TP - LIFPCA Programmation Concurrente et Administration Système

—

Calcul d'image en parallèle

Sylvain Brandel, Yves Caniou, Guillaume Damiand, Meriem Ghali, Laurent Lefèvre, Thibaut Modrzyk, Grégoire Pichon, Alec Sadler, Florence Zara, Jerry Lacmou Zeutouo

Printemps 2024

I Calcul de fractales

I.1 Présentation

Vous devez paralléliser le calcul d'une image fractale. Vous allez utiliser un modèle où le travail est découpé en petites tâches et où les threads lancés se partagent ces tâches.

Dans ce calcul de fractale, il faut évaluer une fonction de calcul de la couleur en chaque pixel 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.

I.2 Prise en main

Téléchargez et décompressez l'archive `mandel.zip` de la page du cours. 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).

I.3 Principe de la parallélisation avec liste de tâches

Le temps nécessaire au calcul de chaque pixel n'est pas constant et 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 est maintenant de lancer plusieurs travailleurs en parallèle, chacun exécutant la fonction `draw_screen_worker()`. Tous les threads partagent la liste de tâches à exécuter.

Lancez maintenant `./mandel --nb-threads 2`. Le programme s'arrête avec le message `draw_screen_thread not yet implemented`.

I.4 Travail à faire

Vous devez maintenant implémenter la fonction `draw_screen_thread()` pour suivre le schéma décrit dans le paragraphe précédent, c'est-à-dire :

— Modifier la fonction pour qu'elle lance `number_of_threads` threads, chacun exécutant `draw_screen_worker`.

> Cf. `draw_screen_threads` : 2 boucles for, chaque thread stocké dans un vecteur pour pouvoir appeler `join()` dessus.

— 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).

> Il n'y a rien à faire, mais notez que la fonction `get_slice()` n'est pas thread-safe. On a même renforcé le problème potentiel en ajoutant un `usleep` : en cas d'accès concurrents, il est possible que plusieurs threads accèdent à la même tranche et/ou que certaines tranches ne soient affectées à personne. Notre code met du rouge quand il détecte qu'un pixel est calculé deux fois (detection non-fiable, car quand ça arrive c'est qu'il y a race condition), et laisse du gris sur les zones qui ne sont pas calculées du tout.

— 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).

Le principe est déjà en place, *mais* il manque les verrouillages/déverrouillages de mutex autour de (ou à l'intérieur de) `get_slice()`.

— 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()`).

Attention à l'utilisation de `std::ref()`, ou passage par pointeur obligatoire.

— 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)

1 mutex à verrouiller autour de `draw_rect()`. On pourrait aussi utiliser `XInitThreads()` pour rendre la bibliothèque X11 thread-safe, mais ce n'est pas le but ici.

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()`).

Si vous avez bien préparé ce TP, vous devriez terminer avant la fin : démarrez le TP3 qui est la suite de celui-ci (mise en place de producteur-consommateur sur ce calcul de fractale).

II Extrait du corrigé complet

```
1   void compute_and_draw_slice(int slice_number) {
2           int y;
3           bool warning_emitted = false;
4
5           if (verbose > 0)
6                   std::cout << "Starting slice " << slice_number << std::endl;
7           for (y = 0; y < height; y += rect_height) {
8                   compute_rect(slice_number, y, warning_emitted);
9                   /* Version where we manage the mutex explicitly. */
10                  xmutex.lock();
```

```
11                      draw_rect(slice_number, y);
12                      xmutex.unlock();
13              }
14              if (verbose > 0)
15                      std::cout << "Finished slice " << slice_number << std::endl;
16      }
17
18      /* Protect concurrent accesses to the set of slices to process */
19      std::mutex last_slice_mutex;
20
21
22      void draw_screen_worker(int & nb_slices) {
23              while (1) {
24                      last_slice_mutex.lock();
25                      int i = get_slice();
26                      last_slice_mutex.unlock();
27                      if (i == -1)
28                              return;
29                      nb_slices++;
30                      compute_and_draw_slice(i);
31              }
32      }
33
34      void draw_screen_worker_prodcons(int & nb_slices, ProdCons<rect> &fifo_x11, ProdCons<int> &
35              while (1) {
```