TP 1 – Threads & Fractales

Note

Le premier exercice devrait être fait en 1h30.

Pour compiler : `g++ fichier.cpp -std=c++11 -pthread -o executable`

1 Premiers pas avec les threads

Vous devez paralléliser un petit programme : `lancement.cpp` disponible sur le site de l'UE. Ce dernier répète un certain nombre de fois un calcul qui n'a pas d'importance. Le but de l'exercice est de savoir si vous pouvez paralléliser les appels à la fonction `fct()`, récupérer le résultat (qui est le temps écoulé pendant le calcul d'après la fonction) et comparer ce temps avec le temps réellement écoulé.

Q1. Dans la fonction `main()` remplacez la boucle qui appelle `nbthreads` fois la fonction `fct()` par le lancement de `nbthreads` threads qui exécutent la fonction.

Q2. Si cela ne vous est pas déjà arrivé, faites une erreur d'argument entre le lancement du thread et la fonction, afin de vous habituer à lire les messages d'erreur de compilation. Par exemple, essayez un passage par référence en oubliant `std::ref()` au moment de l'appel : le message d'erreur est rarement convivial !

Q3. Normalement, dès qu'il y a un nombre de threads important, vous pouvez observer que les messages qu'ils affichent se mélangent. Sachant que les fonctions système sont prévues pour fonctionner en environnement multithread, pourquoi y a-t-il ce mélange et comment l'éviter ?

Q4. À la fin des threads obtenez le résultat de la fonction et affichez le temps que chaque thread pense avoir mis pour s'exécuter ainsi que la somme de ces valeurs.

La commande `time` permet de voir les différents temps utilisés par une commande :

```
$ time ./lancement 20
Th principal : lancement de 10 fois la fonction
0 lancement de la fonction pour 5 iterations
...
Th principal : Le temps total de calcul est 9.87181

real    0m9.876s
user    0m9.863s
sys     0m0.002s
```

- *real* est le temps écoulé pendant le calcul.
- *user* est la somme des temps processeur utilisés par les threads en mode utilisateur (pour faire les calculs eux-mêmes).
- *sys* est la somme des temps processeur passés en mode noyau (pour faire les opérations système comme les affichages).

Q5. Faites plusieurs essais avec 2, 3, 4, 8, 10, 20 threads.

a) Pourquoi le temps utilisateur est-il supérieur au temps réel ?

b) La somme des temps que vous calculez est-elle liée au temps réel ? Au temps utilisateur ? Au temps système ?

c) Est-elle identique, et s'il y a une différence pourquoi ?

Note

Pour la dernière question il faut comparer cela avec le nombre de processeurs/cores de l'ordinateur (`cat /proc/cpuinfo`).

A retenir

Voici ce qu'il faut bien retenir :

- option de compilation pour un code multi threadé ;
- création d'un thread qui va exécuter une fonction ;
- gestion des threads en utilisant un `std::vector<thread> tab_threads` ;
- gestion de la terminaison de chacun des threads lancés ;
- utilisation d'un tableau pour récupérer facilement le résultat de chacun des threads.

2 Calcul de fractales

2.1 Prise en main du code initial

Nous allons paralléliser le calcul d'une image fractale en découpant le travail (calcul de toute l'image) en petites tâches (calcul d'une tranche de l'image). Les threads lancés se partageront ainsi ces tâches. Le calcul de la fractale consiste à évaluer une fonction permettant de déterminer la couleur en chacun des pixels de l'image. Cette fonction mathématique n'est pas à considérer, elle est déjà écrite pour vous. Afin d'accélérer le calcul, nous vous demandons donc de faire réaliser ce travail par plusieurs threads. Chaque thread faisant un certain nombre de tranches d'écran.

- Téléchargez et décompressez l'archive `mandel.zip` se trouvant sur la page web de l'UE. Compilez le programme (`make`) puis lancez-le (`./mandel`).
  Vous verrez apparaître une fenêtre graphique sur laquelle se dessine progressivement la fractale de Mandelbrot. Pour l'instant, l'image est divisée en 10 tranches, et un seul thread calcule ces tranches, les unes après les autres (on peut voir à l'œil nu qu'il n'y a jamais plus d'une tranche en cours de calcul).
- Vous pouvez faire varier le nombre de tranches d'écran avec par exemple : `./mandel --nb-slices 100`.
  Les fichiers `display.*` et `mandel.*` gèrent respectivement l'affichage à l'écran et le calcul de la couleur de chaque pixel de la fractale. Vous n'aurez pas besoin de modifier ces fichiers.
- **Le fichier `main.cpp` est celui qui nous intéresse** : ouvrez-le et regardez son code, en particulier la fonction `main()` et `draw_screen_sequential()` qu'elle appelle par défaut.
  Si le calcul est trop rapide, vous pouvez le ralentir artificiellement avec `--slow` (répétez l'option plusieurs fois pour ralentir encore). Pour afficher plus de traces de debug, utilisez `--verbose`.
  Pour le plaisir des yeux, vous pouvez jouer avec les options `--loop`, `--resize` (zoomer sur la position du pointeur de la souris à chaque itération).

2.2 Principe de la parallélisation avec liste de tâches

Le temps nécessaire au calcul de chaque pixel n'est pas constant car il dépend des coordonnées du pixel considéré. *Il n'est donc pas efficace de donner le même nombre de tranches à chaque thread.*

Pour faire le calcul, vous devez **créer dès le départ un nombre fixé de threads** de calcul et une **liste des tâches à effectuer**. Chaque thread va devoir chercher dans la liste une tâche à faire, effectuer cette tâche et recommencer jusqu'à ce que la liste soit vide. La liste est un objet partagé et contient uniquement le numéro des tranches à traiter. Vous devrez faire bien attention à ce que toutes les tranches soient traitées une et une seule fois. Le code préparé contient plusieurs tests dont le but est de vérifier cela. Le principe de

la gestion de la liste de tâches vous est fourni dans la fonction `draw_screen_worker()`, qui va en boucle chercher du travail (`get_slice()`) puis effectuer la tâche correspondante (`compute_and_draw_slice()`), jusqu'à ce que `get_slice()` renvoie -1.

- Lisez le code de `draw_screen_worker()`. Essayez de remplacer temporairement l'appel à `draw_screen_sequential()` dans `main()` par `draw_screen_worker()` : le résultat est le même. Avec un seul travailleur, le calcul reste séquentiel. Après cette vérification, restaurez le code original avec un appel à `draw_screen_sequential()`.
- Le but de la suite du TP sera de lancer plusieurs travailleurs en parallèle, chacun exécutant la fonction `draw_screen_worker()`, tous les threads partageant la liste de tâches à exécuter.

  Lancez par exemple `./mandel --nb-threads 2`.

  Le programme s'arrête avec le message `draw_screen_thread not yet implemented`.

  Cela va être l'objectif de la suite du TP !

2.3 Implémentation à faire

Vous devez maintenant implémenter la fonction `draw_screen_thread()` pour suivre le schéma décrit dans le paragraphe précédent.

Q1. Modifier la fonction pour lancer `number_of_threads` threads, chacun exécutant `draw_screen_worker`

Q2. Regardez le mécanisme proposé dans le code pour distribuer les tranches d'écran aux threads : il s'agit de la fonction `get_slice()` qui utilise la variable globale, donc partagée, `last_slice` (dernière tranche calculée).

Q3. Faire en sorte que les threads de calcul traitent les tranches jusqu'à ce qu'il ne reste plus de tranche à calculer (fonction `get_slice()` est prévue pour cela et retourne -1 lorsque la liste est vide). Attention, la fonction `get_slice()` n'est pas prévue pour fonctionner en environnement multithread (elle n'est pas *thread safe*, et a même été écrite volontairement pour poser un maximum de problèmes en cas d'accès concurrent). Vous devez faire en sorte que le programme ne calcule pas plusieurs fois la même tranche, et que toute ces tranches soient bien calculées (si ce n'est pas le cas, cela est visible sur l'image obtenue).

Q4. À ce stade, le calcul est complet et doit afficher la fractale correctement. Modifiez-le maintenant pour que le thread principal obtienne le nombre de tranches calculées par chaque thread, faire la somme et vérifier que ce nombre correspond à celui des tranches de l'image (la vérification est faite pour vous dans `main()` à l'appel de `draw_screen_thread()`).

Q5. Nous vous avons fourni une fonction `draw_rect_thread_safe()` qui s'occupe de l'affichage d'un rectangle de l'écran de manière *thread-safe*, c'est à dire qui ne pose pas de problème lors d'accès concurrents. Essayez de remplacer cet appel par `draw_rect()`. Vous devriez obtenir un problème d'accès concurrent à l'affichage (la bibliothèque X11 va afficher une erreur). Cette erreur ne se produit pas sur certains systèmes, donc ne vous acharnez pas si vous ne l'avez pas. À vous de faire en sorte qu'il n'y ait plus d'accès concurrents à l'affichage (sans utiliser `draw_rect_thread_safe()`, qui est en fait le corrigé de cet exercice).

Q6. Vérifiez expérimentalement que le calcul est plus rapide avec plus d'un thread (`--nb-threads ...`). Comparez le nombre de threads optimal avec le nombre de cœurs de la machine (`/proc/cpuinfo`). Le nombre de tranches (`--nb-slices ...`) peut avoir une influence (avec trop peu de tranches, certains threads n'auront plus rien à faire en attendant que la dernière tranche soit calculée, avec trop de tranches on augmente le nombre d'appels à `get_slice()`).