Parallel Cellular Algorithms and Programs

```
import numpy as np

import random


# Step 1: Define the optimization function (Example: minimizing the function f(x) = x^2)

def objective_function(x):

    return x ** 2 - 4*x+4


# Step 2: Initialize parameters

num_cells = 100            # Number of cells (solutions)

grid_size = (10, 10)       # Grid size (10x10)

iterations = 1000          # Number of iterations

neighborhood_size = 3      # Neighborhood size (3x3)

convergence_threshold = 0.000001  # Convergence threshold


# Step 3: Initialize the population (randomly generate cell positions)

def initialize_population():

    # Create a grid with random positions for each cell in the search space [-10, 10]

    population = np.random.uniform(-10, 10, size=(grid_size[0], grid_size[1]))

    return population


# Step 4: Evaluate the fitness of each cell

def evaluate_fitness(population):

    # Apply the objective function to each cell

    fitness = np.vectorize(objective_function)(population)

    return fitness


# Step 5: Define a function to update the state of each cell based on neighboring cells

def update_cell_state(population, fitness, neighborhood_size):

    rows, cols = population.shape

    new_population = population.copy()
```

```
    # Define the neighborhood boundaries
    neighborhood_radius = neighborhood_size // 2

    for i in range(rows):
        for j in range(cols):
            # List of neighboring cell positions, including the current cell
            neighborhood = []
            for di in range(-neighborhood_radius, neighborhood_radius + 1):
                for dj in range(-neighborhood_radius, neighborhood_radius + 1):
                    ni, nj = i + di, j + dj
                    if 0 <= ni < rows and 0 <= nj < cols:  # Ensure indices are within bounds
                        neighborhood.append((population[ni, nj], fitness[ni, nj]))

            # Sort neighbors based on fitness value (ascending order: better solutions have lower fitness)
            neighborhood.sort(key=lambda x: x[1])
            best_neighbor = neighborhood[0]

            # Update the current cell based on the best neighbor (with some random fluctuation)
            new_population[i, j] = best_neighbor[0] + random.uniform(-0.1, 0.1)  # Slight random
movement
    return new_population

# Step 6: Iterate to update the states of the cells
def parallel_cellular_algorithm():
    population = initialize_population()
    fitness = evaluate_fitness(population)

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

```
    for iteration in range(iterations):
        print(f"Iteration {iteration + 1}/{iterations}")

        # Update cell states in parallel (Here we simulate parallel updates by using numpy)
        new_population = update_cell_state(population, fitness, neighborhood_size)

        # Evaluate the new population's fitness
        fitness = evaluate_fitness(new_population)

        # Track the best solution found so far
        min_fitness_index = np.argmin(fitness)
        min_fitness_value = fitness.flatten()[min_fitness_index]

        if min_fitness_value < best_fitness:
            best_fitness = min_fitness_value
            best_solution = new_population.flatten()[min_fitness_index]

        population = new_population  # Update population for next iteration

        # Check for convergence (early stop if we find a very small fitness value)
        if best_fitness < convergence_threshold:
            print("Convergence reached!")
            break

    return best_solution, best_fitness

# Step 7: Run the algorithm and output the best solution
best_solution, best_fitness = parallel_cellular_algorithm()

print(f"The best solution found is: {best_solution}")
print(f"The corresponding fitness (objective function value) is: {best_fitness}")
```

```
Iteration 1/1000
Iteration 2/1000
Iteration 3/1000
Convergence reached!
The best solution found is: 1.99953311880388894
The corresponding fitness (objective function value) is: 2.1797805116463564e-07
```