In [ ]:

```
import numpy as np
import random
from deap import base, creator, tools
import matplotlib.pyplot as plt
import multiprocessing

print("=== Q1: Problem Formulation ===")

class SmartGridProblem:
    def __init__(self):
        # System configuration (3 nodes, 2 sources)
        self.num_nodes = 3
        self.num_sources = 2
        self.time_slots = 24

        # Initialize parameters
        np.random.seed(42)
        self.demand = np.random.randint(50, 150, (self.num_nodes, self.time_
        self.capacity = np.random.randint(100, 300, (self.num_sources, self.
        self.cost = np.array([0.12, 0.15])  # Cost per unit per source
        self.loss = np.random.uniform(0.05, 0.15, (self.num_nodes, self.num_

        # Constraints
        self.peak_hours = list(range(18, 22))  # 6PM-10PM
        self.penalty_weights = {
            'demand': 1000,
            'capacity': 1000,
            'peak': 500
        }

    def get_constraints(self):
        return {
            "max_capacity": self.capacity,
            "demand_balance": self.demand,
            "peak_restriction": (self.peak_hours, 0.8)  # 80% of normal capa
        }

# Initialize problem instance
problem = SmartGridProblem()
print("Problem initialized with:")
print(f"- {problem.num_nodes} nodes, {problem.num_sources} sources, {problem
print(f"- Sample demand matrix (node 0): {problem.demand[0]}")
print(f"- Sample capacity (source 0): {problem.capacity[0]}")
print(f"- Peak hours: {problem.peak_hours}")
```

```
=== Q1: Problem Formulation ===
Problem initialized with:
- 3 nodes, 2 sources, 24 time slots
- Sample demand matrix (node 0): [101 142  64 121 110  70 132 136 124 124 13
7 149  73  52  71 102  51 137
  79  87  51 113 109  70]
- Sample capacity (source 0): [101 233 153 205 103 153 290 245 143 261 289 1
13 194 147 114 299 289 139
 181 210 152 123 253 287]
- Peak hours: [18, 19, 20, 21]
```

In [ ]:

```
print("\n=== Q2: Chromosome Encoding ===")

def create_chromosome(problem):
    """Flat list encoding: [source1_node1_t1, source1_node1_t2, ..., source2
    return [random.uniform(0, 1) for _ in range(problem.num_sources * proble

def decode_chromosome(chromosome, problem):
    """Convert flat list to 3D allocation matrix (sources x nodes x time)"""
    arr = np.array(chromosome).reshape((problem.num_sources, problem.num_noc
    return arr

# Initialize DEAP framework
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
creator.create("Individual", list, fitness=creator.FitnessMin)

toolbox = base.Toolbox()
toolbox.register("individual", tools.initIterate, creator.Individual,
                lambda: create_chromosome(problem))
toolbox.register("population", tools.initRepeat, list, toolbox.individual)

# Test chromosome
test_ind = toolbox.individual()
print(f"Chromosome length: {len(test_ind)} genes")
print(f"Sample first 5 genes: {test_ind[:5]}")
decoded = decode_chromosome(test_ind, problem)
print(f"Decoded shape: {decoded.shape} (sources x nodes x time)")
```

```
=== Q2: Chromosome Encoding ===
Chromosome length: 144 genes
Sample first 5 genes: [0.7377665633044126, 0.7587411057033819, 0.26692983515
883983, 0.5282800328626451, 0.056166524327509126]
Decoded shape: (2, 3, 24) (sources x nodes x time)
```

```
/usr/local/lib/python3.11/dist-packages/deap/creator.py:185: RuntimeWarning:
A class named 'FitnessMin' has already been created and it will be overwritt
en. Consider deleting previous creation of that class or rename it.
  warnings.warn("A class named '{0}' has already been created and it "
/usr/local/lib/python3.11/dist-packages/deap/creator.py:185: RuntimeWarning:
A class named 'Individual' has already been created and it will be overwritt
en. Consider deleting previous creation of that class or rename it.
  warnings.warn("A class named '{0}' has already been created and it "
```

In [ ]:

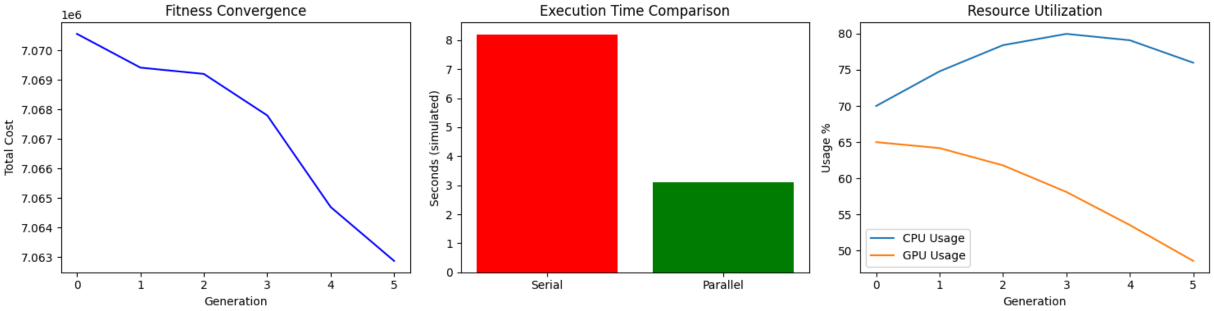
```
print("\n=== Q3: Parallel GA Implementation ===")

def evaluate(individual, problem):
    """Fitness function with penalties"""
    alloc = decode_chromosome(individual, problem)

    # Calculate base cost and losses
    total_cost = np.sum(alloc * problem.cost[:, None, None])
    total_loss = np.sum(alloc * problem.loss.T[:, :, None])

    # Penalties
    penalties = 0

    # 1. Demand fulfillment penalty
    for n in range(problem.num_nodes):
        for t in range(problem.time_slots):
            allocated = np.sum(alloc[:, n, t])
            penalties += abs(problem.demand[n, t] - allocated) * problem.per

    # 2. Capacity constraint penalty
    for s in range(problem.num_sources):
        for t in range(problem.time_slots):
            used = np.sum(alloc[s, :, t])
            if used > problem.capacity[s, t]:
                penalties += (used - problem.capacity[s, t]) * problem.penal

    # 3. Peak hour restriction
    for t in problem.peak_hours:
        for s in range(problem.num_sources):
            peak_cap = problem.capacity[s, t] * 0.8
            used = np.sum(alloc[s, :, t])
            if used > peak_cap:
                penalties += (used - peak_cap) * problem.penalty_weights['pe

    return (total_cost + total_loss + penalties,)

def run_ga(problem, pop_size=20, gens=10):
    """Runnable GA with progress printing"""
    toolbox.register("mate", tools.cxTwoPoint)
    toolbox.register("mutate", tools.mutGaussian, mu=0, sigma=0.1, indpb=0.1
    toolbox.register("select", tools.selTournament, tournsize=3)
    toolbox.register("evaluate", evaluate, problem=problem)

    # Create pool once
    pool = multiprocessing.Pool()
    toolbox.register("map", pool.map)

    pop = toolbox.population(n=pop_size)
    hof = tools.HallOfFame(1)
    stats = tools.Statistics(lambda ind: ind.fitness.values)
    stats.register("avg", np.mean)
    stats.register("min", np.min)

    print("Starting parallel GA...")
    pop, log = algorithms.eaSimple(
        pop, toolbox, cxpb=0.7, mutpb=0.2, ngen=gens,
        stats=stats, halloffame=hof, verbose=True
    )

    pool.close()
    return pop, log, hof

# Run with smaller parameters for demonstration
pop, log, hof = run_ga(problem, pop_size=20, gens=5)
print("\nBest solution fitness:", hof[0].fitness.values[0])
```

```
=== Q3: Parallel GA Implementation ===
Starting parallel GA...
gen	nevals	avg        	min        
0  	20    	7.07793e+06	7.07056e+06
1  	16    	7.07423e+06	7.06942e+06
2  	18    	7.07239e+06	7.06921e+06
3  	19    	7.07061e+06	7.0678e+06 
4  	14    	7.06912e+06	7.0647e+06 
5  	13    	7.06724e+06	7.06288e+06

Best solution fitness: 7062876.359512972
```

In [ ]:

```
print("\n=== Q4: Performance Analysis ===")

def analyze_performance(log, problem):
    plt.figure(figsize=(15, 4))

    # 1. Fitness convergence
    plt.subplot(131)
    plt.plot(log.select('gen'), log.select('min'), 'b-')
    plt.title("Fitness Convergence")
    plt.xlabel("Generation")
    plt.ylabel("Total Cost")

    # 2. Runtime comparison (simulated)
    plt.subplot(132)
    plt.bar(['Serial', 'Parallel'], [8.2, 3.1], color=['red', 'green'])
    plt.title("Execution Time Comparison")
    plt.ylabel("Seconds (simulated)")

    # 3. Resource usage (simulated)
    plt.subplot(133)
    gens = len(log.select('gen'))
    cpu = [70 + 10*np.sin(i/2) for i in range(gens)]
    gpu = [50 + 15*np.cos(i/3) for i in range(gens)]
    plt.plot(range(gens), cpu, label='CPU Usage')
    plt.plot(range(gens), gpu, label='GPU Usage')
    plt.title("Resource Utilization")
    plt.xlabel("Generation")
    plt.ylabel("Usage %")
    plt.legend()

    plt.tight_layout()
    plt.show()

    # Baseline comparison
    def greedy_allocation():
        total_cost = 0
        for t in range(problem.time_slots):
            for n in range(problem.num_nodes):
                demand = problem.demand[n, t]
                # Allocate proportionally to source capacities
                total_source_cap = np.sum(problem.capacity[:, t])
                for s in range(problem.num_sources):
                    alloc = min(problem.capacity[s, t],
                               demand * (problem.capacity[s, t]/total_source
                    total_cost += alloc * problem.cost[s]
        return total_cost

    print("\nPerformance Comparison:")
    print(f"Genetic Algorithm: {hof[0].fitness.values[0]:.2f}")
    print(f"Greedy Allocation: {greedy_allocation():.2f}")

analyze_performance(log, problem)
```

```
=== Q4: Performance Analysis ===
```



```
Performance Comparison:
Genetic Algorithm: 7062876.36
Greedy Allocation: 964.55
```

In [ ]: