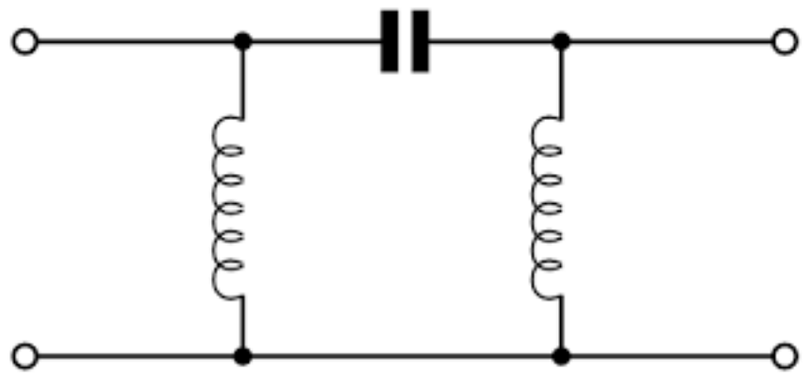
Fine-Grained HEP Analysis Task Graph Optimization with Coffea and Dask

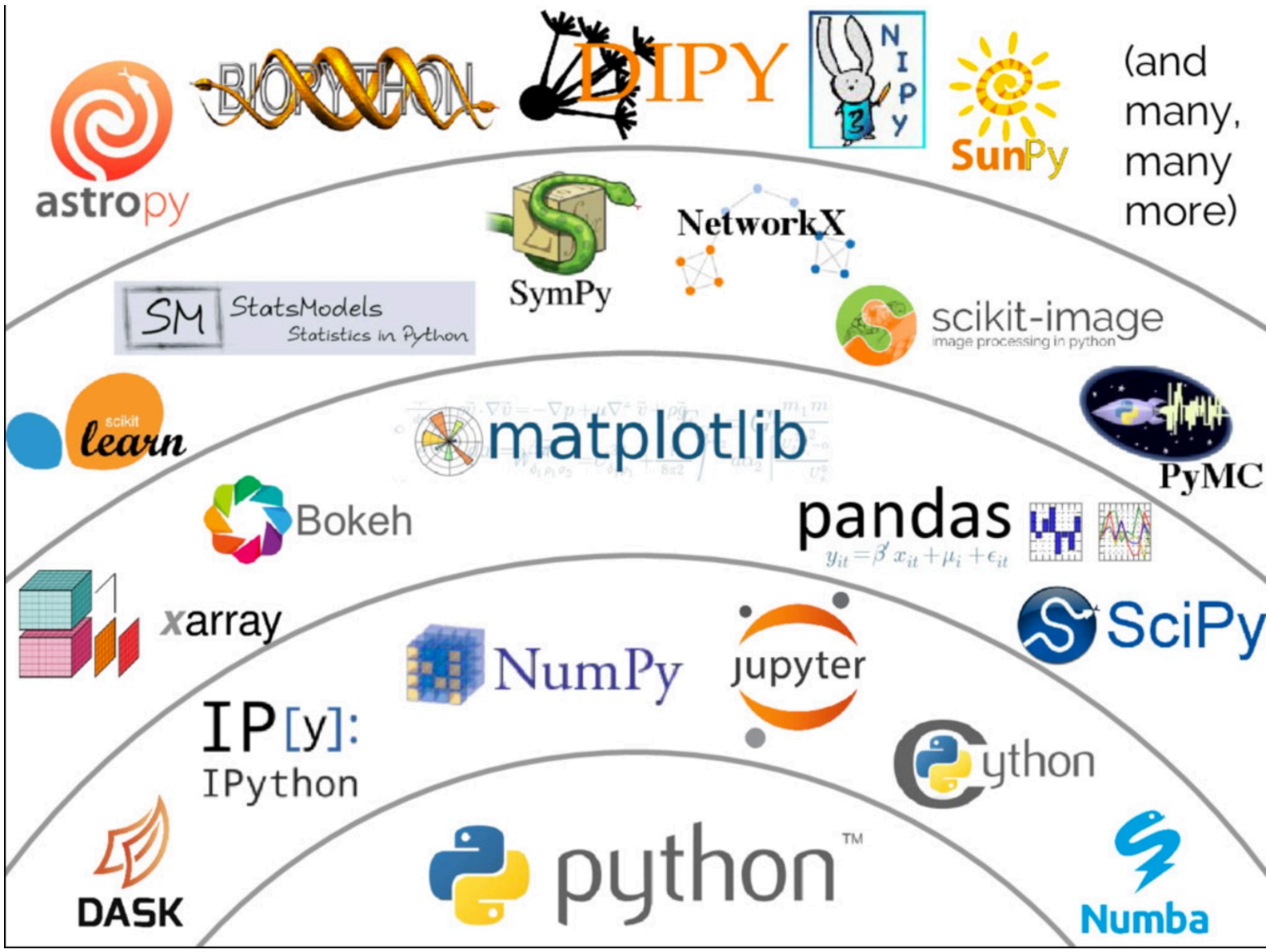
Lindsey Gray, Nick Smith (FNAL),
Doug Davis, Martin Durant (Anaconda),
Angus Hollands, Jim Pivarski (Princeton),
Yi-Mu Chen (UMD),
CHEP 2023 - Norfolk, Virginia
9 May 2023

Impedance Mismatches

- ROOT File <-> Machine Learning (uproot is everywhere nowadays)
- Big data <-> PyROOT (python for-loops are slow)
- HEP Physicist <-> Industry (we are a subset of wider data science)



Scientific Python



Fermilab

Coffea is

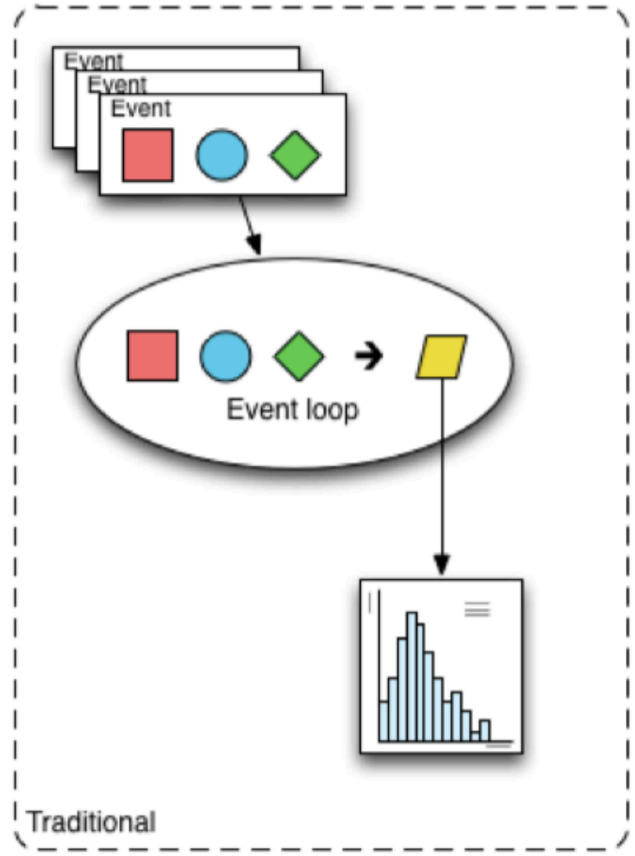
- A package in the scientific python ecosystem
  - `$ pip install coffea`
- A user interface for columnar analysis
  - With missing pieces of the stack filled in
- A minimum viable product
  - We are data analyzers too #dogfooding
- A really strong glue

- Going strong for five years
  - Many published analyses now

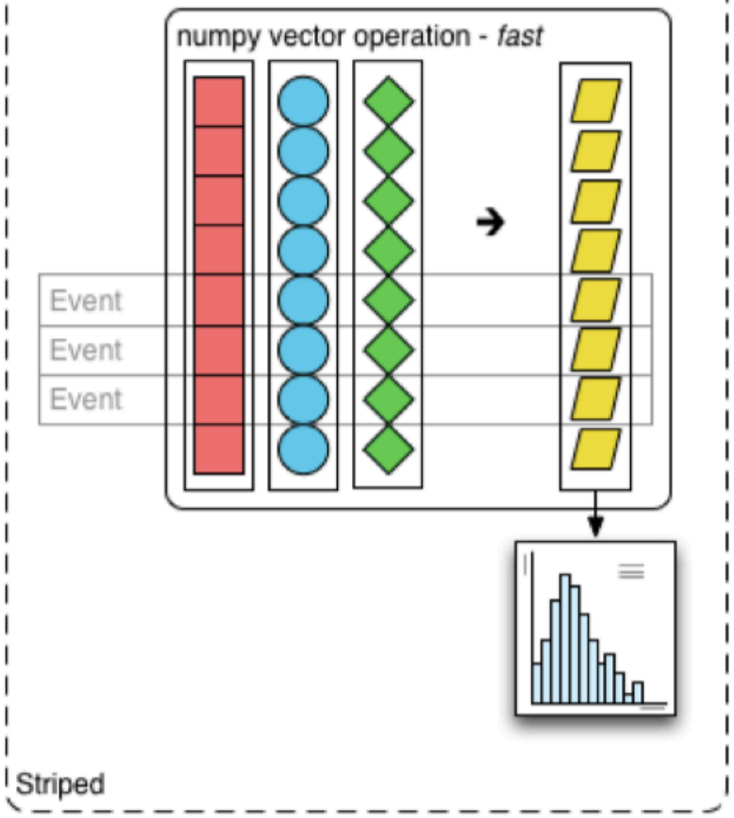
Fermilab

What is columnar analysis?

- Event loop analysis:
  - Load relevant values for a specific event into local variables
  - Evaluate several expressions
  - Store derived values
  - Repeat (explicit outer loop)



- Columnar analysis:
  - Load relevant values for many events into contiguous arrays
  - Evaluate several **array programming** expressions
    - Implicit *inner* loops
    - Plan analysis by composing data manipulations
  - Store derived values



Fermilab

Concrete example:

```
void MyClass::Loop() {
  size_t nEvents;
  // load...

  for (Long64_t iEvent=0; iEvent<nEvents; iEvent++) {
    double MET_pt;
    int nElectron;
    double * Electron_pt;
    double * Electron_eta;
    // load...

    if ( MET_pt > 100. ) continue;

    for(size_t iEl=0; iEl<nElectron; ++iEl) {
      if ( Electron_pt[iEl] > 30. ) {
        hist->Fill(Electron_eta[iEl]);
      }
    }
  }
}
```

Event loop

```
cut = (events.MET.pt < 100.) & (events.Electron.pt > 30.)
hist.fill(eta=events.Electron.eta[cut].flatten())
```

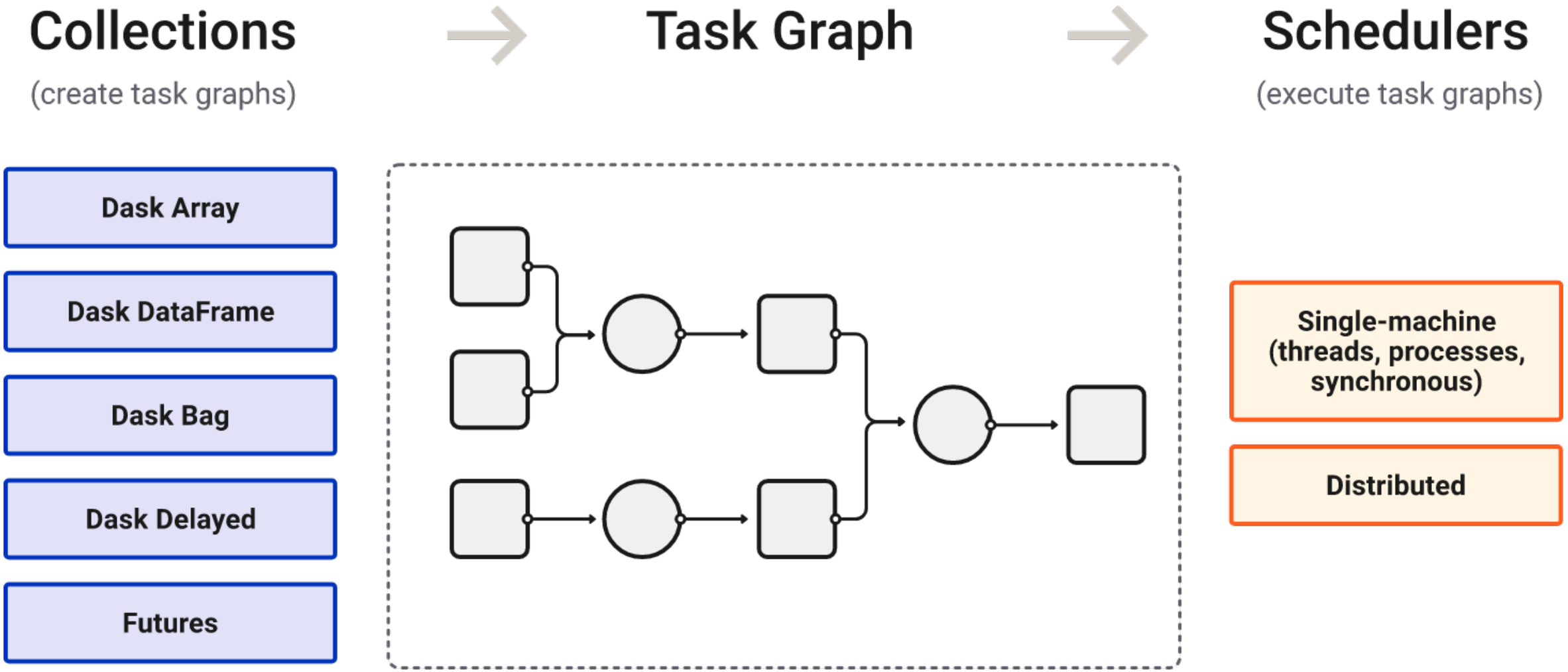
Columnar

This talk:

```
# “array” operations only describe what is to be done
cut = (events.MET.pt < 100.) & (events.Electron.pt > 30.)
hist.fill(eta=events.Electron.eta[cut].flatten())
# in order to render a result, we ask for it
hist.compute()
```

Delayed Columnar

Dask



- Dask provides an interface for specifying/locating input data and then describing manipulations on that data are organized into a task graph
  - This task graph can then be executed on local compute or on a cluster
- Dask Array and Dask Dataframe deal well with rectangular data
  - Provide a scalable interface to describe manipulations of data that may not fit into system memory by mapping transformations onto partitions of the data that fit in memory

awkward array 2.0, dask_awkward, dask_histogram, and coffea

Coffea 0.7

awkward-array
hist
dask, parsl, etc.

Coffea 2023 (yes, we switched to CalVer)

dask_awkward(
    awkward-array
)
hist(dask_histogram)
dask, parsl, etc.

- Awkward array 2.0 features an improved and streamlined backend
  - Only C and python, no C++ metadata handling
  - Removal of ak.virtual delayed computations (to be replaced by dask_awkward)
- dask_awkward and dask_histogram bring delayed, distributed computation to awkward array 2.0 based analyses and libraries
  - Providing access to dask at all layers of analysis yields improved parallelism and better factorization away from compute infrastructure
- Coffea (particularly nanoevents) was almost entirely based on ak.virtual

Practicalities: Writing Code (1)

- Minimal boiler plate to enter delayed, out-of-core computing environment
- Nanoevents interface is the same as with awkward1
  - Arrays from flat input file are organized into physics object concepts
  - Only major difference is now when you want something computed you .compute() it
    - cf. dask.persist() - no time in this talk, it is a whole can of worms, see extras / chat over coffee!
- Largely user needs to change “ak.action” to “dak.action”

```
import dask
import dask_awkward as dak
import hist
import hist.dask as hda
import numpy as np

from coffea import processor
from coffea.nanoevents import NanoEventsFactory

import matplotlib.pyplot as plt

from distributed import Client
client=Client()

# The opendata files are non-standard NanoAOD, so some optional data columns are missing
processor.NanoAODSchema.warn_missing_crossrefs = False

events = NanoEventsFactory.from_root(
    "file:/Users/lgray/coffea-dev/coffea/Run2012B_SingleMu.root",
    treepath="Events",
    chunks_per_file=500,
    permit_dask=True,
    metadata={"dataset": "SingleMu"}
).events()
```

dask_histogram + hist

local dask-distributed cluster (can omit, or extend to condor)

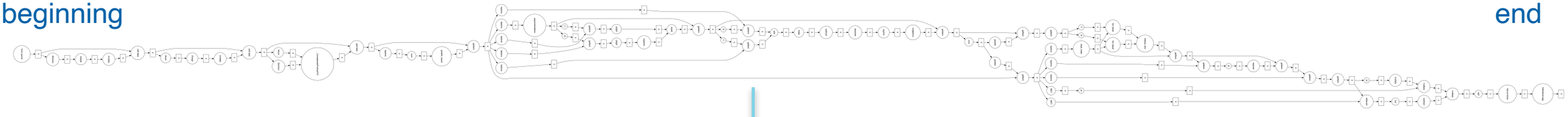
Practicalities: Writing Code (2)

- Example: Query 8
  - from ADL Benchmarks
- Compare to coffea 0.7
  - No need for processor
    - provide facade for backwards compatibility
  - Minimal boilerplate at analysis code
  - Similar interface as coffea 0.7 but with different baseline packages
  - Use dask to dispatch compute
- Similarity of interface hides massive implementation difference
  - H/T to dask_awkward authors for helping to make that happen!
  - Similarity of interface can help encourage adoption in analyses

```
events["Electron", "pdgId"] = -11 * events.Electron.charge
events["Muon", "pdgId"] = -13 * events.Muon.charge
events["leptons"] = dak.concatenate(
    [events.Electron, events.Muon],
    axis=1,
)
events = events[dak.num(events.leptons) >= 3]
pair = dak.argcombinations(events.leptons, 2, fields=["l1", "l2"])
pair = pair[(events.leptons[pair.l1].pdgId == -events.leptons[pair.l2].pdgId)]
x = events.leptons[pair.l1] + events.leptons[pair.l2]

pair = pair[
    dak.singletons(
        dak.argmin(
            abs(
                (events.leptons[pair.l1] + events.leptons[pair.l2]).mass
                - 91.2
            ),
            axis=1,
        )
    )
]
events = events[dak.num(pair) > 0]
pair = pair[dak.num(pair) > 0][:, 0]

l3 = dak.local_index(events.leptons)
l3 = l3[(l3 != pair.l1) & (l3 != pair.l2)]
l3 = l3[dak.argmax(events.leptons[l3].pt, axis=1, keepdims=True)]
l3 = events.leptons[l3][:, 0]

mt = np.sqrt(2 * l3.pt * events.MET.pt * (1 - np.cos(events.MET.delta_phi(l3))))
q8_hist = (
    hda.Hist.new.Reg(
        100, 0, 200, name="mt", label="$\ell$-MET transverse mass [GeV]"
    )
    .Double()
    .fill(mt)
)

q8_hist.compute().plot1d()
```

Optimization Example: Q8

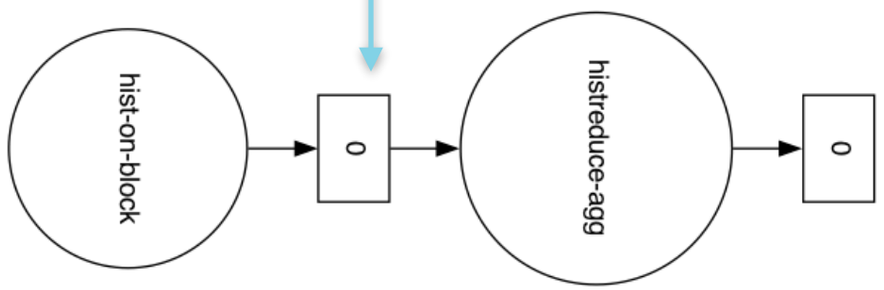
query
beginning



query
end

dask.optimize(q8_hist)

query
beginning



query
end

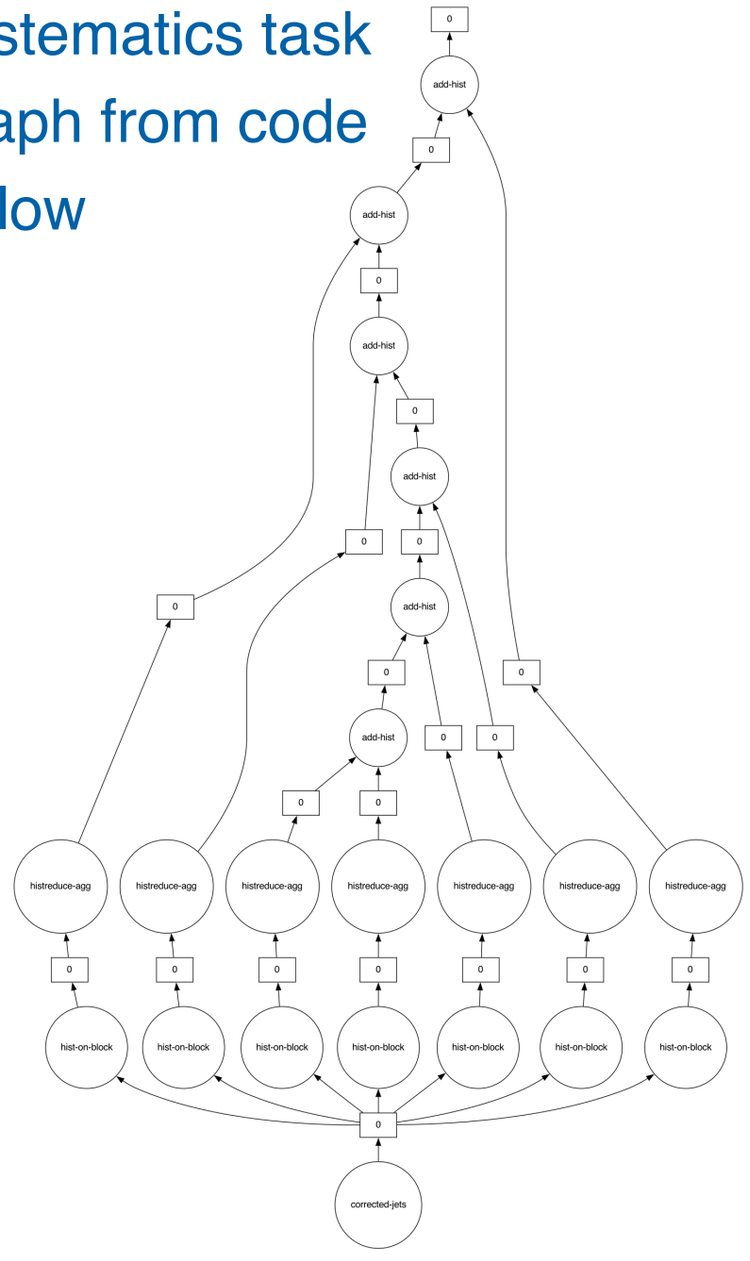
- Raw HEP analysis task graphs get large quickly
  - Reasonably complete analysis, full systematics, is ~7000 layers as written by the user
  - Q8 (top) here is 78 layers
  - Each task-graph node could be executed on a different cluster resource (data transfer!)
- Dask provides standard optimizers to minimize node multiplicity
  - This minimizes data transfer overhead and task-spawning overhead
  - These optimizations are applied by default, yielding 2 layers for Q8
  - Reasonably complete analysis is 234 layers post-optimization (ops fuse to hist filling)

Fermilab

Practicalities: Writing Code (3)

- Systematics are one of the most critical aspects of HEP analysis development
  - Without systematics we cannot do our science
  - Performing critical tasks in code should be clear and intuitive
- In coffea 2023, distributed, parallel systematics loops are written as loops over systematic variations
  - Successive dask_histogram fill calls can be distributed across nodes and resulting sub-histograms aggregated

Shortened example of systematics task graph from code below

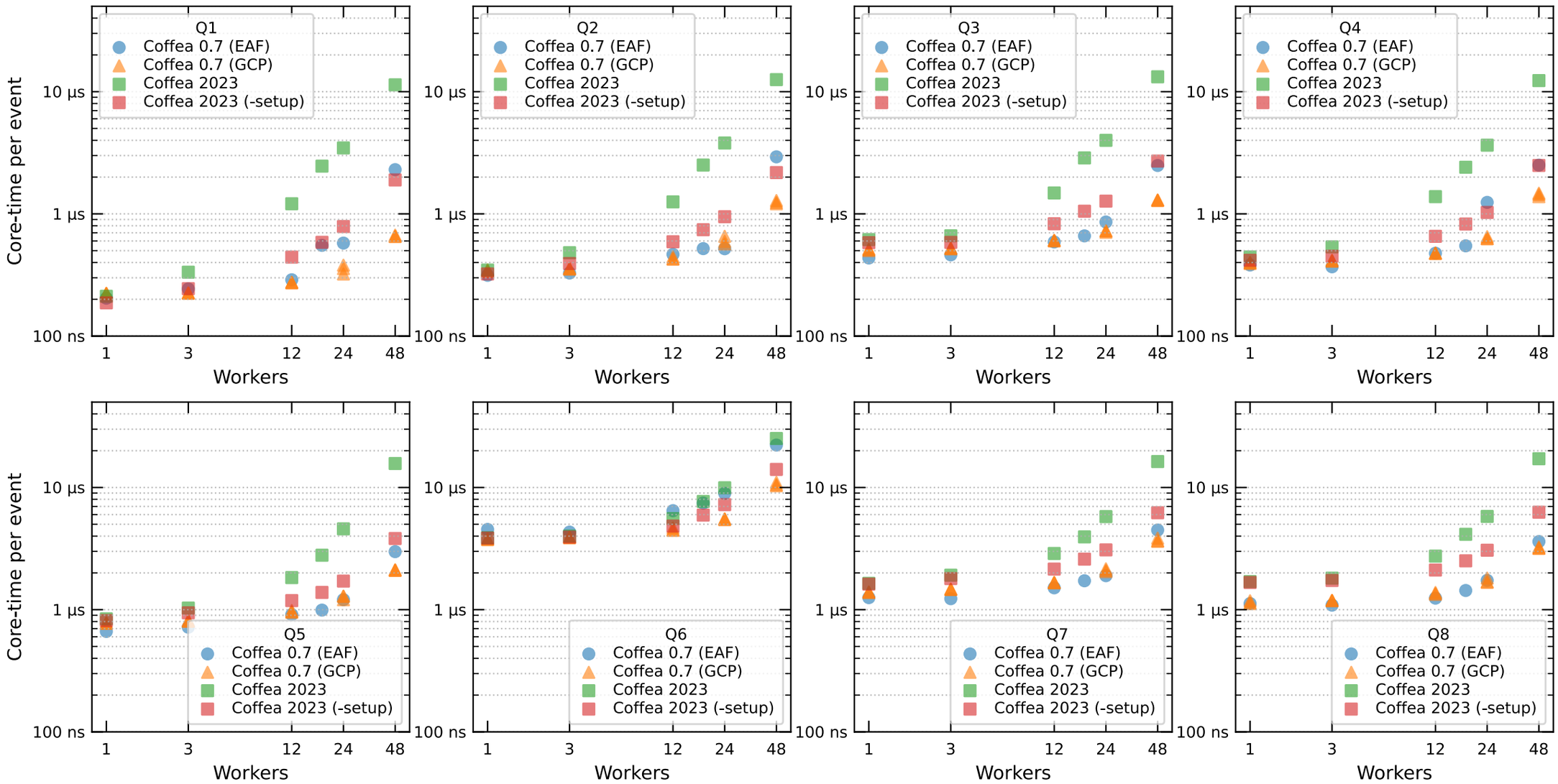



```
dahist = hda.Hist(
    hist.axis.StrCategory([], growth=True, name="systematic"),
    hist.axis.Regular(40, 0, 400, name="pt"),
    storage=hist.storage.Weight(),
)

deepjet_sf = evaluator["deepJet_comb"]
central_weight = deepjet_sf("central", "M", 5, abs(corrected_jets.eta), corrected_jets.pt)
dahist.fill("central", dak.flatten(corrected_jets.pt), weight=dak.flatten(central_weight))

for unc in jet_factory.uncertainties():
    up_weight = deepjet_sf("central", "M", 5, abs(corrected_jets[unc].up.eta), corrected_jets[unc].up.pt)
    down_weight = deepjet_sf("central", "M", 5, abs(corrected_jets[unc].down.eta), corrected_jets[unc].down.pt)
    dahist.fill(systematic=f"{unc}_up", pt=dak.flatten(corrected_jets[unc].up.pt), weight=dak.flatten(up_weight))
    dahist.fill(systematic=f"{unc}_down", pt=dak.flatten(corrected_jets[unc].down.pt), weight=dak.flatten(down_weight))
```

dask-wrapped correctionlib

Fermilab

Benchmark Results comparing to coffea 0.7 / ak1



- New benchmarks using whole-node at FNAL Elastic Analysis Facility (EAF)
  - Confirm no performance degradation compared to coffea 0.7 (further improvement coming)
  - “Setup time” dominated by spinning up full dask worker nodes (subtract off benchmark)
- Graph and column optimization still included in “Coffea 2023 (-setup)”
  - Column optimization runs mock task graph in local single thread

Fermilab

Further Thoughts to Consider

- dask_awkward fundamentally changes how we can describe analysis
- dask_awkward-based analyses, via dask task graphs, are rendered into a general, complete, declarative analysis description language (ADL)
  - It looks curiously reminiscent of lisp, but no one would want to write by hand
  - Luckily, using dask writes it for us so we can reap the advantages
- This means we have a preservable, extensible, and generalized description of HEP analysis code that we can overlay on arbitrary compute resources
  - “achievement unlocked”
- dask_awkward can robustly predict data requirements **without** full execution
  - Using only file metadata, without altering user code (aside from initial adoption)
  - This alone radically changes our ability to optimize compute systems
  - Named data networks, interfaces with network transfer schedulers, can be hidden from users of analysis facilities - enormous potential for system-level optimization
- dask_awkward can make skims in the process of the complete data analysis
  - See extras, skimming + dask.persist() stand to wildly alter analysis data lifecycles and multi-user interaction
- Multiple task scheduling projects are moving to dask task graphs (portability!)

Conclusions and Next Steps

- coffea is in its release candidate phase for coffea 2023
  - full migration to using awkward 2.0 and dask for delayed, out-of-core computation
    - realized by using dask_awkward and dask_histogram
    - no major performance degradation seen so far, with improvements in the pipeline in awkward
  - we also plan to include more user analysis tools (see extras)
    - wrappers for machine learning inference as dask tasks (including Nvidia triton)
    - automatic cutflow and N-1 plot generation as an extension to PackedSelection
  - aim for a complete, robust release this summer or early fall
    - `pip install --pre coffea --upgrade` if you want to try it out now! (works on arm too)

- This update represents the culmination of ~4 years of R&D, in addition to maintaining successful deployment, and supporting analyses
  - The changes as a result of this research set scientific-python based analysis on a course for achieving extreme performance at scale in the busy distributed system of HEP production and analysis computing

- Congratulations to everyone involved in here - let's make some plots :-)

Extras

dask.persist (checkpointing)

riffing on ADL Q8:

```
events["Electron", "pdgId"] = -11 * events.Electron.charge
events["Muon", "pdgId"] = -13 * events.Muon.charge
events["leptons"] = dak.concatenate(
    [events.Electron, events.Muon],
    axis=1,
)
events = events[dak.num(events.leptons) >= 3]
pair = dak.argcombinations(events.leptons, 2, fields=["l1", "l2"])
pair = pair[(events.leptons[pair.l1].pdgId == -events.leptons[pair.l2].pdgId)]
pair = pair.persist()

pair = pair[
    dak.singletons(
        dak.argmin(
            abs(
                (events.leptons[pair.l1] + events.leptons[pair.l2]).mass
                - 91.2
            ),
            axis=1,
        )
    )
]
```

returns immediately, processing runs in background

Further calls and manipulations reference to-be persisted data

- Spawn background processing whose output can be referenced as a new array in the distributed cluster memory
- With local memory cache (given enough memory) can explore / iterate on data extremely quickly
- Resilient through node recycling (in a single dask cluster)
  - Perpetual dask clusters are not an anti-pattern

dak.to_parquet (skimming)

```
events["Electron", "pdgId"] = -11 * events.Electron.charge
events["Muon", "pdgId"] = -13 * events.Muon.charge
events["leptons"] = dak.concatenate(
    [events.Electron, events.Muon],
    axis=1,
)
events = events[dak.num(events.leptons) >= 3]
pair = dak.argcombinations(events.leptons, 2, fields=["l1", "l2"])
pair = pair[(events.leptons[pair.l1].pdgId == -events.leptons[pair.l2].pdgId)]
skim = dak.to_parquet(pair, f"/some/path/to/skim/area/{events.metadata['dataset']}")

pair = pair[
    dak.singletons(
        dak.argmin(
            abs(
                (events.leptons[pair.l1] + events.leptons[pair.l2]).mass
                - 91.2
            ),
            axis=1,
        )
    )
]
```

dask handle for delayed running of skim

Further calls run stepwise as normal. Skim runs in parallel.

- Then you `hists, _ = dask.compute(histograms, skim)`
- In specified directory you get a parquet dataset which you can start further analysis from or share with collaborators
  - ROOT output will happen in time, parquet for skims is functionally equivalent
- Combined with `dask.persist` allows interestingly fine-grained control of data lifecycle in analysis that we don't know best practices for :-)

more user analysis tools: cutflows and N-1 histograms

N-1 plots:

In [3]:
```
nminusone = selection.nminusone("twoElectron", "noMuon", "leadPt20")
labels, nev, masks = nminusone.result()
print(labels)
print(nev)
print(masks)
```

```
['initial', 'N - twoElectron', 'N - noMuon', 'N - leadPt20', 'N']
[40, 10, 3, 5, 3]
[array([False,  True,  True, False, False, False, False, False, False,
        True, False, False, False, False, False,  True,  True, False,
       False, False,  True,  True, False, False, False, False, False,
        True, False,  True, False, False, False,  True, False, False,
       False, False, False, False]), array([False, False,  True, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False,  True,  True, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False]), array([False, False,  True,  True, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
        True, False,  True,  True, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False]), array([False, False,  True, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False,  True,  True, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False])]
```

cutflows:

In [5]:
```
cutflow = selection.cutflow("noMuon", "twoElectron", "leadPt20")
labels, nevonecut, nevcutflow, masksonecut, maskscutflow = cutflow.result()
print(labels)
print(nevonecut, nevcutflow)
print(masksonecut, maskscutflow)
```

```
['initial', 'noMuon', 'twoElectron', 'leadPt20']
[40, 28, 5, 17] [40, 28, 5, 3]
[array([ True,  True,  True,  True, False, False, False,  True,  True,
        True, False,  True,  True,  True, False,  True,  True,  True,
        True,  True,  True,  True,  True, False, False,  True, False,
        True, False,  True, False, False,  True,  True, False,  True,
        True,  True,  True,  True]), array([False, False,  True,  True, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
        True, False,  True,  True, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False]), array([False,  True,  True, False,  True,  True,  True, False, False,
        True, False, False, False, False, False,  True,  True, False,
       False, False,  True,  True, False,  True,  True, False,  True,
        True, False,  True, False,  True, False,  True, False, False,
       False, False, False, False])] [array([ True,  True,  True,  True, False, False, False,  True,  True,
        True, False,  True,  True,  True, False,  True,  True,  True,
        True,  True,  True,  True,  True, False, False,  True, False,
        True, False,  True, False, False,  True,  True, False,  True,
        True,  True,  True,  True]), array([False, False,  True,  True, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
        True, False,  True,  True, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False]), array([False, False,  True, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False,  True,  True, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False])]
```

- Work by Iason Krommydas (Rice) to automate essential early-analysis data exploration
  - N-1 plots and cutflow tables rendered as dask task graphs or eager arrays from book-keeping class “PackedSelection”
  - Expressive, easy-to-use extension to existing, adopted tools within coffea
- Solves an often requested, and otherwise home-grown, feature for coffea

more user analysis tools: common machine learning interface

nvidia triton

```python
# Defining custom wrapper function with awkward padding requirements.
class triton_wrapper_test(triton_wrapper):
    def awkward_to_numpy(self, output_list, jets):
        return [], {
            "output_list": output_list,
            "input_dict": common_awkward_to_numpy(jets),
        }

    def dask_columns(self, output_list, jets):
        return [
            jets.eta,
            jets.phi,
            jets.pfcands.pt,
            jets.pfcands.phi,
            jets.pfcands.eta,
            jets.pfcands.feat1,
            jets.pfcands.feat2,
        ]

# Running the evaluation in lazy and non-lazy forms
tw = triton_wrapper_test(
    model_url="triton+grpc://localhost:8001/pn_test/1",
    client_args=dict(
        ssl=False
    ),  # Solves SSL version mismatch for local inference server
)

ak_jets, dak_jets = prepare_jets_array(njets=256)

# Vanilla awkward arrays
ak_res = tw(["output"], ak_jets)
dak_res = tw(["output"], dak_jets)

for k in ak_res.keys():
    assert ak.all(ak_res[k] == dak_res[k].compute())
```

PyTorch

```python
class torch_wrapper_test(torch_wrapper):
    def awkward_to_numpy(self, jets):
        default = common_awkward_to_numpy(jets)
        return [], {
            "points": default["points"].astype(np.float32),
            "features": default["features"].astype(np.float32),
            "mask": default["mask"].astype(np.float16),
        }

    def dask_columns(self, jets):
        return [
            jets.eta,
            jets.phi,
            jets.pfcands.pt,
            jets.pfcands.phi,
            jets.pfcands.eta,
            jets.pfcands.feat1,
            jets.pfcands.feat2,
        ]

model = torch.jit.load("tests/samples/pn_demo.pt")
tw = torch_wrapper_test(model)
ak_jets, dak_jets = prepare_jets_array(njets=256)

ak_res = tw(ak_jets)
dak_res = tw(dak_jets)

assert np.all(np.isclose(ak_res, dak_res.compute()))
```

xgboost

```python
class xgboost_test(xgboost_wrapper):
    def awkward_to_numpy(self, events):
        ret = np.column_stack([events[name].to_numpy() for name in feature_list])
        return [], dict(data=ret)

    def dask_columns(self, events):
        return [events[f] for f in feature_list]

xgb_wrap = xgboost_test("tests/samples/xgboost_example.xgb")

# Dummy 1000 event array with 20 feature branches
ak_events = ak.zip(
    {f"feat{i}": ak.from_numpy(np.random.random(size=1_000)) for i in range(20)}
)
ak.to_parquet(ak_events, "ml_tools.xgboost.parquet")
dak_events = dak.from_parquet("ml_tools.xgboost.parquet")

ak_res = xgb_wrap(ak_events)
dak_res = xgb_wrap(dak_events)

# Results should be identical
assert ak.all(ak_res == dak_res.compute())
```

- Work by Yi-Mu Chen (UMD) to connect ML inference to dask-based workflows
  - Automatic upload of ML model (if necessary) to dask cluster, fetch to nodes evaluating
  - One entry point, with some configuration for triton, xgboost, PyTorch, et al.
- Aim to provide easy migration of coffea+ML workflows to coffea 2023