OS202 Recherche de Chemin Optimal

Yuheng ZHANG[♢]

On a calculé la fréquence moyenne des images en calculant le temps d'exécution de 1000 images. Dans l'algorithme ACO non parallélisé, la fréquence moyenne des images (FPS) était de 295 images par seconde, et les fourmis ont mangé un total de 286 aliments.

Séparation de l'affichage

On Sépare l'affichage (sur le proc 0) et la gestion des fourmis et phéromones (sur le proc 1) par dans *ants-affichage.py*:

Cette tâche ne peut être exécutée que lorsque le processus est égal à 2, la taille du labyrinthe *size-laby* est 15. Le résultat de FPS est de 613 images par seconde et la colonie reçoit 286 denrées alimentaires, deux fois plus rapide que le programme non parallélisé en calculant:

$$SpeedUp = \frac{t_{serial}}{t_{parallel}} = \frac{FPS_{parallel}}{FPS_{serial}}$$

Séparation des fourmis

Les parties du code parallélisables lorsqu'on partitionne uniquement les fourmis sont :

- La partie du code qui gère le déplacement des fourmis peut être parallélisée, puisque chaque fourmi peut agir indépendamment des autres. On peut diviser le groupe de fourmis en sous-groupes et les assigner à différents processus pour calculer leurs déplacements et interactions avec le labyrinthe et les phéromones.
- La mise à jour des pistes de phéromones peut également être parallélisée. Cela comprend le renouvellement de la phénomone par l'activité des fourmis et la vaporisation de la phénomone elle-même.

```
#Iteration
snapshop_taken = False
while True:
    for event in pg.event.get():
        if event.type == pg.QUIT:
            pg.quit()
            exit(0)
    if rank == 0:
        deb = time.time()
        pherom = comm.recv(source=1, tag=21)
        ants = comm.recv(source=1, tag=22)
        food_counter = comm.recv(source=1, tag=24)

        pherom.display(screen)
        screen.blit(mazeImg, (0, 0))
        ants.display(screen)
        pg.display.update()
        if food_counter == 1 and not snapshop_taken:
            pg.image.save(screen, "MyFirstFood_affichage.png")
            snapshop_taken = True
        end = time.time()
        print(f"FPS : {1./(end-deb):6.2f}, nourriture : {food_counter:7d}", end='\r')

    else:
        food_counter = ants.advance(a_maze, pos_food, pos_nest, pherom, food_counter)
        pherom.do_evaporation(pos_food)
        comm.send(pherom, dest=0, tag=21)
        comm.send(ants, dest=0, tag=22)
        comm.send(food_counter, dest=0, tag=24)
```

Figure 1.1: Parallèlisation de l'affichage

```python
while True:
    for event in pg.event.get():
        if event.type == pg.QUIT:
            pg.quit()
            exit(0)
    count += 1
    if count > 1000:
        break
    if rank == 0:
        ants_globe = Colony(0, pos_nest, max_life)
        pherom_gather = []
        food_counter = 0
        for i in range(1, size):
            comm.send(pherom_globe.pheromon, dest=i, tag=i*10)
            pherom_tmp = comm.recv(source=i, tag=i*10+1)

            ants_tmp = comm.recv(source=i, tag=i*10+2)
            food_counter += comm.recv(source=i, tag=i*10+3)

            ants_globe.merge(ants_tmp)
            pherom_gather.append(pherom_tmp)


        pherom_globe.pheromon = global_update_pherom_avg(pherom_gather)

        pherom_globe.display(screen)
        screen.blit(mazeImg, (0, 0))
        ants_globe.display(screen)
        pg.display.update()

        if food_counter != 0 and not snapshop_taken:
            pg.image.save(screen, "MyFirstFood_proc.png")
            snapshop_taken = True

    else:
        pherom.update_with(comm.recv(source=0, tag=rank*10))
        food_counter = ants.advance(a_maze, pos_food, pos_nest, pherom, food_counter)
        pherom.do_evaporation(pos_food)
        comm.send(pherom.pheromon, dest=0, tag=rank*10+1)
        comm.send(ants, dest=0, tag=rank*10+2)
        comm.send(food_counter, dest=0, tag=rank*10+3)
```

Figure 1.2: Séparation des fourmis - Première approche

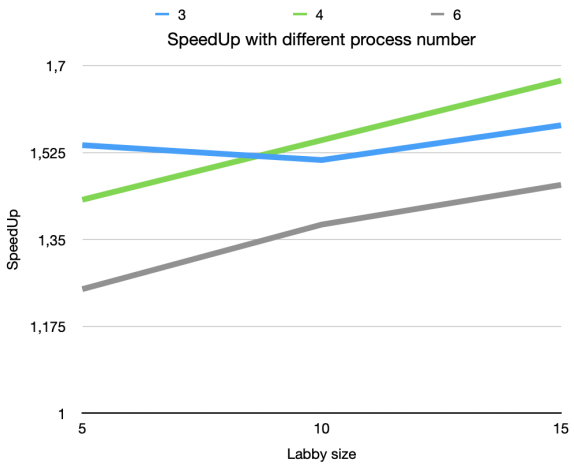

Figure 1.3: SpeedUp de la permière rapproche avec paramètres différents

Dans cette section, nous essayons deux approches différentes de la parallélisation

Première Approche

Dans *ants-proc.py*, Chaque processus calcule une le comportement d'une partie des fourmis ,leur marques des phéromones et la vaporisation des phéromones.

Cet extrait de code 1.2 montre comment la parallélisation peut être utilisée pour diviser le travail de simulation entre différents processus.

On ajuste la taille du labyrinthe *size-labby* dans l'algorithme et le nombre de processus pour observer l'accélération de la tâche :

On fait une analysation sur le résultat 1.3:

$$S(n) = \frac{t_s}{f.t_s + \frac{(1-f)t_s}{n}} \Longrightarrow \frac{1}{f}$$

- L'accélération augmente généralement avec la taille du labyrinthe pour tous les nombres de

processus. Cela pourrait signifier que plus le problème est grand, plus il y a des avantages à utiliser la parallélisation. Il correspond à **la loi d'Amadal**.

- Avec 3 processus, l'accélération semble s'accroître de manière plus significative avec la taille du labyrinthe, ce qui suggère que la parallélisation est efficace pour les petits nombres de processus et les grands problèmes.

- Avec 4 et 6 processus, bien qu'il y ait une augmentation de l'accélération avec la taille du labyrinthe, l'effet est moins marqué. Cela pourrait indiquer des problèmes d'échelonnabilité où l'ajout de processus supplémentaires n'entraîne pas une amélioration proportionnelle du speedup.

On a également observé que les fourmis mettent plus de temps à trouver le chemin de la nourriture, ce qui peut être lié à la manière dont la phéromone est synchronisée et mise à jour. On a utilisé la méthode consistant à obtenir la phéromone globale et à la mettre à jour en faisant la somme et la moyenne au niveau du processus racine après que tous les sous-processus ont été calculés. Si la carte globale des phéromones est obtenue en faisant la moyenne des mises à jour locales, il peut en résulter un lissage excessif des chemins à forte concentration sur la carte des phéromones, ce qui réduit l'incitation de la colonie de fourmis à explorer des chemins inconnus, car les nouveaux chemins, éventuellement plus courts, sont "submergés" par les phéromones des chemins existants.

Seconde Approche

Dans *ants-proc-upgrade.py* 1.4, on essaie seulement de paralléliser le processus de recherche de chemin des fourmis et le processus d'évaporation de la phéromone, alors que la mise à jour des marqueurs de phéromone par les fourmis sera effectuée de manière séquentielle. Cela signifie que chaque fourmi opère indépendamment pendant la recherche de nourriture et que l'évaporation de la phéromone est calculée en parallèle afin d'optimiser le temps de traitement.

Cependant, les mises à jour des phéromones sont effectuées de manière centralisée, pour maintenir la cohérence des données et éviter les conflits d'écriture qui peuvent survenir lorsque plusieurs fourmis essaient de mettre à jour la même partie du chemin en même temps.

On analyse la performance de la nouvelle approche en comparant son temps d'exécution et la qualité de la solution avec l'algorithme non parallèle dans 1.5.

On fait une analysation sur le résultat 1.5:

- La diminution de la vitesse avec 6 processus pour des labyrinthes de plus grande taille pourrait indiquer que les frais généraux associés à la gestion d'un plus grand nombre de processus pourraient l'emporter sur les avantages de la parallélisation. Cela pourrait être dû à la surcharge de communication et à la concurrence pour les ressources partagées.

- La diminution du taux d'accélération par rapport à la première approche peut être liée à l'ajout d'une section série.C'est un exemple de la loi d'Amdahl.

On a également observé que les fourmis étaient plus efficaces pour trouver de la nourriture, ce qui suggère que la méthode de mise à jour en série des phéromones est efficace

Réflexions sur le Parallèle du Labyrinthe

Pour paralléliser le labyrinthe, on doit le diviser en régions et assigner ces régions à différents processus ou threads. Chaque processus ou thread est responsable du comportement des fourmis et de la mise à jour des phéromones dans la région qui lui a été attribuée.

- Partitionnement du labyrinthe : Idéalement, chaque processeur est responsable d'une sous-région de taille à peu près égale.

```python
while True:
    count += 1
    if count > 1000:
        break
    for event in pg.event.get():
        if event.type == pg.QUIT:
            pg.quit()
            exit(0)

    for i in range(1, size):
        if rank == i and i > 1:
            pheron_buffer = comm.recv(source=rank-1, tag=(rank-1)*10+1)
            food_counter_prev = comm.recv(source=rank-1, tag=(rank-1)*10+2)
            pheron_globe.update_with(pheron_buffer)
            food_counter_curr = ants_loc.advance(a_maze, pos_food, pos_nest, pherom_globe, food_counter_prev)
            food_counter =food_counter_curr - food_counter_prev

        if rank == i and i < size-1:
            comm.send(pherom_globe.pheromon, dest=rank+1, tag=rank*10+1)
            comm.send(food_counter_curr, dest=rank+1, tag=rank*10+2)

        if rank == size-1:
            comm.send(pherom_globe.pheromon, dest=1, tag=(size-1)*10+1)
            comm.send(food_counter_curr, dest=1, tag=(size-1)*10+2)

        if rank == 1:
            pheron_buffer = comm.recv(source=size-1, tag=(size-1)*10+1)
            food_counter_prev = comm.recv(source=size-1, tag=(size-1)*10+2)
            pheron_globe.update_with(pheron_buffer)
            food_counter_curr = ants_loc.advance(a_maze, pos_food, pos_nest, pherom_globe, food_counter_prev)
            food_counter =food_counter_curr - food_counter_prev

        pherom_globe.do_evaporation(pos_food)

    comm.barrier() # wait for everybody to be ready

    for i in range(1, size):
        if rank == i:
            comm.send(food_counter, dest=0, tag=rank*100+1)
            comm.send(ants_loc, dest=0, tag=rank*100+2)
        if rank == 1:
            comm.send(pherom_globe.pheromon, dest=0, tag=rank*100+3)


    if rank == 0:
        ants_globe = Colony(0, pos_nest, max_life)
        for j in range(1, size):
            food_counter += comm.recv(source=j, tag=j*100+1)
            ants_tmp = comm.recv(source=j, tag=j*100+2)
            ants_globe.merge(ants_tmp)
        pherom_buffer = comm.recv(source=1, tag=1*100+3)

        pherom_globe.update_with(pherom_buffer)


        pherom_globe.display(screen)
        screen.blit(mazeImg, (0, 0))
        ants_globe.display(screen)

        if food_counter != 0 and not snapshop_taken:
            pg.image.save(screen, "MyFirstFood_proc_update.png")
            snapshop_taken = True
        pg.display.update()
```

Figure 1.4: Séparation des fourmis - Seconde approche

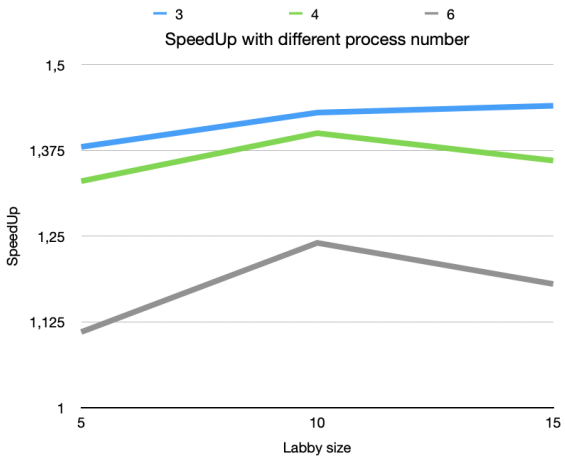

Figure 1.5: SpeedUp de la seconde approche avec paramètres différents

- Traitement des frontières : Lorsque les fourmis se déplacent d'une région à l'autre, les informations les concernant doivent être échangées par l'intermédiaire de l'interface de transmission de messages (MPI). En outre, les mises à jour des phéromones doivent être synchronisées entre les régions. Par conséquent, lors de la mise à jour des sous-régions, nous utilisons l'approche de la cellule fantôme pour obtenir des informations critiques.

- Stratégie parallèle : le processus de déplacement des fourmis et d'évaporation/renforcement de la phéromone est exécuté en parallèle, tandis que la mise à jour globale de la phéromone est utilisée comme point de synchronisation.

- Décomposition des données : En plus de la décomposition du domaine pour partitionner physiquement l'espace du labyrinthe, il faut également utiliser la décomposition des données - un processus pour traiter les informations de localisation de toutes les fourmis et un autre processus pour traiter la carte des phéromones.