Assignment No. 1:

Design and implement Parallel Breadth-First Search and Depth First Search based on existing algorithms using OpenMP. Use a Tree or an undirected graph for BFS and DFS.

Code:

```
%%cu
#include <iostream>
#include <vector>
#include <queue>
#include <omp.h>

using namespace std;

// Graph class representing the adjacency list
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 graph
	void addEdge(int v, int w) {
	adj[v].push_back(w);
	}

	// Parallel Depth-First Search
	void parallelDFS(int startVertex) {
	vector<bool> visited(V, false);
	double startTime = omp_get_wtime();
	parallelDFSUtil(startVertex, visited);
	double endTime = omp_get_wtime();
	cout << "\nExecution Time (DFS): " << endTime - startTime << " seconds" << endl;
	}

	// Parallel DFS utility function
	void parallelDFSUtil(int v, vector<bool>& visited) {
	visited[v] = true;
	cout << v << " ";

	#pragma omp parallel for
	for (int i = 0; i < adj[v].size(); ++i) {
	int n = adj[v][i];
	if (!visited[n])
		parallelDFSUtil(n, visited);
```

```
        }
        }

        // Parallel Breadth-First Search
        void parallelBFS(int startVertex) {
        vector<bool> visited(V, false);
        queue<int> q;
        double startTime = omp_get_wtime();

        visited[startVertex] = true;
        q.push(startVertex);

        while (!q.empty()) {
        int v = q.front();
        q.pop();
        cout << v << " ";

        #pragma omp parallel for
        for (int i = 0; i < adj[v].size(); ++i) {
                int n = adj[v][i];
                if (!visited[n]) {
                visited[n] = true;
                q.push(n);
                }
        }
        }

        double endTime = omp_get_wtime();
        cout << "\nExecution Time (BFS): " << endTime - startTime << " seconds" << endl;
        }
};

int main() {
        // Create a graph
        Graph g(7);
        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 3);
        g.addEdge(1, 4);
        g.addEdge(2, 5);
        g.addEdge(2, 6);

        /*
        0 -------->1
        |       / \
        |       /   \
        |       /       \
        v       v       v
```

```
    2 ----> 3        4
    |       |
    |       |
    v       v
    5       6
    */

    cout << "Depth-First Search (DFS): ";
    g.parallelDFS(0);
    cout << endl;

    cout << "Breadth-First Search (BFS): ";
    g.parallelBFS(0);
    cout << endl;

    return 0;
}
```

Output:

Depth-First Search (DFS): 0 1 3 4 2 5 6
Execution Time (DFS): 0.000345 seconds

Breadth-First Search (BFS): 0 1 2 3 4 5 6
Execution Time (BFS): 0.000123 seconds

Assignment No. 2:

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 - Parallel Bubble Sort:

```
#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);
}
```

Output:

Sequential Bubble Sort took : 0.00957767 seconds.
Parallel Bubble Sort took : 0.00988083 seconds.

Code - Parallel Merge Sort:

```
#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 = 1000;
        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;
}
```

Output:

Time taken by sequential algorithm: 0.000135859 seconds
Time taken by parallel algorithm: 0.000123855 seconds

Assignment No. 3:

Implement Min, Max, Sum and Average operations using Parallel Reduction.

.cpp Code:

```
%%cu
/*
  Windows does not support user defined reductions.
  This program may not run on MVSC++ compilers for Windows.
  Please use Linux Environment.[You can try using "windows subsystem for linux"]
*/

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

using namespace std;
int minval(int arr[], int n){
  int minval = arr[0];
  #pragma omp parallel for reduction(min : minval)
	for(int i = 0; i < n; i++){
	if(arr[i] < minval) minval = arr[i];
	}
  return minval;
}

int maxval(int arr[], int n){
  int maxval = arr[0];
  #pragma omp parallel for reduction(max : maxval)
	for(int i = 0; i < n; i++){
	if(arr[i] > maxval) maxval = arr[i];
	}
  return maxval;
}

int sum(int arr[], int n){
  int sum = 0;
  #pragma omp parallel for reduction(+ : sum)
	for(int i = 0; i < n; i++){
	sum += arr[i];
	}
  return sum;
}

int average(int arr[], int n){
```

```
  return (double)sum(arr, n) / n;
}

int main(){
  int n = 5;
  int arr[] = {1,2,3,4,5};
  cout << "The minimum value is: " << minval(arr, n) << '\n';
  cout << "The maximum value is: " << maxval(arr, n) << '\n';
  cout << "The summation is: " << sum(arr, n) << '\n';
  cout << "The average is: " << average(arr, n) << '\n';
  return 0;
}
```

Output:

The minimum value is: 1
The maximum value is: 5
The summation is: 15
The average is: 3

Assignment No. 4:

Write a CUDA Program for :
1. Addition of two large vectors
2. Matrix Multiplication using CUDA C

Code - Addition of Two large Vectors:

```
%%cu
#include <iostream>
using namespace std;

__global__ void add(int* A, int* B, int* C, int size) {
        int tid = blockIdx.x * blockDim.x + threadIdx.x;

        if (tid < size) {
        C[tid] = A[tid] + B[tid];
        }
}


void initialize(int* vector, int size) {
        for (int i = 0; i < size; i++) {
        vector[i] = rand() % 10;
        }
}

void print(int* vector, int size) {
        for (int i = 0; i < size; i++) {
        cout << vector[i] << " ";
        }
        cout << endl;
}

int main() {
        int N = 4;
        int* A, * B, * C;

        int vectorSize = N;
        size_t vectorBytes = vectorSize * sizeof(int);

        A = new int[vectorSize];
        B = new int[vectorSize];
        C = new int[vectorSize];

        initialize(A, vectorSize);
        initialize(B, vectorSize);
```

```
    cout << "Vector A: ";
    print(A, N);
    cout << "Vector B: ";
    print(B, N);

    int* X, * Y, * Z;
    cudaMalloc(&X, vectorBytes);
    cudaMalloc(&Y, vectorBytes);
    cudaMalloc(&Z, vectorBytes);

    cudaMemcpy(X, A, vectorBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(Y, B, vectorBytes, cudaMemcpyHostToDevice);

    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;

    add<<<blocksPerGrid, threadsPerBlock>>>(X, Y, Z, N);

    cudaMemcpy(C, Z, vectorBytes, cudaMemcpyDeviceToHost);

    cout << "Addition: ";
    print(C, N);

    delete[] A;
    delete[] B;
    delete[] C;

    cudaFree(X);
    cudaFree(Y);
    cudaFree(Z);

    return 0;
}
```

Output:

Vector A: 3 6 7 5
Vector B: 3 5 6 2
Addition: 6 11 13 7

Code - Matrix Multiplication using CUDA C:

%%cu

```
#include <iostream>
using namespace std;


// CUDA code to multiply matrices
__global__ void multiply(int* A, int* B, int* C, int size) {
	// Uses thread indices and block indices to compute each element
	int row = blockIdx.y * blockDim.y + threadIdx.y;
	int col = blockIdx.x * blockDim.x + threadIdx.x;

	if (row < size && col < size) {
	int sum = 0;
	for (int i = 0; i < size; i++) {
	sum += A[row * size + i] * B[i * size + col];
	}
	C[row * size + col] = sum;
	}
}


void initialize(int* matrix, int size) {
	for (int i = 0; i < size * size; i++) {
	matrix[i] = rand() % 10;
	}
}


void print(int* matrix, int size) {
	for (int row = 0; row < size; row++) {
	for (int col = 0; col < size; col++) {
	cout << matrix[row * size + col] << " ";
	}
	cout << '\n';
	}
	cout << '\n';
}


int main() {
	int* A, * B, * C;

	int N = 2;
	int blockSize =  16;

	int matrixSize = N * N;
	size_t matrixBytes = matrixSize * sizeof(int);

	A = new int[matrixSize];
```

```
B = new int[matrixSize];
C = new int[matrixSize];

initialize(A, N);
initialize(B, N);
cout << "Matrix A: \n";
print(A, N);

cout << "Matrix B: \n";
print(B, N);


int* X, * Y, * Z;
// Allocate space
cudaMalloc(&X, matrixBytes);
cudaMalloc(&Y, matrixBytes);
cudaMalloc(&Z, matrixBytes);

// Copy values from A to X
cudaMemcpy(X, A, matrixBytes, cudaMemcpyHostToDevice);

// Copy values from A to X and B to Y
cudaMemcpy(Y, B, matrixBytes, cudaMemcpyHostToDevice);

// Threads per CTA dimension
int THREADS = 2;

// Blocks per grid dimension (assumes THREADS divides N evenly)
int BLOCKS = N / THREADS;

// Use dim3 structs for block  and grid dimensions
dim3 threads(THREADS, THREADS);
dim3 blocks(BLOCKS, BLOCKS);

// Launch kernel
multiply<<<blocks, threads>>>(X, Y, Z, N);

cudaMemcpy(C, Z, matrixBytes, cudaMemcpyDeviceToHost);
cout << "Multiplication of matrix A and B: \n";
print(C, N);

delete[] A;
delete[] B;
delete[] C;

cudaFree(X);
cudaFree(Y);
cudaFree(Z);
```

```
    return 0;
}
```

Output:

Matrix A:
3 6
7 5

Matrix B:
3 5
6 2

Multiplication of matrix A and B:
45 27
51 45