```python
import numpy as np

def sphere_function(position):
    """
    Objective function to minimize.
    Sphere Function: f(x) = sum(x_i^2)
    """
    return np.sum(position**2)

def initialize_population(grid_size, solution_dim, lower_bound, upper_bound):
    """
    Initialize the cellular grid with random positions in the solution space.
    Each cell is assigned a random position (vector).
    """
    grid = np.random.uniform(lower_bound, upper_bound, size=(grid_size, grid_size, solution_dim))
    return grid

def evaluate_fitness(grid):
    """
    Evaluate the fitness of each cell in the grid based on the optimization function.
    """
    fitness = np.apply_along_axis(sphere_function, 2, grid)
    return fitness

def get_neighbors(grid, i, j):
    """
    Get the neighboring cells of cell (i, j) in the grid.
    Wraps around the grid edges (toroidal topology).
    """
    neighbors = []
    grid_size = len(grid)
    for di in [-1, 0, 1]:
        for dj in [-1, 0, 1]:
            if di != 0 or dj != 0:  # Exclude the cell itself
                ni, nj = (i + di) % grid_size, (j + dj) % grid_size
                neighbors.append(grid[ni, nj])
    return np.array(neighbors)

def update_states(grid, fitness, learning_rate):
    """
    Update the state (position) of each cell based on the neighbors and predefined rules.
    Each cell moves towards the best position in its neighborhood.
    """
    grid_size, _, solution_dim = grid.shape
    new_grid = np.copy(grid)
    for i in range(grid_size):
        for j in range(grid_size):
            neighbors = get_neighbors(grid, i, j)
            neighbor_fitness = np.array([sphere_function(n) for n in neighbors])
            best_neighbor = neighbors[np.argmin(neighbor_fitness)]
            # Move cell slightly towards the best neighbor's position
            new_grid[i, j] += learning_rate * (best_neighbor - grid[i, j])
    return new_grid
def parallel_cellular_algorithm(
    grid_size=10, solution_dim=2, lower_bound=-5.0, upper_bound=5.0,
    iterations=100, learning_rate=0.1):
    """
    Main function to execute the Parallel Cellular Algorithm.
    """
    # Step 1: Initialize population
    grid = initialize_population(grid_size, solution_dim, lower_bound, upper_bound)
    best_solution = None
    best_fitness = float('inf')

    for iteration in range(iterations):
        # Step 2: Evaluate fitness
        fitness = evaluate_fitness(grid)

        # Track the best solution
        min_idx = np.unravel_index(np.argmin(fitness), fitness.shape)
        current_best = grid[min_idx]
        current_fitness = fitness[min_idx]
        if current_fitness < best_fitness:
            best_solution = current_best
            best_fitness = current_fitness
```

```
        # Step 3: Update states
        grid = update_states(grid, fitness, learning_rate)

        # Print iteration progress
        print(f"Iteration {iteration+1}/{iterations}: Best Fitness = {best_fitness:.5f}")

    # Step 4: Output the best solution
    print("\nOptimization Complete.")
    print(f"Best Solution: {best_solution}")
    print(f"Best Fitness: {best_fitness:.5f}")

# Run the algorithm
if __name__ == "__main__":
    parallel_cellular_algorithm(grid_size=10, solution_dim=2, iterations=10, learning_rate=0.2)
```

```
Iteration 1/10: Best Fitness = 0.33756
Iteration 2/10: Best Fitness = 0.10905
Iteration 3/10: Best Fitness = 0.10905
Iteration 4/10: Best Fitness = 0.08522
Iteration 5/10: Best Fitness = 0.03044
Iteration 6/10: Best Fitness = 0.02045
Iteration 7/10: Best Fitness = 0.02045
Iteration 8/10: Best Fitness = 0.02045
Iteration 9/10: Best Fitness = 0.00533
Iteration 10/10: Best Fitness = 0.00210

Optimization Complete.
Best Solution: [-0.04568353 -0.00346367]
Best Fitness: 0.00210
```

```
pip install opencv-python scikit-learn matplotlib joblib
```

```
Requirement already satisfied: opencv-python in /usr/local/lib/python3.10/dist-packages (4.10.0.84)
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.10/dist-packages (1.6.0)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.10/dist-packages (3.8.0)
Requirement already satisfied: joblib in /usr/local/lib/python3.10/dist-packages (1.4.2)
Requirement already satisfied: numpy>=1.21.2 in /usr/local/lib/python3.10/dist-packages (from opencv-python) (1.26.4)
Requirement already satisfied: scipy>=1.6.0 in /usr/local/lib/python3.10/dist-packages (from scikit-learn) (1.13.1)
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.10/dist-packages (from scikit-learn) (3.5.0)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (1.3.1)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (4.55.3)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (1.4.7)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (24.2)
Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (11.0.0)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (3.2.0)
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (2.8.2)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.7->matplotlib) (1.17.0)
```

```
import cv2
import numpy as np
from sklearn.cluster import KMeans
from joblib import Parallel, delayed
import matplotlib.pyplot as plt

# Function to segment an image based on k-means clustering
def kmeans_segmentation(image, k=3):
    # Reshape the image into a 2D array of pixels (each pixel has 3 values for RGB)
    image_reshaped = image.reshape((-1, 3))

    # Perform K-means clustering
    kmeans = KMeans(n_clusters=k, random_state=42)
    labels = kmeans.fit_predict(image_reshaped)

    # Reshape the labels back to the image shape
    segmented_image = labels.reshape(image.shape[0], image.shape[1])

    # Convert the segmented labels to a color image
    segmented_image_colored = np.zeros_like(image)
    for i in range(k):
        segmented_image_colored[segmented_image == i] = kmeans.cluster_centers_[i]

    return segmented_image_colored

# Parallel processing function to segment different regions (optional)
def parallel_kmeans_segmentation(image, k=3, n_jobs=-1):
```

```
    # Split the image into multiple parts to process in parallel
    def process_chunk(chunk):
        return kmeans_segmentation(chunk, k)

    height, width, _ = image.shape
    chunk_height = height // 4  # Divide the image into 4 chunks vertically

    chunks = [image[i:i+chunk_height, :] for i in range(0, height, chunk_height)]

    # Apply parallel processing to each chunk
    segmented_chunks = Parallel(n_jobs=n_jobs)(delayed(process_chunk)(chunk) for chunk in chunks)

    # Reassemble the image from the chunks
    segmented_image = np.vstack(segmented_chunks)

    return segmented_image

# Read the image
image_path = '/content/cat-8576777_640.webp'  # You can replace this with the path to your own image
image = cv2.imread(image_path)

# Check if the image is loaded properly
if image is None:
    print(f"Error: Image not found or unable to load the image at {image_path}")
else:
    # Apply k-means segmentation
    segmented_image = kmeans_segmentation(image, k=5)

    # Show the segmented image
    plt.imshow(cv2.cvtColor(segmented_image, cv2.COLOR_BGR2RGB))
    plt.show()
```