# Les fourmis

Laura Montagnier et Ianis Kelfoun

Mars 2024

**Résumé**

Ce document explique notre mise en place de parallélisme pour le calcul du trajet de fourmis dans un labyrinthe dans le cadre du cours OS202 de l'ENSTA Paris.

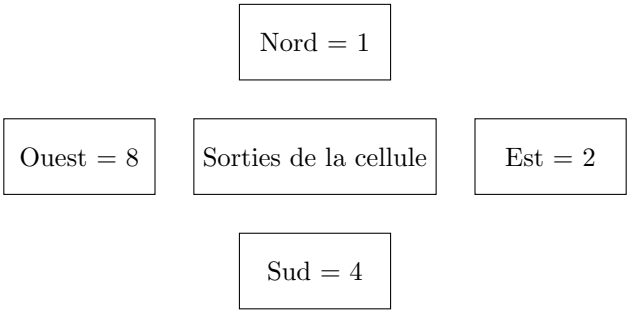
# 1 Le problème des fourmis

## 1.1 Contexte

Le problème s'inscrit dans un cadre de recherche d'**intelligence en essaim** : il est ici question de chercher le chemin le plus court de la fourmilière à une source de nourriture. Nous utiliserons un **labyrinthe où il n'existe qu'un seul chemin sans rebroussement entre la fourmilière et la nourriture**.
Cette modélisation a pour but de trouver un **plus court chemin** dans un labyrinthe.

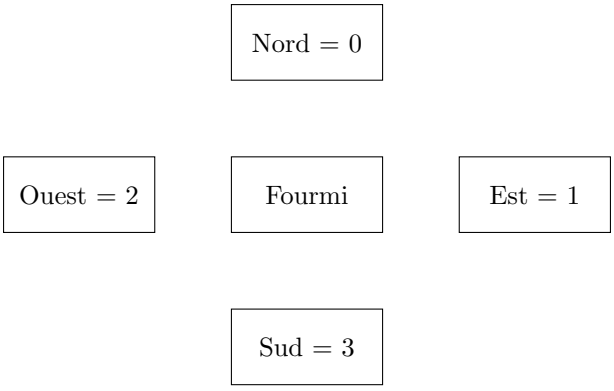
## 1.2 Le fichier maze.py

Ce fichier crée le labyrinthe. C'est un tableau en deux dimension où chaque case est définie par la somme des sorties existantes dans cette cellule (1 pour le Nord, 2 pour l'Est, 4 pour le Sud et 8 pour l'Ouest : ce sont des puissances de 2). Par exemple, une cellule où l'on peut sortir par le Nord et le Sud sera définie par 5.




## 1.3 Le fichier directions.py

Ce document assigne une direction "-1" initialement s'il n'y en a aucune. Ensuite, on assigne 0, 1, 2 ou 3 selon si la fourmi se dirige respectivement vers le nord, l'est, l'ouest ou le sud.



### 1.4 Le fichier pheromones.py

Il contient la classe pheromone, munie de ses différentes fonctions, notamment celle d'affichage.

### 1.5 Le fichier ants.py

C'est le fichier principal que nous n'allons pas modifier, qui va s'occuper de simuler une colonie de fourmis dans un labyrinthe. A l'état initial, ce fichier contient la classe Colony (on nous dit qu'on simule une colony d'un nombre bien défini de fourmis plutôt que de simuler individuellement les fourmis pour des raison de performance).
De plus, ce fichier contient un main qui gère à la fois les calculs et l'affichage des fourmis, du labyrinthe et des phéromones. C'est ce main que nous n'allons pas exécuter, et nous allons plutôt créer un nouveau main dans deux autres fichiers.

# 2 Séparation de l'affichage

Le processus zéro va s'occuper de l'affichage pendant que le processus un va s'occuper de calculer le trajet de la colonie de fourmis.

## 2.1 Première étape

On va subdiviser le *main* en deux fichiers python, un pour l'affichage, qu'on appelle *colonie_affichage*, et un pour calculer les trajets, le comportement des fourmis, si elles meurent, si elles sont chargées, etc... qu'on appelle *colonie_calculs*. On importe dans les deux le fichier *colonie.py* afin de pouvoir accéder à la classe *Colony*.
Dans *colonie.py* on trouve deux nouvelles classes : l'une qui s'appelle ColonieCalcul, et qui contient tout ce qui est relatif aux calculs du comportement des fourmis. L'autre, ColonieAffichage, est une classe utile pour l'affichage.
Nous remarquons que nous n'avions pas forcément à partager la classe *Colony* en deux classes, mais cela a été nécessaire quand nous n'arrivions pas autrement à empêcher le lancement d'une fenêtre d'affichage noire par *colonie_calculs*.

## 2.2 Ecriture du code

A cet état de l'avancement du projet, comme il n'y a que deux processus, nous avons utilisé le **Broadcast**, que nous remplacerons par des **Send** et **Recieve** plus tard (dans notre code présenté dans le Github, ces changements ont été faits).

On a séparé les éléments du *main* entre les deux fichiers. On exécute l'affichage avec le processus 0 et les calculs avec le processus 1. Par défaut, on laissera l'affichage être géré par le processus 0 tout au long du projet.

On exécute les deux fichiers avec les processus 0 et 1 grâce à la ligne de commande :

```
mpirun -n 1 python3 colonie_affichage.py : -n 1 python3 colonie_calculs.py
```

## 2.3 Vérification du bon fonctionnement du code

Nous avons remarqué dans le Github du professeur une image de référence : MyFirstFood_ref.png qui sert de point de repère lorsque la première fourmis trouve de la nourriture.
Afin de vérifier que notre parralélisation ne changeait pas la bonne exécution du programme, nous avons nous aussi défini une image de référence : MyFirstFood_ref_new.png. Elle a les dimensions de notre programme, au contraire de celle sur le Github du professeur.
Nous avons alors utilisé la commande : "diff MyFirstFood.png MyFirstFood_ref_new.png", qui a indiqué qu'il s'agissait bien des deux mêmes fichiers. Ainsi, notre parallélisation n'a apparemment rien changé à la bonne exécution du programme.

## 2.4 Calcul du speed-up

### 2.4.1 Calcul théorique

On s'intéresse aux vitesses d'exécutions du script afin de calculer le Speed-up. On peut se faire une idée des résultats théoriques qu'on devrait obtenir en mesurant le temps nécessaire au code séquentiel **ants.py** pour traiter l'affichage et la gestion des Fourmis. On obtient les résultat suivant :

| Tâche | Temps d'exécution (s) |
|---|---|
| Affichage | 3.1437408924102783 |
| Fourmis | 6.534234523773193 |
| Total | 9.677975416183472 |

On peut donc voir que séparer l'affichage et les fourmis devrait amener à un speed-up théorique d'environ 1,48, puisque, les deux tâches s'effectuant en parallèle, le temps d'exécution total serait aussi long que celui des Fourmis dans la version séquentielle.

Si l'on décide ensuite de partitionner les Fourmis, ce temps sera divisé par le nombre de processus que l'on aura à disposition (hormis le processus 0, responsable de l'affichage).

On aurait donc les speed-ups :

| Nombre de processus dédiés aux fourmis | Speed-up théorique |
|---|---|
| 1 | 1.48 |
| 2 | 2.97 |
| 3 | 3.08 |

### 2.4.2 Résultat expérimental

On fait le choix, pour le calcul du speed-up, de s'intéresser au temps qu'il faut pour qu'une fourmi rapporte la première unité de nourriture à sa colonie.

Pour ce faire, on s'appuie sur le principe de la variable SNAPSHOT_TAKEN qui passe de **False** à **True** quand FOOD_COUNTER prend la valeur 1, et nous a déjà servi pour obtenir l'image : **MyFirstFood.png**.

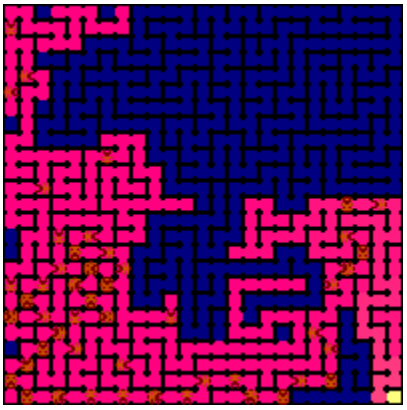
On note l'instant DÉBUT au départ de la boucle principale, puis l'instant FIN au changement de la variable SNAPSHOT_TAKEN. Pour les versions parallélisées du code, il convient simplement de procéder de même dans le processus qui gère le compteur de nourriture, donc les calculs et non l'affichage

Une simple soustraction FIN − DÉBUT fournit les résultats expérimentaux suivants :

| Code | Temps d'exécution (s) |
|---|---|
| Séquentiel | 9.501397609710693 |
| Parallèle - Affichage et Fourmis | 7.356470108032227 |
| Parallèle - Fourmis partitionnées | Pas de résultat obtenu |

On a donc un **Speed-up d'environ 1.29** quand on parallélise en laissant le processus de rang 0 gérer l'affichage, et celui de rang 1 gérer les Fourmis.

On a un speed-up légèrement plus faible que le speed-up théorique pour un seul coeur dédié aux fourmis, qui était de **1.48**, mais le même ordre de grandeur.



# 3 Parraléllisation des fourmis

## 3.1 Petit (ou gros) problème

A cause de considérations de limite de temps, de compétences, et de bugs peu explicites, nous n'avons pas réussi à finir la parraléllisation de nos fourmis. Néanmoins, nous pouvons expliquer point par point notre démarche, qui aurait dû fonctionner.

## 3.2 Le principe de notre mise en oeuvre

Nous voulions exécuter les deux fichiers :

1. **colonie_calculs_question2.py**
2. **colonie_affichage_question2.py**

grâce à la commande :

```
mpirun -n 1 python3 colonie_affichage_question2.py : -n 3 python3 colonie_calculs_question2.py
```

Nous avons en effet 4 processus, 1 sur l'affichage, et 3 sur les fourmis, ce qui fait 52 fourmis par processus.

- Nous créons des "petites colonies" dans le fichier colonie_calcul.
- Dans le fichier colonie_affichage, exécuté par le processus 0, nous transformons les "Receive" en "Gatherv", car nous recevons des informations des 3 processus sans en envoyer.
- Une fois les trois petites colonies gérées par les trois processus et communiquant avec le processus zéro, il ne restera que 2 petits problèmes à régler.
- Le premier problème est celui de la seed, qui est initialisée de la même manière pour la i ème fourmi du premier, du deuxième et du troisième processeur. Il aurait fallu l'initialiser comme la i+fourmis précédentes seed.
- Pour les phéromones, c'est un problème que nous n'avons pas eu le temps d'explorer et de règler.

# 4 Division du labyrinthe

Notre labyrinthe actuel semble entièrement être contenu dans le cache L1, on y accède très facilement. Ainsi, si le labyrinthe était très très grand, il serait sans doute pertinent de le subdiviser.

On pourrait adopter une approche **eulérienne** plutôt que **lagrangienne**. Chaque petite sous-section du labyrinthe serait gérée par un processeur : il aurait à sa charge les fourmis présentes, et l'état des phéromones.

Néanmoins, les fourmis se déplacent **vite**. Il faudrait que le labyrinthe soit vraiment très grand, afin de ne pas devoir constamment faire transiter une fourmi d'un processus à l'autre, ce qui coûterait très cher en termes d'accès mémoire.

On a de nombreux autres problèmes qui risquent de se créer : peut-être qu'en même temps, par exemple, deux cases du labyrinthe soient affichées en même temps mais à des instants d'exécution différents.

Le plus gros problème à notre sens est, qu'à un moment donné, les processus seront déséquilibrés. Les fourmis vont se mettre à suivre le même chemin. Par exemple, dans notre simulation, on n'avait presque aucunes fourmis en haut à droite. Des processus ne feraient donc rien du tout alors que d'autres, celui contenant le nid et la nourriture, croûleraient sous le nombre de fourmis.

Pour cette raison, si jamais quelqu'un voulait diviser le labyrinthe afin de paralléliser, nous lui conseillerons de choisir le système du **maître-esclave**.