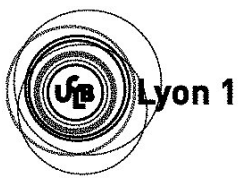
Multiprocessing avec Python

Anne Cadiou

Laboratoire de Mécanique des Fluides et d'Acoustique

Ateliers et Séminaires Pour l'Informatique et le Calcul Scientifique
PMCS2I - LMFA
Vendredi 13 mars 2020



Introduction

Il existe de nombreuses API et bibliothèques qui permettent de faire du calcul parallèle avec Python

`https://wiki.python.org/moin/ParallelProcessing`

Les plus connues sont

- `threading`, basé sur des threads lancés par un processus
- `multiprocessing`, semblable à `threading` mais basée sur des sous-processus

et

- `mpi4py` basé sur MPI

mais on trouve aussi

- `ray` pour le calcul distribué

, etc.

multiprocessing proche du module threading ?

```
#!/usr/bin/env python3

import threading
import time
import random

def hello(n):
    time.sleep(random.randint(1,3))
    print("[{0}] Hello!".format(n))

for i in range(10):
    threading.Thread(target=hello, args=(i,)).start()

print("Done!")
```

```
Done!

[0] Hello!
[7] Hello!
[6] Hello![4] Hello!

[5] Hello!
[1] Hello!
[3] Hello!
[2] Hello!
[8] Hello!
[9] Hello!
```

On voit ici que les lignes `[6] Hello![4] Hello!` se chevauchent (race condition) : lié au fait que les threads accèdent en même temps la fonction `print()`.

Pour imposer que les threads aient fini leur tâche avant l'affichage `Done!`, il suffit de créer les objets à exécuter pour chaque thread, les démarrer (avec `start()`, puis appeler l'instance `join()` pour les terminer. Les objets peuvent être créés dans une liste.

```python
#!/usr/bin/env python3

import threading
import time
import random

def hello(n):
    time.sleep(random.randint(1,3))
    print("[{0}] Hello!".format(n))

threads = [ ]
for i in range(10):
    t = threading.Thread(target=hello, args=(i,))
    threads.append(t)
    t.start()

for one_thread in threads:
    one_thread.join()

print("Done!")
```

```
[2] Hello!
[4] Hello!
[5] Hello!
[3] Hello!
[0] Hello!
[1] Hello!
[7] Hello!
[6] Hello!
[9] Hello!
[8] Hello!
Done!
```

```python
#!/usr/bin/env python3

import multiprocessing
import time
import random

def hello(n):
    time.sleep(random.randint(1,3))
    print("[{0}] Hello!".format(n))

processes = [ ]
for i in range(10):
    t = multiprocessing.Process(target=hello, args=(i,))
    processes.append(t)
    t.start()

for one_process in processes:
    one_process.join()

print("Done!")
```

```
[3] Hello!
[4] Hello!
[8] Hello!
[0] Hello!
[2] Hello!
[1] Hello!
[5] Hello!
[6] Hello!
[7] Hello!
[9] Hello!
Done!
```

- Codes `multithreading` et `multiprocessing` très similaires
- Résultats identiques
- Comportements différents à cause du GIL (Global Interpreter Lock)

Le GIL ou Global Interpreter Lock est un verrou unique auquel l'interpréteur Python fait appel constamment pour protéger tous les objets qu'il manipule contre des accès concurrentiels : un thread à la fois accède à l'interpréteur.

Le module `threading` passe par le GIL (Global Interpreter Lock) ce qui le rend inefficace pour du calcul scientifique.

L'API `multiprocessing` contourne le GIL en faisant appel à des sous-processus.

Rappels

Un thread

est un fil d'exécution d'une tâche générée par un exécutable.

Un processus

est un programme en exécution.

Multithreading

est une technique qui permet à un processus de lancer plusieurs threads simultanément. Les threads partagent les ressources (mémoire, caches, CPU, etc.) du processus.

Multiprocessing

est une technique qui permet de lancer plusieurs process indépendants simultanément. Ils ne partagent leurs ressources qu'au travers des mécanismes de partage de mémoire (Inter Processus Communication) du système ou POSIX.

GIL

L'interpréteur Python utilise un verrou (GIL pour Global Lock Interpreter) auquel il fait appel constamment pour protéger tous les objets qu'il manipule contre des accès concurrentiels :
Python permet de **lancer autant de threads que l'on veut**, mais le **GIL** s'assure qu'**un seul de ces threads soit exécuté à la fois**.

- Pour un code IO-bound, c'est bien adapté car les requêtes d'IO par un thread le mettent en attente. Ils libèrent le GIL et lui permettent de lancer l'exécution d'un autre thread simultanément ;
- Pour un code CPU-bound, c'est très limitant car les threads sont exécutés de façon séquentielle. Le temps d'exécution global du programme multithreadé peut même devenir plus lent que se version séquentielle.

Le multiprocessing induit un overhead lié au temps nécéssaire à la gestion des processus. Cela ne pose pas de problème si la zone parallèle est plus coûteuse en temps de calcul.

Implémentation avec multiprocessing

```
import multiprocessing as mp
import time as time

K = 50
def func(z):
    result = 0
    for k in range(1, K+2):
        result += z**(1./(k**1.5))
    return result

if __name__ == "__main__":
    startTime = time.time()
    N = 1000000
    pool = mp.Pool()
    asyncResult = pool.map_async(func,range(N))
    resultList = asyncResult.get()
    print('Result = ', sum(resultList))
    print('Time elapsed:', time.time() - startTime)
    pool.terminate()
```

Implémentation avec threading

```
import threading as threading
import time as time

class func(threading.Thread):
    def __init__(self, value):
        threading.Thread.__init__(self)
        self.value = value

    def run(self):
        result = 0
        z = self.value
        K = 50
        for k in range(1, K+2):
            result += z**(1./(k**1.5))
            self.value = result
```

...

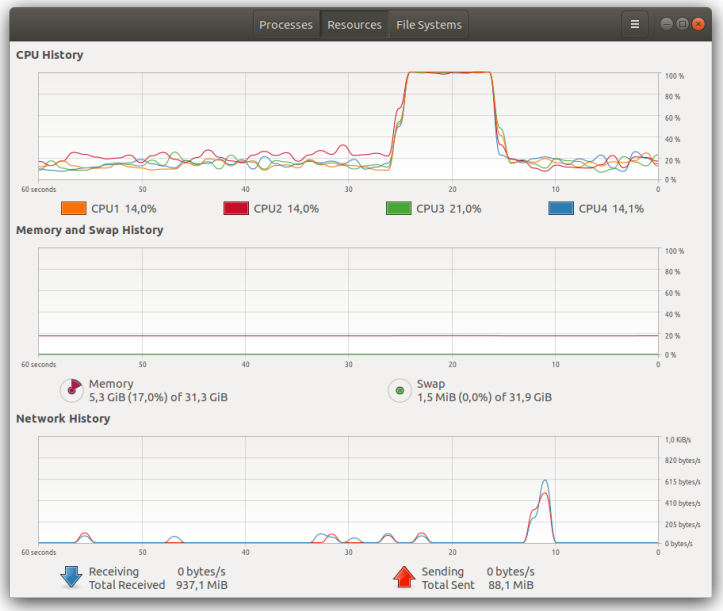
...

```
if __name__ == "__main__":

    # timer
    startTime = time.time()

    # store threads
    threadList = []

    N = 1000000
    for i in range(N):

        curThread = func(i)
        curThread.start()
        threadList.append(curThread)
        resultList = []

    for curThread in threadList:

        curThread.join()
        result = curThread.value
        resultList.append(result)

    print('Result = ', sum(resultList))
    print('Time elapsed:', time.time() - startTime)
    curThread.stop()
```

Résultats

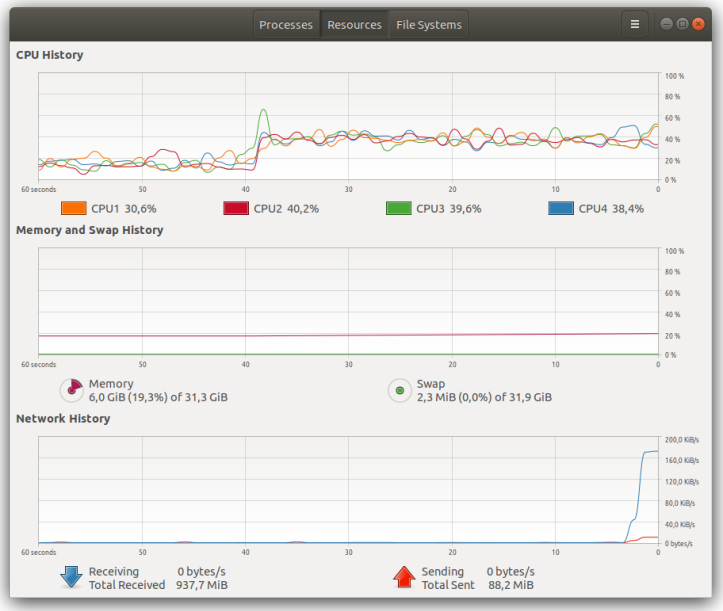
multiprocessing

```
Result = 500172313292.345
Time elapsed: 12.612559080123901
```



threading

```
Result = 500172313292.345
Time elapsed: 92.48981761932373
```



Synthèse

Threads	Processus
Créer des threads est rapide	La création des processus est plus lente
Les threads utilisent un même espace mémoire : • La communication est rapide • Moins de mémoire est utilisée	Les processus utilisent des espaces mémoire séparés : • La communication est lente et difficile • Une plus grande portion de la mémoire est utilisée
Tourne sur un seul processeur	Prend avantage des différents CPUs et cores
Le Global Interpreter Lock (GIL) empêche l'exécution simultanée des threads	Les multiprocess sont indépendants l'un de l'autre, leur permettant ainsi de contourner le GIL
Très bonne méthode de communication (exemple : un serveur qui demande des inputs à un script qui est en train de tourner)	Permet de faire gagner du temps en prenant avantage de toutes les capacités de la machine
Le module 'threading' est un outil très puissant qui contient un grand nombre de fonctionnalités	Le module 'multiprocessing' est conçu pour être aussi proche que possible du module 'threading'
Si un thread crash, tous les autres threads crasheront en conséquence	Un processus qui crash n'affectera pas les autres processus, puisqu'ils sont indépendants les uns des autres

(d'après `https://bioinfo.iric.ca/fr/parallelise-ton-python-y/`)

La classe `Process`

Les processus sont instanciés en créant un objet `Process` et en appelant la méthode `start()` (même principe que `threading.Thread`) pour les démarrer.

- Lorsqu'un objet `Process` est créé, rien n'est exécuté tant que la méthode `start()` ne lui est pas appliquée.
- Pour terminer un processus, utiliser la méthode `join()`. Sans cette méthode, le processus restera IDLE ce qui risque de ne pas libérer les ressources associées.
- `is_alive()` permet de tester si le processus est encore actif.
- Pour passer des arguments au processus, utiliser l'option `args=(,)`

Exemple

```
#!/usr/bin/env python3

import multiprocessing as mp

def f(num):
    print('Hello:', num)

if __name__ == '__main__':
    num = 0
    p = mp.Process(target=f, args=(num,))
    p.start()
    p.join()
    print('actif ? ',p.is_alive())
```

```
Hello: 0
actif ? False
```

Nombre de processeurs disponibles :

```
mp.cpu_count()
```

Identification des processus

```
#!/usr/bin/env python3

import multiprocessing as mp

def f(num):
    my_p = mp.current_process()
    print('Hello:', num, my_p.pid, my_p._parent_pid, my_p.name)

if __name__ == '__main__':
    num = 0
    p = mp.Process(target=f, args=(num,))
    p.start()
    p.join()
```

```
Hello: 0 29118 29117 Process-1
```

Processus parent et fils

```
#!/usr/bin/env python3

import multiprocessing as mp
import os as os

def info(title):
    print(title)
    print('module name:', __name__)
    print('parent process:', os.getppid())
    print('process id:', os.getpid())

def f(num):
    my_p = mp.current_process()
    print('Hello:', num, my_p.pid, my_p._parent_pid, my_p.name)

if __name__ == '__main__':
    info('main program')
    num = 0
    p = mp.Process(target=f, args=(num,))
    p.start()
    print('\nprocess name:', p.name)
    print('process id:', p.pid)
    p.join()
```

```
main program
module name: __main__
parent process: 27071
process id: 29249

process name: Process-1
process id: 29250
Hello: 0 29250 29249 Process-1
```

Lancement de processus simultanés

```
#!/usr/bin/env python3

import multiprocessing as mp
import os as os

def f(num):
    my_p = mp.current_process()
    print('Hello:', num, my_p.pid, my_p._parent_pid, my_p.name)

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

    [p.join() for p in p_lst]
```

```
Hello: 0 29512 29511 proc-0
Hello: 1 29513 29511 proc-1
Hello: 2 29514 29511 proc-2
Hello: 3 29515 29511 proc-3
Hello: 4 29516 29511 proc-4
```

Exemple de travail des processus

```
import multiprocessing as mp

result = []
def f(mylist):
    """ function to square a given list """
    my_p = mp.current_process()
    global result
    # append squares of mylist to global list result
    for num in mylist:
        result.append(num * num)
    # print global list result
    print("result in process",my_p.name,": ",result)

if __name__ == "__main__":
    mylist = [1,2,3,4]
    p = mp.Process(target=f, args=(mylist,))
    p.start()
    p.join()
    print("result in main program:",result)
```

```
result in process Process-1 : [1, 4, 9, 16]
result in main program: []
```

Partage des données

```
import multiprocessing as mp

def f(mylist, result):
    """ function to square a given list """
    my_p = mp.current_process()
    for idx, num in enumerate(mylist):
        result[idx] = num * num
    print("result in process",my_p.name,": ",result[:])

if __name__ == "__main__":
    mylist = [1,2,3,4]
    result = mp.Array('i', len(mylist))
    p = mp.Process(target=f, args=(mylist, result))
    p.start()
    p.join()
    print("result in main program:",result[:])
```

```
result in process Process-1 : [1, 4, 9, 16]
result in main program: [1, 4, 9, 16]
```

```
import multiprocessing as mp

def f(mylist, result):
    """ function to square a given list """
    my_p = mp.current_process()
    for idx, num in enumerate(mylist):
        result[idx] = num * num
    print("result in process",my_p.name,": ",result[:])

if __name__ == "__main__":
    mylist = [1,2,3,4]
    result = mp.Array('i', len(mylist))
    nprocs = 3
    p_lst = [mp.Process(target=f, args=(mylist, result)) for nproc in range(nprocs)
         ]
    for p in p_lst:
        p.start()
    for p in p_lst:
        p.join()
    print("result in main program:",result[:])
```

```
result in process Process-1 : [1, 4, 9, 16]
result in process Process-2 : [1, 4, 9, 16]
result in process Process-3 : [1, 4, 9, 16]
result in main program: [1, 4, 9, 16]
```

La classe Queue

Sert à échanger des données entre les processus, en les passant en argument.

- utilise le module `pickle` pour envoyer les données
- mécanismes de verrous (`locks`) pour les accès concurrentiels
- la fonction `put()` permet d'insérer des données en queue
- la fonction `get()` permet de récupérer les données

```
import multiprocessing as mp

def f(mylist, q):
    """ function to square a given list """
    my_p = mp.current_process()
    for num in mylist:
        q.put((my_p.name, num * num))

def print_queue(q):
    """ function to print queue elements """
    while not q.empty():
        print(q.get())
```

...

...

```
if __name__ == "__main__":
    mylist = [1,2,3,4]
    nprocs = 3
    q = mp.Queue()
    p_lst = [mp.Process(target=f, args=(mylist, q)) for nproc in range(nprocs)]
    r_lst = [mp.Process(target=print_queue, args=(q,)) for nproc in range(nprocs)]
    for p in p_lst:
        p.start()
    for p in p_lst:
        p.join()
    for r in r_lst:
        r.start()
    for r in r_lst:
        r.join()
```

```
('Process-1', 1)
('Process-1', 4)
('Process-1', 9)
('Process-1', 16)
('Process-2', 1)
('Process-2', 4)
('Process-2', 16)
('Process-2', 9)
('Process-3', 1)
('Process-3', 4)
('Process-3', 9)
('Process-3', 16)
```

La classe `Pool`

Permet de travailler avec un réservoir de processus de travail et quatres méthodes d'exécution :

- synchrones
  - `Pool.apply`
  - `Pool.map`
- asynchrones
  - `Pool.apply_async`
  - `Pool.map_async`

Programme séquentiel

```
def square(n):
    return (n*n)

if __name__ == "__main__":
    # input list
    mylist = [1,2,3,4,5]

    # empty list to store result
    result = []

    for num in mylist:
        result.append(square(num))

    print(result)
```

Résultat, exécuté en séquentiel (par un processus) :

```
[1, 4, 9, 16, 25]
```

version multiprocessing

```
import multiprocessing as mp

def square(n):
    my_p = mp.current_process()
    print("Worker process id for ",n,":",my_p.pid)
    return (n*n)

if __name__ == "__main__":
    # input list
    mylist = [1,2,3,4,5]
    nprocs = 3
    # creating a pool object
    pool = mp.Pool(nprocs)

    # map list to target function
    results = pool.map(square, mylist)
    print(results)
```

Résultat, exécuté par 3 processus :

```
Worker process id for 1 : 7421
Worker process id for 2 : 7422
Worker process id for 3 : 7423
Worker process id for 4 : 7421
Worker process id for 5 : 7421
[1, 4, 9, 16, 25]
```

```
Process 7421 : 1, 16, 25
Process 7422 : 4
Process 7423 : 9
```

version multiprocessing par tronçons

```
import multiprocessing as mp

def square(n):
    my_p = mp.current_process()
    print("Worker process id for ",n,":",my_p.pid)
    return (n*n)

if __name__ == "__main__":
    # input list
    mylist = [1,2,3,4,5]
    nprocs = 3
    # creating a pool object
    pool = mp.Pool(nprocs)

    # map list to target function
    results = pool.map(square, mylist,2)
    print(results)
```

Résultat, exécuté par 3 processus :

```
Worker process id for 1 : 25679
Worker process id for 2 : 25679
Worker process id for 3 : 25680
Worker process id for 4 : 25680
Worker process id for 5 : 25681
[1, 4, 9, 16, 25]
```

```
Process 25679 : 1, 4
Process 25680 : 9, 16
Process 25681 : 25
```

version multiprocessing dans un ordre arbitraire

```
import multiprocessing as mp

def square(n):
    my_p = mp.current_process()
    print("Worker process id for ",n,":",my_p.pid)
    return (n*n)

if __name__ == "__main__":
    # input list
    mylist = list(range(1,51))
    nprocs = 3
    # creating a pool object
    pool = mp.Pool(nprocs)

    # map list to target function
    results = list(pool.imap_unordered(square, mylist))
    print(results)
```

fonction à plusieurs arguments

```
import multiprocessing as mp

def power(x,n):
    my_p = mp.current_process()
    print("Worker process id for ",x,":",my_p.pid)
    return (x**n)

if __name__ == "__main__":
    # input list
    mylist = [(x, 2) for x in range(1,51)]
    nprocs = 3
    # creating a pool object
    pool = mp.Pool(nprocs)

    # map list to target function
    results = pool.starmap(power, mylist)
    print(results)
```

Méthode asynchrone

```
import multiprocessing as mp

def power_list(x_list,n):
    return [x ** n for x in x_list]

def slice_data(data, nprocs):
    aver, res = divmod(len(data), nprocs)
    nums = []
    for proc in range(nprocs):
        if proc < res:
            nums.append(aver + 1)
        else:
            nums.append(aver)
    count = 0
    slices = []
    for proc in range(nprocs):
        slices.append(data[count: count+nums[proc]])
        count += nums[proc]
    return slices
```

...

...

```
if __name__ == "__main__":
    # nprocs
    nprocs = 3
    # creating a pool object
    pool = mp.Pool(nprocs)

    # input list
    input_lists = slice_data(range(1,51), nprocs)
    multi_result = [pool.apply_async(power_list, (x, 2)) for x in input_lists]


    # map list to target function
    results = [x for p in multi_result for x in p.get()]
    print(results)
```

Exemple : calcul de π par la méthode de Monte-Carlo

La méthode consiste à tirer au hasard des nombres x et y dans l'intervalle $[0, 1]$.

Si $x^2 + y^2 < 1$, le point $M(x, y)$ appartient au quart de disque de rayon 1. La probabilité pour qu'il en soit ainsi est le rapport des aires du quart de disque de rayon 1 et du carré de côté 1, soit $\pi/4$.

Si n est le nombre total de points générés par une suite aléatoireet p est le nombre de points à l'intérieur du quart de disque, alors $4p/n$ donne une valeur approchée de π.

```python
import random as random
import multiprocessing as mp

def monte_carlo_pi(n):

    count = 0
    for i in range(n):
        x=random.random()
        y=random.random()

        # within the unit circle
        if x*x + y*y <= 1:
            count=count+1
    return count
```

...

...

```
if __name__==’__main__’:

    np = mp.cpu_count()
    print(’number of CPUs: {0:1d}’.format(np))

    # number of points to use for the Pi estimation
    n = 100000000

    # each worker process gets n/np number of points to calculate Pi from
    part_count=[n//np for i in range(np)]
    pool = mp.Pool(processes=np)

    # parallel map
    count=pool.map(monte_carlo_pi, part_count)

    print("estimated value of Pi: ", sum(count)/(n*1.0)*4)
```

```
number of CPUs: 4
estimated value of Pi: 3.14168816
```

Exemple : exploration paramétrique

(`https://fenicsproject.org/docs/dolfin/2019.1.0/python/demos/poisson/demo_poisson.py.html`)

$$-\Delta u = f(\alpha) \text{ dans } \Omega$$
$$u = 0 \text{ sur } \Gamma_D$$
$$\nabla u.n = g \text{ sur } \Gamma_N$$

avec

$$\Omega = [0,1] \times [0,1]$$
$$\Gamma_D = (0,y) \cup (1,y) \subset \delta\Omega \text{ (Dirichelet)}$$
$$\Gamma_N = (x,0) \cup (x,1) \subset \delta\Omega \text{ (Neumann)}$$
$$g = \sin(5x)$$
$$f(\alpha) = \alpha \exp(-((x-0.5)^2 + (y-0.5)^2)/0.02) \text{ (terme source)}$$

Exploration des solutions pour :
$\alpha = 0.1, 1, 10, 100$

```
#!/usr/bin/python3
from dolfin import *

# Define Dirichlet boundary (x = 0 or x = 1)
def boundary(x):
    return x[0] < DOLFIN_EPS or x[0] > 1.0 - DOLFIN_EPS

def poisson(alpha):

    # Mesh
    mesh = UnitSquareMesh(256, 256)
    V = FunctionSpace(mesh, "Lagrange", 1)

    # Define boundary condition
    u0 = Constant(0.0)
    bc = DirichletBC(V, u0, boundary)

    # Define variational problem
    u = TrialFunction(V)
    v = TestFunction(V)
    f = Expression('%g*exp(-(pow(x[0] - 0.5, 2) + pow(x[1] - 0.5, 2)) / 0.02)' % (
          alpha), degree=2)
    g = Expression("sin(5*x[0])", degree=2)
    a = inner(grad(u), grad(v))*dx
    L = f*v*dx + g*v*ds
```

...

...

```
    # Compute solution
    u = Function(V)
    solve(a == L, u, bc)

    # L2 norm
    info('alpha %g -> |u|=%g' % (alpha, u.vector().norm('l2')))

    # Save solution in VTK format
    file = File("poisson_%g.pvd" % alpha)
    file << u

    # Plot solution
    import matplotlib.pyplot as plt
    plot(u)
    plt.title("alpha = %g" % alpha)
    plt.show()
```

...

...

```
alphas = [0.1, 1.0, 10, 100]

import multiprocessing
if __name__ == "__main__":

    pool = multiprocessing.Pool(processes=4)
    pool.map(poisson, alphas)
    pool.close()
```

```
Solving linear variational problem.
Solving linear variational problem.
Solving linear variational problem.
Solving linear variational problem.
alpha 10 -> |u|=38.1301
alpha 0.1 -> |u|=18.4762
alpha 100 -> |u|=240.314
alpha 1 -> |u|=20.0226
```

Références

- `Mutiprocessing avec Python`, de Marco Mancini
- `https://docs.python.org`
- `https://www.geeksforgeeks.org/multiprocessing-python-set-1/`