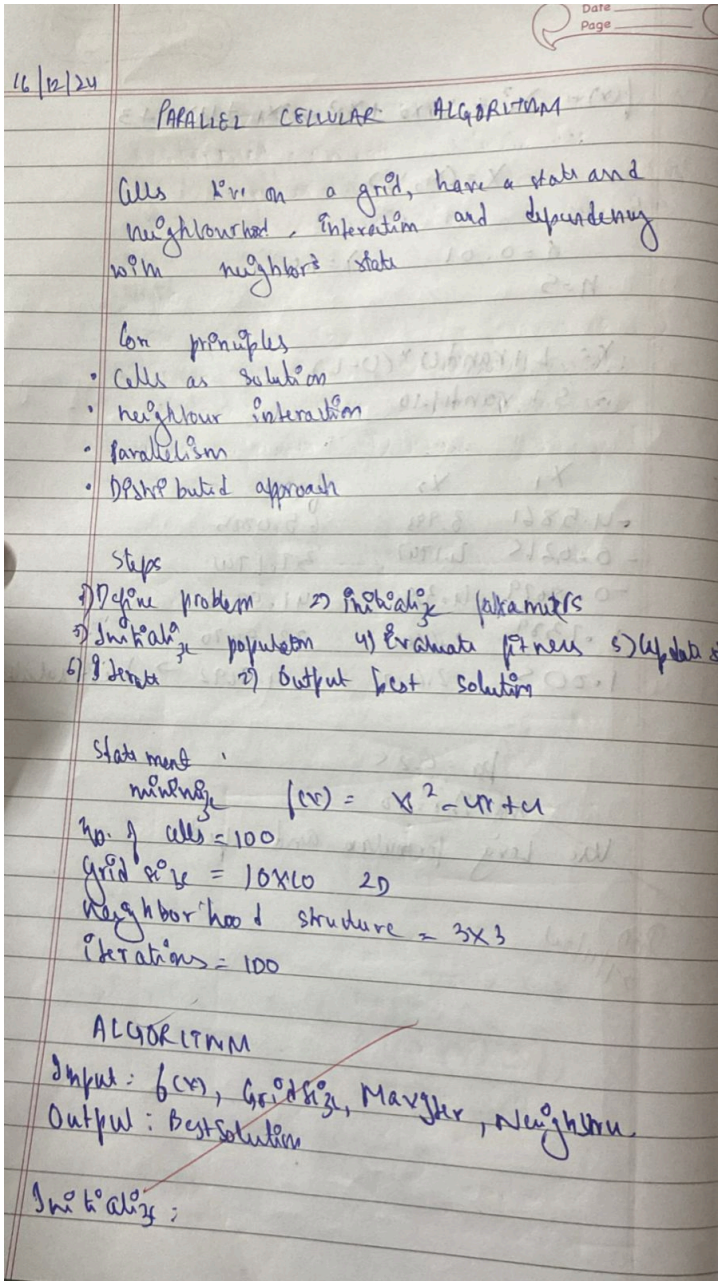
**Program 6**

Problem statement:Parallel Cellular Algorithms and Programs

Algorithm:



16/12/24

PARALLEL CELLULAR ALGORITHM

Cells live on a grid, have a state and neighbourhood. Interaction and dependency with neighbor's state

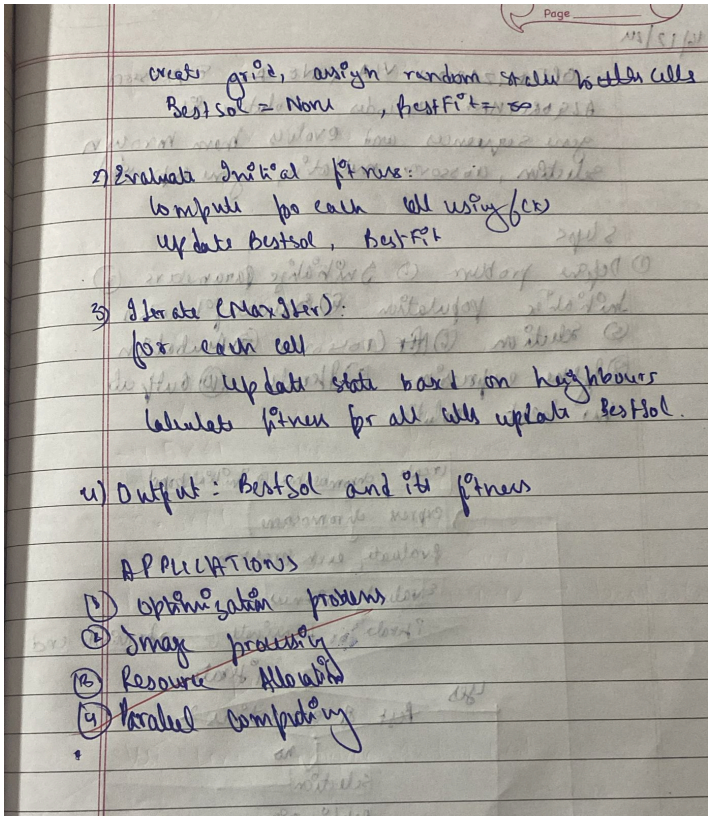
Core principles
- Cells as solution
- neighbour interaction
- parallelism
- Decentralized approach

Steps
1) Define problem 2) Initialize parameters
3) Initialize population 4) Evaluate fitness 5) Update
6) Iterate 7) Output best solution

Statement:
minimize $f(x) = x^2 - 4x + 4$
No. of cells = 100
Grid size = 10×10 2D
Neighborhood structure = 3×3
Iterations = 100

ALGORITHM
Input: f(x), Grid size, Mapper, Neighbour.
Output: Best solution

Initialize:




Code:

```
import numpy as np
def objective_function(x):
    return x**2 - 4*x + 4
class ParallelCellularAlgorithm:
    def __init__(self, objective_function, grid_size, max_iter, lb=-5, ub=5):
        self.objective_function = objective_function
        self.grid_size = grid_size
        self.max_iter = max_iter
        self.lb = lb
        self.ub = ub
        self.cells = np.random.uniform(self.lb, self.ub, (grid_size, grid_size))
        self.fitness = np.array([[self.objective_function(cell) for cell in row] for row in self.cells])
        self.best_cell = self.cells[np.unravel_index(np.argmin(self.fitness), self.fitness.shape)]
        self.best_fitness = np.min(self.fitness)
    def update_state(self, cell, neighbors):
        best_neighbor = min(neighbors, key=lambda x: self.objective_function(x))
        new_cell = best_neighbor + np.random.normal(0, 0.1)
        return np.clip(new_cell, self.lb, self.ub)
    def get_neighbors(self, row, col):
        neighbors = []
        for i in range(max(0, row-1), min(self.grid_size, row+2)):
            for j in range(max(0, col-1), min(self.grid_size, col+2)):
                if i != row or j != col:
```

```
                neighbors.append(self.cells[i, j])
        return neighbors
    def run(self):
        for t in range(self.max_iter):
            for i in range(self.grid_size):
                for j in range(self.grid_size):
                    neighbors = self.get_neighbors(i, j)
                    new_cell = self.update_state(self.cells[i, j], neighbors)
                    new_fitness = self.objective_function(new_cell)
                    if new_fitness < self.fitness[i, j]:
                        self.cells[i, j] = new_cell
                        self.fitness[i, j] = new_fitness
                    if new_fitness < self.best_fitness:
                        self.best_cell = new_cell
                        self.best_fitness = new_fitness
            print(f"Iteration {t+1}/{self.max_iter}: Best Fitness = {self.best_fitness}")
        return self.best_cell, self.best_fitness
grid_size = 5
max_iter = 10
lb = -5
ub = 5
pca = ParallelCellularAlgorithm(objective_function, grid_size, max_iter, lb, ub)
best_cell, best_fitness = pca.run()
print(f"\nBest Cell: {best_cell}")
print(f"Best Fitness: {best_fitness}")
```

```python
import numpy as np
def objective_function(x):
    return x**2 - 4*x + 4
class ParallelCellularAlgorithm:
    def __init__(self, objective_function, grid_size, max_iter, lb=-5, ub=5):
        self.objective_function = objective_function
        self.grid_size = grid_size
        self.max_iter = max_iter
        self.lb = lb
        self.ub = ub
        self.cells = np.random.uniform(self.lb, self.ub, (grid_size, grid_size))
        self.fitness = np.array([[self.objective_function(cell) for cell in row] for row in self.cells])
        self.best_cell = self.cells[np.unravel_index(np.argmin(self.fitness), self.fitness.shape)]
        self.best_fitness = np.min(self.fitness)
    def update_state(self, cell, neighbors):
        best_neighbor = min(neighbors, key=lambda x: self.objective_function(x))
        new_cell = best_neighbor + np.random.normal(0, 0.1)
        return np.clip(new_cell, self.lb, self.ub)
    def get_neighbors(self, row, col):
        neighbors = []
        for i in range(max(0, row-1), min(self.grid_size, row+2)):
            for j in range(max(0, col-1), min(self.grid_size, col+2)):
                if i != row or j != col:
                    neighbors.append(self.cells[i, j])
        return neighbors
    def run(self):
        for t in range(self.max_iter):
            for i in range(self.grid_size):
                for j in range(self.grid_size):
                    neighbors = self.get_neighbors(i, j)
                    new_cell = self.update_state(self.cells[i, j], neighbors)
                    new_fitness = self.objective_function(new_cell)
                    if new_fitness < self.fitness[i, j]:
                        self.cells[i, j] = new_cell
                        self.fitness[i, j] = new_fitness
                    if new_fitness < self.best_fitness:
                        self.best_cell = new_cell
                        self.best_fitness = new_fitness
            print(f"Iteration {t+1}/{self.max_iter}: Best Fitness = {self.best_fitness}")
        return self.best_cell, self.best_fitness
grid_size = 5
max_iter = 10
lb = -5
ub = 5
pca = ParallelCellularAlgorithm(objective_function, grid_size, max_iter, lb, ub)
best_cell, best_fitness = pca.run()
print(f"\nBest Cell: {best_cell}")
print(f"Best Fitness: {best_fitness}")
```

Output:

```
Iteration 1/10: Best Fitness = 4.4338109446151464e-05
Iteration 2/10: Best Fitness = 7.242042420863015e-06
Iteration 3/10: Best Fitness = 7.242042420863015e-06
Iteration 4/10: Best Fitness = 7.242042420863015e-06
Iteration 5/10: Best Fitness = 5.315313718057268e-07
Iteration 6/10: Best Fitness = 5.315313718057268e-07
Iteration 7/10: Best Fitness = 5.315313718057268e-07
Iteration 8/10: Best Fitness = 5.315313718057268e-07
Iteration 9/10: Best Fitness = 5.315313718057268e-07
Iteration 10/10: Best Fitness = 5.315313718057268e-07

Best Cell: 1.9992709380192222
Best Fitness: 5.315313718057268e-07
```