Chunking Arrays in Dask

PARALLEL PROGRAMMING WITH DASK IN PYTHON

Dhavide Aruliah
Director of Training, Anaconda

datacamp

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

```
import numpy as np
a = np.random.rand(10000)
print(a.shape, a.dtype)
```

```
(10000,) float64
```

```
print(a.sum())
```

```
5017.32043995
```

```
print(a.mean())
```

```
0.501732043995
```

Working with Dask arrays

```
import dask.array as da
a_dask = da.from_array(a, chunks=len(a) // 4)
a_dask.chunks
```

```
((2500, 2500, 2500, 2500),)
```

Aggregating in chunks

```
n_chunks = 4
chunk_size = len(a) // n_chunks
result = 0 # Accumulate sum
for k in range(n_chunks):
     offset = k * chunk_size # Track offset
     a_chunk= a[offset:offset + chunk_size] # Slice chunk
     result += a_chunk.sum()
print(result)
```

```
5017.32043995
```

Aggregating with Dask arrays

```
a_dask = da.from_array(a, chunks=len(a)//n_chunks)
result = a_dask.sum()
result
```

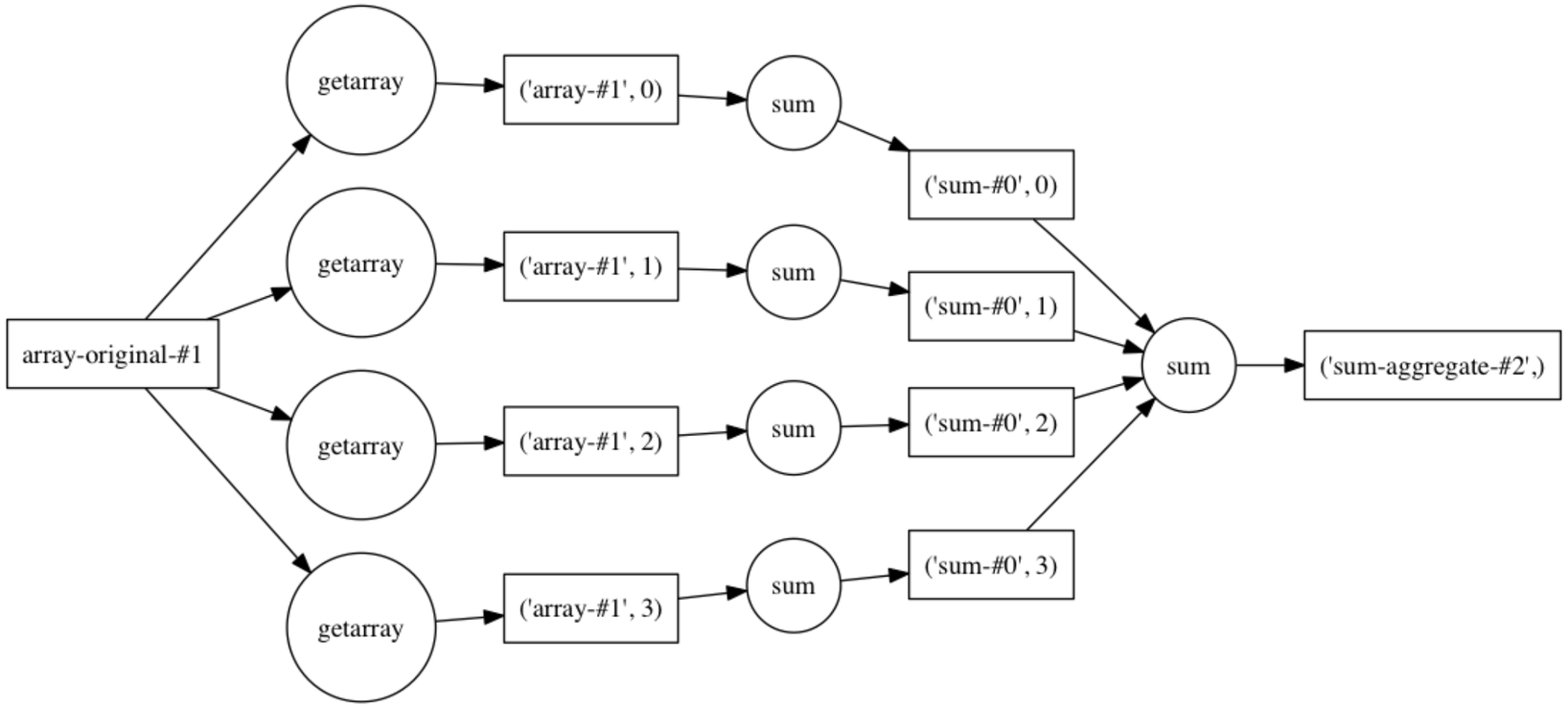
```
dask.array<sum-aggregate, shape=(), dtype=float64, chunksize=()>
```

```
print(result.compute())
```

```
5017.32043995
```

```
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

```
import h5py, time

with h5py.File('dist.hdf5', 'r') as dset:
...:     dist = dset['dist'][:]
dist_dask8 = da.from_array(dist, chunks=dist.shape[0]//8)
t_start = time.time(); \
...: mean8 = dist_dask8.mean().compute(); \
...: t_end = time.time()
t_elapsed = (t_end - t_start) * 1000 # Elapsed time in ms
print('Elapsed time: {} ms'.format(t_elapsed))
```

```
Elapsed time: 180.96423149108887 ms
```

Let's practice!

PARALLEL PROGRAMMING WITH DASK IN PYTHON

Computing with Multidimensional Arrays

PARALLEL PROGRAMMING WITH DASK IN PYTHON

Dhavide Aruliah
Director of Training, Anaconda

datacamp

A Numpy array of time series data

```
import numpy as np
time_series = np.loadtxt('max_temps.csv', dtype=np.int64)
print(time_series.dtype)
```

```
int64
```

```
print(time_series.shape)
```

```
(21,)
```

```
print(time_series.ndim)
```

```
1
```

Reshaping time series data

```
print(time_series)
```

```
[49 51 60 54 47 50 64 58 47 43 50 63 67 68 64 48 55 46 66 51 52]
```

```
table = time_series.reshape((3,7)) # Reshaped row-wise
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!

```
print(time_series)
```

```
[49 51 60 54 47 ... 46 66 51 52]
```

```
 # Incorrect!
time_series.reshape((7,3))
```

```
array([[49, 51, 60],
       [54, 47, 50],
       [64, 58, 47],
       [43, 50, 63],
       [67, 68, 64],
       [48, 55, 46],
       [66, 51, 52]])
```

```
# Column-wise: correct
time_series.reshape((7,3),
    order='F')
```

```
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

```
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]]
```

```
table[0, 4] # value from Week 0, Day 4
```

```
47
```

```
table[1, 2:5] # values from Week 1, Days 2, 3, & 4
```

```
array([43, 50, 63])
```

Indexing in multiple dimensions

```
table[0::2, ::3] # values from Weeks 0 & 2, Days 0, 3, & 6
```

```
array([[49, 54, 64],
       [64, 46, 52]])
```

```
table[0] # Equivalent to table[0, :]
```

```
array([49, 51, 60, 54, 47, 50, 64])
```

Aggregating multidimensional arrays

```
print(table)
```

```
[[49 51 60 54 47 50 64]
 [58 47 43 50 63 67 68]
 [64 48 55 46 66 51 52]]
```

```
table.mean() # mean of *every* entry in table
```

```
54.904761904761905
```

```
# Averages for days
daily_means = table.mean(axis=0)
```

Aggregating multidimensional arrays

```
daily_means # Mean computed of rows (for each day)
```

```
array([ 57.        ,  48.66666667,  52.66666667,  50.        ,
        58.66666667,  56.        ,  61.33333333])
```

```
weekly_means = table.mean(axis=1)
weekly_means # mean computed of columns (for each week)
```

```
array([ 53.57142857,  56.57142857,  54.57142857])
```

```
table.mean(axis=(0,1)) # mean of rows, then columns
```

```
54.904761904761905
```

```
table - daily_means # This works!
```

```
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]])
```

```
table - weekly_means # This doesn't!
```

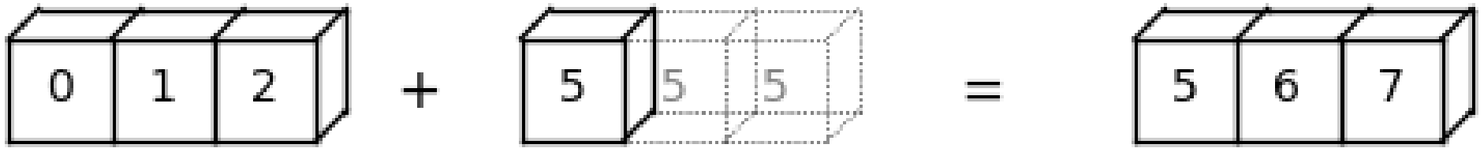
```
ValueError                                Traceback (most recent call last)
    ---> 1 table - weekly_means # This doesn't!

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

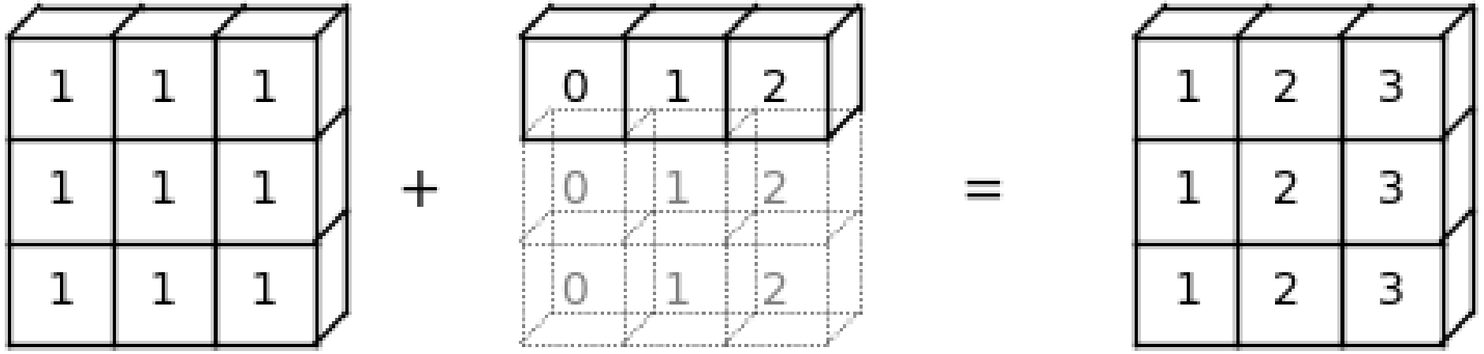
Broadcasting rules

- **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

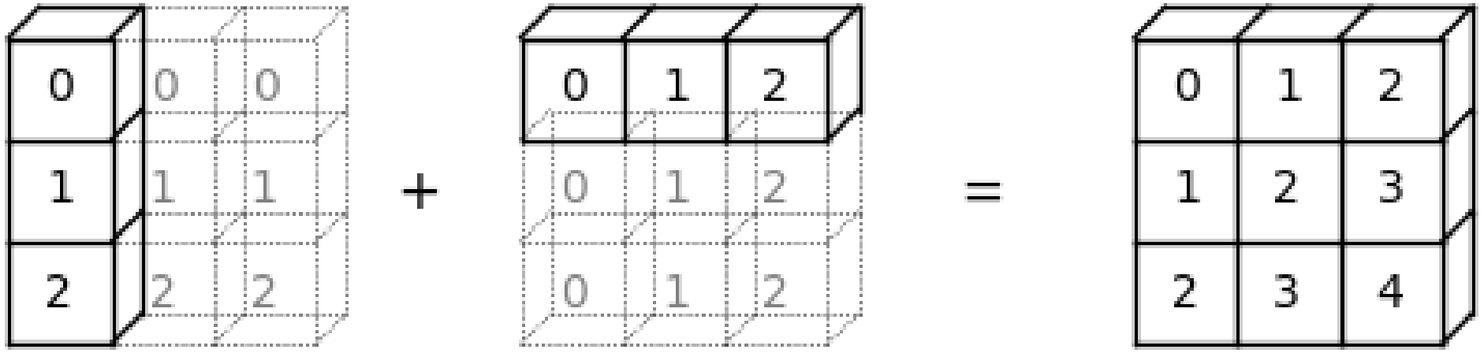
```
np.arange(3)+5
```




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




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



```
print(table.shape)
```

```
(3, 7)
```

```
print(daily_means.shape)
```

```
(7,)
```

```
print(weekly_means.shape)
```

```
(3,)
```

```
# This works now!
result = table -
    weekly_means.reshape((3,1))
```

- `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

```
data = np.loadtxt('', usecols=(1,2,3,4), dtype=np.int64)
data.shape
```

```
(366, 4)
```

```
type(data)
```

```
numpy.ndarray
```

```
data_dask = da.from_array(data, chunks=(366,2))
result = data_dask.std(axis=0) # Standard deviation down columns
result.compute()
```

```
array([ 15.08196053,  14.9456851 ,  15.52548285,  14.47228351])
```

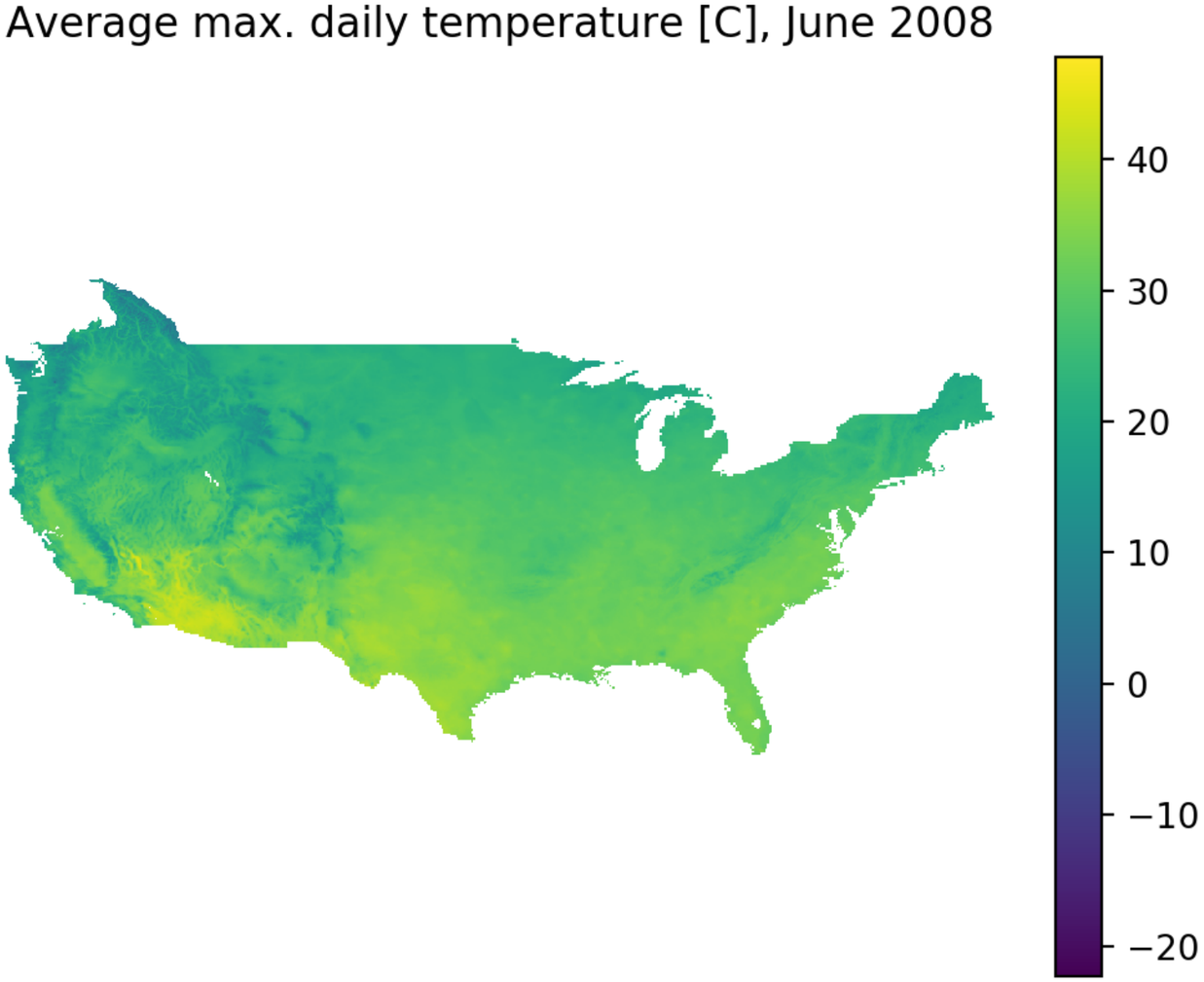
Let's practice!

PARALLEL PROGRAMMING WITH DASK IN PYTHON

Analyzing Weather Data

PARALLEL PROGRAMMING WITH DASK IN PYTHON

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

```
import h5py # import module for reading HDF5 files


# Open HDF5 File object
data_store = h5py.File('tmax.2008.hdf5')
for key in data_store.keys(): # iterate over keys
     print(key)
```

```
tmax
```

Extracting Dask array from HDF5

```
data = data_store['tmax'] # bind to data for introspection
type(data)
```

```
h5py._hl.dataset.Dataset
```

```
data.shape # Aha, 3D array: (2D for each month)
```

```
(12, 444, 922)
```

```
import dask.array as da
data_dask = da.from_array(data, chunks=(1, 444, 922))
```

Aggregating while ignoring NaNs

```
data_dask.min() # Yields unevaluated Dask Array
```

```
dask.array<amin-aggregate, shape=(), dtype=float64, chunksize=()>
```

```
data_dask.min().compute() # Force computation
```

```
nan
```

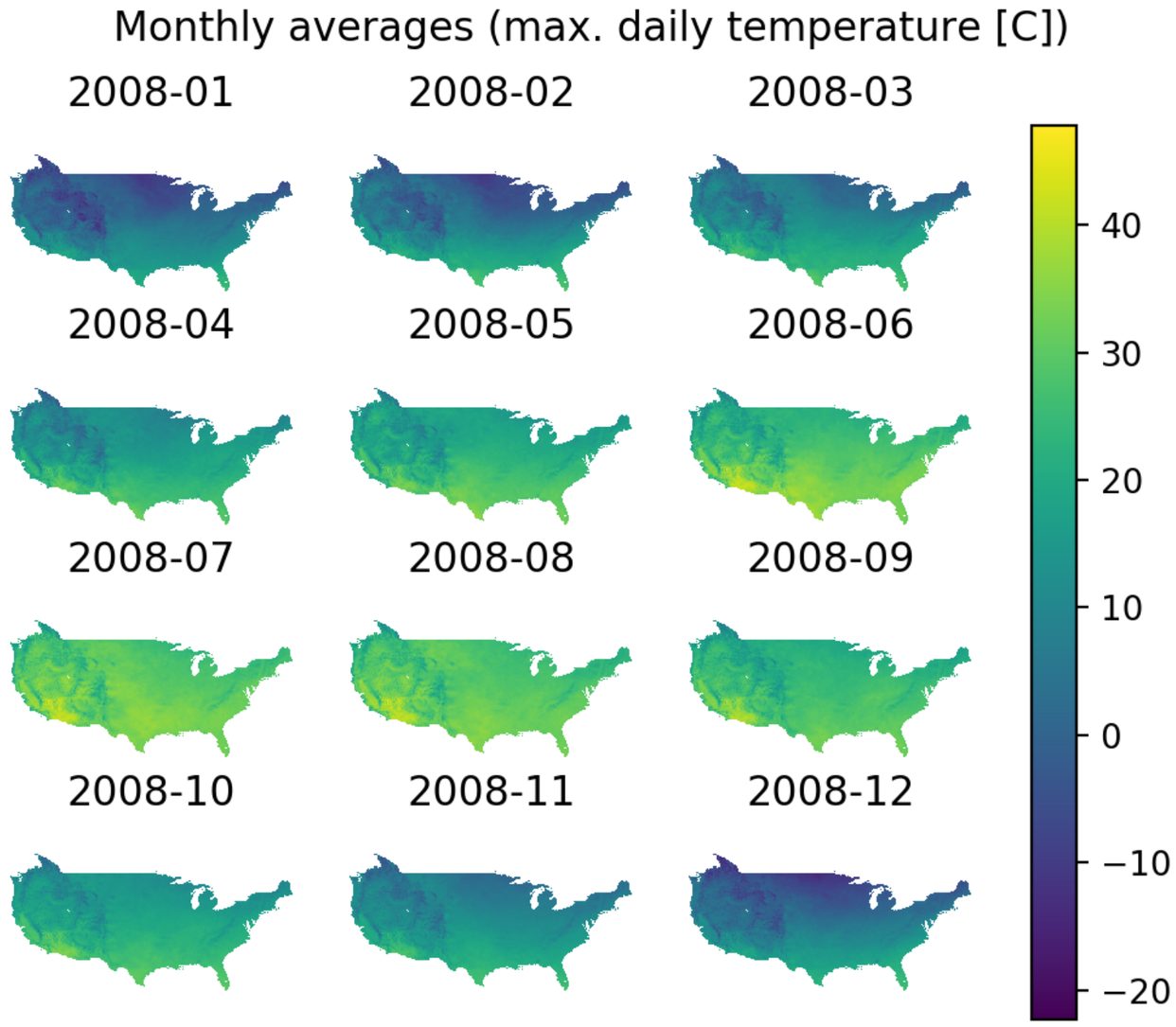
Aggregating while ignoring NaNs

```
da.nanmin(data_dask).compute() # Ignoring nans
```

```
-22.329354809176536
```

```
lo = da.nanmin(data_dask).compute()
hi = da.nanmax(data_dask).compute()
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

```
N_months = data_dask.shape[0] # Number of images
import matplotlib.pyplot as plt
fig, panels = plt.subplots(nrows=4, ncols=3)
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')

plt.suptitle('Monthly averages (max. daily temperature [C])');
plt.colorbar(im, ax=panels.ravel().tolist()); # Common colorbar
plt.show()
```

Stacking arrays

```
import numpy as np
a = np.ones(3); b = 2 * a; c = 3 * a
print(a, '\n'); print(b, '\n'); print(c)
```

```
[ 1.  1.  1.]

[ 2.  2.  2.]

[ 3.  3.  3.]
```

```
np.stack([a, b]) # Makes 2D array of shape (2,3)
```

```
array([[ 1.,  1.,  1.],
       [ 2.,  2.,  2.]])
```

```
np.stack([a, b], axis=0) # Same as above
```

```
array([[ 1.,  1.,  1.],
       [ 2.,  2.,  2.]])
```

```
np.stack([a, b], axis=1) # Makes 2D array of shape (3,2)
```

```
array([[ 1.,  2.],
       [ 1.,  2.],
       [ 1.,  2.]])
```

Stacking one-dimensional arrays

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

```
np.stack([X, Y, Z]) # Makes 3D array of shape (3, 2, 3)
```

```
array([[[ 1.,  1.,  1.],
        [ 2.,  2.,  2.]],

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

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

Stacking two-dimensional arrays

```
# Makes 3D array of shape (2, 3, 3)
np.stack([X, Y, Z], axis=1)
```

```
array([[[ 1.,  1.,  1.],
        [ 2.,  2.,  2.],
        [ 3.,  3.,  3.]],

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

Putting array blocks together

Let's practice!

PARALLEL PROGRAMMING WITH DASK IN PYTHON