Randriampierenana Aaron
Benferchouli Sam

PPC : Circle of life

La simulation d'écosystème est un excellent problème informatique nécessitant une réflexion sur la concurrence et le parallélisme de processus. Le but de ce projet est surtout architectural, centré sur la synchronisation de plusieurs processus indépendants.

1. Conception et choix

1.1 Librairies

Pour la conception de ce projet, nous avons choisi d'utiliser multiprocessing plutôt que threading car le threading ne permet pas un total parallélisme dans le fonctionnement des différents processus, en effet, dans le cadre de notre projet, nous avons besoin que chaque entité (proie ou prédateur) fonctionne indépendamment des autres pour être parfaitement autonome, chaque entité est un processus distinct avec un pid propre à lui. C'est pourquoi nous nous sommes dirigés vers le module **multiprocessing.**
Pour l'interface graphique, nous nous sommes dirigés vers le module **tkinter** nous offrant une plus grande liberté de fonctionnement que matplotlib, nous permettant notamment d'implémenter des boutons ou de changer certains paramètres de départ tels que les seuils d'énergie ou le nombre d'entités de départ.

1.2 Données partagées

Pour assurer le partage d'informations et de mémoire entre processus nous avons utilisé **multiprocessing.Manager()** permettant le partage de données entre processus n'ayant aucun lien de parenté. Nous avons notamment utilisé :

- **Manager.list()** une liste de taille 3 contenant le nombre de proies, le nombre de prédateurs et le niveau d'herbe
- **Manager.dict()** un dictionnaire contenant tous les processus des entités actives de la forme : { id_process : (nature, energie, etat)}
- **Manager.Queue()** une file contenant les états de l'environnement et les messages de reproduction et de mort des processus.

2. Architecture et protocoles

2.1 Architecture globale

La simulation repose sur 4 scripts interconnectés :

- **Le display (display.py):** c'est le script qui lance la simulation et qui implémente la représentation graphique.
- **L'environnement (environment.py):** le processus qui gère la croissance de l'herbe, les aléas climatiques (sécheresse)

- **Les entités (pred.py et prey.py):** les processus autonomes des proies et des prédateurs

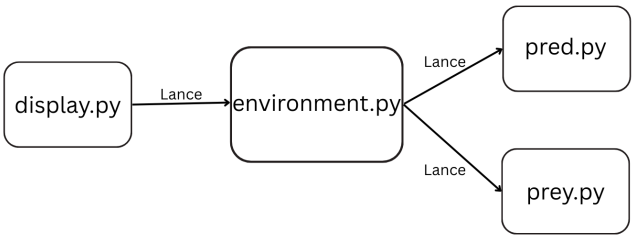


Schéma de fonctionnement de la simulation

2.2 Communication inter-process

- **Communication réseau :** la communication inter-process se fait via une **socket :** le processus environnement sert de serveur, il va lancer le serveur sur localhost:8080 et le mettre en écoute. Ainsi, lorsqu'une entité veut se reproduire elle envoie à ce serveur le message binaire b"NOUVELLE_PROIE" pour les proies et b"NOUVEAU_PREDATEUR" pour les prédateurs. L'environnement va ensuite recevoir ce message et lancer de nouveaux processus en conséquence.
- **Mémoire partagée et synchronisation :** toutes les écritures dans le dictionnaire des entités ou dans la liste de mémoire sont protégées par un **lock** empêchant ainsi plusieurs processus de modifier la mémoire en même temps évitant ainsi les problèmes de synchronisation des données. Cela évite par exemple qu'une proie qui vient de mourir de faim soit mangée par un prédateur.
- **Queue :** Lorsqu'un processus fait une action (un prédateur mange une proie ou une entité se reproduit) il va envoyer un message dans msg_queue qui va ensuite être envoyée au display pour afficher les logs.

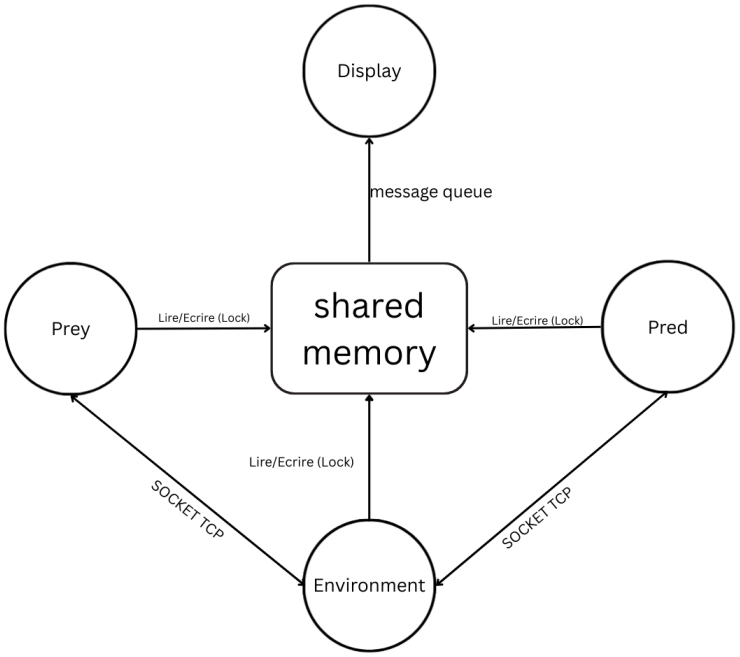
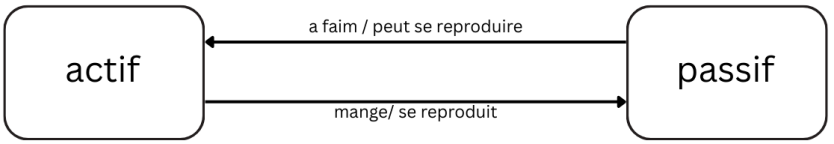


Schéma de communication inter-process

3. Algorithmes

3.1 Etat des entités

Pour éviter que les prédateurs ne mangent toutes les proies d'un coup et pour créer un cycle de vie : faim -> action -> repos, nous avons décidé d'associer à chaque entité un état passif ou actif



Automate des états des entités

L'entité est d'abord passive puis devient active si elle a faim ou si elle peut se reproduire et devient passive après avoir réalisé une action

3.2 Mécanisme de chasse des prédateurs

Pour manger une proie, le prédateur :

- Utilise un **lock**
- Parcourt le dictionnaire des entités jusqu'à tomber sur une proie avec une énergie strictement positive
- Si il en trouve une, il met son énergie à 0 et gagne gain_repas d'énergie
- Envoie un message dans msg_queue pour afficher son action dans les logs de la simulation

L'utilisation du lock permet de garantir la synchronisation des données partagées : sans lock si un prédateur 1 voit la proie A vivante et le prédateur 2 voit la proie A vivante au même moment, si le prédateur 1 mange la proie avant le prédateur 2, ce dernier va quand même manger la proie qui est pourtant morte ce qui va causer une erreur.

3.3 Gestion de l'environnement

Deux boucles sont gérées par l'environnement :

- while true … qte_herbe+=1 : l'herbe croît à chaque itération de la boucle
- **threading.timer** : afin de gérer l'occurrence des changements climatiques, on utilise un threading.timer. Le sujet nous laissait le choix entre l'utilisation de threading.timer et signal.setitimer, nous avons opté pour threading.timer car l'autre module n'est disponible que sous UNIX donc risque de ne pas fonctionner sous windows.

3.4 Fonctionnement des processus de prédateurs

Le fichier pred.py est composé de 3 fonctions :

- **manger_proie** : la fonction qui implémente le mécanisme des prédateurs décrit au point 3.2 et qui renvoie énergie finale du prédateur et un booléen mange qui indique si le prédateur a mangé ou non à la fin de l'exécution.
- **reproduction_predateur** : cette fonction prend en argument l'énergie du prédateur, si cette énergie est suffisante pour se reproduire, elle divise l'énergie par un facteur facteur_reproduction et renvoie (True,énergie) sinon elle renvoie (False,énergie)
- **pred_process** : cette fonction lance le processus du prédateur : elle note l'id du nouveau process, à l'aide d'un lock, elle l'ajoute dans le dictionnaire des entités et incrémente le compteur de prédateur dans la liste mémoire. Elle note ensuite son énergie et son état, puis elle implémente le cycle de vie du prédateur : elle implémente l'automate des états du prédateur.

3.5 Fonctionnement des processus de proies

Le fichier prey.py est composé de 3 fonctions :

- **manger_herbe** : cette fonction lit, à l'aide d'un lock, la quantité d'herbe disponible et si elle est suffisante, elle enlève qte_herbe_mangee à cette valeur puis ajoute gain_repas à l'énergie de la proie. La fonction renvoie (True,énergie) si la proie a mangé et (False,énergie) sinon.

- **reproduction_proie** : cette fonction prend en argument l'énergie du proie, si cette énergie est suffisante pour se reproduire, elle divise l'énergie par un facteur facteur_reproduction et renvoie (True,énergie) sinon elle renvoie (False,énergie)
- **prey_process** : cette fonction lance le processus de la proie : elle note l'id du nouveau process, à l'aide d'un lock, elle l'ajoute dans le dictionnaire des entités et incrémente le compteur de proies dans la liste mémoire. Elle note ensuite son énergie et son état, puis elle implémente le cycle de vie du proie : l'automate des états de la proie.

3.6 Désynchronisation des entités

Afin d'éviter que toutes les entités ne soient synchronisées, on ajoute au début de chaque processus prey ou pred un time.sleep(random.uniform(0,0.5)) évitant ainsi que toutes les proies ne cherchent à accéder à la mémoire en même temps.

4. Implémentation

4.1 Organisation du code

Tous les scripts ne se trouvant pas dans le même dossier, il est nécessaire d'importer les modules sys et os pour exécuter la commande sys.path.append(os.path…) pour ajouter les répertoires nécessaires contenant les différents fichiers.

Le fichier **configs.py** est le fichier contenant toutes les variables communes à tous les scripts, cette implémentation permet de modifier les paramètres dans **display.py** pour permettre à l'utilisateur de changer certains paramètres à sa guise.

4.2 Gestion des erreurs

La simulation est basée sur du multiprocessing, il est donc crucial de gérer correctement les erreurs. En effet, si un processus échoue et que la gestion d'erreur n'est pas optimale, ce processus peut bloquer l'ensemble des processus actifs et donc faire planter la simulation.

Premièrement, il faut gérer la communication asynchrone. L'environnement doit faire pousser l'herbe en continu sans attendre qu'un processus ne se connecte au serveur. Cependant, comme socket.accept() bloque l'exécution par défaut, on a ajouté la commande **setblocking(False)** pour passer en mode non-bloquant et on a utilisé un **try except BlockingIOError** pour ne pas interrompre la pousse de l'herbe.

Dans un second temps, il faut gérer l'affichage des logs dans display.py. En effet, display lit a chaque tour de boucle la valeur de msg_queue sans faire bugger l'interface. Ainsi, si la file est vide Queue.get() bloque le programme, on utilise donc queue.get_nowait pour ne pas bloquer la boucle et on utilise un **try except queue.Empty()** pour arrêter la boucle si la file est vide.

Finalement, il faut aussi gérer les erreurs liées à la partie interactive de la simulation. Le display permet à l'utilisateur de changer des paramètres en changeant des valeurs, cependant si il entre une valeur du mauvais type, une erreur va survenir et le display va

freeze. On utilise donc un **try except ValueError** pour afficher dans les logs le message d'erreur et continuer avec les valeurs précédentes.

4.3 Implémentation de l'interface graphique

Le fichier display.py crée l'interface graphique de la simulation mais agit surtout comme un main process. Pour l'implémenter, il faut d'abord réussir à gérer le partage de mémoire. En effet, display est le processus qui initialise le **multiprocessing.Manager()**, il crée toutes les structures de données qui vont être partagées entre les processus, il lance le processus d'environnement et envoie lui envoie les messages correspondant aux entités initiales. Cela garantit que l'interface graphique lit les bonnes données assurant ainsi une cohérence entre les données affichées et les données réelles.

Deuxièmement, il faut aussi réussir à faire cohabiter l'interface graphique avec les processus qui tournent en arrière-plan. En effet, Tkinter, implémente sa propre boucle **mainloop()** l'utilisation de while True pour lire les données bloquerait l'affichage de l'interface. On a donc utilisé la méthode d'interrogation périodique via une fonction **update_loop** et la méthode **after()** permettant de mettre à jour périodiquement l'interface (ici toute les secondes).

```
self.fenetre.after(1000, self.update_loop)
```

5. Exécution du code

5.1 Démarrage du code

Afin d'exécuter le code, il suffit de lancer le fichier display.py qui va afficher une interface utilisateur permettant d'afficher et de modifier des données ainsi que de deux boutons "Démarrer" et "Réinitialiser".
Lors du lancement, l'écosystème est initialisé : un Process Manager est créé et se lance en arrière-plan. Il est responsable de la mémoire partagée et notamment de la liste des compteurs et du dictionnaire des entités.
De plus, un processus exécutant la fonction **env_process()** provenant du fichier environment.py sera créé, permettant ainsi d'initialiser l'environnement et de gérer l'ensemble des ressources. Cela gère la régénération de l'herbe via la fonction **croissance_herbe()** ainsi que la création de processus de type prédateur/proie en parallèle et ce grâce à l'usage de socket.

5.2 Interaction utilisateur via une interface graphique

Comme spécifié au-dessus, l'interface permet la modification des paramètres de la simulation tels que l'énergie des proies et des prédateurs en temps réel.
On récupère les données entrées sur l'interface graphique et on les modifie dans notre fichier configs grâce à : **self.{donnée_choisie}.insert(0, str(configs.{donnée_choisie}))**

Le bouton de Démarrage permet le lancement de la simulation grâce à la fonction **start_simulation().** Enfin, le bouton de Réinitialisation permet de supprimer tous les processus créés, la réinitialisation des données modifiées et le nettoyage de l'interface graphique. Il agit de la même manière que lors de la fermeture du programme décrite ci-dessous.

5.3 Fermeture du programme

Lors de la fermeture du programme, on utilise **reset_simulation()** précédé par le mot clé "finally" permettant son exécution peu importe la manière dont le code est fermé (Ctrl + C etc).

La fonction reset_simulation() quant à elle permet de terminer le processus environment qui organise toute la structure de la simulation (de force), de shutdown le process manager ainsi que de réinitialiser l'intégralité de la mémoire partagée.

6. Phase de tests et améliorations

6.1 Problèmes rencontrés et solutionnement

Durant le développement du projet, nous avons rencontré quelques problèmes comme :

Le problème de la duplication de la nourriture : Lors des premiers tests, deux prédateurs pouvaient tenter de manger la même proie en même temps. Même si le dictionnaire est partagé, le délai entre la lecture de l'énergie et son passage à zéro permettait une double consommation.

Comme Solution on a opté pour l'encapsulation de la vérification et de l'action dans un **multiprocessing.Lock().**

Le problème de la forte consommation de ressources CPU : Faire tourner des centaines de processus avec des boucles while True sans interruption consommait une quantité beaucoup trop importante de ressources inutilement.

Solution : Introduction de time.sleep() stratégiques et utilisation de la méthode after() de Tkinter pour cadencer l'affichage.

Le problème des Deadlocks : Des processus restaient parfois "suspendus" en attendant un lock déjà possédé par un processus parent.

Pour résoudre cela, on a réduit la portée des locks au strict minimum (juste le temps de la modification mémoire).

6.2 Visualisation de la simulation

Dans le cadre du projet on a pensé à une implémentation plus graphique de la simulation. Avec une interface permettant la visualisation de notre environnement ainsi que des différents process représentant les animaux sous forme de points rouges et blancs.

Pour cela on a imaginé une zone représentant l'environnement qui serait découpé en une grille d'un certain nombre de cases (configurable dans le fichier config). sur laquelle les animaux pourraient se déplacer de manière à pouvoir chasser/s'enfuir et se reproduire.

Un prototype de cette version est disponible sur le github dans la branche second_version.

6.3 Pistes d'amélioration

Pour améliorer le projet actuel, on a pensé à plusieurs idées comme les suivantes.

Rendre la simulation dynamique avec des coordonnées et une mécanique de déplacement (comme implémenté dans la seconde version avec l'interpréteur graphique).

Implémenter un suivi des lignées pour observer quelle "famille" d'entités survit le plus longtemps grâce à la sélection naturelle des paramètres de départ. Et faire en sorte que plusieurs espèces avec des propriétés différentes vivent dans le même écosystème. Avec des herbivores, carnivores et omnivores. Faire en sorte que certaines espèces en chassent d'autres mais se font aussi chasser par d'autres.

Ajouter des éléments de décors comme des arbres ou des grottes etc. Faire varier l'environnement peut permettre d'ajouter des mécaniques propres à chaque environnement.

7. Prise de recul sur le projet

7.1 La complexité du parallélisme et ses enjeux techniques

- **Le choix du parallélisme réel :** Contrairement au threading, limité en Python qui empêche l'exécution simultanée sur plusieurs cœurs CPU, le module multiprocessing nous a permis d'exploiter pleinement l'architecture multicœur. Chaque entité étant un processus indépendant, elle dispose de son propre espace mémoire, garantissant une autonomie totale et évitant qu'un calcul lourd sur une proie ne ralentisse le processus de l'environnement.
- **La gestion de la mémoire partagée :** L'absence de mémoire partagée native entre processus a constitué le défi majeur. L'utilisation de **multiprocessing.Manager()** a été indispensable. Il nous a offert une structure centralisée (le dictionnaire et la liste) capable de survivre à la mort ou à la création dynamique d'agents.
- **Le coût de la synchronisation :** Nous avons appris que le parallélisme n'est pas "gratuit". La mise en place de Locks est un compromis nécessaire : plus on protège de données, plus on risque de transformer notre exécution parallèle en une exécution séquentielle déguisée. Trouver le juste équilibre entre la sécurité des données (éviter les corruptions) et la performance a été l'un des aspects les plus essentiels du projet.

7.2 Conclusion

En conclusion, ce projet est une démonstration concrète de la puissance des systèmes distribués appliqués à la simulation biologique. Nous avons réussi à concevoir un écosystème robuste où la survie des espèces ne dépend pas d'un algorithme centralisé unique, mais de l'interaction de dizaines de processus autonomes luttant pour des ressources communes.

La simulation valide les concepts fondamentaux de notre cursus :

1. **L'atomicité :** Grâce aux verrous lors de la chasse ou de la pousse de l'herbe.
2. **L'asynchronisme :** Via la gestion des sockets non-bloquantes et des timers.
3. **La résilience :** Par une gestion d'erreurs capable de maintenir la simulation active même en cas de mort imprévue d'un agent.

Ce projet offre une base solide pour des évolutions futures. On pourrait imaginer l'introduction de **gènes** transmis lors de la reproduction (vitesse, espérance de vie) comme précisé dans la partie améliorations plus haut. On pourrait même imaginer que si poussé jusqu'au bout, ce projet pourrait être utile à des fins académiques afin d'observer l'évolution de différentes espèces dans un environnement complexe.

Annexe Utilisation de l'IA dans le projet

Dans ce projet, l'IA a été utilisée comme outil de débogage et d'accélération du codage :

- **Débogage et optimisation :** Aide à la détection d'erreurs et proposition de solution.

- **Résolution de problèmes techniques** : L'IA a contribué à la résolution de certains problèmes techniques tels que le problème des freeze de l'interface graphique lié au conflit entre le **mainloop()** de Tkinter et le while True, l'IA a proposé la solution de l'utilisation de la méthode **after()** dans la fonction **update_loop**. Lorsque l'IA propose une solution, on essaye d'abord de comprendre cette solution puis on l'applique pour vérifier son bon fonctionnement.

- **Rapidité** : Copilot a été utilisé notamment pour sa fonctionnalité de pré remplissage de code lors de l'implémentation de certaines parties de codes telles que l'implémentation des widgets Tkinter : leur placement, leur taille, couleur, …

- **Rédaction** : Correction des fautes d'orthographe et structuration de certaines parties du rapport.