TP 2 – Producteur-Consommateur & Fractales

Le but de ce TP est de modifier le code du TP1 concernant le calcul de fractale de Mandelbrot en parallèle afin de **mettre en place un système de producteur-consommateur**.

Note

- Si vous avez bien réussi le TP1, vous pouvez continuer en utilisant votre code.
- Sinon, commencez par récupérer l'archive `mandel-threads.zip` sur la page de l'UE.
- Compilez et exécutez le code et vérifiez qu'il s'agit bien du travail demandé au TP1.

Au TP1, nous avions utilisé un **mutex** pour empêcher les accès concurrents à l'affichage.

Dans ce TP2, nous allons **dédier un thread à l'affichage** (le thread s'occupera d'appeler `draw_rect()`). Ce thread sera le seul à y accéder, donc il n'y aura plus de problème d'exclusion mutuelle.

Les threads de calcul (les `draw_screen_worker` du TP1) n'appelleront plus `draw_rect()`, mais ils enverront le rectangle d'écran à afficher au thread chargé de l'affichage.

La communication entre les threads de calcul et le thread d'affichage se fera au moyen d'un **schema « producteur-consommateur »**, comme vu en TD. Il s'agit d'une classe C++ (un moniteur), qui expose les méthodes suivantes :

- `void put(element)` : appelée par le ou les producteurs pour envoyer un élément au consommateur. Cette fonction est bloquante si la file est pleine.
- `element get()` : appelée par le ou les consommateurs pour récupérer un élément envoyé par un producteur. Cette fonction est bloquante si la file est vide.

Les éléments échangés sont des instances de la structure suivante qui décrit un rectangle à afficher :

```
struct rect {
        int slice_number;
        int y_start;
        rect(int sn, int y) : slice_number(sn), y_start(y) {};
};
```

Les threads de calculs enverront les rectangles avec :

```
prod_cons.put(rect(slice_number, y));
```

Le thread chargé de l'affichage récupérera ces valeurs pour appeler `draw_rect()` dessus avec :

```
rect r = prod_cons.get();
```

1 Une classe pour le producteur-consommateur

Q1. Dans de nouveaux fichiers C++ (`ProdCons.h` et `.cpp`), créez une classe `ProdCons` contenant les deux méthodes `put()` et `get()` décrites ci-dessus. Si vous êtes à l'aise en C++, écrivez une classe template pour que les arguments de `put()` et `get()` soient génériques.

Q2. Ajoutez les données nécessaires pour implémenter une file d'attente (FIFO) dans la classe. En TD, nous avons codé un buffer circulaire à la main avec un tableau ; vous pouvez aussi utiliser une structure toute faite de C++ comme `std::list` ou `std::queue` (vous aurez besoin des méthodes `push()`, `pop()`, `front()` et `size()`).

Q3. Ajoutez le nécessaire pour que cette classe puisse agir comme un moniteur de Hoare.

Q4. Donnez le corps des fonctions `put()` et `get()`.

2 Emploi de la classe pour le producteur-consommateur

Utilisez maintenant votre classe dans le code de calcul de Mandelbrot. Notez que plusieurs noms de fonctions ne refléteront plus exactement ce que les fonctions en question effectuent...

Q1. Instanciez votre classe `ProdCons`. Le plus propre est de l'instancier dans `draw_screen_thread()`, mais vous pouvez aussi le faire en variable globale pour simplifier.

Q2. Écrivez la fonction `x_thread_function()` qui sera exécutée par le thread chargé de l'affichage. Cette fonction doit avoir accès à l'instance de `ProdCons`. Son rôle se limitera à récupérer les éléments contenus dans le `ProdCons`, puis à les afficher.

Q3. Modifier la fonction `draw_screen_worker()` qui sera exécutée par les threads de calculs pour envoyer un rectangle d'écran à afficher au thread d'affichage. Cette fonction doit également avoir accès à l'instance de `ProdCons`.

Q4. Dans la fonction `draw_screen_thread()`, lancez le thread d'affichage en début de calcul. À quel moment pouvez-vous faire un `join()` sur le thread ? Immédiatement ? Plus tard ? Pourquoi ?

Q5. Modifiez la fonction `compute_and_draw_slice()` pour lui faire envoyer les rectangles au thread chargé de l'affichage.

Q6. Une problématique intéressante est celle de la terminaison : comment la fonction `x_thread_function()` est-elle en mesure de déterminer qu'elle a affiché la totalité de la fractale ? Modifier votre code en conséquence, si vous ne l'aviez pas pris en compte.

Q7. Testez l'ensemble de votre code !

Q8. Notre classe `ProdCons` peut aussi être utilisée pour gérer la liste des tâches (la fonction `get_slice()` que nous avons utilisée lors du TP1).
On peut alors instancier un thread chargé d'appeler `get_slice()` et d'envoyer les tranches d'écrans à calculer aux threads de calcul. De cette manière, le thread producteur sera le seul à appeler `get_slice()` (on pourra alors supprimer le mutex qui protégeait son accès).

Si le temps le permet, mettez en place ce producteur-consommateur.

3 Un peu de recul

Cette section est destinée à vous faire réfléchir à ce que vous venez de coder. Son objectif n'est pas pratique, mais réflexif. Comprendre pourquoi vous utilisez un outil plutôt qu'un autre est essentiel. Il n'est pas obligatoire de répondre à ces questions pendant le TP, mais nous vous recommandons de leur consacrer un peu de votre temps à un moment ou un autre.

- Pourquoi utilise-t-on une structure `rect` ? Pourquoi ne pas simplement utiliser deux producteurs-consommateurs d'`int` ? Interrogez-vous sur l'utilité d'une structure de données, l'utilité d'une classe.
- Cette version du producteur-consommateur place les producteurs et les consommateurs en exclusion mutuelle. Il est impossible de consommer en même tant qu'une production, et vice-versa. Cette restriction est-elle inévitable ? Après-tout, les premières valeurs dans la file du producteur-consommateur ne vont pas changer en raison d'une insertion, pour peu qu'on utilise une structure de données qui supporte l'insertion sans déplacer toutes ses valeurs (une liste chaînée par exemple). Pouvez-vous ébaucher une variante de `ProdCons` dans laquelle il est possible de lire une valeur tout en produisant une autre valeur, du moment qu'il y a au moins une valeur de disponible dans le `ProdCons` ?