pg1:Write a Open MP program to sort an array on n elements using both sequential and parallel merge sort (using Section). Record the difference in execution time.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

#define SIZE 100000

// Merge function merges two sorted subarrays arr[left..mid] and arr[mid+1..right]
void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int *L = malloc(n1 * sizeof(int));
    int *R = malloc(n2 * sizeof(int));

    for (int i = 0; i < n1; i++) L[i] = arr[left + i];
    for (int j = 0; j < n2; j++) R[j] = arr[mid + 1 + j];

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) arr[k++] = L[i++];
        else arr[k++] = R[j++];
    }
    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];

    free(L);
    free(R);
}

// Sequential merge sort
void sequentialMergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        sequentialMergeSort(arr, left, mid);
        sequentialMergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}

// Parallel merge sort using OpenMP sections
void parallelMergeSort(int arr[], int left, int right, int depth) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        if (depth <= 4) {  // limit parallel recursion depth
            #pragma omp parallel sections
            {
                #pragma omp section
                parallelMergeSort(arr, left, mid, depth + 1);

                #pragma omp section
                parallelMergeSort(arr, mid + 1, right, depth + 1);
            }
        } else {
            // Fall back to sequential to avoid too many threads
            sequentialMergeSort(arr, left, mid);
            sequentialMergeSort(arr, mid + 1, right);
        }
        merge(arr, left, mid, right);
    }
}

int main() {
    int *arr_seq = malloc(SIZE * sizeof(int));
    int *arr_par = malloc(SIZE * sizeof(int));

    // Initialize arrays with same random values
    for (int i = 0; i < SIZE; i++) {
        int val = rand() % 100000;
        arr_seq[i] = val;
        arr_par[i] = val;
    }

    // Sequential timing
    double start = omp_get_wtime();
    sequentialMergeSort(arr_seq, 0, SIZE - 1);
    double seq_time = omp_get_wtime() - start;

    // Parallel timing
    start = omp_get_wtime();
    parallelMergeSort(arr_par, 0, SIZE - 1, 0);
    double par_time = omp_get_wtime() - start;

    printf("Sequential Merge Sort Time: %.6f seconds\n", seq_time);
    printf("Parallel Merge Sort Time  : %.6f seconds\n", par_time);
    printf("Speedup                   : %.2fx\n", seq_time / par_time);

    // Optional correctness check (uncomment to use)
    /*
    for (int i = 0; i < SIZE; i++) {
        if (arr_seq[i] != arr_par[i]) {
            printf("Mismatch at index %d\n", i);
            break;
        }
    }
    */
    free(arr_seq);
    free(arr_par);

    return 0;
}
```