OS202 - Projet : les fourmis

Thomas TEIXEIRA, Maëlys MORO

Mars 2024

Table des matières

1 Séparation de l'affichage et de la gestion des fourmis

Tout d'abord nous voulons utiliser deux processus, le premier (rank 0) s'occupe de l'affichage, et le second (rank 1) gère l'évolution des fourmis, des pheromones et la quantité de nourriture présente dans la fourmilière. Pour cela, dans la boucle while, nous séparons les éléments du main gérant l'affichage et ceux gérant les calculs. Par simplicité, toutes les définitions d'objets en début de main sont effectués par les deux processus. Quelques autres changements dans le code sont nécessaires. Par exemple, l'initialisation de pygame ou l'affichage du labyrinthe qui doivent être effectués par le processus 0.

Pour communiquer entre les deux processus, nous rencontrons un problème: il est nécessaire d'échanger des informations sur des objets (ants de classe Colony et pherom de classe Pheromon). Pour pouvoir envoyer directement des objets entre les deux processus nous utilisons la fonction pickle de la classe MPI qui permet de sérialiser (dumps) ou désérialiser (loads) des objets python pour récupérer le flux de données. Une fois sérialisés, les objets ants et pherom peuvent être envoyés grâce à la fonction comm.send.

Pour estimer le speed up nous mesurons le temps mis pour que la quantité de nourriture dans la fourmilière dépasse 1000 unités (chiffre choisi arbitrairement). Ce temps est de 24,8 secondes pour l'algorihtme séquentiel. Une fois parallélisé, ce temps est de 21,0 secondes. Nous obtenons donc un speed-up de :

$$S(n=2) \;=\; \frac{Temps\ séquentiel}{Temps\ parallélisé} \;=\; \frac{24{,}8}{21{,}0} \;=\; 1{,}4$$

Nous obtenons un speed up inférieur à 2. Cela peut s'expliquer par le fait que les deux tâches

(affichage et calculs) ne nécessitent pas le même temps d'exécution. Les processus doivent donc s'attendre à chaque itération de la boucle while. Cela explique que l'accélération n'est pas multipliée par 2.

2 Partitionnement de la gestion des fourmis entre plusieurs processus

Pour optimiser un peu plus le code et diminuer son temps d'exécution nous pouvons paralléliser les calculs sur de nouveaux processus. Nous utiliserons toujours le processus 0 pour l'affichage mais nous utiliserons n-1 autres processus pour les calculs sur l'évolution des fourmis et des phéromones. Chacun des n-1 processus se verra attribuer une partie des fourmis. Pour répartir les fourmis sur les n-1 processus, nous créeons n-1 classes Colony avec un nombre de (nb_initial de fourmis / n-1) fourmis dans chacune des colonies. Chaque processus traite ensuite séparement l'avancement de ses fourmis et calcule la quantité de nourriture rapportée par sa colonie. Cependant il faut également traiter l'évaporation des phéromones. Chaque processus va alors mettre à jour sa grille de phéromones en prenant en compte la diffusion des phéromones provenant des fourmis chargées de nourriture et l'évaporation. Ensuite, toutes les données sont envoyées au processus 0. Le processus 0 rassemble ces données ainsi :

- le food_counter est mis à jour en effectuant la somme des quantités de nourriture renvoyées par les n-1 processus
- le tableau des phéromones est mis à jour en prenant, case par case, le maximum des n-1 cases des grilles calculées par les n-1 processus
- l'affichage est mis à jour en récupérant le tableau des positions actuelles des fourmis de chaque processus, et en les concatenant verticalement pour récupérer la position de l'ensemble des fourmis.

Le processus 0 renvoie ensuite le tableau des phéromones aux n-1 autres processus car ils en auront besoin à l'itération suivante.

Nous obtenons les speed-up suivants :

Nombre de processus	Speed up
2	1.48
4	1.43
6	1.20

Pour ce calcul du speed-up, nous avons réalisé les mesures différement. En effet, nous avons remarqué qu'il n'était pas forcément précis de mesurer le temps pour atteindre le score de 1000 nourritures, car le temps mis par les fourmis pour trouver la permière fois la sortie du labyrinthe était aléatoire et pouvait causer de gros écarts lors du calcul du speed-up. Pour résoudre ce soucis, nous avons décidé de diviser chaque résultat par le nombre de passages dans la boucle while, afin d'obtenir le temps moyen d'exécution d'une boucle.

Cependant, les résultats sont plutôt étonnants. En effet, on aurait dû s'attendre à obtenir un speed-up qui augmente avec le nombre de coeur, mais ce n'est pas le cas ici. Nous pensons que le

problème vient du fait que les communications entre les processus prennent trop de temps, ce qui créé un fort écart lorsque qu'on utilise plusieurs processus.

Nous avons donc essayé de réduire au maximum les envois de données (lors d'un passage dans la boucle while, il n'y avait que une communication aller-retour entre le processus n et le processus 0). Mais les résultats ne sont pas plus appréciables.

Bien que nous n'avons pas réussi à résoudre ce soucis, nous imaginons qu'il faudrait faire en sorte d'améliorer les communications pour le rank 0, qui recoit n-1 et envoie n-1 messages par passage dans la boucle while (une idée pourrait être d'utiliser un broadcast au lieu d'envoyer un message à chaque processus par exemple).

3 Partitionnement du labyrinthe entre plusieurs processus

Nous aurions pu partitionner le labyrinthe plutôt que de partitionner les fourmis. Nous pourrions imaginer partitionner le labyrinthe en n-1 sections (nous utiliserions toujours n processus avec le processus 0 qui gère l'affichage). Les fourmis seraient donc distribuées sur les différents processus en fonction de leur position dans le labyrinthe. Ainsi il faudrait séparer le tableau historic_path en fonction de la position des fourmis à chaque itération. Nous rencontrons toutefois un problème pour cette technique car les fonctions explore et advance de la classe Colony sont faites pour calculer l'avancement de toutes les fourmis d'un coup.

Chaque processus pourra mettre à jour la présence des phéromones dans sa fraction de labyrinthe. La difficulté ici sera le calcul pour les zones de frontières entre différentes portions du labyrinthe. En effet, une case du labyrinthe peut recevoir des phéromones d'une fourmi se trouvant dans une autre portion du labyrinthe.

Cette méthode paraît compliquée à implémenter en ne serait sûrement pas très efficace car elle implique de nombreuses communications entre les processus. De plus la population de fourmis n'est pas homogène sur le labyrinthe, donc pour chaque itération les processus ne mettrons pas le même temps de calcul, ils devront s'attendre les uns les autres ce qui diminuera les performances de l'algorithme.