```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <chrono>
#include <random>
#include <omp.h>
#include <iomanip>
#include <numeric>
using namespace std;

class SortingBenchmark {
private:
    vector<int> sizes;
    int numRuns;

    vector<int> generateRandomArray(int size, int min, int max) {
        vector<int> arr(size);
        random_device rd;
        mt19937 gen(rd());
        uniform_int_distribution<> distrib(min, max);

        for (int i = 0; i < size; i++) {
            arr[i] = distrib(gen);
        }

        return arr;
    }

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

    template<typename SortFunc>
    double measureExecutionTime(SortFunc sortFunction, vector<int> arr, const
    string& name) {
        auto start = chrono::high_resolution_clock::now();
        sortFunction(arr);
        auto end = chrono::high_resolution_clock::now();

        chrono::duration<double, milli> duration = end - start;
        if (!isSorted(arr)) {
            cout << "Error: " << name << " failed to sort the array correctly!"
            << endl;
        }

        return duration.count();
    }

public:
    SortingBenchmark(const vector<int>& arraySizes, int runs) :
    sizes(arraySizes), numRuns(runs) {}

    void sequentialBubbleSort(vector<int>& arr) {
        int n = arr.size();
        for (int i = 0; i < n - 1; i++) {
            for (int j = 0; j < n - i - 1; j++) {
                if (arr[j] > arr[j + 1]) {
                    swap(arr[j], arr[j + 1]);
                }
            }
        }
    }

    void parallelBubbleSort(vector<int>& arr) {
        int n = arr.size();

        for (int phase = 0; phase < n; phase++) {
            if (phase % 2 == 0) {
                #pragma omp parallel for
                for (int i = 1; i < n - 1; i += 2) {
                    if (arr[i] > arr[i + 1]) {
                        swap(arr[i], arr[i + 1]);
                    }
                }
            } else {
                #pragma omp parallel for
                for (int i = 0; i < n - 1; i += 2) {
                    if (arr[i] > arr[i + 1]) {
                        swap(arr[i], arr[i + 1]);
                    }
                }
            }
        }
    }
```

```
void sequentialMerge(vector<int>& arr, vector<int>& temp, int left, int
mid, int right) {
    int i = left;
    int j = mid + 1;
    int k = left;

    while (i <= mid && j <= right) {
        if (arr[i] <= arr[j]) {
            temp[k++] = arr[i++];
        } else {
            temp[k++] = arr[j++];
        }
    }

    while (i <= mid) {
        temp[k++] = arr[i++];
    }

    while (j <= right) {
        temp[k++] = arr[j++];
    }

    for (i = left; i <= right; i++) {
        arr[i] = temp[i];
    }
}

void sequentialMergeSortHelper(vector<int>& arr, vector<int>& temp, int
left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        sequentialMergeSortHelper(arr, temp, left, mid);
        sequentialMergeSortHelper(arr, temp, mid + 1, right);

        sequentialMerge(arr, temp, left, mid, right);
    }
}

void sequentialMergeSort(vector<int>& arr) {
    vector<int> temp(arr.size());
    sequentialMergeSortHelper(arr, temp, 0, arr.size() - 1);
}
```

```
void parallelMergeSortHelper(vector<int>& arr, vector<int>& temp, int left,
int right, int depth = 0) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        if (depth < 2) {
            #pragma omp task shared(arr, temp)
            {
                parallelMergeSortHelper(arr, temp, left, mid, depth + 1);
            }

            #pragma omp task shared(arr, temp)
            {
              parallelMergeSortHelper(arr, temp, mid + 1, right, depth +1);
            }

            #pragma omp taskwait
        } else {
            sequentialMergeSortHelper(arr, temp, left, mid);
            sequentialMergeSortHelper(arr, temp, mid + 1, right);
        }

        #pragma omp critical
        {
            sequentialMerge(arr, temp, left, mid, right);
        }
    }
}

void parallelMergeSort(vector<int>& arr) {
    vector<int> temp(arr.size());

    #pragma omp parallel
    {
        #pragma omp single
        {
            parallelMergeSortHelper(arr, temp, 0, arr.size() - 1);
        }
    }
}

void runBenchmark() {
    cout << fixed << setprecision(2);
    cout << "------------------------------------------" << endl;
    cout << "| Array Size | Algorithm       | Time (ms) |" << endl;
cout << "--------------------------------------------" << endl;

for (const auto& size : sizes) {
    vector<double> seqBubbleTime(numRuns);
    vector<double> parBubbleTime(numRuns);
    vector<double> seqMergeTime(numRuns);
    vector<double> parMergeTime(numRuns);

    for (int run = 0; run < numRuns; run++) {
        vector<int> arr = generateRandomArray(size, 1, size * 10);

 vector<int> arr1 = arr;
 vector<int> arr2 = arr;
 vector<int> arr3 = arr;
 vector<int> arr4 = arr;

 seqBubbleTime[run] = measureExecutionTime([this](vector<int>& a)
 { this->sequentialBubbleSort(a); }, arr1, "Sequential Bubble Sort");

 parBubbleTime[run] = measureExecutionTime([this](vector<int>& a)
 { this->parallelBubbleSort(a); },  arr2, "Parallel Bubble Sort");

 seqMergeTime[run] = measureExecutionTime([this](vector<int>& a)
{this->sequentialMergeSort(a); }, arr3, "Sequential Merge Sort");

 parMergeTime[run] = measureExecutionTime([this](vector<int>& a)
{this->parallelMergeSort(a); },arr4, "Parallel Merge Sort");}

    double avgSeqBubble = accumulate(seqBubbleTime.begin(),
    seqBubbleTime.end(), 0.0) / numRuns;
    double avgParBubble = accumulate(parBubbleTime.begin(),
    parBubbleTime.end(), 0.0) / numRuns;
    double avgSeqMerge = accumulate(seqMergeTime.begin(),
    seqMergeTime.end(), 0.0) / numRuns;
    double avgParMerge = accumulate(parMergeTime.begin(),
    parMergeTime.end(), 0.0) / numRuns;

    cout << "| " << setw(10) << size << " | Sequential Bubble | "
          << setw(8) << avgSeqBubble << " |" << endl;
    cout << "| " << setw(10) << size << " | Parallel Bubble   | "
          << setw(8) << avgParBubble << " |" << endl;
    cout << "| " << setw(10) << size << " | Sequential Merge  | "
          << setw(8) << avgSeqMerge << " |" << endl;
    cout << "| " << setw(10) << size << " | Parallel Merge    | "
          << setw(8) << avgParMerge << " |" << endl;
```

```
            double bubbleSpeedup = avgSeqBubble / avgParBubble;
            double mergeSpeedup = avgSeqMerge / avgParMerge;

            cout << "| " << setw(10) << size << " | Bubble Speedup    | "
                  << setw(8) << bubbleSpeedup << "x |" << endl;
            cout << "| " << setw(10) << size << " | Merge Speedup     | "
                  << setw(8) << mergeSpeedup << "x |" << endl;
            cout << "---------------------------------------------" << endl;
        }
    }
};

int main() {
    vector<int> sizes = {1000, 10000, 50000, 100000};
    int numRuns = 5;

    SortingBenchmark benchmark(sizes, numRuns);
    benchmark.runBenchmark();

    return 0;
}
```

Output:

```
---------------------------------------------
| Array Size | Algorithm         | Time (ms) |
---------------------------------------------
|       1000 | Sequential Bubble |     5.84 |
|       1000 | Parallel Bubble   |     4.56 |
|       1000 | Sequential Merge  |     0.24 |
|       1000 | Parallel Merge    |     0.16 |
|       1000 | Bubble Speedup    |    1.28x |
|       1000 | Merge Speedup     |    1.49x |
---------------------------------------------
|      10000 | Sequential Bubble |   450.54 |
|      10000 | Parallel Bubble   |   261.49 |
|      10000 | Sequential Merge  |     3.06 |
|      10000 | Parallel Merge    |     1.84 |
|      10000 | Bubble Speedup    |    1.72x |
|      10000 | Merge Speedup     |    1.67x |
---------------------------------------------
```

```
--------------------------------------------
|      50000 | Sequential Bubble | 14643.43 |
|      50000 | Parallel Bubble   |  6792.09 |
|      50000 | Sequential Merge  |    16.31 |
|      50000 | Parallel Merge    |    10.07 |
|      50000 | Bubble Speedup    |    2.16x |
|      50000 | Merge Speedup     |    1.62x |
--------------------------------------------
|     100000 | Sequential Bubble | 61338.08 |
|     100000 | Parallel Bubble   | 26172.52 |
|     100000 | Sequential Merge  |    31.69 |
|     100000 | Parallel Merge    |    17.17 |
|     100000 | Bubble Speedup    |    2.34x |
|     100000 | Merge Speedup     |    1.85x |
--------------------------------------------
```