Dask

Parallel and Distributed Computing in Python

April 17, 2018

Seminar Distributed Data Mining
Leiden University

Table of Contents

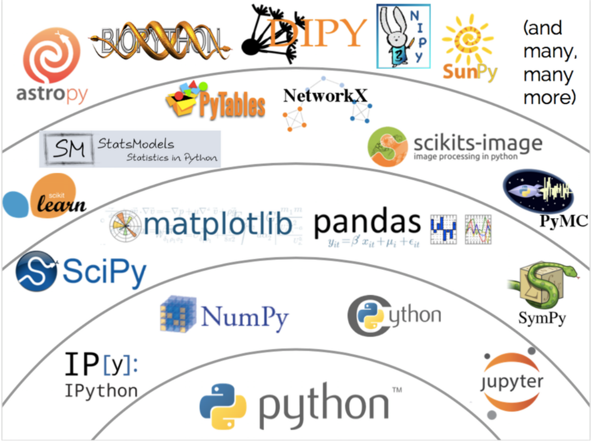
Introduction

What is Dask?

- Relatively new, open source, pure python library (around 2015).
- A project of Continuum Analytics.
- A parallel and distributed computing framework.
- Leverages scientific python ecosystem (numpy, pandas, scikit-learn etc).
- Utilizes block algorithms and task scheduling.

Why Dask?

- Parallelize scientific python ecosystem.

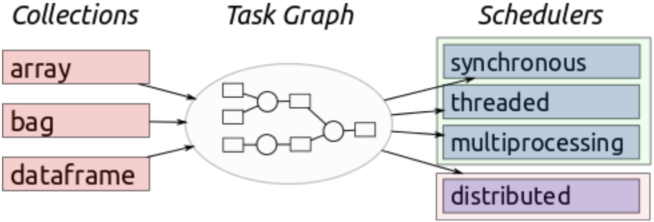


http://danielrothenberg.com/gcpy/python.html

Why Dask? (Cont.)

- Familiar API.
- Scales up to a cluster (Distributed).
- Scales down to a single computer (Parallel).
- Work on data larger than memory.

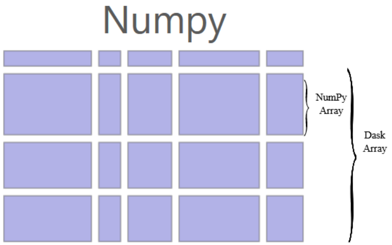
Dask Overview



- Collections create task graphs.
- Scheduler executes graphs on parallel hardware (single machine or cluster).

https://dask.pydata.org/en/latest/docs.html

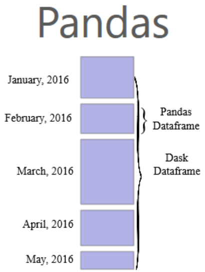
Collections




```python
# NumPy code
import numpy as np
x = np.random.random((1000, 1000))
mean = np.mean(x, axis=1)

# Dask.array code
import dask.array as da
x = da.random.random((100000, 100000), chunks = (1000, 1000))
mean = da.mean(x, axis=1)
mean.compute()
```

https://dask.pydata.org/en/latest/

Dask.dataframe




```
# pandas code
import pandas as pd
data = pd.read_csv('sample_data.csv')
data.groupby(data['reviewerID'])['overall'].mean()

# Dask.dataframe code
import dask.dataframe as dd
data = dd.read_csv('sample_data.csv')
mean_per_reviewerID = data.groupby(data['reviewerID'])['overall'].mean()
mean_per_reviewerID.compute()
```

https://dask.pydata.org/en/latest/

Dask.delayed

- Not every problem is a numpy array, or pandas dataframe.
- What about generic code?
- Dask.delayed mimics for loops, and wraps custom code.

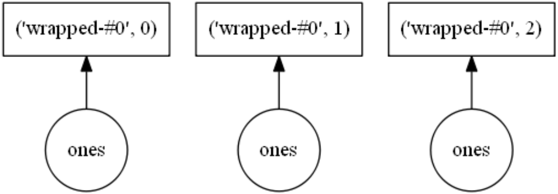
```python
def inc(x):
    return x + 1

def double(x):
    return x + 2

def add(x, y):
    return x + y

data = [1, 2, 3, 4, 5]

output = []
for x in data:
    a = inc(x)
    b = double(x)
    c = add(a, b)
    output.append(c)

total = sum(output)
```

```python
import dask.delayed
# Dask.delayed
output = []
for x in data:
    a = dask.delayed(inc)(x)
    b = dask.delayed(double)(x)
    c = dask.delayed(add)(a, b)
    output.append(c)

total = dask.delayed(sum)(output)

# No computation has happened yet.
# The object "total" contains a task graph of the computation.

total.compute()
```

http://dask.pydata.org/en/latest/delayed.html

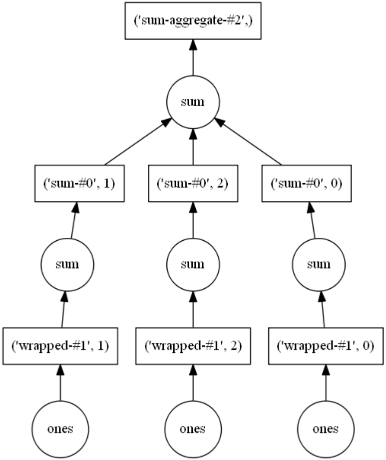
Task Graphs

1D-Array



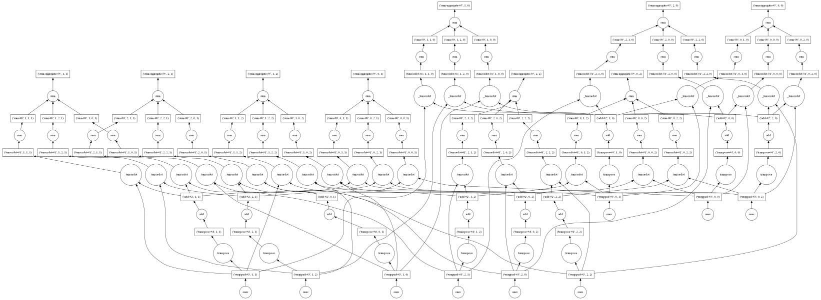

```
In [76]: np.ones((15,))
Out[76]:
array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1.,  1.])

In [77]: x = da.ones((15,), chunks = (5,))
```

1D-Array Sum




```
In [82]: x = da.ones((15,), chunks = (5,))

In [83]: x.sum()
```

ND-Array Matrix Multiplication

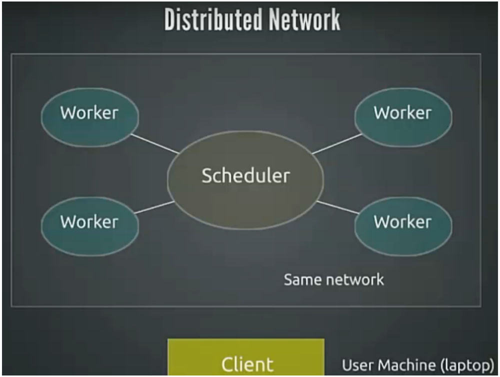


```
In [85]: x = da.ones((15, 15), chunks=(5, 5))

In [86]: x.dot(x.T + 1)
```

Workers and Schedulers

Distributed Network

- The client uses Dask collections to submit graphs to the scheduler.
- The scheduler distributes jobs to the workers.
- Workers communicate peer-to-peer.



http://matthewrocklin.com/slides/pycon-de-2017.html#/8/9

Types of Schedulers

- Dask has two types of schedulers:
  1. Single machine schedulers. Scales only to a single machine.
  2. Distributed scheduler. Scales to a single machine or cluster.
- Single machine schedulers.
  - Synchronous (single-threaded) scheduler. Provides no parallelism, useful for debugging.
  - Threaded scheduler. Provides parallelism for non-Python computations, such as numpy, pandas or any other C/C++ based code.
  - Multiprocessing scheduler. Provides parallelism for generic Python code (strings, lists, dicts etc). However, inter-task data transfer is expensive.
- Distributed Scheduler.
  - Although it's called "Distributed", can be set up either in a local machine or a cluster.
  - Recommended scheduler by Dask developers.

Workers and Scheduler

- Workers provide two functionalities
  - Compute tasks as directed by the scheduler.
  - Store results locally and pass them to other workers or clients.
- The scheduler
  - Dynamically assigns tasks to new workers, as they become available.
  - Informs a worker about which peer-workers have the necessary results to compute the assigned task.
  - Deletes previously computed results from memory.

Communication Between Workers and Scheduler

- Two workers named `Bob` and `Alice`.
- `Scheduler` -> `Alice`: Compute "x <- add(1, 2)"!
  `Alice` -> `Scheduler`: I've computed x and am holding on to it!
  `Scheduler` -> `Bob`: Compute "y <- add(x, 10)"!
  You will need x. Alice has x.
  `Bob` -> `Alice`: Please send me x.
  `Alice` -> `Bob`: Sure. x is 3!
  `Bob` -> `Scheduler`: I've computed y and am holding on to it!
- Task graph execution (gif):
  `https://www.dropbox.com/s/ksjoohokgh8hczl/grid_search_schedule.gif?dl=0`

`http://distributed.readthedocs.io/en/latest/worker.html`

Task Scheduling

Journey of a Task (1)

- We follow a single task through user interface, scheduler, worker nodes, and back.
- Task: Addition of two variables `x, y` already on the cluster. Pulls the result back to the local process.

```
client = Client()
z = client.submit(add, x, y)  # we follow z

print(z.result())
```

Journey of a Task (2)

- Step 1: Client
  - `client.submit()` sends the following message to the scheduler

```
{'op': 'update-graph',
 'tasks': {'z': (add, x, y)},
 'keys': ['z']}
```

- Step 2: Arrival in the scheduler
  - The scheduler creates a graph that shows how to compute `z`.
- Step 3: Select a worker
  1. We consider workers that have either `x` or `y` in local memory.
  2. Select the worker that would have to gather the least number of bytes to get both `x` and `y` locally.
  3. Break ties by selecting the least busy worker to gather `x` or `y`.

Journey of a Task (3)

- Step 4: Transmit to the worker
  - Task `z` is placed into a `worker_queue`, in case the worker has not finished a previous task.
  - Information important for the worker is packed in the following message

```
{'op': 'compute',
 'function': execute_task,
 'args': ((add, 'x', 'y'),),
 'who_has': {'x': {(worker_host, port)},
             'y': {(worker_host, port), (worker_host, port)}},
 'key': 'z'}
```

- Step 5: Execute on the worker
  - Worker unpacks the message.
  - Collects `x` or `y` from other workers (`who_has` key in dictionary).
  - Computes `add(x, y)`.
  - Holds on to the result.
  - Communicates to the scheduler the number of bytes of the result.

Journey of Task (4)

- Step 6: Scheduler aftermath
    - Scheduler sends a new task, if available, to the worker.
    - Scheduler informs workers to delete `x`, `y` if no longer needed.
- Step 7: Gather
    - Scheduler informs worker to return task `z` to the user.
- Step 8: Garbage Collection
    - If no computations depend on `z`, the scheduler removes elements of task `z` from its state.

Conclusions

Conclusions (and Some Weak Points)

- Dask scales computations to a single machine or cluster.
- Handles data larger than memory.
- Collections create task graphs.
- Schedulers execute task graphs on parallel hardware.
- Familiar API, easy to get started.
- Some weak points.
  - Does not always work smoothly (memory leaks).
  - Some computations might be slower with Dask, i.e pandas groupby, merge.

Additional Resources

- Website and documentation: `https://dask.pydata.org/en/latest/`
- Documentation for distributed scheduler: `https://distributed.readthedocs.io/en/latest/`
- Source code: `https://github.com/dask`
- Examples: `https://github.com/dask/dask-examples`
- Matthew Rocklin's blog: `http://matthewrocklin.com/blog/`
- Online course on DataCamp: `https://www.datacamp.com/courses/parallel-computing-with-dask`
- Conference talks on YouTube.

Dask Demo

Thank you for your attention!

Questions?