École Nationale Supérieur des Techniques Avancées

OS202

Projet Final

MAIA DE SOUZA Lethycia
MIRANDA DE PAULA Augusto

Palaiseau
2024

Table des matières

1 Introduction

L'optimisation par colonies de fourmis (ACO), inspirée par le comportement des colonies de fourmis réelles, est un exemple frappant de l'efficacité des algorithmes en essaim. Introduit en 1989 par Beni et al., ce paradigme d'algorithmes a depuis suscité un intérêt croissant dans la résolution de problèmes complexes, en particulier pour les problèmes combinatoires.

Dans ce projet, nous revenons à l'essence originale des algorithmes ACO en développant une version parallèle de l'algorithme Ants. L'objectif est d'améliorer l'efficacité de l'algorithme en exploitant pleinement les capacités massivement parallèles des systèmes informatiques modernes.

L'algorithme Ants, à l'instar de ses homologues réels, simule le comportement des fourmis pour résoudre efficacement le problème du forage. Les fourmis, en quête de nourriture, explorent leur environnement de manière distribuée et communiquent entre elles par le biais de phéromones. Cette interaction indirecte crée un système d'auto-organisation dynamique qui guide les fourmis vers des solutions optimales.

Dans ce rapport, nous présentons une analyse détaillée de notre approche pour paralléliser l'algorithme Ants. Nous examinons les défis rencontrés lors de cette parallélisation et discutons des techniques utilisées pour surmonter ces obstacles. Enfin, nous évaluons les performances de notre implémentation par rapport à la version séquentielle avec différents nombres de processeurs, démontrant ainsi les avantages de l'approche parallèle dans la résolution efficace du problème de forage.

2 Première Paralélisation

Dans la première partie de l'exercice, la parallélisation de l'affichage et des calculs effectués par l'algorithme pour décider du meilleur chemin que la fourmi doit suivre pour ramener la nourriture à la colonie a été réalisée. En utilisant deux cœurs, le rang numéro zéro est responsable de l'affichage du programme réalisé à l'aide de la bibliothèque pygame de Python, et le rang 1 est responsable de la prise de décision des "prochains pas" effectués par les fourmis de la Colonie, comme on peut le voir dans la fonction ***advance***, ainsi que le calcul des positions et des intensités de phéromones émises par les fourmis, comme on peut le voir dans la fonction ***do_evaporation***. La communication entre les processus a été réalisée en utilisant des routines de communication MPI importé de *mpi4py* de Python telles que ***send*** et ***recv***, où le processus 1 envoie au processus 0 les informations dont il a besoin pour afficher le mouvement de la fourmilière à l'écran, à savoir :

1. une liste de toutes les positions actuelles des fourmis,
2. une liste des directions dans lesquelles les fourmis se trouvent à l'instant actuel,
3. les phéromones,
4. la quantité de nourriture collectée par les fourmis.

Le code de cette première partie peut être exécuté en utilisant MPI avec la commande :

```
mpirun -np 2 python ants1.py
```

Ou encore :

```
mpiexec -n (nombre de coeurs) python ants1.py
```

3 Deuxième Paralélisation

Dans la seconde partie, nous avons parallélisé le code en partitionnant le calcul des fourmis et des autres objets (fourmilière, labyrinthe et phéromones) entre les processus dont le rang est non nul, tout en réservant le premier processus (rang=0) pour l'affichage visuel. Nous espérions initialement une amélioration des performances, car en théorie, l'utilisation de davantage de processeurs aurait dû permettre de répartir les tâches de manière plus équilibrée et d'accélérer le traitement. Cependant, en examinant les temps obtenus avec l'augmentation du nombre de processeurs, nous avons constaté que le temps nécessaire aux fourmis pour apporter 500 morceaux de nourriture à la fourmilière augmentait parfois, et dans certains cas, ce temps était même plus long qu'en séquentiel.

Une autre observation intéressante était que même si le temps total augmentait avec 4 processeurs, le temps nécessaire pour la première fourmi diminuait considérablement. Cela s'explique par le fait que les processeurs n'avaient pas beaucoup de communication

jusqu'à ce moment-là, et que l'utilisation de 4 threads partageait davantage la charge de calcul, réduisant ainsi le temps nécessaire.

En comparant ces résultats et en analysant le code, nous avons constaté que ces résultats sont principalement dus aux dimensions du problème (taille du labyrinthe) et au temps de communication entre les processus. Alors, nous avons pensé que pour des faibles dimensions le temps de communication entre les processus devient élevé proportionnellement au temps total, de sorte qu'il est préférable d'utiliser peu de processus. L'augmentation des dimensions entraîne une forte demande de calcul du système, et dans ce cas, le temps passé à effectuer les calculs devient plus important que le temps de communication entre les processus, ce qui rendrait la distribution de ces calculs sur davantage de processeurs plus intéressante. Nous avons vu à posteriori ce n'est pas si évident que ça.

Nous avons obtenu les résultats de temps moyennes (plusieurs executions) suivants en utilisant un labyrinthe de taille 25×25, avec 625 fourmis ayant une durée de vie de 500 pas sans trouver de nourriture, et avec des coefficients de phéromones $\alpha = 0.9$ et $\beta = 0.99$. Ces coefficients indiquent respectivement le taux de diffusion (1-α) et le taux d'évaporation.

Méthode	**1 nourriture (s)**	**500 nourritures (s)**
Séquentiel	6.03	14.46
1ère Parallélisation (nbp=2)	5.00	11.09
2ème Parallélisation (nbp=2)	5.52	12.15
3ème Parallélisation (nbp=3)	5.89	17.96
4ème Parallélisation (nbp=4)	3.32	23.45

TABLE 1 – Temps de calcul en rapport au nombre de processeurs et méthodologie

Méthode	**Speedup (1 nourriture)**	**Speedup (500 nourritures)**
1ère Parallélisation (nbp=2)	1.21	1.31
2ème Parallélisation (nbp=2)	1.09	1.19
3ème Parallélisation (nbp=3)	1.02	0.80
4ème Parallélisation (nbp=4)	1.81	0.62

TABLE 2 – Speedup en rapport au nombre de processeurs et méthodologie

Les tableaux ci-dessus confirment l'importance des temps de communication entre les threads, car nous observons une augmentation du temps total avec le nombre de threads. De plus, en comparant la première et la deuxième parallélisation, nous constatons que la généralisation du code pour utiliser un plus grand nombre de threads entraîne également une augmentation du temps total, ce qui met en évidence l'influence significative du temps de communication.

Nous avons obtenu les résultats de temps moyennes (plusieurs executions) suivants en utilisant un labyrinthe de taille 50×50, avec 2500 fourmis ayant une durée de vie de

2000 pas sans trouver de nourriture, et avec des coefficients de phéromones $\alpha = 0.9$ et $\beta = 0.99$.

Méthode	**1 nourriture (s)**	**500 nourritures (s)**
Séquentiel	38.16	73.30
$1^{\text{ère}}$ Parallélisation (nbp=2)	36.95	61.83
$2^{\text{ème}}$ Parallélisation (nbp=2)	33.64	66.22
$3^{\text{ème}}$ Parallélisation (nbp=3)	53.09	86.46
$4^{\text{ème}}$ Parallélisation (nbp=4)	71.43	127.93

TABLE 3 – Temps de calcul en rapport au nombre de processeurs et méthodologie

Ces résultats nous ont amenés à conclure que les performances d'un code parallélisé dépendent de nombreux paramètres du problème (nombre de fourmis, coefficients de phéromone, taille du labyrinthe,seed du labyrinthe, etc). Il est donc possible qu'ajouter un ou deux processeurs supplémentaires entraîne en réalité une perte de performance, plutôt qu'une amélioration. Cela souligne l'importance de comprendre en profondeur les caractéristiques du problème et les spécificités de l'architecture matérielle pour obtenir une parallélisation efficace.

Le code de cette partie peut être exécuté en utilisant MPI avec la commande :

```
mpirun -np (nombre de coeurs) python ants2.py
```

Ou encore :

```
mpiexec -n (nombre de coeurs) python ants2.py
```

4 Parallélisation en Partitionnant le Labyrinthe

Une autre façon de paralléliser le code est de diviser le labyrinthe entre les processeurs disponibles. Cela peut se faire en divisant le labyrinthe en lignes, en colonnes ou en cellules de taille égale, distribuant ainsi équitablement la charge de travail. Chaque processus serait responsable d'un sous-ensemble des fourmis et des phéromones, calculant les mouvements des fourmis dans sa propre section du labyrinthe et mettant à jour les phéromones au besoin. De plus, les processus devraient communiquer entre eux pour échanger des informations sur les colonies. Cela inclut informer quand une fourmi passe d'une zone à une autre du labyrinthe, changeant ainsi de domaine et, par conséquent, le noyau responsable. L'échange d'informations sur les aliments et la carte de phéromones serait réalisé en utilisant les routines ***send*** et ***recv*** du MPI.

À la fin de chaque itération de l'algorithme, les résultats de chaque processus devraient être collectés et agrégés pour obtenir le résultat global, à présenter par le processeur 0. Cette étape pourrait nécessiter une réduction, au cours de laquelle les résultats seraient combinés de manière appropriée.

Avec de multiples processus échangeant diverses informations entre eux, il est crucial de maintenir la synchronisation des données pour éviter les conflits et garantir la cohérence des données.

En mettant en œuvre ces aspects de manière efficace, il serait possible d'explorer correctement les capacités parallèles du système, améliorant ainsi l'efficacité de l'algorithme des fourmis dans la résolution du problème de recherche de nourriture.

5 Conclusion

En conclusion, ce projet nous a permis d'optimiser l'algorithm des colonies de fourmis (ACO) en le parallélisant. En développant une version parallèle de cet algorithme, nous avons cherché à exploiter les capacités parallèles du système pour améliorer son efficacité.

Au cours de ce projet, nous avons rencontré divers défis liés à la parallélisation, notamment la gestion de la communication entre les processus et la synchronisation des données. Malgré ces obstacles, nous avons pu observer des améliorations de performance dans certains cas, mais également des résultats mitigés dans d'autres. Cela souligne l'importance de choisir avec caution la méthode de parallélisation en fonction des caractéristiques spécifiques du problème à résoudre.

Enfin, ce projet nous a permis de mieux comprendre les implications de la parallélisation dans le domaine de la résolution de problèmes complexes. Bien que des défis en ce qui concerne les temps de communication et la taille du problème.