EPJ Web of Conferences **245**, 07046 (2020) https://doi.org/10.1051/epjconf/202024507046
CHEP 2019

Real-time HEP analysis with funcX, a high-performance platform for function as a service

Anna Elizabeth Woodard[1,*], *Ana* Trisovic[1,**], *Zhuozhao* Li[1], *Yadu* Babuji[1], *Ryan* Chard[2], *Tyler* Skluzacek[1], *Ben* Blaiszik[1,2], *Daniel S.* Katz[3], *Ian* Foster[1,2], and *Kyle* Chard[1]

[1]The University of Chicago
[2]Argonne National Laboratory
[3]University of Illinois at Urbana-Champaign

Abstract. We explore how the function as a service paradigm can be used to address the computing challenges in experimental high-energy physics at CERN. As a case study, we use funcX—a high-performance function as a service platform that enables intuitive, flexible, efficient, and scalable remote function execution on existing infrastructure—to parallelize an analysis operating on columnar data to aggregate histograms of analysis products of interest in real-time. We demonstrate efficient execution of such analyses on heterogeneous resources.

1 Introduction

High-energy physics (HEP) research using accelerators such as the Large Hadron Collider (LHC) requires the processing of massive sets of data recorded from particle collisions. Raw data from detectors must be reconstructed into abstract representations of physics objects that include information on particle properties such as its type, charge, momentum, and energy. This is carried out in successive processing steps that reduce the initial dataset from hundreds of petabytes down to tens of terabytes [1]. Researchers then analyze a subset of this processed data to identify particle signals of interest and produce summary statistics, used to quantify the compatibility between the observed data and theoretical predictions.

After initial data processing, HEP analyses are often carried out using monolithic programs developed by a single researcher or research group. This lack of modularization and reuse leads to duplicated effort, slowing the pace of analysis development. Further, analyses require significant computing resources and typically each user writes and maintains custom batch scripts for their institutional or university cluster. This limits portability because different clusters have different schedulers and hardware configurations. It also reduces scalability because it is difficult to run a single analysis on multiple sets of computing resources simultaneously.

Function as a service (FaaS) is a computing paradigm that simplifies the deployment of applications on computing resources. The user registers a "function," a fragment of code with defined inputs, logic, and outputs with a FaaS system. Users can then execute these functions by supplying input arguments. Importantly, users need not manage the physical (e.g., computers) or virtual infrastructure (containers and virtual machines) on which functions are executed. The registered functions could perform particle reconstruction, application of detector corrections, data selection and transformation, calculation of derived quantities, or any other HEP computation. While the FaaS paradigm aligns well with HEP

*e-mail: annawoodard@uchicago.edu
**e-mail: anatrisovic@g.harvard.edu

analysis needs, most FaaS implementations are offered as pay-per-use services by cloud providers or as services deployed on Kubernetes clusters and thus are unable to leverage the considerable high performance and high throughput of computing resources available to HEP researchers.

FuncX [2] is a free and open-source distributed FaaS platform. It adapts the typically cloud-hosted FaaS model to support research by enabling scientific workloads to be distributed across existing infrastructure including local computers, clusters, clouds, or supercomputers. Users interact with the funcX service via a cloud-hosted REST API. The service provides for registration of remote computing systems by deploying a funcX *endpoint* agent. The service then allows for management and sharing of computing endpoints, function registration and sharing, and secure and reliable execution of registered functions on remote endpoints. While funcX can be employed in any domain, in this paper we explore its use in experimental HEP.

FuncX provides several advantages for research in HEP:

- The funcX platform accelerates software development by facilitating the decomposition of large applications into functions that can be reused and iteratively improved by a community of researchers. When a function is registered with funcX, it is assigned a universally unique identifier (UUID) that can be shared among users and used for subsequent invocations in related workflows.
- FuncX makes it straightforward to access remote and heterogeneous resources by providing a common interface to clouds, campus clusters, HPC systems, and supercomputers. The amount of resources allocated by the endpoint is automatically scaled according to demand. Users need not define batch submission scripts nor do they need to interact with schedulers or manage nodes.
- FuncX can provide, by using containers, a uniform execution environment across different hardware and software infrastructures. This addresses the challenge of portability and increases the likelihood of reproducibility of scientific results, as the code can run identically in different execution environments without requiring the user to synchronize the execution dependencies.
- FuncX allows each portion of the code to run on the most appropriate endpoint. For example, most of a processing task might be appropriate for a conventional CPU, but a portion of the analysis that utilizes machine learning may run more efficiently on a GPU. The FaaS model makes it feasible to assign different parts of a large application to different endpoints based on where the data is located, what hardware is optimal to run the function, and where idle resources are available.
- Separating the functional code from the details of the execution environment makes it possible to easily combine heterogeneous resources. Under the traditional approach, the user must prepare and submit a batch script, authenticate separately on each resource, set up software and data, run the processing task, and then pool results. With funcX, the user authenticates only once, using a permitted identity and an OAuth 2.0 authentication flow, and can execute functions using a single Python SDK interface (see Listing 2).
- Function execution is low-latency, enabling interactive analysis within Jupyter notebooks. Jupyter notebooks combine source code, documentation, and results into a single entity, facilitating faster data exploration and prototyping.

As a case study to demonstrate how funcX can address the computing challenges of HEP research, we have implemented a backend for the Columnar Object Framework For Effective Analysis (Coffea) [3, 4], which transparently scales processing by distributing computations via funcX functions. In Section 2, we introduce funcX and its architecture. In Section 3, we describe the implementation of the funcX backend for Coffea. In Section 4, we present an evaluation of the backend performance using a real-world analysis of data from the Compact Muon Solenoid (CMS) experiment [5] at the LHC. Finally, in Section 5 we summarize our contribution.

2 FuncX

The funcX system is comprised of a single, cloud-hosted web service, currently deployed on Amazon Web Services (AWS), and a distributed pool of funcX endpoints deployed on various edge devices (personal computers, clusters, supercomputers, Raspberry Pis). The funcX architecture is shown in Figure 1.

A funcX function is defined as a Python function body and its input signature. The function body must specify all imported modules. Before a function can be used, it must be registered with the funcX service. Each registered function is available to all users and assigned a UUID by which it can be accessed. Functions are invoked asynchronously; we refer to a function invocation as a task. Invoking a function returns a UUID by which its status can be queried and its results retrieved.

FuncX executes functions on remote *endpoints*, each of which represents a set of computational resources. The endpoint is responsible for authentication, provisioning of computing resources, and monitoring performance. Endpoints may be initiated by any user authorized to start processes on the resource. Each endpoint is registered via the funcX service and assigned a UUID when it is created. The endpoint, built on the Parsl [6] parallel programming library, provisions local resources on-demand and initiates a manager process on each node. The manager starts a set of worker processes (optionally within a user-specified container), each of which executes one task at a time and returns the resulting data. The number of workers corresponds to the number of concurrent tasks that can be run on the node.

All interactions with funcX are carried out via the *funcX client*, which wraps the funcX service's REST API. This interface is used for registering functions, endpoints, and containers. Using the client, authorized users can invoke a registered function on a specified endpoint. Users may optionally specify a container in which the function will be executed. In this case, the manager will either deploy a worker running within the requested container, or if such a worker is already available, assign the task to an existing worker. The interface also provides for monitoring the status of tasks and retrieving results.

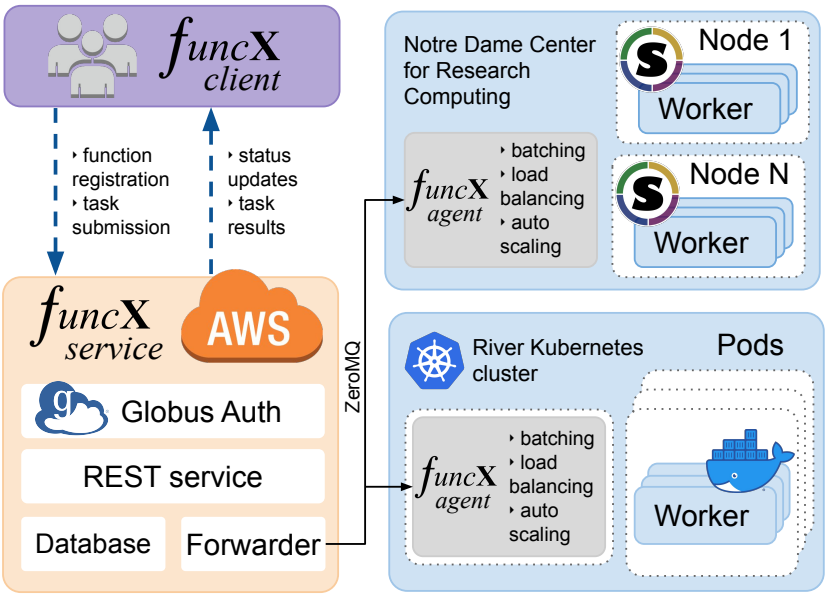

Figure 1. Overview of the funcX system architecture. The The funcX client abstracts inteactions with funcX. The funcX service is used to register and manage endpoints, register and execute functions, and retrieve results. FuncX endpoints are deployed on remote computers (in this case at Notre Dame and the River Kubernetes cluster) and enable function execution on provisioned nodes, optionally using locally supported containers.

```
~ >pip install funcx
~ >funcx-endpoint configure my_endpoint
A default profile has been create for <my_endpoint> at
/afs/crc.nd.edu/user/awoodard/.funcx/my_endpoint/config.py
Configure this file and try restarting with:
    > funcx-endpoint start my_endpoint
~ >funcx-endpoint start my_endpoint
It looks like this is the first time you're accessing this service.
Please log in to Globus at this link:
 https://auth.globus.org/v2/oauth2/authorize?client_id=[...]
Copy and paste the authorization code here:
XXXXXXXXXXXXXXXXXX
Thanks! You're now logged in.
2019-11-02 13:45:30 funcx:252 [INFO]  Endpoint registered with UUID:
f8696260-c060-4f2b-814f-f5ba917f8472
```

Listing 1: Example command-line interaction which 1) installs funcX, 2) configures an endpoint, 3) authenticates, and 4) starts the endpoint. User inputs are highlighted in blue.

2.1 Interacting with funcX

We describe below the process for using funcX. An endpoint is started on the remote resource, authenticated, and registered to the funcX service (see Listing 1). The endpoint process has a default configuration but can be configured based on the interface and capabilities of a particular resource. The configuration specifies what scheduler is used (e.g., Slurm, PBS, Kubernetes), the allocation in which nodes are provisioned, and how many nodes may be requested, among other options.

Listing 2 shows an example of using the Python SDK to register and execute a function (`compute_sum`) on a specific endpoint. First, the funcX client instance is instantiated. A Python function is defined (in this case it simply returns the sum of an input list). The function is then registered with the funcX service, which returns a UUID. The function is then invoked on a specific endpoint (identified by UUID) with a set of inputs. Finally, the result is fetched using the task id returned from the invocation.

2.2 FuncX operation

The actual flow of code and data is as follows. FuncX is registered as a Globus Auth [7] resource server, with specific OAuth scopes defined for programmatic access. Before using funcX, a user must first authenticate via Globus Auth. Here they may authenticate using an identity from any of Globus Auth's hundreds of supported identity providers (e.g., campus identities, ORCID, Google). When accessing funcX via an external client, the user authenticates via Globus Auth which provides OAuth access tokens to the Python script to access the funcX service.

The funcX service provides the API and maintains a registry of users, functions, containers, and endpoints in a persistent database. FuncX endpoints are registered as Globus Auth native clients with access tokens provided by the installing user. Here the funcX endpoint makes a secure call to the funcX service to register the endpoint with the service as the installing user. Endpoints may be shared with other users; however, to ensure that funcX does not violate compute facility security policies, sharing must be enabled by funcX administrators.

A function is then registered by the user through the client by calling `client.register_function`. The Python SDK serializes the body of the function and sends it alongside descriptive metadata and the input signature to the funcX service. The funcX service stores the function signature and the serialized function body. Users may share functions with other users by specifying either a user-managed Globus group or public access.

```python
from funcx.sdk.client import FuncXClient

client = FuncXClient()

def compute_sum(items):
    return sum(items)

func_uuid = client.register_function(
    compute_sum,
    description="A sum function"
)

endpoint_uuid = 'f8696260-c060-4f2b-814f-f5ba917f8472'
task_id = client.run(
    [1, 3, 5],
    endpoint_id=endpoint_uuid,
    function_id=func_uuid
)

result = client.get_result(task_id) # result is now 9
```

Listing 2: Python script demonstrating the use of funcX to register and execute a simple function.

Permitted users can execute functions by calling `client.run`, specifying the function UUID and endpoint UUID on which it should run, and the input arguments to be passed to the function. The user request is added to the funcX service task queue. The task is routed by the service to the endpoint, which adds it to the endpoint's local task queue. Tasks on the endpoint queue are distributed to the node managers, which assign tasks to the workers. The workers execute tasks and return results to the endpoint manager. The endpoint manager returns the function result to the service, where it is stored in a result database. The client requests the result from the funcX service.

3 HEP analysis with funcX

To explore the benefits of the funcX platform, and FaaS in general, in HEP research, we developed a prototype backend for Coffea. Coffea is a HEP analysis framework which provides tools for histogramming, plotting, and performing data transformations and corrections. Coffea achieves substantial performance improvements compared with the traditional approaches that sequentially loop over particle collision events, by replacing the event loop with array programming primitives operating on individual columns of event data. Coffea enforces an interface by which user code is wrapped in a high-level wrapper called a *processor*. Given input event data, the processor returns a reducible object (typically, a set of histograms.) HEP events are statistically independent, making workflows pleasantly parallelizable; this facilitates the use of funcX for low-latency real-time analysis.

FuncX enhances Coffea by allowing it to transparently access (and scale across) external processing resources. All Coffea analyses utilizing the funcX backend use the same registered funcX function. This backend function's UUID is stored in a JSON formatted-file which is saved in the Coffea repository and read in at run-time. As the Coffea code base evolves, modifications to the backend function may be required; after such changes, the function should be re-registered and the JSON file updated. A convenience script developers can use for this purpose is provided. The backend function runs on each worker, sets up data access; stages in, loads, and executes the Coffea processor; stages out the result; and returns a unique identifier by which the results are retrieved. The user can then perform an analysis by providing the Coffea processor and specifying a list of endpoint UUIDs, a set of input files, and the number of events to be assigned to each task.

```
from coffea.util import load
from coffea.processor import funcx_executor, run_uproot_job

ndt3_ep = '1c0434db-1e2d-43c1-86ae-e2c80b6d25ae'
ndcrc_ep = '8bd5cb36-1eec-4769-b001-6b34fa8f9dc7'
wisconsin_ep = 'af21d0db-27f2-4906-beba-6baffac18393'
stageout_url = 'root://deepthought.crc.nd.edu://store/user/awoodard/data'

with open('samplefiles.json') as f:
    dataset = json.load(f)['Hbb_2017']

accumulator, metrics = run_uproot_job(
    dataset,
    'otree',
    load('boostedHbbProcessor.coffea'),
    funcx_executor,
    executor_args={
        'local_path': '/hadoop/store/user/awoodard/data',
        'stageout_url': stageout_url,
        'endpoints': [ndt3_ep, ndcrc_ep, wisconsin_ep],
        'tailretry': 120
    }
)
```

Listing 3: Example Python script demonstrating the use of the funcX executor to scale out a physics analysis to three endpoints.

The input data is specified as a dictionary of datasets and file URLs using any protocol supported by uproot [8] (currently HTTP, local file access, and XrootD). To improve performance in cases where there may be transient data access problems, the funcX backend allows the user to specify a retry timeout for tail tasks. Once this timeout is exceeded, any outstanding tasks are automatically resubmitted. The funcX backend monitors for tasks that might be lost (e.g., due to termination of an endpoint manager or worker on an opportunistic endpoint) and automatically resubmits them. To reduce data transfer through the funcX service, task outputs are staged out directly, either via XrootD [9] or, where possible, over a shared filesystem. Data access and stage-out via XrootD require valid grid credentials. In this prototype implementation, grid credentials for a single user were included in the endpoint configuration; future work will implement a more sophisticated multi-tenant model. Following task submission, the Coffea funcX backend periodically polls the funcX service for any completed tasks. As tasks complete, results are collected from the stage-out locations, reassembled by the funcX backend and returned to the user as a Python object.

The analysis execution environment used by the funcX workers is specified through a Dockerfile[1]: a docker container that is built automatically on DockerHub[2]. The container captures a build of Coffea source code [10] that provides the needed environment to run the funcX backend function.

4 Evaluation

To evaluate the performance of the funcX backend, we carried out scalability tests using a Coffea analysis processor[3] that was developed for an analysis of events recorded by the CMS experiment at the LHC. This analysis searches for events where a Higgs boson is produced and subsequently

[1] https://github.com/globus-labs/coffea/blob/master/docker/kubernetes/funcx/Dockerfile
[2] https://hub.docker.com/repository/docker/funcx/coffea
[3] https://github.com/nsmith-/coffeandbacon

decays to two bottom quarks. We wrote a Python program that uses the Coffea funcX backend to execute this analysis processor on 425 GB of input data and return a typical set of histograms and summary statistics. A maximum of 200000 events per task was used for all experiments.

To distinguish funcX performance limitations from bottlenecks due to XrootD data I/O, we first carried out a set of scaling tests at the University of Chicago's Research Computing Center Midway, a shared campus cluster. Midway has more than 350 compute nodes, each with 28 CPU cores and 64 GB RAM, and interconnected with high-speed InfiBand, with data served via a high-performance General Parallel File System. To simulate the load due to multiple analyzers using the system concurrently, we ran the analysis program simultaneously in separate processes on the cluster login node. We studied both the strong scaling (performance processing a fixed number of tasks scales as the number of cores increases) and weak scaling (how the performance scales as the number of cores increases, with a fixed number of tasks per core) of a single endpoint. The strong scaling is shown in Figure 2 for both one and four concurrent analyses on a single endpoint. As the number of cores increases, the makespan to run a single analysis decreases until about 250 cores are assigned. After this point, additional cores confer no benefit, and as the number increases further, the makespan increases slightly due to the overhead of maintaining the additional workers. The weak scaling is shown in Figure 2. In the ideal case, increasing the number of concurrent analyses while holding the cores per analysis fixed would not increase runtime. We observe that performance begins to decline with three concurrent analyses sharing the same endpoint, but remains under ten minutes with up to ten analysts sharing an endpoint.

To demonstrate the ability of the funcX executor to integrate heterogeneous resources, we deployed endpoints at both the University of Wisconsin-Madison CMS Tier-2 computing cluster and the University of Notre Dame's Center for Research Computing (NDCRC). Computing resources at the University of Wisconsin-Madison present a mix of machines with different numbers of CPU cores ranging from 8 to 48 [11]. There is a similar mix of hardware at the NDCRC, where there are nodes with from 12 to 64 CPU cores and from 12 GB to 3 TB RAM [12]. At the NDCRC, resources are available on an opportunistic basis; thus this test also demonstrates the robustness of the funcX backend to manage situations where resources are only available intermittently. We then ran a test analysis with input data read and staged results out via XrootD, which completed in 9.2 min.

To demonstrate the utility of containers to standardize the execution environment, and the ability of funcX to enable access to diverse resources, we set up an endpoint on River, a 70-node Kubernetes cluster (Kubernetes v1.15), with 48 cores, 256 GB RAM, two 800 GB SSDs, and 10 Gbit NIC on each node. The River endpoint was configured to scale out to a maximum of 200 pods. We ran a test analysis using a container image to standardize the execution environment. This test completed in 8.5 min.

The benchmarking and plotting code, analysis processor, endpoint configurations, and input file lists used in this evaluation are available on GitHub[4].

These results demonstrate the benefits of using funcX for HEP analyses. They show that analyses can be moved between four heterogeneous clusters without modifying the analysis code, that multiple resources can be used concurrently, and that funcX endpoints can scale to support several analysts simultaneously. Further, in all cases, our analyses are able to provide results within 10 minutes—a threshold stated by HEP researchers to be crucial for interactive analyses.

[4]https://github.com/annawoodard/CHEP2019-funcx-coffea-benchmarks

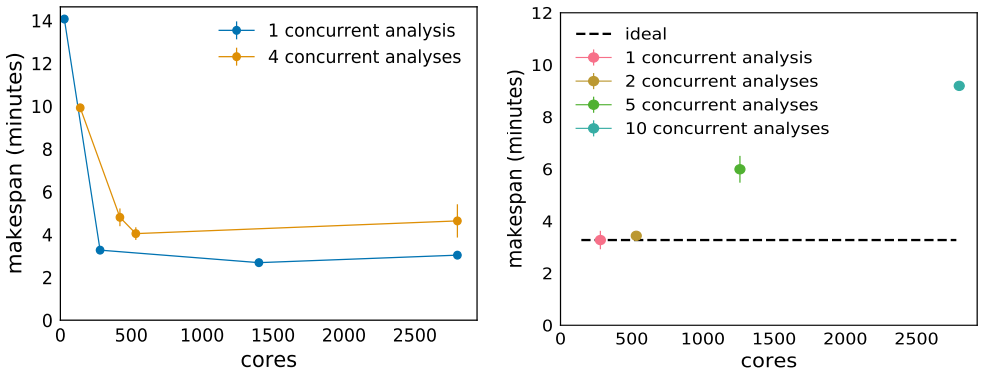

Figure 2. Two views of total execution time as a function of number of dedicated endpoint cores on Midway, for varying numbers of concurrent analyses. For strong scaling (left), number of concurrent analyses is fixed at 1 or 4. For weak scaling (right), number of cores per analysis is fixed for each point at ~256. Error bars denote the standard deviation of time measurements.

5 Conclusion

The FaaS paradigm provides a range of benefits to HEP analyses, most notably the ability to decompose monolithic applications and execute components remotely. Distributed FaaS, as implemented by funcX, allow functions to be executed on the most appropriate computing resource, based on cost, execution time, and resource availability. The use of functions and containers can improve modularity, reuse, and reproducibility of user code. We have tested a prototype implementation of a funcX processing backend for the Coffea physics analysis framework and shown that it achieves good performance while integrating computing resources from multiple sites. Our experiences show that interactive analyses, delivering results within 10 minutes, can be realized on a range of clusters even when endpoints are shared by multiple analysts.

In addition to the analyst use case described here, funcX-based solutions may be useful for other HEP scenarios. For example, the LHC experiments have a long-term commitment to make their physics data openly accessible [13, 14]. However, this data is impossible to analyze without detailed documentation and extensive computational infrastructure due to its vast size and complexity. FuncX provides a possible solution, simplifying remote analysis of CERN Open Data and supporting a range of educational and outreach activities. By hosting a funcX endpoint, the experiments reduce the burden of (1) developing accurate code for data analysis and (2) providing computational infrastructure to citizen scientists and students, and thus allow them to more effectively learn and analyze CERN Open Data.

Acknowledgements

This work was supported in part by NSF 1550588. A. Trisovic is funded by the Sloan Foundation. This research used resources of the Argonne Leadership Computing Facility, a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. We thank the Argonne Leadership Computing Facility for access to the PetrelKube Kubernetes cluster, and Amazon Web Services for providing research credits for rapid service prototyping. This research was also supported in part by the Notre Dame Center for Research Computing and the University of Chicago Research Computing Center.

Special thanks to Kevin Lannon, Kenyi Hurtado, Paul Brenner, and others at Notre Dame, Chad Seys and others at Wisconsin, and Lincoln Bryant and Rob Gardner at the University of Chicago, for providing site access and help with site testing; and to Lindsey Gray and the Coffea Team.

References

[1] J. Albrecht, A.A. Alves, G. Amadio, G. Andronico, N. Anh-Ky, L. Aphecetche, J. Apostolakis, M. Asai, L. Atzori, M. Babik et al., Computing and Software for Big Science **3**, 7 (2019)

[2] R. Chard, T.J. Skluzacek, Z. Li, Y. Babuji, A. Woodard, B. Blaiszik, S. Tuecke, I. Foster, K. Chard, arXiv preprint arXiv:1908.04907 (2019)

[3] *Coffea*, accessed March 9, 2020., https://github.com/CoffeaTeam/coffea

[4] N. Smith, L. Gray, M. Cremonesi, B. Jayatilaka, O. Gutsche, A. Hall, K. Pedro, M. Acosta, A. Melo, S. Belforte et al., *COFFEA - Columnar Object Framework For Effective Analysis*, in *24th International Conference on Computing in High Energy and Nuclear Physics* (2019), https://indi.to/7MnJd

[5] R. Adolphi et al., Jinst **803**, S08004 (2008)

[6] Y. Babuji, A. Woodard, Z. Li, D.S. Katz, B. Clifford, R. Kumar, L. Lacinski, R. Chard, J.M. Wozniak, I. Foster et al., *Parsl: Pervasive Parallel Programming in Python*, in *28th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)* (2019), babuji19parsl.pdf, https://doi.org/10.1145/3307681.3325400

[7] S. Tuecke, R. Ananthakrishnan, K. Chard, M. Lidman, B. McCollam, S. Rosen, I. Foster, *Globus Auth: A research identity and access management platform*, in *12th IEEE International Conference on e-Science* (2016), pp. 203–212, ISSN null

[8] J. Pivarski, P. Das, C. Burr, D. Smirnov, M. Feickert, T. Gal, N. Smith, O. Shadura, N. Biederbeck, M. Proffitt et al., *scikit-hep/uproot: 3.11.3* (2020), https://doi.org/10.5281/zenodo.3662720

[9] S. Campana, D.C. Van der Ster, A. Di Girolamo, A.J. Peters, D. Duellmann, M.C. Dos Santos, J. Iven, T. Bell, *Commissioning of a CERN production and analysis facility based on xrootd*, in *Journal of Physics: Conference Series* (IOP Publishing, 2011), Vol. 331, p. 072006

[10] L. Gray, N. Smith, A. Novak, A. Woodard, D. Taylor, P. Gessinger, J. Pata, Andreas, dnoonan08, Lukas et al., *annawoodard/coffea: v0.6.33-funcx-chep* (2020), https://doi.org/10.5281/zenodo.3899002

[11] *CMS-T2 Resources at Wisconsin-Madison*, accessed March 14, 2020., https://www.hep.wisc.edu/cms/comp/resource.html

[12] *Compute resources at the University of Notre Dame's Center for Research Computing*, accessed March 14, 2020., https://wiki.crc.nd.edu/w/index.php/Available_Hardware

[13] A. Trisovic, Ph.D. thesis, University of Cambridge (2018)

[14] X. Chen, S. Dallmeier-Tiessen, R. Dasler, S. Feger, P. Fokianos, J.B. Gonzalez, H. Hirvonsalo, D. Kousidis, A. Lavasa, S. Mele et al., Nature Physics p. 1 (2018)