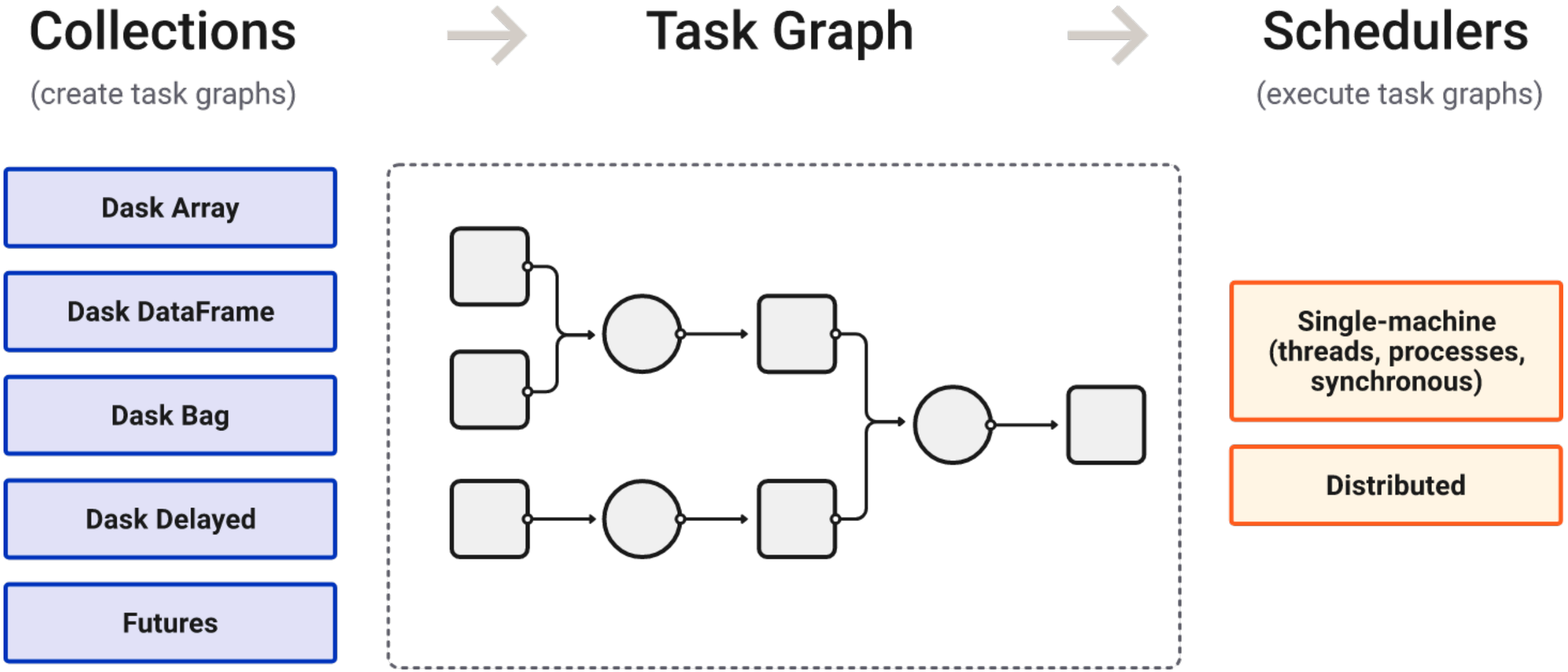
# Introduction to Dask and Dask-awkward

Lindsey Gray

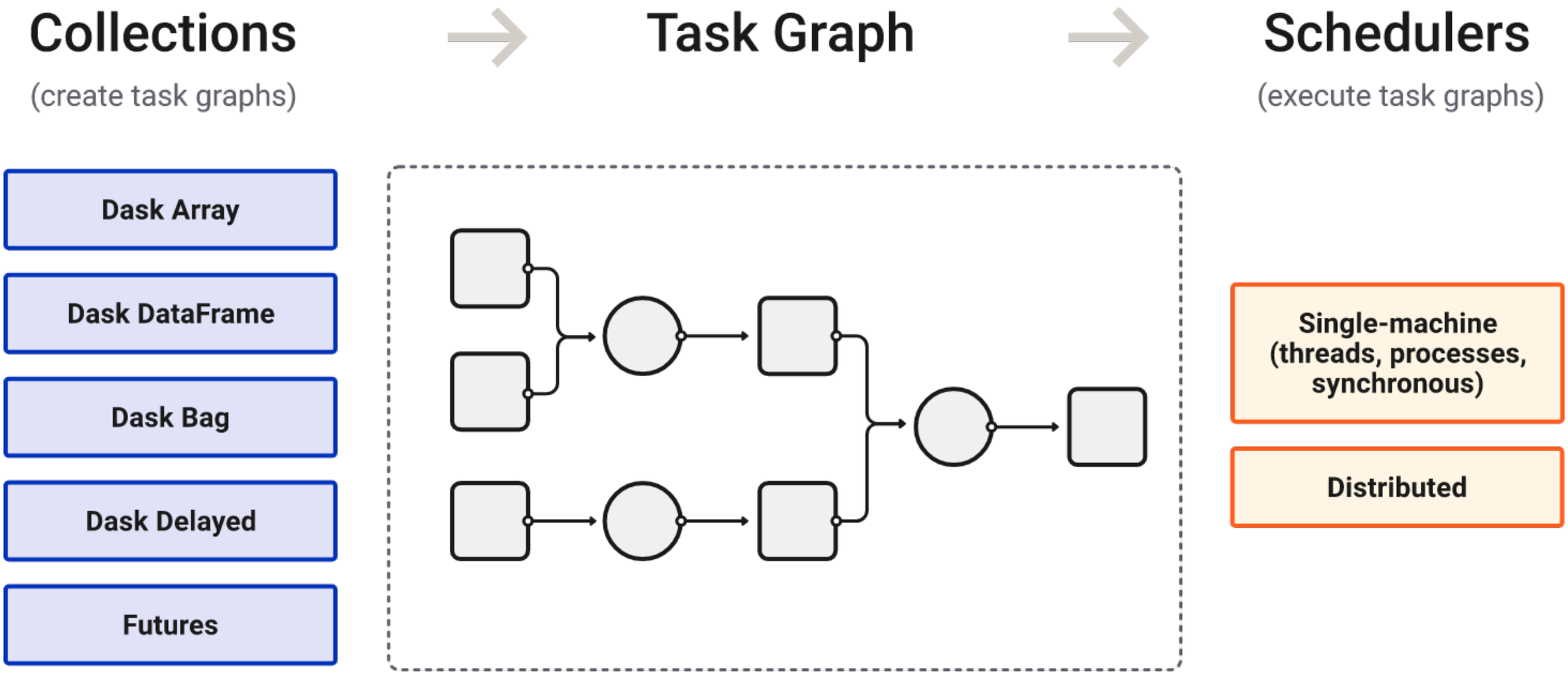
IRIS-HEP Training Event, Princeton
20 June 2024

# 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

# So what does that set of words really mean?




- You use collections to write straightforward python
- That code generates an abstract, declarative, description of your analysis
  - It can then be executed by anything that implements the collection's array interface!
  - This makes analysis code extremely portable for tradeoff in underlying complexity
- I hope to dig into this complexity enough so you can reason about task graphs

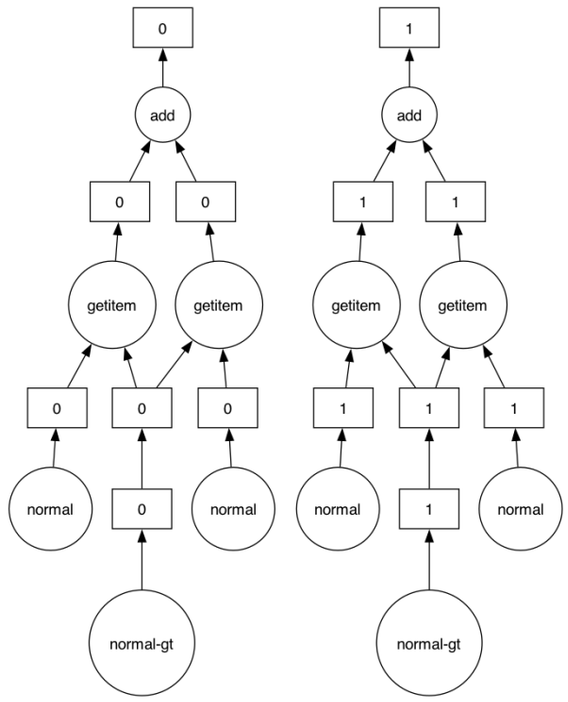
# Major dask “verbs"

- **compute** “dask.compute(stuff); stuff.compute()”
  - This runs optimization routines (by default) and then executes the graph using a specified scheduler or “get” function
  - It blocks until the computation is complete and continues local execution once the request computation job is done
  - All results only exist “client side”, i.e. nothing is cached
- **persist** “dask.persist(stuff); stuff.persist()”
  - Like compute but non-blocking, immediately returning a new dask collection
  - Terminal nodes in the task graph (i.e. final results) are cached and the dask collection points to these cached results
  - A further compute call is required to fetch the cached results!
- **visualize** “dask.visualize(stuff); stuff.visualize()”
  - Display information about the steps that will be executed to compute your requested results
  - Does not cause any actual computation to happen
  - Useful for understanding how efficient an operation might be when executed in parallel

# “High Level” vs “Low Level” Graphs

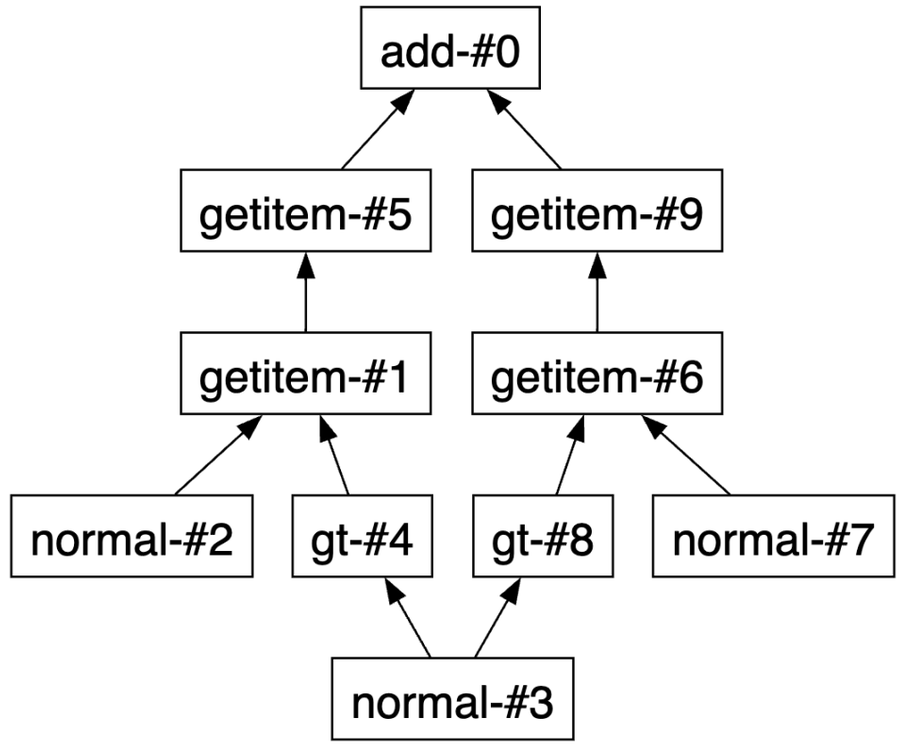
```python
import dask.array as da

x = da.random.normal(size=10000, chunks=(5000,))
y = da.random.normal(size=10000, chunks=(5000,))
z = da.random.normal(size=10000, chunks=(5000,))

pos_x = x > 0
w = y[pos_x] + z[pos_x]

w.visualize(optimize_graph=True)
```

## Low-level graph




## High-level graph

```python
w.dask.visualize()
```




- Dask achieves parallelism by operating over “partitions” or “chunks” of data
- All dask collections will have a “.dask” property that contains the “high-level graph”
  - The high-level graph represents the operations to be done over the whole input dataset
  - The low level graph represent what happens to each input partition and each data access
  - Embarrassingly parallel tasks will have a low level graph that are clones over partitions!

# Keys in a task graph

```
import dask.array as da

x = da.random.normal(size=10000, chunks=(5000,))
y = da.random.normal(size=10000, chunks=(5000,))
z = da.random.normal(size=10000, chunks=(5000,))

pos_x = x > 0
w = y[pos_x] + z[pos_x]
```

```
w.dask.keys()
```

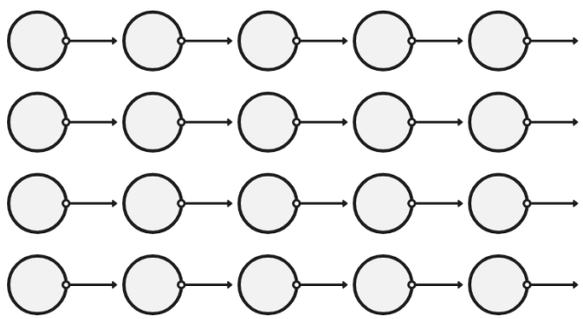
```
dict_keys([('add-409561e07200a6cbfd926597eaf2bdf4', 0), ('add-409561e07200a6cbfd926597eaf2bdf4', 1), ('getitem-54d9c1cbce7524f2d4a981a058e
3a3bf', 0), ('getitem-54d9c1cbce7524f2d4a981a058e3a3bf', 1), ('normal-e98c0b79a1c58e194000dac8df15129e', 0), ('normal-e98c0b79a1c58e194000
dac8df15129e', 1), ('normal-1d35c102a977253c1e109e8580fef013', 0), ('normal-1d35c102a977253c1e109e8580fef013', 1), ('gt-935757e86f5e2010ef
8bd61571fa6c3b', 0), ('gt-935757e86f5e2010ef8bd61571fa6c3b', 1), ('getitem-57d3eebe608615a77585593ceee6ff51', 0), ('getitem-57d3eebe608615
a77585593ceee6ff51', 1), ('getitem-b935203cdb1228959978b901a2d8eec6', 0), ('getitem-b935203cdb1228959978b901a2d8eec6', 1), ('normal-ce21fd
91ed7be1b8be52cb4904a980f8', 0), ('normal-ce21fd91ed7be1b8be52cb4904a980f8', 1), ('getitem-7d684acfe70ecc40e89609d0e70c10cf', 0), ('getite
m-7d684acfe70ecc40e89609d0e70c10cf', 1)])
```

- Task-graphs are “just” big dictionaries where the keys of the dictionary correspond to each output that’s made by your computation
- It is possible (but not often required) to request the computation of any individual key
  - This is occasionally useful for debugging but it’s easier to just evaluate your computation earlier when you’re writing it
- These keys are referenced by other keys in the dictionary, defining the graph

# Basic types of task graphs

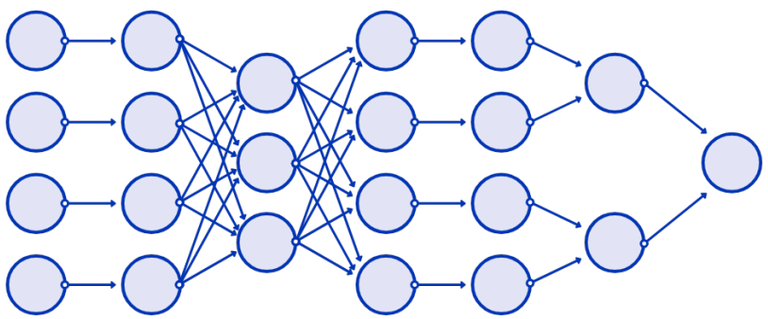
**Embarrassingly Parallel**

Hadoop/Spark/Dask/Airflow/Prefect



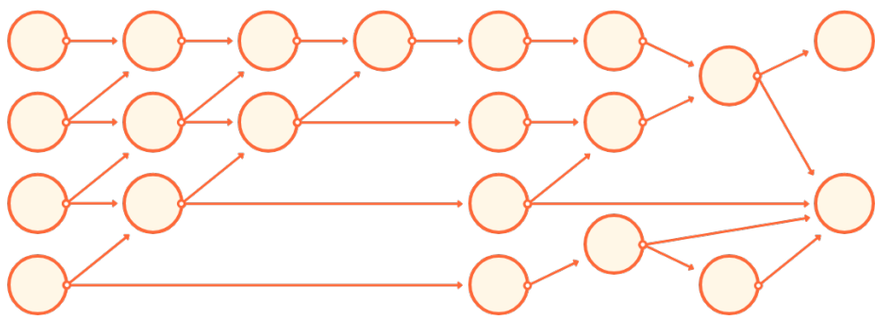
**MapReduce**

Hadoop/Spark/Dask



**Full Task Scheduling**

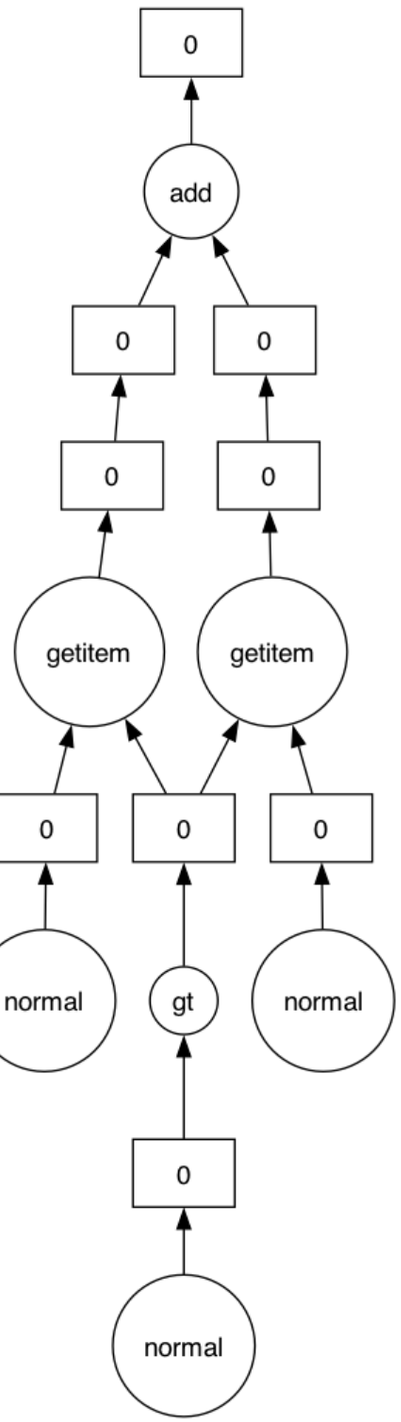
Dask/Airflow/Prefect



- HEP analysis workflows have typically been embarrassingly parallel or map-reduce
  - Skimming (without merging) is embarrassingly parallel
  - Histogramming is fundamentally a map-reduce operation
  - Usually we put anything that's more complex either in a big enough set of operations until it fits those patterns again
- Consider masks applied to many variables, systematics, corrections
  - None of these are actually embarrassingly parallel or map-reduce!
  - By using a dask-collection to write down all your operations new kinds of parallelism can be exploited to possibly* accelerate analysis further

# A simple example of exposing different parallelism

```python
import dask.array as da

x = da.random.normal(size=10_000)
y = da.random.normal(size=10_000)
z = da.random.normal(size=10_000)

pos_x = x > 0
w = y[pos_x] + z[pos_x]

w.visualize(optimize_graph=False)
```




```python
w.dask
```

```
HighLevelGraph with 9 layers.
<dask.highlevelgraph.HighLevelGraph object at 0x13fc960c0>
 0. normal-35b89e4de842487c90ae705f1e9c0a31
 1. normal-590b4df63b34e8b7f89728583149b39d
 2. gt-5dea99c548758ddcf43d02d730a7a81a
 3. getitem-23bd65b746d4f1b0cce6d012003f8347
 4. getitem-f6903ae5194738ea7b40640cdb3f3b6d
 5. normal-1add28117c35b6ff91cf2c17d78356a1
 6. getitem-fdbc6d3c375f625b8bd5e26d4ad9ccd7
 7. getitem-efb2a9517e7f915e44bf2105a8b144cc
 8. add-2d1f4be532b8b0485da5dbcd3118ddec
```



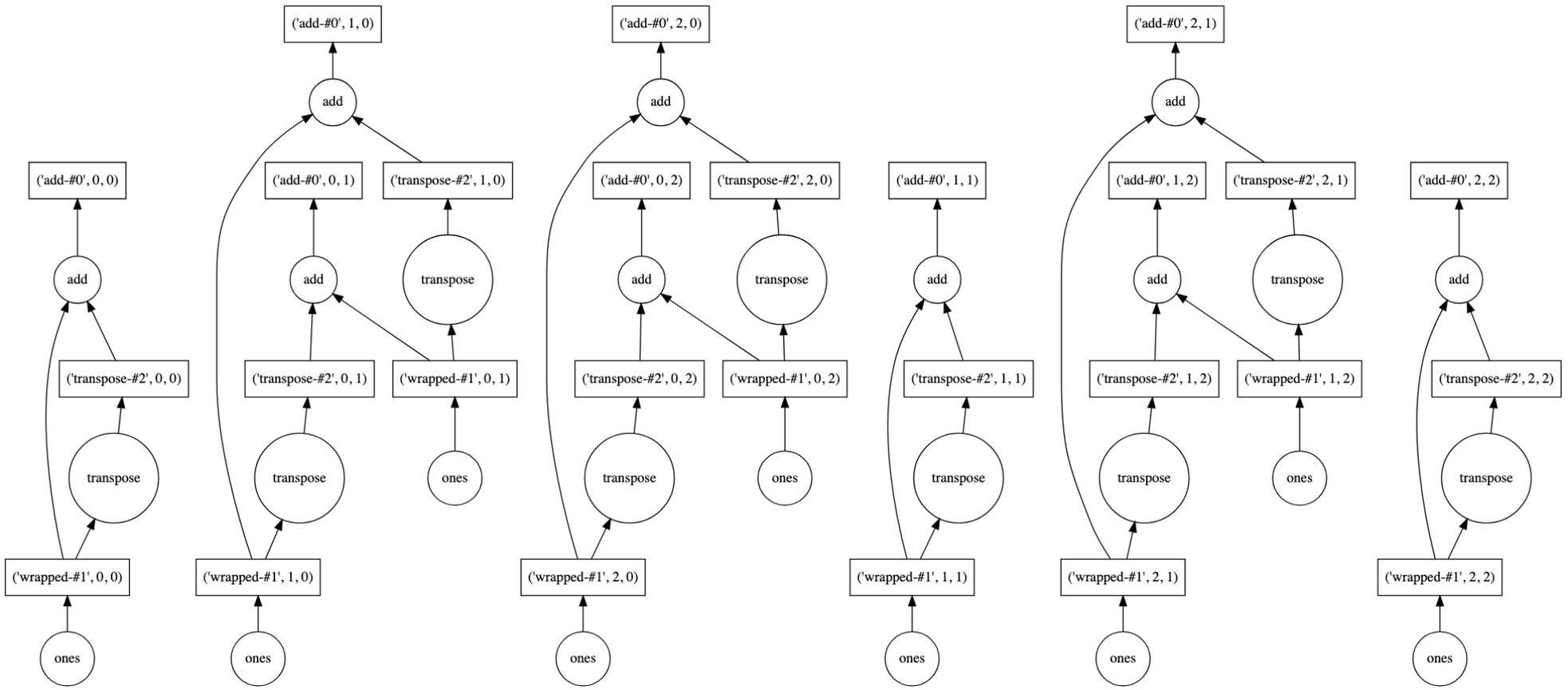
Fermilab

# Another example of parallelism with many input partitions

```python
import dask.array as da
x = da.ones((15, 15), chunks=(5, 5))

y = x + x.T

# y.compute()

# visualize the low level Dask graph
y.visualize(filename='transpose.svg')
```

```python
print(y.dask)
```

```
HighLevelGraph with 3 layers.
<dask.highlevelgraph.HighLevelGraph object at 0x13fc96c60>
 0. ones_like-53cb6f513b7c8c066b24bbb83dc2e948
 1. transpose-aa363d234a9decb5e59827d781c595f1
 2. add-c191726e3feac0c48ae79a20eb4bb1ac
```

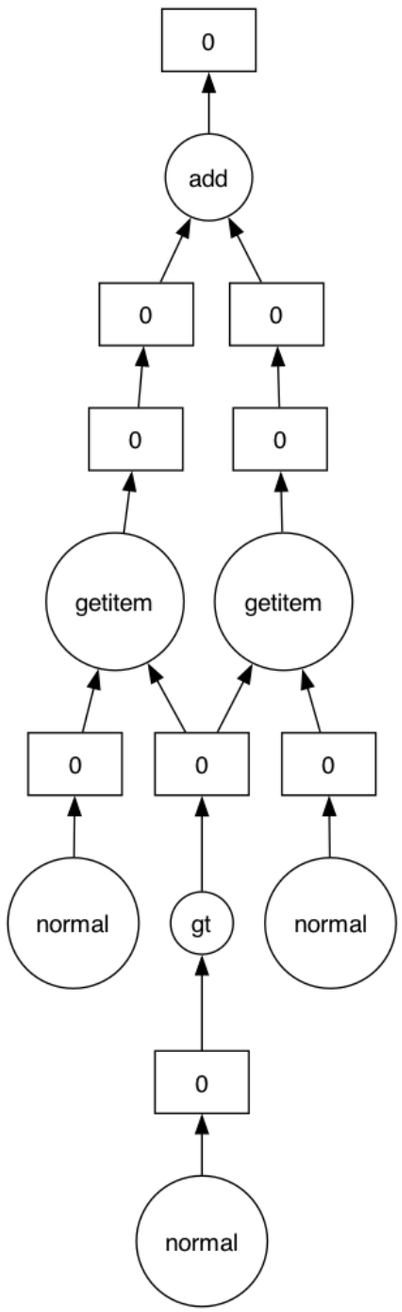




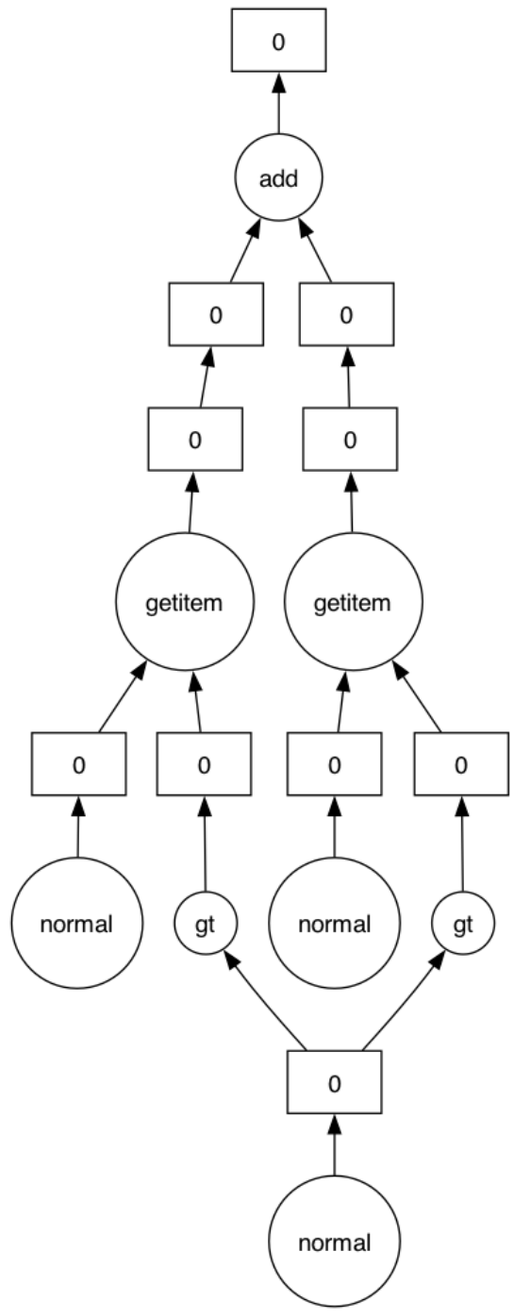

Fermilab

# Dask is extremely literal!

```python
import dask.array as da

x = da.random.normal(size=10_000)
y = da.random.normal(size=10_000)
z = da.random.normal(size=10_000)

pos_x = x > 0
w = y[pos_x] + z[pos_x]

w.visualize(optimize_graph=False)
```



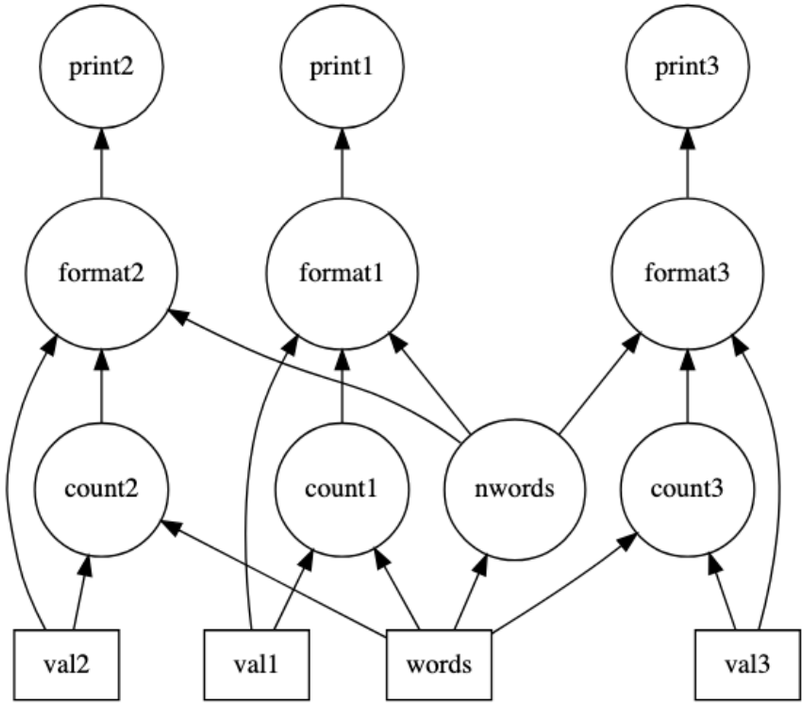

```python
import dask.array as da

x = da.random.normal(size=10_000)
y = da.random.normal(size=10_000)
z = da.random.normal(size=10_000)

w = y[x > 0] + z[x > 0]

w.visualize(optimize_graph=False)
```

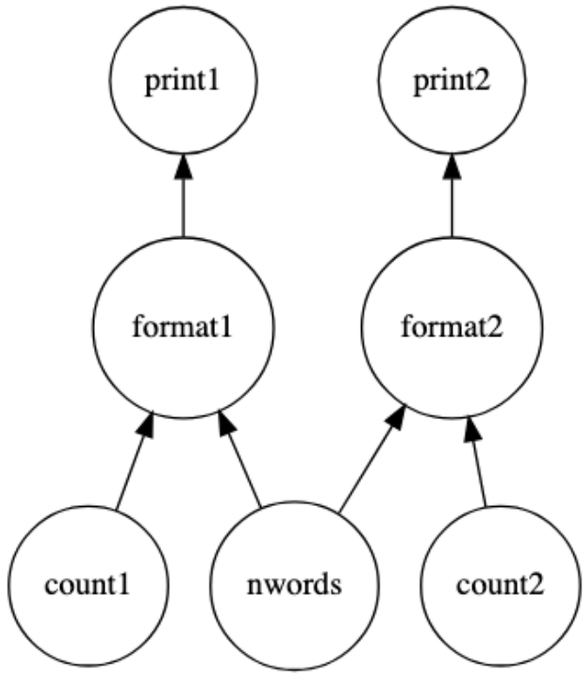


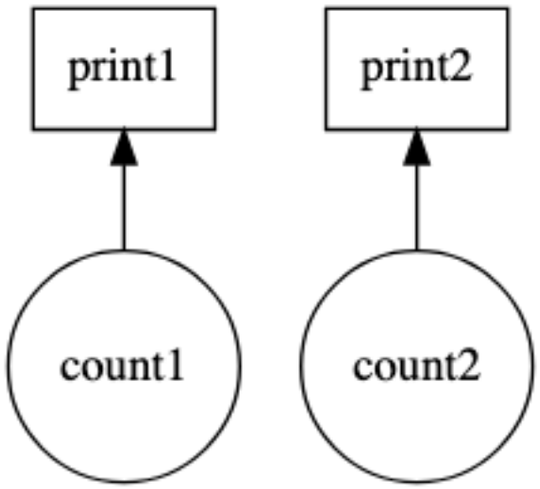



Fermilab

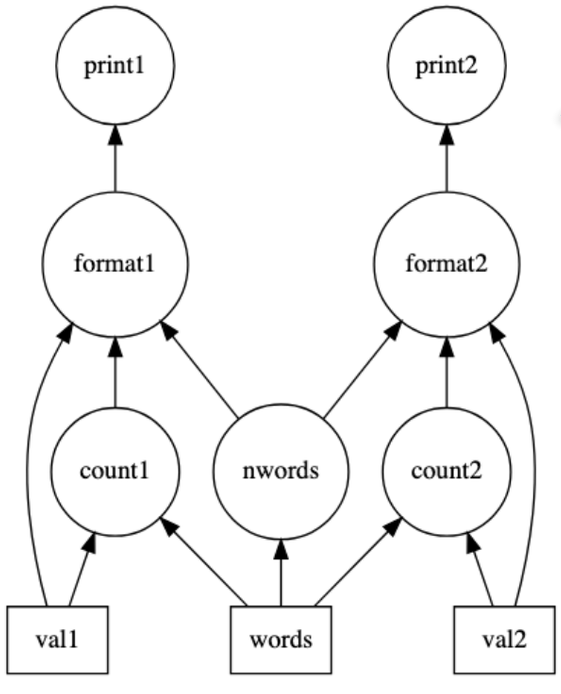
# Task Graph Optimization



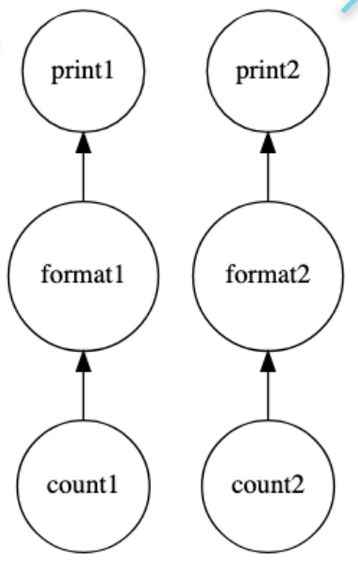

cull




inline

inline_functions

fuse

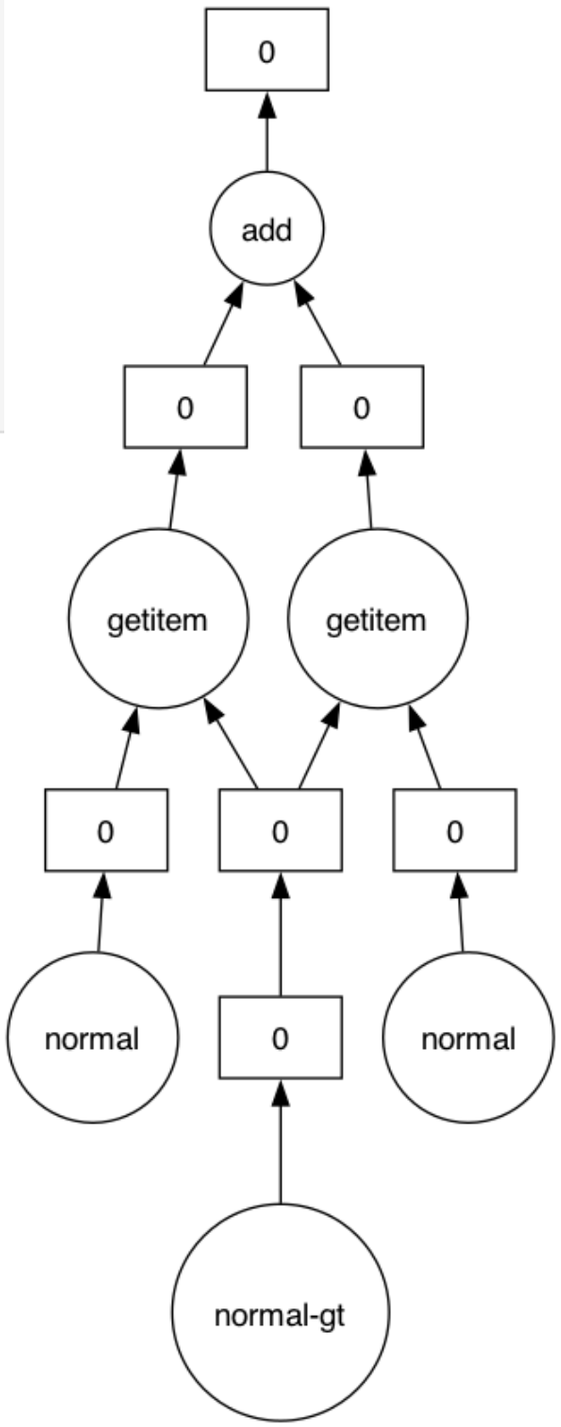




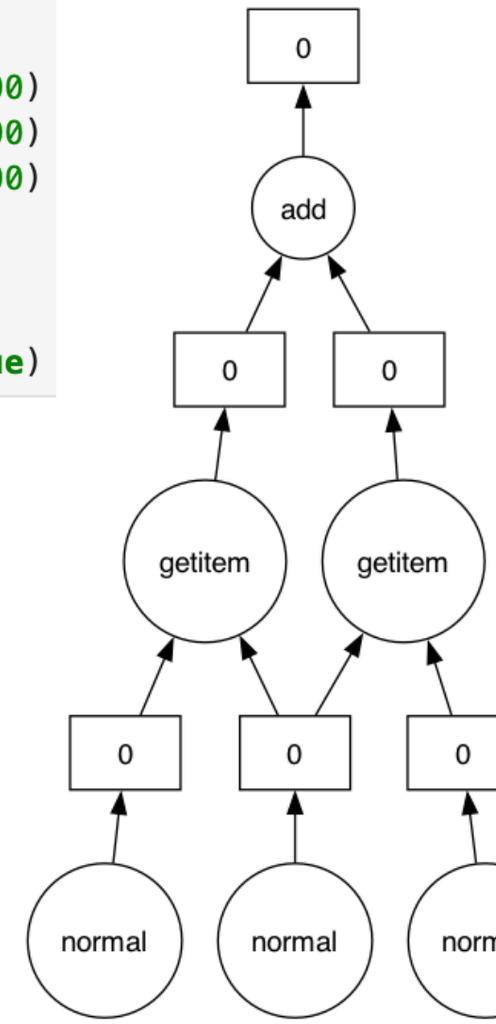

Fermilab

# Dask is extremely literal … and it has consequences

```python
import dask.array as da

x = da.random.normal(size=10000)
y = da.random.normal(size=10000)
z = da.random.normal(size=10000)

pos_x = x > 0
w = y[pos_x] + z[pos_x]

w.visualize(optimize_graph=True)
```




```python
import dask.array as da

x = da.random.normal(size=10000)
y = da.random.normal(size=10000)
z = da.random.normal(size=10000)

w = y[x > 0] + z[x > 0]

w.visualize(optimize_graph=True)
```






Fermilab

# Benchmarking with parallelism is important!

```python
import dask.array as da

x = da.random.normal(size=100_000_000, chunks=(5_000_000,))
y = da.random.normal(size=100_000_000, chunks=(5_000_000,))
z = da.random.normal(size=100_000_000, chunks=(5_000_000,))

pos_x = x > 0
w = y[pos_x] + z[pos_x]

%timeit w.compute(scheduler="sync")
```

```
6.09 s ± 163 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```python
import dask.array as da

x = da.random.normal(size=100_000_000, chunks=(5_000_000,))
y = da.random.normal(size=100_000_000, chunks=(5_000_000,))
z = da.random.normal(size=100_000_000, chunks=(5_000_000,))

pos_x = x > 0
w = y[pos_x] + z[pos_x]

%timeit w.compute(scheduler="threads")
```

```
1.07 s ± 206 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```python
import dask.array as da

x = da.random.normal(size=100_000_000, chunks=(5_000_000,))
y = da.random.normal(size=100_000_000, chunks=(5_000_000,))
z = da.random.normal(size=100_000_000, chunks=(5_000_000,))

w = y[x > 0] + z[x > 0]

%timeit w.compute(scheduler="sync")
```

```
6.13 s ± 92.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```python
import dask.array as da

x = da.random.normal(size=100_000_000, chunks=(5_000_000,))
y = da.random.normal(size=100_000_000, chunks=(5_000_000,))
z = da.random.normal(size=100_000_000, chunks=(5_000_000,))

w = y[x > 0] + z[x > 0]

%timeit w.compute(scheduler="threads")
```

```
876 ms ± 29 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```python
import dask.array as da

x = da.random.normal(size=100_000_000, chunks=(5_000_000,))
y = da.random.normal(size=100_000_000, chunks=(5_000_000,))
z = da.random.normal(size=100_000_000, chunks=(5_000_000,))

w = x + y + z

w = w[x > 0]

%timeit w.compute(scheduler="sync")
```

```
5.55 s ± 18.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```python
import dask.array as da

x = da.random.normal(size=100_000_000, chunks=(5_000_000,))
y = da.random.normal(size=100_000_000, chunks=(5_000_000,))
z = da.random.normal(size=100_000_000, chunks=(5_000_000,))

w = x + y + z

w = w[x > 0]

%timeit w.compute(scheduler="threads")
```

```
827 ms ± 26.5 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

# Getting to: dask-awkward

- Up to now, have looked at what's available in the base dask collections
  - Particularly array, since it's more pertinent to what we're after in the end
- dask-awkward doesn't really operate alone to get HEP analyses
  - There's also dask-histogram, which provides distributed histogram filling as well
  - These two packages together let allow you to write parallelizable, portable analyses
- You get all features of dask while writing rather familiar code
  - Task graph optimization comes for free
  - Dask-awkward also has instrumentation to optimize automatically what data is read from an input file
  - We'll get into these parts more in the notebook

# 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)

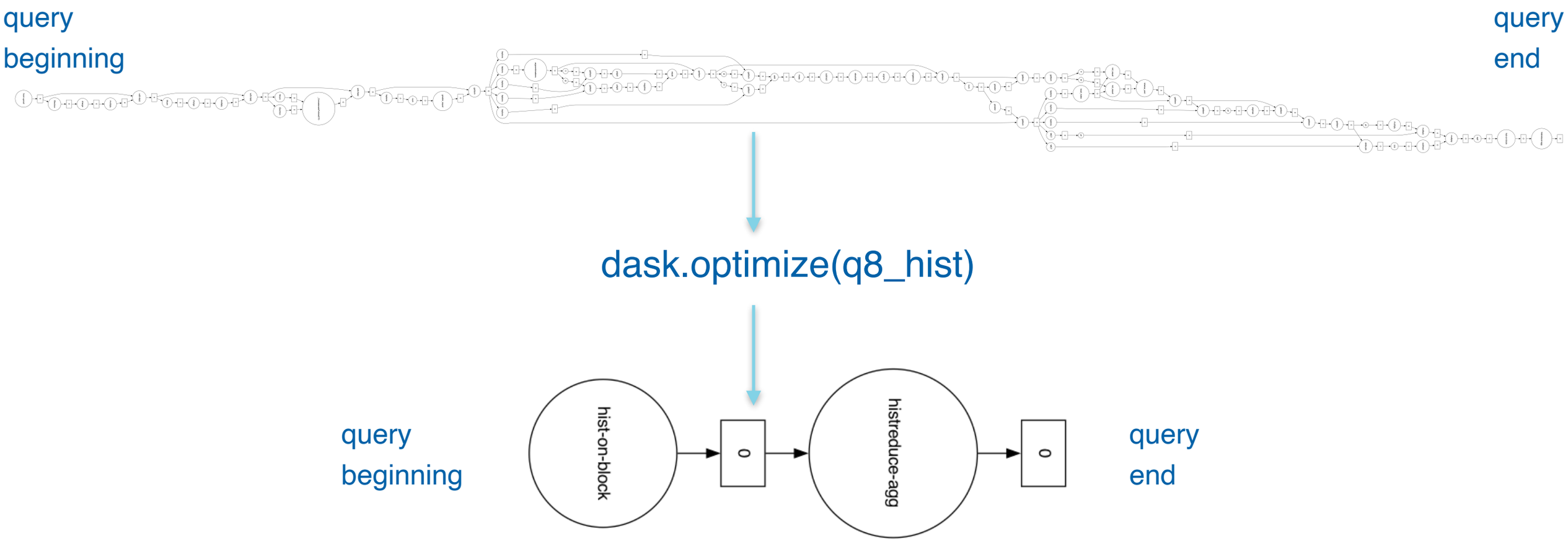
- Example: Query 8
  - from ADL Benchmarks

- Finds dilepton pairs close to z-pole and a third lepton
  - Calculates and plots the transverse mass of the system

- Aside from the .compute() statement this code is identical to the eager awkward-array code you would use to write this!

```
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

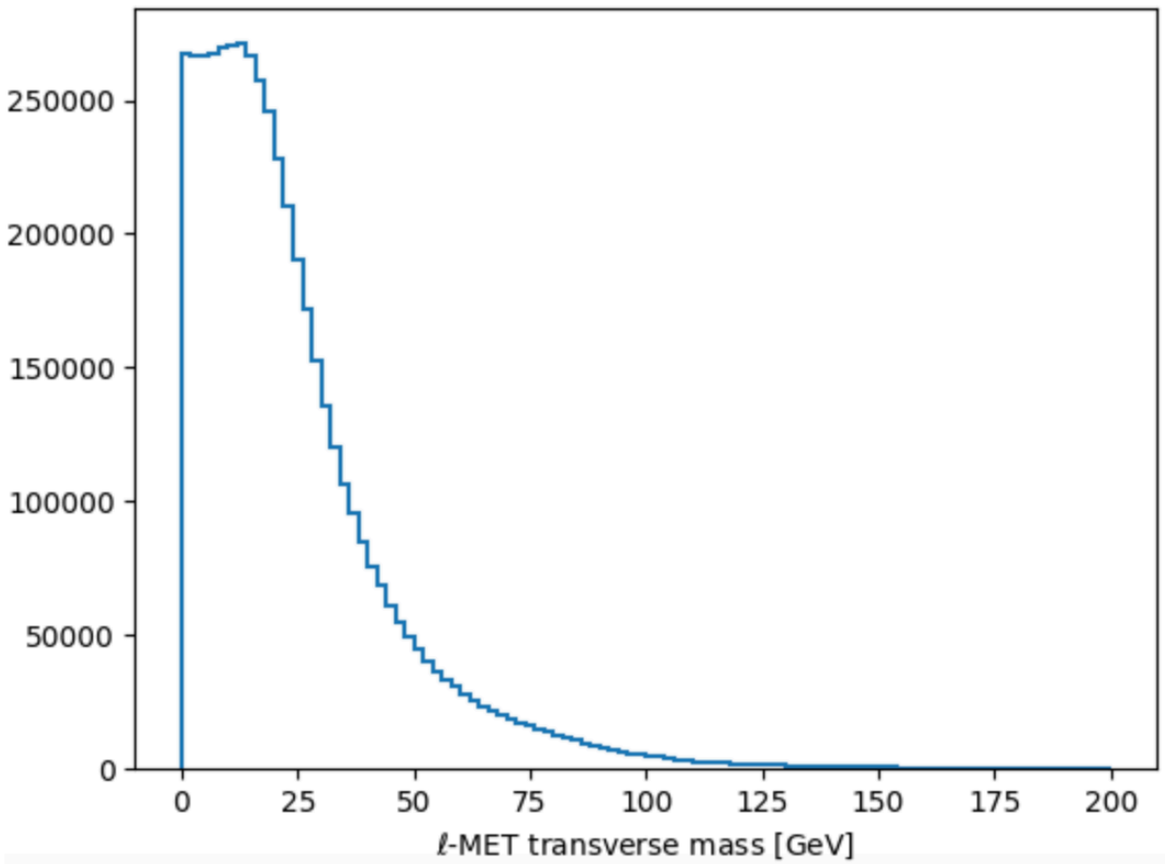



- Raw HEP analysis task graphs get large quickly
  - Reasonably complete analysis, full systematics, is ~7000 layers as written by the user
  - Q8 (top) here is 78 layers
  - Each task-graph node could be executed on a different cluster resource (data transfer!)
- Dask provides standard optimizers to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

# Data-access optimization

```
q8_hist.compute().plot1d(flow="none")
dak.necessary_columns(q8_hist)
```

```
{'from-uproot-d895a5d3a21bc4fd7e5f7c35639408ec': frozenset({'Electron_charge',
            'Electron_eta',
            'Electron_mass',
            'Electron_phi',
            'Electron_pt',
            'MET_phi',
            'MET_pt',
            'Muon_charge',
            'Muon_eta',
            'Muon_mass',
            'Muon_phi',
            'Muon_pt',
            'nElectron',
            'nMuon'})}
```






Fermilab

# Concluding remarks before practical tutorial

- Introduced the dask parallel processing library
  - Walked through how it decomposes processing tasks into steps in a *taskgraph*
  - Dug into some of the details of what these task graphs are and how they work

- Demonstrated the kinds of parallelism that dask makes available
  - Depending on how heavy pieces of data are, it is possible to tune the kind of parallelism that's possible in the graph
  - Demonstrated that some times doing **more** work can be more efficient because there are fewer synchronization points and correspondingly simpler optimized task graphs

- Introduced dask-awkward and demonstrated that it is very similar to raw awkward array in terms of user-facing behavior
  - Also benefits immediately from dask infrastructure to optimize and inspect task graphs
  - Come with the capability to *automatically* optimize what data is read from files
  - Demonstrated some analysis-like code and interfacing to histogramming via desk-histogram

- Let's dig into this a bit more via the notebook for this session!