並列分散処理 最終レポート

チーム E　大城 慶知　眞榮城 隆守　伊波卓浩　宮良友也

July 22, 2018

最終報告書に載せるやつ (あとで消すやつ)

演習の背景、目的、方法、結果、考察を A410 ページ以内で適切にまとめる。個々のメンバーの役割分担を明記する。記載がない場合、あるいは、曖昧な場合には、減点の対象となる。例えば、あるタスクを複数名で担当した場合でも、個々のメンバーの役割をできる限り区別して説明する。最終報告書にはソースコードの github リポジトリも記載する。

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 のスレッドと concurrent を用いて並列化を行うとともに、async await の使うことで非同期処理を行いさらに速度向上を目指す。

3　メンバーの担当

大城 慶知 素早いおじさん
眞榮城 隆守 Python ファイルの記述。グラフの生成
伊波卓浩 FizzBuzz のデモ作成
宮良友也 カービィー

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。

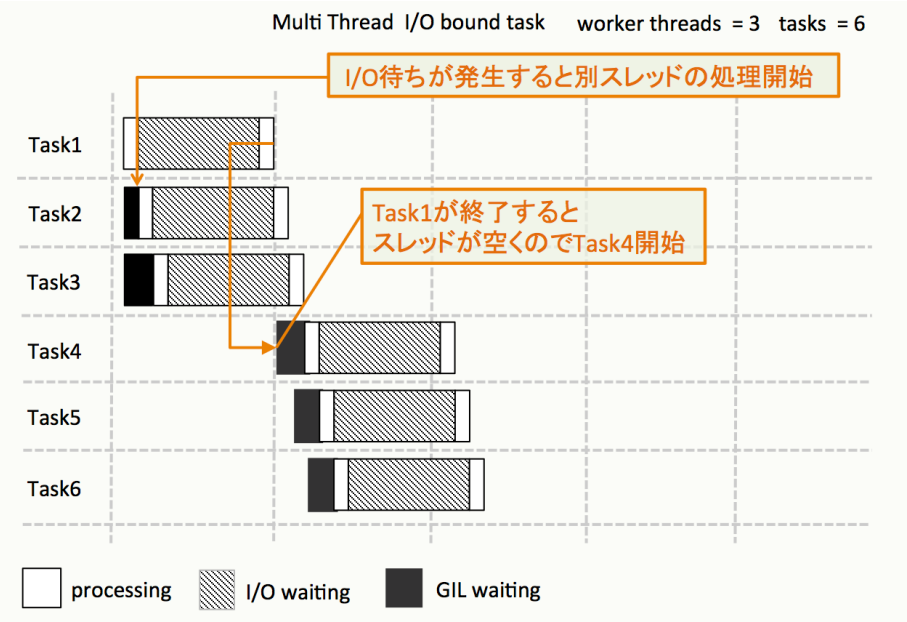

Figure 1: マルチスレッド I/O バウンド

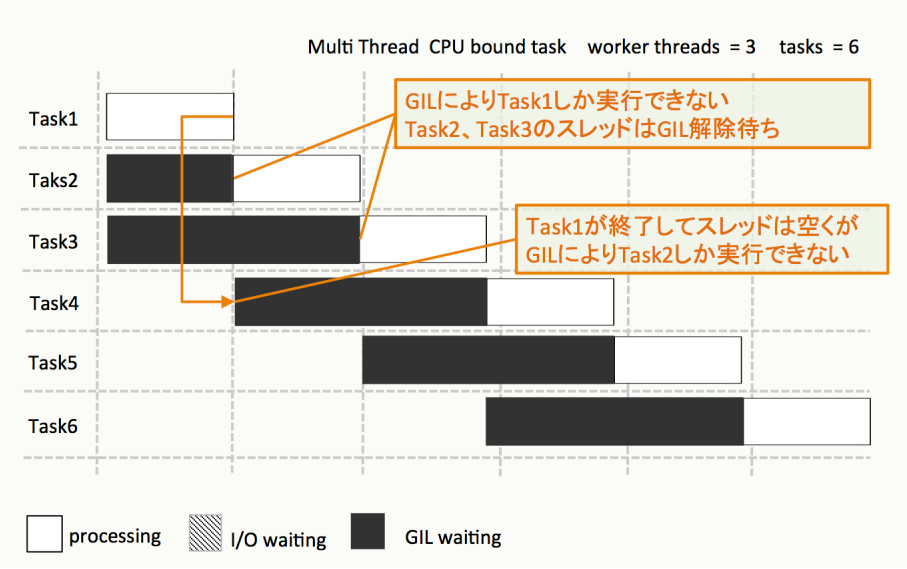

Figure 2: マルチスレッド CPU バウンド

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を実行して動作を確認した。

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

```
75219:6:Fizz
8282:6:Fizz
57464:6:Fizz
75220:6:Fizz
8283:6:Fizz
57465:6:Fizz
75221:6:Fizz
8284:6:Fizz
57466:6:Fizz
75222:6:Fizz
8285:6:Fizz
57467:6:Fizz
75223:6:Fizz
```

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からだと考えられる。ただ実行は CPU バウンドなので 1 スレッドしか利用できずに、速度としてはかなり遅かった。

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への GET を 20 回連続で行った。

また、上記の順でコード 1 3、コード 2 4、コード 3 5、コード 4 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=20) 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。

コード 7: example1.py

```
1.7703397274017334
3.2814269065856934
4.895288705825806
6.453948736190796
8.205788850784302
9.720278978347778
11.497416973114014
13.36806869506836
15.045061826705933
16.66911792755127
18.479264974594116
20.14133882522583
21.714859008789062
23.392934799194336
24.951546907424927
26.690969705581665
28.501687049865723
30.250661849975586
31.964433908462524
33.642555713653564
```

コード 8: example2.py

```
6.742665767669678
6.7426910400390625
6.742694139480591
6.742694854736328
6.742696762084961
6.7426979541778564
6.742699861526489
6.742701053619385
6.742702960968018
6.742703914642334
6.742705821990967
6.742706775665283
6.742708921432495
6.7427098751068115
6.742712020874023
6.74271297454834
6.742713928222656
6.742715835571289
6.742717742919922
6.742718935012817
```

コード 9: example3.py

```
1.8773348331451416
1.877375841140747
1.8773789405822754
1.8773798942565918
1.8773820400238037
3.4774909019470215
3.477499008178711
3.477501153945923
3.4775028228759766
3.477504014968872
5.168514013290405
5.16852593421936
5.168529033660889
5.1685309410095215
```

```
5.168533802032471
6.861561059951782
6.861573934555054
6.861576795578003
6.861578941345215
6.861582040786743
```

コード 10: example4.py

```
6.301663875579834
6.301697015762329
6.301699161529541
6.30170202255249
6.301703214645386
6.30170048835754395
6.301707029342651
6.301707983016968
6.30171012878418
6.30171201361328125
6.301713228225708
6.301714897155762
6.301717042922974
6.30171799659729
6.301720142364502
6.301722049713135
6.301723003387451
6.301724910736084
6.301727056503296
6.301728010177612
```

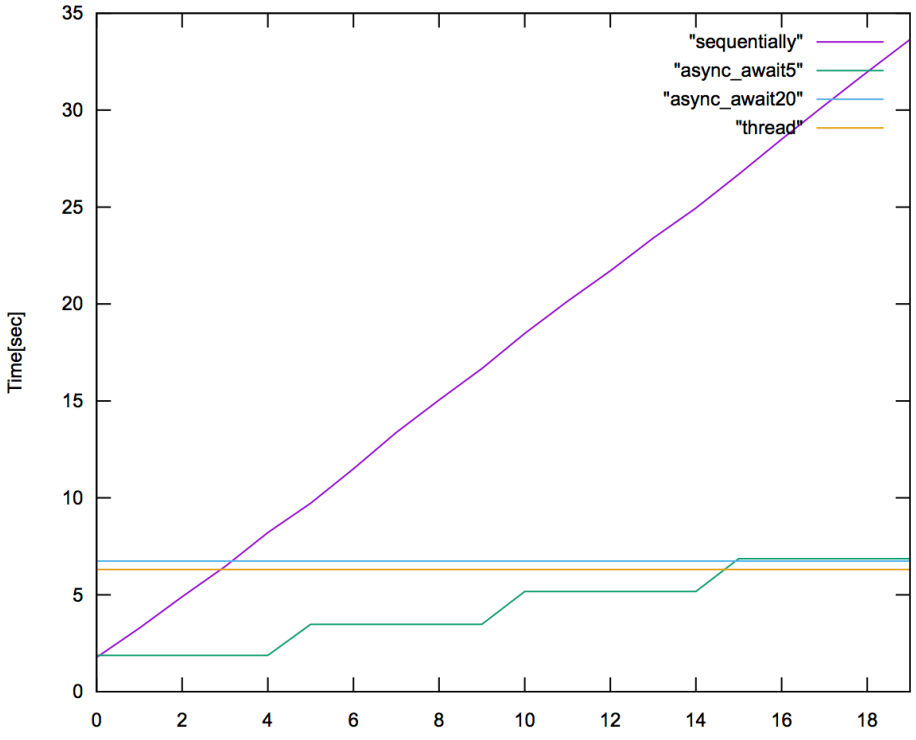

Figure 3: 実行結果

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。

8　感想・意見

9　GitHub の URL

https://github.com/e165719/ParallelDistributedProcessing