Parallel Cellular Algorithm

```
import numpy as np

from copy import deepcopy

def get_neighbors(grid, i, j, neighborhood='moore'):
    """
    Get neighbor coordinates (with boundary checks).
    neighborhood: 'moore' (8 neighbors) or 'von_neumann' (4
neighbors)
    """
    rows, cols = grid.shape
    neighbors = []

    # Define relative positions
    if neighborhood == 'moore':
        directions = [(-1,-1), (-1,0), (-1,1),
                      (0,-1),          (0,1),
                      (1,-1),  (1,0),  (1,1)]
    elif neighborhood == 'von_neumann':
        directions = [(-1,0), (0,-1), (0,1), (1,0)]
    else:
        raise ValueError("Invalid neighborhood type")

    for dr, dc in directions:
        r, c = i + dr, j + dc
        if 0 <= r < rows and 0 <= c < cols:
            neighbors.append(grid[r, c])

    return neighbors


def parallel_cellular_algorithm(
        rows=10, cols=10,
        max_steps=20,
        neighborhood='moore',
        init_rule=None,
        update_rule=None,
        verbose=False):
    """
    Generic Parallel Cellular Algorithm Framework
    """

    # Initialize grid
    grid = np.zeros((rows, cols))
    if init_rule is not None:
        for i in range(rows):
            for j in range(cols):
```

```
                grid[i, j] = init_rule(i, j)
    else:
        # Default random initialization
        grid = np.random.randint(0, 2, (rows, cols))

    # Evolution loop
    for step in range(max_steps):
        new_grid = deepcopy(grid)

        # Parallel update (conceptually; real parallelism could use
multiprocessing)
        for i in range(rows):
            for j in range(cols):
                neighbors = get_neighbors(grid, i, j, neighborhood)
                if update_rule is not None:
                    new_grid[i, j] = update_rule(grid[i, j],
neighbors)

        grid = new_grid

        if verbose:
            print(f"\nStep {step+1}:")
            print(grid.astype(int))

    return grid
# --- Add this at the end of your file ---

def init_rule(i, j):
    # Randomly start with 0 (dead) or 1 (alive)
    return np.random.randint(0, 2)

def life_rule(cell, neighbors):
    alive_neighbors = sum(neighbors)
    if cell == 1 and alive_neighbors in [2, 3]:
        return 1
    elif cell == 0 and alive_neighbors == 3:
        return 1
    else:
        return 0

# Run the algorithm
final_grid = parallel_cellular_algorithm(
    rows=4,
    cols=4,
    max_steps=4,
    neighborhood='moore',
    init_rule=init_rule,
    update_rule=life_rule,
```

```
    verbose=True
)

print("\nFinal grid state:")
print(final_grid.astype(int))
```

Output:

```
Step 1:
[[0 1 1 0]
 [1 0 0 0]
 [1 1 1 0]
 [0 1 1 0]]

Step 2:
[[0 1 0 0]
 [1 0 0 0]
 [1 0 1 0]
 [1 0 1 0]]

Step 3:
[[0 0 0 0]
 [1 0 0 0]
 [1 0 0 0]
 [0 0 0 0]]

Step 4:
[[0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]]

Final grid state:
[[0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]]
```