Dask

DataFrames: Read and Write Data

Dask Dataframes can read and store data in many of the same formats as Pandas dataframes. In this example we read and write data with the popular CSV and Parquet formats, and discuss best practices when using these formats.

Start Dask Client for Dashboard

Starting the Dask Client is optional. It will provide a dashboard which is useful to gain insight on the computation.

The link to the dashboard will become visible when you create the client below. We recommend having it open on one side of your screen while using your notebook on the other side. This can take some effort to arrange your windows, but seeing them both at the same is very useful when learning.

In [ ]:

```
from dask.distributed import Client
client = Client(n_workers=1, threads_per_worker=4, processes=False, me
mory_limit='2GB')
client
```

```
#ojo esto os puede fallar en sistemas remotos, se puede prescindir par
a la ejecución, en local os deberi ir, siempre que tengamos las depend
encias necesarias
```

Create artificial dataset

First we create an artificial dataset and write it to many CSV files.

You don't need to understand this section, we're just creating a dataset for the rest of the notebook.

In [ ]:

```
import dask
df = dask.datasets.timeseries()
df
```

In [ ]:

```
import os
import datetime

if not os.path.exists('data'):
    os.mkdir('data')

def name(i):
    """ Provide date for filename given index
    Examples
    --------
    >>> name(0)
    '2000-01-01'
    >>> name(10)
    '2000-01-11'
    """
    return str(datetime.date(2000, 1, 1) + i * datetime.timedelta(days
=1))

df.to_csv('data/*.csv', name_function=name);
```

Read CSV files

We now have many CSV files in our data directory, one for each day in the month of January 2000. Each CSV file holds timeseries data for that day. We can read all of them as one logical dataframe using the `dd.read_csv` function with a glob string.

In [ ]:

```
!ls data/*.csv | head
```

In [ ]:

```
!head data/2000-01-01.csv
```

In [ ]:

```
!head data/2000-01-30.csv
```

We can read one file with `pandas.read_csv` or many files with `dask.dataframe.read_csv`

In [ ]:

```
import pandas as pd

df = pd.read_csv('data/2000-01-01.csv')
df.head()
```

In [ ]:

```
import dask.dataframe as dd

df = dd.read_csv('data/2000-*-*.csv')
df
```

In [ ]:

```
df.head()
```

Tuning read_csv

The Pandas `read_csv` function has *many* options to help you parse files. The Dask version uses the Pandas function internally, and so supports many of the same options. You can use the `?` operator to see the full documentation string.

In [ ]:

```
pd.read_csv?
```

In [ ]:

```
dd.read_csv?
```

In this case we use the `parse_dates` keyword to parse the timestamp column to be a datetime. This will make things more efficient in the future. Notice that the dtype of the timestamp column has changed from `object` to `datetime64[ns]`.

In [ ]:

```
df = dd.read_csv('data/2000-*-*.csv', parse_dates=['timestamp'])
df
```

Do a simple computation

Whenever we operate on our dataframe we read through all of our CSV data so that we don't fill up RAM. This is very efficient for memory use, but reading through all of the CSV files every time can be slow.

In [ ]:

```
%time df.groupby('name').x.mean().compute()
```

In [ ]:

Write to Parquet

Instead, we'll store our data in Parquet, a format that is more efficient for computers to read and write.

In [ ]:

```
df.to_parquet('data/2000-01.parquet', engine='pyarrow')
```

In [ ]:

```
!ls data/2000-01.parquet/
```

Read from Parquet

In [ ]:

```
df = dd.read_parquet('data/2000-01.parquet', engine='pyarrow')
df
```

In [ ]:

```
%time df.groupby('name').x.mean().compute()
```

Select only the columns that you plan to use

Parquet is a column-store, which means that it can efficiently pull out only a few columns from your dataset. This is good because it helps to avoid unnecessary data loading.

In [ ]:

```
%%time
df = dd.read_parquet('data/2000-01.parquet', columns=['name', 'x'], en
gine='pyarrow')
df.groupby('name').x.mean().compute()
```

Here the difference is not that large, but with larger datasets this can save a great deal of time.

DataFrames: Groupby

This notebook uses the Pandas groupby-aggregate and groupby-apply on scalable Dask dataframes. It will discuss both common use and best practices.

Start Dask Client for Dashboard

Starting the Dask Client is optional. It will provide a dashboard which is useful to gain insight on the computation.

The link to the dashboard will become visible when you create the client below. We recommend having it open on one side of your screen while using your notebook on the other side. This can take some effort to arrange your windows, but seeing them both at the same is very useful when learning.

In [ ]:

```
from dask.distributed import Client
client = Client(n_workers=1, threads_per_worker=4, processes=False, me
mory_limit='2GB')
client
```

Artifical dataset

We create an artificial timeseries dataset to help us work with groupby operations

In [ ]:

```
import dask
df = dask.datasets.timeseries()
df
```

This dataset is small enough to fit in the cluster's memory, so we persist it now.

You would skip this step if your dataset becomes too large to fit into memory.

In [ ]:

```
df = df.persist()
```

Groupby Aggregations

Dask dataframes implement a commonly used subset of the Pandas groupby API (see Pandas Groupby Documentation.

We start with groupby aggregations. These are generally fairly efficient, assuming that the number of groups is small (less than a million).

In [ ]:

```
df.groupby('name').x.mean().compute()
```

Performance will depend on the aggregation you do (mean vs std), the key on which you group (name vs id), and the number of total groups

In [ ]:

```
%time _ = df.groupby('id').x.mean().compute()
```

In [ ]:

```
%time _ = df.groupby('name').x.mean().compute()
```

In [ ]:

```
%time df.groupby('name').agg({'x': ['mean', 'std'], 'y': ['mean', 'cou
nt']}).compute().head()
```

This is the same as with Pandas. Generally speaking, Dask.dataframe groupby-aggregations are roughly same performance as Pandas groupby-aggregations, just more scalable.
You can read more about Pandas' common aggregations in the Pandas documentation.

Custom Aggregations

Dask dataframe Aggregate is available for custom aggregations (See Dask dataframe Aggregate Documentation)

Many groups

By default groupby-aggregations (like groupby-mean or groupby-sum) return the result as a single-partition Dask dataframe. Their results are *usually* quite small, so this is *usually* a good choice.
However, sometimes people want to do groupby aggregations on *many* groups (millions or more). In these cases the full result may not fit into a single Pandas dataframe output, and you may need to split your output into multiple partitions. You can control this with the `split_out=` parameter

In [ ]:

```
# Computational graph of a single output aggregation (for a small numb
er of groups, like 1000)
df.groupby('name').x.mean().visualize(node_attr={'penwidth': '6'})
```

In [ ]:

```
# Computational graph of an aggregation to four outputs (for a larger
number of groups, like 1000000)
df.groupby('id').x.mean(split_out=4).visualize(node_attr={'penwidth':
'6'})
```

Groupby Apply

Groupby-aggregations are generally quite fast because they can be broken down easily into well known operations. The data doesn't have to move around too much and we can just pass around small intermediate values across the network.
For some operations however the function to be applied requires *all* data from a given group (like every record of someone named "Alice"). This will force a great deal of communication and be more expensive, but is still possible with the Groupby-apply method. This should be avoided if a groupby-aggregation works.

In the following example we train a simple Scikit-Learn machine learning model on every person's name.

In [ ]:

```
from sklearn.linear_model import LinearRegression

def train(partition):
    est = LinearRegression()
    est.fit(partition[['x', 'id']].values, partition.y.values)
    return est
```

In [ ]:

```
%time df.groupby('name').apply(train, meta=object).compute().sort_inde
x()
```

Gotcha's from Pandas to Dask

This notebook highlights some key differences when transfering code from `Pandas` to run in a `Dask` environment.
Most issues have a link to the Dask documentation for additional information.

In [1]:

```
# since Dask is activly beeing developed - the current example is runn
ing with the below version
```

```
import dask
import dask.dataframe as dd
import pandas as pd
print(f'Dask versoin: {dask.__version__}')
print(f'Pandas versoin: {pd.__version__}')
Dask versoin: 1.2.2
Pandas versoin: 0.24.2
```

Start Dask Client for Dashboard

Starting the Dask Client is optional. In this example we are running on a `LocalCluster`, this will also provide a dashboard which is useful to gain insight on the computation. For additional information on Dask Client see documentation
The link to the dashboard will become visible when you create a client (as shown below). When running within `Jupyter Lab` an extenstion can be installed to view the various dashboard widgets.

In [2]:

```
from dask.distributed import Client
# client = Client(n_workers=1, threads_per_worker=4, processes=False,
memory_limit='2GB')
client = Client()
client
```

Out[2]:

Client • **Scheduler:** tcp://127.0.0.1:58069 • **Dashboard:** http://127.0.0.1:8787/status	Cluster • **Workers:** 4 • **Cores:** 4 • **Memory:** 8.50 GB

See documentation for addtional cluster configuration

Create 2 DataFrames for comparison:

1. for Dask
2. for Pandas

   Dask comes with builtin dataset samples, we will use this sample for our example.

In [3]:

```
ddf = dask.datasets.timeseries()
ddf
```

Out[3]:

Dask DataFrame Structure:

	id	name	x	y
npartitions=30				
2000-01-01	int32	object	float64	float64
2000-01-02	...	...	...	...

	id	name	x	y
npartitions=30				
...	...	...	...	...
2000-01-30	...	...	...	...
2000-01	...	...	...	...

	id	name	x	y
npartitions=30				
-31				

Dask Name: make-timeseries, 30 tasks

- Remember `Dask framework` is **lazy** thus in order to see the result we need to run compute() (or `head()` which runs under the hood compute()))

In [4]:

```
ddf.head(2)
```

Out[4]:

	id	name	x	y
timestamp				
2000-01-01 00:00:00	972	Alice	0.532136	0.229624
2000-01-0	964	Patricia	0.004575	-0.3445

	id	name	x	y
timestamp				
1000:00:01				98

Pandas Dataframe

In order to create a `Pandas` dataframe we can use the `compute()` method from a `Dask dataframe`

In [5]:

```
pdf = ddf.compute()
print(type(pdf))
pdf.head(2)
<class 'pandas.core.frame.DataFrame'>
```

Out[5]:

	id	**name**	**x**	**y**
timestamp				
2000-01-01 00:00:00	972	Alice	0.532136	0.229624
2000-01-0	964	Patricia	0.004575	-0.3445

	id	name	x	y
timestamp				
1000:00:01				98

dataframe.shape

We can also see *dask laziness* when using the shape attribute

In [6]:

```
print(f'Pandas shape: {pdf.shape}')
print('---------------------------')
print(f'Dask lazy shape: {ddf.shape}')
Pandas shape: (2592000, 4)
---------------------------
Dask lazy shape: (Delayed('int-f7698154-62ad-435d-a7fa-ec5212b44a1c'),
4)
```

We cannot get the full shape before accessing all the partitions - running `len` will do so

In [7]:

```
print(f'Dask computed shape: {len(ddf.index):,}')  # expensive
Dask computed shape: 2,592,000
```

Creating a `Dask` `dataframe` from `Pandas`

In order to utilize `Dask` capablities on an existing `Pandas` `dataframe` (pdf) we need to convert the `Pandas` `dataframe` into a `Dask` `dataframe` (ddf) with the from_pandas method. You must supply the number of partitions or chunksize that will be used to generate the dask dataframe

In [8]:

```
ddf2 = dask.dataframe.from_pandas(pdf, npartitions=10)
ddf2
```

Out[8]:

Dask DataFrame Structure:

	id	name	x	y
npartitions=10				
2000-01-01 00:00:	int32	object	float64	float64

	id	name	x	y
npartitions=100				
00				
2000-01-04 00:00:00	...	...	...	...

	id	name	x	y
npartitions=10				
...	...	...	...	...
2000-01-28 00:00:00	...	...	...	...

	id	name	x	y
npartitions=10				
2000-01-30 23:59:59	...	...	...	...

Dask Name: from_pandas, 10 tasks

Partitions in Dask Dataframes

Notice that when we created a `Dask dataframe` we needed to supply an argument of `npartitions`.

The number of partitions will assist `Dask` on how to breakup the `Pandas Datafram` and parallelize the computation.

Each partition is a *separate* dataframe. For additional information see partition documentation
An example for this can be seen when examing the `reset_ index()` method:

In [9]:

```
pdf2 = pdf.reset_index()
# Only 1 row
pdf2.loc[0]
```

Out[9]:

```
timestamp    2000-01-01 00:00:00
id                           972
name                       Alice
x                       0.532136
y                       0.229624
Name: 0, dtype: object
```

In [10]:

```
ddf2 = ddf2.reset_index()
# each partition has an index=0
ddf2.loc[0].compute()
```

```
c:\users\jsber\.virtualenvs\dask-examples-3r4mgfnb\lib\site-packages\d
istributed\worker.py:3101: UserWarning: Large object of size 9.42 MB d
etected in task graph:
  (slice(0, 0, None), None, 'reset_index-8356d669ad1 ... 54123c422adde
')
Consider scattering large objects ahead of time
with client.scatter to reduce scheduler burden and
keep data on workers

    future = client.submit(func, big_data)    # bad

    big_future = client.scatter(big_data)     # good
    future = client.submit(func, big_future)  # good
  % (format_bytes(len(b)), s)
```

Out[10]:

	timestamp	id	name	x	y
0	2000-01-01	972	Alice	0.532136	0.229624
0	2000-01-04	994	Ingrid	0.380598	-0.739793
0	2000-01-07	974	Bob	0.927427	0.390628

	timestamp	id	name	x	y
0	2000-01-10	1016	Wendy	-0.485949	-0.738365
0	2000-01-13	973	Xavier	0.708157	-0.507682
0	2000-01-16	966	Edith	0.161352	-0.822572

	timestamp	id	name	x	y
0	2000-01-19	971	Yvonne	-0.193811	0.302984
0	2000-01-22	976	Yvonne	-0.829687	-0.782505
0	2000-01-25	1026	Laura	-0.597932	-0.762958

	timestamp	id	name	x	y
0	2000-01-28	958	Ingrid	0.773340	0.877205

Dask Dataframe vs Pandas Dataframe

Now that we have a `dask` (ddf) and a `pandas` (pdf) dataframe we can start to compair the interactions with them.

Conceptual shift - from Update to Insert/Delete

Dask does not update - thus there are no arguments such as `inplace=True` which exist in Pandas.
For more detials see [issue#653 on github]

Rename Columns

- using `inplace=True` is not considerd to be *best practice*.

In [11]:

```
# Pandas
print(pdf.columns)
# pdf.rename(columns={'id':'ID'}, inplace=True)
pdf = pdf.rename(columns={'id':'ID'})
pdf.columns
Index(['id', 'name', 'x', 'y'], dtype='object')
```

Out[11]:

```
Index(['ID', 'name', 'x', 'y'], dtype='object')
```

# Dask - Error # ddf.rename(columns={'id':'ID'}, inplace=True) # ddf.columns ''' python -- TypeError Traceback (most recent call last)

Start Dask Client for Dashboard

Starting the Dask Client is optional. It will provide a dashboard which is useful to gain insight on the computation.

The link to the dashboard will become visible when you create the client below. We recommend having it open on one side of your screen while using your notebook on the other side. This can take some effort to arrange your windows, but seeing them both at the same is very useful when learning.

In [ ]:

```
from dask.distributed import Client
client = Client(n_workers=1, threads_per_worker=4, processes=False, me
mory_limit='2GB')
client
```

Create artificial dataset

First we create an artificial dataset and write it to many CSV files.

You don't need to understand this section, we're just creating a dataset for the rest of the notebook.

In [ ]:

```
import dask
df = dask.datasets.timeseries()
df
```

In [ ]:

```
import os
import datetime

if not os.path.exists('data'):
    os.mkdir('data')

def name(i):
    """ Provide date for filename given index

    Examples
    --------
    >>> name(0)
    '2000-01-01'
    >>> name(10)
    '2000-01-11'
    """
    return str(datetime.date(2000, 1, 1) + i * datetime.timedelta(days
=1))

df.to_csv('data/*.csv', name_function=name, index=False);
```

Read CSV files

We now have many CSV files in our data directory, one for each day in the month of January 2000. Each CSV file holds timeseries data for that day. We can read all of them as one logical dataframe using the `dd.read_csv` function with a glob string.

In [ ]:

```
!ls data/*.csv | head
```

In [ ]:

```
import dask.dataframe as dd

df = dd.read_csv('data/2000-*-*.csv')
df
```

In [ ]:

```
df.head()
```

Let's look at some statistics on the data

In [ ]:

```
df.describe().compute()
```

Make some messy data

Now this works great, and in most cases dd.read_csv or dd.read_parquet etc are the preferred way to read in large collections of data files into a dask dataframe, but real world data is often very messy and some files may be broken or badly formatted. To simulate this we are going to create some fake messy data by tweaking our example csv files. For the file `data/2000-01-05.csv` we will replace with no data and for the file `data/2000-01-07.csv` we will remove the `y` column

In [ ]:

```
# corrupt the data in data/2000-01-05.csv
with open('data/2000-01-05.csv', 'w') as f:
    f.write('')
```

In [ ]:

```
# remove y column from data/2000-01-07.csv
import pandas as pd
df = pd.read_csv('data/2000-01-07.csv')
del df['y']
df.to_csv('data/2000-01-07.csv', index=False)
```

In [ ]:

```
!head data/2000-01-05.csv
```

In [ ]:

```
!head data/2000-01-07.csv
```

Reading the messy data

Let's try reading in the collection of files again

In [ ]:

```
df = dd.read_csv('data/2000-*-*.csv')
```

In [ ]:

```
df.head()
```

Ok this looks like it worked, let us calculate the dataset statistics again

In [ ]:

```
df.describe().compute()
```

So what happened?

When creating a dask dataframe from a collection of files, dd.read_csv samples the first few files in the dataset to determine the datatypes and columns available. Since it has not opened all the files it does not now if some of them are corrupt. Hence, `df.head()` works since it is only looking at the first file. `df.describe.compute()` fails because of the corrupt data in `data/2000-01-05.csv`

Building a delayed reader

To get around this problem we are going to use a more advanced technique to build our dask dataframe. This method can also be used any time some custom logic is required when reading each file. Essentially, we are going to build a function that uses pandas and some error checking and returns a pandas dataframe. If we find a bad data file we will either find a way to fix/clean the data or we will return and empty pandas dataframe with the same structure as the good data.

In [ ]:

```
import numpy as np
import io

def read_data(filename):

    # for this to work we need to explicitly set the datatypes of our
pandas dataframe
    dtypes = {'id': int, 'name': str, 'x': float, 'y': float}
    try:
        # try reading in the data with pandas
        df = pd.read_csv(filename, dtype=dtypes)
    except:
        # if this fails create an empty pandas dataframe with the same
dtypes as the good data
        df = pd.read_csv(io.StringIO(''), names=dtypes.keys(), dtype=d
types)

    # for the case with the missing column, add a column of data with
NaN's
    if 'y' not in df.columns:
        df['y'] = np.NaN

    return df
```

Let's test this function on a good file and the two bad files

In [ ]:

```
# test function on a normal file
read_data('data/2000-01-01.csv').head()
```

In [ ]:

```
# test function on the empty file
```

```
read_data('data/2000-01-05.csv').head()
```

In [ ]:

```
# test function on the file missing the y column
read_data('data/2000-01-07.csv').head()
```

Assembling the dask dataframe

First we take our `read_data` function and convert it to a dask delayed function

In [ ]:

```
from dask import delayed
read_data = delayed(read_data)
```

Let us look at what the function does now

In [ ]:

```
df = read_data('data/2000-01-01.csv')
df
```

It creates a delayed object, to actually run read the file we need to run `.compute()`

In [ ]:

```
df.compute()
```

Now let's build a list of all the available csv files

In [ ]:

```
# loop over all the files
from glob import glob
files = glob('data/2000-*-*.csv')
files
```

Now we run the delayed read_data function on each file in the list

In [ ]:

```
df = [read_data(file) for file in files]
df
```

Then we use dask.dataframe.from_delayed. This function creates a Dask DataFrame from a list of delayed objects as long as each delayed object returns a pandas dataframe. The structure of each individual dataframe returned must also be the same.

In [ ]:

```
df = dd.from_delayed(df, meta={'id': int, 'name': str, 'x': float, 'y'
: float})
df
```

Note: we provided the dtypes in the `meta` keyword to explicitly tell Dask Dataframe what kind of dataframe to expect. If we did not do this Dask would infer this from the first delayed object which could be slow if it was a large csv file

Now let's see if this works

In [ ]:

```
df.head()
```

In [ ]:

```
df.describe().compute()
```

Async/Await and Non-Blocking Execution

Dask integrates natively with concurrent applications using the Tornado or Asyncio frameworks, and can make use of Python's `async` and `await` keywords.

This example shows a small example how how to start up a Dask Client in asynchronous mode.

The `asynchronous=True` parameter

Dask LocalCluster and Client objects can operate in async-await mode if you pass the `asynchronous=True` parameter.

In [ ]:

```python
from dask.distributed import Client
client = await Client(asynchronous=True)
```

In [ ]:

```python
def inc(x: int) -> int:
    return x + 1

future = client.submit(inc, 10)
future
```

In [ ]:

```python
await future
```

Collections

Note that blocking operations like the `.compute()` method aren't ok to use in asynchronous mode. Instead you'll have to use the `Client.compute` method.

In [ ]:

```python
import dask
df = dask.datasets.timeseries()
df
```

In [ ]:

```python
df = df.persist()             # persist is non-blocking, so it's ok
```

In [ ]:

```python
total = df[['x', 'y']].sum()  # lazy computations are also ok
```

In [ ]:

```python
# total.compute()             # but compute is bad, because compute bl
ocks until done
```

In [ ]:

```python
future = client.compute(total)
future
```

In [ ]:

```python
await future
```

Within a script

Running async/await code in Jupyter is a bit atypical. Jupyter already has an event loop running, so it's easy to use async/await syntax directly within it. In a normal Python script this won't be the case. Here is an example script that should run within a normal Python interpreter or as a script.

```
import asyncio
from dask.distributed import Client


def inc(x: int) -> int:
    return x + 1


async def f():
    async with Client(asynchronous=True) as client:
        future = client.submit(inc, 10)
        result = await future
        print(result)


if __name__ == '__main__':
    asyncio.get_event_loop().run_until_complete(f())
```

Dask Arrays

Dask arrays coordinate many Numpy arrays, arranged into chunks within a grid. They support a large subset of the Numpy API.

Start Dask Client for Dashboard

Starting the Dask Client is optional. It will provide a dashboard which is useful to gain insight on the computation.

The link to the dashboard will become visible when you create the client below. We recommend having it open on one side of your screen while using your notebook on the other side. This can take some effort to arrange your windows, but seeing them both at the same is very useful when learning.

In [ ]:

```
from dask.distributed import Client, progress
client = Client(processes=False, threads_per_worker=4,
                n_workers=1, memory_limit='2GB')
client
```

Create Random array

This creates a 10000x10000 array of random numbers, represented as many numpy arrays of size 1000x1000 (or smaller if the array cannot be divided evenly). In this case there are 100 (10x10) numpy arrays of size 1000x1000.

In [ ]:

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

Use NumPy syntax as usual

In [ ]:

```
y = x + x.T
z = y[::2, 5000:].mean(axis=1)
z
```

Call `.compute()` when you want your result as a NumPy array.
If you started `Client()` above then you may want to watch the status page during computation.

In [ ]:

```
z.compute()
```

Persist data in memory

If you have the available RAM for your dataset then you can persist data in memory.

This allows future computations to be much faster.

In [ ]:

```
y = y.persist()
```

In [ ]:

```
%time y[0, 0].compute()
```

In [ ]:

```
%time y.sum().compute()
```

Dask Bags

Dask Bag implements operations like `map`, `filter`, `groupby` and aggregations on collections of Python objects. It does this in parallel and in small memory using Python iterators. It is similar to a parallel version of itertools or a Pythonic version of the PySpark RDD.

Dask Bags are often used to do simple preprocessing on log files, JSON records, or other user defined Python objects.
Full API documentation is available here: http://docs.dask.org/en/latest/bag-api.html

Start Dask Client for Dashboard

Starting the Dask Client is optional. It will provide a dashboard which is useful to gain insight on the computation.

The link to the dashboard will become visible when you create the client below. We recommend having it open on one side of your screen while using your notebook on the other side. This can take some effort to arrange your windows, but seeing them both at the same is very useful when learning.

In [ ]:

```
from dask.distributed import Client, progress
client = Client(n_workers=4, threads_per_worker=1)
client
```

Create Random Data

We create a random set of record data and store it to disk as many JSON files. This will serve as our data for this notebook.

In [ ]:

```
import dask
import json
import os

os.makedirs('data', exist_ok=True)              # Create data/ directo
ry

b = dask.datasets.make_people()                 # Make records of peop
le
b.map(json.dumps).to_textfiles('data/*.json')   # Encode as JSON, writ
e to disk
```

Read JSON data

Now that we have some JSON data in a file lets take a look at it with Dask Bag and Python JSON module.

In [ ]:

```
!head -n 2 data/0.json
```

In [ ]:

```
import dask.bag as db
import json

b = db.read_text('data/*.json').map(json.loads)
b
```

In [ ]:

```
b.take(2)
```

Map, Filter, Aggregate

We can process this data by filtering out only certain records of interest, mapping functions over it to process our data, and aggregating those results to a total value.

In [ ]:

```
b.filter(lambda record: record['age'] > 30).take(2)  # Select only peo
ple over 30
```

In [ ]:

```
b.map(lambda record: record['occupation']).take(2)  # Select the occup
ation field
```

In [ ]:

```
b.count().compute()  # Count total number of records
```

Chain computations

It is common to do many of these steps in one pipeline, only calling `compute` or `take` at the end.

In [ ]:

```
result = (b.filter(lambda record: record['age'] > 30)
           .map(lambda record: record['occupation'])
           .frequencies(sort=True)
           .topk(10, key=1))
result
```

As with all lazy Dask collections, we need to call `compute` to actually evaluate our result. The `take` method used in earlier examples is also like `compute` and will also trigger computation.

In [ ]:

```
result.compute()
```

Transform and Store

Sometimes we want to compute aggregations as above, but sometimes we want to store results to disk for future analyses. For that we can use methods like `to_textfiles` and `json.dumps`, or we can convert to Dask Dataframes and use their storage systems, which we'll see more of in the next section.

In [ ]:

```
(b.filter(lambda record: record['age'] > 30)  # Select records of inte
rest
  .map(json.dumps)                             # Convert Python objects
to text
  .to_textfiles('data/processed.*.json'))      # Write to local disk
```

Convert to Dask Dataframes

Dask Bags are good for reading in initial data, doing a bit of pre-processing, and then handing off to some other more efficient form like Dask Dataframes. Dask Dataframes use Pandas internally, and so can be much faster on numeric data and also have more complex algorithms.

However, Dask Dataframes also expect data that is organized as flat columns. It does not support nested JSON data very well (Bag is better for this).

Here we make a function to flatten down our nested data structure, map that across our records, and then convert that to a Dask Dataframe.

In [ ]:

```
b.take(1)
```

In [ ]:

```
def flatten(record):
    return {
        'age': record['age'],
        'occupation': record['occupation'],
        'telephone': record['telephone'],
        'credit-card-number': record['credit-card']['number'],
        'credit-card-expiration': record['credit-card']['expiration-da
te'],
        'name': ' '.join(record['name']),
        'street-address': record['address']['address'],
        'city': record['address']['city']
    }
```

```
b.map(flatten).take(1)
```

In [ ]:

```
df = b.map(flatten).to_dataframe()
df.head()
```

We can now perform the same computation as before, but now using Pandas and Dask dataframe.

In [ ]:

```
df[df.age > 30].occupation.value_counts().nlargest(10).compute()
```

Simplest possible example

We make some simple functions, inc and add, that sleep for a while to simulate work. We time running these functions normally.

In the next section we'll parallelize this code

In [ ]:

```
from time import sleep

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

def add(x, y):
    sleep(1)
    return x + y
```

In [ ]:

```
%%time
# This takes three seconds to run because we call each function sequen
tially, one after the other
x = inc(1)
y = inc(2)
z = add(x, y)
```

Parallelize with dask.delayed decorator

Those two increment calls *could* be called in parallel.

In this section we call `inc` and `add`, wrapped with `dask.delayed`. This changes those functions so that they don't run immediately, but instead put those functions and arguments into a task graph. Now when we run our code this runs immediately, but all it does it create a graph. We then separately compute the result by calling the `.compute()` method.

In [ ]:

```
import dask
```

In [ ]:

```
%%time
# This runs immediately, all it does is build a graph
```

```
x = dask.delayed(inc)(1)
y = dask.delayed(inc)(2)
z = dask.delayed(add)(x, y)
```

In [ ]:

```
%%time
# This actually runs our computation using a local thread pool
z.compute()
```

What just happened?

The `z` object is a lazy `dask.Delayed` object. This object holds everything we need to compute the final result. We can compute the result with `.compute()` as above or we can visualize the result with `.visualize()`.

In [ ]:

```
z
```

In [ ]:

```
z.visualize(rankdir="LR")
```

Some questions to consider:

- Why did we go from 3s to 2s? Why weren't we able to parallelize down to 1s?
- What would have happened if the inc and add functions didn't include the `sleep(1)`? Would Dask still be able to speed up this code?
- What if we have multiple outputs or also want to get access to x or y?

Exercise: Parallelize a for loop

For loops are one of the most common things that we want to parallelize. Use dask.delayed on `inc` and `sum` to parallelize the computation below:

In [ ]:

```
data = [1, 2, 3, 4, 5, 6, 7, 8]
```

In [ ]:

```
%%time
# Sequential code

results = []
for x in data:
    y = inc(x)
    results.append(y)

total = sum(results)
```

In [ ]:

```
total
```

In [ ]:

```
%%time
# Parallel code

results = []
for x in data:
    # TODO
```

In [ ]:

```
total
```

Parallelizing for-loop code with control flow

Often we want to delay only *some* functions, running a few of them immediately. This is especially helpful when those functions are fast and help us to determine what other slower functions we should call. This decision, to delay or not to delay, is usually where we need to be thoughtful when using dask.delayed.

In the example below we iterate through a list of inputs. If that input is even then we want to call `inc`. If the input is odd then we want to call `double`. This `iseven` decision to call `inc` or `double` has to be made immediately (not lazily) in order for our graph-building Python code to proceed.

In [ ]:

```
def double(x):
    sleep(1)
    return 2 * x

def iseven(x):
    return x % 2 == 0

data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

In [ ]:

```
%%time
# Sequential code

results = []
for x in data:
    if iseven(x):
        y = double(x)
    else:
        y = inc(x)
    results.append(y)

total = sum(results)
total
```

In [ ]:

```
%%time
# Parallel code
# TODO: parallelize the sequential code above using dask.delayed
# You will need to delay some functions, but not all
```

In [ ]:

```
total.visualize()
```

In [ ]:

```
%time total.compute()
```

In [ ]:

```
%load solutions/01-delayed-inc-double.py
```

Dask for Machine Learning

This is a high-level overview demonstrating some the components of Dask-ML. Visit the main Dask-ML documentation, see the dask tutorial notebook 08, or explore some of the other machine-learning examples.

In [ ]:

```
from dask.distributed import Client, progress
client = Client(processes=False, threads_per_worker=4,
                n_workers=1, memory_limit='2GB')
client
```

Distributed Training

Scikit-learn uses joblib for single-machine parallelism. This lets you train most estimators (anything that accepts an `n_jobs` parameter) using all the cores of your laptop or workstation.

Alternatively, Scikit-Learn can use Dask for parallelism. This lets you train those estimators using all the cores of your *cluster* without significantly changing your code.

This is most useful for training large models on medium-sized datasets. You may have a large model when searching over many hyper-parameters, or when using an ensemble method with many individual estimators. For too small datasets, training times will typically be small enough that cluster-wide parallelism isn't helpful. For too large datasets (larger than a single machine's memory), the scikit-learn estimators may not be able to cope (see below).

Create Scikit-Learn Estimator

In [ ]:

```
from sklearn.datasets import make_classification
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV
import pandas as pd
```

We'll use scikit-learn to create a pair of small random arrays, one for the features `X`, and one for the target `y`.

In [ ]:

```
X, y = make_classification(n_samples=1000, random_state=0)
X[:5]
```

We'll fit a Support Vector Classifier, using grid search to find the best value of the hyperparameter.

In [ ]:

```
param_grid = {"C": [0.001, 0.01, 0.1, 0.5, 1.0, 2.0, 5.0, 10.0],
              "kernel": ['rbf', 'poly', 'sigmoid'],
              "shrinking": [True, False]}

grid_search = GridSearchCV(SVC(gamma='auto', random_state=0, probabili
ty=True),
                           param_grid=param_grid,
                           return_train_score=False,
                           cv=3,
                           n_jobs=-1)
```

To fit that normally, we would call

```
    grid_search.fit(X, y)
```

To fit it using the cluster, we just need to use a context manager provided by joblib.

In [ ]:

```
import joblib

with joblib.parallel_backend('dask'):
    grid_search.fit(X, y)
```

We fit 48 different models, one for each hyper-parameter combination in `param_grid`, distributed across the cluster. At this point, we have a regular scikit-learn model, which can be used for prediction, scoring, etc.

In [ ]:

```
pd.DataFrame(grid_search.cv_results_).head()
```

In [ ]:

```
grid_search.predict(X)[:5]
```

In [ ]:

```
grid_search.score(X, y)
```

For more on training scikit-learn models with distributed joblib, see the dask-ml documentation.

Training on Large Datasets

Most estimators in scikit-learn are designed to work on in-memory arrays. Training with larger datasets may require different algorithms.
All of the algorithms implemented in Dask-ML work well on larger than memory datasets, which you might store in a dask array or dataframe.

In [ ]:

```
%matplotlib inline
```

In [ ]:

```
import dask_ml.datasets
import dask_ml.cluster
import matplotlib.pyplot as plt
```

In this example, we'll use `dask_ml.datasets.make_blobs` to generate some random *dask* arrays.

In [ ]:

```
X, y = dask_ml.datasets.make_blobs(n_samples=10000000,
                                   chunks=1000000,
                                   random_state=0,
                                   centers=3)
X = X.persist()
X
```

We'll use the k-means implemented in Dask-ML to cluster the points. It uses the `k-means||` (read: "k-means parallel") initialization algorithm, which scales better than `k-means++`. All of the computation, both during and after initialization, can be done in parallel.

In [ ]:

```
km = dask_ml.cluster.KMeans(n_clusters=3, init_max_iter=2, oversamplin
g_factor=10)
km.fit(X)
```

We'll plot a sample of points, colored by the cluster each falls into.

In [ ]:

```
fig, ax = plt.subplots()
ax.scatter(X[::10000, 0], X[::10000, 1], marker='.', c=km.labels_[::10
000],
           cmap='viridis', alpha=0.25);
```

Parallelize image filters with dask

This notebook will show how to parallize CPU-intensive workload using dask array. A simple uniform filter (equivalent to a mean filter) from `scipy.ndimage` is used for illustration purposes.

In [1]:

```
%pylab inline
from scipy.ndimage import uniform_filter
import dask.array as da

def mean(img):
    "ndimage.uniform_filter with `size=51`"
    return uniform_filter(img, size=51)
```

```
Populating the interactive namespace from numpy and matplotlib
```

Get the image

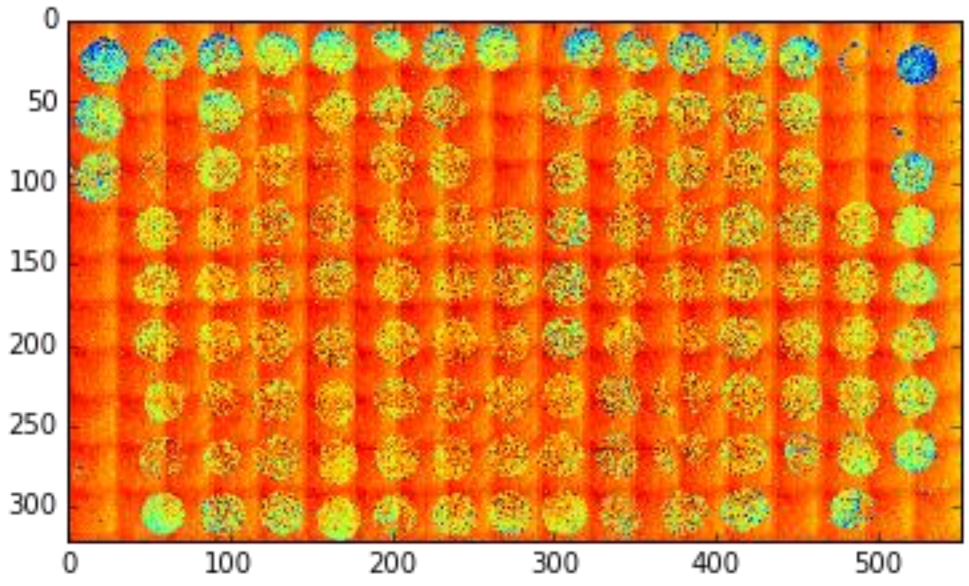
In [2]:

```
!if [ ! -e stitched--U00--V00--C00--Z00.png ]; then wget -q https://gi
thub.com/arve0/master/raw/master/stitched--U00--V00--C00--Z00.png; fi
img = imread('stitched--U00--V00--C00--Z00.png')
img = (img*255).astype(np.uint8) # image read as float32, image is 8 b
it grayscale
imshow(img[::16, ::16])
mp = str(img.shape[0] * img.shape[1] * 1e-6 // 1)
```

```
'%s Mega pixels, shape %s, dtype %s' % (mp, img.shape, img.dtype)
```

Out[2]:

```
'45.0 Mega pixels, shape (5122, 8810), dtype uint8'
```

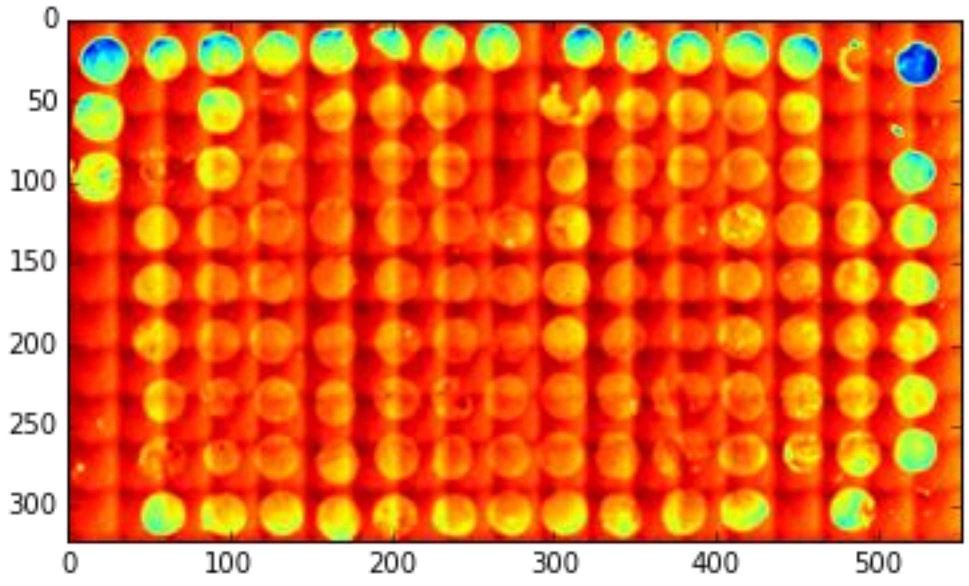


Initial speed

Lets try the filter directly on the image.

In [3]:

```
# filter directly
%time mean_nd = mean(img)
imshow(mean_nd[::16, ::16]);
CPU times: user 2.52 s, sys: 64.2 ms, total: 2.59 s
Wall time: 2.7 s
```



With dask

First, we'll create the dask array with one chunk only (`chunks=img.shape`).

In [4]:

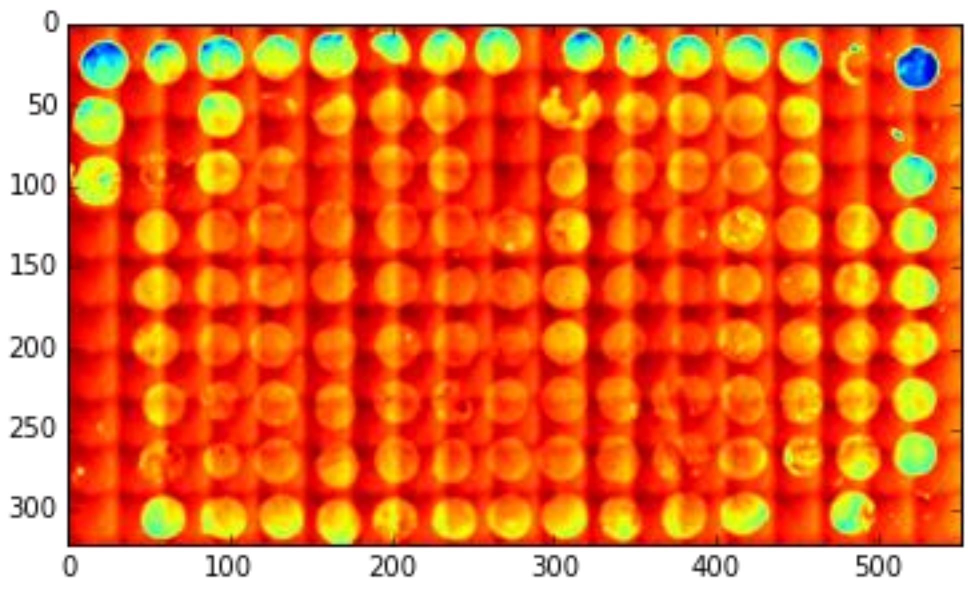
```
img_da = da.from_array(img, chunks=img.shape)
```

`depth` defines the overlap. We have one chunk only, so overlap is not necessary. `compute` must be called to start the computation.

In [5]:

```
%time mean_da = img_da.map_overlap(mean, depth=0).compute()
imshow(mean_da[::16, ::16]);
```

```
CPU times: user 2.43 s, sys: 59 ms, total: 2.49 s
Wall time: 2.53 s
```



As we can see, the performance is the same as applying the filter directly.

Now, lets chop up the image in chunks so that we can leverage all the cores in our computer.

In [6]:

```
from multiprocessing import cpu_count
cpu_count()
```

Out[6]:

```
4
```

We have four cores, so lets split the array in four chunks.

In [7]:

```
img.shape, mean_da.shape, mean_nd.shape
```

Out[7]:

```
((5122, 8810), (5122, 8810), (5122, 8810))
```

Pixels in both axes are even, so we can split the array in equally sized chunks. If we had odd shapes, chunks would not be the same size (given four cpu cores). E.g. 101x101 image => 50x50 and 51x51 chunks.

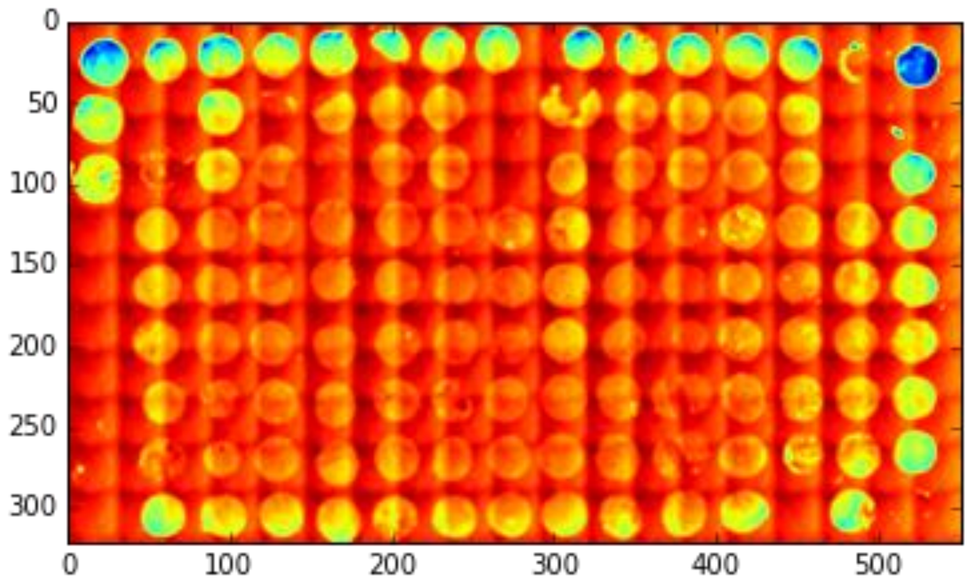
In [8]:

```
chunk_size = [x//2 for x in img.shape]
img_da = da.rechunk(img_da, chunks=chunk_size)
```

Now, lets see if the filtering is faster.

In [9]:

```
%time mean_da = img_da.map_overlap(mean, depth=0).compute()
imshow(mean_da[::16, ::16]);
CPU times: user 3.09 s, sys: 165 ms, total: 3.26 s
Wall time: 1.23 s
```



It is :-)

If one opens the process manager, one will see that the python process is eating more then 100% CPU.

As we are looking at neighbor pixels to compute the mean intensity for the center pixel, you might wonder what happens in the seams between chunks? Lets examine that.

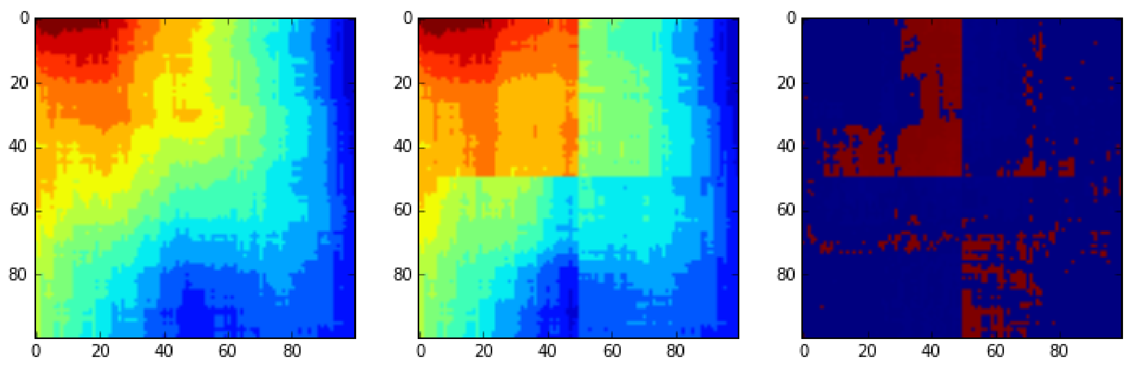
In [10]:

```
size = 50
mask = np.index_exp[chunk_size[0]-size:chunk_size[0]+size, chunk_size[
1]-size:chunk_size[1]+size]

figure(figsize=(12,4))
subplot(131)
imshow(mean_nd[mask]) # filtered directly
subplot(132)
imshow(mean_da[mask]) # filtered in chunks with dask
subplot(133)
imshow(mean_nd[mask] - mean_da[mask]); # difference
```

Out[10]:

```
<matplotlib.image.AxesImage at 0x10d86cc18>
```



To overcome this edge effect in the seams, we need to define a higher `depth` so that dask does the computation with an overlap. We need an overlap of 25 pixels (half the size of the neighborhood in `mean`).

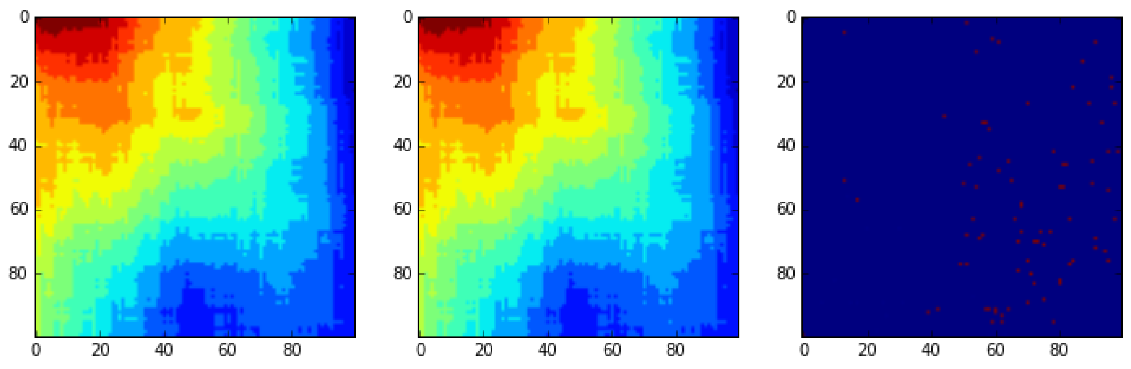
In [11]:

```
%time mean_da = img_da.map_overlap(mean, depth=25).compute()

figure(figsize=(12,4))
subplot(131)
imshow(mean_nd[mask]) # filtered directly
subplot(132)
imshow(mean_da[mask]) # filtered in chunks with dask
subplot(133)
imshow(mean_nd[mask] - mean_da[mask]); # difference
CPU times: user 3.47 s, sys: 206 ms, total: 3.67 s
Wall time: 1.23 s
```

Out[11]:

```
<matplotlib.image.AxesImage at 0x111f62b00>
```



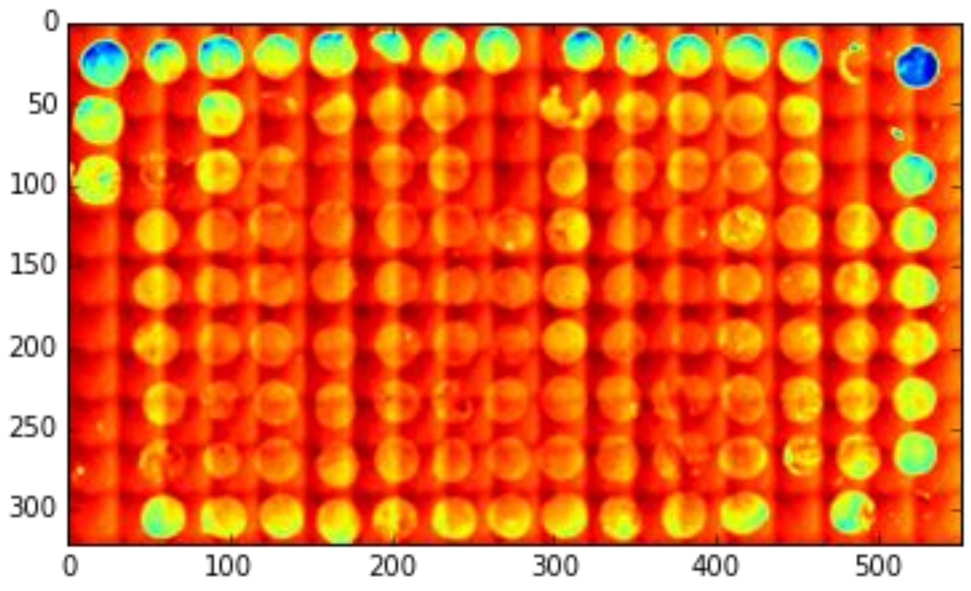
Edge effect is gone, nice! The dots in the difference is due to uniform_filter's limited precision. From the manual:

The multi-dimensional filter is implemented as a sequence of one-dimensional uniform filters. The intermediate arrays are stored in the same data type as the output. Therefore, for output types with a limited precision, the results may be imprecise because intermediate results may be stored with insufficient precision.

Lets see if we can improve the performance. As we do not get 4x speedup, there might be that computation is not only CPU-bound. Chunksize of 1000 is a good place to start.

In [12]:

```
img_da = da.rechunk(img_da, 1000)
%time mean_da = img_da.map_overlap(mean, depth=25).compute()
imshow(mean_da[::16, ::16]);
CPU times: user 2.39 s, sys: 183 ms, total: 2.57 s
Wall time: 1.24 s
```



As you see, adjusting the chunk size did not affect the performance significant, though its a good idea to identify your bottleneck and adjust the chunk size accordingly.

That's all! By chopping up the computation we utilized all cpu cores and got a speedup at best:

In [14]:

```
'%0.1fx' % (2.7/1.24)
```

Out[14]:

```
'2.2x'
```

Custom Workloads with Futures

Start Dask Client

Unlike for arrays and dataframes, you need the Dask client to use the Futures interface. Additionally the client provides a dashboard which is useful to gain insight on the computation.

The link to the dashboard will become visible when you create the client below. We recommend having it open on one side of your screen while using your notebook on the other side. This can take some effort to arrange your windows, but seeing them both at the same is very useful when learning.

In [ ]:

```
from dask.distributed import Client, progress
client = Client(threads_per_worker=4, n_workers=1)
client
```

Create simple functions

These functions do simple operations like add two numbers together, but they sleep for a random amount of time to simulate real work.

In [ ]:

```
import time
import random

def inc(x):
    time.sleep(random.random())
    return x + 1

def double(x):
    time.sleep(random.random())
    return 2 * x

def add(x, y):
    time.sleep(random.random())
    return x + y
```

We can run them locally

In [ ]:

```
inc(1)
```

Or we can submit them to run remotely with Dask. This immediately returns a future that points to the ongoing computation, and eventually to the stored result.

In [ ]:

```
future = client.submit(inc, 1)   # returns immediately with pending fut
ure
future
```

If you wait a second, and then check on the future again, you'll see that it has finished.

In [ ]:

```
future  # scheduler and client talk constantly
```

You can block on the computation and gather the result with the `.result()` method.

In [ ]:

```
future.result()
```

Chain dependencies

You can submit tasks on other futures. This will create a dependency between the inputs and outputs. Dask will track the execution of all tasks, ensuring that downstream tasks are run at the proper time and place and with the proper data.

In [ ]:

```
x = client.submit(inc, 1)
y = client.submit(double, 2)
z = client.submit(add, x, y)
z
```

In [ ]:

```
z.result()
```

Note that we never blocked on `x` or `y` nor did we ever have to move their data back to our notebook.

Submit many tasks

So we've learned how to run Python functions remotely. This becomes useful when we add two things:

1. We can submit thousands of tasks per second
2. Tasks can depend on each other by consuming futures as inputs

We submit many tasks that depend on each other in a normal Python for loop

In [ ]:

```
zs = []
```

In [ ]:

```
%%time

for i in range(256):
    x = client.submit(inc, i)     # x = inc(i)
    y = client.submit(double, x)  # y = inc(x)
    z = client.submit(add, x, y)  # z = inc(y)
    zs.append(z)
```

In [ ]:

```
total = client.submit(sum, zs)
```

To make this go faster, add an additional workers with more cores

(although we're still only working on our local machine, this is more practical when using an actual cluster)

In [ ]:

```
client.cluster.scale(10)  # ask for ten 4-thread workers
```

Custom computation: Tree summation

As an example of a non-trivial algorithm, consider the classic tree reduction. We accomplish this with a nested for loop and a bit of normal Python logic.

```
    finish           total             single output
        ^          /        \
        |        c1          c2        neighbors merge
        |       /  \        /  \
        |     b1    b2    b3    b4     neighbors merge
        ^    / \   / \   / \   / \
    start   a1  a2 a3 a4 a5 a6 a7 a8    many inputs
```

In [ ]:

```
L = zs
while len(L) > 1:
    new_L = []
    for i in range(0, len(L), 2):
        future = client.submit(add, L[i], L[i + 1])  # add neighbors
        new_L.append(future)
    L = new_L                                   # swap old list for ne
w
```

If you're watching the [dashboard's status page]() then you may want to note two things:

1. The red bars are for inter-worker communication. They happen as different workers need to combine their intermediate values
2. There is lots of parallelism at the beginning but less towards the end as we reach the top of the tree where there is less work to do.

Alternatively you may want to navigate to the dashboard's graph page and then run the cell above again. You will be able to see the task graph evolve during the computation.

Building a computation dynamically

In the examples above we explicitly specify the task graph ahead of time. We know for example that the first two futures in the list `L` will be added together.

Sometimes this isn't always best though, sometimes you want to dynamically define a computation as it is happening. For example we might want to sum up these values based on whichever futures show up first, rather than the order in which they were placed in the list to start with.
For this, we can use operations like as_completed.

We recommend watching the dashboard's graph page when running this computation. You should see the graph construct itself during execution.

In [ ]:

```
del future, L, new_L, total  # clear out some old work
```

In [ ]:

```
from dask.distributed import as_completed

zs = client.map(inc, zs)
seq = as_completed(zs)

while seq.count() > 1:  # at least two futures left
    a = next(seq)
    b = next(seq)
    new = client.submit(add, a, b)  # add them together
    seq.add(new)                    # add new future back into loop
```