Optimisation Par Colonie De Fourmi D'un Chemin Sur Un Paysage Fractal

Les algorithmes coopératifs

Le terme "*Intelligence en essaim*" a été introduit en 1989 par Beni et al. Cette classe d'algorithmes, souvent inspirés par le comportement des insectes sociaux, met en place une population d'agents simples interagissant et communicant indirectement avec leur environnement. Ces algorithmes constituent une classe d'algorithmes massivement parallèles pour résoudre une tâche donnée.

L'algorithme en essaim le plus connu est l'optimisation par colonies de fourmis (ACO) pour les problèmes combinatoires. Dans ce projet, on revient à l'inspiration originale des algorithmes ACO où une population d'agents simples (qui peuvent être vue comme imitant le comportement de fourmis réelles) résout efficacement le problème de fourragement (chercher le chemin le plus cours de la fourmilière à une source de nourriture).

Modèle simple d'ACO

Dans cette partie, nous allons décrire un modèle simple de colonies de fourmis qui permet de résoudre le problème de fourragement. Il a été montré pour cette algorithme qu'il converge de façon sur-linéaire par rapport au nombre de fourmis.

Description du modèle

On considère un terrain représenté par une grille cartésienne 2D, dont chaque cellule comporte une valeur donnant l'unité de temps pour la traverser (valeur comprise entre zéro et un). On considère un ensemble de m fourmis artificielles qui évoluent sur ces cellules et sur lesquelles elles mettent à jour des "taux de phéromones". Chaque cellule s de la grille stocke en plus deux valeurs réelles, correspondant à deux types de phéromones : un phéromone $V_1(s)$ permettant d'indiquer aux autres fourmis le chemin d'exploration effectué, et un phéromone $V_2(s)$ permettant à la fourmi de pouvoir retourner au nid lorsqu'elle a trouvé de la nourriture.

Sur la grille, on trouve quatre types de cellules : une cellule correspondant à la fourmilière, une cellule correspondant à la source de nourriture (qui pourrait ne pas être unique), des cellules indésirables que le fourmis ne peuvent pas traverser, et le reste des cellules sont *libres*, c'est à dire explorables par les fourmis.

Une fourmi peut être dans deux états possibles : elle peut porter de la nourriture (état "chargée") ou elle peut ne rien porter (état "non chargée"). En évoluant sur la grille, l'état des fourmis peut changer selon les règles naturelles suivantes :

- si une fourmi arrive sur une cellule contenant de la nourriture, son état devient "chargée";
- Si une fourmi arrive à la fourmilière son état change à "non chargée".

- Quand une fourmi arrive à la fourmilière avec l'état chargée, un compteur d'"unités de nourriture" est incrémenté. Ce compteur servira d'indice de performance globale pour la population de fourmi choisie.

Décrivons la dynamique du modèle. Au départ :

- Le compteur d'unités de nourriture est mis à zéro;
- les m fourmis sont initialisés à des positions arbitraires (soit tous dans la fourmilière soit initialisées uniformément sur la grille)
- Toutes les fourmis sont non chargées;
- les phéromones sont tous mis à zéro sur la grille.

'A chaque pas de temps, une fourmi fait deux choses :

1. Elle met à jour les taux de phéronomes $V_1(s)$ et $V_2(s)$ de la cellule sur laquelle elle se trouve en utilisant celles des quatre cellules voisines (on note $N(s)$ les voisins de s). La mise à jour des taux de phéronomes requiert uniquement la connaissance du maximum et de la moyenne des cellules voisines : $\max_i(N(s)) \equiv \max_{s' \in N(s)} V_i(s')$ et $avg_i(N(s)) \equiv \frac{1}{4}\sum_{s' \in N(s)} V_i(s')$ où l'indice $i \in \{1, 2\}$ indique le type de phéronome considéré. Précisément, la mise à jour des phéronomes se fait selon les calculs suivants :

   $$V_1(s) \rightarrow \begin{cases} 1 & \text{si la source de nourriture est en } s \\ \alpha \max_1(N(s)) + (1-\alpha) avg_1(s) & \text{sinon} \end{cases}$$

   et

   $$V_2(s) \rightarrow \begin{cases} 1 & \text{si la fourmilière est en } s \\ \alpha \max_2(N(s)) + (1-\alpha) avg_2(s) & \text{sinon} \end{cases}$$

   où $0 \leq \alpha \leq 1$
2. Elle avance sur une de ses cellules voisines si elle n'a pas la valeur -1 :
   - avec une probabilité ε ($0 \leq \varepsilon \leq 1$) qu'on appelera *taux d'exploration*, elle avancera sur une cellule voisine choisie aléatoirement parmi ses quatre voisines;
   - avec une probabilité $1 - \varepsilon$, elle avancera sur la cellule ayant le taux de $V_1(s)$ le plus grand si elle n'est pas chargée, et sur la cellule ayant le taux de $V_2(s)$ le plus grand si elle est chargée.

Le paramêtre α est appelé paramêtre de bruit 3. la fourmi avancera dans le même pas de temps d'autant qu'elle n'a pas épuisé ses points de mouvement : à chaque pas de temps, la fourmi possède un taux de mouvement de 1 qu'elle doit dépenser en fonction de la valeur unité de temps du terrain qu'elle traverse (ce qui traduit la difficulté plus ou moins grande de traverser une zone). Sur la carte affichée à l'écran, plus une zone est claire, plus elle est difficile à traverser.

À chaque pas de temps, les phéronomes posés par les fourmis s'évaporent suivant un cœfficient d'évaporation β : $V_i(s) \rightarrow \beta V_i(s)$. La valeur β prend typiquement une valeur proche de un.

Ainsi, pour l'instanciation du modèle, les paramêtres suivants doivent être précisisés :

- **l'environnement** : c'est l'ensemble des cellules avec leur unité de temps, leur type (libre, indésirable, fourmilière ou nourriture)
- **Le nombre de fourmis** m
- L'initialisation de la position des fourmis : soit toutes au départ dans la fourmière soit réparties sur la grille uniformément;
- Les paramêtres de bruit α, d'évaporation β et d'exploration ε.

Le modèle décrit ci–dessus est constitué d'agents réactifs simples qui communiquent indirectement au travers de l'environnement. Contrairement aux modèles classiques, il n'est pas nécessaire ici de conserver le trajet d'une fourmi par rapport à sa fourmilière grâce à l'utilisation d'un deuxième phéronome, ce qui rend nos agents complètement réactifs.

Création de l'environnement

Notre simulation nécéssite l'utilisation d'une carte d'unités de temps traduisant la nature plus ou moins accidentée du terrain traversé par les fourmis (se traduisant par une valeur de temps que doit dépenser une fourmi pour rentrer dans une cellule). La génération d'une telle carte peut se faire soit de façon analytique (à l'aide d'une fonction) soit de façon stochastique en générant un plasma.

C'est la deuxième solution qui a été retenue pour ce projet. Un plasma est une carte générée de façon aléatoire où on s'assure que le gradient de valeurs entre deux points de la carte ne dépasse pas une certaine valeur d nommée déviation.

On considère une grille composée de $n \times n$ sous–grilles. Ces sous–grilles ont un nombre de cellules par direction de $ns = 2^k + 1$ cellules et se recouvrent avec les sous–grilles voisines à l'aide d'une rangée de cellule. L'algorithme est alors le suivant :

- On génère les coins de chaque sous–grille en générant à l'aide d'un générateur pseudo-aléatoire dépendant d'une graine et de la coordonnée du sommet à évaluer, le premier coin de la première sous–grille puis en générant les autres coins séquentiellement à partir de ce premier coin en calculant une déviation inférieure à $d \times ns$.
- Puis on divise récursivement chaque sous–grille en quatre sous–grilles égales en calculant la valeur des cellules se trouvant au milieu de chaque bord et la valeur de la cellule se trouvant au centre de la sous-grille, en calculant des déviations inférieures à $d \times \frac{ns}{2}$. Il faudra tout de même veiller à chaque instant aux bords, communs à chaque sous–grille

Une fois toutes les valeurs générées, on les normalise pour obtenir une carte de valeurs comprises entre zéro et un.

Mesure du temps passer par itération

Mesurer et noter le temps passé dans les différentes parties du code.

Vectorisation et parallélisation en mémoire partagée du code

Vectorisation

Dans un premier temps, on modifiera le code afin de "vectoriser" les algorithmes. Cela consiste à créer des tableaux contenant l'un les coordonnées de toutes les fourmis, le second leurs état, le troisième leurs graines, etc. Une fourmi ne sera ensuite repérée que par son indice dans ces tableaux.

Mesurer de nouveau le temps passé dans les différentes parties du code et comparer avec la version d'origine

Parallélisation en mémoire partagée

Paralléliser ensuite à l'aide d'OpenMP les boucles qui vous semblent adaptées et permettant une amélioration nette des performances du code.

Mesurer de nouveau le temps passé dans les différentes partie du code en fonction du nombre de cœurs de calcul utilisés et faites un tableau donnant l'accélération obtenue en fonction du nombre de threads (et en fonction du nombre de cœurs de l'ordinateur sur lequel vous exécuter le code).

Parallélisation du code

Il existe deux façon de paralléliser le code (en fait trois, mais le troisième est beaucoup plus complexe à réaliser) :

Première façon

L'idée ici est que chaque processus contienne l'environnement en entier et ne contrôle qu'une partie des fourmis tandis que lors de la mise à jour des phéronomes lors de la phase d'évaporation, chaque processus s'ocuppe d'une partie de la carte pour laquelle il calculera l'évaporation, à l'aide d'une parallélisation openmp si besoin.

Lorsque deux fourmis appartenant à deux processus différents se trouvent sur une même cellule, la valeur de la case va dépendre des valeurs des phéronomes des cellules voisines, qui peuvent être différent selon si d'autres fourmis locales sont passées avant ou non ! Cependant, l'algorithme peut normalement prendre les fourmis dans un ordre arbitraire, et donc dans ce cas, on choisira de prendre la valeur la plus grande d'entre tous les processus comme valeur de phéronome pour une cellule donnée !

Le problème de cette approche est qu'elle n'est efficace que si le nombre de fourmi est grand et la carte assez petite pour tenir en mémoire sur chaque processus.

De plus, l'évaporation des phéronomes se fait en parallèle, assurant ainsi un bon degré de parallélisme. De plus, cette méthode assure un bon équilibrage des tâches.

Par contre, une grande quantité de données est échangée entre les processus.

Mesurer les temps pris par les différentes parties du code en parallèle en fonction du nombre de processus (en prenant un cœur par processus) et calculer l'accélération correspondante

Seconde façon

Cette fois-ci, chaque processus prend en compte qu'une partie de la carte et ne gère que les fourmis qui sont sur la carte. La difficulté ici est de gérer les bords de chaque sous-carte. De plus, il est possible qu'un processus n'est pas de fourmis à gérer tandis qu'un autre en possède beaucoup (en particuliers celui possédant la fourmilière). L'équilibre des tâches ne sera donc pas optimal, mais si le nombre de fourmis est assez grand ainsi que le coefficient d'exploration, il est très probable que les fourmis soient relativement bien distribuées sur la carte.

L'avantage de cette méthode est que la mémoire occupée par l'application est bien plus petite qu'à l'aide de la première solution. De plus, il n'y a qu'un échange au bord des sous-cartes et donc peu de données échangées en définitive.

Indiquez votre stratégie pour programmer une telle parallélisation.

Bonus : Mettez en œuvre votre stratégie et calculer l'accélération en fonction du nombre de processus.