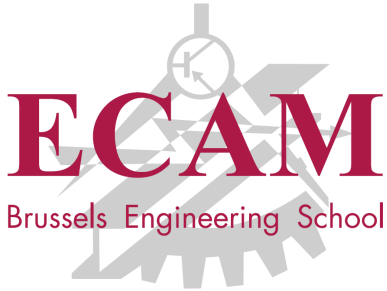


PI2T Développement informatique

Séance 5

Programmation concurrente

Sébastien Combéfis, Quentin Lurkin

8 mars 2016

Objectifs

- Calcul parallèle et distribué
  - Système d'exploitation
  - Processus et threads
  - Multiprocesseur
- Librairie Python de calcul distribué et parallèle
  - Multiprocessing
  - Threading
  - Calcul distribué avec `dispy`

Calcul parallèle

IBM 704

- Exécution séquentielle des programmes l'un après l'autre

Système d'exploitation

- Couche logicielle intermédiaire entre l'utilisateur et le hardware

  Le seul programme qui tourne toujours sur une machine allumée

- Abstraction des installations matérielles sous-jacentes

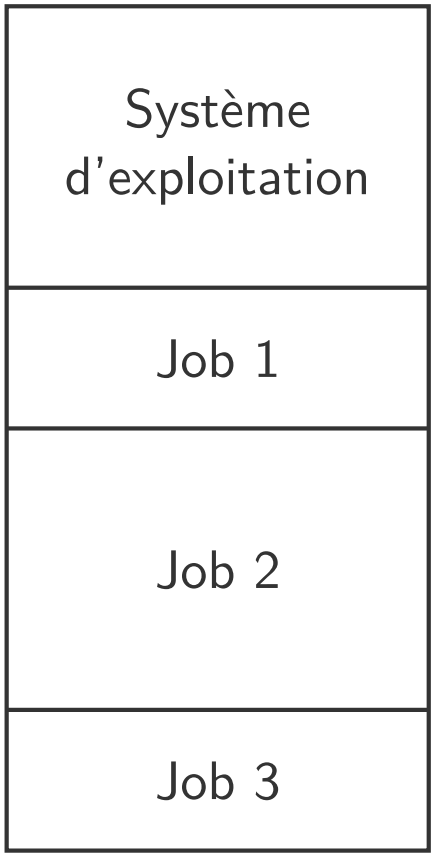
  Permet aux programmes de communiquer avec le hardware

- Gestion de plusieurs aspects liés à l'environnement

  Processus, fichier, mémoire, socket, dessin à l'écran...

Multiprogrammation

Système simple par lot



Traitement par lot multiprogrammé

Processus (1)

- Un processus est un programme en cours d'exécution

  Plusieurs processus peuvent exister pour un même programme

- Plusieurs caractéristiques liés aux processus
  - Processus gérés par le système d'exploitation
  - Trois flux connectés : entrée/sortie standard
  - Code de retour après exécution (0 terminaison normale)

- Informations sur le processus avec le module `sys`

Processus (2)

- Flux accessibles avec `sys.stdout`, `sys.stderr` et `sys.stdin`

  Mêmes méthodes que pour manipuler des fichiers

- Forcer l'écriture des sorties avec `flush`

  Le système d'exploitation stocke temporairement dans des buffers

```
1  import sys
2
3  print('Année de naissance ? ', end='')
4  sys.stdout.flush()
5  data = sys.stdin.read()
6  print('Vous avez', 2016 - int(data), 'ans.')
7  sys.exit(0)
```

Note : il faut utiliser CTRL+D pour envoyer l'entrée standard

Calcul parallèle et distribué

- Exécution concurrente de plusieurs activités

  Rendue possible avec la multiprogrammation

- Exécution parallèle de plusieurs activités

  Rendue possible avec les multi-processeurs

- Plusieurs sources de parallélisme (`os.cpu_count()`)
  - Processeurs multicœurs, multithreadé, co-processeurs
  - Plusieurs processeurs
  - Plusieurs machines reliées par le réseau (cluster)

Concurrence $\neq$ **Parallélisme**

Taxonomie de Flynn

- **SISD** (single-instruction, single-data)

  Ordinateur séquentiel sans aucun parallélisme (von Neumann)

- **SIMD** (single-instruction, multiple-data)

  Parallélisme de mémoire, même opération sur plusieurs données (processeur vectoriel)

- **MIMD** (multiple-instruction, multiple-data)

  Plusieurs unités de calcul traitent des données différentes

Application multi-processus

- Exécution de plusieurs processus indépendants

  Chaque processus possède ses propres ressources

- Communication et partage de données pas évident
  - Fournir les données au lancement du nouveau processus
  - Communication par le réseau
  - Utilisation de services spécifiques au système d'exploitation

Module subprocess

- Exécution de processus et récupération d'informations

  Connexion aux entrées/sorties et récupération code de retour

- Exécution d'un processus avec la fonction `run`

  Prends la liste des arguments de l'appel en paramètre

```
1  import subprocess
2
3  p = subprocess.run(['pwd'])
4  print('Valeur de retour :', p.returncode)
```

```
/Users/combefis/Desktop
Valeur de retour : 0
```

Redirection des flux

- Possibilité de rediriger les entrées et sorties (std et err)
- Plusieurs redirections possibles
  - Dans le néant avec `subprocess.DEVNULL`
  - Vers un fichier avec un descripteur
  - Vers un nouveau pipe créé avec `subprocess.PIPE`

```
1 import subprocess
2
3 with open('result.txt', 'w') as file:
4     p = subprocess.run(['pwd'], stdout=file, stderr=subprocess.
      DEVNULL)
5     print('Valeur de retour :', p.returncode)
```

```
Valeur de retour : 0
```

Objet Popen

- Processus représenté par un objet `Popen`

  Beaucoup plus flexible que de passer par la fonction `run`

- Constructeur de la classe similaire à la fonction `run`

  Liste des arguments, puis autres options

```
1  import subprocess
2
3  proc = subprocess.Popen(['pwd'])
4  proc.wait()
5  print('Fini avec code de retour :', proc.returncode)
```

Communication avec un processus

- Méthode `communicate` pour envoyer données sur `stdin`
  - Communication en `str` grâce à `universal_newlines=True`
  - Redirection des flux vers `subprocess.PIPE`

```
1 import subprocess
2
3 proc = subprocess.Popen(['python3'], stdin=subprocess.PIPE, stdout=
  subprocess.PIPE, universal_newlines=True)
4
5 out, err = proc.communicate('print(1 + 2)\nimport os\nprint(os.
  getcwd())\nexit()')
6 print(out)
```

```
3
/Users/combefis/Desktop
```

Module `multiprocessing`

- Parallélisme par exploitation des multi-processeurs

  Permet de lancer plusieurs processus localement ou à distance

- Parallélisme de données à l'aide d'objets `Pool`

  Exécution d'une même fonction sur plusieurs données

```
1  import multiprocessing as mp
2
3  def compute(data):
4      return data ** 2
5
6  with mp.Pool(3) as pool:
7      print(pool.map(compute, [1, 7, 8, -2]))
```

```
[1, 49, 64, 4]
```

Lancer un processus

- Processus représenté par un objet `Process`

  Code du processus défini par une fonction passée au constructeur

- Méthode `start` pour lancer le processus

```
1  import multiprocessing as mp
2
3  def sayhello(name):
4      print('Hello', name)
5
6  proc = mp.Process(target=sayhello, args=('Dan',))
7  proc.start()
8
9  proc.join() # Attendre la fin du processus
10 print('Terminé avec code', proc.exitcode)
```

```
Hello Dan
Terminé avec code 0
```

Communication interprocessus

- Possibilité d'échanger des objets entre processus

  Permet une communication bidirectionnelle entre deux processus

- Deux constructions différentes proposées
  - **File** de communication (`Queue`)
  - **Tube** de communication (`Pipe`)

- Attendre la fin d'un processus avec la méthode `join`

  Méthode `join` bloquante

Communication par Queue

- Création de la `Queue` et passation au processus fils

Méthodes `put` et `get` pour écrire et lire dans la file

```
1  import multiprocessing as mp
2
3  def compute(q):
4      q.put('Hey!')
5
6  q = mp.Queue()
7  proc = mp.Process(target=compute, args=(q,))
8  proc.start()
9  print(q.get())
10
11 proc.join()
12 print('Terminé avec code', proc.exitcode)
```

```
Hey!
Terminé avec code 0
```

Communication par `Pipe`

- Création du `Pipe` et passation d'un bout au processus fils

  Méthodes `send` et `recv` pour envoyer et recevoir des données

```
1  import multiprocessing as mp
2
3  def compute(child):
4      child.send('Hey!')
5      child.close()
6
7  parent, child = mp.Pipe()
8  proc = mp.Process(target=compute, args=(child,))
9  proc.start()
10 print(parent.recv())
11
12 proc.join()
13 print('Terminé avec code', proc.exitcode)
```

```
Hey!
Terminé avec code 0
```

Pool de workers

- Création d'un `Pool` de processus

  Nombre de processus dans le pool à spécifier au constructeur

- Plusieurs méthodes pour lancer une exécution
  - `apply` fait exécuter une fonction par un worker
  - `map` fait exécuter une fonction avec plusieurs données
  - `starmap` lorsque plusieurs paramètres pour la fonction

```
1  import multiprocessing as mp
2
3  def compute(a, b):
4      return a + b
5
6  with mp.Pool(3) as pool:
7      print(pool.starmap(compute, [(1, 2), (7, -4), (9, 3)]))
```

Mode d'exécution synchrone

- Mode d'exécution synchrone par défaut

  Le processus parent attend la fin de l'exécution de son fils

```
1  import multiprocessing as mp
2  import random
3  import time
4
5  def temperature(city):
6      print('Lancé', city)
7      time.sleep(random.randint(0, 3))
8      return random.randint(0, 25)
9
10 with mp.Pool(3) as pool:
11     for city in ['Belgium', 'Croatia', 'Poland']:
12         print('>', city, pool.apply(temperature, (city,)))
```

```
Lancé Belgium
> Belgium 20
Lancé Croatia
> Croatia 21
Lancé Poland
> Poland 3
```

Mode d'exécution asynchrone (1)

- Version asynchrone des méthode de `Pool`

  Appel pas bloquant renvoie un objet de type `AsyncResult`

- Lance le processus et se termine directement

  Attente de la réponse avec méthode `get` de `AsyncResult`

Mode d'exécution asynchrone (2)

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

def temperature(city):
    print('Lancé', city)
    time.sleep(random.randint(0, 3))
    return random.randint(0, 25)

with mp.Pool(3) as pool:
    results = []
    for city in ['Belgium', 'Croatia', 'Poland']:
        results.append((city, pool.apply_async(temperature, (city,))))
    i = 0
    while i < len(results):
        city, result = results[i]
        print('>', city, result.get())
        i += 1
```

```
Lancé Belgium
Lancé Croatia
Lancé Poland
> Belgium 21
> Croatia 23
> Poland 1
```

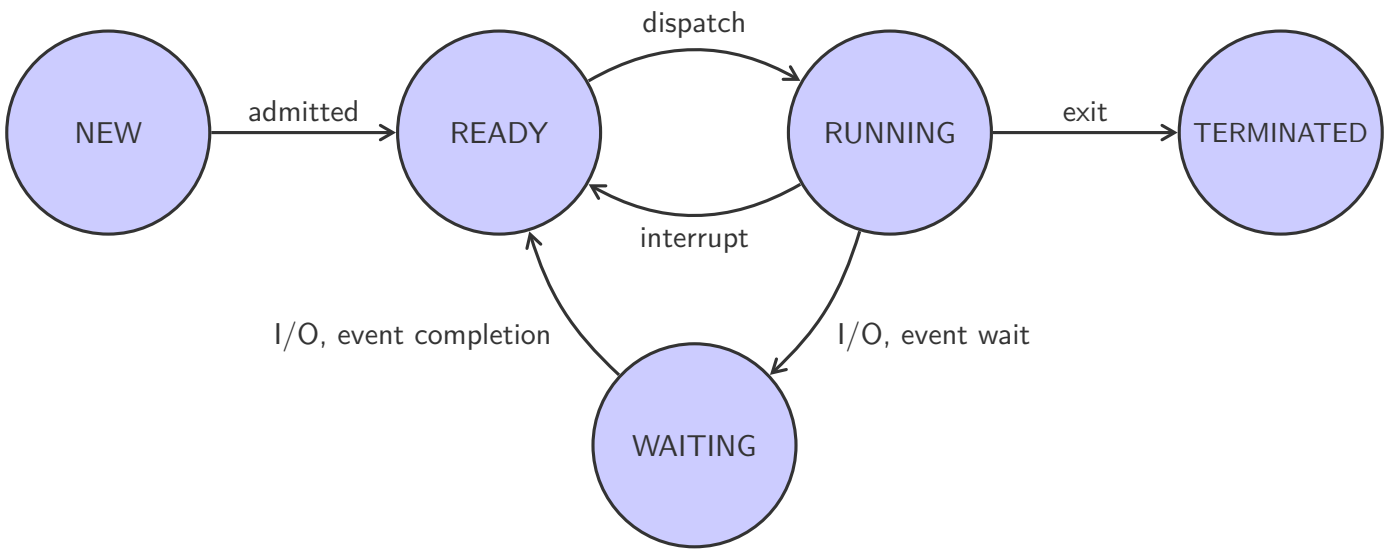
Threads

Thread

- Processus léger à l'intérieur d'un processus

  Plusieurs états lors de l'exécution d'un thread

- Un thread terminé ne peut pas être redémarré



Application multi-threads

- Exécution de plusieurs threads d'exécution indépendants

  Partage des ressources avec le processus hôte

- Gestion des threads possible par le système d'exploitation

  Ou gestion directe par le processus

- Gestion de la concurrence pas évident

  - Synchronisation et plusieurs threads
  - Évitement des interblocages (deadlocks)

Attendre un certain temps

- Blocage du programme avec fonction `sleep` du module `time`

  Blocage uniquement du thread qui a exécuté l'appel

- Spécification du temps de blocage en secondes (flottant)

```
1  import sys
2  import time
3
4  print('Bonjour', end='')
5  sys.stdout.flush()
6  time.sleep(1.5)
7  print(' toi.')
```

```
Bonjour toi.
```

Planifier une action

- Planification d'une exécution à l'aide d'un objet `Timer`

  Choix de la fonction à exécuter et du temps d'attente

- Création d'un objet `Timer` puis appel à la méthode `start`

```
1  import threading as th
2
3  def sayhello(name):
4      print('Hello', name)
5
6  t = th.Timer(5.5, sayhello, args=('Bob',))
7  t.start()
8  print('Timer started')
```

```
Timer started
Hello Bob
```

Répéter une action

- Répétition d'actions avec le module `sched`

  Ajout d'action à planifier avec la méthode `enter`

- Création d'un objet `scheduler` puis appel à la méthode `run`

```
1  import sched
2  from datetime import datetime
3  import sys
4  import time
5
6  def printhour():
7      print('\r{0:%H:%M:%S}'.format(datetime.now()), end='')
8      sys.stdout.flush()
9      scheduler.enter(1, 1, printhour)
10
11 scheduler = sched.scheduler(time.time, time.sleep)
12 scheduler.enter(1, 1, printhour)
13 scheduler.run()
```

Module threading

- Parallélisme par création de processus légers

  Création plus rapide par rapport à un processus

- Un thread possède un nom

  Accessible par l'attribut name

```
1  import concurrent.futures as cf
2
3  def compute(data):
4      return data ** 2
5
6  with cf.ThreadPoolExecutor(3) as executor:
7      print(list(executor.map(compute, [1, 7, 8, -2])))
```

```
[1, 49, 64, 4]
```

Lancer un thread

- Thread représenté par un objet `Thread`

  Code du thread défini par une fonction passée au constructeur

- Méthode start pour lancer le thread

```
1  import threading as th
2
3  def sayhello(name):
4      print('Hello', name)
5
6  thread = th.Thread(target=sayhello, args=('Tom',))
7  thread.start()
8
9  thread.join() # Attendre la fin du thread
10 print('Thread', thread.name, 'terminé')
```

```
Hello Tom
Thread Thread-1 terminé
```

Protection des données partagées (1)

- Les threads partagent les données du processus

  Attention aux modifications concurrentes des données partagées

- Utilisation d'un lock pour protéger l'accès

  Un seul thread à la fois accède à la donnée partagée

- Lock représenté par objet `Lock`

  Méthode `acquire` pour obtention et `release` pour libération

Protection des données partagées (2)

```
1  from random import randint
2  import threading as th
3  from time import sleep
4
5  counter = 0
6  results = {}
7
8  def compute(name):
9      global counter
10     counter += 1
11     sleep(randint(0, 1))
12     results[name] = counter
13
14 threads = [th.Thread(target=compute, args=(name,)) for name in ['
   Dan', 'Tom']]
15 for thread in threads:
16     thread.start()
17 for thread in threads:
18     thread.join()
19 print(results)
```

```
{'Tom': 2, 'Dan': 2}
```

Protection des données partagées (3)

```
1  # ...
2
3  counter = 0
4  results = {}
5  lock = th.Lock()
6
7  def compute(name):
8      global counter
9      lock.acquire()
10     counter += 1
11     sleep(randint(0, 1))
12     results[name] = counter
13     lock.release()
14
15 # ...
```

```
1  # ...
2
3  counter = 0
4  results = {}
5  lock = th.Lock()
6
7  def compute(name):
8      global counter
9      with lock:
10         counter += 1
11         sleep(randint(0, 1))
12         results[name] = counter
13
14
15 # ...
```

```
{'Tom': 2, 'Dan': 1}
```

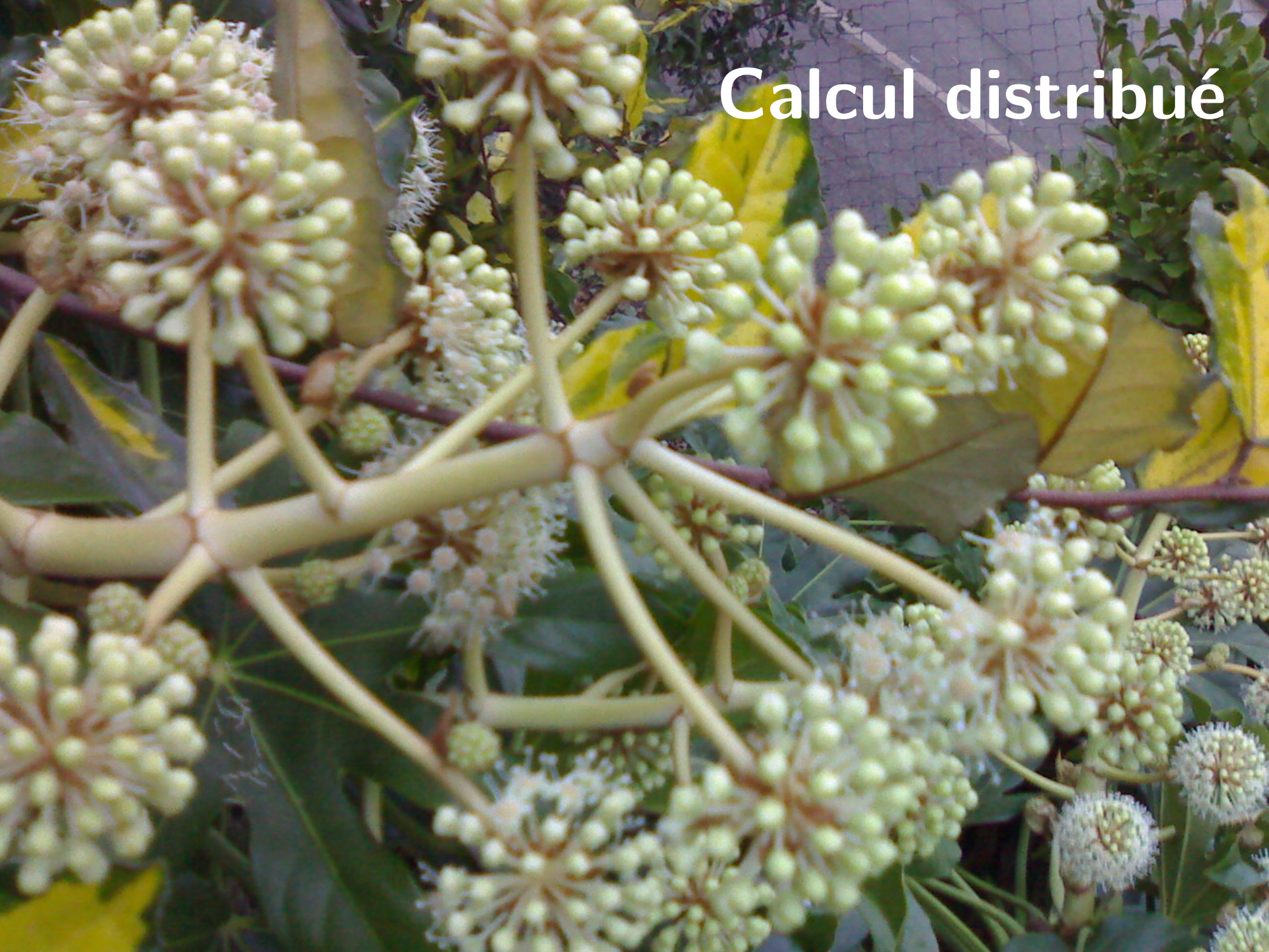
Pool d'executor

- Création d'un `Pool` de threads

  Nombre de threads dans le pool à spécifier au constructeur

- Plusieurs méthodes pour lancer une exécution
  - `submit` fait exécuter une fonction par un executor
  - `map` fait exécuter une fonction avec plusieurs données

```
1  import concurrent.futures as cf
2
3  def compute(data):
4      a, b = data
5      return a + b
6
7  with cf.ThreadPoolExecutor(3) as executor:
8      print(list(executor.map(compute, [(1, 2), (7, -4), (9, 3)])))
```



Calcul distribué

Framework `dispy`

- Création et utilisation de clusters pour du calcul parallèle
  - Plusieurs processeurs sur la même machine (*SMP*)
  - À travers plusieurs machines dans un cluster
- Adapté pour le parallélisme de données (SIMD)

  Le même programme est exécuté avec des données différentes
- Gestion dynamique des nodes du cluster

  L'ordonnanceur de `dispy` répartit les jobs

Composants

- **dispy** (client)

  Création d'un cluster (`JobCluster` ou `SharedJobCluster`)

- **dispynode** (serveur)

  Exécution de jobs reçus d'un client, tourne sur chaque node

- **dispyscheduler** (exécution partagée)

  Ordonnance les jobs lorsque en mode `SharedJobCluster`

- **dispynetrelay** (serveurs à distance)

  Réparti les jobs lorsque les nodes sont sur différents réseaux

Démarrage d'un node

- Démarrage du programme `dispynode.py` sur chaque node

  Possibilité d'interroger le serveur pour avoir des stats

```
$ dispynode.py
2016-03-06 10:26:22,305 - dispynode - dispynode version 4.6.9
2016-03-06 10:26:22,780 - dispynode - serving 4 cpus at
192.168.1.4:51348
Enter "quit" or "exit" to terminate dispynode,
  "stop" to stop service, "start" to restart service,
  "cpus" to change CPUs used, anything else to get status: BLABLA

  Serving 4 CPUs
  Completed:
    0 Computations, 0 jobs, 0.000 sec CPU time
  Running:

Enter "quit" or "exit" to terminate dispynode,
  "stop" to stop service, "start" to restart service,
  "cpus" to change CPUs used, anything else to get status:
```

Exécution d'un job (1)

- Définition d'un job par une fonction

  Rien de particulier à faire, fonction tout à fait classique

```
1  import dispy
2  import random
3  import time
4
5  def compute(n):
6      result = n ** 2
7      time.sleep(random.randint(1, 5))
8      return (n, result)
9
10 # ...
```

Exécution d'un job (2)

- Création d'un nouveau cluster de type `JobCluster`

  Envoi des jobs sur le cluster avec la méthode `submit`

- Attente de la fin de l'exécution d'un job avec `job()`

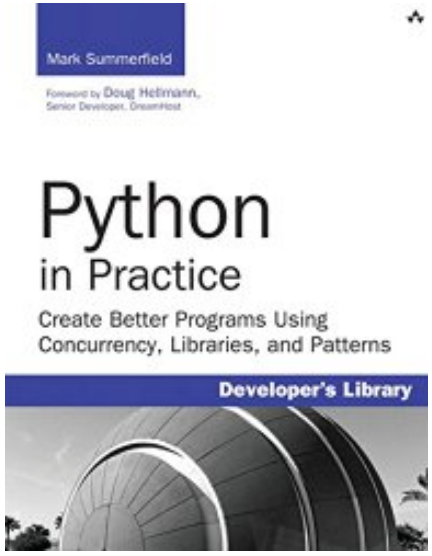
  Possibilité de récupérer pleins de statistiques sur le job

```
1  # ...
2
3  cluster = dispy.JobCluster(compute)
4  jobs = []
5  for i in range(10):
6      job = cluster.submit(random.randint(0, 1000))
7      job.id = i
8      jobs.append(job)
9
10 for job in jobs:
11     n, result = job()
12     print('Job #{} : Le carré de {} est {}'.format(job.id, n,
       result))
```

Livres de référence

ISBN

978-1-783-28839-7



ISBN

978-0-321-90563-5

Crédits

- Photos des livres depuis Amazon
- https://www.flickr.com/photos/berkeleylab/4025867073
- https://en.wikipedia.org/wiki/File:IBM_Electronic_Data_Processing_Machine_-_GPN-2000-001881.jpg
- https://www.flickr.com/photos/lovelihood/5399818533
- https://www.flickr.com/photos/toxi/1795660108