Master ISICG

Rapport : Mise en correspondance d'images

Parallélisme et Application

Florian AUBERVAL
Timothée BEHUET

1 Introduction

Nous avons choisi de travailler sur le projet de mise en correspondance d'images.

Tout d'abord, nous avons fait une version de référence du projet qui recherche l'image de manière séquentielle. Cette version est compilable et exécutable avec le fichier « main.c ».

Plus de détail sur la compilation et l'exécution des différentes versions peut être retrouvé dans le fichier README.md du projet

Le projet peut être trouvé sur le dépôt GitHub suivant : `https://github.com/Me-k-01/ParallelisationCorrespondanceImage`

2 Version OpenMP

Nous avons fait une version parallèle de toutes nos fonctions de référence.

La fonction «search» a été sujette à deux implémentations différentes.

- Une implémentation plus classique parallélisant les boucles for, et une implémentation utilisant le mécanisme de taches.
- Et une implémentation utilisant les taches, qui se trouve légèrement moins efficace au global, cette solution n'a pas été retenu pour l'implémentation finale.

Nous avons dû faire un choix entre deux fonctions pour paralléliser le programme final.

Étant donné que la fonction « search » se sert de la fonction « evaluator », nous ne pouvons pas paralléliser les deux fonctions en même temps. Cela est dû au fait que OpenMP ne permet pas d'imbriquer les instructions de parallélisation entre elles.

Une fois que l'on rentre dans une zone exécutée parallèlement par les processus, nous ne pouvons pas refaire appelle à « #pragma omp for ».

Pour faire notre choix, nous avons comparé les temps d'exécutions suivant la parallélisation de « search » et de « evaluator ». Si on parallélise la fonction de recherche, le temps d'exécution sur l'image « space.png » est d'environ 3s contre 8s si on parallélise la fonction d'évaluation.

Ce qui nous a amené à conclure qu'il était plus intéressant de paralléliser « search ». Nous allons donc laisser « evaluator » en séquentiel pour la version du programme parallélisé avec OpenMP.

Dans la suite, nous allons nous servir de MPI pour contourner ce problème, en parallélisant justement « search » avec MPI pour pouvoir utiliser la version d'« evaluator » parallélisé sous OpenMP.

3 Versions MPI

Avec MPI, nous allons uniquement paralléliser la fonction de recherche «search», cela nous permettra plus tard d'envisager une version hybride utilisant conjointement MPI et OpenMP. Nous aurons donc la fonction de recherche «search», et la fonction d'évaluation «evaluator» toutes les 2 parallélisées.

Précédemment, lorsque nous voulions paralléliser ces fonctions, nous étions bloqués, car nous ne pouvions pas faire appel à OpenMP pour paralléliser « evaluator », car elle était appelée dans un bloc parallélisé.

Version répartition du travail par le processus racine

Nous avons tout d'abord tenté de diviser le travail entre les machines en se servant d'un système similaire au mécanisme de tache d'OpenMP.

Nous avons décidé que le processus racine commencerait par envoyer à toutes les machines un premier travail à effectuer, qui correspond à la recherche de l'entropie sur une partie de l'image par rapport à l'image que l'on recherche.

Chaque machine conserve dans son propre espace mémoire le meilleur x et y qu'il a trouvé, afin d'éviter de surcharger de travail la machine racine.

Le processus racine va ensuite lui-même travailler sur la prochaine taches, et à chaque fois qu'il termine une tâche, il regarde, de manière non bloquante, si les autres machines lui ont envoyé une demande de travail.

Si une machine demande du travail, le processus racine lui en envoie, et ce dernier continue la boucle.

Lorsque le processus racine n'a plus de travail, il envoie un signal de fin à tout le monde pour qu'on lui renvoie le x et y et la meilleure entropie locale que chaque machine a trouvé sur ses différents travaux.

Depuis la machine racine, on récupère ses différentes valeurs et on déduit ainsi le meilleur x et y global.

— **Avantage :** Cette version a pour avantage de parfaitement répartir les taches entre les machines. Les machines plus lentes auront moins de travail, tandis que les plus rapides en auront plus. De plus, cette version sera très efficace si on a un très grand nombre de machines (autant qu'il y a de morceau d'image à explorer), car la répartition des taches est optimale.

— **Inconvénient :** Cependant, on pourra noter qu'elle a pour défaut de surutiliser le réseau. Il y a beaucoup de communication entre les machines étant donné qu'à chaque fois qu'une des machines a fini de travailler, elle doit demander à la machine racine du travail et se mettre à attendre.

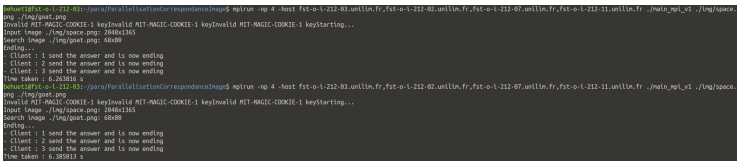

FIGURE 1 – Résultat d'execution de la version 1 du programme

En voyant que le temps d'exécution de cet algorithme n'était pas très bon sur des petits clusters (voir image 1), nous avons décidé de faire une seconde version.

Version découpage du travail suivant un axe

Pour cette seconde version de « search » parallélisé avec MPI, nous avons découpé l'espace de travail selon l'axe x de l'image.

Nous avons donc réparti le travail en se servant simplement de la place des processus dans le « world rank » pour savoir qu'elle partit de l'image chaque machine doit traiter.

À la fin du travail de toutes les machines, le processus racine récupère les x et y que stockent les machines avec l'aide d'une réduction MPI sur l'entropie SSD des machines. Pour récupérer les x et les y pendant la réduction, il a fallu créer nous même une structure de donnée et l'opération de min sur cette structure de donnée.

- **Avantage :**
  La répartition du travail de cet algorithme a pour avantage de ne pas provoquer de délais d'attentes entre les travaux des machines. Ici tout est immédiat. De plus, la réduction nous fait gagner du temps lors de la récupération final du résultat.
- **Inconvénient :** Cependant, sur une très grande quantité de machine, elle sera incapable de répartir correctement le travail, étant donné que l'on découpe l'image selon un seul axe.

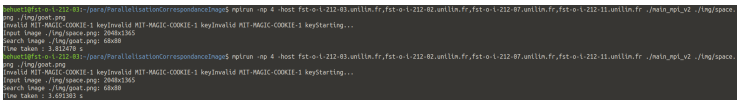

FIGURE 2 – Résultat d'execution de la version 2 du programme

4 Version Hybride

Chacune des 2 versions utilisant MPI a été hybridé avec OpenMP, afin d'essayer d'en tirer de meilleures performances. La fonction de recherche «search» est parallélisée à l'aide de MPI. La fonction d'évaluation « evaluator » elle, est parallélisée à l'aide d'OpenMP.

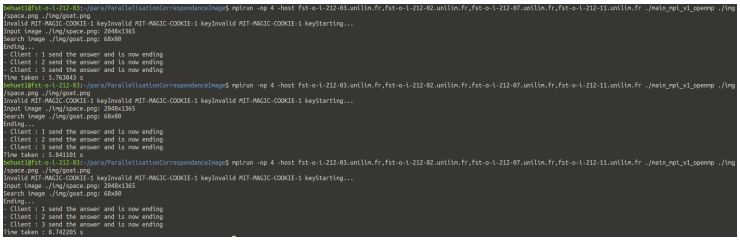

FIGURE 3 – Version 1 hybride avec OpenMP

On remarque que cette version hybride est sujette de grosse variation de temps d'exécution. Cette variance est peut-être liée aux aléas du réseau, d'autant que cette version effectue beaucoup de communication durant son exécution, elle serait donc d'autant plus sensible à ce genre de choses.

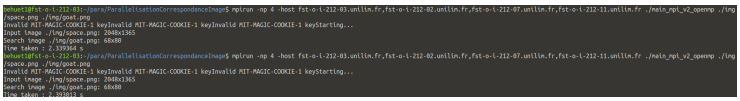

FIGURE 4 – Version 2 hybride avec OpenMP

On peut constater que pour la version 2 (voir image 4), l'hybridation permet d'obtenir de meilleurs résultats. Ceci est d'autant plus intéressant qu'elle n'est pas soumise à de grosses variations.

5 Comparaison des résultats

Les programmes ont été compiler en utilisant l'option -O3 du compilateur. En exécutant plusieurs fois les programmes sur la même image «space.png». (les temps ont été calculer sur les machines de la salle I-212)
Les versions MPI ont été testées sur 4 machines. On obtient en moyenne pour les programmes non-hybride :

Séquentielle	OpenMP	Version 1 MPI	Version 2 MPI
13.59s	2.92s	6.45s	3.644 s

On constate que la version séquentielle est vraiment très lente, alors que la version OpenMP propose déjà des performances bien plus intéressantes.

Même si on peut voir que la version 2 sous MPI est plus rapide que la version 1 sous MPI, on constate qu'elle n'arrive pas pour autant à atteindre les performances de la version OpenMP.

On obtient en moyenne pour les programmes hybrides :

Hybridation Version 1-OpenMP	Hybridation Version 2-OpenMP
6.78s	2.37s

La version hybride de MPI 1 et d'OpenMP offre des résultats très décevants, en faisant en moyenne moins bien que sa version non hybridée. Cependant, nous tenons à rappeler que cette version hybride souffre d'une variance élevée.

La version hybride de MPI 2 et d'OpenMP offre, elle, en comparaison, des résultats bien plus intéressants.

6 Conclusion

En conclusion, l'optimisation qui est la plus rapide dans le cadre de la mise en correspondance d'images, est le programme hybride OpenMP et MPI, en version 2. C'est en divisant en amont le travail de recherche dans l'image sur les différentes machines avec MPI, et en parallélisant les autres fonctions que nous obtenons le résultat le plus rapidement.

Nous pouvons tout de même supposer que sur un nombre très grand de machines (plus qu'il y ait de pixel sur l'axe x de l'image), la première version hybride avec MPI et OpenMP pourrait être plus rapide que celle-ci, car elle répartirait mieux le travail sur les différentes machines.