Assignment 1

Code

```
#include <iostream>
#include <vector>
#include <queue>
#include <ctime>
#include <omp.h>

using namespace std;

// Function to perform BFS from a given vertex
void bfs(int startVertex, vector<bool> &visited, vector<vector<int>> &graph)
{
   // Create a queue for BFS
   queue<int> q;

   // Mark the start vertex as visited and enqueue it
   visited[startVertex] = true;
   q.push(startVertex);

   // Loop until the queue is empty
   while (!q.empty())
   {
      // Dequeue a vertex from the queue
      int v = q.front();
      q.pop();

// Enqueue all adjacent vertices that are not visited
#pragma omp parallel for
      for (int i = 0; i < graph[v].size(); i++)
      {
         int u = graph[v][i];
#pragma omp critical
         {
            if (!visited[u])
            {
               visited[u] = true;
               q.push(u);
            }
         }
      }
   }
}

// Parallel Breadth-First Search
void parallelBFS(vector<vector<int>> &graph, int numCores)
{
   int numVertices = graph.size();
   vector<bool> visited(numVertices, false); // Keep track of visited vertices

   double startTime = omp_get_wtime(); // Start timer

// Perform BFS from all unvisited vertices using specified number of cores
```

```
#pragma omp parallel for num_threads(numCores)
    for (int v = 0; v < numVertices; v++)
    {
        if (!visited[v])
        {
            bfs(v, visited, graph);
        }
    }

    double endTime = omp_get_wtime(); // End timer

    cout << "Number of cores used: " << numCores << endl;
    cout << "Time taken: " << endTime - startTime << " seconds" << endl;
    cout << "----------------------" << endl;
}

int main()
{
    // Generate a random graph with 10,000 vertices and 50,000 edges
    int numVertices = 10000;
    int numEdges = 50000;
    vector<vector<int>> graph(numVertices);
    srand(time(0));
    for (int i = 0; i < numEdges; i++)
    {
        int u = rand() % numVertices;
        int v = rand() % numVertices;
        graph[u].push_back(v);
        graph[v].push_back(u);
    }

    // Array containing number of cores
    int numCoresArr[] = {1, 2, 3, 4, 5, 6, 7, 8};

    // Loop over different number of cores and execute parallel BFS
    for (int i = 0; i < sizeof(numCoresArr) / sizeof(numCoresArr[0]); i++)
    {
        int numCores = numCoresArr[i];
        cout << "Running parallel BFS with " << numCores << " core(s)..." << endl;
        parallelBFS(graph, numCores);
    }

    return 0;
}
```

Output

```
PS D:\Gaurav\College\4 BE\Sem 8\High Performance Computing\Lab Assignments\1> g++ dfs.cpp -o dfs -fopenmp -pthread
PS D:\Gaurav\College\4 BE\Sem 8\High Performance Computing\Lab Assignments\1> ./dfs
Running parallel DFS with 1 core(s)...
Number of cores used: 1
Time taken: 10.921 seconds
------------------------
Running parallel DFS with 2 core(s)...
Number of cores used: 2
Time taken: 0.0769999 seconds
------------------------
Running parallel DFS with 3 core(s)...
Number of cores used: 3
Time taken: 0.0639999 seconds
------------------------
Running parallel DFS with 4 core(s)...
Number of cores used: 4
Time taken: 0.0469999 seconds
------------------------
Running parallel DFS with 5 core(s)...
Number of cores used: 5
Time taken: 0.043 seconds
------------------------
Running parallel DFS with 6 core(s)...
Number of cores used: 6
Time taken: 0.0400002 seconds
------------------------
Running parallel DFS with 7 core(s)...
Number of cores used: 7
Time taken: 0.0380001 seconds
------------------------
Running parallel DFS with 8 core(s)...
Number of cores used: 8
Time taken: 0.0370002 seconds
------------------------
PS D:\Gaurav\College\4 BE\Sem 8\High Performance Computing\Lab Assignments\1>
```

Assigment 2

Code

```
#include <iostream>
#include <vector>
#include <stack>
#include <ctime>
#include <omp.h>

using namespace std;

// Function to perform DFS from a given vertex
void dfs(int startVertex, vector<bool> &visited, vector<vector<int>> &graph)
{
   // Create a stack for DFS
   stack<int> s;

   // Mark the start vertex as visited and push it onto the stack
   visited[startVertex] = true;
   s.push(startVertex);

   // Loop until the stack is empty
   while (!s.empty())
   {
      // Pop a vertex from the stack
      int v = s.top();
      s.pop();

// Push all adjacent vertices that are not visited onto the stack
#pragma omp parallel for
      for (int i = 0; i < graph[v].size(); i++)
      {
         int u = graph[v][i];
#pragma omp critical
         {
            if (!visited[u])
            {
               visited[u] = true;
               s.push(u);
            }
         }
      }
   }
}

// Parallel Depth-First Search
void parallelDFS(vector<vector<int>> &graph, int numCores)
{
   int numVertices = graph.size();
   vector<bool> visited(numVertices, false); // Keep track of visited vertices

   double startTime = omp_get_wtime(); // Start timer

// Perform DFS from all unvisited vertices using specified number of cores
```

```
#pragma omp parallel for num_threads(numCores)
    for (int v = 0; v < numVertices; v++)
    {
        if (!visited[v])
        {
            dfs(v, visited, graph);
        }
    }

    double endTime = omp_get_wtime(); // End timer

    cout << "Number of cores used: " << numCores << endl;
    cout << "Time taken: " << endTime - startTime << " seconds" << endl;
    cout << "----------------------" << endl;
}

int main()
{
    // Generate a random graph with 10,000 vertices and 50,000 edges
    int numVertices = 10000;
    int numEdges = 50000;
    vector<vector<int>> graph(numVertices);
    srand(time(0));
    for (int i = 0; i < numEdges; i++)
    {
        int u = rand() % numVertices;
        int v = rand() % numVertices;
        graph[u].push_back(v);
        graph[v].push_back(u);
    }

    // Array containing number of cores
    int numCoresArr[] = {1, 2, 3, 4, 5, 6, 7, 8};

    // Loop over different number of cores and execute parallel DFS
    for (int i = 0; i < sizeof(numCoresArr) / sizeof(numCoresArr[0]); i++)
    {
        int numCores = numCoresArr[i];
        cout << "Running parallel DFS with " << numCores << " core(s)..." << endl;
        parallelDFS(graph, numCores);
    }

    return 0;
}
```

Output

```
PS D:\Gaurav\College\4 BE\Sem 8\High Performance Computing\Lab Assignments\1> g++ bfs.cpp -o dfs -fopenmp -pthread
PS D:\Gaurav\College\4 BE\Sem 8\High Performance Computing\Lab Assignments\1> ./bfs
Running parallel BFS with 1 core(s)...
Number of cores used: 1
Time taken: 11.01 seconds
------------------------
Running parallel BFS with 2 core(s)...
Number of cores used: 2
Time taken: 0.076 seconds
------------------------
Running parallel BFS with 3 core(s)...
Number of cores used: 3
Time taken: 0.0679998 seconds
------------------------
Running parallel BFS with 4 core(s)...
Number of cores used: 4
Time taken: 0.0539999 seconds
------------------------
Running parallel BFS with 5 core(s)...
Number of cores used: 5
Time taken: 0.0469999 seconds
------------------------
Running parallel BFS with 6 core(s)...
Number of cores used: 6
Time taken: 0.043 seconds
------------------------
Running parallel BFS with 7 core(s)...
Number of cores used: 7
Time taken: 0.0419998 seconds
------------------------
Running parallel BFS with 8 core(s)...
Number of cores used: 8
Time taken: 0.0409999 seconds
------------------------
PS D:\Gaurav\College\4 BE\Sem 8\High Performance Computing\Lab Assignments\1>
```

Assigment 3

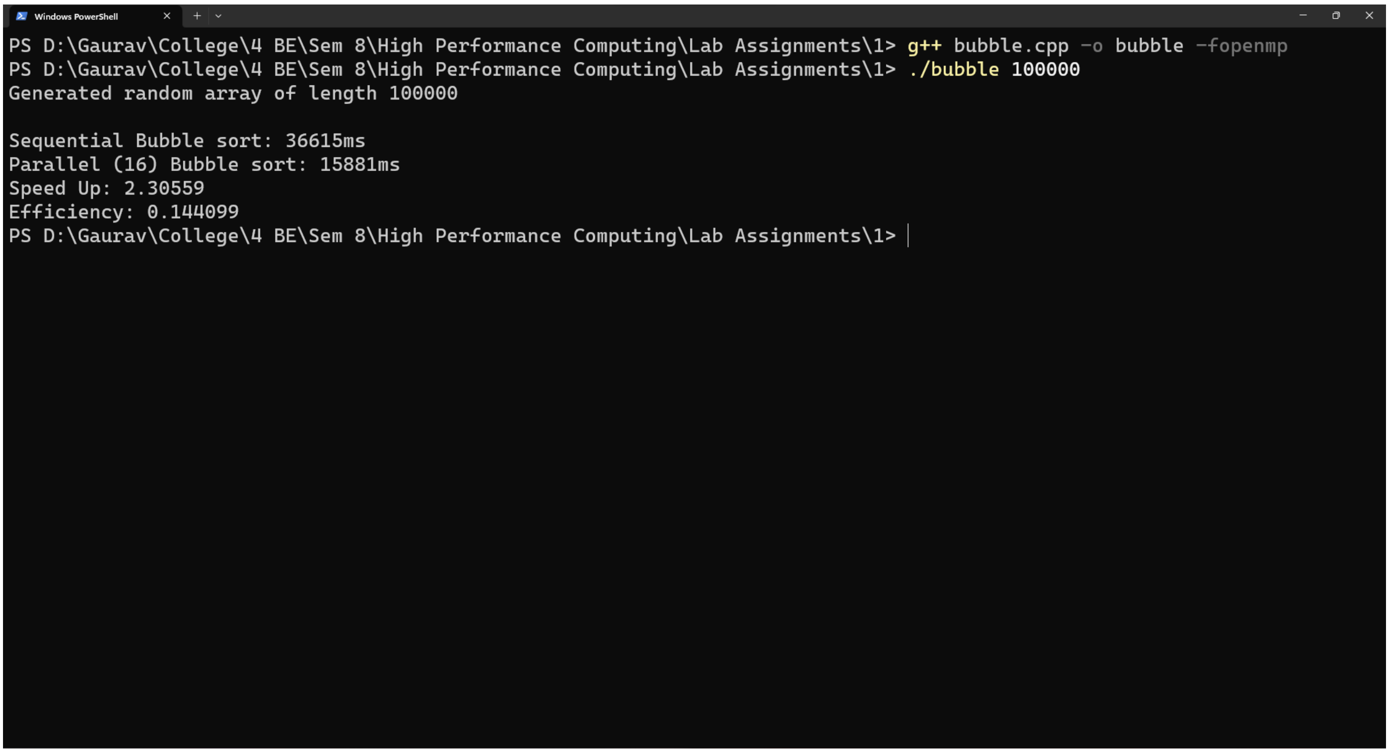
Code

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

#include <array>
#include <chrono>
#include <functional>
#include <iostream>
#include <string>
#include <vector>

using std::chrono::duration_cast;
using std::chrono::high_resolution_clock;
using std::chrono::milliseconds;
using namespace std;

void s_bubble(int *, int);
void p_bubble(int *, int);
void swap(int &, int &);

void s_bubble(int *a, int n)
{
    for (int i = 0; i < n; i++)
    {
        int first = i % 2;
        for (int j = first; j < n - 1; j += 2)
        {
            if (a[j] > a[j + 1])
            {
                swap(a[j], a[j + 1]);
            }
        }
    }
}

void p_bubble(int *a, int n)
{
    for (int i = 0; i < n; i++)
    {
        int first = i % 2;
#pragma omp parallel for shared(a, first) num_threads(16)
        for (int j = first; j < n - 1; j += 2)
        {
            if (a[j] > a[j + 1])
            {
                swap(a[j], a[j + 1]);
            }
        }
    }
}

void swap(int &a, int &b)
```

```
{
    int  test;
    test = a;
    a = b;
    b = test;
}

int bench_traverse(std::function<void()> traverse_fn)
{
    auto start = high_resolution_clock::now();
    traverse_fn();
    auto stop = high_resolution_clock::now();

    // Subtract stop and start timepoints and cast it to required unit.
    // Predefined units are nanoseconds, microseconds, milliseconds, seconds,
    // minutes, hours. Use duration_cast() function.
    auto duration = duration_cast<milliseconds>(stop - start);

    // To get the value of duration use the count() member function on the
    // duration object
    return duration.count();
}

int main(int argc, const char **argv)
{
    if (argc < 2)
    {
        cout << "Specify array length.\n";
        return 1;
    }
    int *a, n;
    n = stoi(argv[1]);
    a = new int[n];

    for (int i = 0; i < n; i++)
    {
        a[i] = rand() % n;
    }

    int *b = new int[n];
    copy(a, a + n, b);
    cout << "Generated random array of length " << n << "\n\n";

    int sequentialTime = bench_traverse([&]
                                        { s_bubble(a, n); });

    omp_set_num_threads(16);
    int parallelTime = bench_traverse([&]
                                      { s_bubble(a, n); });

    float speedUp = (float)sequentialTime / parallelTime;
    float efficiency = speedUp / 16;
```

```
    cout
        << "Sequential Bubble sort: " << sequentialTime << "ms\n";

    cout << "Parallel (16) Bubble sort: " << parallelTime << "ms\n";

    cout << "Speed Up: " << speedUp << "\n";

    cout << "Efficiency: " << efficiency << "\n";

    return 0;
}
```

Outpu



Assigment 4

Code

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <iostream>

using namespace std;

#define ARRAY_SIZE 5000

void merge(int arr[], int left[], int left_size, int right[], int right_size)
{
  int i = 0, j = 0, k = 0;
  while (i < left_size && j < right_size)
  {
    if (left[i] <= right[j])
    {
      arr[k] = left[i];
      i++;
    }
    else
    {
      arr[k] = right[j];
      j++;
    }
    k++;
  }
  while (i < left_size)
  {
    arr[k] = left[i];
    i++;
    k++;
  }
  while (j < right_size)
  {
    arr[k] = right[j];
    j++;
    k++;
  }
}

void merge_sort(int arr[], int size)
{
  if (size < 2)
  {
    return;
  }
  int mid = size / 2;
  int left[mid], right[size - mid];
  for (int i = 0; i < mid; i++)
  {
```

```
        left[i] = arr[i];
    }
    for (int i = mid; i < size; i++)
    {
        right[i - mid] = arr[i];
    }
#pragma omp parallel sections
    {
#pragma omp section
        {
            merge_sort(left, mid);
        }
#pragma omp section
        {
            merge_sort(right, size - mid);
        }
    }
    merge(arr, left, mid, right, size - mid);
}

int main()
{
    int arr[ARRAY_SIZE];
    int num_threads_array[] = {16};
    int num_threads_array_size = sizeof(num_threads_array) / sizeof(int);

    // Initialize the array with random values
    for (int i = 0; i < ARRAY_SIZE; i++)
    {
        arr[i] = rand() % ARRAY_SIZE;
    }

    // Sort the array using normal merge sort
    clock_t start_time = clock();
    merge_sort(arr, ARRAY_SIZE);
    clock_t end_time = clock();
    double normal_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;

    // Sort the array in parallel using OpenMP
    for (int i = 0; i < num_threads_array_size; i++)
    {
        int num_threads = num_threads_array[i];
        printf("Number of threads: %d\n", num_threads);
        start_time = clock();
        omp_set_num_threads(num_threads);
#pragma omp parallel
        {
#pragma omp single
            {
                merge_sort(arr, ARRAY_SIZE);
            }
        }
```

```
    end_time = clock();
    double parallel_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;

    // Print the time taken by both merge sorts
    printf("Time taken (normal merge sort): %f seconds\n", normal_time);
    printf("Time taken (parallel merge sort): %f seconds\n", parallel_time);

    float speedUp = normal_time / parallel_time;
    float efficiency = speedUp / num_threads;

    cout << "Speed Up: " << speedUp << "\n";
    cout << "Efficiency: " << efficiency << "\n";
    printf("\n");
  }

  return 0;}
```

Output:

```
PS D:\Gaurav\College\4 BE\Sem 8\High Performance Computing\Lab Assignments\1> g++ merge.cpp -o merge -fopenmp
PS D:\Gaurav\College\4 BE\Sem 8\High Performance Computing\Lab Assignments\1> ./merge
Number of threads: 16
Time taken (normal merge sort): 0.083000 seconds
Time taken (parallel merge sort): 0.070000 seconds
Speed Up: 1.18571
Efficiency: 0.0741071

PS D:\Gaurav\College\4 BE\Sem 8\High Performance Computing\Lab Assignments\1>
```