OS202- Systèmes parallèles et distribués

Projet de Parallélisation du Code Fourmis

Guimarães Lino de Paula Eduardo
Sarmanho Freitas Maria Teresa

Mars 2024

Table des matières

1 Description du problème **2**

2 Analyse du code **2**

3 Analyse de la parallélisation **2**
3.1 Étape 1 - Séparer l'affichage 3
3.2 Étape 2 - Partitionner les fourmis 4

4 Mesure de temps et speed-up **5**

5 Partition du Labyrinthe **6**

6 Conclusion **6**

1 Description du problème

Le problème proposé consiste à paralléliser un code Python qui effectue la recherche d'un chemin optimal appliqué à un modèle simple de colonies de fourmis. Le code original est disponible à l'adresse suivante : https://github.com/JuvignyEnsta/Fourmi2024.

Pour la parallélisation, nous exécuterons les étapes suivantes :

— Séparer l'affichage de la gestion des fourmis/phéromones

— Partitionner les fourmis entre les processus restants

— Réfléchir à la manière dont nous pourrions partitionner le labyrinthe pour gérer les fourmis en parallèle sur un labyrinthe distribué entre les processus

Tout d'abord, nous ferons la description et l'analyse des principales parties du code principal, en mettant particulièrement l'accent sur celles susceptibles d'être parallélisées. Ensuite, nous présenterons les stratégies utilisées pour paralléliser le code et partitionner chaque fonction. Enfin, nous discuterons de la manière de mettre en œuvre un code parallèle qui divise également le labyrinthe entre les processus.

2 Analyse du code

Le code original est subdivisé en plusieurs classes :

— Colony : responsable de la création d'une colonie de fourmis et de la gestion de son comportement

— Pheromon : responsable de la mise à jour des phéromones, qui sont soit laissées par des fourmis chargées, soit évaporées selon une loi pré-définie.

— Maze : responsable de la construction du labyrinthe et de son image représentative.

Dans Colony, la fonction "advance" est responsable de faire progresser le code en gérant le tableau des fourmis chargées et non chargées, le tableau des positions des fourmis, la mise à jour de l'historique du chemin des fourmis, la mise à jour des sorties possibles et la marque des phéromones. Donc ce fonction fait uniquement les calculs référents à la gestion des fourmis d'une colonie. La fonction "explore" évolue les fourmis non chargées, en mettant à jour tous les attributs nécessaires : les âges, les directions, etc. La fonction "return_to_nest" fait un travail pareil, mais pour les fourmis chargées.

Les autres fichiers et classes, comme Pheromone et Maze sont utilisés juste pour créer le labyrinthe et la matrice de phéromones qui aide les fourmis à trouver un chemin optimale jusqu'à la nourriture et au nid (notre point de début).

3 Analyse de la parallélisation

La parallélisation était divisé en étapes et les codes résultants se trouvent sur le dépôt GitHub : https://github.com/DuduGuima/Fourmi2024/

Le groupe a fait une mélange de deux types de parallélisation : une parallélisation de données, où chaque processeur/core de calcul est responsable pour un petit groupe de fourmis, et une parallélisation fonctionnelle, où le processeur 0 est responsable par l'affichage et le reste par les calculs.

3.1 Étape 1 - Séparer l'affichage

En premier lieu, pour la 'Question 1', nous avons séparé l'affichage sur le processus 0 et laissé la gestion des fourmis et des phéromones sur le processus 1. Sachant qu'à l'étape 2 il était nécessaire de procéder à une autre division des processus pour gérer les fourmis, nous avions déjà créé des communicateurs différents pour la gestion de l'affichage et des fourmis. Le communicateur responsable de l'affichage est appelé "comm_display" et comprend les processus 0 et 1. Le communicateur responsable de la gestion des fourmis est appelé "new_comm" et comprend tous les processus sauf le 0. Le communicateur 'original' est "comm", qui contient tous les processeurs et est utilisé pour créer les autres.

Étant les processus sur le communicateur "new_comm" responsables de toute la partie calcul et avancement des fourmis et phéromones, la fonction "advance" a été modifiée pour permettre uniquement à ce communicateur d'exécuter ces parties du code dans n'importe quel appel de la fonction. Pour l'affichage des données sur le terminal, la variable "food_counter" modifiée par cette fonction doit être envoyée au processus 0.

Le processus 1 dans "comm_display" doit ensuite envoyer toutes les informations utilisées par la fonction "display" au processus 0 du même communicateur. Cela est fait de façon optimisé en effectuant un Broadcast des variables d'intérêt à l'intérieur de la fonction "advance", qui est responsable de modifier ces paramètres.

```
def advance(self, the_maze, pos_food, pos_nest, pheromones,
   food_counter=0):
    if not new_comm == MPI.COMM_NULL:
        loaded_ants = np.nonzero(self.is_loaded == True)[0]
        {... Calcul ...}
        comm.send(food_counter,dest= 0)

    if not comm_display == MPI.COMM_NULL:
        food_counter = comm_display.bcast(food_counter,root =
   1)
        self.age = comm_display.bcast(np.array(self.age),root
   =1)
        self.historic_path = comm_display.bcast(np.array(self.
   historic_path),root =1)
        self.directions = comm_display.bcast(np.array(self.
   directions),root =1)
        pheromones.pheromon=comm_display.bcast(pheromones.
   pheromon,root=1)
    return food_counter

def display(self, screen):
    [screen.blit(self.sprites[self.directions[i]], (8*self.
   historic_path[i, self.age[i], 1], 8*self.historic_path[i,
   self.age[i], 0])) for i in range(self.directions.shape[0])]
```

Dans la fonction principale, une série de conditions"if rank_i == 0" était utilisée pour laisser au processus 0 les lignes de code responsables de l'affichage avec Pygame, des fonctions display des classes Maze, Pheromon et Ants, ainsi que de la capture de la première nourriture et de l'affichage des données sur le terminal.

Le code contenant les modifications mentionnées se trouve sur "Question 1" dans le dépôt GitHut.

3.2 Étape 2 - Partitionner les fourmis

À la second étape de la parallélisation, la partition de la gestion de fourmis a été parallélisé. Comment déjà mentionnée, ce partie est exécutée par les processus du communicateur "new_comm".

La stratégie choisi consiste en chaque processus créer et gérer sa propre colonie, qui contient une fraction du nombre de fourmis de la colonie du code séquentiel.Ces colonies restent partie d'une même fourmilière parce qu'elles habitent au même labyrinthe (même position de la nourriture et du nid) et se déplacent en suivant les mêmes marques des phéromones.

Pour garantir que chaque colonie suit la même generation de choix aléatoires, il faut modifier comme chaque processeur de calcul crée les seeds :

```
def __init__(self, nb_ants, pos_init, max_life, index_min,
   index_max):
        {...}

    self.seeds = np.arange(index_min+1, index_max+1, dtype=np.
   int64)

        {...}
```

où (index_min,index_max) vont aider avec le nombre de fourmis que chaque processeur ira gérer. Le constructeur

Les classes Pheromon et Maze restent sans modifications, vu qu'elles sont utilisées par la colonie mais ne sont pas partie d'elle.

Dans Colony, les fonctions "return_to_nest" et "explore" sont gérées de façon indépendant par chaque processus, vu qu'elles réalisent changements locaux des fourmis qui n'affectent pas directement le comportement de la fourmilière sauf en relation au nombre de fourmis. Cependant, la fonction "advance" dévient la responsable par maintenir les processus opérant sur le même labyrinthe. Cet-à-dire, ce fonction contient des commandes pour faire communiquer à tous les processus sur le "new_comm" la quantité actuelle de nourriture et la position actualisée des phéromones.

Les phéromones de chaque petite colonie sont mis à jour avec l'expression contenue dans le notebook Jupyter. Pour faire une combinaison avec les matrices de phéromones de processeurs de calcul différents, le code utilise la fonction Reduce et un Allgather parmi les membres de "new_comm" :

```
def advance(self, the_maze, pos_food, pos_net, pheromones,
   food_counter=0):
        {...}
    food_counter = comm.reduce(food_counter,op = MPI.SUM,root
   =0)
        {...}
    if not new_comm == MPI.COMM_NULL:
            {...}
        old_pheromones = pheromones.pheromon.copy()

        [pheromones.mark(self.historic_path[i, self.age[i], :],
                [has_north_exit[i], has_east_exit[i],
                has_west_exit[i], has_south_exit[i]],
```

```
                old_pheromones) for i in range(self.directions.
    shape[0])]

        result= np.zeros_like(pheromones.pheromon)
        new_comm.Allreduce(pheromones.pheromon,result, op = MPI
    .MAX)
        pheromones.pheromon = result.copy()

        {...}
```

La mise à jour globale du "food_counter" est faite juste après sa modification par l'appelle de la fonction "return_to_nest". Nous utilisons un Reduce réalisé par le communicateur global "comm", vu que ce valeur est utilisé par le communicateur d'affichage et celui de gestion des fourmis.

Les autres attributs et actions des fourmis sont calculés de façon indépendant par chaque processus à partir des données partagées. Les choix des chemins aléatoires de chaque colonie avant le marquage du phéromone, par exemple, est fait à partir des générations indépendantes, de façon que les déplacements des fourmis ne soient pas répètes entre les processus. Les choix des chemins après la marquage du phéromone utilise le tableau de phéromones souvent mis-à-jour.

4 Mesure de temps et speed-up

Pour comparer la vitesse d'exécution du code séquentiel et parallèle, nous avons choisi deux stratégies. En première lieu, nous avons mesuré le temps écoulé entre le début du programme et le moment où la variable food_counter atteint la valeur de 1000 dans la fonction principale (main). Ce temps prend déjà en compte la partie non parallélisable du code, c'est-à-dire l'initialisation.

```
if __name__ == "__main__":
{...}
deb_1 =time.time()
while True:
        {...}
        food_counter = ants.advance(a_maze, pos_food, pos_nest,
    pherom, food_counter)
        pherom.do_evaporation(pos_food)
        end = time.time()
        {...}
        if food_counter>=1000:
            print("Time to reach 1000 foods: ",end - deb_1)
            pg.quit()
            exit(0)
```

Lors de l'étape 1 de la parallélisation, lorsque nous avons divisé les calculs de l'affichage, le temps d'exécution est passé de 26,50s en séquentiel à 23,13s en parallèle. Cela nous donne le speed-up :

$$S(n) = \frac{t_s}{t_p(n)} = 1,14$$

Pour l'étape 2, nous avons utilisé l'approche de mesurer, pendant l'exécution du code, le FPS moyenne stabilisé après la marquage du phéromone des deux approches et les comparer.

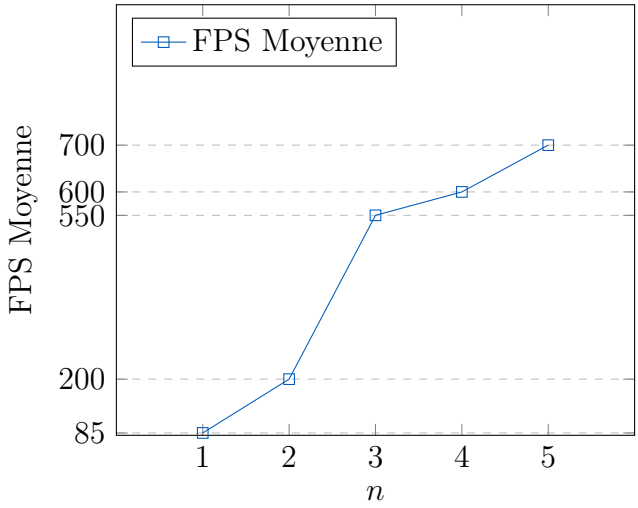
Étape 2 FPS Moyenne 85 n=1

FPS Moyenne 200 n = 2

FPS Moyenne 550 n = 3

FPS Moyenne () n = 4

FPS Moyenne () n = 5



5 Partition du Labyrinthe

Pour implémenter le code parallèle en partitionnant le labyrinthe entre les processus, il serait intéressant d'utiliser une stratégie maître-esclave. Le processus 0, appelé maître, enverrait initialement des parties du labyrinthe aux autres processus, qui seraient chargés de calculer la quantité de fourmis dans chaque région du labyrinthe. La valeur serait renvoyée au processus 0, qui répartirait les régions de manière aussi équitable que possible entre les processus, de sorte que chacun contienne approximativement la même quantité de fourmis à gérer. Après l'avance des états des fourmis, des phéromones, de la quantité de nourriture, etc., le processus 0 reçoit et rassemble toutes les informations. Enfin, le labyrinthe serait à nouveau réparti entre les processus pour calculer le nombre de fourmis avec les informations mises à jour, et le cycle se répéterait.

6 Conclusion