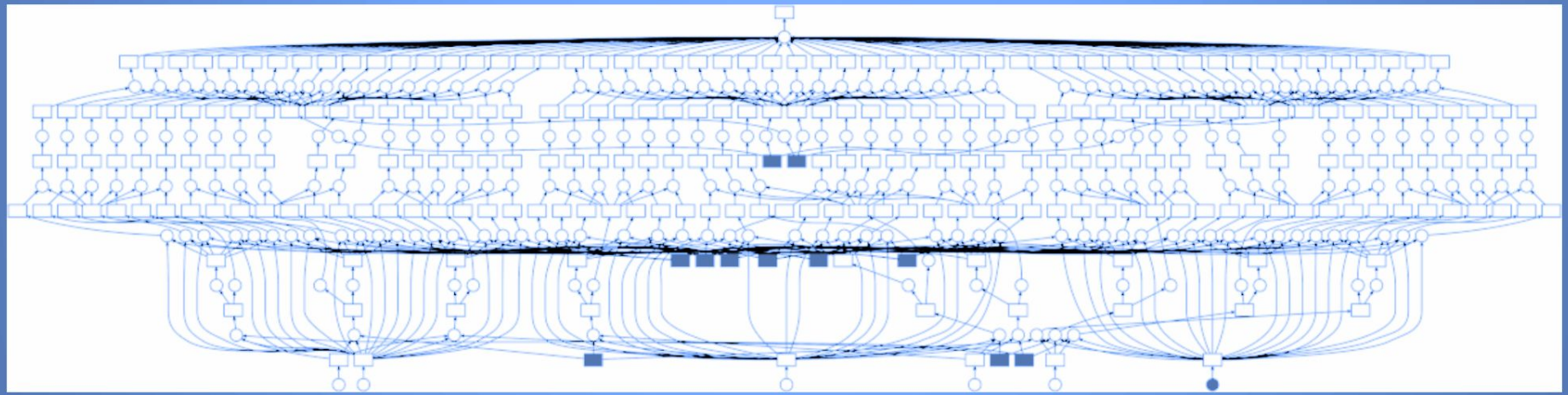
# Intro to Dask

Serafina Di Gioia
Postdoctoral researcher in ML @ICTP

# Low Level: Dask is a task scheduler

Like make, but where each task is a short Python function

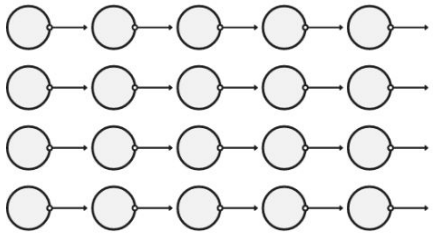
```
(X + X.T) - X.mean(axis=0)  # Dask code turns into task graphs
```



# Approaches to task scheduling

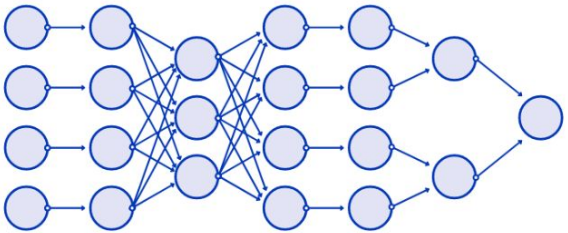
**Embarrassingly Parallel**

Hadoop/Spark/Dask/Airflow/Prefect



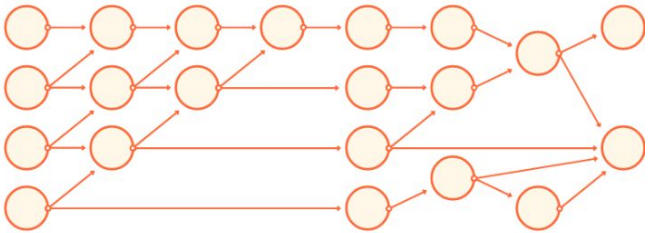
**MapReduce**

Hadoop/Spark/Dask



**Full Task Scheduling**

Dask/Airflow/Prefect



# Dask approach to task scheduling

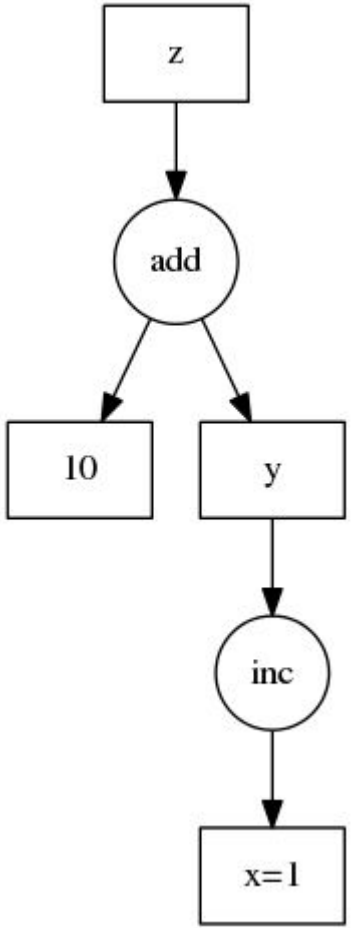
## Task Graphs

Internally, Dask encodes algorithms in a simple format involving Python dicts, tuples, and functions. This graph format can be used in isolation from the dask collections. Working directly with dask graphs is rare, though, unless you intend to develop new modules with Dask. Even then, dask.delayed is often a better choice. If you are a *core developer*, then you should start here.

- Specification
- Custom Graphs
- Optimization
- Advanced graph manipulation
- Custom Collections
- High Level Graphs

# Example

```
import dask

@dask.delayed
def inc(x):
   return x + 1

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

a = inc(1)       # no work has happened yet
b = inc(2)       # no work has happened yet
c = add(a, b)    # no work has happened yet

c = c.compute()  # This triggers all of the above computations
```



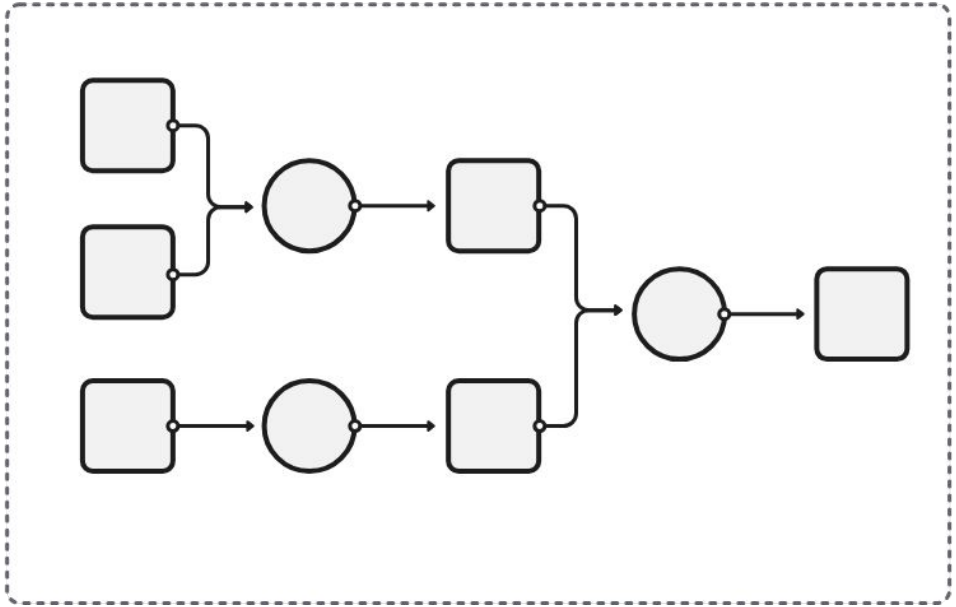
# High-level Dask

## Collections

(create task graphs)

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

## Task Graph



## Schedulers

(execute task graphs)

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

# High Level: Dask scales other Python libraries

- Pandas + Dask

```
# df = pandas.read_csv('my-file.csv')
df = dask.dataframe.read_csv('s3://path/to/*.csv')
df.groupby(df.timestamp.dt.hour).value.mean()
```

- Numpy + Dask

```
# X = numpy.random.random((1000, 1000))
X = dask.array.random((100000, 100000), chunks=(1000, 1000))
(X + X.T) - X.mean(axis=0)
```

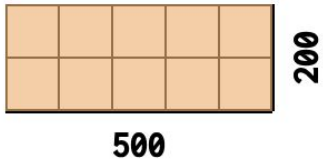
- Scikit-Learn + Dask + ...

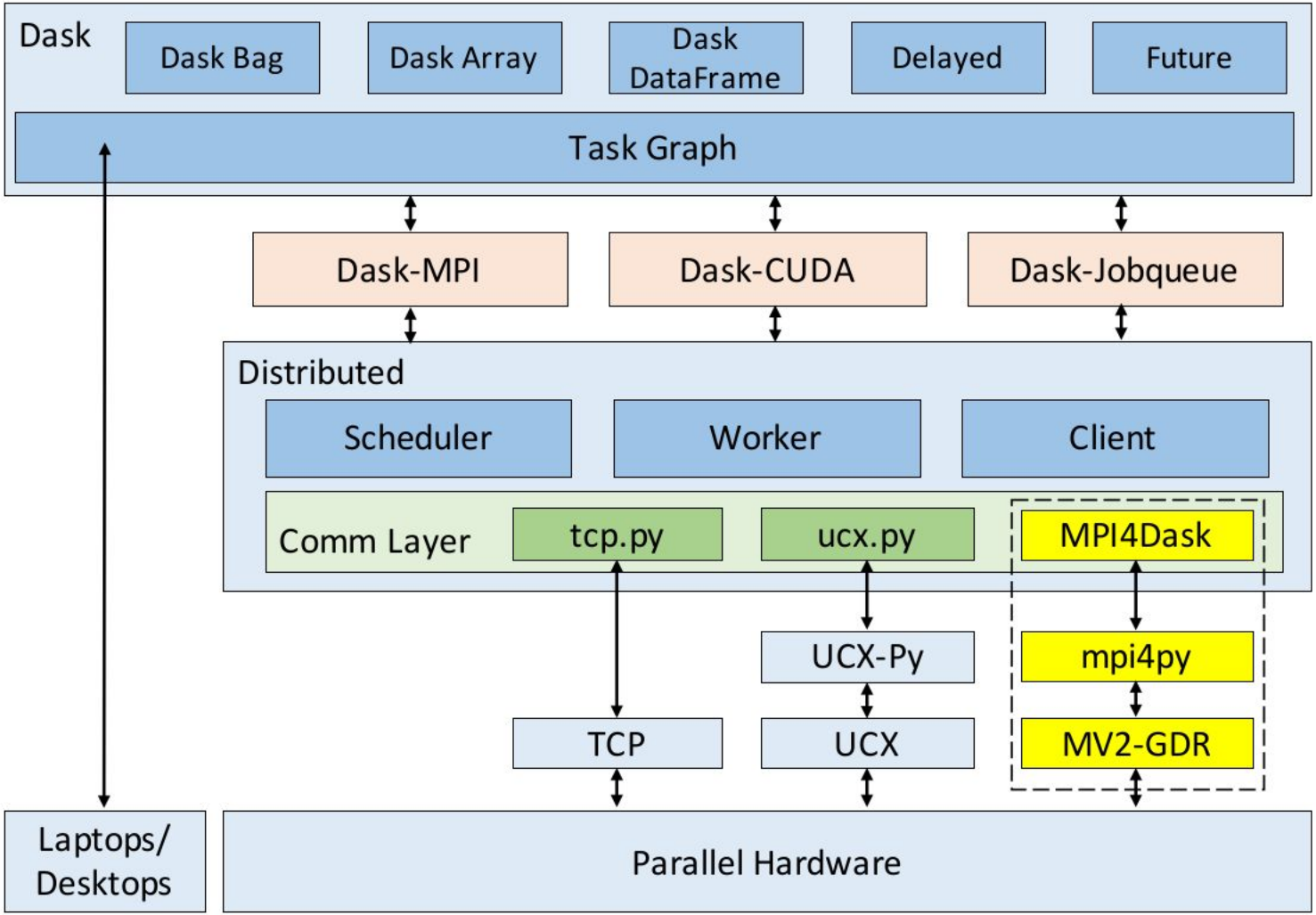
```
from scikit_learn.linear_models import LogisticRegression
from dask_ml.linear_models import LogisticRegression

model = LogisticRegression()
model.fit(X, y)
```

- ... and several other applications throughout PyData

```
import numpy as np
import dask.array as da

data = np.arange(100_000).reshape(200, 500)
a = da.from_array(data, chunks=(100, 100))
a
```

| | Array | Chunk |
|---|---|---|
| Bytes | 781.25 kiB | 78.12 kiB |
| Shape | (200, 500) | (100, 100) |
| Dask graph | 10 chunks in 1 graph layer | |
| Data type | int64 numpy.ndarray | |




Dask
Dask Bag
Dask Array
Dask DataFrame
Delayed
Future
Task Graph
Dask-MPI
Dask-CUDA
Dask-Jobqueue
Distributed
Scheduler
Worker
Client
Comm Layer
tcp.py
ucx.py
MPI4Dask
UCX-Py
mpi4py
TCP
UCX
MV2-GDR
Laptops/ Desktops
Parallel Hardware

# Dask.distributed

It serves to complement the existing PyData analysis stack. In particular it meets the following needs:

- **Low latency:** Each task suffers about 1ms of overhead. A small computation and network roundtrip can complete in less than 10ms.
- **Peer-to-peer data sharing:** Workers communicate with each other to share data. This removes central bottlenecks for data transfer.
- **Complex Scheduling:** Supports complex workflows (not just map/filter/reduce) which are necessary for sophisticated algorithms used in nd-arrays, machine learning, image processing, and statistics.
- **Pure Python:** Built in Python using well-known technologies. This eases installation, improves efficiency (for Python users), and simplifies debugging.
- **Data Locality:** Scheduling algorithms cleverly execute computations where data lives. This minimizes network traffic and improves efficiency.
- **Familiar APIs:** Compatible with the concurrent.futures API in the Python standard library. Compatible with dask API for parallel algorithms
- **Easy Setup:** As a Pure Python package distributed is `pip` installable and easy to set up on your own cluster.

# Setup Dask.distributed the Easy Way

If you create a client without providing an address it will start up a local scheduler and worker for you.

```
>>> from dask.distributed import Client
>>> client = Client()  # set up local cluster on your laptop
>>> client
<Client: scheduler="127.0.0.1:8786" processes=8 cores=8>
```

# Setup Dask.distributed the Hard Way

This allows dask.distributed to use multiple machines as workers.

Set up scheduler and worker processes on your local computer:

```
$ dask scheduler
Scheduler started at 127.0.0.1:8786

$ dask worker 127.0.0.1:8786
$ dask worker 127.0.0.1:8786
$ dask worker 127.0.0.1:8786
```

# Map and Submit Functions

Use the `map` and `submit` methods to launch computations on the cluster. The `map/submit` functions send the function and arguments to the remote workers for processing. They return `Future` objects that refer to remote data on the cluster. The `Future` returns immediately while the computations run remotely in the background.

```
>>> def square(x):
        return x ** 2

>>> def neg(x):
        return -x

>>> A = client.map(square, range(10))
>>> B = client.map(neg, A)
>>> total = client.submit(sum, B)
>>> total.result()
-285
```

# Dask dashboard

By default, when starting a scheduler on your local machine the dashboard will be served at `http://localhost:8787/status`. You can type this address into your browser to access the dashboard, but may be directed elsewhere if port 8787 is taken. You can also configure the address using the `dashboard_address` parameter (see **`LocalCluster`**).

There are numerous diagnostic plots available. In this guide you'll learn about some of the most commonly used plots shown on the entry point for the dashboard:

- Bytes Stored and Bytes per Worker: Cluster memory and Memory per worker
- Task Processing/CPU Utilization/Occupancy/Data Transfer: Tasks being processed by each worker/ CPU Utilization per worker/ Expected runtime for all tasks currently on a worker.
- Task Stream: Individual task across threads.
- Progress: Progress of a set of tasks.

# Dashboard for memory profiling

**process**

Overall memory used by the worker process (RSS), as measured by the OS

**managed**

Sum of the `sizeof` of all Dask data stored on the worker, excluding spilled data.

**unmanaged**

Memory usage that Dask is not directly aware of. It is estimated by subtracting managed memory from the total process memory and typically includes:

- The Python interpreter code, loaded modules, and global variables
- Memory temporarily used by running tasks
- Dereferenced Python objects that have not been garbage-collected yet
- Unused memory that the Python memory allocator did not return to libc through free yet
- Unused memory that the user-space libc free function did not release to the OS yet (see memory allocators below)
- Memory fragmentation
- Memory leaks

**unmanaged recent**

Unmanaged memory that has appeared within the last 30 seconds. This is not included in the ‘unmanaged’ memory