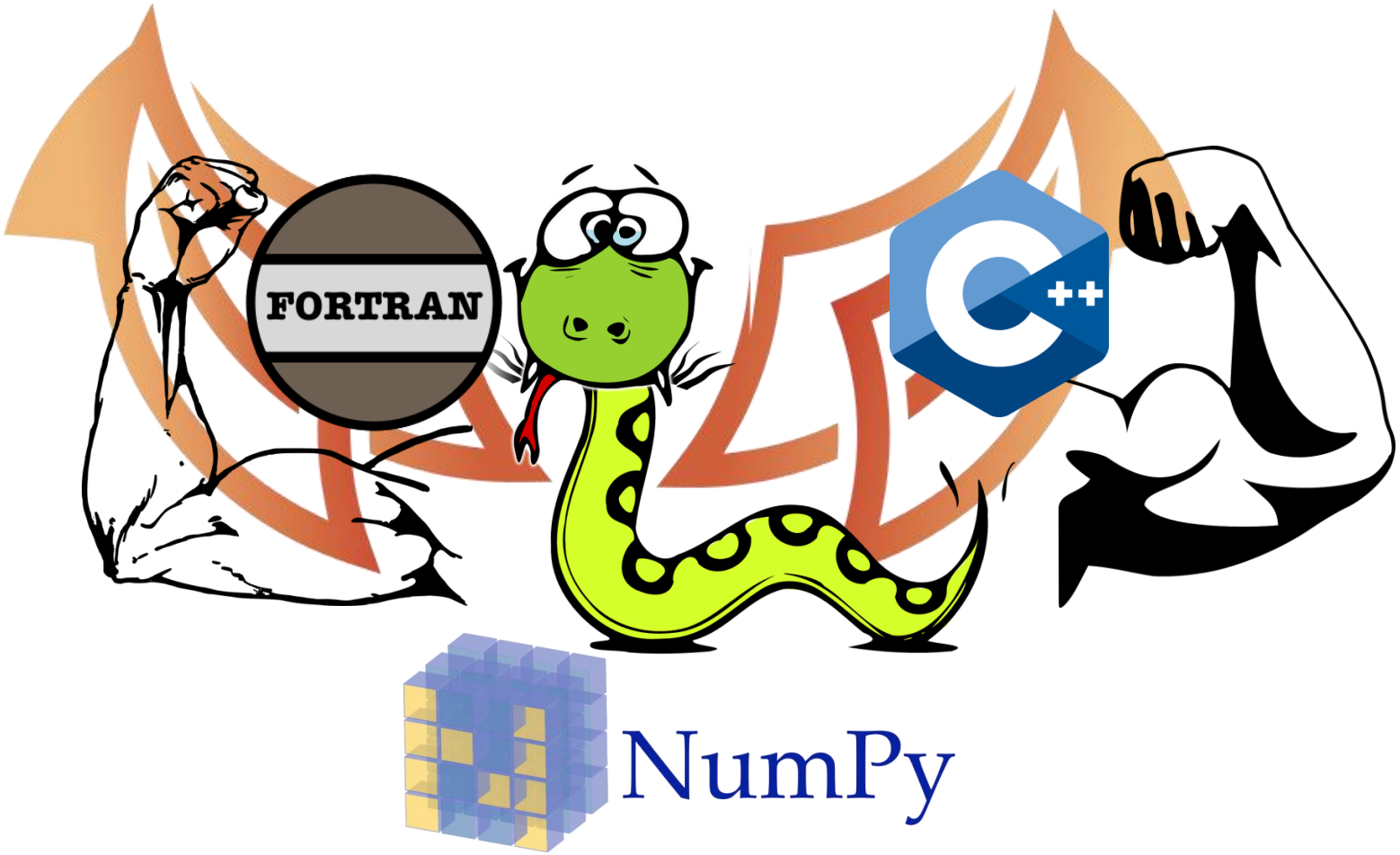
Parallelize Python using Dask

Yubo “Paul” Yang, THW-IL, 2019/11/20

Code available on GitHub: https://github.com/Paul-St-Young/thw-dask-para

Images from https://pixabay.com/vectors



Why parallelizing Python is problematic: The Global Interpreter Lock (**GIL**)

The GIL **prevents multiple threads** from executing CPython bytecodes at once.

This lock is necessary mainly because CPython's **memory management is not thread-safe.**

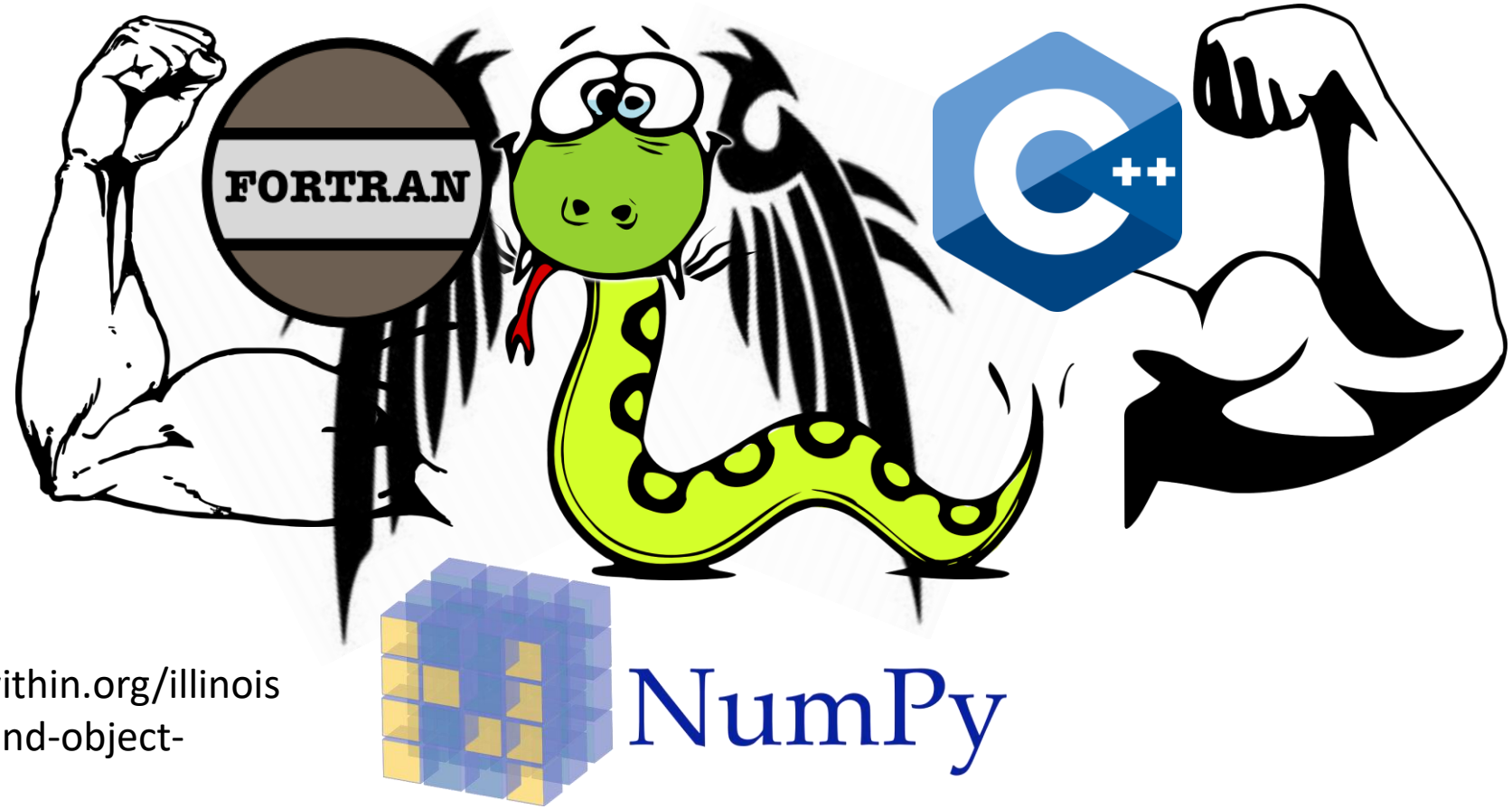
Take home: pure Python script is dragged down by the GIL to a single processing thread!

https://wiki.python.org/moin/GlobalInterpreterLock

Why parallelizing Python is problematic: The Global Interpreter Lock (**GIL**)

The GIL can be circumvented by calling low-level C/Fortran functions, which numpy does!

Problem: require significant coding, and still limited to a single machine.



See 2019/04/17 talk
http://www.thehackerwithin.org/illinois/posts/python-as-glue-and-object-oriented-programming

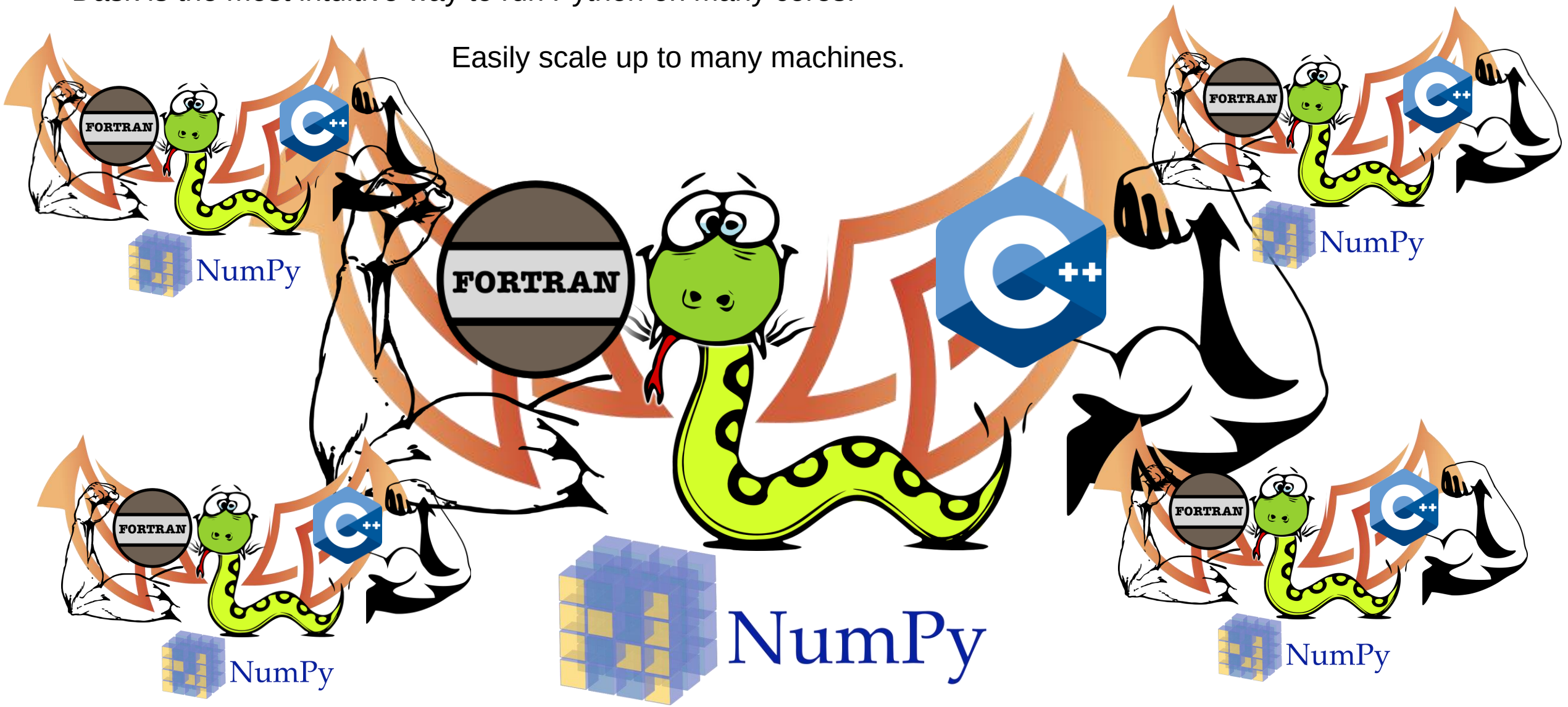
What is Dask? Graph execution engine

dask uses: psutil, multiprocessing, mpi4py, and sockets under the hood.

Dask is a graph execution engine.

Dask is the most intuitive way to run Python on many cores.

Easily scale up to many machines.



Why Dask?

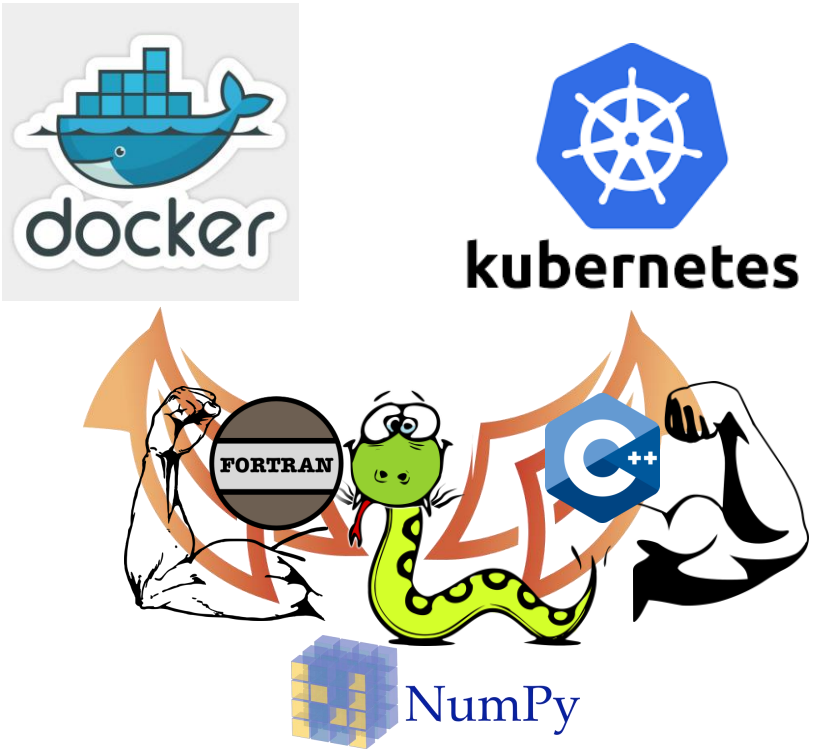
Reason 1: Dask stands on solid foundation

- "Dask does not seek to disrupt or displace the existing ecosystem, but rather to complement and benefit it from within."
- strong and diverse backers

Reason 2: Dask developers care

- well-written documentation: docs, distributed, ml, examples.dask.org
- easy to diagnose: dashboard, serial->supercomputer
- many video tutorials: showcase, dashboard, lab extension

Reason 3: Dask reaches the cloud



Supported By

We thank these institutions for generously supporting the project

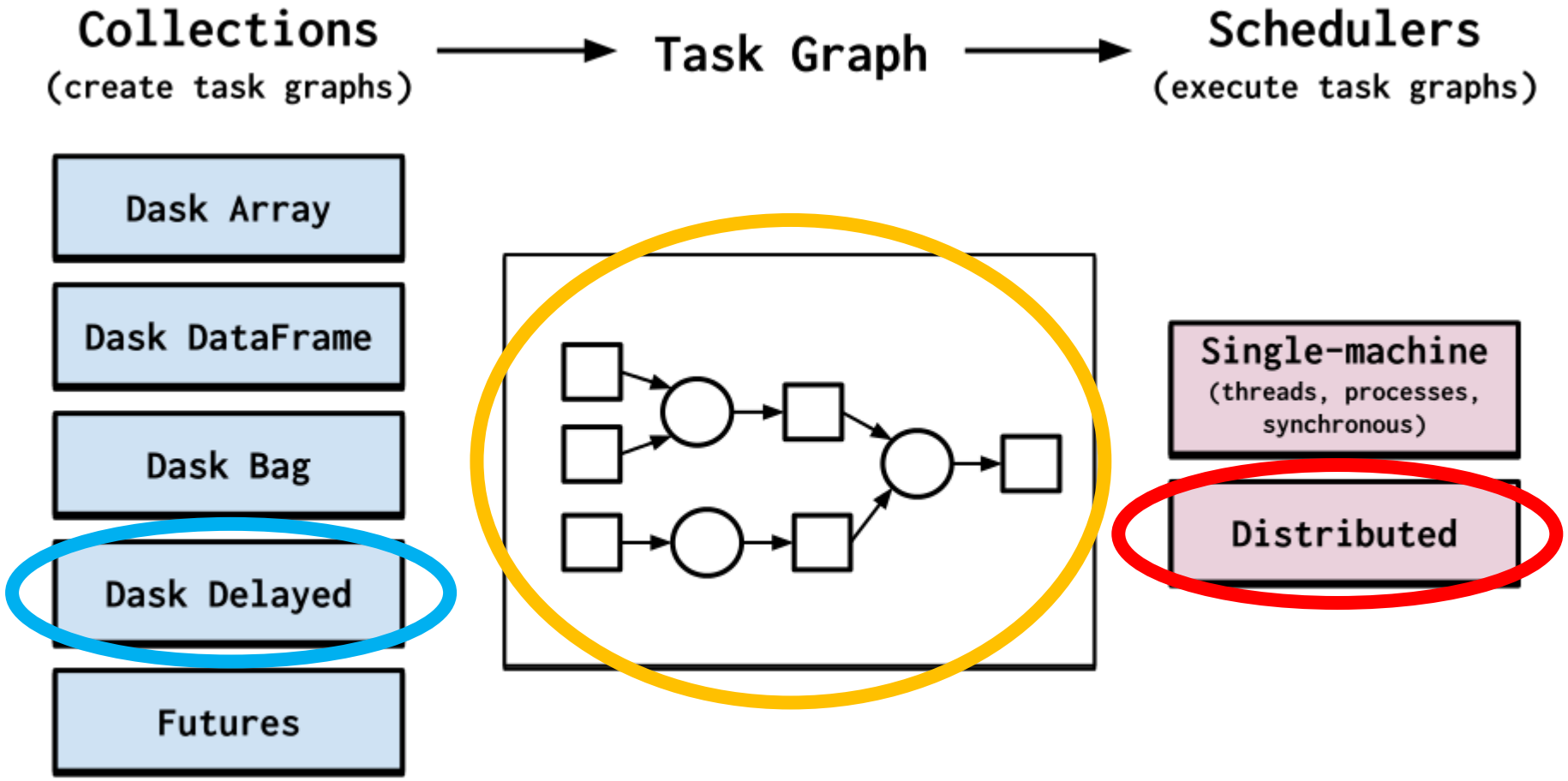
How to use Dask? Complete pipeline ≡ create task graph -> scheduler -> execute

USER INTERFACE
User Interfaces
Array
Bag
DataFrame
Delayed
Futures
Machine Learning
Best Practices
API

SCHEDULING
Scheduling
Distributed Scheduling

DIAGNOSTICS
Understanding Performance
Visualize task graphs
Diagnostics (local)
Diagnostics (distributed)

Start by making one choice for each step:
task graph builder: dask.delayed
task graph executioner: dask.distributed.Client(LocalCluster())



Step 1: Build and visualize the task graph

Use dask.delayed to lazy-evaluate costly computation. This constructs the task graph without doing any real computation.

A delayed object can be visualized

or executed.

```
%%time
results = []

for x in data:
    y = delayed(inc)(x)
    results.append(y)

total = delayed(sum)(results)
print("Before computing:", total)  # Let's see what type of thing total is
result = total.compute()
print("After computing :", result)  # After it's computed
```

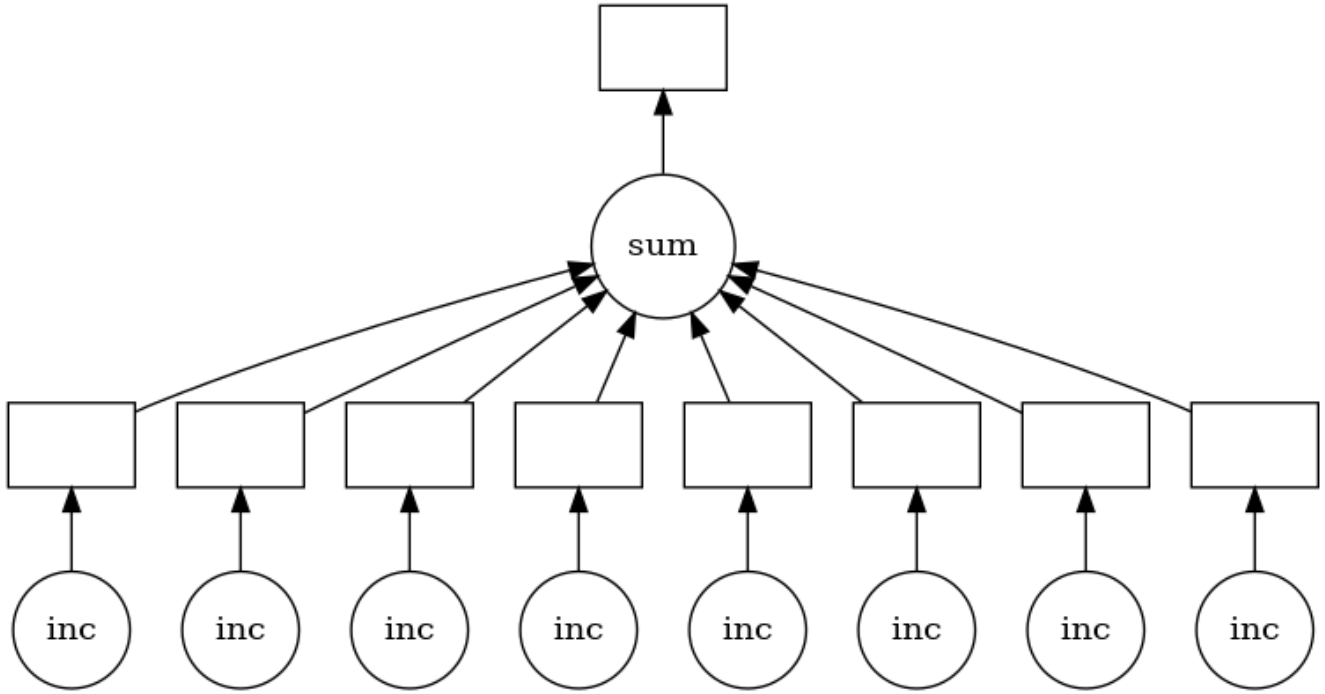
```
Before computing: Delayed('sum-f61ea7b0-b27f-4e1b-9507-3d0154fb6840')
After computing : 44
CPU times: user 14.9 ms, sys: 23.1 ms, total: 37.9 ms
Wall time: 1.03 s
```

```
total.visualize()
```

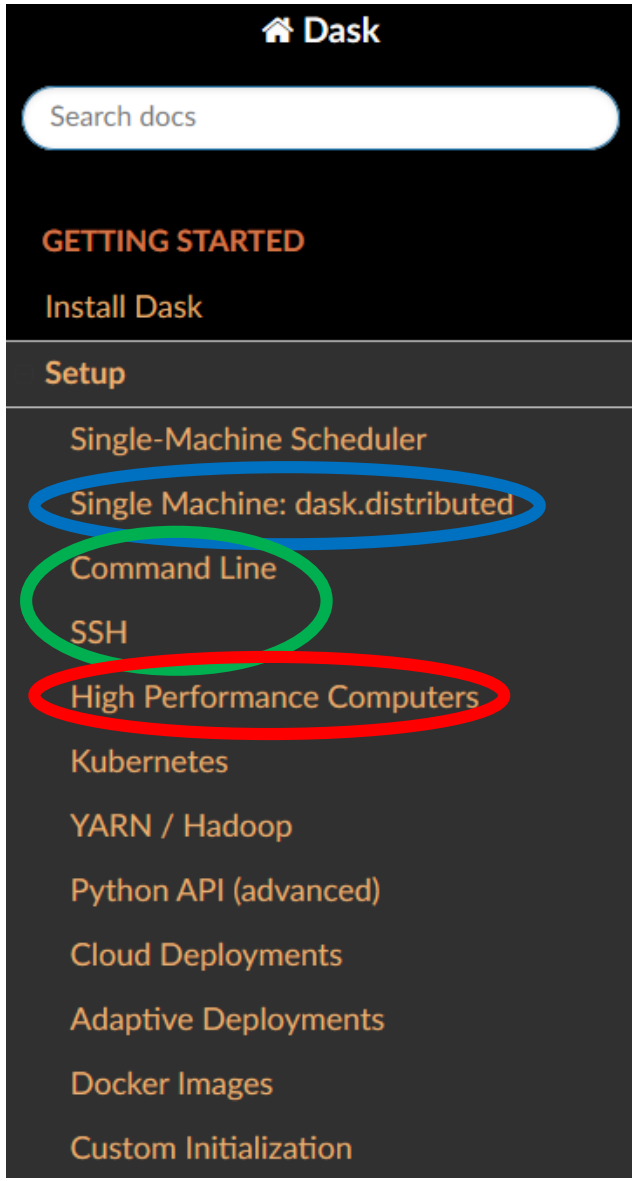
Example taken from dask tutorial

Tutorial 01 delayed task graph

https://github.com/dask/dask-tutorial/blob/master/01_dask.delayed.ipynb



Step 2: Construct a scheduler to efficiently execute the task graph



LocalCluster processes=False -> like pure OpenMP, one process saves memory and communication, but Python script must release the GIL!

LocalCluster processes=ncpu -> like pure MPI, no need to worry about GIL, but uses ncpu× memory

Client('127.0.0.1:8787') -> use a few loosely connected workstations to pool memory and CPUs together
Requires setup:
on local machine:
dask-scheduler
dask-worker [tcp://myscheduler:8787] --nprocs 8
on remote machine:
dask-worker [tcp://myscheduler:8787] --nprocs 8

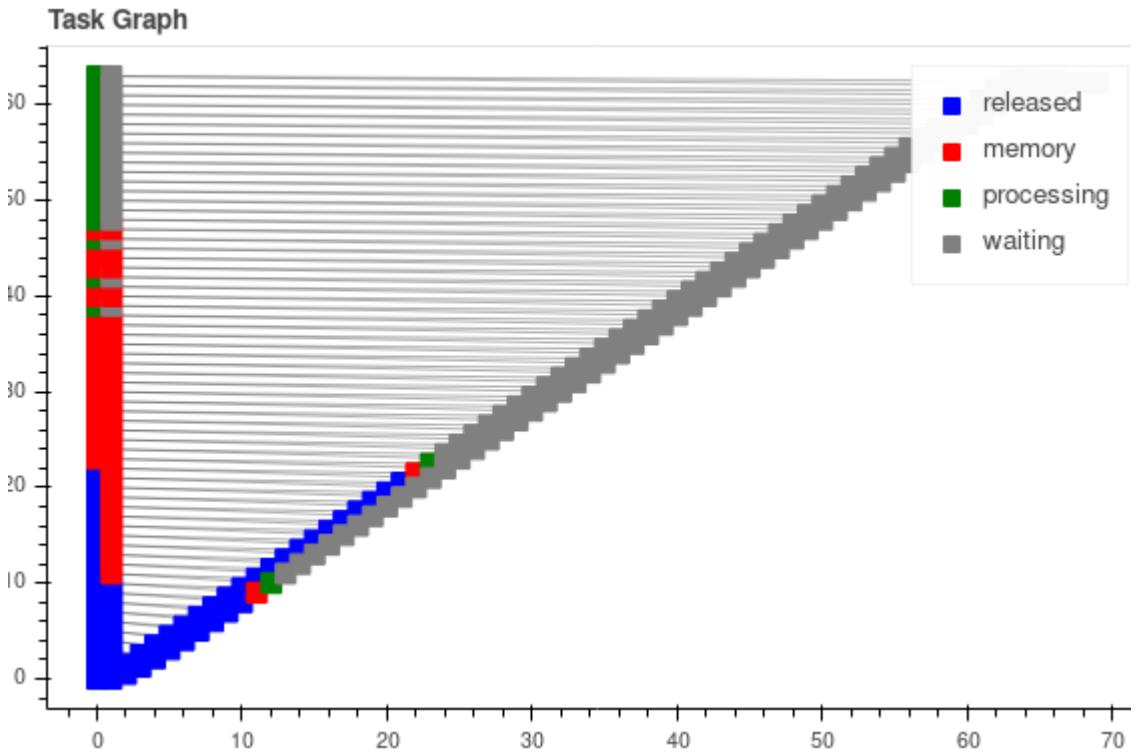
PBSCluster -> use a super computer with torque scheduler
cluster = PBSCluster(cores=16, memory="16GB", queue="secondary", walltime="00:15:00", interface="ib0")
cluster.scale(jobs=4) # get 4 nodes

Step 3: Use dashboard to profile and optimize

One task stream for each worker thread,
want to see lots of solid colors (computation).

Progress bar show headroom for more cores.

Task graph shows parallelism
inherit in computation.



Conclusions

Dask is an intuitive and flexible graph execution engine for Python.

Dask.delayed can easily parallelize existing Python code.

Dask.distributed.Client can use a variety of Clusters to suit the task at hand.
Eg. Laptop -> LocalCluster,
Workstations -> SSHCluster,
Supercomputer -> PBSCluster*
Cloud->KubeCluster

*note: use dask.future instead of dask.delayed if Client has a job submission system

Dask dashboard provides many tools to debug and profile parallel execution.

Resources

Videos by Matthew Rocklin

cmd setup cluster
dashboard
delayed

array, bag, database APIs

Tutorials

dask/dask-tutorial

Documentation

examples.dask.org
docs.dask.org
distributed.dask.org

Take-home: smash that GIL with f2py/numpy/pybind, then dask to clusters!

NumPy