Dask for High Energy Physics

Dask: Flexible parallel execution library for analytic computing

Martin Durant, Anaconda Inc.

Introducing Dask

- easy
- efficient
- scalable
- familiar
- low-latency

The HPC gulf

Local machine	HPC cluster
few GB of RAM, ~TB storage	TB RAM, PB storage
<10 cores	>10k cores
python	compiled languages
simple programming	dedicated parallelism framework
interactive/exploratory work	scheduling system

Dask: How to scale up with a minimum of hassle

Run dask on your laptop, or on a large cluster: just specify the scheduler address.

In [1]:

```
import dask.distributed
client = dask.distributed.Client('dask-scheduler:8786')
client
```

Out[1]:

Client

- **Scheduler:** tcp://dask-scheduler:8786
- **Dashboard:** http://dask-scheduler:8787/status (http://dask-scheduler:8787/status)

Cluster

- **Workers:** 8
- **Cores:** 16
- **Memory:** 54.19 GB

```
@dask.delayed
def f(x, y):
    do_thing_with_inputs
    return output
```

In [2]:

```
%%writefile work.py
import dask
import time
import random

@dask.delayed
def load(fn):
    time.sleep(random.random())
    return fn

@dask.delayed
def load_from_sql(fn):
    time.sleep(random.random() * 3)
    return fn

@dask.delayed
def normalize(in1, in2):
    time.sleep(random.random())
    return in1

@dask.delayed
def roll(in1, in2, in3):
    time.sleep(random.random())
    return in1

@dask.delayed
def compare(in1, in2):
    time.sleep(1)
    return in1

@dask.delayed
def reduction(inlist):
    return True
```

```
Overwriting work.py
```

In [4]:

```
# Normal make-work functions annotated with "delayed"
from work import load, load_from_sql, normalize, roll, compare, reduction, random
filenames = ['mydata-%d.dat' % i for i in range(30)]
data = [load(fn) for fn in filenames]

reference = load_from_sql('sql://mytable')
processed = [normalize(d, reference) for d in data]

rolled = []
for i in range(len(processed) - 2):
    r = roll(processed[i], processed[i + 1], processed[i + 2])
    rolled.append(r)

compared = []
for i in range(20):
    a = random.choice(rolled)
    b = random.choice(rolled)
    c = compare(a, b)
    compared.append(c)

out = reduction(compared)
out
```

Out[4]:

```
Delayed('reduction-59fbe054-4c7b-4c7a-96b9-950c36b9c8ce')
```

In [6]:

```
out.visualize()
```

Out[6]:

In [7]:

```
f = client.compute(out)
```

In [9]:

```
str(f)
```

Out[9]:

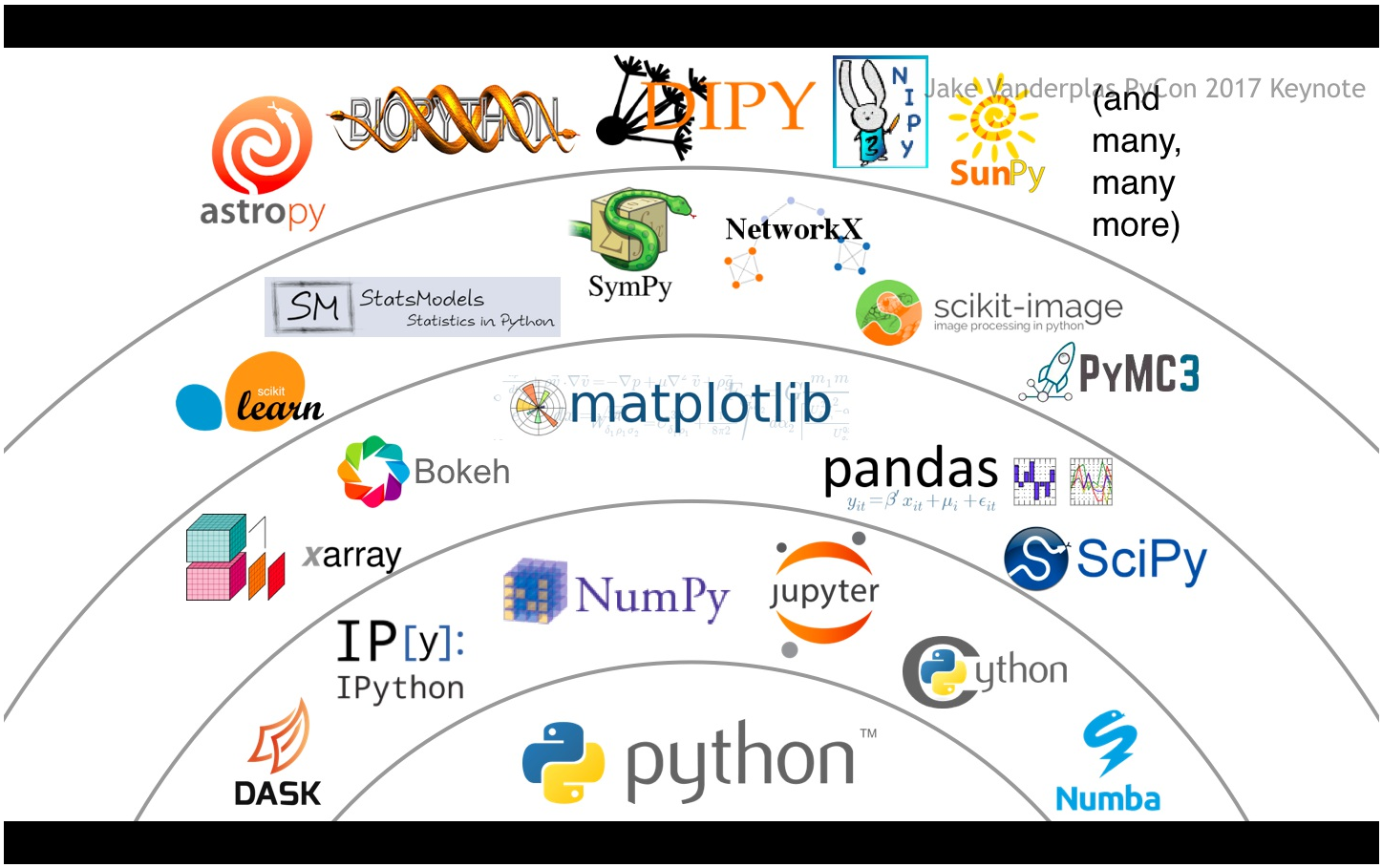
```
'<Future: status: finished, type: bool, key: reduction-59fbe054-4c7b-4c7a-96b9-950c36b9c8ce>'
```

In [10]:

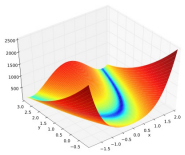
```
del f
```

Why Python?

- fast prototyping, minimum keystrokes
- this is what new students use
- interactive, fast feedback
- very complete numerical ecosystem
- easy to accelerate critical code
- easy linkage with legacy C/C++/fortran



Dynamic programming: minimization



In [11]:

```
import time
def rosenbrock(point):
    """Compute the rosenbrock function and return the point and result"""
    time.sleep(0.1)  # fake delay
    score = (1 - point[0])**2 + 2 * (point[1] - point[0]**2)**2
    return point, score
```

In [12]:

```
scale = 5                  # Intial random perturbation scale
best_point = (0, 0)        # Initial guess
best_score = float('inf')  # Best score so far
```

In [13]:

```
from dask.distributed import as_completed
from random import uniform
# initial 10 random points
futures = [client.submit(rosenbrock, (uniform(-scale, scale), uniform(-scale, scale)))
           for _ in range(10)]
iterator = as_completed(futures)

for res in iterator:
    # take a completed point, is it an improvement?
    point, score = res.result()
    if score < best_score:
        best_score, best_point = score, point

    x, y = best_point

    # add new point, dynamically, to work on the cluster
    new_point = client.submit(rosenbrock, (x + uniform(-scale, scale),
                                           y + uniform(-scale, scale)))
    iterator.add(new_point)  # Start tracking new task as well

    # Narrow search and consider stopping
    scale *= 0.99
    if scale < 0.001:
        break
point
```

Out[13]:

```
(1.000363479684451, 0.999097972615762)
```

In [14]:

```
del futures[:], new_point, iterator, res
```

High-level APIs

Dataframes: distributed pandas

In [15]:

```
import dask.dataframe as dd
d = dd.read_csv('gcs://anaconda-public-data/airline/*.csv',
                dtype={'CRSElapsedTime': float, 'CancellationCode': object,
                       'TailNum': object, 'Distance': float}, encoding='latin1')
d.groupby(d.DayOfWeek).ArrDelay.mean().compute()
```

Out[15]:

```
DayOfWeek
1    6.669515
2    5.960421
3    7.091502
4    8.945047
5    9.606953
6    4.187419
7    6.525040
Name: ArrDelay, dtype: float64
```

Arrays: distributed numpy

In [16]:

```
import dask.array as da
d = da.random.random((2000, 2000, 2000), chunks=(100, 1000, 1000))  # 60GB
d.argmax(axis=2).compute()
```

Out[16]:

```
array([[1701, 1523, 1837, ...,  385,  225,   17],
       [ 114,  886,  636, ...,  741, 1778, 1464],
       [ 942,  802, 1902, ...,  232,  995,  568],
       ...,
       [1988, 1254, 1869, ...,  900, 1417, 1490],
       [1255,  432,   56, ..., 1920, 1383,  796],
       [1824, 1233, 1339, ..., 1262, 1594,  746]])
```

Sequences: distributed functional programming

In [17]:

```
import dask.bag as db
import json
lines = db.read_text('s3://anaconda-public-datasets/enron-email/edrm-enron-v2_'
                     'c*/merged.txt', storage_options={'anon': True})
print(lines.take(3))
b = (lines.map(str.split).
     flatten().
     filter(lambda x: len(x) > 5).
     frequencies().
     topk(10, lambda x: x[1]))
b.compute()
```

```
('Date: Wed, 7 Mar 2001 10:10:00 -0800 (PST)\n', 'From: Larry Campbell\n', 'To: Team Flagstaff-Sta3, Team Flagstaff, Team Gallup-Sta4, Team Gallup\n')
```

Out[17]:

```
[('ENERGY', 63313),
 ('Technologies,', 48538),
 ('**********', 48516),
 ('09/11/2000', 42332),
 ('Subject:', 33449),
 ('Pending', 31214),
 ('provide', 31115),
 ('TRANSPORTATION,', 30959),
 ('please', 30840),
 ('COMPANY', 29831)]
```

In [18]:

```
lines.to_delayed() # , b.dask
```

Out[18]:

```
[Delayed(('bag-from-delayed-8e47584b649c338ca8464d31bd84edd8', 0)),
 Delayed(('bag-from-delayed-8e47584b649c338ca8464d31bd84edd8', 1)),
 Delayed(('bag-from-delayed-8e47584b649c338ca8464d31bd84edd8', 2)),
 Delayed(('bag-from-delayed-8e47584b649c338ca8464d31bd84edd8', 3)),
 Delayed(('bag-from-delayed-8e47584b649c338ca8464d31bd84edd8', 4)),
 Delayed(('bag-from-delayed-8e47584b649c338ca8464d31bd84edd8', 5)),
 Delayed(('bag-from-delayed-8e47584b649c338ca8464d31bd84edd8', 6))]
```

Other

In [ ]:

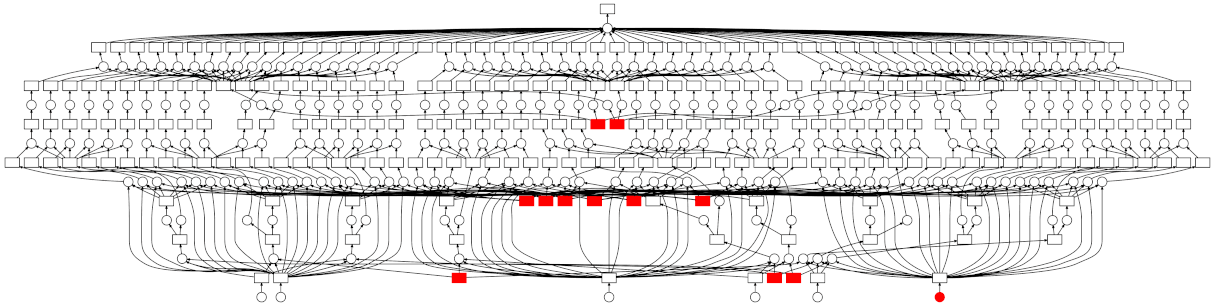
```
# xarray: labelled ND-arrays
>>> xr.open_dataset('/data_store/mydata*.nc')
<xarray.Dataset>
Dimensions:   (location: 3, time: 731)
Coordinates:
  * time      (time) datetime64[ns] 2000-01-01 2000-01-02 2000-01-03 ...
  * location  (location) <U2 'IA' 'IN' 'IL'
Data variables:
    tmin      (time, location) float64 -8.037 -1.788 -3.932 -9.341 -6.558 ...
    tmax      (time, location) float64 12.98 3.31 6.779 0.4479 6.373 4.843 ...

# machine learning drop-ins
from dask_ml.linear_model import LogisticRegression, PartialSGDRegressor
from dask_ml.model_selection import GridSearchCV

# interaction with XGBoost, TensorFlow ...
# streams
# sparse...
```

Summary: why use Dask?

- all python
- efficient, low-latency
- flexible algorithms, real-time scheduling
- familiar functional, array, dataframe APIs



In [ ]:

```
client.restart()
```

gNlZz9FKrznpEl/JwX6C6nQX0iF/kIK9BdSob+QCv2FFOgvxeEobSNYtmxZPPfcc/H000/HlClTYubMmbF+/fqoqKilrl27Rrdu3WLbbbeNdu3axbbbbhtbbbVVrF69Oj799799NNYsWJFLF++POBmRNLliyJilj27dvH/wwHwMHDoyBAwfG3/3d30WrVq1y/ixJhTxTJwJMmck3c
.btn-link:focus { color: #777777; text-decoration: none; } .btn-lg, .btn-gro