```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
```

```python
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 = 0.39113
Iteration 2/10: Best Fitness = 0.01925
Iteration 3/10: Best Fitness = 0.01925
Iteration 4/10: Best Fitness = 0.01925
Iteration 5/10: Best Fitness = 0.00125
Iteration 6/10: Best Fitness = 0.00125
Iteration 7/10: Best Fitness = 0.00125
Iteration 8/10: Best Fitness = 0.00125
Iteration 9/10: Best Fitness = 0.00125
Iteration 10/10: Best Fitness = 0.00125

Optimization Complete.
Best Solution: [-0.00959221  0.03401244]
Best Fitness: 0.00125
```