USN: 1BM22CS235

LAB-6: Parallel Cellular Algorithms and Programs

CODE:

```
#pcap import numpy as np



# Define the problem: A simple optmizaton functon (e.g., Sphere Functon)
def optmizaton_functon(positon):
    """Example: Sphere Functon for minimizaton."""
    return sum(x**2 for x in positon)

# Initalize Parameters
GRID_SIZE = (10, 10) # Grid size (rows, columns)
NEIGHBORHOOD_RADIUS = 1 # Moore neighborhood radius
DIMENSIONS = 2 # Number of dimensions in the soluton space
ITERATIONS = 30 # Number of iteratons

# Initalize Populaton
def initalize_populaton(grid_size, dimensions):
    """Initalize a grid with random positons."""
    populaton = np.random.uniform(-10, 10, size=(grid_size[0], grid_size[1], dimensions))
    return populaton

# Evaluate Fitness
def evaluate_fitness(populaton):
    """Calculate the fitness of all cells."""
    fitness = np.zeros((populaton.shape[0], populaton.shape[1]))
    for i in range(populaton.shape[0]):
        for j in range(populaton.shape[1]):
            fitness[i, j] = optmizaton_functon(populaton[i, j])
```

```
    return fitness

# Get Neighborhood
def get_neighborhood(grid, x, y, radius):
    """Get the neighbors of a cell within the specified radius."""
    neighbors = []
    for i in range(-radius, radius + 1):
        for j in range(-radius, radius + 1):
            if i == 0 and j == 0:
                contnue # Skip the current cell
            ni, nj = x + i, y + j
            if 0 <= ni < grid.shape[0] and 0 <= nj < grid.shape[1]:
                neighbors.append((ni, nj))
    return neighbors

# Update States
def update_states(populaton, fitness):
    """Update the state of each cell based on its neighbors."""
    new_populaton = np.copy(populaton)
    for i in range(populaton.shape[0]):
        for j in range(populaton.shape[1]):
            neighbors = get_neighborhood(populaton, i, j, NEIGHBORHOOD_RADIUS)
            best_neighbor = populaton[i, j]
            best_fitness = fitness[i, j]

            # Find the best positon among neighbors
            for ni, nj in neighbors:
                if fitness[ni, nj] < best_fitness:
                    best_fitness = fitness[ni, nj]
                    best_neighbor = populaton[ni, nj]
```

```
            # Update the cell state (move towards the best neighbor) new_populaton[i, j] =
            (populaton[i, j] + best_neighbor) / 2 # Average positon
    return new_populaton

# Main Algorithm
def parallel_cellular_algorithm():
    """Implementaton of the Parallel Cellular Algorithm."""
    populaton = initalize_populaton(GRID_SIZE, DIMENSIONS)
    best_soluton = None
    best_fitness = float('inf')

    for iteraton in range(ITERATIONS):
        # Evaluate fitness
        fitness = evaluate_fitness(populaton)

        # Track the best soluton
        min_fitness = np.min(fitness)
        if min_fitness < best_fitness:
            best_fitness = min_fitness
            best_soluton = populaton[np.unravel_index(np.argmin(fitness), fitness.shape)]

        # Update states based on neighbors
        populaton = update_states(populaton, fitness)

        # Print progress
        print(f"Iteraton {iteraton + 1}: Best Fitness = {best_fitness}")

    print("\nBest Soluton Found:")
    print(f"Positon: {best_soluton}, Fitness: {best_fitness}")

# Run the algorithm
```

```
if __name__ == "__main__":
    parallel_cellular_algorithm()
```

OUTPUT:

```
Iteration 1: Best Fitness = 0.43918427791098213
Iteration 2: Best Fitness = 0.43918427791098213
Iteration 3: Best Fitness = 0.062221279350329436
Iteration 4: Best Fitness = 0.030149522005462108
Iteration 5: Best Fitness = 0.015791278460696168
Iteration 6: Best Fitness = 0.0025499667118763104
Iteration 7: Best Fitness = 0.0025499667118763104
Iteration 8: Best Fitness = 0.00019007166980743008
Iteration 9: Best Fitness = 0.00019007166980743008
Iteration 10: Best Fitness = 1.0432171933623911e-05
Iteration 11: Best Fitness = 8.406928148912647e-06
Iteration 12: Best Fitness = 5.511032710180021e-07
Iteration 13: Best Fitness = 4.3084388056725156e-07
Iteration 14: Best Fitness = 2.315054420755622e-07
Iteration 15: Best Fitness = 5.245753459404661e-08
Iteration 16: Best Fitness = 5.245753459404661e-08
Iteration 17: Best Fitness = 4.341357920017173e-08
Iteration 18: Best Fitness = 1.145644119860328e-08
Iteration 19: Best Fitness = 3.147791691706415e-09
Iteration 20: Best Fitness = 2.8192306881167533e-09
Iteration 21: Best Fitness = 9.788374665398935e-11
Iteration 22: Best Fitness = 9.788374665398935e-11
Iteration 23: Best Fitness = 9.788374665398935e-11
Iteration 24: Best Fitness = 9.788374665398935e-11
Iteration 25: Best Fitness = 7.537171686605552e-11
Iteration 26: Best Fitness = 7.234639306921671e-11
Iteration 27: Best Fitness = 7.028872029493468e-11
Iteration 28: Best Fitness = 3.340290444524624e-11
Iteration 29: Best Fitness = 1.4953679944431498e-11
Iteration 30: Best Fitness = 1.0817118995466254e-11

Best Solution Found:
Position: [-2.92599538e-06 -1.50188883e-06], Fitness: 1.0817118995466254e-11
```