Emma BEGARD
Audrey NICOLLE
3 ETI B

Programmation Concurrente

– Projet –

Exercices d'introduction aux calculs parallèles

03/06/2022
10/06/2022

CPE LYON

Professeurs référents :
Tahar LIMANE
Alexandre
Anthony CHOMIENNE

Table des matières

I) Estimation de PI

Dans cet exercice on cherche à estimer la valeur de pi et nous allons comparer les résultats qui sont fait de manière séquentielle et parallèle. Dans les deux cas, nous avons utilisé la méthode Monte Carlo ou Hit-Miss. Le principe est d'échantillonner un point, soit un couple $(x, y) \in [0.0,1.0\,]$ qui se situe dans le quart du cercle unitaire et nous déterminons, d'après l'équation du cercle, si la valeur de $x^2 + y^2 \leq 1$. Si celle-ci respecte cette équation, le point est dans le quart unitaire du cercle et est donc un *hit*, sinon il est en dehors et c'est donc un *miss*. Avec un nombre d'itérations de cette condition sur un nombre de point suffisamment grand, le nombre de *hits* correspond à un quart de la surface du cercle unitaire d'où la valeur de pi.

I.1) Version séquentielle

La version séquentielle était fournie dans le sujet et nous avons juste ajouté la mesure du temps du calcul. En moyenne, pour 10 000 000 itérations le temps moyen est de 1,47392 secondes pour une valeur de pi d'environ 3,14155. Nous allons par la suite comparer ce temps avec la version parallèle du calcul.

On obtenait le résultat suivant dans le terminal :

```
audrey@audrey-Ubuntu:~/Ecole/3A/CS_PC/Projet$ python3 estimation_pi_seq.py
Valeur estimée Pi par la méthode Mono-Processus :  3.14254
Temps : 1.58526
```

I.2) Version parallèle

Dans la version parallèle, au lieu de faire les calculs des *hit* et *miss* par un seul processus, on va partager ce calcul par 4 processus. En effet, chaque processus va calculer 2 500 000 points, estimer la valeur de pi et partager cette valeur aux autres processus à l'aide d'une variable partagée appelée *share* dans le programme. À la fin de chaque processus, la valeur contenue dans *share* est la valeur estimée de pi. De part son calcul en parallèle, le temps de calcul est réduit, en effet, en moyenne le temps de calcul est de 0,59625 secondes pour une valeur de pi équivalente à celle de la version parallèle, 3,14150.

On obtenait le résultat suivant dans le terminal :

```
audrey@audrey-Ubuntu:~/Ecole/3A/CS_PC/Projet$ python3 estimation_pi_par.py
Valeur estimée Pi par la méthode Multi-Processus 1 :  3.14138
Temps : 0.49408
```

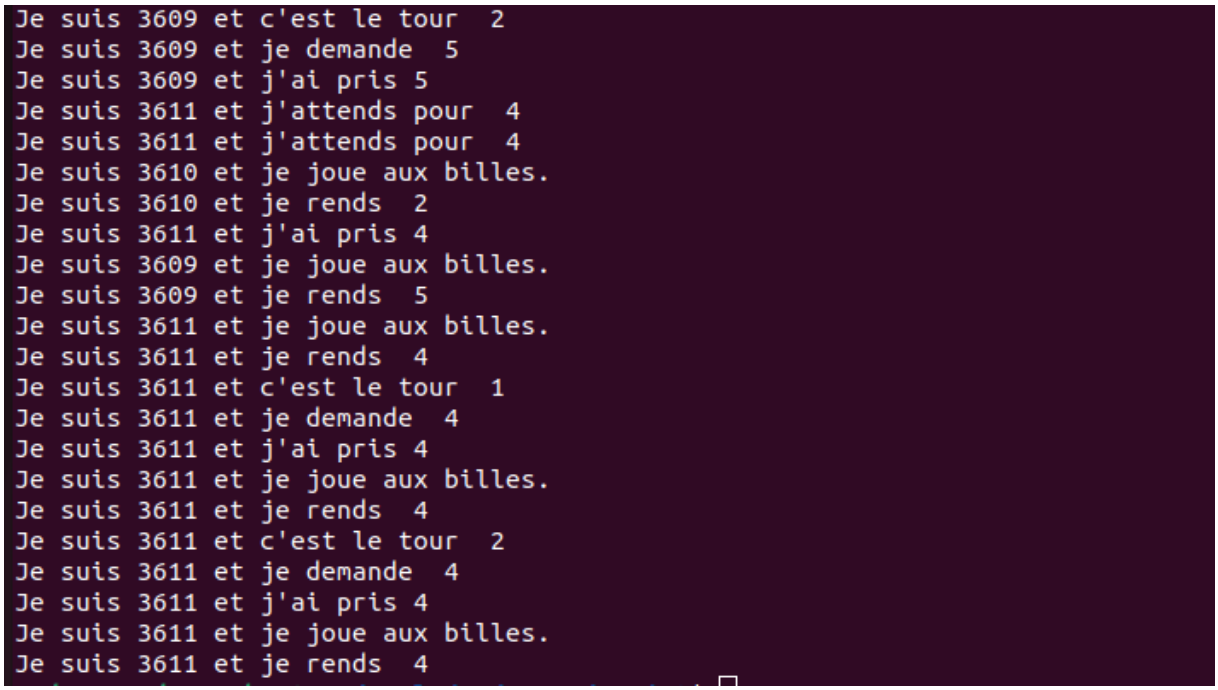
Ainsi, on peut noter qu'avec l'utilisation de plusieurs processus pour le calcul, celui-ci est beaucoup plus rapide dans notre cas environ 3 fois plus rapide.

II) Gestion de Billes

Dans cet exercice, on simule un gestionnaire de billes. En effet, le nombre de billes est limité, dans le programme ce nombre est de 9, et différents processus, ici 3, vont récupérer un certain nombre de billes, les utiliser puis les rendre. Un processus, dans le programme *surveillant*, va vérifier que le nombre de billes maximales n'est jamais dépassé. Ci cela arrive, il affiche un message d'arrêt et arrête tous les processus en cours d'exécution.
Comme le nombre bille voulu par un processus n'est pas forcément accessible tout de suite lorsqu'un processus effectue l'appel à la fonction *demander,* celui-ci est mis en attente à l'aide d'un sémaphore, via l'instruction *sem.acquire(),* tant que le nombre de bille voulu est supérieur au nombre disponible, boucle *while* dans le programme. À chaque fois qu'un processus rend des billes via la fonction *rendre*, celui-ci libère le jeton du sémaphore, soit l'instruction *sem.release().* Ainsi, le processus qui était en attente est libéré revérifie que le nombre de billes qu'il souhaite est disponible grâce à la boucle *while.* Si c'est le cas, il récupère des billes, sinon il est mis en attente par le sémaphore et attend qu'un processus libère un jeton du sémphore pour vérifier de nouveau la condition.

On obtient ainsi ce type de résultat, lorsque les processus demandent 3 fois des billes, où l'on retrouve bien le fonctionnement décrit précédemment :

```
audrey@audrey-Ubuntu:~/Ecole/3A/CS_PC/Projet$ python3 gestionnaire_billes.py
Je suis 3609 et c'est le tour  0
Je suis 3609 et je demande  5
Je suis 3609 et j'ai pris 5
Je suis 3610 et c'est le tour  0
Je suis 3610 et je demande  2
Je suis 3610 et j'ai pris 2
Je suis 3611 et c'est le tour  0
Je suis 3611 et je demande  4
Je suis 3611 et j'attends pour  4
Je suis 3611 et j'attends pour  4
Je suis 3610 et je joue aux billes.
Je suis 3610 et je rends  2
Je suis 3609 et je joue aux billes.
Je suis 3609 et je rends  5
Je suis 3610 et c'est le tour  1
Je suis 3610 et je demande  2
Je suis 3609 et c'est le tour  1
Je suis 3610 et j'ai pris 2
Je suis 3609 et je demande  5
Je suis 3609 et j'ai pris 5
Je suis 3611 et j'attends pour  4
Je suis 3611 et j'attends pour  4
Je suis 3610 et je joue aux billes.
Je suis 3609 et je joue aux billes.
Je suis 3610 et je rends  2
Je suis 3610 et c'est le tour  2
Je suis 3609 et je rends  5
Je suis 3610 et je demande  2
Je suis 3610 et j'ai pris 2
```



III) Un système multi-tâche de simulation d'un restaurant

Dans cet exercice, nous avons simulé l'activité d'un restaurant. Nous avons des clients qui vont commander des plats, qui sont représentés par les lettres de l'alphabet. Ces commandes vont ensuite est récupérer par les serveurs qui vont les transmettre au cuisiner. Les cuisiniers vont les préparer, ce qui sera simuler par l'utilisation de la fonction *sleep* du module *time* pour simuler un délai. Ensuite, ils donnent les plats aux serveurs qui les servent. Le major d'homme affiche quelles commandes sont en cours de préparation, lesquelles sont en attente et combien.

Les 5 serveurs sont simulés par 5 processus, de même les 3 cuisiniers sont simulés par 3 processus. Les clients ne sont simulés que par un seul process et de même pour le major d'homme. Le fonctionnement du code est le suivant :

- Le processus client crée une commande et la dépose dans une liste de valeur partagée de taille 50. La commande est écrite sous le format : 000_A, soit le numéro du client et le plat choisi. Pour effectuer cette écriture dans la variable partagée, on vérifie que le processus est le seul à effectuer une action sur ces variables à l'aide d'un sémaphore. Cela est le cas pour tous les autres processus qui travaillerons avec la variable partagée excepté le major d'homme.

- Le processus serveur lit le contenu des variables partagées à son tour et regarde qu'elles sont les commandes non prises encore en compte donc de la forme : numClient_lettre. Dès qu'il lit une commande de ce type, il la « donne » via les variables partagées aux cuisiniers sous le format : numClient_lettre_numServeur. Cette action est simulée par un temps de 2 secondes avec la fonction *sleep.*

- Le processus cuisinier va à son tour regarder le tableau de valeurs partagées, et récupérer celles qui sont de la forme numClient_lettre_numServeur. Il va simuler la préparation de la commande par un temps de délai, ajouter son numéro, la commande sera de la forme numClient_lettre_numServeur_numCuisinier et la « rendre » aux serveurs via le tableau de valeurs partagées.

- Le processus serveur va servir la commande, simuler avec un délai, et le signifier en ajoutant S au nom de la commande qui sera donc du format suivant numClient_lettre_numServeur_numCuisinier_S. Cette action est simulée par un délai de 2 secondes.

- Le processus major d'homme va lire ce tableau de valeurs partagées et à l'aide des noms des différentes commandes afficher les différentes commandes en cours pour obtenir cet affichage :

```
Le cuistot 0 prepare un I pour le client 008 suivie par le serveur 1.
Le cuistot 1 prepare un W pour le client 010 suivie par le serveur 4.

Le cuistot 2 prepare un X pour le client 007 suivie par le serveur 4.

Les commandes clients en attente : [('Client :011', 'Commande :J')] .

Nombres de commandes en attente: 0 .

Le serveur 4 a servie le client 010.
```

  Quand ce processus a fini un affichage, il notifie les autres processus pour les autoriser à travailler et donc modifier le tableau de valeurs partagées. Cet affichage est obtenu avec la commande : *python3 simu_resto.py 2> out.*

En effet d'autres informations sont affichées dans le terminal pour permettre de mieux suivre le déroulement de la commande avec la commande : *python3 simu_resto.py > out.* Cette commande montre le résultat suivant :

```
audrey@audrey-Ubuntu:~/Ecole/3A/CS_PC/Projet$ python3 simu_resto.py > out
000_B
in 000_B
('prise000_B_0', 0)
('cuistot000_B_0_0', 0)
('servie000_B_0_0S', 3)
001_I
in 001_I
('prise001_I_3', 3)
('cuistot001_I_3_0', 0)
('servie001_I_3_0S', 4)
002_T
in 002_T
('prise002_T_1', 1)
('cuistot002_T_1_1', 1)
('servie002_T_1_1S', 1)
003_K
in 003_K
('prise003_K_3', 3)
('cuistot003_K_3_0', 0)
('servie003_K_3_0S', 3)
004_R
in 004_R
('prise004_R_2', 2)
('cuistot004_R_2_0', 0)
005_K
('servie004_R_2_0S', 2)
006_K
in 006_K
('prise006_K_4', 4)
('cuistot006_K_4_1', 1)
```

Cela montre un des problèmes du programme. En effet, parfois, certaines commandes ne sont pas prises en compte comme ici la commande 005_K. Un autre problème du programme est l'affichage du nombre de commande en attente qui de part la structure de celui-ci ne se fait pas correctement.

IV) Course hippique

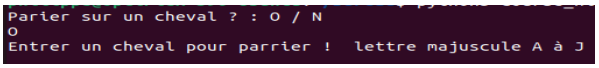
Dans cet exercice nous devions rajouter des fonctionnalités au code de la course de chevaux. Nous devions commencer par ajouter un arbitre, qui affiche le début du classement des chevaux. Pour cela, nous avons créé un tableau partagé qui contient les positions en temps réel de chaque cheval (tab). Ensuite, nous avons mis à jour la liste contenant le classement des chevaux ainsi qu'en fonction de leur position (*up_date* : ligne 108 et tri : ligne 119). Enfin nous avons affiché les résultats en bas de l'écran (disp_*score* : ligne 126).
Nous pouvions également ajouter la possibilité de parier, il faut donc demander au joueur s'il veut parier et si oui sur quel cheval et afficher quels chevaux sont disponibles pour parier (cela dépend du nombre de chevaux dans la course). On utilise des *inputs* avec des vérifications des valeurs saisies puis on affiche les résultats du pari.

Finalement nous pouvions changer l'aspect physique des chevaux, nous avons choisit :

```
      (A>                 (A>             (A>
/|-----|     puis    /|-----|   puis   /|-----| ,   avec une lettre par cheval.

 | | | |               \\\\             ////
```

Demande de pari :



Course si le joueur a parié :

Si le joueur n'a pas voulu parier :

V) Le jeu de la vie

Nous avons codé une classe appelée *game_of_life*, elle prend en entrée le nombre de lignes et de colonnes souhaitées pour le tableau.
Une fonction d'initialisation crée un tableau partagé contenant assez de valeur pour compléter le tableau donné en entrée de la classe. Nous lançons ensuite le traitement de l'environnent, pour récupérer les états des cellules voisines. Ensuite, nous comptons le nombre de cellules vivantes ou mortes autour pour savoir si la cellule va vivre ou mourir. Dans la partie *main* de la classe, on initialise le tableau, nous répartissons le tableau entre deux processus et nous les démarrons.

Dans la boucle *while* on appel la méthode display pour afficher le tableau avec l'état des cellules. Cependant pour mettre fin au programme il faut faire un *contrôle+C* car nous ne savions pas quand arrêter la simulation.

Voici un aperçu d'un tableau en 15x15 :