並列分散処理 最終報告書

チーム E　大城 慶知　眞榮城 隆守　伊波卓浩　宮良友也

August 3, 2018

1　演習の背景

「講義で説明した並列分散処理を実践し、他者に有益な情報となるように共有せよ。」という課題が渡された。
　b3 前期はメンバーが忙しく時間も取れないので軽量かつ有益な情報をということで、GPU マシンを使った並列処理を断腸の思いで断念し、Python における非同期処理を用いた I/O の並列処理を行うことにした。

2　目的

Python のネットワーク I/O に対して、async await の使うことで非同期処理を行い、スレッドと concurrent を用いて並列化を行うことで速度向上率を調べる。

3　メンバーの担当

大城 慶知 総括
　眞榮城 隆守・宮良友也 Python ファイルの記述。グラフの生成
　伊波卓浩 FizzBuzz のデモ作成

4　Python の事前知識

4.1　スレッドの制約

Python では、GIL(Global Interpreter Lock) と呼ばれる制約がある。GIL とは、Figure 1 と Figure2 のように、CPU バウンドで Python を実行した際に一つだけしかスレッドのリソースを起動できない制約である。そのため、Python の CPU バウンドの並列処理はプロセスを使って、I/O バウンドの並列処理はスレッドを行う事が多い。

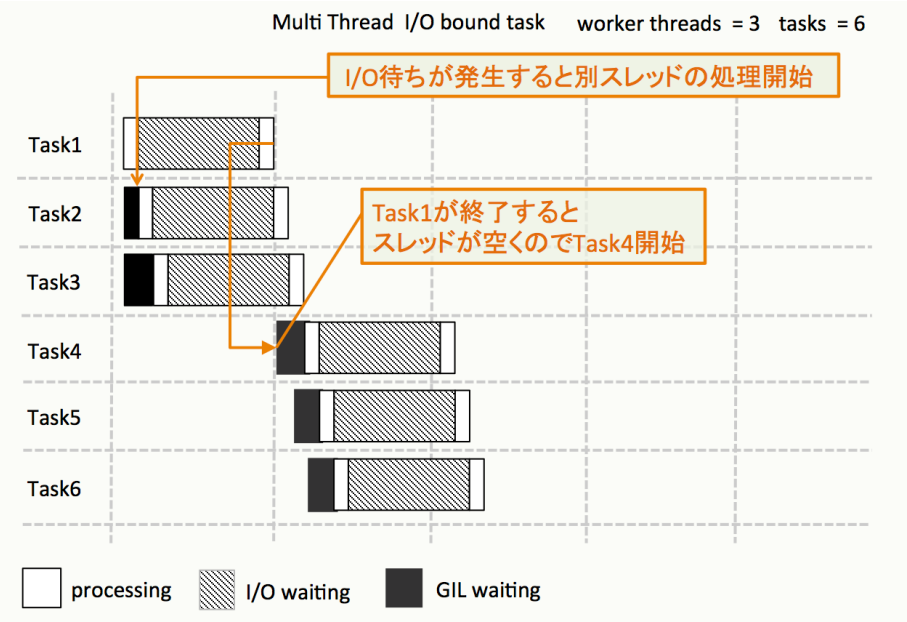

Figure 1: マルチスレッド I/O バウンド

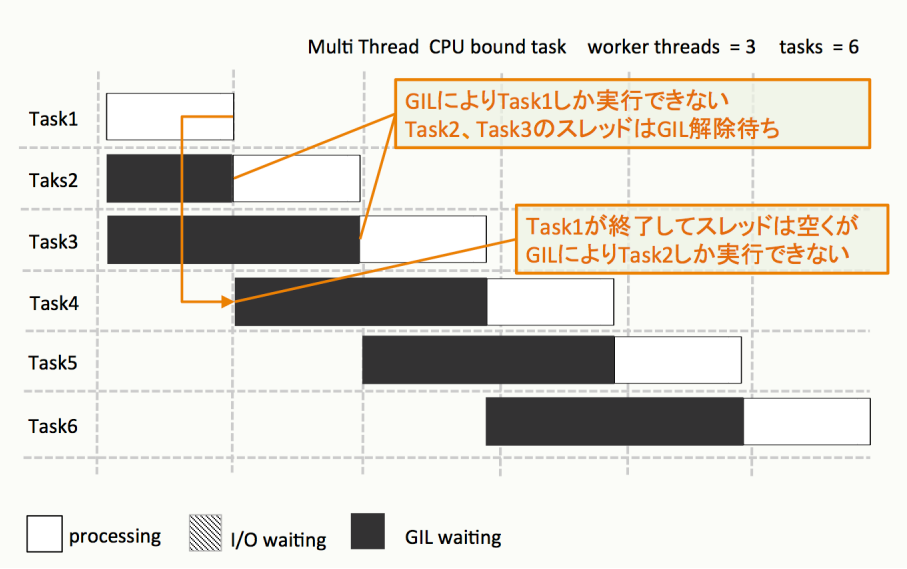

Figure 2: マルチスレッド CPU バウンド

4.2 async と await

async と await がどのような処理を行うのか以下のコード 1 を実行して動作を確認した。async$_a wait\Gamma async\Gamma\Gamma\Gamma\Gamma\Gamma\Gamma\Gamma await\Gamma\Gamma\Gamma\Gamma$

コード 1: 非同期処理の FizzBuzz

```
# −*− coding: utf−8 −*−
# from asyncio import Queue
# from queue import Queue
import asyncio

def fizzbuzz(i):
    if i == 15:
        return 'FizzBuzz'
    if i % 5 == 0:
        return 'Buzz'
    if i % 3 == 0:
        return 'Fizz'
    return str(i)

async def task_fizzbuzz(prefix):
    for x in range(1, 31):
        await asyncio.sleep(0.3)
        print(prefix + "{}:".format(str(x)) + fizzbuzz(x))
    return None

loop = asyncio.get_event_loop()
# コルーチン個生成 1000
tasks = asyncio.wait([task_fizzbuzz(str(i) + ":") for i in range(1, 1000)])
loop.run_until_complete(tasks)
loop.close()
```

コード 2: FizzBuzz 実行結果

```
9:12:Fizz
4:12:Fizz
5:12:Fizz
2:12:Fizz
6:12:Fizz
3:12:Fizz
7:12:Fizz
1:12:Fizz
8:12:Fizz
9:13:13
4:13:13
5:13:13
2:13:13
6:13:13
3:13:13
7:13:13
1:13:13
8:13:13
9:14:14
4:14:14
5:14:14
2:14:14
6:14:14
3:14:14
7:14:14
1:14:14
8:14:14
9:15:FizzBuzz
4:15:FizzBuzz
```

コード 1 は、30 まで FizzBuzz を 10 回行う処理を非同期処理で実装している。

実行結果から FizzBuzz 関数を呼び出す際に sleep していると、コンテキストスイッチと呼ばれる、実行できるプログラムを先に実行する処理を行うのが確認できる。

5　実験方法

HTTP の GET を用いて実験を行う。GET を複数回実行する場合を考えると、逐次処理の場合ではレスポンスがあるまで次の GET を送信することができない。async-await または ThreadPoolExecutor を用いることで、レスポンスを待つことなく次の GET を実行することで効率よく結果を受け取ることができると想定した。

それぞれ逐次処理、async-await(20 回渡し)、async-await のみ (5 回ずつ)、async-await + ThreadPoolExecutor に分けて琉球大学情報工学科 HP への GET を 20 回連続で行うコードを記述した。

また、上記の順でコード 3、コード 4、コード 5、コード 6 に示したとおりである。

コード 3: 逐次処理

```
# Example 1: synchronous requests
import requests
import time

start = time.time()

num_requests = 20

def http_get():
    requests.get('https://ie.u-ryukyu.ac.jp/')
    print(time.time() - start)

responses = [
    http_get()
    for i in range(num_requests)
]
```

コード 4: async-await のみ (20 個渡し)

```
# Example 2: asynchronous requests
import asyncio
import requests
import time

start = 0

async def main():
    start = time.time()

    loop = asyncio.get_event_loop()
    futures = [
        loop.run_in_executor(
            None,
            requests.get,
            'https://ie.u-ryukyu.ac.jp/'
        )
        for i in range(20)
    ]
    for response in await asyncio.gather(*futures):
        print(time.time() - start)
        pass

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
```

コード 5: async-await のみ (4 回ずつ)

```
# Example 2: asynchronous requests
import asyncio
import requests
import time

start = 0

async def main():
    start = time.time()

    for num in range(4):
        loop = asyncio.get_event_loop()
        futures = [
            loop.run_in_executor(
                None,
                requests.get,
                'https://ie.u-ryukyu.ac.jp/'
            )
            for i in range(5)
        ]
        for response in await asyncio.gather(*futures):
            print(time.time() - start)
            pass

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
```

コード 6: async-await + ThreadPoolExecutor

```
# Example 3: asynchronous requests with larger thread pool
import asyncio
import concurrent.futures
import requests
import time

start = 0

async def main():
    start = time.time()

    with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:

        loop = asyncio.get_event_loop()
        futures = [
            loop.run_in_executor(
                executor,
                requests.get,
                'https://ie.u-ryukyu.ac.jp/'
            )
            for i in range(20)
        ]
        for response in await asyncio.gather(*futures):
            print(time.time() - start)
            pass


loop = asyncio.get_event_loop()
loop.run_until_complete(main())
```

6　実行結果

実行結果を Figure3 に示した。Figure 3 を見ると、逐次処理、async-await のみ (4 回ずつ)、async-await のみ (20 個渡し)、async-await + ThreadPoolExecutor の順で時間がかかっていることがわかる。

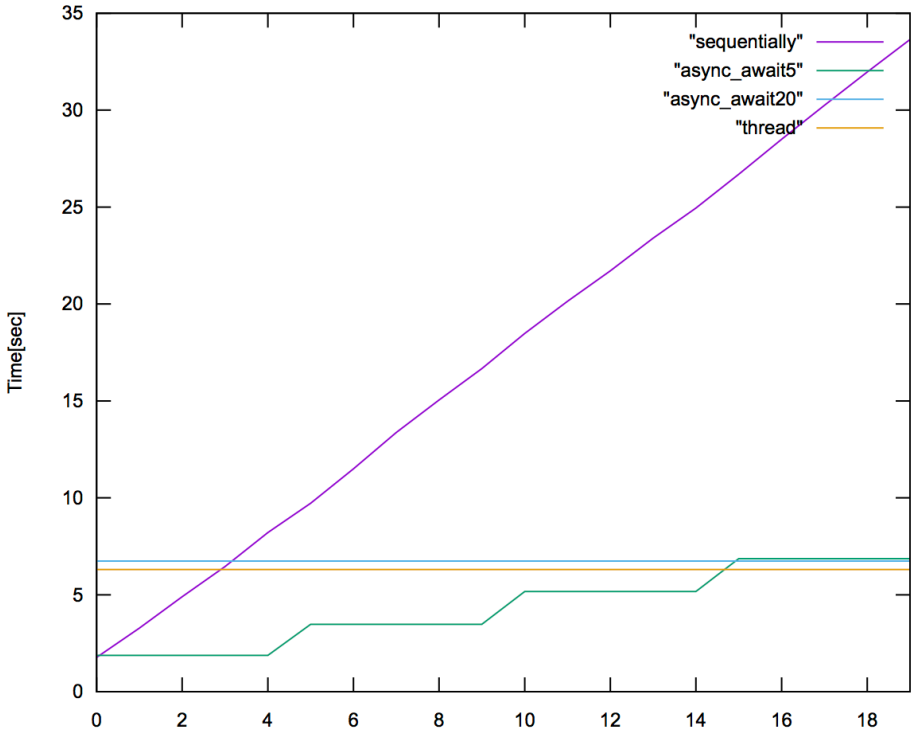

Figure 3: 実行結果

また、実験として Figure: 4 に示したように、async-await での Thread 数を 5 と 20 で比較してみた。Figure: 4 から見て取れるように thread 数が 4 倍になると 2 倍実行速度が早くなっていることがわかる。やはり Thread 数が多いと並列度もあがるようだが、thread 数を 4 倍にして 2 倍の実行速度であれば、生成コストも大きいように見える。

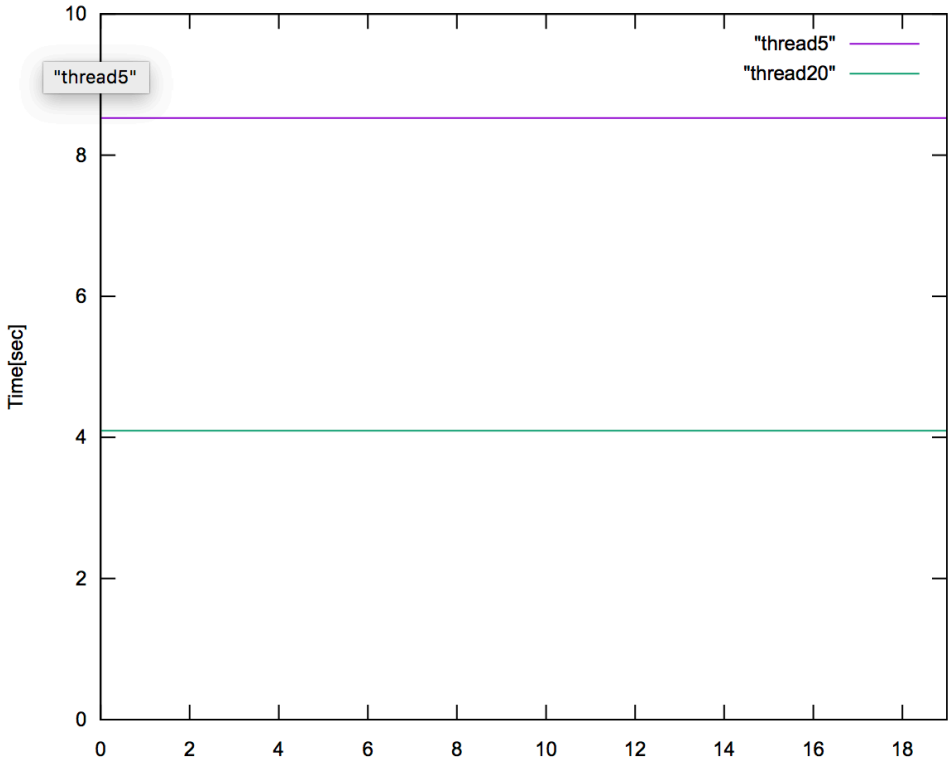

Figure 4: Thread 数比較

7 考察

async-await を用いた非同期処理は、コルーチンを呼び出して行っているので高速であることがグラフから読み取れた。また、async-await でプログラムを 4 回に分けて渡したとしても実行時間が変わらなかったことから、async-await は動的にタスクを増やしても実行時間に影響が出にくいことが考えられる。

async-await + ThreadPoolExecutor は少しだけ実行速度が早くなったことから、Thread の数が増えて並列度が増したと考えられる。

つまり、async-await を用いてプログラムの並行に走らせ、Thread で並列に実行させることでより並列度が高い処理を実装することが今回の実行結果から読み取れる。しかし、Thread 数を 2 とすると実行速度が下がったことから Thread の生成コストが並列度悪くする場合もあるので、使う際には Thread 数に気をつける必要がある。

また、コードには記載してないが async-await を用いずに ThreadPoolExector のみで実行した場合の速度比較を行いたかったが、参考文献 [2] より ThreadPoolExector + Requests だとデッドロックが起こり実行されないということがわかった。そのため速度比較の例としては取り上げていない。

ただ async-await を使うことによって、メモリのデッドロックが起こらないことがわかった。原因としてはコルーチンで Thread を呼び出す動作がメモリが予め確保される状態になるので、request を投げる際にデッドロックを回避できるのではないかと推測した。request と ThreadPoolExector のデッドロックの原因をソースを見て確かめたわけではないのであくまで推測でしかないが、機会があるときにまた詳しく調べたいと思う。

8　感想・意見

b3 前期が忙しかったために凝った内容にできなかった。Python での CPU バウンドでの並行処理はあまり使われているイメージが無く、あったとしてもライブラリで完結しているものが多いので CPU バウンドで Python を取り上げるのがふさわしいのかわからないと感じた。

しかし、GPU を用いた機械学習や画像処理は Python でもだいぶ重要になってくるので率先してやりたかったが、期間内に 0 から学ぶことを考えると手が出しづらかった。

9　GitHub の URL

https://github.com/e165719/ParallelDistributedProcessing

References

[1] Python をとりまく並行/非同期の話, https://tell-k.github.io/pyconjp2017/

[2] ThreadPoolExecutor + Requests == deadlock?, https://stackoverflow.com/questions/40891497/threadpoolexecutor-requests-deadlock