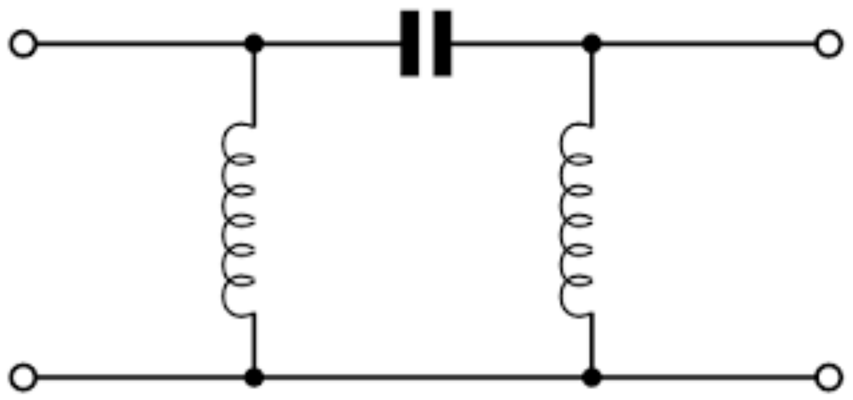
Fine-Grained HEP Analysis Task Graph Optimization with Coffea and Dask

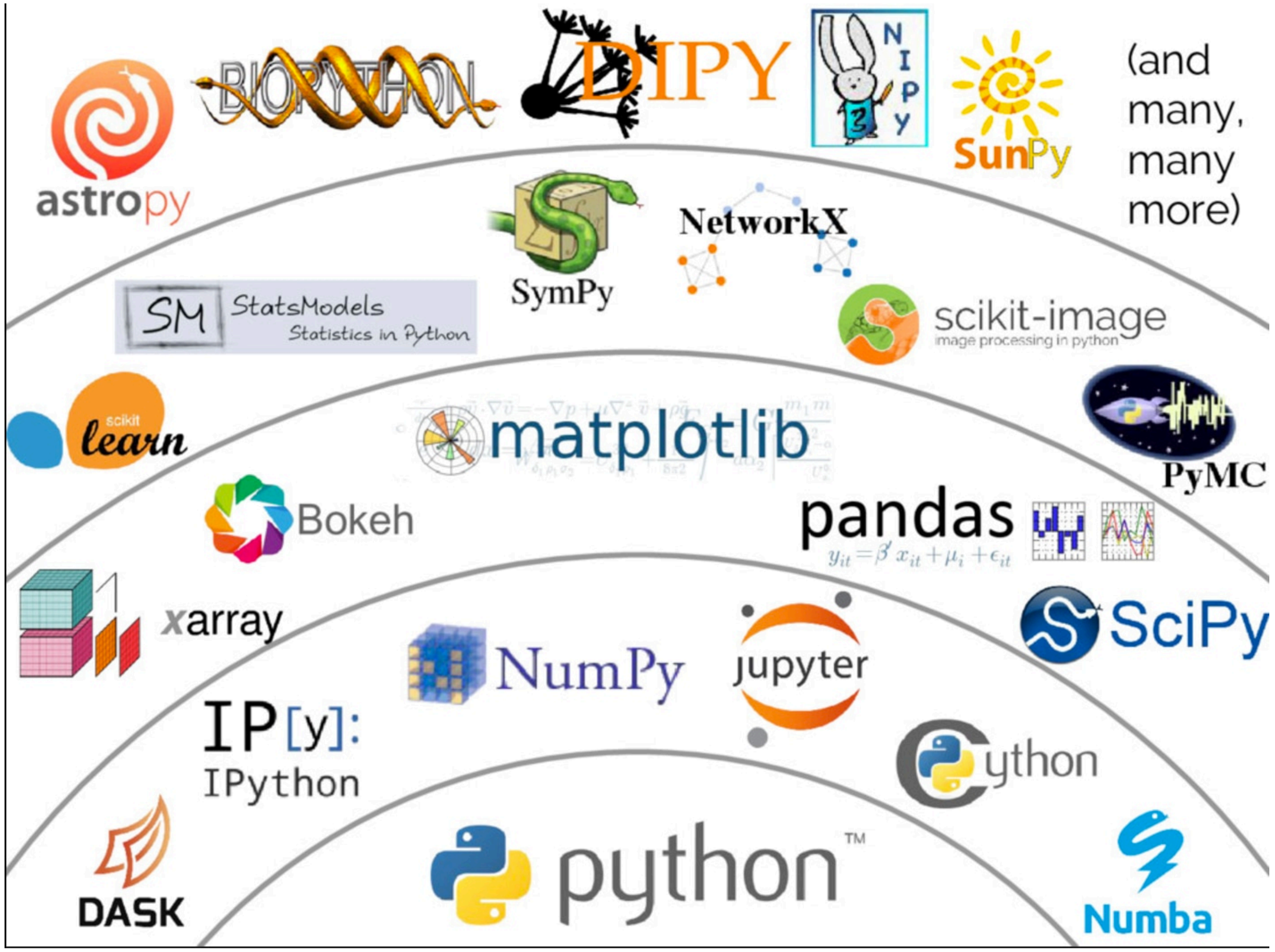
Lindsey Gray, Nick Smith (FNAL),
Doug Davis, Martin Durant (Anaconda),
Angus Hollands, Jim Pivarski (Princeton),
Yi-Mu Chen (UMD),
CHEP 2023 - Norfolk, Virginia
9 May 2023

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)



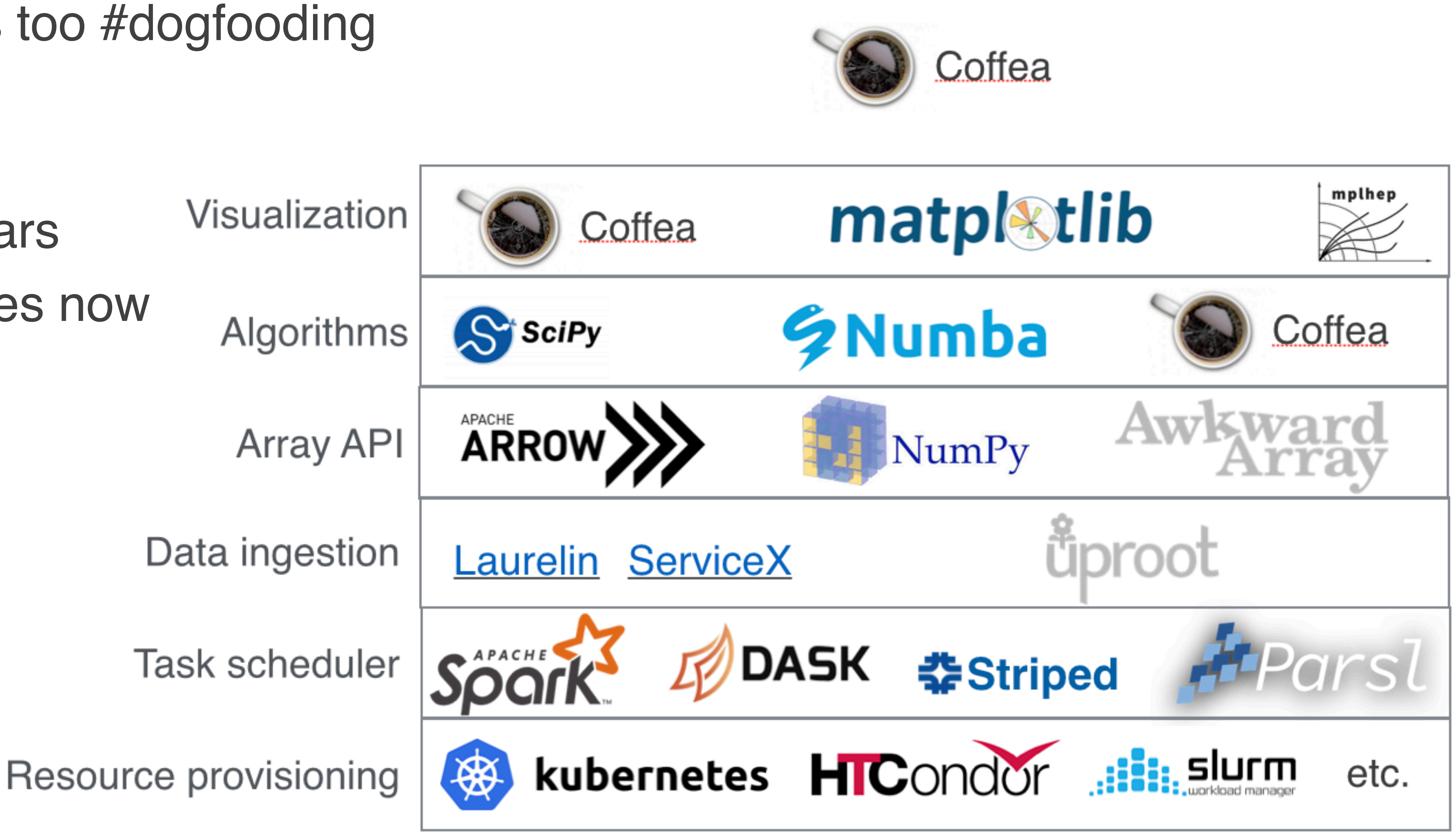
Scientific Python



Coffea is

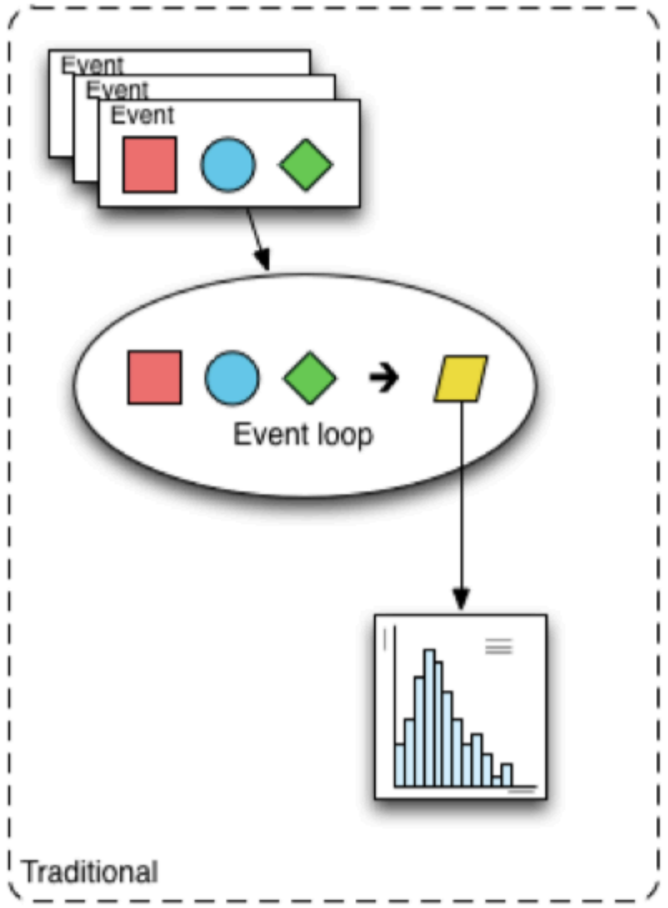
- A package in the scientific python ecosystem
  - `$ pip install coffea`
- A user interface for columnar analysis
  - With missing pieces of the stack filled in
- A minimum viable product
  - We are data analyzers too #dogfooding
- A really strong glue

- Going strong for five years
  - Many published analyses now

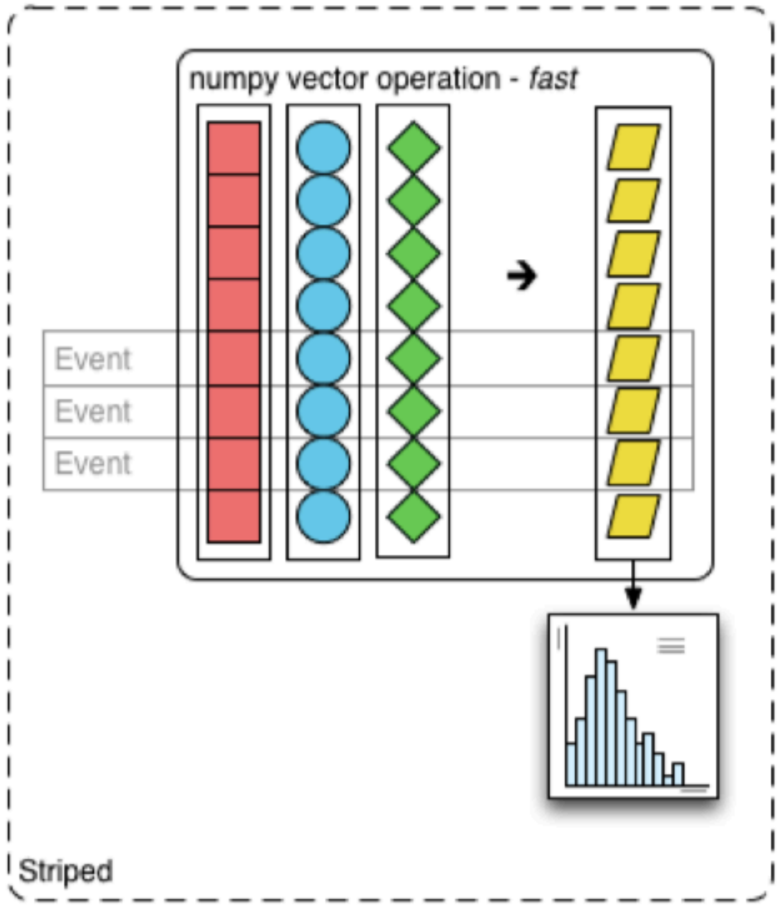


What is columnar analysis?

- 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



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

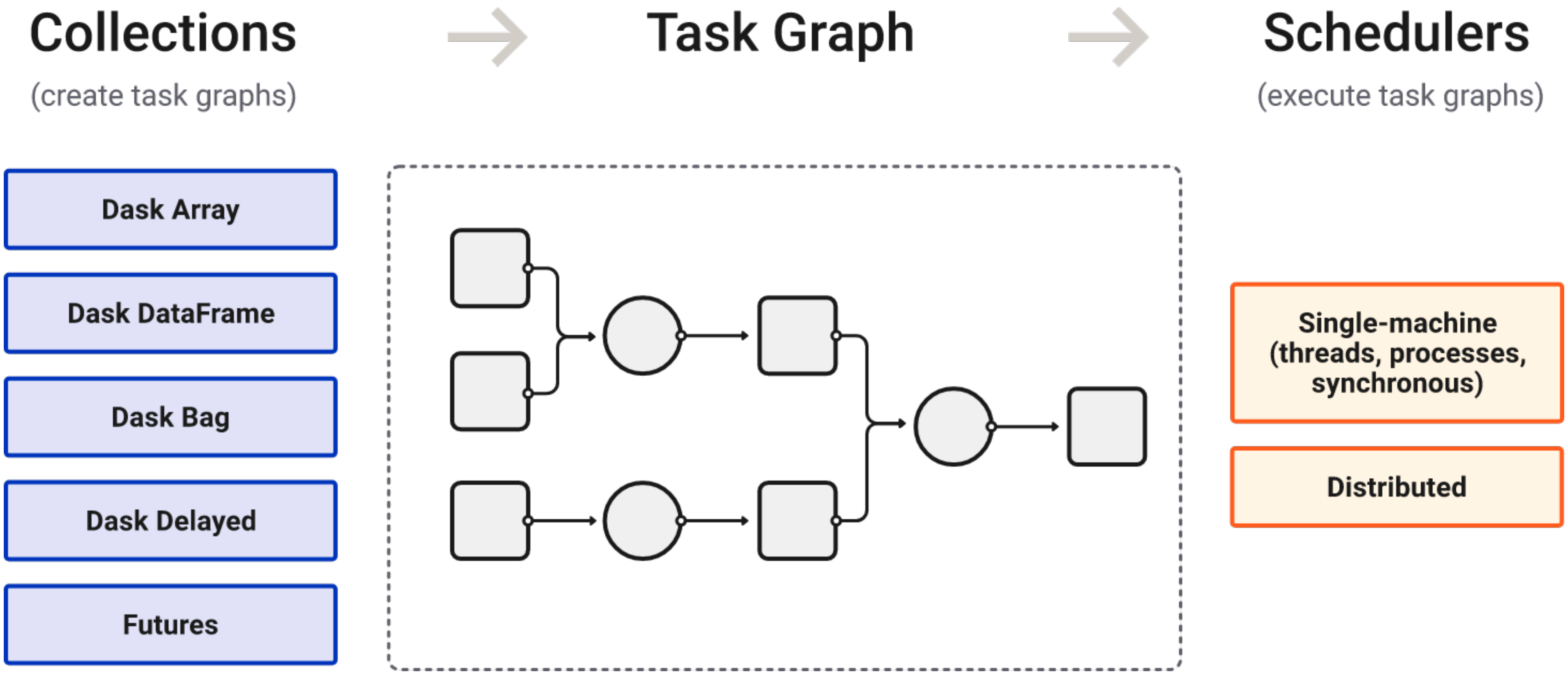
This talk:

```
# “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

Fermilab

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

Coffea 2023 (yes, we switched to CalVer)

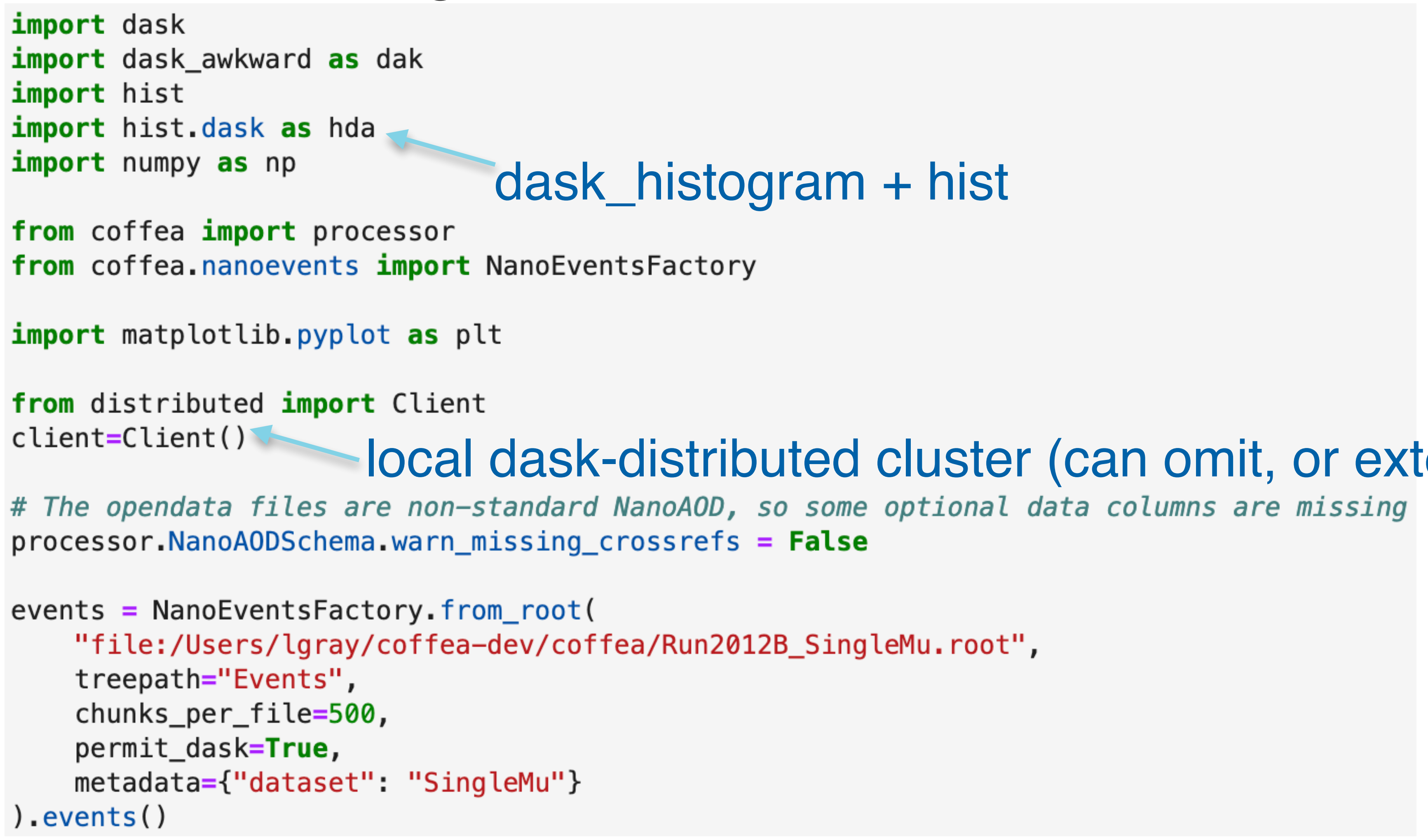
(awkward-array
hist
dask, parsl, etc.)

(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”



```python
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

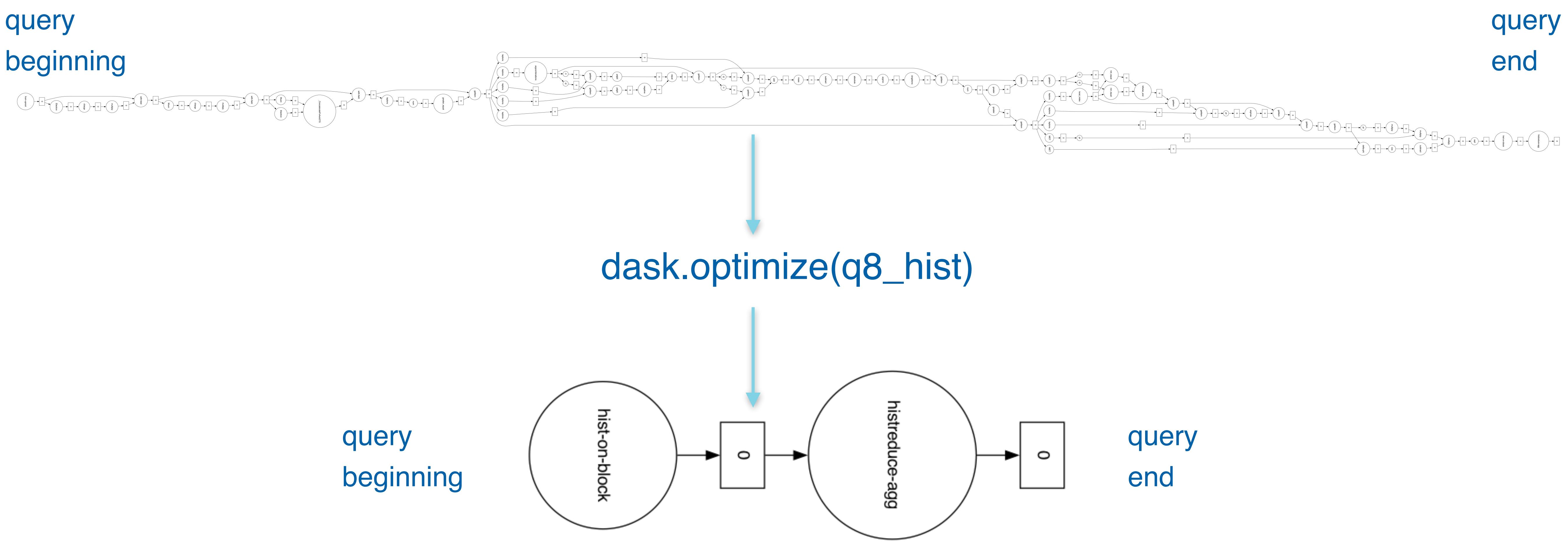
local dask-distributed cluster (can omit, or extend to condor)

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

```python
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()
```

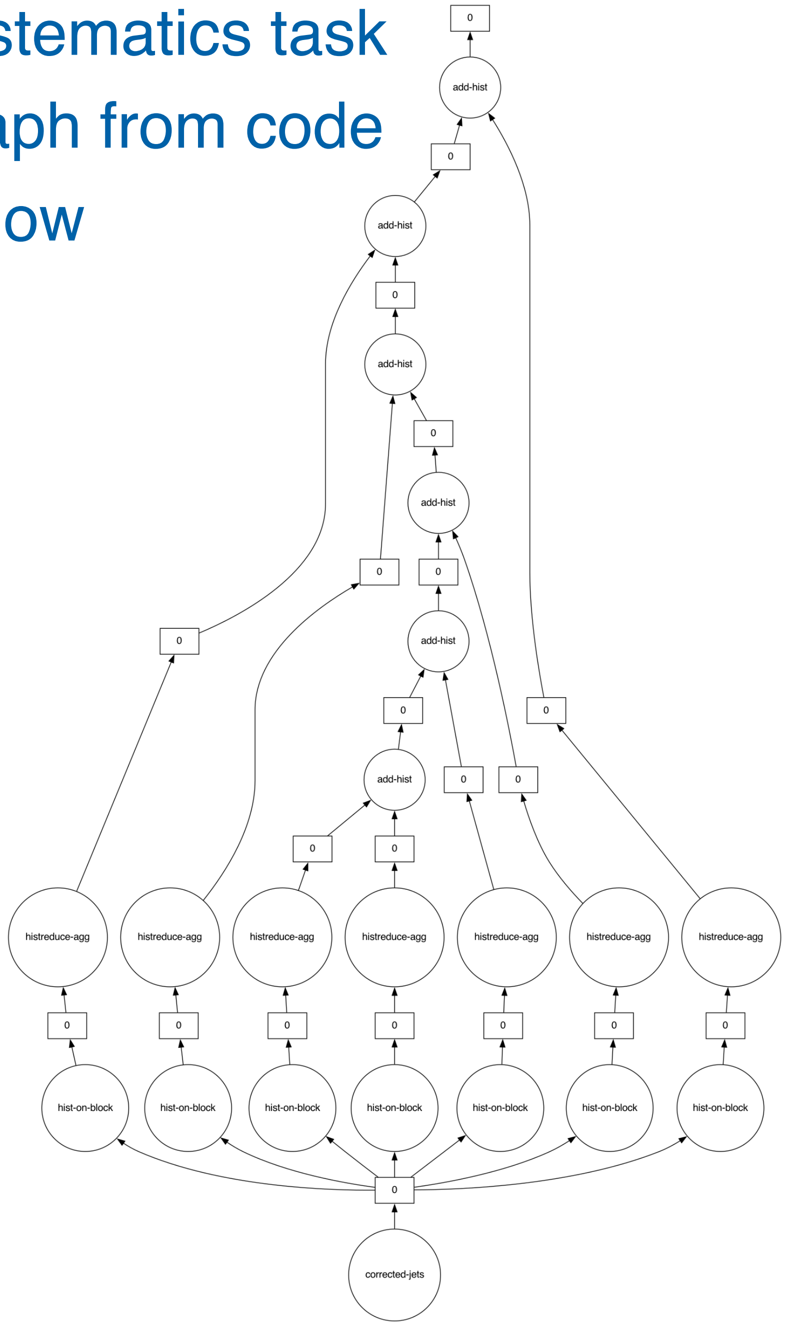
Optimization Example: Q8



- 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
  - Reasonably complete analysis is 234 layers post-optimization (ops fuse to hist filling)

Practicalities: Writing Code (3)

Shortened example of systematics task graph from code below



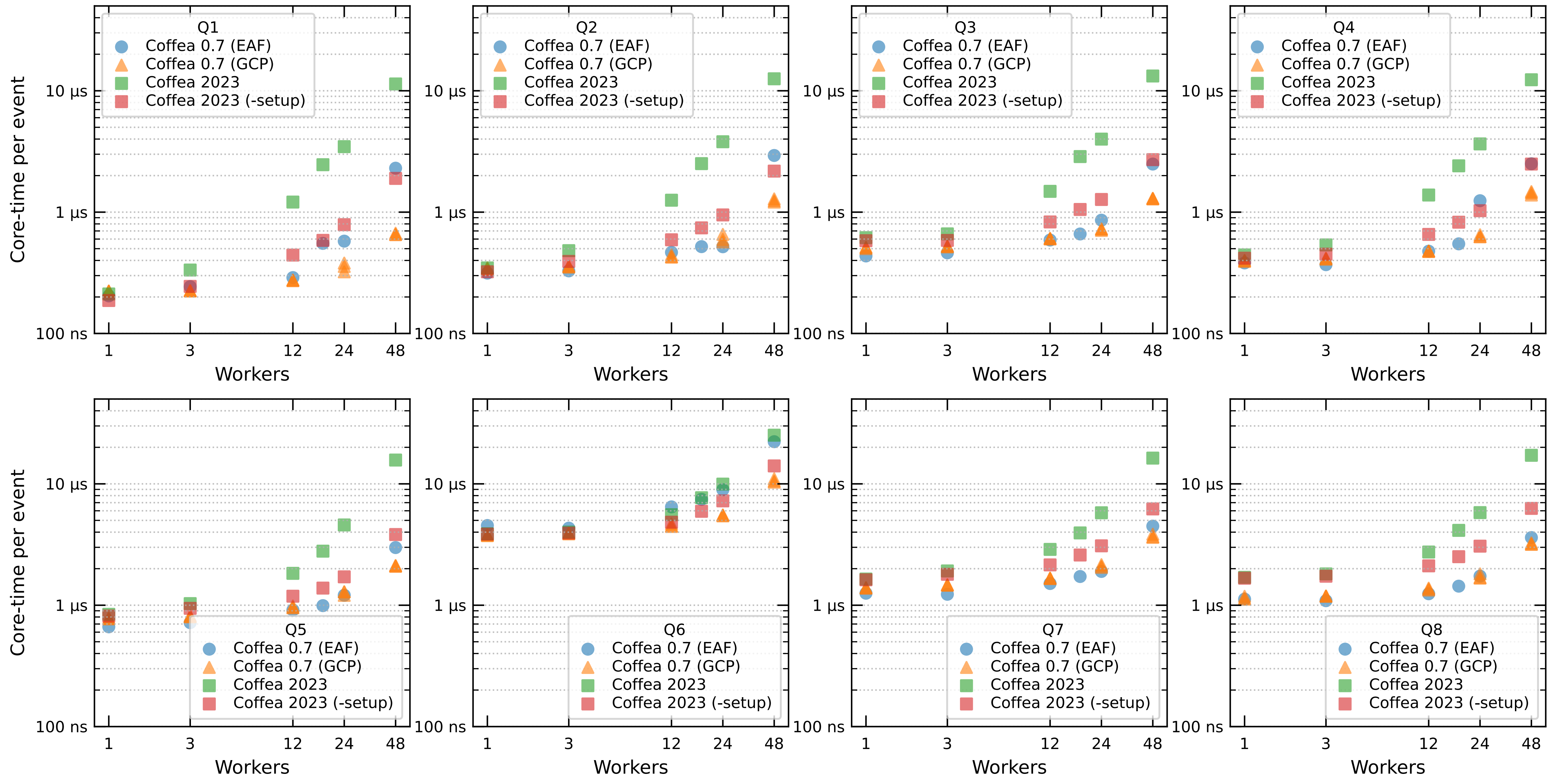
- Systematics are one of the most critical aspects of HEP analysis development
  - Without systematics we cannot do our science
  - Performing critical tasks in code should be clear and intuitive
- In coffea 2023, distributed, parallel systematics loops are written as loops over systematic variations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

```python
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)
  - Compared to last round of benchmarks no significant performance degradation
  - “Setup time” dominated by spinning up full disk 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

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 and complete 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 comfortable, 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