```python
import numpy as np
import random

# Objective function (Sphere function)
def objective_function(x):
    return np.sum(x ** 2)

# Initialize the grid (population)
def initialize_grid(grid_size, dim, bounds):
    return np.random.uniform(bounds[0], bounds[1], (grid_size, grid_size, dim))

# Evaluate fitness of the grid
def evaluate_grid(grid, objective_function):
    fitness = np.zeros((grid.shape[0], grid.shape[1]))
    for i in range(grid.shape[0]):
        for j in range(grid.shape[1]):
            fitness[i, j] = objective_function(grid[i, j])
    return fitness

# Selection using the best individual in the neighborhood
def select_best_neighbor(grid, fitness, x, y):
    neighbors = [
        ((x - 1) % grid.shape[0], y),   # Up
        ((x + 1) % grid.shape[0], y),   # Down
        (x, (y - 1) % grid.shape[1]),   # Left
        (x, (y + 1) % grid.shape[1]),   # Right
    ]
    best_pos = min(neighbors, key=lambda pos: fitness[pos[0], pos[1]])
    return grid[best_pos[0], best_pos[1]]

# Crossover operation
def crossover(parent1, parent2):
    alpha = np.random.rand()
    return alpha * parent1 + (1 - alpha) * parent2

# Mutation operation
def mutate(individual, bounds, mutation_rate=0.1):
    for i in range(len(individual)):
        if random.random() < mutation_rate:
            individual[i] += np.random.uniform(-1, 1)
            individual[i] = np.clip(individual[i], bounds[0], bounds[1])
    return individual

# Main Parallel Cellular Genetic Algorithm
def parallel_cellular_ga(objective_function, grid_size=5, dim=2, bounds=(-5, 5), max_iter=100, mutation_rate=0.1):
    # Initialize the grid and fitness
    grid = initialize_grid(grid_size, dim, bounds)
    fitness = evaluate_grid(grid, objective_function)

    for iteration in range(max_iter):
        new_grid = np.copy(grid)

        for i in range(grid_size):
            for j in range(grid_size):
                # Select parents from the neighborhood
                parent1 = grid[i, j]
                parent2 = select_best_neighbor(grid, fitness, i, j)

                # Apply crossover and mutation
                offspring = crossover(parent1, parent2)
                offspring = mutate(offspring, bounds, mutation_rate)

                # Replace if offspring is better
                offspring_fitness = objective_function(offspring)
                if offspring_fitness < fitness[i, j]:
                    new_grid[i, j] = offspring
                    fitness[i, j] = offspring_fitness

        grid = new_grid

        # Output the best solution in the grid
        best_position = np.unravel_index(np.argmin(fitness), fitness.shape)
        best_fitness = fitness[best_position]
        print(f"Iteration {iteration + 1}: Best Fitness = {best_fitness}")

    # Return the best solution
    best_position = np.unravel_index(np.argmin(fitness), fitness.shape)
    return grid[best_position[0], best_position[1]], fitness[best_position]

# Parameters
```

```
grid_size = 5         # Size of the grid
dim = 2               # Dimensionality of the problem
bounds = (-5, 5)      # Search space boundaries
max_iter = 50         # Number of iterations
mutation_rate = 0.1   # Mutation rate

# Run PCGA
best_solution, best_fitness = parallel_cellular_ga(objective_function, grid_size, dim, bounds, max_iter, mutation_rate)

# Output the best solution
print(f"\nBest solution: {best_solution}")
print(f"Best fitness: {best_fitness}")
```

```
Iteration 1: Best Fitness = 0.06571016935620341
Iteration 2: Best Fitness = 0.06571016935620341
Iteration 3: Best Fitness = 0.06571016935620341
Iteration 4: Best Fitness = 0.01975646174919851
Iteration 5: Best Fitness = 0.01975646174919851
Iteration 6: Best Fitness = 0.01975646174919851
Iteration 7: Best Fitness = 0.016219175585871014
Iteration 8: Best Fitness = 0.014781837298537073
Iteration 9: Best Fitness = 0.01354681521242492
Iteration 10: Best Fitness = 0.009014417900499692
Iteration 11: Best Fitness = 0.00366251453912872
Iteration 12: Best Fitness = 0.00366251453912872
Iteration 13: Best Fitness = 0.00366251453912872
Iteration 14: Best Fitness = 0.0034887883082367005
Iteration 15: Best Fitness = 0.0034887883082367005
Iteration 16: Best Fitness = 0.0034887883082367005
Iteration 17: Best Fitness = 0.003125259778309727
Iteration 18: Best Fitness = 0.0024241284090699574
Iteration 19: Best Fitness = 0.002010479631378075
Iteration 20: Best Fitness = 0.0015112033412028746
Iteration 21: Best Fitness = 0.0015112033412028746
Iteration 22: Best Fitness = 0.0014810883705566
Iteration 23: Best Fitness = 0.001465339275066465
Iteration 24: Best Fitness = 0.001293529849566275
Iteration 25: Best Fitness = 0.001247161323675447
Iteration 26: Best Fitness = 0.0012410534665905446
Iteration 27: Best Fitness = 0.0012410534665905446
Iteration 28: Best Fitness = 0.0012410534665905446
Iteration 29: Best Fitness = 0.00123313097275597
Iteration 30: Best Fitness = 0.0012306022902823684
Iteration 31: Best Fitness = 0.0012306022902823684
Iteration 32: Best Fitness = 0.00111090617256
Iteration 33: Best Fitness = 0.00111090617256
Iteration 34: Best Fitness = 0.00111090617256
Iteration 35: Best Fitness = 0.00111090617256
Iteration 36: Best Fitness = 0.00111090617256
Iteration 37: Best Fitness = 0.00111090617256
Iteration 38: Best Fitness = 0.00111090617256
Iteration 39: Best Fitness = 0.00111090617256
Iteration 40: Best Fitness = 0.00111090617256
Iteration 41: Best Fitness = 0.00111090617256
Iteration 42: Best Fitness = 0.00111090617256
Iteration 43: Best Fitness = 0.00111090617256
Iteration 44: Best Fitness = 0.00111090617256
Iteration 45: Best Fitness = 0.00111090617256
Iteration 46: Best Fitness = 0.00111090617256
Iteration 47: Best Fitness = 0.00111090617256
Iteration 48: Best Fitness = 0.00111090617256
Iteration 49: Best Fitness = 0.00111090617256
Iteration 50: Best Fitness = 0.00111090617256

Best solution: [0.03175863 0.01011413]
Best fitness: 0.00111090617256
```