Daisy: Data analysis integrated software system for X-ray experiments

Yu Hu[1], *Ling Li*[1,2], *Haolai Tian*[1,2,3*], *Zhibing Liu*[1,2], *Qiulan Huang*[1,2], *Yi Zhang*[1], *Hao Hu*[1], *Fazhi Qi*[1,2,#]

[1]Computing Center, Institute of High Energy Physics, Beijing, 100049, China.
[2]University of Chinese Academy of Sciences, Beijing 100049, China
[3]Spallation Neutron Source Science Center, Dongguan 523803, China

Abstract. Daisy (Data Analysis Integrated Software System) has been designed for the analysis and visualisation of X-ray experiments. To address the requirements of the Chinese radiation facilities community, spanning an extensive range from purely algorithmic problems to scientific computing infrastructure, Daisy sets up a cloud-native platform to support on-site data analysis services with fast feedback and interaction. Furthermore, the plug-in based application is convenient to process the expected high throughput data flow in parallel at next-generation facilities such as the High Energy Photon Source (HEPS). The objectives, functionality and architecture of Daisy are described in this article.

1 Introduction

Large scale research facilities are becoming prevalent in the modern scientific landscape. One of these facilities' primary responsibilities is to make sure that users can process and analyse measurement data for publication. In order to allow for barrier-less access to those highly complex experiments, almost all beamlines require fast feedback capable of manipulating and visualising data online to offer convenience for the decision process of the experimental strategy. And recently, the advent of beamlines at fourth-generation synchrotron sources and high resolution with high sample rate detector has made significant progress that pushes the demand for computing resources to the edge of current workstation capabilities. On top of this, most synchrotron light sources have shifted to prolonged remote operation because of the outbreak of a global pandemic, with the need for remote access to the online instrumental system during the operation. Another issue is the vast data volume produced by specific experiments makes it difficult for users to create local data copies. In this case, as shown in Fig. 1, on-site data analysis services are necessary both during and after experiments.

* Corresponding author: tianhl@ihep.ac.cn
# Corresponding author: qfz@ihep.ac.cn

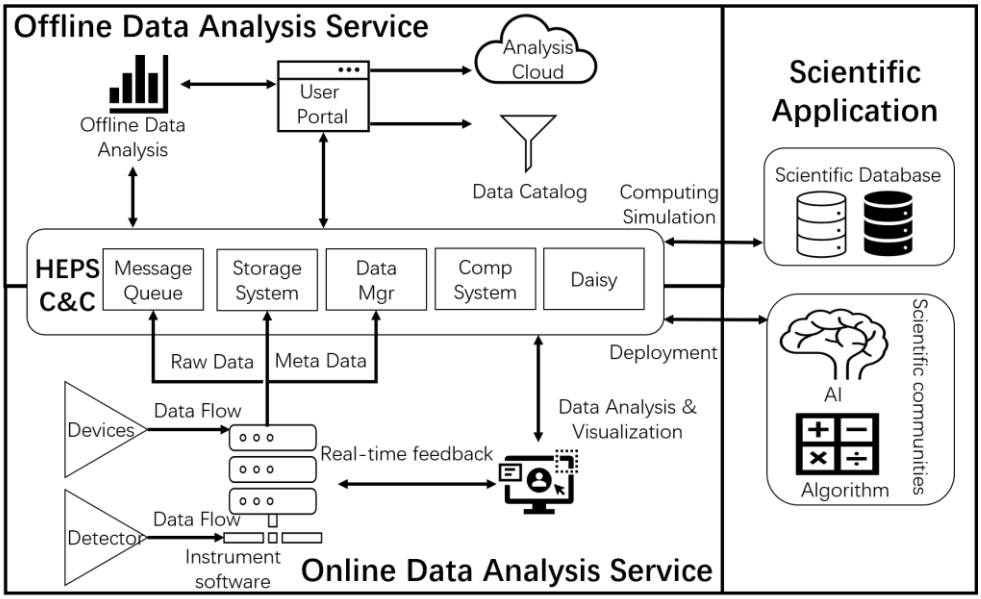

Fig. 1. The integrated computing platform of the HEPS enable on-site data analysis services both for online and offline

The High Energy Photon Source (HEPS) [1] is the first high-performance fourth-generation synchrotron radiation light source with a diffraction-limited storage ring in China. It creates opportunities for researchers to perform multi-dimensional in-situ characterisation of complex structures and dynamic processes with a high beam energy of 6 GeV and ultra-low emittance of 0.06 nm*rad. Some state-of-the-art experimental techniques, such as phase-contrast tomography and ptychography approaches, will be deployed. However, customised algorithms developed in the scientific communities are needed to use these techniques to the full extent.

However, it poses a critical problem of integrating this algorithmic development into a novel computing environment used in the experimental workflow [2]. The solution requires collaboration with the user research groups, instrument scientists and computational scientists. A unified software platform that provides an integrated working environment with generic functional modules and services is necessary to meet these requirements [3,4,5]. Scientists can work on their ideas, implement the prototype and check the results following some conventions without dealing with the technical details and the migration between different HPC environments. Thus, one of the vital considerations is integrating extensions into the software in a flexible and configurable way. Another challenge resides in the interactions between instrumental sub-systems, such as control system, data acquisition system, computing infrastructures, data management system [6], data storage system and so on, which can be quite complicated.

Based on an object-oriented plug-in architecture, the Daisy (Data Analysis Integrated Software System) has been designed to address these challenges. The project aims to bridge the gap with an all-encompassing framework between sophisticated computing infrastructure

and the users focusing on scientific issues. This article describes the main components of this software and the status of the project.

2 Design

The main goals of Daisy are the following:

1. Provision of an extensible framework that can easily port existing algorithms.
2. Establish scientific workflows that can involve reusable algorithms in parallel computing systems.
3. Integrated capabilities that can be adopted from other application.
4. Data visualisation and analysis services that can be accessed via a browser and native application

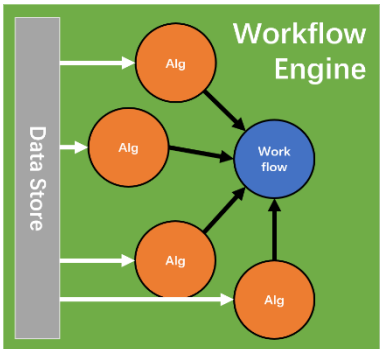

Fig. 2. Daisy framework design

As shown in Fig. 2, the Daisy framework consists of a highly modular C++/Python architecture composed of four pillars: data store, algorithm, workflow and workflow engine. The primary design consideration is the separation of data objects and algorithms [7]. The data store is the data object container that passes on data objects between algorithms. And the algorithm, the handler of existing software code created by third parties, gets input data objects from the data store, passes them out to the external libraries, generates output data objects and sends them back to the data store. The workflow, a sequence of algorithms defined by scientists, is scheduled by the workflow engine. The architecture supporting customised plug-in functions can easily access visualisation tools and the Python-based scientific computing ecosystem. The graphical user interface employs the Model-View-Controller (MVC) architecture for handling data objects, analysis workflow and presentation. Web and native applications based on Jupyter notebook [8] and PyQt [9] integrate script editors, data visualising and processing widgets.

Daisy is designed to separate two main layers: (1) a scientific domain layer focusing on the algorithm, workflow and user interface, and (2) an IT infrastructure layer that interacts with the computing environment and other instrumental sub-systems. The scientific domain layer considers three types of scientists: less experienced users who only wish to execute the pre-defined workflow through the graphical interface or command-line interface; instrument scientists and experts who implement the workflow, specify its parameters, and develop customisable interfaces; and data scientists who study the data processing algorithms and tools for analysis tasks. And the IT infrastructure layer, including the data store and workflow engine, are developed and tested by software engineers. This stratification allows the separation of the technical implementation and the description of sequential calculation. This generic and modular workflow structure can support an extensive range of scientific applications.

2.1 Algorithm, Data Object, and Workflow

A Daisy process relies on a workflow instance to execute a sequence of algorithm instances. The interaction with the Daisy framework occurs through the Python-based Application

Program Interface (API) and the Graphical User Interface (GUI). An algorithm module with pre-defined methods can be initialised, configured, executed, and finalised by the workflow module, whose responsibility is to perform a particular action, such as invoking an external library or a calculation task. To ensure performance as well as flexibility, Python/C++ are both supported to implement the algorithm module. The Python algorithm class is derived from the C++ one, and the framework provides automated Python bindings via the Boost Python library for C++ algorithms.

The data object managed by the data store module is an in-memory transient data structure that facilitates information exchange between algorithms. They can be loaded from various file formats, live streams, or created by algorithms. Except for the persistence algorithms (e.g. LoadAlg and SaveAlg), ordinary algorithms should not access the persistency store but instead use the data objects registered in the data store module. In order to combine multiple scientific software packages where the data structure will be quite complex and unpredictable, the framework does not provide a unitary data model. Still, it compulsorily requires NumPy numerical types as its basic data structures acceptable by Python and C++ execution modules.

In practical term, the algorithm and the data object are both managed by the global mapping relation between the name and the instance, and the details will be described in the following section. The workflow module thus connects a sequence of algorithms to be executed in order and coordinates the flow of data objects between algorithms. The workflow also keeps a set of parameters given by users and holds a set of properties required by algorithms. Since a workflow containing a set of properties can act with the same behaviours as an algorithm to perform the execution task, the workflow class is driven by the algorithm class. It is recommended to organise a complex workflow in a tree structure: a group of algorithms and sub-workflow (optional) embodies a workflow.

The framework needs customisation to implement different data processing applications in various environments. Usually, those scientific specific codes will eventually be encapsulated in two places: algorithm and workflow. With a set of well-defined functions for certain types of interaction, they both support run-time loading via dynamic libraries and allow us to implement actual plug-and-play applications. The examples of those two modules sharing similar interfaces can be achieved, as shown in Fig. 3.

A

```
import numpy as np
import tomopy
from Daisy import DaisyAlg
class AlgTomopyRecon(DaisyAlg):
    def __init__(self, name):
        super().__init__(name)

    def initialize(self):
        self.data = self.get("DataStore").data()
        self.LogInfo("initialized, Tomopy Reconstruction")
        return True

    def execute(self, input_dataobj, theta, center, alg_type, output_dataobj):
        projs  = self.data[input_dataobj]
        thetas = self.data[theta]
        dataobj = tomopy.recon(projs, thetas, center=center, algorithm=alg_type)
        self.data[output_dataobj] = dataobj
        return True

    def finalize(self):
        self.LogInfo("finalized")
        return True
```

B

```
@Daisy.Singleton
class WorkflowCTReconstruct(Daisy.PyWorkflow):
 def execute(self):
  self.engine['loadhdf5'].execute(input_path='/entry/tomo', output_dataobj='tomodata')
  self.engine['loadhdf5'].execute(input_path='/entry/dark', output_dataobj='darkdata')
  self.engine['loadhdf5'].execute(input_path='/entry/flat', output_dataobj='flatdata')
  self.engine['normalize'].execute(projs_dataobj='tomodata',darks_dataobj='darkdata',\
                                   flats_dataobj='flatdata', output_dataobj='normdata')
  self.engine['angles'].execute(input_dataobj='normdata', output_dataobj='thetas')
  self.engine['minuslog'].execute(input_dataobj='normdata', output_dataobj='mlogdata')
  self.engine['reconstruct'].execute(input_dataobj='mlogdata', theta='thetas',\
                                     center=1030, alg_type='fbp',output_dataobj='recodata')
  self.engine['savehdf5'].execute(input_dataobj='recodata',output_path='/entry/reco')

wf = WorkflowCTReconstruct('WorkflowCTReconstruct')
wf.initialize(workflow_engine='PyWorkflowEngine', \
          workflow_environment = init_dict, algorithms_cfg = cfg_dict)
wf.execute()

data =wf.data_keys()
algs =wf.algorithm_keys()

wf.finalize()
```

Fig. 3. Examples of (A) how to invoke an external software within the Algorithm module, and (B) how to implement and execute a Workflow module

2.2 Workflow Engine and Data Store

The data-centric architecture is implemented through the data store to manage the data propagation, an intermediate buffer to minimise the coupling between algorithms. In the execution process, the data object generated by the algorithm registers into the data store and can be accessed by other algorithms in the workflow. The data store is automatically initialised by the workflow engine before the workflow execution and can be accessed by every algorithm (as shown in Fig. 2 A, initialise function). The algorithm requires data objects through the data store by their names, which must be compulsory unique in the workflow. If the new data object is registered with the same name as an older one, the Data Store will replace the expired one with the new one. Thus, there is no need for the user to consider memory management in this process, which can be complicated, especially on the distributed system. As shown in Fig. 3, the data object's name is defined in the workflow, by which the algorithm can access the data object's instance in the data store. This approach is adopted by Gaudi [7] for high energy physics and later Mantid software [5] for neutron scattering experiments. Considering the development of computing techniques, however, the data store is provided as a backend that the workflow engine can define in Daisy. This mechanism ensures the migration of the workflow from one platform to another and preserves the heritage of scientific communities.

```
init_dict ={'loaddata':{'class_name':'LoadHDF5',\
                        'init_paras':{'inputfile_name':'scan_00575_data_000001.h5'}\
                       },\
            'loadmask':{'class_name':'LoadHDF5',\
                        'init_paras':{'inputfile_name':'scan_00575_master.h5'}\
                        },\
            'azintalg':{'class_name':'AlgFAIIntegrate',\
                        'init_paras':{'wavelength':'0.7293'}\
                        },\
            'savedata':{'class_name':'SaveHDF5',\
                        'init_paras':{'outputfile_name':'test_scan.h5'}\
                        }\
            }
```

A

```
cfg_dict = {'azintalg':{'directDist':169,\
                        'centerX':1049.967,\
                        'centerY':1063.892,\
                        'pixelX':75,\
                        'pixelY':75,\
                        'PlanRotation':64.66877,\
                        'tilt':0.3753,\
                        'azimin':93,\
                        'azimax':115,\
                        'radmin':10.0,\
                        'radmax':20,\
                        'ntth':40}\
            }
```

B

Fig. 4. Examples of workflow configurations

As shown in Fig. 3 B, The workflow engine schedules the algorithms explicitly through their names defined in the workflow. The workflow engine, the intermediate layer between the computing environments and the scientific application, is responsible for initialising the data store and the algorithm, sequential or parallel execution of algorithms, and other essential

services such as logging and error tracking. Therefore, computing resources and initial parameters need to be specified in the initialisation phase of the workflow execution. As shown in Fig. 4, two types of JSON schemes are presented here. Fig. 4 A shows the requirements of the module, which includes the module name, the version, and other information required by the Computing Infrastructure. And Fig. 4 B shows the parameters needed by the algorithms, which are usually stored in the Data Management System and maintained by instrument scientists.

The foundation of the in-process workflow engine is an in-house developed framework, SNiPER [10], which supports many experiments led by IHEP, like JUNO [11], LHAASO [12], and CSNS [13]. Its lightweight plug-in architecture inspired by the GAUDI [7] framework uses a key-value dictionary as the index for locating and invoking the dynamic library across the C++ and Python modules. Furthermore, as shown in Fig. 5, the distributed workflow engine is based on Apache Spark [14], a unified analytics engine for large-scale data processing. Firstly, the Spark workflow engine creates a SparkContext, which employs the Resilient Distributed Dataset (RDD) as its data store. Secondly, the workflow is split into many small Spark tasks (sub-workflows) distributed to worker nodes. Then, the SNiPER workflow engines on the worker nodes receive data from the RDD and execute the sub-workflows simultaneously. Finally, the output data generated by the sub-workflows is collected in the RDD and sent back to the Spark Driver Program (top-workflow). In this case, the Spark workflow engine translates the top-workflow script into multiple sub-workflows executed by the SNiPER workflow engine on the cluster.

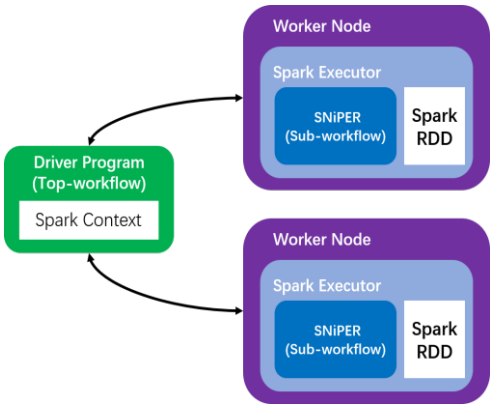

Fig. 5. Workflow on Apache Spark

3 Prototype: HEPS testbed at 3W1A of BSRF

Beijing Synchrotron Radiation Facility (BSRF) [15] sets up a testbed at beamline 3W1A dedicated to verifying state-of-the-art techniques. As shown in Fig. 6 A, The HEPS computing and communication group (HEPS-CC) also establishes an integrated computing environment providing the advanced information infrastructure for high-throughput analysis services. The Data Management System [6], based on SciCat [16], aims to manage the whole data lifecycle. The Data Storage System employs Lustre [17] to satisfy the requirements of massive data production in the HEPS. The Computing System focuses on Kubernetes [18], Docker [19] and JupyterHub [20] to implement a scalable and flexible computing environment for users. Mamba, a science-oriented data acquisition software on top of Bluesky [21], offers a highly customisable solution for experimental plans and high-performance data multiplexing, as well as the capabilities of natural integration with Daisy.

Integrating an open-source 3D tomographic reconstruction package, TomoPy [22], into the framework, Daisy provides a remote accessing client employing the Jupyter message protocol and data visualisation based on Jupyter Widget (as shown in Fig. 6 B). Mamba triggers the pre-defined workflow script, which is automatically scheduled by a Kubernetes

cluster. The parallelisation supported by the Spark cluster and its incorporation with DMS is on the way.

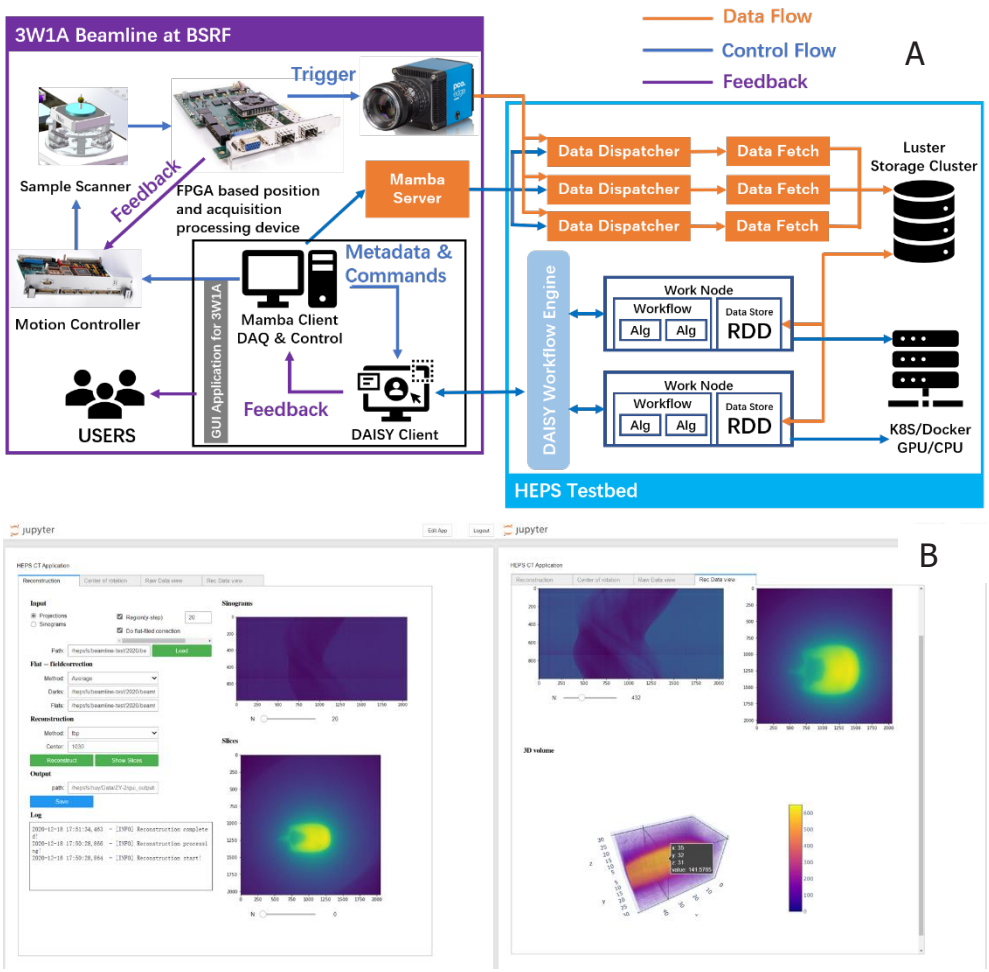

Fig. 6. (A)The scheme of HEPS testbed for tomographic experiment at 3W1A beamline of BSRF; (B)The Web-based graphic user interface powered by Jupyter Widget.

4 Status and Outlook

A proof-of-concept design has been implemented, which is now being used to perform tomography reconstruction application on the HEPS testbed. Development is proceeding by expanding an extensive range of necessary components, including algorithms that integrate the TomoPy and HDF IO [23] module h5py [24] and data visualisation service based on Jupyter Widget. In the area of the distributed computing system, the development of a Spark workflow engine is continuing that involves PySpark and an HDF5 connector [25] for Apache Spark. In addition, the user interface is also extended to support PyQt5 and provide remote access through the Jupyter message protocol. Furthermore, collaborations with scientific communities are also essential to develop this framework further. A user-friendly

integrated development environment and graphic user interface should be provided for end-users to integrate algorithms and set up data analysis applications.

References

1. Yi Jiao, Zhe Duan, Yuanyuan Gao, et al., Physics Procedia **84**, 40 – 46 (2016)
2. Incardona, M.-F., Bourenkov, G. P., Levik, K., Pieritz, R. A., Popov, A. N. & Svensson, O., J. Synchrotron Radiat. **16**, 872–879 (2009).
3. Basham, M., Filik, J., Wharmby, M. T., et al., J. Synchrotron Radiat. **22**, 853–858 (2015).
4. Tian, H. L., Zhang, J. R., Yan, L. L., et al., Nucl. Instrum. MethodsPhys. Res. Sect. A Accel. Spectrometers, Detect. Assoc. Equip. **834**, 24–29 (2016).
5. O. Arnold, J. Bilheux, J. Borreguero, A. Buts, S. Campbell, et al, Nucl. Instrum. MethodsPhys. Res. Sect. A: Accel. Spectrom. Detect.Assoc.Equip.**764**, 156–166 (2014)
6. Hao Hu, Fazhi Qi, Hongmei Zhang, Haolai Tian and Qi Luo, J. Synchrotron Rad. **28**, 169–175 (2021).
7. G. Barrand, I. Belyaev, P. Binko, M. Cattaneo, R. Chytracek, G. Corti, M. Frank, G. Gracia, J. Harvey, E. Herwijnen, P. Maley, P. Mato, S. Probst, F. Ranjard, Comput.Phys.Commun.**140**(1) 45–55(2001)
8. *Project Jupyter,* https://jupyter.org (2021), accessed: 2021-05-01
9. *Qt for Python,* https://doc.qt.io/qtforpython/ (2021), accessed: 2021-05-01
10. J. H. Zou, et al., 2015 J. Phys.: Conf. Ser. **664**, 072053 (2015)
11. Cedric Cerna (behalf of the JUNO collaboration), Nucl. Instrum. MethodsPhys. Res. Sect. A: Accel. Spectrom. Detect.Assoc.Equip.**958**, 162183 (2020)
12. Cao Zhen, et al., Chinese Phys. C **34** 249 (2010)
13. F.Wang, T.Liang, W.Yin, Q.Yu, L.He, J.Tao, T.Zhu, X.Jia, S.Zhang, Sci. ChinaPhys. Mech. Astron. **56**, 2410–2424 (2013).
14. *Apache Spark,* https://spark.apache.org/ (2021), accessed: 2021-05-01
15. *Beijing Synchrotron Radiation Facility,* http://english.bsrf.ihep.cas.cn/ (2021), accessed: 2021-05-01
16. *The ICAT Project,* https://icatproject.org (2021), accessed: 2021-05-01
17. *Lustre,* https://www.lustre.org/ (2021), accessed: 2021-0-01
18. *Kubernetes,* https://kubernetes.io/ (2021), accessed: 2021-05-01
19. *Docker,* https://www.docker.com/ (2021), accessed: 2021-05-01
20. *Project Jupyter,* https://jupyter.org/hub (2021), accessed: 2021-05-01
21. Koerner, L. J., Caswell, T. A., Allan, D. B. & Campbell, S. I., IEEE Trans. Instrum. Meas. **69**, 1698–1707(2019)
22. Gursoy, D., De Carlo, F., Xiao, X. & Jacobsen, C., J. Synchrotron Radiat. **21**, 1188–1193 (2014).
23. Rees, N., Billich, H., Götz, A., Koziol, Q., Pourmal, E., Rissi, M. & Wintersberger, E., Proc. ICALEPCS 2015 (2015).
24. *HDF5 for Python,* https://www.h5py.org (2021), accessed: 2021-05-01

25. *Apache Spark Connector,* https://www.hdfgroup.org/solutions/enterprise-support/apache-spark-connector/ (2021), accessed: 2021-05-01