Dask Custom Schedulers

Ben Tovar btovar@nd.edu

NDCMS - Center for Research Computing - Cooperative Computing Lab

July 2023

UNIVERSITY OF NOTRE DAME

CRC CENTER FOR RESEARCH COMPUTING

CCTools

About me

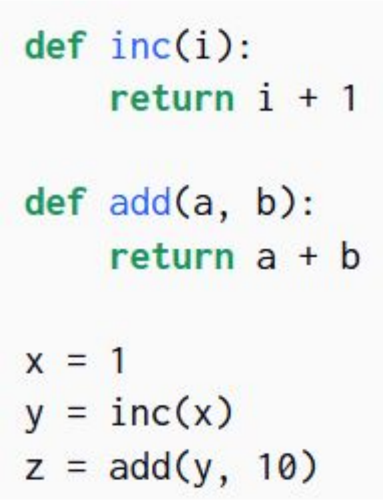
- HPC engineer at Notre Dame working for CMS
- Computer Scientist
- Tools to construct and execute scientific workflows (TaskVine, WorkQueue)
  - How to measure and allocate resources for maximum throughput?

Outline

1. Basics on Dask Task Graph
2. Custom Schedulers for Task Graphs
3. Development Opportunities

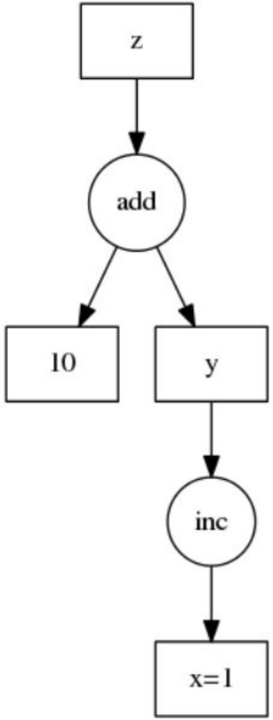
https://docs.dask.org/en/latest/graphs.html

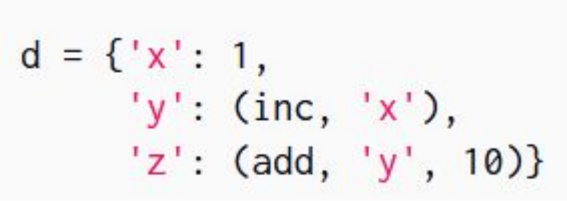
The Dask Task Graph



```
def inc(i):
    return i + 1

def add(a, b):
    return a + b

x = 1
y = inc(x)
z = add(y, 10)
```

user code



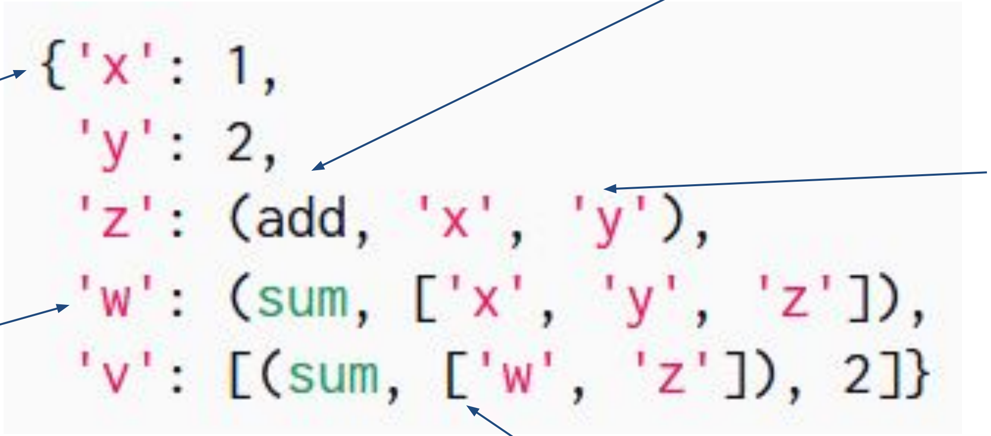


```
d = {'x': 1,
     'y': (inc, 'x'),
     'z': (add, 'y', 10)}
```

task graph
representation

https://docs.dask.org/en/latest/graphs.html

The Dask Task Graph



```
{'x': 1,
 'y': 2,
 'z': (add, 'x', 'y'),
 'w': (sum, ['x', 'y', 'z']),
 'v': [(sum, ['w', 'z']), 2]}
```

function calls are tuples which first element is callable

dict-like object

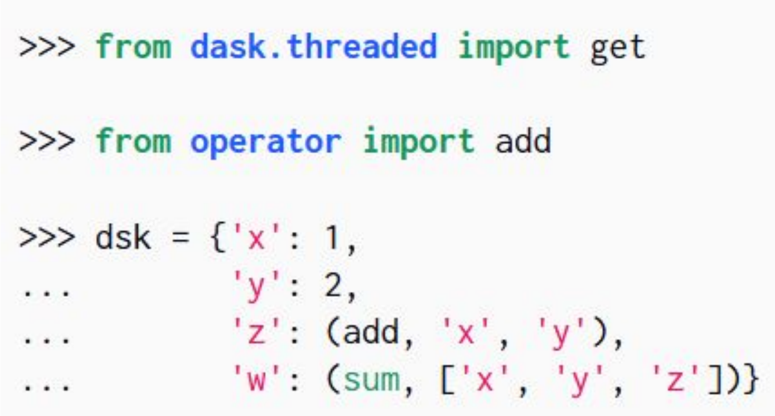
arguments to functions are either keys to graph nodes, other function calls, python objects, or lists of the previous.

graph nodes can be any hashable object, except for tuples which first element is callable

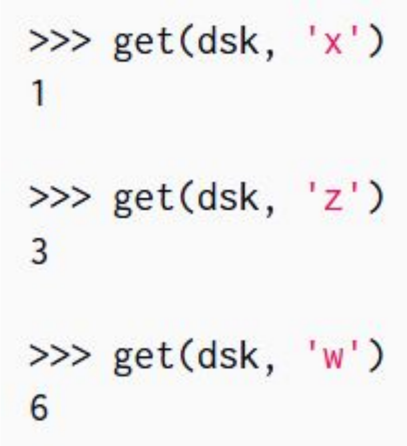
lists represent potential parallelism (or maybe the function call just wants a list as an argument)

think s-expressions

Executing the Graph



```
>>> from dask.threaded import get

>>> from operator import add

>>> dsk = {'x': 1,
...        'y': 2,
...        'z': (add, 'x', 'y'),
...        'w': (sum, ['x', 'y', 'z'])}
```

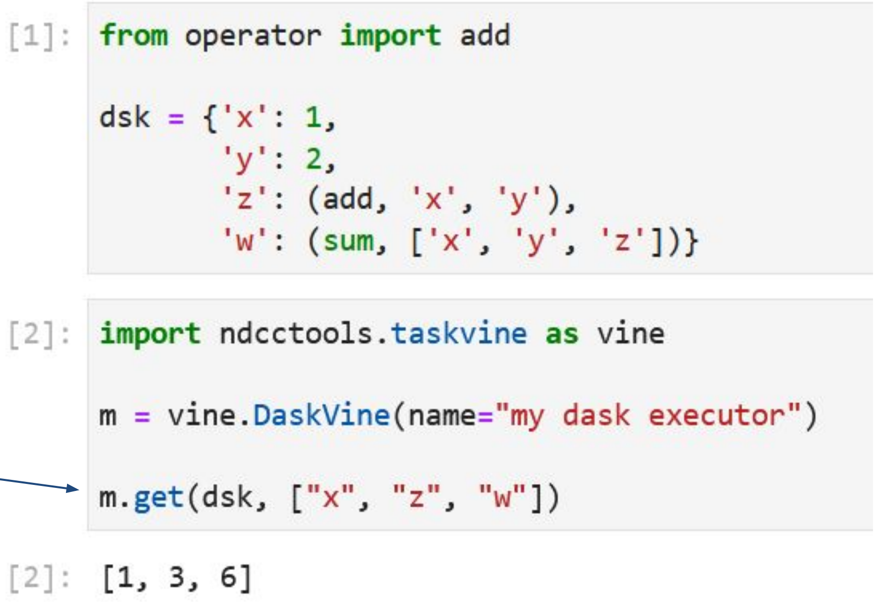


```
>>> get(dsk, 'x')
1

>>> get(dsk, 'z')
3

>>> get(dsk, 'w')
6
```

Note that the graph itself has no dask dependencies.
Here we just happen to use a dask scheduler (dask.threaded)

Executing the Graph | Custom Scheduler

anything that can receive a task graph and a (possibly nested) listed of keys to compute, can work as a dask scheduler



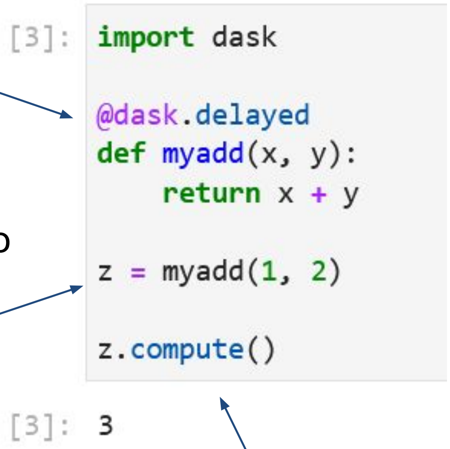
```
[1]: from operator import add

     dsk = {'x': 1,
            'y': 2,
            'z': (add, 'x', 'y'),
            'w': (sum, ['x', 'y', 'z'])}
```

```
[2]: import ndcctools.taskvine as vine

     m = vine.DaskVine(name="my dask executor")

     m.get(dsk, ["x", "z", "w"])

[2]: [1, 3, 6]
```

Using .compute()

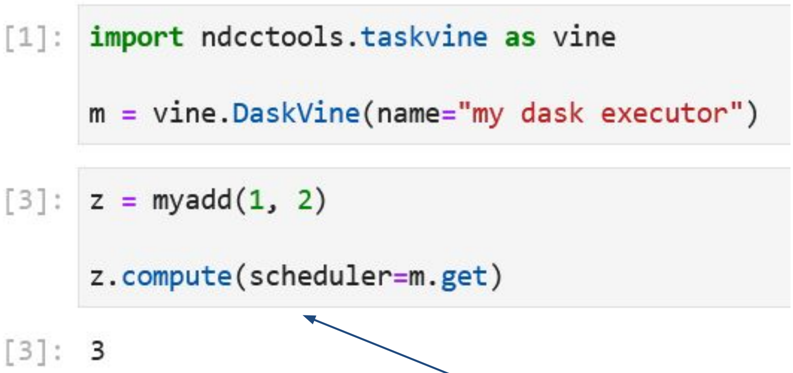
dask data types and decorators to construct the task graph

We don't explicitly refer to the task graph dict, only to *futures* that know their graph.



```
[3]: import dask

     @dask.delayed
     def myadd(x, y):
         return x + y

     z = myadd(1, 2)

     z.compute()

[3]: 3
```

use dask default executor



```
[1]: import ndcctools.taskvine as vine

     m = vine.DaskVine(name="my dask executor")

[3]: z = myadd(1, 2)

     z.compute(scheduler=m.get)

[3]: 3
```

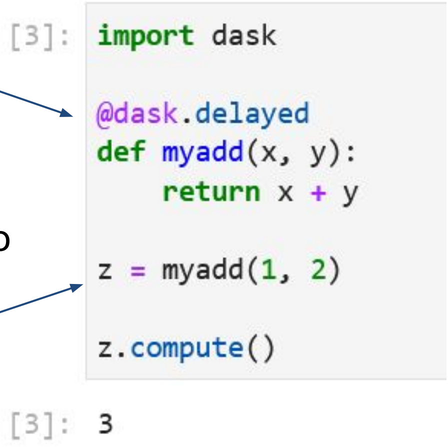
tell dask which executor to use

UNIVERSITY OF NOTRE DAME

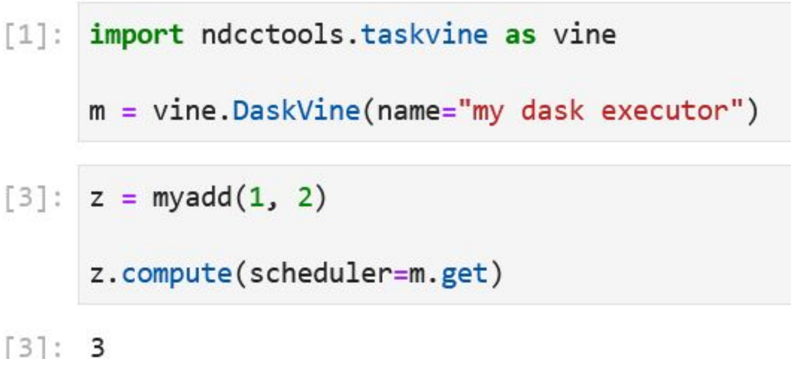
Using .compute()

dask data types and decorators to construct the task graph

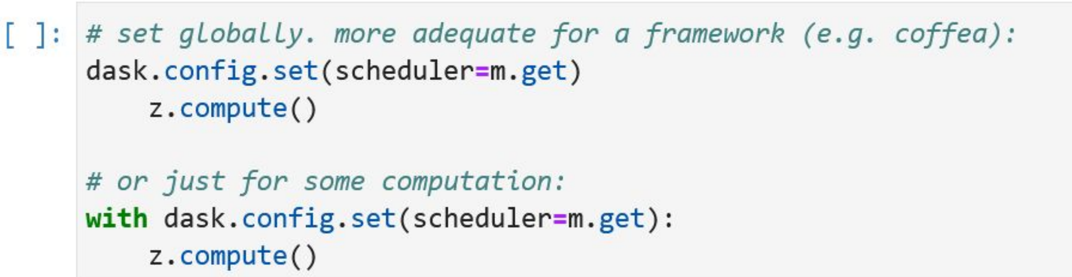
We don't explicitly refer to the task graph dict, only to *futures* that know their graph.



```
[3]: import dask

     @dask.delayed
     def myadd(x, y):
         return x + y

     z = myadd(1, 2)

     z.compute()

[3]: 3
```



```
[1]: import ndcctools.taskvine as vine

     m = vine.DaskVine(name="my dask executor")

[3]: z = myadd(1, 2)

     z.compute(scheduler=m.get)

[3]: 3
```

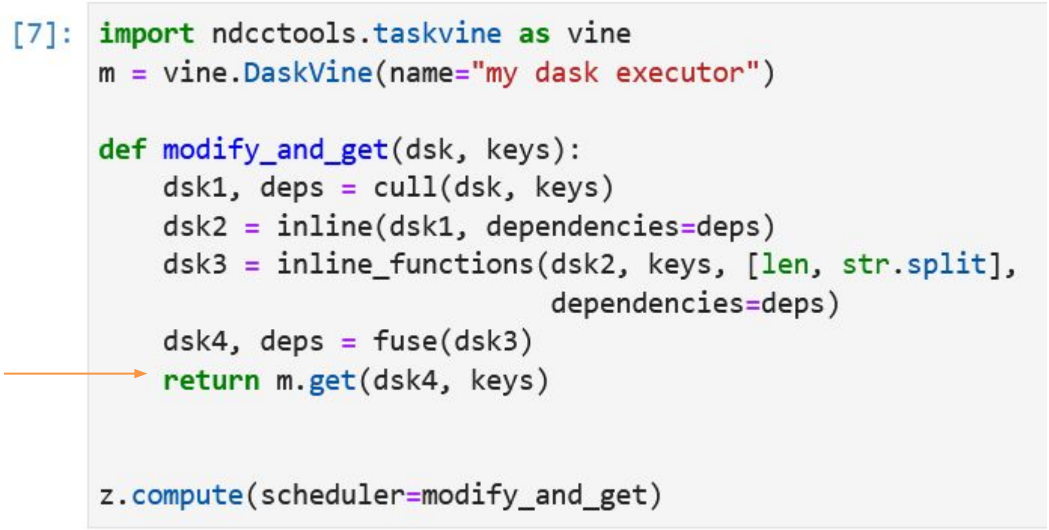


```
[ ]: # set globally. more adequate for a framework (e.g. coffea):
     dask.config.set(scheduler=m.get)
         z.compute()

     # or just for some computation:
     with dask.config.set(scheduler=m.get):
         z.compute()
```

Modifying the Task Graph

Change the graph
before sending it for execution.

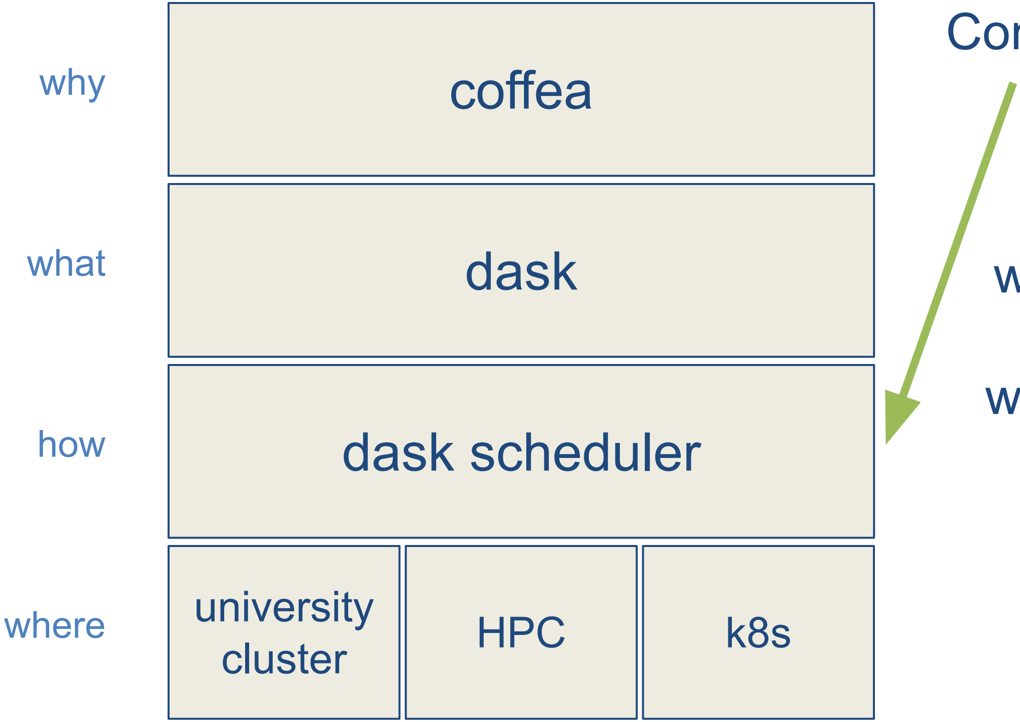
(E.g., merging ephemeral independent calls together, or splitting subcalls for improved parallelism.)



```
import ndcctools.taskvine as vine
m = vine.DaskVine(name="my dask executor")

def modify_and_get(dsk, keys):
    dsk1, deps = cull(dsk, keys)
    dsk2 = inline(dsk1, dependencies=deps)
    dsk3 = inline_functions(dsk2, keys, [len, str.split],
                            dependencies=deps)
    dsk4, deps = fuse(dsk3)
    return m.get(dsk4, keys)


z.compute(scheduler=modify_and_get)
```

https://docs.dask.org/en/latest/optimize.html

Questions I would like to explore



why
coffea

what
dask

how
dask scheduler

where
university cluster
HPC
k8s

Control **how** the computation should occur, e.g.:

how many resources to use?
which python environment to use?
how to activate environment?
what temp files should be cached?
what should be serialized?
how much should be serialized?
what should be retried?
how to retry?

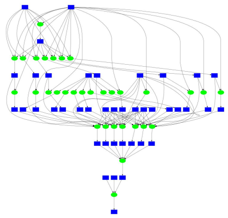
TaskVine

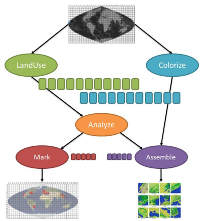
TaskVine Application Stack

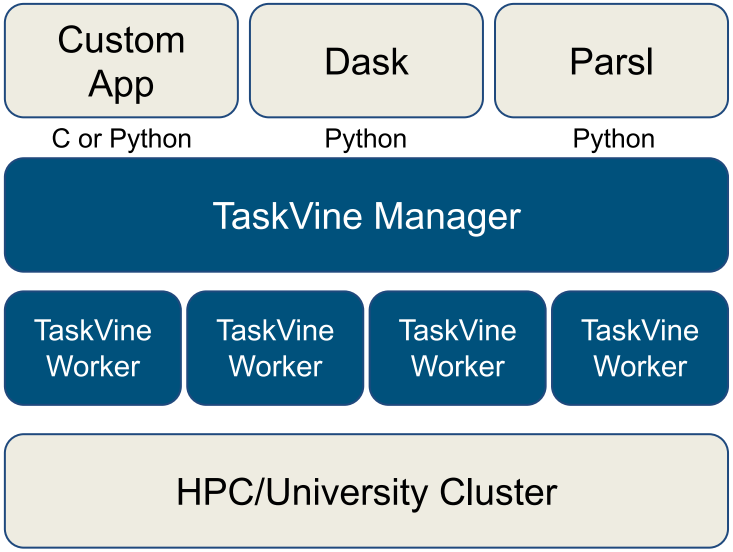
TaskVine (successor of Work Queue) is a system for executing **data intensive** scientific workflows on clusters, clouds, and grids from very small to massive scale.

TaskVine controls the **computation and storage** capability of a large number of workers, striving to carefully manage, transfer, and re-use data and software wherever possible.

```
import taskvine

file = URL(www)
m.submit(task)
task = m.wait(5)
```







Custom App | Dask | Parsl

C or Python | Python | Python

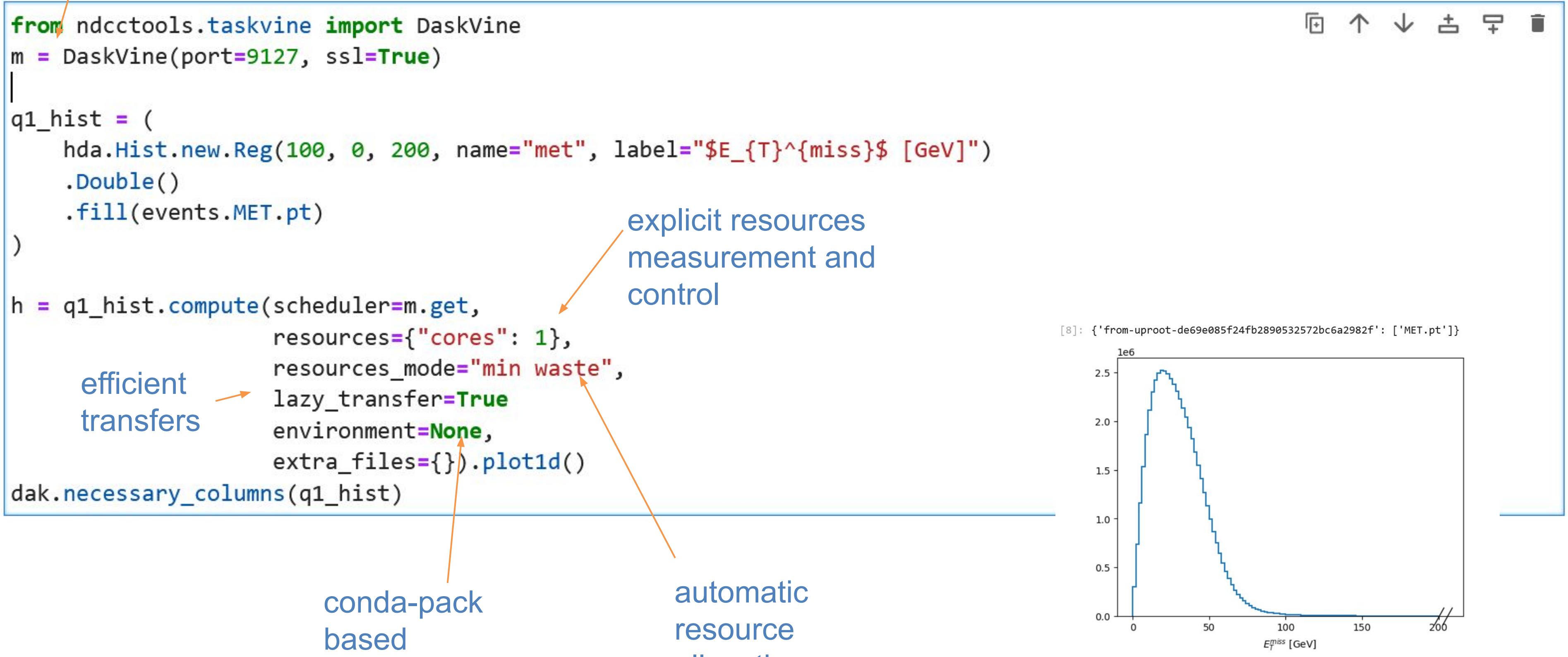
TaskVine Manager

TaskVine Worker | TaskVine Worker | TaskVine Worker | TaskVine Worker

HPC/University Cluster

TaskVine in Action

specialized manager to execute dask
only final results loaded into memory



```
from ndcctools.taskvine import DaskVine
m = DaskVine(port=9127, ssl=True)

q1_hist = (
    hda.Hist.new.Reg(100, 0, 200, name="met", label="$E_{T}^{miss}$ [GeV]")
    .Double()
    .fill(events.MET.pt)
)

h = q1_hist.compute(scheduler=m.get,
                    resources={"cores": 1},
                    resources_mode="min waste",
                    lazy_transfer=True
                    environment=None,
                    extra_files={}).plot1d()
dak.necessary_columns(q1_hist)
```

```
[8]: {'from-uproot-de69e085f24fb2890532572bc6a2982f': ['MET.pt']}
```

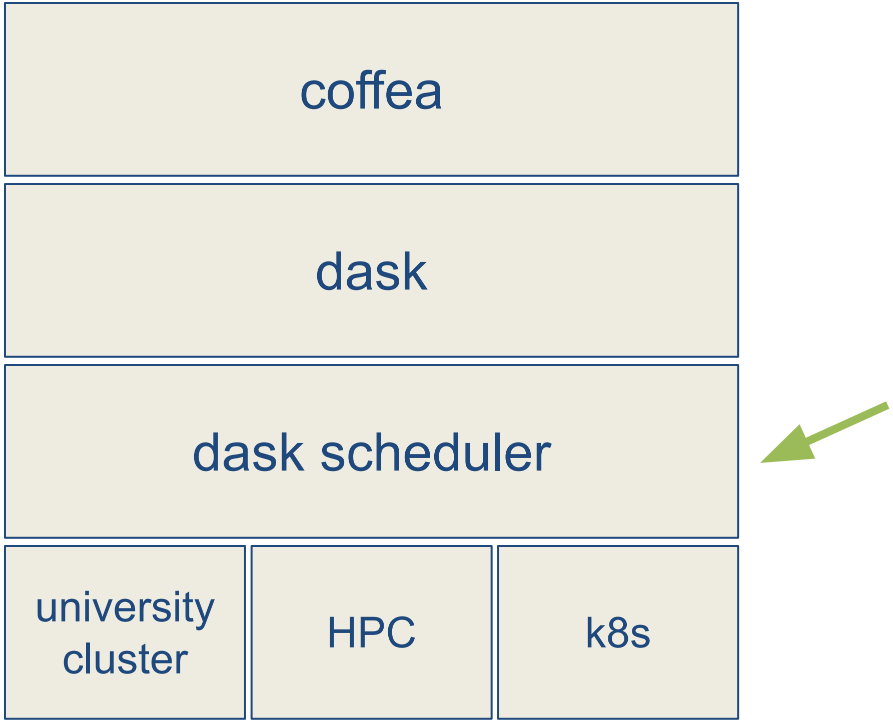
explicit resources
measurement and
control

efficient
transfers

conda-pack
based
env delivery

automatic
resource
allocation

Conclusions



coffea

dask

dask scheduler

university cluster

HPC

k8s

Advantageous place to be to control **how** the computations should occur.

Harness current computer science developments and research in workflow execution.

Harness previous experience on how to execute workflows at scale.

UNIVERSITY OF NOTRE DAME

CCTools

TaskVine

https://cctools.readthedocs.io

https://github.com/cooperative-computing-lab/cctools

```
conda install -c conda-forge ndcctools
```

Thanks to team and collaborators

ND CMS

Prof. Kevin Lannon
Prof. Mike Hildreth
Kelci Mohrman
Brent R. Yates
Andrew Wightman
John Lawrence
Andrea Trapote
Irena Johnson
Kenyi Hurtado

CCL

Prof. Douglas Thain
Thanh Son Phung
Barry Sly-Delgado
Colin Thomas
David Simonetti
Andrew Hennessee
Jachob Dolak

Other Collaborators

Parsl team

CRC
CENTER FOR RESEARCH COMPUTING

This work was supported by
NSF Award OAC-1931348