PARALLEL COMPUTING WITH DASK

Chunking Arrays in Dask

Dhavide Aruliah
Director of Training, Anaconda

What we've seen so far...

- Measuring memory usage
- Reading large files in chunks
- Computing with generators
- Computing with `dask.delayed`

Working with Numpy Arrays

```
In [1]: import numpy as np

In [2]: a = np.random.rand(10000)

In [3]: print(a.shape, a.dtype)
(10000,) float64

In [4]: print(a.sum())
5017.32043995

In [5]: print(a.mean())
0.501732043995
```

Working with Dask Arrays

```
In [6]: import dask.array as da

In [7]: a_dask = da.from_array(a, chunks=len(a) // 4)

In [8]: a_dask.chunks
Out[8]: ((2500, 2500, 2500, 2500),)
```

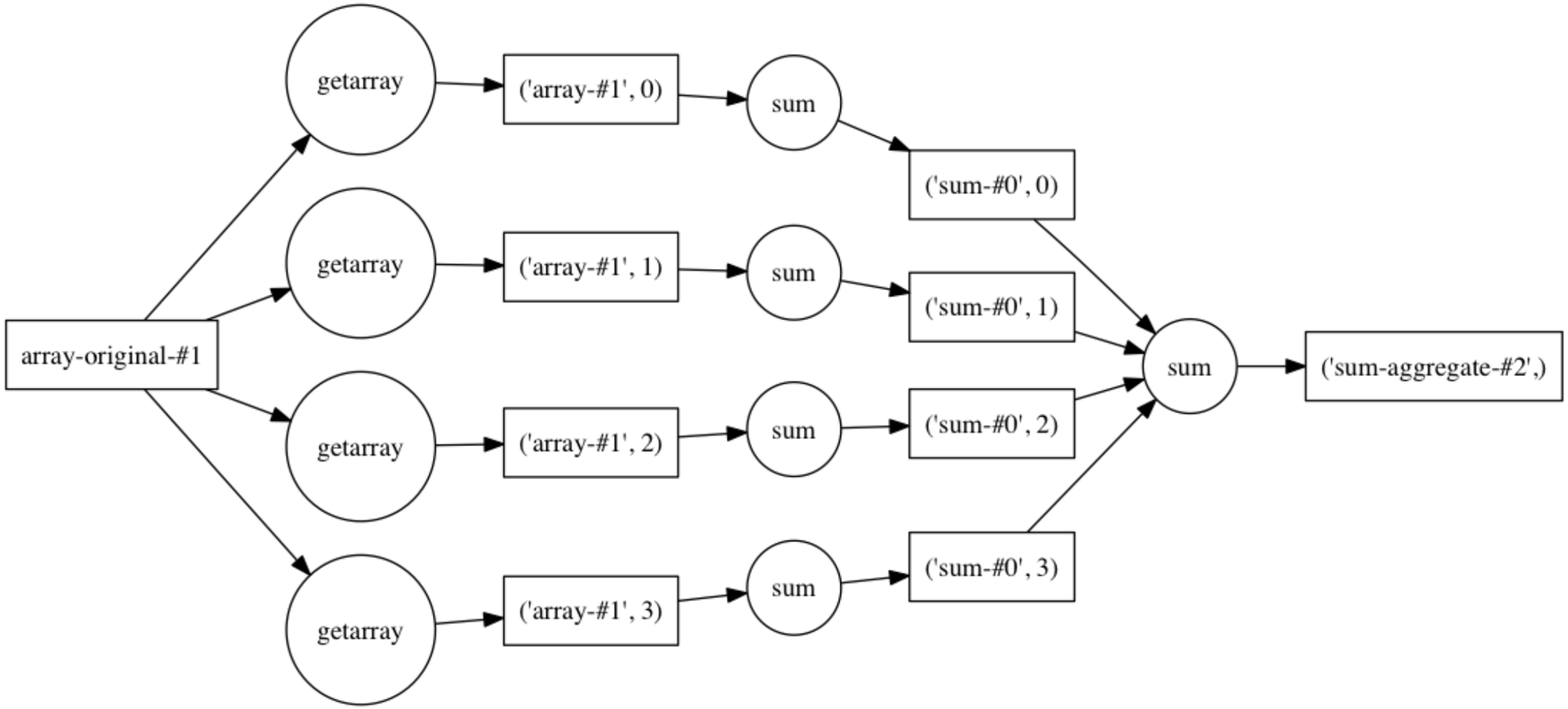
Aggregating in Chunks

```
In [9]: n_chunks = 4

In [10]: chunk_size = len(a) // n_chunks

In [11]: result = 0 # Accumulate sum explicitly

In [12]: for k in range(n_chunks):
    ...:     offset = k * chunk_size # track offset explicitly
    ...:     a_chunk= a[offset:offset + chunk_size] # slice chunk explicitly
    ...:     result += a_chunk.sum()

In [13]: print(result)
5017.32043995
```

Aggregating with Dask Arrays

```
In [14]: a_dask = da.from_array(a, chunks=len(a)//n_chunks)

In [15]: result = a_dask.sum()

In [16]: result
Out[16]: dask.array<sum-aggregate, shape=(), dtype=float64, chunksize=()>

In [17]: print(result.compute())
5017.32043995

In [18]: result.visualize(rankdir='LR')
```

Task Graph


array-original-#1
getarray
getarray
getarray
getarray
('array-#1', 0)
('array-#1', 1)
('array-#1', 2)
('array-#1', 3)
sum
sum
sum
sum
('sum-#0', 0)
('sum-#0', 1)
('sum-#0', 2)
('sum-#0', 3)
sum
('sum-aggregate-#2',)

Dask Array Methods/Attributes

- Attributes: `shape`, `ndim`, `nbytes`, `dtype`, `size`, etc.
- Aggregations: `max`, `min`, `mean`, `std`, `var`, `sum`, `prod`, etc.
- Array transformations: `reshape`, `repeat`, `stack`, `flatten`, `transpose`, `T`, etc.
- Mathematical operations: `round`, `real`, `imag`, `conj`, `dot`, etc.

Timing Array Computations

```
In [20]: import h5py, time

In [21]: with h5py.File('dist.hdf5', 'r') as dset:
    ...:     dist = dset['dist'][:]
    ...:

In [22]: dist_dask8 = da.from_array(dist, chunks=dist.shape[0]//8)

In [23]: t_start = time.time(); \
    ...: mean8 = dist_dask8.mean().compute(); \
    ...: t_end = time.time()

In [24]: t_elapsed = (t_end - t_start) * 1000 # Elapsed time in ms

In [25]: print('Elapsed time: {} ms'.format(t_elapsed))
Elapsed time: 180.96423149108887 ms
```

PARALLEL COMPUTING WITH DASK

Let's practice!

PARALLEL COMPUTING WITH DASK

Computing with Multidimensional Arrays

Dhavide Aruliah
Director of Training, Anaconda

A Numpy Array of Time Series Data

```
In [1]: import numpy as np

In [2]: time_series = np.loadtxt('max_temps.csv', dtype=np.int64)

In [3]: print(time_series.dtype)
int64

In [4]: print(time_series.shape)
(21,)

In [5]: print(time_series.ndim)
1
```

Reshaping Time Series Data

```
In [6]: print(time_series)
[49 51 60 54 47 50 64 58 47 43 50 63 67 68 64 48 55 46 66 51 52]

In [7]: table = time_series.reshape((3,7)) # reshaped *row-wise*

In [8]: print(table) # Display the result
[[49 51 60 54 47 50 64]
 [58 47 43 50 63 67 68]
 [64 48 55 46 66 51 52]]
```

Reshaping: Getting the Order Correct!

```
In [9]: print(time_series)
[49 51 60 54 47 50 64 58 47 43 50 63 67 68 64 48 55 46 66 51 52]

In [10]: time_series.reshape((7,3)) # Incorrect!
Out[10]:
array([[49, 51, 60],
       [54, 47, 50],
       [64, 58, 47],
       [43, 50, 63],
       [67, 68, 64],
       [48, 55, 46],
       [66, 51, 52]])

In [11]: time_series.reshape((7,3), order='F') # Column-wise: correct
Out[11]:
array([[49, 58, 64],
       [51, 47, 48],
       [60, 43, 55],
       [54, 50, 46],
       [47, 63, 66],
       [50, 67, 51],
       [64, 68, 52]])
```

Using reshape: Row- & Column-Major Ordering

- *Row-major* ordering (outermost index changes fastest)
  - `order='C'` (consistent with C; default)
- *Column-major* ordering (innermost index changes fastest)
  - `order='F'` (consistent with FORTRAN)

Indexing in Multiple Dimensions

```
In [12]: print(table) # Display the result
[[49 51 60 54 47 50 64]
 [58 47 43 50 63 67 68]
 [64 48 55 46 66 51 52]]

In [13]: table[0, 4] # value from Week 0, Day 4
Out[13]: 47

In [14]: table[1, 2:5] # values from Week 1, Days 2, 3, & 4
Out[14]: array([43, 50, 63])

In [15]: table[0::2, ::3] # values from Weeks 0 & 2, Days 0, 3, & 6
Out[15]:
array([[49, 54, 64],
       [64, 46, 52]])

In [16]: table[0] # Equivalent to table[0, :]
Out[16]: array([49, 51, 60, 54, 47, 50, 64])
```

Aggregating Multidimensional arrays

```
In [16]: print(table)
[[49 51 60 54 47 50 64]
 [58 47 43 50 63 67 68]
 [64 48 55 46 66 51 52]]

In [17]: table.mean() # mean of *every* entry in table
Out[17]: 54.904761904761905

In [18]: daily_means = table.mean(axis=0)

In [19] daily_means # mean computed of rows (for each day)
Out[19]:
array([ 57.        ,  48.66666667,  52.66666667,  50.        ,
        58.66666667,  56.        ,  61.33333333])

In [20]: weekly_means = table.mean(axis=1)

In [21]: weekly_means # mean computed of columns (for each week)
Out[21]: array([ 53.57142857,  56.57142857,  54.57142857])

In [22]: table.mean(axis=(0,1)) # mean of rows, then columns
Out[22]: 54.904761904761905
```

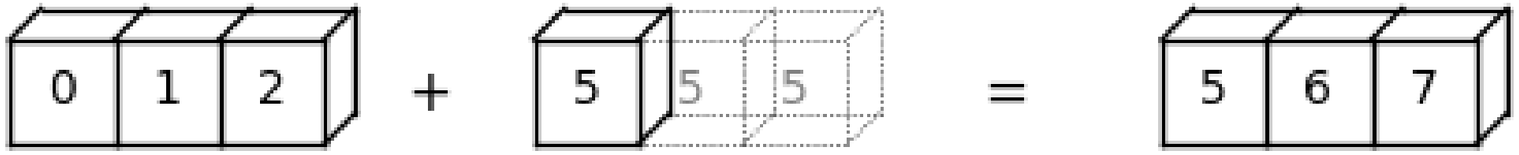
Broadcasting Arithmetic Operations

```
In [23]: table - daily_means # This works!
Out[23]:
array([[ -8.        ,   2.33333333,   7.33333333,   4.        ,
        -11.66666667,  -6.        ,   2.66666667],
       [  1.        ,  -1.66666667,  -9.66666667,   0.        ,
          4.33333333,  11.        ,   6.66666667],
       [  7.        ,  -0.66666667,   2.33333333,  -4.        ,
          7.33333333,  -5.        ,  -9.33333333]])

In [24]: table - weekly_means # This doesn't!
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
----> 1 table - weekly_means # This doesn't!

ValueError: operands could not be broadcast together with shapes (3,7) (3,)
```

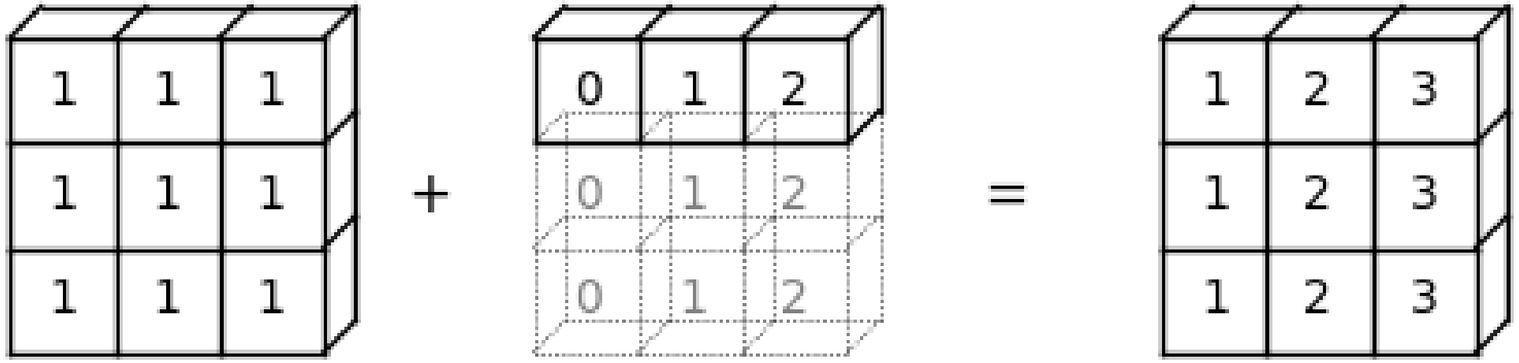
Broadcasting Rules

- **Compatible Arrays**:
    1. same `ndim`: all dimensions same or 1
    2. different `ndim`: smaller shape prepended with ones & #1. applies
- **Broadcasting**: copy array values to missing dimensions, then do arithmetic

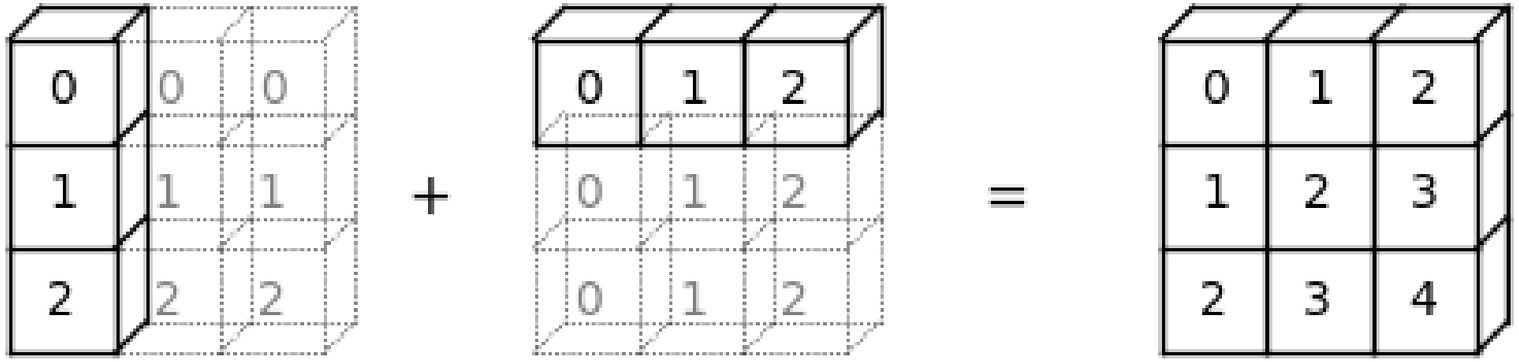
```
np.arange(3)+5
```




```
np.ones((3,3))+np.arange(3)
```




```
np.arange(3).reshape((3,1))+np.arange(3)
```



Broadcasting with table & its means

```
In [24]: print(table.shape)
(3, 7)

In [25]: print(daily_means.shape)
(7,)

In [26]: print(weekly_means.shape)
(3,)

In [27]: result = table - weekly_means.reshape((3,1)) # This works now!
```

- `table - daily_means`: `(3,7) - (7,)` → `(3,7) - (1,7)`: *compatible*
- `table - weekly_means`: `(3,7) - (3,)` → `(3,7) - (1,3)`: *incompatible*
- `table - weekly_means.reshape((3,1))`: `(3,7) - (3,1)`: *compatible*

Connecting with Dask

```
In [28]: data = np.loadtxt('', usecols=(1,2,3,4), dtype=np.int64)

In [29]: data.shape
Out[29]: (366, 4)

In [30]: type(data)
Out[30]: numpy.ndarray

In [31]: data_dask = da.from_array(data, chunks=(366,2))

In [32]: result = data_dask.std(axis=0) # Standard deviation down columns

In [33]: result.compute()
Out[33]: array([ 15.08196053,  14.9456851 ,  15.52548285,  14.47228351])
```

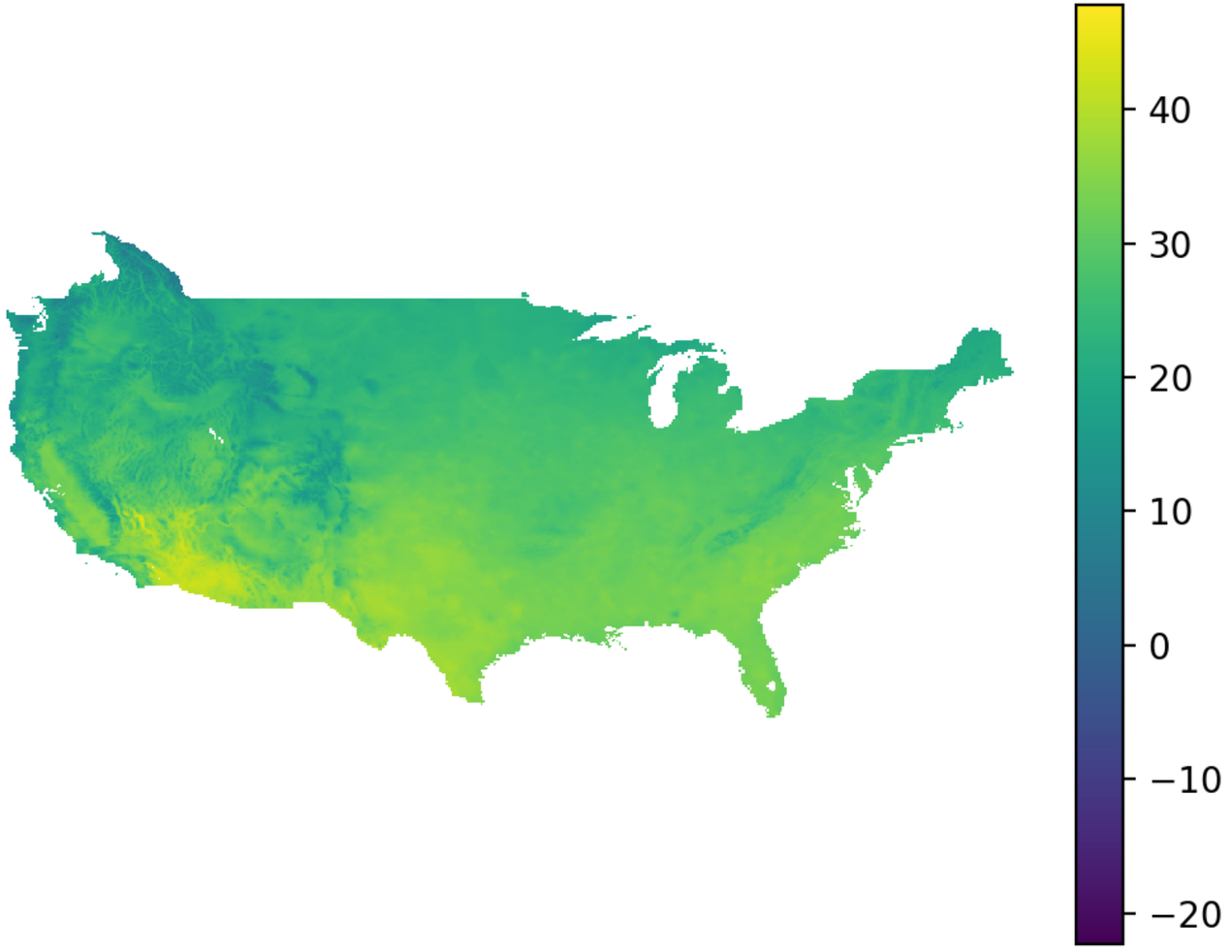
PARALLEL COMPUTING WITH DASK

Let's practice!

PARALLEL COMPUTING WITH DASK

Analyzing Weather Data

Dhavide Aruliah
Director of Training, Anaconda


Average max. daily temperature [C], June 2008
40
30
20
10
0
−10
−20

HDF5 Format

Using HDF5 Files

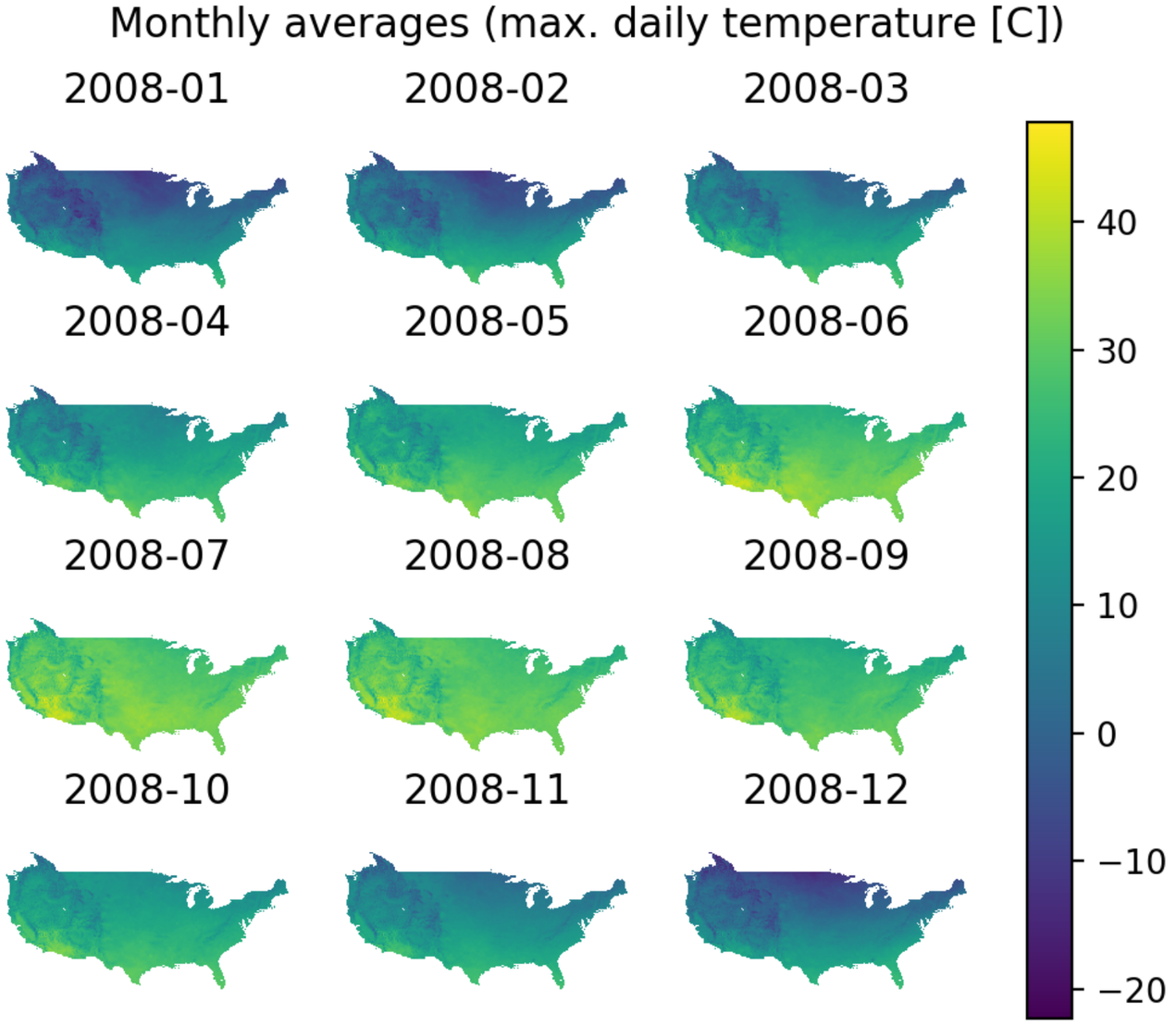
```
In [1]: import h5py # import module for reading HDF5 files

In [2]: data_store = h5py.File('tmax.2008.hdf5') # Open HDF5 File object

In [3]: for key in data_store.keys(): # iterate over keys
   ...:     print(key)

tmax
```

Extracting Dask Array from HDF5

```
In [4]: data = data_store['tmax'] # bind to data for introspection

In [5]: type(data)
Out[5]: h5py._hl.dataset.Dataset

In [6]: data.ndim
Out[6]: 3

In [7]: data.shape # Aha, 3D array: (2D for each month)
Out[7]: (12, 444, 922)

In [8]: import dask.array as da

In [9]: data_dask = da.from_array(data, chunks=(1, 444, 922))
```

Aggregating while Ignoring NaNs

```
In [10]: data_dask.min() # Yields unevaluated Dask Array
Out[10]: dask.array<amin-aggregate, shape=(), dtype=float64, chunksize=()>

In [11]: data_dask.min().compute() # Force computation
Out[11]: nan

In [12]: da.nanmin(data_dask).compute() # Ignoring nans
Out[12]: -22.329354809176536

In [13]: lo = da.nanmin(data_dask).compute()

In [14]: hi = da.nanmax(data_dask).compute()

In [15]: print(lo, hi)
-22.3293548092 47.7625806255
```


Monthly averages (max. daily temperature [C])
2008-01
2008-02
2008-03
2008-04
2008-05
2008-06
2008-07
2008-08
2008-09
2008-10
2008-11
2008-12
40
30
20
10
0
−10
−20

Producing a Visualization of data_dask

```
In [16]: N_months = data_dask.shape[0] # Number of images

In [17]: import matplotlib.pyplot as plt

In [18]: fig, panels = plt.subplots(nrows=4, ncols=3)

In [19]: for month, panel in zip(range(N_months), panels.flatten()):
    ...:     im = panel.imshow(data_dask[month, :, :],
    ...:                       origin='lower',
    ...:                       vmin=lo, vmax=hi)
    ...:
    ...:     panel.set_title('2008-{:02d}'.format(month+1))
    ...:     panel.axis('off')

In [20]: plt.suptitle('Monthly averages (max. daily temperature [C])');

In [21]: plt.colorbar(im, ax=panels.ravel().tolist()); # Common colorbar

In [22]: plt.show()
```

Stacking Arrays

```
In [23]: import numpy as np

In [24]: a = np.ones(3); b = 2 * a; c = 3 * a

In [25]: print(a, '\n'); print(b, '\n'); print(c)
[ 1.  1.  1.]

[ 2.  2.  2.]

[ 3.  3.  3.]
```

Stacking One-Dimensional Arrays (I)

```
In [26]: np.stack([a, b]) # makes 2D array of shape (2,3)
Out[26]:
array([[ 1.,  1.,  1.],
       [ 2.,  2.,  2.]])

In [27]: np.stack([a, b], axis=0) # same as above
Out[27]:
array([[ 1.,  1.,  1.],
       [ 2.,  2.,  2.]])

In [28]: np.stack([a, b], axis=1) # makes 2D array of shape (3,2)
Out[28]:
array([[ 1.,  2.],
       [ 1.,  2.],
       [ 1.,  2.]])
```

Stacking One-Dimensional Arrays (II)

```
In [29]: X = np.stack([a, b]); \
    ...: Y = np.stack([b, c]); \
    ...: Z = np.stack([c, a])

In [30]: print(X, '\n'); print(Y, '\n'); print(Z, '\n')
[[ 1.  1.  1.]
 [ 2.  2.  2.]]

[[ 2.  2.  2.]
 [ 3.  3.  3.]]

[[ 3.  3.  3.]
 [ 1.  1.  1.]]
```

Stacking Two-Dimensional Arrays

```
In [31]: np.stack([X, Y, Z]) # makes 3D array of shape (3, 2, 3)
Out[31]:
array([[[ 1.,  1.,  1.],
        [ 2.,  2.,  2.]],

       [[ 2.,  2.,  2.],
        [ 3.,  3.,  3.]],

       [[ 3.,  3.,  3.],
        [ 1.,  1.,  1.]]])

In [32]: np.stack([X, Y, Z], axis=1) # makes 3D array of shape (2, 3, 3)
Out[32]:
array([[[ 1.,  1.,  1.],
        [ 2.,  2.,  2.],
        [ 3.,  3.,  3.]],

       [[ 2.,  2.,  2.],
        [ 3.,  3.,  3.],
        [ 1.,  1.,  1.]]])
```

Putting Array Blocks Together

PARALLEL COMPUTING WITH DASK

Let's practice!