```
import numpy as np

def objective_function(x):
    # Example: A simple quadratic function to minimize
    return sum([xi**2 for xi in x])

def initialize_population(grid_size, dimensions, bounds):
    return np.random.uniform(bounds[0], bounds[1], size=(grid_size, grid_size, 

def get_neighbors(grid, x, y):
    # Define the Moore neighborhood (8 neighbors)
    neighbors = []
    for dx in [-1, 0, 1]:
        for dy in [-1, 0, 1]:
            if dx == 0 and dy == 0:
                continue
            neighbors.append(grid[(x + dx) % grid.shape[0], (y + dy) % grid.sha
    return neighbors

def parallel_cellular_algorithm(
    objective_function,
    dimensions,
    bounds,
    grid_size=10,
    num_iterations=100
):
    # Initialize the cellular grid
    grid = initialize_population(grid_size, dimensions, bounds)
    fitness_grid = np.zeros((grid_size, grid_size))

    # Evaluate the initial fitness of each cell
    for i in range(grid_size):
        for j in range(grid_size):
            fitness_grid[i, j] = objective_function(grid[i, j])

    best_solution = None
    best_fitness = float("inf")

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

        for i in range(grid_size):
            for j in range(grid_size):
                # Get the neighbors of the current cell
                neighbors = get_neighbors(grid, i, j)

                # Find the best neighbor
```

```
                best_neighbor = min(neighbors, key=objective_function)

                # Update the cell's state (e.g., move towards the best neighbor
                new_grid[i, j] = (grid[i, j] + best_neighbor) / 2

                # Enforce bounds
                new_grid[i, j] = np.clip(new_grid[i, j], bounds[0], bounds[1])

        # Evaluate the fitness of the new grid
        for i in range(grid_size):
            for j in range(grid_size):
                fitness_grid[i, j] = objective_function(new_grid[i, j])

                # Track the best solution
                if fitness_grid[i, j] < best_fitness:
                    best_fitness = fitness_grid[i, j]
                    best_solution = new_grid[i, j]

        # Update the grid
        grid = new_grid

    return best_solution, best_fitness

if __name__ == "__main__":
    # Define problem parameters
    dimensions = 2  # Number of dimensions
    bounds = (-10, 10)  # Bounds for the search space

    # Run Parallel Cellular Algorithm
    best_solution, best_value = parallel_cellular_algorithm(objective_function,

    print("Best Solution:", best_solution)
    print("Best Value:", best_value)
```

```
Best Solution: [4.63795973e-13 1.02550128e-12]
Best Value: 1.2667595861122983e-24
```