Lab 6

Parallel Cellular Algorithms and Programs

Code:-

```
import numpy as np
import random
import math
import concurrent.futures

# Define the Rastrigin function (objective function to optimize)
def rastrigin_function(x):
    A = 10
    return A * len(x) + sum([(xi**2 - A * math.cos(2 * math.pi * xi)) for xi in x])

# Initialize the population of cells (solutions) randomly
def initialize_population(num_cells, grid_size):
    population = []
    for _ in range(num_cells):
        cell = np.random.uniform(-5.12, 5.12, grid_size)  # Rastrigin function bounds [-
5.12, 5.12]
        population.append(cell)
    return population

# Evaluate the fitness of a cell (solution)
def evaluate_fitness(cell):
    return rastrigin_function(cell)

# Function to update the state of each cell based on its neighbors
def update_cell_state(cell, neighbors, grid_size):
    # For simplicity, we'll use an average of the neighbors' states (here it's just a
random update)
    best_neighbor = min(neighbors, key=lambda n: evaluate_fitness(n))
    # Move towards the best neighbor in the fitness landscape
```

```
    new_cell = cell + 0.1 * (best_neighbor - cell)
    return np.clip(new_cell, -5.12, 5.12)  # Clamping to the function's bounds

# Parallelized function to perform the main update step for each cell
def update_population(population, grid_size):
    updated_population = []
    # Parallelize the update step for all cells in the population
    with concurrent.futures.ThreadPoolExecutor() as executor:
        for i in range(len(population)):
            # Get the neighbors (simple example: using adjacent cells)
            neighbors = population[max(i - 1, 0):min(i + 2, len(population))]
            updated_cell = executor.submit(update_cell_state, population[i], neighbors,
grid_size)
            updated_population.append(updated_cell)
        updated_population = [cell.result() for cell in updated_population]
    return updated_population

# Main function to perform the optimization
def parallel_cellular_algorithm(num_cells, grid_size, num_iterations):
    # Step 1: Initialize Population
    population = initialize_population(num_cells, grid_size)

    # Step 2: Main Loop (Iterate for a fixed number of iterations)
    best_solution = None
    best_fitness = float('inf')

    for iteration in range(num_iterations):
        # Step 3: Evaluate fitness of all cells
        fitness_scores = [evaluate_fitness(cell) for cell in population]

        # Track the best solution
        min_fitness = min(fitness_scores)
        if min_fitness < best_fitness:
            best_fitness = min_fitness
```

```
            best_solution = population[fitness_scores.index(min_fitness)]

        print(f"Iteration {iteration+1}/{num_iterations}, Best Fitness: {best_fitness}")

        # Step 4: Update population based on neighbors
        population = update_population(population, grid_size)

    # Output the best solution found
    return best_solution, best_fitness

# Main function to handle user input and execution
if __name__ == "__main__":
    # Get user input for parameters
    num_cells = int(input("Enter the number of cells: "))
    grid_size = int(input("Enter the grid size (dimension of each solution): "))
    num_iterations = int(input("Enter the number of iterations: "))

    # Run the Parallel Cellular Algorithm
    best_solution, best_fitness = parallel_cellular_algorithm(num_cells, grid_size,
num_iterations)

    print("\nOptimization complete!")
    print("Best Solution Found:", best_solution)
    print("Best Fitness:", best_fitness)
```

output:-

```
Enter the number of cells: 10
Enter the grid size (dimension of each solution): 5
Enter the number of iterations: 10
Iteration 1/10, Best Fitness: 59.99174778100239
Iteration 2/10, Best Fitness: 59.99174778100239
Iteration 3/10, Best Fitness: 59.99174778100239
Iteration 4/10, Best Fitness: 59.99174778100239
Iteration 5/10, Best Fitness: 49.03282796977294
Iteration 6/10, Best Fitness: 49.03282796977294
Iteration 7/10, Best Fitness: 49.03282796977294
Iteration 8/10, Best Fitness: 49.03282796977294
Iteration 9/10, Best Fitness: 49.03282796977294
Iteration 10/10, Best Fitness: 49.03282796977294

Optimization complete!
Best Solution Found: [-0.06683258  2.82252253  2.30744881 -2.93031518 -1.88095198]
Best Fitness: 49.03282796977294
```