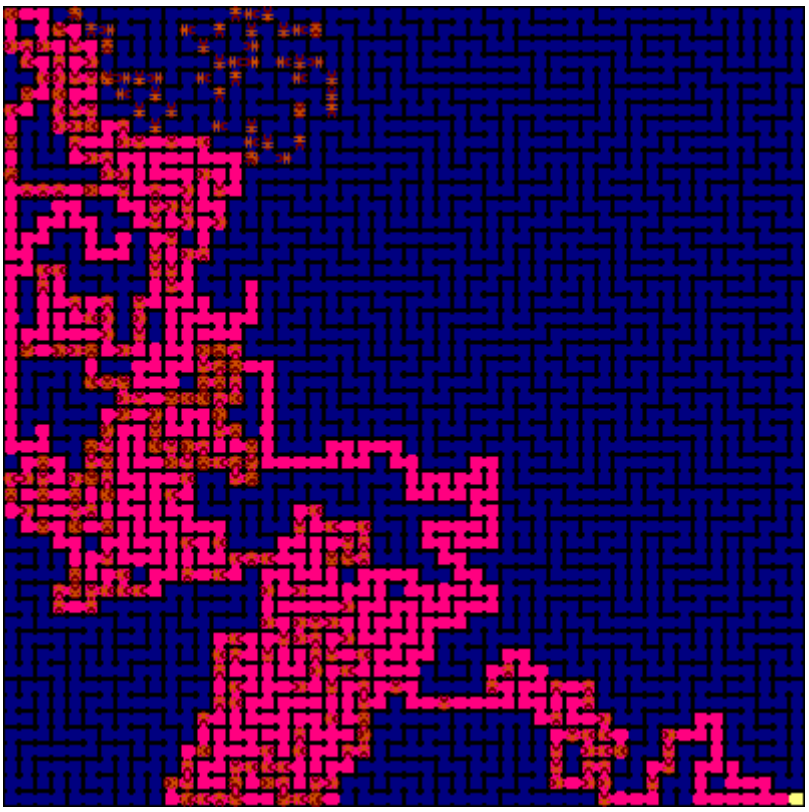
Quentin LORIAUX
Samy VINCENT

Projet Fourmis

OS202



Analyse du problème et solution proposée :

Ce problème semble à la fois CPU bound et Memory bound.

En effet, d'un côté, l'algorithme implique beaucoup de calculs, notamment pour mettre à jour les niveaux de phéromones et pour choisir les chemins en fonction de ces niveaux (ants.advance et pheromones.do_evaporation). Si le nombre de fourmis ou le nombre d'intersections dans le labyrinthe est très grand, l'algorithme peut être limité par le CPU.

D'un autre côté, l'algorithme nécessite également de stocker le labyrinthe, qui peut être très grand selon les paramètres choisis, et chaque fourmi stocke son historique de chemin.

Au final, on retient qu'on peut améliorer les performances en parallélisant les calculs, mais dans notre implémentation, chaque processus a besoin de sa propre copie du labyrinthe, ce qui augmente les exigences de mémoire.

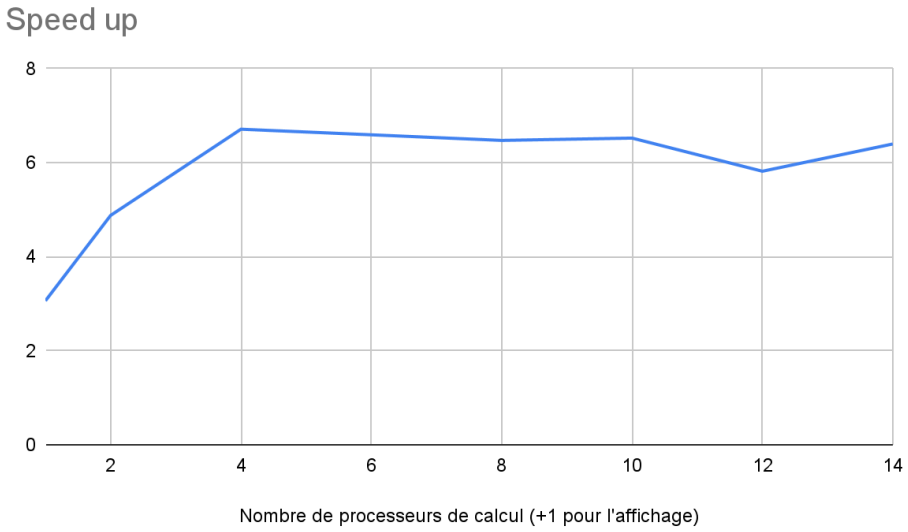
On commence par séparer l'affichage (gui.py) et les calculs (grid.py). Exécuter cette commande : mpirun -np 1 python3 gui.py : -np 1 python3 grid.py

On remarque que les fourmis sont indépendantes dans leurs déplacements, ce qui fait que le problème semble *embarrassingly parallel*. En effet, il suffit de diviser le nombre total de fourmis par le nombre de processus, et d'effectuer les calculs de chaque groupe de fourmis

sur chaque processus. Cependant, les fourmis doivent tenir compte de tous les phéromones, y compris ceux déposés dans les autres processus. Il faut donc rassembler les phéromones locales en une classe de phéromones globale pour l'affichage et pour le calcul du déplacement des fourmis. Il faut cependant n'actualiser que les phéromones locales, sinon chaque itération amplifierait les phéromones autant de fois qu'il y a de processus.

Malgré cela, la charge de travail de chaque processeur est bien équilibrée car il y a toujours autant de fourmis sur chaque processus. Ce point est important car il est nécessaire que tous les processus aient terminé leurs calculs pour passer à l'itération suivante, du fait de la nécessité de synchroniser les phéromones.

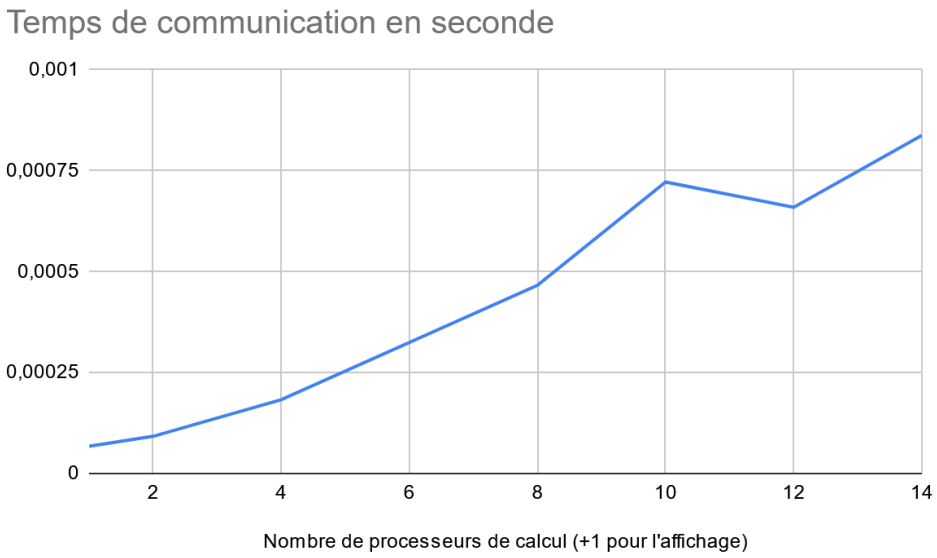
Analyse des performances :
Nous avons effectué les tests avec comme paramètres de labyrinthe: 25x25, durée de vie : 500 et (alpha, beta) = (0.9 , 0.99).
On compare le temps de calcul initial d'une itération (8,5 ms) à la moyenne des temps de calcul pour chaque processeur après parallélisation, en comptant le temps de communication.



On observe un plateau de speedup à partir de 4 processeurs et même une baisse de speedup à 12. A partir de 4 processeurs, le gain en performances est négligeable car on réduit de manière moins conséquente le nombre de fourmis par processus et cela ne compense pas l'augmentation du nombre de communications.

Notre implémentation est à granularité relativement fine car nous ne communiquons l'ensemble des données qu'en 3 étapes au bout de chaque itération. Nous aurions pu le faire en 2 en envoyant les phéromones avec les fourmis et food_counter mais utiliser mpi_reduce sur les phéromones nous a semblé plus optimal.

Le temps de communication est négligeable (relativement au temps de calcul) lorsqu'on utilise peu de processeurs, mais commence à devenir plus significatif en augmentant le nombre de processeurs (et donc de communications). Pour voir l'effet du temps de communication, on fixe le nombre de fourmis par processeur (ici, 25) et on augmente le nombre de processeurs. On multiplie par 12 le temps de communication entre l'utilisation d'un processeur et l'utilisation de 14 processeurs.



Nous avons essayé d'exécuter le programme avec un grand labyrinthe 200x200 et le programme est bien plus lent, mais toujours plus rapide lorsqu'on augmente le nombre de processeurs de 1 à 5 (SpeedUp = 4) et le temps de communication est multiplié par 2. Le programme semble donc plus CPU Bound que Memory Bound.

Idées de parallélisation de la grille :

Le problème de notre implémentation pour paralléliser la grille (et donc l'actualisation des phéromones), c'est que les fourmis sur un processus peuvent être à n'importe quel endroit de la grille, et donc dans une zone qui n'était pas traitée par le processus au départ.

Il faudrait donc faire autrement, cette fois-ci en créant sur chaque processus une colonie de taille variable. Ce ne seraient plus les phéromones qui seraient échangées entre les processus, mais les fourmis. Le problème devient similaire à celui du lifegame.

Prenons l'exemple de 4 processus coupant la grille en 4 rectangles (1 pour chaque coin).
Dans un processus worker :
Gains en performance :

- Les fourmis ne se déplacent qu'en considérant la sous grille.

- Lorsqu'une fourmi veut quitter la sous-grille pour aller dans une autre. Pour limiter les communications, on peut créer un tableau contenant les fourmis qui quittent la grille avec leur destination et leur direction.
- Lorsqu'une fourmi meurt, elle est envoyée sur le processus coin haut/gauche de la même manière. Cela crée cependant un déséquilibre car le processus coin haut/gauche sera plus sollicité (d'autant plus que c'est le point de départ et d'arrivée des fourmis).
- Il faut envoyer les fourmis locales au processus d'affichage, pas besoin de les renvoyer.
- Le calcul des phéromones ne se fait que sur la sous grille et ne dépend pas des autres grilles.
- A la fin d'une itération, il faut envoyer au processus d'affichage le tableau local de phéromones qu'il faudra réassembler (mais pas besoin de le renvoyer).

Le processus d'affichage:

- reçoit d'un coup avec gather les fourmis locales, les tableaux de fourmis en transit/mortes, et les phéromones,
- reconstruit le tableau de phéromones complet,
- affiche les phéromones puis la grille puis les fourmis,
- trie les fourmis en transit par destination,
- envoie d'un bloc les fourmis arrivant à chaque sous grille pour éviter trop de communications.

Comme on ne peut avancer que d'itération en itération, on ne peut pas adopter de stratégie maître esclave simplement ici, et le défaut a priori de la parallélisation de la grille sera le load balancing puisque la sous-grille avec la fourmilière aura plus de calcul à réaliser que que les autres.