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;
```

```
}
```

output:

```
Sequential Merge Sort Time: 0.024535 seconds
Parallel Merge Sort Time  : 0.014064 seconds
Speedup                   : 1.74x
```

with explainaion:

```
#include <stdio.h>   // For printf
#include <stdlib.h>  // For malloc, rand, free
#include <omp.h>     // For OpenMP timing and parallelism

#define SIZE 100000  // Size of array to sort

// Merge function to merge 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;  // Length of left subarray
    int n2 = right - mid;     // Length of right subarray

    // Allocate temporary arrays to hold subarrays
    int *L = malloc(n1 * sizeof(int));
    int *R = malloc(n2 * sizeof(int));

    // Copy data to temp arrays L and R
    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;  // Initial indexes for temp arrays
    int k = left;      // Initial index for merged array

    // Merge back into arr[] while both arrays have elements
    while (i < n1 && j < n2) {
        if (L[i] <= R[j])
            arr[k++] = L[i++];  // Take element from L if smaller or equal
        else
            arr[k++] = R[j++];  // Otherwise take from R
    }

    // Copy any remaining elements of L[], if any
    while (i < n1) arr[k++] = L[i++];

    // Copy any remaining elements of R[], if any
    while (j < n2) arr[k++] = R[j++];
    // Free temporary arrays
    free(L);
    free(R);
}

// Sequential merge sort implementation
void sequentialMergeSort(int arr[], int left, int right) {
    if (left < right) { // Base case: if the subarray has more than 1 element
        int mid = left + (right - left) / 2;  // Find middle index

        // Recursively sort left half
        sequentialMergeSort(arr, left, mid);

        // Recursively sort right half
        sequentialMergeSort(arr, mid + 1, right);

        // Merge the two sorted halves
        merge(arr, left, mid, right);
    }
}

// Parallel merge sort using OpenMP sections and recursion depth control
void parallelMergeSort(int arr[], int left, int right, int depth) {
    if (left < right) { // Continue splitting if more than one element
        int mid = left + (right - left) / 2;

        if (depth <= 4) { // Limit parallelism depth to avoid too many threads
            // Create parallel sections to sort left and right halves concurrently
            #pragma omp parallel sections
            {
                #pragma omp section
                parallelMergeSort(arr, left, mid, depth + 1);     // Sort left half in one thread

                #pragma omp section
                parallelMergeSort(arr, mid + 1, right, depth + 1); // Sort right half in another thread
            }
        } else {
            // Beyond depth limit, fall back to sequential sorting to reduce overhead
            sequentialMergeSort(arr, left, mid);
            sequentialMergeSort(arr, mid + 1, right);
        }

        // Merge the sorted halves
        merge(arr, left, mid, right);
    }
}

int main() {
    // Allocate two arrays for sequential and parallel sorts
    int *arr_seq = malloc(SIZE * sizeof(int));
    int *arr_par = malloc(SIZE * sizeof(int));
    // Initialize both arrays with the same random values
    for (int i = 0; i < SIZE; i++) {
        int val = rand() % 100000; // Random number between 0 and 99999
        arr_seq[i] = val;          // Fill sequential array
        arr_par[i] = val;          // Fill parallel array
    }

    // Record start time for sequential merge sort
    double start = omp_get_wtime();
    sequentialMergeSort(arr_seq, 0, SIZE - 1);  // Run sequential merge sort
    double seq_time = omp_get_wtime() - start; // Calculate elapsed time

    // Record start time for parallel merge sort
    start = omp_get_wtime();
    parallelMergeSort(arr_par, 0, SIZE - 1, 0); // Run parallel merge sort starting at depth 0
    double par_time = omp_get_wtime() - start; // Calculate elapsed time

    // Print out timing results
    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 verify that sorting results match)
    /*
    for (int i = 0; i < SIZE; i++) {
        if (arr_seq[i] != arr_par[i]) {
            printf("Mismatch at index %d\n", i);
            break;
        }
    }
    */

    // Free allocated memory
    free(arr_seq);
    free(arr_par);

    return 0;
}
```