ÉCOLE NATIONALE SUPÉRUEURE DE TECHNIQUES AVANCÉES

OS202 - Programmation Parallèle

Projet

Diego Bernardes Rafael Pincer

Tiago Lopes Rezende

Palaiseau, France 2024

Table des matières

1 Introduction **3**

2 Parallélisation du Code **3**

2.0.1 Q1 3

2.0.2 Q2 3

2.0.3 Q3 3

3 Analyse des Résultats **5**

4 Conclusion **6**

5 Bibliographie **7**

Table des figures

1 Introduction

Dans le cadre de ce projet, nous explorons le domaine fascinant de la recherche de chemin optimal, en mettant l'accent sur l'utilisation des algorithmes coopératifs inspirés par l'intelligence collective des essaims. L'intelligence en essaim, un concept introduit en 1989 par Beni et al., s'inspire du comportement des insectes sociaux et implique une population d'agents simples qui interagissent et communiquent indirectement avec leur environnement pour résoudre des tâches de manière massivement parallèle. Le coeur de notre étude repose sur l'optimisation par colonies de fourmis (ACO), un algorithme phare qui simule le comportement des fourmis pour trouver des solutions optimales aux problèmes combinatoires. Ce projet vise à approfondir notre compréhension des mécanismes sous-jacents à ces algorithmes et à explorer leur potentiel pour améliorer les techniques de recherche de chemin dans diverses applications.

Notre approche a consisté à utiliser la bibliothèque mpi4py pour paralléliser certaines parties du code dans le but d'accélérer son exécution. En tirant parti des capacités de mpi4py, nous avons pu décomposer le problème en plusieurs sessions de code qui peuvent être exécutées simultanément sur différents processeurs. Cette stratégie de parallélisation nous a permis d'exploiter efficacement les ressources de calcul disponibles, réduisant ainsi le temps nécessaire pour atteindre des solutions optimales. L'utilisation de fonctions spécifiques pour orchestrer cette parallélisation a été un aspect clé de notre méthode, permettant une distribution équilibrée des tâches et une synchronisation efficace entre les processus parallèles.

2 Parallélisation du Code

2.0.1 Q1

- funçao main, separaçao entre 2 processos -

2.0.2 Q2

2.0.3 Q3

Pour partitionner un labyrinthe pour le traitement parallèle en utilisant mpi4py, nous commençons par initialiser MPI avec la bibliothèque mpi4py pour déterminer le rang de chaque processus ainsi que la taille totale du communicateur. Le rang est essentiellement l'identité de chaque processus dans la communication, tandis que la taille indique le nombre total de processus disponibles.

Une fois que nous avons cette information, nous procédons à la division du labyrinthe en sections, qui peuvent être attribuées à chaque processus. Cette division peut se faire de plusieurs manières, mais deux approches courantes sont la décomposition en blocs, qui divise le labyrinthe en sous-sections égales, ou la décomposition cyclique, où les cellules

individuelles sont distribuées alternativement entre les processus. Cette dernière peut être bénéfique pour l'équilibre de charge entre les processus.

Chaque processus gère alors une structure de données locale représentant sa section du labyrinthe, contenant des informations sur les murs, les chemins et la localisation actuelle des fourmis à l'intérieur de cette section. Lorsque les fourmis se déplacent vers la limite d'une section, elles sont communiquées au processus responsable de la section adjacente. Pour éviter les blocages dans la communication, il est courant d'établir une convention pour l'envoi et la réception d'informations, comme les processus de rang pair envoyant des informations en premier et ceux de rang impair recevant en premier.

La communication entre les processus adjacents est réalisée par des opérations d'envoi et de réception MPI. La communication peut être point à point ou collective, en fonction du besoin spécifique de la simulation. De plus, dans les simulations où les fourmis déposent des phéromones, il est essentiel de synchroniser les niveaux de phéromone aux frontières entre les sections des processus. Cela se fait généralement après chaque itération ou après un nombre déterminé d'itérations pour maintenir la cohérence de la carte de phéromones.

Dans des simulations complexes, il peut être nécessaire d'implémenter un équilibrage de charge dynamique. Cela devient nécessaire lorsque certaines sections du labyrinthe ont plus d'activité que d'autres, conduisant à des temps de traitement inégaux. L'équilibrage de charge redistribue le travail de manière plus équitable entre les processus.

Enfin, une fois les tâches des fourmis terminées, les résultats de chaque processus sont collectés. Cela peut se faire par des opérations d'agrégation telles que Gather ou Reduce, en fonction du type de données à collecter. Avec les résultats agrégés et le traitement terminé, l'environnement MPI est finalisé.

3 Analyse des Résultats

4 Conclusion

5 Bibliographie

[1] mpi4py Developers, "Documentation de mpi4py," [En ligne]. Disponible sur : `https://mpi4py.readthedocs.io/en/stable/`. [Accédé : 08-05-2024].

[2] X. Juvigny, "Cours 1 : l'introduction à la programmation parallèle," Cours de Programmation Parallèle, ONERA, Jan. 2023.

[3] X. Juvigny, "Cours 2 : les outils de performance et algorithmes embarrassingly parallel," Cours de Programmation Parallèle, ONERA, Jan. 2024.

[4] X. Juvigny, "Cours 3 : la théorie des algorithmes de tri parallèles," Cours de Programmation Parallèle, ONERA, Jan. 2024.

[5] X. Juvigny, "Cours 4 : les algorithmes de tri parallèles et leur mise en œuvre," Cours de Programmation Parallèle, ONERA, Jan. 2024.

[6] X. Juvigny, "Cours 5 : les stratégies de parallélisation et optimisation," Cours de Programmation Parallèle, ONERA, Jan. 2024.