OS 202

Rapport de projet

Julien SEGONNE

8 mars 2024

Table des matières

Introduction

Voici le rapport du projet de OS 202 sur la parallélisation d'un algorithme en essaim : l'optimisation par colonie de fourmis.

L'algorithme représente des fourmis cherchant le chemin le plus court de leur fourmilière à la nourriture. Il génère un labyrinthe, dont nous choisissons les dimensions, dans lequel il fera évoluer une colonie de fourmis guidées par des phéromones déposés par les fourmis ayant trouvé de la nourriture.

Le but de ce projet est de paralléliser le code avec la bibliothèque mpi4py pour séparer les tâches et étudier les configurations les plus efficaces (diviser le labyrinthe, diviser les colonies de fourmis, séparer les tâches,...).

Tous les résultats de speedup seront présentés de la même manière : le nombre de processus varie de 1 à 4 (lorsque cela est possible), la taille du labyrinthe est choisie parmi 10, 20 et 25 (pour simplifier le labyrinthe sera carré) et la condition d'arrêt (la quantité de nourriture rapportée) parmi 200, 1500 et 3000.

1 Première approche

1.1 Explication

Dans un premier temps, nous allons séparer l'affichage, qui sera géré par le processus 0, et la gestion des fourmis et des phéromones, gérés par le processus 1. Nous n'utiliserons donc ici que 2 processus.

Pour mettre en oeuvre cette approche, nous modifierons `maze.py` et ants.py que nous appellerons `ants_1.py`.

1.1.1 `maze.py`

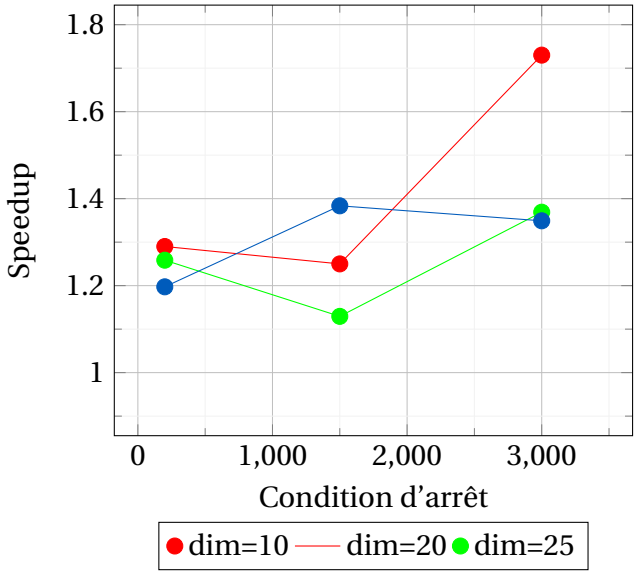
Dans ce code, nous allons simplement stipuler que, lors de l'initialisation du labyrinthe, le chargement de celui-ci ne se fait que sur le processus 0.

1.1.2 `ants_1.py`

Dans ce code, le processus 1 se charge uniquement de faire avancer la colonie de fourmis et d'actualiser la carte de phéromones. Le processus 0 gère l'affichage mais pour ce faire, il a besoin des données de phéromones et de la colonie. Ainsi, le processus 1 envoie à chaque cycle ces données et le processus 0 les reçoit pour tout afficher.

Remarque : Les échanges de données ont été fait une première fois de manière un peu naïve avec les fonctions *send* et *recv* de MPI, une implémentation avec la fonction *Gatherv* aurait aussi pu être envisagée pour plus de lisibilité et d'organisation.

1.2 Résultats



1.3 Analyse

On constate dans un premier temps, qu'il y a bien une amélioration du temps d'exécution car tous les speedups sont supérieurs ou égaux à 1.
Dans un second temps, l'amélioration la plus flagrante demeure sur une dimension de labyrinthe petite (dim=10) où les speedups sont assez élevés.

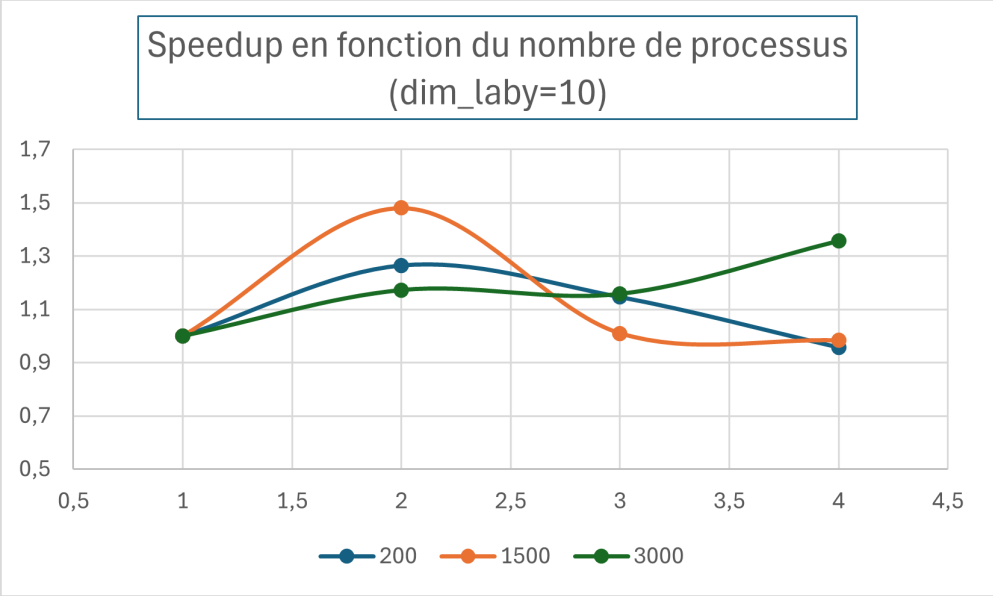
2 Seconde approche

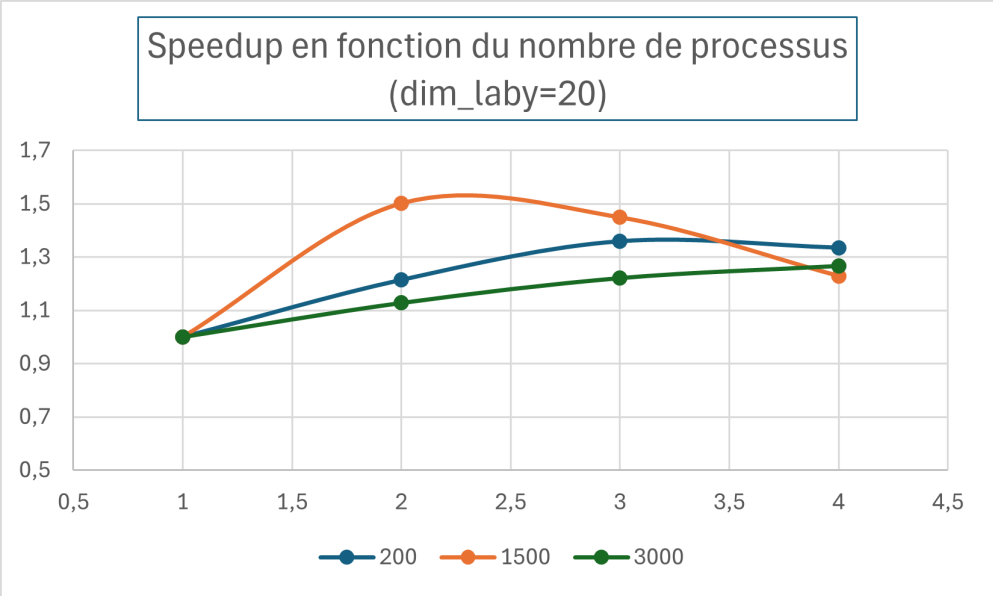
2.1 Explication

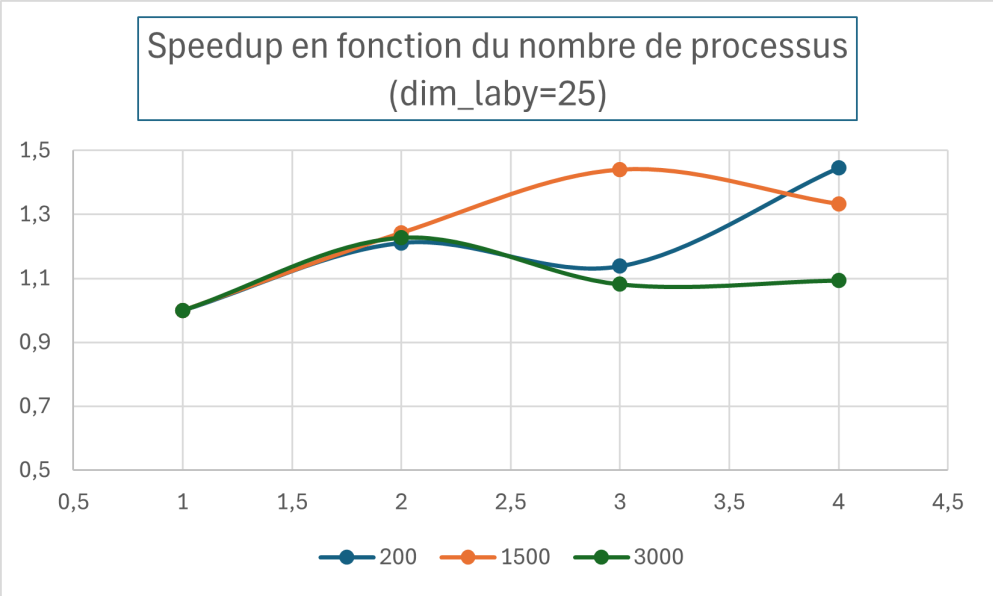
Cette fois, nous disposons de plusieurs processus (pour mon ordinateur, je n'ai que 4 processeurs maximum). La stratégie est différente : le processus 0 s'occupe toujours de l'affichage et les autres processus ont leur propre colonie de fourmis et leur propre carte de phéromones. De la même manière, chaque processus envoie ses données au processus 0 afin qu'il puisse afficher mais il faut trouver un moyen de mettre en commun les nbp-1 cartes de phéromones. pour ce faire, nous allons sélectionner pour chaque case le maximum de la valeur du phéromone parmi les nbp-1 cartes et l'inscrire dans la carte du processus 0 qui sera ensuite redistribuée aux autres processus afin qu'il puisse l'utiliser pour leur propre colonie et donc mutualiser les phéromones. On affiche ensuite bien sûr les phéromones via le processus 0. Tout ceci se fait à chaque cycle. Le nouveau fichier s'appelle `ants_2.py`.

Remarque : Même remarque précédemment.

2.2 Résultats







2.3 Analyse

Certaines valeurs semblent être erronées car on observe parfois des décrochages assez brusques (dim_laby=10, 1500 ou dim_laby=25, 3000), cependant la courbe pour dim_laby=20, 1500 est parfaitement exploitable : on observe une croissance du speedup jusqu'à 3 processus puis une légère décroissance pour 4 processus ce qui semble donc ne pas être optimal même si le speedup est très intéressant. On remarque aussi que les speedups pour une dim_laby=20 sont en général plus élevés que pour dim_laby=25. Cela peut être dû au fait qu'un plus grand labyrinthe se repose beaucoup sur le temps que met la première fourmi à trouver la nourriture et donc se repose sur l'aléatoire. Ce premier laps de temps, plus long pour un grand labyrinthe, déterminera grandement le temps d'exécution du code.

3 Réflexion sur la partition du labyrinthe

Une autre manière de paralléliser serait de partitionner le labyrinthe pour le répartir entre les processus. La colonie de fourmis et les phéromones seraient alors partagés entre tous les processus. Cela pourrait simplifier l'idée du code : chaque processus n'à qu'à connaître les fourmis qui voyagent dans sa propre partie du labyrinthe. Il suffirait ensuite d'utiliser la méthode *advance* sur les fourmis en question et prévenir le processus concerné lorsqu'une fourmi entre sur une autre partie du labyrinthe. Cependant, on observe rapidement que les fourmis n'utilisent pas en même proportion toutes les parties du labyrinthe (surtout après la diffusion de phéromones, globalement les mêmes chemins sont empruntés), il y aurait donc des processus qui travailleraient beaucoup moins que d'autres et donc la parallélisation serait moins efficace. Toutefois, à grande échelle, cette idée pourrait peut être bien fonctionner car la phase d'exploration (dans toutes les directions) durerait beaucoup plus longtemps.