OS202 Fourmi

Sofiia Chorna

8 mars 2024

Table des matières

1 Caractéristiques du matériel **3**

2 Simulation d'optimisation de fourmis **3**
2.1 Introduction 3

3 Description du Problème **3**

4 Analyse du Code **4**

5 Analyse de la Parallélisation **4**

6 Exécution du Programme **4**
6.1 Résultats de la simulation 5
6.2 Analyse des résultats 6
6.3 Gains de Performances 6

7 Partitionnement du Labyrinthe **6**
7.1 Approche Maître-Esclave 6
7.2 Gestion du Labyrinthe 6

8 Conclusion **7**

1 Caractéristiques du matériel

```
~$ lscpu
Architecture:            x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         39 bits physical, 48 bits virtual
  Byte Order:            Little Endian
CPU(s):                  8
  On-line CPU(s) list:   0-7
Vendor ID:               GenuineIntel
  Model name:            11th Gen Intel(R) Core(TM)
      i7-1165G7 @ 2.80GHz
    CPU family:          6
    Model:               140
    Thread(s) per core:  2
    Core(s) per socket:  4
    Socket(s):           1
    Stepping:            1
    CPU max MHz:         4700,0000
    CPU min MHz:         400,0000
    BogoMIPS:            5606.40
Virtualization features:
  Virtualization:        VT-x
Caches (sum of all):
  L1d:                   192 KiB (4 instances)
  L1i:                   128 KiB (4 instances)
  L2:                    5 MiB (4 instances)
  L3:                    12 MiB (1 instance)
```

2 Simulation d'optimisation de fourmis

2.1 Introduction

La simulation d'optimisation de fourmis est une méthode inspirée du comportement des fourmis réelles pour résoudre des problèmes d'optimisation, tels que le problème du voyageur de commerce. Dans cette simulation, nous modélisons un environnement virtuel dans lequel un groupe de fourmis cherche à trouver le chemin le plus court entre une source de nourriture et leur nid.

3 Description du Problème

Le problème abordé concerne la parallélisation d'un code Python implémentant la recherche d'un chemin optimal pour un modèle simplifié de colonies de fourmis. Le code original est disponible sur GitHub à l'adresse suivante : `https://github.com/JuvignyEnsta/Fourmi2024`. Les étapes de parallélisation prévues sont les suivantes :

— Séparation de l'affichage de la gestion des fourmis/phéromones
— Partitionnement des fourmis entre les processus restants
— Réflexion sur le partitionnement du labyrinthe pour gérer les fourmis en parallèle sur un labyrinthe distribué entre les processus

Dans cette analyse, nous allons décrire et examiner les principales parties du code principal, en mettant en évidence celles qui pourraient être parallélisées. Ensuite, nous présenterons les stratégies envisagées pour paralléliser le code et partitionner chaque fonction. Enfin, nous discuterons de l'approche pour mettre en œuvre un code parallèle qui divise également le labyrinthe entre les processus.

4 Analyse du Code

Le code original est organisé en plusieurs classes :
— `Colony` : responsable de la création d'une colonie de fourmis et de la gestion de son comportement
— `Pheromon` : responsable de la mise à jour des phéromones, laissées par les fourmis ou évaporées selon une loi prédéfinie
— `Maze` : responsable de la construction du labyrinthe et de son image représentative

Dans la classe `Colony`, la fonction `advance` est chargée de faire progresser la colonie en gérant les fourmis chargées et non chargées, les positions des fourmis, l'historique des chemins, les sorties possibles et les phéromones. Les fonctions `explore` et `return_to_nest` mettent à jour les attributs des fourmis non chargées et chargées, respectivement.

Les autres classes comme `Pheromon` et `Maze` sont utilisées pour créer le labyrinthe et la matrice de phéromones qui guident les fourmis vers la nourriture et le nid.

5 Analyse de la Parallélisation

La parallélisation a été réalisée en plusieurs étapes, et les codes résultants sont disponibles sur le dépôt GitHub :
`https://github.com/sofiia-chorna/OS202/tree/main/Projet`. Le groupe a combiné deux types de parallélisation : parallélisation des données, où chaque processeur est responsable d'un groupe de fourmis, et parallélisation fonctionnelle, où les fonctions sont exécutées en parallèle sur différents processus.

Cette analyse servira de base pour comprendre les modifications apportées au code original et les stratégies utilisées pour la parallélisation.

6 Exécution du Programme

Pour exécuter le programme parallélisé, vous pouvez utiliser une commande similaire à celle-ci :

```
mpiexec -n [number_of_processes] python3 main_mpi.py
    [height] [width] [max_life] [alpha] [beta]
    [max_iterations=n]
```

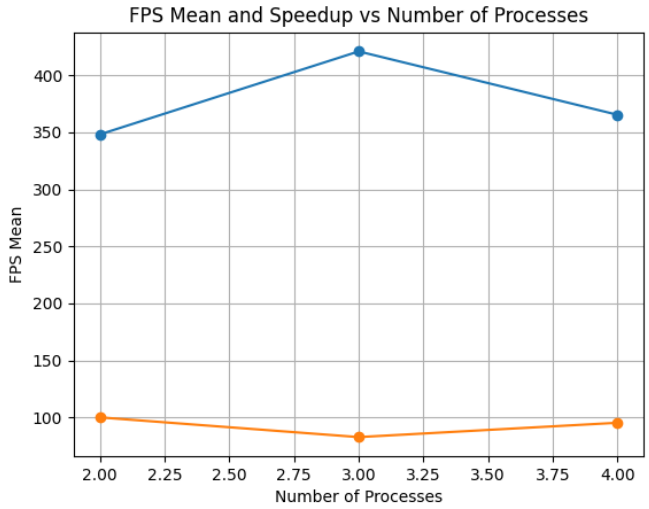
Les paramètres par défaut du programme sont les suivants :
— Nombre de fourmis : Calculé en fonction de la taille du labyrinthe.
— Durée de vie maximale des fourmis : 500.
— Position de la nourriture : En bas à droite du labyrinthe.
— Position du nid : En haut à gauche du labyrinthe.
— Alpha : 0.9.
— Beta : 0.99.
— Nombre maximal d'itérations : 5000.

6.1 Résultats de la simulation

Les résultats présentés dans le tableau 1 montrent les performances du programme en termes de FPS moyen et de speedup pour différentes configurations de nombre de processus.

Nombre de Processus	FPS Moyen	Speedup
2	348.21	100.00
3	421.05	82.70
4	365.43	95.29

TABLE 1 – Performances du programme pour différentes configurations de nombre de processus.



6.2 Analyse des résultats

— Lorsque le nombre de processus augmente de 2 à 3, le FPS moyen augmente de manière significative, passant de 348.21 à 421.05. Cependant, le speedup diminue de 100.00 à 82.70, indiquant une inefficacité croissante de la parallélisation à mesure que le nombre de processus augmente.
— Pour 4 processus, bien que le FPS moyen soit légèrement inférieur à celui de 3 processus, le speedup augmente à 95.29. Cela suggère que l'utilisation de 4 processus permet d'améliorer l'efficacité de la parallélisation par rapport à 3 processus.
— Le fait que le speedup diminue après 2 processus indique des inefficacités potentielles dans la gestion des ressources ou des communications entre les processus.
— Il est possible que des gains de performances supplémentaires soient obtenus en optimisant davantage la parallélisation ou en ajustant d'autres paramètres du système.

En conclusion, bien que l'ajout de processus supplémentaires puisse améliorer les performances dans certains cas, il est essentiel de surveiller l'efficacité de la parallélisation et de trouver un équilibre entre le nombre de processus et les performances globales du système.

6.3 Gains de Performances

Les gains de performances (speed-up) dépendront de divers facteurs tels que le nombre de processus disponibles, la taille du labyrinthe, le nombre de fourmis, etc. Plus précisément, le speed-up dépendra de la capacité à répartir efficacement la charge de travail entre les différents processus sans introduire de temps d'attente excessif pour la synchronisation des données.

7 Partitionnement du Labyrinthe

7.1 Approche Maître-Esclave

Pour envisager la partition du labyrinthe entre les différents processus, une approche maître-esclave pourrait être adoptée. Dans cette configuration, le processus maître serait responsable de diviser le labyrinthe en régions et de les distribuer aux différents esclaves (processus). Chaque esclave serait chargé de calculer les déplacements des fourmis dans sa région attribuée et de mettre à jour les phéromones localement. Une fois ces calculs effectués, les esclaves renverraient les informations pertinentes au processus maître, qui serait chargé de rassembler et de mettre à jour les données globales.

7.2 Gestion du Labyrinthe

Après la réception et la mise à jour des données globales, le processus maître redistribuerait les régions du labyrinthe entre les différents esclaves pour que

chaque région contienne approximativement la même quantité de fourmis à gérer. Ce cycle de distribution des régions, calculs locaux, mise à jour globale et redistribution des régions serait répété à chaque itération du programme.

8 Conclusion

En mettant en œuvre cette approche, nous pourrions obtenir des gains de performances supplémentaires en parallélisant non seulement les calculs des fourmis, mais aussi la gestion du labyrinthe lui-même. Cependant, cela nécessiterait une gestion plus complexe de la communication entre les différents processus, ainsi qu'une répartition équitable et efficace des tâches entre eux.