Rapport
Gestion de la Concurrence

LERAS - MOLINIER

21 décembre 2019

1 Introduction

Ce rapport rend compte d'un projet visant à étudier les contraintes de la programmation concurrente liées au parallélisme et aux accès concurrents sur les données. Au travers de différents scénarios nous verrons quelles solutions peuvent être apportées en Python.

2 Principe de fonctionnement des algorithmes

2.1 Scénario 0 : thread unique

Algorithm 1 Scénario 0

```
1: procedure ALGORITHM0
2: people ← sorted(people, key=lambda x: (pow(x[0], 2) + pow(x[1], 2)))
3: while len(people) != 0:
4:     for p in people do
5:         if p can move outside then
6:             people.pop(p)
7:         if p can move diagonally then
8:             move
9:         if p can move horizontally then
10:            move
11:        if p can move vertically then
12:            move
```

2.2 Scénario 1 : une thread par personnes

Algorithm 2 Scénario 1

```
1: procedure ALGORITHM1
2: lock ← Lock()
3: tList ← []
4:     for p in people do
5:         tList.append(Thread(target=GoToExit, args=(p, lock)))
6:     for t in tList do
7:         t.start()
8:     for thread in tList do
9:         thread.join()
```

La méthode *GoToExit()* exécute la même logique que le pseudo-code du scénario 0 en cherchant un déplacement possible. La différence est qu'avant d'effectuer cette recherche il faut faire *lock.acquire()* et après avoir bougé faire

lock.release() afin de protéger la liste et d'éviter qu'un autre thread vienne prendre la place trouvée.

2.3 Scénario 2 : 4 threads

Algorithm 3 Scénario 2

```
1: procedure ALGORITHM2
2: tList ← []
3:     for i in range(1,4) do
4:         tList.append(Thread(target=subAlgorithm2, args=(i,)))
5:     for t in tList do
6:         t.start()
7:     for thread in tList do
8:         thread.join()
```

La méthode *subAlgorithm2()* s'occupe selon l'entier reçu de travailler sur l'une des 4 zones fraichement crées. Tout comme avec le scénario 1 il faut s'assurer de prendre les verrous avant chaque recherche et de les rendre après chaque déplacement. La particularité ici est qu'il faut parfois prendre 2 verrous lorsque la personne change de zone et donc change de liste.

3 Fonctionnement des bibliothèques utilisés

3.1 Bibliothèque threading.Thread

Cette bibliothèque Python permet l'implémentation du parallélisme grâce à l'utilisation de thread (multithreading). Le problème est que le GIL (Global Interpreter Lock) réduit fortement l'utilité des threads en Python car il ne permet l'exécution que d'un seul thread natif à la fois. Cela permet d'augmenter fortement les performances des programmes monothread qui sont bien plus répandu.

Il semble donc que le GIL tue le multithreading Python mais pas tout à fait. Il empêche généralement de profiter de plusieurs coeurs sur une seule machine mais permet de profiter de la latence entre les entrées/sorties.

Les scénarios qui vont suivre permettront peut-être de montrer le "faux" parallélisme du multithreading Python.

3.2 Librairie threading.Lock

La bibliothèque threading de Python fournit un mécanisme de verrouillage simple qui permet de synchroniser les threads. Ce système comprend 2 méthodes. La méthode *acquire()* permet de prendre le vérrou et de forcer les autres threads à attendre que celui-ci soit libéré par la méthode *release()*.

4 Analyse des performances

Afin de mesurer les performances des algorithmes, le paramètre -m permet de donner le temps moyen que met la foule à sortir du terrain. On reproduit 5 fois l'algorithme et on supprime les 2 valeurs extrêmes pour obtenir un résultat cohérent. Ainsi ce résultat n'est pas sensible au temps d'échauffement du processeur et les performances plus stables.

4.1 Scénario 0 : thread unique

Voici le tableau des performances du scénario 0 :

Taille de la foule	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9
Temps execution (ms)	1	1	3	7	16	38	122	390	1451
Temps CPU (ms)	1	1	3	7	16	38	122	390	1449

Table 1: Performances scénario 0

Les résultats ci-dessus montrent que le temps d'exécution de l'algorithme est très similaire au temps d' utilisation du CPU. Cela n'est pas surprenant car cet algorithme n'utilise pas de parallélisme. Les calculs ne sont réalisés que par un seul thread du processus main.

4.2 Scénario 1 : une thread par personnes

Voici le tableau des performances du scénario 1 :

Taille de la foule	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9
Temps execution (ms)	1	2	7	22	59	164	730	6248	86753
Temps CPU (ms)	1	3	7	32	85	236	1012	8101	102378

Table 2: Performances scénario 1

Tout d'abord le temps d'utilisation CPU a fortement augmenté avec ce deuxième algorithme. Il dépasse même le temps d'exécution ce qui n'est pas étonnant car celui-ci utilise autant de threads que de personnes il y a donc des calculs qui s'effectuent en parallèle. Par contre afin d'assurer la cohérence lors d'une recherche d'un déplacement possible et le déplacement en lui-même la liste est protégée par un verrou. Comme tous les threads tentent d'accéder à la même liste le temps d'attente pour prendre le verrou augmente fortement avec le nombre de personnes.

4.3 Scénario 2 : 4 threads

Voici le tableau des performances du scénario 2 :

Taille de la foule	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9
Temps execution (ms)	60	64	72	163	295	1099	1177	1748	4379
Temps CPU (ms)	57	64	72	163	297	1102	1183	1759	4424

Table 3: Performances scénario 2

Pour ce dernier scénario le temps d'utilisation CPU est légèrement supérieur au temps d'exécution. De plus le temps d'exécution reste bien supérieur à celui du scénario 1. En effet prendre et rendre un verrou sont des opérations coûteuses et le découpage en 4 zones ne solutionne pas le problème de bouchon près de la sortie.

5 Conclusion

Pour conclure, l'utilisation du parallélisme et du multithreading python n'est pas forcément synonyme de meilleures performances. Les scénarios précédents montrent clairement qu'assurer la protection et la cohérence des données entre les différents calculs sont des opérations coûteuses qui impactent lourdement les performances.