lab4

April 28, 2021

1 Paradygmaty programowania

1.1 Materiały przygotowawcze do Laboratorium nr. 4

Temat: Przetwarzanie równoległe i mechanizmy synchronizacji

1.1.1 Wprowadzenie

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

W języku *Python* dostępne są następujące mechanizmy wykonywania zadań równolegle:

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

Na zajęciach laboratoryjnych będą Państwo wykorzystywali tylko trzy pierwsze mechanizmy.

1.1.2 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:

[1]:
```
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:

[2]:
```
import time
def work(number):
    print ("Thread number:",number)
    current_time = time.time()
```

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

teraz zdefiniujemy grupę wątków:

[3]:
```
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 konkretnego wątku, należy wywołać metodę **join** na rzecz tego wątku

[4]:
```
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**
- głowne ciało funkcji wykonuje dziesięć razy funkcję **main** i wyświetla końcową wartość zmiennej globalnej **data**

Tip: Numerację linii kodu można włączyć naciskając kombinację: `Shift+l`

[5]:
```
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}: x = {1}".format(i+1,data))
```

```
Iteration 1: x = 311324
Iteration 2: x = 400000
Iteration 3: x = 349781
Iteration 4: x = 327742
Iteration 5: x = 400000
Iteration 6: x = 368896
Iteration 7: x = 333892
Iteration 8: x = 364395
Iteration 9: x = 259823
Iteration 10: x = 365273
```

Jak widać, końcowa wartość zmiennej globalnej (dla każdej iteracji) nie zawsze równa się oczekiwanej wartości (4 x 100000=>400000), tylko przyjmuje "różne wartości". Zjawisko to wynika z braku synchronizacji dostępu do zmiennej globalnej.

Po to, by wyeliminować to zjawisko, należy zastosować jakiś mechanizm synchronizacji. Najprostrzym 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 temu fragment kodu realizujący inkrementację zmiennej globalnej jest realizowany zawsze tylko przez **jeden** wątek.

[16]:
```
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}: 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
```

1.1.3 Multiprocessing

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 *processing* (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 pdpowiednie klasy:

[7]:
```
from multiprocessing import Process
```

najprostszą metodą wskazania zadania do wykonania w ramach procesu jest:

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

Zdefiniujmy przykładową funkcję:

[8]:
```
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:

[9]:
```
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).

[10]:
```
for process in processes:
        process.start()
for process in processes:
        process.join()
```

```
Process: 0 ; Process id: 34303 ; Parent process id: 34151
Process: 1 ; Process id: 34306 ; Parent process id: 34151
Process: 2 ; Process id: 34309 ; Parent process id: 34151
```

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()* – wkłada dowolny obiekt do kolejki,
- *get()* – usuwa z kolejki i zwraca dostępny obiekt.

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 o zakończeniu generacji danych przez producenta, do kolejki wstawiany jest obiekt listy, której pierwszym elementem jest ciąg znaków **END** (takie rozwiązanie wymusza by liczba producentów i konsumentów była identyczna).

[11]:
```
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 get operation =",q.qsize())

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()
```

```
producer:1 putting data 0 ; queue size before put operation = 0
END OF MAIN
producer:1 putting data 1 ; queue size before put operation = 1
producer:2 putting data 0 ; queue size before put operation = 0
producer:1 putting data 2 ; queue size before put operation = 2
producer:2 putting data 1 ; queue size before put operation = 1
consumer: 2 getting data: 0 from producer:2 ; queue size after get operation = 1
producer:2 putting data 2 ; queue size before put operation = 2
consumer: 2 getting data: 1 from producer:1 ; queue size after get operation = 1
producer:2 putting data 3 ; queue size before put operation = 2
consumer: 2 getting data: 1 from producer:2 ; queue size after get operation = 1
consumer: 1 getting data: 0 from producer:1 ; queue size after get operation = 0
producer:2 putting data 4 ; queue size before put operation = 2
```

```
consumer: 2 getting data: 2 from producer:2 ; queue size after get operation = 1
consumer: 2 getting data: 3 from producer:2 ; queue size after get operation = 2
consumer: 1 getting data: 4 from producer:2 ; queue size after get operation = 1
END signal from producer: 2
producer:1 putting data 4 ; queue size before put operation = 1
consumer: 1 getting data: 2 from producer:1 ; queue size after get operation = 0
producer:1 putting data 3 ; queue size before put operation = 1
consumer: 1 getting data: 3 from producer:1 ; queue size after get operation = 0
consumer: 1 getting data: 4 from producer:1 ; queue size after get operation = 1
END signal from producer: 1
```

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.

1.1.4 Pule procesów

Istnieje grupa problemów obliczeniowych dla których ich 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:

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

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

W celu numerycznego policzenia wartości całki oznaczonej należy:

- zdenifiować funkcję (np. I), która dla zadanej wartości `x` zwróci wartość funkcji całkowanej `f(x)`:
- zdefiniować górną i dolną granicę całkowania: np. *lowerLimit*, *upperLimit*
- wywować 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

Rozważny następujący problem, policzyć:

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

wykorzystując obliczenia równoległe.

Do rozwiązania tego problemu wykorzystana zostanie klasa *Pool*, która kontroluje zadaną grupę procesów. W pierwszym kroku zaimportować do skryptu odpowiednie klasy:

[12]:
```
from multiprocessing import Pool
from scipy import integrate
```

W linii nr. 2 importowany jest cały moduł *integrate* udestępniający grupę metod numerycznych do wyznaczania wartości całek numerycznie.

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

```
pip3 install scipy
```

[13]:
```
def work(integrationLimits):
    def integrant(x): # funkcja całkowana
        return 3*x**2 + 1
    return integrate.quad(integrant,integrationLimits[0],integrationLimits[1])␣
↪#  algorytm numeryczny do całkowania
```

Powyższy fragment kodu implementuje, zadanie obliczeniowe dla każdego procesu. W ramach funkcji `work` zdefiniowano funkcję wewnętrzną `integrant` obliczającą wartość funkcji całkowanej dla zadanego x.

W linii nr. 4 zostaje wywołana właściwa metoda numeryczna a jej wynik końcowy zostaje zwócony do otoczenia.

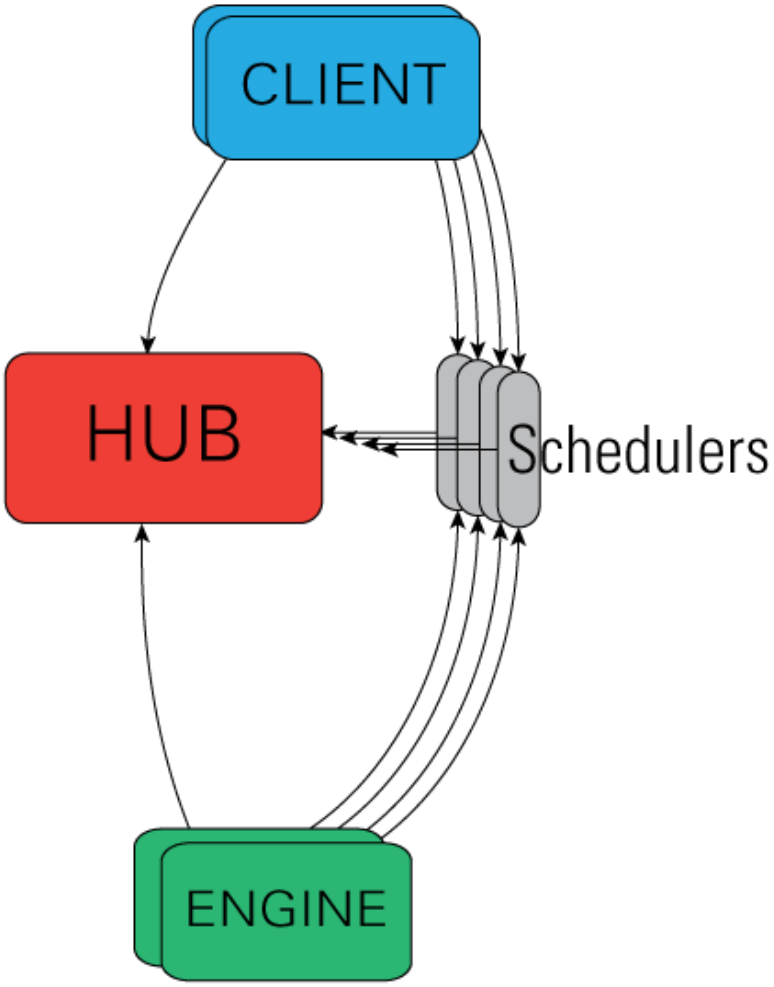
[14]:
```
def main():
    myPool=Pool(processes=5) #Tworzona jest pula procesów
    results=myPool.map(work,[(0,1),(1,2),(2,3),(3,4),(4,5)]) # mapowanie zadań␣
↪do procesów wraz z zakresami całkowania
    myPool.close()
    finalResult=0
    for result in results: # ta pętla służy do obliczenia końcowej wartości -␣
↪bez uwzgędnienia niepewności obliczeń
        finalResult+=result[0]
    print(finalResult)
    #print(integrate.quad(lambda x:3*x**2+1,0,5)) #sprawdzenie poprawnościi␣
↪wyników -

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

```
130.0
```

1.1.5 Pakiet ipyparallel

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

1.1.6 Architektura



- **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"

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

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

```
[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 (można je 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

[ ]: