OS202 - Systèmes parallèles et distribués

Projet Python 2024: Algorithme de Simulation de Fourmis pour la Résolution de Labyrinthes

GALVÃO, Mateus

ODORISSIO PEREIRA, Vitor

Palaiseau

2024

TABLE DES MATIÈRES

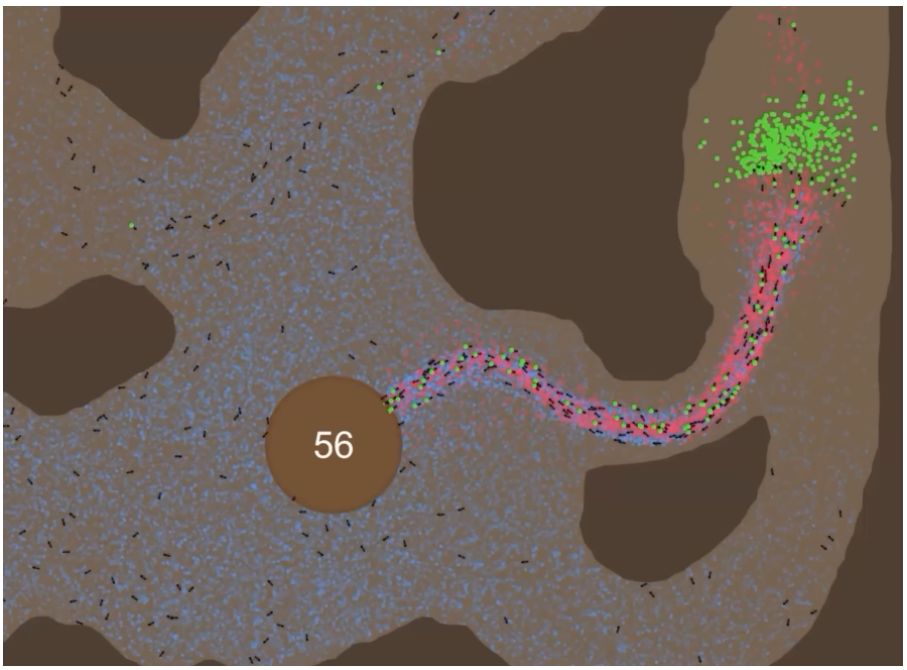
1 INTRODUCTION

1.1 Explication de l'algorithme

L'algorithme d'optimisation de colonie de fourmis (ACO) est une classe d'algorithmes d'optimisation inspirée du comportement de recherche de nourriture des vraies fourmis. Des "fourmis"artificielles trouvent des chemins optimaux en naviguant à travers un labyrinthe, qui représente toutes les solutions possibles. En se déplaçant à travers le labyrinthe, les fourmis laissent des pistes de phéromones le long de leurs chemins pour guider l'exploration des fourmis suivantes. La force et la durabilité de ces pistes de phéromones sont déterminées par la qualité du chemin trouvé, influençant le processus de prise de décision collective vers la route la plus efficace.

Essentiellement, cela repose sur l'idée qu'en prenant des chemins aléatoires, celui le plus emprunté est probablement le plus court. Par conséquent, en augmentant les chances que les chemins réussis soient choisis à nouveau, éventuellement, le seul chemin utilisé devient le plus court.

Illustration 1 – ACO implémenté dans une carte 2D



Source: (LAGUE, 2021)

1.2 Le stimulus pour paralléliser cet algorithme

Étant donné que la méthode repose sur de nombreux petits calculs répétitifs pour déterminer la probabilité de chaque fourmi choisissant un chemin particulier, elle est bien adaptée au traitement parallèle. Cela peut être réalisé en répartissant différentes tâches, ou même la même tâche, sur différents cœurs de processeur, ce qui peut potentiellement entraîner une accélération significative.

2 RÉFLEXIONS INDUITES PAR LE PROJET

2.1 **Q1:** Quelles parties du code sont parallélisables lorsque l'on partitionne uniquement les fourmis ?

Il existe de nombreuses façons différentes de paralléliser cet algorithme, même en considérant la parallélisation des colonies différentes pour chaque processus. Dans ce cas, les approches généralement possibles se réduisent à deux : **la parallélisation des tâches** ou **les multi-colonies**.

2.1.1 Parallélisme des tâches

Cette méthode consiste essentiellement à utiliser plusieurs processus pour exécuter davantage du même code séquentiel. Par conséquent, l'accélération n'est pas influencée par l'augmentation du nombre de processus. Dans le cas de notre problème, l'accélération provient de la séparation des tâches de visualisation et des tâches de calcul des fourmis entre 2 cœurs de processeurs différents.

- **Avantages:**
  - Coût de communication le plus bas possible.
  - Implémentation simple.
  - N'utilise que 2 cœurs.
- **Inconvénients :**
  - Ne bénéficie pas de l'utilisation de plus de 2 cœurs.

2.1.2 Multi-colonies

Les multi-colonies impliquent la même base que le parallélisme des tâches mais avec une communication entre les différents processus. En d'autres termes, l'apprentissage de l'algorithme se produit à travers plusieurs processus.

- **Avantages :**

  - L'accélération augmente avec le nombre de processus.

- **Inconvénients :**
  - L'accélération peut être entravée par le coût de la communication.
  - Implémentation plus complexe.

En détail, la possibilité d'atteindre des accélérations plus élevées s'accompagne d'une augmentation considérable de la complexité du code, car l'accélération provient de l'apprentissage partagé entre les processus, mais le partage lui-même coûte du temps.

Les variations possibles de cette méthode viennent du choix entre **Quoi**, **Comment** et **À quelle fréquence** partager les données du problème. En se concentrant sur notre programme, les différentes approches se résument à :

- **Quoi** partager : la matrice de phéromones entière ou le chemin le plus optimal trouvé.
- **Comment** partager : communications bloquantes ou non bloquantes ; quelles fonctions utiliser.
- **À quelle fréquence** partager : à chaque itération, dans quelques itérations, seulement après un événement, etc.

2.2 **Q2:** Quels gains ont été réalisés (accélération) ?

Pour mieux comprendre le résultat du speed up pour les deux stratégies de parallélisation, les deux codes ont été développés afin de pouvoir analyser les avantages et les inconvénients de chacun dans la pratique.

La Table 1 montre l'accélération obtenue avec la stratégie de parallélisation des tâches.

Temps séquentiel	Temps parallélisé	Speed up
13.9219 s	6.8777 s	2.0242

Table 1 – Accélération pour la parallélisation des tâches.

L'Illustration 2 montre l'accélération en utilisant la stratégie de multicolonies en faisant varier le nombre de processus de 2 à 9.

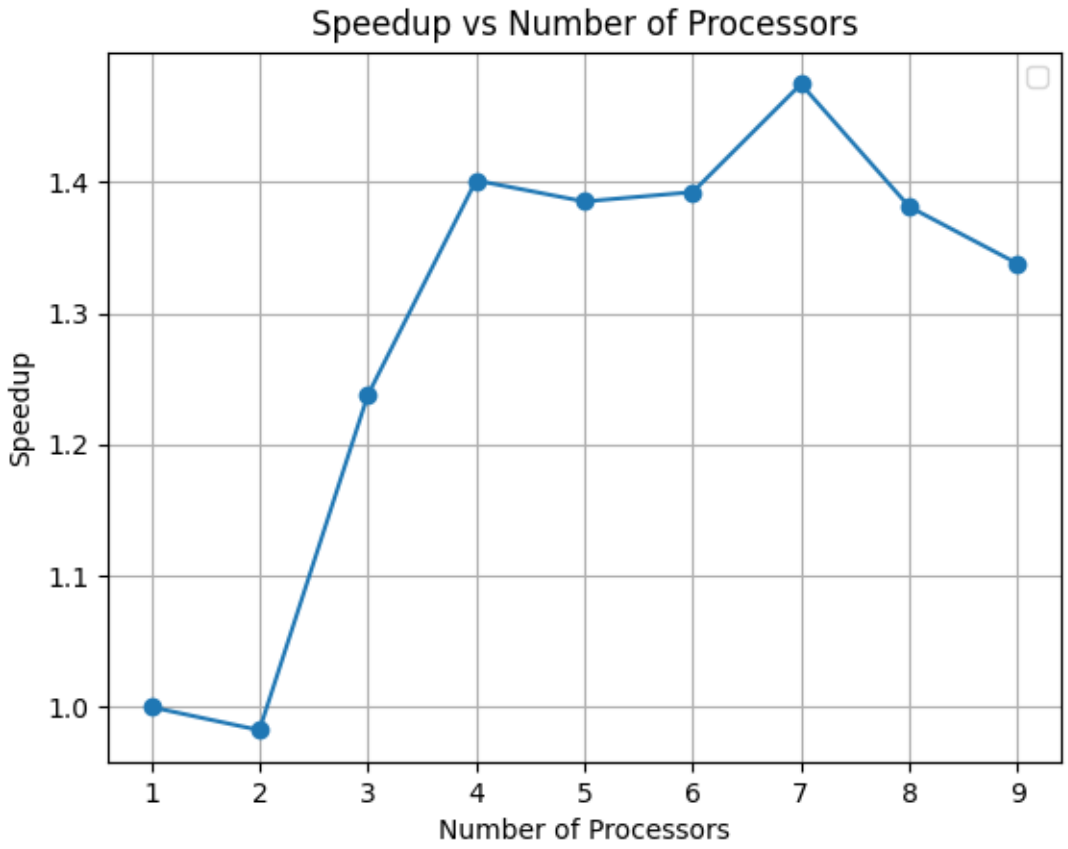


Illustration 2 – Accélération pour la parallélisation multi-colonies.

Comme on peut le voir, la stratégie de parallélisation des tâches a été très efficace, obtenant un accélération proche de 2, étant donné que son code est beaucoup plus simple et ne comporte pas une charge de communication élevée. Ainsi, ce code ne bénéficierait pas d'une augmentation de l'utilisation des processeurs, car son implémentation point à point ne garantirait pas une accélération similaire, chaque processus effectuant toujours le même calcul et envoyant ensuite les données au processus 0 pour l'affichage.

En ce qui concerne la parallélisation des multicolonies, nous pouvons observer une augmentation de l'accélération avec l'augmentation du nombre de processeurs jusqu'à un point où elle devient proche d'une valeur constante (environ 1.4), et tend à diminuer de plus en plus avec l'augmentation du nombre de processeurs, car la quantité de communication entre les processus augmente également beaucoup. La communication constante entraîne

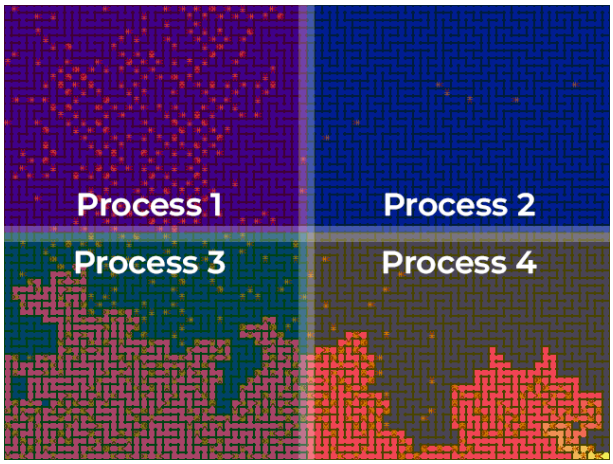
un temps d'attente supplémentaire à chaque fois qu'un processus attend une réponse d'un autre processus pendant chaque itération, ce qui nuit à l'accélération pour un grand nombre de processus utilisés.

2.3 **Q3:** Décrire vos réflexions sur la mise en œuvre du code en parallèle si l'on partitionne également le labyrinthe entre les différents processus.

Pour diviser le labyrinthe, il est possible de le diviser simplement en parties symétriques, mais cela conduirait à un terrible équilibrage de la charge, car le nombre de calculs est proportionnel au nombre de fourmis, et elles ne se répartissent pas de manière homogène dans tout le labyrinthe, comme on peut le voir sur l'Illustration 3.

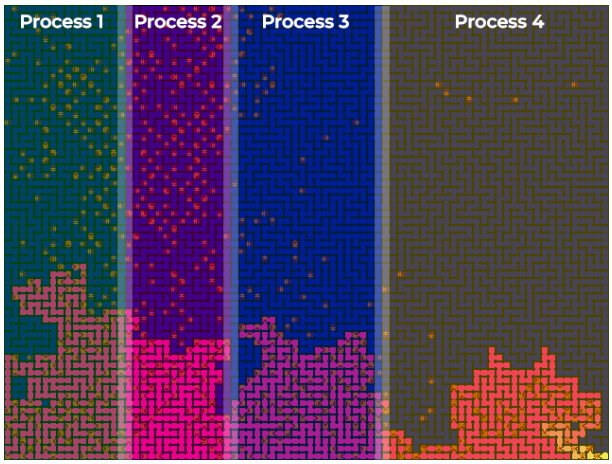
Une autre considération concerne les cellules frontalières, qui doivent chevaucher les différentes partitions afin que les fourmis puissent décider correctement où aller, sans laisser aucune possibilité derrière elles.

Illustration 3 – Partition symétrique



Source : Élaboré par les auteurs.

Illustration 4 – Partition équilibrée



Source : Élaboré par les auteurs.

Par conséquent, pour garantir un équilibrage correct de la charge, une structure de parallélisation maître-esclave est nécessaire car la division du labyrinthe doit s'adapter dynamiquement à la répartition des fourmis, comme illustré sur l'Illustration 4.

Ce type de distribution de processus est très courant dans les programmes de simulation de physique des particules, car la vérification des collisions est beaucoup plus efficace lors de la division de différents groupes d'objets en fonction de leur proximité, plutôt que de vérifier chaque objet par rapport à tous les autres.

Dans ces types de distributions, un algorithme très efficace est la carte arborescente, qui correspondrait très bien aux besoins de ce problème pour diviser les fourmis.

Ainsi, le processus zéro serait le maître, s'occupant de la visualisation de Pygame, de la division de la carte arborescente des fourmis, puis envoyant aux processus esclaves les fourmis que chacun traiterait.

RÉFÉRENCES

LAGUE, S. *Coding Adventure: Ant and Slime Simulations*. 2021. Disponível em: <https://www.youtube.com/watch?v=X-iSQQgOd1A>. 2