TP2
OpenMP

GIF 4104
Programmation parallèle et distribuée

Pier-Luc Auger (PLAUG2 - 910 098 011)
Mathieu Garon (MAGAR220 - 111 005 164)

Université Laval

Rapport TP2

Mise en contexte

Pour ce deuxième TP, on vous demande d'utiliser OpenMP afin de paralléliser un programme pour le filtrage numérique d'images. Un code s'exécutant de manière séquentielle était fourni lequel a servi de base au code paralléliser que nous avons réalisé pour accélérer le traitement des images.

Méthode utilisée

La librairie omp.h a été utilisée pour paralléliser le code.

Le script utilisé a été roulé en boucle de 20 itérations: 20 exécutions du code séquentiel suivi de 20 exécutions du code parallélisé. Cela a permis d'utiliser une moyenne de temps d'exécution selon une configuration plutôt que le résultat d'une seule exécution prise au hasard.

La commande `#pragma omp parallel private(i) num_threads(nbThreads)` a été utilisée pour paralléliser le traitement en boucle effectué sur les blocs de l'images. On appelle OpenMP sur cette boucle avec `#pragma omp for schedule(static) nowait`.

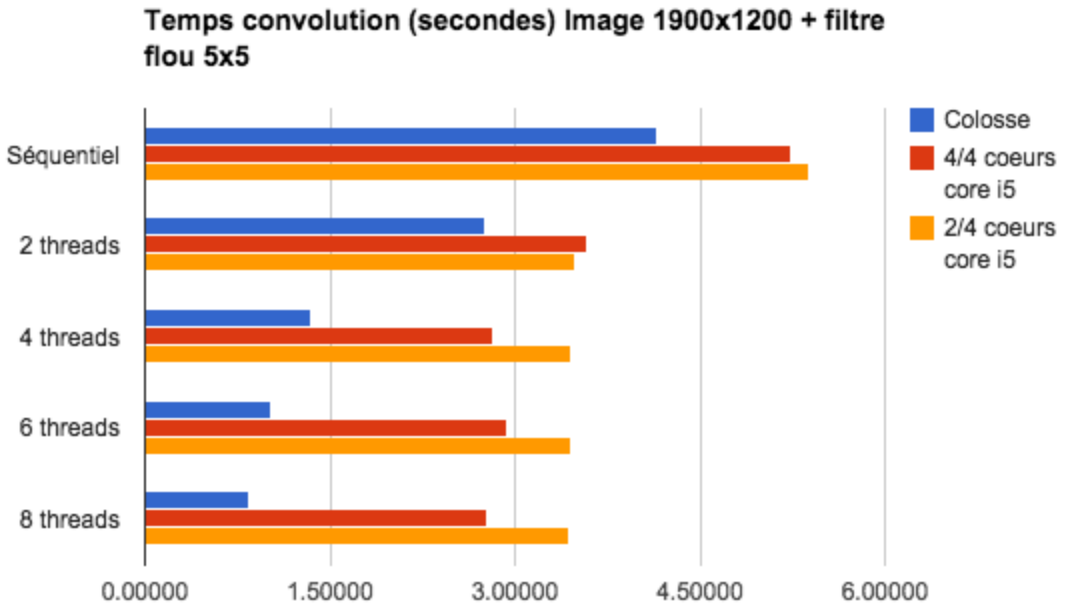
```
#pragma omp parallel private(i) num_threads(nbThreads)
{
    #pragma omp for schedule(static) nowait
    for (i = 0; i < blocks; ++i) {
        int x = i%convoWidth + hf;
        int y = i/convoWidth + hf;
        int lR = 0.;
        int lG = 0.;
        int lB = 0.;
```

Pour tester plusieurs configuration simplement, le programme prend en entrée un paramètre qui correspond au nombre de threads que l'on veut faire générer par OpenMP. Si on ne passe rien au programme, le nombre de coeur physique sur la machine qui exécute le programme est utilisé.

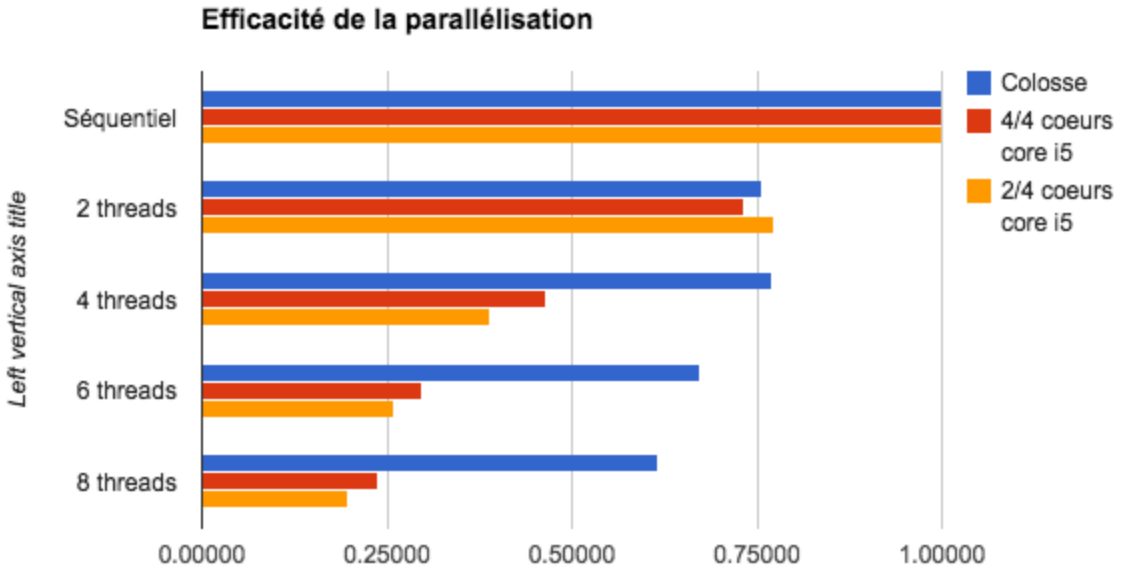
Résultats obtenus

Voici les résultats que nous avons obtenus, avec différentes configurations.

Temps convolution (secondes) Image 1900x1200 + filtre flou 5x5			
	Colosse	**4/4 coeurs core i5**	**2/4 coeurs core i5**
Séquentiel	4.15741	5.23961	5.38379
2 threads	2.74894	3.57707	3.48349
4 threads	1.34955	2.81575	3.45607
6 threads	1.02812	2.94127	3.45857
8 threads	0.84271	2.76561	3.43137



Efficacité de la parallélisation			
	Colosse	**4/4 coeurs core i5**	**2/4 coeurs core i5**
Séquentiel	1.00000	1.00000	1.00000
2 threads	0.75618	0.73239	0.77276
4 threads	0.77015	0.46521	0.38944
6 threads	0.67395	0.29690	0.25944
8 threads	0.61667	0.23682	0.19612



Analyse des résultats

On peut ressortir certaines tendances des graphiques montrant nos résultats. On remarque bien que sur Colosse, la programmation parallèle est efficace et plus le nombre de coeur utilisé est grand plus le programme s'exécute rapidement.

En calculant l'efficacité de la parallélisation avec la formule:

$$E_p = \frac{S_p}{p} = \frac{T_1}{pT_p}.$$

on remarque aussi que l'efficacité est meilleure sur Colosse que sur une machine virtuelle tournant sur Linux. En effet, au-delà de 2 threads, la machine virtuelle voit son efficacité chutée. Cela s'avère cohérant puisque le nombre de coeur physique sur la dite machine est de 2. Sur une machine avec 4 coeurs physiques, le déterioration de l'efficacité se serait alors fort probablement manifestée lors d'une parallélisation sur plus de 4 coeurs.

Lancement de l'exécutable

Pour utiliser notre programme, il s'agit d'utiliser `cmake` pour générer le makefile. Le CMakeList est fourni avec le code. Cela permettra ensuite d'utiliser la commande `make` pour compiler le programme. Le programme prend en entrée deux paramètres obligatoires et trois optionnels .

Obligatoires:

- chemin vers la photo
- chemin vers le filtre

Optionnels:

- nom de la photo résultante
- nombre de threads OpenMP a utilisé
- nombre d'itérations à faire pour calculer une moyenne du temps d'exécution

Sur colosse, le code est déjà disponible et exécutable. Des scripts Moab sont dans le répertoire `/TP2` sur notre espace `/home/user46`. Ceux-ci suggèrent différentes utilisations du programme.