TDLOG séance 9 - concurrence

Xavier Clerc – xavier.clerc@enpc.fr

8 décembre 2025

Table des matières

La programmation concurrente constitue un sujet complexe, qui pourrait sans difficulté occuper un semestre entier. L'objectif de ce document est en conséquence seulement de présenter un aperçu de la terminologie ainsi que les questions et concepts élémentaires.

1 Généralités

1.1 *The free lunch is over*

L'expression *free lunch* en informatique désigne le fait que l'augmentation exponentielle de la puissance des machines pouvait permettre aux développeurs d'*attendre* l'arrivée de machines plus rapides plutôt que d'optimiser leurs codes...

En effet, si la puissance des machines double tous les 18 mois et qu'à un instant donné un programme est deux fois trop lent pour l'usage que l'on souhaite en faire, il *suffit* d'attendre 18 mois et d'acheter une nouvelle machine pour que le programme devienne *gratuitement* (du point de vue du développeur) suffisamment rapide...

Or il se trouve que la loi de Moore, qui prédit empiriquement un doublement du nombre de transistors par CPU tous les 18 mois, s'est également observée sur les fréquences des processeurs (et donc, en première approximation, sur leurs puissances) pendant plusieurs décennies. Malheureusement, depuis une vingtaine d'années, la fréquence des processeurs plafonne : ils ont atteint les 3 GHz au début des années 2000 et atteignent au mieux 4 GHz aujourd'hui.

Cependant, la loi de Moore continue à s'observer : il y a toujours une augmentation exponentielle du nombre de transistors, mais ceux-ci se répartissent dans plusieurs cœurs. On observe ainsi aujourd'hui une variante de la loi de Moore sur le nombre de cœurs plutôt que sur la fréquence des processeurs.

C'est pour cette raison que l'on dit que le *free lunch is over*, car pour pouvoir tirer parti de la puissance accrue des processeurs, il est maintenant nécessaire d'être capable de mobiliser tous les cœurs d'exécution. Le repas n'est plus gratuit parce qu'un programme doit être réécrit pour être en mesure de s'exécuter en parallèle sur plusieurs cœurs.

De plus, outre ces contraintes issues du matériel, l'évolution de la quantité de données à traiter est, elle aussi, exponentielle (*cf. big data*). La combinaison des nouvelles architectures (multicœurs, GPU, *etc*) et des volumes de données à traiter (souvent dans des temps très courts) nécessitent de plus en plus l'écriture de codes parallèles.

1.2 Concurrence, parallélisme et distribution

L'objet de la programmation concurrente est de s'intéresser à l'exécution d'un programme au sein duquel plusieurs parties s'exécutent dans un même laps de temps. Cette définition est volontairement un peu floue car la programmation concurrente possède une acception assez large et pour certains auteurs subtilement différente de la programmation parallèle.

En effet, lorsque l'on parle de programmation parallèle, on indique clairement que plusieurs parties d'un programme s'exécutent effectivement de manière simultanée alors que lorsque l'on parle de programmation concurrente, on indique que plusieurs parties d'un programme sont au même instant éligibles pour s'exécuter. Dans le second cas, les différentes parties du programme peuvent ou non s'exécuter de manière simultanée selon des contraintes logicielles et/ou matérielles. Par exemple, sur un système qui ne dispose que d'un seul cœur d'exécution, on peut lancer des calculs en plusieurs points du programme mais l'exécution ne pourra être simultanée : les différents points d'exécution seront en concurrence pour l'accès au cœur d'exécution.

Cependant, le partage du temps entre les différentes tâches se fait selon des intervalles tellement courts que l'utilisateur observant le système peut avoir l'impression que les tâches s'exécutent simultanément. De plus, la concurrence est utile par exemple lors d'entrées-sorties : pendant qu'un point du programme est *p. ex.* en attente de la lecture/écriture d'une valeur dans un fichier, un autre point peut s'exécuter.

En marge de ces notions de programmation concurrente et de programmation parallèle, on ajoute souvent un troisième mode d'exécution par le biais de la programmation distribuée. La programmation distribuée concerne l'exécution d'un logiciel sur plusieurs machines simultanément et implique donc une programmation parallèle. Cependant, elle ne se réduit pas à cela et ajoute de

nouvelles contraintes. En effet, lorsqu'un programme s'exécute sur plusieurs machines, de nouvelles erreurs peuvent se produire. Typiquement, des problèmes de réseau peuvent empêcher ou retarder l'envoi de données depuis une machine vers une autre. Il est également possible qu'une machine cesse de fonctionner alors que les autres continuent leur exécution. Ces nouveaux types d'erreurs, qui n'existent pas lorsque le programme s'exécute sur une seule machine, doivent bien entendu être pris en considération par le développeur en plus des problèmes déjà complexes introduits par la programmation concurrente que nous allons aborder maintenant.

1.3 Organisation de la mémoire

Une distinction fondamentale au sein de la programmation concurrente repose sur l'organisation de la mémoire et la manière dont le développeur peut y accéder. On distingue deux cas : la mémoire partagée et la mémoire distribuée. Dans le premier cas, les différentes tâches qui s'exécutent peuvent toutes accéder à l'ensemble des valeurs manipulées par le programme. Dans le second cas, chaque tâche a accès à seulement une partie des valeurs manipulées et le partage de valeurs entre différentes tâches se fait par le biais de communications explicites, par exemple en envoyant ces valeurs via le réseau. Il est important de noter que le caractère distribué ou non des calculs n'a qu'une influence majeure mais pas totale sur le caractère distribué ou non de la mémoire. Ainsi, il est possible d'être dans un cadre de mémoire distribuée même si l'exécution ne se déroule que sur un seul ordinateur (*cf.* processus ci-après). Il est également possible (mais nettement plus rare) d'être dans un cadre de mémoire partagée même si l'exécution se déroule sur plusieurs machines.

1.4 Performances

Lorsque l'on s'intéresse aux performances d'un code parallèle, on effectue le plus souvent une comparaison en fonction du nombre de cœurs que l'on utilise pour le calcul. Ainsi, il est d'usage de définir l'accélération (ou *speedup*) de la manière suivante :

$$S(n) = \frac{T_s}{T_p(n)}$$

avec :

- T_s le temps d'exécution du programme séquentiel ;
- T_p le temps d'exécution du programme parallèle ;
- n le nombre de cœurs utilisés.

On peut alors observer trois cas :

- $1 < S(n) < n$, le plus commun, qui correspond à un gain plus ou moins important de la version parallèle sur la version séquentielle ;
- $S(n) < 1$ qui correspond soit à une erreur de programmation, soit au fait que le surcoût du parallélisme (*p. ex.* transmission de données entre les différentes unités d'exécution) est supérieur à son gain ;
- $S(n) > n$ qui correspond soit à une erreur de mesure, soit au fait que la répartition des calculs sur plusieurs unités d'exécution permet de mieux utiliser les *caches* (permettant une accélération qualifiée de *superlinéaire*).

Il existe pour chaque programme, en fonction de sa structure, une limite à l'accélération que l'on peut espérer obtenir. En effet, chaque programme est composé de parties qui peuvent s'exécuter en parallèle et de parties qui sont intrinsèquement séquentielles. Si l'on note p la fraction du

programme qui est parallélisable, l'accélération maximale en fonction de n le nombre de cœurs est donnée par la loi de Amdahl :

$$\frac{1}{(1-p)+\frac{p}{n}}$$

Bien entendu, cette loi repose sur un modèle simple qui ne tient pas compte, entre autres, des synchronisations et communications entre cœurs ou encore des effets des caches. Elle n'a donc pas un caractère prédictif mais seulement indicatif.

Enfin, un autre critère important est celui du passage à l'échelle (*scalability*), qui correspond à la capacité d'un code parallèle à mobiliser utilement plus d'unités d'exécution. En effet, il est fréquent que l'on observe une accélération jusqu'à un certain nombre d'unités avant de constater un ralentissement. Cela tient au fait qu'en multipliant les unités d'exécution on divise d'autant la taille des calculs à effectuer par unité, et que cette taille passe sous le seuil où les transmissions sont plus coûteuses que le calcul lui-même.

2 Programmation à l'aide de *threads*

2.1 Notion de thread

Lors de la programmation d'une application utilisant le réseau, il est fréquent de devoir gérer simultanément plusieurs connexions, ou d'être en mesure de poursuivre des traitements alors même que l'on attend des données en provenance d'une autre machine. Il devient alors précieux de pouvoir exécuter plusieurs opérations de manière concurrente au sein d'un même programme.

La notion de *thread* est une réponse à cette problématique. On peut voir un programme classique comme étant *mono-thread* : il y a un fil d'exécution et à chaque instant le programme exécute une instruction. Dès lors que l'on crée des threads additionnels, il y a plusieurs fils d'exécution, ce qui signifie qu'à chaque instant le programme exécute *simultanément* plusieurs instructions. Chaque thread dispose de sa propre pile mais les différents threads s'exécutent dans le même espace mémoire. En pratique, cela signifie que chaque thread possède sa propre copie des variables locales mais que les variables globales sont partagées entre les threads. Nous sommes donc dans le cadre d'une mémoire partagée.

On distingue classiquement quatre usages principaux pour les threads :

- le calcul parallèle : chaque cœur du microprocesseur se charge d'un thread, ce qui permet d'augmenter d'autant la puissance de calcul ;
- les interfaces graphiques : le programme continue ses calculs et est réactif, attendant simultanément des actions de l'utilisateur ;
- les communications (qui incluent le cas particulier de l'utilisation de fichiers) : pendant qu'un programme envoie ou reçoit des données, il est à même de mener des calculs ;
- la structuration d'un programme qui peut s'exprimer plus clairement ainsi (*p. ex.* dans le cas d'une architecture en *pipeline* ou producteur-consommateur).

Ces cas d'usage amènent naturellement à se poser une question centrale de la programmation multi-thread : celle de la synchronisation. En effet, si plusieurs threads tentent d'accéder au même instant à une même donnée, que va-t-il se passer ? Les accès en lecture ne posent pas de problème, et les différents threads récupéreront la même valeur. En revanche, si plusieurs threads tentent de modifier une valeur, il est crucial pour la correction d'un algorithme de savoir quelle valeur va être effectivement écrite.

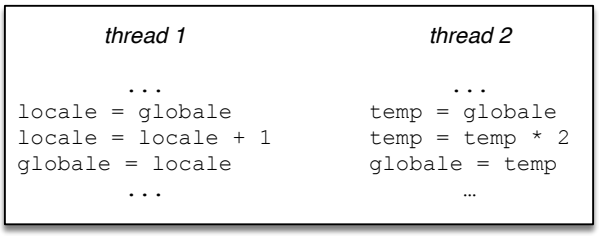

FIGURE 2.1 – Exécution de threads (sans verrou).

Supposons que notre programme soit composé de deux threads et qu'à un instant donné ceux-ci exécutent les portions de code montrées par la figure Figure 2.1. Les deux threads s'exécutent en concurrence, et il n'y a aucune contrainte d'ordonnancement des instructions. Ainsi, il est possible d'avoir l'enchevêtrement suivant :

- exécution de `locale = globale;`
- exécution de `temp = globale;`
- exécution de `locale = locale + 1;`
- exécution de `globale = locale;`
- exécution de `temp = temp * 2;`
- exécution de `globale = temp`.

On voit sur cet exemple que la modification de la valeur `globale` faite par le thread numéro 2 est bien prise en compte alors que celle faite par le thread numéro 1 est perdue.

Afin de réguler de tels accès (dits *concurrents*) à une même donnée, on utilise classiquement un concept de synchronisation : le *verrou*. L'idée sous-jacente est de garantir qu'à un instant donné un seul thread manipule les données associées à ce verrou. Un thread peut demander à acquérir un verrou : si aucun thread n'a déjà acquis le verrou alors il l'obtient pour son usage exclusif, si un thread a déjà acquis le verrou alors il doit attendre qu'il soit disponible. Symétriquement, le thread qui détient le verrou peut décider de le *relâcher*, ce qui signifie qu'il le rend disponible aux autres threads.

Équipés de la notion de verrou, nous pouvons modifier notre programme comme le montre la figure Figure 2.2 pour nous assurer que chaque thread n'utilisera la variable `globale` que lorsqu'il en aura le droit, c'est-à-dire après avoir acquis le verrou (lui-même stocké comme une variable globale). Pour qualifier le code se trouvant entre l'acquisition et le relâchement d'un verrou, on parle de *section atomique* [1]. Un seul thread à la fois peut se trouver dans la section atomique et l'atomicité est due au fait que l'on peut alors raisonner sur le programme comme si l'ensemble des instructions de la section était exécuté en une seule opération, c'est-à-dire sans qu'un autre thread demandant le même verrou puisse exécuter d'autres instructions.

2.2 Utilisation de *threads*

Pour déclarer un thread en Python, il faut écrire une classe dérivant de la classe `threading.Thread`. Au sein de cette classe, la méthode `run()` doit contenir le code que le thread exécutera lorsqu'il sera lancé, le lancement effectif du thread se faisant en appelant la méthode `start()` sur une instance de la classe.

Une autre méthode importante de la classe `threading.Thread` est `join()` qui permet d'attendre la fin de l'exécution d'un thread. Le code Code 2.1 déclare une classe `T` dont les ins-

1. On parle également de section critique, ou de bloc *protégé*.

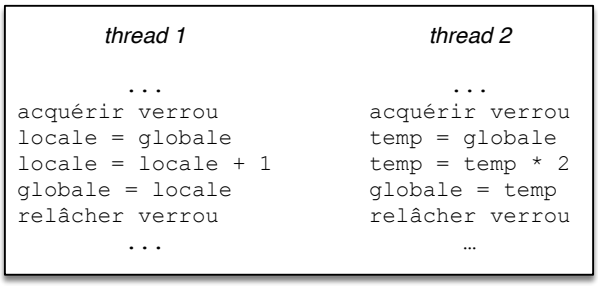

FIGURE 2.2 – Exécution de threads (avec verrou).

tances afficheront à l'écran dix fois le même message, en attendant entre 0 et 2 secondes entre deux affichages. On notera qu'il y a au total 3 threads : le thread principal (créé au début du programme) et les deux threads liés aux instances de `T`. Le thread principal se contente dans cet exemple de lancer les autres threads puis d'attendre qu'ils aient fini leurs traitements.

Code 2.1 – Threads affichant des messages.

```
1  import threading
2  import time
3  import random
4
5  class T(threading.Thread):
6    def __init__(self, msg):
7      super().__init__()
8      self.message = msg
9    def run(self):
10     for i in range(10):
11       time.sleep(random.randint(0,2))
12       print(self.message)
13
14 t1 = T("bonjour")
15 t2 = T("hello")
16 t1.start()
17 t2.start()
18 t1.join()
19 t2.join()
```

Du fait de l'aléa existant dans le programme sur les temps d'attente entre deux affichages, différentes exécutions vont produire des résultats différents. Cela pose inévitablement des problèmes de mise au point et de debuggage des programmes car il n'est pas facile de reproduire un comportement observé. Pour caractériser le fait que différentes exécutions ne donnent pas forcément le même résultat même si on fournit les mêmes entrées, on parle de non-déterminisme. Il est très important d'être conscient qu'il est tout à fait possible d'avoir un programme non-déterministe même si l'on n'introduit pas explicitement d'aléa dans le code source du programme. En effet, lorsque plusieurs threads sont utilisés de manière concurrente dans un programme, il n'est pas toujours possible de savoir lequel(s) va (vont) avoir accès à un (des) cœurs d'exécution. Cela reste par ailleurs vrai même si le programme lance moins de threads qu'il n'y a de cœurs d'exécution sur la machine : en effet, les systèmes d'exploitation sont multi-tâches et des cœurs

peuvent être accordés de manière variable aux différents programmes au fil du temps. En outre, si le programme accède à des fichiers ou reçoit des informations par le réseau, les temps d'accès et d'attente sont variables et influent donc sur la manière dont chaque thread exécute la tâche dont il a la charge.

2.3 Utilisation de verrou

Lors de l'exécution du programme correspondant au code Code 2.1, on constate parfois (selon les tirages aléatoires) des "collisions" entre les messages qui peuvent s'afficher sur une même ligne. Si l'on souhaite s'assurer d'un affichage correct, il est nécessaire d'ajouter un verrou. Plus généralement un verrou permet de s'assurer qu'une portion donnée du code ne peut être exécutée que par un seul thread à la fois. Cela se fait en créant une instance par `verrou = threading.Lock()` puis en encadrant l'instruction d'affichage des threads par :

— `verrou.acquire()` avant ;
— `verrou.release()` après.

L'appel à `acquire` tente d'acquérir le verrou : s'il est libre le thread appelant l'acquiert, s'il est déjà acquis par un autre thread alors le thread attend que le premier le relâche. L'appel à `release` permet au thread détenant le verrou de le relâcher.

On garantit ainsi que `print(self.message)` ne peut être exécuté que par un seul thread à la fois. Cependant, il faut être attentif au fait qu'à chaque appel à `acquire` doit correspondre un appel à `release`, ce qui n'est pas toujours facile à garantir, par exemple si le code se trouvant entre les deux appels est susceptible de lever une exception. Pour s'assurer que `release` sera toujours appelé, le plus simple et le plus robuste est d'utiliser le verrou comme une ressource par le biais d'une instruction `with`. Dans notre cas cela se fera par `with verrou: ...`. Contrairement à d'autres usages de `with`, il est crucial de ne pas créer un nouveau verrou via cette construction syntaxique. En effet, il est absolument nécessaire que les différents threads partagent le même verrou pour que la synchronisation puisse se faire.

Les verrous que nous avons vus sont les plus communs et les plus simples à manipuler. Il existe une variante proposant les mêmes méthodes qui est désignée par le terme de verrou réentrant. Seule la méthode `acquire` diffère :

— dans un verrou classique, un appel est bloquant si le verrou est déjà détenu par un thread, même s'il s'agit du thread appelant `acquire` ;
— dans un verrou réentrant, `acquire` n'est bloquant que si le verrou est détenu par un thread autre que le thread appelant.

Dans l'ensemble, les verrous doivent être vus comme une stratégie d'implémentation d'une notion plus générale : celle de section critique. Une section critique (on trouvera parfois également l'adjectif *atomique* dans la littérature) est une portion de code qui ne peut être exécutée que par un seul thread à fois. D'où le terme *atomique* puisqu'il est possible de raisonner sur le programme en considérant que la portion de code s'exécute "en une seule fois".

2.4 Synchronisation avancée

Le concept de verrou fournit un moyen de base permettant la synchronisation de plusieurs threads. Il existe des mécanismes d'abstraction plus élevée, tels que les conditions ou les files.

Les conditions reposent sur les verrous mais donnent accès à des fonctionnalités plus avancées : outre la prise et le relâchement, elles permettent de demander d'attendre qu'un événement soit signalé et de signaler un événement. L'attente se fait par le biais de la méthode `wait` de la classe

`threading.Condition`; le thread qui appelle cette méthode est alors arrêté jusqu'à ce qu'un autre thread appelle la méthode `notify` (ou `notify_all`) sur la même instance de `threading.Condition`. Les conditions sont classiquement utilisées dans des systèmes de type producteur consommateur, comme illustré par le programme Code 2.2. Dans un tel système, le producteur génère des valeurs qui sont passées au consommateur pour traitement.

Code 2.2 – Threads producteur et consommateur.

```
import threading

cond = threading.Condition()
valeurs = []

# le thread consommateur attend qu'une valeur soit disponible
...
with cond:
  while len(valeurs) == 0: cond.wait()
  valeur = valeurs.pop()
  ... traitement sur 'valeur'

# le thread producteur signale chaque valeur ajoutée
...
with cond:
  valeur = ...
  valeurs.append(valeur)
  cond.notify()
```

Dans le programme Code 2.2, on voit que les appels aux méthodes de la condition se font au sein d'une construction `with`. En effet, il ne faut appeler ces méthodes que si l'on a acquis la condition (par la méthode `acquire`, comme pour un verrou). Afin de s'en assurer, et également pour ne pas oublier de libérer le verrou lorsque l'on n'en a plus besoin, on préfère placer ces appels dans une construction `with`. Au début du bloc, la condition est acquise et elle est relâchée à la sortie du bloc.

L'exemple de synchronisation utilisant une condition pour un système producteur consommateur est si commun en pratique que Python propose une classe "file d'attente" qui permet de le programmer facilement : la classe `Queue.Queue` qui se charge des synchronisations. L'exemple Code 2.2 peut alors être écrit de manière nettement plus simple, comme le montre le code Code 2.3.

Code 2.3 – Threads producteur et consommateur (bis).

```
import threading
import Queue

valeurs = Queue.Queue()

# le thread consommateur attend qu'une valeur soit disponible
...
```

(suite sur la page suivante)

(suite de la page précédente)

```
valeur = valeurs.get() # bloquant si la file est vide

# le thread producteur signale chaque valeur ajoutée
...
valeur = ...
valeurs.put(valeur)
```

2.5 Concurrence implicite

Plutôt que de créer explicitement des threads et collecter leurs résultats en utilisant variables globales et verrous, il est possible de soumettre des tâches à exécuter qui le seront dès que des threads seront disponibles. Le code Code 2.4 montre comment créer un pool de threads (dont la taille s'adapte aux nécessités mais peut-être bornée par le paramètre `max_workers` passé au constructeur). Ensuite, une tâche peut être soumise par le biais de la méthode `submit` qui retourne une instance de la classe `Future`. Enfin, on peut attendre que la tâche soit terminée en appelant `result` qui retourne le résultat renvoyé par la fonction passée à `submit`. Si la tâche n'a pas terminé son exécution, cet appel est bloquant.

Code 2.4 – Exemple de future.

```
import concurrent.futures

def tache(x):
  ...
  return ...

pool = concurrent.futures.ThreadPoolExecutor(max_workers=4)

f1 = pool.submit(tache, param)
...
print(f1.result())
```

L'utilisation d'un pool de threads ne permet pas seulement de soumettre des tâches, il permet de mutualiser l'utilisation des threads. En effet, lorsqu'une tâche est soumise, elle est automatiquement allouée à un thread le plus souvent préexistant, ce qui permet d'éviter de payer le coût de la création d'un nouveau thread. Lorsqu'une tâche est terminée, le thread retourne dans le pool pour être de nouveau utilisé par une prochaine soumission.

Outre `submit`, il est également possible d'utiliser la méthode `map` sur un pool de threads. Elle se comporte comme la fonction `map` à ceci près qu'elle est capable de mobiliser plusieurs threads pour son exécution (les différents threads calculent les résultats pour les différents éléments de la liste passée). Bien entendu, le résultat n'est équivalent à une exécution séquentielle que si les différents appels sont indépendants.

Enfin, pour demander la destruction des threads utilisés par le pool, il est nécessaire d'appeler la méthode `shutdown`. Il n'est alors plus possible de demander de nouveaux calculs au pool.

2.6 Notion de *deadlock*

On a vu la nécessité de synchroniser les accès à des données dès lors que plusieurs threads sont susceptibles de tenter d'y accéder de manière concurrente. Les verrous permettent de sérialiser les accès afin de garantir l'intégrité des données. Cependant, les verrous sont susceptibles de provoquer un certain nombre de problèmes, le plus connu étant celui appelé *deadlock*.

Cela se produit typiquement lorsqu'un thread *t1* détient un verrou *v1* et tente d'acquérir un verrou *v2* lui-même détenu par un thread *t2* qui tente d'acquérir le verrou *v1*. Dans cette situation, chaque thread va attendre indéfiniment qu'un verrou soit relâché, ce qui ne peut se produire.

Pour qu'un *deadlock* se produise, il est nécessaire que quatre conditions se combinent :
- exclusion mutuelle ;
- détention et tentative d'acquisition d'un verrou ;
- impossibilité de préempter un verrou ;
- attente circulaire.

Comme la combinaison de ces quatre conditions est nécessaire, il suffit de s'assurer que l'une ne se produit jamais pour garantir l'absence de *deadlock*. On peut en tirer un certain nombre de stratégies, les deux plus simples étant :
- une stratégie fondée sur l'ordre ;
- une stratégie fondée sur le retrait.

La stratégie fondée sur l'ordre consiste à s'assurer que tous les threads acquièrent toujours les verrous dans le même ordre (il suffit de fixer un ordre arbitraire sur les verrous). La stratégie fondée sur le retrait consiste, lorsque l'on cherche à acquérir un verrou, à procéder en deux étapes :

1. tenter d'acquérir un verrou par `acquire(False)` qui ne bloque jamais et renvoie un booléen indiquant si le verrou a été effectivement acquis ;
2. si le verrou n'a pas pu être acquis, relâcher les verrous déjà détenus.

Ces stratégies semblent relativement simples à implémenter, mais le problème pratique réside dans le fait qu'il faut s'assurer que la stratégie s'applique sur **tous les chemins d'exécution**.

2.7 Notion de *Global Interpreter Lock*

Les éléments présentés dans cette section (threads, verrous, conditions) sont essentiellement indépendants du langage de programmation utilisé. Python jusqu'à 3.12 souffre cependant d'un problème qui affecte relativement peu de langages : l'existence d'un *Global Interpreter Lock*. Cela signifie en pratique que toute exécution de code Python se fait sous un verrou. Et donc en conséquence que même si plusieurs threads sont lancés, à un instant donné, seul un peut s'exécuter.

La conséquence immédiate et insurmontable est que l'utilisation de threads avant Python 3.13 ne peut pas servir à mobiliser les différents cœurs d'une machine dans le but d'accélérer les traitements par leur parallélisation (si Python appelle du code C, plusieurs threads C peuvent s'exécuter en parallèle - voir séance sur l'interopérabilité). En revanche, le modèle des threads reste utile dans les autres circonstances évoquées en cours de section : en particulier, les interfaces graphiques et communications (*p. ex.* liées aux fichiers ou au réseau) dans lesquelles un thread attend une action de l'utilisateur ou l'arrivée d'une donnée.

À partir de Python 3.13, le *global interpreter lock* n'existe essentiellement plus, et on peut donc utiliser les threads our mobiliser les différents cœurs de la machine.

3 Programmation à l'aide de processus

3.1 Notion de processus

Du fait du *Global Interpreter Lock*, si l'on désire paralléliser des calculs en Python avant 3.13, il est nécessaire de mobiliser plusieurs processus. Les processus sont simplement des instances de programmes qui s'exécutent de manière concurrente au sein du système d'exploitation. S'agissant de programmes disjoints (même s'il s'agit de plusieurs instances du même programme), ils possèdent chacun leur propre espace mémoire et l'on se trouve donc dans le cadre d'une mémoire distribuée (y compris si les différentes instances de programmes s'exécutent sur la même machine).

3.2 Utilisation de processus

Le plus simple est d'avoir recours au module `multiprocessing`. Dans ce cas, il ne s'agit plus d'utiliser plusieurs threads qui partagent le même espace mémoire, mais d'utiliser plusieurs processus qui doivent explicitement communiquer par messages pour partager des données. Le code Code 2.3 montre comment créer un processus en lui passant le nom de la fonction à exécuter et les arguments qu'elle recevra. La création se fait sous la condition `if __name__ == '__main__':` car elle ne doit avoir lieu que si l'on se trouve dans le programme principal. La mécanique du module `multiprocessing` va se charger de créer le processus avec un autre nom et d'appeler au sein de ce nouveau processus la fonction désignée. Comme pour les threads, la méthode `join` permet d'attendre la fin de l'exécution d'un processus.

Code 3.1 – Processus : création et communication explicites.

```
import multiprocessing

def tache(c, x):
  ...
  c.send(...)
  ...
  c.send(...)
  ...
  c.close()

if __name__ == "__main__":
  connexion_parent, connexion_enfant = multiprocessing.Pipe()
  processus_enfant = multiprocessing.Process(target=tache,
                                             args=(connexion_enfant, ..
↪.))
  processus_enfant.start()
  ...
  print(connexion_parent.recv())
  ...
  print(connexion_parent.recv())
  ...
  processus_enfant.join()
```

La communication se fait par le biais d'une instance de `Pipe` qui permet une communication bidirectionnelle par les méthodes `send` et `receive`. Dans notre exemple, la fonction chargée de l'exécution de la tâche envoie successivement deux données qui sont affichées par le processus principal.

L'utilisation des classes `Process` et `Pipe` donne un maximum de flexibilité, mais nécessite de passer explicitement les données. Dans les cas les plus simples, il est possible d'effectuer les créations de processus et les communications de manière implicite, par le biais de la classe `Pool`. Le code Code 3.2 donne un exemple d'utilisation de pool : sa création se fait en passant simplement le nombre de processus que l'on souhaite utiliser. Ensuite, il est possible de demander l'exécution d'une fonction sur différentes valeurs en appelant la méthode `map` qui est similaire à la fonction du même nom. Les données et les valeurs de retour de la fonction sont automatiquement envoyées d'un processus à l'autre. De plus, la méthode `map` se charge de répartir les données de `l` entre les différents processus : dès qu'un processus envoie le résultat de la fonction pour une certaine donnée, il reçoit une nouvelle donnée à traiter.

Code 3.2 – Processus : création et communication implicites.

```
import multiprocessing

def tache(x):
  ...
  return ...

if __name__ == "__main__":
  pool = multiprocessing.Pool(4)
  l = [ ... ]
  print(pool.map(tache, l))
```

Ce mode d'exécution correspond au modèle que l'on appelle *map-reduce* et a été popularisé par Google qui l'utilise intensivement pour distribuer les calculs sur des clusters de machines. Le modèle s'appelle *map-reduce* car outre l'exécution de la fonction par `map`, une seconde étape est exécutée pour "assembler" les résultats en un seul, typiquement par la fonction `reduce`. Pour mémoire, `functools.reduce(f, l, z)` retourne $f(z, f(e_0, f(e_1, \dots)))$ où $l = [\ e_0, e_1, \dots, e_n\ ]$. Bien entendu, tous les problèmes ne peuvent pas s'exprimer sous la forme d'une combinaison de `map` et de `reduce`, mais il est utile d'avoir ce modèle à l'esprit car il permet une parallélisation très facile des problèmes qui peuvent s'exprimer dans ce modèle contraint.

Annexes

À retenir

Pour comprendre les possibilités et difficultés liées à l'utilisation conjointe de plusieurs unités d'exécution, il est nécessaire de connaître les notions de concurrence/parallélisme/distribution et de mémoire partagée/distribuée. Il est important de faire la différence entre threads et processus et de savoir comment les utiliser en Python. Pour éviter des déconvenues, il est également important de savoir que Python (avant 3.13) est basé sur un *Global Interpreter Lock*. Enfin, comprendre la nécessité de la synchronisation des accès est déterminant pour être capable d'écrire

des programmes corrects.

Références

— https://docs.python.org/3/library/concurrency.html est la partie de la documentation Python présentant les différentes possibilités pour la programmation concurrente en Python.