MERGESORT-BUBBLE SORT

g++ -fopenmp -o output filename.cpp

./output

1

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

using namespace std;

void merge(int arr[], int low, int mid, int high) {
   // Create arrays of left and right partititons
   int n1 = mid - low + 1;
   int n2 = high - mid;

   int left[n1];
   int right[n2];

   // Copy all left elements
   for (int i = 0; i < n1; i++) left[i] = arr[low + i];

   // Copy all right elements
   for (int j = 0; j < n2; j++) right[j] = arr[mid + 1 + j];

   // Compare and place elements
   int i = 0, j = 0, k = low;
```

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

    // If any elements are left out
    while (i < n1) {
        arr[k] = left[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = right[j];
        j++;
        k++;
    }
}
```

```
void parallelMergeSort(int arr[], int low, int high) {
    if (low < high) {
        int mid = (low + high) / 2;

        #pragma omp parallel sections
        {
            #pragma omp section
            {
                parallelMergeSort(arr, low, mid);
            }

            #pragma omp section
            {
                parallelMergeSort(arr, mid + 1, high);
            }
        }
        merge(arr, low, mid, high);
    }
}

void mergeSort(int arr[], int low, int high) {
    if (low < high) {
        int mid = (low + high) / 2;
        mergeSort(arr, low, mid);
        mergeSort(arr, mid + 1, high);
```

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

int main() {
    int n = 10;
    int arr[n];
    double start_time, end_time;

    // Create an array with numbers starting from n to 1.
    for(int i = 0, j = n; i < n; i++, j--) arr[i] = j;

    // Measure Sequential Time
    start_time = omp_get_wtime();
    mergeSort(arr, 0, n - 1);
    end_time = omp_get_wtime();
    cout << "Time taken by sequential algorithm: " << end_time - start_time << "
seconds\n";

    // Reset the array
    for(int i = 0, j = n; i < n; i++, j--) arr[i] = j;

    //Measure Parallel time
    start_time = omp_get_wtime();
    parallelMergeSort(arr, 0, n - 1);
    end_time = omp_get_wtime();
```

```
    cout << "Time taken by parallel algorithm: " << end_time - start_time << "
seconds";


    return 0;
}


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

using namespace std;

void bubble(int array[], int n){
    for (int i = 0; i < n - 1; i++){
        for (int j = 0; j < n - i - 1; j++){
            if (array[j] > array[j + 1]) swap(array[j], array[j + 1]);
        }
    }
}

void pBubble(int array[], int n){
    //Sort odd indexed numbers
    for(int i = 0; i < n; ++i){
        #pragma omp for
        for (int j = 1; j < n; j += 2){
        if (array[j] < array[j-1])
        {
          swap(array[j], array[j - 1]);
```

```
      }
    }

    // Synchronize
    #pragma omp barrier

    //Sort even indexed numbers
    #pragma omp for
    for (int j = 2; j < n; j += 2){
      if (array[j] < array[j-1])
      {
        swap(array[j], array[j - 1]);
      }
    }
  }
}

void printArray(int arr[], int n){
    for(int i = 0; i < n; i++) cout << arr[i] << " ";
    cout << "\n";
}

int main(){
    // Set up variables
    int n = 10;
    int arr[n];
```

```
    int brr[n];
    double start_time, end_time;

    // Create an array with numbers starting from n to 1
    for(int i = 0, j = n; i < n; i++, j--) arr[i] = j;

    // Sequential time
    start_time = omp_get_wtime();
    bubble(arr, n);
    end_time = omp_get_wtime();
    cout << "Sequential Bubble Sort took : " << end_time - start_time << "
seconds.\n";
    printArray(arr, n);

    // Reset the array
    for(int i = 0, j = n; i < n; i++, j--) arr[i] = j;

    // Parallel time
    start_time = omp_get_wtime();
    pBubble(arr, n);
    end_time = omp_get_wtime();
    cout << "Parallel Bubble Sort took : " << end_time - start_time << "
seconds.\n";
    printArray(arr, n);
}
```

2

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <chrono>
#include <omp.h>

using namespace std;

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

// Function to perform parallel bubble sort
void parallel_bubble_sort(int arr[], int n)
{

// Parallelize the outer loop
#pragma omp parallel for shared(arr, n)
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n - 1; j++)
        {
            if (arr[j] > arr[j + 1])
            {
                // Swap elements if they are in the wrong order
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

// Function to merge two sorted arrays
void merge(int arr[], int l, int m, int r)
{
    int n1 = m - l + 1;
    int n2 = r - m;
    int L[n1], R[n2];
    for (int i = 0; i < n1; i++)
    {
        L[i] = arr[l + i];
```

```
    }
    for (int j = 0; j < n2; j++)
    {
        R[j] = arr[m + 1 + j];
    }
    int i = 0, j = 0, k = l;
    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 sequential_merge_sort(int arr[], int l, int r)
{
    if (l < r)
    {
        int m = l + (r - l) / 2;
        sequential_merge_sort(arr, l, m);
        sequential_merge_sort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}

// Function to perform parallel merge sort
void parallel_merge_sort(int arr[], int low, int high)
{
    if (low < high)
    {
        int mid = (low + high) / 2;
#pragma omp parallel sections
        {
#pragma omp section
            parallel_merge_sort(arr, low, mid);
#pragma omp section
            parallel_merge_sort(arr, mid + 1, high);
```

```
        }
        merge(arr, low, mid, high);
    }
}
void print_array(int *arr, int n)
{
    for (int i = 0; i < n; i++)
    {
        cout << arr[i] << " ";
    }
}
// Function to test the performance of sequential and parallel bubble
sort
void test_bubble_sort_performance(int n)
{
    int *arr = new int[n];
    int *arr_copy = new int[n];

    // Initialize the array with random values
    srand(time(NULL));
    for (int i = 0; i < n; i++)
    {
        arr[i] = rand();
        arr_copy[i] = arr[i];
    }

    // Measure the execution time of sequential bubble sort
    auto start_time = chrono::high_resolution_clock::now();
    sequential_bubble_sort(arr, n);
    auto end_time = chrono::high_resolution_clock::now();
    auto sequential_sort_time =
chrono::duration_cast<chrono::microseconds>(end_time -
start_time).count();

    // Measure the execution time of parallel bubble sort
    start_time = chrono::high_resolution_clock::now();
    parallel_bubble_sort(arr_copy, n);
    end_time = chrono::high_resolution_clock::now();
    auto parallel_sort_time =
chrono::duration_cast<chrono::microseconds>(end_time -
start_time).count();

    // Print the execution times of the sequential and parallel bubble
sort
    cout << "Sequential bubble sort time: " << sequential_sort_time <<
" microseconds" << endl;
    print_array(arr, n);

    cout << "\nParallel bubble sort time: " << parallel_sort_time << "
microseconds" << endl;
    print_array(arr_copy, n);

    // Free the memory allocated for the arrays
    delete[] arr;
    delete[] arr_copy;
}

void test_merge_sort_performance(int n)
```

```
{
    int *arr = new int[n];
    int *arr_copy = new int[n];

    // Initialize array with random values
    srand(time(NULL));
    for (int i = 0; i < n; i++)
    {
        arr[i] = rand() % 1000;
        arr_copy[i] = arr[i];
    }

    // Perform sequential merge sort and measure time
    double start = omp_get_wtime();
    sequential_merge_sort(arr, 0, n - 1);
    double end = omp_get_wtime();
    std::cout << "Sequential merge sort took " << end - start << "
seconds\n";
    print_array(arr, n);

    // Perform parallel merge sort and measure time
    // int threads = omp_get_max_threads();
    start = omp_get_wtime();
    // int num_threads = omp_get_max_threads();
    parallel_merge_sort(arr_copy, 0, n - 1);
    end = omp_get_wtime();
    std::cout << "\nParallel merge sort took " << end - start << "
seconds\n";
    print_array(arr_copy, n);

    delete[] arr;
}

// Example usage
int main()
{
    int n = 10;
    test_bubble_sort_performance(n);
    cout << "\n------------------\n";
    test_merge_sort_performance(n);
    return 0;
}
```