

# TP - ASR7 Programmation Concurrente

## Producteur-Consommateur

Sylvain Brandel, Guillaume Damiand, Laurent Lefèvre, Matthieu Moy, Grégoire Pichon

Printemps 2022

### I Producteur-Consommateur

Le but de ce TP est de modifier le code du TP2 (calcul de fractale de Mandelbrot en parallèle) pour mettre en place un système de producteur-consommateur.

Si vous avez bien réussi le TP2, vous pouvez continuer en utilisant votre code. Sinon, commencez par récupérer l'archive `mandel-threads.zip` sur la page du cours. Compilez et exécutez le code qu'elle contient et vérifiez qu'il s'agit bien du travail demandé au TP2.

Au TP2, nous avons utilisé un mutex pour empêcher les accès concurrents à l'affichage. Cette fois-ci, 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 TP2, n'appelleront plus `draw_rect()`, mais 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 schéma « 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 :

```

1  struct rect {
2      int slice_number;
3      int y_start;
4      rect(int sn, int y) : slice_number(sn), y_start(y) {};
5  };

```

Les threads de calculs enverront les rectangles, par exemple avec :

```

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

```

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

```

1  rect r = prod_cons.get();

```

## I.1 Travail à réaliser

### I.1.1 Une classe pour le producteur-consommateur

- Créez un nouveau fichier C++ dans lequel vous allez écrire le producteur-consommateur. 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.
- Ajoutez le nécessaire pour que cette classe puisse agir comme un moniteur.
- 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 mieux, `std::queue` (vous aurez besoin des méthodes `push()`, `pop()`, `front()` et `size()`).
- Donnez le corps des fonctions `put()` et `get()`.

Cf. les fichiers `ProdCons.h` et `ProdCons.tpp` dans l'archive `tp3/mandel_corr.zip`.

### I.1.2 Utiliser 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, ne vous en étonnez pas.

- Instanciez votre classe `ProdCons`. Le plus propre est de l'instancier dans la fonction `draw_screen_thread()`, mais vous pouvez aussi le faire en variable globale pour simplifier.
- 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.
- 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?
- Modifiez la fonction `compute_and_draw_slice()` pour lui faire envoyer les rectangles au thread chargé de l'affichage. `x_thread_function`.

### I.1.3 Achever son œuvre

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?

- Une première intuition, naïve, serait de se dire qu'il suffit d'attendre que le `ProdCons` soit vide. Cette intuition n'est cependant pas correcte. Pourquoi donc?
- Au vu de cette observation, il faut apporter une information à la fonction `x_thread_function()` afin de lui permettre de détecter la terminaison. Les solutions sont nombreuses. Voici une piste de réflexion : rappelez-vous que pour détecter la fin

d'une liste chaînée en C(++), on exploite le fait que la dernière cellule de la liste ne possède pas de pointeur vers une cellule suivante. C'est le concept de la sentinelle, une valeur qui marque la terminaison. Ce concept peut-il être appliqué au producteur-consommateur pour indiquer la terminaison ? Si oui, essayez de le mettre en place. Si non, pourquoi donc et comment est-il possible de l'adapter pour résoudre la problématique ?

- Il existe de nombreuses solutions et variantes de ces solutions pour ce problème, qui est très classique. Pouvez-vous en proposer d'autres ? Quels sont leurs avantages et inconvénients ?

Cf. le fichier `main.cpp`.

Testez l'ensemble !

#### I.1.4 Pistes de réflexion

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` ?

- Une structure de données, au même titre qu'une classe, sert à rassembler des données, des valeurs, qui contribuent ensemble à former un tout cohérent. Les classes ajoutent la possibilité d'avoir des méthodes qui opèrent sur ces données, en effaçant les détails d'implémentation pour le programmeur. Il ignore comment fonctionne la méthode, mais il sait quelle est son utilité et quel est son résultat. Séparer les composants d'un `rect` entre deux producteurs-consommateurs scinde l'aspect unifié de la structure. D'une part, cela réduit la compréhension du code, un numéro de tranche seul est moins représentatif que lorsqu'il est accompagné d'une hauteur. D'autre part, cela pose des problèmes de concurrence assez désagréables à résoudre. Interrogez-vous sur quels problèmes peuvent survenir.
- Une solution à cette question serait trop longue pour être détaillée ici, voici donc quelques idées. Tout d'abord, une telle structure est possible et repose sur l'idée d'avoir deux vues différentes sur le contenu du producteur-consommateur : une vue sur l'endroit où écrire

la prochaine valeur produite et une vue sur l'endroit où récupérer la prochaine valeur à consommer. Tant que ces deux vues ne sont pas en collision, il n'y a pas de problème. Toute l'astuce consiste à faire en sorte que ces deux vues ne soient manipulées que par les bons threads : seuls les producteurs doivent modifier la vue sur la production, seuls les consommateurs doivent modifier la vue sur la consommation. L'absence de partage de données permet de supprimer le mutex qui sépare les producteurs et les consommateurs. Un problème survient toutefois lorsque les vues sont en collision : le consommateur doit attendre. L'absence de mutex partagé entre les producteurs et les consommateurs rend l'attente plus subtile à traiter que dans le cas simple. Quelle(s) primitive(s) de synchronisation vue(s) en cours peu(ven)t être utilisée pour permettre l'attente ?

## II Gérer la liste des tâches avec un producteur-consommateur

Notre classe `ProdCons` peut aussi être utilisée pour gérer la liste des tâches (la fonction `get_slice()` que nous avons utilisée lors du TP2) : on peut 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()` (donc on pourra supprimer le mutex qui protégeait son accès).

Si le temps le permet, mettez en place ce producteur-consommateur en plus de celui de la section précédente.