ASSIGNMENT NO – 1

#include <iostream>

#include <vector>

#include <queue>

#include <stack>

#include <omp.h>

using namespace std;

class Graph {

    int V; // Number of vertices

    vector<vector<int>> adj; // Adjacency list

public:

    Graph(int V) : V(V), adj(V) {}

    // Add an edge to the undirected graph

    void addEdge(int v, int w) {

        adj[v].push\_back(w);

        adj[w].push\_back(v);

    }

    // Print the adjacency list

    void printGraph() {

        cout << "\nGraph Adjacency List:\n";

        for (int i = 0; i < V; i++) {

            cout << "Vertex " << i << ": ";

            for (int neighbor : adj[i]) {

                cout << neighbor << " ";

            }

            cout << endl;

        }

    }

    // Parallel Breadth-First Search

    void parallelBFS(int start) {

        vector<bool> visited(V, false);

        queue<int> q;

        double start\_time = omp\_get\_wtime();

        visited[start] = true;

        q.push(start);

        cout << "\nParallel BFS starting from vertex " << start << ":\n";

        while (!q.empty()) {

            #pragma omp parallel

            {

                #pragma omp single

                {

                    int level\_size = q.size();

                    for (int i = 0; i < level\_size; i++) {

                        int v = q.front();

                        q.pop();

                        cout << v << " ";

                        // Process neighbors in parallel

                        #pragma omp task firstprivate(v) shared(visited, q)

                        {

                            for (int neighbor : adj[v]) {

                                #pragma omp critical

                                {

                                    if (!visited[neighbor]) {

                                        visited[neighbor] = true;

                                        q.push(neighbor);

                                    }

                                }

                            }

                        }

                    }

                }

            }

        }

        double end\_time = omp\_get\_wtime();

        cout << "\n\nParallel BFS completed in " << (end\_time - start\_time) \* 1000 << " milliseconds\n";

    }

    // Parallel Depth-First Search (using iterative approach)

    void parallelDFS(int start) {

        vector<bool> visited(V, false);

        stack<int> s;

        double start\_time = omp\_get\_wtime();

        s.push(start);

        visited[start] = true;

        cout << "\nParallel DFS starting from vertex " << start << ":\n";

        while (!s.empty()) {

            int v;

            #pragma omp critical

            {

                v = s.top();

                s.pop();

            }

            cout << v << " ";

            // Process neighbors in parallel

            #pragma omp parallel for

            for (int i = 0; i < adj[v].size(); i++) {

                int neighbor = adj[v][i];

                #pragma omp critical

                {

                    if (!visited[neighbor]) {

                        visited[neighbor] = true;

                        s.push(neighbor);

                    }

                }

            }

        }

        double end\_time = omp\_get\_wtime();

        cout << "\n\nParallel DFS completed in " << (end\_time - start\_time) \* 1000 << " milliseconds\n";

    }

};

int main() {

    int V, E, start\_vertex;

    int num\_threads;

    cout << "Enter number of vertices: ";

    cin >> V;

    cout << "Enter number of edges: ";

    cin >> E;

    Graph g(V);

    cout << "Enter edges (vertex pairs, 0-based indexing):\n";

    for (int i = 0; i < E; i++) {

        int v, w;

        cin >> v >> w;

        if (v >= V || w >= V || v < 0 || w < 0) {

            cout << "Invalid vertex! Vertices must be between 0 and " << V-1 << endl;

            i--; // Retry this edge

            continue;

        }

        g.addEdge(v, w);

    }

    cout << "Enter starting vertex for BFS/DFS (0-" << V-1 << "): ";

    cin >> start\_vertex;

    if (start\_vertex < 0 || start\_vertex >= V) {

        cout << "Invalid starting vertex! Using 0 as default.\n";

        start\_vertex = 0;

    }

    cout << "Enter number of threads to use: ";

    cin >> num\_threads;

    omp\_set\_num\_threads(num\_threads);

    g.printGraph();

    // Run parallel BFS

    g.parallelBFS(start\_vertex);

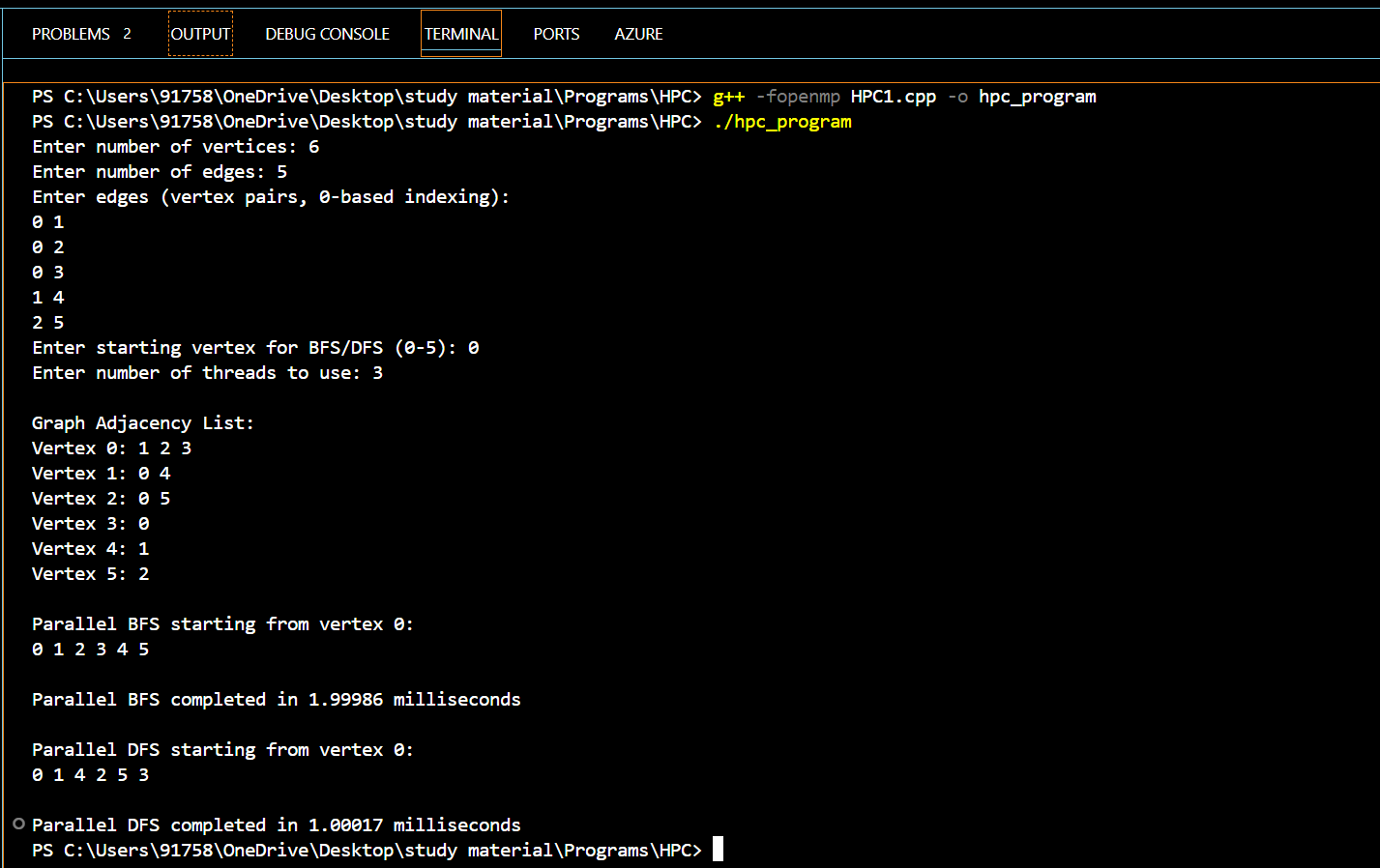
    // Run parallel DFS

    g.parallelDFS(start\_vertex);

    return 0;

}

OUTPUT



ASSIGNMENT NO – 2

#include <iostream>

#include <ctime>

#include <cstdlib>

#include <omp.h>

#include <chrono>

using namespace std;

using namespace std::chrono;

void bubbleSort(int arr[], int n)

{

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 merge(int arr[], int l, int m, int r)

{

int i, j, k;

int n1 = m - l + 1;

int n2 = r - m;

int \*L = new int[n1];

int \*R = new int[n2];

for (i = 0; i < n1; ++i)

{

L[i] = arr[l + i];

}

for (j = 0; j < n2; ++j)

{

R[j] = arr[m + 1 + j];

}

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;

}

delete[] L;

delete[] R;

}

void mergeSort(int arr[], int l, int r)

{

if (l < r)

{

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

#pragma omp parallel sections

{

#pragma omp section

{

mergeSort(arr, l, m);

}

#pragma omp section

{

mergeSort(arr, m + 1, r);

}

}

merge(arr, l, m, r);

}

}

void printArray(int arr[], int size)

{

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

{

cout << arr[i] << " ";

}

cout << endl;

}

int main()

{

int n;

cout << "Enter the size of the array: ";

cin >> n;

int \*arr = new int[n];

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

{

cout << "Enter element : ";

cin >> arr[i];

}

cout << "Original array: ";

printArray(arr, n);

// Time measurement variables

auto start = high\_resolution\_clock::now();

auto end = high\_resolution\_clock::now();

auto duration = duration\_cast<microseconds>(end - start);

// Sequential Bubble Sort

start = high\_resolution\_clock::now();

bubbleSort(arr, n);

end = high\_resolution\_clock::now();

duration = duration\_cast<microseconds>(end - start);

cout << "\nSequential Bubble Sorted array: ";

printArray(arr, n);

cout << "Time taken: " << duration.count() << " microseconds\n";

// Parallel Bubble Sort

start = high\_resolution\_clock::now();

#pragma omp parallel

{

#pragma omp single

{

bubbleSort(arr, n);

}

}

end = high\_resolution\_clock::now();

duration = duration\_cast<microseconds>(end - start);

cout << "Parallel Bubble Sorted array: ";

printArray(arr, n);

cout << "Time taken: " << duration.count() << " microseconds\n";

// Sequential Merge Sort

start = high\_resolution\_clock::now();

mergeSort(arr, 0, n - 1);

end = high\_resolution\_clock::now();

duration = duration\_cast<microseconds>(end - start);

cout << "Sequential Merge Sorted array: ";

printArray(arr, n);

cout << "Time taken: " << duration.count() << " microseconds\n";

// Parallel Merge Sort

start = high\_resolution\_clock::now();

#pragma omp parallel

{

#pragma omp single

{

mergeSort(arr, 0, n - 1);

}

}

end = high\_resolution\_clock::now();

duration = duration\_cast<microseconds>(end - start);

cout << "Parallel Merge Sorted array: ";

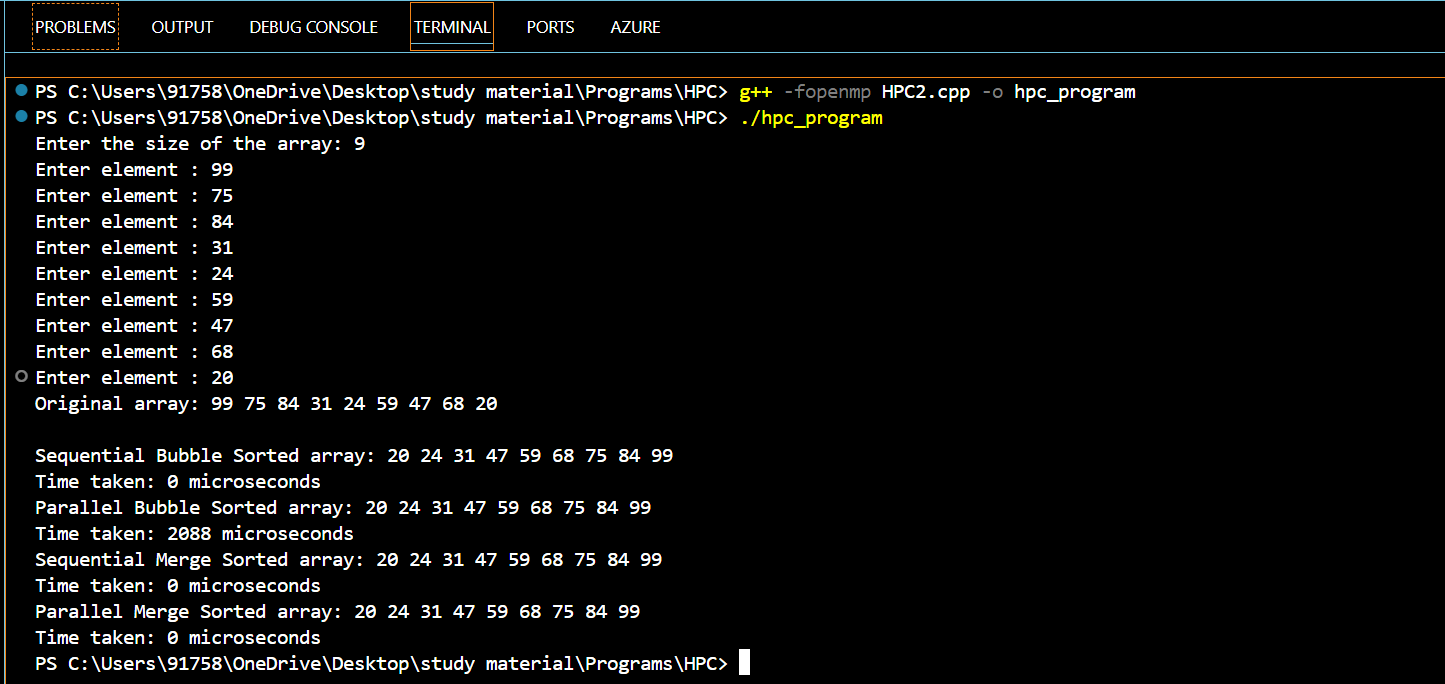
printArray(arr, n);

cout << "Time taken: " << duration.count() << " microseconds\n";

delete[] arr;

return 0;

}OUTPUT



ASSIGNMENT NO – 3

#include <iostream>

#include <omp.h>

#include <climits>

using namespace std;

// Function to find the minimum value in an array using parallel reduction

void min\_reduction(int arr[], int n) {

  int min\_value = INT\_MAX;

  // Use OpenMP parallel for loop with reduction clause (min)

  #pragma omp parallel for reduction(min: min\_value)

  for (int i = 0; i < n; i++) {

    if (arr[i] < min\_value) {

      min\_value = arr[i];

    }

  }

  cout << "Minimum value: " << min\_value << endl;

}

// Function to find the maximum value in an array using parallel reduction

void max\_reduction(int arr[], int n) {

  int max\_value = INT\_MIN;

  // Use OpenMP parallel for loop with reduction clause (max)

  #pragma omp parallel for reduction(max: max\_value)

  for (int i = 0; i < n; i++) {

    if (arr[i] > max\_value) {

      max\_value = arr[i];

    }

  }

  cout << "Maximum value: " << max\_value << endl;

}

// Function to calculate the sum of elements in an array using parallel reduction

void sum\_reduction(int arr[], int n) {

  int sum = 0;

  // Use OpenMP parallel for loop with reduction clause (+)

  #pragma omp parallel for reduction(+: sum)

  for (int i = 0; i < n; i++) {

    sum += arr[i];

  }

  cout << "Sum: " << sum << endl;

}

// Function to calculate the average of elements in an array using parallel reduction

void average\_reduction(int arr[], int n) {

  int sum = 0;

  // Use OpenMP parallel for loop with reduction clause (+)

  #pragma omp parallel for reduction(+: sum)

  for (int i = 0; i < n; i++) {

    sum += arr[i];

  }

  // Calculate average using the reduced sum (note: consider division by n-1 for unbiased average)

  double average = (double)sum / (n - 1);

  cout << "Average: " << average << endl;

}

int main() {

  int n;

  cout << "\nEnter the total number of elements: ";

  cin >> n;

  int \*arr = new int[n];

  for (int i = 0; i < n; i++) {

cout << "\nEnter the element : ";

    cin >> arr[i];

  }

  min\_reduction(arr, n);

  max\_reduction(arr, n);

  sum\_reduction(arr, n);

  average\_reduction(arr, n);

  delete[] arr;

  return 0;

}

OUTPUT

