Université de Limoges

Faculté des Sciences & Techniques

Mini rapport Parallélisme et Applications

Projet image : Mise en correspondance d'images

Allan BLANCHARD
Antoine COUTY

18 mai 2022

1 Informations

Pour compiler notre programme et l'exécuter, il faut aller dans notre dossier src/ puis exécuter les deux commandes suivantes successivement :

```
gcc -fopenmp -o main main.c utils.c -lm
./main ../img/IMAGE_SOURCE.png ../img/VOTRE_FILTRE.png
```

Nous avons légèrement modifié l'architecture du code fourni. Nous avons tout d'abord à la racine du projet ce pdf et un dossier nommé code. Dans ce dossier, nous avons un dossier lib contenant stbi, un dossier image contenant les différentes images fournis et notre code.

Puisque l'image qu'il faut retrouver contient canal alpha il a faut que nous commencions par réfléchir à comment gérer cela. Nous partons donc du principe que tout les pixels d'une image à retrouver comportant de la transparence doivent être ignorés.

2 Séquentiel

2.1 Naïf

Pour commencer nous avons créé un algorithme nommé "sequential" qui comme son nom l'indique sera notre algorithme séquentiel de référence par la suite. Ce programme s'efforcera de trouver l'image qu'on lui demande de trouver dans l'image de référence même si elle ne s'y trouve pas.

Sa méthode : pour chaque pixels de l'image source, il va calculer la différence entre le patch et la sous-image (en utilisant la formule du sujet). A la fin de chaque patch, on vérifie si nous n'avons pas déjà une valeur stocké inférieur à celle-ci. Si ce n'est pas le cas on stocke la nouvelle valeur ainsi que les coordonnées du pixel courant représentant le pixel en haut à gauche du pattern. Cette méthode fonctionne mais est très lourde à calculer avec l'utilisation de quatre boucles (deux pour parcourir l'image source en x et y et deux pour parcourir le patch).

2.2 Linéarisé

Ainsi, nous avons réfléchi à une deuxième option pour le séquentiel où cette fois-ci, nous linéarisons nos boucles for, passant de quatre boucles à deux. Ensuite plutôt que tester tous les patchs différents et conserver leurs valeurs, étant donné que si l'emplacement du pattern est trouvé, la différence entre le patch et la sous image est 0. Nous pouvons donc arrêter le calcul dés qu'une image correspond et ne pas chercher dans le reste de l'image. De ce fait, si lors de la différence entre l'image et le patch nous obtenons pour un pixel une valeur plus grande que 0 nous pouvons nous arrêter car nous savons

que l'image ne correspondra pas au final. Cette méthode est beaucoup plus rapide que celle d'avant mais très peut robuste, au moindre pixel qui change de nuance elle ne reconnaîtra pas l'image. L'algorithme en cause s'appelle "sequentialLinear".

3 OpenMP

Nous sommes ensuite partis de ces deux algorithmes séquentiel pour les convertir en algorithme parallèle à l'aide d'OpenMP.

3.1 Naïf

Pour ce qui est du premier algorithme, nommé "ompNaif", nous appliquons à sa première boucle for un : `#pragma omp parallel for schedule(dynamic)` . Cela nous permet de diviser les threads dans notre parcours et le dynamique permet de spécifier aux threads qu'il n'ont pas besoin d'attendre que tout les autres aient finis pour reprendre un nouveau tour de boucle.

Ensuite, avec la parallélisation il nous faut nous assurer qu'il n'y ai pas d'accès concurrent en écriture par la suite. Première chose, lors du calcul de la différence des patches nous avons essayé deux méthodes différentes. La première consiste à rajouter la clause `reduction(+:value)` à la notre première boucle for. La deuxième consiste ensuite à effectuer une opération atomic au moment du calcul de la valeur(`#pragma omp atomic`). Ces deux méthodes sont plus ou moins équivalentes et permettent dans les deux cas de réduire à peu près par 10 le temps de calcul de l'algorithme séquentiel. Etant donné que la réduction était très légèrement plus longue à calculer, nous somme rester sur l'utilisation de l'opération atomic. Pour finir, sur notre dernière condition nous effectuons un `#pragma omp critical` pour que chaque thread passe une par une et être sur d'obtenir la valeur minimale du résultat des n threads fonctionnels.

3.2 Linéarisé

La parallélisation du deuxième algorithme séquentiel n'a pas été de tout repos. Étant donné qu'il n'est pas possible d'invoquer le mot clef break dans une région parallèle, nous avons du légèrement modifier l'algorithme original. Pour cela, nous utilisons maintenant un flag permettant d'avertir les autres threads lorsque l'image à chercher a été trouvé et qu'il n'est donc plus nécessaire de continuer la recherche. Cela se traduit simplement par l'utilisation de la commande `#pragma omp parallel for schedule(dynamic) shared(flag)` .

Avant la boucle parallèle le flag est initialisé à 1 (true), ensuite durant la boucle, le flag ne peux changer que pour la valeur 0 (false). Lorsque le flag est à false, les threads terminent leur travail sans effectuer de calcul inutile.

Cette méthode garde en plus de cela l'optimisation lui permettant de ne pas avoir besoin d'aller jusqu'au bout du calcule du patch dès qu'il rencontre au moins une différence supérieur à 0. Nous avons nommé cet algorithme "ompNaifLinear".

Cet algorithme est en effet plus rapide que son prédécesseur mais reste tout aussi peu robuste, c'est pourquoi nous avons implémenté un nouvelle version de cet algorithme qui peut prendre en compte un seuil de tolérance du total des différences entre le patch et la sous-image. Ce seuil peut être régler dés le début de l'algorithme pour accéléré le calcul mais de base est mise sur INT_MAX. Au fur et à mesure que les threads trouvent des patchs de meilleurs en meilleurs, ils mettent à jour ce seuil pour au final arriver à trouver la sous-image la plus ressemblante à celle cherché. Le seuil étant une donnée partagée par les threads avec shared(bestSeuil) en clause, une barrière de synchronisation implicite se forme avant de comparer les différents résultats des threads, évitant ainsi tout accès concurrentiel. Cette algorithme nommé "ompTerminator" est en quelque sorte un hybride entre nos deux approches. Elle est à la fois plus rapide que notre version naïve parallèle mais moins que celle évoqué au dessus mais peut être vraiment robuste, tout dépend du paramétrage.

3.3 Autre approche parallèle

Pour contourner le problème d'accès concurrent survenant lorsque nous avons tenté de passé notre version séquentiel naïve vers du parallèle, nous avons pensé à une autre approche profitant mieux d'OpenMP. Cette algorithme va simplement créer un tableau notant les scores de chaque patch. Ces scores peuvent tous indépendamment être calculé par un seul thread qui n'aura donc pas de problème d'accès conçurent et au final une fois le tableau remplis il nous suffis de retrouver à l'aide d'un seul thread ou plusieur quel patch à eu la meilleur note. Cet algorithme à été implémenté sous le nom de "ompScoreBoard". Il a de meilleur résultat que les algorithmes séquentiel naif et le parallèle naif mais mois que les version linéarisé. Mais il reste très robuste.

3.4 Résultat

Pour les comparatifs qui vont être fait ici, le nom ref est donné à notre algorithme séquentiel, linear à notre algorithme séquentiel linéarisé, OMP_naif à notre algorithme parallélisant naïvement notre ref, OMP_linear notre version parallèle de linear, seuil à la version d'OMP_linear utilisant un seuil et scoreBoard à notre algorithme nommé ompScoreBoard.

Pour tous les temps énoncé ci dessous nous comptabilisons aussi la formation de la boite rouge autour de notre sous-image car elle fait partie de l'exercice. Nous avons fait nos comparatifs et sur le tableaux 1 vous pouvez voir les résultats obtenue.

	259X195	820X532	1366X768	2048X1365
	beach	jungle	chevre	space
ref	0,61	9,35	24,22	70,92
linear	0,015	0,038	0,3	0,896
OMP_naif	0,198	2,973	7,706	23,65
OMP_linear	0,003	0,022	0,097	0,274
seuil	0,094	0,228	2,119	7,265
ScoreBoard	0,151	2,74	6,844	19,409

FIGURE 1 – Temps en secondes des algorithmes en fonctions des images

Fort heureusement tous nos algorithme surpassent la version séquentiel naïf. Pour pouvoir mieux visualiser la différence entre les algorithmes, nous proposons trois graphes avec respectivement tous les algorithmes (figure 2) réunis. Et comme une différence notable est observable entre les trois algorithmes plus rapide dérivé de linear et les trois autres dérivé de ref, nous proposons deux autres graphique. Le première composé des trois plus lents (figure 3) et le deuxième composé des trois plus rapides(figure 4).

On peut voir à l'allure des courbes que tous nos algorithmes battent au la main ref et sont au minimum trois fois plus rapide. Il est tout de même important de noter que la différence entre les rapides des lents se fait surtout du fait que les rapides ne testent que s'il existe l'image tel quel dans l'image de recherche. Ici seuil à été configuré avec un seuil de départ à INT_MAX. Donc nous pouvons réellement comparer ref, omp_naif, scoreboard et seuil entre eux et linear et omp_linear entre eux. Bien que omp_naif et scoreboard aient des approchent différentes, ils ont des temps très similaire qui sont probablement dût à la synchronisation qui les freinent tout deux. Seuil est donc le grand vainqueur de ce côté là. Et pour ce qui en est des deux autres il est assez logique qu'un même algorithme soit supérieur dans sa version parallélisé.

4 MPI

Pour ce qui est de l'approche MPI, nous ne l'avons pas implémenté. Pour ce que nous voulions faire, nous avions comme approche de faire traiter aux différentes machines des sous-patchs par rapport à celui qui était actullement calculé. Puis pour chaque sous-patchs évaler et retourner un booléen pour annoncer si le sous-patchs est conformes entre l'image source et le pattern.

Nous n'avons pas implémenter cette méthode par manque de temps et nous avons voulu nous concentrer sur OpenMP.

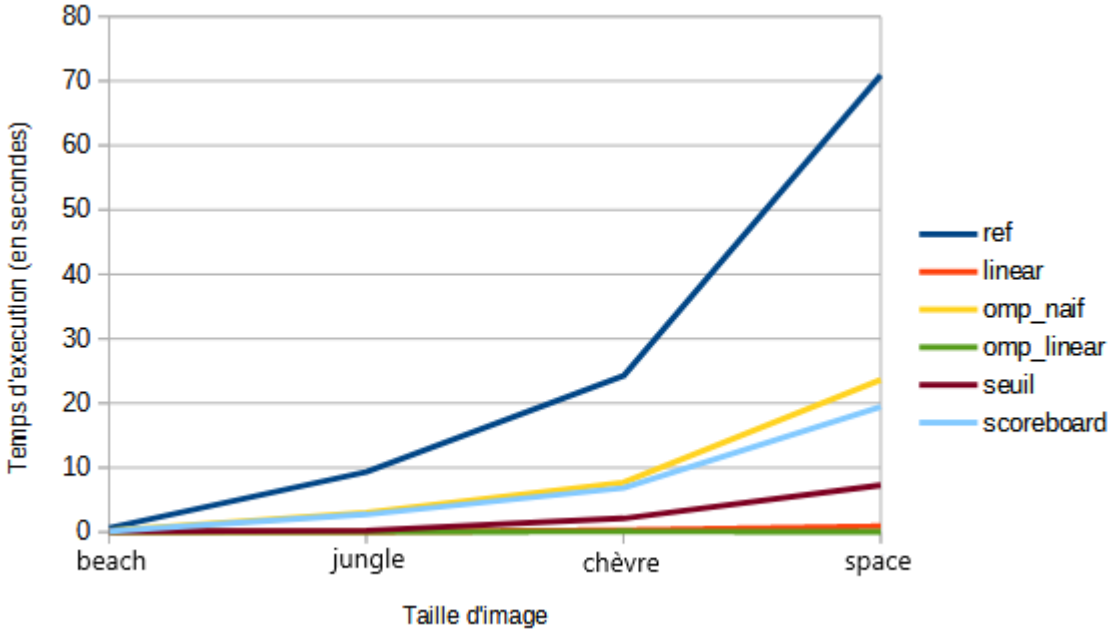

FIGURE 2 – Résultats en secondes pour chaque images avec tous les algorithmes

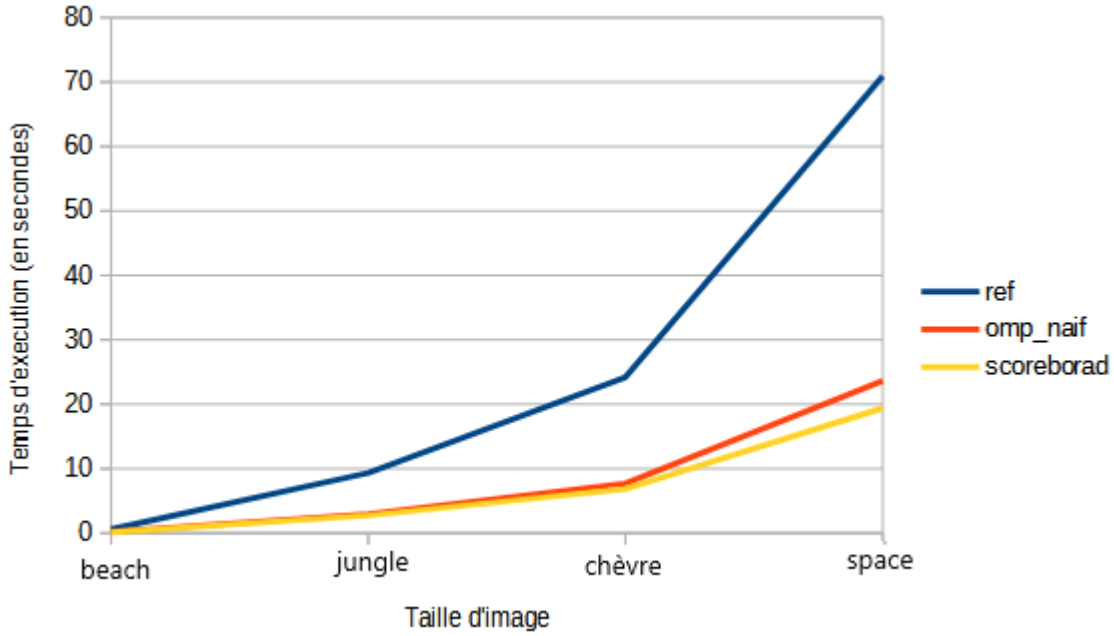

FIGURE 3 – Résultats en secondes pour chaque images avec les trois plus lents algorithmes

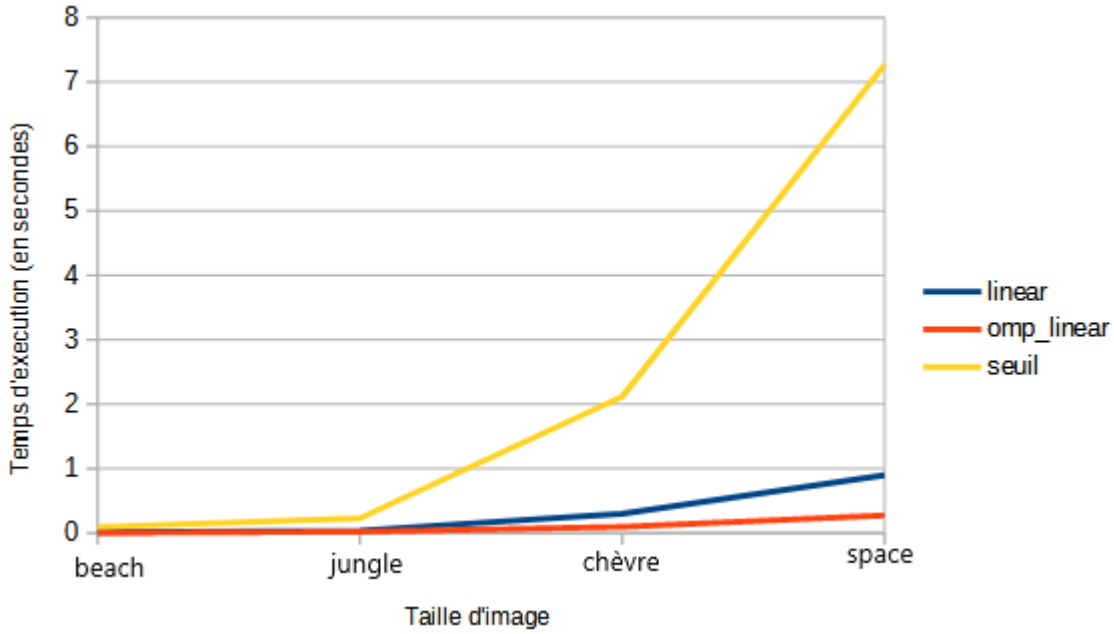

FIGURE 4 – Résultats en secondes pour chaque images avec les trois plus rapides algorithmes