Travail Pratique 2

Pthread

Équipe 7:
Diego Silva Pires
Hervé Simard

26 février 2018

Introduction

L'exercice est de transformer une implémentation séquentielle d'une convolution d'image afin de paralléliser le traitement et améliorer ainsi la performance. L'objectif est d'utiliser la librairie Open MP afin de se familiariser avec son fonctionnement et de paralléliser le code séquentiel de la convolution d'image.

Le programme doit accepter, sur une ligne de commande au moins 2 paramètres, soit le le nom du fichier à convoluer et la matrice à appliquer sur le fichier. Un troisième paramètre indiquant le nom du fichier de sortie est optionnel, par défaut le fichier s'appellera "output.png" à moins d'indication contraire.

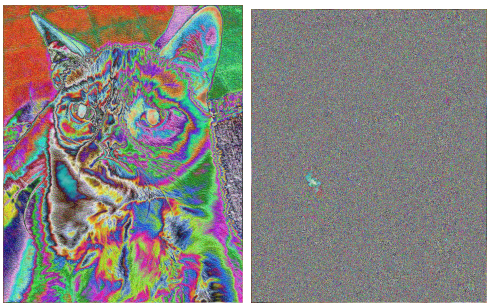
Par la suite nous voulons exposer le temps que le programme séquentiel a pris pour convoluer l'image versus le temps pris par l'implantation parallèle afin d'en comparer la performance et, ainsi, établir le speedup obtenu.

Procédé

Notre programme a testé plusieurs approches avant d'obtenir des résultats concluants. La première constatation a été de bien implémenter le parallélisme pour la première double boucle; en effet, remarquez l'exemple suivant:

```
#pragma omp parallel for //private(lToken) //schedule(auto) //collapse(2)
for (int i = 0; i < lK; i++) {
   for (int j = 0; j < lK; j++) {
       // #pragma omp atomic read
       lTok.getNextToken(lToken);
       lFilter[i * lK + j] = atof(lToken.c_str());
   }
}
```

Les tests nous ont démontrés que le décodage de l'image jouait un rôle important dans le rendu final. Trois paramètres de parallélisations ont été testés comme l'indique le code ci-haut, mais sans résultats. Voici d'ailleurs quelques exemples d'images affectés par ce traitement:



Nous avons aussi tenté d'appliquer un "`pragma omp atomic read`" afin de pallier au problème, mais nous ne sommes pas parvenus à faire compiler l'application, lorsque la commande "read" s'attend à avoir une assignation de variable dans la ligne suivant et non pas un appel de méthode.

Notre hypothèse était que l'obtention du jeton par la méthode "getNextToken" devait se faire de façon séquentielle. Enfin, l'implantation d'une boucle parallèle ordonnée a réglé le problème:

```
#pragma omp for ordered
for (int i = 0; i < lK; i++) {
   for (int j = 0; j < lK; j++) {
       lTok.getNextToken(lToken);
       lFilter[i * lK + j] = atof(lToken.c_str());
   }
}
```

L'autre problème se situait au niveau de la quadruple boucle for où la convolution se produisait. Nos tests nous ont rapidement démontrés qu'une implantation de boucles parallèles avec des variables privées et partagées sont nécessaires:

```
#pragma omp parallel for shared(lImage, lWidth, lHeight) private(lR, lG,
lB, fy, fx) //collapse(2) // schedule(static, 2)
for(int x = lHalfK; x < maxWidth; x++)
{
   for (int y = lHalfK; y < maxHeight; y++)
   ...
```

Si, par exemple, nous implémentons une boucle parallèle de type "collapse", celle-ci ne génère pas de bonnes images malgré des résultats intéressants. Des boucles for "schedule" ne donnent pas de résultats si différent pour le speedup.

Nous avons tentés, par exemple, de paralléliser les boucles intérieures:

```
//#pragma  omp  parallel  for  schedule(dynamic)  shared(lImage,  lWidth,
lHeight)
for (int j = -lHalfK; j <= lHalfK; j++) {
   fy = j + lHalfK;
   for (int i = -lHalfK; i <= lHalfK; i++) {
       fx = i + lHalfK;
       ...
```

Les temps obtenus sont pires encore, donc nous n'avons pas opté pour cette solution. Enfin, nous avons aussi tenté de déclarer les threads au début de l'exécution du code (#pragma omp parallel num_threads(4)) mais nous n'avons pas noté d'améliorations notables.

Résultats partiels

Le Mac utilisé possède les spécifications suivantes:
Processeur: 2.9 GHz Intel Core i5
Memory: 8 GB 1867 MHz DDR3
Graphique: Intel Iris Graphics 6100 1536 MB

Au bout du compte nous avons appliqué 2 parallélismes distinct, soit:

1. Implantation d'une boucle ordonnée sur le décodage de l'image;
2. Implantation d'une boucle

Nous n'avons pas modifié l'algorithme original autrement que par l'ajout des boucles parallèles. Les résultats finaux ont été obtenus avec le noyau flou sur l'exemple fournis.

Le temps parallèle indiqué est une moyenne sur plusieurs appels subséquents:

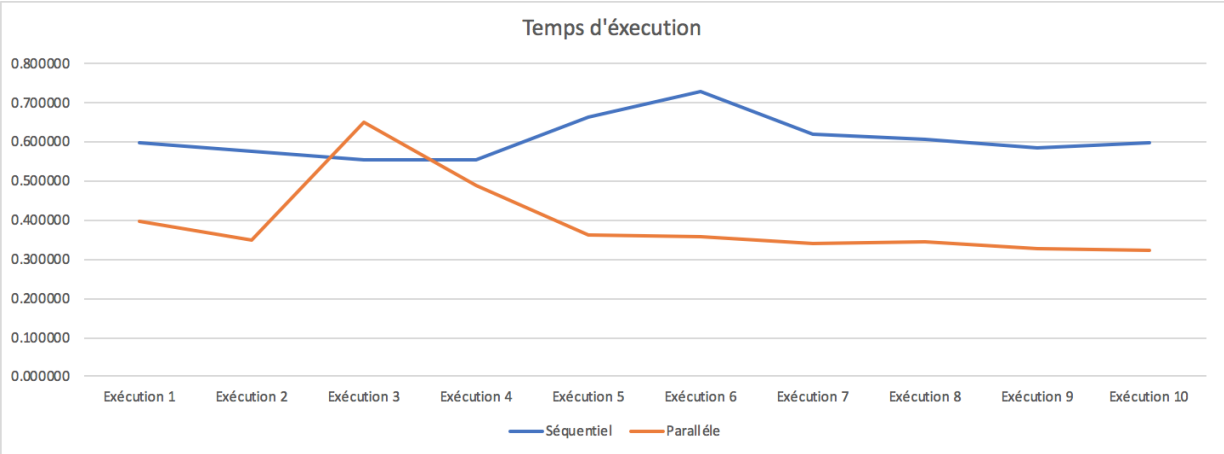
- Temps d'exécution séquentiel: 3.11816 secondes
- Temps d'exécution parallèle: 2.30416 secondes

Ceci nous donne un speedup de plus ou moins 1.35.

Lorsque nous avons effectué plusieurs exécutions de chaque méthode afin de compiler la moyenne, nous avons réalisé que le temps utilisé pour charger et enregistrer un fichier avec la librairie "lodepng" avait un impact sur la performance s'il était déjà présent, ça provoquait beaucoup d'inconsistance dans l'exécution du code. Nous avons donc effectué une pause à ce moment afin de bien évaluer la performance:

```
lChrono.pause();
//Appeler lodepng
decode(iFilename.c_str(), lImage, lWidth, lHeight);
lChrono.resume();
```

Voici une compilation de temps avec compilés sur MAC réalisés dans une boucle de 10 appels.



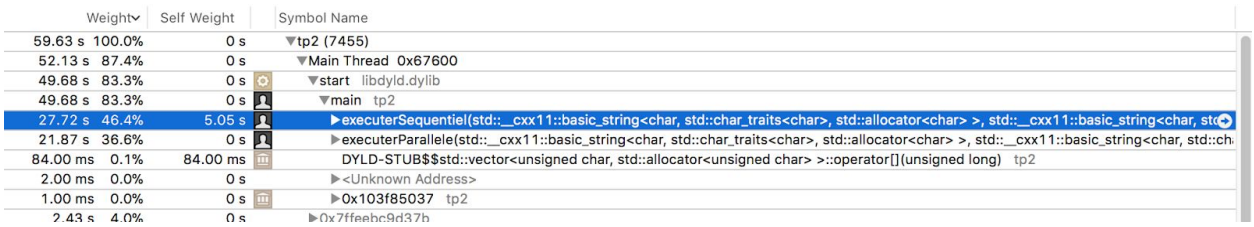
Temps d'exécution séquentiel moyen = 0.607352 sec
Temps d'execution parallele moyen = 0.393694 sec

Conclusion

Open MP offre de nombreuses possibilité pour améliorer le traitement parallèle des boucles et du code, mais il faut en même temps en mesurer l'impact. Quelques tests ont permis d'avoir un speedup intéressant, mais au prix d'une application qui ne fonctionne pas, résultant ainsi à une optimisation inutile.

Il faut donc mesurer où et comment on effectue notre optimisation, par exemple en introduisant les boucles ordonnées (parallel for ordered).

Nous avons utilisé l'outil "Instruments" dans Mac avec la fonctionnalité "Time Profiler" pour inspecter l'utilisation des coeurs du Mac:



On remarque que le programme parallèle utilise 36.6% du temps du processeur pour le programme au complet tandis que le séquentielle en prend 46.4%.

En regardant plus en détail l'utilisation du processeur, on note une moyenne de 2 secondes pour compléter l'exécution du code parallèle. C'est intéressant de noter une instruction d'initialisation de la thread pour chaque exécution.