Parallel computing in Python using Dask, Modin and Joblib

April 16, 2024

Dask basics

Material taken from [Dask documentation] and other links (indicated as appropriate)

- flexible library for parallel computing in Python
- it is implemented on top of `multiprocessing` and `multithreading`
- Composed of two parts:
  - dynamic task scheduling optimized for interactive computational workloads
  - big data collections: parallel arrays, dataframes and lists (extends common interfaces like `numpy`, `pandas` or `iterators`)

Dask basics

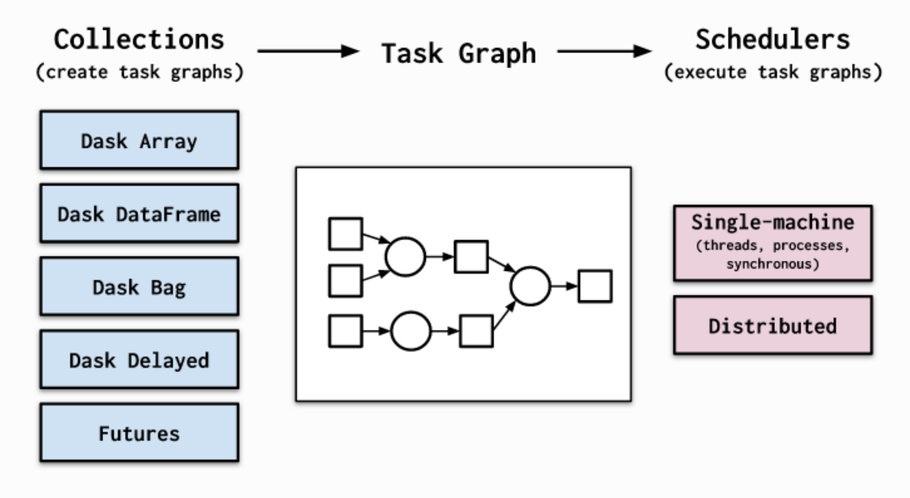
- Advantages:
  - Familiar: provides parallelized numpy array and pandas dataframe objects
  - Flexible: provides a task scheduling interface
  - Native: enables ditributed computing in pure Python with access to the PyData stack
  - Fast: low overhead, low latency, and minimal serialization
  - Scales up: runs on clusters with 1000s of cores
  - Scales down: can trivially run in a laptop using a single process
  - Responsive: designed for interactive computing, providing rapid feedback

Dask vs. PySpark

- In general, Dask is smaller and lighter weight than Spark
- It has fewer features and, instead, is used in conjunction with other libraries, particularly those in the numeric Python ecosystem
- It couples with libraries like Pandas or Scikit-Learn
- Spark is written in Scala and supports various languages, dask is written in Python and only supports Python
- specifically, PySpark runs on JVM

Dask basics

General overview of Dask components



Dask will run in a single machine, but if using `dask.distributed`, it will create processes in several machines

Dask familiar user interface: dataframe example

```
import pandas as pd
df = pd.read_csv('2015-01-01.csv')
df.groupby(df.user_id).value.mean()
```

```
import dask.dataframe as dd
df = dd.read_csv('2015-*-*.csv')
df.groupby(df.user_id).value.mean().compute()
```

Dask familiar user interface: numpy example

```python
import numpy as np
f = h5py.File('myfile.hdf5')
x = np.array(f['/small-data'])

x - x.mean(axis=1)
```

```python
import dask.array as da
f = h5py.File('myfile.hdf5')
x = da.from_array(f['/big-data'],
                  chunks=(1000, 1000))
x - x.mean(axis=1).compute()
```

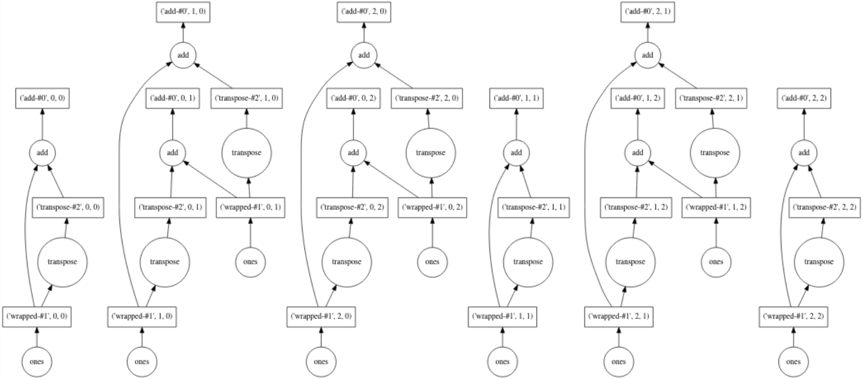
chunks tell dask.array how to break up the underlying array into chunks (refer to [Dask chunks documentation])

Dask familiar user interface: numpy example

```python
import dask.array as da
x = da.ones((15, 15), chunks=(5, 5))

y = x + x.T

# y.compute()
y.visualize(filename='transpose.svg')
```



Dask familiar user interface: iterators, Toolz and PySpark

```
import dask.bag as db
b = db.read_text('2015-*-*.json.gz').map(json.loads)
b.pluck('name').frequencies().topk(10, lambda pair: pair[1]).compute()
```

Dask familiar user interface: for loop and custom code

```python
from dask import delayed
L = []
for fn in filenames:
    data = delayed(load)(fn)
    L.append(delayed(process)(data))

result = delayed(summarize)(L)
result.compute()
```

Dask familiar user interface: futures

```python
from dask.distributed import Client
client = Client('scheduler:port')

futures = []
for fn in filenames:
    future = client.submit(load, fn)
    futures.append(future)

summary = client.submit(summarize, futures)
summary.result()
```

Dask familiar user interface: futures

- `Client` is needed to use with `future` interfaces

```
from dask.distributed import Client

client = Client()  # start local workers as processes
# or
client = Client(processes=False)  # start local workers as threads
```

Dask distributed

- If you create a `Client` without providing an address it will start up a local scheduler and worker
- Dask distribute allows you to manage clusters

  `python -m pip install dask distributed --upgrade`

```
$ 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
```

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

Dask task graphs

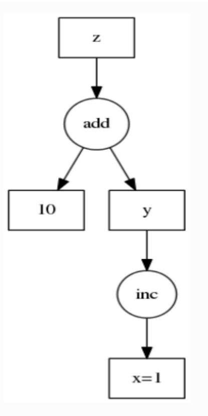
Dask encodes programs as dictionaries or similar, which are represented as graphs

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

def add(a, b):
    return a + b

x = 1
y = inc(x)
z = add(y, 10)
```

```python
d = {'x': 1,
     'y': (inc, 'x'),
     'z': (add, 'y', 10)}
```



Dask dataframes

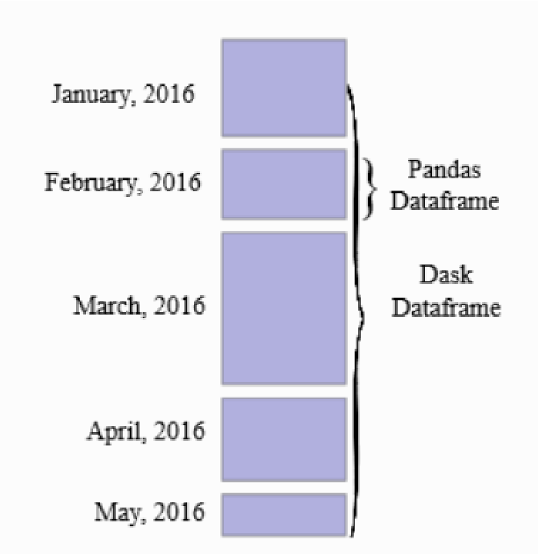
Material from Dask dataframe doc

- Dask dataFrame: used usually when Pandas fails due to data size or computation speed:
  - Manipulating large datasets, even when those datasets don't fit in memory
  - Accelerating long computations by using many cores
  - Distributed computing on large datasets with standard pandas operations like groupby, join, and time series computations

Dask dataframes

- Dask dataFrame may not be the best choice in the following situations:
  - If the dataset fits into RAM
  - If the dataset doesn't fit into the pandas tabular model (in that case if the data fits, you may use dask.bag or dask.array)
  - If you need functions that are not implemented in Dask dataFrame
  - If you need database optimized operations

Dask dataframe example



Dask dataframes and pandas

- Trivially parallelizable operations: element-wise, row-wise, loc, aggregations, etc
- Cleverly parallelizable operations: groupby (agg and index), counts, drop_duplicates, merge etc

Dask dataframes and pandas

- Dask **does not implement all** pandas interface
- Some limitations:
  - setting a new index from an unsorted column is expensive
  - operations like groupby-apply and join on unsorted columns require setting the index, which as said above, is expensive
  - operations that are slow in pandas, like iterating row-by-row will remain slow in dask

Dask dataframes overheads

- A note on GIL (Global Interpreter Lock):
  - pandas is more GIL bound than numpy, therefore operations on dask arrays should be faster than operations on dask dataframes

Dask ML

```
>>> from dask_glm.datasets import make_classification
>>> X, y = make_classification()
>>> lr = LogisticRegression()
>>> lr.fit(X, y)
>>> lr.decision_function(X)
>>> lr.predict(X)
>>> lr.predict_proba(X)
>>> lr.score(X, y)
```

```
from dask_ml.xgboost import XGBRegressor

est = XGBRegressor(...)
est.fit(train, train_labels)
```

Modin

[Modin documentation in github]
[Modin documentation in readthedocs]

Scale your pandas workflows by changing one line of code! (is this serious?? :)

```python
# import pandas as pd
import modin.pandas as pd
```

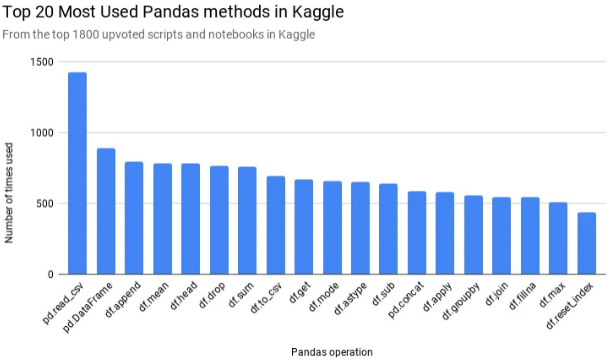
Modin

```
pip install modin[ray] # Install Modin dependencies and Ray to run on Ray
pip install modin[dask] # Install Modin dependencies and Dask to run on Dask
pip install modin[all] # Install all of the above
```

Modin

Q: What is Modin?

A: An alternative to handle 100GB or 1TB datasets not supported by pandas



Pandas implemented functions from Kaggle's most used

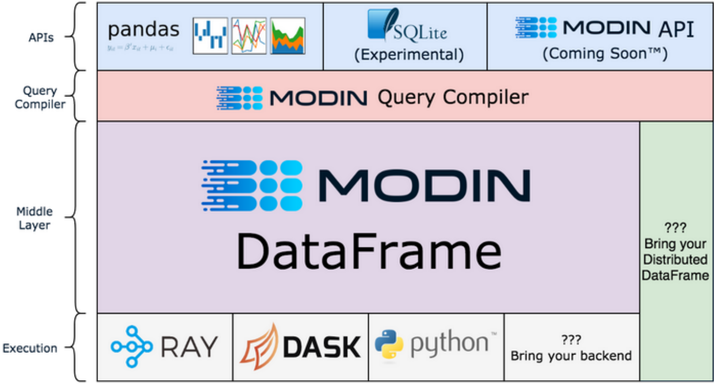
Source: `https://towardsdatascience.com/how-to-speed-up-pandas-with-modin-84aa6a87bcdb`

Modin

pandas Object	Modin's Ray Engine Coverage	Modin's Dask Engine Coverage	Modin's Unidist Engine Coverage
`pd.DataFrame`	api coverage 90.8%	api coverage 90.8%	api coverage 90.8%
`pd.Series`	api coverage 88.05%	api coverage 88.05%	api coverage 88.05%
`pd.read_csv`	✅	✅	✅
`pd.read_table`	✅	✅	✅
`pd.read_parquet`	✅	✅	✅
`pd.read_sql`	✅	✅	✅
`pd.read_feather`	✅	✅	✅
`pd.read_excel`	✅	✅	✅
`pd.read_json`	✳️	✳️	✳️
`pd.read_<other>`	✴️	✴️	✴️

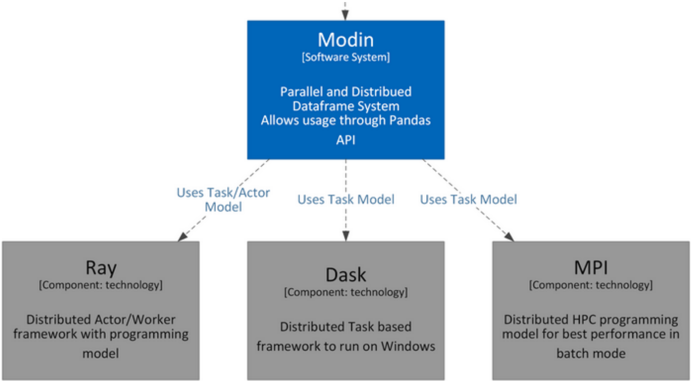
Source: `https://github.com/modin-project/modin?tab=readme-ov-file#pandas-api-coverage`

Modin abstract architecture



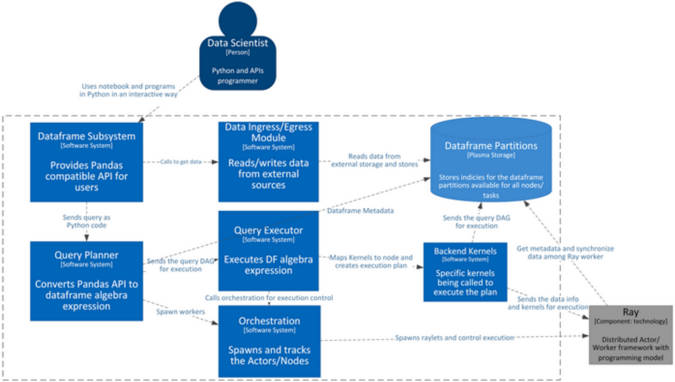
Source: `https://modin.readthedocs.io/en/stable/development/architecture.html`

Modin execution engines



Source: `https://modin.readthedocs.io/en/stable/development/architecture.html`

Modin details for Ray execution



Source: `https://modin.readthedocs.io/en/stable/development/architecture.html`

Joblib

Joblib

- Transparent and fast disk-caching
- Embarrassingly parallel
- Fast compressed Persistence

Joblib

Switching different Parallel Computing Back-ends:

- "loky"
- "multiprocessing"
- "threading"
- "dask"