**Problem Statement:** Write a program to implement Parallel Bubble Sort and Merge sort using OpenMP. Use existing algorithms and measure the performance of sequential and parallel algorithms.

**Code:**

#include <iostream>

#include <vector>

#include <cstdlib>

#include <ctime>

#include <omp.h>

using namespace std;

// Sequential Bubble Sort

void bubbleSort(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]);

}

// Parallel Bubble Sort using Odd-Even Transposition

void parallelBubbleSort(vector<int>& arr) {

int n = arr.size();

for (int i = 0; i < n; i++) {

#pragma omp parallel for

for (int j = (i % 2); j < n - 1; j += 2) {

if (arr[j] > arr[j + 1])

swap(arr[j], arr[j + 1]);

}

}

}

// Sequential Merge Sort

void merge(vector<int>& arr, int l, int m, int r) {

int n1 = m - l + 1, n2 = r - m;

vector<int> L(n1), R(n2);

for (int i = 0; i < n1; i++) L[i] = arr[l + i];

for (int i = 0; i < n2; i++) R[i] = arr[m + 1 + i];

int i = 0, j = 0, k = l;

while (i < n1 && j < n2)

arr[k++] = (L[i] <= R[j]) ? L[i++] : R[j++];

while (i < n1) arr[k++] = L[i++];

while (j < n2) arr[k++] = R[j++];

}

void mergeSort(vector<int>& arr, int l, int r) {

if (l < r) {

int m = l + (r - l) / 2;

mergeSort(arr, l, m);

mergeSort(arr, m + 1, r);

merge(arr, l, m, r);

}

}

// Parallel Merge Sort using OpenMP

void parallelMergeSort(vector<int>& arr, int l, int r, int depth = 0) {

if (l < r) {

int m = l + (r - l) / 2;

if (depth < 4) { // limit depth to avoid oversubscription

#pragma omp parallel sections

{

#pragma omp section

parallelMergeSort(arr, l, m, depth + 1);

#pragma omp section

parallelMergeSort(arr, m + 1, r, depth + 1);

}

} else {

mergeSort(arr, l, m);

mergeSort(arr, m + 1, r);

}

merge(arr, l, m, r);

}

}

int main() {

int n;

cout << "Enter size of array: ";

cin >> n;

vector<int> arr(n);

srand(time(0));

for (int i = 0; i < n; i++)

arr[i] = rand() % 10000;

// Sequential Bubble Sort

vector<int> b1 = arr;

double start = omp\_get\_wtime();

bubbleSort(b1);

double end = omp\_get\_wtime();

cout << "Sequential Bubble Sort time: " << (end - start) << " seconds\n";

// Parallel Bubble Sort

vector<int> b2 = arr;

start = omp\_get\_wtime();

parallelBubbleSort(b2);

end = omp\_get\_wtime();

cout << "Parallel Bubble Sort time: " << (end - start) << " seconds\n";

// Sequential Merge Sort

vector<int> m1 = arr;

start = omp\_get\_wtime();

mergeSort(m1, 0, n - 1);

end = omp\_get\_wtime();

cout << "Sequential Merge Sort time: " << (end - start) << " seconds\n";

// Parallel Merge Sort

vector<int> m2 = arr;

start = omp\_get\_wtime();

parallelMergeSort(m2, 0, n - 1);

end = omp\_get\_wtime();

cout << "Parallel Merge Sort time: " << (end - start) << " seconds\n";

return 0;

}

**Steps to run the code:**

1. Enter Command “ g++ -fopenmp -O2 file\_name -o file\_name ”
2. Enter command “.\file\_name ”

**Output:**

C:\Users\Shravan\OneDrive\Desktop\Engineering Degree Stuff\4th Year Stuff\8th Sem Stuff\LP-5 Problem Statement & Programs\Programs\HPC Practical 2>g++ -fopenmp -O2 bubble\_merge\_sort.cpp -o bubble\_merge\_sort

C:\Users\Shravan\OneDrive\Desktop\Engineering Degree Stuff\4th Year Stuff\8th Sem Stuff\LP-5 Problem Statement & Programs\Programs\HPC Practical 2>.\bubble\_merge\_sort

Enter size of array: 50000

Sequential Bubble Sort time: 18.643 seconds

Parallel Bubble Sort time: 11.143 seconds

Sequential Merge Sort time: 0.029 seconds

Parallel Merge Sort time: 0.0480001 seconds

The code begins by including several standard C++ libraries. #include <iostream> brings in the iostream library, which provides input and output functionalities like printing to the console (cout) and reading user input (cin). #include <vector> includes the vector library, allowing the use of dynamic arrays (vector), which are used to store the elements to be sorted. #include <cstdlib> includes the cstdlib library, which provides general utility functions, including the rand() function for generating pseudo-random numbers and srand() for seeding the random number generator. #include <ctime> includes the ctime library, which provides functions for working with time, such as time(), used here to seed the random number generator. #include <omp.h> includes the OpenMP header file, which provides directives and functions for parallel programming, enabling the parallel versions of the sorting algorithms. using namespace std; brings all identifiers from the standard namespace into the current scope, making it easier to use elements like cout, cin, vector, and swap without explicitly specifying std::.

The code then defines a function bubbleSort which implements the sequential bubble sort algorithm. void bubbleSort(vector<int>& arr) declares a function that takes a reference to a vector of integers arr as input and sorts it in place. int n = arr.size(); gets the number of elements in the input vector arr and stores it in the integer variable n. The outer for (int i = 0; i < n-1; i++) loop iterates through the array n-1 times. The inner for (int j = 0; j < n-i-1; j++) loop iterates through the unsorted part of the array. if (arr[j] > arr[j+1]) checks if the current element arr[j] is greater than the next element arr[j+1]. If it is, swap(arr[j], arr[j+1]); swaps the positions of these two adjacent elements, effectively "bubbling" the larger elements towards the end of the array.

Next, the code defines parallelBubbleSort, which implements a parallel version of bubble sort using the odd-even transposition sort algorithm. void parallelBubbleSort(vector<int>& arr) declares a function that takes a reference to an integer vector arr and sorts it in parallel. int n = arr.size(); gets the size of the array. The outer for (int i = 0; i < n; i++) loop iterates n times. #pragma omp parallel for is an OpenMP directive that instructs the compiler to parallelize the following for loop across multiple threads. for (int j = (i % 2); j < n - 1; j += 2) iterates through the array, but the starting index j depends on whether the current iteration i is even or odd (i % 2). If i is even, it starts at 0 (even indices); if i is odd, it starts at 1 (odd indices). The loop increments by 2 in each step, processing either all even-indexed pairs or all odd-indexed pairs in parallel. if (arr[j] > arr[j + 1]) compares adjacent elements. swap(arr[j], arr[j + 1]); swaps them if they are in the wrong order. This odd-even transposition ensures that after n phases, the array will be sorted.

The code then defines functions for sequential merge sort. void merge(vector<int>& arr, int l, int m, int r) is a helper function that merges two sorted subarrays of arr. It takes the array arr, the left index l, the middle index m, and the right index r as input. int n1 = m - l + 1, n2 = r - m; calculates the sizes of the two subarrays to be merged. vector<int> L(n1), R(n2); creates two temporary vectors L and R to store the left and right subarrays, respectively. The two for loops copy the elements from the original array into the temporary L and R vectors. int i = 0, j = 0, k = l; initializes index variables i for L, j for R, and k for the merged part of arr. The while (i < n1 && j < n2) loop compares elements from L and R and copies the smaller one back to arr[k], incrementing the corresponding index (i or j) and k. The two subsequent while loops handle any remaining elements in L or R after one of them is exhausted. void mergeSort(vector<int>& arr, int l, int r) is the main sequential merge sort function. It takes the array arr, the left index l, and the right index r. if (l < r) is the base case for the recursion: if the left index is less than the right index, it means there is more than one element to sort. int m = l + (r - l) / 2; calculates the middle index m. mergeSort(arr, l, m); recursively sorts the left half of the array. mergeSort(arr, m + 1, r); recursively sorts the right half of the array. merge(arr, l, m, r); merges the two sorted halves.

The code then defines parallelMergeSort, a parallel version of merge sort using OpenMP. void parallelMergeSort(vector<int>& arr, int l, int r, int depth = 0) takes the array, left index, right index, and an optional depth parameter (initialized to 0) to control the level of parallelism. if (l < r) is the base case. int m = l + (r - l) / 2; calculates the middle index. if (depth < 4) checks if the current recursion depth is less than 4 (this is a heuristic to limit the creation of too many threads). #pragma omp parallel sections is an OpenMP directive that creates a parallel region with multiple sections. #pragma omp section marks a block of code to be executed by one thread. The two #pragma omp section blocks recursively call parallelMergeSort for the left and right halves of the array in parallel, incrementing the depth. If the depth is not less than 4, the code falls back to the sequential mergeSort for the subarrays to avoid excessive thread creation overhead. Finally, merge(arr, l, m, r); merges the sorted (either sequentially or recursively parallelized) left and right halves.

The int main() function is the entry point of the program. int n; declares an integer n for the size of the array. cout << "Enter size of array: "; prompts the user to enter the array size. cin >> n; reads the size from the user. vector<int> arr(n); creates a vector of integers arr of size n. srand(time(0)); seeds the pseudo-random number generator using the current time, ensuring different random numbers each time the program runs. The for loop for (int i = 0; i < n; i++) arr[i] = rand() % 10000; fills the array arr with n random integers between 0 and 9999. The code then proceeds to perform and time the sequential bubble sort. vector<int> b1 = arr; creates a copy of the original array arr named b1 to be sorted by the sequential bubble sort. double start = omp\_get\_wtime(); records the starting time using OpenMP's wall-clock time function. bubbleSort(b1); calls the sequential bubble sort function on b1. double end = omp\_get\_wtime(); records the ending time. cout << "Sequential Bubble Sort time: " << (end - start) << " seconds\n"; calculates and prints the execution time. The same process is repeated for the parallel bubble sort using a copy b2 of the original array and the parallelBubbleSort function. Similarly, the sequential merge sort is performed on a copy m1 using mergeSort, and its execution time is measured and printed. Finally, the parallel merge sort is performed on a copy m2 using parallelMergeSort, and its execution time is also measured and printed. return 0; indicates successful program execution.