Program 6 - Parallel Cellular Algorithm for Function Optimization :

Design and implement a Parallel Cellular Algorithm (PCA) in Python to optimize a mathematical function. Each cell in a 1D cellular grid represents a potential solution, and cells interact with their neighbors to explore the search space. The objective is to find the maximum value of a mathematical function.

Algorithm:

Algorithm for Parallel Cellular Algorithm

1. **Define the Problem**:
   - Use f(x)=−x2+8x+20f(x) = -x^2 + 8x + 20f(x)=−x2+8x+20 as the objective function.
   - Define the search space as [0,10][0, 10][0,10].
2. **Initialize Parameters**:
   - Set the grid size NNN (number of cells).
   - Define the number of iterations III.
   - Define the neighborhood structure (e.g., each cell interacts with its immediate neighbors).
3. **Initialize Population**:
   - Generate random initial values for each cell within the search space.
4. **Evaluate Fitness**:
   - Calculate f(x)f(x)f(x) for each cell to determine its fitness.
5. **Update States**:
   - For each cell, calculate its new state by considering its own value and the values of its neighbors.
6. **Iterate**:
   - Repeat the evaluation and state updating process for III iterations.
7. **Output the Best Solution**:
   - Track and output the value and fitness of the best cell after all iterations.

code:

```
import numpy as np

# Define the fitness function
def fitness_function(x):
    return -x**2 + 8*x + 20

# Parallel Cellular Algorithm implementation
def parallel_cellular_algorithm():
    # Input parameters
    grid_size = int(input("Enter grid size (number of cells): "))
    num_iterations = int(input("Enter number of iterations: "))
    x_min = float(input("Enter minimum value of x: "))
    x_max = float(input("Enter maximum value of x: "))
```

```
    # Initialize the cellular grid
    cells = np.random.uniform(x_min, x_max, grid_size)
    fitness = np.array([fitness_function(x) for x in cells])

    # Optimization loop
    for _ in range(num_iterations):
        new_cells = np.copy(cells)
        for i in range(grid_size):
            # Get neighbors (handle edge cases)
            left_neighbor = cells[i - 1] if i > 0 else cells[-1]
            right_neighbor = cells[i + 1] if i < grid_size - 1 else cells[0]

            # Update state based on neighbors
            new_cells[i] = (cells[i] + left_neighbor + right_neighbor) / 3
            new_cells[i] = np.clip(new_cells[i], x_min, x_max)  # Ensure
within bounds

        # Update fitness values
        cells = new_cells
        fitness = np.array([fitness_function(x) for x in cells])

    # Find the best solution
    best_index = np.argmax(fitness)
    best_solution = cells[best_index]
    best_fitness = fitness[best_index]

    return best_solution, best_fitness

# Run the Parallel Cellular Algorithm
best_solution, best_fitness = parallel_cellular_algorithm()
print(f"Best Solution: {best_solution}, Fitness: {best_fitness}")
print("Name-pooja Gaikwad(1BM22CS194)")
```

Output:

```
Enter grid size (number of cells): 10
Enter number of iterations: 50
Enter minimum value of x: 0
Enter maximum value of x: 10
Best Solution: 6.856947134091286, Fitness: 27.837853073007587
Name-pooja Gaikwad(1BM22CS194)
```

Page No.
Date

(5) Parallel Cellular Algo

1. Start
2. Initialize grid (rows, col)
   initialize grid of x col. with random states.
   (0 for dead, 1 for alive)

3. Count neighbours (grid, x, y)
   - Count the no. of neighbours around the cell (x, y)

4. Apply rules (Grid, x, y)
   for each cell (x, y) is parallel:
   Count neighbours (x, y)

   Apply Rules:
   If cell $(x, y) = 1$ & neighbour (grid, x, y) ≥ 2:
   newGrid $[x][y] = 1$ (cell stays alive)

   If cell $(x, y) = 0$ & neighbour (grid, x, y) $= 3$
   newGrid $[x][y] = 1$ (cell becomes alive)

   ELSE:
   newGrid $[x][y] = 0$ (cell stays dead)

   Grid $(x, y) \leftarrow$ new Grid $[x][y]$

5. For iteration 1 to max.
   Repeat the above steps

Return the grid after max generation