MULTIPROCESSING AVEC PYTHON

ACTION FEDERATRICE DEVCAL

(Marco Mancini)

SOMMAIRE

THREADING (THREAD-BASED PARALLELISM)

- Interface d'haut niveau pour manipulation des processus légers
- classe `Thread` représente une activité lancée dans un autre thread

https://docs.python.org/3/library/threading.html

THREADING (EXEMPLE)

cat python/threading_0.py

```
import threading

def worker(num):
    """thread worker function"""
    print 'Worker: %s' % num
    return

threads = []
for i in range(5):
    t = threading.Thread(target=worker, args=(i,))
    threads.append(t)
    t.start()
```

```
python python/threading_0.py
Worker: 0
Worker: 1
Worker: 2
Worker: 3
Worker: 4
```

QUELQUES PROBLÈME AVEC *THREADING*

Considérons la fonction intensive (CPU-bound):

```python
"""python/intensive.py"""
def busy_sleep(n):
    while n > 0:
        n -= 1
```

Avec un seul thread:

```python
"""python/seq_gil.py"""
from intensive import busy_sleep
N = 100000000
busy_sleep(N)
```

QUELQUES PROBLÈME AVEC *THREADING*

Avec deux threads:

```python
"""python/par_gil.py"""
from intensive import busy_sleep
from threading import Thread
N = 100000000
t1 = Thread(target=busy_sleep, args=(N//2, ))
t2 = Thread(target=busy_sleep, args=(N//2, ))
t1.start()
t2.start()
t1.join()
t2.join()
```

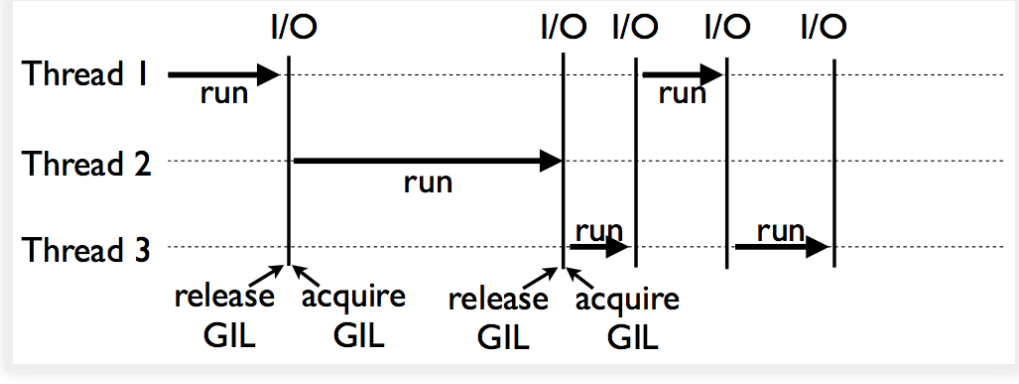
on obtient:

```
time python python/seq_gil.py  ==>  4.796s
time python python/par_gil.py  ==>  5.243s
```

GIL : GLOBAL INTERPRETER LOCK

Un feu rouge : un seul thread à la fois accède à l'interpréteur.

Les threads sont serializés (exceptions: I/O, numpy,..)



Aucune performance dans le cas de multithreading

https://docs.python.org/3/c-api/init.html

MULTIPROCESSING

- Semblable à *threading* mais basée sur les processus
- Countournement du GIL
- API de niveau encore plus élevée
- classe de base `Process`

docs.python.org/3/library/multiprocessing.html

On verra:

- `Process`
- `Queue`
- `Pool`

On verra pas:

- `Pipe`
- `Connection`
- `Shared ctypes Objets`

La classe *Process*

Process représente un processus:

```
p = Process(target=f, args=('bob',))
```

- *target* : associe une fonction au processus
- *args* : la list des arguments dans une n-tuple
- `p.start()` lance le processus
- `p.join([timeout])` attend son arret
- `p.name` nom du processus
- ... `pid, daemon, is_alive(), terminate()`

La classe *Process* : Exemple precedent

```python
"""python/par_gil_process.py"""
from intensive import busy_sleep
from multiprocessing import Process
N = 100000000
p1 = Process(target=busy_sleep, args=(N//2, ))
p2 = Process(target=busy_sleep, args=(N//2, ))
p1.start()
p2.start()
p1.join()
p2.join()
```

on obtient:

```
time python python/seq_gil.py  ==>  4.796s
time python python/par_gil_process.py  ==> 2.586
```

https://docs.python.org/3/library/multiprocessing.html

La classe *Process* : Exemple (Demon)

```
"""daemon.py"""
from multiprocessing import Process, current_process
import time

def non_daemon(num):
    print( "in non_daemon block ", current_process().name)

def daemon(num):
    time.sleep(4)
    print( "in daemon block ", current_process().name)

if __name__ == '__main__':
    p1 = Process(name ='worker1', target=non_daemon, args=(1,))
    p2 = Process(name ='service', target=daemon, args=(2,))
    p2.daemon =True
    p1.start()
    p2.start()
    time.sleep(1)
    #p2.join()
    print( "All finished")
```

La classe *Process* : Exemple (Demon)

Avec `sleep(4)` dans le daemon:

```
python daemon.py
in non_daemon block  worker1
All finished
```

Avec `sleep(4)` dans le non_daemon:

```
python daemon.py
in daemon block  service
All finished
in non_daemon block  worker1
```

La classe *Process* : Un Exercice tout simple

Créer n processus qui lancent un worker.

Le worker fait un print du nom du processus et de son PID.

La classe *Process* : Un Exercice tout simple

```
"""multiprocessing_0.py"""
import multiprocessing as mp

def worker(num):
    """thread worker function"""
    my_p = mp.current_process()
    print('Worker:', num,my_p.pid, my_p._parent_pid, my_p.name)

if __name__ == '__main__':
    jobs = []
    for i in range(5):
        p = mp.Process(target=worker, args=(i,), name='proc-'+str(i) )
        jobs.append(p)
        p.start()

    for i in jobs:
        i.join()
```

La classe *Process* : Un Exercice tout simple

```
python multiprocessing_0.py
Worker: 0 21173 21172 proc-0
Worker: 2 21175 21172 proc-2
Worker: 1 21174 21172 proc-1
Worker: 3 21176 21172 proc-3
Worker: 4 21177 21172 proc-4
```

La classe *Queue*

Queue sert à échanger des données (et des objets) entre les processus (FIFO structure).

```
q = Queue()
```

- `q.empty()` si la queue est vide
- `q.get()` récupère et vide la queue de son contenu
- `q.put(item)` met des données dans la queue

La classe *Queue* : Exemple

```
from multiprocessing import Process, Queue
import random

def rand_num(queue):
    num = random.random()
    #print(num)
    queue.put(num)

if __name__ == "__main__":
    queue = Queue()

    processes = [Process(target=rand_num, args=(queue,)) for x in range(4)]

    for p in processes:
        p.start()

    for p in processes:
        p.join()

    results = [ queue.get() for p in processes ]
    print (results)
```

La classe *Queue* : Exercice

Céer un systeme avec double queue:

- Une queue utilisée pour amener des valeurs dans les worker
- Une queue utilisée pour récupérer les résultats.
- On supposera que chaque valeur initial est un multiple de 100000
- Chaque processus ajoutera son PID à ce valeur.

La classe *Queue*: Exercice

```
import multiprocessing as mp

def add_pid(queue_in,queue_out):
    num = mp.current_process().pid

    val = queue_in.get()
    #print(num)
    queue_out.put(num + val)

if __name__ == "__main__":
    queue0 = mp.Queue()
    queue1 = mp.Queue()

    # fullfil the input queue
    [queue0.put(ii*10000) for ii in range(4)]

    # initialize process
    processes = [mp.Process(target=add_pid, args=(queue0,queue1,)) for x in range(4)]

    for p in processes:  p.start()
    for p in processes:  p.join()

    # get results from output queue
```

```
    results = [ queue1.get() for p in processes ]
    print (results)
```

La classe *Pool*

- *Pool* est une classe que permet de paralléliser facilement.
- `p = Pool(processes=4)` est un ensemble de 4 processus sur lequel une fonction peut être appliquée.
- `p.map` permet de "mapper" la fonction:

```
'''pool_simple_map_apply.py'''
import multiprocessing as mp

def cube(x):
    return x**3

p = Pool(processes=4)
results = p.map(cube, range(1,7))
print(results) #[1, 8, 27, 64, 125, 216]
```

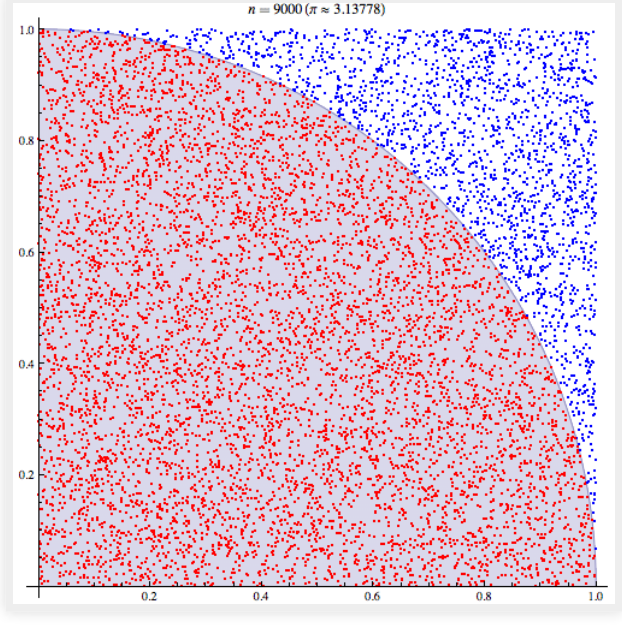
- `p.apply` est utile pour les fonctions avec plusieurs arguments:

```
p= Pool(processes=4)
results = [p.apply(cube, args=(x,)) for x in range(1,7)]
print(results) #[1, 8, 27, 64, 125, 216]
```

- Il exist d'autres fonctions : `p.imap`, `p.imap_unordered`, `p.map_async`, `p.starmap`

La classe *Pool*: Exemple (integration Montecarlo)

On suppose de vouloir faire une intégration de type statistique :



En lançant un numéro important de points sur la courbe d'une fonction on peut calculer l'intégral de la fonction.

La classe *Pool* : Exemple (integration Montecarlo)

Code pour le calcul de l'intégral

```
def integral(function, xbound, numPoints=1000000):
    xmin, xmax = xbound
    ymin, ymax = find_yminymax(function, xbound, numPoints)
    rectArea = (xmax - xmin) * (ymax - ymin)
    ctr = 0
    for j in range(numPoints):
        x = xmin + (xmax - xmin) * random.random()
        y = ymin + (ymax - ymin) * random.random()
        if math.fabs(y) <= math.fabs(function(x)):
            if function(x) > 0 and y > 0 and y <= function(x):
                ctr += 1 # area over x-axis is positive
            if function(x) < 0 and y < 0 and y >= function(x):
                ctr -= 1 # area under x-axis is negative

    baseArea = ymin * (xmax - xmin)
    fnArea = rectArea * float(ctr) / numPoints
    return fnArea + baseArea
```

La classe *Pool*: Exemple (integration Montecarlo)

Où la fonction pour calculer le max et le min de la coordonnée y est :

```
# find ymin-ymax
def find_yminymax(function, xbound, numSteps=1000000):
    ymin = function(xbound[0])
    ymax = ymin

    for i in range(numSteps):
        x = xbound[0] + (xbound[1] - xbound[0]) * float(i) / numSteps
        y = function(x)
        #print(x,y)
        if y < ymin: ymin = y
        if y > ymax: ymax = y

    return ymin,ymax
```

Et pour lancer le calcul:

```
seq_area = integral(f, [xmin, xmax],N)
```

La classe *Pool* : Exemple (integration Montecarlo)

Exercice : Parallélisation de l'intégration avec *Pool*

- Comment partager le calcul ?
- `Pool.map` ne gére pas bien les fonction de plusieurs arguments :

```
def integral_wrap(args):
  # * collects all the positional arguments in a tuple
  return integral(*args)
```

Solutions

```
# Without Order in output
lista = ((f,[(xmax-xmin)/nproc*ii+xmin,
            (xmax-xmin)/nproc*(ii+1)+xmin], N//nproc) for ii in range(nproc))
p = mp.Pool(processes=nproc)
par_area = 0
for part in p.imap_unordered(integral_wrap,lista):  par_area += part
```

```
# Using Apply : some problem with python 3.5 or on my laptop
lista2 = ([(xmax-xmin)/nproc*ii+xmin,
           (xmax-xmin)/nproc*(ii+1)+xmin] for ii in range(nproc))
p = mp.Pool(processes=nproc)
results = (p.apply(integral, args=(f,x,N//nproc)) for x in lista2)
par_area = sum(results)
```

```
python montecarlo_integration.py

Test integration of the function :
            1/(1+x**2)
Exact integration = 0.785398163397448
---------------------------------
Result for N = 1000000
 parallel   integration = 0.7853487146516074
 sequential integration = 0.785393457099523
Timing:
 sequential              = 1.8331952095031738
 parallel with 16 procs  = 0.34990477561950684
---------------------------------
Result for N = 10000000
 parallel   integration = 0.7852551714744861
 sequential integration = 0.7853972742304417
Timing:
 sequential              = 18.81016755104065
 parallel with 16 procs  = 1.770831823348999
---------------------------------
Result for N = 50000000
 parallel   integration = 0.7854155342916894
 sequential integration = 0.7853978153100228
Timing:
 sequential              = 89.51818585395813
 parallel with 16 procs  = 7.661694526672363
```

La classe *Pool* : Exercice (Calcul de Pi en parallel)

Conseils :

- Créer un fonction (et aussi son "wrap")
- Pour semplicité considerer seulement un nombre `nb` divisible par `nrpoc`

La classe *Pool* : Exercice (Calcul de Pi en parallel)

On considere la suivante code sequentiel :

```
'''pi_base.py'''
import multiprocessing
import math
import time

nb = 1000000 #int(input("number of bins : "))
nproc = 1

start = time.time()

ppi = 0.
for ii in range(0,nb):
    jj = (ii + 0.5)/nb
    ppi += 1./(1. + jj**2)

ppi = 4./nb * ppi
end = time.time()

print("Valeur approchee de pi : {} (delta={:e})".format(ppi,abs(ppi-math.pi)))
print("Calculee en {} par {} processus".format(end-start,nproc))
```

La classe *Pool* : Exercice (Calcul de Pi en parallel)

Solution :

```
'''pi_base.py'''
import multiprocessing
import math
import time

def partial_pi(izero, iend, nb):
    ppi = 0
    nb_inv = 1. / nb
    for ii in range(izero,iend):
        jj = (ii + 0.5) * nb_inv
        ppi += 1./(1. + jj**2)
    return ppi

def partial_pi_wrap(args):
    return partial_pi(*args)
```

La classe Pool: Exercice (Calcul de Pi en parallel)

Solution :

```
if __name__ == '__main__':
    nb = 100000000 #int(input("number of bins : "))
    nproc = mp.cpu_count()//2

    start = time.time()
    div   = nb // nproc
    if nb % nproc != 0 : div += 1
    nb = nproc * div
    pool = mp.Pool(nproc)

    # using imap_unordered
    lista = ((ii*div,(ii+1)*div,nb)  for ii in range(nproc))
    result = pool.imap_unordered(partial_pi_wrap,lista)
    ppi = 4./nb * sum(result)
    end = time.time()

    print("Valeur approchee de pi : {} (delta={:e})".format(ppi,abs(ppi-math.pi)))
    print("Calculee en {} par {} processus".format(end-start,nproc))
```

Une solution alternative :

```
    # using Apply does not work
    lista = ((ii*div,(ii+1)*div)  for ii in range(nproc))
    result = (pool.apply(partial_pi, args=(x1,x2,nb)) for x1,x2 in lista)
```