

Dask for Parallel Processing

Adrian Jackson, EPCC, The University of Edinburgh
a.jackson@epcc.ed.ac.uk



Reusing this material



This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

<https://creativecommons.org/licenses/by-nc-sa/4.0/>

This means you are free to copy and redistribute the material and adapt and build on the material under the following terms: You must give appropriate credit, provide a link to the license and indicate if changes were made. If you adapt or build on the material you must distribute your work under the same license as the original.

Note that this presentation contains images owned by others. Please seek their permission before reusing these images.

Partners



Engineering and
Physical Sciences
Research Council

Natural
Environment
Research Council



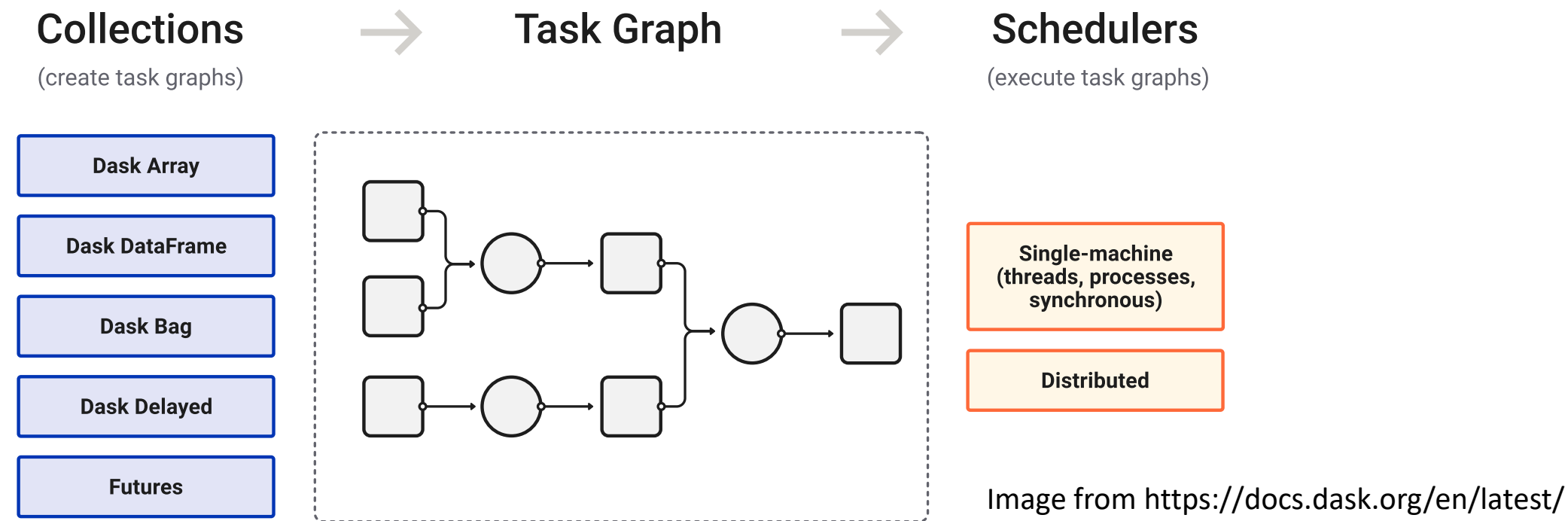
THE UNIVERSITY
of EDINBURGH



**Hewlett Packard
Enterprise**

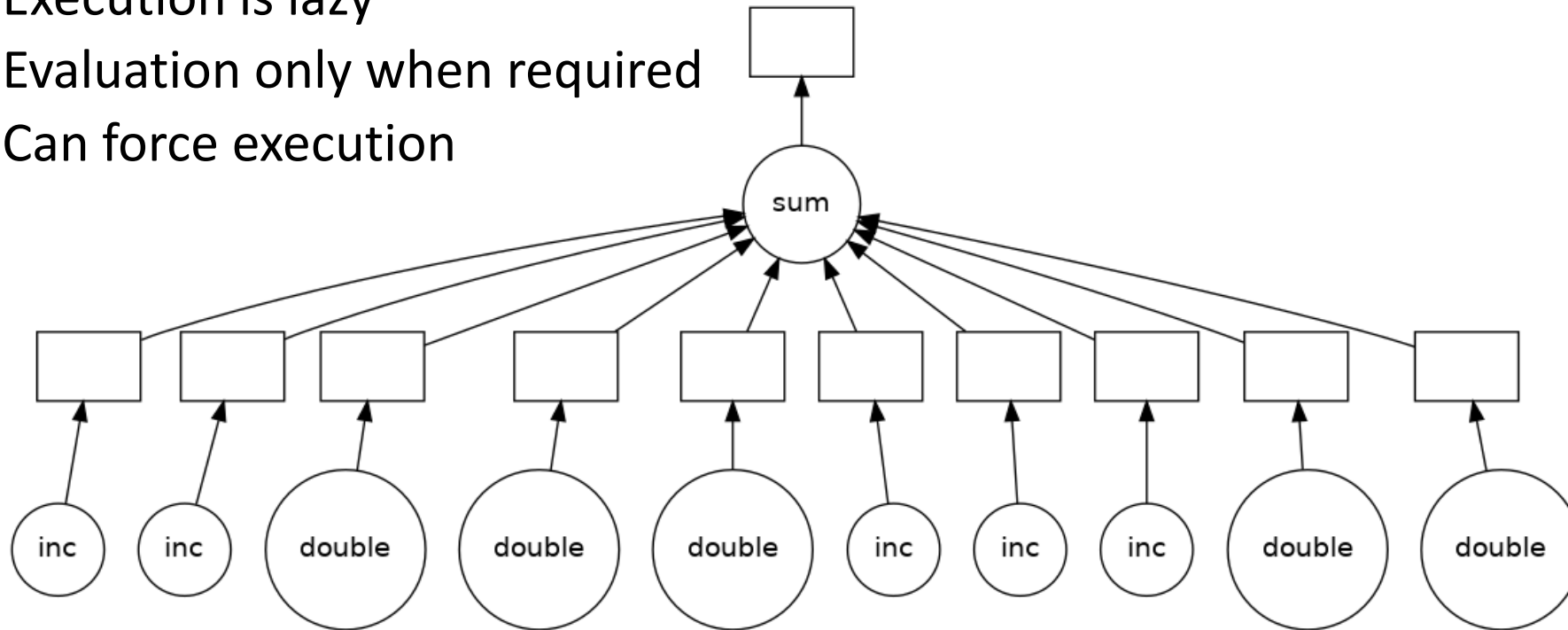
Dask

- Dask is a library to do parallel computing in Python
- Two main components
 - Data collections/types
 - Task scheduling functionality with parallel backends



Parallel computing

- Split work into tasks to be undertaken
- Schedule those tasks on available compute resources
- Execute the task schedule
 - Execution is lazy
 - Evaluation only when required
 - Can force execution

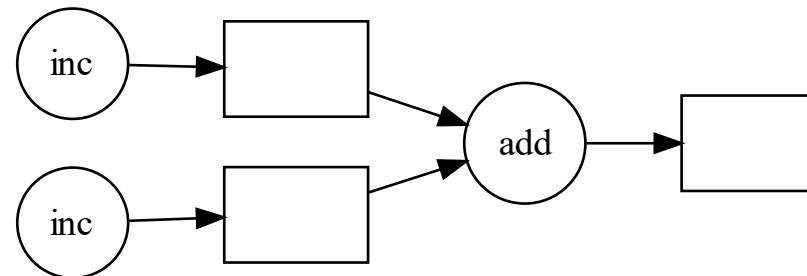


Collections/data types

- Array
 - n-dimensional array (similar to numpy arrays)
 - Out-of-memory functionality
- DataFrame
 - pandas DataFrame functionality
 - 2-d table/spreadsheet
- Bag
 - Unstructured data types
 - Python iterators
- Dask functionality allows all of these to be distributed across compute nodes but still processed as if local

Direct parallelisation

- `dask.delayed` interface
 - Construct custom parallelization
 - Dask task graph functions
- **dask Futures** functionality
 - Enable immediate task generation
 - Sidesteps delayed evaluation
- Data movement functions
 - **gather**, **scatter**, or realise data from futures
- Coordination/synchronisation functionality
 - Queues, Variables, Locks, etc...



Dask Array

- Create array with chunk size

- `import dask.array as da`
- `x = da.random.random((10000, 10000), chunks=(1000, 1000))`

- Many ways to create arrays

- Random

```
random.binomial(n, p[, size, chunks])
random.normal([loc, scale, size, chunks])
random.poisson([lam, size, chunks])
random.random([size, chunks])
```

- <https://docs.dask.org/en/stable/array-api.html#random>

- numpy arrays

```
import numpy as np
import dask.array as da
np_array = np.ones((10000,10000))
x = da.from_array(np_array, chunks=(1000, 1000))
```

- Input must have a `.shape`, `.ndim`, `.dtype` and support numpy-style slicing

- https://docs.dask.org/en/stable/generated/dask.array.from_array.html#dask.array.from_array

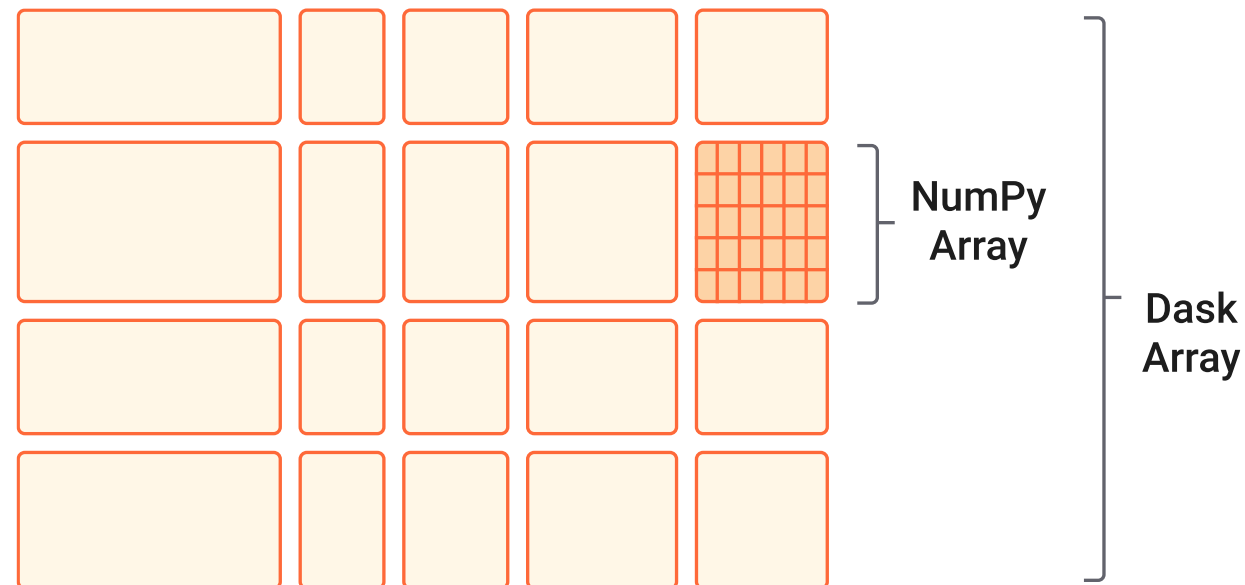


Image from <https://docs.dask.org/en/stable/array.html/>

Dask Array

- From files
 - `.npz` are numpy binary files
 - `.zarr` are binary files designed for blocked/chunked and compressed data
 - Load groups of files into a single array
 - `dask.array.from_npy_stack(dirname, mmap_mode='r')`
 - `dask.array.from_zarr(url, component=None, storage_options=None, chunks=None, name=None, inline_array=False, **kwargs)`
- From dask arrays
 - `concatenate`: create single dimension array from existing arrays
 - `stack`: create new dimension of data with existing arrays
- `delayed`
 - From delayed functions that return things that dask arrays can be constructed from
- Remember, tasking means everything is lazy

Array

- Dask supports a range of numpy like functionality:
 - Arithmetic and scalar mathematics: +, *, exp, log, ...
 - Reductions along axes: sum(), mean(), std(), sum(axis=0), ...
 - Tensor contractions / dot products / matrix multiply: tensordot
 - Axis reordering / transpose: transpose
 - Slicing: x[:100, 500:100:-2]
 - Indexing along single axes with lists or NumPy arrays: x[:, [10, 1, 5]]
 - Array protocols like `__array__` and `__array_ufunc__`
 - Some linear algebra: svd, qr, solve, solve_triangular, lstsq
- Dask Array lacks the following features:
 - Much of np.linalg has not been implemented
 - Arrays with unknown shapes do not support all operations
 - Sorts are not fully supported
 - tolist
 - Iterators can be inefficient
- Full API is at: <https://docs.dask.org/en/stable/array-api.html>

Dask DataFrame

- Create like a standard pandas dataframe

```
import dask.dataframe as dd
df = dd.read_csv('mydata.csv')
```

- Can also specify blocksize (chunks)

```
dask.dataframe.read_csv(urlpath, blocksize='default',
lineterminator=None, compression='infer', sample=256000,
sample_rows=10, enforce=False, assume_missing=False,
storage_options=None, include_path_column=False, **kwargs)
```

- Many ways to create dataframes
- <https://docs.dask.org/en/stable/dataframe-create.html#>
- From files/data sources

```
read_csv(urlpath[, blocksize, ...])
read_parquet(path[, columns, filters, ...])
read_hdf(pattern, key[, start, stop, ...])
read_orc(path[, engine, columns, index, ...])
read_json(url_path[, orient, lines, ...])
read_sql_table(table_name, con, index_col..)
read_sql_query(sql, con, index_col[, ...])
read_sql(sql, con, index_col, **kwargs)
read_table(urlpath[, blocksize, ...])
read_fwf(urlpath[, blocksize, ...])
```

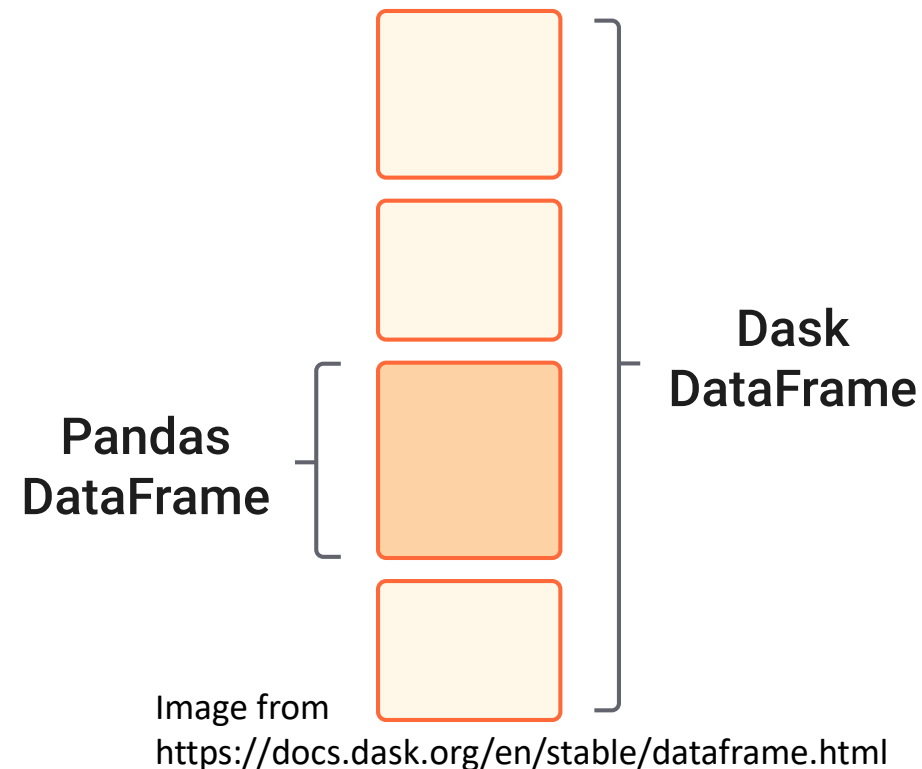
- <https://docs.dask.org/en/stable/array-api.html#random>

- From dask objects

```
from_delayed(dfs[, meta, divisions, prefix, ...])
from_dask_array(x[, columns, index, meta])
dask.bag.core.Bag.to_dataframe([meta, ...])
```

- From other objects

```
• from_bcolz(x[, chunksize, categorize, ...])
• from_array(x[, chunksize, columns, meta])
```



DataFrames

- DataFrames cover part of the pandas API:
 - Independent operations:
 - Element-wise operations: `df.x + df.y`, `df * df`
 - Row-wise selections: `df[df.x > 0]`
 - Loc: `df.loc[4.0:10.5]`
 - Common aggregations: `df.x.max()`, `df.max()`
 - Is in: `df[df.x.isin([1, 2, 3])]`
 - Date time/string accessors: `df.timestamp.month`
 - Group operations:
 - groupby-aggregate (with common aggregations): `df.groupby(df.x).y.max()`, `df.groupby('x').max()`
 - groupby-apply on index: `df.groupby(['idx', 'x']).apply(myfunc)`, where `idx` is the index level name
 - value_counts: `df.x.value_counts()`
 - Drop duplicates: `df.x.drop_duplicates()`
 - Join on index: `dd.merge(df1, df2, left_index=True, right_index=True)`
 - Join with Pandas DataFrames: `dd.merge(df1, df2, on='id')`
 - Element-wise operations with different partitions / divisions: `df1.x + df2.y`
 - Date time resampling: `df.resample(...)`
 - Rolling averages: `df.rolling(...)`
 - Pearson's correlation: `df[['col1', 'col2']].corr()`
 - Group operations requiring data reordering
 - Set index: `df.set_index(df.x)`
 - groupby-apply not on index (with anything): `df.groupby(df.x).apply(myfunc)`
 - Join not on the index: `dd.merge(df1, df2, on='name')`

DataFrames

- DataFrame has the following limitations:
 - Setting a new index from an unsorted column is expensive
 - Many operations like groupby-apply and join on unsorted columns require setting the index, which as mentioned above, is expensive
 - The Pandas API is very large. Dask DataFrame does not attempt to implement many Pandas features or any of the more exotic data structures like NDFrames
 - Operations that were slow on Pandas, like iterating through row-by-row, remain slow on Dask DataFrame
- Full API is <https://docs.dask.org/en/stable/dataframe-api.html>

Dask Bag

- Bag is like a list or set
 - Unordered collection of data with repeats, i.e. {1, 2, 2, 3}
 - Immutable
- Operations
 - map, groupby, filter, fold, etc...
 - Full API <https://docs.dask.org/en/latest/bag-api.html>
- Parallelise simple computations
 - unstructured or semi-structured data
 - i.e. text data, log files, JSON records, or user defined Python objects.
- Implemented using multi-processing (not threads)
 - Reduces communication efficiency between bag elements

- Creating bags:
 - `from_sequence(seq[, partition_size, npartitions])`
 - `from_delayed(values)`
 - `from_url(urls)`
 - `range(n, npartitions)`
 - `read_text(urlpath[, blocksize, compression, ...])`
 - `read_avro(urlpath[, blocksize, ...])`
 - `DataFrame.to_bag([index, format])`
- Bag operations
 - `Bag.accumulate(binop[, initial])`

```
import dask.bag as db
from operator import add
b = db.from_sequence([1, 2, 3, 4, 5], npartitions=2)
b.accumulate(add).compute()
```

 - `Bag.all(split_every=None)`
 - `Bag.any(split_every=None)`
 - `Bag.count([split_every])`
 - `Bag.max([split_every])`
 - `Bag.min([split_every])`
 - `Bag.msum([split_every])`

```
import dask.bag as db
bool_bag = db.from_sequence([True, True, False])
bool_bag.all().compute()
```

Bag

- `Bag.reduction(perpartition, aggregate[, ...])`
 - `Bag.random_sample(prob, random_state=None)`
 - `Bag.filter(predicate)`
- ```
def iseven(x):
 return x % 2 == 0
```
- ```
import dask.bag as db
b = db.from_sequence(range(5))
list(b.filter(iseven))
```
- `Bag.groupby(grouper[, method, npartitions, ...])`
- ```
import dask.bag as db
b = db.from_sequence(range(10))
iseven = lambda x: x % 2 == 0
dict(b.groupby(iseven))
```
- `Bag.foldby(key, binop[, initial, combine, ...])`
    - Combined **reduction** and **groupby**
    - Efficient parallel split-apply-combine tasks.
  - Bags can't be changed
    - Immutable
  - Arrays and DataFrame are faster than Bags
  - Bag **groupby** is slow.
    - **foldby** faster alternative if possible



# delayed

- Direct task graph creation
  - Delay python functions`dask.delayed(func)(inputs...)`
  - Code annotation`@dask.delayed`
  - `compute` still required to complete
- Delay function specifics matter
  - `dask.delayed(f(x, y))`
    - Calculates `f` first then delays the output
  - `dask.delayed(f)(x, y)`
    - Enables lazy evaluation
- Don't delay other dask functionality

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

def double(x):
 return x * 2

def add(x, y):
 return x + y

data = [1, 2, 3, 4, 5]

output = []
for x in data:
 a = inc(x)
 b = double(x)
 c = add(a, b)
 output.append(c)

total = sum(output)

import dask

...
...
```

```
import dask

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

@dask.delayed
def double(x):
 return x * 2

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

data = [1, 2, 3, 4, 5]

output = []
for x in data:
 a = inc(x)
 b = double(x)
 c = add(a, b)
 output.append(c)

total = dask.delayed(sum)(output)
total.compute()
```

```
output = []
for x in data:
 a = dask.delayed(inc)(x)
 b = dask.delayed(double)(x)
 c = dask.delayed(add)(a, b)
 output.append(c)
```

```
total = dask.delayed(sum)(output)
total.compute()
```

# Futures

- Futures provide more complex functionality to build arbitrary task graphs
  - Similar to `delayed`, but tasks executed as soon as available
- Create task:

```
Client.submit(func, *args[, key, workers, ...])
Client.map(func, *iterables[, key, workers, ...])
Future.result([timeout])
```
- Move data:

```
Client.gather(futures[, errors, direct, ...])
Client.scatter(data[, workers, broadcast, ...])
```

# Futures

```
from dask.distributed import Client
client = Client() # start local workers as processes
or
client = Client(processes=False) # start local workers as threads
def inc(x):
 return x + 1

def add(x, y):
 return x + y

a = client.submit(inc, 10)
b = client.submit(inc, 20)
c = client.submit(add, a, b)
c.result()
futures = client.map(inc, range(1000))
```

# Parallelisation

- Dask generally defaults to threaded parallelisation
  - Single node
  - Maximum cores available
  - Dask array and dataframe
- Bag uses the multiprocessing scheduler by default
- Threads are lightweight workers for the main process (program)
  - Easy to parallelise
  - Share data easily between workers
  - Often doesn't scale up to all cores efficiently
- Processes are heavier weight workers (copies of the main program)
  - More scope for independent work
  - More heavy weight in startup costs
  - Explicit communication needed between workers if required

# Dask scheduling

- Scheduling choices can be modified by the user
- Default is generally threading:
 

```
import dask
dask.config.set(scheduler='threads')
```

  - Works well for Array, DataFrame, and Delayed
  - Relies on threading, so native python code will be restricted by the GIL
- Can change to processes:
 

```
import dask
dask.config.set(scheduler='processes')
```

  - No GIL issues but slower for inter-task communications
- More advanced scheduling can be done using distributed
 

```
from dask.distributed import Client
client = Client()
```

  - Defaults to processes
  - `client = Client(processes=False)`
    - Might require `distributed` to be installed (not part of core dask install)
- Distributed can do single node or multi node
  - asynchronous API (Futures)
  - Has a dashboard
  - Improved data locality functionality for multi process work



# Dask scheduling

- Can customise amount of resources and scheduler
  - Per run basis:

```
x.sum().compute(scheduler='processes')
```
  - As a context:

```
with dask.config.set(scheduler='threads'):
 x.compute()
```
  - Globally:

```
dask.config.set(scheduler='threads')
```
  - Number of workers

```
from multiprocessing.pool import ThreadPool
dask.config.set(pool=ThreadPool(8))
```
  - Distributed scheduler

```
client = Client(processes=False, n_workers=4)
```

# Distributed scheduler

- Distributed Client
 

```
from dask.distributed import Client
client = Client(...)
df.x.sum().compute()
```
- Local cluster (single node)
 

```
from dask.distributed import Client, LocalCluster
cluster = LocalCluster()
client = Client(cluster)
df.x.sum().compute()
```
- `dask_jobqueue` provides batch system interfaces
 

```
HTCondorCluster([n_workers, loop, security, ...])
LSFCluster([n_workers, loop, security, ...])
MoabCluster([n_workers, loop, security, ...])
OARCluster([n_workers, loop, security, ...])
PBSCluster([n_workers, loop, security, ...])
SGECluster([n_workers, loop, security, ...])
SLURMCluster([n_workers, loop, security, ...])
```
- `dask_mpi` provides MPI launch interfacing
 

```
from dask_mpi import
initialize
initialize()
from dask.distributed import Client
Client = Client()
mpirun -np 4 python my_client_script.py
• Or
mpirun -np 4 dask-mpi --scheduler-file ~/dask-scheduler.json
from dask.distributed import Client
client = Client(scheduler_file='~/dask-scheduler.json')
```

<https://docs.dask.org/en/latest/deploying-hpc.html>

# Guiding principles

- If you don't need Dask, don't use it
  - Numpy array, panda dataframe, etc... all faster for small scale, in-memory
- Chunking/granularity important for performance
  - Too big chunks -> not enough parallelism
  - Too small chunks -> large parallelisation overhead
- **compute** as infrequently as possible
  - **compute** forces evaluation of the task graph
  - Might need multiple compute calls if task graph gets too big
- Mix threads and processes if doing larger parallelisation
  - Threads good for small scale parallelisation
- Load data with dask
- Persist datasets to memory when reduced



# Dask on ARCHER2

- Dask is available in the cray-python
  - Threading/shared memory backends
- Distributed dask needs to be installed

```
module load cray-python/3.9.7.1
export PYTHONUSERBASE=/work/t01/t01/auser/.local
export PATH=$PYTHONUSERBASE/bin:$PATH
python -m pip install --user dask distributed --upgrade
python -m pip install --user dask-jobqueue --upgrade
```
- Currently need to submit from compute nodes
  - Dask runs a scheduler that needs connection from the workers

# Threaded Dask on ARCHER2

- Running single node dask as normal job works fine
  - Default mode is threading
  - Requires process binding and thread placement to be sensible

```
#!/bin/bash
#SBATCH --job-name=my_job
#SBATCH --nodes=1
#SBATCH --tasks-per-node=1
#SBATCH --cpus-per-task=128
#SBATCH --partition=standard
#SBATCH --qos=short
#SBATCH --account=z19
#SBATCH --time=0:10:0
python dask-program.py
```

- Can also do an interactive run:

```
srun --nodes=1 --tasks-per-node=1 --cpus-per-task=128 --exclusive --partition=standard --qos=short
--reservation=shortqos --account=z19 --time=0:20:0 python dask-program.py
```

# Distributed Dask on ARCHER2

```
from dask_jobqueue import SLURMCluster
cluster = SLURMCluster(cores=128,
 processes=128,
 memory='256GB',
 queue='standard',
 header_skip=['--mem'],
 job_extra=['--qos="standard"'],
 python='srun python',
 project='t01',
 walltime="01:00:00",
 shebang="#!/bin/bash --login",
 local_directory='$PWD',
 interface='hsn0',
 env_extra=['module load cray-python',
 'export PYTHONUSERBASE=/work/t01/t01/auser/.local/',
 'export PATH=$PYTHONUSERBASE/bin:$PATH',
 'export PYTHONPATH=$PYTHONUSERBASE/lib/python3.8/site-
packages:$PYTHONPATH'])
cluster.scale(jobs=2) # Deploy two single-node jobs
from dask.distributed import Client
client = Client(cluster) # Connect this local process to remote workers
import dask.array as da
x = da.random.random((10000, 10000), chunks=(1000, 1000))
mean = x.mean().compute()
```