DASK

Introduction to Dask

A lightweight library for distributed computing in Python

Presented by @jessecdaniel for @PyData_Denver

7/12/2017

Outline

- What Is Dask + Why Should I Care?
- Parallel Computation 101
- *Hands-On:* From Pandas to Dask
- Scaling Out with *dask.distributed*

What Is Dask + Why Should I Care?

Dask is a flexible parallel computing library

- *Task graph scheduler*
- *"Large" collection objects*

Dask is easy to install

Dask easily scales from a laptop to a large cluster

Dask is written in Python and makes use of the C/Fortran stack

What Is Dask + Why Should I Care?

The Pandas API is very intuitive, but data must fit in RAM!

- This limitation continues to look more unreasonable as time passes

Compute intensive algorithms need to scale-out

- GIL makes multicore processing difficult
- Must rely on outside solutions - Cython/Numba/etc.

Why not use PySpark?

- Non-trivial configuration
- JVM overhead (Py4J, serialization)

Parallel Computing 101

A Workload in Serial Execution

$$f = \sum(A * B + C)$$

A	B	C	
3	2	5	11
4	3	8	31
2	3	6	43
1	9	0	52
2	5	4	66

Parallel Computing 101

Distribute compute workload to more resources

1. Break workload into *chunks*
2. Send workers a unit of work
3. Monitor workers and react as appropriate
4. Collect results

Parallel Computing 101

A Workload in Parallel Execution (Two Workers)

$$f = \sum(A * B + C)$$

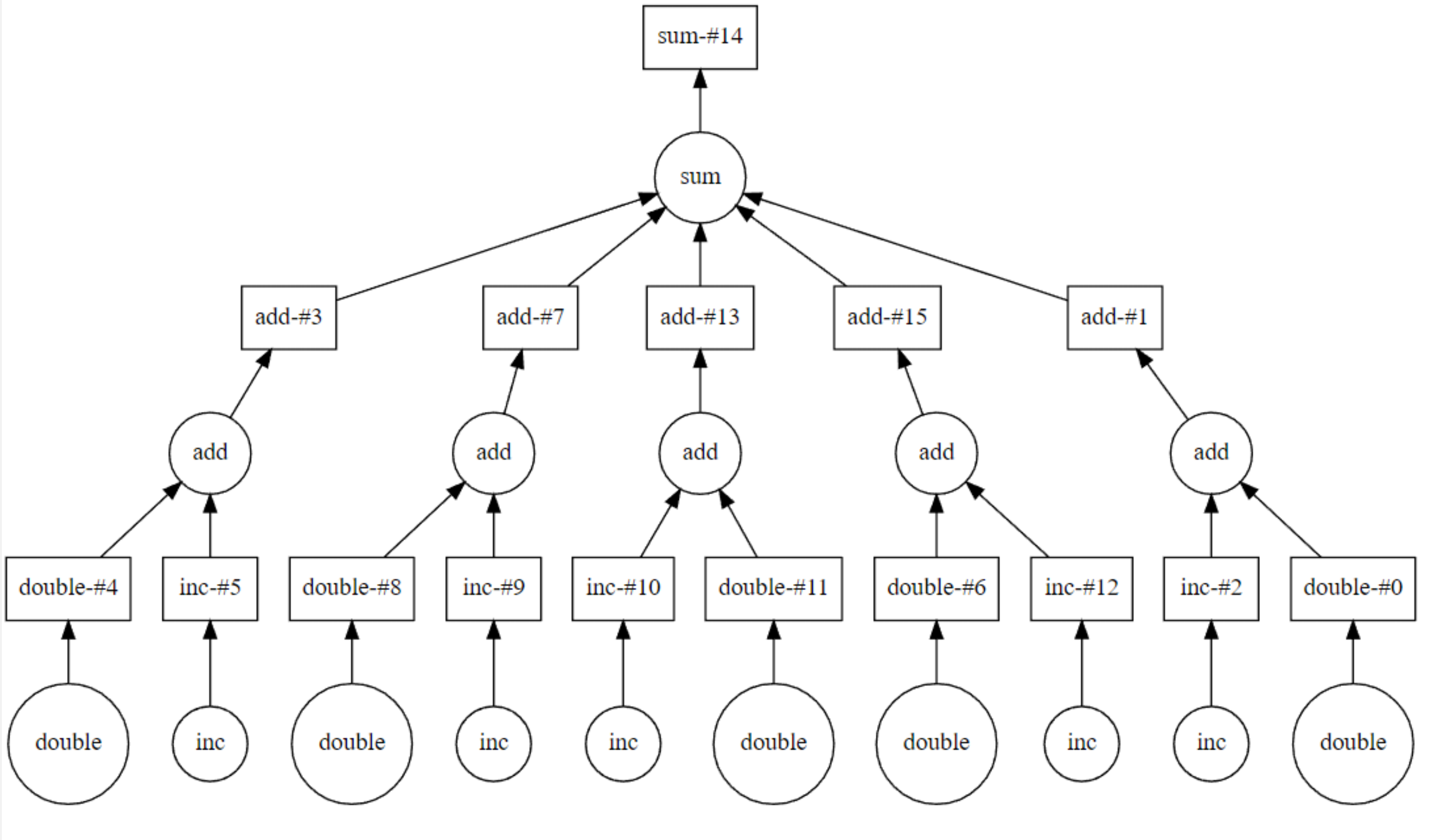
A	B	C	
3	2	5	11
2	3	6	23
2	5	4	37

A	B	C	
4	3	8	20
1	9	0	29

37 + 29 = 66

Parallel Computing 101

A Dask workload consists of *data* and a *task graph*



Parallel Computing 101

A few additional notes before we jump in to some code…

- Dask data objects are lazy
- *dask.dataframe* does not implement the entire Pandas API
- Reindexing and sorting are very expensive and should be avoided if possible
- It’s worth it to learn how to make custom task graphs!

Hands-On: From Pandas to Dask

It's easy to see how similar the Dask APIs are to the Pandas API!

If arrays, bags, or dataframes aren't suitable for your solution, *dask.delayed* can be used to generate custom workloads

- The *dask.delayed* API supports most Python operators, item access, slicing, attribute access, and method calls
- The *dask.delayed* API ***does not*** support iteration, mutating operators, or predicates
  - Tail recursive functions *are* supported

Scaling out with *dask.distributed*

With a solid grasp on the *dask.delayed* API, task graphs, and the Dask DataFrame API, it's not hard at all to take the next step by scaling out

- I demonstrated a local cluster, but setting up remote workers on different machines is very easy!

```
$ dask-scheduler
Scheduler started at 127.0.0.1:8786

$ dask-worker 127.0.0.1:8786
$ dask-worker 127.0.0.1:8786
$ dask-worker 127.0.0.1:8786
```

Remote workers just need TCP access to the scheduler's port

```
>>> from dask.distributed import Client
>>> client = Client('127.0.0.1:8786')
```

Scheduler's IP & port; make sure notebook server has access!

Scaling out with *dask.distributed*

Dask workers can be deployed using Docker, in AWS or other cloud architectures; the sky's the limit!

Hybrid/cross-platform clusters are natively supported

Task scheduler API allows fine grain of control on data persistence, publishing, package sharing, etc.

More Information

Dask: http://dask.pydata.org/en/latest/

dask.distributed: https://distributed.readthedocs.io/en/latest/index.html

GitHub repo: https://github.com/jcdaniel91/pydata-meetup-dask

Thank You!

Please connect with me on LinkedIn @ linkedin.com/in/jcdaniel91

Or follow me on Twitter @jessecdaniel