並列分散処理　最終課題

185405E 浅香幸佑

2021/08/07

1　目的

逐次処理と並列処理の比較を通して並列プログラムの効果を検証する．

2　方法

開発，実行は python で行なった．並列処理についての理解を深めるために大きな数値に対して fizz_buzz を行う CPU バウンドな処理のサンプルプログラムを用意した．

fizz_buzz とは 1 から順番に数字を言って，3 の倍数では「Fizz」，5 の倍数では「Buzz」，3 かつ 5 の倍数の場合「Fizz Buzz」を数の代わりに発言するという英語圏で行われるゲームである．

リストの中の 5 つの大きな値について 1 つずつ逐次的に 1 からその数値まで fizz_buzz していき，最後にどのくらい時間がかかったのかを time を使って出力する．

github の URL

ソースコード 1　serial_fizbuz

```
1  import time
2
3  def fizz_buzz(num: int):
4      result_list = []
5      for i in range(1, num + 1):
6          result = ''
7          if i % 3 == 0:
8              result += 'fizz'
9          if i % 5 == 0:
10             result += 'buzz'
11         if not result:
12             result = str(i)
13         result_list.append(result)
14     return result_list
15
16
17 start = time.time()
18 num_list = [12000000, 38000000, 28000000, 2520000, 22850000]
19 for n in num_list:
20     fizz_buzz(n)
21 stop = time.time()
22 print(f'Sequential␣processing:␣{stop␣-␣start:.3f}␣seconds')
```

次にマルチスレッドを実現するために threading を用いたプログラムを用意した．

ソースコード 2　multithread_fizbuz

```
1  import threading
2  import time
3
4  class MyThread(threading.Thread):
5      def __init__(self, num):
```

```
6           super().__init__()
7           self.__num = num
8
9       def fizz_buzz(self, num: int):
10          result_list = []
11          for i in range(1, num + 1):
12              result = ''
13              if i % 3 == 0:
14                  result += 'fizz'
15              if i % 5 == 0:
16                  result += 'buzz'
17              if not result:
18                  result = str(i)
19              result_list.append(result)
20          return result_list
21
22      def run(self):
23          self.fizz_buzz(self.__num)
24
25
26  start = time.time()
27  threads = []
28  num_list = [12000000, 38000000, 28000000, 2520000, 22850000]
29  for n in num_list:
30      thread = MyThread(n)
31      thread.start()
32      threads.append(thread)
33  for th in threads:
34      th.join()
35  stop = time.time()
36  print(f'multi␣threads:␣{stop␣-␣start:.3f}␣seconds')
```

最後にマルチプロセスを実現するために multiprocessing を用いたプログラムを用意した.

ソースコード 3　multiprocess_fizbuz

```
1   from multiprocessing import Process
2   import time
3
4   class MyProcessor(Process):
5
6       def __init__(self, num):
7           super().__init__()
8           self.__num = num
9
10      def fizz_buzz(self, num: int):
11          result_list = []
12          for i in range(1, num + 1):
13              result = ''
14              if i % 3 == 0:
15                  result += 'fizz'
16              if i % 5 == 0:
17                  result += 'buzz'
18              if not result:
```

```
19                      result = str(i)
20                  result_list.append(result)
21              return result_list
22
23          def run(self):
24              self.fizz_buzz(self.__num)
25
26
27  start = time.time()
28  processes = []
29  num_list = [12000000, 38000000, 28000000, 2520000, 22850000]
30  for n in num_list:
31      process = MyProcessor(n)
32      process.start()
33      processes.append(process)
34  for p in processes:
35      p.join()
36  stop = time.time()
37  print(f'multi␣process:␣{stop␣-␣start:.3f}␣seconds')
```

3 考察

まず逐次処理を行うサンプルプログラムを実行した結果，処理時間は 31.307s となった．

次にマルチスレッドを用いたプログラムでは処理時間が 32.206s となった．

最後にマルチプロセスを用いたプログラムでは処理時間が 18.698s となった．

マルチスレッドを用いたプログラムの方が処理時間が長くなってしまった理由として，マルチスレッドで処理を行う場合基本的にスレッドセーフである必要がある．スレッドセーフとは複数のスレッドを並列的に使用しても問題が発生しないことを意味する．

しかし python はスレッドセーフではない．そこで python は GIL(Global Interpreter lock) という排他ロックを使用することでこの問題を回避している．GIL により python のプロセスは 1 スレッドしか実行できない．

つまり 1 つのプロセスに複数スレッドが存在してもロックを持つ単一スレッドでしかコードが実行できずに，その他のスレッドは待機状態になる．

そのため python におけるマルチスレッドでは並列処理ではなく並列処理になってしまうため，CPU バウンドな処理ではマルチスレッドでは高速化しないという現象が起きてしまった．

対してマルチプロセスには GIL 制約などは存在しないので，並行処理ではなく並列処理となって処理時間が短くなったと考えられる．

4

参考文献

[1] Python における並行・並列処理について調べてみた https://okiyasi.hatenablog.com/entry/2020/09/22/030505 閲覧日:2021/08/02

[2] threading — スレッドベースの並列処理 https://docs.python.org/ja/3/library/threading.html 閲覧日:2021/08/02