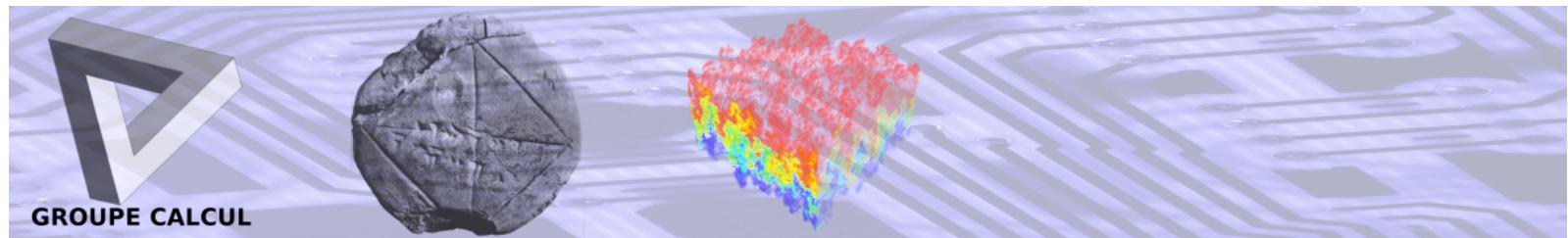


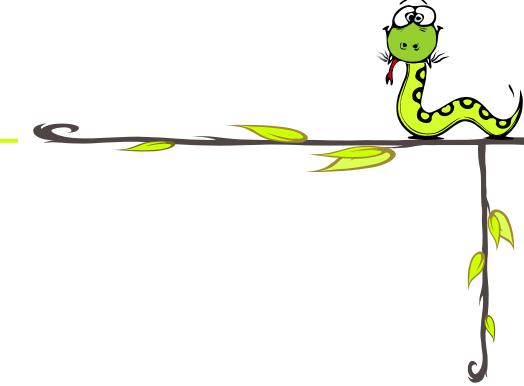


Présentation du Parallélisme et mise en œuvre avec Python



Groupe Calcul du CNRS
Mardi 19 décembre 2017
Rennes - [IRMAR](#)

Votre interlocuteur

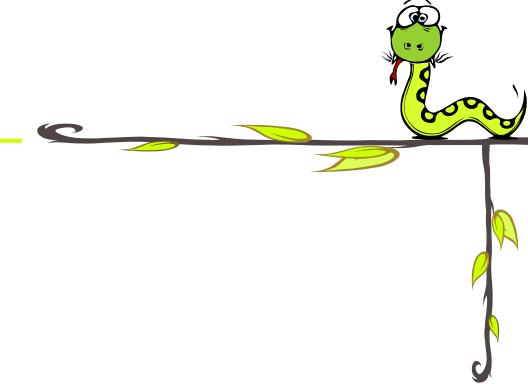


- Gaël Pegliasco <gael@peglasco.com>
- Développe en Python depuis 2004
- Applications web : Pendant 10 ans Zope, Plone, Django
- Calcul scientifique : Depuis 3-4 ans
Numpy, Scipy, Pandas, Matplotlib, Celery, Scikit-Learn, ...
- Sociétés Makina Corpus et Cogitec

**Il est gentil et aura le plaisir de vous guider
dans la découverte de l'écosystème du calcul parallèle
avec Python**



Plan de la présentation



- ***Les concepts du parallélisme***
 - Mécanismes et limites
 - Les différentes formes du parallélisme
- ***Les bases de la programmation parallèle en Python***
 - *Asyncio*
 - *La librairie threading*
 - *La librairie multiprocessing*
- ***Présentation du paysage des librairies de calcul parallèle en Python***
 - *Vue d'ensemble*
 - *Celery, Dask, Numba*



Calcul parallèle



Sunway TaihuLight, le plus puissant supercalculateur au monde

- 10 649 600 coeurs (41000 puces de 260 coeurs à 1.45GHz)
- 93,01 petaflops/s (Linpack test)
93 millions de milliards d'opérations en virgule flottante par seconde
- **Attention**, la norme IEEE754 n'est pas d'une précision fabuleuse
`>>> 0.1 + 0.1 + 0.1 - 0.3 == 0`
- Consommation électrique 15 371 KW par seconde
- *C'est environ la consommation totale annuelle d'une maison de 100m² de 4 personnes*
- 20^{ème} au top500 des green supercomputer
- Et **sur quels OS fonctionnent les supercalculateurs ?**

Source <https://www.top500.org/lists/2017/11>
Novembre 2017

Kalray MPPA2®-256 (Bostan), probablement le CPU le plus parallélisé au monde
<http://www.kalray.eu/kalray/products/>

- 288 coeurs et 128 crypto co-processeurs
- 4.5 GigaFlops
- 64-bit Very Long Instruction Word (VLIW)
- 600 MHz
- Consommation électrique 1 Watt
- Performances comparables aux systèmes ASICs



Présentation Parallélisme : Pourquoi Paralléliser ?



- Une des « *Loi de Moore* » stipule que le nombre de transistors sur les microprocesseurs double tous les 2 ans.
- Une formulation plus commune de cette loi indique que la fréquence (et donc la puissance de calcul) des processeurs double tous les 18 mois.
- Or bien qu'en 2006 IBM et Georgia Tech aient décroché un record de vitesse d'horloge à 500Ghz à -268°C et 350Ghz à température ambiante, depuis 2004 la fréquence des processeurs grand public tend à stagner en raison de problèmes de bruit parasites et de dissipation thermique
- Pour maintenir cette augmentation de la puissance de calcul des ordinateurs la parallélisation est devenue une solution incontournable.



Gordon Earle Moore

Présentation – Parallelisme : Quelques termes



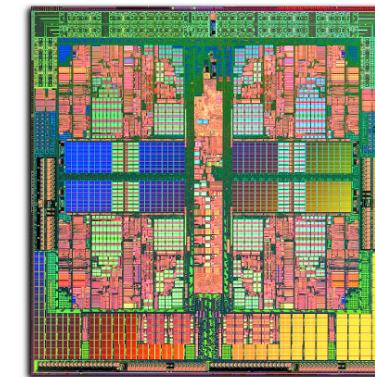
- **Multiprocesseurs**

C'est l'utilisation d'au moins 2 processeurs (CPU) à l'intérieur d'un même ordinateur.



- **Processeur multicœurs**

C'est un processeur possédant plusieurs cœurs physiques/unités de calculs fonctionnant simultanément.

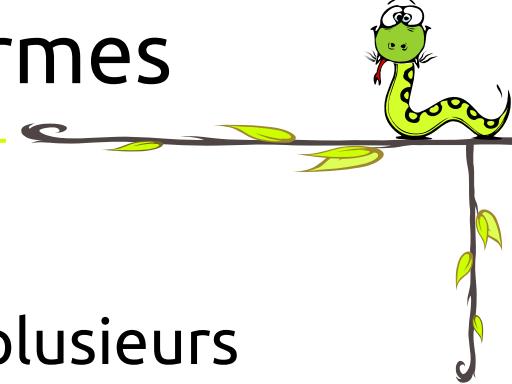


Processeur Quad-core AMD

- **Multiprocessing**

On utilise ce terme quand un système d'exploitation est capable d'exécuter plusieurs programmes en parallèle.

Présentation – Parallelisme : Quelques termes



- **Multitâche**

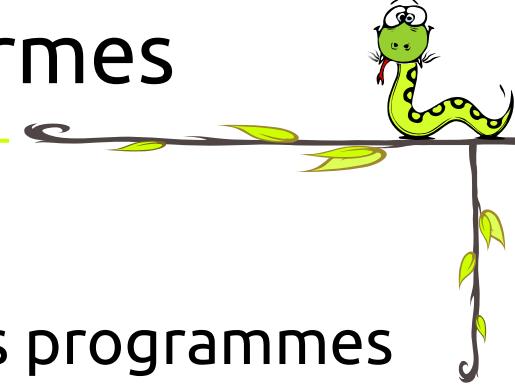
En informatique, le multitâche est une méthode où plusieurs tâches, aussi appelées processus, partagent des ressources de traitement communes comme une unité centrale.

Un système d'exploitation est multitâche lorsqu'il permet de passer d'une tâche à une autre rapidement. Soit en les exécutant parallèlement soit l'une après l'autre pour donner l'impression de simultanéité (cas des OS monoprocesseur et monocœur)

- **Multithreading**

Le multithreading consiste à appliquer l'idée du multitâche au sein d'une même application. C'est à dire de permettre d'exécuter en parallèle différentes actions (comme des fonctions) au sein d'un même programme

Présentation – Parallelisme : Quelques termes



- **Cœur physique**

C'est un ensemble de circuits capable d'exécuter des programmes de façon autonome. Il possède donc ses propres registres, unités de calculs, compteur ordinal, etc. Il possède souvent son propre cache

- **HyperThreading**

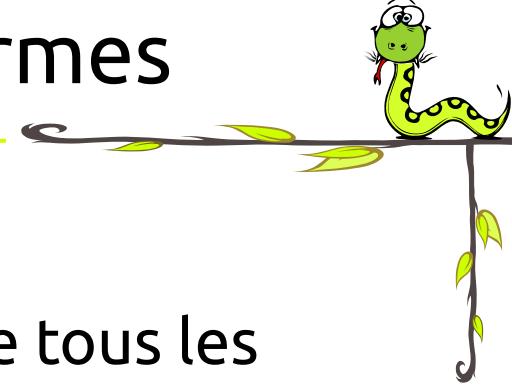
Technologie développée par Intel permettant de créer 2 processeurs logiques sur un même cœur physique.

Les ordinateurs Pentium P4 ou i7 en sont un exemple.

Un seul thread peut s'exécuter à la fois, quand l'un des deux est en attente d'une entrée sortie, l'autre prend le relais.

- Les gamers ne les aiment pas...

Présentation – Parallelisme : Quelques termes



- **SMP**

Symmetric MultiProcessing. On parle de SMP lorsque tous les processeurs sont connectés à une unique mémoire partagée.
Cas des processeurs multicœurs

```
Bash$ uname -a
Linux SMP Tue Sep 19 17:28:18 UTC 2017 x86_64 x86_64 x86_64 GNU/Linux
```

- **NUMA**

Non Uniform Memory Access. On parle d'architecture NUMA lorsque la mémoire entre les processeurs est distribuée dans différents nœuds. Son temps d'accès n'est plus constant et dépend de l'emplacement auquel tente d'accéder un processeur.

- On les programme généralement avec des librairies de type **MPI** (Message Passing Interface)

Présentation – Parallelisme : Quelques termes



- **Différence CPU/GPU**

Un CPU (Central Processing Unit) correspond aux processeurs que l'on trouve dans les ordinateurs portables actuels.

- Ils sont conçus pour exécuter des programmes séquentiels le plus rapidement possible, et, aujourd'hui en parallèle
 - Ils savent gérer les entrées/sorties/interruptions matérielles, partager le cache mémoire et sont de ce fait particulièrement bien adaptés pour faire tourner des systèmes d'exploitation
 - Ils possèdent rarement plus d'une dizaine de cœurs
- Un GPU (Graphics Processing Unit) est un processeur initialement dédié à l'affichage graphique
 - Ils sont conçus pour exécuter la même instruction en parallèle sur de multiples données
 - Ils possèdent des centaines voire milliers de cœurs
 - Imbattables pour le calcul parallèle « *ils ne sont pas taillés* » pour faire tourner un OS

Le CPU est un 4x4 qui s'adapte à tous les besoins, le GPU une formule 1 qui ne roule bien que sur circuit de course...

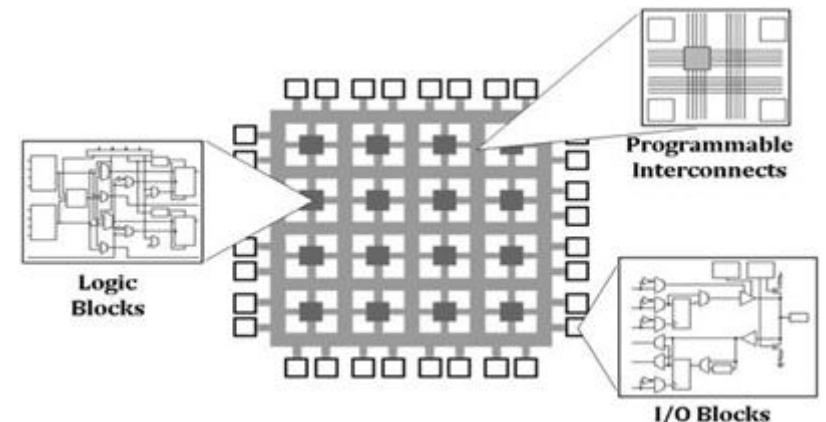
Présentation – Parallelisme : Quelques termes



- **FPGA**

Un FPGA (Field Programmable Gate Arrays) est un matériel qui contient de nombreuses cellules logiques librement reconfigurables.

- Il est composé de nombreuses cellules logiques élémentaires et bascules logiques librement connectables.
- Contrairement aux processeurs, les FPGA utilisent du matériel dédié pour traiter la logique et n'ont pas de système d'exploitation.
- Un FPGA tout seul peut remplacer des milliers de composants discrets en incorporant des millions de portes logiques dans un seul et unique circuit intégré.
- Il peut traiter des opérations en parallèles, comme le feraient les cœurs d'un processeur.
- C'est une solution matérielle donc très performante et qui est reconfigurable.



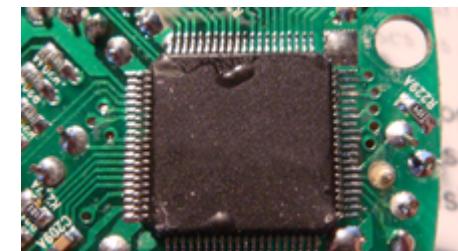
Présentation – Parallélisme : Quelques termes



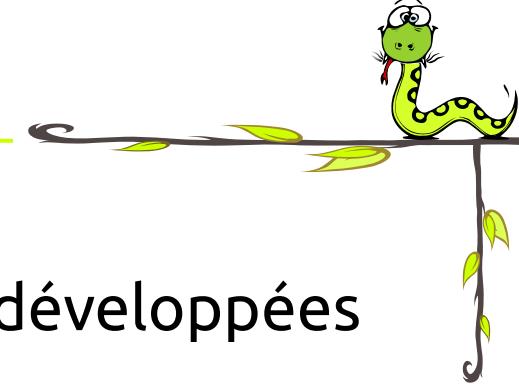
- **ASIC**

Un ASIC (Application Spécifique Integrated Circuit) est une solution matérielle de type circuit intégré spécialisé, non reconfigurable.

- Tout comme le FPGA il est conçu pour concevoir des matériels réalisant des traitements logiques spécifiques.
- Contrairement aux processeurs FPGA le matériel est gravé une fois pour toutes et ne peut pas être reconfiguré pour d'autres traitements.
- Son coût de conception est très grand comparé à l'achat d'un FPGA (souvent plusieurs millions d'euros pour faire le moule).
- Mais le coût de la fabrication des puces est très faible (quelques centimes) lorsque le moule est validé.
- C'est une solution matérielle donc très performante.



Présentation Parallélisme : Les bases

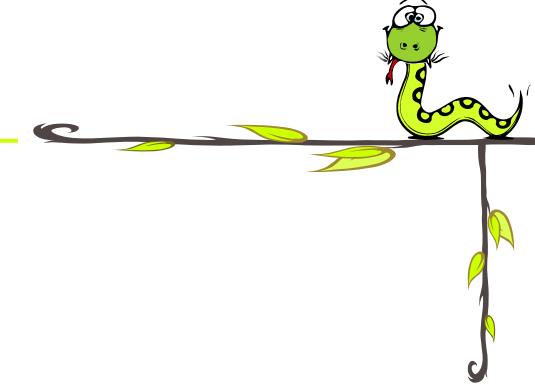


- Les techniques de programmation parallèle ont été développées initialement sur les supercalculateurs
- Elles changent de façon significative notre manière d'écrire des programmes
- Pour bien comprendre les systèmes parallèles
 - « Votre boulangerie a un système d'exploitation multiprocesseur »
- Et leurs limites
 - « Et plus vite si affinités »

Présentation Parallélisme : Les bases



Limites du parallélisme



Tout n'est pas parallélisable

« 1 femme fait un bébé en 9 mois, mais 9 femmes ne peuvent pas faire un bébé en 1 mois » *Frederick Brooks*

Limites du parallélisme



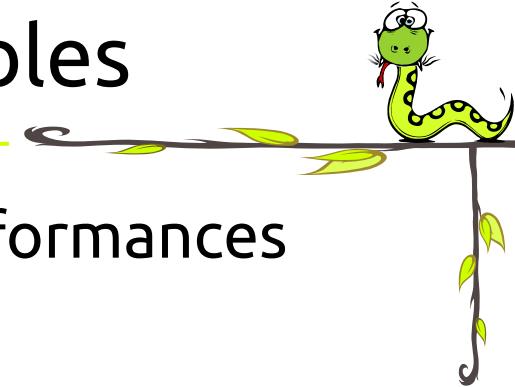
La loi d'Amdahl indique que tout programme possède une partie séquentielle et que tôt ou tard, c'est cette portion de programme qui limitera l'accélération qui peut en être faite. https://fr.wikipedia.org/wiki/Loi_d'Amdahl

Par exemple si votre programme dure 10 secondes sur une machine, et que 90 % de ce temps est passé dans une portion de code à scalabilité linéaire (parallélisable) et les 10 % restant dans une portion de code séquentielle incompressible, alors :

- Si on le fait tourner sur 9 processus, le gain sera de x5
 $1s + 9s / 9 = 2s$
- Au mieux le programme ne pourra jamais avoir une accélération supérieure à x10
 $1s + 9s/\infini = 1s$

Il est important de pouvoir identifier ses portions de code avant de se lancer tête baissée dans la parallélisation....

Identifier les portions de code parallélisables



Sous Unix il existe des commandes pour mesurer les performances d'un programme (profiling) :

- Gprof (C, Pascal or Fortran)
- Valgrind/callgrind pour tout exécutable

Sous Python 2 librairies sont disponibles

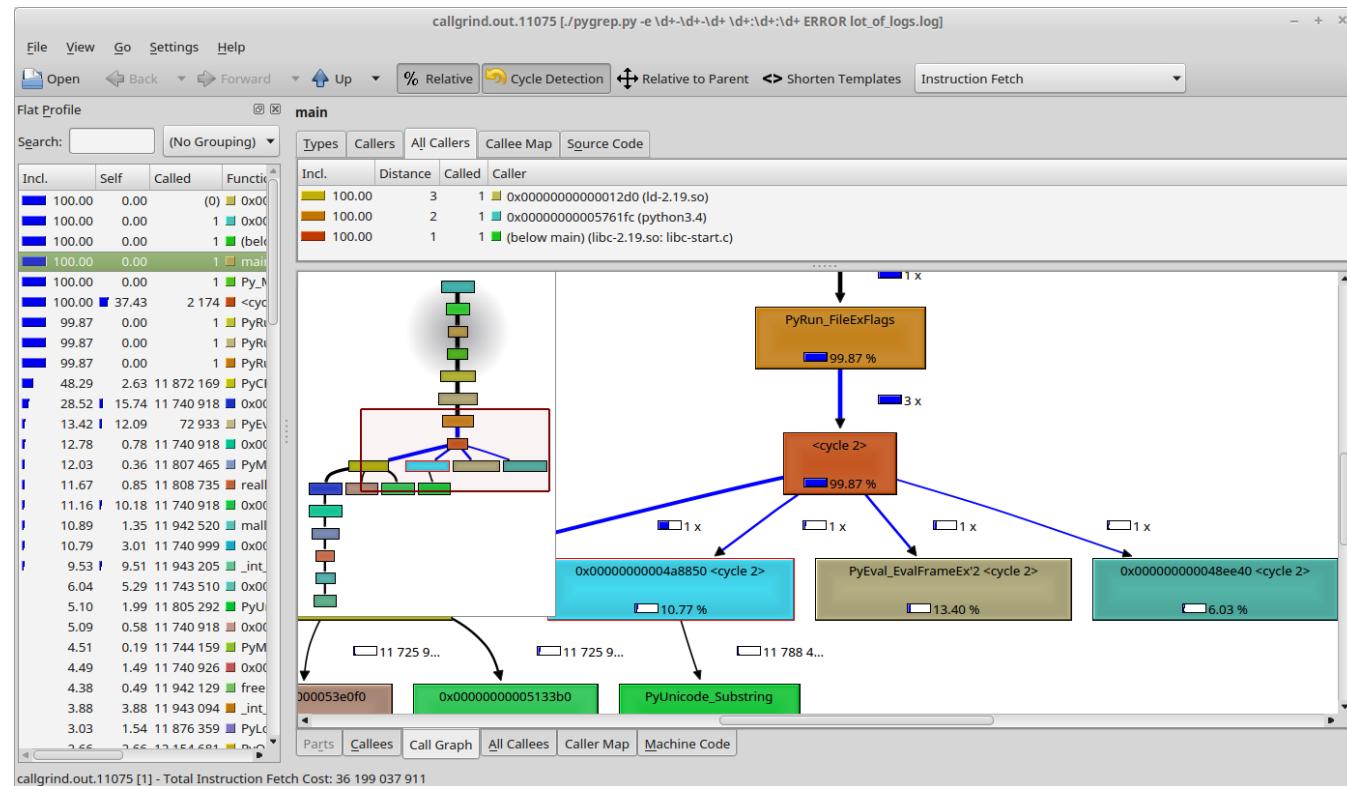
- `timeit`
<https://docs.python.org/3/library/timeit.html>
- `cProfile`
<https://docs.python.org/3/library/profile.html>

Identifier les portions de code parallélisables



La commande « valgrind » couplée à l'outil « callgrind » vous permet de générer un graphe des appels de votre programme et de visualiser :

- Le nombre de fois qu'une fonction est appelée
- Le temps passé dans chaque fonction

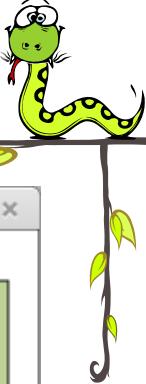


Présentation – Limites physiques du parallélisme



- Les processeurs actuels ont des fréquences supérieures au GigaHertz.
 - Dans les années 1980 la mémoire (< 1Mo en général) était intégrée au processeur et on y accédait en 1 cycle machine/1 instruction
 - Mais la fréquence des processeurs a cru d'environ 50 % par an contre une augmentation de la vitesse de lecture des mémoires de 7 % environ, laquelle est maintenant externe aux processeurs
- Fin 2015 les temps d'accès à une adresse mémoire en lecture ou écriture sont de l'ordre de 100 nanosecondes. Soit 100 cycles d'un processeur !
Pour pallier à cela :
 - Les processeurs modernes utilisent des caches locaux très rapides mais de capacité très réduite
 - <http://ecariou.perso.univ-pau.fr/cours/archi/cours-5-memoire.pdf>
 - Architecture et micro-architecture des processeurs

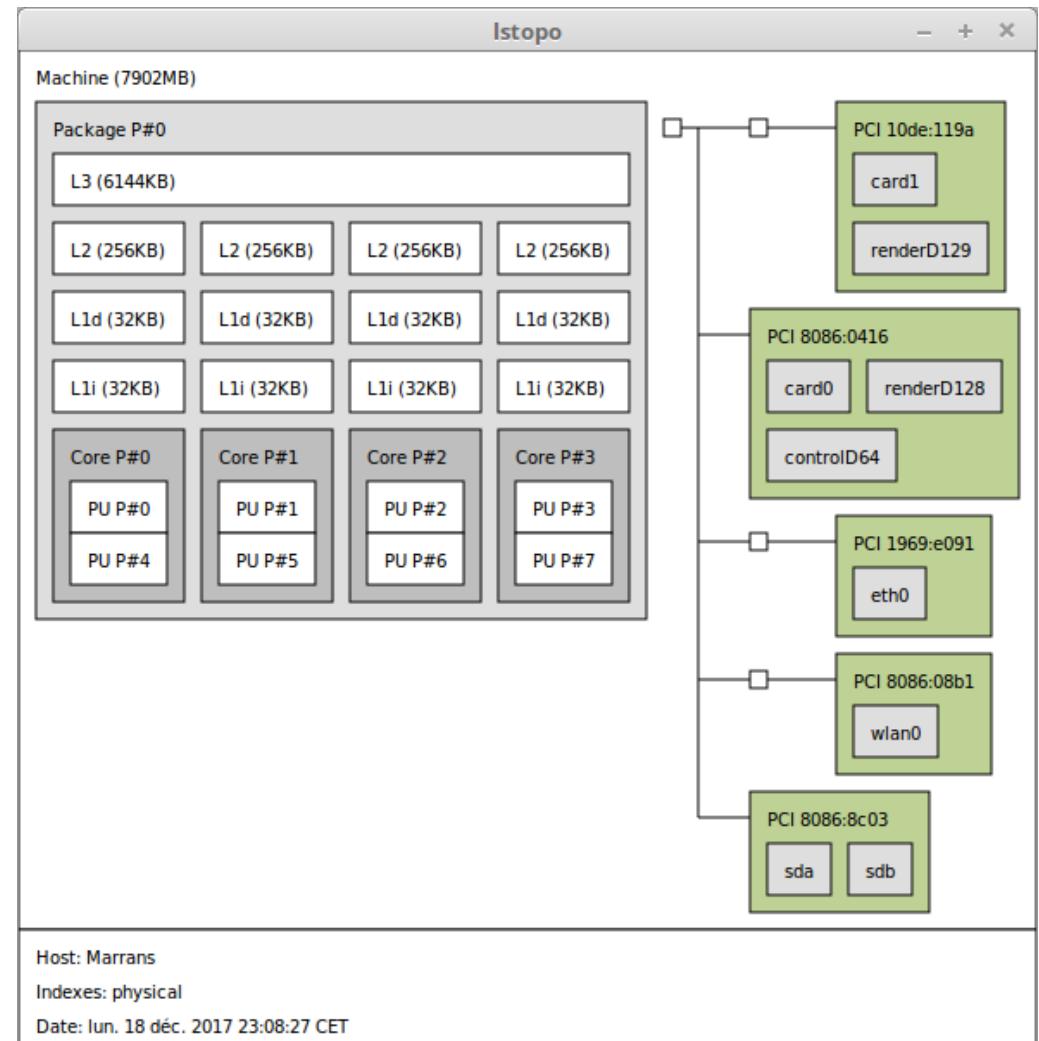
Affinité/Invalidation de cache



Pourquoi les gamers n'aiment pas l'hyperthreading et donc mon i7 ?

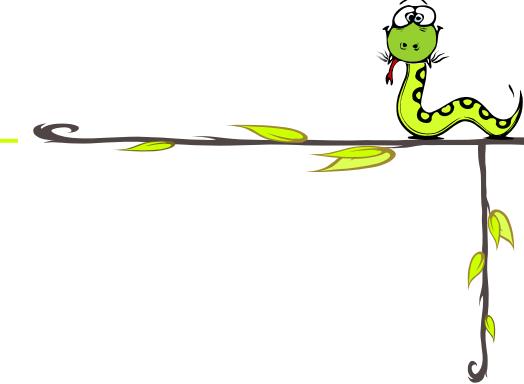
- Le cœur physique possède 2 hyperthreads
- Un seul peut s'exécuter à la fois
- Quand le premier est en attente d'une entrée sortie l'autre prend le relais
- Mais si le second a besoin d'accéder à une zone mémoire non présente dans le cache du premier
- Il va le vider et le recharger avec les nouvelles données (invalidation de cache)
- C'est lent car la mémoire est plus lente que le CPU
- Quand le premier reprendra la main il fera de même
- Les performances peuvent ainsi chuter terriblement !

La librairie [ATLAS](#) de calcul numérique conseille carrément de le désactiver !



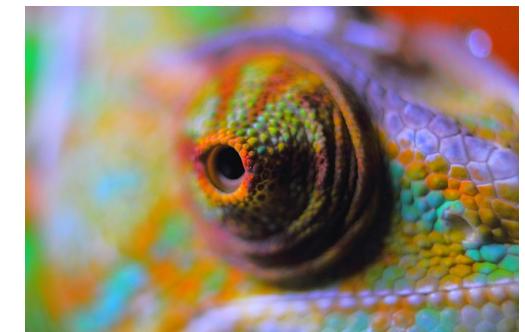
Architecture d'un processeur Intel i7 réalisée via la commande
bash\$ **hwloc-ls**

Les différentes formes du parallélisme



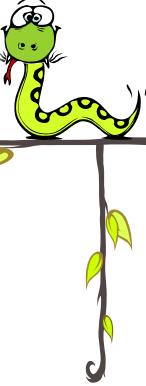
Il existe de nombreuses manières de paralléliser vos applications :

- **Dans un même programme**
 - En utilisant des jeux d'instructions dites vectorielles **SIMD** (Single Instruction Multiple Data) comme SSE/AVX/NEON
 - En utilisant des threads
- **Sur un même ordinateur**
 - En utilisant la programmation concurrente
 - En utilisant plusieurs processeurs/cœurs (multiprocessing/**OpenMP**/MPI)
- **Sur plusieurs ordinateurs**
 - Calcul distribué
 - Cloud computing
 - Grid computing

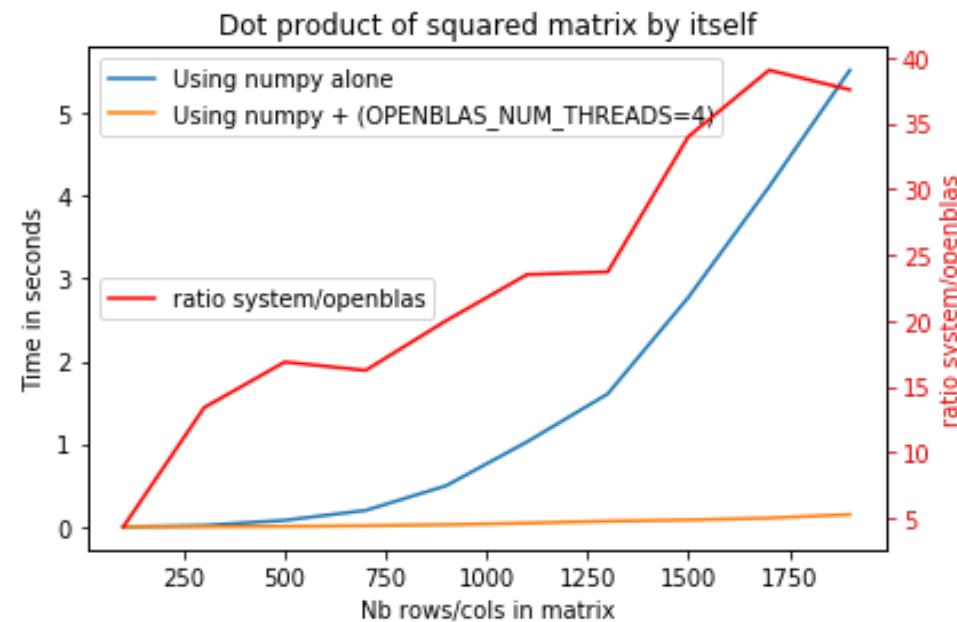


<http://semelin.developpez.com/cours/parallelisme/>

Démo : bien installer Numpy



- On pense souvent qu'installer numpy se limite à exécuter la commande « pip install numpy »
- Mais si vous ne disposez pas de librairies mathématiques vectorisées/parallélisées, vous allez passer à côté de l'essentiel : **La performance !**
- Il est donc conseillé d'installer Numpy avec une librairie comme :
 - [Atlas](#), [Blas](#), [Lapack](#)
 - [OpenBlas](#)
 - Intel [MKL](#)
 - AMD [ACML](#)

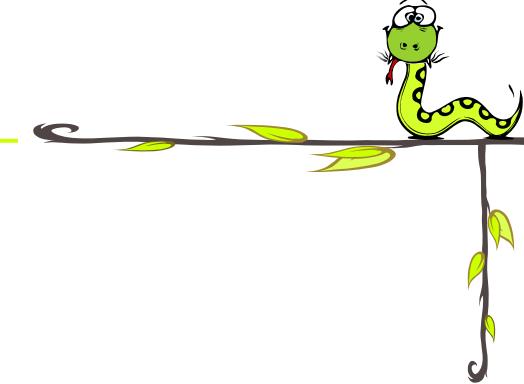




Programmation Parallèle en Python

Les bases algorithmiques

Parallélisme avec Python



Python offre en standard la possibilité de réaliser :

- De la programmation concorrente avec les librairies « `concurrent.futures` » et « `asyncio` »
<https://docs.python.org/3/library/asyncio.html>
<https://docs.python.org/3/library/concurrent.futures.html>
- De faire du multithreading avec la librairie « `threading` »
<https://docs.python.org/3/library/threading.html>
- De faire du multiprocessing avec la librairie du même nom
<https://docs.python.org/3/library/multiprocessing.html>
- De faire du calcul distribué avec la librairie « `multiprocessing` » et les objets « `Manager` »

Multithreading et Multiprocessing



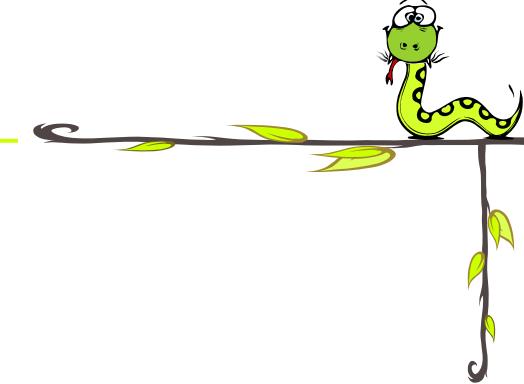
Les librairies pour le multithreading et multiprocessing ont une interface quasiment identique en Python. En renommant la librairie importée il devient ainsi possible de basculer de l'un à l'autre en ne modifiant qu'une ligne de code :

- `from threading import Thread as Task`
- `from multiprocessing import Process as Task`

Multithreading



Multithreading : Les bases



Utilisation de la librairie « threading » :

- `from threading import Thread as Task`
- **Créer un thread :**
`t = Task(target, args, kwargs, daemon)`
 - target : nom de fonction ou méthode
 - args : tuple contenant les arguments positionnels passés à la fonction
 - kwargs : dictionnaire contenant les arguments nommés passés à la fonction
 - Daemon : Si True, le thread est considéré comme un daemon.
Le programme peut s'arrêter même si le thread continue d'exister
- **Exécuter un thread :**
`t.start()`
- **Attendre la fin de l'exécution d'un thread :**
`t.join()`

Multithreading : Les bases



```
from threading import Thread as Task
import time

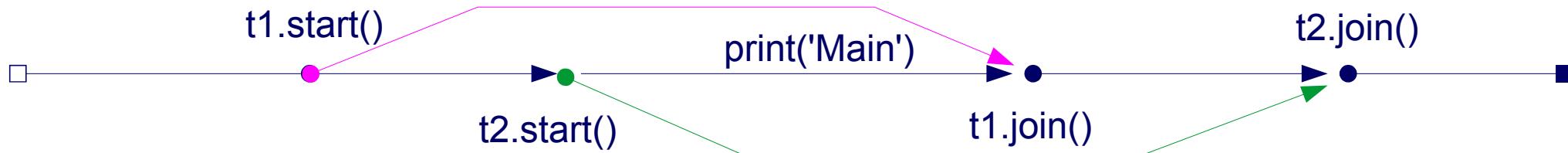
def f1(name):
    for ind in range(10):
        print("%s : %d" % (name, ind))
        time.sleep(1)

t1 = Task(target=f1, args=("Thread 1",))
t2 = Task(target=f1, args=("Thread 2",))
t1.start()
t2.start()

for ind in range(10):
    print("Main", ind)
    time.sleep(1)

# wait tasks to be finished
[ t.join() for t in (t1, t2) ]
print('Fin')
```

```
Thread 1 : 0
Thread 2 : 0
Main 0
Main 1
Thread 1 : 1
Thread 2 : 1
Main 2
Thread 1 : 2
Thread 2 : 2
Main 3
Thread 1 : 3
Thread 2 : 3
Main 4
Thread 2 : 4
Thread 1 : 4
...
```



Multithreading : Les verrous



```
from threading import Thread as Task

class BankAccount(object):
    def __init__(self, owner, initial_deposit):
        self.amount = initial_deposit
        self.owner = owner

    def __str__(self):
        return "%s: %s€" % (self.owner, self.amount)

    def virement(self, money):
        self.amount += money

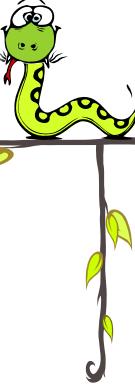
my_account = BankAccount("Me", 1.0)
print("Start:", my_account)

tasks = []
for ind in range(0, 100):
    t = Task(target=my_account.virement, args=(100,))
    t.start()
    tasks.append(t)

print('Waiting end of tasks')
[ t.join() for t in tasks]
print("End:", my_account)
```

Start: Me: 1.0€
Waiting end of tasks
End: Me: 10001.0€

Multithreading : Les verrous



Mais c'est bientôt Noël et les réseaux bancaires sont saturés...

- Modifions la méthode virement comme suit
- Puis relançons le programme

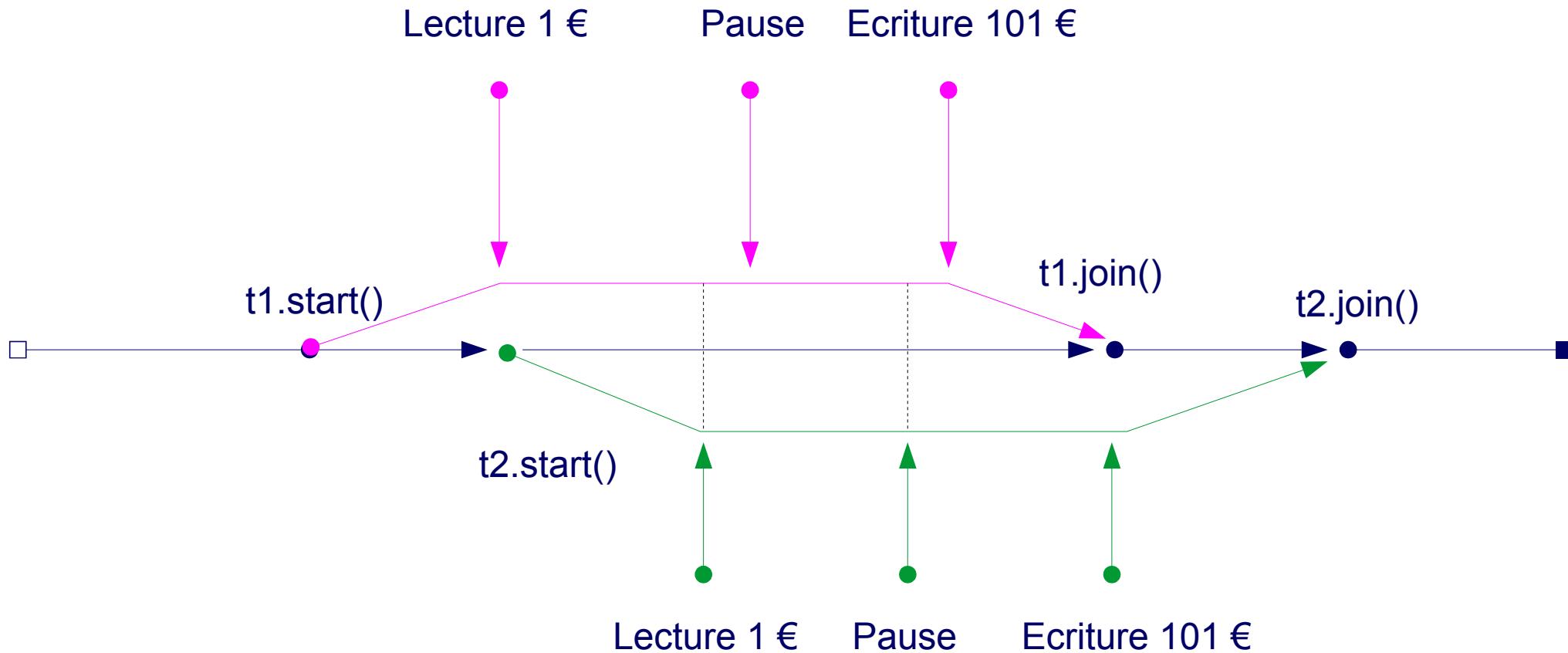
```
def virement(self, money):  
    # get current amount  
    current_amount = self.amount  
  
    # network slow time  
    sleep(0.01)  
  
    # doing computation  
    current_amount += money  
  
    # hard drive latency  
    sleep(0.01)  
  
    self.amount = current_amount
```

Start: Me: 1.0€
Waiting end of tasks
End: Me: 201.0€



*Mais qui a vampirisé toute ma
Précieuse monnaie ?*

Multithreading : Les verrous



Multithreading : Les verrous



Pour contourner ce problème on peut utiliser un verrou.

Le verrou n'est accessible par une seule tâche à la fois. Une fois pris le code qui suit son acquisition n'est exécutable que par la tâche jusqu'à ce qu'elle le libère :

- **Création d'un verrou :**

```
verrou = Lock()
```

- **Acquisition d'un verrou :**

```
verrou.acquire()
```

- **Libération d'un verrou :**

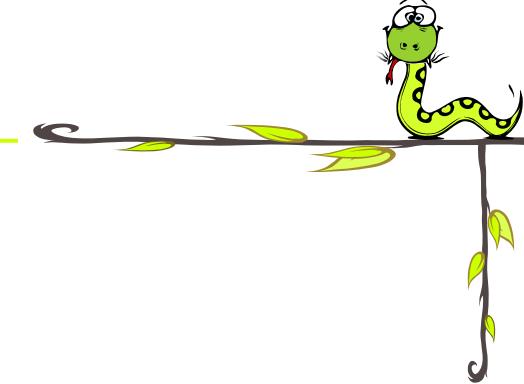
```
verrou.release()
```

- **Un verrou accepte les « contexts managers » (mot clef « with »)**

```
verrou = Lock()
with verrou:
    # vos traitements nécessitant une protection
    # contre des accès concurrents
```

```
verrou = Lock()
verrou.acquire()
try:
    # vos traitements nécessitant une protection
    # contre des accès concurrents
finally:
    verrou.release()
```

Multithreading : Les verrous



Avec l'ajout d'un verrou :

- La fonction virement n'est plus parallélisable
- Mais « my money is back ! »

```
from threading import Thread as Task, Lock

class BankAccount(object):
    def __init__(self, owner, initial_deposit):
        ...
        self.lock = Lock()

    def __str__(self):
        return "%s: %s€" % (self.owner, self.amount)

    def virement(self, money):

        with self.lock:
            current_amount = self.amount
            sleep(0.01)
            current_amount += money
            sleep(0.01)
            self.amount = current_amount
```

Start: Me: 1.0€
Waiting end of tasks
End: Me: 10001.0€

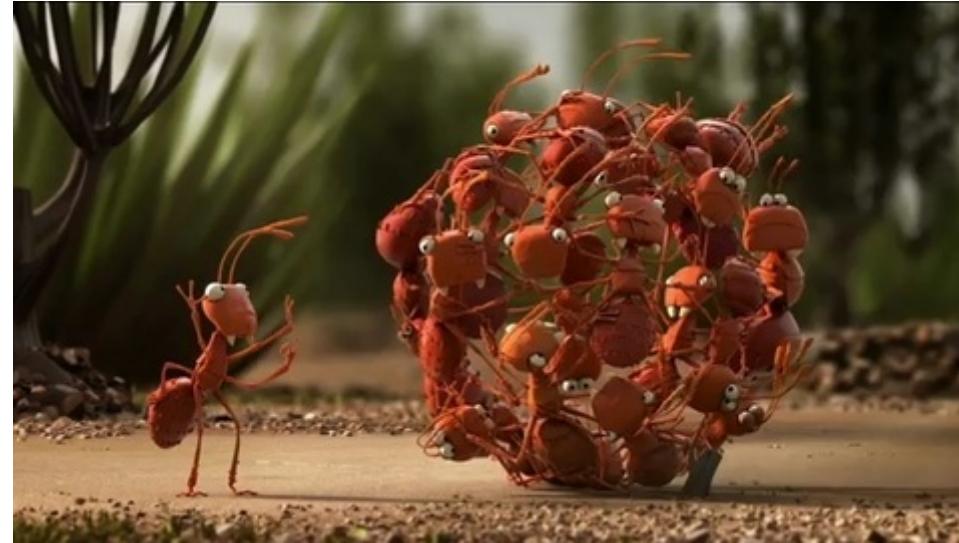
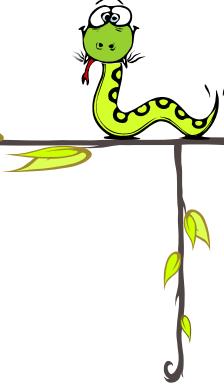
Multithreading : Autres fonctionnalités



Le module `threading` propose de nombreuses autres fonctionnalités :

- Verrous ré-entrants
- Sémaphores
- Files, Piles
- Timer
- Event
- Barrier
- Condition
- ...

Multiprocessing



L'union fait la force

Multiprocessing : Comme threading, ou presque...



Le module multiprocessing a une interface identique à celle du module threading.

De ce fait, pour passer d'un module à l'autre vous n'avez qu'une ligne de code à changer...

- `from threading import Thread as Task`
- **Devient**
`from multiprocessing import Process as Task`

Enfin, presque...

```
Start: Me: 1.0€  
Waiting end of tasks  
End: Me: 1.0€
```



*Les gremlins sont de retour !
Et ils sont encore plus méchants !*

Multiprocessing : Comme threading, ou presque...



Que s'est-il passé ?

- Quand vous exécutez un processus il crée une copie du programme principal, via un « fork » au niveau de l'OS
- Ceci duplique aussi les données de votre programme
- Et chaque process se retrouve avec sa propre copie du compte bancaire
- Cela complique un peu votre tâche
 - Il convient de mettre le compte bancaire en mémoire partagée
 - Vous êtes alors limités aux types du langage C
 - Par contre les variables en mémoire partagée possèdent leur propre verrou
 - Et une propriété « value » permettant la conversion avec les types Python

Multiprocessing : Comme threading, ou presque...



```
from multiprocessing import Process as Task
from multiprocessing.sharedctypes import Value
from ctypes import c_double

class BankAccount(object):
    def __init__(self, owner, initial_deposit):
        self.amount = Value(c_double, initial_deposit)
        self.owner = owner

    def __str__(self):
        return "%s: %s€" % (self.owner, self.amount.value)

    def virement(self, money):

        with self.amount:
            # get current amount
            current_amount = self.amount.value

            # network slow time
            sleep(0.01)

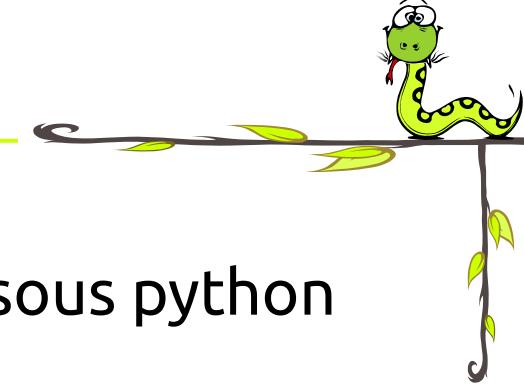
            # doing computation
            current_amount += money

            # hard drive latency
            sleep(0.01)

            self.amount.value = current_amount
```

Start: Me: 1.0€
Waiting end of tasks
End: Me: 10001.0€

Multithreading : Les limites



Afin de bien comprendre les limites du multithreading sous python réalisez les opérations suivantes :

- Exécutez la fonction `countdown()` avec une valeur très grande (100 millions) par exemple
 - 2 fois en série
 - Avec 2 threads en parallèle
 - Avec 2 process en parallèle
- En théorie les versions parallèles devraient demander 2 fois moins de temps que l'exécution en série si vous possédez une machine multicœurs/multiprocesseurs
- Mais ce n'est pas le cas avec le module `threading...`

```
def countdown(n):
    while n > 0:
        n -= 1
```

Multithreading : Les limites



```
def countdown(n):
    while n > 0:
        n -= 1

def serial(n, repeat=2):
    for ind in range(repeat):
        countdown(n)

def threads(n, repeat=2):
    tasks = []
    for ind in range(repeat):
        t = Thread(target=countdown, args=(n,))
        t.start()
        tasks.append(t)

    [t.join() for t in tasks]

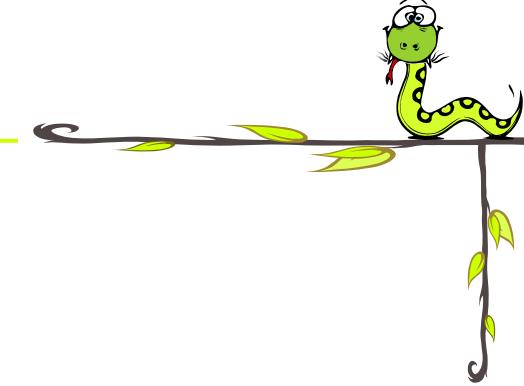
def processes(n, repeat=2):
    tasks = []
    for ind in range(repeat):
        t = Process(target=countdown, args=(n,))
        t.start()
        tasks.append(t)

    [t.join() for t in tasks]

N = 10**8
repeat = 2
```

Exécution en série
serial: 7.866269341997395
threads: 8.027223899996898
processes: 4.158569907005585

Multithreading : Comprendre les limites



Le multithreading avec Python ça semble bien.

Mais « **ça craint du boudin** » :

- En fait Python, n'est pas capable d'utiliser l'architecture multicœurs des processeurs modernes dans son implémentation multithread.
- L'interpréteur Python ne sait exécuter qu'un seul thread à la fois
- C'est uniquement quand un thread est en attente sur une opération I/O que l'interpréteur changera de thread.
- Pour bien comprendre ces limites vous pouvez consulter cet excellent document de David Beazley qui est devenu la référence sur le sujet :

<http://www.dabeaz.com/python/UnderstandingGIL.pdf>

Multithreading : Savoir à quoi s'en tenir



- Les threads sont intéressants avec Python tant que vos traitements réalisent beaucoup d'opérations d'entrées/sorties.
- Si vos threads font énormément de calculs, ils ne seront pas efficaces. Ils peuvent même être plus lents.
- C'est un problème interne à l'implémentation du langage
 - Jusque là le problème a été maintenu en connaissance de causes pour ne pas pénaliser les ordinateurs monoprocesseur/monocoeur.
 - Une solution technique compatible avec ces derniers et permettant d'utiliser le potentiel des nouvelles architectures a apparemment été trouvée et sera implémentée dans une prochaine version



http://python-notes.curiosefficiency.org/en/latest/python3/multicore_python.html

<https://wiki.python.org/moin/GlobalInterpreterLock>

Multiprocessing : Process Pools



Les « pools » de processus permettent d'assigner un ensemble de tâches à un groupe de processus :

- L'objet « Pool » gère un nombre donné de processus auxquels il distribuera les tâches qu'il recevra.
Si vous ne fournissez pas ce nombre il utilisera la valeur de `os.cpu_count()`
- Il peut tuer un worker après que celui-là ait traité un certain nombre de tâches pour en redémarrer un tout frais
- Il peut demander l'exécution les tâches de manière *synchrone* (et rester bloqué en attente du résultat) ou de manière *asynchrone* (ce qui est préférable pour la programmation parallèle)
- Elle peut s'utiliser avec un « context manager » depuis Python 3.3

Multiprocessing : Process Pools



Le constructeur de la classe « Pool » accepte les paramètres suivants :

- « processes »
Le nombres de processus utilisés pour traiter les tâches
- « initializer », « initargs »
Une fonction utilisée pour initialiser les processus créés et les paramètres qui lui sont passés
- « maxtasksperchild »
Le nombre maximum de tâches qu'un processus pourra exécuter avant d'être tué et remplacé

```
# 3 processus pour traiter les tâches,  
# une fonction init_process pour les initialiser  
# et qui ne prend pas de paramètres  
# 2 tâches exécutées par processus  
pool = Pool(3, init_process, (), 2)
```

Multiprocessing : Process Pools



Elle possèdent différentes méthodes pour gérer l'ensemble des processus fils :

- « close »
Interdit l'ajout de nouvelles tâches.
Une fois les tâches en cours terminées, les workers s'arrêteront
- « join »
Attend la fin des processus workers. « close » doit avoir été appelée au préalable
- « terminate »
Arrête tous les workers en cours, sans qu'ils terminent leurs tâches

Multiprocessing : Process Pools - map



La méthode « `map` » se comporte comme la fonction « `map` » de python en appelant une fonction sur un ensemble de valeurs. Chaque appel de fonction est réparti entre les différents workers.

- C'est une fonction synchrone (bloquante)
- Elle accepte un paramètre « `chunksize` » qui permet d'envoyer plusieurs valeurs à calculer aux différents processus à chaque appel
- Le résultat retourné est ordonné, comme pour la fonction « `map` »

Multiprocessing : Process Pools - map



```
from multiprocessing import Pool

from time import sleep
import os
from functools import reduce

def carre(x):
    print("Calcul du carré de %s par proccesus %s" % (x, os.getpid()))
    sleep(1)
    return (x, x * x)

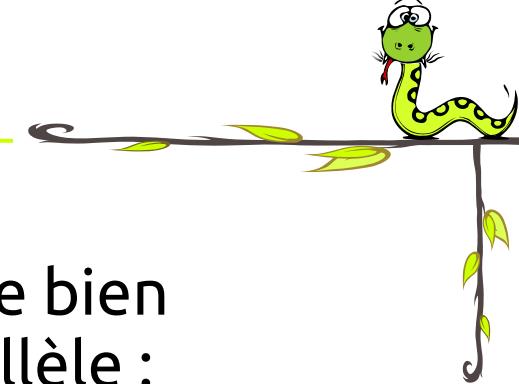
def init_process():
    print("Initializing process %s" %(os.getpid(),))

if __name__ == "__main__":
    pool = Pool(processes=3, initializer=init_process, initargs=(),
                maxtasksperchild=2)

    with pool:
        print("Running map")
        data = pool.map(carre, range(20))
        print("End map")
        print(data)
```

```
Initializing process 6140
Initializing process 6141
Initializing process 6142
Running map
Calcul du carré de 0 par proccesus 6140
Calcul du carré de 2 par proccesus 6141
Calcul du carré de 4 par proccesus 6142
Calcul du carré de 5 par proccesus 6142
Calcul du carré de 3 par proccesus 6141
Calcul du carré de 1 par proccesus 6140
Calcul du carré de 6 par proccesus 6141
[...]
Initializing process 6146
Calcul du carré de 12 par proccesus 6146
Initializing process 6147
Calcul du carré de 14 par proccesus 6147
Initializing process 6148
Calcul du carré de 16 par proccesus 6148
Calcul du carré de 13 par proccesus 6146
Calcul du carré de 15 par proccesus 6147
Calcul du carré de 17 par proccesus 6148
Calcul du carré de 18 par proccesus 6146
Calcul du carré de 19 par proccesus 6146
End map
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25),
(6, 36), [...](18, 324), (19, 361)]
```

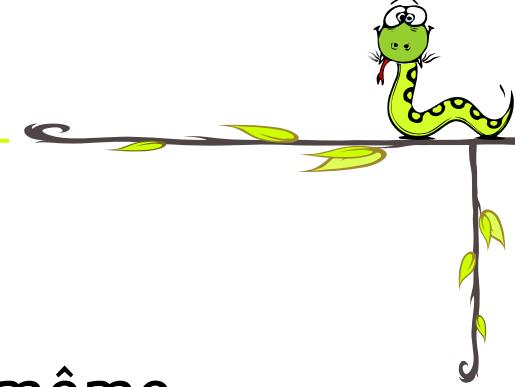
Multiprocessing : Compléments



Le pool de processus est la notion qu'il est primordial de bien comprendre pour se lancer dans l'univers du calcul parallèle :

- La plupart des librairies de calcul distribué sont basées sur ce modèle
- La librairie propose de nombreuses autres fonctions :
 - map_async
 - apply, apply_async
 - imap, imap_async
 - starmap, starmap_async

Multiprocessing : Proxy et managers



Les managers permettent plusieurs choses :

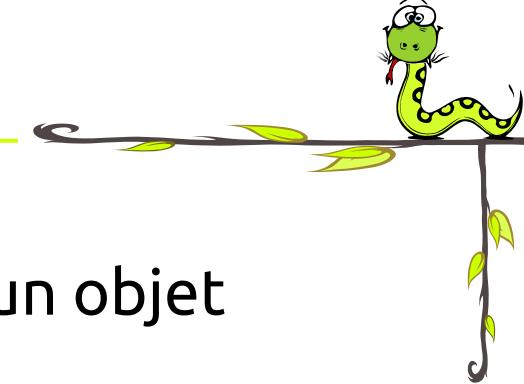
- Partager des objets entre plusieurs processus sur la même machine
- Partager des objets entre plusieurs processus sur fonctionnant en réseau sur plusieurs machines.

Ils sont très puissants.

Ces objets sont manipulés via des proxys.

Tous les types de structures Queue, Lock, Value, Barrier, et autres peuvent être « proxyfiés » via un manager

Multiprocessing : Proxy et managers



La classe « Manager » du module multiprocessing crée un objet « multiprocessing.managers.SyncManager »

- Il implémente les verrous, variables, et autres objets de synchronisation
- Il permet de créer des listes et dictionnaires
- <https://docs.python.org/3.5/library/multiprocessing.html#multiprocessing.managers.SyncManager>

```
from multiprocessing import Manager  
manager = Manager()  
q = manager.Queue()
```

Multiprocessing : Managers



Une des grandes forces des managers est de permettre à des processus répartis sur différentes machines d'échanger des données.
Pour cela différentes méthodes existent :

- La classe `BaseManager` accepte 2 paramètres « `address` » et « `authkey` » :
 - `Address` contient l'adresse du Manager sur lequel se connecter ou sur laquelle il écoutera. `None` correspond à une écoute sur toutes les IP
 - `Authkey` est la clef d'authentification qui devra être utilisée par les clients
- « `get_server` »
Cette méthode retourne le serveur qui contrôle le manager pour le lancer (process serveur) ou s'y connecter (process client)

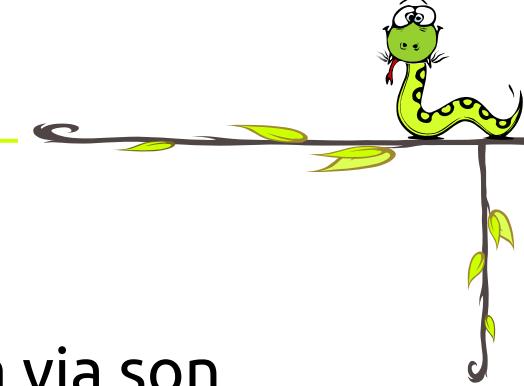
Multiprocessing : Managers



Une des grandes forces des managers est de permettre à des processus répartis sur différentes machines d'échanger des données.
Pour cela différentes méthodes existent :

- «serve_forever»
Appliquée sur le serveur retourné par «get_server» exécute ce dernier
- «connect »
Permet aux clients de se connecter à un manager
- « shutdown »
Arrête le manager et son serveur

Multiprocessing : Managers



- «register»

Permet d'indiquer quels objets le manager exposera via son interface.

Les objets peuvent être des variables globales ou des attributs de la classe Manager

Elle accepte plusieurs paramètres dont :

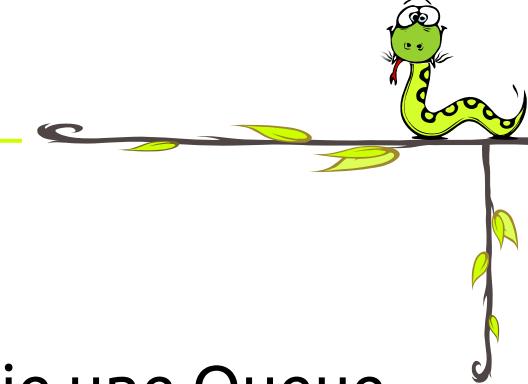
- « typeid »

Le nom par lequel les clients accèderont à la donnée

- « callable »

Un objet callable (généralement une fonction) retournée lorsque les clients utiliseront manager. « typeid »

Multiprocessing : Managers



La classe QueueManager.

Elle hérite tout simplement de BaseManager et instancie une Queue sous la forme d'une variable de classe.

Mais l'objet Queue aurait pu être une variable globale du module.

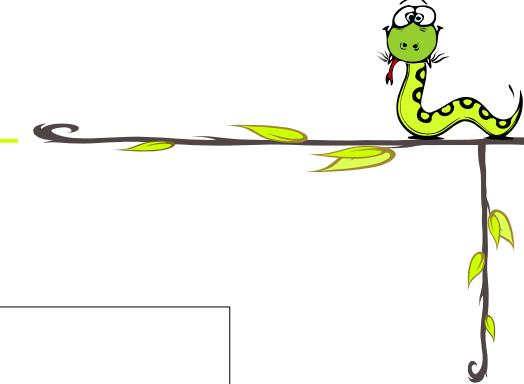
L'astuce réside ici dans le fait que pour retourner la « Queue » nous créons une fonction lambda, car l'objet Queue ne propose pas de fonction pour se retourner lui-même.

```
from multiprocessing import managers, Queue

class QueueManager(managers.BaseManager):
    queue = Queue()

QueueManager.register('get_queue', callable=lambda: QueueManager.queue)
```

Multiprocessing : Managers - TP



Le script du server.

```
from QueueManager import QueueManager

manager = QueueManager(address='', 8000), authkey=b'secret')
server = manager.get_server()
server.serve_forever()

print("done")
```

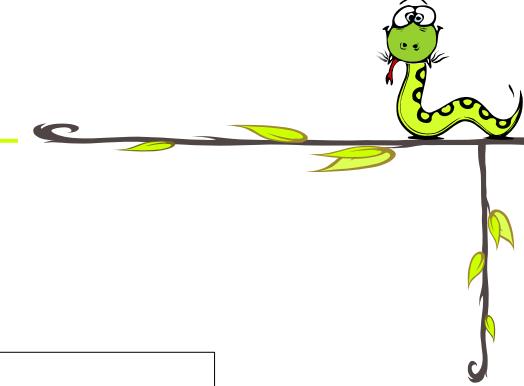
Le script du provider.

```
from QueueManager import QueueManager

m = QueueManager(address='127.0.0.1', 8000), authkey=b'secret')
m.connect()
q = m.get_queue()
for i in range(10):
    q.put("Task %s" % i)

print("Provider done")
```

Multiprocessing : Managers - TP



Le script du worker.

```
from QueueManager import QueueManager
from queue import Empty

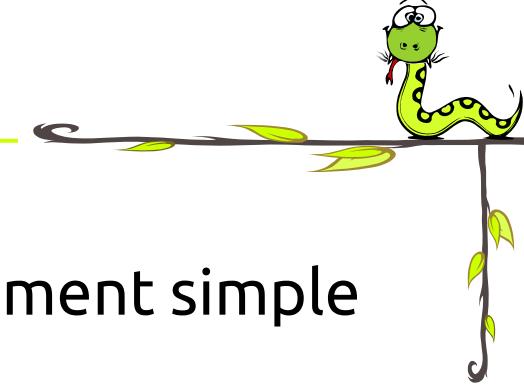
m = QueueManager(address=('127.0.0.1', 8000), authkey=b'secret')
m.connect()

q = m.get_queue()

try:
    while True:
        data = q.get(timeout=4)
        print("processing %s" % data)
except Empty:
    pass

print("Client done")
```

Conclusion



- Le parallélisme avec Python est très riche et relativement simple
- Le multithreading est clairement à éviter
- Le multiprocessing, les managers et pool de workers sont les éléments qui vous permettront de véritablement paralléliser votre application à un niveau industriel
- Les grandes difficultés du parallélisme sont :
 - Une gestion asynchrone des traitements qui impose de repenser la façon de concevoir ses programmes
 - Les problèmes de partage de données entre processus
 - Les problèmes de synchronisation des processus
 - Pour cela différentes recommandations sont fournies dans la documentation de Python :
<https://docs.python.org/3.5/library/multiprocessing.html#programming-guidelines>

Conclusion



- Mais la grande force de Python est de disposer de projets reconnus proposant des solutions clefs en mains très puissantes permettant de mettre en œuvre une infrastructure de « programmation parallèle » de très haut niveau :

<https://wiki.python.org/moin/ParallelProcessing>



Paysage du calcul parallèle en Python

- **Compilation**

Générer un code directement exécutable par la machine hôte.

- **Programmation concurrence/asynchrone**

Exécuter des fonctions en alternance lorsqu'elles font des entrées/sorties

- **Multithreading**

Exécuter des fonctions en parallèle dans un même programme

- **Grid Computing**

Du calcul distribué utilisant différents réseaux, architectures généralement hétérogènes et délocalisés : exemples [seti@home](http://setiathome.berkeley.edu) ou le LHC

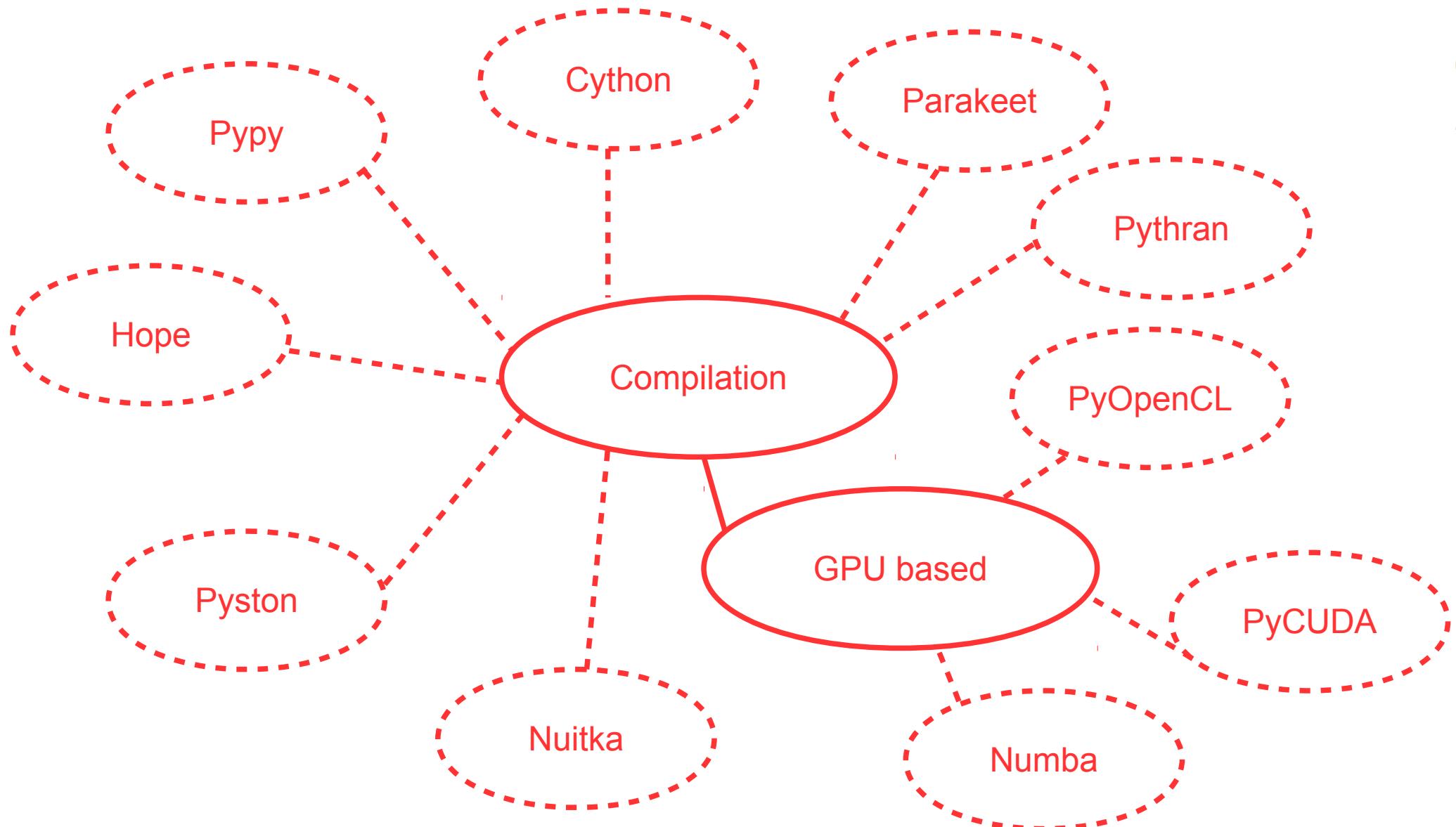
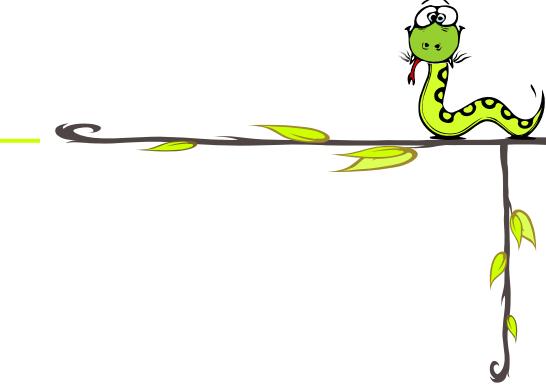
- **Cloud computing**

Du calcul distribué dans le web

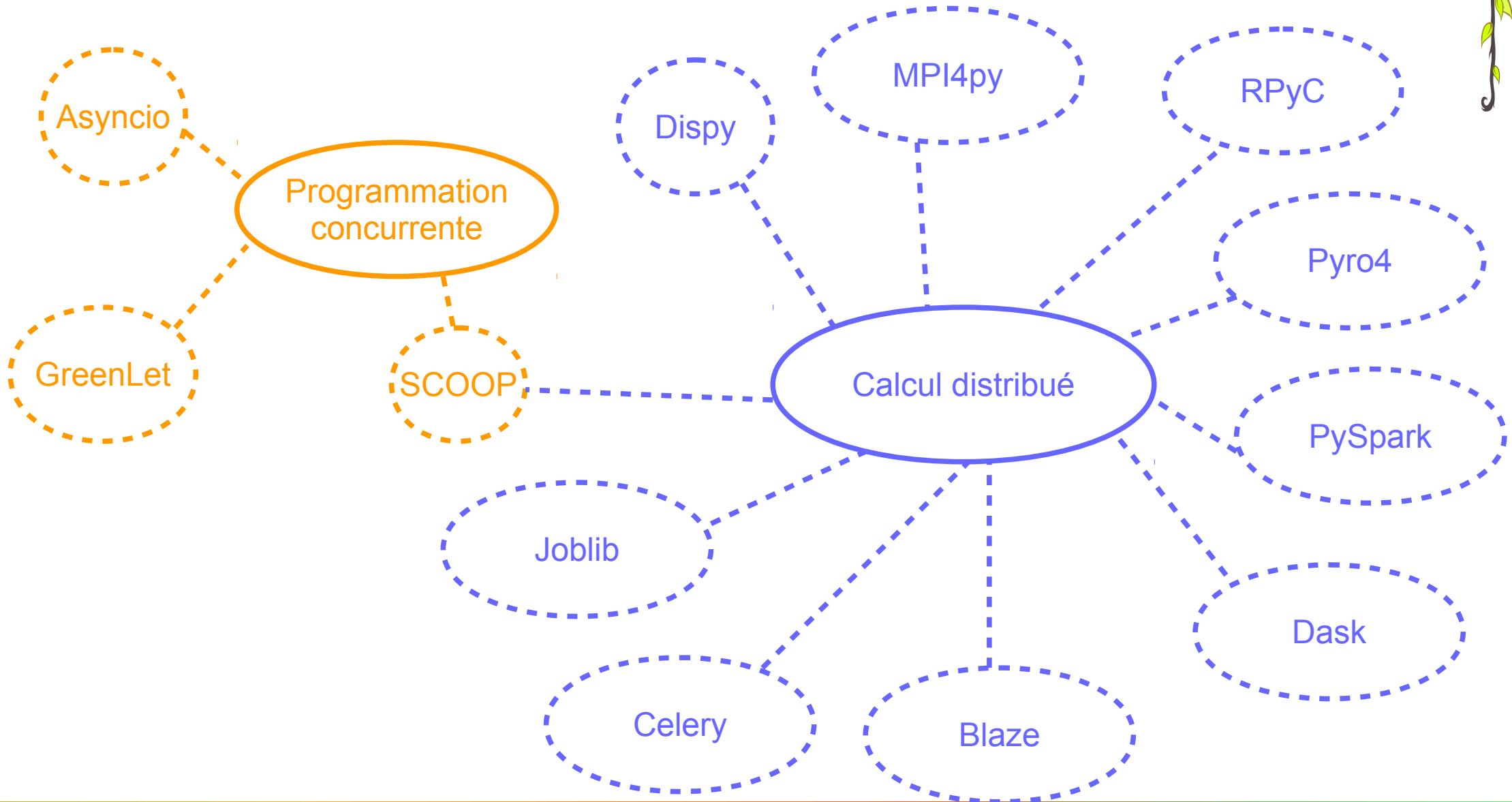
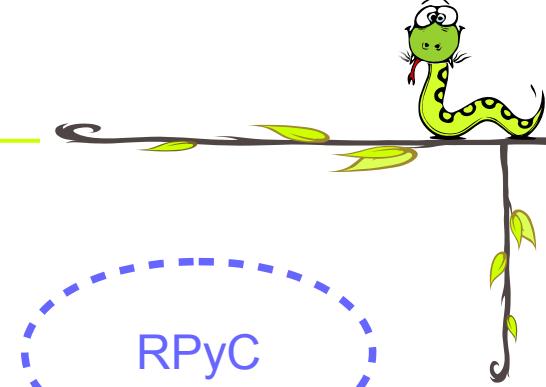
- **Calcul distribué**

Répartir des calculs sur plusieurs processeurs d'une même machine ou sur plusieurs machines

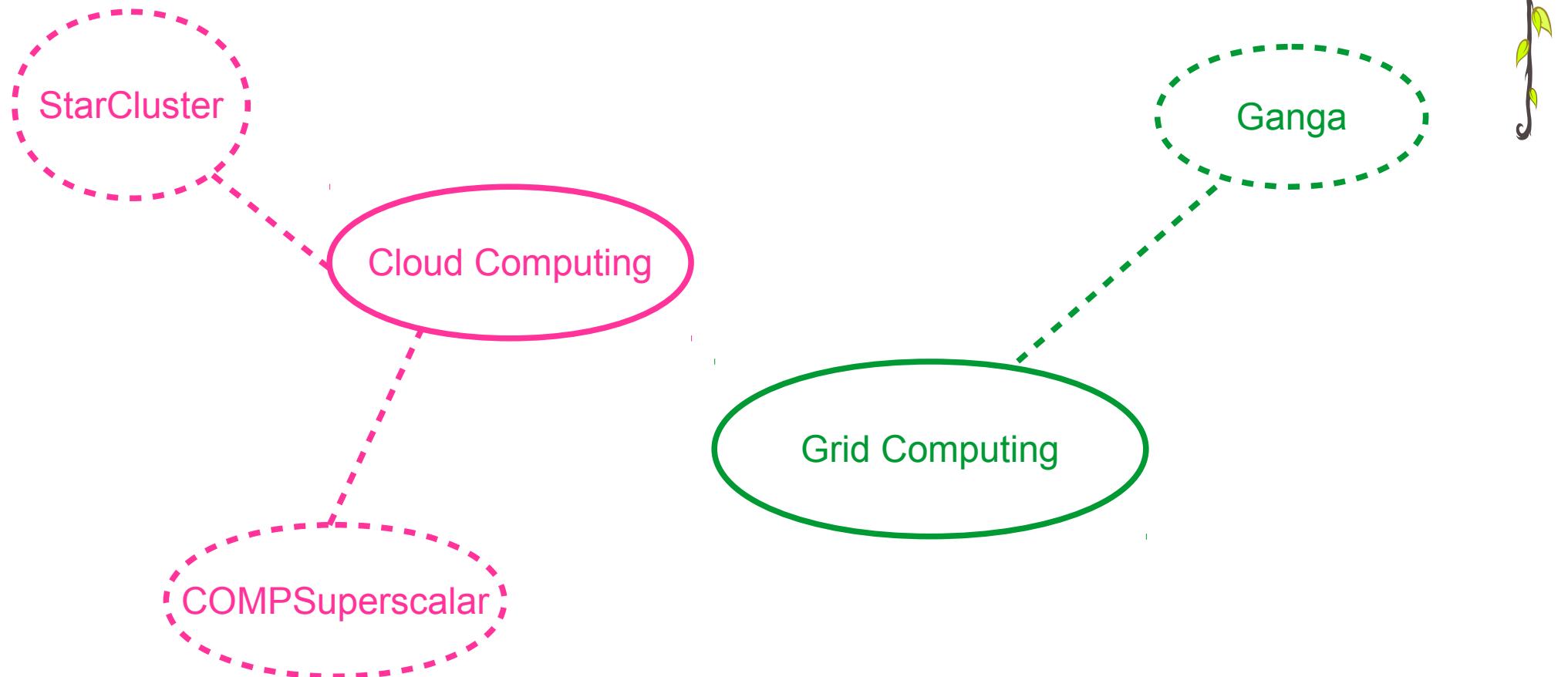
Paysage des librairies de calcul parallèle



Paysage des librairies de calcul parallèle



Paysage des librairies de calcul parallèle

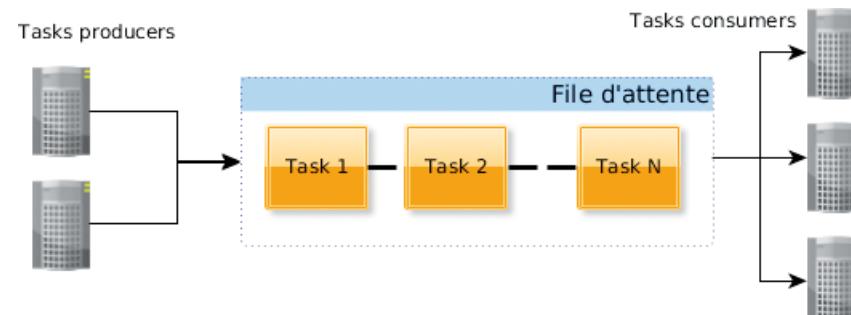


Celery

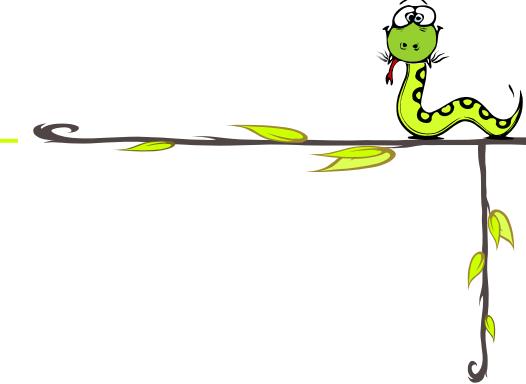


Celery

- Celery est un projet de calcul distribué en Python
<http://www.celeryproject.org/>
 - Contrairement à la plupart des autres projets il se contente d'implémenter les workers
 - Pour la distribution des tâches il s'appuie sur des serveurs utilisant le protocole AMQP
 - Par défaut il préconise d'utiliser le serveur RabbitMQ
- RabbitMQ
<http://www.rabbitmq.com/>
 - RabbitMQ est un serveur développé en Erlang et implémentant le protocole AMQP
 - Il propose des API pour de nombreux langages dont Python



Celery - Installation



- Installation de Rabbit MQ

```
$ sudo apt-get install rabbitmq-server
```

<http://www.rabbitmq.com/install-debian.html>

- Installation de Celery
Dans votre environnement virtuel Python

```
$ pip install celery
```

Celery - Vérification de l'installation



- Rabbit MQ

```
$ which rabbitmqctl
/usr/sbin/rabbitmqctl
$ sudo rabbitmqctl status
Status of node rabbit@Marrans ...
[{:pid, 7361},
 {:running_applications, [{:rabbit, "RabbitMQ", "3.2.4"},
                           {mnesia, "MNESIA CXC 138 12", "4.11"},
                           {os_mon, "CPO CXC 138 46", "2.2.14"},
                           {xmerl, "XML parser", "1.3.5"},
                           {sasl, "SASL CXC 138 11", "2.3.4"},
                           {stdlib, "ERTS CXC 138 10", "1.19.4"},
                           {kernel, "ERTS CXC 138 10", "2.16.4}]},
 {:os, {unix, linux}},
 {:erlang_version, "Erlang R16B03 (erts-5.10.4) [source] [64-bit] [smp:8:8] [async-
 threads:30] [kernel-poll:true]\n"},
 {:memory, [{:total, 36438616},
            {:connection_procs, 2704},
            {:queue_procs, 5408},
            {:plugins, 0},
            {:other_proc, 13607208},
            {:mnesia, 60496},
            ...]}
```

Celery - Vérification de l'installation



- **Celery**

```
$ which celery  
/path/to/anaconda3/venv/bin/celery  
$ celery --help  
Usage: celery <command> [options]
```

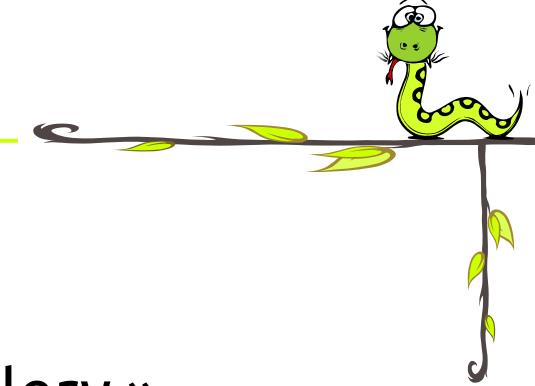
Show help screen and exit.

Options:

```
-A APP, --app=APP      app instance to use (e.g. module.attr_name)  
-b BROKER, --broker=BROKER  
                      url to broker. default is 'amqp://guest@localhost//'  
--loader=LOADER       name of custom loader class to use.  
--config=CONFIG        Name of the configuration module  
--workdir=WORKING_DIRECTORY  
                      Optional directory to change to after detaching.  
...  
----- Commands -----
```

```
+ Main:  
|   celery worker  
|   ...
```

Première application



Pour créer une application Celery il convient :

- 1) De créer un script Python qui instancie la classe « Celery »
- 2) Créer une tâche liée à cette instance
- 3) Démarrer votre instance de Celery, appelée application, avec la commande « celery »



Première application – étape 1



Créer l'application Celery.

Beaucoup de paramètres peuvent être spécifiés lorsque l'on démarre une application Celery, mais tous ont une valeur par défaut, lesquelles suffisent bien souvent.

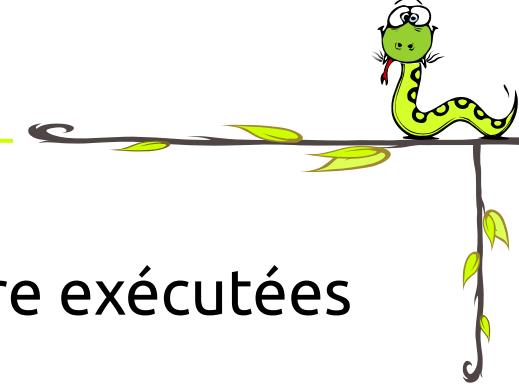
application.py

```
from celery import Celery  
  
the_app = Celery('MonApplication')
```

L'application a pour charge :

- De gérer le pool de processus (workers) qui traiteront les tâches
- Ajouter les tâches dans les files d'attentes du broker
- Distribuer ces tâches auprès des workers
- Envoyer les résultats vers le backend

Première application – étape 2



Il convient ensuite de définir les tâches qui pourront être exécutées par l'application

Définir une tâche revient à créer une fonction et lui affecter un décorateur du nom de l'instance celery suivi de « @<instance>.task »

application.py

```
from celery import Celery

the_app = Celery('MonApplication')

@the_app.task
def add(x, y):
    print("Doing task %s + %s" % (x, y))
    return x + y
```

Première application – Exécution de tâches



Maintenant que l'application fonctionne, il convient de lui soumettre des tâches.

Pour cela il suffit d'appeler les fonctions déclarées comme tâches

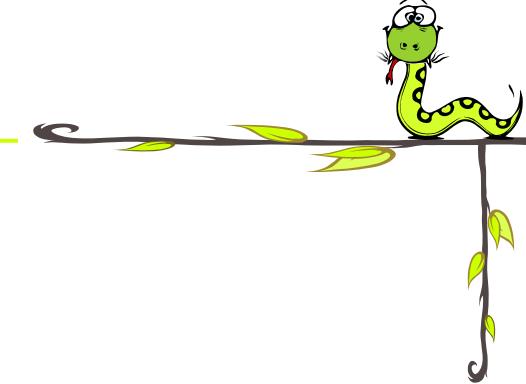
producer.py

```
from celery1.application import add  
  
for i in range(10):  
    add.delay(i, i)
```

\$ python producer.py

```
[2016-01-22 14:57:49,268: INFO/MainProcess] Received task: celery1.application.add[eaea7701-d280-4acf-be9c-c0c65341ac48]  
[2016-01-22 14:57:49,268: INFO/MainProcess] Received task: celery1.application.add[f02d9cdb-46f0-4e05-9082-99e8837d41da]  
...  
[2016-01-22 14:57:49,272: WARNING/Worker-2] Doing task 0 mul by 0  
[2016-01-22 14:57:49,272: WARNING/Worker-5] Doing task 1 mul by 1  
[2016-01-22 14:57:49,272: WARNING/Worker-8] Doing task 3 mul by 3  
[2016-01-22 14:57:49,272: WARNING/Worker-7] Doing task 2 mul by 2  
[2016-01-22 14:57:49,272: WARNING/Worker-6] Doing task 6 mul by 6  
...
```

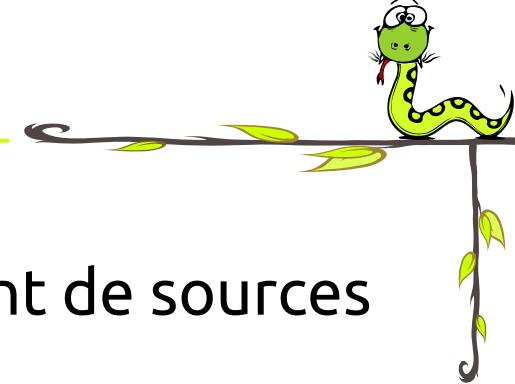
Questions



Merci de votre attention.

Des questions ?

Source des images

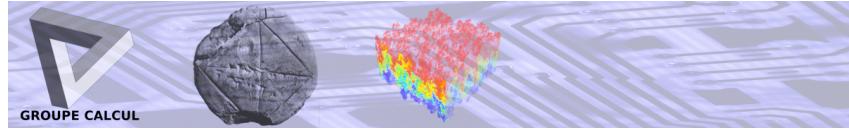


Les images utilisées dans cette présentation proviennent de sources libres :

- Wikipédia/Wikimedia
<http://www.wikipedia.fr>
- Pixabay
<https://pixabay.com>



Présentation du Parallélisme et mise en œuvre avec Python



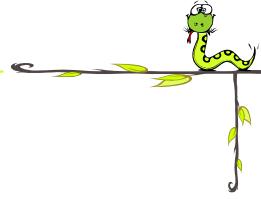
Groupe Calcul du CNRS
Mardi 19 décembre 2017
Rennes - IRMAR

Présentation des différents concepts du parallélisme
et du paysage des bibliothèques de calcul distribué en Python

Votre interlocuteur

- Gaël Pegliasco <gael@pegliasco.com>
- Développe en Python depuis 2004
- Applications web : Pendant 10 ans Zope, Plone, Django
- Calcul scientifique : Depuis 3-4 ans
Numpy, Scipy, Pandas, Matplotlib, Celery, Scikit-Learn, ...
- Sociétés Makina Corpus et Cogitec

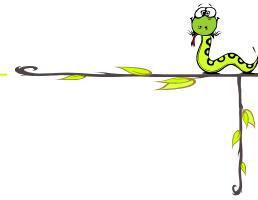
**Il est gentil et aura le plaisir de vous guider
dans la découverte de l'écosystème du calcul parallèle
avec Python**



Présentation des différents concepts du parallélisme
et du paysage des librairies de calcul distribué en Python

Page 2

Plan de la présentation



- ***Les concepts du parallélisme***
 - *Mécanismes et limites*
 - *Les différentes formes du parallélisme*
- ***Les bases de la programmation parallèle en Python***
 - *Asyncio*
 - *La librairie threading*
 - *La librairie multiprocessing*
- ***Présentation du paysage des librairies de calcul parallèle en Python***
 - *Vue d'ensemble*
 - *Celery, Dask, Numba*

Présentation des différents concepts du parallélisme
et du paysage des librairies de calcul distribué en Python

Page 3



Calcul parallèle



Sunway TaihuLight, le plus puissant supercalculateur au monde

- 10 649 600 coeurs (41000 puces de 260 coeurs à 1.45GHz)
- 93.01 petaflops/s (Linpack test)
- 93 millions de milliards d'opérations en virgule flottante par seconde
- **Attention**, la norme IEEE754 n'est pas d'une précision fabuleuse
$$>>> 0.1 + 0.1 + 0.1 - 0.3 == 0$$
- Consommation électrique 15 371 KW par seconde
- *C'est environ la consommation totale annuelle d'une maison de 100m² de 4 personnes*
- 20^{ème} au top500 des green supercomputer
- Et sur quels OS fonctionnent les supercalculateurs ?

Source <https://www.top500.org/lists/2017/11>
Novembre 2017

Kalray MPPA2®-256 (Boston), probablement le CPU le plus parallélisé au monde
<http://www.kalray.eu/kalray/products/>

- 288 coeurs et 128 crypto co-processeurs
- 4.5 GigaFlops
- 64-bit Very Long Instruction Word (VLIW)
- 600 MHz
- Consommation électrique 1 Watt
- Performances comparables aux systèmes ASICs



Présentation des différents concepts du parallélisme
et du paysage des bibliothèques de calcul distribué en Python

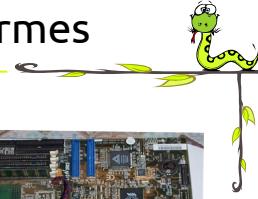
Présentation Parallélisme : Pourquoi Paralléliser ?



Gordon Earle Moore

- Une des « *Loi de Moore* » stipule que le nombre de transistors sur les microprocesseurs double tous les 2 ans.
- Une formulation plus commune de cette loi indique que la fréquence (et donc la puissance de calcul) des processeurs double tous les 18 mois.
- Or bien qu'en 2006 IBM et Georgia Tech aient décroché un record de vitesse d'horloge à 500Ghz à -268°C et 350Ghz à température ambiante, depuis 2004 la fréquence des processeurs grand public tend à stagner en raison de problèmes de bruit parasites et de dissipation thermique
- Pour maintenir cette augmentation de la puissance de calcul des ordinateurs la parallélisation est devenue une solution incontournable.

Présentation – Parallélisme : Quelques termes



- **Multiprocesseurs**

C'est l'utilisation d'au moins 2 processeurs (CPU) à l'intérieur d'un même ordinateur.



- **Processeur multicœurs**

C'est un processeur possédant plusieurs cœurs physiques/unités de calculs fonctionnant simultanément.

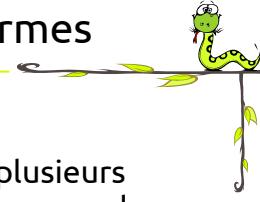


Processeur Quad-core AMD

- **Multiprocessing**

On utilise ce terme quand un système d'exploitation est capable d'exécuter plusieurs programmes en parallèle.

Présentation – Parallélisme : Quelques termes



- **Multitâche**

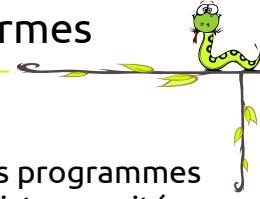
En informatique, le multitâche est une méthode où plusieurs tâches, aussi appelées processus, partagent des ressources de traitement communes comme une unité centrale.

Un système d'exploitation est multitâche lorsqu'il permet de passer d'une tâche à une autre rapidement. Soit en les exécutant parallèlement soit l'une après l'autre pour donner l'impression de simultanéité (cas des OS monoprocesseur et monocœur)

- **Multithreading**

Le multithreading consiste à appliquer l'idée du multitâche au sein d'une même application. C'est à dire de permettre d'exécuter en parallèle différentes actions (comme des fonctions) au sein d'un même programme

Présentation – Parallélisme : Quelques termes



- **Cœur physique**

C'est un ensemble de circuits capable d'exécuter des programmes de façon autonome. Il possède donc ses propres registres, unités de calculs, compteur ordinal, etc. Il possède souvent son propre cache

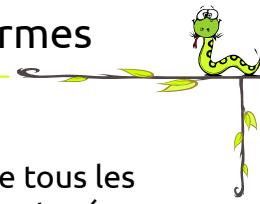
- **HyperThreading**

Technologie développée par Intel permettant de créer 2 processeurs logiques sur un même cœur physique. Les ordinateurs Pentium P4 ou i7 en sont un exemple.

Un seul thread peut s'exécuter à la fois, quand l'un des deux est en attente d'une entrée sortie, l'autre prend le relais.

- Les gamers ne les aiment pas...

Présentation – Parallélisme : Quelques termes



- **SMP**

Symmetric MultiProcessing. On parle de SMP lorsque tous les processeurs sont connectés à une unique mémoire partagée.
Cas des processeurs multicœurs

```
Bash$ uname -a
Linux SMP Tue Sep 19 17:28:18 UTC 2017 x86_64 x86_64 x86_64 GNU/Linux
```

- **NUMA**

Non Uniform Memory Access. On parle d'architecture NUMA lorsque la mémoire entre les processeurs est distribuée dans différents nœuds. Son temps d'accès n'est plus constant et dépend de l'emplacement auquel tente d'accéder un processeur.

- On les programme généralement avec des librairies de type [MPI](#) (Message Passing Interface)

Présentation – Parallélisme : Quelques termes



• Différence CPU/GPU

Un CPU (Central Processing Unit) correspond aux processeurs que l'on trouve dans les ordinateurs portables actuels.

- Ils sont conçus pour exécuter des programmes séquentiels le plus rapidement possible, et, aujourd'hui en parallèle
 - Ils savent gérer les entrées/sorties/interruptions matérielles, partager le cache mémoire et sont de ce fait particulièrement bien adaptés pour faire tourner des systèmes d'exploitation
 - Ils possèdent rarement plus d'une dizaine de cœurs
- Un GPU (Graphics Processing Unit) est un processeur initialement dédié à l'affichage graphique
 - Ils sont conçus pour exécuter la même instruction en parallèle sur de multiples données
 - Ils possèdent des centaines voire milliers de cœurs
 - Imbattables pour le calcul parallèle « *ils ne sont pas taillés* » pour faire tourner un OS

Le CPU est un 4x4 qui s'adapte à tous les besoins, le GPU une formule 1 qui ne roule bien que sur circuit de course...

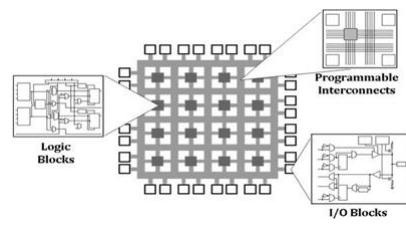
Présentation – Parallélisme : Quelques termes



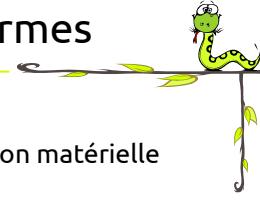
- **FPGA**

Un FPGA (Field Programmable Gate Arrays) est un matériel qui contient de nombreuses cellules logiques librement reconfigurables.

- Il est composé de nombreuses cellules logiques élémentaires et bascules logiques librement connectables.
- Contrairement aux processeurs, les FPGA utilisent du matériel dédié pour traiter la logique et n'ont pas de système d'exploitation.
- Un FPGA tout seul peut remplacer des milliers de composants discrets en incorporant des millions de portes logiques dans un seul et unique circuit intégré.
- Il peut traiter des opérations en parallèles, comme le feraient les cœurs d'un processeur.
- C'est une solution matérielle donc très performante et qui est reconfigurable.



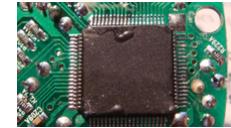
Présentation – Parallélisme : Quelques termes



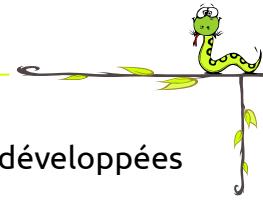
- **ASIC**

Un ASIC (Application Spécifique Integrated Circuit) est une solution matérielle de type circuit intégré spécialisé, non reconfigurable.

- Tout comme le FPGA il est conçu pour concevoir des matériels réalisant des traitements logiques spécifiques.
- Contrairement aux processeurs FPGA le matériel est gravé une fois pour toutes et ne peut pas être reconfiguré pour d'autres traitements.
- Son coût de conception est très grand comparé à l'achat d'un FPGA (souvent plusieurs millions d'euros pour faire le moule).
- Mais le coût de la fabrication des puces est très faible (quelques centimes) lorsque le moule est validé.
- C'est une solution matérielle donc très performante.

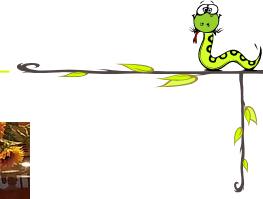


Présentation Parallélisme : Les bases



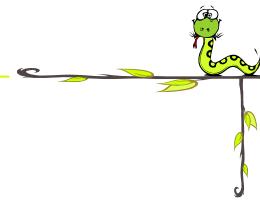
- Les techniques de programmation parallèle ont été développées initialement sur les supercalculateurs
- Elles changent de façon significative notre manière d'écrire des programmes
- Pour bien comprendre les systèmes parallèles
 - « [Votre boulangerie a un système d'exploitation multiprocesseur](#) »
- Et leurs limites
 - « [Et plus vite si affinités](#) »

Présentation Parallélisme : Les bases



Limites du parallélisme

Tout n'est pas parallélisable



« *1 femme fait un bébé en 9 mois, mais 9 femmes ne peuvent pas faire un bébé en 1 mois* » *Frederick Brooks*

Limites du parallélisme



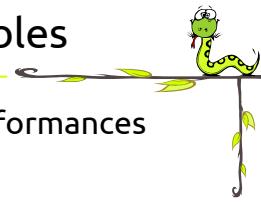
La loi d'Amdahl indique que tout programme possède une partie séquentielle et que tôt ou tard, c'est cette portion de programme qui limitera l'accélération qui peut en être faite. https://fr.wikipedia.org/wiki/Loi_d'Amdahl

Par exemple si votre programme dure 10 secondes sur une machine, et que 90 % de ce temps est passé dans une portion de code à scalabilité linéaire (parallélisable) et les 10 % restant dans une portion de code séquentielle incompressible, alors :

- Si on le fait tourner sur 9 processus, le gain sera de x5
 $1s + 9s / 9 = 2s$
- Au mieux le programme ne pourra jamais avoir une accélération supérieure à x10
 $1s + 9s/\infini = 1s$

Il est important de pouvoir identifier ses portions de code avant de se lancer tête baissée dans la parallélisation....

Identifier les portions de code parallélisables



Sous Unix il existe des commandes pour mesurer les performances d'un programme (profiling) :

- Gprof (C, Pascal or Fortran)
- Valgrind/callgrind pour tout exécutable

Sous Python 2 librairies sont disponibles

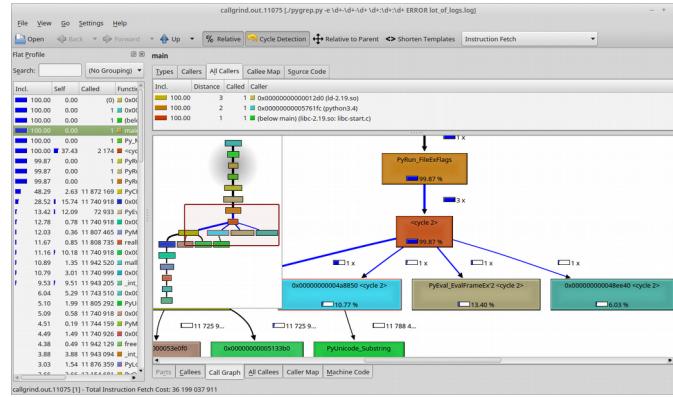
- `timeit`
<https://docs.python.org/3/library/timeit.html>
- `cProfile`
<https://docs.python.org/3/library/profile.html>

Identifier les portions de code parallélisables



La commande « valgrind » couplée à l'outil « callgrind » vous permet de générer un graphe des appels de votre programme et de visualiser :

- Le nombre de fois qu'une fonction est appelée
 - Le temps passé dans chaque fonction



Présentation des différents concepts du parallélisme et du paysage des librairies de calcul distribué en Python

Page 18

Présentation – Limites physiques du parallélisme



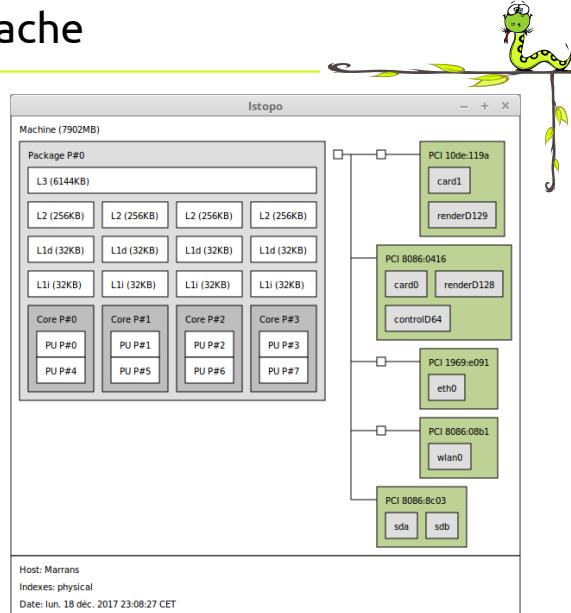
- Les processeurs actuels ont des fréquences supérieures au GigaHertz.
 - Dans les années 1980 la mémoire (< 1Mo en général) était intégrée au processeur et on y accédait en 1 cycle machine/1 instruction
 - Mais la fréquence des processeurs a cru d'environ 50 % par an contre une augmentation de la vitesse de lecture des mémoires de 7 % environ, laquelle est maintenant externe aux processeurs
- Fin 2015 les temps d'accès à une adresse mémoire en lecture ou écriture sont de l'ordre de 100 nanosecondes. Soit 100 cycles d'un processeur ! Pour pallier à cela :
 - Les processeurs modernes utilisent des caches locaux très rapides mais de capacité très réduite
 - <http://ecariou.perso.univ-pau.fr/cours/archi/cours-5-memoire.pdf>
 - [Architecture et micro-architecture des processeurs](#)

Affinité/Invalidation de cache

Pourquoi les gamers n'aiment pas l'hyperthreading et donc mon i7 ?

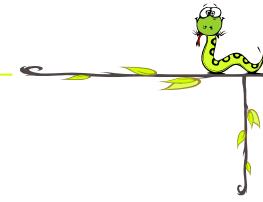
- Le cœur physique possède 2 hyperthreads
- Un seul peut s'exécuter à la fois
- Quand le premier est en attente d'une entrée sortie l'autre prend le relais
- Mais si le second a besoin d'accéder à une zone mémoire non présente dans le cache du premier
- Il va le vider et le recharger avec les nouvelles données (invalidation de cache)
- C'est lent car la mémoire est plus lente que le CPU
- Quand le premier reprendra la main il fera de même
- Les performances peuvent ainsi chuter terriblement !

La librairie [ATLAS](#) de calcul numérique conseille carrément de le désactiver !



*Architecture d'un processeur Intel i7 réalisée via la commande
bash\$ hwloc-ls*

Les différentes formes du parallélisme



Il existe de nombreuses manières de paralléliser vos applications :

- **Dans un même programme**
 - En utilisant des jeux d'instructions dites vectorielles SIMD (Single Instruction Multiple Data) comme SSE/AVX/NEON
 - En utilisant des threads
- **Sur un même ordinateur**
 - En utilisant la programmation concurrente
 - En utilisant plusieurs processeurs/cœurs (multiprocessing/OpenMP/MPI)
- **Sur plusieurs ordinateurs**
 - Calcul distribué
 - Cloud computing
 - Grid computing

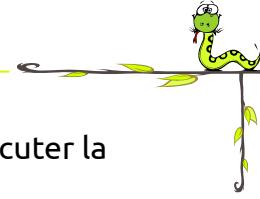


<http://semelin.developpez.com/cours/parallelisme/>

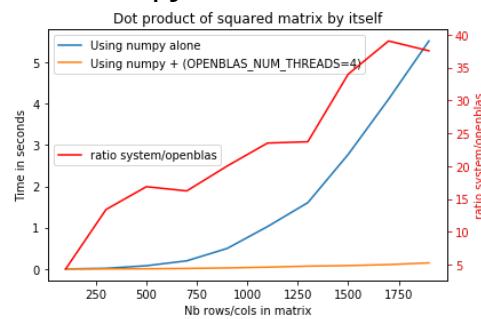
Présentation des différents concepts du parallélisme et du paysage des librairies de calcul distribué en Python

Page 21

Démo : bien installer Numpy



- On pense souvent qu'installer numpy se limite à exécuter la commande « pip install numpy »
- Mais si vous ne disposez pas de librairies mathématiques vectorisées/parallélisées, vous allez passer à côté de l'essentiel : **La performance !**
- Il est donc conseillé d'installer Numpy avec une librairie comme :
 - [Atlas](#), [Blas](#), [Lapack](#)
 - [OpenBlas](#)
 - Intel [MKL](#)
 - AMD [ACML](#)



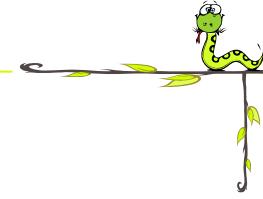


Programmation Parallèle en Python

Les bases algorithmiques

Présentation des différents concepts du parallélisme
et du paysage des librairies de calcul distribué en Python

Parallélisme avec Python



Python offre en standard la possibilité de réaliser :

- De la programmation concorrente avec les librairies « concurrent.futures » et « asyncio »
<https://docs.python.org/3/library/asyncio.html>
<https://docs.python.org/3/library/concurrent.futures.html>
- De faire du multithreading avec la librairie « threading »
<https://docs.python.org/3/library/threading.html>
- De faire du multiprocessing avec la librairie du même nom
<https://docs.python.org/3/library/multiprocessing.html>
- De faire du calcul distribué avec la librairie « multiprocessing » et les objets « Manager »

Multithreading et Multiprocessing



Les librairies pour le multithreading et multiprocessing ont une interface quasiment identique en Python. En renommant la librairie importée il devient ainsi possible de basculer de l'un à l'autre en ne modifiant qu'une ligne de code :

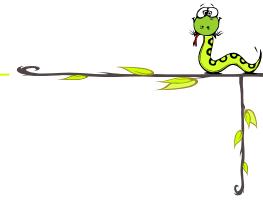
- `from threading import Thread as Task`
- `from multiprocessing import Process as Task`

Multithreading



Présentation des différents concepts du parallélisme
et du paysage des librairies de calcul distribué en Python

Multithreading : Les bases



Utilisation de la librairie « threading » :

- `from threading import Thread as Task`
- **Créer un thread :**
`t = Task(target, args, kwargs, daemon)`
 - target : nom de fonction ou méthode
 - args : tuple contenant les arguments positionnels passés à la fonction
 - kwargs : dictionnaire contenant les arguments nommés passés à la fonction
 - Daemon : Si True, le thread est considéré comme un daemon.
Le programme peut s'arrêter même si le thread continue d'exister
- **Exécuter un thread :**
`t.start()`
- **Attendre la fin de l'exécution d'un thread :**
`t.join()`

Multithreading : Les bases

```
from threading import Thread as Task
import time

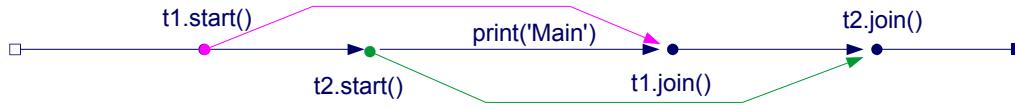
def f1(name):
    for ind in range(10):
        print("%s : %d" % (name, ind))
        time.sleep(1)

t1 = Task(target=f1, args=("Thread 1",))
t2 = Task(target=f1, args=("Thread 2",))
t1.start()
t2.start()

for ind in range(10):
    print("Main", ind)
    time.sleep(1)

# wait tasks to be finished
[ t.join() for t in (t1, t2)]
print('Fin')
```

Thread 1 : 0
 Thread 2 : 0
 Main 0
 Main 1
 Thread 1 : 1
 Thread 2 : 1
 Main 2
 Thread 1 : 2
 Thread 2 : 2
 Main 3
 Thread 1 : 3
 Thread 2 : 3
 Main 4
 Thread 2 : 4
 Thread 1 : 4
 ...



Présentation des différents concepts du parallélisme et du paysage des librairies de calcul distribué en Python

Page 28

Multithreading : Les verrous

```
from threading import Thread as Task

class BankAccount(object):
    def __init__(self, owner, initial_deposit):
        self.amount = initial_deposit
        self.owner = owner

    def __str__(self):
        return "%s: %s€" % (self.owner, self.amount)

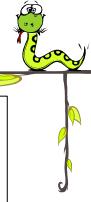
    def virement(self, money):
        self.amount += money

my_account = BankAccount("Me", 1.0)
print("Start:", my_account)

tasks = []
for ind in range(0, 100):
    t = Task(target=my_account.virement, args=(100,))
    t.start()
    tasks.append(t)

print('Waiting end of tasks')
[ t.join() for t in tasks]
print("End:", my_account)
```

Start: Me: 1.0€
Waiting end of tasks
End: Me: 10001.0€



Multithreading : Les verrous



Mais c'est bientôt Noël et les réseaux bancaires sont saturés...

- Modifions la méthode virement comme suit
- Puis relançons le programme

```
def virement(self, money):
    # get current amount
    current_amount = self.amount

    # network slow time
    sleep(0.01)

    # doing computation
    current_amount += money

    # hard drive latency
    sleep(0.01)

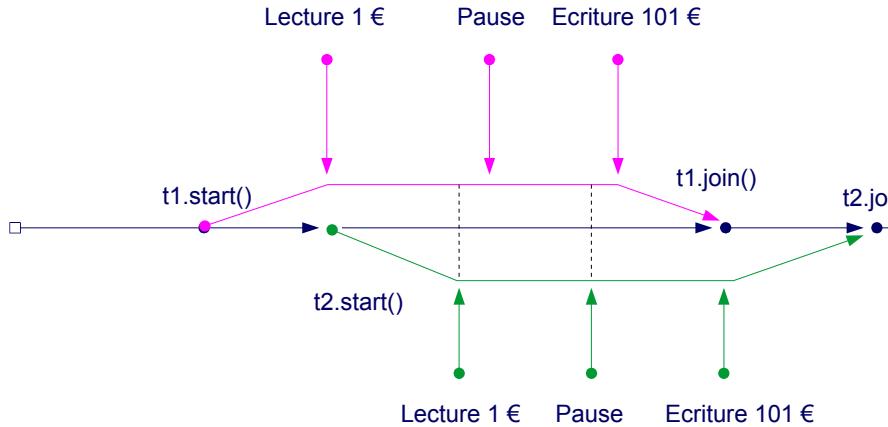
    self.amount = current_amount
```

```
Start: Me: 1.0€
Waiting end of tasks
End: Me: 201.0€
```



*Mais qui a vampirisé toute ma
Précieuse monnaie ?*

Multithreading : Les verrous



Multithreading : Les verrous



Pour contourner ce problème on peut utiliser un verrou.

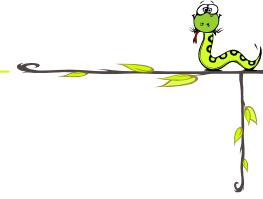
Le verrou n'est accessible par une seule tâche à la fois. Une fois pris le code qui suit son acquisition n'est exécutable que par la tâche jusqu'à ce qu'elle le libère :

- **Création d'un verrou :**
verrou = Lock()
- **Acquisition d'un verrou :**
verrou.acquire()
- **Libération d'un verrou :**
verrou.release()
- **Un verrou accepte les « contexts managers » (mot clef « with »)**

```
verrou = Lock()
with verrou:
    # vos traitements nécessitant une protection
    # contre des accès concurrents
```

```
verrou = Lock()
verrou.acquire()
try:
    # vos traitements nécessitant une protection
    # contre des accès concurrents
finally:
    verrou.release()
```

Multithreading : Les verrous



Avec l'ajout d'un verrou :

- La fonction virement n'est plus parallélisable
- Mais « my money is back ! »

```
from threading import Thread as Task, Lock

class BankAccount(object):
    def __init__(self, owner, initial_deposit):
        ...
        self.lock = Lock()

    def __str__(self):
        return "%s: %s€" % (self.owner, self.amount)

    def virement(self, money):
        with self.lock:
            current_amount = self.amount
            sleep(0.01)
            current_amount += money
            sleep(0.01)
            self.amount = current_amount
```

Start: Me: 1.0€
Waiting end of tasks
End: Me: 10001.0€

Multithreading : Autres fonctionnalités



Le module `threading` propose de nombreuses autres fonctionnalités :

- Verrous ré-entrants
- Sémaphores
- Files, Piles
- Timer
- Event
- Barrier
- Condition
- ...

Multiprocessing



L'union fait la force

Présentation des différents concepts du parallélisme
et du paysage des librairies de calcul distribué en Python

Page 35

Multiprocessing : Comme threading, ou presque...



Le module multiprocessing a une interface identique à celle du module threading.

De ce fait, pour passer d'un module à l'autre vous n'avez qu'une ligne de code à changer...

- from threading import Thread as Task

- **Devient**

```
from multiprocessing import Process as Task
```

Enfin, presque...

```
Start: Me: 1.0e
Waiting end of tasks
End: Me: 1.0e
```



*Les gremlins sont de retour !
Et ils sont encore plus méchants !*

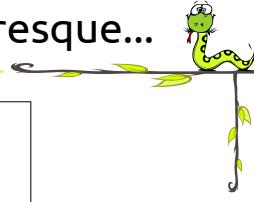
Multiprocessing : Comme threading, ou presque...



Que s'est-il passé ?

- Quand vous exécutez un processus il crée une copie du programme principal, via un « fork » au niveau de l'OS
- Ceci duplique aussi les données de votre programme
- Et chaque process se retrouve avec sa propre copie du compte bancaire
- Cela complique un peu votre tâche
 - Il convient de mettre le compte bancaire en mémoire partagée
 - Vous êtes alors limités aux types du langage C
 - Par contre les variables en mémoire partagée possèdent leur propre verrou
 - Et une propriété « value » permettant la conversion avec les types Python

Multiprocessing : Comme threading, ou presque...



```
from multiprocessing import Process as Task
from multiprocessing.sharedctypes import Value
from ctypes import c_double

class BankAccount(object):
    def __init__(self, owner, initial_deposit):
        self.amount = Value(c_double, initial_deposit)
        self.owner = owner

    def __str__(self):
        return "%s: %s€" % (self.owner, self.amount.value)

    def virement(self, money):
        with self.amount:
            # get current amount
            current_amount = self.amount.value

            # network slow time
            sleep(0.01)

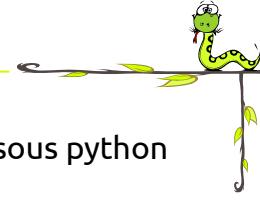
            # doing computation
            current_amount += money

            # hard drive latency
            sleep(0.01)

        self.amount.value = current_amount
```

Start: Me: 1.0€
Waiting end of tasks
End: Me: 10001.0€

Multithreading : Les limites



Afin de bien comprendre les limites du multithreading sous python réalisez les opérations suivantes :

- Exécutez la fonction `countdown()` avec une valeur très grande (100 millions) par exemple
 - 2 fois en série
 - Avec 2 threads en parallèle
 - Avec 2 process en parallèle
- En théorie les versions parallèles devraient demander 2 fois moins de temps que l'exécution en série si vous possédez une machine multicœurs/multiprocesseurs
- Mais ce n'est pas le cas avec le module `threading...`

```
def countdown(n):
    while n > 0:
        n -= 1
```

Multithreading : Les limites

```
def countdown(n):
    while n > 0:
        n -= 1

def serial(n, repeat=2):
    for ind in range(repeat):
        countdown(n)

def threads(n, repeat=2):
    tasks = []
    for ind in range(repeat):
        t = Thread(target=countdown, args=(n,))
        t.start()
        tasks.append(t)

    [t.join() for t in tasks]

def processes(n, repeat=2):
    tasks = []
    for ind in range(repeat):
        t = Process(target=countdown, args=(n,))
        t.start()
        tasks.append(t)

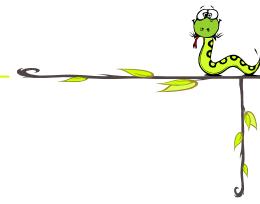
    [t.join() for t in tasks]

N = 10**8
repeat = 2
```

Exécution en série
serial: 7.866269341997395
threads: 8.027223899996898
processes: 4.158569907005585



Multithreading : Comprendre les limites



Le multithreading avec Python ça semble bien.

Mais « **ça craint du boudin** » :

- En fait Python, n'est pas capable d'utiliser l'architecture multicœurs des processeurs modernes dans son implémentation multithread.
- L'interpréteur Python ne sait exécuter qu'un seul thread à la fois
- C'est uniquement quand un thread est en attente sur une opération I/O que l'interpréteur changera de thread.
- Pour bien comprendre ces limites vous pouvez consulter cet excellent document de David Beazley qui est devenu la référence sur le sujet :

<http://www.dabeaz.com/python/UnderstandingGIL.pdf>

Multithreading : Savoir à quoi s'en tenir



- Les threads sont intéressants avec Python tant que vos traitements réalisent beaucoup d'opérations d'entrées/sorties.
- Si vos threads font énormément de calculs, ils ne seront pas efficaces. Ils peuvent même être plus lents.
- C'est un problème interne à l'implémentation du langage
 - Jusque là le problème a été maintenu en connaissance de causes pour ne pas pénaliser les ordinateurs monoprocesseur/monocoeur.
 - Une solution technique compatible avec ces derniers et permettant d'utiliser le potentiel des nouvelles architectures a apparemment été trouvée et sera implémentée dans une prochaine version

http://python-notes.curiousoftware.org/en/latest/python3/multicore_python.html

<https://wiki.python.org/moin/GlobalInterpreterLock>

Multiprocessing : Process Pools



Les « pools » de processus permettent d'assigner un ensemble de tâches à un groupe de processus :

- L'objet « Pool » gère un nombre donné de processus auxquels il distribuera les tâches qu'il recevra.
Si vous ne fournissez pas ce nombre il utilisera la valeur de `os.cpu_count()`
- Il peut tuer un worker après que celui-là ait traité un certain nombre de tâches pour en redémarrer un tout frais
- Il peut demander l'exécution les tâches de manière *synchrone* (et rester bloqué en attente du résultat) ou de manière *asynchrone* (ce qui est préférable pour la programmation parallèle)
- Elle peut s'utiliser avec un « context manager » depuis Python 3.3

Multiprocessing : Process Pools

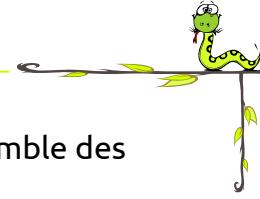


Le constructeur de la classe « Pool » accepte les paramètres suivants :

- « processes »
Le nombres de processus utilisés pour traiter les tâches
- « initializer », « initargs »
Une fonction utilisée pour initialiser les processus créés et les paramètres qui lui sont passés
- « maxtasksperchild »
Le nombre maximum de tâches qu'un processus pourra exécuter avant d'être tué et remplacé

```
# 3 processus pour traiter les tâches,  
# une fonction init_process pour les initialiser  
# et qui ne prend pas de paramètres  
# 2 tâches exécutées par processus  
pool = Pool(3, init_process, (), 2)
```

Multiprocessing : Process Pools



Elle possèdent différentes méthodes pour gérer l'ensemble des processus fils :

- « close »
Interdit l'ajout de nouvelles tâches.
Une fois les tâches en cours terminées, les workers s'arrêteront
- « join »
Attend la fin des processus workers. « close » doit avoir été appelée au préalable
- « terminate »
Arrête tous les workers en cours, sans qu'ils terminent leurs tâches

Multiprocessing : Process Pools - map



La méthode « `map` » se comporte comme la fonction « `map` » de python en appelant une fonction sur un ensemble de valeurs. Chaque appel de fonction est réparti entre les différents workers.

- C'est une fonction synchrone (bloquante)
- Elle accepte un paramètre « `chunksize` » qui permet d'envoyer plusieurs valeurs à calculer aux différents processus à chaque appel
- Le résultat retourné est ordonné, comme pour la fonction « `map` »

Multiprocessing : Process Pools - map



```
from multiprocessing import Pool
from time import sleep
import os
from functools import reduce

def carre(x):
    print("Calcul du carré de %s par processus %s" % (x, os.getpid()))
    sleep(1)
    return (x, x * x)

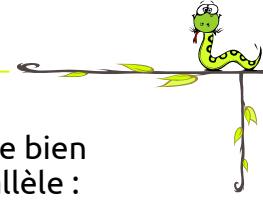
def init_process():
    print("Initializing process %s" %(os.getpid(),))

if __name__ == "__main__":
    pool = Pool(processes=3, initializer=init_process, initargs=(),
                maxtasksperchild=2)

    with pool:
        print("Running map")
        data = pool.map(carre, range(20))
        print("End map")
        print(data)
```

```
Initializing process 6140
Initializing process 6141
Initializing process 6142
Running map
Calcul du carré de 0 par processus 6140
Calcul du carré de 1 par processus 6140
Calcul du carré de 2 par processus 6141
Calcul du carré de 4 par processus 6142
Calcul du carré de 5 par processus 6142
Calcul du carré de 3 par processus 6141
Calcul du carré de 1 par processus 6140
Calcul du carré de 6 par processus 6141
...
Initializing process 6146
Calcul du carré de 12 par processus 6146
Initializing process 6147
Calcul du carré de 14 par processus 6147
Initializing process 6148
Calcul du carré de 16 par processus 6148
Calcul du carré de 13 par processus 6146
Calcul du carré de 15 par processus 6147
Calcul du carré de 17 par processus 6148
Calcul du carré de 18 par processus 6146
Calcul du carré de 19 par processus 6146
End map
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25),
(6, 36), [...] (18, 324), (19, 361)]
```

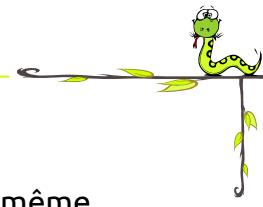
Multiprocessing : Compléments



Le pool de processus est la notion qu'il est primordial de bien comprendre pour se lancer dans l'univers du calcul parallèle :

- La plupart des librairies de calcul distribué sont basées sur ce modèle
- La librairie propose de nombreuses autres fonctions :
 - map_async
 - apply, apply_async
 - imap, imap_async
 - starmap, starmap_async

Multiprocessing : Proxy et managers



Les managers permettent plusieurs choses :

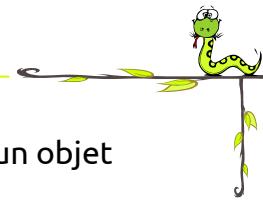
- Partager des objets entre plusieurs processus sur la même machine
- Partager des objets entre plusieurs processus sur fonctionnant en réseau sur plusieurs machines.

Ils sont très puissants.

Ces objets sont manipulés via des proxys.

Tous les types de structures Queue, Lock, Value, Barrier, et autres peuvent être « proxyfiés » via un manager

Multiprocessing : Proxy et managers



La classe « Manager » du module multiprocessing crée un objet « multiprocessing.managers.SyncManager »

- Il implémente les verrous, variables, et autres objets de synchronisation
- Il permet de créer des listes et dictionnaires
- <https://docs.python.org/3.5/library/multiprocessing.html#multiprocessing.managers.SyncManager>

```
from multiprocessing import Manager  
manager = Manager()  
q = manager.Queue()
```

Multiprocessing : Managers



Une des grandes forces des managers est de permettre à des processus répartis sur différentes machines d'échanger des données.
Pour cela différentes méthodes existent :

- La classe `BaseManager` accepte 2 paramètres « `address` » et « `authkey` »:
 - `Address` contient l'adresse du Manager sur lequel se connecter ou sur laquelle il écoutera. `None` correspond à une écoute sur toutes les IP
 - `Authkey` est la clef d'authentification qui devra être utilisée par les clients
- « `get_server` »
Cette méthode retourne le serveur qui contrôle le manager pour le lancer (process serveur) ou s'y connecter (process client)

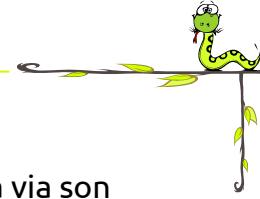
Multiprocessing : Managers



Une des grandes forces des managers est de permettre à des processus répartis sur différentes machines d'échanger des données.
Pour cela différentes méthodes existent :

- «serve_forever»
Appliquée sur le serveur retourné par «get_server» exécute ce dernier
- «connect »
Permet aux clients de se connecter à un manager
- « shutdown »
Arrête le manager et son serveur

Multiprocessing : Managers



- «register»

Permet d'indiquer quels objets le manager exposera via son interface.

Les objets peuvent être des variables globales ou des attributs de la classe Manager

Elle accepte plusieurs paramètres dont :

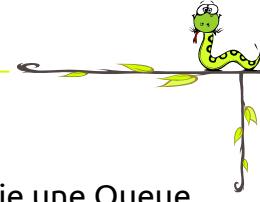
- « typeid »

Le nom par lequel les clients accèderont à la donnée

- « callable »

Un objet callable (généralement une fonction) retournée lorsque les clients utiliseront manager. « typeid »

Multiprocessing : Managers



La classe QueueManager.

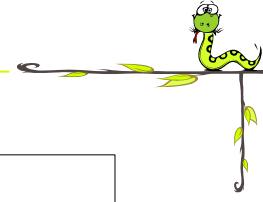
Elle hérite tout simplement de BaseManager et instancie une Queue sous la forme d'une variable de classe.

Mais l'objet Queue aurait pu être une variable globale du module.

L'astuce réside ici dans le fait que pour retourner la « Queue » nous créons une fonction lambda, car l'objet Queue ne propose pas de fonction pour se retourner lui-même.

```
from multiprocessing import managers, Queue
class QueueManager(managers.BaseManager):
    queue = Queue()
QueueManager.register('get_queue', callable=lambda: QueueManager.queue)
```

Multiprocessing : Managers - TP



Le script du server.

```
from QueueManager import QueueManager

manager = QueueManager(address='127.0.0.1', 8000, authkey=b'secret')
server = manager.get_server()
server.serve_forever()

print("done")
```

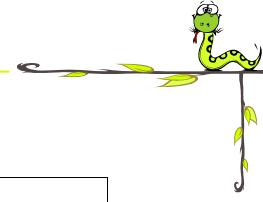
Le script du provider.

```
from QueueManager import QueueManager

m = QueueManager(address='127.0.0.1', 8000, authkey=b'secret')
m.connect()
q = m.get_queue()
for i in range(10):
    q.put("Task %s" % i)

print("Provider done")
```

Multiprocessing : Managers - TP



Le script du worker.

```
from QueueManager import QueueManager
from queue import Empty

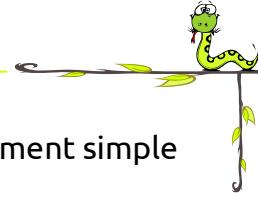
m = QueueManager(address='127.0.0.1', 8000, authkey=b'secret')
m.connect()

q = m.get_queue()

try:
    while True:
        data = q.get(timeout=4)
        print("processing %s" % data)
except Empty:
    pass

print("Client done")
```

Conclusion



- Le parallélisme avec Python est très riche et relativement simple
- Le multithreading est clairement à éviter
- Le multiprocessing, les managers et pool de workers sont les éléments qui vous permettront de véritablement paralléliser votre application à un niveau industriel
- Les grandes difficultés du parallélisme sont :
 - Une gestion asynchrone des traitements qui impose de repenser la façon de concevoir ses programmes
 - Les problèmes de partage de données entre processus
 - Les problèmes de synchronisation des processus
 - Pour cela différentes recommandations sont fournies dans la documentation de Python :
<https://docs.python.org/3.5/library/multiprocessing.html#programming-guidelines>

Conclusion



- Mais la grande force de Python est de disposer de projets reconnus proposant des solutions clefs en mains très puissantes permettant de mettre en œuvre une infrastructure de « programmation parallèle » de très haut niveau :
<https://wiki.python.org/moin/ParallelProcessing>



Paysage du calcul parallèle en Python

- **Compilation**

Générer un code directement exécutable par la machine hôte.

- **Programmation concurrence/asynchrone**

Exécuter des fonctions en alternance lorsqu'elles font des entrées/sorties

- **Multithreading**

Exécuter des fonctions en parallèle dans un même programme

- **Grid Computing**

Du calcul distribué utilisant différents réseaux, architectures généralement hétérogènes et délocalisés : exemples [seti@home](#) ou le LHC

- **Cloud computing**

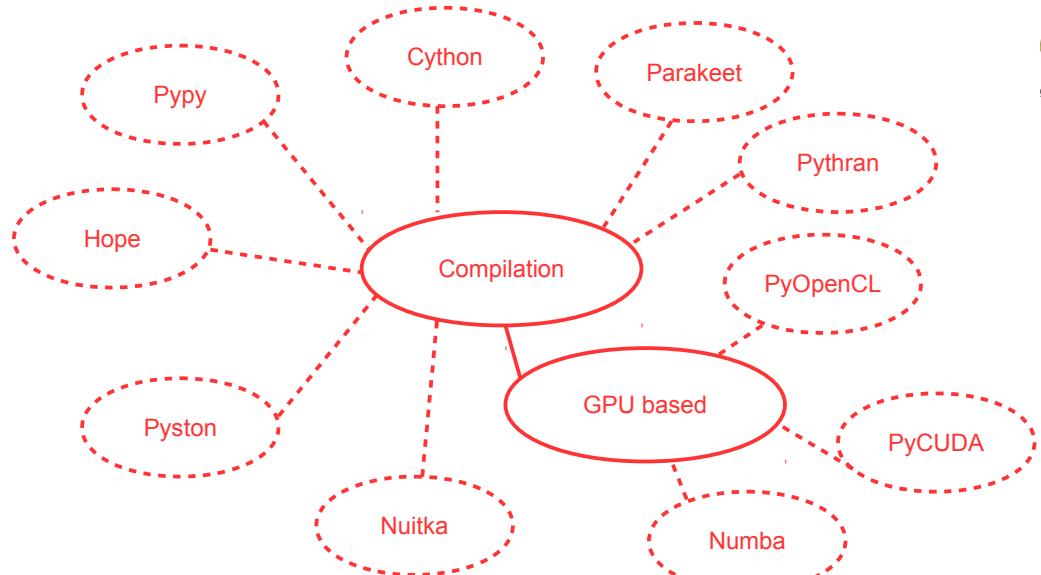
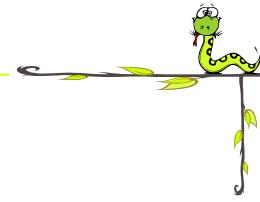
Du calcul distribué dans le web

- **Calcul distribué**

Répartir des calculs sur plusieurs processeurs d'une même machine ou sur plusieurs machines

Présentation des différents concepts du parallélisme et du paysage des bibliothèques de calcul distribué en Python

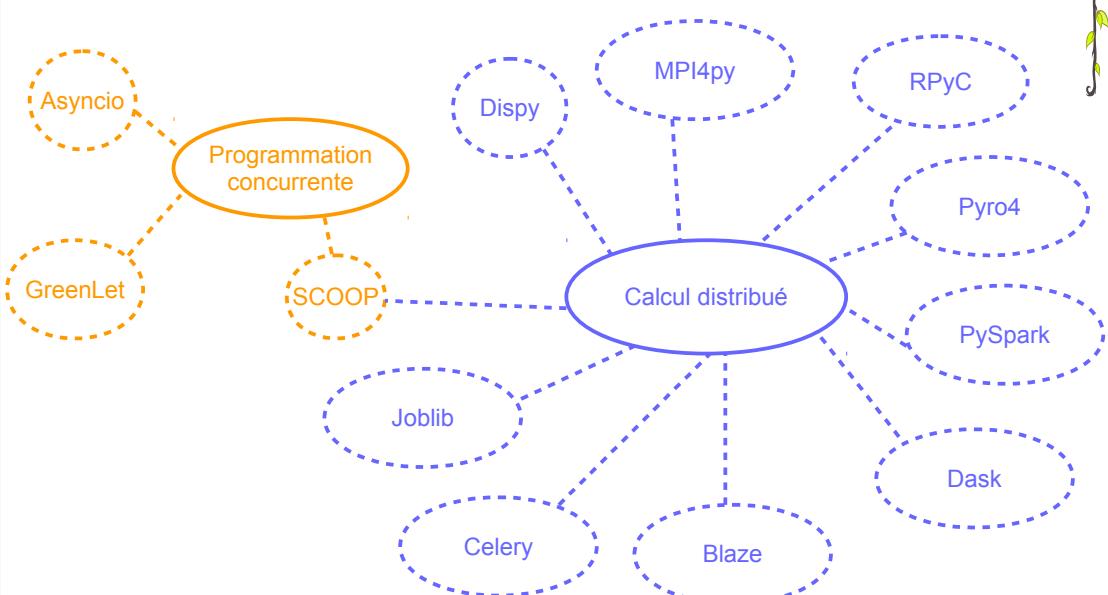
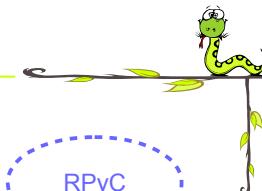
Paysage des librairies de calcul parallèle



Présentation des différents concepts du parallélisme
et du paysage des librairies de calcul distribué en Python

Page 60

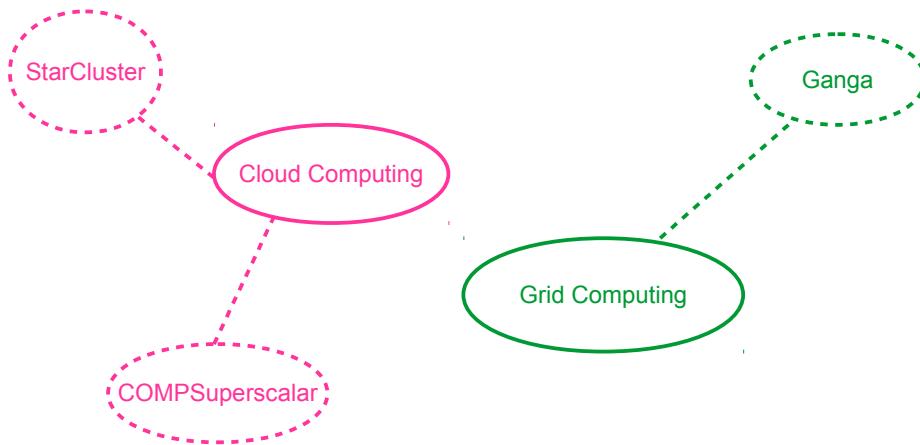
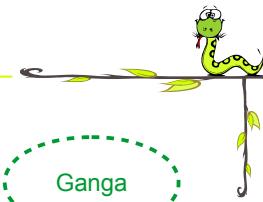
Paysage des librairies de calcul parallèle



Présentation des différents concepts du parallélisme et du paysage des librairies de calcul distribué en Python

Page 61

Paysage des librairies de calcul parallèle

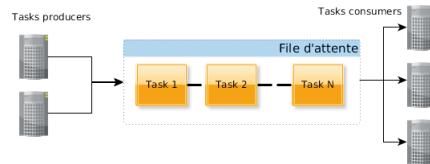


Celery



Celery

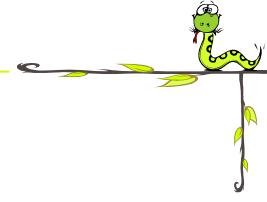
- Celery est un projet de calcul distribué en Python
<http://www.celeryproject.org/>
 - Contrairement à la plupart des autres projets il se contente d'implémenter les workers
 - Pour la distribution des tâches il s'appuie sur des serveurs utilisant le protocole AMQP
 - Par défaut il préconise d'utiliser le serveur RabbitMQ
- RabbitMQ
<http://www.rabbitmq.com/>
 - RabbitMQ est un serveur développé en Erlang et implémentant le protocole AMQP
 - Il propose des API pour de nombreux langages dont Python



Présentation des différents concepts du parallélisme
et du paysage des librairies de calcul distribué en Python

Page 63

Celery - Installation



- Installation de Rabbit MQ

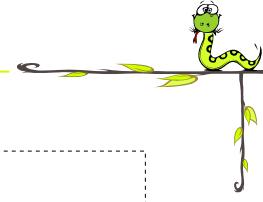
```
$ sudo apt-get install rabbitmq-server
```

<http://www.rabbitmq.com/install-debian.html>

- Installation de Celery
Dans votre environnement virtuel Python

```
$ pip install celery
```

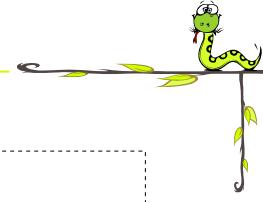
Celery - Vérification de l'installation



- Rabbit MQ

```
$ which rabbitmqctl
/usr/sbin/rabbitmqctl
$ sudo rabbitmqctl status
Status of node rabbit@Marrans ...
[{"pid":7361},
 {"running_applications,[{"rabbit","RabbitMQ","3.2.4"}, {"mnesia","MNESIA CXC 138 12","4.11"}, {"os_mon,"CPO CXC 138 46","2.2.14"}, {"xmerl,"XML parser","1.3.5"}, {"sasl,"SASL CXC 138 11","2.3.4"}, {"stdlib,"ERTS CXC 138 10","1.19.4"}, {"kernel,"ERTS CXC 138 10","2.16.4"}]}, {"os,{unix,linux}}, {"erlang_version,"Erlang R16B03 (erts-5.10.4) [source] [64-bit] [smp:8:8] [async-threads:30] [kernel-poll:true]\n"}, {"memory,[{"total",36438616}, {"connection_procs,2704}, {"queue_procs,5408}, {"plugins,0}, {"other_proc,13607208}, {"mnesia,60496}, ... ]}]
```

Celery - Vérification de l'installation



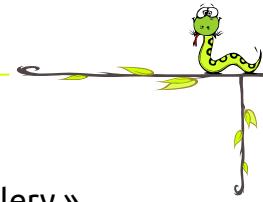
- Celery

```
$ which celery
/path/to/anaconda3/venv/bin/celery
$ celery --help
Usage: celery <command> [options]

Show help screen and exit.

Options:
-A APP, --app=APP      app instance to use (e.g. module.attr_name)
-b BROKER, --broker=BROKER
                      url to broker. default is 'amqp://guest@localhost//'
--loader=LOADER        name of custom loader class to use.
--config=CONFIG         Name of the configuration module
--workdir=WORKING_DIRECTORY
                      Optional directory to change to after detaching.
...
----- Commands -----
+ Main:
|   celery worker
|   ...
```

Première application



Pour créer une application Celery il convient :

- 1) De créer un script Python qui instancie la classe « Celery »
- 2) Créer une tâche liée à cette instance
- 3) Démarrer votre instance de Celery, appelée application, avec la commande « celery »



Première application – étape 1



Créer l'application Celery.

Beaucoup de paramètres peuvent être spécifiés lorsque l'on démarre une application Celery, mais tous ont une valeur par défaut, lesquelles suffisent bien souvent.

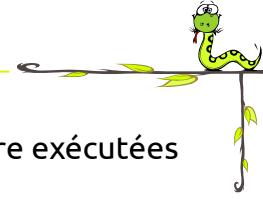
application.py

```
from celery import Celery  
the_app = Celery('MonApplication')
```

L'application a pour charge :

- De gérer le pool de processus (workers) qui traiteront les tâches
- Ajouter les tâches dans les files d'attentes du broker
- Distribuer ces tâches auprès des workers
- Envoyer les résultats vers le backend

Première application – étape 2



Il convient ensuite de définir les tâches qui pourront être exécutées par l'application

Définir une tâche revient à créer une fonction et lui affecter un décorateur du nom de l'instance celery suivi de « @<instance>.task »

application.py

```
from celery import Celery  
  
the_app = Celery('MonApplication')  
  
@the_app.task  
def add(x, y):  
    print("Doing task %s + %s" % (x, y))  
    return x + y
```

Première application – Exécution de tâches



Maintenant que l'application fonctionne, il convient de lui soumettre des tâches.

Pour cela il suffit d'appeler les fonctions déclarées comme tâches

producer.py

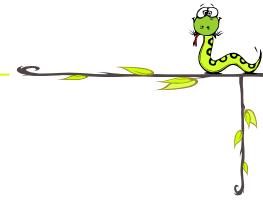
```
from celery1.application import add  
  
for i in range(10):  
    add.delay(i, i)
```

\$ python producer.py

```
[2016-01-22 14:57:49,268: INFO/MainProcess] Received task: celery1.application.add[eaea7701-d280-4acf-be9c-c0c65341ac48]  
[2016-01-22 14:57:49,268: INFO/MainProcess] Received task: celery1.application.add[f02d9cdb-46f0-4e05-9082-99e8837d41da]  
...  
[2016-01-22 14:57:49,272: WARNING/Worker-2] Doing task 0 mul by 0  
[2016-01-22 14:57:49,272: WARNING/Worker-5] Doing task 1 mul by 1  
[2016-01-22 14:57:49,272: WARNING/Worker-8] Doing task 3 mul by 3  
[2016-01-22 14:57:49,272: WARNING/Worker-7] Doing task 2 mul by 2  
[2016-01-22 14:57:49,272: WARNING/Worker-6] Doing task 6 mul by 6  
...
```

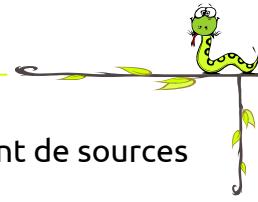
Questions

Merci de votre attention.



Des questions ?

Source des images



Les images utilisées dans cette présentation proviennent de sources libres :

- Wikipédia/Wikimedia
<http://www.wikipedia.fr>
- Pixabay
<https://pixabay.com>