!apt-get update

!apt-get install -y mpich

!pip install mpi4py

import numpy as np

import time

from mpi4py import MPI

import matplotlib.pyplot as plt

# Sequential QuickSort

def quicksort(arr, start, end):

    if start < end:

        pivot = partition(arr, start, end)

        quicksort(arr, start, pivot - 1)

        quicksort(arr, pivot + 1, end)

def partition(arr, start, end):

    pivot = arr[end]

    i = start - 1

    for j in range(start, end):

        if arr[j] <= pivot:

            i += 1

            arr[i], arr[j] = arr[j], arr[i]

    arr[i + 1], arr[end] = arr[end], arr[i + 1]

    return i + 1

# Merge function for combining sorted subarrays

def merge\_sorted\_arrays(arrays):

    result = []

    pointers = [0] \* len(arrays)

    while any(p < len(arr) for p, arr in zip(pointers, arrays)):

        min\_val = float('inf')

        min\_idx = -1

        for i, arr in enumerate(arrays):

            if pointers[i] < len(arr) and arr[pointers[i]] < min\_val:

                min\_val = arr[pointers[i]]

                min\_idx = i

        result.append(min\_val)

        pointers[min\_idx] += 1

    return result

# Parallel QuickSort using MPI

def parallel\_quicksort(data, comm, rank, size):

    if size == 1:

        quicksort(data, 0, len(data) - 1)

        return data

    # Scatter data to all processes

    local\_size = len(data) // size

    local\_data = np.zeros(local\_size, dtype=int)

    comm.Scatter(data, local\_data, root=0)

    # Sort local data

    quicksort(local\_data, 0, len(local\_data) - 1)

    # Gather sorted data to root

    gathered\_data = [np.zeros(local\_size, dtype=int) for \_ in range(size)]

    comm.Gather(local\_data, gathered\_data, root=0)

    # Merge sorted chunks at root

    if rank == 0:

        return merge\_sorted\_arrays(gathered\_data)

    return None

# Main function to evaluate performance

def main():

    comm = MPI.COMM\_WORLD

    rank = comm.Get\_rank()

    size = comm.Get\_size()

    if rank == 0:

        input\_sizes = [10000, 100000, 1000000]

        seq\_times = []

        par\_times = []

        for n in input\_sizes:

            # Generate random data

            data = np.random.randint(0, 1000000, n)

            # Sequential QuickSort

            data\_seq = data.copy()

            start\_time = time.time()

            quicksort(data\_seq, 0, n - 1)

            seq\_time = time.time() - start\_time

            seq\_times.append(seq\_time)

            # Parallel QuickSort

            start\_time = time.time()

            sorted\_data = parallel\_quicksort(data, comm, rank, size)

            par\_time = time.time() - start\_time

            par\_times.append(par\_time)

            # Verify correctness

            if sorted\_data is not None:

                assert np.array\_equal(sorted\_data, np.sort(data)), f"Sorting failed for n={n}"

        # Print results

        if rank == 0:

            print(f"{'Input Size':<15} {'Sequential (s)':<15} {'Parallel (s)':<15} {'Speedup':<10}")

            for n, st, pt in zip(input\_sizes, seq\_times, par\_times):

                speedup = st / pt if pt > 0 else float('inf')

                print(f"{n:<15} {st:<15.6f} {pt:<15.6f} {speedup:<10.2f}")

            # Plot results

            plt.figure(figsize=(10, 6))

            plt.plot(input\_sizes, seq\_times, label='Sequential', marker='o')

            plt.plot(input\_sizes, par\_times, label='Parallel', marker='s')

            plt.xscale('log')

            plt.yscale('log')

            plt.xlabel('Input Size')

            plt.ylabel('Execution Time (s)')

            plt.title(f'Performance: Sequential vs Parallel QuickSort (MPI, {size} processes)')

            plt.legend()

            plt.grid(True)

            plt.savefig('performance\_plot.png')

if \_\_name\_\_ == '\_\_main\_\_':

    main()

Output:

Input Size Sequential (s) Parallel (s) Speedup

10000 0.213045 0.153259 1.39

100000 1.906256 0.917122 2.08

1000000 12.527014 12.452296 1.01

