Lab - 6

Parallel Cellular Algorithms and Programs

```
import numpy as np

# Define the mathematical function to optimize (e.g., Sphere 
function)
def objective_function(x, y):
    return x**2 + y**2  # Minimize this function

# Define the problem parameters
grid_size = 10  # Size of the 2D grid
iterations = 100  # Number of iterations
neighborhood_size = 1  # Neighborhood radius

# Initialize the population of cells (random positions in the 
solution space)
def initialize_population(grid_size, lower_bound, upper_bound):
    grid = np.random.uniform(lower_bound, upper_bound, (grid_size, 
grid_size, 2))  # (x, y) for each cell
    return grid

# Evaluate the fitness of each cell
def evaluate_fitness(grid):
    fitness = np.zeros((grid_size, grid_size))
    for i in range(grid_size):
        for j in range(grid_size):
            x, y = grid[i, j]
            fitness[i, j] = objective_function(x, y)
    return fitness

# Update the state of each cell based on its neighborhood
def update_states(grid, fitness):
    new_grid = grid.copy()
    for i in range(grid_size):
        for j in range(grid_size):
            # Get the neighborhood
            neighborhood = []
            for di in range(-neighborhood_size, neighborhood_size 
+ 1):
                for dj in range(-neighborhood_size, 
neighborhood_size + 1):
                    ni, nj = (i + di) % grid_size, (j + dj) % 
grid_size
                    neighborhood.append((fitness[ni, nj], grid[ni, 
nj]))

            # Find the best neighbor
            best_neighbor = min(neighborhood, key=lambda x: x[0])
```

```python
            new_grid[i, j] = best_neighbor[1]  # Update to the best neighbor's position

    return new_grid

# Main function to execute the Parallel Cellular Algorithm
def parallel_cellular_algorithm():
    # Initialize parameters
    lower_bound, upper_bound = -10, 10  # Solution space bounds
    grid = initialize_population(grid_size, lower_bound, upper_bound)
    best_solution = None
    best_fitness = float('inf')

    for _ in range(iterations):
        fitness = evaluate_fitness(grid)

        # Update best solution
        min_fitness = fitness.min()
        if min_fitness < best_fitness:
            best_fitness = min_fitness
            best_index = np.unravel_index(fitness.argmin(), fitness.shape)
            best_solution = grid[best_index]

        # Update the states of the grid
        grid = update_states(grid, fitness)

    return best_solution, best_fitness

# Run the algorithm
best_solution, best_fitness = parallel_cellular_algorithm()
print("Best Solution:", best_solution)
print("Best Fitness:", best_fitness)
```

Output:

```
Best Solution: [ 0.13267371 -0.92210353]
Best Fitness: 0.867877235058378
```