Shamanth K Murthy 1BM22CS251

6) Parallel Cellular Algorithm:

```
import numpy as np

def objective_function(x):
    return np.sum(x**2)

def initialize_population(num_cells, dim, bounds):
    """
    Generate an initial population of cells with random positions.
    """
    return np.random.uniform(bounds[0], bounds[1], size=(num_cells,
dim))

def evaluate_fitness(population):
    """
    Evaluate the fitness of each cell in the population.
    """
    return np.array([objective_function(cell) for cell in population])

def get_neighbors(index, grid_size, neighborhood='moore'):
    """
    Get the indices of neighboring cells for a given index.
    Moore neighborhood includes all adjacent cells (diagonals
included).
    """
    x, y = divmod(index, grid_size)
    neighbors = []
    for dx in [-1, 0, 1]:
        for dy in [-1, 0, 1]:
            if dx == 0 and dy == 0:
                continue
            nx, ny = (x + dx) % grid_size, (y + dy) % grid_size
            neighbors.append(nx * grid_size + ny)
    return neighbors

def update_states(population, fitness, grid_size):
    """
    Update the state of each cell based on its neighbors.
    """
    new_population = np.copy(population)
    for i in range(len(population)):
        neighbors = get_neighbors(i, grid_size)
        best_neighbor = min(neighbors, key=lambda idx: fitness[idx])
        if fitness[best_neighbor] < fitness[i]:
            new_population[i] = population[best_neighbor]
    return new_population
```

```python
def parallel_cellular_algorithm(num_cells, grid_size, dim, bounds,
num_iterations):
    """
    Main implementation of the parallel cellular algorithm.
    """
    # Initialize population
    population = initialize_population(num_cells, dim, bounds)
    # Iterate
    best_solution = None
    best_fitness = float('inf')
    for iteration in range(num_iterations):
        # Evaluate fitness
        fitness = evaluate_fitness(population)
        # Track the best solution
        current_best_idx = np.argmin(fitness)
        if fitness[current_best_idx] < best_fitness:
            best_fitness = fitness[current_best_idx]
            best_solution = population[current_best_idx]
        # Update states
        population = update_states(population, fitness, grid_size)

    return best_solution, best_fitness

# Parameters
num_cells = 100  # Number of cells
grid_size = int(np.sqrt(num_cells))  # Assume a square grid
dim = 2  # Dimensionality of the solution space
bounds = [-10, 10]  # Bounds for the solution space
num_iterations = 50  # Number of iterations

# Run the algorithm
best_solution, best_fitness = parallel_cellular_algorithm(
    num_cells, grid_size, dim, bounds, num_iterations
)

print(f"Best Solution: {best_solution}")
print(f"Best Fitness: {best_fitness}")
```

Output

```
Best Solution: [-0.0998082  -0.32800116]
Best Fitness: 0.11754643693165355
```

Application: 2. Traffic flow optimization

```
import numpy as np
import multiprocessing as mp
import time

# Parameters
GRID_SIZE = (20, 20)
NUM_CELLS = GRID_SIZE[0] * GRID_SIZE[1]
NEIGHBORHOOD = [(0, 1), (1, 0), (0, -1), (-1, 0)]
ITERATIONS = 500
LANE_CAPACITY = 5


def optimization_function(cell, traffic_density):
    return -traffic_density[cell]  # Lower density is better


def initialize_population(grid_size):
    return np.random.randint(0, LANE_CAPACITY + 1, grid_size)


def evaluate_fitness(grid):
    fitness = np.zeros_like(grid, dtype=float)
    rows, cols = grid.shape

    for i in range(rows):
        for j in range(cols):
            congestion = grid[i, j]
            fitness[i, j] = optimization_function((i, j), grid)

    return fitness


def update_states(grid, fitness):
    new_grid = np.copy(grid)
    rows, cols = grid.shape

    for i in range(rows):
        for j in range(cols):
            neighborhood_states = []
            neighborhood_coords = []
            for dx, dy in NEIGHBORHOOD:
                ni, nj = i + dx, j + dy
                if 0 <= ni < rows and 0 <= nj < cols:
                    neighborhood_states.append(grid[ni, nj])
                    neighborhood_coords.append((ni, nj))
```

```
            # Rule: Route vehicles to less congested neighbors
            if neighborhood_states:
                avg_congestion = np.mean(neighborhood_states)
                if avg_congestion < grid[i, j]:
                    new_grid[i, j] = max(0, grid[i, j] - 1)
                    next_cell_idx = np.argmin(neighborhood_states)
                    ni, nj = neighborhood_coords[next_cell_idx]
                    new_grid[ni, nj] = min(LANE_CAPACITY, new_grid[ni,
nj] + 1)

    return new_grid

# Main simulation loop
def simulate_traffic(grid_size, iterations):
    # Step 1: Initialize population
    grid = initialize_population(grid_size)
    print(grid)
    best_solution = None
    best_fitness = float('-inf')

    for t in range(iterations):


        # Step 2: Evaluate fitness
        fitness = evaluate_fitness(grid)

        # Track best solution
        max_fitness = np.max(fitness)
        if max_fitness > best_fitness:
            best_fitness = max_fitness
            best_solution = np.unravel_index(np.argmax(fitness),
grid.shape)

        # Step 3: Update states
        grid = update_states(grid, fitness)

        # Visualize current grid
    print(grid)

    # Step 4: Output the best solution
    print(f"Best solution found at {best_solution} with fitness
{best_fitness}")

if __name__ == "__main__":
    simulate_traffic(GRID_SIZE, ITERATIONS)
```

Output:

```
[[1 1 5 0 0 2 2 3 1 1 0 3 5 2 1 0 0 3 1 5]
 [2 5 0 0 2 5 2 0 1 3 0 4 0 0 1 2 3 4 5 4]
 [1 0 4 3 4 2 0 3 1 3 4 4 5 1 0 5 5 4 1 4]
 [1 5 3 2 3 2 4 4 2 4 4 0 0 4 5 5 2 1 5 1]
 [3 0 1 5 4 1 4 4 0 5 1 5 2 0 3 0 2 5 1 2]
 [4 4 2 4 2 3 0 4 2 1 5 1 5 0 5 4 4 5 1 3]
 [4 3 1 4 0 3 2 3 2 0 5 3 2 4 0 5 0 4 1 3]
 [1 3 2 0 0 4 5 2 2 4 4 4 4 3 2 4 3 4 4 1]
 [0 2 4 1 4 2 3 2 0 3 4 4 0 5 1 4 0 5 4 5]
 [4 1 1 4 0 3 0 2 5 0 2 2 0 2 3 4 5 1 0 0]
 [4 1 4 2 1 1 0 5 3 2 5 4 5 2 0 4 2 0 3 2]
 [2 1 4 2 2 1 4 1 0 5 3 2 0 0 0 3 3 5 0 3]
 [5 0 0 0 5 1 4 4 0 2 4 5 1 1 4 0 4 0 5 2]
 [1 1 5 1 4 5 4 1 0 3 3 1 3 0 4 2 2 0 0 0]
 [0 1 2 3 1 0 4 3 3 4 4 2 1 4 4 2 1 5 3 5]
 [2 4 1 3 5 1 4 0 4 2 2 3 0 3 1 0 2 1 5 1]
 [4 4 5 2 4 5 2 2 0 3 3 4 3 3 3 0 0 5 2 3]
 [2 4 4 3 5 1 3 4 1 1 3 4 1 3 5 1 2 2 5 4]
 [2 2 1 3 0 3 0 1 4 0 4 0 2 4 1 3 4 0 2 0]
 [1 4 1 2 2 1 5 3 1 4 1 0 2 4 5 0 4 4 4 1]]
[[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 4]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 3 2 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 2 2 2 2]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 2 2 3 3 2]
 [1 1 1 1 1 1 1 1 1 2 1 1 2 2 2 2 3 2 4 3]
 [1 1 1 1 1 1 1 1 1 3 2 2 1 4 3 3 3 2 4 3]
 [1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 3 5 5 3]
 [1 1 1 1 1 1 1 2 3 1 2 2 3 3 4 2 3 3 3 4]
 [1 1 1 1 1 1 1 3 1 3 2 2 3 3 2 3 4 3 4 4]]
Best solution found at (0, 3) with fitness 0.0
```