Speed Up Your Data Processing
Parallel and Asynchronous Programming in Data Science

By: Chin Hwee Ong (@ongchinhwee)

23 July 2020
About me

Ong Chin Hwee 王敬惠

- Data Engineer @ ST Engineering
- Background in aerospace engineering + computational modelling
- Contributor to pandas 1.0 release
- Mentor team at BigDataX

@ongchinHwee
A typical data science workflow

1. Extract raw data
2. Process data
3. Train model
4. Evaluate and deploy model
Bottlenecks in a data science project

- Lack of data / Poor quality data
- Data processing
  - The 80/20 data science dilemma
    - In reality, it’s closer to 90/10
Data Processing in Python

● For loops in Python
  ○ Run on the interpreter, not compiled
  ○ Slow compared with C

```python
a_list = []
for i in range(100):
    a_list.append(i*i)
```
Data Processing in Python

- **List comprehensions**
  - *Slightly faster* than for loops
  - No need to call append function at each iteration

```python
a_list = [i*i for i in range(100)]
```
Challenges with Data Processing

● Pandas
  ○ Optimized for in-memory analytics using DataFrames
  ○ Performance + out-of-memory issues when dealing with large datasets (> 1 GB)

```python
import pandas as pd
import numpy as np
df = pd.DataFrame(list(range(100)))
squared_df = df.apply(np.square)
```
Challenges with Data Processing

● “Why not just use a Spark cluster?”

**Communication overhead**: Distributed computing involves communicating between (independent) machines across a network!

“**Small Big Data**”(*): Data too big to fit in memory, but not large enough to justify using a Spark cluster.

What is parallel processing?
Let’s imagine I work at a cafe which sells toast.
Task 1: Toast 100 slices of bread

Assumptions:
1. I’m using single-slice toasters. (Yes, they actually exist.)
2. Each slice of toast takes 2 minutes to make.
3. No overhead time.

Image taken from:
https://www.mitsubishielectric.co.jp/home/breadoven/product/to-st1-t/feature/index.html
Sequential Processing

= 25 bread slices

@ongchinhwee
Sequential Processing

= 25 bread slices

Processor/Worker:
Toaster

@ongchinhwee
Sequential Processing

= 25 bread slices  Processor/Worker: Toaster  = 25 toasts

@ongchinhwee
Sequential Processing

**Execution Time** = 100 toasts × 2 minutes/toast
= **200 minutes**

@ongchinhwee
Parallel Processing

= 25 bread slices

@ongchinhwee
Parallel Processing
Parallel Processing

Processor (Core):
Toaster

@ongchinhwee
Parallel Processing

Processor (Core): Toaster

Task is executed using a pool of 4 toaster subprocesses.

Each toasting subprocess runs in parallel and independently from each other.
Parallel Processing

**Processor (Core):** Toaster

Output of each toasting process is **consolidated** and **returned** as an overall output (which may or may not be ordered).

@ongchinhwee
Parallel Processing

Execution Time
= 100 toasts × 2 minutes/toast ÷ 4 toasters
= 50 minutes

Speedup
= 4 times
Synchronous vs Asynchronous Execution
What do you mean by “Asynchronous”?
Task 2: Brew coffee

Assumptions:
1. I can do other stuff while making coffee.
2. One coffee maker to make one cup of coffee.
3. Each cup of coffee takes 5 minutes to make.

Image taken from: https://www.crateandbarrel.com/breville-barista-espresso-machine/s267619
Synchronous Execution

Task 2: Brew a cup of coffee on coffee machine
Duration: 5 minutes
Synchronous Execution

Task 1: Toast two slices of bread on single-slice toaster after Task 2 is completed
Duration: 4 minutes

Task 2: Brew a cup of coffee on coffee machine
Duration: 5 minutes
Synchronous Execution

Task 1: Toast two slices of bread on single-slice toaster after Task 2 is completed
Duration: 4 minutes

Task 2: Brew a cup of coffee on coffee machine
Duration: 5 minutes

Output: 2 toasts + 1 coffee

Total Execution Time = 5 minutes + 4 minutes = 9 minutes
Asynchronous Execution

While brewing coffee:

Make some toasts:

@ongchinhwee
Asynchronous Execution

Output: 2 toasts + 1 coffee
Total Execution Time = 5 minutes

@ongchinhwee
When is it a good idea to go for parallelism?

(or, “Is it a good idea to simply buy a 256-core processor and parallelize all your codes?”)
Practical Considerations

- Is your code already optimized?
  - Sometimes, you might need to rethink your approach.
  - Example: Use list comprehensions or map functions instead of for-loops for array iterations.
Practical Considerations

● Is your code already optimized?
● Problem architecture
  ○ Nature of problem limits how successful parallelization can be.
  ○ If your problem consists of processes which depend on each others’ outputs **(Data dependency)** and/or intermediate results **(Task dependency)**, maybe not.
Practical Considerations

- Is your code already optimized?
- Problem architecture
- Overhead in parallelism
  - There will always be parts of the work that cannot be parallelized. → **Amdahl’s Law**
  - Extra time required for coding and debugging (parallelism vs sequential code) → **Increased complexity**
  - **System overhead** including **communication overhead**
Amdahl’s Law and Parallelism

Amdahl’s Law states that the theoretical speedup is defined by the fraction of code $p$ that can be parallelized:

$$S = \frac{1}{(1 - p) + \frac{p}{N}}$$

$S$: Theoretical speedup (theoretical latency)
$p$: Fraction of the code that can be parallelized
$N$: Number of processors (cores)
Amdahl’s Law and Parallelism

If there are no parallel parts ($p = 0$): \textbf{Speedup} = 0
Amdahl’s Law and Parallelism

If there are no parallel parts \( (p = 0) \): \textbf{Speedup = 0}

If all parts are parallel \( (p = 1) \): \textbf{Speedup = N \rightarrow \infty}
Amdahl’s Law and Parallelism

If there are no parallel parts \((p = 0)\): Speedup = 0

If all parts are parallel \((p = 1)\): Speedup = \(N \rightarrow \infty\)

Speedup is limited by fraction of the work that is not parallelizable - will not improve even with infinite number of processors

@ongchinhwee
Multiprocessing vs Multithreading

**Multiprocessing:**
System allows executing multiple processes at the same time using multiple processors.
Multiprocessing vs Multithreading

**Multiprocessing:**
System allows executing **multiple processes** at the same time using **multiple processors**

**Multithreading:**
System executes **multiple threads** of sub-processes at the same time within a **single processor**
Multiprocessing vs Multithreading

**Multiprocessing:**
System allows executing multiple processes at the same time using multiple processors.
Better for processing large volumes of data

**Multithreading:**
System executes multiple threads of sub-processes at the same time within a single processor.
Best suited for I/O or blocking operations

@ongchinwee
Some Considerations

Data processing tends to be more compute-intensive

→ Switching between threads become increasingly inefficient

→ Global Interpreter Lock (GIL) in Python does not allow parallel thread execution
How to do Parallel + Asynchronous in Python?

(without using any third-party libraries)
Parallel + Asynchronous Programming in Python

`concurrent.futures` module

- High-level API for launching asynchronous (async) parallel tasks
- Introduced in Python 3.2 as an abstraction layer over `multiprocessing` module
- Two modes of execution:
  - `ThreadPoolExecutor()` for async multithreading
  - `ProcessPoolExecutor()` for async multiprocessing

@ongchinhwee
ProcessPoolExecutor vs ThreadPoolExecutor

From the Python Standard Library documentation:

For `ProcessPoolExecutor`, this method chops iterables into a number of chunks which it submits to the pool as separate tasks. The (approximate) size of these chunks can be specified by setting `chunksize` to a positive integer. For very long iterables, using a large value for `chunksize` can significantly improve performance compared to the default size of 1. With `ThreadPoolExecutor`, `chunksize` has no effect.
ProcessPoolExecutor vs ThreadPoolExecutor

ProcessPoolExecutor:
- System allows executing multiple processes asynchronously using multiple processors
- Uses multiprocessing module - side-steps GIL

ThreadPoolExecutor:
- System executes multiple threads of sub-processes asynchronously within a single processor
- Subject to GIL - not truly “concurrent”

@ongchinhwee
submit() in concurrent.futures

`Executor.submit()` takes as input:

1. The **function** (callable) that you would like to run, and
2. Input arguments (*args, **kwargs) for that function;

and returns a **futures object** that **represents the execution of the function**.
map() in concurrent.futures

Similar to map(), Executor.map() takes as input:

1. The function (callable) that you would like to run, and
2. A list (iterable) where each element of the list is a single input to that function;

and returns an iterator that yields the results of the function being applied to every element of the list.
Case: Network I/O Operations

**Dataset:** Data.gov.sg Realtime Weather Readings

**API Endpoint URL:** https://api.data.gov.sg/v1/environment/

**Response:** JSON format
Initialize Python modules

```python
import numpy as np
import requests
import json
import sys
import time
import datetime
from tqdm import trange, tqdm
from time import sleep
from retrying import retry
import threading
```
@retry(wait_exponential_multiplier=1000, wait_exponential_max=10000)
def get_airtemp_data_from_date(date):
    print('{}: running {}'.format(threading.current_thread().name, date))
    # for daily API request
    url = "https://api.data.gov.sg/v1/environment/air-temperature?date=" 
        + str(date)
    JSONContent = requests.get(url).json()
    content = json.dumps(JSONContent, sort_keys=True)
    sleep(1)
    print('{}: done with {}'.format(threading.current_thread().name, date))
    return content
Initialize Submission List

date_range = np.array(sorted([datetime.datetime.strftime(
    datetime.datetime.now() - datetime.timedelta(i),
    '%Y-%m-%d') for i in trange(100)]))
Using List Comprehensions

```python
start_cpu_time = time.clock()

data_np = [get_airtemp_data_from_date(str(date)) for date in tqdm(date_range)]

data_np = [get_airtemp_data_from_date(str(date)) for date in tqdm(date_range)]

end_cpu_time = time.clock()
print(end_cpu_time - start_cpu_time)
```

@ongchinhwee
Using List Comprehensions

```python
start_cpu_time = time.clock()
data_np = [get_airtemp_data_from_date(str(date)) for date in tqdm(date_range)]
end_cpu_time = time.clock()
print(end_cpu_time - start_cpu_time)
```

List Comprehensions: **977.88 seconds (~ 16.3mins)**
Using ThreadPoolExecutor

```python
from concurrent.futures import ThreadPoolExecutor, as_completed

start_cpu_time = time.clock()

with ThreadPoolExecutor() as executor:
    future = {executor.submit(get_airtemp_data_from_date, date): date
               for date in tqdm(date_range)}
    resultarray_np = [x.result() for x in as_completed(future)]

end_cpu_time = time.clock()
total_tpe_time = end_cpu_time - start_cpu_time
sys.stdout.write('Using ThreadPoolExecutor: {} seconds.
'.format(total_tpe_time))
```

@ongchinhwee
Using ThreadPoolExecutor

```python
from concurrent.futures import ThreadPoolExecutor, as_completed

start_cpu_time = time.clock()

with ThreadPoolExecutor() as executor:
    future = {executor.submit(get_airtemp_data_from_date, date): date
               for date in tqdm(date_range)}
    resultarray_np = [x.result() for x in as_completed(future)]

end_cpu_time = time.clock()

total_tpe_time = end_cpu_time - start_cpu_time

sys.stdout.write('Using ThreadPoolExecutor: {} seconds.\n'.format(total_tpe_time))
```

ThreadPoolExecutor (40 threads): 46.83 seconds (~20.9 times faster)

@ongchinhwee
Case: Image Processing

**Dataset:** Chest X-Ray Images (Pneumonia)  

**Size:** 1.15GB of x-ray image files with normal and pneumonia (viral or bacterial) cases

**Data Quality:** Images in the dataset are of different dimensions

@ongchinhwee
Initialize Python modules

```python
import numpy as np
from PIL import Image
import os
import sys
import time
```
Initialize image resize process

def image_resize(filepath):
    '''Resize and reshape image'''
    sys.stdout.write('{}: running {}
'.format(os.getpid(), filepath))
    im = Image.open(filepath)
    resized_im = np.array(im.resize((64, 64)))
    sys.stdout.write('{}: done with
{}\n'.format(os.getpid(), filepath))
    return resized_im
Initialize File List in Directory

```python
DIR = './chest_xray/train/NORMAL/'

train_normal = [DIR + name for name in os.listdir(DIR) if os.path.isfile(os.path.join(DIR, name))]
```

No. of images in 'train/NORMAL': **1431**
Using map()

```python
start_cpu_time = time.clock()

result = map(image_resize, train_normal)

output = np.array([x for x in result])

end_cpu_time = time.clock()
total_tpe_time = end_cpu_time - start_cpu_time
sys.stdout.write('Map completed in {} seconds.
'.format(total_tpe_time))
```

@ongchinhwee
Using map()

```python
start_cpu_time = time.clock()
result = map(image_resize, train_normal)
output = np.array([x for x in result])

end_cpu_time = time.clock()
total_tpe_time = end_cpu_time - start_cpu_time
sys.stdout.write('Map completed in {} seconds.
'.format(total_tpe_time))
```

@ongchinhwee

map(): 29.48 seconds
Using List Comprehensions

```
start_cpu_time = time.clock()

listcomp_output = np.array([image_resize(x) for x in train_normal])

end_cpu_time = time.clock()
total_tpe_time = end_cpu_time - start_cpu_time
sys.stdout.write('List comprehension completed in {} seconds.
'.format(total_tpe_time))
```
Using List Comprehensions

```
start_cpu_time = time.clock()
listcomp_output = np.array([image_resize(x) for x in train_normal])
end_cpu_time = time.clock()
total_tpe_time = end_cpu_time - start_cpu_time
sys.stdout.write('List comprehension completed in {} seconds.
'.format(total_tpe_time))
```

List Comprehensions: 29.71 seconds

@ongchinhwee
Using ProcessPoolExecutor

```python
from concurrent.futures import ProcessPoolExecutor

start_cpu_time = time.clock()

with ProcessPoolExecutor() as executor:
    future = executor.map(image_resize, train_normal)

array_np = np.array([x for x in future])

end_cpu_time = time.clock()

total_tpe_time = end_cpu_time - start_cpu_time

sys.stdout.write('ProcessPoolExecutor completed in {} seconds.
'.format(total_tpe_time))

@ongchinhwee
```
Using ProcessPoolExecutor

```python
from concurrent.futures import ProcessPoolExecutor

start_cpu_time = time.clock()

with ProcessPoolExecutor() as executor:
    future = executor.map(image_resize, train_normal)

array_np = np.array([x for x in future])

end_cpu_time = time.clock()

total_tpe_time = end_cpu_time - start_cpu_time

sys.stdout.write('ProcessPoolExecutor completed in {} seconds.\n'.format(total_tpe_time))
```

ProcessPoolExecutor (8 cores): 6.98 seconds (~4.3 times faster)
Key Takeaways

@ongchinhwee
Not all processes should be parallelized

- Parallel processes come with overheads
  - Amdahl’s Law on parallelism
  - System overhead including communication overhead
  - If the cost of rewriting your code for parallelization outweighs the time savings from parallelizing your code, consider other ways of optimizing your code instead.
References

Official Python documentation on concurrent.futures (https://docs.python.org/3/library/concurrent.futures.html)

Source code for ThreadPoolExecutor (https://github.com/python/cpython/blob/3.8/Lib/concurrent/futures/thread.py)

Reach out to me!

LinkedIn: ongchinhwee
Twitter: @ongchinhwee
GitHub: hweecat

And check out my slides on:
GitHub: hweecat/talk_parallel-async-python