LAB - 06 **(16-12-24)**

Parallel Cellular Algorithms and Programs:

Parallel Cellular Algorithms are inspired by the functioning of biological cells that operate in a highly parallel and distributed manner. These algorithms leverage the principles of cellular automata and parallel computing to solve complex optimization problems efficiently. Each cell represents a potential solution and interacts with its neighbors to update its state based on predefined rules. This interaction models the diffusion of information across the cellular grid, enabling the algorithm to explore the search space effectively. Parallel Cellular Algorithms are particularly suitable for large-scale optimization problems and can be implemented on parallel computing architectures for enhanced performance.

Implementation Steps:

1.Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the number of cells, grid size, neighborhood structure, and number of iterations.
3. Initialize Population: Generate an initial population of cells with random positions in the solution space.
4. Evaluate Fitness: Evaluate the fitness of each cell based on the optimization function.
5.Update States: Update the state of each cell based on the states of its neighboring cells and predefined update rules.
6. Iterate: Repeat the evaluation and state updating process for a fixed number of iterations or until convergence criteria are met.
7.Output the Best Solution: Track and output the best solution found during the iterations

ALGORITHM/LOGIC-

PARALLEL CELLULAR ALGORITHM

PURPOSE:

The parallel cellular algorithm (PCA) is inspired by biological cell functioning and cellular automata. It is designed to solve complex large scale optimization problems efficiently.

APPLICATION:

1. Optimization Problems: solve ~~functionlog~~ functions for minimizing or maximizing specific values.
2. Image Processing: Used for edge detection and noise reduction
3. Routing & scheduling: optimize resource allocation and logistics efficiently

ALGORITHM:

1. Initialize the grid and random population of cells
2. Evaluate the fitness for all cells using fitness function.
3. For each iteration:
   - Update each cell based on average of neighbours
   - Recalculate fitness for all cells
   - Track the best solution
4. Stop when convergence or iteration limit is reached
5. Output the best solution.

Date 16/12/24
Page 24

IMPLEMENTATION:

1. Define the problem
2. Initialize parameters: grid size, Number of cells, iterations
3. Initialize Population
4. Evaluate fitness
5. Update states: synchronous, asynchronous, update rule example
6. Iterate: repeat until convergence or maximum iterations are reached
7. Output the best solution

OUTPUT:

Best solution: 2.0
Best fitness: 0.0

INPUT-

```
import numpy as np
def fitness_function(x):
    return x**2 - 4*x + 4

def initialize_grid(grid_size, value_range):
    return np.random.uniform(value_range[0], value_range[1], grid_size)

def evaluate_fitness(grid):
    return np.array([[fitness_function(x) for x in row] for row in grid])
```

```
def update_states(grid):
    updated_grid = np.copy(grid)
    rows, cols = grid.shape
    for i in range(rows):
        for j in range(cols):
            neighbors = [
                grid[x, y]
                for x in range(max(0, i - 1), min(rows, i + 2))
                for y in range(max(0, j - 1), min(cols, j + 2))
                if (x, y) != (i, j)
            ]
            updated_grid[i, j] = np.mean(neighbors)
    return updated_grid

# Parallel Cellular Algorithm
def parallel_cellular_algorithm(grid_size, value_range, iterations):
    grid = initialize_grid(grid_size, value_range)
    best_solution = None
    best_fitness = float("inf")

    for _ in range(iterations):
        fitness_grid = evaluate_fitness(grid)
        min_fitness = fitness_grid.min()
        if min_fitness < best_fitness:
            best_fitness = min_fitness
            best_solution = grid[np.unravel_index(fitness_grid.argmin(), fitness_grid.shape)]

        grid = update_states(grid)

    return best_solution, best_fitness

grid_size = (10, 10)
value_range = (-10, 10)
iterations = 100

best_solution, best_fitness = parallel_cellular_algorithm(grid_size, value_range, iterations)
print(f"Best Solution: {best_solution}")
print(f"Best Fitness: {best_fitness}")
```

OUTPUT-

```
Best Solution: 2.0008925527960155
Best Fitness: 7.966504940171149e-07
```