Practical 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 .

```
%%writefile main.cpp

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

using namespace std;

// Node structure representing a tree node
struct TreeNode {
    int data;
    vector<TreeNode*> children;

    TreeNode(int val) : data(val) {}
};

// Tree class representing the tree structure
class Tree {
    TreeNode* root;

public:
    Tree(int val) {
        root = new TreeNode(val);
    }

    // Add a child to a parent node
    void addChild(TreeNode* parent, int val) {
        TreeNode* newNode = new TreeNode(val);
        parent->children.push_back(newNode);
    }

    // Method to get the root node
    TreeNode* getRoot() {
        return root;
    }

    // Parallel Depth-First Search
    void parallelDFS(TreeNode* node) {
        cout << node->data << " ";

        #pragma omp parallel for
        for (size_t i = 0; i < node->children.size(); ++i) {
            parallelDFS(node->children[i]);
        }
    }

    // Parallel Breadth-First Search
    void parallelBFS() {
        queue<TreeNode*> q;
        q.push(root);

        while (!q.empty()) {
            TreeNode* current = q.front();
```

```cpp
            q.pop();
            cout << current->data << " ";

            #pragma omp parallel for
            for (size_t i = 0; i < current->children.size(); ++i) {
                q.push(current->children[i]);
            }
        }
    }
};

int main() {
    // Create a tree
    Tree tree(1);
    TreeNode* root = tree.getRoot();
    tree.addChild(root, 2);
    tree.addChild(root, 3);
    tree.addChild(root, 4);

    TreeNode* node2 = root->children[0];
    tree.addChild(node2, 5);
    tree.addChild(node2, 6);

    TreeNode* node4 = root->children[2];
    tree.addChild(node4, 7);
    tree.addChild(node4, 8);

    /*
               1
             / | \
            2  3  4
          / \    / \
         5   6  7   8
    */

    cout << "Depth-First Search (DFS): ";
    tree.parallelDFS(root);
    cout << endl;

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

    return 0;
}
```

```
Writing main.cpp
```

OUTPUT

```
%%script bash
g++ main.cpp -std=c++11
./a.out
```

```
Depth-First Search (DFS): 1 2 5 6 3 4 7 8
Breadth-First Search (BFS): 1 2 3 4 5 6 7 8
```

Practical 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.

Bubble Sort

```
%%writefile main.cpp

#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);
}
```

```
Overwriting main.cpp
```

OUTPUT

```
%%script bash
g++ main.cpp -std=c++11 -fopenmp -lgomp
./a.out
```

```
Sequential Bubble Sort took : 1.646e-06 seconds.
1 2 3 4 5 6 7 8 9 10
Parallel Bubble Sort took : 2.5e-06 seconds.
1 2 3 4 5 6 7 8 9 10
```

Merge Sort

```
%%writefile main.cpp

#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);
    }
}

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

```
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";
    printArray(arr, 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\n";
    printArray(arr, n);

    return 0;
}
```

```
Overwriting main.cpp
```

OUTPUT

```
%%script bash
g++ main.cpp -std=c++11 -fopenmp -lgomp
./a.out
```

```
Time taken by sequential algorithm: 1.528e-06 seconds
1 2 3 4 5 6 7 8 9 10
Time taken by parallel algorithm: 0.00014441 seconds
1 2 3 4 5 6 7 8 9 10
```

Practical 3

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

```
%%writefile main.cpp

#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;
}
```

```
Overwriting main.cpp
```

OUTPUT

```
%%script bash
g++ main.cpp -std=c++11
./a.out
```

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

Practical 4

Write a CUDA Program for :

1. Addition of two large vectors
2. Matrix Multiplication using CUDA C

```
!nvcc --version
```

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2023 NVIDIA Corporation
Built on Tue_Aug_15_22:02:13_PDT_2023
Cuda compilation tools, release 12.2, V12.2.140
Build cuda_12.2.r12.2/compiler.33191640_0
```

```
!pip install git+https://github.com/afnan47/cuda.git
%load_ext nvcc_plugin
```

```
Collecting git+https://github.com/afnan47/cuda.git
  Cloning https://github.com/afnan47/cuda.git to /tmp/pip-req-build-u12yi4kr
  Running command git clone --filter=blob:none --quiet https://github.com/afnan47/cuda.git /tmp/pip
  Resolved https://github.com/afnan47/cuda.git to commit aac710a35f52bb78ab34d2e52517237941399eff
  Preparing metadata (setup.py) ... done
directory /content/src already exists
Out bin /content/result.out
```

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;
}
```

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

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;
}
```

```
Matrix A:
3 6
7 5

Matrix B:
3 5
6 2

Multiplication of matrix A and B:
45 27
51 45
```