Dask.distributed

Dask.distributed is a lightweight library for distributed computing in Python. It extends both the `concurrent.futures` and `dask` APIs to moderate sized clusters.

Architecture

Dask.distributed is a centrally managed, distributed, dynamic task scheduler. The central `dask-scheduler` process coordinates the actions of several `dask-worker` processes spread across multiple machines and the concurrent requests of several clients.

The scheduler is asynchronous and event driven, simultaneously responding to requests for computation from multiple clients and tracking the progress of multiple workers. The event-driven and asynchronous nature makes it flexible to concurrently handle a variety of workloads coming from multiple users at the same time while also handling a fluid worker population with failures and additions. Workers communicate amongst each other for bulk data transfer over TCP.

Scikit-Learn & Joblib

Many Scikit-Learn algorithms are written for parallel execution using Joblib, which natively provides thread-based and process-based parallelism.

Dask can scale these Joblib-backed algorithms out to a cluster of machines by providing an alternative Joblib backend. The following demonstrates how to use Dask to parallelize a grid search across a cluster.

Distributed Learning

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.

In [25]:
```
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
```

Create Scikit-Learn Estimator

In [10]:
```
from sklearn.datasets import make_classification
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV
import pandas as pd
import numpy as np
```

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 [11]:
```
X, y = make_classification(n_samples=20000, random_state=0)
print("X: ", np.size(X))
print("y: ",np.size(y))
```

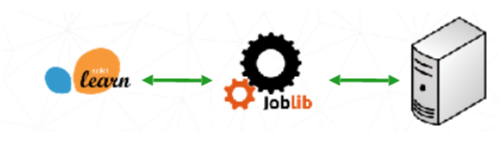
```
X:  400000
y:  20000
```

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

In [11]:
```
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]}
```

In [12]:
```
grid_search = GridSearchCV(SVC(gamma='auto', random_state=0, probability=True),
                           param_grid=param_grid,
                           return_train_score=False,
                           iid=True,
                           cv=3,
                           n_jobs=-1)
```

Single-Machine Parallelism with Scikit-Learn



In [13]:
```
%%time
grid_search.fit(X, y);
print("")
```

```
CPU times: user 1min 39s, sys: 1.59 s, total: 1min 40s
Wall time: 51min 20s
```

Multi-Machine Parallelism with Dask

In [26]:
```
from dask.distributed import Client
tritonCluster = Client("tcp://192.168.86.33:8786")
```

In [27]:
```
tritonCluster # 4 raspiberry pi, laptop, and desktop
```

Out[27]:

Client	Cluster
Scheduler: tcp://192.168.86.33:8786 **Dashboard:** http://192.168.86.33:8787/status	**Workers:** 6 **Cores:** 30 **Memory:** 29.57 GB

In [19]:
```
import joblib
```

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

In [20]:
```
%%time

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

```
CPU times: user 1min 50s, sys: 2.11 s, total: 1min 52s
Wall time: 25min 34s
```

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 [21]:
```
pd.DataFrame(grid_search.cv_results_).head()
```

Out[21]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_C	param_kernel	param_shrinking	params	split0_test_score	spli
0	450.046474	154.745756	15.068107	1.539063	0.001	rbf	True	{'C': 0.001, 'kernel': 'rbf', 'shrinking': True}	0.821509	
1	356.718916	81.222665	14.177350	0.523499	0.001	rbf	False	{'C': 0.001, 'kernel': 'rbf', 'shrinking': False}	0.821509	
2	257.365047	118.632740	11.509992	0.919539	0.001	poly	True	{'C': 0.001, 'kernel': 'poly', 'shrinking': True}	0.796610	
3	167.464683	2.180848	11.047203	0.714162	0.001	poly	False	{'C': 0.001, 'kernel': 'poly', 'shrinking': Fa...	0.796610	
4	387.264651	115.009896	11.571863	0.599657	0.001	sigmoid	True	{'C': 0.001, 'kernel': 'sigmoid', 'shrinking':...	0.834708	

In [22]:
```
grid_search.predict(X)[:5]
```

Out[22]:
```
array([0, 1, 0, 1, 1])
```

In [23]:
```
grid_search.score(X, y)
```

Out[23]:
```
0.87735
```

To use the Dask backend to Joblib you have to create a Client, and wrap your code with `joblib.parallel_backend('dask')`.

```
from dask.distributed import Client
import joblib

client = Client(processes=False) # create local cluster
# client = Client("scheduler-address:8786")  # or connect to remote cluster

with joblib.parallel_backend('dask'):
    # Your scikit-learn code
```

Another Intense Calculation Example

Parallel Monte Carlo with Dask

we will look at how to implement a Monte Carlo simulation

In [1]:
```
import numpy as np
```

In [25]:
```
def random_walk(s0, mu, sigma, days):
    dt = 1/365.
    prices = np.zeros(days)
    shocks = np.zeros(days)
    prices[0] = s0
    for i in range(1, days):
        e = np.random.normal(loc=mu * dt, scale=sigma * np.sqrt(dt))
        prices[i] = prices[i-1] * (1 + e)
    return prices
```

In [26]:
```
days = 365 * 4  # days to expire
s0 = 100        # current underlying price
mu = 0.02       # drift
sigma = 0.2     # volatility
K = 100         # strike price
```

In [27]:
```
A = np.average(random_walk(s0, mu, sigma, days))
```

In [28]:
```
C = max(0, A - K)
```

In [29]:
```
n = 10000
```

In [30]:
```
%time np.average([max(0, np.average(random_walk(s0, mu, sigma, days)) - K) for i in range(0, n)])
```

```
CPU times: user 2min 48s, sys: 1.12 s, total: 2min 49s
Wall time: 2min 49s
```

Out[30]:
```
11.969836076845231
```

In [32]:
```
from dask import delayed
```

In [33]:
```
s0 = 100
K = 100
mu = 0.02
sigma = 0.2
days = 365*4
n = 10000
```

In [34]:
```
result = delayed(np.average)([
    delayed(max)(
        0,
        delayed(np.average)(random_walk(s0, mu, sigma, days)) - K
    ) for i in range(0, n)
])
```

When we create a `Client` object it registers itself as the default Dask scheduler. All `.compute()` methods will automatically start using the distributed system.

In [36]:
```
%time result.compute()
```

```
CPU times: user 4.68 s, sys: 778 ms, total: 5.45 s
Wall time: 39.1 s
```

Out[36]:
```
11.491952733801552
```

Handwritten Digit Recognition Using scikit-learn

This is an example where big cluster-wide parallelism isn't helpful

Goal

The goal is to take an image of a handwritten single digit, and determine what that digit is. For every ImageId in the test set, you should predict the correct label.

In [37]:
```
import matplotlib as mpl
import matplotlib.pyplot as plt

%matplotlib inline
```

Load Data

In [3]:
```
from sklearn.datasets import load_digits
digits = load_digits()
print(digits.data.shape)
```

```
(1797, 64)
```

In [39]:
```
digits.keys()
```

Out[39]:
```
dict_keys(['data', 'target', 'frame', 'feature_names', 'target_names', 'images', 'DESCR'])
```

Data

In [40]:
```
print("Label Data Shape", digits.target.shape)
```

```
Label Data Shape (1797,)
```

In [41]:
```
pd.DataFrame(digits.data).head()
```

Out[41]:

	0	1	2	3	4	5	6	7	8	9	...	54	55	56	57	58	59	60	61	62	63
0	0.0	0.0	5.0	13.0	9.0	1.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	6.0	13.0	10.0	0.0	0.0	0.0
1	0.0	0.0	0.0	12.0	13.0	5.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	11.0	16.0	10.0	0.0	0.0
2	0.0	0.0	0.0	4.0	15.0	12.0	0.0	0.0	0.0	0.0	...	5.0	0.0	0.0	0.0	0.0	3.0	11.0	16.0	9.0	0.0
3	0.0	0.0	7.0	15.0	13.0	1.0	0.0	0.0	0.0	8.0	...	9.0	0.0	0.0	0.0	7.0	13.0	13.0	9.0	0.0	0.0
4	0.0	0.0	0.0	1.0	11.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	2.0	16.0	4.0	0.0	0.0

5 rows × 64 columns

Target

In [42]:
```
print("Label Data Shape", digits.target.shape)
```

```
Label Data Shape (1797,)
```

In [43]:
```
pd.DataFrame(digits.target).head()
```

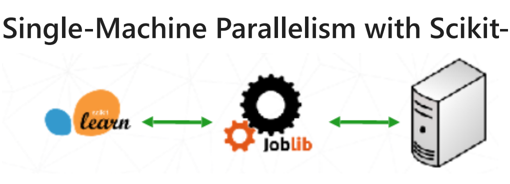
Out[43]:

	0
0	0
1	1
2	2
3	3
4	4

Splitting Data into Training and Test Sets (Digits Dataset)

In [5]:
```
from sklearn.model_selection import train_test_split
X,_,y,_ = train_test_split(digits.data, digits.target, test_size=0.25, random_state=0)
```

Single-Machine Parallelism with Scikit-Learn



In [50]:
```
from sklearn.ensemble import RandomForestClassifier
```

In [57]:
```
clf = RandomForestClassifier(n_estimators=8000)
```

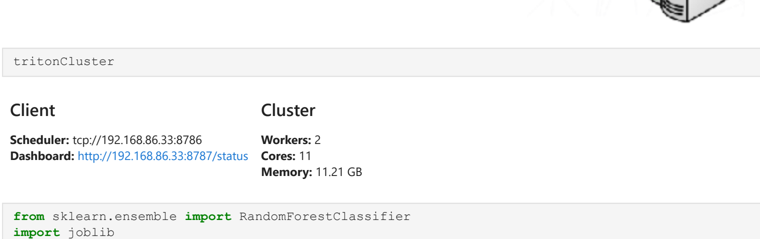
In [58]:
```
%%time
clf.fit(X,y)
```

```
CPU times: user 15min, sys: 3.95 s, total: 15min 4s
Wall time: 15min 1s
```

Out[58]:
```
RandomForestClassifier(n_estimators=8000)
```

Multi-Machine Parallelism with Dask

num of PI: 1



In [5]:
```
tritonCluster
```

Out[5]:

Client	Cluster
Scheduler: tcp://192.168.86.33:8786 **Dashboard:** http://192.168.86.33:8787/status	**Workers:** 2 **Cores:** 11 **Memory:** 11.21 GB

In [6]:
```
from sklearn.ensemble import RandomForestClassifier
import joblib
```

In [7]:
```
clf = RandomForestClassifier(n_estimators=8000)
```

In [8]:
```
import joblib
```

In [12]:
```
%%time
with joblib.parallel_backend('dask'):
    clf.fit(X,y)
```

```
CPU times: user 55.9 s, sys: 5.65 s, total: 1min 1s
Wall time: 3min 57s
```

num of PI: 2

In [13]:
```
tritonCluster
```

Out[13]:

Client	Cluster
Scheduler: tcp://192.168.86.33:8786 **Dashboard:** http://192.168.86.33:8787/status	**Workers:** 3 **Cores:** 15 **Memory:** 13.14 GB

In [14]:
```
%%time
with joblib.parallel_backend('dask'):
    clf.fit(X,y)
```

```
CPU times: user 52 s, sys: 3.73 s, total: 55.7 s
Wall time: 3min 1s
```

num of PI: 3

In [15]:
```
tritonCluster
```

Out[15]:

Client	Cluster
Scheduler: tcp://192.168.86.33:8786 **Dashboard:** http://192.168.86.33:8787/status	**Workers:** 4 **Cores:** 19 **Memory:** 15.08 GB

In [16]:
```
%%time
with joblib.parallel_backend('dask'):
    clf.fit(X,y)
```

```
CPU times: user 50.9 s, sys: 3 s, total: 53.9 s
Wall time: 2min 32s
```

num of PI: 4

In [17]:
```
tritonCluster
```

Out[17]:

Client	Cluster
Scheduler: tcp://192.168.86.33:8786 **Dashboard:** http://192.168.86.33:8787/status	**Workers:** 5 **Cores:** 23 **Memory:** 17.02 GB

In [18]:
```
%%time
with joblib.parallel_backend('dask'):
    clf.fit(X,y)
```

```
CPU times: user 41.5 s, sys: 2.5 s, total: 44 s
Wall time: 2min 14s
```

num of PI: 4 + Desktop

In [20]:
```
tritonCluster
```

Out[20]:

Client	Cluster
Scheduler: tcp://192.168.86.33:8786 **Dashboard:** http://192.168.86.33:8787/status	**Workers:** 6 **Cores:** 30 **Memory:** 29.57 GB

In [22]:
```
%%time
with joblib.parallel_backend('dask'):
    clf.fit(X,y)
```

```
CPU times: user 32 s, sys: 1.91 s, total: 33.9 s
Wall time: 2min 4s
```