Przetwarzanie równoległe i mechanizmy synchronizacji

14 grudnia 2021

Paradygmaty Programowania – mat. przygotowawcze do lab. 4.

Wprowadzenie

Obliczenia równoległe jest to sposób wykonywania zadań obliczeniowych, w którym wiele instrukcji jest wykonywanych jednocześnie.

W języku *Python* dostępne są następujące mechanizmy, pozwalające na równoległe wykonywanie zadań:

1. wątki – moduł **threading**,
2. procesy – moduł **multiprocessing**,
3. obliczenia rozproszone – np: moduł **ipyparallel**,
4. programowanie asynchroniczne – moduł **asyncio** (wykonywanie, koordynacja oraz przełączanie zadań dokonuje się w ramach pojedynczego procesu).

Na zajęciach laboratoryjnych będą Państwo wykorzystywali tylko trzy pierwsze mechanizmy.

Wątki

Moduł **threading** (dokumentacja) – dostarcza definicję klasy `Thread`, która zapewnia pełną funkcjonalność dla pojedynczego wątku.

Aby móc korzystać z tej funkcjonalności, należy w pierwszym kroku zaimportować do skryptu odpowiednie klasy:

```
from threading import Thread
```

Najprostszą metodą wskazania zadania do wykonania w ramach wątku jest:

- zdefiniowanie funkcji, która będzie wykonana w ramach wątku,
- przekazanie tej funkcji jako argumentu do konstruktora tworzonego obiektu nowego wątku.

Zdefiniujmy przykładową funkcję:

```
import time
def work(number):
    print ("Thread number:",number)
    current_time = time.time()
```

```
    while (time.time() < current_time+2):
        pass
```

Teraz zdefiniujemy grupę wątków:

```
threads = [Thread(target=work, args=(number,)) for number in range(5)]
```

Spowoduje to utworzenie listy pięciu obiektów typu `Thread`. Każdemu z nich w momencie utworzenia zostanie "przydzielona" do wykonania funkcja `work` oraz lista argumentów tej funkcji – tutaj jednoelementowa, zawierająca numer wątku, czyli kolejną liczbę całkowitą z zakresu [0, 5).

Dodatkowo, żeby zablokować główny wątek wywołujący do momentu zakończenia działania nowego ("roboczego") wątku, należy wywołać na rzecz wątku roboczego metodę `join`. Jest to implementacja mechanizmu synchronizacji znanego jako **bariera**.

```
for thread in threads:
    thread.start()
for thread in threads:
    thread.join()
```

```
Thread number: 0
Thread number: 1
Thread number: 2
Thread number: 3
Thread number: 4
```

> Uwaga:
>
> W najpowszechniej używanym interpreterze Pythona *Cpython*, z powodu istnienia mechanizmu Global Interpreter Lock, tylko jeden wątek może wykonywać Pythonowy `bytecode`. Dlatego też mechanizm ten (w przypadku Pythona) używany jest głównie do zrównoleglania operacji I/O (wejścia/wyjscia).

Ponieważ wątki współdzielą pamięć, w przypadku wykorzystania mechanizmu wątków pojawia się problem synchronizacji.

Poniższy skrypt obrazuje zjawisko tzw **wyścigów**:

- W linii nr. 4 zdefiniowana została zmienna globalna **data** – wspólna dla wszystkich wątków.
- Funkcja **work** wykonuje 100000 razy inkrementację zmiennej globalnej.
- Funkcja **main** zeruje wartość zmiennej globalnej i uruchamia cztery wątki wykonujace zadanie zdefiniowane w funkcji **work**.
- Główne ciało funkcji wykonuje dziesięć razy funkcję **main** i wyświetla końcową wartość zmiennej globalnej **data**.

Tip: Numerację linii kodu w programie *Jupyter Lab* można włączyć, wchodząc w tryb wydawania poleceń za pomocą kombinacji klawiszy `Shift + M`, a następnie wciskając `L`. Powrót do trybu edycji następuje po kliknięciu w komórkę.

```
from threading import Thread

# global variable
data = 0
```

```
def work():
    global data
    for _ in range(100000):
        data += 1

def main():
    global data
    data = 0
    threads = [ Thread(target=work) for _ in range(4) ]
    for thread in threads:
        thread.start()
    for thread in threads:
          thread.join()

if __name__ == "__main__":
    for i in range(10):
        main()
        print("Iteration {0:2}: x = {1}".format(i+1, data))
```

```
Iteration  1: x = 300000
Iteration  2: x = 300000
Iteration  3: x = 400000
Iteration  4: x = 400000
Iteration  5: x = 400000
Iteration  6: x = 400000
Iteration  7: x = 400000
Iteration  8: x = 400000
Iteration  9: x = 400000
Iteration 10: x = 400000
```

Jak widać, końcowa wartość zmiennej globalnej (dla każdej iteracji) nie zawsze równa się oczekiwanej wartości ($4 \times 100000 = 400000$), lecz czasami przyjmuje inne wartości (jeśli efekt ten nie wystąpił, prosze ponownie wykonać powyższy kod). Zjawisko to wynika z braku synchronizacji dostępu do zmiennej globalnej.

Po to, by wyeliminować to zjawisko, należy zastosować jakiś mechanizm synchronizacji. Najprostszym jest tzw. `Lock`.

W kolejnym skrypcie w wierszu nr 5 zdefiniowany został – wspólny dla wszystkich wątków – **obiekt blokady** typu `Lock`. Jest on czymś w rodzaju pałeczki w sztafecie, w której zawodnikami są wątki. Jeśli obiekt blokady jest dostępny, pierwszy wątek, który wywoła jego metodę `acquire`, przejmuje go na wyłączność (wiersz 9). Późniejsze wywołania metody `acquire` przez inne wątki powodują wstrzymanie ich pracy do momentu, w którym obiekt blokady zostanie zwolniony przez posiadający go wątek. Służy do tego metoda `release` (wiersz 11).

Dzięki użyciu obiektu `lock` fragment kodu realizujący inkrementację zmiennej globalnej jest realizowany zawsze tylko przez **jeden** wątek. Proszę zaobserwować, jak użycie synchronizacji wpłynęło na:

- poprawność wyniku,
- czas wykonania całego skryptu.

```python
from threading import Thread, Lock

# global variable
data = 0
lock = Lock()

def work():
    global data
    for _ in range(100000):
        lock.acquire()
        data += 1
        lock.release()

def main():
    global data
    data = 0
    threads = [ Thread(target=work) for _ in range(4) ]
    for thread in threads:
        thread.start()
    for thread in threads:
            thread.join()

if __name__ == "__main__":
    for i in range(10):
        main()
        print("Iteration {0:2}: x = {1}".format(i+1,data))
```

```
Iteration  1: x = 400000
Iteration  2: x = 400000
Iteration  3: x = 400000
Iteration  4: x = 400000
Iteration  5: x = 400000
Iteration  6: x = 400000
Iteration  7: x = 400000
Iteration  8: x = 400000
Iteration  9: x = 400000
Iteration 10: x = 400000
```

Procesy

Obliczenia równoległe można realizować także poprzez uruchamianie wielu procesów. Będą one działały niezależnie od siebie (nie współdzielą obszarów pamięci) i dlatego, aby zapewnić współpracę między nimi, stosuje się komunikację IPC.

W języku Python dostępny jest pakiet **multiprocessing** (dokumentacja), który udostępnia funkcjonalność uruchamiania oddzielnych procesów, a jego *API* jest bardzo podobne do tego używanego w ramach modułu **threading**.

Aby móc korzystać z tej funkcjonalności, należy w pierwszym kroku zaimportować do skryptu

odpowiednie klasy:

```
from multiprocessing import Process
```

Najprostszą metodą wskazania zadania do wykonania w ramach procesu jest:

- zdefiniowanie funkcji, która będzie wykonana w ramach procesu, a następnie
- przekazanie tej funkcji jako argumentu do konstruktora tworzonego obiektu nowego procesu.

Zdefiniujmy przykładową funkcję:

```
import os

def work(name):
    print("Process:",name,"; Process id:",os.getpid(), "; Parent process id:",os.getppid())
```

Będzie ona wyświetlała kolejno: *numer procesu*, jego *process id*, oraz *process id* jego rodzica.

Teraz zdefiniujemy grupę procesów:

```
processes=[Process(target=work, args=(i,)) for i in range(3)]
```

Następnie – podobnie jak w przypadku wątków – dla każdego obiektu procesu należy wywołać metodę **start**. Dodatkowo, żeby zablokować główny wątek wywołujący do momentu zakończenia działania wszystkich potomnych procesów, należy wywołać metodę **join** (dla każdego procesu).

```
for process in processes:
        process.start()
for process in processes:
        process.join()
```

```
Process:Process:  10 ; Process id:Process: 12035  2; Parent process id:   ; Process id:3570 ; P
 1204012034  ; Parent process id:; Parent process id: 3570
 3570
```

Jednym ze sposobów komunikacji międzyprocesowej (IPC) jest wykorzystanie bezpiecznych kolejek **FIFO** lub **LIFO**. Są one bezpieczne w tym sensie, że w danej chwili tylko jeden proces może modyfikować zawartość kolejki. Pakiet *multiprocessing* dostarcza definicję klas `Queue` (FIFO), `LifoQueue` oraz `PriorityQueue`.

Udostępniają one API, które składa się miedzy innnymi z następujących metod:

- `put(obiekt)` – wkłada `obiekt` do kolejki,
- `get()` – usuwa z kolejki i zwraca kolejny obiekt, przy czym definicja "kolejnego obiektu" zależy od typu kolejki: FIFO, LIFO czy priorytetowa.

Kolejny skrypt zademonstruje sposób współpracy między wątkami.

Mamy w nim dwa obiekty – procesy producentów, które wkładają dane do kolejki **FIFO** oraz dwa obiekty – procesy konsumentów, które odczytują dane. Kolejka ma maksymalny rozmiar **3**, a każdy producent ma wyprodukować po **5** danych – tutaj musi zajść synchronizacja działań procesów. Do kolejki można wstawiać dowolne obiekty – w tym przypadku wstawiane są listy dwuelementowe, zawierające nazwę producenta oraz numer wytworzonego przez niego obiektu. W celu zasygnalizowania procesowi konsumenta faktu zakończenia generacji danych przez producenta, do kolejki wstawiany jest obiekt listy, której pierwszym elementem jest ciąg znaków **END**. Takie rozwiązanie narzuca wymaganie, by liczba producentów i konsumentów była identyczna.

```
from multiprocessing import Process, Queue

def producer(name,q):
    for i in range(5):
        item=["producer:"+str(name),i]
        print(item[0],"putting data",item[1],"; queue size before put operation =",q.qsize())
        q.put(item)
    item=["END",name]
    q.put(item)

def consumer(name,q):
    while True:
        item=q.get()
        if item[0]=="END":
            print("END signal from producer:",item[1])
            break
        else:
            print ("consumer:",name,"getting data:",item[1], "from",item[0],"; queue size after

def main():
    queue=Queue(3)
    producers=[Process(target=producer, args=(i+1,queue)) for i in range(2)]
    consumers=[Process(target=consumer, args=(i+1,queue)) for i in range(2)]
    for producer_process in producers:
        producer_process.start()
    for consumer_process in consumers:
        consumer_process.start()
    print ("END OF MAIN")

if __name__ == "__main__":
        main()
```

Proszę zauważyć, że w funkcji **main** nie ma wywołania metod **join** dla poszczególnych procesów producentów i konsumentów, w związku z czym główny proces skryptu kończy swoje działanie **przed** zakończeniem działania procesów potomnych.

Pule procesów

Istnieje grupa problemów obliczeniowych, których zrównoleglenie nie wymaga współdzielenia obszarów pamięci, jak i komunikacji pomiędzy poszczególnymi zadaniami obliczeniowymi. Przykładowo problem numerycznego obliczania całek oznaczonych należy do tej grupy z uwagi na własność:

$$I = \int_a^b f(x)dx = \int_a^k f(x)dx + \int_k^b f(x)dx, \qquad a < k < b,$$

gdzie przedział całkowania można podzielic na rozłaczne podprzedziały i dla nich wykonywać niezależnie obliczenia.

W celu numerycznego policzenia wartości całki oznaczonej należy:

- zdefiniować funkcję (np. `I`), która dla zadanej wartości x zwróci wartość funkcji całkowanej $f(x)$:
- zdefiniować górną i dolną granicę całkowania: np. `lowerLimit`, `upperLimit`,
- wywołać metodę numeryczną z parametrami: `met_num(I, lowerLimit, upperLimit)`.

W języku Python dostępnych jest wiele pakietów, które umożliwiają wykonanie obliczeń całkowania numerycznego. Na potrzeby bieżącego przykładu wybrany został pakiet `scipy` i medoda quad

Rozważmy problem obliczenia:

$$I = \int_0^5 (3x^2 + 1)dx$$

z użyciem obliczeń równoległych. Do rozwiązania tego problemu wykorzystana zostanie klasa `Pool`, która kontroluje zadaną grupę procesów. W pierwszym kroku należy zaimportować do skryptu odpowiednie klasy:

```
from multiprocessing import Pool
from scipy import integrate
```

W linii nr. 2 importowany jest cały moduł `integrate`, udostępniający grupę metod służących do numerycznego wyznaczania wartości całek oznaczonych.

UWAGA: Pakiet można w razie potrzeby można doinstalować wywołując komendę:

```
pip3 install scipy
```

```
def work(integrationLimits):
    def integrant(x): # funkcja całkowana
        return 3*x**2 + 2
    # integrant - algorytm numeryczny do całkowania
    return integrate.quad(integrant, integrationLimits[0], integrationLimits[1])
```

Powyższy fragment kodu implementuje zadanie obliczeniowe dla pojedynczego procesu. W ramach funkcji `work` zdefiniowano funkcję wewnętrzną `integrant`, obliczającą wartość funkcji całkowanej dla zadanego przedziału. W linii nr. 4 zostaje wywołana właściwa metoda numeryczna, a jej wynik końcowy jest zwracany do otoczenia. Poniżej zaprezentowany jest przykład użycia tak zdefiniowanej funkcji.

```
def main():
    # Utworzenie puli procesów.
    myPool = Pool(processes=5)
    # Przypisanie ("mapowanie")  do procesów zadań wraz z zakresami całkowania
    # i uruchomienie procesów.
    results = myPool.map(work, [(0,1),(1,2),(2,3),(3,4),(4,5)])
    myPool.close()
    finalResult = 0
    # Agregacja wyników cząstkowych.
    for result in results:
        finalResult += result[0]
    print(finalResult)
    #print(integrate.quad(lambda x:3*x**2+1,0,5)) #sprawdzenie poprawności wyników -
```

```python
if __name__ == '__main__':
    main()
```

```
135.0
```

Pakiet ipyparallel

Jest to pakiet umożliwiający obliczenia równoległe i rozproszone z wykorzystaniem interaktywnej powłoki `ipython`.

Architektura

Poniższy rysunek przedstawia architekturę środowiska obliczeniowego tworzonego za pomocą tego pakietu.

- **Engines** ("silniki obliczeniowe") – *ipython kernels* – nasłuchują poleceń, wykonują zadania obliczeniowe, zwracają uzyskane wyniki.
- **Schedulers** ("planiści") – są to pośrednicy: przekazują zadania do wykonania do "silników obliczeniowych"; udostępniają "nieblokującą" warstwę pośrednią.
- **Client** ("klient") – główny obiekt umożliwiający komunikację użytkownikowi z klastrem obliczeniowym. Dla każdego modelu obliczeń rozproszonych udostępniany jest dedykowany model "widoku" np. *DirectView*, *LoadBalancedView*.
- **Hub** – centralny proces klastra obliczeniowego, który zarządza połączeniami z "silnikami", "planistami", obiektem "klienta", wynikami.

1. Instalacja pakietu:

```
pip3 install ipyparallel
```

2. Uruchomienie klastra obliczeniowego wraz z "silnikami" (w oddzielnej powłoce):

```
$ipcluster start -n 4
```

Parametr `-n` decyduje o liczbie uruchomionych "silników obliczeniowych".

Wynik działania polecenia:

```
[IPClusterStart] Starting ipcluster with [daemon=False]
[IPClusterStart] Creating pid file: /home/marek/.ipython/profile_default/pid/ipcluster.pid
[IPClusterStart] Starting Controller with LocalControllerLauncher
[IPClusterStart] Starting 4 Engines with LocalEngineSetLauncher
[IPClusterStart] Engines appear to have started successfully
```

3. Uruchomienie skryptu z "klientem", który będzie zlecał wykonanie zadań działającym "silnikom":

```python
import ipyparallel as parallel
client = parallel.Client()
print(client.ids)
```

Taki skrypt pokaże listę działających silników (dla n=4 otrzymamy)

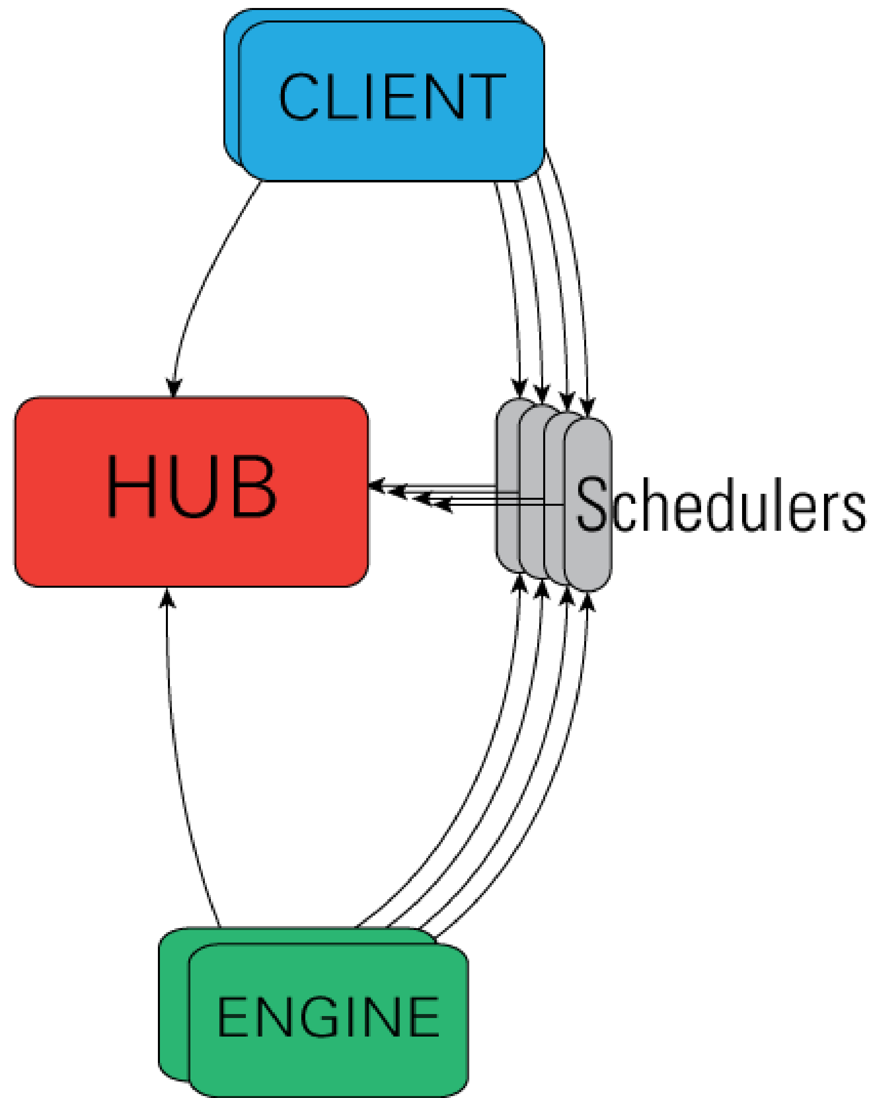

Figure 1:

```
[0, 1, 2, 3]
```

Aby wykonać zadania w trybie równoległym z równoważeniem obciążenia (*load balancing*), należy stworzyć odpowiedni obiekt widoku:

```
lview = client.load_balanced_view()
```

Należy także zdefiniować zadanie, jakie będzie wykonywane w sposób równoległy. Może to być np. funkcja:

```
def work(x):
    return x*x*x
```

Następnie należy powiązać zadania do wykonania z "silnikami":

```
result=lview.map(work,[ x for x in range(15) ])
```

Pierwszy argument metody `map` to zadanie do wykonania, drugi to zbiór argumentów, dla których zadania będą wykonywane. Wynik obliczeń otrzymujemy w postaci listy, którą można wyświetlić poleceniem:

```
print(result.get())
```

```
[0, 1, 8, 27, 64, 125, 216, 343, 512, 729, 1000, 1331, 1728, 2197, 2744]
```

Autor materiałów: dr inż. Marek Niewiński