Rapport de projet de programmation concurrente

Rémi De Pretto

June 23, 2024

Contents

1 Course hippique

Pour la course hippique, le code des chevaux était déjà fourni ainsi que l'affichage dans le terminal. Ici, il fallait donc adapter le code pour pouvoir mettre en place un arbitre qui annonce à chaque instant quel cheval est premier et quel cheval est dernier. Puis à la fin de la course (lorsqu'un cheval atteint la colonne 50), l'arbitre annonce le(s) gagnant(s).

Pour cela, il suffit de mettre en place une liste partagée de la librairie multiprocessing. Et avec quelques fonctions simples déterminant le ou les maximum et minimum, l'arbitre est capable d'afficher les positions désirées.

La liste partagée est fournie avec un verrou mais pour mieux comprendre leurs fonctionnements, j'ai aussi ajouté les miens par dessus.

2 Client-serveur de calculs

2.1 Introduction

Pour les client-serveur de calculs, des codes représentatifs de la situation ont été donnés mais en séquentiel. Il fallait donc dans un premier temps, réaliser une implémentation en parallèle puis ajouter plusieurs serveurs et clients.

Pour cela la fonction "calculette" joue le rôle de serveur et la fonction client, celui d'un client.

2.2 Serveur

Le rôle des serveurs est simple : ils attendent qu'un client demandent à effectuer une opération. Puis calculent le résultat qu'ils renvoient au client demandeur. Pour gérer la communication et l'attente des serveurs, il faut des queues, qui sont fournies par multiprocessing. Un serveur qui cherche à obtenir des opérations va lire la queue "op_to_do". Si elle est vide, il passe en attente passive. C'est le système d'exploitation qui le réveillera lors de l'arrivée de données et l'absence de serveur devant lui. Pour les résultats, il lui suffira de les déposer dans la queue du client demandeur qui a l'identifiant "c_id". Alors cette queue se trouve dans "resultats_queues" à l'indice "c_id".

2.3 Client

Le rôle des clients est de demander des opérations à évaluer. On modélise le comportement d'un client par effectuer une séquence d'action en boucle. Il prend deux chiffres et une opération aléatoirement dans une liste définie. Les dépose dans la queue "op_to_do_queue" puis se met en attente d'une réponse. Comme pour les serveurs, ceci est géré par la méthode pour obtenir un élément sur la queue "results_queue". Cette queue est unique car le client ne va pas espionner ce que les autres font.

2.4 Affichage

Pour pouvoir suivre les échanges, les clients affichent ce qu'ils ont demandé et la réponse, le tout une fois que la réponse est obtenue. Puis les serveurs affichent la requête reçue et le résultat. Il a donc fallu protéger la ressource "stdout" qui est l'affichage dans la console.

3 K-means

3.1 Introduction

Pour le Clustering K-means, j'ai implémenter cette méthode pour un nuage de points de $[0, 100]^2$. Un pseudo code était fourni pour nous aider à l'écriture de notre fonction. Il était également conseillé de commencer par l'implémentation séquentielle avant celle en parallèle.

3.2 Initialisation

Pour l'initialisation, il suffit de générer N points aléatoires sur la partie du plan considérée (c'est le rôle de la fonction "init_Lst"). Puis il fallait choisir les k barycentres initiaux (rôle de "centroid_init"). Soit nous pouvions les choisir aléatoirement, soit nous pouvions essayer de choisir les points les plus éloignés les un des autres avec seulement le premier point choisi aléatoirement.

J'ai choisi la deuxième méthode. Pour réaliser cela, j'ai d'abord pensé à considérer, pour chaque point, l'espérance et la variance de la distance euclidienne entre les i premiers barycentres choisis et le point. Cependant ces critères ne suffisent pas ! Voici les possibilités :

1. Nous voulons maximiser l'espérance puis en cas d'égalité ou de valeur proche (car ce sont des float) minimiser la variance. En d'autres termes, nous voulons les distances les plus grandes possibles en moyenne et ces distances doivent être assez homogène.

   Cependant les points vont tous se trouver uniquement dans les quatres coins de l'ensemble. En effet, ils sont loin les uns des autres en moyenne mais deux points ne sont qu'espacé suffisamment s'ils n'appartiennent pas au même coin. Ceci est embêtant si $k > 5$ car nous aurions alors au moins deux points distincts très proches dans un coin. Ce n'est pas une bonne solution.

2. À l'inverse, minimisons d'abord la variance puis en cas de variance proche, maximisons l'espérance. C'est-à-dire choisir des points de distances homogènes quitte à avoir des espérances plus faibles.

   Cette méthode est bien plus efficace pour éparpiller les barycentres initiaux mais ne donne pas toujours la solution optimale. Il arrive que le plus petit ensemble englobant (l'enveloppe convexe) les barycentres initiaux ne soit que les deux tiers de $[0, 100]^2$. Cependant c'est la méthode que j'ai conservée puisqu'elle diffuse bien les barycentres initiaux.

3.3 Boucle

Dans la boucle, il suffisait de convertir le pseudo code en python. J'aimerais juste revenir sur la condition d'arrêt que je ne trouve pas satisfaisante. Si l'un des barycentres ne bouge presque plus, cela n'est a fortiori pas le signe que notre clustering est fini. Exposons un contre-exemple, l'initialisation des barycentres donne des barycentres qui sont bien répartis dans l'ensemble entier. Mais lorsque nous avons k assez grand comparé à N (exemple k = 10, N = 50), il arrive souvent qu'un barycentre ne bouge pas et provoque l'arrêt de la boucle. Donc j'ai préféré prendre le maximum plutôt que le minimum. Ainsi si le plus grand déplacement de barycentre est inférieur à epsilon, tous le sont. Donc la méthode a convergé vers une répartition. Puis nous affichons à chaque tour de boucle la répartition.

Notons que pour des soucis d'efficacité, j'ai essayé le plus possible de travailler avec des distances euclidiennes au carré pour éviter de calculer une racine qui est coûteuse.

4 Schéma lecteur / rédacteur

4.1 Introduction

Pour le schéma lecteur / rédacteur, le but est de mettre en place une structure qui permet protéger une ressource partagée. Nous voulons ici donner la priorité aux rédacteurs. Le déroulé des opérations à réaliser pour commencer ou finir une rédaction ou une lecture nous est fourni. Il fallait donc simplement l'implémenter en python.

4.2 Structure

Pour cela, nous allons utiliser trois verrous et deux variables partagées. L'affichage ajoute à cela encore un verrou.

Voici le fonctionnement des verrous :

1. "mutex" sert à ce que les étapes d'initialisation et de fin de lecture ou de rédaction soient réalisées d'une traite par un processus sans qu'un autre modifie l'état des variables internes. C'est un point essentiel ! Seul le premier rédacteur prend un jeton pour la priorité aux rédacteurs ("prio_redacteur"). Sinon les rédacteurs suivant feraient la queue avec les lecteurs et il n'y aurait plus de priorité. Puis seul le dernier rédacteur relâche le jeton. Sinon les lecteurs accèderaient à la ressource en même temps que le deuxième rédacteur. Puis une situation similaire est aussi évitée grâce au "mutex" : cela concerne les lecteurs et le verrou d'exclusion entre lecteurs et rédacteurs ("redacteur_lecteur_ex").
2. "redacteur_lecteur_ex" permet d'exclure l'accès simultané à deux rédacteurs ou entre rédacteurs et lecteurs.
3. "prio_redacteur" est le verrou qui empêche tout lecteur d'accéder à la ressource partagée s'il y a déjà au moins un rédacteur qui veut modifier cette dernière.
4. "stdout_lock" sert à ce que l'affiche d'un processus soit effectué entièrement avant qu'un autre commence son affichage.

Et celui des variables partagées :

1. "nbr_process_redacteur" est le nombre de rédacteurs qui veulent apporter une modification.
2. "nbr_process_lecteur" est le nombre de lecteurs voulant accéder à la ressource.

5 Game of life

5.1 Introduction

Le but de cette partie est d'implémenter le célèbre jeu de la vie en parallèle. Pour connaître les règles, se référer au sujet du projet ou sur internet. Pour paralléliser le jeu de la vie, il faut remarquer que chaque cellule (ou case) dépend de ses huit voisines de la génération (ou itération ou tick) précédente. Donc il suffit d'avoir deux versions de la grille, que l'on nommera "ville", l'actuelle et la future que nous cherchons à calculer. Ainsi chaque cellule future est indépendante de ses voisines mais dépend de ses voisines de la génération précédente. Donc nous pourrions lancer un processus sur chaque case pour calculer son évolution. Ceci, en faisant attention à synchroniser les générations : une fois que tous les états futurs sont calculés, ils deviennent les états présents.

5.2 Proposition de parallélisation

Cependant en réalité, nous n'avons pas un nombre conséquent de coeurs sur les processeurs pour faire ce parallélisme physiquement en même temps et non pas un partage du temps de calcul. Dès lors, j'ai opté pour diviser ma ville carrée en quatre quartiers carrés. Chacun attribué à un processus qui pourra réellement tourné en parallèle des autres.

Il a donc fallu synchroniser ces quatre processus et l'affichage. Pour cela, j'ai utilisé six sémaphores.

1. Chaque quartier dépose son jeton dans "afficher_prochain_tick" pour libérer l'affichage qui en a besoin de quatre et indiquer que la prochaine génération est prête.
2. Puis lorsque l'affichage à copier la génération actuelle, il rend ses quatre jetons dans "calculer_prochain_tick" pour recommencer le calcul de la génération suivante.
3. Pour finir, quatre sémaphores "push_ville_evo" permettent de synchroniser la mise à jour de la ville par les quartiers. En d'autres termes, une fois que les quartiers ont tous fini de calculer, ils peuvent à ce moment-là uniquement transférer la prochaine génération sur l'actuelle.

Remarques :

1. La ville est simplement une liste partagée séparée en plusieurs tronçons. Chaque ligne d'un quartier est un tronçon, donc une liste partagée.
2. Pour plus de détails sur les calculs réaliser voir le code et sa documentation. Notamment "affiche_ville_aide()".