OS201

Fourmis

Agathe BEUCHER

Table des matières

Table des figures

1 Introduction du projet

1.1 Objectif

L'objectif du projet est de paralléliser le calcul de l'évolution d'une colonie de fourmie dans un labyrinthe.

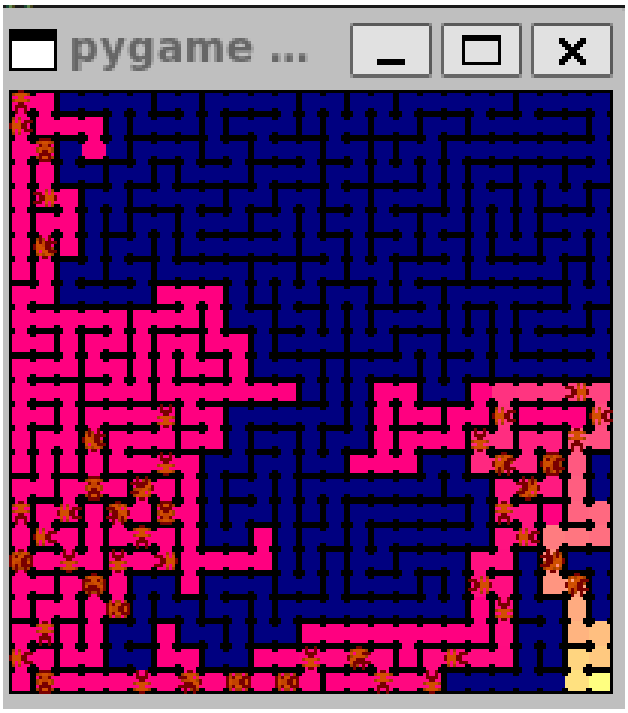

FIGURE 1 – Répartition des fourmis dans les processeurs

Pour cela, nous avons à disposition un template du jeu séquentiel, sous la forme de trois classes :

1. **Classe Colony** : cette classe simule la vie d'une colonie de fourmis, leurs déplacements et leurs interactions dans un environnement structuré, en utilisant des principes d'intelligence collective et d'algorithmes de colonie de fourmis
2. **Classe pheromon** : modélise le comportement et la gestion des phéromones
3. **Classe Maze** : modélise l'affichage et le déplacement dans un labyrinthe.

Le code nous est fourni sous format séquentiel. L'objectif est donc d'optimiser le temps de calcul du mouvement des fourmis en parallélisant d'abord les calculs sur plusieurs processeurs à l'aide d'OpenMPI.

2 Stratégie de parallélisation

2.1 Stratégie globale

Après analyse du programme séquentiel, on remarque que l'on peut diviser ce dernier en deux parties plus ou moins indépendantes : affichage par fenêtre graphique et calcul de l'avancement des fourmis. Il paraît donc judicieux dans un premier temps de séparer ces derniers sur différents processeurs : le processeur principal (ici le processeur 0) s'occupe de l'affichage tandis que les autres processeurs s'occupent des calculs.

2.2 Stratégie de séparation des fourmis

Pour paralléliser, les calculs, deux options s'offrent à nous : dans un premier temps, on choisit de séparer les calculs en paquets réduits de fourmis. Pour cela, on repartit de manière la plus homogène possible les fourmis dans les différents processeurs. Pour cela on répartie de manière homogène toutes les fourmis possibles, les fourmis qui restent sont ajoutées au dernier processeur :

```
# Répartition des fourmis par processus
ants_per_process = nb_ants//(size-1)
if rank==size-1:
        ants_per_process = ants_per_process+(nb_ants%(size-1))
        # répartition des fourmis restantes
```

FIGURE 2 – Répartition des fourmis dans chaque processeur

Une fois que chaque processeur s'est vu assigner un nombre de fourmis à gérer, il crée une colonie locale, tandis que le processeur maître crée une colonie globale avec le nombre total de fourmi :

```
# Initialisation d'une colonie locale dans chaque processus non nul
ants_local = Colony(ants_per_process, pos_nest, max_life,rank)

if rank==0:
    # Initialisation de la colonie globale
    ants_global=Colony(nb_ants, pos_nest, max_life,rank)
```

FIGURE 3 – Colonies locales et globale

2.3 Stratégie de parallélisation des calculs

Ainsi, chaque processeur fait évoluer sa propre colonie indépendamment des autres dans la boucle while :

```
# Sur les processus de calcul
else:
    # Copie de la version globale précédente des phéromones
    old_pheromone=pherom.pheromon.copy()
    # Mise à jour de la colonie locale
    food_counter = ants_local.advance(a_maze, pos_food, pos_nest, pherom, old_pheromone)
    pherom.do_evaporation(pos_food)
    pherom_updates_local=pherom.pheromon
```

FIGURE 4 – Evolution des colonies locales

2.4 Stratégie de gestion des phéromones

La gestion des phéromones dans un environnement distribué pose effectivement un défi :

1. **Calcul Local des Phéromones** : Chaque processus calcule les modifications des phéromones localement pour les fourmis qu'il gère. Cela inclut à la fois l'évaporation et l'augmentation des phéromones sur les chemins empruntés par les fourmis.

2. **Réception des Phéromones** : Chaque processus reçoit la version globale des phéromones qu'ils copie comme "old_pheromones", et utilise pour calculer les nouveaux phéromones, afin de ne pas modifier directement la version globale des phéromones utilisée simultanément par tous les processeurs.
3. **Synchronisation des Phéromones** : À intervalles réguliers, à la fin de chaque itération de simulation, les modifications locales des phéromones doivent être combinées pour obtenir une vue cohérente et à jour des phéromones sur l'ensemble du labyrinthe. Chaque processus envoie ses modifications de phéromones au processus principal, qui les combine et modifie la version globale des phéromones en prenant pour chaque case du labyrinthe, la valeur maximale de phéromones obtenue à travers les différents processeurs.

```
pherom_update_global = [update for update in pherom_update_global if update is not None]
pherom_update_global = np.amax(pherom_update_global, axis=0)
pherom.pheromon = np.maximum(pherom.pheromon, pherom_update_global)
pherom.pheromon = pherom_update_global
```

FIGURE 5 – Evolution des colonies locales

4. **Transmission des Modifications plutôt que des États Complets** : Pour minimiser la quantité de données transférées entre les processus, il est préférable de transmettre uniquement les modifications des phéromones (par exemple, les augmentations dues au passage des fourmis et les diminutions dues à l'évaporation) plutôt que l'état complet des phéromones. Chaque processus applique alors ses modifications à sa copie locale de la carte des phéromones.

2.5 Stratégie de regroupement des données

La méthode responsable de l'affichage des fourmis dans l'interface est '*Colony.display*'. Toutefois, il serait non seulement trop coûteux mais également inutile de transmettre toutes les colonies locales d'un processeur à un autre. Afin d'afficher les fourmis dans le processeur maître, seules certaines données sont nécessaires. :

— *ants.historic_path*
— *ants.directions*
— *ants.age*

2.5.1 Gather

Une première approche de regroupement des données se base sur l'utilisation de "gather" d'OpenMPI. "Gather" et "allreduce" sont des **opérations collectives** qui rassemblent les données de tous les processus d'un groupe et les assemble dans un seul processus, le processus maître ou combine les données de tous les processus en utilisant une opération de réduction (comme la somme, le maximum, le minimum...) et distribue le résultat à tous les processus.. En tant qu'opérations collectives, ils présentent de nombreux avantages :

— **Simplicité** : Un seul appel de fonction est nécessaire pour collecter les données de tous les processus, réduisant ainsi la complexité du code.
— **Performance** : Optimise la manière dont les données sont collectées en parallèle, réduisant le temps de communication global.
— **Fiabilité** : La gestion des erreurs est plus simple car il y a moins de points où la communication peut échouer

. Cette méthode est à priori plus efficace et rassemble directement en une seule liste python toutes les informations provenant de chaque processeur :

```
# Regroupement des résultats reçus en tableau NUMPY
food_counter=comm.allreduce(food_counter, op=MPI.SUM)
pherom_update_global=comm.gather(pherom_updates_local, root=0)
all_historic=comm.gather(ants_local.historic_path,root=0)
all_directions=comm.gather(ants_local.directions,root=0)
all_age=comm.gather(ants_local.age,root=0)
```

FIGURE 6 – Transmission des informations par "gather"

Une fois toutes les données rassemblées en un tableau, on concatène les éléments des tableaux en tableaux numpy et on met à jour la colonie globale avec ces données locales.

2.5.2 Send

Une autre approche serait d'utiliser un processus de communication point à point : "Send" et "recv" sont des opérations de **communication point à point** où un processus envoie explicitement des données à un autre processus, et ce processus reçoit explicitement les données. Ils présentent également leur propres avantages :

- **Flexibilité** : Ils offrent plus de contrôle sur la communication, permettant des schémas de communication complexes et spécifiques.
- **Précision** : Les données peuvent être envoyées et reçues exactement où et quand elles sont nécessaires, ce qui est utile dans des scénarios de communication très spécifiques.

On envoie donc une à une les données sur les colonies locales (*ants.historic_path, ants.age, ants.destination, food_counter,pherom_update_local*) au processeur maître qui les reçoit les une après les autres et les concatène :

```
# Envoie des données nécéssaires à l'affichage à P0
comm.send(local_ants.age, dest=0, tag=0)
comm.send(local_ants.historic_path, dest=0, tag=1)
comm.send(local_ants.directions,dest=0, tag=2)
comm.send(food_counter, dest=0, tag=3)
comm.send(pherom.pheromon,dest=0,tag=4)
```

FIGURE 7 – Transmission des informations par "send"

```
for process in range(1, size):
    # Pour chaque processeur, reception des données d'intêret
    age=comm.recv(source=process, tag=0)
    historic_path=comm.recv(source=process, tag=1)
    directions=comm.recv(source=process, tag=2)
    food_counter += comm.recv(source=process, tag=3)
    pherom_local = comm.recv(source=process, tag=4)
    # Agrégation des résultats reçus en tableau NUMPY
    all_age = np.concatenate((all_age,age),axis=0)
    all_directions = np.concatenate((all_directions,directions), axis=0)
    all_historic_path[(process-1)*ants_per_process:process*ants_per_process,:,:] = historic_path
    pherom.pheromon=np.array(pherom_local)
```

FIGURE 8 – Réception et concaténation des informations par "recv"

3 Analyse

3.1 MEM ou CPU Bound ?

On considère un algorithme qui effectue une série de calculs, notamment pour déterminer les mouvements des fourmis, calculer les phéromones, et mettre à jour les états des fourmis, soit qui effectue des calculs intensifs. Cela suggère une charge de travail potentiellement CPU-bound. L'utilisation de OpenMPI pour distribuer le travail sur plusieurs processeurs vise à surmonter les limitations du CPU en répartissant la charge de calcul. Notre programme est donc potentiellement CPU-bound, en cherchant à maximiser l'utilisation du CPU à travers le parallélisme.

3.2 Parallélisme embarrassant ou pas

Chaque processus peut simuler une portion de la colonie de fourmis de manière relativement indépendante avec un besoin minimal de synchronisation ou de communication entre elles. Toutefois le besoin de synchronisation des phéromones est élevé mais comme la synchronisation représente une fraction important du temps d'exécution total (cf. speedup), cela va contre l'indication d'un parallélisme embarrassant.

3.3 Partie forcément séquentielle

Avant de commencer la simulation, le labyrinthe et ses paramètres ainsi que les phéromones, des colonies ou autre objet doivent être initialisés de manière séquentielle. Par ailleurs, à la fin de chaque itération de calcul de mise à jour des colonies, il est nécessaire de collecter et d'agréger les résultats provenant de différents processeurs de la simulation. Cela inclut la mise à jour de la colonie globale de fourmi, des quantités de phéromones et le comptage de la nourriture collectée.

3.4 Équilibre de charge

Si on divise de manière homogène le nombre de fourmis, on a un bon équilibre de charge. En revanche, si $nb\_ants \% nb\_processeur \neq 0$, alors il restera un nombre fini faible de fourmi non répartie, que l'on choisit de mettre dans le dernier processeur, les charges seront donc un peu moins équilibrées dans ce cas-là.

4 Mesure et analyse des performances

4.1 Speed-up

On cherche à tracer le Speed-up de nos programmes en fonction du nombre de processeurs exécutés en parallèle. Pour cela, on code un script python d'automatisation, qui fait varier le nombre de processeurs et qui lit le terminal pour récupérer une cinquantaine de données de FPS :

```
for nb_proc in nb_procs:
    # Exécute le programme parallèle gather
    cmd_par_gather = f'mpiexec -n {nb_proc} python3 ants_gather.py'
    average_fps_par_gather = execute_command_and_collect_data(cmd_par_gather,max_data_count)
    if average_fps_par_gather:
        times_parallel_gather.append(1 / average_fps_par_gather)

    # Exécute le programme parallèle send
    cmd_par_send = f'mpiexec -n {nb_proc} python3 ants_send.py'
    average_fps_par_send = execute_command_and_collect_data(cmd_par_send,max_data_count)
    if average_fps_par_send:
        times_parallel_send.append(1 / average_fps_par_send)
```

FIGURE 9 – Automatisation de la parallélisation avec subprocess

```
# Exécute le programme séquentiel une seule fois
cmd_seq = f'python3 ants_sequentiel.py'
average_fps_seq = execute_command_and_collect_data(cmd_seq,max_data_count)
if average_fps_seq :
    time_sequential = 1 / average_fps_seq
```

FIGURE 10 – Automatisation du séquentiel avec subprocess

Pour chaque nombre de processeur, le programme calcule alors une moyenne de temps d'exécution en parallèle t_p, puis calcule le temps d'exécution du programme séquentiel :

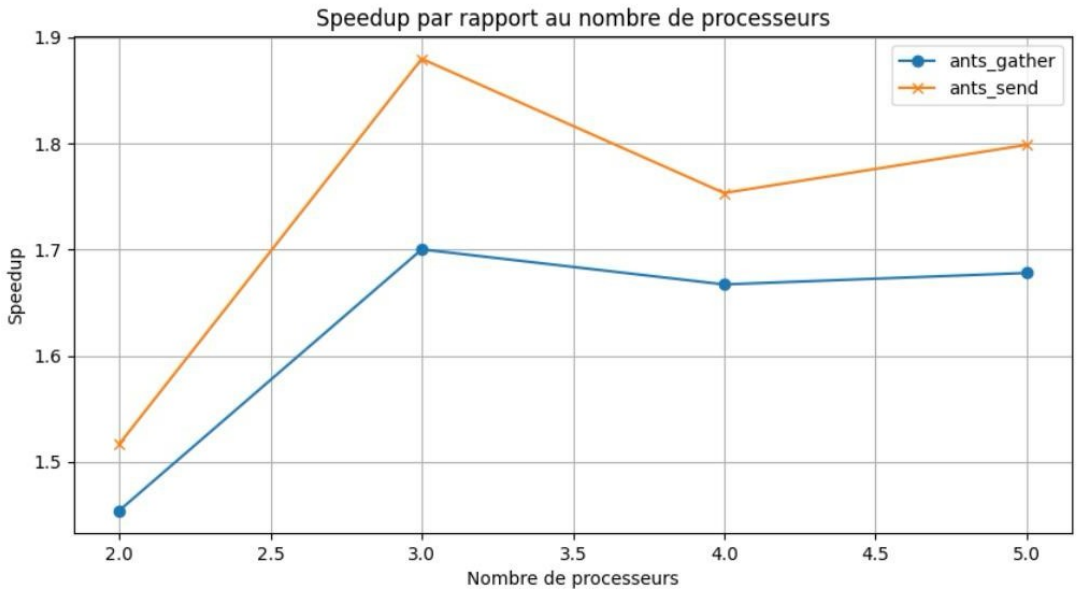

FIGURE 11 – Calcul du Speed-up

L'analyse des courbes de speedup nous permet d'analyser comment le speedup varie avec le nombre de processeurs pour nos deux stratégies de communication différentes dans un environnement de calcul parallèle :

4.1.1 Communication collective

Sur la courbe *ants_ gather*, on observe une augmentation linéaire du speedup en passant de 2 à 3 processeurs. Cependant, entre 3 et 5 processeurs, le speedup stagne et ne montre pas d'amélioration significative. L'amélioration initiale indique que l'algorithme parallèle avec communication collective tire avantage d'avoir plus de ressources de calcul. La stagnation pourrait indiquer que d'autres facteurs, comme la synchronisation entre les processus ou les délais de communication, deviennent prédominants et limitent l'efficacité du speedup.

4.1.2 Communication point à point

La courbe *ants_ send* courbe montre une croissance rapide du speedup en passant de 2 à 3 processeurs, suivie d'une diminution légèrement entre 3 et 4 processeurs, puis d'une chute légère augmentation vers 5 processeurs. La croissance rapide du speedup jusqu'à 3 processeurs suggère que l'algorithme est capable de tirer parti efficacement de la communication point à point. La diminution du speedup après 3 processeurs pourrait être due à l'overhead de la gestion des communications point à point qui devient plus important avec le nombre croissant de processus

4.1.3 Comparaison des deux méthodes

En comparant les deux courbes, plusieurs points sont à noter :
- **Efficacité initiale** : ants_send commence avec un speedup plus élevé que ants_gather à 2 processeurs, ce qui suggère que pour un nombre limité de processeurs, la communication point à point peut être plus efficace.
- **Scalabilité** : ants_gather semble être plus stable et scalable avec l'ajout de processeurs, alors que ants_send présente une chute plus prononcée, indiquant une perte d'efficacité avec l'augmentation des processeurs.
- **Performance relative** : ants_gather montre moins de variation dans le speedup et maintient une meilleure performance relative sur la gamme de processeurs testés, tandis que ants_send présente une plus grande variance, ce qui peut indiquer une sensibilité plus élevée aux problèmes de surcharge de communication.

5 Problèmes rencontrés

1. Dimension des données : Colony.historic_path est un tableau numpy tridimensionnel de dimensions (nb_ants, max_life+1, 2), donc on récupère dans chaque processeur un tableau numpy tridimensionnel de dimension (my_ants_count, max_life+1,2). On veut ensuite les rassembler dans le processeur 0, mais pour ça il faut le concaténer seulement sur la première dimension (axis=0) => Problème résolu
2. Parallélisation des seeds : Quand on prend nb_ants=3 fourmis avec 4 processeurs, on ne voit à l'affichage qu'une seule fourmi. En réalité, c'est bien trois fourmis différentes superposées qui effectuent les mêmes déplacements, parce qu'à chaque fois, elles sont initialisées par la même "seed". En effet, l'initialisation de la classe "Colony" crée self.seeds avec un tableau d'entiers successifs servant de graines uniques pour chaque

fourmi dans la simulation, responsable des choix aléatoires de directions dans le labyrinthe. Chaque fourmi, de l'indice 1 à nb_ants, reçoit une graine aléatoire unique qui peut être utilisée pour générer des séquences aléatoires indépendantes dans des simulations ou des calculs ultérieurs, assurant ainsi que le comportement aléatoire de chaque fourmi est distinct. Donc lors de l'initialisation de ants_local pour seulment 1fourmi/processeur, chaque colonie de fourmi reçoit une série de nombre identique. => idée de résolution : modifier self__init__.seeds pour générer une série de seeds entre [rank, rank+my_ants_counts]

6 Parallélisation du labyrinthe

6.1 Découpage du Labyrinthe

On divise le labyrinthe en 4 sections égales afin que chaque processus travaille sur une partie distincte. Par exemple, pour un labyrinthe de taille N x N, avec P processus, on divise le labyrinthe en 4 sous-labyrinthes :

- Processeur 1 : [0 :N//2,0 :N//2]
- Processeur 2 : [N//2 :N,0 :N//2]
- Processeur 3 : [0 :N//2,N//2 :N]
- Processeur 4 : [N//2 :N, N//2 :N]

On attribue à chaque processus une section unique du labyrinthe à gérer. Chaque processus est responsable de la mise à jour de sa section du labyrinthe, y compris le déplacement des fourmis et la gestion des phéromones dans sa zone.

6.2 Partitionnement des fourmis

Initialement, on, reparti les fourmis en fonction de leur position : chaque fourmi est géré par le même processus qui gère la partie du labyrinthe dans laquelle elle se trouve.

6.3 Calcul des processus

Chaque processus doit faire avancer sa colinie locale de fourmis, mettre à jour les phéromones dans sa parties du labyrinthe (avec les cellules fantômes), puis il doit vérifier que ses fourmis n'ont pas atteint une bordure de son sous-labyrinthe et s'apprête à entrer dans une zone gérée par un autre processus en vérifiant les positions des fourmis par rapport aux limites de la section du labyrinthe qu'elles occupent.

6.4 Communication inter-processus

Lorsqu'une fourmi traverse la frontière, le processus gérant la section d'origine doit envoyer les informations pertinentes sur cette fourmi (position, direction, état de chargement, etc.) au processus gérant la section de destination. Cela peut se faire à travers des opérations MPI telles que MPI_Send et MPI_Recv.

6.5 Rassemblement des informations

Enfin, le processus maitre reçoit les sous-labyrinthes, les colonies locales et les rassembles dans un labyrinthe globale et une colonie globale et une liste de phéromone globale pour gérer l'affichage, avec MPI_gather.