ES201

Projet - Recherche de Chemin Optimal

Guidado Aissatou Hairiya
Taulois Braga Bernardo

Mars 2024

Lien Github

Table des matières

1 Présentation de l'ordinateur sur lequel le TP a été effectué :

L'ordinateur sur lequel le TP a été réalisé est un Intel core I7 possédant 8 coeurs physiques de calcul sous Linux. une synthèse de la sortie donnée par lscpu est :

- Architecture : x86_64
- Mode(s) opératoire(s) des processeurs : 32-bit, 64-bit
- Taille des adresses : 39 bits physical, 48 bits virtual
- Processeur(s) : 8
- Liste de processeur(s) en ligne : 0-7
- Identifiant constructeur : GenuineIntel
- Nom de mpdèle : 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz
- Famille de processeur : 6
- Modèle : 140
- Thread(s) par cœur : 2
- Cœur(s) par socket : 4
- Socket(s) : 1
- Caches (sum of all) :
  - L1d : 192 KiB (4 instances)
  - L1i : 128 KiB (4 instances)
  - L2 : 5 MiB (4 instances)
  - L3 : 12 MiB (1 instance)

2 Analyse à priori du problème

Le problème de fourragement des fourmis est un problème classique d'optimisation combinatoire dans lequel les fourmis cherchent le chemin le plus court entre leur nid et une source de nourriture, en utilisant des phéromones pour marquer les chemins parcourus et guider les autres fourmis vers la source de nourriture.

Le code soumis à notre étude est une version séquentielle permettant de résoudre le problème de fourragement des fourmis et nous devons le paralléliser.

De prime à bord, on constate que le problème est plus **memory bound** que **CPU bound** car une grande partie du temps de calcul est consacrée à la mise à jour des différentes données. Ces opérations impliquent beaucoup d'opérations de lecture et d'écriture, en effet, les phéromones sont stockés dans des tableaux numpy et tout a long de l'exécution du code, des valeurs sont lues et mise à jour dans ces tableaux.

Ensuite, on constate qu'il s'agit d'un problème **Nearly embarassingly parallel** car le parallélisme du code se fait en envoyant des données au processus 0 qui va se charger d'exploiter ces informations pour l'affichage. De plus, il y a un déséquilibre des charges dans ce cas car l'affichage est beaucoup plus rapide que les calculs. Le problème étant Nearly Embarassingly parallel, il peut être totalement parallélisable et les résultats seront dont ensuite rassemblées ensemble dans le processus 0.

3 Séparation de l'affichage et de la gestion des fourmis/-phéromones sur deux processus

Ici, le processus 0 se charge de l'affichage et le processus 1 se charge de la gestion des fourmis/phéromones. Pour ce faire, il faut répertorier toutes les valeurs qui sont nécessaires pour effectuer l'affichage et assurer leur envoie du processus 1 au processus 0.

La procédure de parallélisation que nous avons effectué est la suivante :

— Nous nous sommes rassurés de n'utiliser la bibliothèque pygame que dans le processus 0 qui est chargé de l'affichage, Pour ce faire, il a fallu modifier la définition de certaines méthodes. De plus, les classes colony et pheromon avaient des méthodes display que nous avons supprimés pour créer des fontions propre au processus 0 et qui ne seront exécutées que par lui :

```
# functions that replace the object methods
def getColor_sent_pheromon(i: int, j: int, pheromon):
    val = max(min(pheromon[i, j], 1), 0)
    return [255*(val > 1.E-16), 255*val, 128.]
def display_sent_pheromon(pheromon, screen):
    [[screen.fill(getColor_sent_pheromon(i, j, pheromon),
   (8*(j-1), 8*(i-1), 8, 8)) for j in range(1, pheromon.
   shape[1]-1)] for i in range(1, pheromon.shape[0]-1)]

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

— Dans chacun des processus, on a déclaré uniquement les variables nécessaires pour cette dernière pour réduire au maximum les problèmes de concurrences d'accès mémoire. La seule variable qui est utilisée par les deux processus est : **a_maze**.

— On a constaté que lorsqu'on fermait la fenêtre du labyrinthe, le code n'arrêtait pas de s'exécuter pour le processus 1 mais uniquement pour le processus 0. On a donc écrit une fonction qui permet la gestion de ce problème, il envoie un message au processus 1 qui se charge à son tour d'arrêter son exécution :

```
def end_all_processes():
    pg.quit()
    globCom.send(True, dest=1)
    exit(0)
```

4 Séparation de l'affichage et de la gestion des fourmis et phéromones sur plusieurs processus

La parallélisation du code est plus difficile dans cette section car en plus d'assurer la communication avec le processus 0, on doit également assurer la communication avec l'ensemble des autres processus qui se charge du calcul. La méthode que nous avons utilisé est la suivante :

— On détermine le nombre de fourmis qui seront affecté à chaque processus en divisant le nombre total de fourmis par le nombre de processus -1 (en retirant le processus 0).

— Pour envoyer les résultats des calculs effectué par les processus au processus 0, nous avons crée un tableau de tableaux permettant le stockage de toutes les variables qui sont mises à jour à la suite des calculs et qui participent à l'affichage. Cela permet la minimisation des communications parfois coûteuses de MPI. C'est ce tableau qui est envoyé au processus 0 :

```
food_counter = ants.advance(a_maze, pos_food, pos_nest,
   pherom, food_counter)
pherom.do_evaporation(pos_food)
send_nd_array = np.array((pherom.pheromon, ants.directions,
    ants.historic_path, ants.age, food_counter), dtype=
   object)
globCom.send(send_nd_array, dest=0)
```

— Le processus 0 quand il reçoit les données, les récupère chacune dans un tableau différent en appliquant l'opération np.concatenate pour rassembler les valeurs envoyés par chaque processus i et pour permettre de les passer en paramètres aux fonctions **display_sent_ants** et **display_sent_pheromon**.

```
for i in range(nbp - 1):
    received = globCom.recv(source=(i + 1))
    if(i == 0):
        sent_pheromon = received[0]
        directions_concatenated = received[1]
        historic_path_concatenated = received[2]
        age_concatenated = received[3]
    else:
        directions_concatenated = np.concatenate((
   directions_concatenated, received[1]))
        sent_pheromon = np.maximum(sent_pheromon, received
   [0])
        historic_path_concatenated = np.concatenate((
   historic_path_concatenated, received[2]), axis=0)
        age_concatenated = np.concatenate((age_concatenated
   , received[3]))
    total_food_found += received[4]
display_sent_pheromon(sent_pheromon, screen)
screen.blit(mazeImg, (0, 0))
display_sent_ants(screen, sprites, directions_concatenated,
    historic_path_concatenated, age_concatenated)
pg.display.update()
```

— Ensuite, le processus 0 renvoie les valeurs concaténées aux autres processus car ils en ont besoin pour poursuivre leurs calculs.

```
for i in range(nbp - 1):
globCom.send(sent_pheromon, dest=(i + 1), tag=1)
```

5 Calcul du Speedup

: Soit ts le temps d'exécution séquentielle et tp(n) le temps d'exécution sur n unités de calcul, la formule de calcul du speedup est la suivante :

$$S(n) = \frac{ts}{tp(n)}$$

Nous avons évalué le temps pour atteindre la première nourriture et le temps lorsque le nombre de nourriture atteint 1000. On a consigné les résultats dans le tableau suivant :

Type de parallélisation		Temps une nourriture (s)	Temps 1000 nourritures (s)	Speed up une nourriture	Speed up 1000 nourritures
Serial		6.01	22.26	-	-
Séparer affichage		4.41	15.07	1.36	1.48
Séparer affichage et fourmis	n=1	6.88	25.44	0.87	0.88
	n=2	6.76	25.49	0.89	0.87
	n=3	4.67	24.19	1.29	0.92

TABLE 1 – Comparaison des temps obtenus pour chaque type de parallélisation.

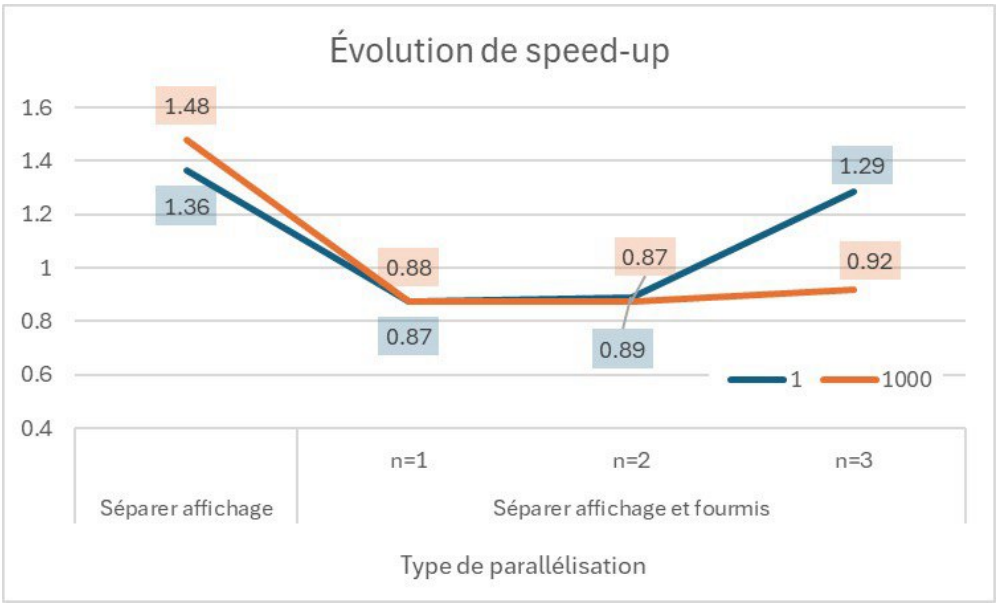

FIGURE 1 – Évolution du speedup selon le type de parallélisation réalisé.

5.1 Analyse

Lorsqu'on ne sépare que l'affichage et le calcul entre 2 processus, le speedup est optimal. En effet,on atteint la première nourriture 26% plus vite qu'en séquentiel et on atteint la 1000e nourriture 32% plus vite. Cependant, lorsqu'on parallélise les fourmis entre les différents processus non nuls, le speedup se détériore par rapport à la valeur précédente. Néanmoins, on constate que lorsqu'on augment le nombre de processus (n=3), on obtient une valeur plus acceptable ce qui nous amène à penser que si notre ordinateur disposait de plus de processus, on aurait pu obtenir un meilleur résultat.

6 Proposition de solution pour le partitionnement du labyrinthe pour gérer les fourmis en parallèle sur un labyrinthe distribué entre les processsus :

Pour gérer les fourmis en parallèle sur un labyrinthe distribué, il faut tout d'abord utiliser un algorithme **Maitre-esclave** car on aura un problème de déséquilibre de charges vu que les différentes sections de labyrinthe qui seront attribué aux processus n'auront pas la même charge de calcul.

On constate que les fourmis et les phéromones seront plus concentrées le long du chemin le plus court vers la nourriture. On pourrait donc au début, réaliser une partition égale mais ajuster les partitions au fur et à mesure (lorsqu'on aura accès au chemin le plus court) en répartissant le plus de processus le long de ce chemin.

Ensuite, pour déterminer les partitions, il va falloir définir des ghost cells au bord des différentes sections car les fourmis ont besoin des informations des cellules voisines pour déterminer leur mouvement dans la méthode *explore* de la classe *Colony*.

Il est important de noter qu'il faut s'assurer que les fourmis peuvent se déplacer sur l'ensemble du labyrinthe et il faut impérativement maintenir une communication entre les processus pour permettre ce mouvement.

7 Conclusion

Il était question pour nous de paralléliser le programme séquentiel qui permet de résoudre le problème de fourragement des fourmis. Nous avons pour ce faire commencer par diviser l'affichage et le calcul sur deux processus distincts et nous avons obtenu des résultats plutôt satisfaisants avec une valeur de speedup acceptable. Ensuite, nous avons parallélisé les fourmis sur plusieurs processus différents et nous avons utilisé le processus 0 pour l'affichage. Nous avons aussi obtenu d'assez bons résultats mais un peu moins convaincants que ceux obtenus précédemment. Enfin, nous avons proposé une solution pour permettre la gestion des fourmis en parallèle sur un labyrinthe distribué.