Parallel Cellular Algorithms and Programs:

```
import numpy as np


def objective_function(x):
    """
    Define the mathematical function to optimize.
    Example: Sphere function f(x) = sum(x^2).
    """
    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

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: [-0.78384895 -0.94859388]
Best Fitness: 1.5142495245776773
```