[4] https://github.com/cloudpipe/cloudpickle.

### 4.2 *OSCAR* services

A minimum of two *OSCAR* services need to be created, a *Mapper* and a *Reducer*. Additionally, we may need a coordination service depending on the way the reduction is performed. At its current state, *OSCAR* does not support usage of wildcards in the path specified to *MinIO* when creating a service. Thus, supporting common use cases such as multiple users running their analysis or a single user running more than one application requires two extra steps. First, the exact path where the service will be listening for events must be specified during its creation. Second, each service needs to be destroyed, together with the data it exchanged with the OSS, once the analysis has completed successfully.

#### 4.2.1 The *Mapper* service

For the *Mapper* we define a service that listens file-creation events in the *mapper-jobs* folder of the root directory. The *backend* writes in the OSS as many files as enumerated in the *Ranges* list. Each one of those files will have the information, serialized, of the particular *range*, launching thus the execution of as many concurrent *Mapper* services as specified at the initialization of the *RDataFrame*. Invoking a *Mapper* service consists of reading from the OSS the *Mapper* function. The *backend* deserializes the function, turns it into an object and applies it to its assigned *range* of entries. The access to *MinIO* storage system is protected with credentials so, to obtain the *Mapper* function, it is necessary to have access to these credentials and the path to the file that contains the *Mapper* function.

When a *Mapper* has finished processing the analysis part, it writes the result returned by the *Mapper* function in a temporary file of the container, following the philosophy of *OSCAR*, so that it is in charge of managing the upload of those results to *MinIO*. *OSCAR* will write this result to the *partial-results* folder, which will trigger the reduction process. This event will or will not directly launch another service, depending on the reduction model used.

#### 4.2.2 The *Reducer* service

The reduction phase is much faster than the map phase, since it simply consists of merging partial results and there are no complex computations involved. Nonetheless, performing the reduction in a stateless scenario is not a trivial problem. Compared to a traditional MapReduce workflow, there is no central scheduler that can decide when the reduction phase should start and how the reducers should be run. Thus, this work proposes two ways of performing this phase:

- *Uncoordinated Reduction*. Under this model of coordination, the result written by each *Mapper* will launch a *Reducer*. The problem of a reduction in a *serverless* environment without coordination or uncoordinated lies on how to check the state of the operation. The *mapping* process will generate a given set of partial results. If we set an order in the *Mapper* services invoked assigning an identifier to each one of them, lets say, the rank in the list of *range*, limiting the reduction

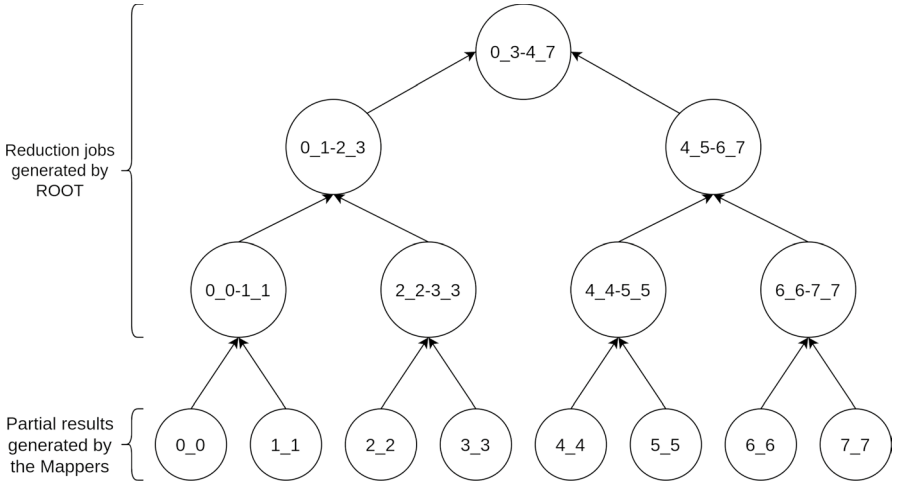


**Fig. 3** Algorithm of reduction

to be done in a given order, the reduction can be carried out without the presence of a coordinator "manager". Each partial result needs to know which other partial result it should be reduced with. This additional information, necessary for this solution to work, can be found in the *reducer-jobs* folder. This folder contains as many files as reduction jobs. The files names are selected according to the reduction process. For instance, let there be two *Mappers* with identifiers `1_1` and `2_2`, respectively, then the name of the fale representing their reduction is `1_1-2_2`. A *Mapper*, upon completion, performs the reduction if the data of its counterpart is available in the file system. Otherwise, simply stores the partial result, that will wait for the other mapper to finish. It is necessary to find a way to determine the generation of the *Mapper* names in a deterministic way so that whatever be the number of *Mappers* invoked, the reduction is carried out successfully. The algorithm proposed to solve this problem focuses on generating a binary tree from the leaves to the root, in which the leaves are the identifiers of the *Mappers*, and at each level from bottom to top the names are combined two by two until reaching the root node. The name combination is done by taking the identifiers of the extremes that will match the minimum and maximum identifiers that have been reduced up to that point. For their part, the *Reducer* must write in *partial-results* the result with the name corresponding to the reduction of the two reduced jobs so that the new *Reducer* that is invoked is able of reducing itself with another partial result. This generation of reduced jobs is built into the *backend* of *ROOT* but has been developed in this section for convenience. This idea can be easily generalized to non binary trees. We show a reduction example in Fig. 3 assuming a division of the analysis load into 8 parts. In these trees, the nodes of the deepest level will be discarded since they correspond to the *Mappers* and will write the name to the corresponding folder. The main advantage of this approach is that we can get rid of a manager process or a coordination function that should permanently be monitoring the process, with

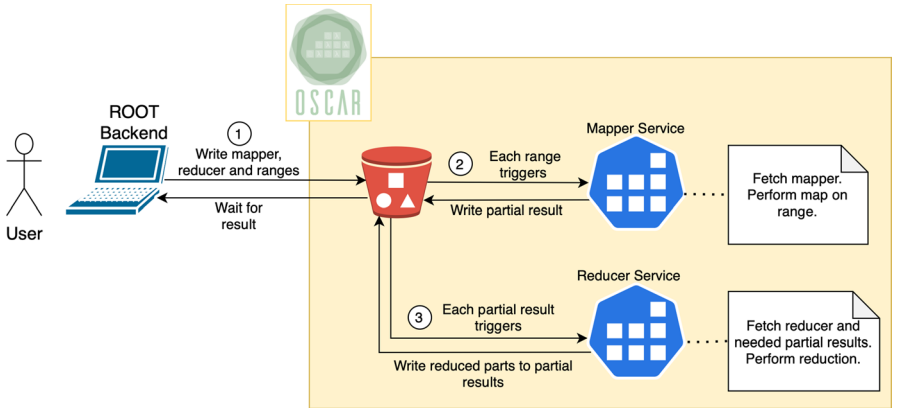


**Fig. 4** Component interaction in the uncoordinated reduction process. Numbers denote events order

the consequent waste of resources, which is very discouraged on platforms, such as *AWS Lambda*. On the contrary, there exists a disadvantage. If two *Mappers* that are to be reduced together terminate and write their partial result nearly at the same time, they both will check that their counterpart already wrote a partial result in their corresponding file. In consequence, both *Mappers* will probably perform the same reduction at the same time. This fact, although produces a correct final result, also produces in turn a performance penalty. Figure 4 shows a schema of the uncoordinated reduction process.

- *Coordinated reduction*: Another alternative consists of using an additional service, i.e., a coordination process which is in charge of the whole reduction process. The *backend* invokes this service once, at the starting. This service monitors the *Mappers* once they are created and the folder where they write the partial results (*partial-results*). The monitoring process is done similarly as the client does when waits for the final result, receiving an event by the time a partial result is written in the file. The coordination service is responsible of launching reduction jobs when necessary and of writing in the folder *reducer-jobs* the work to be done. On the one hand, the *backend* will generate a file with the necessary configuration of the coordinating service. This configuration consists of a list of integers where each value represents the number of partial results that should be merged once a *Reducer* is invoked. The coordinator watches the folder in which the partial results are written and, upon notification of a new result, it will save the name of the given file. Once the number of partial results corresponding to an element of the configuration list is reached, the coordinator will generate a reduction job including the names of all the partial results that should be merged. Once done, it will pass to the next element in the reduction list. An optimization carried out during this work is that the last reduction job is performed directly by the coordinator service, so that no extra time is spent waiting. It should be noted that since the *Reducer* function only accepts two input parameters, these "multiple" reductions are performed by iterating over the jobs to be reduced, but they avoid invoking multiple *Reducers* and some of the limitations of the uncoordinated reduction approach. Writing the jobs

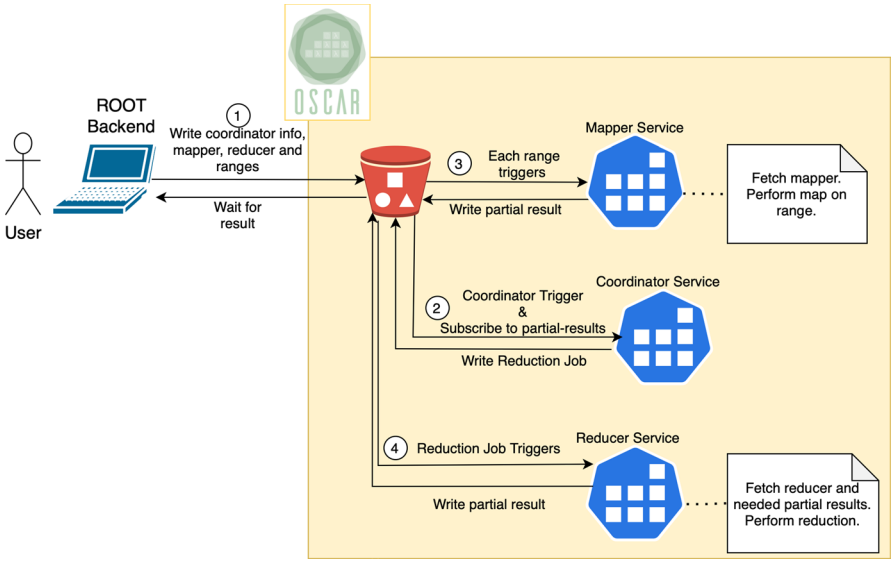


**Fig. 5** Component interaction in the coordinated reduction process. Numbers denote events order

is done by the coordinator directly to *MinIO* without interference from the normal *OSCAR* workflow. In fact, usually *OSCAR* only uploads files once a service has terminated. But the coordinator service should not finish until all the MapReduce workflow has ended. In this case the *Reducer* will be called when the files are written to the *reducer-jobs* folder instead of *partial-results* folder, as it was in the uncoordinated case. The reduction service needs an additional modification. There is no longer need to check whether there are other partial results pending. There is only need to deserialize the content of the job, get the partial results from *MinIO* and reduce them all. Once it has finished reducing them, it will write that partial result to a file with the result that *OSCAR* will upload to *MinIO*, finally triggering a notification to the coordinator. With the coordinated reduction is not necessary to keep the order of the file names. Indeed, the coordinator can structure the process of reduction as convenient, e.g., into two parts or choosing an imbalanced splitting (80–20%). Figure 5 shows a schema of the coordinated reduction process.

### 4.3 Summary of the *backend*

Preliminary attempts of integrating the *backend* with *OSCAR* tried using the *OSCAR* API to call the *Mapper* and *Reducer* services. The final implementation instead accesses directly the files on *MinIO* to generate events. This choice conveniently provides the possibility to resume the execution at any point of failure during the MapReduce workflow.

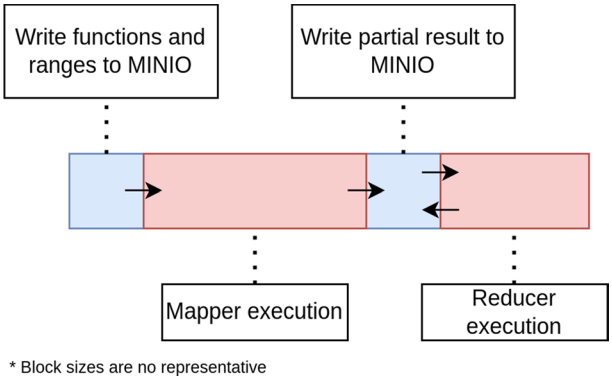


**Fig. 6** Stages of execution experiments

Another difference between the two models is that in the uncoordinated reduction the jobs are written before the *Mappers* start whilst in the coordinated one the jobs are written by the coordinator when necessary.

One of the current requirements of this *backend* is the need to define the number of *Mappers* to be invoked for the analysis. In other *backends* it is possible to obtain the number of *cores* of the cluster but, in *OSCAR*, this information is not available at the time the services are created. In fact, the whole serverless approach relies on the idea that users can scale the number of function calls automatically depending on the amount of work provided. A possibility for future research would be to attempt at predicting the best number of functions that should be invoked, based on the workload given by the user application.

## 5 Experiments

One of the most important performance metrics for a HEP analysis is the so-called *time-to-plot*. That is, the time elapsed between the moment a user starts the analysis and the moment they can see a plot of the results on their screen. This metric determines the real scalability of the implementation of the proposed backends.

The experiments that will be described in this section all have to go through the usual two stages of the MapReduce scheme, plus some extra scheduling and orchestration that is more specific to *OSCAR*. Figure 6 shows the stages of the process where the blue squares represent the work being done outside of the physics analysis, such as container start and kill, *MinIO* iteration and others. The red squares correspond to the time devoted to the analysis itself. In order to obtain this information the code of the *OSCAR* services is wrapped with a Python script that invokes the mapper or reducer functions and monitors the hardware resources and time spent executing.

**Table 1** Hardware used for the experiments

| | |
|---|---|
| 6 nodes | CPU:16 cores Skylake Architecture. No hyper threading. Each core has 64KB of L1 cache, 4 MB of L2 cache and 16 MB of L3 cache<br>Memory: 62.5 GB |
| Cluster private network | 10GbE |
| Cluster public network | 10GbE |
| *MinIO* | The object storage system is hosted on a dedicated disc with the following performance: 3500MB/s sequential read and 3000 MB/s sequential write |

### 5.1 Setup

A total of 6 working nodes were made available for the purposes of this work. Each working node is a virtual machine hosted on a *OSCAR*-enabled computing cluster, located in Valencia (Spain). The underlying hardware is described in Table 1. Each *OSCAR* service created, i.e., the mapper and the reducer, is configured to invoke functions with 1 vCPU and 3 GB of RAM available.

### 5.2 Methodology

Two types of analysis were employed as benchmarks in this work. The first type is a real physics analysis, processing data from events recorded by the CMS experiment in 2012 to finally plot a histogram of the di-muon mass spectrum [49]. The amount of data for this experiment consists of 200GB obtained from replicating one hundred times the original dataset file of 2GB. This analysis was carried out with two types of benchmarks, differing for the location of the dataset at runtime: either stored in the CERN data center or in the *MinIO* storage attached to the *OSCAR* cluster, much closer to the computing nodes.

The second type of analysis consists of a simulated workload that reads no data from disk or network and can better highlight the best CPU usage the backend can drive. A simulated dataset is created at runtime in-memory, with roughly the same amount of data that would be processed in the other type of analysis. The *Mapper* of this experiment will generate the required data locally and will apply a series of operations over the data.

For each of the three experiments, an increasing number of splits of the input dataset (also called partitions) was studied: 1, 2, 4, 8, 16, 32, 48, 64, 80. Each split corresponds to one mapper task, thus one invocation of the corresponding *OSCAR* service, which are then all executed simultaneously on the available nodes. The overall number of cores in the cluster would be 96, but in each node the Kubernetes service consumes a small amount of CPU thus hindering performance when all cores of a single VM are used. For this reason, the 96 partition count is omitted

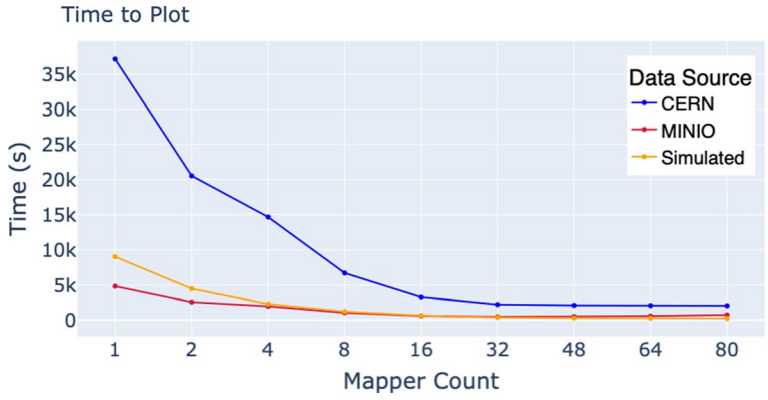


**Fig. 7** *Time-To-Plot* evolution

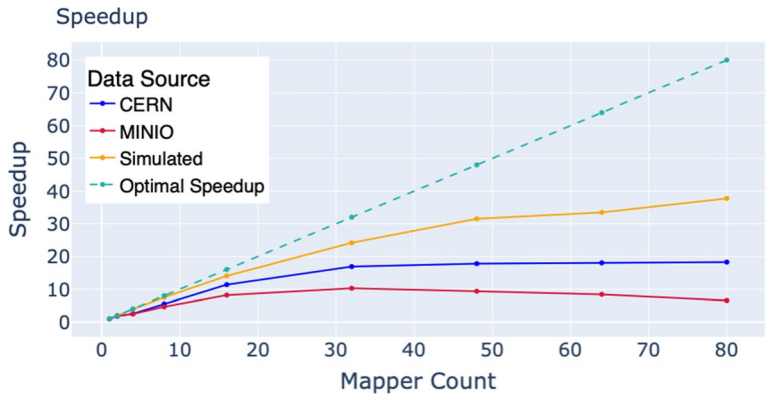


**Fig. 8** *Speedup* evolution

The Docker images are downloaded to the cluster before the experiments are run to avoid fetching the 4GB sized images during the first experiment.

### 5.3 Results

In Fig. 7 the *time-to-plot* for all the experiments can be observed. As expected, the experiment with the data located within the cluster is faster than the experiment with the data located at CERN's servers.

In Fig. 8, the speedup for all the experiments is shown alongside the optimal speedup that should be expected from this kind of analysis, as all the data to be processed are independent. For the experiments that use real data we can observe that CERN outperforms the data stored in *MinIO*, something that should not happen unless hitting some bottleneck with the disk where *MinIO* is operating. This assumption is based on the fact that *MinIO* storage is placed within the

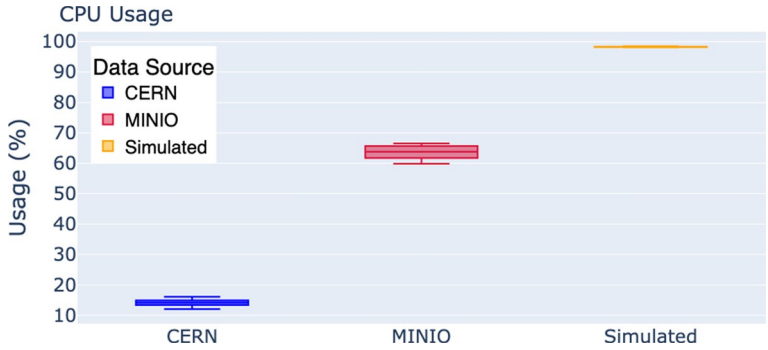


**Fig. 9** *Mapper* CPU usage

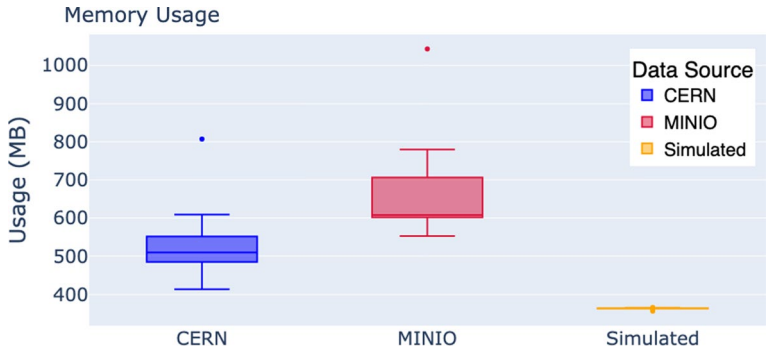


**Fig. 10** *Mapper* memory usage

same network and the same machines that run the *OSCAR* services. Thus, latency should be minimal and I/O throughput should be higher than reading remotely from CERN, due to the geographical distance.

For the CERN experiment, when we use 48 *Mappers* or more, the speedup remains constant, being probably a network bottleneck. Similarly, also the simulated workload shows a poor performance scaling, reaching less than half of the optimal speedup as the core count increases.

Regarding the CPU and Memory utilization, Fig. 9 shows the expected behavior. The simulated workload has a CPU usage close to 100% and the experiment with the data stored in *MinIO* also makes a better usage of the CPU as it has to spend less time waiting for the data to arrive due to its locality. Looking at Fig. 10 we can see that none of the experiments is close to reaching the 3GiB memory limit set, so we can discard this limit as a reason for the poor performance.

In order to better investigate the lack of scaling, the focus is shifted to the simulated workload. The other applications that read data may be influenced by I/O with the network or the local filesystem, so it is preferable to have a fully CPU bound workload to avoid any noise from other factors. Table 2 describes the difference in time between the fastest *Mapper* and the slowest one for the simulated workload.

**Table 2** *Mapper*'s variability in execution time for simulated workload

| Mapper count | Absolute time difference (s) | Relative difference time (%) |
|---|---|---|
| 1 | 0.0 | 0.0 |
| 2 | 119.0 | 2.79 |
| 4 | 154.0 | 7.41 |
| 8 | 105.5 | 10.03 |
| 16 | 63.5 | 11.72 |
| 32 | 40.5 | 13.64 |
| 48 | 32.5 | 15.44 |
| 64 | 48.0 | 29.45 |
| 80 | 39.0 | 27.76 |

The absolute time difference represents the difference between the runtime of the slowest mapper and the runtime of the fastest mapper. The relative time difference is computed with respect to the runtime of the fastest mapper

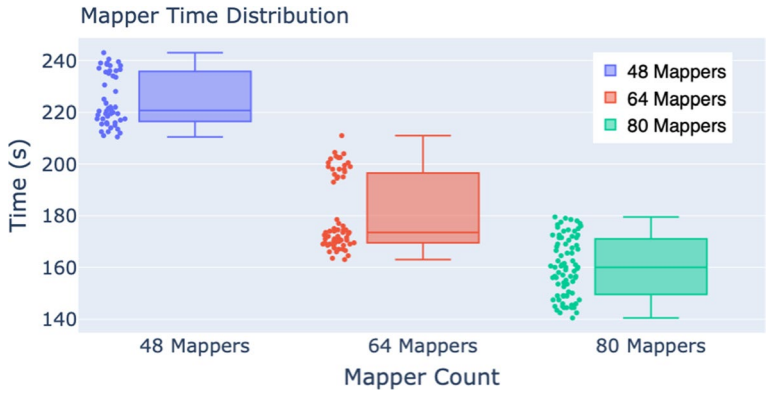


**Fig. 11** Distribution of the runtime of *Mapper* invokations for 48, 64 and 80 mappers, respectively, from left to right in the image

In this workload the difference should be within a reasonable margin (e.g., 2–3%), which may be justifiable due to noise on the system, but we can see that the difference in time reaches up to 29.45%. Figure 11 shows this information graphically. The same workload was run with distributed RDataFrame outside of the *OSCAR* cluster, on a single physical machine. The difference in time between the fastest and the slowest *Mapper* in this controlled condition is on average 3%. With this information we can conclude that there are unknown sources of bottlenecks coming from the *OSCAR* cluster.

**Table 3** Time to plot results for different workload partitioning of the coordinated reduction strategy

| Workload partitioning (%) | Time to plot [s] |
|---|---|
| 0 | 205 |
| 50 | 208 |
| 87.5 | 202 |

### 5.4 Comparison between reduction strategies

The previous results were relative to the uncoordinated reduction strategy. Section 4 introduced another possible way of addressing the reduction phase, which is further discussed in this section and compared to the first one.

In the uncoordinated reduction the degree of the tree is fixed to two. The coordinated reduction model is more flexible. Taking as a starting point the overhead introduced by *OSCAR*, we will focus on a two step reduction. This means that the coordinator will launch a reduction service to merge a determined amount of partial results, while the remaining partial results will be merged by the coordinator itself. For this coordinated reduction three different configurations of load partitioning have been studied:

- 0%: all the reductions are performed by the coordinator.
- 50%: the invoked reducers process 50% of the reductions and the remaining reductions are performed by the coordinator
- 87.5%: the reducers are invoked for 87.5% of cases, the coordinator performs the remaining 12.5%.

Table 3 shows the results for this fine-tuning, all the configurations have similar results and the differences can be attributed to the runtime variability of the various mappers. For the comparison with the uncoordinated reduction the 87.5% will be used.

For this fine-tuning experiment the tests have been carried out using 80 *Mapper* services as it generates the biggest amount of reductions required. Figure 12 shows that, despite the variability of the system, the coordinated reduction is faster than the binary reduction, and this difference increases with the amount of *Mappers*. This is due to the fact that for the uncoordinated reduction, the last *Mapper* service to finish has to travel the entire path to the root of the tree invoking a service on each step and the greater the amount of *Mappers* the deeper the tree. This could be partially solved if instead of performing only one reduction in the uncoordinated reduction, the *Reducer* also checks if it can perform any other reduction in the path to the root.

## 6 Conclusion

In this work we have extended the functionality of *ROOT* with another *serverless* engine that improves some of the shortcomings of the previous one based on *AWS Lambda*. The open-source platform provided by *OSCAR* is a solid basis for a new

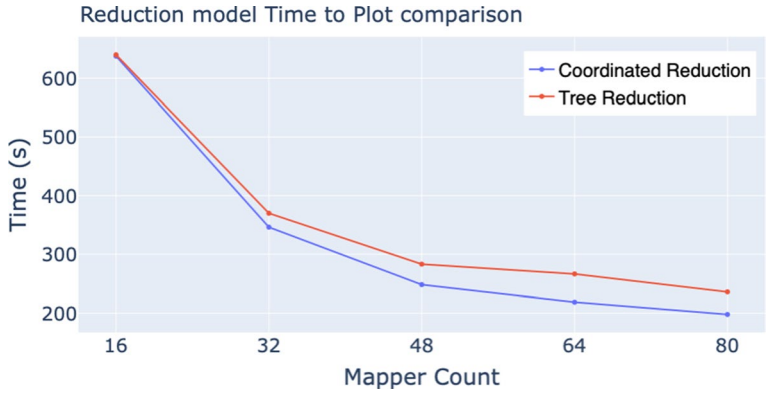


**Fig. 12** Reduction time comparison

*DistRDF* backend that could be used by physicists to easily deploy their distributed environments. A few areas of improvement were found for *OSCAR*, in particular regarding the asynchronous function invocations, which rely on the underlying Kubernetes batch job scheduler. The experimental results still favor the current distributed backends offered via Spark and Dask. Nonetheless, the *serverless* approach is one of the best options to afford the necessary scalability requested by future HEP computing needs. In that regard, this work paves the road to addressing this challenge.

**Author contributions** The following statement is based on CRediT taxonomy, giving contributions of each author in the work: VEP: Conceptualization, Investigation, Writing—review and editing, Software, Methodology. POC: Conceptualization, Investigation, Writing—original draft, Software, Methodology. PA-J: Conceptualization, Supervision, Writing—original draft, Validation. ETS: Conceptualization, Supervision. SR: Conceptualization, Software, Methodology, Writing—review and editing. GM: Conceptualization, Software, Methodology, Writing—review and editing

**Funding** Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature. This work was supported by the research projects PID2020-113656RB-C22 (MCIN/AEI/10.13039/501100011033). Also, Grant PID2020-113126RB-I00 funded by MCIN/AEI/10.13039/501100011033 and Project PDC2021-120844-I00 funded by MCIN/AEI/10.13039/501100011033 funded by the European Union NextGenerationEU/PRTR all support the research on the *OSCAR* software tool.

**Data availability** The code and Docker images of this work are available at https://github.com/Break95/TFM-ServerlessRoot. The data used for the benchmarks in Sect. 5 is open and available at the CERN OpenData portal [49].

## Declarations

**Conflict of interest** Financial interests are disclosed in Funding. The authors declare no other competing interest for this work.

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

**Ethics approval** Not applicable.

## References

1. Albrecht J, Alves AA, Amadio G et al (2019) A roadmap for HEP software and computing R &D for the 2020s. Comput Softw Big Sci 3(1):7. https://doi.org/10.1007/s41781-018-0018-8
2. Alvarruiz F, de Alfonso C, Caballer M, et al (2012) An energy manager for high performance computer clusters. In: 2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications, p 231–238. https://doi.org/10.1109/ISPA.2012.38
3. Amazon Web Services (2022a) Lambda. https://aws.amazon.com/releasenotes/release-aws-lambda-on-2014-11-13. Accessed 4 Dec 2022
4. Amazon Web Services (2022b) Organizing objects in the Amazon S3 console using folders. https://docs.aws.amazon.com/AmazonS3/latest/userguide/using-folders.html. Accessed 4 Dec 2022
5. Amazon Web Services (2022c) S3: Simple Storage Service. https://aws.amazon.com/s3. Accessed 4 Dec 2022
6. Apache Software Foundation (2022) OpenWhisk. https://openwhisk.apache.org/. Accessed 4 Dec 2022
7. Apollinari G, Béjar Alonso I, Brüning O et al (2017) High-luminosity large hadron collider (HL-LHC): technical design report V.0.1. Tech Rep CERN. https://doi.org/10.23731/CYRM-2017-004
8. Beswick J (2022) Using Amazon EFS for AWS Lambda in your serverless applications. https://aws.amazon.com/blogs/compute/using-amazon-efs-for-aws-lambda-in-your-serverless-applications/. Accessed 4 Dec 2022
9. Bila N, Dettori P, Kanso A, et al (2017) Leveraging the serverless architecture for securing linux containers. In: 2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW), p 401–404. https://doi.org/10.1109/ICDCSW.2017.66
10. Bird I, Buncic P, Carminati F, et al (2014) Update of the computing models of the WLCG and the LHC experiments. Tech Rep CERN. https://cds.cern.ch/record/1695401
11. Blomer J, Buncic P, Fuhrmann T (2011) CernVM-FS: delivering scientific software to globally distributed computing resources. In: Proceedings of the First International Workshop on Network-aware Data Management. Association for Computing Machinery, New York, p 49-56. https://doi.org/10.1145/2110217.2110225
12. Blomer J, Ganis G, Mosciatti S et al (2019) Towards a serverless CernVM-FS. EPJ Web Conf 214(09):007. https://doi.org/10.1051/epjconf/201921409007
13. Brun R, Rademakers F (1997) ROOT-an object oriented data analysis framework. Nuclear instruments and methods in physics research section A: accelerators, spectrometers, detectors and associated equipment. New Comput Tech Phys Res V 389(1):81–86. https://doi.org/10.1016/S0168-9002(97)00048-X
14. Caballer M, de Alfonso C, Alvarruiz F et al (2013) EC3: elastic cloud computing cluster. J Comput Syst Sci 79(8):1341–1351. https://doi.org/10.1016/j.jcss.2013.06.005
15. Caballer M, Blanquer I, Moltó G et al (2015) Dynamic management of virtual infrastructures. J Grid Comput 13(1):53–70. https://doi.org/10.1007/s10723-014-9296-5
16. Carver B, Zhang J, Wang A, et al (2020) Wukong: a scalable and locality-enhanced framework for serverless parallel computing. In: Proceedings of the 11th ACM Symposium on Cloud Computing. Association for Computing Machinery, New York, p 1–15. https://doi.org/10.1145/3419111.3421286
17. Dean J, Ghemawat S (2004) MapReduce: simplified data processing on large clusters. In: OSDI'04: Sixth Symposium on Operating System Design and Implementation. San Francisco, CA, p 137–150
18. Dorigo A, Elmer P, Furano F et al (2005) XROOTD—a highly scalable architecture for data access. WSEAS Trans Comput 4:348–353

19. Giménez-Alventosa V, Moltó G, Caballer M (2019) A framework and a performance assessment for serverless MapReduce on AWS Lambda. Future Gener Comput Syst 97:259–274. https://doi.org/10.1016/j.future.2019.02.057
20. Google (2022) Cloud Functions. https://cloud.google.com/functions. Accessed 4 Dec 2022
21. Grzesik P, Augustyn DR, Wyciślik L et al (2021) Serverless computing in omics data analysis and integration. Brief Bioinform. https://doi.org/10.1093/bib/bbab349
22. Harris CR, Millman KJ, van der Walt SJ et al (2020) Array programming with NumPy. Nature 585(7825):357–362. https://doi.org/10.1038/s41586-020-2649-2
23. HEPix (2017) Hepix benchmarking working group. https://w3.hepix.org/benchmarking.html. Accessed 4 Dec 2022
24. Jonas E, Pu Q, Venkataraman S, et al (2017) Occupy the cloud: distributed computing for the 99%. In: Proceedings of the 2017 Symposium on Cloud Computing. Association for Computing Machinery, New York, p 445-451. https://doi.org/10.1145/3127479.3128601
25. Kuśnierz J, Padulano VE, Malawski M, et al (2022) A serverless engine for high energy physics distributed analysis. In: 2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid), p 575–584. https://doi.org/10.1109/CCGrid54584.2022.00067
26. Lavrijsen WTLP, Dutta A (2016) High-performance python-C++ bindings with PyPy and Cling. In: PyHPC '16. IEEE Press, p 27-35. http://wlav.web.cern.ch/wlav/Cppyy_LavrijsenDutta_PyHPC16.pdf
27. Le DN, Pal S, Pattnaik PK (2022) OpenFaaS. John Wiley & Sons, p 287–303. https://doi.org/10.1002/9781119682318.ch17
28. Li Z, Guo L, Chen Q, et al (2022) Help rather than recycle: alleviating cold startup in serverless computing through inter-function container sharing. In: 2022 USENIX Annual Technical Conference (USENIX ATC 22). USENIX Association, Carlsbad, p 69–84. https://www.usenix.org/conference/atc22/presentation/li-zijun-help
29. McKinney W (2010) Data structures for statistical computing in python. In: Stéfan van der Walt, Jarrod Millman (eds) Proceedings of the 9th Python in Science Conference, p 56–61. https://doi.org/10.25080/Majora-92bf1922-00a
30. Merkel D (2014) Docker: lightweight linux containers for consistent development and deployment. Linux J 2014(239):2
31. MinIO (2022) White paper: high performance multi-cloud object storage. Tech Rep MinIO Inc., Palo Alto, CA. https://min.io/resources/docs/MinIO-High-Performance-Multi-Cloud-Object-Storage.pdf
32. Müller I, Marroquín R, Alonso G (2020) Lambada: interactive data analytics on cold data using serverless cloud infrastructure. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. Association for Computing Machinery, New York, p 115–130. https://doi.org/10.1145/3318464.3389758
33. Nguyen HD, Yang Z, Chien AA (2021) Motivating high performance serverless workloads. In: Proceedings of the 1st Workshop on High Performance Serverless Computing. Association for Computing Machinery, New York, p 25–32. https://doi.org/10.1145/3452413.3464786
34. Oakes E, Yang L, Zhou D, et al (2018) SOCK: rapid task provisioning with serverless-optimized containers. In: 2018 USENIX Annual Technical Conference (USENIX ATC 18). USENIX Association, Boston, p 57–70. https://www.usenix.org/conference/atc18/presentation/oakes
35. ONEDATA (2022) https://onedata.org. Accessed 4 Dec 2022
36. Padulano VE, Villanueva JC, Guiraud E et al (2020) Distributed data analysis with ROOT RDataFrame. EPJ Web Conf 245(03):009. https://doi.org/10.1051/epjconf/202024503009
37. Pheatt C (2008) Intel®threading building blocks. J Comput Sci Coll 23(4):298
38. Piparo D, Canal P, Guiraud E et al (2019) RDataFrame: easy parallel ROOT analysis at 100 threads. EPJ Web Conf 214(06):029. https://doi.org/10.1051/epjconf/201921406029
39. Pérez A, Moltó G, Caballer M et al (2018) Serverless computing for container-based architectures. Future Gener Comput Syst 83:50–59. https://doi.org/10.1016/j.future.2018.01.022
40. Pérez A, Risco S, Naranjo DM, et al (2019) On-premises serverless computing for event-driven data processing applications. In: 2019 IEEE 12th International Conference on Cloud Computing (CLOUD). https://doi.org/10.1109/CLOUD.2019.00073
41. Rocklin M (2015) Dask: parallel computation with blocked algorithms and task scheduling. In: Huff K, Bergstra J (eds) Proceedings of the 14th Python in Science Conference. SciPy, online, p 130–136
42. Serguei C et al (2008) The CMS experiment at the CERN LHC. JINST 3(S08):004. https://doi.org/10.1088/1748-0221/3/08/S08004

43. Sexton-Kennedy E (2018) HEP software éevelopment in the next decade; the views of the HSF community. J Phys Conf Series 1085(022):006. https://doi.org/10.1088/1742-6596/1085/2/022006
44. Shankar V, Krauth K, Vodrahalli K, et al (2020) Serverless linear algebra. In: Proceedings of the 11th ACM Symposium on Cloud Computing. Association for Computing Machinery, New York, p 281–295. https://doi.org/10.1145/3419111.3421287
45. The Knative Authors (2022) Knative. https://knative.dev. Accessed 4 Dec 2022
46. The Kubernetes Authors (2022) Kubernetes. https://kubernetes.io/. Accessed 4 Dec 2022
47. Vassilev V, Canal P, Naumann A et al (2012) Cling–the new interactive interpreter for ROOT 6. J Phys Conf Series. https://doi.org/10.1088/1742-6596/396/5/052071
48. WLCG (2022) Homepage. http://wlcg.web.cern.ch/. Accessed 4 Dec 2022
49. Wunsch S (2019) Analysis of the di-muon spectrum using data from the CMS detector taken in 2012. https://doi.org/10.7483/OPENDATA.CMS.AAR1.4NZQ
50. Zaharia M, Chowdhury M, Franklin MJ, et al (2010) Spark: cluster computing with working sets. In: Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing. USENIX Association, Boston, p 10. https://www.usenix.org/conference/hotcloud-10/spark-cluster-computing-working-sets



## Authors and Affiliations

**Vincenzo Eduardo Padulano[1,2] · Pablo Oliver Cortés[1] · Pedro Alonso-Jordá[1] · Enric Tejedor Saavedra[2] · Sebastián Risco[3] · Germán Moltó[3]**

Pablo Oliver Cortés
olivercortespablo@gmail.com

Pedro Alonso-Jordá
palonso@upv.es

Enric Tejedor Saavedra
enric.tejedor.saavedra@cern.ch

Sebastián Risco
srisco@i3m.upv.es

Germán Moltó
gmolto@dsic.upv.es

[1] Department of Computer Systems and Computation, Universitat Politècnica de València, Camino de Vera s/n, 46022 Valencia, Spain

[2] EP-SFT, CERN, 1211 Meyrin, Geneva, Switzerland

[3] Instituto de Instrumentación para Imagen Molecular (I3M). Centro mixto CSIC, Universitat Politècnica de València, Camino de Vera s/n, 46022 Valencia, Spain