Program 6: Parallel Cellular Algorithms and Programs

Code:

```
import numpy as np
import random

# Define any optimization function to minimize (can be changed as needed)
def custom_function(x):
    return x**2  # Example function: x^2 to minimize

# Initialize the population (grid of cells) with random values
def initialize_population(grid_size, dim, lower_bound, upper_bound):
    # Initialize a grid of cells with random positions
    population = np.random.uniform(lower_bound, upper_bound, (grid_size, grid_size, dim))
    return population

# Evaluate the fitness of each cell in the grid
def evaluate_fitness(population, fitness_function):
    # Calculate fitness of each cell based on the optimization function
    fitness = np.zeros(population.shape[:-1])  # Create an empty fitness matrix
    for i in range(population.shape[0]):
        for j in range(population.shape[1]):
            fitness[i, j] = fitness_function(population[i, j])  # Assign fitness value
    return fitness

# Update the state of each cell based on the best neighbor within a neighborhood
def update_states(population, fitness, neighborhood_radius, lower_bound, upper_bound):
    grid_size, _, dim = population.shape
    new_population = population.copy()  # Make a copy to store updated values

    # Iterate over each cell in the grid
    for i in range(grid_size):
        for j in range(grid_size):
            # Get the neighbors of the current cell
            neighbors = get_neighbors(i, j, grid_size, neighborhood_radius)
            best_neighbor = None
            best_fitness = float('inf')

            # Find the best neighbor based on fitness
```

```
            for ni, nj in neighbors:
                if fitness[ni, nj] < best_fitness:
                    best_fitness = fitness[ni, nj]
                    best_neighbor = (ni, nj)

            # Update the current cell towards the best neighbor
            if best_neighbor:
                ni, nj = best_neighbor
                # Add a small random perturbation to the updated cell value
                new_population[i, j] = population[ni, nj] + np.random.uniform(-0.5, 0.5, dim)
                # Ensure the new population is within bounds
                new_population[i, j] = np.clip(new_population[i, j], lower_bound, upper_bound)

    return new_population

# Get the neighbors of a cell within a given neighborhood radius
def get_neighbors(i, j, grid_size, radius):
    neighbors = []
    for di in range(-radius, radius + 1):
        for dj in range(-radius, radius + 1):
            ni, nj = (i + di) % grid_size, (j + dj) % grid_size
            if (di != 0 or dj != 0):  # Exclude the cell itself
                neighbors.append((ni, nj))
    return neighbors

# Main function to run the parallel cellular algorithm
def parallel_cellular_algorithm(grid_size, dim, lower_bound, upper_bound, max_iterations,
neighborhood_radius, fitness_function):
    # Initialize the population (grid of cells)
    population = initialize_population(grid_size, dim, lower_bound, upper_bound)

    # Initialize fitness of the population
    fitness = evaluate_fitness(population, fitness_function)

    best_solution = None
    best_fitness = float('inf')

    # Iterate to update states (based on number of iterations)
    for iteration in range(max_iterations):
        # Update states based on neighbor interactions
```

```
        population = update_states(population, fitness, neighborhood_radius, lower_bound,
upper_bound)

        # Re-evaluate fitness
        fitness = evaluate_fitness(population, fitness_function)

        # Track the best solution found so far
        min_fitness = fitness.min()
        if min_fitness < best_fitness:
            best_fitness = min_fitness
            best_solution = population[np.unravel_index(fitness.argmin(), fitness.shape)]

        # Print output every 10 iterations
        if (iteration + 1) % 10 == 0:
            print(f"Iteration {iteration + 1}/{max_iterations}, Best Fitness: {best_fitness}")

    return best_solution, best_fitness

# Parameters for the algorithm
grid_size = 10  # Number of cells per side (10x10 grid = 100 cells)
dim = 1  # One-dimensional solution space (this can be extended to higher dimensions if needed)
lower_bound = -5  # Lower bound for the solution space (can be adjusted for different problem
ranges)
upper_bound = 5  # Upper bound for the solution space
max_iterations = 100  # Number of iterations (how long to run the algorithm)
neighborhood_radius = 1  # Radius for neighborhood search (defines how far neighboring cells
are considered)

# Run the parallel cellular algorithm
best_solution, best_fitness = parallel_cellular_algorithm(grid_size, dim, lower_bound,
upper_bound, max_iterations, neighborhood_radius, custom_function)

# Output the best solution found
print("\nBest Solution Found:", best_solution)
print("Best Fitness Value:", best_fitness)
```

Output:

```
Iteration 10/100, Best Fitness: 6.151748023183363e-06
Iteration 20/100, Best Fitness: 1.675707538339441e-07
Iteration 30/100, Best Fitness: 5.9534590538844534e-08
Iteration 40/100, Best Fitness: 5.9534590538844534e-08
Iteration 50/100, Best Fitness: 5.9534590538844534e-08
Iteration 60/100, Best Fitness: 5.9534590538844534e-08
Iteration 70/100, Best Fitness: 5.9534590538844534e-08
Iteration 80/100, Best Fitness: 5.9534590538844534e-08
Iteration 90/100, Best Fitness: 5.9534590538844534e-08
Iteration 100/100, Best Fitness: 5.3029915870773e-10

Best Solution Found: [2.30282253e-05]
Best Fitness Value: 5.3029915870773e-10
```