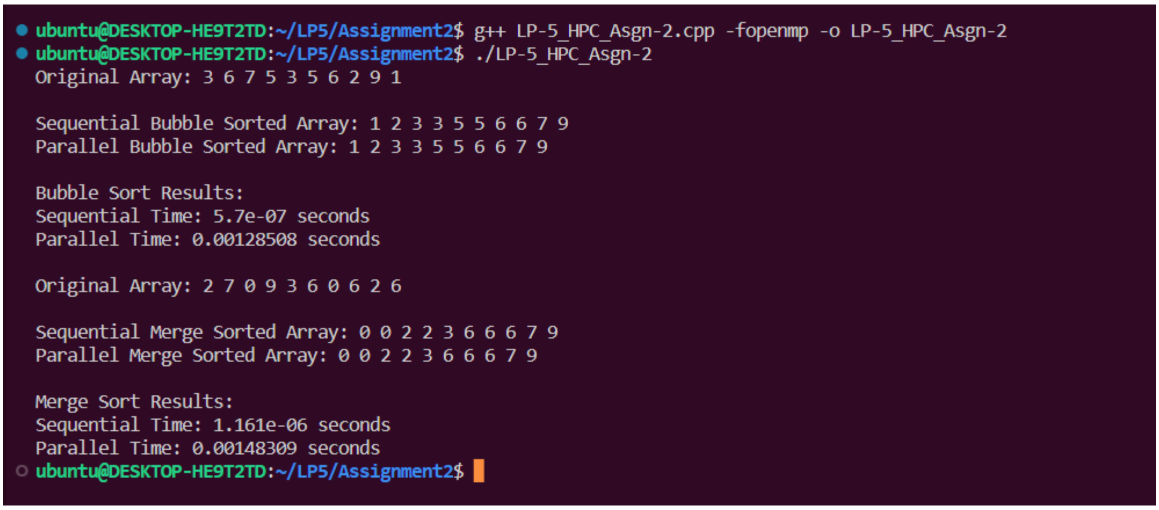
Code

```
#include <iostream>
#include <omp.h>

using namespace std;
int n = 10;

// function to perform sequential bubble sort
void bubbleSort(int arr[], int n) {
   int i, j, temp;
   for (i = 0; i < n-1; i++) {
      for (j = 0; j < n-i-1; j++) {
         if (arr[j] > arr[j+1]) {
            temp = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = temp;
         }
      }
   }
}

// function to perform parallel bubble sort using OpenMP
void parallelBubbleSort(int arr[], int n) {
   int i, j, temp;
   #pragma omp parallel for private(i, j, temp) num_threads(16)
   for (i = 0; i < n-1; i++) {
      for (j = 0; j < n-i-1; j++) {
         if (arr[j] > arr[j+1]) {
            temp = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = temp;
         }
      }
   }
}

// function to merge two subarrays in ascending order
void merge(int arr[], int left, int mid, int right) {
   int i, j, k;
   int n1 = mid - left + 1;
   int n2 = right - mid;
   int L[n1], R[n2];
   for (i = 0; i < n1; i++) {
      L[i] = arr[left + i];
   }
   for (j = 0; j < n2; j++) {
      R[j] = arr[mid + 1 + j];
   }
   i = 0;
```

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

// function to perform sequential merge sort
void mergeSort(int arr[], int left, int right, int n, bool isLastCall) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid, n, false);
        mergeSort(arr, mid+1, right, n, false);
        merge(arr, left, mid, right);
    }

}

// function to perform parallel merge sort using OpenMP
void parallelMergeSort(int arr[], int left, int right, int num_threads, int n) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        #pragma omp parallel sections num_threads(2)
        {
            #pragma omp section
            {
                parallelMergeSort(arr, left, mid, num_threads/2, n);
            }
            #pragma omp section
            {
                parallelMergeSort(arr, mid+1, right, num_threads/2, n);
            }
        }
        merge(arr, left, mid, right);
    }

}


int main() {

   int arr[n];

    cout << "Original Array: ";
    // initialize array with random values
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % n;
        cout << arr[i] <<" ";
    }

    // copy array for parallel sorting
    int arr_copy[n];
    for (int i = 0; i < n; i++) {
        arr_copy[i] = arr[i];
    }

    // measure time for sequential bubble sort
    double start_time = omp_get_wtime();
    bubbleSort(arr, n);
    double end_time = omp_get_wtime();
    double sequential_bubble_time = end_time - start_time;
    cout << "\n\nSequential Bubble Sorted Array: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] <<" ";
    }

    // measure time for parallel bubble sort
    start_time = omp_get_wtime();
    parallelBubbleSort(arr_copy, n);
    end_time = omp_get_wtime();
    double parallel_bubble_time = end_time - start_time;
    cout << "\nParallel Bubble Sorted Array: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] <<" ";
    }
    // output results for bubble sort
    cout << "\n\nBubble Sort Results:" << endl;
    cout << "Sequential Time: " << sequential_bubble_time << " seconds" << endl;
    cout << "Parallel Time: " << parallel_bubble_time << " seconds" << endl;

    cout << "\nOriginal Array: ";
    // reset array for merge sort
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % n;
        cout << arr[i] <<" ";
    }

    // copy array for parallel sorting
    for (int i = 0; i < n; i++) {
        arr_copy[i] = arr[i];
    }

    // measure time for sequential merge sort
    start_time = omp_get_wtime();
    mergeSort(arr, 0, n-1, n, true);
    end_time = omp_get_wtime();
    double sequential_merge_time = end_time - start_time;
    cout << "\n\nSequential Merge Sorted Array: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] <<" ";
    }
    // measure time for parallel merge sort
    start_time = omp_get_wtime();
    #pragma omp parallel num_threads(4)
    {
        #pragma omp single
        {
            parallelMergeSort(arr_copy, 0, n-1, omp_get_num_threads(), n);
        }
    }
    end_time = omp_get_wtime();
    double parallel_merge_time = end_time - start_time;
    cout << "\nParallel Merge Sorted Array: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] <<" ";
    }
    // output results for merge sort
    cout <<endl << "\nMerge Sort Results:" << endl;
    cout << "Sequential Time: " << sequential_merge_time << " seconds" << endl;
    cout << "Parallel Time: " << parallel_merge_time << " seconds" << endl;

    return 0;
}
```

Output



```
● ubuntu@DESKTOP-HE9T2TD:~/LP5/Assignment2$ g++ LP-5_HPC_Asgn-2.cpp -fopenmp -o LP-5_HPC_Asgn-2
● ubuntu@DESKTOP-HE9T2TD:~/LP5/Assignment2$ ./LP-5_HPC_Asgn-2
  Original Array: 3 6 7 5 3 5 6 2 9 1

  Sequential Bubble Sorted Array: 1 2 3 3 5 5 6 6 7 9
  Parallel Bubble Sorted Array: 1 2 3 3 5 5 6 6 7 9

  Bubble Sort Results:
  Sequential Time: 5.7e-07 seconds
  Parallel Time: 0.00128508 seconds

  Original Array: 2 7 0 9 3 6 0 6 2 6

  Sequential Merge Sorted Array: 0 0 2 2 3 6 6 6 7 9
  Parallel Merge Sorted Array: 0 0 2 2 3 6 6 6 7 9

  Merge Sort Results:
  Sequential Time: 1.161e-06 seconds
  Parallel Time: 0.00148309 seconds
○ ubuntu@DESKTOP-HE9T2TD:~/LP5/Assignment2$ 
```