



ÉCOLE NATIONALE SUPÉRIEURE DE TECHNIQUES AVANCÉES

OS202 - Programmation Parallèle

Projet

Diego Bernardes Rafael Pincer

Tiago Lopes Rezende

Palaiseau, France 2024

Table des matières

1	Introduction	4
2	Organisation sur GitHub	5
3	Parallélisation du Code	6
3.1	Explication du Modèle implémenté	6
3.1.1	Subdivision des classes	6
3.1.2	Réorganisation de la fonction main	6
3.1.3	Répartition entre processeurs	6
3.2	Parallélisation du labyrinthe	7
4	Analyse des Résultats	8
4.1	Calcul FPS	8
4.2	Calcul Speed-up	8
4.3	Gains Obtenus	9
5	Conclusion	10
6	Bibliographie	11

Table des figures

1	Affichage du pygame en cours d'exécution.	4
2	Organisation des codes sur GitHub.	5
3	Extrait de code originale - Calcul FPS.	8
4	Extrait de code originale - Affichage FPS.	8
5	Extrait de code modifié - Calcul FPS moyenne.	8

Liste des tableaux

- 1 Comparaison entre le nombre de processeurs utilisés et le FPS moyen obtenu. 9

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.

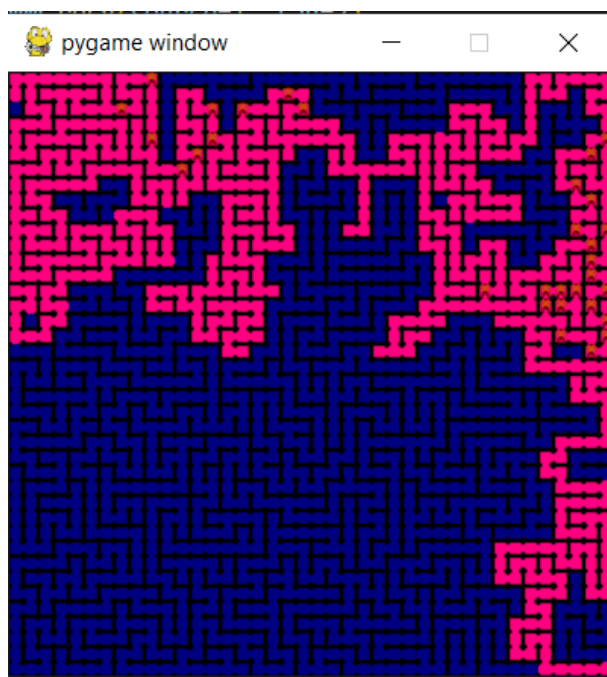
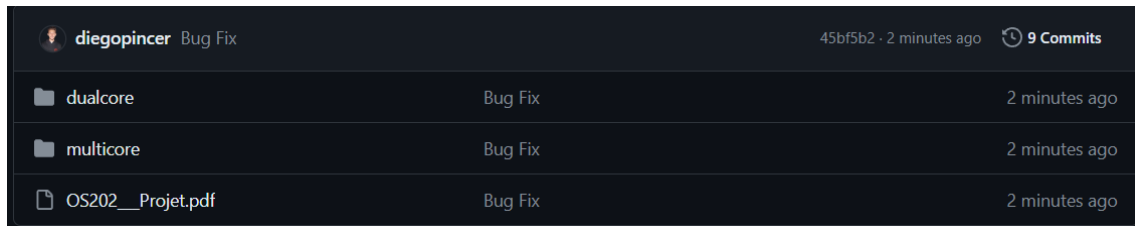


FIGURE 1 – Affichage du pygame en cours d'exécution.

2 Organisation sur GitHub

Nous avons réalisé un fork du dépôt de travail sur GitHub. Avec cela, nous avons codé le projet et effectué un commit. Le lien pour y accéder est le suivant : https://github.com/diegopincer/OS202_Projet.

Il existe deux dossiers dans le dépôt, un relatif au code parallélisé pour seulement deux processeurs (dualcore) et l'autre pour les autres processeurs (multicore). De plus, ce PDF est disponible là.



The screenshot shows a GitHub repository page for 'diegopincer' with the title 'Bug Fix'. The commit hash '45bf5b2' and the time '2 minutes ago' are displayed, along with '9 Commits'. Below this is a table listing the commit history:

Commit Hash	Message	Time
45bf5b2	Bug Fix	2 minutes ago
...
...
...

FIGURE 2 – Organisation des codes sur GitHub.

3 Parallélisation du Code

3.1 Explication du Modèle implémenté

3.1.1 Subdivision des classes

La première approche pour paralléliser le code a été de tenter de séparer la partie affichage du programme de la partie de traitement des fourmis. Pour cela, une séparation a été réalisée où certaines méthodes ont été retirées de certaines classes afin de permettre l'exécution locale. Ainsi, les méthodes `display` de la classe `Colony`, `display` et `getColor` de `Pheromon`, et `display` de `Maze` ont été déplacées vers les classes respectives `Colony_show`, `Pheromon_show`, `Maze_show`, qui comprenaient également la méthode `init`.

3.1.2 Réorganisation de la fonction main

De plus, le code à l'intérieur de la fonction `main` a été divisé entre celui exécuté par le rang 0, la partie d'affichage, et par le rang 1, la partie de calcul des fourmis. Dans le rang 0, `pygame` a été initialisé et les classes d'affichage ont été appelées, puis la réception de certains objets qui seraient utilisés plus tard dans la boucle `while`. Dans le rang 1, les `clDivision` de la fonction principale `assess` `Maze` et `Colony` ont été appelées pour initialiser le calcul des fourmis et des objets ont été envoyés à 0. La boucle `while` a également été divisée, le processus de rang 0 utilisant `pygame` pour effectuer l'affichage et recevant du processus 1 à l'aide de la fonction `recv` les données nécessaires pour appeler les méthodes. Le rang 1 appelle la méthode `ants.advance` pour calculer le `food_counter` et `ants.returns` qui a été créé pour retourner `directions`, `historic_path` et `age`, utilisés pour appeler la méthode `ants_show.display`, ce qui envoie les données de retour en utilisant la fonction `send`. L'utilisation de l'étiquette était nécessaire dans toutes les communications de ce processus car le code réalisé comportait de nombreuses communications entre les processus et l'identification des messages était nécessaire.

3.1.3 Répartition entre processeurs

Après cela, le traitement des fourmis a été séparé en plusieurs processus. Tout d'abord, la transmission des données a été remodelée et simplifiée : avant la fonction `main`, seul le processus 1 envoyait des données au processus 0 pour transmettre les paramètres nécessaires au démarrage de l'affichage. Ensuite, à l'intérieur de la boucle `while`, seul un envoi était réalisé du processus 0 à tous les autres processus, et seulement une autre communication venant des autres processus vers le processus 0, ce qui a permis de supprimer les balises, qui empêchaient le fonctionnement de la réception des messages envoyés par tout processus autre que 0 ou 1. La raison de ce problème n'a pas été trouvée malgré les efforts investis. De plus, le nombre de fourmis était divisé par le nombre de processus qui allaient calculer les fourmis, et le fonctionnement à l'intérieur de la boucle `while` a subi quelques adaptations : le phéromone était unifié en utilisant la valeur la plus élevée retournée par tous les processus, et la somme de nourriture retournée était effectuée, le tout dans le processus 0. Comme les

fonctions qui géraient les fourmis et les phéromones étaient locales, les valeurs de phéromone et de `food_counter` étaient renvoyées du processus 0 pour permettre le calcul du prochain cycle réalisé par les fourmis. Avec ces modifications, il a été possible de diviser le traitement des fourmis en plusieurs processus.

Il est important de souligner que d'autres outils de MPI auraient pu être utilisés, tels que la diffusion (`diffusion`) pour additionner `food_counter` tout en rassemblant les données, et la dispersion (`scatter`) pour envoyer toutes les informations simultanément à tous les processus. Cependant, l'utilisation de ces outils s'est avérée très problématique, et une mise en œuvre plus simple utilisant les fonctions `recv` et `send` a été adoptée.

3.2 Parallélisation du labyrinthe

Pour partitionner un labyrinthe pour le traitement parallèle en utilisant `mpi4py`, nous commençons par initialiser MPI pour déterminer le rang de chaque processus ainsi que la taille totale du communicateur.

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.

4 Analyse des Résultats

4.1 Calcul FPS

Pour analyser le gain de performance obtenu avec la parallélisation, la stratégie choisie a été d'analyser le nombre moyen de Frames par Seconde (FPS) atteint par l'ordinateur. Pour ce faire, nous avons profité d'une partie de la fonctionnalité du code fourni par le professeur :

```
deb = time.time()
pherom.display(screen)
screen.blit(mazeImg, (0, 0))
ants.display(screen)
pg.display.update()

food_counter = ants.advance(a_maze, pos_food, pos_nest, pherom, food_counter)
pherom.do_evaporation(pos_food)
end = time.time()
```

FIGURE 3 – Extrait de code originale - Calcul FPS.

À chaque boucle du while, l'affichage est mis à jour, comme montré dans le code ci-dessus, enregistrant le temps de début et de fin dans les variables *deb* et *end*, respectivement. Enfin, le FPS est calculé comme étant 1 divisé par la différence entre le temps final et initial.

```
print(f"FPS : {1./(end-deb):6.2f}, nourriture : {food_counter:7d}", end='\r')
```

FIGURE 4 – Extrait de code originale - Affichage FPS.

Cela dit, une petite modification a été faite pour qu'il soit possible de calculer le FPS moyen à chaque boucle while. Pour cela, le pas suivant a été réalisé :

```
fps.append(int(1./(end-deb)))
counter += 1
print(f"FPS : {1./(end-deb):6.2f}, nourriture : {food_counter:7d}, average fps: {np.sum(fps)/counter}", end='\r')
```

FIGURE 5 – Extrait de code modifié - Calcul FPS moyenne.

4.2 Calcul Speed-up

Comme vu dans certains exemples en classe, la méthode choisie pour calculer l'accélération (speed-up) obtenue avec la parallélisation des processus a consisté à diviser le nombre de FPS du processus parallélisé par le nombre de FPS obtenu par le processus linéaire. Pour illustration, en considérant que dans le code exécuté sur un seul processeur nous avons obtenu une moyenne de 146 FPS et sur 3 processeurs 256, l'accélération pour ce cas a été de 175 %.

$$Speed - up = \left(\frac{256}{146} \times 100\% \right) - 100\% = 75.3\%$$

4.3 Gains Obtenus

Après avoir expliqué les méthodes de calcul des FPS et de l'accélération, il est possible de faire une comparaison générale entre tous les cas simulés, cherchant à comprendre dans quels cas un meilleur comportement a été observé et pourquoi.

Processeurs	FPS Moyenne	Speed-up
1	146	0%
2 - code uniquement pour 2 processus	165	13.0%
2	210	43.4%
3	256	75.3%
4	206	41.1%
5	170	16.4%
6	147	-0.7%

TABLE 1 – Comparaison entre le nombre de processeurs utilisés et le FPS moyen obtenu.

En analysant le tableau présenté, on peut observer une augmentation significative du speed-up lors du passage de 1 à 3 processeurs, suivie d'une décroissance quand on utilise 4 processeurs, et une légère amélioration avec 5 processeurs. Cette tendance suggère que, bien que l'ajout de processeurs supplémentaires puisse améliorer la performance initialement, il existe un point de saturation au-delà duquel l'ajout de ressources supplémentaires n'apporte pas d'amélioration significative, voire détériore la performance. Une explication possible pourrait être le surcoût lié à la communication entre les processus. En effet, à mesure que le nombre de processeurs augmente, le temps nécessaire pour l'envoi et la réception des données entre les processus peut devenir substantiel, compensant ainsi les gains de performance obtenus par la parallélisation des calculs. De plus, cela peut être exacerbé par des problèmes de synchronisation et de contention des ressources, où les processeurs doivent attendre les uns les autres pour progresser, ce qui ralentit davantage le calcul global.

5 Conclusion

À travers ce projet, nous avons exploré l'application fascinante de la parallélisation du code pour optimiser la recherche de chemin dans un labyrinthe, inspirée par les mécanismes d'intelligence collective et de coopération observés chez les fourmis. Notre exploration s'est appuyée sur l'utilisation de la bibliothèque `mpi4py` pour répartir le traitement du problème sur plusieurs processeurs, permettant ainsi une accélération significative de l'exécution du code. Nous avons constaté, grâce à une analyse méthodique des performances, que la parallélisation apporte des améliorations notables en termes de FPS et de speed-up, surtout jusqu'à un certain point où l'ajout de processeurs supplémentaires ne se traduit plus par des gains significatifs. Cette observation souligne l'importance de trouver un équilibre entre le nombre de processus parallèles et l'efficacité de la communication et de la synchronisation entre eux.

Notre travail contribue à une meilleure compréhension de la manière dont la programmation parallèle peut être appliquée efficacement pour résoudre des problèmes complexes. Il met en lumière les défis inhérents à la parallélisation, tels que la gestion des communications entre processus et l'équilibrage de la charge de travail, tout en démontrant le potentiel d'accélération qu'elle offre.

Pour les recherches futures, il serait intéressant d'explorer des stratégies d'équilibrage de charge dynamique plus avancées, ainsi que l'impact de différentes topologies de communication sur les performances. De plus, l'extension de notre approche à d'autres problèmes d'optimisation combinatoire et la comparaison avec d'autres méthodes de parallélisation pourraient offrir de nouvelles perspectives sur la manière d'exploiter au mieux les capacités des architectures parallèles modernes.

En conclusion, ce projet souligne l'efficacité de la programmation parallèle pour améliorer les performances de l'algorithme d'optimisation par colonies de fourmis, tout en révélant les limites et les défis associés à cette approche. Les leçons tirées de cette étude offrent des pistes précieuses pour l'optimisation d'autres algorithmes complexes, ouvrant ainsi la voie à des applications encore plus larges de la programmation parallèle dans le domaine de l'intelligence artificielle et de l'optimisation.

6 Bibliographie

[1] mpi4py Developers, “Documentation de mpi4py,” [En ligne]. Disponible sur : <https://mpi4py.readthedocs.io/en/stable/>. [Accédé : 08-03-2024].

[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.