TP **OpenMP**

1 Vérification des capacités du système [F. Butelle]

Si vous avez un système linux de type debian/ubuntu, on peut recueuillir des informations.

- Pour savoir quelle version d'**OpenMP** est disponible : `dpkg --status libgomp1`
- Pour savoir combien de coeurs votre machine dispose : `cat /proc/cpuinfo`

Généralement Linux les numérote linéairement de 0 à $n-1$ (voir les lignes "processor" dans le résultat par exemple avec ***grep***).

Cela peut être des coeurs dédoublés vituellement (hyperthreading) ; pour avoir le nombre de coeurs physiques, regardez les lignes "processor". On peut aussi consulter la commande lscpu si elle est disponible.

1. Écrire un `Makefile` avec comme contenu uniquement :

```
CXX = gcc
CFLAGS = -Wall -fopenmp -O3

fichier:
    $(CXX) $(CFLAGS) fichier.c -o fichier
```

   Cela permettra d'utiliser la commande `make fichier` pour compiler le fichier `fichier.c`.

2. Écrire un programme exo1.c avec les directives et fonctions **OpenMP** nécessaires pour afficher :
   - le nombre de processeurs
   - le nombre de threads max/limite
3. Votre programme ne doit pas forcer le nombre de threads. Compilez-le et exécutez-le en essayant de changer la variable d'environnement `OMP_NUM_THREADS`.
   Vous pourrez pour cela utiliser au niveau *shell* : `OMP_NUM_THREADS=6; export OMP_NUM_THREADS`.

2 Opérations sur les éléménts d'un tableau [F. Butelle]

On va travailler sur un tableau de taille N. Ce tableau doit être alloué dynamiquement et N doit être passé en premier paramètre du main. Le nombre de threads utilisé doit être passé en deuxième paramètre (ceci pour pouvoir écrire des scripts *Bash* facilement en changeant le nombre de threads). Le tableau doit d'abord être initialisé avec des nombres flottants aléatoires (utilisez rand et srand définis dans `stdlib.h`).
A chaque élément du tableau on appliquera une fonction f. Pour chaque fonction vous essaierez avec $2, 3, .., 16$ threads et des tailles de N de 10^3, 10^5, 10^7 et 10^8.
N'hésitez pas à lancer l'exécutable plusieurs fois avec les mêmes paramètres, il peut y avoir de grandes variations...

1. Écrire d'abord la version séquentielle avec la mesure du temps de calcul et la paralléliser avec **OpenMP**. Ne comptez pas le temps d'initialisation du tableau.
   On mesurera le temps de calcul avec la fonction `omp_get_wtime()` qui renvoie le temps courant en secondes comme un `double`. Il faut évidemment compiler avec le plus haut niveau d'optimisation. On utilisera la fonction :

$$f(x) = 2.17 \times x$$

2. Qu'obtenez-vous comme accélération avec 2,3,..,16 threads ? Ne comptez pas le temps de l'initialisation (la fonction rand n'est pas thread safe et ne doit pas être utilisée en multithreads).
3. Essayez avec la fonction :

$$f(x) = 2.17 \times ln(x) \times cos(x)$$

   Attention c'est la fonction log en C et il faut ajouter la librairie maths par l'option de compilation `-lm` (le plus simple est de rajouter `LDLIBS=-lm` dans le fichier makefile).

3 Somme des éléments d'un tableau [F. Butelle]

1. Écrire un programme réalisant la somme en parallèle des éléments d'un tableau de taille N (rempli aléatoirement comme dans la section précédente) en utilisant une réduction.
2. Mesurer le temps écoulé, faites varier N et le nombre de threads pour voir l'évolution. Quel est l'accélération ? N'hésitez pas à lancer l'exécutable plusieurs fois, il peut y avoir de grandes variations...
3. Pour essayer d'améliorer l'accélération, essayez de jouer avec la clause schedule, en particulier la taille des blocs/chunks.
4. Et sans utiliser la réduction, mais en rendant atomique l'affectation à la variable d'accumulation, c'est mieux ? Et avec une section critique ?

4 Détection des angles par la méthode de Shi-Tomasi

La détection des angles Shi-Tomasi a été publiée par J.Shi et C.Tomasi dans leur article intitulé "Good Features to Track" [1]. L'idée principale est de détecter les coins en recherchant un changement significatif dans toutes les directions de l'image.

En considérent une fenêtre de petite taille, déplacer cette fenêtre dans n'importe quelle direction entraînerait un changement important de l'apparence, si cette fenêtre se situe au même endroit qu'un coin. La Figure 1 illustre les différentes possibilités.

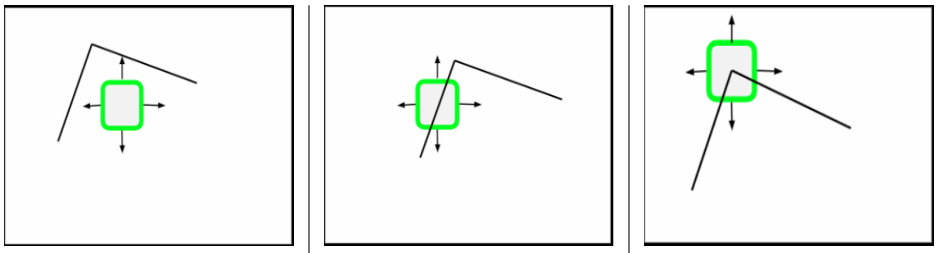

FIGURE 1 – (1) Pas de changement d'apparence. (2) Changement mineur d'apparence. (3) Changement majeur d'apparence.

4.1 Fondement mathématique

Pour une fenêtre W située à la position en pixels (X, Y) avec une intensité de pixel $I(X, Y)$, la formule pour la détection des coins de Shi-Tomasi est donnée par :

$$f(X,Y) = \Sigma(I(X_k, Y_k) - I(X_k + \Delta X, Y_k + \Delta Y))^2 \qquad \text{Où } (X_k, Y_k) \in W$$

Si on parcourt l'image avec une fenêtre (comme pour le kernel d'un filtre gaussien) et qu'on remarque qu'il y a une zone où il y a un changement majeur, alors on peut déduire avec une grande probabilité qu'il y a un coin à cet endroit.

Le calcul de $f(X, Y)$ étant très lent, l'expansion de Taylor (de premier ordre) est utilisée pour simplifier la fonction [2].

Après simplification $f(X, Y)$ peut être réécrites comme :

$$f(X,Y) \approx I(X,Y) \times \mathcal{M}$$

La matrice $\mathcal{M}$ est le résultat obtenu après le calcul de la convolution de l'image. Finalement le score permettantde détecter un coins dans la fenêtre choisie est donné par R :

$$R = min(\lambda_1, \lambda_2) \qquad \lambda_1 \text{ et } \lambda_2 \text{ sont les valeurs propres de } \mathcal{M}$$

1. Shi, Jianbo ; Tomasi, Carlo. "Good features to track." 1994 Proceedings of IEEE conference on computer vision and pattern recognition. IEEE, 1994.
2. Equations disponibles sur : `https://users.cs.duke.edu/~tomasi/papers/shi/TR_93-1399_Cornell.pdf`

4.2 Travail demandé

Le projet qui permet de détecter les coins dans une image sera mis à disposition. Il sert à mettre en oeuvre, sur des images type `.jpeg ou .png`, la fonction de détection de coins en précisant la taille de la fenêtre W et le nombre de coins désiré. La figure 2 ci-dessous donne un aperçu de la détection de coins dans une partie de l'image.

L'objectif est d'évaluer les performances entre deux versions. La première dite "non-optimisée" (qui intègre des itérations) et la deuxième dite "optimisée" (après optimisations **OpenMP**).

1. Analyser le code fourni pour la détection des coins
   - Analyser le fichier Makefile.
   - Repérer les fonctions de mesure de temps.
   - Repérer les portions qui peuvent être optimisées.
2. En s'inspirant du code fourni, réecrire une version optimisée et parallélisée.
3. Comparer les résultats obtenus par la version "non-optimisée" et la version "optimisée". Mesurer les accélérations obtenues.
4. Varier la taille de la fenêtre utilisée, le nombre maximal de coins à détecter, la taille des images à l'entrée du programme et le nombre de threads utilisées puis comparer la version séquentielle et la version parallélisée. Que pouvez-vous déduire ?
5. Tracer une courbe qui donne la variation du paramètre ***CPP*** (Nombre de cycles par pixel) en fonction de la résolution de l'image. Que pouvez-vous déduire ?

Rappel : $CPP = \frac{Temps\_de\_calcul\_de\_la\_fonction}{Resolution \times Temps\_cycle\_machine}$

FIGURE 2 – Exemple de détection de coins sur une partie de l'image