LAB 4

AIM : To implement multithreading and multiprocessing in a distributed environment

Objectives:

- To understand the difference between multithreading and multiprocessing in parallel execution.
- To explore how concurrency and parallelism improve system performance.
- To implement multithreading techniques for efficient task sharing within a single process.
- To implement multiprocessing techniques for independent process execution on multiple cores.
- To learn how task synchronization and communication are handled across threads and processes.
- To evaluate the scalability and fault-tolerance of distributed systems using parallel execution.

Theory:

Multithreading:

Multithreading is a technique where multiple threads run inside the same process. A thread is the smallest unit of execution, and all threads share the same memory space of the program. This makes multithreading lightweight and efficient for tasks that involve waiting, such as file reading, web requests, or database queries. Since threads share memory, they can communicate easily, but this also creates problems like race conditions. In Python, multithreading is mainly useful for I/O-bound tasks, because the Global Interpreter Lock (GIL) prevents true parallel execution of threads for CPU-heavy work.

- A **thread** is the smallest unit of execution inside a program.
- **Multithreading** means running **multiple threads** within the **same process**.

How it works?

- All threads share the **same memory** of the program.
- Useful when tasks can run **concurrently** (e.g., downloading files, handling multiple users in a web server).

Multiprocessing:

Multiprocessing, on the other hand, uses multiple processes, each with its own memory space. Unlike threads, processes do not share memory and can run on separate CPU cores, which allows true parallelism. This makes multiprocessing very effective for CPU-bound tasks such as mathematical calculations, image processing, and matrix operations. However, processes are heavier than threads because they require more memory and have higher communication overhead.

- Uses **multiple processes**, each with its **own memory space**.
- Each process can run on a **different CPU core**, so tasks run **in parallel**.

How it works?

- The program is divided into **separate processes**.
- Each process is independent, so they don't block each other.

In simple terms, multithreading is best when a program needs to do many I/O operations at once (like a web server handling multiple users), while multiprocessing is best when the goal is to speed up heavy computations by using all available CPU cores. In this experiment, when we perform matrix addition using both approaches, multiprocessing gives better performance because it allows true parallel execution, while multithreading is limited by the GIL.

Gunicorn: Python WSGI HTTP Server

Gunicorn, short for **"Green Unicorn"**, is a lightweight and efficient **WSGI (Web Server Gateway Interface)** HTTP server designed for Python web applications. It is widely used for deploying Python web frameworks like Flask, Django, and FastAPI. Here's a quick overview:

Key Features

- **Pre-fork Worker Model**: Gunicorn spawns multiple worker processes to handle requests, ensuring better performance and reliability.
- **Framework Compatibility**: Works seamlessly with most Python web frameworks.
- **Ease of Use**: Simple to set up and configure.
- **Resource Efficiency**: Minimal resource usage while maintaining speed and scalability.

Procedure:

Step 1:

```
python3 --version
```

Step 2: Create a Project Folder

```
mkdir flask-gunicorn-demo

cd flask-gunicorn-demo
```

Step 3: Create a Virtual Environment

```
python3 -m venv .venv
```

Step 4: Activate the Virtual Environment

- **Linux/Mac**

```
source .venv/bin/activate
```

Step 5: Install Dependencies

```
pip install flask gunicorn numpy
```

Step 6: Create Flask Apps

A) Multithreading Code → `app_threading.py`

```
from flask import Flask, jsonify

import numpy as np

import threading, time


app = Flask(__name__)
```

```
def add_matrix_threads(A, B, num_threads=4):
    rows = A.shape[0]
    result = np.zeros_like(A)
    chunk = rows // num_threads
    threads = []

    def worker(start, end):
        result[start:end] = A[start:end] + B[start:end]

    for i in range(num_threads):
        start = i * chunk
        end = rows if i == num_threads - 1 else (i + 1) * chunk
        t = threading.Thread(target=worker, args=(start, end))
        threads.append(t)
        t.start()
    for t in threads:
        t.join()
    return result

@app.route("/threading")
def threading_add():
    N = 500
    A = np.random.randint(0, 100, (N, N))
```

```
    B = np.random.randint(0, 100, (N, N))
    t0 = time.time()
    result = add_matrix_threads(A, B, num_threads=4)
    t1 = time.time()
    return jsonify({
        "status": "ok",
        "method": "Multithreading",
        "execution_time_sec": round(t1 - t0, 4),
        "sample": result[0, :10].tolist()
    })

@app.route("/")
def index():
    return "Matrix Addition (Multithreading) API"
```

B) Multiprocessing Code → `app_multiprocessing.py`

```
from flask import Flask, jsonify
import numpy as np
import multiprocessing as mp
import time

app = Flask(__name__)
```

```
def worker_process(A, B, start, end, queue):
    part = A[start:end] + B[start:end]
    queue.put((start, part))


def add_matrix_processes(A, B, num_processes=4):
    rows = A.shape[0]
    result = np.zeros_like(A)
    queue = mp.Queue()
    procs = []
    chunk = rows // num_processes
    for i in range(num_processes):
        start = i * chunk
        end = rows if i == num_processes - 1 else (i + 1) * chunk
        p = mp.Process(target=worker_process, args=(A, B, start, end,
queue))
        procs.append(p); p.start()
    for p in procs:
        p.join()
    while not queue.empty():
        start, part = queue.get()
        result[start:start+part.shape[0]] = part
    return result
```

```
@app.route("/multiprocessing")
def processing_add():
    N = 500
    A = np.random.randint(0, 100, (N, N))
    B = np.random.randint(0, 100, (N, N))
    t0 = time.time()
    result = add_matrix_processes(A, B, num_processes=4)
    t1 = time.time()
    return jsonify({
        "status": "ok",
        "method": "Multiprocessing",
        "execution_time_sec": round(t1 - t0, 4),
        "sample": result[0, :10].tolist()
    })


@app.route("/")
def index():
    return "Matrix Addition (Multiprocessing) API"
```

Step 7: Run with Gunicorn

A) Run Multithreading app

```
gunicorn -w 1 --threads 4 -b 127.0.0.1:8001 app_threading:app
```

Open in browser: http://127.0.0.1:8001/threading

B) Run Multiprocessing app

```
gunicorn -w 1 -b 127.0.0.1:8002 app_multiprocessing:app
```

Open in browser: http://127.0.0.1:8002/multiprocessing

Step 8: Benchmark with ApacheBench (Optional)

Install ApacheBench:

```
sudo apt-get install apache2-utils   # Ubuntu/Debian
brew install httpd                   # macOS
```

Run tests:

1. **Test Multithreading API**

```
ab -n 100 -c 10 http://127.0.0.1:8000/threading
```

save results:

```
ab -n 200 -c 20 http://127.0.0.1:8001/threading > threading_ab.txt
```

2. **Test Multiprocessing API**

```
ab -n 100 -c 10 http://127.0.0.1:8000/multiprocessing
```

save results:

```
ab -n 200 -c 20 http://127.0.0.1:8002/multiprocessing > multiprocessing_ab.txt
```

Example Output (ApacheBench)

For multithreading run:

```
Concurrency Level:      10
Time taken for tests:   2.134 seconds
Complete requests:      100
```

```
Failed requests:        0
Requests per second:    46.85 [#/sec]
Time per request:       213.4 [ms]
```

For multiprocessing run:

```
Concurrency Level:      10
Time taken for tests:   1.052 seconds
Complete requests:      100
Failed requests:        0
Requests per second:    95.06 [#/sec]
Time per request:       105.2 [ms]
```

Metrics to Collect

Metric	How to Measure	Notes
Throughput (RPS)	From `ab` output	Higher is better
Latency	From `ab` output (`Time per request`)	Lower is better
CPU usage	`htop` or `top` during test	See how many cores are used
Memory usage	`htop` or `top` during test	Processes use more memory than threads

Conclusion:

multithreading and multiprocessing are useful techniques for achieving concurrency and parallelism, but their efficiency depends on the type of task. Multithreading is lightweight and suitable for I/O-bound tasks, where multiple operations like file access or network communication can run concurrently. However, due to Python's Global Interpreter Lock (GIL), it cannot achieve true parallelism in CPU-heavy operations. Multiprocessing, on the other hand, creates independent processes that run on multiple CPU cores, which enables true parallel execution. This makes it more effective for CPU-bound tasks such as matrix addition and other computations. In our experiment, matrix addition executed faster using multiprocessing compared to multithreading, proving that multiprocessing is the better choice for computationally intensive problems.