HPC 2 MERGE SORT

```
#include <iostream>
#include <vector>
#include <random>
#include <omp.h>
#include <chrono>

using namespace std;

bool isSorted(const vector<int>& arr) {
  int n = arr.size();
  for (int i = 1; i < n; i++) {
    if (arr[i] < arr[i - 1]) {
      return false;
    }
  }
  return true;
}

void merge(vector<int>& arr, int low, int mid, int high) {
  int i = low;
  int j = mid + 1;
  vector<int> merged(high - low + 1);

  for (int k = 0; k < merged.size(); k++) {
    if (i > mid) {
      merged[k] = arr[j++];
    } else if (j > high) {
      merged[k] = arr[i++];
    } else if (arr[i] <= arr[j]) {
      merged[k] = arr[i++];
    } else {
      merged[k] = arr[j++];
    }
  }

  for (int k = 0; k < merged.size(); k++) {
    arr[low + k] = merged[k];
  }
}

void mergeSort(vector<int>& arr, int low, int high) {
  if (low < high) {
    int mid = (low + high) / 2;

    // Recursively sort the left and right halves.
    mergeSort(arr, low, mid);
    mergeSort(arr, mid + 1, high);

    // Merge the sorted halves.
    #pragma omp parallel
    {
      #pragma omp for
      for (int i = low; i <= high; i++) {
        // Do nothing, just for parallelization.
      }

      merge(arr, low, mid, high);
    }
  }
}
```

```
int main() {
  int n = 1000;

  vector<int> arr(n);

  // Generate random numbers
  random_device rd;
  mt19937 generator(rd());
  uniform_int_distribution<int> distribution(1, 1000);

  for (int i = 0; i < n; i++) {
    arr[i] = distribution(generator);
  }

  cout << "Unsorted array: ";
  for (int i = 0; i < n; i++) {
    cout << arr[i] << " ";
  }
  cout << endl;

  // Merge sort.
  cout << "Sequential merge sort: ";
  auto start = chrono::high_resolution_clock::now();
  mergeSort(arr, 0, n - 1);
  auto end = chrono::high_resolution_clock::now();
  chrono::duration<double> elapsed = end - start;
  cout << elapsed.count() << " seconds" << endl;

  // Check if the array is sorted.
  if (isSorted(arr)) {
    cout << "The array is sorted." << endl;
  } else {
    cout << "The array is not sorted." << endl;
  }

  return 0;
}
```

Explanation of code

1. The code includes necessary libraries for input/output (`iostream`), vectors (`vector`), random number generation (`random`), OpenMP for parallelization (`omp`), and time measurement (`chrono`).

2. The `isSorted` function takes a vector `arr` as input and checks if it is sorted in ascending order. It iterates through the vector and compares each element with the previous one. If it finds any element smaller than the previous one, it returns `false`, indicating that the array is not sorted. Otherwise, it returns `true`.

3. The `merge` function performs the merge operation in the merge sort algorithm. It takes a vector `arr` and three indices `low`, `mid`, and `high`. It creates a temporary vector `merged` to store the merged result. The function merges the two subarrays from indices `low` to `mid` and `mid+1` to `high` into the `merged` vector. It compares the elements from the two subarrays and adds the smaller element to the `merged` vector. Finally, it copies the merged result back to the original `arr` vector.

4. The `mergeSort` function implements the merge sort algorithm. It takes a vector `arr` and two indices `low` and `high` to specify the range of elements to be sorted. If `low` is less than `high`, it calculates the middle index `mid` and recursively calls `mergeSort` on the left and right halves of the array. After that, it merges the sorted left and right halves using the `merge` function.

5. In the `main` function, an integer `n` is set to 1000, representing the number of elements in the array.

6. A vector `arr` of size `n` is created to store the array elements.

7. Random numbers are generated using the `random_device`, `mt19937`, and `uniform_int_distribution` classes. The `random_device` is used to obtain a random seed, the `mt19937` is the random number engine, and the `uniform_int_distribution` defines the range of random numbers (1 to 1000 in this case). The generated random numbers are stored in the `arr` vector.

8. The unsorted array is printed to the console.

9. The sequential merge sort is performed on the `arr` vector. The execution time is measured using the `chrono` library. The start time is recorded, `mergeSort` is called to sort the array, and the end time is recorded. The elapsed time is calculated by subtracting the start time from the end time.

10. The sorted array is printed to the console.

11. The `isSorted` function is called to check if the array is sorted. If it returns `true`, a message stating that the array is sorted is displayed. Otherwise, a message indicating that the array is not sorted is shown.

12. The program ends with a return statement.

Overall, this code generates a sequence of 1000 random numbers, performs sequential merge sort on the array, and checks if the array is sorted.

ORAL QUESTIONS –

1. OpenMP ? - OpenMP (Open Multi-Processing) is an application programming interface (API) that supports shared-memory parallel programming in C, C++, and Fortran. It is used to write parallel programs that can run on multicore processors, multiprocessor systems, and parallel computing cluster.
2. What is Merge Sort - Merge sort is a sorting algorithm that uses a divide-and-conquer approach to sort an array or a list of elements. The algorithm works by recursively dividing the input array into two halves, sorting each half, and then merging the sorted halves to produce a sorted output
3. What is parallel merge sort - Parallel merge sort is a parallelized version of the merge sort algorithm that takes advantage of multiple processors or cores to improve its performance. In parallel merge sort, the input array is divided into smaller subarrays, which are sorted in parallel using multiple processors or cores.
4. How to measure the performance of sequential and parallel algorithms?
   There are several metrics that can be used to measure the performance of sequential and parallel merge sort algorithms:
   a. Execution time: Execution time is the amount of time it takes for the algorithm complete its sorting operation. This metric can be used to compare the speed of sequential and parallel merge sort algorithms.
   b. Speedup: Speedup is the ratio of the execution time of the sequential merge sort algorithm to the execution time of the parallel merge sort algorithm.
   c. Efficiency: Efficiency is the ratio of the speedup to the number of processors or cores used in the parallel algorithm. This metric can be used to determine how well the parallel algorithm is utilizing the available resources.
   d. Scalability: Scalability is the ability of the algorithm to maintain its performance as the input size and number of processors or cores increase.

5. Example of merge sort –

12 31 25 8 32 17 40 42

Divide the array

12 31 25 8   32 17 40 42

Divide again

12 31   25 8   32 17   40 42

Again divide

12   31   25   8   32   17   40   42

Sort and merge

12 31   8 25   17 32   40 42

Merge again

8 12 25 31   17 32 40 42

Merge again

8 12 17 25 31 40 42