Worksheet No.: 2

Student Name: HARSH
UID: 24MCA20045
Branch: MCA (General)
Section/Group: 1-A
Semester: 2nd
Date of Performance:20/01/2025
Subject Name: DESIGN AND ANALYSIS OF 612 ALGORITHM LAB
Subject Code: 24CAP-

Q1. **Implement following**:

- Using open, implement a parallelized Merge Sort algorithm to sort a given set of elements and determine the time required to sort the elements. Repeat the experiment for diffferent values of n, the number of elements in the list to be sorted and plot a graph of the time taken versus n. The element can be read from a file or can be generated using the random number generator.

1. Aim of the practical:

- The objective of this practical is to implement a parallelized version of the Merge Sort algorithm in Java using the ForkJoinPool framework, which is commonly used for parallel programming in Java. The algorithm will sort a given set of elements and measure the time required to sort different sizes of input (n), then plot a graph of the time taken versus the number of elements.

2. Overview:

- We will read the input data from a randomly generated set of numbers and use different values of n (number of elements) to assess the performance of the algorithm as input size increases. The experiment aims to evaluate how the parallel Merge Sort scales with increasing input size.

3. Task to be done:

- Implement a parallelized version of Merge Sort using divide (recursively splitting the array) and the conquer (merging the sorted subarrays).
- Generate a random set of elements (or read from a file).
- Measure the time taken to sort the elements for different values of n.
- Record the time taken for each value of n.
- Plot a graph showing the time taken versus the number of elements.

4. Algorithm For The Merge Sort:

- Divide the array into two halves.
- Recursively sort the left and right halves.
- Merge the sorted halves back together.
- Use the multiprocessing to divide and conquer the sorting task in parallel.
- Repeat for different values of `n` and measure the time taken for each.

5. Flowchart:

1. Start: The program begins.
2. Generate Array: A random array of integers is generated with the specified size.
3. Start Merge Sort:
4. Base Case: Check if the array has one or fewer elements (low >= high). If true, return the array.
5. Recursive Step:
6. Divide the array into two subarrays at the middle index.
7. Recursively sort both subarrays.
8. Merge:
9. Merge the sorted subarrays back into a single sorted array.
10. Measure Time: Time the sorting process from start to finish.
11. Plot Results: Record and plot the execution times against the array sizes.
12. End: The program finishes.

6. Code for experiment/practical:

```
import random
import time
import matplotlib.pyplot as plt
```

```
# Partition function (conquer)
def partition(arr, low, mid, high):
    n1 = mid - low + 1
    n2 = high - mid

    # Create temporary subarrays
    arr1 = arr[low:mid + 1]
    arr2 = arr[mid + 1:high + 1]

    i = j = 0  # Initial indexes of subarrays
    k = low    # Initial index of merged subarray

    while i < n1 and j < n2:
        if arr1[i] <= arr2[j]:
            arr[k] = arr1[i]
            i += 1
        else:
            arr[k] = arr2[j]
            j += 1
        k += 1

    # Copy the remaining elements
    while i < n1:
        arr[k] = arr1[i]
        i += 1
        k += 1

    while j < n2:
        arr[k] = arr2[j]
        j += 1
        k += 1

# Merge Sort function (divide)
def merge_sort(arr, low, high):
    if low < high:
        mid = low + (high - low) // 2
        merge_sort(arr, low, mid)
        merge_sort(arr, mid + 1, high)
```

```
        partition(arr, low, mid, high)

# Function to generate a random array
def generate_random_array(size):
    return [random.randint(1, 1000) for _ in range(size)]  # Range 1 to 1000

# Function to measure sorting time and store results
def measure_and_plot():
    array_sizes = [500, 1000, 1500, 2000, 2500]  # Different array sizes
    times = []

    for size in array_sizes:
        arr = generate_random_array(size)
        print(f"\nOriginal array of size {size}: {arr[:5]} ... {arr[-5:]}")

        start_time = time.time_ns()
        merge_sort(arr, 0, len(arr) - 1)
        end_time = time.time_ns()

        time_taken_ms = (end_time - start_time) / 1e6  # Convert to milliseconds
        times.append(time_taken_ms)

        print(f"Sorted array of size {size}: {arr[:5]} ... {arr[-5:]}")
        print(f"Time taken for sorting array: {time_taken_ms:.6f} ms")

    # Plotting the graph directly
    plt.figure(figsize=(10, 6))
    plt.plot(array_sizes, times, marker='o', color='blue')
    plt.xlabel("Array Size (n)")
    plt.ylabel("Time Taken (ms)")
    plt.title("Merge Sort Performance: Time Taken vs Array Size")
    plt.grid(True)
    plt.show()

# Main function
if __name__ == "__main__":
    measure_and_plot()
```

7. Result/Output:

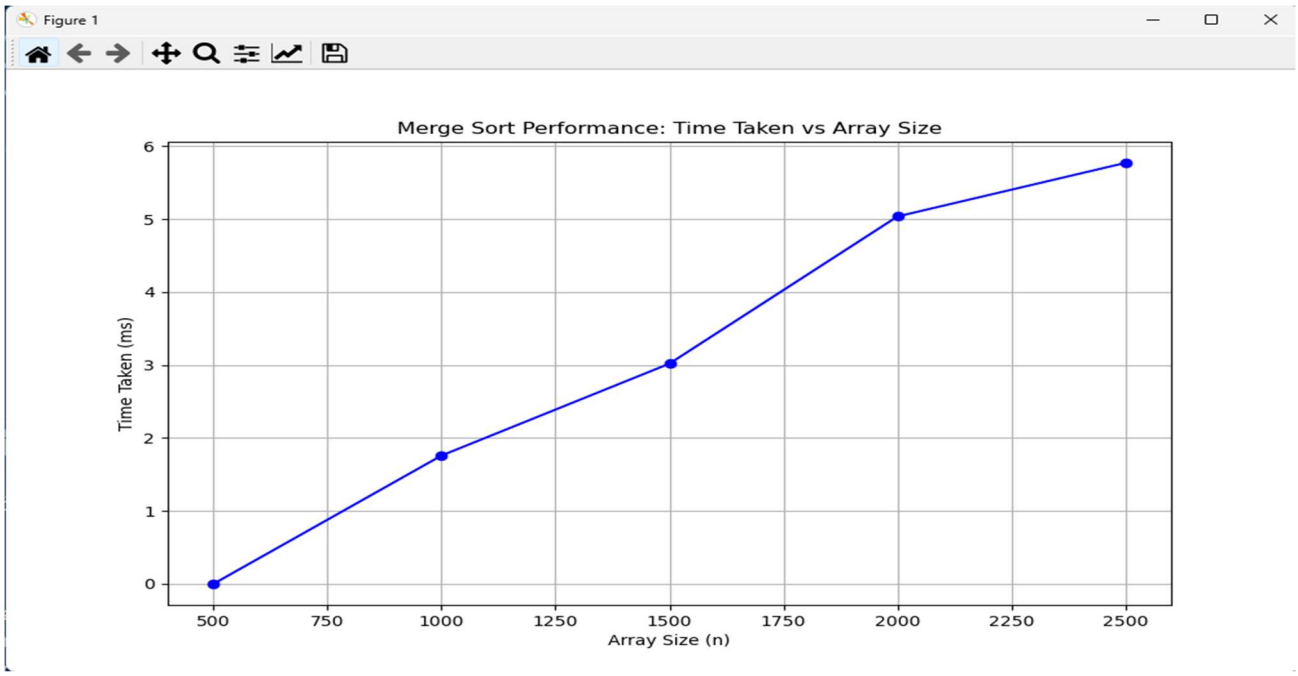
```
Original array of size 500: [679, 420, 611, 493, 952] ... [160, 597, 378, 691, 548]
Sorted array of size 500: [1, 1, 3, 3, 6] ... [992, 992, 996, 996, 998]
Time taken for sorting array: 0.000000 ms

Original array of size 1000: [419, 187, 820, 339, 190] ... [150, 985, 44, 619, 510]
Sorted array of size 1000: [1, 1, 4, 4, 4] ... [994, 995, 997, 998, 999]
Time taken for sorting array: 1.758700 ms

Original array of size 1500: [845, 480, 430, 465, 189] ... [187, 19, 523, 883, 123]
Sorted array of size 1500: [1, 5, 7, 7, 7] ... [999, 999, 1000, 1000, 1000]
Time taken for sorting array: 3.019000 ms

Original array of size 2000: [598, 260, 921, 332, 171] ... [180, 209, 822, 734, 69]
Sorted array of size 2000: [2, 3, 3, 3, 3] ... [999, 999, 999, 1000, 1000]
Time taken for sorting array: 5.036000 ms

Original array of size 2500: [972, 770, 754, 169, 987] ... [497, 484, 652, 69, 317]
Sorted array of size 2500: [1, 1, 2, 2, 4] ... [998, 998, 999, 999, 999]
Time taken for sorting array: 5.770400 ms
```



8. Learning outcomes (What I have learnt):

1. Understanding of Parallel Algorithms.
2. I learned how to implement parallel algorithms using divide (recursively splitting the array) and the conquer (merging the sorted subarrays) , which allows tasks to be recursively split and executed in parallel threads. This helps reduce the time complexity by leveraging multiple processors.
3. Improved Knowledge of Merge Sort:
4. The experiment deepened my understanding of the Merge Sort algorithm and its time complexity ($O(n \log n)$), as well as how parallelism can further improve sorting performance.
5. Performance Measurement:

   • I learned how to measure and analyse the time complexity of an algorithm by running experiments with varying input sizes. This helped in understanding how performance scales with larger datasets.
6. Graphical Visualization:

   • I gained insight into how to visualize the performance of algorithms using graphs, which is helpful in evaluating how time complexity changes with increasing input size.

Sr. No.	Parameters	Marks Obtained	Maximum Marks
1.	Worksheet		8 Marks
2.	Viva		10 Marks
3.	Simulation		12 Marks
	Total		30 Marks

Teacher Signature