DASK FOR SCALABLE COMPUTING CHEAT SHEET

Visit the Dask homepage: **http://dask.org/**

These instructions use the conda environment manager. Get yours at **http://bit.ly/getconda**

DASK QUICK INSTALL	
Install Dask with conda	`conda install dask`
Install Dask with pip	`pip install dask[complete]`
USER INTERFACES	**EASY TO USE BIG DATA COLLECTIONS**
DASK DATAFRAMES	**SCALABLE PANDAS DATAFRAMES FOR LARGE DATA**
Import	`import dask.dataframe as dd`
Read CSV data	`df = dd.read_csv('my-data.*.csv')`
Read Parquet data	`df = dd.read_parquet('my-data.parquet')`
Filter and manipulate data with Pandas syntax	`df['z'] = df.x + df.y`
Standard groupby aggregations, joins, etc.	`result = df.groupby(df.z).y.mean()`
Compute result as a Pandas dataframe	`out = result.compute()`
Or store to CSV, Parquet, or other formats	`result.to_parquet('my-output.parquet')`

EXAMPLE	`df = dd.read_csv('filenames.*.csv')`
	`df.groupby(df.timestamp.day)\`
	`  .value.mean().compute()`

DASK ARRAYS	**SCALABLE NUMPY ARRAYS FOR LARGE DATA**
Import	`import dask.array as da`
Create from any array-like object	`import h5py` `dataset = h5py.File('my-data.hdf5')['/group/dataset']`
Including HFD5, NetCDF, Zarr, or other on-disk formats.	`x = da.from_array(dataset, chunks=(1000, 1000))`
Alternatively generate an array from a random distribution.	`da.random.uniform(shape=(1e4, 1e4), chunks=(100, 100))`
Perform operations with NumPy syntax	`y = x.dot(x.T - 1) - x.mean(axis=0)`
Compute result as a NumPy array	`result = y.compute()`
Or store to HDF5, NetCDF or other on-disk format	`out = f.create_dataset(...)` `x.store(out)`

EXAMPLE	`with h5py.File('my-data.hdf5') as f:`
	`  x = da.from_array(f['/path'], chunks=(1000, 1000))`
	`  x -= x.mean(axis=0)`
	`  out = f.create_dataset(...)`
	`  x.store(out)`

DASK BAGS	**PARELLEL LISTS FOR UNSTRUCTURED DATA**
Import	`import dask.bag as db`
Create Dask Bag from a sequence	`b = db.from_sequence(seq, npartitions)`
Or read from text formats	`b = db.read_text('my-data.*.json')`

Map and filter results	`import json`
	`records = b.map(json.loads)`
	`           .filter(lambda d: d["name"] == "Alice")`

Compute aggregations like mean, count, sum	`records.pluck('key-name').mean().compute()`
Or store results back to text formats	`records.to_textfiles('output.*.json')`

EXAMPLE	`db.read_text('s3://bucket/my-data.*.json')`
	`  .map(json.loads)`
	`  .filter(lambda d: d["name"] == "Alice")`
	`  .to_textfiles('s3://bucket/output.*.json')`

ANACONDA.

CONTINUED ON BACK

DASK COLLECTIONS (CONTINUED)

ADVANCED

Read from distributed file systems or cloud storage	`df = dd.read_parquet('s3://bucket/myfile.parquet')`
Prepend prefixes like hdfs://, s3://, or gcs:// to paths	`b = db.read_text('hdfs:///path/to/my-data.*.json')`
Persist lazy computations in memory	`df = df.persist()`
Compute multiple outputs at once	`dask.compute(x.min(), x.max())`

CUSTOM COMPUTATIONS — FOR CUSTOM CODE AND COMPLEX ALGORITHMS

DASK DELAYED — LAZY PARALLELISM FOR CUSTOM CODE

Import

```
import dask
```

Wrap custom functions with the @dask.delayed annotation

Delayed functions operate lazily, producing a task graph rather than executing immediately

Passing delayed results to other delayed functions creates dependencies between tasks

```
@dask.delayed
def load(filename):
    ...

@dask.delayed
def process(data):
    ...
```

Call functions in normal code

```
data = [load(fn) for fn in filenames]
results = [process(d) for d in data]
```

Compute results to execute in parallel

```
dask.compute(results)
```

CONCURRENT.FUTURES — ASYNCHRONOUS REAL-TIME PARALLELISM

Import	`from dask.distributed import Client`
Start local Dask Client	`client = Client()`
Submit individual task asynchronously	`future = client.submit(func, *args, **kwargs)`
Block and gather individual result	`result = future.result()`

Process results as they arrive

```
for future in as_completed(futures):
    ...
```

EXAMPLE

```
L = [client.submit(read, fn) for fn in filenames]
L = [client.submit(process, future) for future in L]
future = client.submit(sum, L)
result = future.result()
```

SET UP CLUSTER — HOW TO LAUNCH ON A CLUSTER

MANUALLY

Start scheduler on one machine

```
$ dask-scheduler
Scheduler started at SCHEDULER_ADDRESS:8786
```

Start workers on other machines
Provide address of the running scheduler

```
host1$ dask-worker SCHEDULER_ADDRESS:8786
host2$ dask-worker SCHEDULER_ADDRESS:8786
```

Start Client from Python process

```
from dask.distributed import Client
client = Client('SCHEDULER_ADDRESS:8786')
```

ON A SINGLE MACHINE

Call Client() with no arguments for easy setup on a single host	`client = Client()`

CLUSTER DEPLOYMENT

On Kubernetes using Helm	`helm install stable/dask`
On Kubernetes from Python	`pip install dask-kubernetes`
On Hadoop/Yarn with dask-yarn	`conda install -c conda-forge dask-yarn`
On HPC with dask-jobqueue	`conda install -c conda-forge dask-jobqueue`

MORE RESOURCES

User Documentation docs.dask.org
Technical documentation for distributed scheduler distributed.dask.org
Report a bug github.com/dask/dask/issues