BIS LAB EXAMINATION - 1BM22CS243

QUESTION : PARALLEL CELLULAR ALGORITHM

```
import numpy as np

def fitness_function(position):
    return np.sum(position**2)

# Initialize the population (grid of cells)
def initialize_population(grid_size, solution_dim, bounds):
    population = np.random.uniform(bounds[0], bounds[1], (grid_size[0],
grid_size[1], solution_dim))
    return population

# Evaluate the fitness of each cell
def evaluate_fitness(population):
    fitness = np.zeros((population.shape[0], population.shape[1]))
    for i in range(population.shape[0]):
        for j in range(population.shape[1]):
            fitness[i, j] = fitness_function(population[i, j])
    return fitness

# Update the state of a cell based on its neighbors
def update_cell_state(cell_position, neighbor_positions, bounds):
    best_neighbor = min(neighbor_positions, key=fitness_function)
    new_position = cell_position + np.random.uniform(-1, 1,
len(cell_position)) * (best_neighbor - cell_position)
    new_position = np.clip(new_position, bounds[0], bounds[1])  #
Ensure position stays within bounds
    return new_position

# Get neighbors of a cell
def get_neighbors(population, x, y):
    neighbors = []
    rows, cols = population.shape[:2]
    for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1), (-1, -1), (-1, 1),
(1, -1), (1, 1)]:
        nx, ny = x + dx, y + dy
        if 0 <= nx < rows and 0 <= ny < cols:
            neighbors.append(population[nx, ny])
    return neighbors

# Parallel Cellular Algorithm
def parallel_cellular_algorithm(grid_size, solution_dim, bounds,
num_iterations):
    # Step 1: Initialize population
    population = initialize_population(grid_size, solution_dim, bounds)
```

```python
    best_solution = None
    best_fitness = float('inf')

    # Step 2: Iterative optimization
    for _ in range(num_iterations):
        # Evaluate fitness
        fitness = evaluate_fitness(population)

        # Update each cell in parallel
        new_population = np.copy(population)
        for i in range(grid_size[0]):
            for j in range(grid_size[1]):
                neighbors = get_neighbors(population, i, j)
                new_population[i, j] = update_cell_state(population[i,
j], neighbors, bounds)

        # Update population
        population = new_population

        # Track the best solution
        for i in range(grid_size[0]):
            for j in range(grid_size[1]):
                if fitness[i, j] < best_fitness:
                    best_fitness = fitness[i, j]
                    best_solution = population[i, j]

    return best_solution, best_fitness

# Parameters
grid_size = (5, 5)  # 5x5 grid of cells
solution_dim = 2    # 2D solution space
bounds = (-10, 10)  # Search space bounds for each dimension
num_iterations = 100

best_solution, best_fitness = parallel_cellular_algorithm(grid_size,
solution_dim, bounds, num_iterations)
print("Best Solution:", best_solution)
print("Best Fitness:", best_fitness)
```

Ouput : `Best Solution: [-2.60896483e-04 -6.74215553e-05]`

`Best Fitness: 1.6573487963525238e-08`

Application :

```python
import numpy as np

# Define the fitness function
# This example assumes a simplified fitness function combining area,
power, and delay
# Lower fitness values represent better solutions
def fitness_function(placement):
    area = np.sum(placement ** 2)  # Example: area metric
    power = np.sum(placement)      # Example: power metric
    delay = np.sum(1 / (placement + 1e-5))  # Example: routing delay
(avoid division by zero)

    # Weighted sum of the three metrics
    return area + 0.5 * power + 0.3 * delay

# Update function for a cell, based on its neighbors
def update_cell(cell, neighbors):
    # Perform a local search by averaging the neighbors and adding
randomness
    new_value = np.mean(neighbors) + np.random.uniform(-0.1, 0.1)
    return max(0, new_value)  # Ensure non-negative values (physical
constraints)

# Parallel Cellular Algorithm for VLSI optimization
def parallel_cellular_algorithm(grid_size, num_iterations):
    # Initialize the grid with random placements
    grid = np.random.uniform(0, 1, (grid_size, grid_size))

    # Main optimization loop
    for iteration in range(num_iterations):
        # Create a copy of the grid to store updates
        new_grid = np.copy(grid)

        for i in range(grid_size):
            for j in range(grid_size):
                # Define the neighborhood (wrap-around for edge cells)
                neighbors = [
                    grid[(i - 1) % grid_size, j],  # Top
                    grid[(i + 1) % grid_size, j],  # Bottom
                    grid[i, (j - 1) % grid_size],  # Left
                    grid[i, (j + 1) % grid_size],  # Right
                ]

                # Update the cell based on its neighbors
                new_grid[i, j] = update_cell(grid[i, j], neighbors)

        # Evaluate fitness of the new placements
```

```
        fitness_values = np.array([fitness_function(cell) for cell in
new_grid.flatten()])

        # Print iteration progress
        print(f"Iteration {iteration + 1}: Best fitness =
{np.min(fitness_values)}")

        # Update the grid for the next iteration
        grid = new_grid

    # Return the best placement configuration
    best_placement = grid.flatten()[np.argmin(fitness_values)]
    return grid, best_placement

# Parameters
grid_size = 10  # Example: 10x10 grid for chip components
num_iterations = 50  # Number of iterations for optimization

# Run the Parallel Cellular Algorithm
optimized_grid, best_placement = parallel_cellular_algorithm(grid_size,
num_iterations)

# Output the results
print("\nOptimized Placement Grid:")
print(optimized_grid)
print(f"\nBest Placement Fitness Value:
{fitness_function(best_placement)}")
```

Ouput :

```
Iteration 1: Best fitness = 1.093759429071725
Iteration 2: Best fitness = 1.0937733052318777
Iteration 3: Best fitness = 1.0937597427641015
Iteration 4: Best fitness = 1.0937604252122122
Iteration 5: Best fitness = 1.093774042189116
Iteration 6: Best fitness = 1.093760238518852
Iteration 7: Best fitness = 1.0937945124687956
Iteration 8: Best fitness = 1.0937594571377907
Iteration 9: Best fitness = 1.0937701929483863
Iteration 10: Best fitness = 1.0937752313924058
Iteration 11: Best fitness = 1.0937716982776078
Iteration 12: Best fitness = 1.0937625010222347
Iteration 13: Best fitness = 1.0937611397141669
Iteration 14: Best fitness = 1.0937595076236364
Iteration 15: Best fitness = 1.093798257934321
Iteration 16: Best fitness = 1.0940324118030251
Iteration 17: Best fitness = 1.09375991595108
Iteration 18: Best fitness = 1.0937646768078553
Iteration 19: Best fitness = 1.0937601511672617
```

```
Iteration 20: Best fitness = 1.0937594148822356
Iteration 21: Best fitness = 1.09379449855488?
Iteration 22: Best fitness = 1.0937717412414463
Iteration 23: Best fitness = 1.0937796008790792
Iteration 24: Best fitness = 1.0938189105332903
Iteration 25: Best fitness = 1.0937860232352876
Iteration 26: Best fitness = 1.0937594152569206
Iteration 27: Best fitness = 1.0937605610137737
Iteration 28: Best fitness = 1.0937594989457997
Iteration 29: Best fitness = 1.0937660872926682
Iteration 30: Best fitness = 1.0937609074463568
Iteration 31: Best fitness = 1.0937979043461397
Iteration 32: Best fitness = 1.093760287367451
Iteration 33: Best fitness = 1.09380218806767
Iteration 34: Best fitness = 1.0937628779868813
Iteration 35: Best fitness = 1.0939643038357463
Iteration 36: Best fitness = 1.0937643533532837
Iteration 37: Best fitness = 1.093926657874358
Iteration 38: Best fitness = 1.09375952539712
Iteration 39: Best fitness = 1.0938373210667434
Iteration 40: Best fitness = 1.0937615760046835
Iteration 41: Best fitness = 1.0937622121142787
Iteration 42: Best fitness = 1.093774726757427
Iteration 43: Best fitness = 1.0937596048253844
Iteration 44: Best fitness = 1.093770535266736
Iteration 45: Best fitness = 1.0937673677272848
Iteration 46: Best fitness = 1.0937594298572078
Iteration 47: Best fitness = 1.093759478492679
Iteration 48: Best fitness = 1.0937974219890827
Iteration 49: Best fitness = 1.0937636711139107
Iteration 50: Best fitness = 1.093795615988007

Optimized Placement Grid:
[[0.5448364  0.55449296 0.48121833 0.63598355 0.55036385 0.63935853
  0.53840992 0.62039716 0.53057381 0.65624571]
 [0.54482809 0.5675153  0.48236952 0.60468648 0.45202945 0.66835638
  0.65079122 0.62928497 0.46618118 0.5135742 ]
 [0.589523   0.5132246  0.51771704 0.4415972  0.48260586 0.49429477
  0.54631839 0.67927139 0.43188207 0.5367487 ]
 [0.46701986 0.53495057 0.59478348 0.42011678 0.42498629 0.44158062
  0.51293566 0.32481127 0.44787185 0.49756932]
 [0.39522676 0.56852248 0.41666377 0.44939049 0.4915825  0.46817642
  0.52217218 0.57914246 0.40405991 0.41521746]
 [0.4629958  0.5027735  0.50474777 0.44581659 0.49738639 0.58070696
  0.57063019 0.38038914 0.50996599 0.42138778]
 [0.54073983 0.57922622 0.36826435 0.52566595 0.41547065 0.67072089
  0.41605567 0.62398027 0.49548048 0.40476056]
 [0.5233397  0.61811951 0.41241562 0.47377007 0.43775875 0.46431779
  0.59229993 0.48807834 0.45675068 0.47668198]
 [0.55269353 0.4239985  0.49080007 0.40462697 0.44082014 0.49033246
  0.56954711 0.53134339 0.45317833 0.49642594]
 [0.45468898 0.52821272 0.43224621 0.53449184 0.42367134 0.61643639
  0.53647134 0.45493013 0.58816288 0.50239614]]

Best Placement Fitness Value: 1.093795615988007
```