OS202 : TP Fourmi 2024

Thibault Cochet

Mars 2024

Pré-requis

- Importer la bibliothèque mpi4py au début du code
- Initialiser les variables comm puis size et rank dans le "main"

1 Première partie : Séparation des calculs et de l'affichage

Dans le "main" du code ants.py, on repère les parties du code qui relèvent de l'affichage. Ces parties concernent les instances "display()" des classes.
Avant d'entrer dans la boucle "While True", on peut donner au processeur l'affichage du labyrinthe :

```
if rank == 0 :
        mazeImg = a_maze.display()
```

Le processeur 0 gère aussi l'affichage des FPS et la capture d'écran :

```
if rank == 0 :

    if food_counter == 1 and not snapshop_taken:
        pg.image.save(screen, "MyFirstFood.png")

    print(f"FPS : {1./(end-deb):6.2f}, nourriture : {food_counter:7d}", end='\r')
```

Maintenant, il faut séparer la partie "calculs" de la partie "affichage".

Pour cela, on effectue une itération avec la fonction "ant.advance" sur le processeur 1.
On envoie ensuite sur le processeur 0 les différentes instances des classes Pheromone et Colony pour que l'affichage soit à jour :

```
if rank == 1 :

    food_counter = ants.advance(a_maze, pos_food, pos_nest, pherom, food_counter)
    pherom.do_evaporation(pos_food)
    comm.Send(ants.directions , dest = 0 , tag = 1)
    comm.Send(pherom.pheromon , dest = 0 , tag = 2)
    comm.send(food_counter , dest = 0 , tag = 3)
    comm.Send(ants.historic_path , dest = 0 , tag = 4)
    comm.Send(ants.age , dest = 0 , tag = 5)
```

Ensuite, le processeur 0 initialise les instances à récupérer puis les reçois du processeur 1, avant d'afficher les différentes classes :

```
if rank == 0 :

    ants.directions = d.DIR_NONE*np.ones(nb_ants, dtype=np.int8)
```

```
    pherom.pheromon  = np.zeros((size_laby[0]+2, size_laby[1]+2), dtype=np.double)
    ants.historic_path = np.zeros((nb_ants, max_life+1, 2), dtype=np.int16)
    ants.age = np.zeros(nb_ants, dtype=np.int64)

    comm.Recv(ants.directions  , source=1, tag = 1)
    comm.Recv(pherom.pheromon ,source=1, tag = 2)
    food_counter = comm.recv(source=1, tag=3)
    comm.Recv(ants.historic_path , source=1, tag = 4)
    comm.Recv(ants.age , source=1, tag = 5)


    pherom.display(screen)
    screen.blit(mazeImg, (0, 0))
    ants.display(screen)
    pg.display.update()
```

Pour évaluer les performance de la parallélisation étant donné que le programme tourne en boucle, j'ai choisi de chronométrer le temps qu'il fallait pour que la première nourriture arrive au nid.
Ainsi, pour l'exécution simple du programme avec la commande `"python3 ants.py"`, le temps est de 5.47 secondes. Pour l'exécution sur 2 processeurs, pour la taille de labyrinthe initiale de 25 par 25, avec la commande `"mpiexec -n 2 python3 ants_2proc3.py"`, le temps est de 3.79 secondes.

Le speed-up vaut donc $\frac{5.47}{3.79} = 1.44$.

2 Seconde partie : Partition des fourmis entre les processus de rang non nul

Pour paralléliser le programme entre les différents processeurs en répartissant les fourmis, j'ai décidé de créer une colonie de `nb_ants //(size-1)` fourmis par processeur. Cette colonie évoluera à chaque itération.
Toutefois, pour assurer la cohérence de la colonie dans son ensemble, il est nécessaire de permettre aux processus de se communiquer les phéromones générés localement. Au début, j'initialise donc deux classes de phéromones : `global_pheromon` et `local_pheromon`.
Pour faire le lien entre les phéromones calculés localement et les phéromones globales, on calcule le maximum entre chaque phéromone local sur le processus 0. On renvoie ensuite le résultat vers les autres processus :

```
local_pherom.pheromon  = np.zeros((size_laby[0]+2, size_laby[1]+2))
global_pherom.pheromon  = np.zeros((size_laby[0]+2, size_laby[1]+2))

for i in range(1,size):

    comm.Recv(local_pherom.pheromon , source=i, tag = 2)
```

```
    for j in range(local_pherom.pheromon.shape[0]) :
                for k in range (local_pherom.pheromon.shape[1]) :
                    if local_pherom.pheromon[j,k] > global_pherom.pheromon[j,k] :
                        global_pherom.pheromon[j,k] = local_pherom.pheromon[j,k]

global_pherom.display(screen)
                                        ...
for i in range (1,size) :
    comm.Send(global_pherom.pheromon , dest = i , tag = 3)
```

De même pour la mise à jour du compteur de nourriture sur le processeur 0, il faut additionner les différents compteurs locaux afin d'avoir la nourriture totale ramenée au nid:

```
food_counter = 0
for i in range(1,size):
    local_food = comm.recv(source=i, tag = 1)
    food_counter += local_food
```

Pour l'affichage des fourmis, j'ai procédé comme la première partie avec les commandes Send et Recv sauf que l'affichage se fait colonie pour chaque colonie locale, toujours sur le processus 0.

Concernant les performances du programme, j'ai utilisé la même métrique que pour la partie 1 à savoir le temps avant que la première nourriture arrive au nid.
J'obtiens un temps de 4.78 secondes avec `"mpiexec -n 4 python3 ants_parallel.py"` l'execution contre 5.57 secondes pour l'éxecution classique de ants.py

Le speed-up vaut donc $\frac{5.57}{4.78} = 1.17$.

Remarques

- Les temps d'exécution dépendent du moment ou on exécute les codes. Cela est sûrement du aux processus qui tournent ou non sur la machine en arrière-plan. Dans chaque partie, je me suis assuré d'exécuter les codes à la suite pour le calcul du speed-up.

- La manière de calculer le speed-up n'est en fin de compte peut-être pas la plus appropriée. En effet, le code faisant intervenir le facteur aléatoire de manière assez importante, il aurait peut-être été plus judicieux de calculer la moyenne des FPS sur un certain temps afin de véritablement se baser sur le speed-ups des calculs sans faire intervenir l'aléatoire.

3 Conclusion

On remarque que le speed-up est plus important dans le cas où on parallélise seulement l'affichage. L'accélération faible dans le cas de la parallélisation des fourmis, avec la méthode que j'ai utilisée est certainement due au nombre important de communications entre les processus qui freine l'exécution.

4 Réflexion sur la parallélisation du code en répartissant le labyrinthe partitionné entre les processus

Partitionner le labyrinthe de manière à paralléliser équitablement les calculs semble assez peu intéressant à première vue car au lancement, toutes les fourmis sont en haut à gauche de labyrinthe. De même lorsque le chemin optimal a été trouvé, les fourmis restent sur ce même chemin.

Pour répondre à l'énonce, il pourrait être judicieux d'utiliser la méthode maître-esclave afin de partitionner dynamiquement le labyrinthe en fonction du nombre de fourmis qui s'y trouvent. Cela nécessiterait néanmoins de revoir la définition des différentes instances des classes Colony et Pheromon. En effet, il sera alors par exemple nécessaire d'ajouter ou de retirer les fournis des différentes cases et ainsi des colonies locales.

Cela fait de cette méthode une méthode difficile à implémenter et qui risque de s'avérer peu efficace en raison du nombre important de communications qui vont devoir être implémentées.