Lab 6: parallel cellular algorithms & programs

Algorithm

1. Initialization
i) Define the grid dimensions
ii) initialize each cell based on the problem
iii) Define the neighborhood for each cell
iv) Define transition rule

2. Set initial condition
. populate the grid with all initial states for all cells

3. parallel update
For each time step:
For each C(i,j,...) in grid (in parallel):
1. Retrieve the current state of C(i,j,...)
2. Retrieve the states of the neighboring cells
3. Apply the transition rule to compute the next state
if C(i,j,...)
update all cells simultaneously to their new states.

4. Check stopping conditions
Terminate if a predefined condition is met
· A fixed no. of iterations completed
· System reaches a steady state

```
import numpy as np

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

def initialize_grid(grid_size, dim):
    return np.random.uniform(-10, 10, (
    grid_size, (grid_size, dim))

def evaluate_fit(grid):
    return np.apply_along_axis(fitness_
    fn, 2, grid)

def update_cell(cell, neighbors, learning_
    rate=0.5):
        return cell + learning_rate * np.mean
        (neighbors - cell, axis=0)

def get_neighbors(grid, x, y)
    neighbors=[]
    for i in [-1, 0, 1]
        for j in [-1, 0, 1]
            if i==0 & j==0
                continue
            xi, yi = (x+i) % grid.shape[0],
            (y+j) % grid.shape[1]
            neighbors.append(grid[xi, yi])

    return np.array(neighbors)
```

```
def Algo(grid_size=10, dim=2, max_iteration=100):
    grid = initialize_grid(grid_size, dim)
    best_sol = None
    best_fit = float('inf')

    for _ in range(max_iteration):
        fitness_grid = evaluate_fitness(grid)

        for x in range(grid_size):
            for y in range(grid_size):
                neighbors = get_neighbors(grid, x, y)
                grid[x,y] = update_cell(grid[x,y],
                    neighbors)

        max_fit = np.max(fitness_grid)
        if(max_fit < best_fitness):
            best_fit = max_fit
            best_solution = grid[np.unravel_index
            (np.argmin(fit_grid), shape[:2])]

    return best_solution, best_fitness
best_solution, best_fit = parallel_cellular_Algo
print("Best solution:", best_solution)
print("Best fitness:", best_fitness)
```

Output:

Best solution = [-0.5505 -0.42444]

Best fitness = 0.2002454