Of course!  
I'll give you **a complete and detailed explanation** in two parts:

**🧠 Code Explanation:**

**1. Importing Modules:**

import numpy as np

import time

from concurrent.futures import ThreadPoolExecutor

* **numpy:** To easily create and manage arrays.
* **time:** To measure execution time.
* **ThreadPoolExecutor:** To run parts of the sorting in parallel threads.

**2. Merging Two Sorted Parts:**

def merge(arr, low, mid, high):

...

* **left = arr[low:mid+1]** → Left half
* **right = arr[mid+1:high+1]** → Right half
* Then compares elements one by one and **merges** into a sorted order back into the original array.

**3. Sequential Merge Sort:**

def merge\_sort(arr, low, high):

if low < high:

mid = (low + high) // 2

merge\_sort(arr, low, mid)

merge\_sort(arr, mid + 1, high)

merge(arr, low, mid, high)

* **Divide and conquer:**
  + Recursively splits the array into two halves.
  + Sorts each half.
  + Merges the halves back into a sorted array.

**4. Parallel Merge Sort:**

def parallel\_merge\_sort(arr, low, high, depth=0):

...

* Works exactly like normal merge sort but uses **threads**.
* **ThreadPoolExecutor** divides the work among 2 threads.
* **depth parameter** controls **how deep** we go into parallelism (to avoid creating too many threads).

**5. Main Execution:**

n = 10

arr = np.arange(n, 0, -1)

* Creates an array [10, 9, 8, ..., 1].
* Then runs:
  + **Sequential Merge Sort**, prints sorted array and time.
  + **Parallel Merge Sort**, prints sorted array and time.

**📚 Theory Related to This Practical:**

**➡️ What is Merge Sort?**

* **Merge Sort** is a **Divide and Conquer** algorithm.
* It divides the array into two halves, sorts each half, and merges them into a sorted array.
* **Stable sorting algorithm.**

**➡️ Why Parallel Merge Sort?**

* In traditional Merge Sort, each recursive call happens **sequentially**.
* In **Parallel Merge Sort**, we solve left and right halves **simultaneously** using threads.
* Useful for **multi-core processors** where tasks can actually run at the same time!

**➡️ Time Complexity:**

| **Algorithm** | **Best** | **Average** | **Worst** | **Space Complexity** |
| --- | --- | --- | --- | --- |
| Merge Sort | O(n log n) | O(n log n) | O(n log n) | O(n) extra |

* **O(n log n)** → Because array splits log n times, and merging takes O(n) time each level.

**➡️ OpenMP in C++ vs Python Threads:**

* In C++, you used #pragma omp parallel sections for parallelism.
* In Python, true parallelism is tricky because of **GIL (Global Interpreter Lock)**.
* That's why **ThreadPoolExecutor** is used here, and real parallelism is better achieved with **multiprocessing** (for large tasks).

**➡️ Advantages of Parallel Merge Sort:**

* Faster sorting for **very large arrays**.
* Efficient CPU usage on **multi-core systems**.
* Reduces total execution time significantly if the dataset is large enough.

**✨ In Short:**

| **Sequential Merge Sort** | **Parallel Merge Sort** |
| --- | --- |
| Single-threaded | Multi-threaded |
| Good for small datasets | Better for large datasets |
| Easy to implement | Slightly complex |
| Slower on multi-core CPUs | Faster on multi-core CPUs |

**✅ Summary:**

In this practical,  
you **implemented**, **measured**, and **compared** the performance of  
**Sequential Merge Sort** vs **Parallel Merge Sort**  
in **Python** using **ThreadPoolExecutor**.

Would you also like me to give you a **ready-made practical write-up** (like aim, code, output, result, conclusion)  
✅ that you can directly submit in your journal or college report? 📓✨  
(Just say yes!)