RICE UNIVERSITY

U.S. DEPARTMENT OF ENERGY | Office of Science

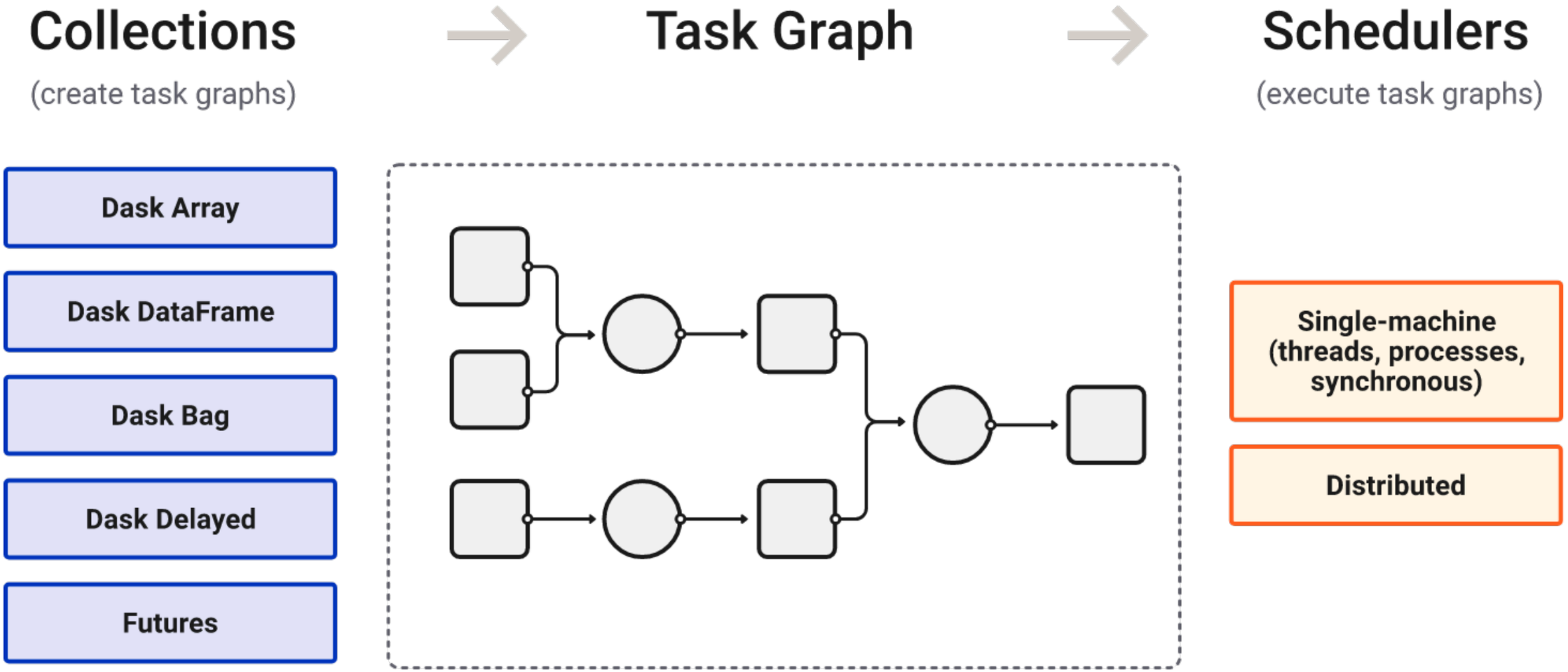
Fermilab

Introduction to Dask

Iason Krommydas, originally written by Lindsey Gray

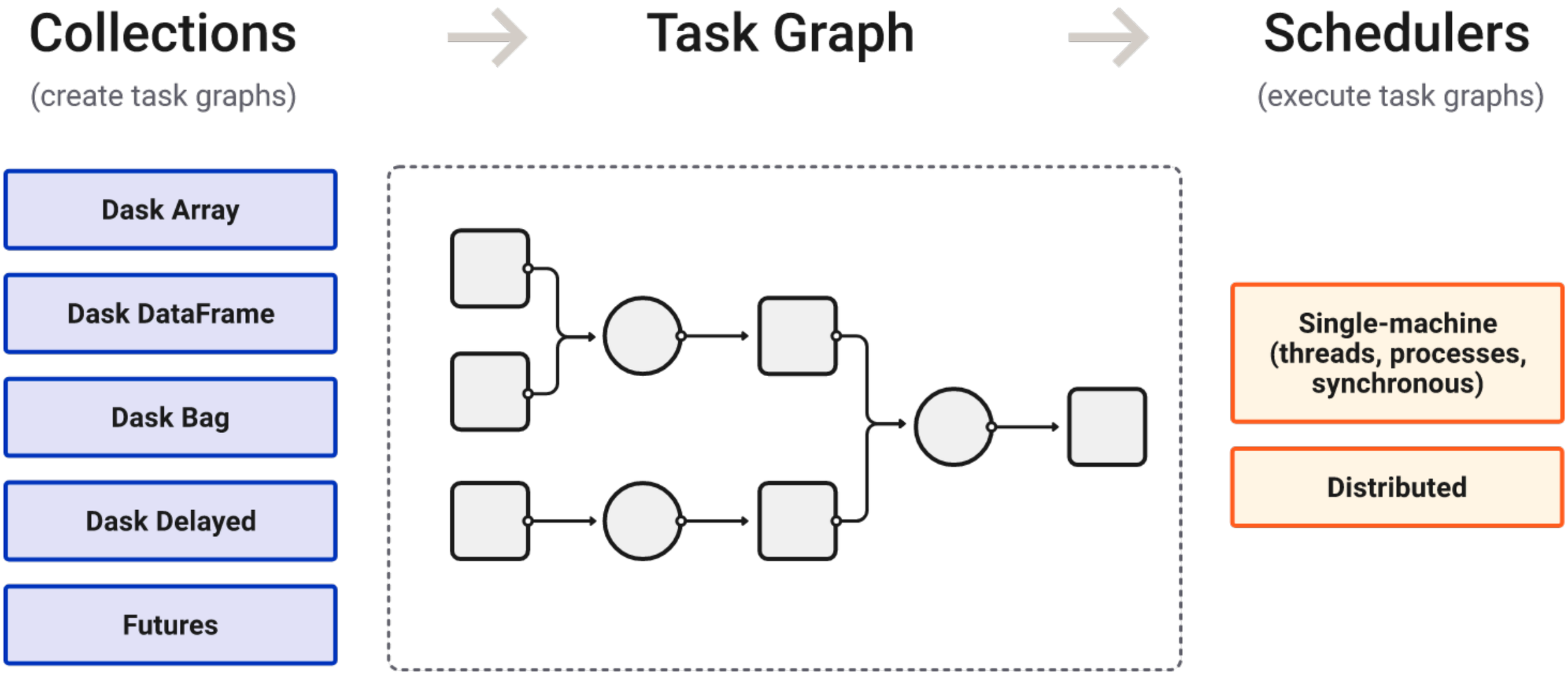
IRIS-HEP Training Event, University of Michigan
24 July 2025

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

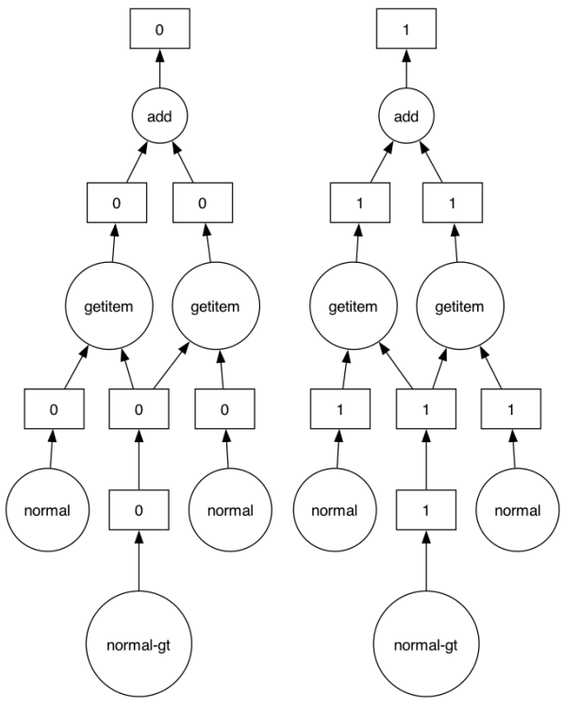
Major dask “verbs"

- **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

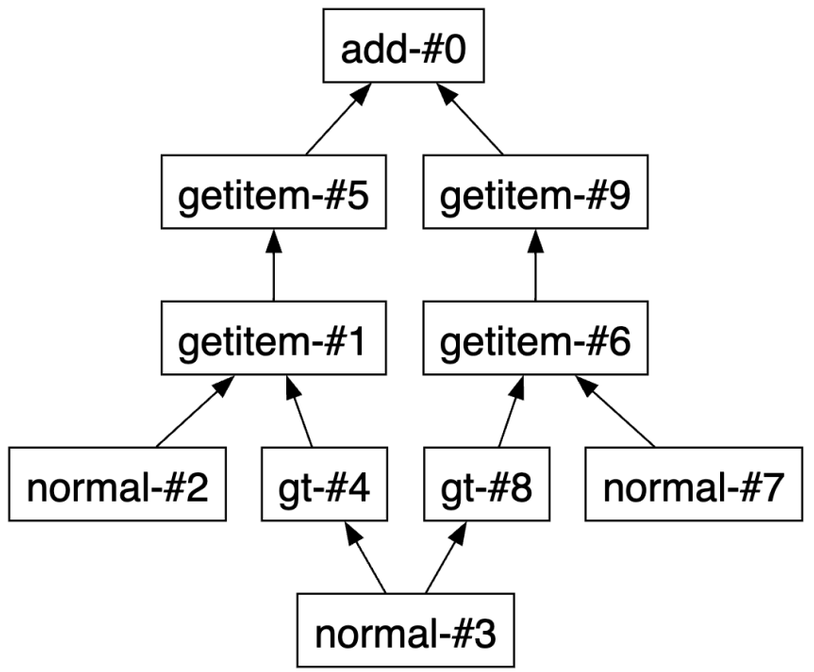
```python
import dask.array as da

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”
  - Please take everything about high level graphs with a grain of salt as dask is currently changing a lot of the internals of high level graphs to use something they call expressions and plans to deprecate high level graphs in some way that is yet unclear.
  - 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

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()
```

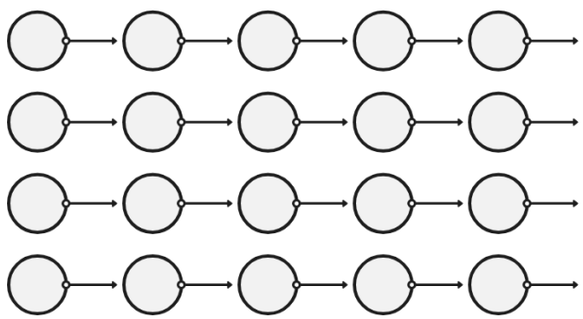
```
dict_keys([('add-409561e07200a6cbfd926597eaf2bdf4', 0), ('add-409561e07200a6cbfd926597eaf2bdf4', 1), ('getitem-54d9c1cbce7524f2d4a981a058e
3a3bf', 0), ('getitem-54d9c1cbce7524f2d4a981a058e3a3bf', 1), ('normal-e98c0b79a1c58e194000dac8df15129e', 0), ('normal-e98c0b79a1c58e194000
dac8df15129e', 1), ('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

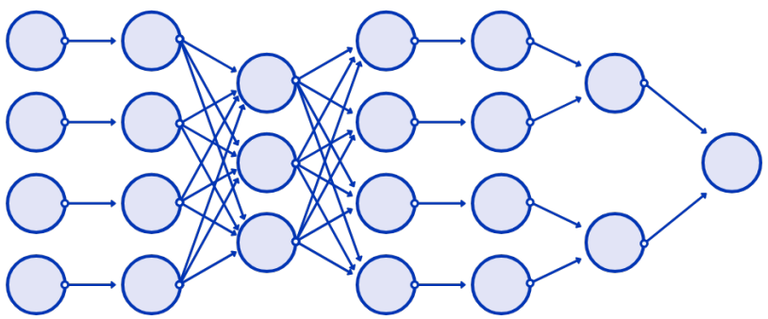
Embarrassingly Parallel

Hadoop/Spark/Dask/Airflow/Prefect



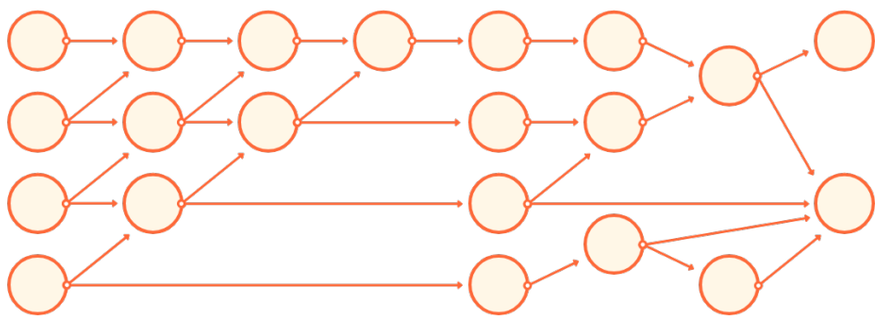
MapReduce

Hadoop/Spark/Dask



Full Task Scheduling

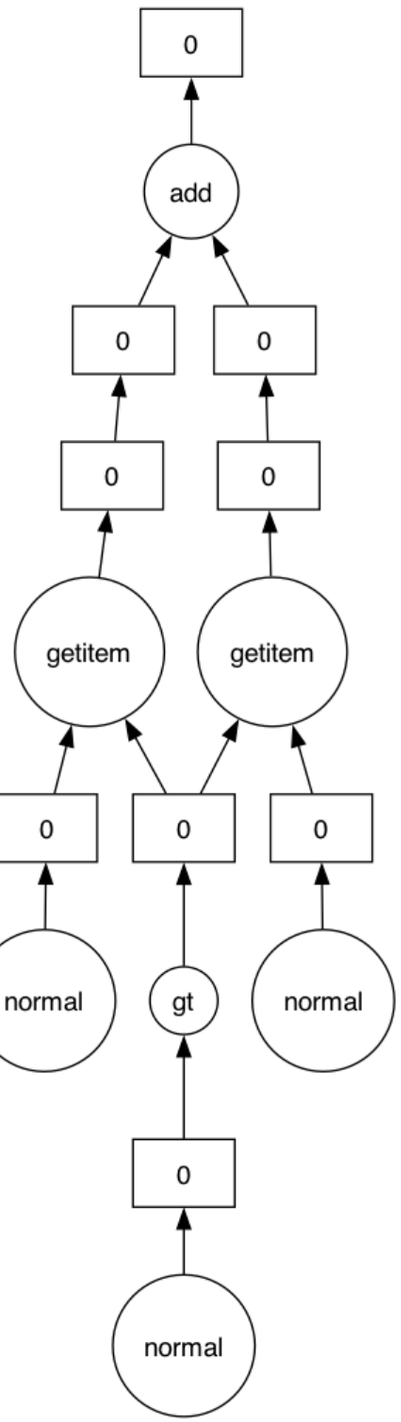
Dask/Airflow/Prefect



- 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
```

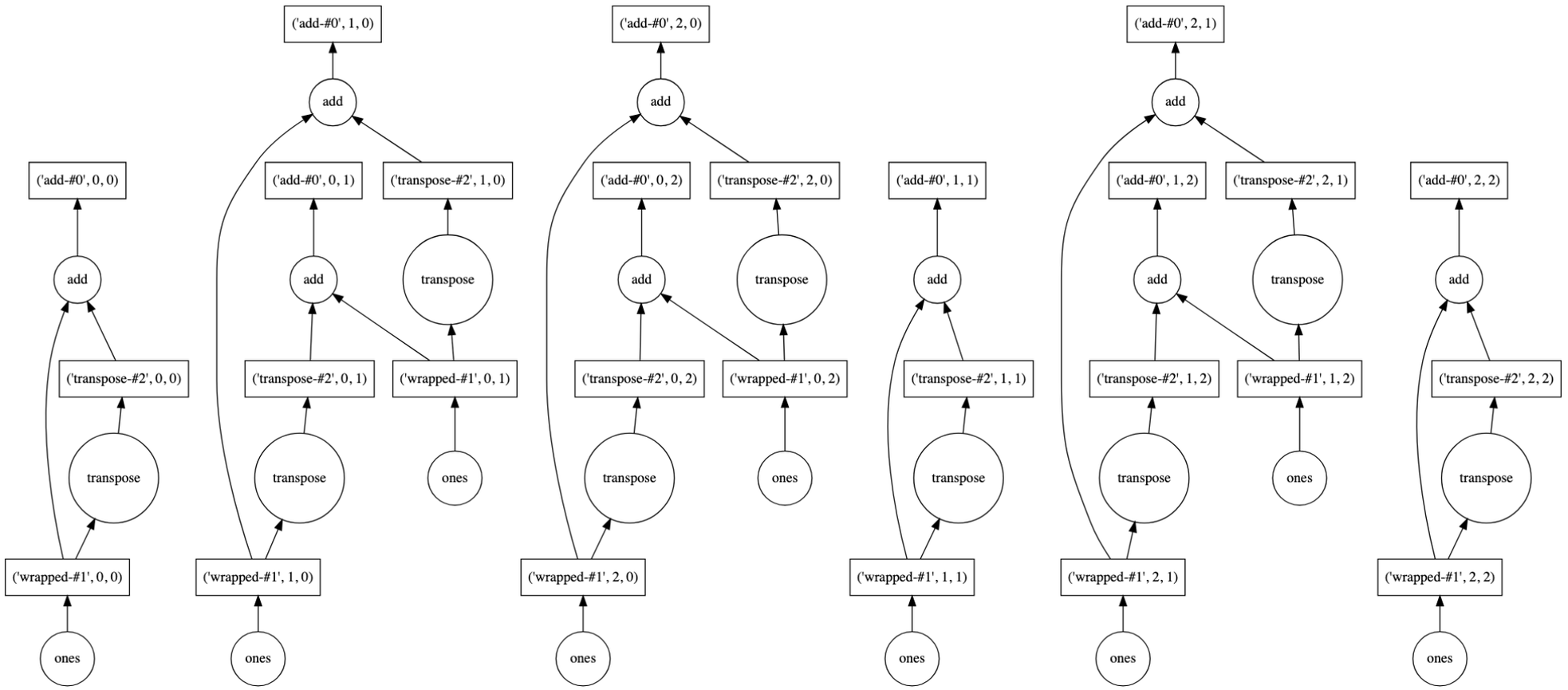
```
HighLevelGraph with 9 layers.
<dask.highlevelgraph.HighLevelGraph object at 0x13fc960c0>
 0. normal-35b89e4de842487c90ae705f1e9c0a31
 1. normal-590b4df63b34e8b7f89728583149b39d
 2. gt-5dea99c548758ddcf43d02d730a7a81a
 3. getitem-23bd65b746d4f1b0cce6d012003f8347
 4. getitem-f6903ae5194738ea7b40640cdb3f3b6d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

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')
```

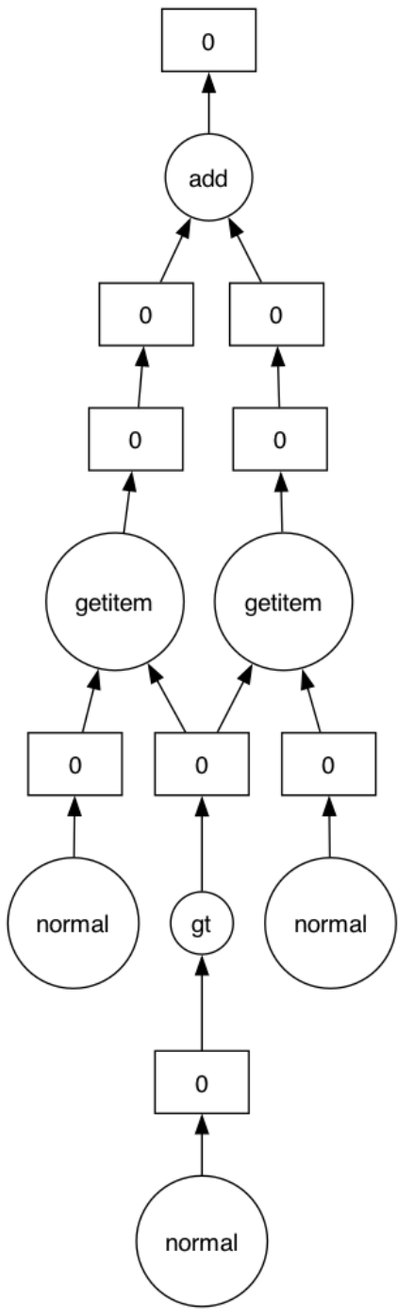
```python
print(y.dask)
```

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

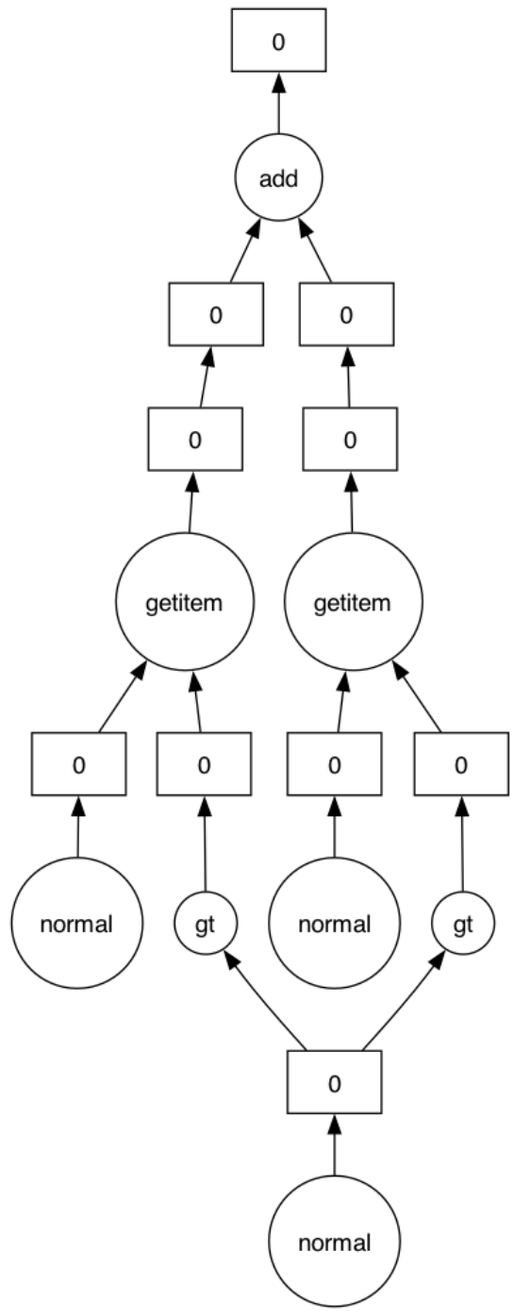


Dask is extremely literal!

```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)
```



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

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