Distributed statistical inference with `pyhf` powered by `funcX`

Matthew Feickert

matthew.feickert@cern.ch
@HEPfeickert
matthewfeickert

Physics and Astronomy Mini-symposia
SciPy 2021
July 15th, 2021

Project team

Lukas Heinrich

CERN

Matthew Feickert

University of Illinois
Urbana-Champaign

Giordon Stark

UCSC SCIPP

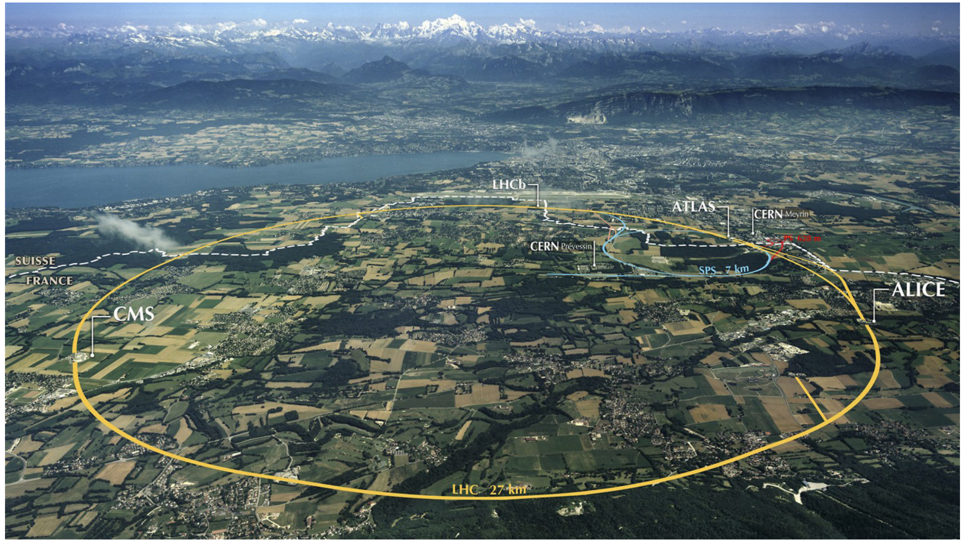
Ben Galewsky

National Center for
Supercomputing
Applications/Illinois

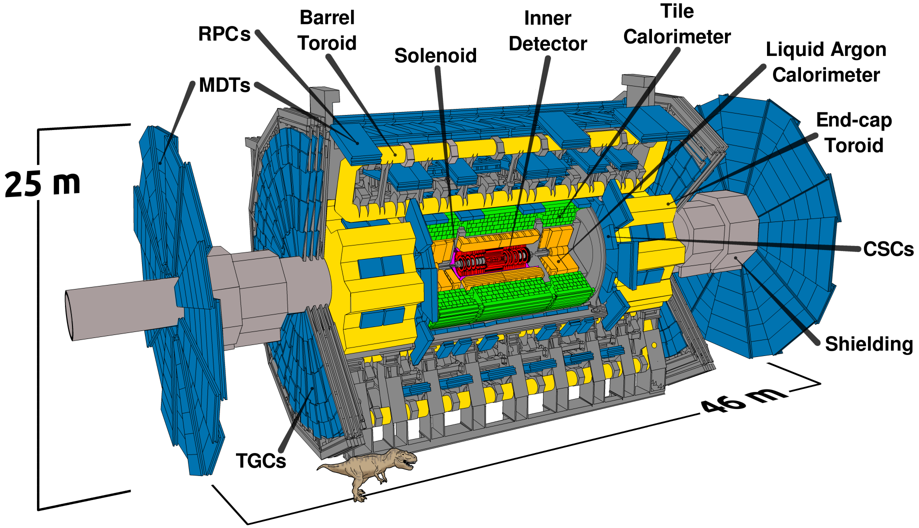
`pyhf` **Core Developers**

`funcX` **Developer**

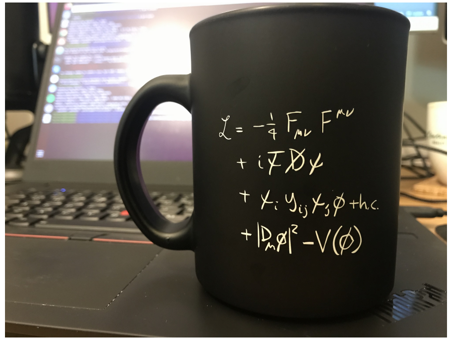
We're high energy particle physicists



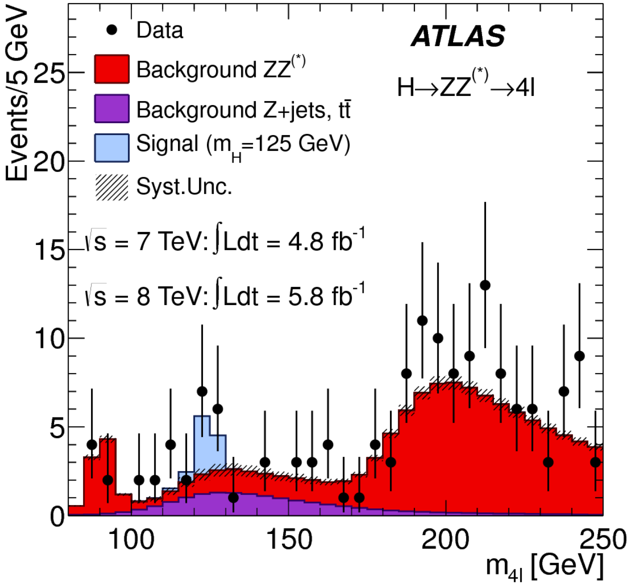
LHC



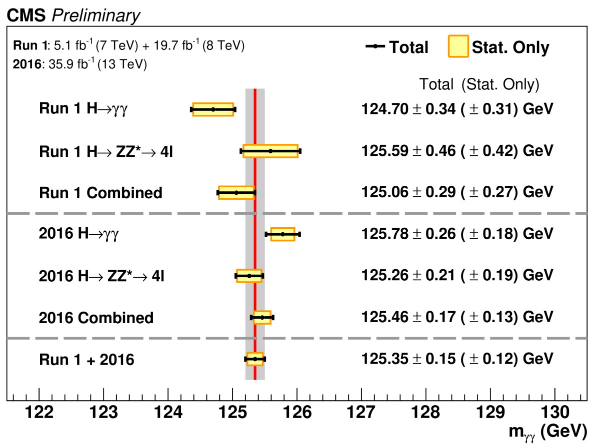
ATLAS



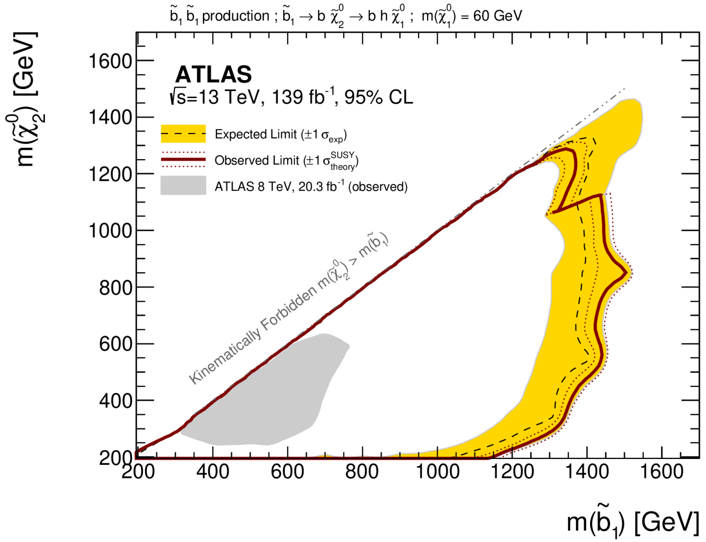
Goals of physics analysis at the LHC



Search for new physics



Make precision measurements

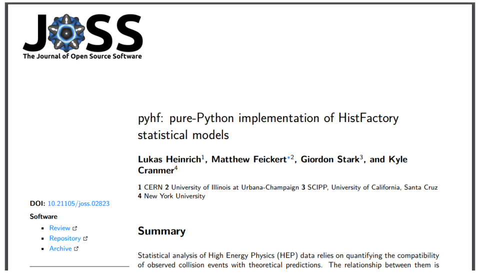


Provide constraints on models through setting best limits

- All require **building statistical models** and **fitting models** to data to perform statistical inference
- Model complexity can be huge for complicated searches
- Problem: Time to fit can be **many hours**
- `pyhf` Goal: Empower analysts with fast fits and expressive models

pyhf: pure-Python HistFactory statistical models

- Pure Python implementation of ubiquitous high energy physics (HEP) statistical model specification for multi-bin histogram-based analysis
- Supports **multiple computational backends** and optimizers (defaults of NumPy and SciPy)
- JAX, TensorFlow, and PyTorch backends can leverage hardware acceleration (GPUs, TPUs) and automatic differentiation
- Possible to outperform traditional C++ implementations that are default in HEP
- Ways to learn more:



JOSS
The Journal of Open Source Software

pyhf: pure-Python implementation of HistFactory statistical models

Lukas Heinrich[1], Matthew Feickert[*2], Giordon Stark[3], and Kyle Cranmer[4]

1 CERN 2 University of Illinois at Urbana-Champaign 3 SCIPP, University of California, Santa Cruz 4 New York University

DOI: 10.21105/joss.02823

Software
- Review
- Repository
- Archive

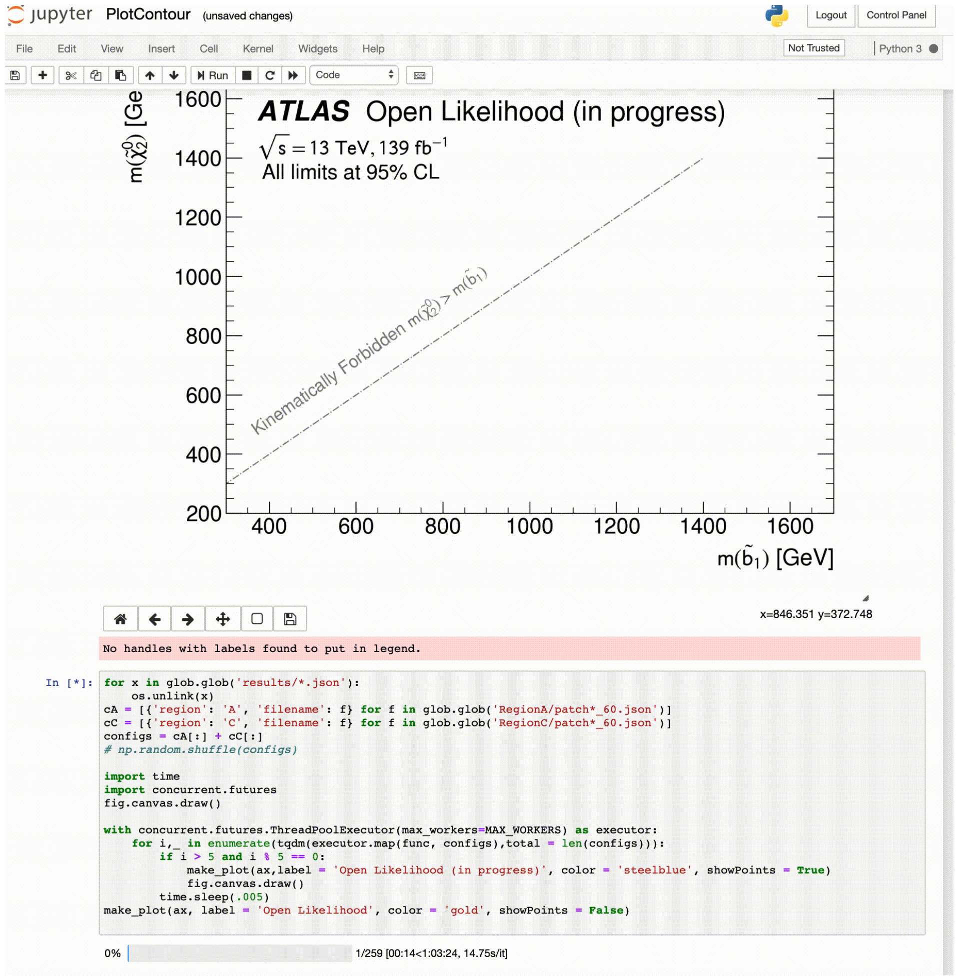
Summary

Statistical analysis of High Energy Physics (HEP) data relies on quantifying the compatibility of observed collision events with theoretical predictions. The relationship between them is

JOSS 10.21105/joss.02823

Functions as a Service natural habitat: Cloud

- Cloud service providers give an excellent Functions as a Service (FaaS) platform that can scale elastically
- Example: Running `pyhf` across 25 worker nodes on Google Cloud Platform
  - Results being plotted as they are streamed back
  - Fit of all signal model hypotheses in analysis takes **3 minutes**!
- Powerful resource, but in (academic) sciences experience is still growing
- "Pay for priority" model
  - fast and reliable
  - requires funding even with nice support from cloud providers



(GIF sped up by 8x)

(Fitting) FaaS with `pyhf` on HPCs

- HPC facilities are more **commonly available** for use in HEP and provide an opportunity to **efficiently perform statistical inference** of LHC data
- Can pose problems with orchestration and efficient scheduling
- Want to leverage `pyhf` hardware accelerated backends at HPC sites for real analysis speedup
    - Reduce fitting time from hours to minutes
- Idea: Deploy a `pyhf` based **(fitting) Function as a Service** to HPC centers
- Example use cases:
    - Large scale ensemble fits for statistical combinations
    - Large dimensional scans of theory parameter space (e.g. Phenomenological Minimal Supersymmetric Standard Model scans)
    - Pseudo-experiment generation ("toys")

```
$ nvidia-smi --list-gpus | awk 'NF{NF-=2};1'
GPU 0: GeForce RTX 2080 Ti
$ cat benchmarks/gpu/gpu_jax.txt
# time pyhf cls --backend jax HVTWZ_3500.json

{
    "CLs_exp": [
        0.07675154647551732,
        0.17259685242090003,
        0.3571957128757839,
        0.6318389054097654,
        0.8797833319522873
    ],
    "CLs_obs": 0.25668814241306653
}

real    0m53.790s
user    0m59.982s
sys     0m4.725s
```

Model that takes over an hour with traditional C++ framework fit in under 1 minute with pyhf on local GPU

funcX: high-performance FaaS platform

- Designed to orchestrate **scientific workloads** across **heterogeneous computing resources** (clusters, clouds, and supercomputers) and task execution providers (HTCondor, Slurm, Torque, and Kubernetes)
- Leverages Parsl parallel scripting library for efficient parallelism and managing concurrent task execution
- Allows users to **register** and then **execute** Python functions in "serverless supercomputing" workflow
- `funcX` SDK provides a Python API to `funcX` service
- Controlling "endpoint" and submission machine can be **totally seperate** (communications routed through Globus)
    - Allows for "fire and forget" remote execution where you can run on your laptop, close it, and then retrieve output later
- Tool in a growing ecosystem of distributed computing
    - Currently looking into other tools like `Dask.distributed` and `Dask-Jobqueue` as well

funcX endpoints deployment

- Deployment of `funcX` endpoints is straightforward
- Example: Deployment to University of Chicago's RIVER cluster with Kubernetes

```
(venv) $ git clone git@github.com:matthewfeickert/talk-scipy-2021.git && cd talk-scipy-2021  # This talk as example
(venv) $ python -m pip install -r core-requirements.txt  # Install funcx and runtime dependencies
(venv) $ funcx-endpoint configure pyhf  # Generate endpoint and authenticate with Globus
(venv) $ cp funcx/river-config.py ~/.funcx/pyhf/config.py  # Move our custom endpoint config into place
(venv) $ funcx-endpoint start pyhf  # Startup endpoint...
(venv) $ funcx-endpoint list  # ...and good to go!
YYYY-MM-DD HH:MM:SS endpoint.endpoint_manager:173 [INFO]  Starting endpoint with uuid: 12345678-abcd-abcd-abcd-123456789101
YYYY-MM-DD HH:MM:SS endpoint.endpoint_manager:238 [INFO]  Launching endpoint daemon process
+---------------+---------+--------------------------------------+
| Endpoint Name | Status  |             Endpoint ID              |
+===============+=========+======================================+
| pyhf          | Running | 12345678-abcd-abcd-abcd-123456789101 |
+---------------+---------+--------------------------------------+
```

funcX endpoints on HPC

- funcX endpoint: logical entity that represents a compute resource
- Managed by an agent process allowing the funcX service to dispatch **user defined functions** to resources for execution
- Agent handles:
    - Authentication (Globus) and authorization
    - Provisioning of nodes on the compute resource
    - Monitoring and management
- Through funcX endpoint config can use expert knowledge of resource to optimize for task

```
from funcx_endpoint.endpoint.utils.config import Config
from funcx_endpoint.executors import HighThroughputExecutor
from funcx_endpoint.providers.kubernetes.kube import KubernetesProvider
from funcx_endpoint.strategies import KubeSimpleStrategy
from parsl.addresses import address_by_route

config = Config(
    executors=[
        HighThroughputExecutor(
            max_workers_per_node=1,
            address=address_by_route(),
            strategy=KubeSimpleStrategy(max_idletime=3600),
            container_type="docker",
            scheduler_mode="hard",
            provider=KubernetesProvider(
                init_blocks=0,
                min_blocks=1,
                max_blocks=100,
                init_cpu=2,
                max_cpu=3,
                init_mem="2000Mi",
                max_mem="4600Mi",
                image="bengal1/pyhf-funcx:3.8.0.3.0-1",
                worker_init="pip freeze",
                namespace="servicex",
                incluster_config=True,
            ),
        )
    ],
    # ...
)
```

Execution with funcX: Define user functions

```python
import json
from time import sleep

import pyhf
from funcx.sdk.client import FuncXClient
from pyhf.contrib.utils import download


def prepare_workspace(data, backend):
    import pyhf

    pyhf.set_backend(backend)
    return pyhf.Workspace(data)


def infer_hypotest(workspace, metadata, patches, backend):
    import time
    import pyhf

    pyhf.set_backend(backend)

    tick = time.time()
    model = workspace.model(...)
    data = workspace.data(model)
    test_poi = 1.0
    return {
        "metadata": metadata,
        "cls_obs": float(
            pyhf.infer.hypotest(test_poi, data, model, test_stat="qtilde")
        ),
        "fit-time": time.time() - tick,
    }

...
```

- As the analyst user, define the functions that you want the funcX endpoint to execute
- These are run as individual jobs and so require all dependencies of the function to **be defined inside the function**

```python
import numpy  # Not in execution scope

def example_function():
    import pyhf  # Import here

    ...

    pyhf.set_backend("jax")  # To use here
```

Execution with funcX: Register and run functions

```python
...
def main(args):

    ...

    # Initialize funcX client
    fxc = FuncXClient()
    fxc.max_requests = 200

    with open("endpoint_id.txt") as endpoint_file:
        pyhf_endpoint = str(endpoint_file.read().rstrip())

    # register functions
    prepare_func = fxc.register_function(prepare_workspace)

    # execute background only workspace
    bkgonly_workspace = json.load(bkgonly_json)
    prepare_task = fxc.run(
        bkgonly_workspace, backend, endpoint_id=pyhf_endpoint, function_id=prepare_func
    )

    # retrieve function execution output
    workspace = None
    while not workspace:
        try:
            workspace = fxc.get_result(prepare_task)
        except Exception as excep:
            print(f"prepare: {excep}")
            sleep(10)
...
```

- With the user functions defined, they can then be registered with the funcX client locally
  - `fx.register_function(...)`
- The local funcX client can then execute the request to the remote funcX endpoint, handling all communication and authentication required
  - `fx.run(...)`
- While the jobs run on the remote HPC system, can make periodic requests for finished results
  - `fxc.get_result(...)`
  - Returning the output of the user defined functions

Execution with funcX: Scaling out jobs

```
...

    # register functions
    infer_func = fxc.register_function(infer_hypotest)

    patchset = pyhf.PatchSet(json.load(patchset_json))

    # execute patch fits across workers and retrieve them when done
    n_patches = len(patchset.patches)
    tasks = {}
    for patch_idx in range(n_patches):
        patch = patchset.patches[patch_idx]
        task_id = fxc.run(
            workspace,
            patch.metadata,
            [patch.patch],
            backend,
            endpoint_id=pyhf_endpoint,
            function_id=infer_func,
        )
        tasks[patch.name] = {"id": task_id, "result": None}

    while count_complete(tasks.values()) < n_patches:
        for task in tasks.keys():
            if not tasks[task]["result"]:
                try:
                    result = fxc.get_result(tasks[task]["id"])
                    tasks[task]["result"] = result
                except Exception as excep:
                    print(f"inference: {excep}")
                    sleep(15)
...
```

- The workflow
  - `fx.register_function(...)`
  - `fx.run(...)`

  can now be used to scale out **as many custom functions as the workers can handle**
- This allows for all the signal patches (model hypotheses) in a full analysis to be **run simultaneously across HPC workers**
  - Run from anywhere (e.g. laptop)!
- The user analyst has **written only simple pure Python**
  - No system specific configuration files needed

Scaling of statistical inference

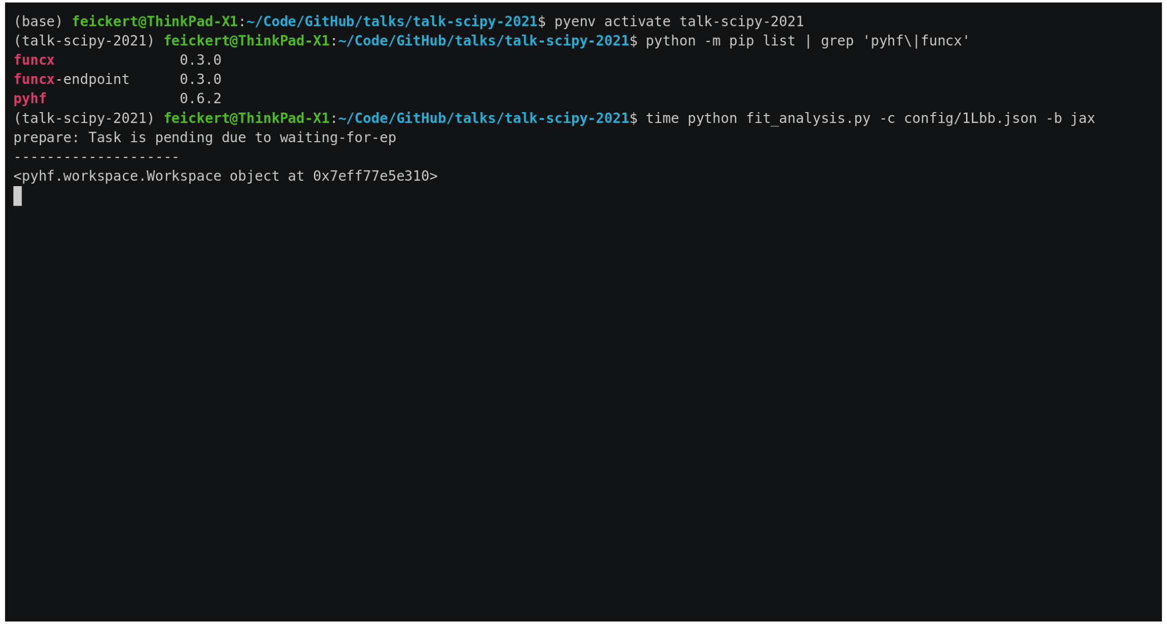
- **Example**: Fitting all 125 models from `pyhf` pallet for published ATLAS SUSY 1Lbb analysis
  - DOI: https://doi.org/10.17182/hepdata.90607
- Wall time **under 2 minutes 30 seconds**
  - Downloading of `pyhf` pallet from HEPData (submit machine)
  - Registering functions (submit machine)
  - Sending serialization to funcX endpoint (remote HPC)
  - funcX executing all jobs (remote HPC)
  - funcX retrieving finished job output (submit machine)
- Deployments of funcX endpoints currently used for testing
  - University of Chicago River HPC cluster (CPU)
  - NCSA Bluewaters (CPU)
  - XSEDE Expanse (GPU JAX)

```
feickert@ThinkPad-X1:~$ time python fit_analysis.py -c config/1Lbb.json
prepare: waiting-for-ep
prepare: waiting-for-ep
--------------------
<pyhf.workspace.Workspace object at 0x7fb4cfe614f0>
Task C1N2_Wh_hbb_1000_0 complete, there are 1 results now
Task C1N2_Wh_hbb_1000_100 complete, there are 2 results now
Task C1N2_Wh_hbb_1000_150 complete, there are 3 results now
Task C1N2_Wh_hbb_1000_200 complete, there are 4 results now
Task C1N2_Wh_hbb_1000_250 complete, there are 5 results now
Task C1N2_Wh_hbb_1000_300 complete, there are 6 results now
Task C1N2_Wh_hbb_1000_350 complete, there are 7 results now
Task C1N2_Wh_hbb_1000_400 complete, there are 8 results now
Task C1N2_Wh_hbb_1000_50 complete, there are 9 results now
Task C1N2_Wh_hbb_150_0 complete, there are 10 results now
...
Task C1N2_Wh_hbb_900_150 complete, there are 119 results now
Task C1N2_Wh_hbb_900_200 complete, there are 120 results now
inference: waiting-for-ep
Task C1N2_Wh_hbb_900_300 complete, there are 121 results now
Task C1N2_Wh_hbb_900_350 complete, there are 122 results now
Task C1N2_Wh_hbb_900_400 complete, there are 123 results now
Task C1N2_Wh_hbb_900_50 complete, there are 124 results now
Task C1N2_Wh_hbb_900_250 complete, there are 125 results now
--------------------
...

real	2m17.509s
user	0m6.465s
sys	0m1.561s
```

Scaling of statistical inference

- **Example**: Fitting all 125 models from `pyhf` pallet for published ATLAS SUSY 1Lbb analysis
  - DOI: https://doi.org/10.17182/hepdata.90607
- Wall time **under 2 minutes 30 seconds**
  - Downloading of `pyhf` pallet from HEPData (submit machine)
  - Registering functions (submit machine)
  - Sending serialization to funcX endpoint (remote HPC)
  - funcX executing all jobs (remote HPC)
  - funcX retrieving finished job output (submit machine)
- Deployments of funcX endpoints currently used for testing
  - University of Chicago River HPC cluster (CPU)
  - NCSA Bluewaters (CPU)
  - XSEDE Expanse (GPU JAX)



Click me to watch an asciinema!

Scaling of statistical inference: Results

- Remember, the returned output is just the **function's return**
- Our hypothesis test user function from earlier:

```python
def infer_hypotest(workspace, metadata, patches, backend):
    import time
    import pyhf

    pyhf.set_backend(backend)

    tick = time.time()
    model = workspace.model(...)
    data = workspace.data(model)
    test_poi = 1.0
    return {
        "metadata": metadata,
        "cls_obs": float(
            pyhf.infer.hypotest(
                test_poi, data, model, test_stat="qtilde"
                )
        ),
        "fit-time": time.time() - tick,
    }
```

- Allowing for easy and rapid serialization and manipulation of results
- Time from submitting jobs to plot can be minutes

```
feickert@ThinkPad-X1:~$ python fit_analysis.py -c config/1Lbb.json
# Returned results saved in results.json
feickert@ThinkPad-X1:~$ jq .C1N2_Wh_hbb_1000_0.result results.json

{
  "metadata": {
    "name": "C1N2_Wh_hbb_1000_0",
    "values": [
      1000,
      0
    ]
  },
  "cls_obs": 0.5856783708143126,
  "fit-time": 24.032342672348022
}

feickert@ThinkPad-X1:~$ jq .C1N2_Wh_hbb_1000_0.result.cls_obs results.json
0.5856783708143126
```

FasS constraints and trade-offs

- The nature of FaaS that makes it highly scalable also leads to a problem for taking advantage of just-in-time (JIT) compiled functions
  - JIT is super helpful for performing pseudo-experiment generation
- To leverage JITed functions there needs to be **memory that is preserved across invocations** of that function
- FaaS: Each function call is self contained and **doesn't know about global state**
  - funcX endpoint listens on a queue and invokes functions
- Still need to know and tune funcX config to specifics of endpoint resource
  - No magic bullet when using HPC center batch

```
In [1]: import jax.numpy as jnp
   ...: from jax import jit, random

In [2]: def selu(x, alpha=1.67, lmbda=1.05):

   ...:     return lmbda * jnp.where(x > 0, x, alpha * jnp.exp(x) - alpha)
   ...:

In [3]: key = random.PRNGKey(0)

   ...: x = random.normal(key, (1000000,))

In [4]: %timeit selu(x)

850 µs ± 35.4 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

In [5]: selu_jit = jit(selu)


In [6]: %timeit selu_jit(x)
17.2 µs ± 105 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

50X speedup from JIT

Summary

- Through the combined use of the pure-Python libraries **funcX and `pyhf`**, demonstrated the ability to **parallelize and accelerate** statistical inference of physics analyses on HPC systems through a **(fitting) FaaS solution**
- Without having to write any bespoke batch jobs, inference can be registered and executed by analysts with a client Python API that still **achieves the large performance gains** compared to single node execution that is a typical motivation of use of batch systems.
- Allows for transparently switching workflows between **provider systems** and from **CPU to GPU** environments
- Not currently able to leverage benefits of **JITed operations**
  - Looking for ways to bridge this
- All code used **public and open source** on GitHub!

Thanks for listening!

Come talk with us!

www.scikit-hep.org/pyhf

Backup

funcX endpoints on HPC: Config Example

Example Parsl `HighThroughputExecutor` config (from Parsl docs) that **funcX extends**

```
from parsl.config import Config
from libsubmit.providers.local.local import Local
from parsl.executors import HighThroughputExecutor

config = Config(
    executors=[
        HighThroughputExecutor(
            label='local_htex',
            workers_per_node=2,
            provider=Local(
                min_blocks=1,
                init_blocks=1,
                max_blocks=2,
                nodes_per_block=1,
                parallelism=0.5
            )
        )
    ]
)
```

- block: Basic unit of resources acquired from a provider
- `max_blocks`: Maximum number of blocks that can be active per executor
- `nodes_per_block`: Number of nodes requested per block
- `parallelism`: Ratio of task execution capacity to the sum of running tasks and available tasks

- 9 tasks to compute
- Tasks are allocated to the first block until its `task_capacity` (here 4 tasks) reached
- Task 5: First block full and `5/9 > parallelism` so Parsl provisions a new block for executing the remaining tasks

View of fitting FaaS Analysis Facility Blueprint

FaaS Team

Development

pyhf evolves over time. Code on GitHub released to PyPI and conda-forge.

New pyhf computations that may be interesting to expose.

GitHub PyPI conda-forge

Building

FuncX encapsulation of Python functions.

Images are published to a cloud registry (DockerHub?), so they can be accessed.

Docker Cloud Registry

Deploying

Kubernetes is used to deploy the functions.

High scalability plays nicely with computational expensive workflows.

Kubernetes

v1
v1.1
v2

Governance

Governance model required.

Someone needs to coordinate new deployments across the stack.

In addition to enable / disable access through an auth DB.

Auth database

Continuous effort

End users

Ask for access

Access request to the service.

Given the amount of computing power the service could use, auth is required.

Some ticketing procedure must be defined (GitHub issues?).

Access request

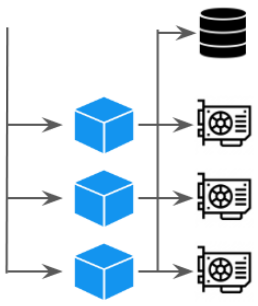
Continuous effort

Fit

Users send **HTTP** requests.

Users query the service, with some basic auth information.

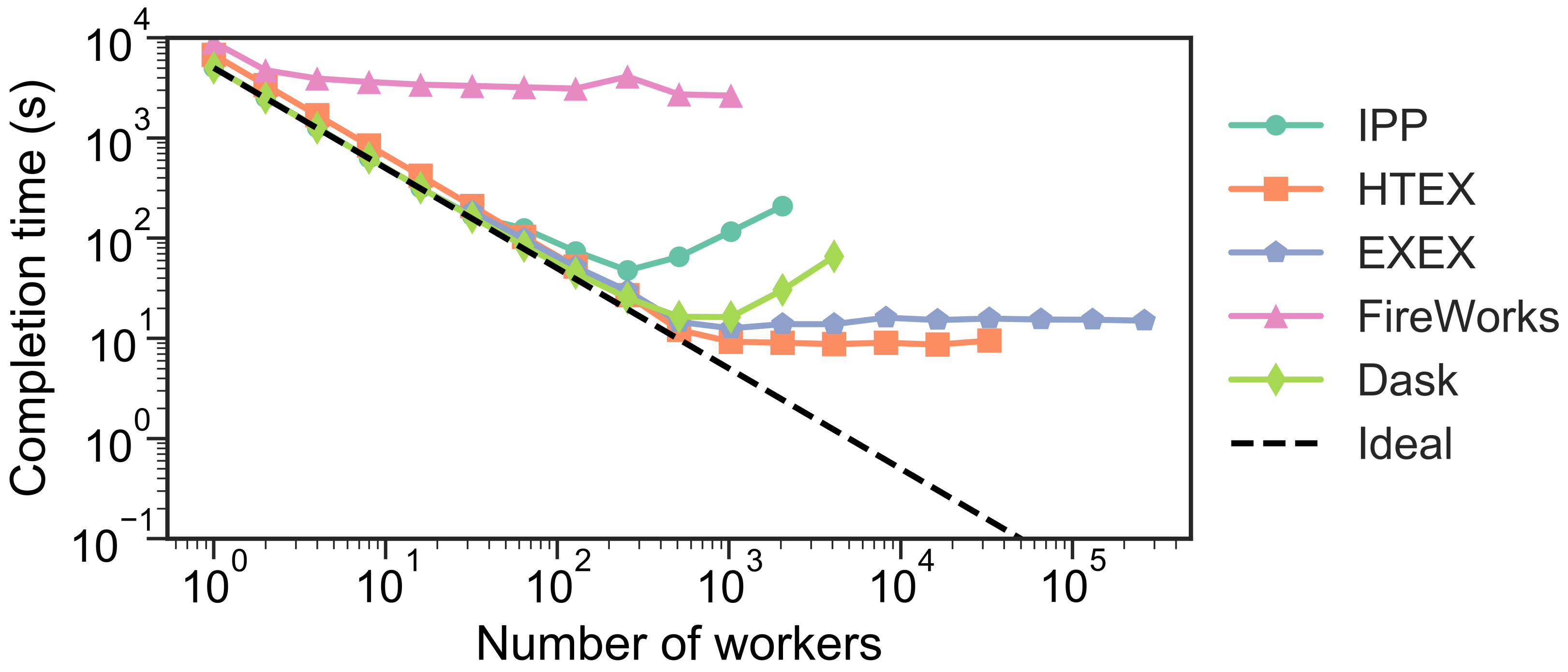
Service validates user auth before proceeding forward.



Pods GPUs

Why look at funcX when Dask is so established?

- funcX provides a managed service secured by Globus Auth
- Endpoints can be set up by a site administrator and shared with authorized users through Globus Auth Groups
- Testing has shown that Dask may not scale well **to thousands of nodes**, whereas the funcX High Throughput Executor (HTEX) provided through Parsl scales efficiently



References

1. Lukas Heinrich, *Distributed Gradients for Differentiable Analysis*, Future Analysis Systems and Facilities Workshop, 2020.
2. Babuji, Y., Woodard, A., Li, Z., Katz, D. S., Clifford, B., Kumar, R., Lacinski, L., Chard, R., Wozniak, J., Foster, I., Wilde, M., and Chard, K., Parsl: Pervasive Parallel Programming in Python. 28th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC). 2019. https://doi.org/10.1145/3307681.3325400

The end.