07-AsynchronousProcessing

August 10, 2020

1 Asynchronous Processing

While many parallel applications can be described as maps, some can be more complex. In this section we look at the asynchronous `concurrent.futures` interface, which provides a simple API for ad-hoc parallelism. This is useful for when your computations don't fit a regular pattern.

1.0.1 Executor.submit

The `submit` method starts a computation in a separate thread or process and immediately gives us a `Future` object that refers to the result. At first, the future is pending. Once the function completes the future is finished.

We collect the result of the task with the `.result()` method, which does not return until the results are available.

[1]:
```
%%time
from time import sleep

def slowadd(a, b, delay=1):
    sleep(delay)
    return a + b

slowadd(1,1)
```

```
CPU times: user 1.88 ms, sys: 269 µs, total: 2.15 ms
Wall time: 1 s
```

[1]:
```
2
```

[2]:
```
from concurrent.futures import ThreadPoolExecutor

e = ThreadPoolExecutor()
future = e.submit(slowadd, 1, 2)
future
```

[2]:
```
<Future at 0x7f66444d8d90 state=running>
```

[3]:
```
future.result()
```

[3]: 3

1.0.2 Submit many tasks, receive many futures

Because submit returns immediately we can submit many tasks all at once and they will execute in parallel.

[4]:
```
%%time
results = [slowadd(i, i, delay=1) for i in range(8)]
print(results)
```

```
[0, 2, 4, 6, 8, 10, 12, 14]
CPU times: user 130 µs, sys: 4.07 ms, total: 4.2 ms
Wall time: 8.01 s
```

[5]:
```
%%time
e = ThreadPoolExecutor()
futures = [e.submit(slowadd, i, i, delay=1) for i in range(8)]
results = [f.result() for f in futures]
print(results)
```

```
[0, 2, 4, 6, 8, 10, 12, 14]
CPU times: user 5.18 ms, sys: 155 µs, total: 5.34 ms
Wall time: 2 s
```

- Submit fires off a single function call in the background, returning a future.
- When we combine submit with a single for loop we recover the functionality of map.
- When we want to collect our results we replace each of our futures, `f`, with a call to `f.result()`
- We can combine submit with multiple for loops and other general programming to get something more general than map.

1.0.3 Exercise 7.1

Parallelize the following code with e.submit

1. Replace the `results` list with a list called `futures`
2. Replace calls to `slowadd` and `slowsub` with `e.submit` calls on those functions
3. At the end, block on the computation by recreating the `results` list by calling `.result()` on each future in the `futures` list.

[6]:
```
%%time
from time import sleep

def slowadd(a, b, delay=1):
```

```
    sleep(delay)
    return a + b

def slowsub(a, b, delay=1):
    sleep(delay)
    return a - b

results = []
for i in range(4):
    for j in range(4):
        if i < j:
            results.append(slowadd(i, j, delay=1))
        elif i > j:
            results.append(slowsub(i, j, delay=1))

print(results)
```

```
[1, 2, 3, 1, 3, 4, 2, 1, 5, 3, 2, 1]
CPU times: user 5.78 ms, sys: 0 ns, total: 5.78 ms
Wall time: 12 s
```

1.1 Extract daily stock data from google

[7]:
```
import os  # library to get directory and file paths
import tarfile # this module makes possible to read and write tar archives

def extract_data(name, where):
    datadir = os.path.join(where,name)
    if not os.path.exists(datadir):
       print("Extracting data...")
       tar_path = os.path.join(where, name+'.tgz')
       with tarfile.open(tar_path, mode='r:gz') as data:
          data.extractall(where)

extract_data('daily-stock','data') # this function call will extract json files
```

1.2 Convert data to pandas DataFrames and save it in hdf5 files

HDF5 is a data model, library, and file format for storing and managing data. This format is widely used and is supported by many languages and platforms.

[8]:
```
import json
import pandas as pd
import os, glob
```

```
here = os.getcwd()
datadir = os.path.join(here,'data','daily-stock')
filenames = sorted(glob.glob(os.path.join(datadir, '*.json')))
filenames
```

[8]:
```
['/home/runner/work/big-data/big-data/notebooks/data/daily-stock/aet.json',
 '/home/runner/work/big-data/big-data/notebooks/data/daily-stock/afl.json',
 '/home/runner/work/big-data/big-data/notebooks/data/daily-stock/aig.json',
 '/home/runner/work/big-data/big-data/notebooks/data/daily-stock/al.json',
 '/home/runner/work/big-data/big-data/notebooks/data/daily-stock/amgn.json',
 '/home/runner/work/big-data/big-data/notebooks/data/daily-stock/avy.json',
 '/home/runner/work/big-data/big-data/notebooks/data/daily-stock/b.json',
 '/home/runner/work/big-data/big-data/notebooks/data/daily-stock/bwa.json',
 '/home/runner/work/big-data/big-data/notebooks/data/daily-stock/ge.json',
 '/home/runner/work/big-data/big-data/notebooks/data/daily-stock/hal.json',
 '/home/runner/work/big-data/big-data/notebooks/data/daily-stock/hp.json',
 '/home/runner/work/big-data/big-data/notebooks/data/daily-stock/hpq.json',
 '/home/runner/work/big-data/big-data/notebooks/data/daily-stock/ibm.json',
 '/home/runner/work/big-data/big-data/notebooks/data/daily-stock/jbl.json',
 '/home/runner/work/big-data/big-data/notebooks/data/daily-stock/jpm.json',
 '/home/runner/work/big-data/big-data/notebooks/data/daily-stock/luv.json',
 '/home/runner/work/big-data/big-data/notebooks/data/daily-stock/met.json',
 '/home/runner/work/big-data/big-data/notebooks/data/daily-stock/pcg.json',
 '/home/runner/work/big-data/big-data/notebooks/data/daily-stock/tgt.json',
 '/home/runner/work/big-data/big-data/notebooks/data/daily-stock/usb.json',
 '/home/runner/work/big-data/big-data/notebooks/data/daily-stock/xom.json']
```

1.2.1 Sequential version

[9]:
```
%%time
from concurrent.futures import ProcessPoolExecutor
import json
import pandas as pd

def load_parse_store(fn):

    with open(fn) as f:
        data = [json.loads(line) for line in f] # load

    df = pd.DataFrame(data) # parse

    out_filename = fn[:-5] + '.h5'
    df.to_hdf(out_filename, '/data') # store
    print("Finished : %s" % (out_filename.split(os.path.sep)[-1]))

    return True
```

```
results = [load_parse_store(file) for file in filenames]
```

```
Finished : aet.h5
Finished : afl.h5
Finished : aig.h5
Finished : al.h5
Finished : amgn.h5
Finished : avy.h5
Finished : b.h5
Finished : bwa.h5
Finished : ge.h5
Finished : hal.h5
Finished : hp.h5
Finished : hpq.h5
Finished : ibm.h5
Finished : jbl.h5
Finished : jpm.h5
Finished : luv.h5
Finished : met.h5
Finished : pcg.h5
Finished : tgt.h5
Finished : usb.h5
Finished : xom.h5
CPU times: user 8.39 s, sys: 1.01 s, total: 9.4 s
Wall time: 9.26 s
```

1.2.2 Exercise 7.2

Parallelize the loop above using `ThreadPoolExecutor` and `map`.

1.3 Read files and load dataframes.

[10]:
```
filenames = sorted(glob.glob(os.path.join('data', 'daily-stock', '*.h5')))
series ={}
for fn in filenames:
    series[fn] = pd.read_hdf(fn)['close']
```

```
␣
↪---------------------------------------------------------------------------

ValueError                                Traceback (most recent call␣
↪last)
```

```
<ipython-input-10-2dab4a15ac74> in <module>
  2 series ={}
  3 for fn in filenames:
----> 4     series[fn] = pd.read_hdf(fn)['close']


/usr/share/miniconda3/envs/big-data/lib/python3.8/site-packages/pandas/
↪io/pytables.py in read_hdf(path_or_buf, key, mode, errors, where, start, stop,␣
↪columns, iterator, chunksize, **kwargs)
    400             for group_to_check in groups[1:]:
    401                 if not _is_metadata_of(group_to_check,␣
↪candidate_only_group):
--> 402                     raise ValueError(
    403                         "key must be provided when HDF5 "
    404                         "file contains multiple datasets."


ValueError: key must be provided when HDF5 file contains multiple␣
↪datasets.
```

1.4 Application

Given our HDF5 files from the last section we want to find the two datasets with the greatest pair-wise correlation. This forces us to consider all $n \times (n-1)$ possibilities.

[11]:
```
%%time

results = {}

for a in filenames:
    for b in filenames:
        if a != b:
            results[a, b] = series[a].corr(series[b])

((a, b), corr) = max(results.items(), key=lambda kv: kv[1])
print("%s matches with %s with correlation %f" % (a, b, corr))
```

```
␣
↪---------------------------------------------------------------------------

KeyError                                  Traceback (most recent call␣
↪last)

<timed exec> in <module>
```

```
KeyError: 'data/daily-stock/aet.h5'
```

We use matplotlib to visually inspect the highly correlated timeseries

[12]:
```
%matplotlib inline
import matplotlib.pyplot as plt
plt.figure(figsize=(10, 4))
plt.plot(series[a]/series[a].max())
plt.plot(series[b]/series[b].max())
plt.xticks(visible=False);
```

```
␣
↪---------------------------------------------------------------------------

KeyError                                  Traceback (most recent call␣
↪last)

<ipython-input-12-1d4fb0c2882b> in <module>
  2 import matplotlib.pyplot as plt
  3 plt.figure(figsize=(10, 4))
----> 4 plt.plot(series[a]/series[a].max())
  5 plt.plot(series[b]/series[b].max())
  6 plt.xticks(visible=False);


KeyError: 'data/daily-stock/aet.h5'
```

```
<Figure size 720x288 with 0 Axes>
```

1.5 Analysis

This computation starts out by loading data from disk. We already know how to parallelize it:

```
series = {}
for fn in filenames:
    series[fn] = pd.read_hdf(fn)['x']
```

It follows with a doubly nested for loop with an if statement.

```
results = {}
for a in filenames:
    for b in filenames:
        if a != b:
            results[a, b] = series[a].corr(series[b])
```

It *is* possible to solve this problem with `map`, but it requires some cleverness. Instead we'll learn `submit`, an interface to start individual function calls asynchronously.

It finishes with a reduction on small data. This part is fast enough.

```
((a, b), corr) = max(results.items(), key=lambda kv: kv[1])
```

[13]:
```
%%time

from concurrent.futures import ThreadPoolExecutor as PoolExecutor


e = PoolExecutor()

def corr( serie_a, serie_b):

    return serie_a.corr(serie_b)

futures = {}

for a in filenames:
    for b in filenames:
        if a != b:
            futures[a, b] = e.submit( corr, series[a], series[b])

results = {k : f.result() for k, f in futures.items()}


((a, b), corr) = max(results.items(), key=lambda kv: kv[1])
print("%s matches with %s with correlation %f" % (a, b, corr))
```

```
␣
↪---------------------------------------------------------------------------

KeyError                                  Traceback (most recent call␣
↪last)

<timed exec> in <module>


KeyError: 'data/daily-stock/aet.h5'
```

1.5.1 Exercise 7.3

- Parallelize pair-wise correlations with `e.submit`

- Implement two versions one using Processes, another with Threads by replacing `e` with a ProcessPoolExecutor:

Threads

```
from concurrent.futures import ThreadPoolExecutor
e = ThreadPoolExecutor(4)
```

Processes Be careful, a `ProcessPoolExecutor` does not run in the jupyter notebook cell. You must run your file in a terminal.

```
from concurrent.futures import ProcessPoolExecutor
e = ProcessPoolExecutor(4)
```

- How does performance vary?

1.6 Some conclusions about futures

- `submit` functions can help us to parallelize more complex applications
- It didn't actually speed up the code very much
- Threads and Processes give some performance differences
- This is not very robust.