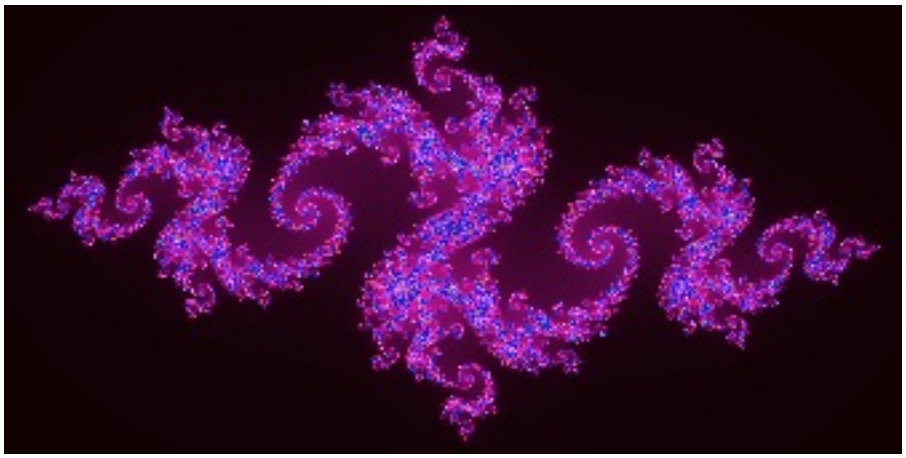
# TP2: Julia, threads, et répartition de charge

## Programmation multi-threadée

### ★ Exercice 1: La fractale de Julia



L'objectif de cet exercice est de calculer efficacement une représentation de l'ensemble de Julia. On explorera plusieurs façon de répartir les calculs entre plusieurs threads.

▷ **L'ensemble de Julia.** Il s'agit d'une fractale, c'est-à-dire un ensemble présentant des auto-similarités à différentes échelles (lorsque l'on zoome sur la figure, on retrouve les mêmes motifs à différentes échelles). C'est une généralisation de l'ensemble de Mandelbrot plus célèbre. On trouve de nombreuses visualisations psychédéliques de ces ensembles sur Internet, par exemple sur wikipédia.

Trouver la représentation graphique de l'ensemble de Julia demande des calculs relativement intensifs. Pour chaque point du plan complexe noté $z$, on calcule les termes successifs de la série $z_i$ définie ci-dessous. La couleur du pixel correspondant à $z$ dépend du nombre d'itérations nécessaires pour que $|z_n| > 2$.

$$\begin{cases} z_0 = z \\ z_{n+1} = z_n^2 + c \text{ (avec } c = -0.79 + 0.15i \text{ — d'autres valeurs donnent d'autres images)} \end{cases}$$

▷ **Format d'images BMP et matrices en mémoire.** Le format d'images BMP est l'un des plus simple à réaliser, puisque la matrice de pixels est stockée en l'état et sans compression après l'entête de fichier. Chaque pixel de l'image est alors représenté par 3 entiers R, G et B, représentant les composantes rouge, verte et bleue du pixel. Les lignes sont représentées les unes après les autres dans le fichier, en commençant par celle du bas de l'image. Il s'agit donc d'une représentation par lignes *(row-major)* de la matrice des pixels.

Dans le cas général, la position de la case $(x, y)$ dans un tableau stocké sous la forme *row-major* est `tab[y*width + x]`. Il faut adapter légèrement la formule dans notre cas, puisque chaque case du tableau fait trois octets au lieu d'un seul.

▷ **Code fourni.** Trois fichiers source sont fournis avec ce sujet.

- `julia/compute_julia_pixel.c`: Code de la fonction C du même nom, dont l'objectif est de calculer les composantes R, G et B du pixel passé en paramètre. Pour pouvoir l'utiliser, il faut avoir réservé suffisament de mémoire pour stocker la matrice des pixels de l'image, et passer la bonne case en paramètre à cette fonction.
- `julia/write_bmp_header.c`: Cette fonction se charge d'écrire une matrice de pixels sur disque sous forme d'un fichier au format BMP.
- `julia/CMakeLists.txt`: Le projet cmake, permettant de compiler l'ensemble.

▷ **Question 1:** Complétez le projet en écrivant un fichier `julia.c` qui se charge d'allouer suffisament de mémoire pour stocker l'image. La résolution verticale $n$ sera passée en paramètre au programme, et la résolution horizontale sera $2n$. La taille du tableau à allouer est donc de $n \times (2n) \times 3$ pour stocker 3 octets par pixel.

Nous allons maintenant utiliser tous les threads de l'ordinateur pour réaliser le calcul. Le code ci-contre permet de retrouver le nombre de coeurs utilisables. Il fonctionne sous Linux et sous le Windows Subsystem Linux (WSL).

```
int cpucount() {
  cpu_set_t cpuset;
  sched_getaffinity(0, sizeof(cpuset), &cpuset);
  return CPU_COUNT(&cpuset);
}
```

▷ **Question 2: Implémentez un découpage 1D** des calculs à réaliser. Pour cela, attribuez une bande de hauteur $\frac{height}{nb\ threads}$ pixels à chaque thread. Votre programme devrait aller significativement plus vite. Si vous ne voyez pas la différence, augmentez la taille de l'image.

▷ **Question 3: Implémentez maintenant un découpage 2D** de l'image à calculer. Le plus simple est de découper l'image à calculer sur une matrice carrée, par exemple de taille $2 \times 2$ ou $3 \times 3$. Mais si le nombre de coeurs de votre ordinateur n'est pas un carré parfait, utilisez une autre géométrie de grille de threads. La forme de la grille ne devrait pas avoir d'impact sur les performances de votre programme.

▷ **Question 4:** Le problème des répartitions 1D et 2D réalisées jusqu'à présent est que la quantité de travail n'est pas la même pour tous les threads. Aux bordures de l'image, une itération ou deux suffisent à atteindre la condition $|z_n| > 2$, et le bloc est très vite colorié en noir. Au coeur de la fractale en revanche, chaque pixel demande 300 itérations avant d'être colorié en blanc. **Il faut implémenter une répartition dynamique.**

Certains problèmes nécessitent des stratégies compliquées dites de vol de travail. Dans notre cas, la meilleure solution est de découper le travail en parties plus petites, et stocker la liste de choses à faire dans une structure partagée entre tous les threads.

On découpera par exemple l'image en 100 bandes distinctes à faire dans différents threads. Ensuite, une variable partagée `next` dénote de la prochaine bande à calculer (un simple entier suffit pour cela). Comme cette variable est partagée, un mutex est nécessaire pour synchroniser les manipulations de cette variable.

Ensuite, tant qu'il reste du travail à faire, chaque thread acquière une tranche de l'image et la calcule. Le pseudo-code est détaillé ci-contre.

```
Init:
  int next = 0
  mutex mut

Thread:
  fini = false
  ma_bande = 0
  while not fini:
    mutex_lock(mut)
    if next == 100:
      fini = true
    else:
      ma_bande = next
      next++
    mutex_unlock(mut)
    if not fini:
      calcule la bande ma_bande
```

▷ **Question 5:** Implémentez une répartition dynamique 2D. Explorez le compromis sur la taille des blocs : des blocs trop petits induisent des coûts de synchronisation trop importants, tandis que des blocs trop gros induisent une mauvaise répartition des calculs.