Practical No: 08
Title: Implement DEAP (Distributed Evolutionary Algorithms) using Python
Problem Statement:. Develop a distributed evolutionary algorithm using DEAP (Distributed Evolutionary Algorithms in Python) to optimize a complex problem that requires intensive computational resources.
Prerequisite:
Basics of Python
Software Requirements: Jupyter
Hardware Requirements:
Core I3 Processor, 4GB RAM, 500 GB HDD
Learning Objectives:

1. Implement a scalable and efficient distributed evolutionary algorithm using DEAP to solve the optimization problem.

2. Improve the optimization process by leveraging parallel processing capabilities and distributing the workload across multiple nodes.

Outcomes:
After completion of this assignment students are able to understand how to developed distributed evolutionary algorithm will significantly reduce the optimization time and enable the solution of larger and more complex optimization problems.
Theory:
DEAP (Distributed Evolutionary Algorithms in Python) is a framework for building and analyzing distributed evolutionary algorithms. It provides tools for implementing genetic algorithms and other evolutionary computation techniques in a flexible and easy-to-use manner. DEAP allows users to parallelize their evolutionary algorithms across multiple processors or computers, making it suitable for tackling complex optimization problems that require significant computational resources.
DEAP provides a wide range of tools and functionalities to facilitate the implementation and experimentation of evolutionary algorithms. Some key features of DEAP include:
1. **Genetic Operators:** DEAP provides a set of standard genetic operators such as crossover, mutation, and selection, which can be easily customized and combined to suit the problem being solved.
2. **Fitness Evaluation:** DEAP allows users to define custom fitness functions to evaluate the quality of candidate solutions.
3. **Population Management:** DEAP provides tools for managing populations of candidate solutions, including initialization methods and selection strategies.
4. **Statistics and Logging:** DEAP includes utilities for tracking and logging the evolution of candidate solutions over time, including statistics such as average fitness and best fitness.
5. **Parallelization:** DEAP supports parallelization of evolutionary algorithms, allowing users to take advantage of multi-core processors and distributed computing environments to speed up optimization tasks.
6. **Integration:** DEAP can be easily integrated with other Python libraries and frameworks, making it a versatile tool for a wide range of optimization problems.
Overall, DEAP provides a convenient and efficient way to implement evolutionary algorithms in Python, making it a popular choice for researchers and practitioners working in the field of evolutionary computation.
Use and Applications:

- DEAP is used to solve optimization problems where traditional methods may be impractical or inefficient.
- It is commonly used in various fields such as engineering, biology, finance, and data science for optimization tasks.
- Applications include parameter optimization, feature selection, function optimization, and more.

Code:
Implement DEAP (Distributed Evolutionary Algorithms) using Python

pip install deap

```
Requirement already satisfied: deap in c:\users\cc\appdata\roaming\python\python39\site-packages (1.4.1)
Collecting numpy (from deap)
  Downloading numpy-1.26.4-cp39-cp39-win_amd64.whl.metadata (61 kB)
     ---------------------------------------- 0.0/61.0 kB ? eta -:--:--
     ------------- -------------------------- 20.5/61.0 kB ? eta -:--:--
     -------------------------- ------------ 41.0/61.0 kB 393.8 kB/s eta 0:00:01
     ------------------------------- ------ 51.2/61.0 kB 440.4 kB/s eta 0:00:01
     -------------------------------------- 61.0/61.0 kB 325.6 kB/s eta 0:00:00
Downloading numpy-1.26.4-cp39-cp39-win_amd64.whl (15.8 MB)
   ---------------------------------------- 0.0/15.8 MB ? eta -:--:--
   ---------------------------------------- 0.0/15.8 MB ? eta -:--:--
   ---------------------------------------- 0.1/15.8 MB 871.5 kB/s eta 0:00:19
   ---------------------------------------- 0.2/15.8 MB 1.4 MB/s eta 0:00:12
   ---------------------------------------- 0.2/15.8 MB 1.4 MB/s eta 0:00:12
    --------------------------------------- 0.3/15.8 MB 1.1 MB/s eta 0:00:14
    --------------------------------------- 0.4/15.8 MB 1.3 MB/s eta 0:00:13

   -------------------------------------- -- 14.8/15.8 MB 3.6 MB/s eta 0:00:01
   --------------------------------------- - 15.2/15.8 MB 3.4 MB/s eta 0:00:01
   ---------------------------------------- 15.8/15.8 MB 3.5 MB/s eta 0:00:00
Installing collected packages: numpy
Successfully installed numpy-1.26.4
Note: you may need to restart the kernel to use updated packages.
```

```python
import random
from deap import base, creator, tools, algorithms
# Define the evaluation function (minimize a simple mathematical function)
def eval_func(individual):
   # Example evaluation function (minimize a quadratic function)
   return sum(x ** 2 for x in individual),
# DEAP setup
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
creator.create("Individual", list, fitness=creator.FitnessMin)

toolbox = base.Toolbox()

# Define attributes and individuals
toolbox.register("attr_float", random.uniform, -5.0, 5.0)  # Example: Float values between -5 and 5
toolbox.register("individual", tools.initRepeat, creator.Individual, toolbox.attr_float, n=3)  # Example: 3-dimensional individual
toolbox.register("population", tools.initRepeat, list, toolbox.individual)

# Evaluation function and genetic operators
toolbox.register("evaluate", eval_func)
toolbox.register("mate", tools.cxBlend, alpha=0.5)
toolbox.register("mutate", tools.mutGaussian, mu=0, sigma=1, indpb=0.2)
toolbox.register("select", tools.selTournament, tournsize=3)

# Create population
population = toolbox.population(n=50)

# Genetic Algorithm parameters
generations = 20
```

```
# Run the algorithm
for gen in range(generations):
    offspring = algorithms.varAnd(population, toolbox, cxpb=0.5, mutpb=0.1)

    fits = toolbox.map(toolbox.evaluate, offspring)
    for fit, ind in zip(fits, offspring):
        ind.fitness.values = fit

    population = toolbox.select(offspring, k=len(population))

# Get the best individual after generations
best_ind = tools.selBest(population, k=1)[0]
best_fitness = best_ind.fitness.values[0]

print("Best individual:", best_ind)
print("Best fitness:", best_fitness)
```

// **output**

```
Best individual: [-0.012337462846870904, 0.0024526280250416702, 0.01195996830863327]
Best fitness: 0.00030126921567065186
```

Conclusion: This way DEAP (Distributed Evolutionary Algorithms) using Python is done.