Государственное образовательное учреждение
высшего профессионального образования
«Московский Государственный Технический Университет
имени Н.Э. Баумана»

Отчет
По лабораторной работе №3
По курсу «Анализ Алгоритмов»
На тему «Конвейер»

Щербатюк Дарья, ИУ7-54

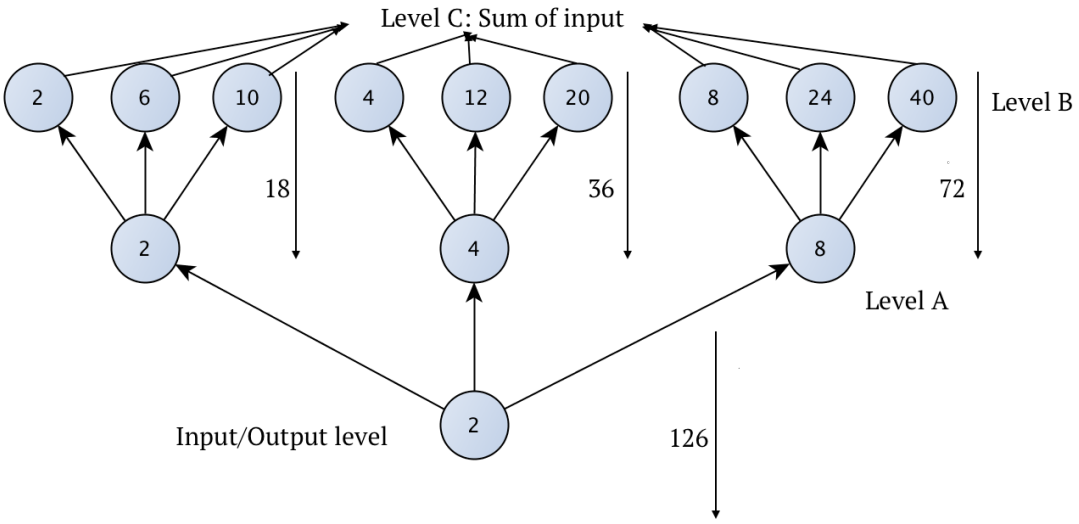
Москва, 2017

Оглавление

Постановка задачи

Реализовать алгоритм работы конвейер. Использовать методы паралеллизации.

Схема конвейера



Описание алгоритма

На вход алгоритм получает одно число. Начальная функция пробрасывает число на уровень A.

Уровень A: с помощью входного числа генерируются соответсвующие числа для уровня B и пробрасываются в него. При этом уровень A начинает ожидать результата от уровня B. После его получения числа результата суммируются.

Уровень B: аналогичен уровню A. Пробрасывает результат в C.

Уровень C: отправляет числа в функцию суммированию и возвращает результат в Уровень B.

На выходе получаем сумму чисел, дошедших до уровня C. Все действия выполяются асинхронно. Функция суммирования работает в 9 потоков.

Листинг

CONVEYOR.PY:

```
1  import asyncio
2  import logging
3  import time
4  from concurrent.futures import ThreadPoolExecutor
5
6  logging.basicConfig(format="[%(thread)-5d]%(asctime)s: %(
       message)s")
7  logger = logging.getLogger('async')
8  logger.setLevel(logging.INFO)
9
10 executor = ThreadPoolExecutor(max_workers=9)  # thread pool
11 loop = asyncio.get_event_loop()  # event loop
12
13
14 def cpu_bound_op(exec_time, *data):    # fake long-running
       func
15         logger.info("Running cpu-bound op on {} for {}
               seconds".format(data, exec_time))
16         time.sleep(exec_time)
17         return sum(data)
18
19
20 async def process_pipeline(data):
21         # just pass the data along to level_a and return the
               results
22         results = await level_a(data)  # Waiting for the
               level a
23         return results
24
25
26 async def level_a(data):
27         level_b_inputs = data, 2 * data, 4 * data
28         results = await asyncio.gather(*[level_b(val) for
               val in level_b_inputs])  # aggregate results from
                the level b
29         result = await loop.run_in_executor(executor,
               cpu_bound_op, 3, *results)
30         return result
31
32
33 async def level_b(data):
34         # similar to level a
35         level_c_inputs = data, 3 * data, 5 * data
36         results = await asyncio.gather(*[level_c(val) for
               val in level_c_inputs])
37         result = await loop.run_in_executor(executor,
               cpu_bound_op, 2, *results)
```

```
38                 return result
39
40
41         async def level_c(data):
42                 result = await loop.run_in_executor(executor,
                       cpu_bound_op, 1, data)
43                 return result
44
45
46         def main():
47                 start_time = time.time()
48                 result = loop.run_until_complete(process_pipeline(2)
                       )
49                 logger.info("Completed ({}) in {} seconds".format(
                       result, time.time() - start_time))
50
51
52         if __name__ == '__main__':
53                 main()
```

TEST.PY:

```
1          start_time = time.time()
2          start_clock = time.clock()
3
4          inpt = 2
5
6          lvl_a = inpt, inpt * 2, inpt * 4
7
8          logger.info("Running cpu-bound op on {}".format(lvl_a))
9          lvl_b_0 = lvl_a[0], lvl_a[0] * 3, lvl_a[0] * 5
10         logger.info("Running cpu-bound op on {}".format(lvl_b_0))
11         lvl_b_1 = lvl_a[1], lvl_a[1] * 3, lvl_a[1] * 5
12         logger.info("Running cpu-bound op on {}".format(lvl_b_1))
13         lvl_b_2 = lvl_a[2], lvl_a[2] * 3, lvl_a[2] * 5
14         logger.info("Running cpu-bound op on {}".format(lvl_b_2))
15
16         lvl_c = lvl_b_0
17
18         branch_0 = sum(lvl_c)
19
20         lvl_c = lvl_b_1
21
22         branch_1 = sum(lvl_c)
23
24         lvl_c = lvl_b_2
25
26         branch_2 = sum(lvl_c)
27
28         time.sleep(3 + 2 * 3 + 1 * 9)
29
30         result = sum((branch_0, branch_1, branch_2))
```

```
31          logger.info("Running cpu-bound op on {}".format(branch_0,
                branch_1, branch_2))
32
33          logger.info("Completed ({}) in {} seconds and {} cpu-time".
                format(result, time.time() - start_time, time.clock() -
                start_clock))
```

Выводы

В результате проведенных испытаний алгоритма было установлено, что:

1. Наилучший результат достигается при работе в 9ти потоках, то есть число потоков должно быть равно числу входящих чисел на уровень C.

Число потоков	Время на входном числе data = 2	CPU-время
1	18.050 sec	0.0111 sec
2	11.020 sec	0.0105 sec
3	8.025 sec	0.0091 sec
4	8.020 sec	0.0096 sec
5	7.019 sec	0.0076 sec
6	7.022 sec	0.00088 sec
7	7.019 sec	0.0094 sec
8	7.019 sec	0.0092 sec
9	6.011 sec	0.0078 sec
10	6.013 sec	0.0086 sec
11	6.015 sec	0.0091 sec
12	6.014 sec	0.0089 sec

2. Если сделать похожую не асинхронную программу(test.py), игнорируя время вызова функций и не используя асинхронное программирование, то можно увидеть, что реальное время, затраченное на выполнение задачи превышает работу асинхронной программы в 3 раза. Однако время, затраченное программой на процессоре меньше на порядок. Вттаблице ниже через черту представлено время выполнения программы и cpu-время.

Входное число	Асиннхронно	Не асинхронно
2	6.015 sec | 0.00812 sec	18.006 sec | 0.00069 sec
100	6.016 sec | 0.00817 sec	18.005 sec | 0.00071 sec
10 000	6.015 sec | 0.00896 sec	18.004 sec | 0.00068 sec
10 000 000	6.016 sec | 0.00776 sec	18.003 sec | 0.00073 sec

3. Из выше приведенной таблицы видно, что время выполнение обеих программ не зависит от размера входного параметра.

Заключение

В ходе лабораторной работы было изучено асинхронное программирование с помощью встроенной библиотеки Asyncio языка Python.