UNIVERSITÀ DEGLI STUDI DI PADOVA

J. Pazzini
PADOVA UNIVERSITY

12 - DASK

Management and Analysis of Physics Datasets - Module B

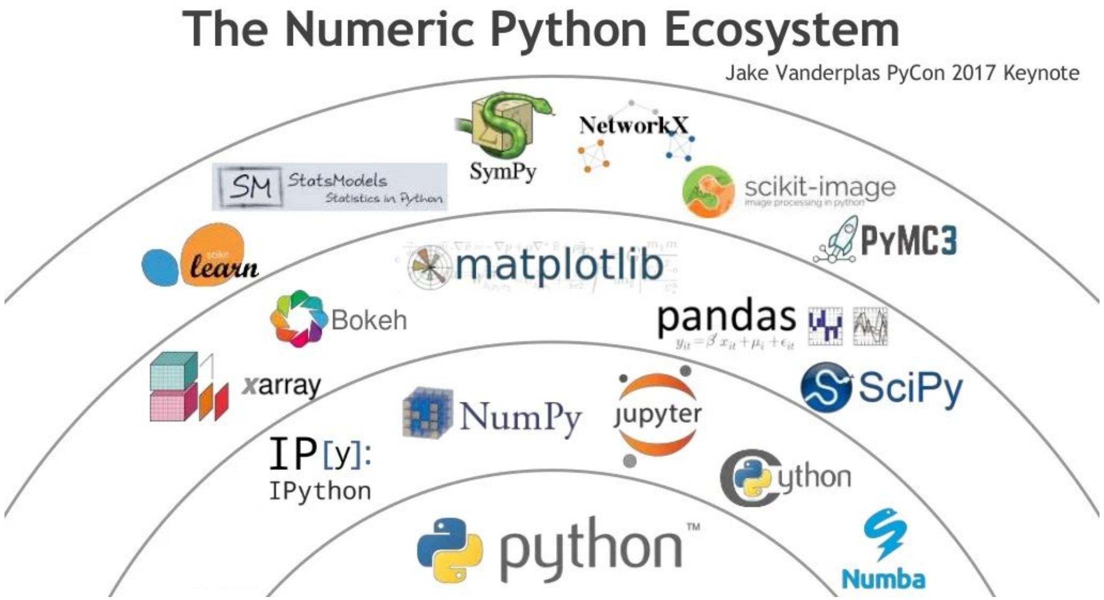
Physics of Data

A.A. 2023/2024

DASK

Open source library for parallel and distributed computing in Python

Developed starting around 2015 to address a main concern with the Python ecosystem:



Python has a strong analytics ecosystem
but
is mostly limited to single-core applications and in-memory datasets

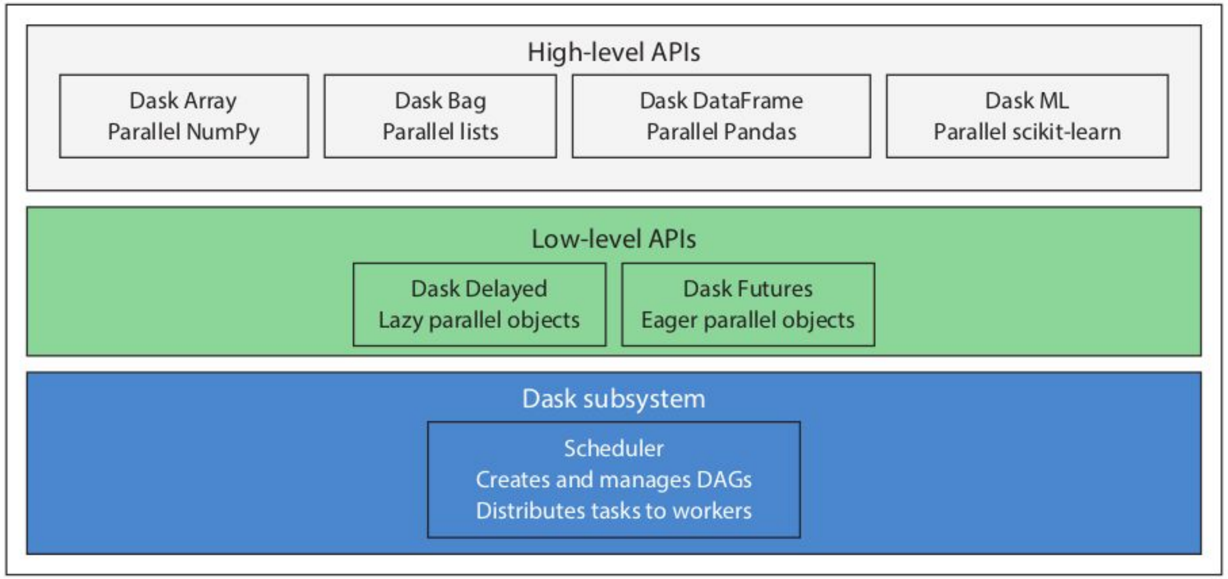
Matthew Rocklin (Dask main creator)

DASK

Python-based library built with two main functionalities in mind:

1. Efficient task scheduling for parallel and distributed computing on single-nodes / clusters / HPC. Including both distributed computation and job submission

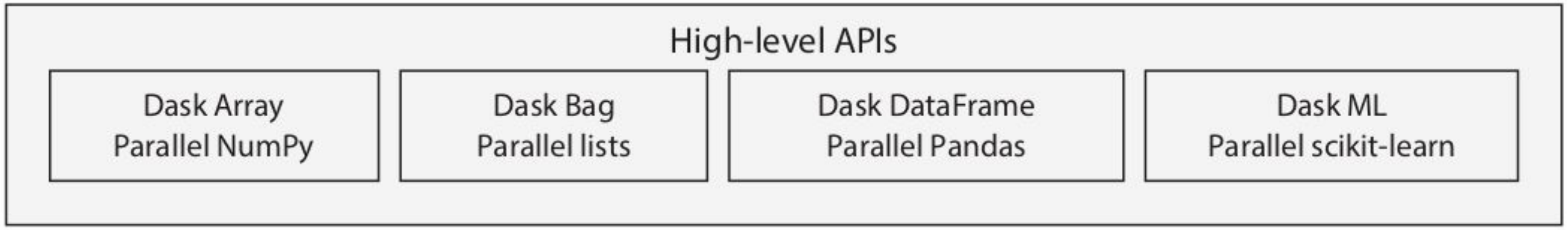
2. Familiar "big-data" collections for Python users, to scale python to larger-than-memory datasets



DASK - HIGH LEVEL API

High level → **distributed pythonic data collections**

Dask provides high-level distributed data collections that mimic Python lists, NumPy arrays, Pandas dataframes



The main idea behind the high-level data collection APIs is that Dask extends their plain-python equivalent for usage on parallel and distributed systems

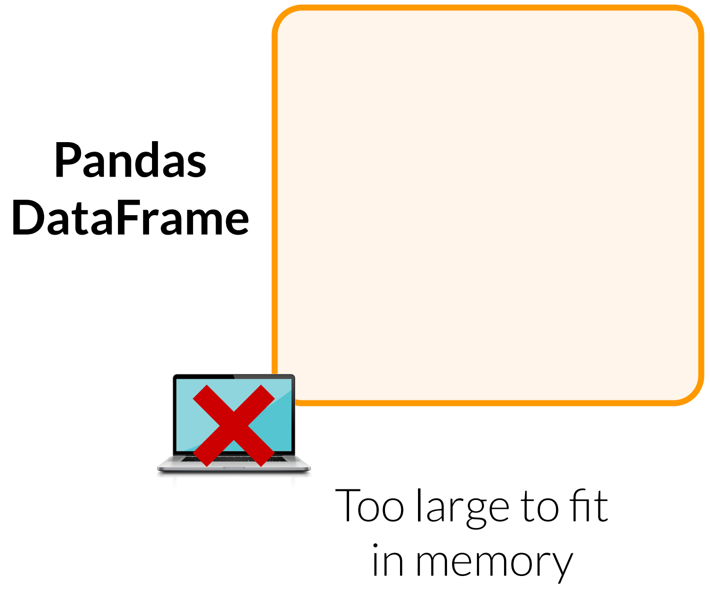
A Dask DataFrame is made up of several smaller Pandas DataFrames, a Dask Array is made up of smaller NumPy Arrays, etc

Each of the smaller underlying objects (a chunk, or a partition) can be stored in individual machines within a cluster, or queued up and worked on one piece at a time on a local node

DASK - HIGH LEVEL API

High level → **distributed pythonic data collections**

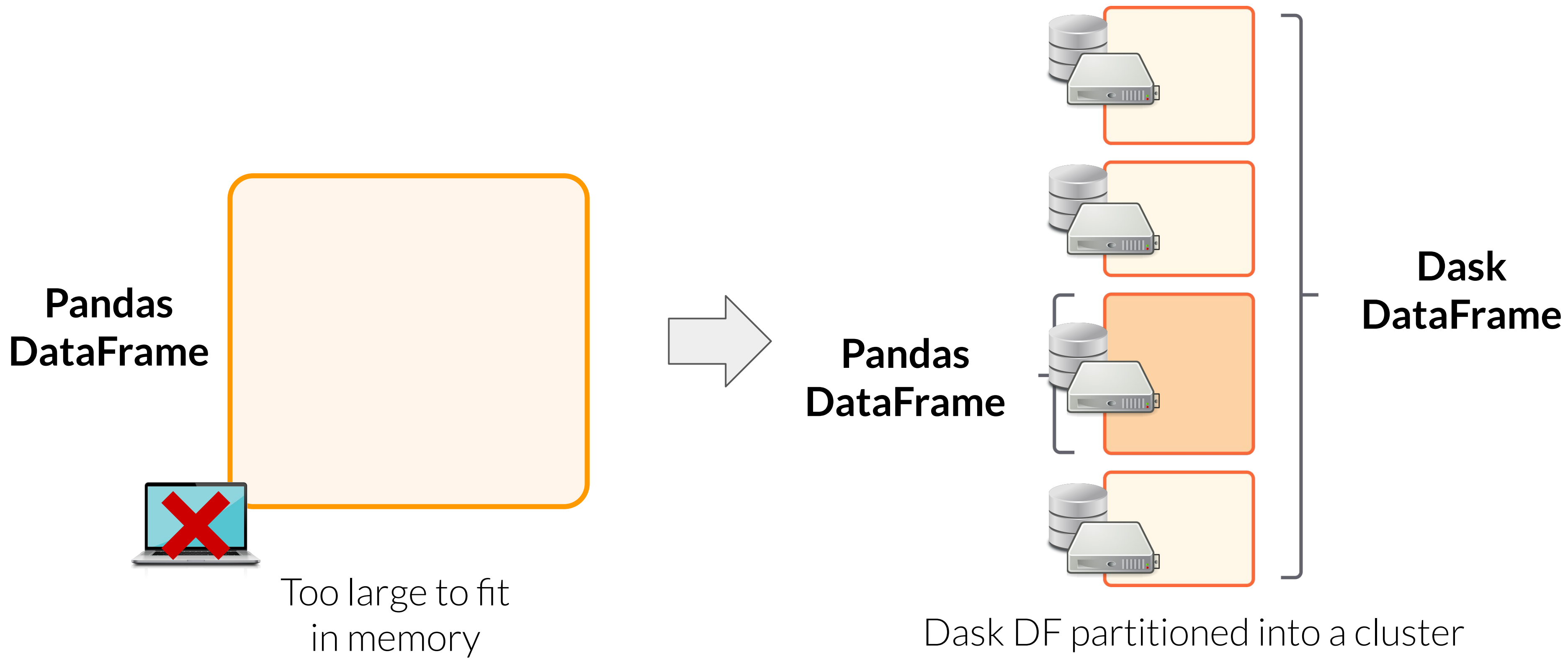
Dask provides high-level **distributed data collections that mimic Python** lists, NumPy arrays, Pandas dataframes



DASK - HIGH LEVEL API

High level → **distributed pythonic data collections**

Dask provides high-level **distributed data collections that mimic Python** lists, NumPy arrays, Pandas dataframes

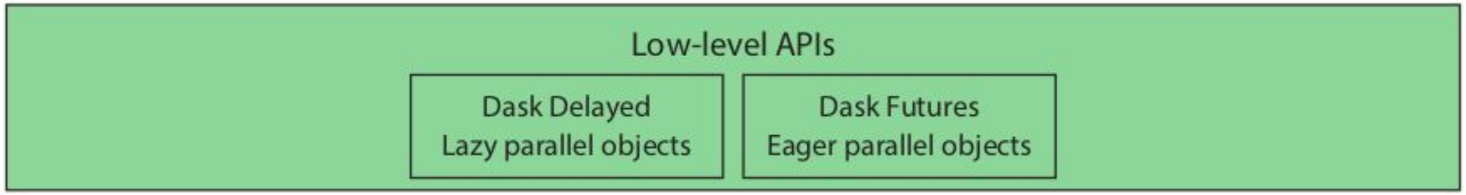


Pandas DataFrame
Too large to fit in memory
Pandas DataFrame
Dask DataFrame
Dask DF partitioned into a cluster

DASK - LOW LEVEL API

Low level → generic Python parallel & distributed computing objects

Dask provides high-granularity for the parallel/distributed execution of any Python code, regardless of the usage of the high-level APIs



Any Python instruction can be issued to Dask either for **lazy** or **eager** execution:

- Lazy → Not to be executed right away. The parallel execution of multiple Lazy operations can thus be optimized by the Dask task scheduler
- Eager → Executed immediately over the cluster. Mostly used to provide a rough equivalent of a *submit* function to a batch system

The execution of plain-Python code can be distributed with Dask by means of Low-level APIs but it's important to realize that a speed-up can be achieved only if the code is written in such a way to optimize the parallel execution

DASK - LOW LEVEL API

Low level → generic Python parallel & distributed computing objects

Parallelize and distribute any python code not necessarily relying on the map-reduce paradigm

Useful both for embarrassingly-parallel (SPMD) tasks, and to parallelize parts of a complex task

```
def my_python_function(x):
  ...
  return something
```

e.g. a custom Python function written to scan a part of a parameter space, or to compute metrics on a subset of data

```
res = delayed(my_python_function)(some_data)
```

Will **not execute**, but **wait and optimize** the execution of the code on the available computing resources

```
res.compute()
```

Will **effectively start the execution** of the task in parallel over a number of threads or nodes

DASK - LOW LEVEL API

Low level → generic Python parallel & distributed computing objects

Parallelize and distribute any python code not necessarily relying on the map-reduce paradigm

Useful both for embarrassingly-parallel (SPMD) tasks, and to parallelize parts of a complex task

```
def my_python_function(x):
  ...
  return something
```

e.g. a custom Python function written to scan a part of a parameter space, or to compute metrics on a subset of data

```
p = Client.submit(my_python_function, some_data)
```

Will **immediately submit the task to the cluster**.
'`p`' is the "promise" of the results, that will be stored and held into the cluster after the execution, like in a batch system

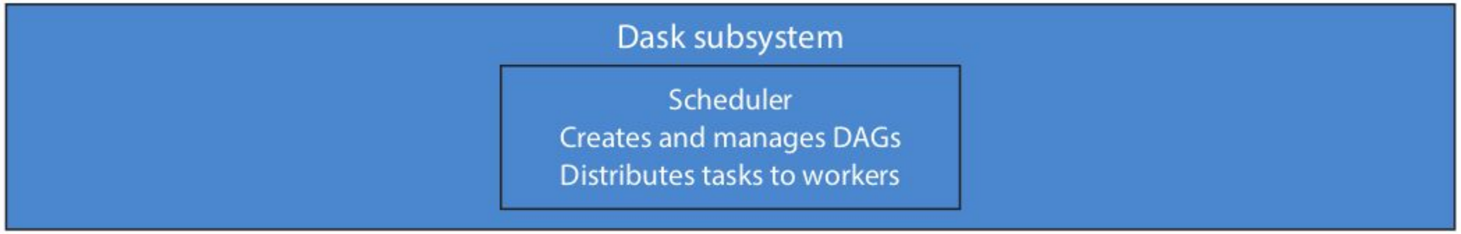
```
Client.gather(p)
```

Will **retrieve the results from the cluster**

DASK - DAG SCHEDULER

Core level → DAG scheduler and cluster resource manager

At its core, Dask is a **D**istributed t**ASK** scheduler, optimized for the deployment of python code in a number of parallel and distributed computing environments



The tasks are optimized via the Dask DAG scheduler

Dask also includes dynamic task schedulers to deploy and execute the task graphs in parallel

- **Single-machine scheduler**: to work on a pool of processors/threads on a single machine (does not scale)

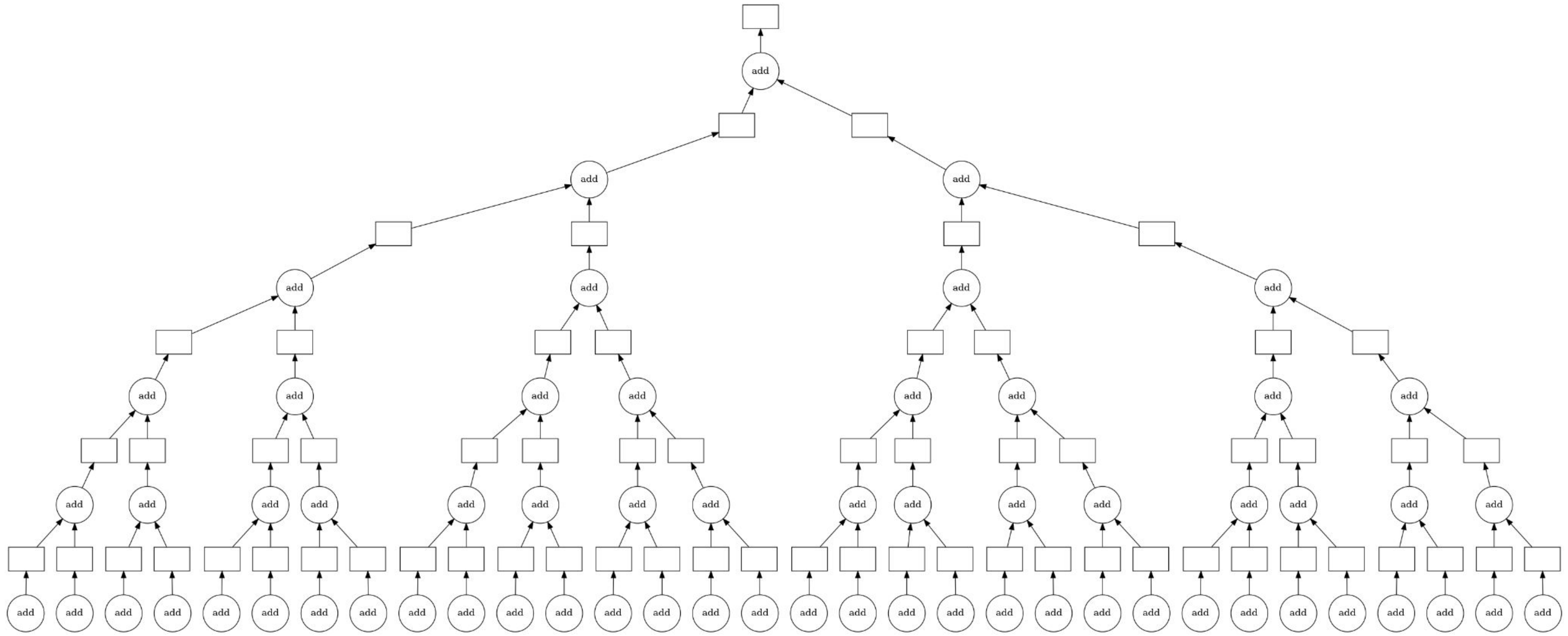
- **Distributed scheduler**: offers features to run on a set of custom nodes, or to use a batch system such as SLURM or PBS.

DASK - DAG SCHEDULER

Core level → DAG scheduler and cluster resource manager

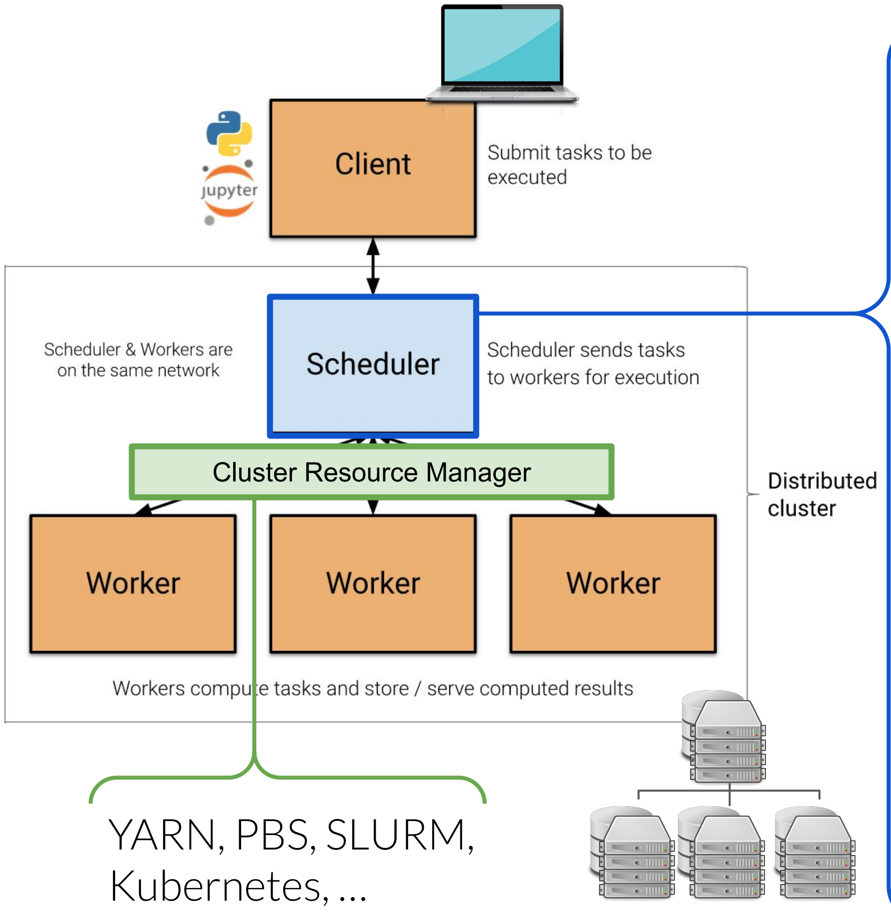
```
res = delayed(my_python_function)(some_data)
```

At this stage Dask computes the task DAG, before passing the execution to the scheduler



DASK - DAG SCHEDULER

Core level → DAG scheduler and cluster resource manager


Client
Submit tasks to be executed
jupyter
Scheduler & Workers are on the same network
Scheduler
Scheduler sends tasks to workers for execution
Cluster Resource Manager
Distributed cluster
Worker
Worker
Worker
Workers compute tasks and store / serve computed results

YARN, PBS, SLURM, Kubernetes, …

Local Threads
Default scheduler for single machine deployment (no setup required)

Local Processes
Simple scheduler for local machine, more heavy-weight than thread-based scheduler

Single-threaded synchronous
No parallel processing (for debugging purposes)

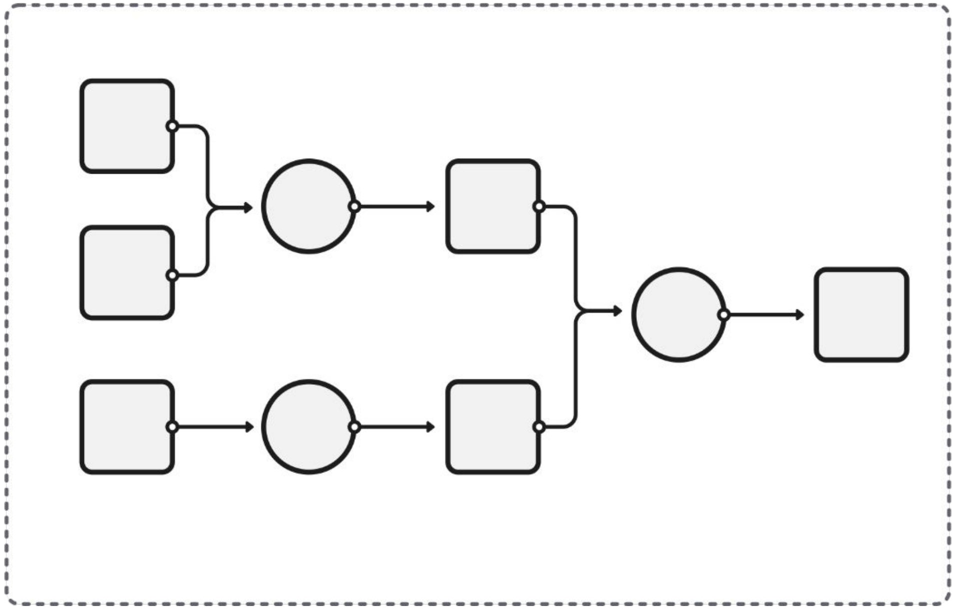
Distributed
The main scheduler for job deployment on distributed systems such as a cluster of worker nodes.
Despite its name, it can also be run locally on a single machine, with the "worker nodes" being defined as the pool of available cores (similar to multiprocessing/threading)

DASK WORKFLOW

Collections → **Task Graph** → **Schedulers**

(create task graphs) (execute task graphs)

- Dask Array
- Dask DataFrame
- Dask Bag
- Dask Delayed
- Futures



- Single-machine (threads, processes, synchronous)
- Distributed

DASK VS SPARK

Despite some important differences, both can be used to run the same applications

Spark is adopted vastly in the industry, due to its all-in-one solution framework and the support of SQL

Dask popularity is steadily rising for scientific and data-science applications, based on the Python ecosystem

dask

Written in Python and only really supports Python. It has quite some deficits in the management of memory

Not a framework, mostly just a scheduler. But… a component of the larger Python ecosystem

Dask internals are focused on "lower level". Fundamentally based on generic task scheduling

Dask DataFrame reuses the Pandas API (no SQL nor query optimizers)

APACHE Spark

Written in Scala and based on Java VMs executors

All-in-one framework, well integrated with other Apache projects. Recently a strong effort towards the Python API

Spark is aimed at "higher level" all-around tasks, providing good optimizations for simpler "Map-Reduce"-like computations

Vast set of functionalities for structured datasets, including SQL language and an efficient query optimizer