Accélération de calculs 2D avec des *Threads*

Dans ce TP on s'intéresse à des traitements qui manipulent des images sous forme de tableaux en 2 dimensions qui contiennent des pixels.

1 Introduction

Pour faciliter le développement, nous allons utiliser un petit environnement nommé EASYPAP permettant d'afficher des images, de lancer des calculs dessus, et de visualiser interactivement les évolutions de l'image à chaque itération.
Une fois l'archive récupérée et décompressée, placez vous dans le répertoire `easypap` et suivez les instructions des premières pages de la documentation (pour compiler l'environnement et lancer vos premières exécutions). La documentation se trouve dans le répertoire `easypap/doc/`. Les 6 premières pages devraient suffire.
Si tout va bien, vous devriez pouvoir observer une image animée lorsque vous tapez :

```
./run -k spin
```

2 Découverte du code

Le fichier `kernel/c/spin.c` contient plusieurs variantes du noyau `spin`. Commençons par modifier (temporairement) la variante séquentielle pour bien comprendre la façon dont l'image est parcourue.
Appliquez la modification suivante (ligne n°7) dans le code de la fonction `spin_compute_seq` (Figure 1).
Recompilez et relancez pour vérifier l'effet de votre modification :

```
./run --kernel spin --variant seq
```

Regardons à présent la variante `tiled` (Figure 2), qui découpe virtuellement l'image en `GRAIN` × `GRAIN` tuiles carrées (de côté $\frac{\texttt{DIM}}{\texttt{GRAIN}}$).
La fonction `do_tile` se charge de calculer les pixels à l'intérieur d'une zone rectangulaire définie par les paramètres `(x, y, w, h)`. Même si elle travaille avec des tuiles carrées dans cette variante, nous nous en servirons plus tard pour travailler sur des rectangles.
Appliquez la modification suggérée (Figure 2, ligne n°7) dans le code de la fonction `spin_compute_tiled`, recompilez et vérifiez l'effet à l'écran :

```
./run --kernel spin --variant tiled
```

Faites variez le nombre de tuiles en utilisant l'option `--grain` suivie d'une puissance de 2. Par exemple :

```
unsigned spin_compute_seq (unsigned nb_iter)
{
  for (unsigned it = 1; it <= nb_iter; it++) {

    for (int i = 0; i < DIM; i++)
      for (int j = 0; j < DIM; j++)
        if (j < DIM/2)                  // Try also: if (i < DIM/2 || j < DIM/2)
        cur_img (i, j) = compute_color (i, j);

    rotate (); // Slightly increase the base angle
  }

  return 0;
}
```

FIGURE 1 – Version séquentielle simple du noyau `spin` (`kernel/c/spin.c`).

```
./run --kernel spin --variant tiled --grain 32
```

Prenez un moment pour contempler.

```
unsigned spin_compute_tiled (unsigned nb_iter)
{
  for (unsigned it = 1; it <= nb_iter; it++) {

    for (int ty = 0; ty < GRAIN; ty++)
      for (int tx = 0; tx < GRAIN; tx++)
        if ((tx + ty) % 2)
        do_tile (tx * BLOCK_SIZE /* column */, ty * BLOCK_SIZE /* row */,
                 BLOCK_SIZE /* width */, BLOCK_SIZE /* height */,
                 0 /* thread id */);

    rotate ();
  }

  return 0;
}
```

FIGURE 2 – Version tuilée du noyau `spin`.

3 Version multithread

Il s'agit maintenant d'écrire une version accélérée au moyen de threads. Un squelette de la version `thread` vous est fourni (Figure 3).
La fonction `easypap_requested_number_of_threads` retourne le nombre de threads spécifié par l'utilisateur (au moyen de la variable d'environnement `OMP_NUM_THREADS`), ou le nombre de cœurs de la machine par défaut.

```c
static unsigned nb_threads = 0;
static unsigned iterations = 0;

unsigned spin_compute_thread (unsigned nb_iter)
{
  iterations = nb_iter;
  nb_threads = easypap_requested_number_of_threads ();

  // TODO: create threads (each with a different number
  //       in range [0..nb_threads-1], and wait for their
  //       completions

  return 1; // We temporarily stop after one iteration
}
```

FIGURE 3 – Squelette de la version multithread du noyau `spin`.

3.1 Création des threads

Écrivez une fonction `void *thread_starter (void *arg)` qui sera exécutée par chaque thread.
Ajoutez une boucle dans `spin_compute_thread` pour créer `nb_threads` threads, en faisant en sorte que chaque thread puisse récupérer son index (entre 0 en `nb_threads` −1) en paramètre. N'oubliez pas d'attendre la terminaison des threads au moyen de `pthread_join`.
Afin de testez que le lancement se déroule bien, contentez-vous dans un premier temps d'un `printf` dans la fonction exécutée par les threads. Vérifiez que le programme crée bien le nombre de threads demandé :

```
[my-machine] OMP_NUM_THREADS=4 ./run -k spin -v thread
Using kernel [spin], variant [thread]
Thread 0/4 started
Thread 2/4 started
Thread 1/4 started
Thread 3/4 started
Computation stopped after 1 iterations
```

3.2 Calcul de la bande attribuée à chaque thread

Maintenant que la création/terminaison des threads fonctionne, il nous faut faire en sorte que chaque thread calcule une zone distincte de l'image. Nous allons simplement découper l'image en `nb_threads` bandes horizontale, et faire en sorte que le thread 0 calcule la bande du haut, le thread 1 la bande juste en-dessous, etc.
Ajoutez des variables dans la fonction `thread_starter` pour calculer respectivement l'indice de la première ligne et le nombre de lignes que doit calculer chacun des threads. Chaque thread travaillera sur une bande d'épaisseur $\lfloor \frac{\texttt{DIM}}{\texttt{nb\_threads}} \rfloor$, sauf le dernier qui aura une bande un peu plus épaisse si la division ne tombe pas juste. Faites en sorte que chaque thread affiche l'intervalle de lignes qui lui revient. Exemple avec 6 threads :

```
[my-machine] OMP_NUM_THREADS=6 ./run -k spin -v thread
Using kernel [spin], variant [thread]
Thread 0/6 started, computing slice [   0- 169]
Thread 5/6 started, computing slice [ 850-1023]
Thread 2/6 started, computing slice [ 340- 509]
Thread 1/6 started, computing slice [ 170- 339]
Thread 4/6 started, computing slice [ 680- 849]
Thread 3/6 started, computing slice [ 510- 679]
Computation stopped after 1 iterations
```

Une fois que les affichages confirment que la distribution des bandes de l'image est correcte, remplacez le `printf` par un appel à `do_tile` et vérifiez que l'image s'affiche correctement !

3.3 Plusieurs itérations

Notre code n'effectue pour l'instant qu'une seule itération. Il faut maintenant :

- ajouter une boucle autour de `do_tile` pour que chaque thread effectue `nb_iter` itérations ;
- faire en sorte qu'un seul thread (par exemple celui de numéro 0) appelle `rotate ()` à la fin de chaque itération ;
- retourner 0 à la fin de la fonction `spin_compute_thread`.

Vérifiez. Ca marche ? Même en effectuant plusieurs itérations "de suite" entre deux affichages comme ceci ?

```
./run -k spin -v thread -r 4
```

3.4 Barrière de synchronisation

Le problème est un problème de synchronisation : le thread 0 doit d'abord attendre que tous les threads aient terminé leur itération avant d'appeler `rotate`. Réciproquement, tous les threads doivent ensuite attendre que `rotate` soit effectué avant d'enchainer une nouvelle itération.
Ajoutez des barrières de synchronisation dans `thread_starter` pour corriger le problème.
Faites varier le nombre de threads lancés. Vérifiez visuellement ce paramètre en utilisant l'option `--monitoring` (ou `-m`) :

```
OMP_NUM_THREADS=12 ./run -k spin -v thread -m
```

3.5 Mesure de l'accélération

Pour déterminer l'accélération obtenue, c'est-à-dire le gain obtenu en passant d'une version séquentielle à une version multithread, on mesure d'abord précisément le temps obtenu par chacune des versions en désactivant l'affichage. Par exemple :

```
[my-machine] ./run -k spin -v seq -i 200 -n
Using kernel [spin], variant [seq]
Computation completed after 200 iterations
6203.184
```

```
[my-machine] ./run -k spin -v thread -i 200 -n
Using kernel [spin], variant [thread]
Computation completed after 200 iterations
841.775
```

Sur cette machine, l'accélération est de $\frac{6203}{841} \approx 7.375$. Calculez celle obtenue au CREMI.

4 Calcul non-homogène sur l'image

Le noyau `mandel` affiche une représentation graphique de l'ensemble de Mandelbrot (https://fr.wikipedia.org/wiki/Ensemble_de_Mandelbrot) qui est une fractale définie comme l'ensemble des points du plan complexe pour lesquels les termes d'une suite ont un module borné par 2.
À chaque itération, le programme affiche une image dont les pixels ont une couleur qui dépend de la convergence de lq suite associée au point complexe du plan. Entre chaque itération, un (dé)zoom est appliqué afin de changer légèrement de point de vue.
Vous pouvez lancer le programme de la façon suivante :

```
./run -k mandel
```

Sans surprise, la fonction `mandel_compute_seq` se trouve dans le fichier `kernel/c/mandel.c`.

4.1 Découpage statique en bandes

Copiez votre implémentation multithread du noyau `spin` dans le fichier `mandel.c` et adaptez le code.
Examinez le comportement de votre implémentation, et notamment la répartition de la charge de calcul sur les threads :

```
./run -k mandel -m
```

Conclusion ?

4.2 Deux bandes par thread

Pour tenter de mieux équilibrer la charge dans cet exemple, on pourrait découper l'image en bandes deux fois moins épaisses, et faire en sorte que chaque thread prenne en charge deux bandes : une dans la moitié haute de l'image, et l'autre dans la moitié basse. Intuitivement, deux bandes symétriques par rapport à la médiane serait une bonne distribution.
Donnez une nouvelle implémentation « `thread_sym` » du noyau `mandel`. Observez la répartition des bandes de l'image :

```
./run -k mandel -v thread_sym -m
```

Comparez les performances avec la version précédente :

```
./run -k mandel -i 30 -n -v thread
# puis
./run -k mandel -i 30 -n -v thread_sym
```

4.3 Répartition cyclique des lignes

Une façon plus générique et plus fine de mieux répartir la charge de calcul est d'attribuer les lignes de façon cyclique. Si on lance P threads, le thread n°0 calcule les lignes 0, P, $2P$, etc. Le thread n°1 calcule les lignes 1, $P+1$, $2P+1$, etc.
Écrivez une implémentation sur ce principe nommée « `thread_cyclic`. »
Comparez les performances obtenues avec les versions précédentes. Faites varier le nombre de threads pour trouver la meilleure accélération possible par rapport à la version séquentielle.

4.4 Distribution dynamique des lignes

Pour maintenir les threads occupés le plus uniformément possible, on peut faire en sorte qu'ils calculent une ligne à la fois, et qu'ils viennent « chercher » la prochaine ligne à traiter à chaque fois qu'ils ont terminé une ligne. L'idée est donc de disposer d'un distributeur de lignes auprès duquel les threads vont s'adresser pour obtenir la prochaine ligne qu'ils doivent traiter.
Ce distributeur peut être implémenté par une fonction `unsigned get_next_line (void)`
À chaque appel, la fonction renvoie le numéro de la prochaine ligne non encore traitée : d'abord 0, puis 1, 2, 3, etc. Attention, même si plusieurs threads appellent la fonction en même temps, ils doivent chacun obtenir un numéro différent. Lorsque le numéro obtenu par un thread est supérieur ou égal à `DIM`, il peut se terminer.
Implémentez cette variante dynamique que vous nommerez `thread_dyn`.