Paralelização da Geração de Fractais

Você recebeu um código Python que gera diversos fractais (Triângulo de Sierpinski, Samambaia de Barnsley, Conjunto de Mandelbrot, Conjunto de Julia, Curva de Koch, Árvore Fractal, Tapete de Sierpinski e Esponja de Menger). Cada fractal é gerado sequencialmente, o que pode levar algum tempo, especialmente para fractais mais complexos como o Conjunto de Mandelbrot e o Conjunto de Julia.

Para melhorar o desempenho do programa, você deve paralelizar a geração dos fractais usando threads . O objetivo é permitir que múltiplos fractais sejam gerados simultaneamente, aproveitando melhor os recursos computacionais disponíveis.

Objetivo

Modifique o código fornecido para implementar a geração dos fractais em threads separadas . Cada fractal deve ser gerado em uma thread independente, e as imagens resultantes devem ser salvas na área de trabalho, como no código original.

Requisitos

1. Use a biblioteca threading do Python para criar threads.
2. Cada fractal deve ser gerado em uma thread separada.
3. Garanta que todas as threads terminem antes de finalizar o programa.
4. Certifique-se de que os arquivos de imagem sejam salvos corretamente sem conflitos.
5. Mantenha a estrutura modular do código original, criando uma função para cada fractal.
6. Adicione uma função principal (main) que inicie as threads e aguarde sua conclusão.
7. Os arquivos de imagem gerados devem ser salvos na área de trabalho , como no código original.
8. O nome dos arquivos deve seguir o padrão: nome_do_fractal.png.
9. Verifique se todos os fractais são gerados corretamente e salvos na área de trabalho.
10. Meça o tempo de execução com e sem threads para comparar o desempenho.

Código a ser paralelizado

```python
import matplotlib.pyplot as plt
import numpy as np
import random
import os

# Função para gerar fractais usando um Sistema de Funções Iteradas (IFS)
def gerar_fractal(transformacoes, probabilidades, iteracoes=100000):
    if not abs(sum(probabilidades) - 1.0) < 1e-6:
        raise ValueError("As probabilidades devem somar 1.")

    x, y = 0.0, 0.0
    pontos = []

    for _ in range(iteracoes):
        r = random.random()
        acumulado = 0.0
        for i, prob in enumerate(probabilidades):
            acumulado += prob
            if r < acumulado:
                transformacao = transformacoes[i]
                break
```

```
            x, y = transformacao(x, y)
            pontos.append((x, y))

    return pontos

# Triângulo de Sierpinski
def sierpinski():
    transformacoes = [
        lambda x, y: (0.5 * x, 0.5 * y),
        lambda x, y: (0.5 * x + 0.5, 0.5 * y),
        lambda x, y: (0.5 * x + 0.25, 0.5 * y + 0.5)
    ]
    probabilidades = [1/3, 1/3, 1/3]
    pontos = gerar_fractal(transformacoes, probabilidades, iteracoes=100000)

    x_vals, y_vals = zip(*pontos)
    plt.figure()
    plt.scatter(x_vals, y_vals, s=0.1, color='black', marker='.')
    plt.title("Triângulo de Sierpinski")
    plt.axis('off')
    plt.savefig(os.path.join(os.path.expanduser("~"), "Desktop", "sierpinski.png"), bbox_inches='tight', dpi=300)
    plt.close()

# Samambaia de Barnsley
def samambaia_barnsley():
    transformacoes = [
        lambda x, y: (0.0, 0.16 * y),
        lambda x, y: (0.85 * x + 0.04 * y, -0.04 * x + 0.85 * y + 1.6),
        lambda x, y: (0.2 * x - 0.26 * y, 0.23 * x + 0.22 * y + 1.6),
        lambda x, y: (-0.15 * x + 0.28 * y, 0.26 * x + 0.24 * y + 0.44)
    ]
    probabilidades = [0.01, 0.85, 0.07, 0.07]
    pontos = gerar_fractal(transformacoes, probabilidades, iteracoes=100000)

    x_vals, y_vals = zip(*pontos)
    plt.figure()
    plt.scatter(x_vals, y_vals, s=0.1, color='green', marker='.')
    plt.title("Samambaia de Barnsley")
    plt.axis('off')
    plt.savefig(os.path.join(os.path.expanduser("~"), "Desktop", "samambaia_barnsley.png"), bbox_inches='tight', dpi=300)
    plt.close()

# Conjunto de Mandelbrot
def mandelbrot(width=800, height=800, max_iter=100):
    x_min, x_max = -2.0, 1.0
    y_min, y_max = -1.5, 1.5
    image = np.zeros((height, width))

    for row in range(height):
        for col in range(width):
            c = complex(x_min + (x_max - x_min) * col / width,
                        y_min + (y_max - y_min) * row / height)
            z = 0.0j
            n = 0
            while abs(z) <= 2 and n < max_iter:
                z = z * z + c
                n += 1
            image[row, col] = n

    plt.figure()
    plt.imshow(image, extent=(x_min, x_max, y_min, y_max), cmap='hot', interpolation='bilinear')
    plt.title("Conjunto de Mandelbrot")
    plt.axis('off')
    plt.savefig(os.path.join(os.path.expanduser("~"), "Desktop", "mandelbrot.png"), bbox_inches='tight', dpi=300)
    plt.close()
```

```python
# Conjunto de Julia
def julia(c=-0.7 + 0.27015j, width=800, height=800, max_iter=100):
    x_min, x_max = -1.5, 1.5
    y_min, y_max = -1.5, 1.5
    image = np.zeros((height, width))
    for row in range(height):
        for col in range(width):
            z = complex(x_min + (x_max - x_min) * col / width,
                        y_min + (y_max - y_min) * row / height)
            n = 0
            while abs(z) <= 2 and n < max_iter:
                z = z * z + c
                n += 1
            image[row, col] = n
    plt.figure()
    plt.imshow(image, extent=(x_min, x_max, y_min, y_max), cmap='twilight_shifted', interpolation='bilinear')
    plt.title("Conjunto de Julia")
    plt.axis('off')
    plt.savefig(os.path.join(os.path.expanduser("~"), "Desktop", "julia.png"), bbox_inches='tight', dpi=300)
    plt.close()

# Curva de Koch (usando matplotlib)
def koch_curve(order=4, size=300):
    def koch_curve_recursive(points, order):
        if order == 0:
            return points
        new_points = []
        for i in range(len(points) - 1):
            p1, p2 = points[i], points[i + 1]
            dx, dy = p2[0] - p1[0], p2[1] - p1[1]
            new_points.append(p1)
            new_points.append((p1[0] + dx / 3, p1[1] + dy / 3))
            new_points.append((p1[0] + dx / 2 - dy * np.sqrt(3) / 6, p1[1] + dy / 2 + dx * np.sqrt(3) / 6))
            new_points.append((p1[0] + 2 * dx / 3, p1[1] + 2 * dy / 3))
        new_points.append(points[-1])
        return koch_curve_recursive(new_points, order - 1)

    points = [(0, 0), (size, 0)]
    points = koch_curve_recursive(points, order)

    x_vals, y_vals = zip(*points)
    plt.figure()
    plt.plot(x_vals, y_vals, color='blue', linewidth=1)
    plt.title("Curva de Koch")
    plt.axis('equal')
    plt.axis('off')
    plt.savefig(os.path.join(os.path.expanduser("~"), "Desktop", "koch_curve.png"), bbox_inches='tight', dpi=300)
    plt.close()

# Árvore Fractal (usando matplotlib)
def fractal_tree():
    def draw_tree(ax, x, y, length, angle, depth):
        if depth == 0:
            return
        x_end = x + length * np.cos(np.radians(angle))
        y_end = y + length * np.sin(np.radians(angle))
        ax.plot([x, x_end], [y, y_end], color='brown', linewidth=1)
        draw_tree(ax, x_end, y_end, length * 0.7, angle - 30, depth - 1)
        draw_tree(ax, x_end, y_end, length * 0.7, angle + 30, depth - 1)

    fig, ax = plt.subplots()
    draw_tree(ax, 0, 0, 100, 90, 8)
    plt.title("Árvore Fractal")
    plt.axis('equal')
    plt.axis('off')
    plt.savefig(os.path.join(os.path.expanduser("~"), "Desktop", "fractal_tree.png"), bbox_inches='tight', dpi=300)
    plt.close()
```

```
# Tapete de Sierpinski
def sierpinski_carpet(size=3, iterations=4):
    carpet = np.ones((size**iterations, size**iterations))

    def recursive_remove(grid, x, y, size, iteration):
        if iteration == 0:
            return
        sub_size = size // 3
        for i in range(3):
            for j in range(3):
                if i == 1 and j == 1:
                    grid[x + sub_size:x + 2 * sub_size, y + sub_size:y + 2 * sub_size] = 0
                else:
                    recursive_remove(grid, x + i * sub_size, y + j * sub_size, sub_size, iteration - 1)
    recursive_remove(carpet, 0, 0, size**iterations, iterations)
    plt.figure()
    plt.imshow(carpet, cmap='gray_r')
    plt.title("Tapete de Sierpinski")
    plt.axis('off')
    plt.savefig(os.path.join(os.path.expanduser("~"), "Desktop", "sierpinski_carpet.png"), bbox_inches='tight', dpi=300)
    plt.close()

# Esponja de Menger
def menger_sponge(iterations=2):
    def generate_sponge(grid, x, y, z, size, iteration):
        if iteration == 0:
            return
        sub_size = size // 3
        for i in range(3):
            for j in range(3):
                for k in range(3):
                    if i == 1 and j == 1 or i == 1 and k == 1 or j == 1 and k == 1:
                        continue
                    generate_sponge(grid, x + i * sub_size, y + j * sub_size, z + k * sub_size, sub_size, iteration - 1)
    grid_size = 3**iterations
    grid = np.ones((grid_size, grid_size, grid_size))
    generate_sponge(grid, 0, 0, 0, grid_size, iterations)

    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.voxels(grid, edgecolor='k')
    plt.title("Esponja de Menger")
    plt.savefig(os.path.join(os.path.expanduser("~"), "Desktop", "menger_sponge.png"), bbox_inches='tight', dpi=300)
    plt.close()

# Função para gerar todos os fractais
def gerar_todos_fractais():
    print("Gerando Triângulo de Sierpinski...")
    sierpinski()
    print("Gerando Samambaia de Barnsley...")
    samambaia_barnsley()
    print("Gerando Conjunto de Mandelbrot...")
    mandelbrot()
    print("Gerando Conjunto de Julia...")
    julia()
    print("Gerando Curva de Koch...")
    koch_curve()
    print("Gerando Árvore Fractal...")
    fractal_tree()
    print("Gerando Tapete de Sierpinski...")
    sierpinski_carpet()
    print("Gerando Esponja de Menger...")
    menger_sponge()
# Executa a geração de todos os fractais
if __name__ == "__main__":
    gerar_todos_fractais()
```