12.010 Computational Methods of Scientific Programming 2021

Lecture 23: Working with large data files: NetCDF, databases

Summary

- Large problem sources

- Tools
  - Dask
  - Dask + xarray
  - Dask + xarray + open data sets (in zarr)

- Other tools
  - Dask + Pandas
  - Hadoop, spark

Large sources of digital data abound

- Physics
  - Particle
  - Astro

- Medical
  - Imaging
  - Sequencing

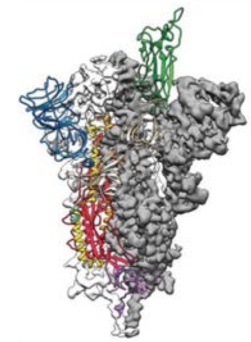
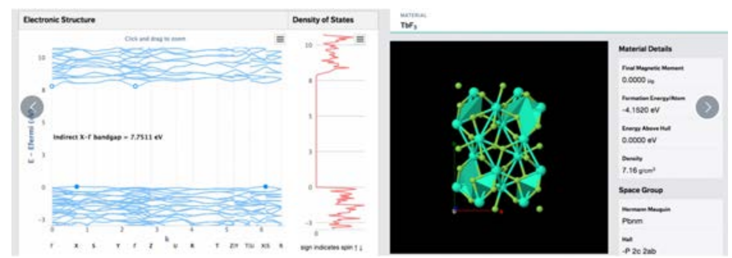


Image courtesy of Jason McLellan, University of Texas at Austin. Used with permission.

- Earth and environment
  - Biodiversity and ecosystems
  - Topography
  - Fire, Water, Land Use

- Materials



Digital sources

- Sequencing

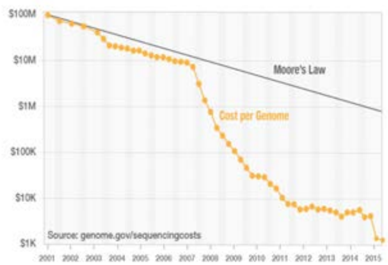


Image courtesy of NIH.
Image is in the public domain.

- Simulation

Image courtesy of DOE. Image is in the public domain.

- CCD

Common theme is generation of PiB of digital information, useful for analysis.

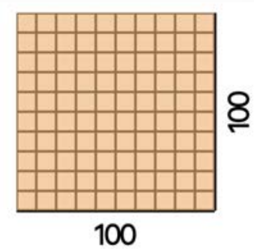
Need some tools that scale.

Tools for large data repositories

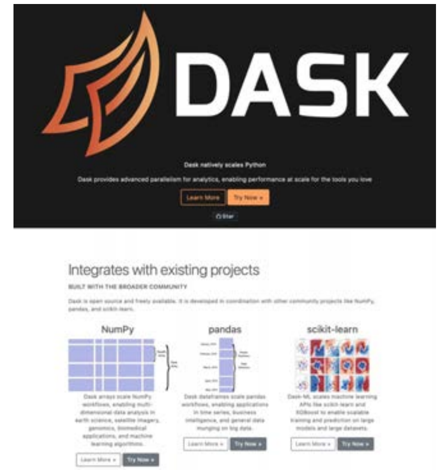
- Dask (and more)
  - Dask is a library that is designed and maintained to be compatible with Numpy, Dataframes and SciKit.
  - It provides
    - lazily evaluated arrays and other data structures
    - distributed (multi-process and multi-node) analysis
  - It has handy features for reading in collections of files in standard forms
  - Builtin to xarray.
  - Designed to help with array like problems that don't fit in memory and/or can leverage multiple processors for speed.

```
[1]: import numpy as np
     xnp=np.ones((100,100))
     xnp

[1]: array([[1., 1., 1., ..., 1., 1., 1.],
            [1., 1., 1., ..., 1., 1., 1.],
            [1., 1., 1., ..., 1., 1., 1.],
            ...,
            [1., 1., 1., ..., 1., 1., 1.],
            [1., 1., 1., ..., 1., 1., 1.],
            [1., 1., 1., ..., 1., 1., 1.]])

[2]: import dask.array as da
     xda=da.ones((100,100),chunks=(10, 10))
     xda

[2]:
```

	Array	Chunk
Bytes	78.12 kiB	800 B
Shape	(100, 100)	(10, 10)
Count	100 Tasks	100 Chunks
Type	float64	numpy.ndarray



Dask



- First developed in 2016
- Official web site https://dask.org
- Provides "distributed" data structures that are like those in Numpy, Dataframes, Scikit
  - except the data structures can be distributed across processors and computers
  - computations on the distributed data structures can execute in parallel

Dask for Numpy

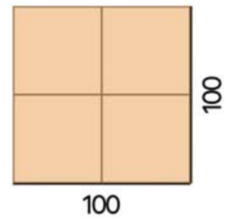
- Dask “array” has same interfaces as Numpy, but for Dask objects.

- Introduces “chunks” that allow for parallel execution
- computation is evaluated lazily and can be launched on separate local remote processes (in “clusters”)

[1]:
```
import dask.array as da
xda=da.ones((100,100),chunks=(50, 50))
xda
```

[1]:

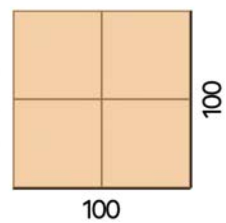
	Array	Chunk
Bytes	78.12 kiB	19.53 kiB
Shape	(100, 100)	(50, 50)
Count	4 Tasks	4 Chunks
Type	float64	numpy.ndarray



[2]:
```
z=xda+xda.T
z
```

[2]:

	Array	Chunk
Bytes	78.12 kiB	19.53 kiB
Shape	(100, 100)	(50, 50)
Count	12 Tasks	4 Chunks
Type	float64	numpy.ndarray



[3]:
```
z.compute()
```

[3]:
```
array([[2., 2., 2., ..., 2., 2., 2.],
       [2., 2., 2., ..., 2., 2., 2.],
       [2., 2., 2., ..., 2., 2., 2.],
       ...,
```

Example showing dask array.

It has a ones() function like numpy, but can take a “chunk” size.

.T is a transpose, same as numpy.

computation is not executed immediately, only when required.

.compute() can be used to trigger computation.

Dask lazy chunk evaluation?

- Dask works fine with numpy, but it behaves a little differently.
- Arrays can be created with “chunks” that correspond to parallel parts to operate on.
- Computations (e.g. max()) are first formed into a “graph” of operations, but not executed.
- They are only executed when needed. For example, by compute().

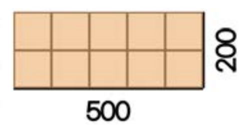
[1]:
```
import numpy as np
import dask.array as da
data = np.arange(100_000).reshape(200, 500)
print(data)
data.max()
```

```
[[    0     1     2 ...   497   498   499]
 [  500   501   502 ...   997   998   999]
 [ 1000  1001  1002 ...  1497  1498  1499]
 ...
 [98500 98501 98502 ... 98997 98998 98999]
 [99000 99001 99002 ... 99497 99498 99499]
 [99500 99501 99502 ... 99997 99998 99999]]
```

[1]: 99999

[2]:
```
data_dask=da.from_array(data, chunks=(100, 100))
display(data_dask)
data_dask.max()
```

	Array	Chunk
Bytes	781.25 kiB	78.12 kiB
Shape	(200, 500)	(100, 100)
Count	10 Tasks	10 Chunks
Type	int64	numpy.ndarray



[2]:

	Array	Chunk
Bytes	8 B	8.0 B
Shape	()	()
Count	26 Tasks	1 Chunks
Type	int64	numpy.ndarray

[3]:
```
data_dask.max().compute()
```

[3]: 99999

Numpy arrays can be mapped to “chunked” dask arrays.

Numpy computations are immediate.

Dask computations are lazy i.e. defined as a “graph” of operations and then executed when needed/requested.

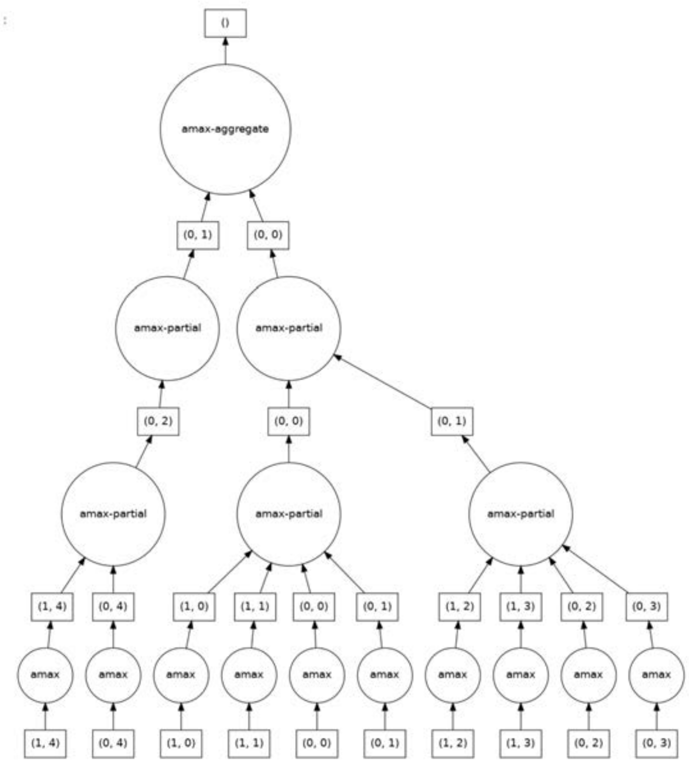
Dask compute graph

- Computing on a Dask data structure generates a “graph” of operations.
- The “graph” is a tree of dependent and independent operations called “tasks” that can be grouped into sets that can execute concurrently.
- The tasks (square boxes) are nodes of the graph. The boxes start at base as chunks for the data structure.
- The arrows show “edges”
- The graph for max is a “directed acyclic graph” (DAG).

[4]:
```
dm=data_dask.max()
display(dm)
dm.visualize()
```

	Array	Chunk
Bytes	8 B	8.0 B
Shape	()	()
Count	26 Tasks	1 Chunks
Type	int64	numpy.ndarray

[4]:

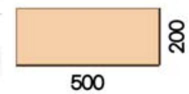


Dask chunk examples

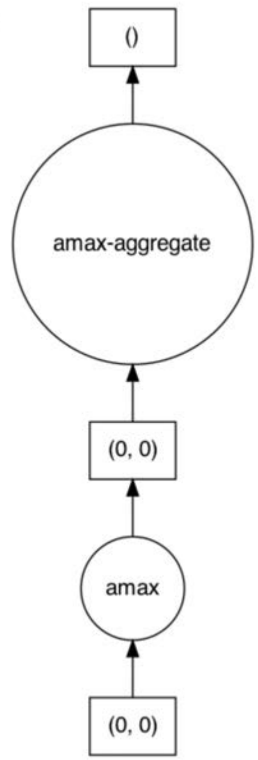
- Dask uses "chunks" to sub-divide work that can execute concurrently.
- The graph generated starts from the chunking
- When dask arrays are created a chunk size can be specified.

[8]:
```
data_dask=da.from_array(data, chunks=(200, 500))
display(data_dask)
data_dask.max()
dm=data_dask.max()
dm.visualize()
```

	Array	Chunk
Bytes	781.25 kiB	781.25 kiB
Shape	(200, 500)	(200, 500)
Count	1 Tasks	1 Chunks
Type	int64	numpy.ndarray



[8]:



Dask lazy examples

- The Numpy math operations are defined for dask arrays.

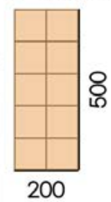
- Math operations will be lazy.

```
display( a.mean()  )
print(   a.mean()  )
display( a.T       )
print(   a.T       )
display( np.sin(a) )
print(   np.sin(a) )
```

	Array	Chunk
Bytes	8 B	8.0 B
Shape	()	()
Count	26 Tasks	1 Chunks
Type	float64	numpy.ndarray

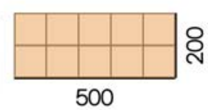
```
dask.array<mean_agg-aggregate, shape=(), dtype=float64, chunksize=(), chunktype=numpy.ndarray>
```

	Array	Chunk
Bytes	781.25 kiB	78.12 kiB
Shape	(500, 200)	(100, 100)
Count	20 Tasks	10 Chunks
Type	int64	numpy.ndarray




```
dask.array<transpose, shape=(500, 200), dtype=int64, chunksize=(100, 100), chunktype=numpy.ndarray>
```

	Array	Chunk
Bytes	781.25 kiB	78.12 kiB
Shape	(200, 500)	(100, 100)
Count	20 Tasks	10 Chunks
Type	float64	numpy.ndarray



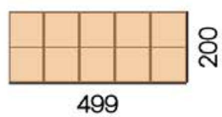

```
dask.array<sin, shape=(200, 500), dtype=float64, chunksize=(100, 100), chunktype=numpy.ndarray>
```

Dask operations can span chunks

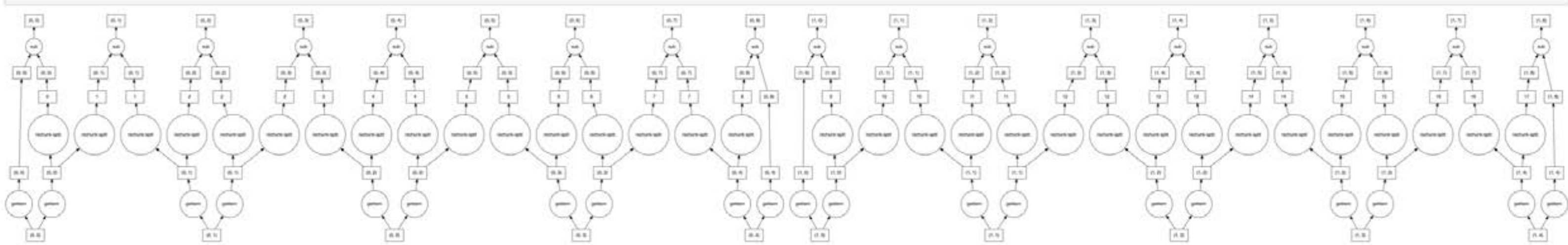
e.g. Can apply operations like np.diff() to dask arrays. Graph is more complex.

```
np.diff(a)
```

	Array	Chunk
Bytes	779.69 kiB	77.34 kiB
Shape	(200, 499)	(100, 99)
Count	116 Tasks	18 Chunks
Type	int64	numpy.ndarray

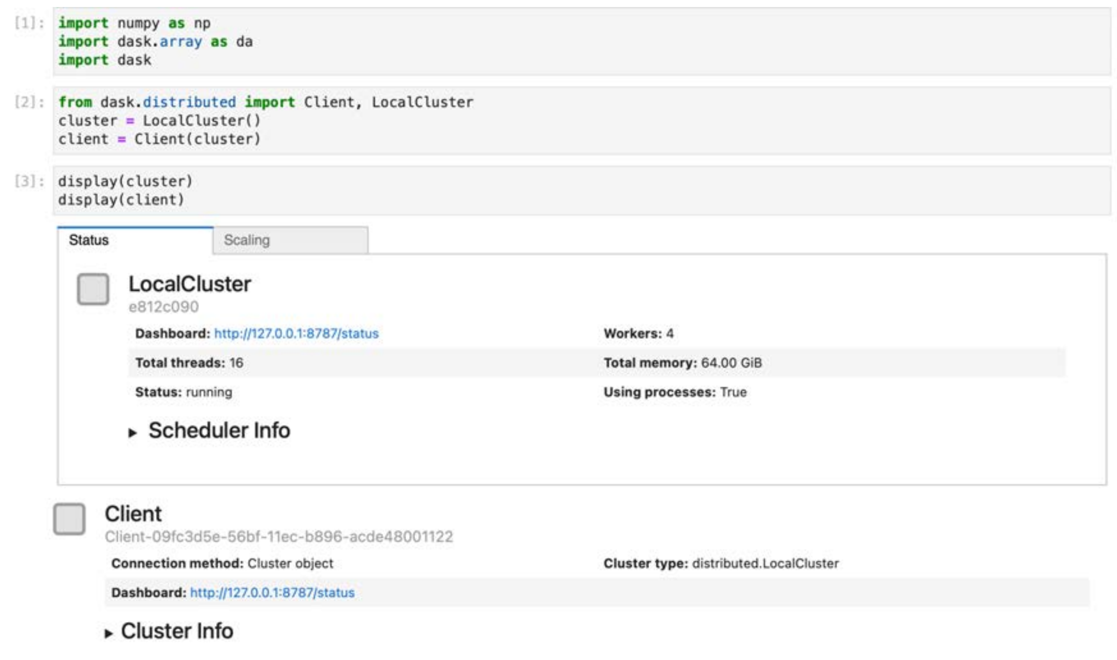



```
np.diff(a).visualize()
```



Dask clusters under the hood - I

- Dask “chunks”, “blocks”, “tasks” and “graphs” can be a way to harness parallelism on a single node or on tens to hundreds of nodes.
- Single node has a default cluster.
- Can customize local machine cluster or create cluster on collection of nodes.
- Tasks are allocated to cluster through client interface
- Dask Arrays handle allocation transparently



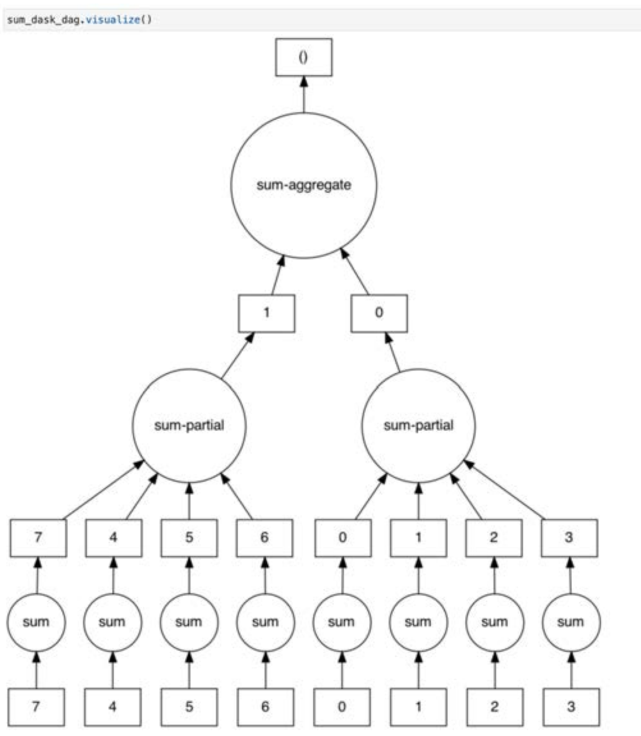
Dask clusters under the hood - II

- Client interface is used to launch parallel tasks by Dask
- With Array, Dataframe, xarray interface this is done automatically

```
[4]: def square(x):
         return x ** 2
     def neg(x):
         return -x
     def loop(x):
         while ( True ):
             continue
         return 0
     A = client.map(square, range(10))
     B = client.map(neg, A)
     L = client.map(loop, [0])
     total = client.submit(sum, B)
     total.result()

[4]: -285
```

Laptop test

x2.5 using Dask



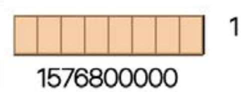

```
[1]: import numpy as np
     import dask.array as da
```

```
[2]: data = np.random.rand(50 * 365 * 24 * 60 * 60)
     print(data.shape)

     %time data_sum = data.sum()
     data_sum
```

```
(1576800000,)
CPU times: user 971 ms, sys: 1.41 ms, total: 972 ms
Wall time: 972 ms
[2]: 788419656.3967832
```

```
[3]: data_dask = da.from_array(data, chunks=len(data) // 8)
     display(data_dask)
     sum_dask_dag = data_dask.sum()
     %time sum_dask = sum_dask_dag.compute()
     sum_dask
```

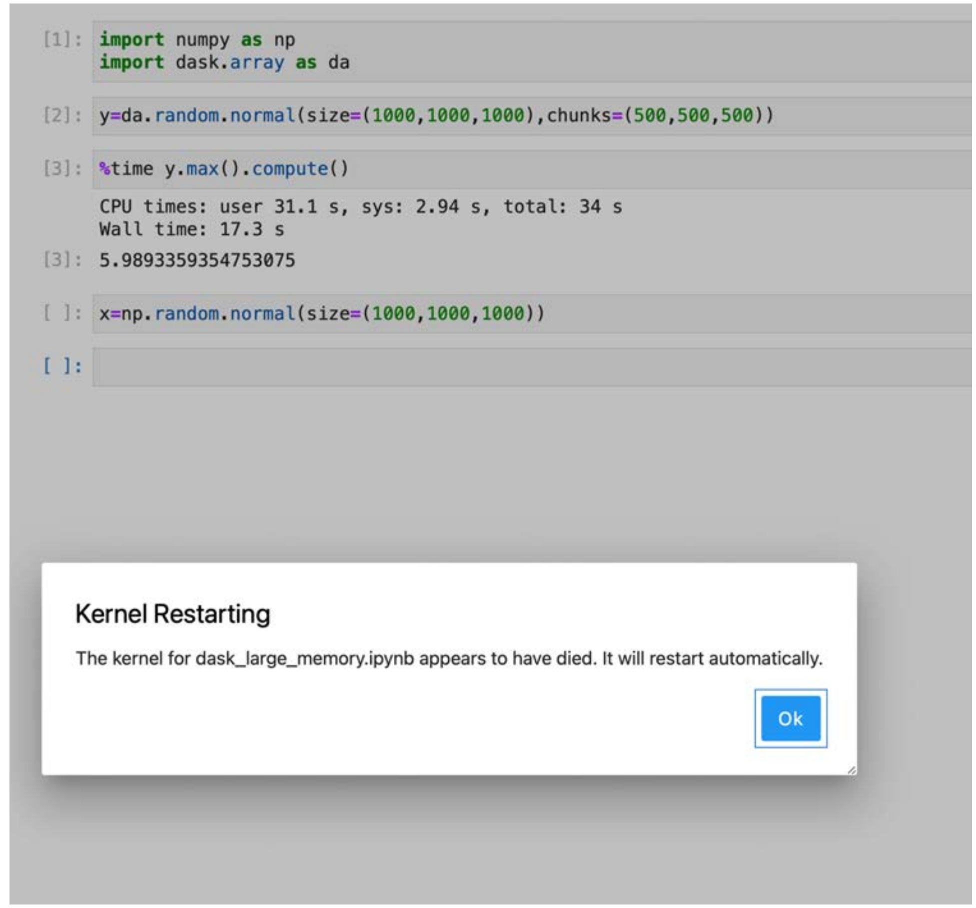
	Array	Chunk
Bytes	11.75 GiB	1.47 GiB
Shape	(1576800000,)	(197100000,)
Count	8 Tasks	8 Chunks
Type	float64	numpy.ndarray




```
CPU times: user 2.87 s, sys: 19.9 ms, total: 2.89 s
Wall time: 375 ms
[3]: 788419656.396776
```

Dask out of core

- Things that won't fit in memory.
- On a small memory system Dask automates running things in chunks that fit in memory
- Can use this to work with arrays larger than memory



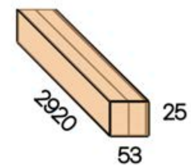
Dask and xarray

- Dask works with xarray and with Dataframes
- With both this can be used to work with larger collections of input data in collections of multiple files.
- Previously looked at xarray, without delving into dask.

```
[1]: import xarray as xr

[2]: ds = xr.tutorial.open_dataset('air_temperature',
                                   chunks={'lat': 25, 'lon': 25, 'time': -1})

[3]: ds.air
```

[3]: xarray.DataArray 'air' (**time**: 2920, **lat**: 25, **lon**: 53)

	Array	Chunk
Bytes	14.76 MiB	6.96 MiB
Shape	(2920, 25, 53)	(2920, 25, 25)
Count	4 Tasks	3 Chunks
Type	float32	numpy.ndarray



▾ Coordinates:

lat	(lat)	float32	75.0 72.5 70.0 ... 20.0 17.5 15.0
lon	(lon)	float32	200.0 202.5 205.0 ... 327.5 330.0
time	(time)	datetime64[ns]	2013-01-01 ... 2014-12-31T18:00:00

▸ Attributes: (11)

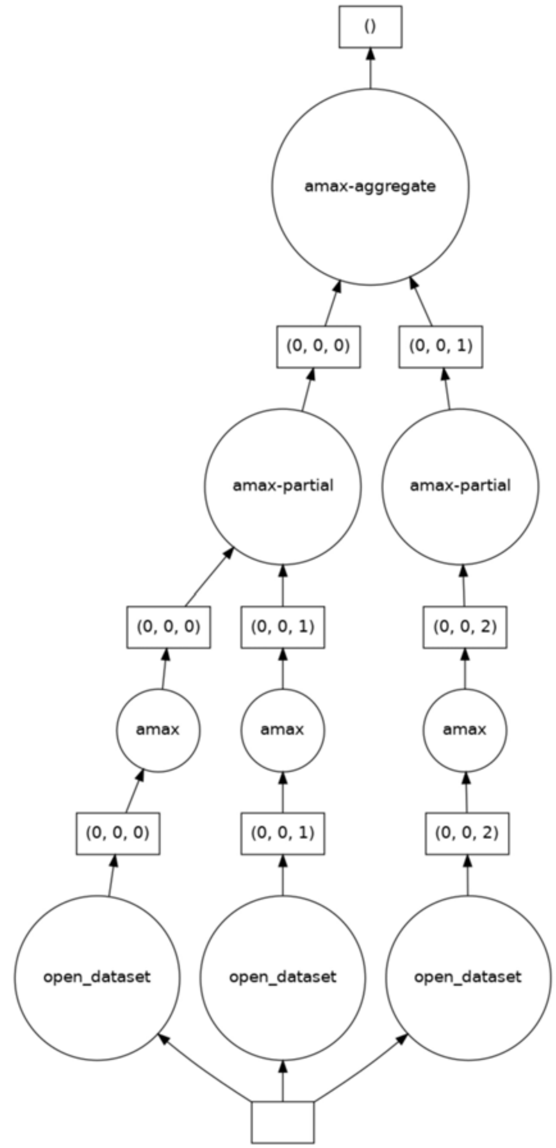
In the xarray example earlier, the air-temperature variable is a Dask array chunked in the third dimension.

Dask and xarray

- Dask supports operations on xarrays with a task graph

```
# We can look at how to evaluation works in this case
ds.air.data.max().visualize()
```

- in this way even the file open, for example, is handled lazily.



Dask and xarray, multiple netcdf files

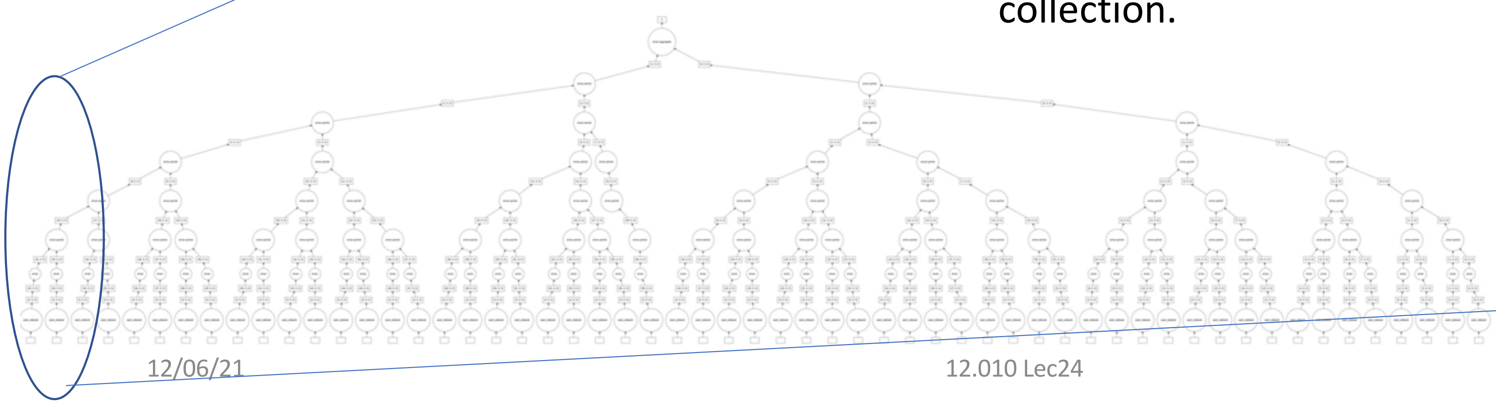
- With xarray wrapper can load multiple files as chunks

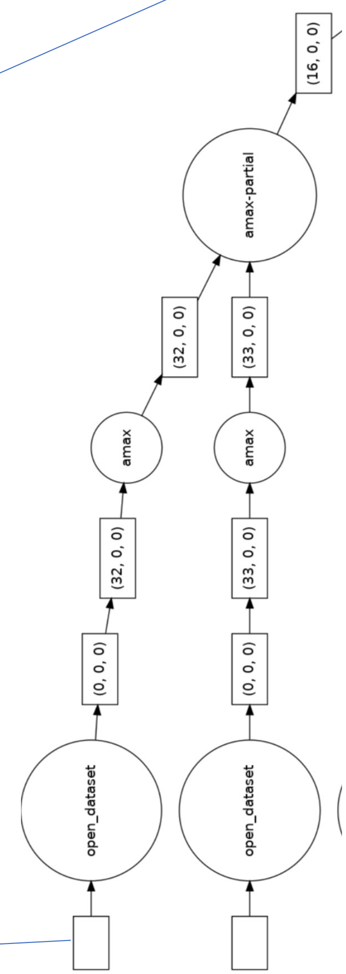
```
# we can extend to chunk multiple files
!git clone https://github.com/pangeo-data/tutorial-data.git
```

```
dsnc=xr.open_mfdataset('tutorial-data/sst/*.nc')
```

```
dsnc.sst.data.max().visualize()
```

Here the graph includes sweeping over all the files in the collection.

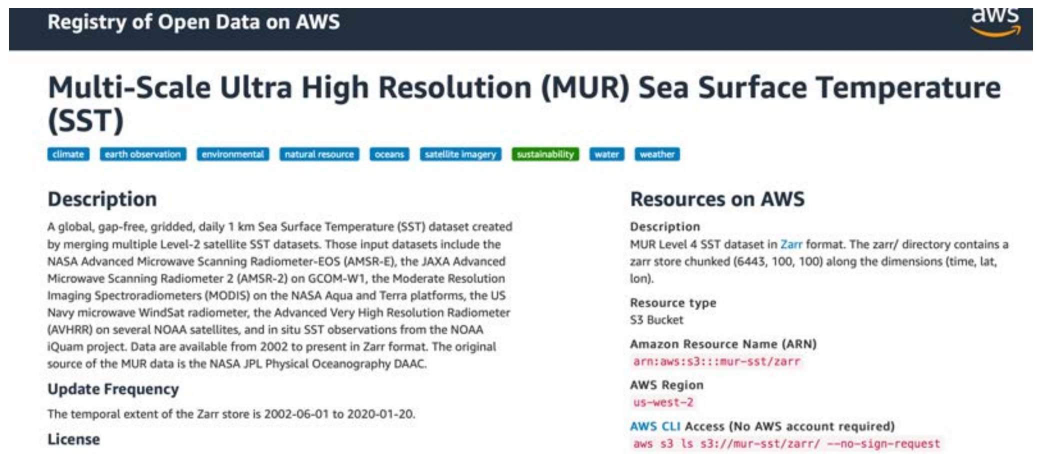




Dask and xarray, multi TiB cloud data

- Multi-file and lazy evaluation graphs from Dask, provides the basis for working with multi-TiB data sets.

- This works especially well if
    - the datasets are openly available (not behind a pay-wall, license wall etc..)
    - the datasets themselves are chunked, for example using the zarr format.


Registry of Open Data on AWS

Multi-Scale Ultra High Resolution (MUR) Sea Surface Temperature (SST)

climate · earth observation · environmental · natural resource · oceans · satellite imagery · sustainability · water · weather

Description

A global, gap-free, gridded, daily 1 km Sea Surface Temperature (SST) dataset created by merging multiple Level-2 satellite SST datasets. Those input datasets include the NASA Advanced Microwave Scanning Radiometer-EOS (AMSR-E), the JAXA Advanced Microwave Scanning Radiometer 2 (AMSR-2) on GCOM-W1, the Moderate Resolution Imaging Spectroradiometers (MODIS) on the NASA Aqua and Terra platforms, the US Navy microwave WindSat radiometer, the Advanced Very High Resolution Radiometer (AVHRR) on several NOAA satellites, and in situ SST observations from the NOAA iQuam project. Data are available from 2002 to present in Zarr format. The original source of the MUR data is the NASA JPL Physical Oceanography DAAC.

Update Frequency

The temporal extent of the Zarr store is 2002-06-01 to 2020-01-20.

License

Resources on AWS

Description
MUR Level 4 SST dataset in Zarr format. The zarr/ directory contains a zarr store chunked (6443, 100, 100) along the dimensions (time, lat, lon).

Resource type
S3 Bucket

Amazon Resource Name (ARN)
`arn:aws:s3:::mur-sst/zarr`

AWS Region
`us-west-2`

AWS CLI Access (No AWS account required)
`aws s3 ls s3://mur-sst/zarr/ --no-sign-request`

https://registry.opendata.aws/mur/

Dask and xarray and MUR SST

- MUR SST is a NASA daily sea-surface temperature product, covering 2002 to present day at ~1km resolution.
  - the zarr archive in AWS covers 2002 – 2020.
  - in numbers it has about 4×10^{12} values and is about 15TiB.
  - in principle can download for analysis, but many times want to look at some part in space and time.

```
import xarray as xr
mur_sst = xr.open_zarr('https://mur-sst.s3.us-west-2.amazonaws.com/zarr-v1',consolidated=True)
display(mur_sst)
display(mur_sst.analysed_sst.data.blocks[0,0,0])
```

xarray.Dataset

▶ Dimensions: (**time**: 6443, **lat**: 17999, **lon**: 36000)

▼ Coordinates:

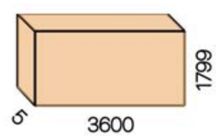
lat	(lat)	float32	-89.99 -89.98 ... 89.98 89.99
lon	(lon)	float32	-180.0 -180.0 ... 180.0 180.0
time	(time)	datetime64[ns]	2002-06-01T09:00:00 ... 2020-01-...

▼ Data variables:

analysed_sst	(time, lat, lon)	float32	dask.array<chunksize=(5, 1799, 3600), me...
analysis_error	(time, lat, lon)	float32	dask.array<chunksize=(5, 1799, 3600), me...
mask	(time, lat, lon)	float32	dask.array<chunksize=(5, 1799, 3600), me...
sea_ice_fraction	(time, lat, lon)	float32	dask.array<chunksize=(5, 1799, 3600), me...

▶ Attributes: (47)

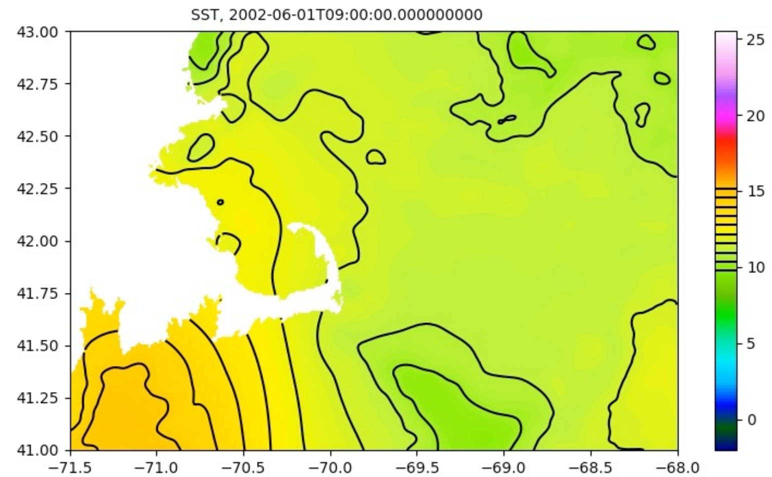
	Array	Chunk
Bytes	123.53 MiB	123.53 MiB
Shape	(5, 1799, 3600)	(5, 1799, 3600)
Count	141792 Tasks	1 Chunks
Type	float32	numpy.ndarray



Two commands using xarray and dask provide lazy access to the entire data.

Dask and xarray and MUR SST

- Movie of temperature in region around Boston
- This involves extracting more data
- Somewhat slow to my laptop at home
- To cloud machine, using parallel dask can download all 20 years in < 60 seconds.



Dask and xarray and MUR SST

- Key pieces
  - Starting dask workers

```
$ dask-scheduler
```

```
$ dask-worker --nprocs 30 --memory-limit
2GB --nthreads 2 tcp://127.0.0.1:8786
```

  - Filtering to region of interest

```
ds=mur_sst.analysed_sst
mask_lon=(ds.lon >= -71.5) & ( ds.lon <= -68)
mask_lat=(ds.lat >=  41) & ( ds.lat <=  43)
import dask
with dask.config.set(**{'array.slicing.split_large_chunks':
False}):
    ds_masked=ds.where(mask_lon & mask_lat, drop=True)
```

  - Filtering and dask together make process x1000+ faster

Dask and dataframes

- Dask also works similarly with Pandas Dataframes to speed up larger Dataframe processing.

- There is also a proprietary library "RAPIDS" that works with a GPU dataframe (cudf) to allow dataframes on multiple GPUs in parallel.

```
import pandas as pd
import numpy as np
import dask.dataframe as dd
```

```
index = pd.date_range("2021-09-01", periods=2400, freq="1
df = pd.DataFrame({"a": np.arange(2400), "b": list("abcad
display(df)
ddf = dd.from_pandas(df, npartitions=10)
display(ddf)
```

	a	b
2021-09-01 00:00:00	0	a
2021-09-01 01:00:00	1	b
2021-09-01 02:00:00	2	c
2021-09-01 03:00:00	3	a
2021-09-01 04:00:00	4	d
...	...	...
2021-12-09 19:00:00	2395	a
2021-12-09 20:00:00	2396	d
2021-12-09 21:00:00	2397	d
2021-12-09 22:00:00	2398	b
2021-12-09 23:00:00	2399	e

2400 rows × 2 columns

Dask DataFrame Structure:

	a	b
npartitions=10		
2021-09-01 00:00:00	int64	object
2021-09-11 00:00:00	...	...
...	...	...
2021-11-30 00:00:00	...	...
2021-12-09 23:00:00	...	...

Dask Name: from_pandas, 10 tasks

Spark and Hadoop

- Some other tools that are used for large data analysis are
  - Hadoop
  - Spark

- These largely leverage “map, reduce” abstraction.

- They are very common in “business analytics”