PROJET OS202
Recherche de chemin optimal

Baghino, Gianluca & Gallego, Natalia
AST
ENSTA Paris
gianluca.baghino@ensta-paris.fr
natalia.gallego@ensta-paris.fr

I. Introduction

L'intelligence en essaim a généré une classe d'algorithmes coopératifs inspirés du comportement des insectes sociaux.

Ce projet utilise une base de code avec les fondements originaux des algorithmes ACO, où une population d'agents imitant le comportement des fourmis résout efficacement le problème de la recherche de nourriture dans un labyrinthe 2D. Chaque fourmi artificielle, aux caractéristiques différentes, parcourt le labyrinthe, ajustant son orientation, son état et son histoire en fonction de l'environnement.

L'objectif est de développer un modèle parallèle pour améliorer l'efficacité du code de simulation. La parallélisation se déroulera en plusieurs étapes, à commencer par la séparation de la gestion des fourmis/phéromones et l'affichage des responsabilités entre les différents processus. Ensuite, la partition de la classe **ant** entre les processus sera implémentée, avec l'écran de gestion du processus principal. Enfin, la possibilité de diviser le labyrinthe pour la gestion simultanée des fourmis dans un maillage distribué entre processus sera discutée.

Ce travail documentera les aspects parallélisables du code lors du partitionnement **ant**, évaluera les gains réalisés en termes de vitesse d'exécution et explorera les perspectives d'un partitionnement du labyrinthe pour une gestion parallèle plus poussée. Les résultats seront présentés pour donner un aperçu des considérations et des résultats de cette parallélisation.

II. Code séquentiel

Ce code simule le comportement de la colonie de fourmis, il est exécuté séquentiellement dans un premier temps.

```
if __name__ == "__main__":
    import sys
    import time
    pg.init()
    size_laby = 25, 25
    if len(sys.argv) > 2:
        size_laby = int(sys.argv[1]),int(sys.argv
    [2])

    resolution = size_laby[1]*8, size_laby[0]*8
    screen = pg.display.set_mode(resolution)
    nb_ants = size_laby[0]*size_laby[1]//4
    max_life = 500
    if len(sys.argv) > 3:
        max_life = int(sys.argv[3])
    pos_food = size_laby[0]-1, size_laby[1]-1
    pos_nest = 0, 0
    a_maze = maze.Maze(size_laby, 12345)
    ants = Colony(nb_ants, pos_nest, max_life)
    unloaded_ants = np.array(range(nb_ants))
    alpha = 0.9
    beta  = 0.99
    if len(sys.argv) > 4:
        alpha = float(sys.argv[4])
    if len(sys.argv) > 5:
        beta = float(sys.argv[5])
    pherom = pheromone.Pheromon(size_laby,
                                pos_food, alpha,
    beta)
    mazeImg = a_maze.display()
    food_counter = 0


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

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

        food_counter = ants.advance(a_maze,pos_food,
                                    pos_nest,pherom,
                                    food_counter)
        pherom.do_evaporation(pos_food)
        end = time.time()
        if food_counter == 1 and not snapshop_taken:
            pg.image.save(screen, "MyFirstFood.png")
            snapshop_taken = True
        # pg.time.wait(500)
        print(f"FPS : {1./(end-deb):6.2f},
    nourriture :
              {food_counter:7d}", end='\r')
```

Dans ce code, **pygame** est d'abord initialisé (**pg.init()**) et les dimensions par défaut du labyrinthe sont définies (**size_laby**), qui peuvent être saisies à partir de la ligne de commande.

Ensuite, vous définissez la résolution de la fenêtre en fonction des dimensions du labyrinthe et créez la fenêtre à l'aide de **pygame**. Le nombre de fourmis dans la colonie (**nb_ants**) est également défini, calculé comme une fraction des dimensions du labyrinthe et de l'âge maximum qu'une fourmi peut atteindre avant de mourir (**max_life**).

Poursuivant l'initialisation du labyrinthe, la position de l'aliment dans le labyrinthe (**pos_food**) est définie dans le

coin inférieur droit et la position du nid dans le labyrinthe (‘pos_nest‘) dans le coin supérieur gauche.

Une fois ces variables système définies, nous commençons par l’initialisation du programme. Pour ce faire, nous commençons par **a_maze**, qui est l’objet qui représente le labyrinthe et fournit des fonctions pour le visualiser. En plus de **ants**, qui représente la colonie de fourmis et **unloaded_ants** qui est la liste d’index des fourmis qui ne transportent pas de nourriture. Les phéromones et la visualisation initiale sont également initialisées, ceci à l’aide de **pherom**, qui est l’objet qui gère la logique des phéromones dans le labyrinthe, **mazeImg**, image du labyrinthe à afficher dans la fenêtre de visualisation et **food_counter**, le compteur de la quantité de nourriture trouvée par les fourmis.

Les tâches pour démarrer le programme sont exécutées de manière séquentielle ; pour cela, la visualisation du labyrinthe, des fourmis et des phéromones est d’abord affichée sur l’écran. D’un autre côté, les fourmis avancent dans la simulation (**ants.advance**), collectant de la nourriture, laissant des phéromones et mettant à jour le compteur de nourriture. En plus d’évaporer les phéromones dans le labyrinthe. Si de la nourriture est trouvée pour la première fois, une capture d’écran est prise.

Enfin, des informations sur le **FPS** et la quantité de nourriture trouvée sont imprimées sur la console.

Dans un premier temps, pour connaître le comportement du programme, on a défini un labyrinthe de dimension **25x25** et une durée de vie maximale pour les fourmis égale à **500**. Avec cette configuration, l’image suivante a été obtenue comme la première fois que de la nourriture était obtenue dans le colonie.

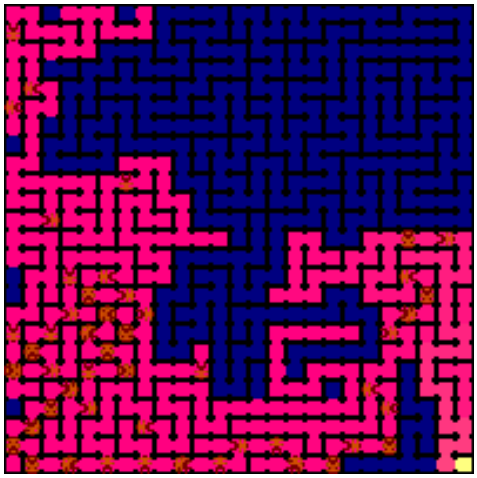

Fig. 1. Premier repas, programme séquence, dimensions 25x25

Avec cette même configuration, une étude générale du comportement du FPS a été réalisée, car celle-ci pourra être utilisée pour de futures comparaisons avec des systèmes parallélisés qui seront lancés dans le futur. La valeur FPS moyenne obtenue est de **102.53** et une image de son comportement dans les 900 itérations est présentée.

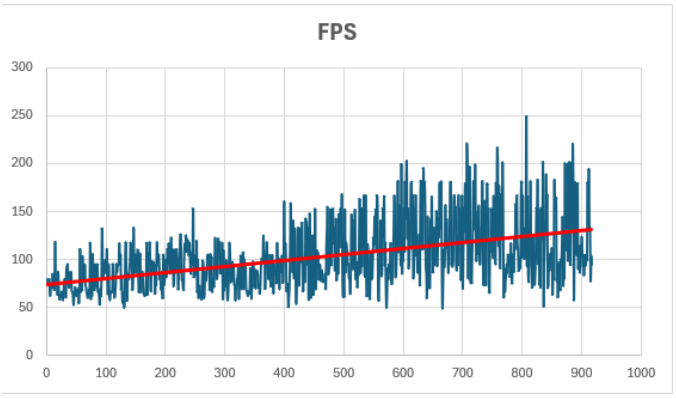

Fig. 2. Graphique de comportement FPS

Le système a également été configuré avec un labyrinthe **50x50** pour étudier son comportement. En augmentant les dimensions du labyrinthe, sa difficulté augmente également, donc les fourmis ont besoin de plus de temps pour trouver de la nourriture, c’est pourquoi il a fallu augmenter leur vie maximale pour qu’elles puissent trouver de la nourriture avant de mourir, cette vie maximale a été définie dans **1000**.

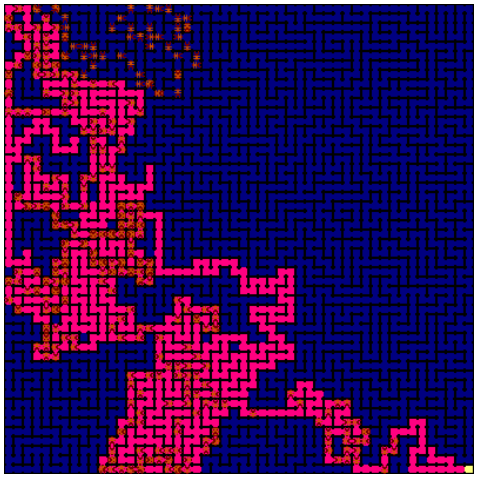

Fig. 3. Premier repas, programme séquentiel, dimensions 50x50

III. Première parallélisation: Séparant affichage (sur le proc 0) et gestion des fourmiS/phéromones (sur le proc 1)

Ce code présente une structure de programme parallèle utilisant la bibliothèque **mpi4py** pour la communication interprocessus dans un environnement de programmation distribué. Le code est divisé en deux blocs principaux, gérés par des processus différents (**rank 0 et 1**), dans le but de séparer les responsabilités de visualisation et de gestion des fourmis et des phéromones.

```
if __name__ == "__main__":
    import sys
    import time
    from mpi4py import MPI

    comm = MPI.COMM_WORLD
```

```
    rank = comm.Get_rank()
    size = comm.Get_size()

    pg.init()
    size_laby = 25, 25
    if len(sys.argv) > 2:
        size_laby = int(sys.argv[1]),int(sys.argv[2])

    resolution = size_laby[1]*8, size_laby[0]*8
    screen = pg.display.set_mode(resolution)
    nb_ants = size_laby[0]*size_laby[1]//4
    max_life = 500
    if len(sys.argv) > 3:
        max_life = int(sys.argv[3])
    pos_food = size_laby[0]-1, size_laby[1]-1
    pos_nest = 0, 0
    a_maze = maze.Maze(size_laby, 12345)
    ants = Colony(nb_ants, pos_nest, max_life)
    unloaded_ants = np.array(range(nb_ants))
    alpha = 0.9
    beta  = 0.99
    if len(sys.argv) > 4:
        alpha = float(sys.argv[4])
    if len(sys.argv) > 5:
        beta = float(sys.argv[5])
    pherom = pheromone.Pheromon(size_laby,
                                pos_food, alpha, beta)
    mazeImg = a_maze.display()
    food_counter = 0
    snapshop_taken = False

    if rank == 1:
        while True:
            food_counter = ants.advance(a_maze,
                                        pos_food,
                                        pos_nest,
                                        pherom,
                                        food_counter)
            pherom.do_evaporation(pos_food)
            end = time.time()
            ant_data = [ants.age, ants.is_loaded,
                        ants.directions]
            last_positions = [ants.historic_path[
                                i, ants.age[i], :]
                              for i in range(
                                  ants.directions.shape[0])]
            comm.send((pherom, food_counter,
                       snapshop_taken,
                       ant_data, last_positions), dest=0)

    elif rank == 0:
        while True:
            for event in pg.event.get():
                if event.type == pg.QUIT:
                    pg.quit()
                    comm.Abort(0)
                    sys.exit()

            deb = time.time()

            pherom, food_counter, snapshop_taken,
            ant_data, last_positions = comm.recv(source=1)

            ants.age, ants.is_loaded, ants.directions = ant_data
            for i, last_position in enumerate(
                last_positions):
                ants.historic_path[i, ants.age[i],
                                   :] = last_position
            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, "MyFirstFood2.png")
                snapshop_taken = True
            end = time.time()  # Define 'end' here
            print(f"FPS : {1./(end-deb):6.2f}, nourriture :
                  {food_counter:7d}", flush=True)
```

L'exécution du code est divisée en deux branches : une pour le processus de **rank 1** (gestion des fourmis et des phéromones) et une autre pour le processus de **rank 0** (affichage).

Dans le processus de **rank 1**, les données pertinentes telles que les objets phéromones, les compteurs de nourriture, les informations sur les fourmis et les dernières positions sont envoyées au processus de **rank 0**.

Dans le processus de **rank 0**, sont reçues les données envoyées par le processus de **rank 1**. Ces données comprennent l'état mis à jour des phéromones, le compteur de nourriture, les indicateurs de capture d'écran, les données des fourmis et leurs dernières positions. Les informations sur les fourmis et leurs derniers mouvements sont utilisées pour mettre à jour l'état des fourmis lors du processus de visualisation.

Il convient de noter que dans une première version du code de **rank 1**, à chaque cycle, il envoyait la matrice complète avec l'historique du voyage des fourmis qui, bien qu'il améliorait les performances par rapport au programme sérialisé, ne permettait pas au système de se comporter de la meilleure façon possible, en raison de la grande quantité de données; De ce fait, il a été défini pour envoyer uniquement la dernière valeur de cette matrice à chaque itération et prendre soin de l'ajouter à sa propre matrice au **rank 0**.

Dans le processus avec **rank 0**, l'affichage est effectué en continu en boucle tout en écoutant les mises à jour envoyées par le processus avec **rank 1**. Les informations sur le **FPS** et la quantité de nourriture trouvée dans le processus sont imprimées avec **rank 0**.

En divisant les responsabilités de visualisation et de gestion des fourmis/phéromones entre deux processus, une séparation efficace des tâches est obtenue et permet une exécution parallèle plus efficace. Ceci est essentiel pour améliorer la vitesse de simulation, en particulier dans les environnements distribués.

Afin de permettre une comparaison directe entre les systèmes à l'avenir, les mêmes tests du système ont également été effectués pour déterminer son comportement. Autrement dit, le programme a été testé avec un labyrinthe **25x25** et un labyrinthe **50x50**.

Pour la première configuration, l'image de la première fois que la nourriture est obtenue dans la fourmilière est présentée,

dont on voit qu'elle est identique à celle du programme sérialisable.

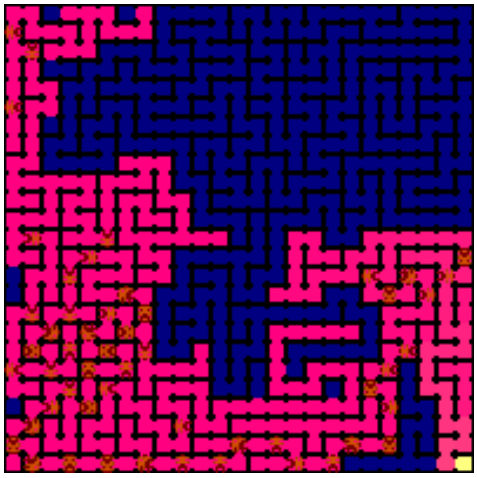

Fig. 4. Premier repas, parallélisation sur deux processeurs, dimensions 25x25

Cependant, cette fois, nous avons obtenu un FPS moyen de **191.73**, ce qui est proche du double du résultat obtenu précédemment. Son comportement au cours des 900 premières itérations est visible dans le graphique ci-dessous :

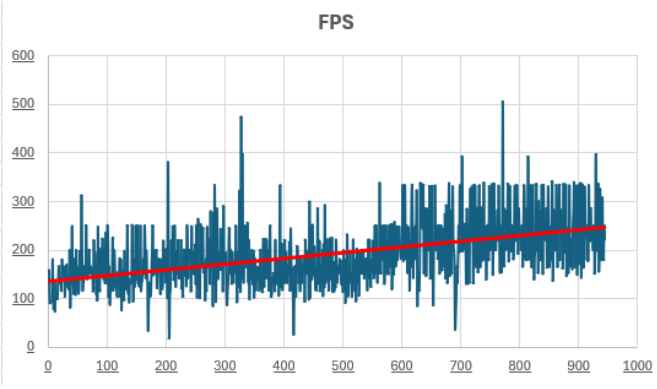

Fig. 5. Graphique de comportement FPS

Le résultat du premier repas obtenu dans un labyrinthe **50x50** est également partagé dans l'image suivante. Par rapport au cas **25x25**, dans ce cas, les résultats coïncident également avec ceux obtenus dans le programme qui fonctionne en série.

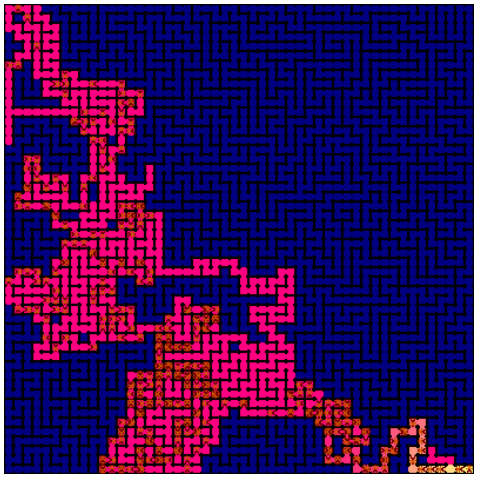

Fig. 6. Premier repas, parallélisation sur deux processeurs, dimensions 50x50

La différence de performances (**FPS**) entre le premier programme et le programme parallélisé est due à la capacité de traitement simultané exploitée par la parallélisation.

Dans le programme parallélisé, les tâches de gestion des fourmis et des phéromones (processus 1) et de visualisation (processus 0) sont exécutées simultanément dans des processus distincts, ce qui permet à ces tâches d'être exécutées simultanément, ce qui permet une meilleure utilisation des ressources du système et une accélération de l'exécution.

De plus, étant donné que le système sur lequel le code est exécuté possède plusieurs cœurs de processeur, la parallélisation peut permettre à chaque processus de s'exécuter sur un cœur distinct, tirant ainsi le meilleur parti des ressources multicœurs disponibles.

IV. Deuxième parallélisation: Répartir les fourmis entre les processus dont le rang est non nul (0 continue de gérer l'affichage)

Ce code présente une structure de programme parallèle utilisant la bibliothèque **mpi4py** pour la communication interprocessus dans un environnement de programmation distribué.

```
if __name__ == "__main__":
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()

    pg.init()
    size_laby = 50, 50
    if len(sys.argv) > 2:
        size_laby = int(sys.argv[1]), int(sys.argv
    [2])
    resolution = size_laby[1] * 8, size_laby[0] * 8
    screen = pg.display.set_mode(resolution)
    nb_ants = size_laby[0] * size_laby[1] // 4
    max_life = 2000
    if len(sys.argv) > 3:
        max_life = int(sys.argv[3])
    pos_food = size_laby[0] - 1, size_laby[1] - 1
    pos_nest = 0, 0
    ants = Colony(nb_ants, pos_nest, max_life)
    a_maze = maze.Maze(size_laby, 12345)
    unloaded_ants = np.array(range(nb_ants))
    alpha = 0.9
    beta = 0.99
```

```
    if len(sys.argv) > 4:
        alpha = float(sys.argv[4])
    if len(sys.argv) > 5:
        beta = float(sys.argv[5])
    pherom = pheromone.Pheromon(size_laby,
                                pos_food,
                                alpha, beta)
    mazeImg = a_maze.display()
    food_counter = 0
    snapshop_taken = False

    nb_ants_l = nb_ants//(size-1)
    nb_remainder = nb_ants%(size-1)

    if rank == size-1:
        ants_l = Colony(nb_ants_l+nb_remainder,
                        pos_nest, max_life)
    elif rank > 0:
        ants_l = Colony(nb_ants_l, pos_nest,
                        max_life)

    if rank > 0:
        while True:
            food_counter = ants_l.advance(a_maze,
                                          pos_food,
                                          pos_nest,
                                          pherom,

    food_counter)
            pherom.do_evaporation(pos_food)
            end = time.time()
            ant_data = np.array([ants_l.age,
                                 ants_l.is_loaded,
                                 ants_l.directions])
            last_positions = np.array([ants_l.
    historic_path[
                                       i, ants_l.
    age[i], :]
                                      for i in
    range
                                      (ants_l.
    directions.shape[0])])
            comm.send((pherom, food_counter,
    snapshop_taken,
                       ant_data, last_positions),
    dest=0)

    if rank == 0:
        while True:
            for event in pg.event.get():
                if event.type == pg.QUIT:
                    pg.quit()
                    comm.Abort(0)
                    sys.exit()

            deb = time.time()
            age_l = []
            loaded_l = []
            direction_l = []
            last_positions = []
            food_counter = 0

            for source_rank in range(1, size):

                pherom, food_counter_l,
    snapshop_taken, ant_data_l,
                last_positions_l = comm.recv(source=
    source_rank)

                age_l.append(ant_data_l[0])
                loaded_l.append(ant_data_l[1])
                direction_l.append(ant_data_l[2])
                last_positions.append(
    last_positions_l)

                food_counter += food_counter_l

                if food_counter == 1 and not
    snapshop_taken:
                    pg.image.save(screen, "
    MyFirstFood3_2_2.png")
                    snapshop_taken = True

            age_l = np.concatenate(age_l)
            loaded_l = np.concatenate(loaded_l)
            direction_l = np.concatenate(direction_l
    )
            last_positions = np.concatenate(
    last_positions, axis=0)

            ants.age = age_l
            ants.is_loaded = loaded_l
            ants.directions = direction_l

            for i, last_position in enumerate(
    last_positions):
                ants.historic_path[i, ants.age[i],
    :] = last_position

            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}", flush=True)
            deb = time.time()
```

Le code est conçu pour exécuter une simulation parallèle de fourmis utilisant MPI (Message Passing Interface) pour la communication entre les processus. Voici comment le parallélisme est mis en œuvre :

1) **Initialisation de MPI :** Le code commence par initialiser MPI en créant un communicateur pour tous les processus (`comm`), en récupérant le rang de chaque processus (`rank`), et en déterminant le nombre total de processus (`size`).
2) **Configuration de la simulation :** Le code utilise Pygame pour créer une interface graphique et définir les paramètres de la simulation, tels que la taille du labyrinthe, le nombre de fourmis, la durée de vie maximale des fourmis, les positions initiales de la nourriture et du nid, ainsi que les paramètres des phéromones.
3) **Répartition de la charge de travail :** Le nombre total de fourmis est réparti de manière équitable entre les processus MPI. Chaque processus est responsable de gérer un sous-ensemble des fourmis dans la simulation.
4) **Boucle principale de la simulation :**
   - Chaque processus, à l'exception du processus zéro, exécute une boucle infinie pour simuler le comportement des fourmis qu'il contrôle.
   - Les fourmis explorent le labyrinthe, collectent de la nourriture, retournent au nid, et déposent des phéromones en fonction de leur comportement.
   - Les processus échangent périodiquement des informations sur l'état des fourmis et des phéromones avec le processus zéro.

5) **Communication entre les processus :**

- Les processus MPI communiquent en envoyant et en recevant des données via MPI. Le processus zéro joue un rôle central dans la coordination des échanges d'informations.
- Les processus esclaves envoient des mises à jour sur l'état des fourmis et des phéromones au processus zéro.
- Le processus zéro agrège les données provenant de tous les processus esclaves, met à jour l'état global de la simulation, puis diffuse les mises à jour à tous les processus.

6) **Affichage graphique :**

- Le processus zéro gère l'affichage graphique de la simulation à l'aide de Pygame.
- Il récupère les données sur l'état des fourmis et des phéromones à partir des autres processus et met à jour l'interface utilisateur graphique en conséquence.

En résumé, chaque processus MPI est responsable d'une partie de la simulation et communique avec les autres processus pour coordonner l'ensemble de la simulation. Le processus zéro agit comme un point central de contrôle et de coordination pour la simulation parallèle.

Le code précédent a été testé avec les deux configurations proposées ci-dessus (**25x25** et **50x50**) en utilisant pour cela 3 processus, c'est-à-dire que le nombre total de fourmis serait divisé en 2 pour travailler en parallèle. Les images du premier repas dans chacun de ces cas sont visibles ci-dessous, en plus du résultat du FPS, qui a une valeur moyenne de **208.92**.

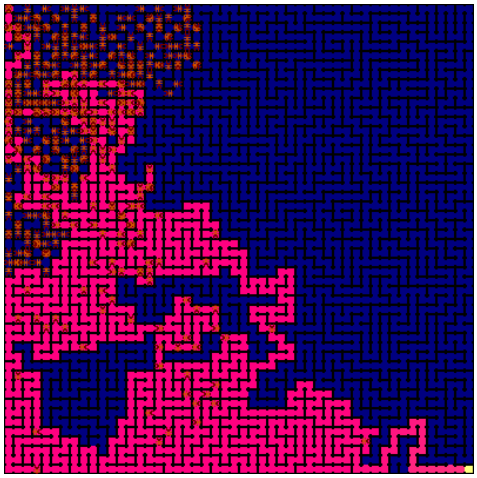

Fig. 8. Premier repas, parallélisation sur trois processeurs, dimensions 50x50

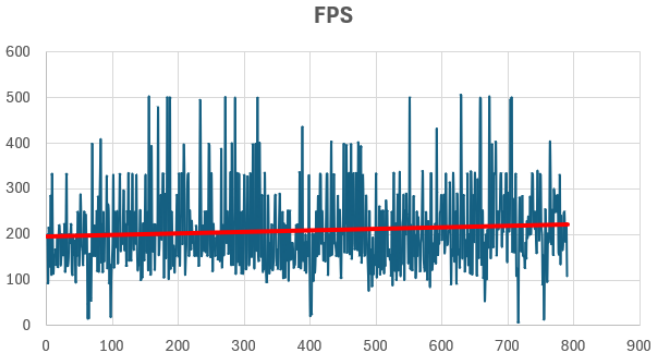

Fig. 9. Graphique de comportement FPS

Exactement la même chose a été faite, mais en utilisant 6 processus au lieu de 3, pour voir comment le comportement du système changeait à mesure que des processus étaient ajoutés. Cette configuration donnait une moyenne de **127.93** FPS

Fig. 7. Premier repas, parallélisation sur trois processeurs, dimensions 25x25

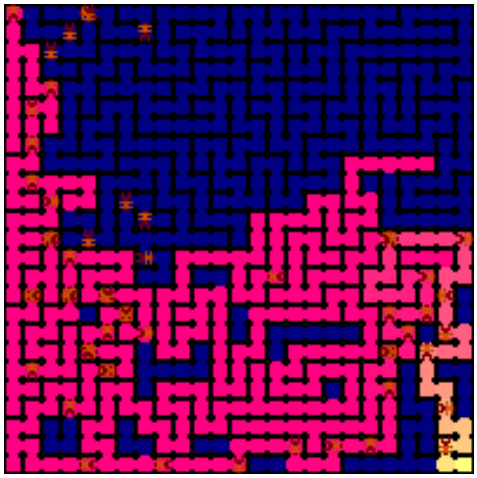

Fig. 10. Premier repas, parallélisation sur six processeurs, dimensions 25x25

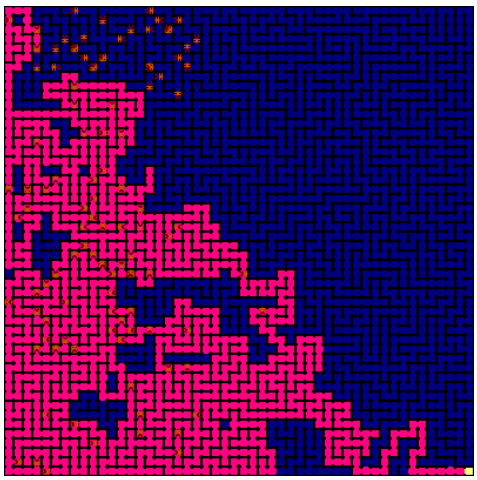

Fig. 11. Premier repas, parallélisation sur six processeurs, dimensions 50x50

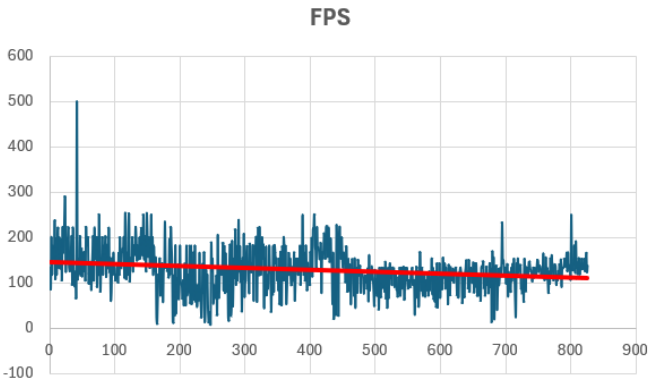

Fig. 12. Graphique de comportement FPS

La diminution des performances (FPS) lors de l'augmentation du nombre de cœurs peut être attribuée à des phénomènes tels que la surcharge de communication, la concurrence sur les ressources partagées ou les limitations de la nature du problème.

En augmentant le nombre de cœurs, la communication entre eux peut introduire une surcharge importante. Autrement dit, la fréquence et le volume des communications nécessaires à la synchronisation des processus augmentent, dépassant les avantages de la parallélisation.

En outre, l'évolutivité, qui fait référence à la capacité d'un programme à améliorer ses performances en augmentant le nombre de cœurs ou de processeurs, peut être limitée.

V. Troisième parallélisation: Réfléchir comment on pourrait partitionner le labyrinthe pour gérer les fourmis en parallèle sur un labyrinthe distribué entre les processus

Pour distribuer le labyrinthe entre les processus tout en gérant les fourmis en parallèle, une division cohérente du labyrinthe est nécessaire pour garantir que les fourmis puissent se déplacer entre les partitions sans rencontrer de problèmes. Voici une approche :

1) **Partitionnement du Labyrinthe** : Diviser le labyrinthe en plusieurs sous-labyrinthes, chaque sous-labyrinthe étant assigné à un processus. On doit assurer que chaque sous-labyrinthe soit une section cohérente du labyrinthe global et que les passages entre les sous-labyrinthes soient correctement gérés.
2) **Gestion des Passages entre les Partitions** : Définir des passages ou des connexions entre les sous-labyrinthes pour permettre le mouvement des fourmis. Les passages doivent être définis aux intersections ou aux jonctions entre les sous-labyrinthes pour assurer un mouvement fluide des fourmis.
3) **Communication entre les Processus** : Les processus doivent communiquer pour faciliter le déplacement des fourmis entre les partitions du labyrinthe. Lorsqu'une fourmi atteint le bord de sa partition, elle doit être transférée à la partition adjacente appropriée. Cela nécessite une coordination entre les processus pour assurer un mouvement fluide des fourmis.
4) **Synchronisation et Cohérence** : Synchroniser les mouvements des fourmis entre les processus pour éviter les conflits et assurer la cohérence de la simulation. Les processus doivent se coordonner pour garantir que les fourmis se déplacent de manière ordonnée et que les mises à jour du labyrinthe soient cohérentes.
5) **Optimisation des Échanges de Données** : Minimiser les échanges de données entre les processus en partageant des informations locales sur les limites des partitions ou en préchargeant les informations nécessaires pour les déplacements des fourmis.

En mettant en œuvre ces concepts, il est possible de distribuer efficacement le labyrinthe entre les processus tout en permettant une gestion parallèle des fourmis. Cela exploite pleinement les capacités de calcul parallèle et permet des simulations de labyrinthes à grande échelle.

Plusieurs algorithmes et techniques de parallélisation peuvent être utilisés pour distribuer le labyrinthe et gérer les fourmis en parallèle :

- **Décomposition de Domaine** : Diviser le labyrinthe en domaines plus petits, assignés à différents processus. Chaque processus est responsable d'une partie du labyrinthe, permettant un traitement parallèle.
- **Collectives MPI** : Utiliser des opérations collectives MPI telles que *scatter*, *gather* et *broadcast* pour échanger des données entre les processus. Cela simplifie la coordination des mouvements des fourmis et des mises à jour du labyrinthe.

En combinant ces algorithmes et méthodes de parallélisation avec le partitionnement du labyrinthe et la gestion des fourmis décrits précédemment, il est possible de construire un système robuste et efficace pour simuler le déplacement des fourmis dans un labyrinthe distribué entre les processus.

Implémenter ces concepts nécessite une coordination et une synchronisation minutieuses entre les processus. Une manière dont nous pourrions les mettre en œuvre :

- Chaque processus, à l'exception du processus 0, est responsable de la gestion d'une partie du labyrinthe et d'une partie de la colonie de fourmis.
- Le processus 0 gère l'affichage et coordonne la communication entre les processus.
- Les processus communiquent pour échanger des informations sur les positions des fourmis, y compris leur position et leur direction, ainsi que les mises à jour du labyrinthe.
- Lorsque les fourmis atteignent le bord de leur partition, les processus coordonnent leur transfert vers la partition adjacente. Plus précisément :
  - Le processus actuel identifie les fourmis approchant du bord de la partition.
  - Il envoie un message au processus adjacent, signalant le transfert de la fourmi et incluant sa position et sa direction.
  - Le processus adjacent vérifie s'il peut accueillir la fourmi.
  - Si possible, il accepte la fourmi et met à jour ses structures de données locales.
- Les passages entre les partitions sont soigneusement gérés pour garantir que les fourmis peuvent naviguer entre les sous-labyrinthes sans heurts.
- Les mouvements de fourmis sont synchronisés entre les processus pour éviter les conflits et assurer la cohérence de la simulation.
- Les échanges de données entre les processus sont minimisés en partageant efficacement des informations sur les limites des partitions et en préchargeant les données nécessaires aux mouvements des fourmis.

VI. Réflexions induits par le projet

A. Quelles sont les parties du code parallélisable lorsqu'on partitionne uniquement les fourmis

Les parties du code qui pourraient être parallélisables en divisant uniquement les fourmis sont principalement celles où chaque fourmi effectue des actions indépendantes des autres. Par exemple:

- Explorer le labyrinthe : La fonction **explore** est l'endroit où les fourmis déchargées explorent le labyrinthe. Au sein de cette fonction, il existe des boucles et des calculs qui pourraient être parallélisés. Par exemple, le calcul des sorties possibles pour chaque fourmi et la mise à jour des positions pourraient être effectués en parallèle pour différentes fourmis.
- Mise à jour des positions et des directions : Dans la fonction **explore**, plusieurs opérations impliquent la mise à jour des positions et des directions des fourmis.
- Vieillissement des fourmis : La section où vieillissent les fourmis (**self.age[unloaded_ants] += 1**) pourrait être réalisée en parallèle, puisque chaque fourmi vieillit indépendamment.
- Retour à la colonie : La fonction **return_to_nest** où une fourmi arrive au nid et met à jour son état pourrait être parallélisable.
- Marquage des phéromones : La partie du code où les phéromones sont marquées (**[pheromones.mark(self.historic_path[i, self.age[i], :]) pour i in range(self.directions.shape[0])**) pourrait être parallélisable si les conditions de concurrence sont gérées correctement.

Il est important de noter que lors de l'introduction de la parallélisation, les problèmes potentiels de concurrence tels que les conditions de concurrence et la synchronisation entre les fourmis doivent être résolus. Par exemple, nous pourrions utiliser des outils comme des verrous pour gérer ces problèmes et assurer la cohérence des données partagées.

B. Gains obtenu (speed-up)

L'accélération est communément définie comme le rapport entre le temps d'exécution sur un seul processeur (série) et le temps d'exécution sur plusieurs processeurs (parallèle).

$$S(n) = \frac{T_{série}}{T_{parallèle}} \tag{1}$$

Dans le contexte des FPS, on peut considérer l'inverse du temps d'exécution comme une mesure analogue. Dans ce cas, puisque on mesure les FPS (images par seconde) et qu'un FPS plus élevé indique de meilleures performances, on peut utiliser une variante de la formule :

$$S(n) = \frac{FPS_{parallèle}}{FPS_{série}} \tag{2}$$

Les résultats SpeedUp pour les deux parallélisations réalisées dans ce projet sont visibles dans le tableau suivant.

Configuration	FPS Moyens	Speed Up
En série	102.53	
Parallélisation (2 processus)	191.73	1.87
Parallélisation (3 processus)	208.93	2.04
Parallélisation (6 processus)	127.93	1.25

La parallélisation initiale avec 2 processus montre une accélération significative, doublant presque le FPS par rapport à l'exécution en série. En augmentant le nombre de processus à 3, une augmentation supplémentaire du FPS et de l'accélération est observée. Cela suggère qu'il existe une certaine efficacité dans la parallélisation en répartissant la charge de travail sur davantage de processus. Cependant, lorsqu'on passe à 6 processus, le FPS diminue et l'accélération est réduite. Ce comportement peut être dû à plusieurs facteurs, tels que la surcharge de communication inter-processus ou les ressources disponibles limitées.

Cela suggère qu'il existe un point auquel l'ajout de plus de parallélisme n'apporte pas une amélioration proportionnelle des performances.

VII. Annexes

Ici vous pouvez voir le code complet de l'ensemble du projet.

GitHub Natalia: github.com.

GitHub Gianluca: github.com.