Laboratorul 11: Utilizarea bibliotecilor Python pentru paralelism

Introducere

Pentru o imagine de ansamblu asupra paralelismului în Python, se recomandă parcurgerea următoarelor resurse:

- **„Learning concurrency in python“ - Elliot Forbes**
- **„Python parallel programming cookbook“ - Giancarlo Zaccone**
- Laboratorul 5 de la disciplina *Paradigme de programare* (pentru comunicarea inter-proces)
- Cursul 11 de la disciplina *Paradigme de programare*
- https://docs.python.org/3/library/concurrency.html

De asemenea, consultați documentația oficială pentru modulele:

- multiprocessing: https://docs.python.org/3/library/multiprocessing.html
- asyncio: https://docs.python.org/3/library/asyncio.html
- threading: https://docs.python.org/3/library/threading.html
- concurrent.futures: https://docs.python.org/3/library/concurrent.futures.html
- subprocess: https://docs.python.org/3/library/subprocess.html

Exemple

Exemplul 1

Pentru început, se va porni de la exemplul din curs utilizat pentru analiza comparativă de performanțe pentru diverse biblioteci utilizate în cadrul exemplelor.

```
import threading
import multiprocessing
from concurrent.futures import ThreadPoolExecutor
import time


def countdown():
    x = 100000000
    while x > 0:
        x -= 1


def ver_1():
    thread_1 = threading.Thread(target=countdown)
    thread_2 = threading.Thread(target=countdown)
    thread_1.start()
    thread_2.start()
    thread_1.join()
    thread_2.join()


def ver_2():
    countdown()
    countdown()
```

```
def ver_3():
    process_1 = multiprocessing.Process(target=countdown)
    process_2 = multiprocessing.Process(target=countdown)
    process_1.start()
    process_2.start()
    process_1.join()
    process_2.join()


def ver_4():
    with ThreadPoolExecutor(max_workers=2) as executor:
        future = executor.submit(countdown())
        future = executor.submit(countdown())


if __name__ == '__main__':
    start = time.time()
    ver_1()
    end = time.time()
    print("\n Timp executie pseudoparalelism cu GIL")
    print(end - start)
    start = time.time()
    ver_2()
    end = time.time()
    print("\n Timp executie secvential")
    print(end - start)
    start = time.time()
    ver_3()
    end = time.time()
    print("\n Timp executie paralela cu multiprocessing")
    print(end - start)
    start = time.time()
    ver_4()
    end = time.time()
    print("\n Timp executie paralela cu concurrent.futures")
    print(end - start)
```

Exemplul 2

Se pornește de la exemplul de fir cu condiție din curs care este prezentat mai jos:

```
from threading import Thread, Condition
import time
elemente = []
conditie = Condition()


class Consumator(Thread):
    def __init__(self):
        Thread.__init__(self)

    def consumator(self):
        global conditie
        global elemente
        conditie.acquire()
        if len(elemente) == 0:
```

```
            conditie.wait()
            print('mesaj de la consumator: nu am nimic disponibil')
        elemente.pop()
        print('mesaj de la consumator : am utlizat un element')
        print('mesaj de la consumator : mai am disponibil',
              len(elemente),
              'elemente')
        conditie.notify()
        conditie.release()

    def run(self):
        for i in range(5):
            self.consumator()

class Producator(Thread):
    def __init__(self):
        Thread.__init__(self)

    def producator(self):
        global conditie
        global elemente
        conditie.acquire()
        if len(elemente) == 10:
            conditie.wait()
            print('mesaj de la producator : am disponibile',
                  len(elemente),
                  'elemente')
            print('mesaj de la producator : am oprit productia')
        elemente.append(1)
        print('mesaj de la producator : am produs',
              len(elemente),
              'elemente')
        conditie.notify()
        conditie.release()

    def run(self):
        for i in range(5):
            self.producator()

if __name__ == '__main__':
    producator = Producator()
    consumator = Consumator()
    producator.start()
    consumator.start()
    producator.join()
    consumator.join()
```

Exemplul 3

Exemplu de utilizare a unui proces în sub clasă:

```
import multiprocessing


class ProcesTest(multiprocessing.Process):
    def run(self):
```

```
        print(f'am apelat metoda run() in procesul: {self.name}')
        return

if __name__ == '__main__':
    jobs = []
    for i in range(5):
        p = ProcesTest()
        jobs.append(p)
        p.start()
        p.join()
```

Exemplul 4: Task asyncio

```
import asyncio


async def factorial(name, number):
    f = 1
    for i in range(2, number + 1):
        print(f"Task {name}: Compute factorial({i})...")
        await asyncio.sleep(1)
        f *= i
    print(f"Task {name}: factorial({number}) = {f}")


async def main():
    # Schedule three calls *concurrently*:
    await asyncio.gather(
        factorial("A", 2),
        factorial("B", 3),
        factorial("C", 4),
    )


if __name__ == '__main__':
    asyncio.run(main())
```

Exemplul 5: Buclă de evenimente cu asyncio

```
import asyncio

async def print_number(number):
    print(number)


if __name__ == "__main__":
    loop = asyncio.get_event_loop()

    loop.run_until_complete(
        asyncio.wait([
            print_number(number)
            for number in range(10)
        ])
    )
```

Exemplul 6: Analiză comparativă

```
import multiprocessing
import multiprocessing.pool
from multiprocessing import cpu_count
from concurrent.futures import ThreadPoolExecutor, ProcessPoolExecutor
import math
import time

def time_it(func):
    def wrapper(*args, **kwargs):
        begin = time.time()
        output = func(*args, **kwargs)
        end = time.time()
        print(func.__name__, "executed in", end - begin, "sec")
        return output
    return wrapper

def factorial0(n):
    result = 1
    for num in range(2, n + 1):
        result *= num
    return result

def factorial1(n):
    return math.factorial(n)


@time_it
def test_thread_pool_executor(function, list_of_integers):
    with ThreadPoolExecutor(cpu_count()) as pool:
        return pool.map(function, list_of_integers)


@time_it
def test_process_pool_executor(function, list_of_integers):
    with ProcessPoolExecutor(cpu_count()) as pool:
        return pool.map(function, list_of_integers)

@time_it
def test_pool0(function, list_of_integers):
    with multiprocessing.pool.Pool(cpu_count()) as pool:
        return pool.map(function, list_of_integers)

@time_it
def test_pool1(function, list_of_integers):
    with multiprocessing.Pool(cpu_count()) as pool:
        return pool.map(function, list_of_integers)

@time_it
def test_thread_pool(function, list_of_integers):
    with multiprocessing.pool.ThreadPool(cpu_count()) as pool:
        return pool.map(function, list_of_integers)


if __name__ == '__main__':
```

```
    list_of_integers = list(range(5, 2000))
    tests = [test_thread_pool_executor, test_process_pool_executor,
             test_pool0, test_pool1, test_thread_pool]
    for function in [factorial0, factorial1]:
        print(function.__name__)
        for test in tests:
            test(function, list_of_integers)
        print()
```

Aplicații și teme

Aplicații de laborator:

- Să se modifice și să se testeze exemplul 1 după cum urmează:
    1. Se va înlocui în funcția *countdown* bucla while cu:
        - sortarea unui vector de x elemente generate aleator (vectorul va fi generat global, toate apelurile funcției *countdown* folosind același vector), creând o copie sortată a vectorului inițial și se va relua măsurarea;
        - filtrarea numerelor prime dintr-un ***set*** generat aleator.
    2. Se vor modifica toate funcțiile de test (*ver_1, ver_2, ver_3, ver_4*) astfel încât să se primească ca parametri colecțiile de numere generate aleator și să se execute operații paralele reale (filtrare numere prime, ridicarea la pătrat a elementelor unei colecții), adică colecțiile vor fi segmentate, iar segmentele vor fi trimise la câte un thread/proces (fiecare thread/proces operează doar pe segmentul primit)
    3. Se va crea o funcție care incrementează cu 1 fiecare valoare din colecția de numere primită ca parametru (o listă cu x zerouri). Se vor modifica funcțiile de test astfel încât fiecare thread/proces să apeleze funcția de incrementare. Apoi, se vor adăuga operațiile necesare pentru a garanta rezolvarea coerenței datelor și se va relua măsurarea.

- Să se testeze și să se modifice exemplul 2 astfel încât să se elimine nevoia de a utiliza cele două variabile globale (vezi comunicarea între thread-uri).

- Să se testeze exemplul 3, apoi să se realizeze o procesare după modelul pipeline a unui ADT de numere întregi. Primul thread din pipe înmulțește toate elementele din vectorul V cu o constantă *alpha*, următorul thread din pipe va ordona mulțimea, iar la final ultimul thread o va afișa. Este necesară sincronizarea?

Tema pe acasă:

1. Să se realizeze calculul $\sum_{0}^{n} i$ simultan pentru patru valori diferite ale lui *n* luate dintr-o coadă de către patru corutine diferite (se va utiliza modulul *asyncio*).

2. Să se citească de la tastatură o comandă care folosește pipeline-uri UNIX (spre exemplu: `ip a | grep inet | wc -l`), apoi să se împartă această comandă după pipe-uri (caracterul |) și folosind PIPE-urile din modulul *subprocessing*, output-ul de la comanda din stânga să fie trimis ca input pentru următoarea (se va recrea practic pipeline-ul de procese).

3. Să se implementeze un ThreadPool de la zero care să îndeplinească următoarele cerințe:
    - să fie auto-closeable (adică să poată fi folosit ca un Pool obișnuit într-un bloc *with*)
    - să implementeze funcția *map*, aceasta distribuind automat (load balancing) volumul de lucru (spre exemplu, dacă ThreadPool-ul conține 4 thread-uri și se primește o colecție cu 9 elemente, primul thread va opera pe 3 elemente, iar următoarele thread-uri pe câte 2 elemente)
    - să se implementeze funcțiile de *join* și *terminate*