École Nationale Supérieur des Techniques Avancées

OS202 - Projet : Fourmis 2024

FALABELLA Leonardo
MAGALHÃES CONTENTE Sergio

Palaiseau
2023

Table des matières

Table des figures

1 Introduction

Avant de commencer à discuter des résultats du projet, voici les configurations de l'ordinateur utilisé pour le projet :

- CPU(s) : 16
  - Liste des CPU(s) en ligne : 0-15
- ID du vendeur : AuthenticAMD
  - Nom du modèle : AMD Ryzen 7 4800H avec Radeon Graphics
    - Famille de CPU : 23
    - Modèle : 96
    - Thread(s) par cœur : 2
    - Cœur(s) par socket : 8
    - Socket(s) : 1
    - Pas : 1
    - Augmentation de fréquence : activée
    - CPU MHz max : 2900.0000
    - CPU MHz min : 1400.0000
  - Caches (somme de tous) :
    - L1d : 256 KiB (8 instances)
    - L1i : 256 KiB (8 instances)
    - L2 : 4 MiB (8 instances)
    - L3 : 8 MiB (2 instances)

Les paramètres utilisés lors de la réalisation du projet étaient les suivants :

- Nombre de cycles = 7000
- alpha = 0.9
- beta = 0.99
- Taille du labyrinthe = 25, 25
- Vie maximale = 1000

Ils ont été choisis afin de permettre une simulation rapide, mais suffisamment précise pour refléter les modifications des algorithmes qui parallélisent le code fourni par le professeur.

1.1 Algorithme sans parallélisation

Pour l'algorithme sans type de parallélisation, le temps nécessaire pour calculer les mouvements des fourmis et les phéromones, ainsi que le temps nécessaire pour afficher ces informations à l'écran ont été calculés. Ensuite, ces valeurs ont été additionnées pour trouver le temps total.

- Temps total : 43.42931509017944 (s)

2 Séparation de la affichage et de calculs

2.1 En utilisant seulement 2 processus

Afin de paralléliser le code entre 2 processus, il est important de choisir une stratégie qui permet aux processus de diviser le tâche entre affichage et calcules.

La stratégie qui a été employée dans ce projet est celle où le processus de rang 0 est uniquement responsable de l'affichage des fourmis, du phéromone et du labyrinthe dans la fenêtre de Pygame. L'autre processus, dans ce cas, est chargé de réaliser tous les calculs subséquents à chaque cycle de la boucle principale. Ces calculs comprennent la mise à jour du "historic_path" de chaque fourmi, de ses directions, le marquage du phéromone et également le compteur de nourriture qui arrive à la fourmilière.

2.1.1 Processus 0

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

            Status = MPI.Status()
            ants_attributes, pherom, food_counter = comm.recv(source=1, status=Status)

            # Updating ants
            ants.seeds = ants_attributes[0]
            ants.is_loaded = ants_attributes[1]
            ants.age = ants_attributes[2]
            ants.historic_path = ants_attributes[3]
            ants.directions = ants_attributes[4]


            snapshop_taken = False
            for event in pg.event.get():
                if event.type == pg.QUIT:
                    pg.quit()
                    exit(0)

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


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

            end = time.time()

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

FIGURE 1 – Logique du Rank 0 pour le problème 1

Premièrement, le processus commence par l'affichage du labyrinthe à travers la fonction display de sa propre classe et le stocke dans la variable "mazeImg".

Après cela, commence la communication entre lui et le deuxième processus. Nous avons implémenté l'approche point à point dans laquelle le processus 0 recevra les valeurs des attributs "seeds", "is_loaded", "age", "historic_path", "directions", en plus du "pheromone" et "food_counter" calculés dans l'objet "ants" local du processus 1, à travers la fonction MPI "recv" où la source est justement le processus de rang 1.

Et puis, nous mettons à jour la valeur de ces mêmes attributs dans l'objet "ants" local du processus 0, afin de le mettre à jour avec les informations calculées par le processus 1 et finalement nous faisons l'affichage des fourmis, du phéromone et du compteur de nourriture.

2.1.2 Processus 1

```
if rank == 1:
            food_counter = ants.advance(a_maze.maze, pos_food, pos_nest, pherom, food_counter)
            pherom.do_evaporation(pos_food)

            comm.send(([ants.seeds, ants.is_loaded, ants.age, ants.historic_path, \
                        ants.directions], pherom, food_counter), dest=0)
```

FIGURE 2 – Logique du Rank 1 pour le problème 1

Le processus de rang 1 met à jour les informations de la fourmi de son objet "ants" local, ainsi que la valeur de "food_counter" et pheromone local également. Ceci est réalisé en appelant les fonctions "ants.advance(a_maze.maze, pos_food, pos_nest, pherom, food_counter)" et "pherom.do_evaporation(pos_food)".

Enfin, le processus envoie ces valeurs à l'autre processus de rang 0 à travers le communicateur global et la méthode "send" de MPI.

2.2 Quels gains on a obtenu (speed-up) ?

Comme mentionné dans l'introduction, les valeurs de référence sont celles données par l'algorithme sans parallélisation. En utilisant le code parallélisé avec 2 processus, nous avons trouvé les temps suivants :

— Temps total : 27.267609119415283

$$\text{Accélération (Speed up)} = \frac{\text{Temps Ancien}}{\text{Temps Nouveau}}$$

$$\text{Speed up (total)} = \frac{43.42931509017944}{27.267609119415283} \approx 1.59$$

De cette manière, il est évident que les valeurs trouvées ont reçu une amélioration considérable (60%) en utilisant seulement 2 processeurs.

3 Partitionant le calculs de les fourmis

3.1 Description du code

Poursuivant la parallélisation du code, dans cette section, nous décrivons comment nous avons mis en œuvre la séparation du calcul des fourmis entre différents processus. Pendant ce temps, l'affichage a été maintenu dans un seul processus isolé.

Tout d'abord, nous avons une partie d'initialisation des paramètres et des objets pertinents du code. Cette partie peut être considérée comme séquentielle, car nous n'avons pas encore de processus simultanés effectuant des calculs distincts. Nous avons également défini les communicateurs qui seront utilisés ultérieurement, le premier, appelé communicateur global, impliquant tous les processus de calcul ou d'affichage, et le second, appelé communicateur de calcul, incluant uniquement les processus générant les fourmis.

```
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
nbp = comm.Get_size()

color = 0 if rank == 0 else 1

comm_calc = comm.Split(color)
rank_calc = comm_calc.Get_rank()
size_calc = comm_calc.Get_size()
```

FIGURE 3 – Définition des communicateurs

Ensuite, nous créons uniquement dans le processus d'affichage les objets servant de conteneurs pour recevoir les informations des autres processus, ants_global et pherom.

```
if rank == 0:
    ants_global = Colony(0, nb_ants, pos_nest, max_life)
    pherom = pheromone.Pheromon(size_laby, pos_food, alpha, beta)
```

FIGURE 4 – Définition des objets globales

Il a également été nécessaire de répartir les fourmis entre les processus effectuant les calculs, de sorte que chacun reçoive un nombre similaire de fourmis à gérer. Il a également été nécessaire de définir quatre autres tableaux, recv_cout, displacements, historic_recvcounts et historic_displacements. Ceux-ci seront utilisés ultérieurement pour assurer la bonne communication entre les processus en tant qu'arguments de la fonction *Gatherv*. De plus, les objets locaux qui seront gérés par chaque processus et les tampons utilisés dans les communications entre les processus de calcul ont été définis.

Concernant les variables "food_counter" et "pheromone", l'opération *Reduce* est employée afin d'assurer que les valeurs recueillies sur l'ensemble des processus soient prises en compte dans le calcul final de la quantité de nourriture. Il est nécessaire de mettre à jour la valeur de "pheromone_loc" après avoir effectué l'opération de *MAX*, pour garantir que la concentration de phéromone la plus élevée soit correctement répartie parmi tous les processus.

Ainsi, la partie d'initialisation est terminée. Nous examinerons maintenant la boucle principale du programme. Tout d'abord, les unités de calcul font avancer la position de leurs fourmis respectives et de la phéromone. Ensuite, les informations de toutes les unités de calcul sont envoyées au processus

de calcul principal. Enfin, celui-ci envoie l'ensemble des informations concernant toutes les fourmis au processus d'affichage.

```
if color != 0:
          # Calculation processes
          food_counter_local = np.array(ants_local.advance(a_maze, pos_food, \
              pos_nest, pherom_local, food_counter_local), dtype=np.int64)
          pherom_local.do_evaporation(pos_food)
          pherom_send = np.array(pherom_local.pheromon.copy())


          comm_calc.Reduce([food_counter_local, MPI.INT64_T], \
                           [food_counter_colored, MPI.INT64_T], op=MPI.SUM, root=0)
          comm_calc.Reduce([pherom_send, MPI.DOUBLE], [pheromon_colored, MPI.DOUBLE],\
                          op=MPI.MAX, root=0)
          pherom_local.pheromon = comm_calc.bcast(pheromon_colored, root=0)
          comm_calc.Gatherv(ants_local.age, \
                           [age_colored, recv_count, displacements, MPI.INT64_T], root=0)
          comm_calc.Gatherv(ants_local.historic_path, \
                           [historic_path_colored, historic_recvcounts,\
                             historic_displacements, MPI.INT16_T], root=0)
          comm_calc.Gatherv(ants_local.directions, \
                           [directions_colored, recv_count, displacements, MPI.INT8_T], root=0)
          comm.send([seeds_colored, is_loaded_colored, age_colored,\
                      historic_path_colored, directions_colored, \
                          food_counter_colored, pheromon_colored], dest=0)
```

FIGURE 5 – Communication collective entre processus

Pendant ce temps, le processus d'affichage affiche la carte du labyrinthe et attend de recevoir les informations mises à jour. Une fois reçues, il met à jour l'objet ants_global qui génère toutes les fourmis, le compteur de nourriture et la phéromone. Ensuite, il affiche les fourmis et les phéromones mises à jour dans la fenêtre.

```
mazeImg = a_maze.display()

            ants_attributes = comm.recv(source=1)
            # Updating ants
            ants_global.seeds = ants_attributes[0]
            ants_global.is_loaded = ants_attributes[1]
            ants_global.age = ants_attributes[2]
            ants_global.historic_path = ants_attributes[3]
            ants_global.directions = ants_attributes[4]
            food_counter = ants_attributes[5]
            pherom.pheromon = ants_attributes[6]

                 ## Print screen pygame window
            snapshop_taken = False
            for event in pg.event.get():
                if event.type == pg.QUIT:
                    pg.quit()
                    exit(0)

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

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

FIGURE 6 – Affichage des valeurs

3.2 Speed up

Comme mentionné dans l'introduction, les valeurs de référence sont celles données par l'algorithme sans parallélisation. Désormais, il faut analyser le speed up pour différents nombres de processeurs.

$$\text{Accélération (Speed up)} = \frac{\text{Temps Ancien}}{\text{Temps Nouveau}}$$

Processeurs	**Temps total**	**Speed up (total)**
2	28.74717450142	1.51
3	25.59487009048	1.70
4	27.22495317459	1.60
5	28.41768026352	1.53
6	29.43900108337	1.48
7	29.08656096458	1.49
8	30.95087361336	1.40

TABLE 1 – Performance avec différents nombres de processeurs

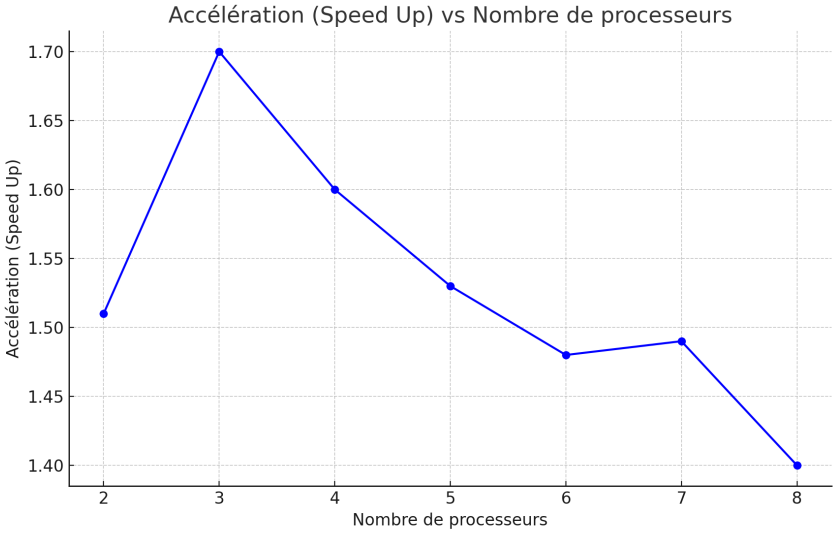

FIGURE 7 – Speed up por différents processeurs

L'analyse des résultats obtenus en fonction du nombre de processeurs utilisés pour la parallélisation du code montre une tendance intéressante concernant l'accélération (Speed Up) du processus. Comme indiqué, le Speed Up atteint son pic avec 3 processeurs, où l'accélération est d'environ 1.70, avant de commencer à diminuer avec l'ajout de processeurs supplémentaires. Cette observation peut être attribuée à plusieurs facteurs principaux liés aux limites de l'amélioration de performance qu'apporte la parallélisation.

Tout d'abord, il est important de noter que l'augmentation du nombre de processeurs au-delà d'un certain point entraîne des coûts supplémentaires liés à la gestion de la communication et de la synchronisation entre les processus. Chaque processus supplémentaire introduit une complexité accrue dans la coordination des tâches et le partage des données, ce qui peut ralentir l'exécution globale en raison du temps perdu dans les communications inter-processus et la gestion de la cohérence des données.

De plus, il est possible que le problème lui-même ne soit pas parfaitement divisible de manière à bénéficier pleinement de l'ajout de processeurs supplémentaires. La loi d'Amdahl souligne cette limitation, en indiquant que l'amélioration de performance obtenue par la parallélisation est limitée par la partie du programme qui doit s'exécuter séquentiellement.

4 Décrire vos réflexions de comment vous voyez la mise en oeuvre du code en parallèle si on partionne en plus le labyrinthe entre les divers procesus.

Par la suite, nous décrirons comment nous pourrions gérer en parallèle des sous-ensembles du labyrinthe, de manière à ce que chaque processus soit responsable du mouvement des fourmis présentes dans son sous-ensemble. Pour cela, il est envisageable de considérer deux approches : statique ou dynamique.

L'approche statique divise le labyrinthe de manière à ce que chaque processeur prenne en charge des blocs du labyrinthe de même taille. Cette méthode n'est pas efficace, puisque les fourmis ne sont pas réparties également dans le labyrinthe, et par conséquent, la charge de chaque processus sera différente. Il est donc plus judicieux de choisir une approche dynamique.

Pour achever la distribution du labyrinthe entre les processus, il est possible de suivre les étapes suivantes. Soit n_f le nombre de fourmis, n_p le nombre de processus et $\delta \in \mathbb{N}, \delta > 1$ une constante arbitraire, de sorte que l'objectif soit de distribuer $(n_f/n_p) \pm \delta$ à chaque processus. Pour réaliser la division des secteurs du labyrinthe, on peut suivre l'algorithme suivant. Étant donnée une certaine distribution de fourmis dans le labyrinthe, il est possible de le diviser en 2 parties égales verticalement par exemple. Ensuite, on compte combien de fourmis se trouvent dans chaque section, N_f, si $N_f \in [(n_f/n_p) - \delta, (n_f/n_p) + \delta]$, la zone correspondante est associée à un processeur pour la gérer. Et l'autre partie continue d'être divisée, maintenant dans le sens contraire au précédent (horizontal), jusqu'à ce que tout le labyrinthe soit distribué entre tous les processeurs.

Si le N_f n'est pas dans l'intervalle accepté, on peut appliquer un raisonnement similaire à une recherche binaire, "déplaçant" la division vers la moitié de la zone précédente dans la direction adéquate. Jusqu'à trouver une zone du labyrinthe qui contient le nombre adéquat de fourmis.

L'algorithme est illustré par l'exemple suivant, avec 1000 fourmis, 4 processeurs et $\delta = 100$. Pour un cas idéal, où N_f de l'une des zones appartient à l'intervalle, on a une situation comme :

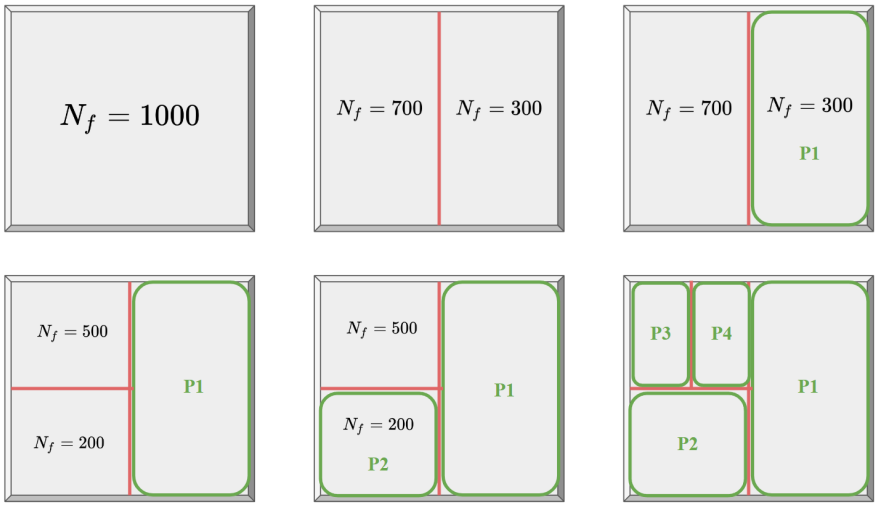

FIGURE 8 – Division des secteurs pour une situation idéale

Ou par le suivant dans une situation où il est nécessaire de redéfinir les secteurs. Dans un premier temps, l'algorithme identifie 600 fourmis d'un côté et 400 de l'autre. Aucune de ces valeurs ne se trouve dans l'intervalle d'intérêt. De ce fait, une nouvelle division est générée, à la moitié de la zone à droite, et les fourmis sont comptées. Si aucun N_f ne satisfait, la division est refaite au point médian entre la première et la deuxième division, suivant le principe de la recherche binaire. Pour cet exemple, à ce moment, nous trouvons une zone qui possède le nombre adéquat de fourmis et elle est attribuée au processus 1. Ensuite, l'algorithme continuerait normalement, divisant la zone restante perpendiculairement.

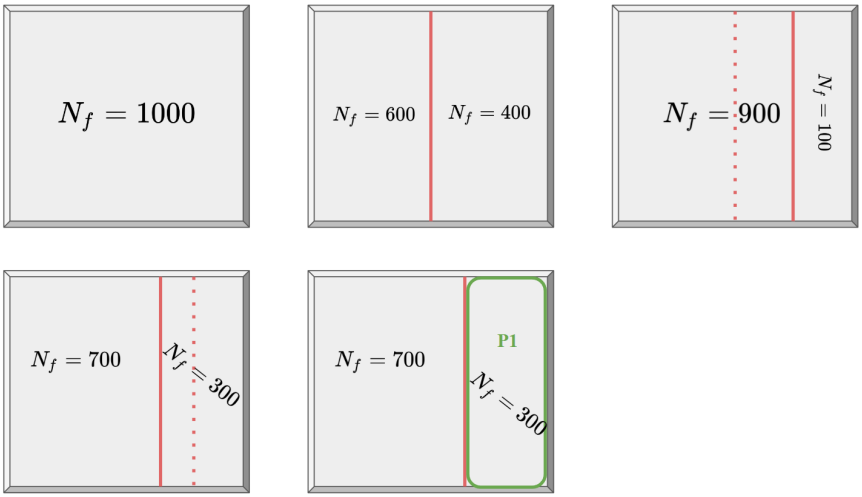

FIGURE 9 – Division des secteurs pour une situation non-idéale