```
#include <iostream> // --> Includes input/output stream library
#include <omp.h> // --> Includes OpenMP library for parallel programming
#include <vector> // --> Includes vector container from STL
using namespace std; // --> Allows using standard namespace to avoid prefixing std::

// Merge two sorted subarrays
void merge(vector<int> &arr, int l, int m, int r) // --> Merges two sorted subarrays in arr from index l to m and m+1 to r
{
  int n1 = m - l + 1; // --> Calculates size of first subarray
  int n2 = r - m; // --> Calculates size of second subarray

  vector<int> L(n1), R(n2); // --> Creates temporary vectors L and R
  for (int i = 0; i < n1; i++) // --> Copies elements to vector L
    L[i] = arr[l + i]; // --> Copies element from arr to L
  for (int j = 0; j < n2; j++) // --> Copies elements to vector R
    R[j] = arr[m + 1 + j]; // --> Copies element from arr to R

  int i = 0, j = 0, k = l; // --> Initializes indices for L, R, and arr

  while (i < n1 && j < n2) // --> Merges L and R until one is exhausted
    arr[k++] = (L[i] <= R[j]) ? L[i++] : R[j++]; // --> Places smaller element into arr
  while (i < n1) // --> Copies remaining elements from L
    arr[k++] = L[i++]; // --> Places remaining elements from L
  while (j < n2) // --> Copies remaining elements from R
    arr[k++] = R[j++]; // --> Places remaining elements from R
}

// Sequential Merge Sort
void mergeSortSequential(vector<int> &arr, int l, int r) // --> Recursively sorts arr from index l to r using sequential merge sort
{
  if (l < r) // --> Continues recursion while more than one element
  {
    int m = l + (r - l) / 2; // --> Calculates midpoint to divide array
    mergeSortSequential(arr, l, m); // --> Recursively sorts left half
    mergeSortSequential(arr, m + 1, r); // --> Recursively sorts right half
    merge(arr, l, m, r); // --> Merges sorted halves
  }
}

// Parallel Merge Sort using OpenMP
void mergeSortParallel(vector<int> &arr, int l, int r, int depth = 0) // --> Recursively sorts arr using parallel merge sort with depth control
{
  if (l < r) // --> Continues recursion while more than one element
  {
    int m = l + (r - l) / 2; // --> Calculates midpoint to divide array

    if (depth < 4) // --> Limits parallel depth to avoid thread overhead
    {
#pragma omp parallel sections // --> Begins parallel sections block
      {
#pragma omp section // --> First parallel section
        mergeSortParallel(arr, l, m, depth + 1); // --> Parallel sort for left half

#pragma omp section // --> Second parallel section
        mergeSortParallel(arr, m + 1, r, depth + 1); // --> Parallel sort for right half
      }
    }
    else
    {
      mergeSortSequential(arr, l, m); // --> Falls back to sequential sort for left half
```

```
        mergeSortSequential(arr, m + 1, r); // --> Falls back to sequential sort for right half
      }

      merge(arr, l, m, r); // --> Merges sorted halves
    }
}

int main() // --> Main function
{
    int n; // --> Variable to store number of elements
    cout << "Enter number of elements: "; // --> Prompts user for input size
    cin >> n; // --> Reads number of elements

    vector<int> arr(n), arrSeq(n); // --> Declares vectors for parallel and sequential sorting

    // User input for array elements
    cout << "Enter the elements:\n"; // --> Prompts user for elements
    for (int i = 0; i < n; i++) // --> Loops through each index
    {
        cin >> arr[i]; // --> Reads element into arr
    }
    arrSeq = arr; // --> Copies arr to arrSeq for sequential sort

    // Sequential sort timing
    double start = omp_get_wtime(); // --> Records start time for sequential sort
    mergeSortSequential(arrSeq, 0, n - 1); // --> Performs sequential merge sort
    double end = omp_get_wtime(); // --> Records end time for sequential sort
    double seqTime = end - start; // --> Calculates sequential sort duration

    // Parallel sort timing
    start = omp_get_wtime(); // --> Records start time for parallel sort
    mergeSortParallel(arr, 0, n - 1); // --> Performs parallel merge sort
    end = omp_get_wtime(); // --> Records end time for parallel sort
    double parTime = end - start; // --> Calculates parallel sort duration

    // Output sorted array
    cout << "\nSorted array:\n"; // --> Displays sorted array label
    for (int i = 0; i < n; i++) // --> Loops through sorted array
        cout << arr[i] << " "; // --> Prints each element
    cout << "\n"; // --> Newline after sorted array

    // Calculate performance metrics
    double speedup = seqTime / parTime; // --> Calculates speedup factor
    int numThreads = omp_get_max_threads(); // --> Gets max available threads
    double efficiency = speedup / numThreads; // --> Calculates efficiency

    // Display metrics
    cout << "\nPerformance Metrics:"; // --> Outputs performance header
    cout << "\n---------------------"; // --> Outputs separator
    cout << "\nSequential Time: " << seqTime << " seconds"; // --> Outputs sequential time
    cout << "\nParallel Time  : " << parTime << " seconds"; // --> Outputs parallel time
    cout << "\nSpeedup        : " << speedup; // --> Outputs speedup
    cout << "\nEfficiency     : " << efficiency << endl; // --> Outputs efficiency

    return 0; // --> Indicates successful program termination
}

// Run Commands:
// g++ -fopenmp -o merge_sort .\3_Merge_Sort.cpp // --> Command to compile with OpenMP
// .\merge_sort // --> Command to run executable
```