HPC Practicals

Assignment 1

Q. 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.

To explain and implement **parallel Breadth-First Search (BFS)** and **Depth-First Search (DFS)** using **OpenMP**, let’s break it down into the **theory**, followed by **code implementation** using a **graph (adjacency list)** structure. We'll go with an **undirected graph** to generalize the application.

Theory of Design

Graph Representation

We'll use an **adjacency list** to represent an undirected graph. Each node will keep a list of its neighbors.

Breadth-First Search (BFS)

Sequential BFS

- Uses a **queue** to explore nodes level by level.
- Start at a source node, explore its neighbors, then the neighbors' neighbors, etc.

Parallel BFS (with OpenMP)

- At each **level**, all nodes in the current frontier can be explored **in parallel**.
- Use OpenMP to parallelize the loop that iterates through current-level nodes.

BFS is more naturally parallelizable because each level can be processed in parallel.

Depth-First Search (DFS)

Sequential DFS

- Uses a **stack** (or recursion) to go deep along one branch before backtracking.

Parallel DFS (with OpenMP)

- DFS is inherently recursive and not easy to parallelize because it explores deep paths sequentially.
- We can spawn threads from the root for each unvisited neighbor (i.e., parallelize top-level recursion).

Parallel DFS usually leverages **task-based parallelism**.

Code

```
#include <iostream>
#include <vector>
#include <queue>
#include <stack>
#include <omp.h>
using namespace std;

class Graph {
public:
    int V;
    vector<vector<int>> adj;

    Graph(int V) {
        this->V = V;
        adj.resize(V);
    }

    void addEdge(int u, int v) {
        adj[u].push_back(v);
        adj[v].push_back(u);  // Undirected graph
    }

    void parallelBFS(int start) {
        vector<bool> visited(V, false);
        queue<int> q;

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

        while (!q.empty()) {
            int qSize = q.size();
            vector<int> frontier;

            // Collect current level nodes
            for (int i = 0; i < qSize; ++i) {
                int node = q.front(); q.pop();
                cout << node << " ";

                frontier.push_back(node);
            }

            // Explore neighbors in parallel
            #pragma omp parallel for schedule(dynamic)
            for (int i = 0; i < frontier.size(); ++i) {
                int u = frontier[i];
                for (int v : adj[u]) {
                    if (!visited[v]) {
                        #pragma omp critical
                        {
                            if (!visited[v]) {
                                visited[v] = true;
                                q.push(v);
                            }
                        }
                    }
                }
            }
        }
        cout << endl;
    }
```

```
    void parallelDFSUtil(int node, vector<bool>& visited) {
        visited[node] = true;
        cout << node << " ";

        #pragma omp parallel for schedule(dynamic)
        for (int i = 0; i < adj[node].size(); ++i) {
            int neighbor = adj[node][i];
            if (!visited[neighbor]) {
                #pragma omp task firstprivate(neighbor)
                {
                    parallelDFSUtil(neighbor, visited);
                }
            }
        }
        #pragma omp taskwait
    }

    void parallelDFS(int start) {
        vector<bool> visited(V, false);
        #pragma omp parallel
        {
            #pragma omp single
            {
                parallelDFSUtil(start, visited);
            }
        }
        cout << endl;
    }
};

int main() {
    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);

    cout << "Parallel BFS starting from node 0:\n";
    g.parallelBFS(0);

    cout << "Parallel DFS starting from node 0:\n";
    g.parallelDFS(0);

    return 0;
}
```

Header Files

```
#include <iostream>     // For input/output
#include <vector>       // To use dynamic arrays (adjacency list)
#include <queue>        // For BFS (FIFO structure)
#include <stack>        // (Not used here but typical for DFS)
#include <omp.h>        // OpenMP for parallel programming
```

Graph Class Definition

```
class Graph {
public:
    int V;                          // Number of vertices
    vector<vector<int>> adj;        // Adjacency list for graph
```

Constructor and Edge Addition

```
    Graph(int V) {
        this->V = V;
        adj.resize(V);              // Create a list for each vertex
    }

    void addEdge(int u, int v) {
        adj[u].push_back(v);
        adj[v].push_back(u);        // Because it's an undirected graph
    }
```

Parallel Breadth-First Search (BFS)

```
 void parallelBFS(int start) {
        vector<bool> visited(V, false);  // Keep track of visited nodes
        queue<int> q;

        visited[start] = true;      // Mark the start node as visited
        q.push(start);              // Start BFS from this node
```

While queue isn't empty

```
        while (!q.empty()) {
            int qSize = q.size();           // Number of nodes at current level
            vector<int> frontier;
            // Step 1: Extract current level
            for (int i = 0; i < qSize; ++i) {
                int node = q.front(); q.pop();
                cout << node << " ";        // Process the node
                frontier.push_back(node);  // Save for parallel exploration
            }
```

Step 2: Explore neighbors in parallel

```
            #pragma omp parallel for schedule(dynamic)
            for (int i = 0; i < frontier.size(); ++i) {
                int u = frontier[i];
                for (int v : adj[u]) {
                    if (!visited[v]) {
                        #pragma omp critical
                        {
                            if (!visited[v]) {
                                visited[v] = true;
                                q.push(v);
                            }
                        }
                    }
                }
            }
        }
        cout << endl;
    }
```

What's going on:

- **Frontier**: Nodes in the current BFS level.
- **Parallel for**: Multiple threads explore neighbors of current-level nodes.
- **Critical section**: Protects concurrent writes to `visited` and `queue` to avoid race conditions.

Recursive DFS with OpenMP Tasks

```
void parallelDFSUtil(int node, vector<bool>& visited) {
    visited[node] = true;
    cout << node << " ";
```

Parallel for-loop over neighbors

```
    #pragma omp parallel for schedule(dynamic)
    for (int i = 0; i < adj[node].size(); ++i) {
        int neighbor = adj[node][i];
        if (!visited[neighbor]) {
            #pragma omp task firstprivate(neighbor)
            {
                parallelDFSUtil(neighbor, visited);
            }
        }
    }
    #pragma omp taskwait   // Wait for all tasks to finish
}
```

What's happening:

- DFS is **recursive**, and we use `#pragma omp task` to run each neighbor's DFS in parallel.
- `firstprivate(neighbor)`: Ensures each task gets its own copy of `neighbor`.
- `taskwait`: Makes sure all recursive branches complete before returning.

DFS Entry Point

```
void parallelDFS(int start) {
    vector<bool> visited(V, false);
    #pragma omp parallel
    {
        #pragma omp single
        {
            parallelDFSUtil(start, visited);
        }
    }
    cout << endl;
}
```

What this does:

- Starts a **single root task** to kick off DFS.
- The actual parallelism happens in the recursive utility function using **tasks**.

Main Function: Setup and Run

```
int main() {
    Graph g(7);              // Create a graph with 7 nodes

    // Add edges (undirected)
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 3);
    g.addEdge(1, 4);
    g.addEdge(2, 5);
    g.addEdge(2, 6);

    cout << "Parallel BFS starting from node 0:\n";
    g.parallelBFS(0);

    cout << "Parallel DFS starting from node 0:\n";
    g.parallelDFS(0);

    return 0;
}
```

Summary of Key OpenMP Features Used

Directive	Purpose
`#pragma omp parallel for`	Parallelizes loops over independent data (e.g., frontier nodes)
`#pragma omp critical`	Protects critical section from concurrent access
`#pragma omp task`	Spawns a new task (useful for recursive DFS)
`#pragma omp single`	Ensures a single thread runs the initial DFS call
`#pragma omp taskwait`	Waits for all tasks launched so far to complete

Assignment 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.

Below is an explanation of the theory behind parallelizing two common sorting algorithms—Bubble Sort (using the odd–even transposition method) and Merge Sort—and sample C++ code that implements both sequential and parallel versions using OpenMP. The code also measures performance to compare the sequential and parallel implementations.

Theory

Bubble Sort and Its Parallel Variant

Sequential Bubble Sort:
Bubble sort repeatedly compares adjacent pairs in an array and swaps them if they are out of order. It performs this pass over the array repeatedly until no swaps are needed. Although simple, it has a worst-case time complexity of $O(n^2)$.

Parallel Bubble Sort (Odd–Even Transposition Sort):
The odd–even transposition sort is a variant of bubble sort that is more amenable to parallelization. It divides the sorting pass into two phases:

- **Odd Phase:** Compare and swap elements at positions (1, 2), (3, 4), …
- **Even Phase:** Compare and swap elements at positions (0, 1), (2, 3), …

Within each phase, comparisons are independent and can be performed concurrently. OpenMP's parallel loops can be used to execute the comparisons in parallel, while a shared flag (with a reduction) helps determine if another pass is needed.

Merge Sort and Its Parallel Variant

Sequential Merge Sort:
Merge sort is a classic divide-and-conquer algorithm. It recursively divides the array into halves until each subarray has one element, then merges the sorted subarrays back together. Its time complexity is $O(n \log n)$.

Parallel Merge Sort:
Merge sort is naturally suited for parallelization since its recursive division of the array can be performed concurrently. By using OpenMP tasks, the two halves of the array can be sorted in parallel. To prevent too many tasks from being spawned (which might lead to overhead), a depth limit is often introduced. After reaching a certain recursion depth, the sequential merge sort is used.

Bubble Sort

```cpp
#include <iostream>
#include <omp.h>
using namespace std;

void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

void bubbleSortSeq(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        bool swapped = false;
        for (int j = 0; j < n - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }
        if (!swapped)
            break;
    }
}

void bubbleSortParallel(int arr[], int n) {
    // Perform n iterations of odd-even phases
    for (int i = 0; i < n; i++) {
        // Alternate starting index: 0 for even phase, 1 for odd phase
        int start = i % 2;
        #pragma omp parallel for
        for (int j = start; j < n - 1; j += 2) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
   }}}}

int main() {
    int n;
    cout << "Enter number of elements: ";
    cin >> n;

    int *seq = new int[n];
    int *par = new int[n];

    cout << "Enter elements: ";
    for (int i = 0; i < n; i++) {
        cin >> seq[i];
        par[i] = seq[i]; // Copy data for parallel sort
    }

    double t1 = omp_get_wtime();
    bubbleSortSeq(seq, n);
    double t2 = omp_get_wtime();
    cout << "Sequential Sorted: ";
    for (int i = 0; i < n; i++)
        cout << seq[i] << " ";
    cout << "\nTime taken (sequential): " << t2 - t1 << " seconds\n";

    t1 = omp_get_wtime();
    bubbleSortParallel(par, n);
    t2 = omp_get_wtime();
    cout << "Parallel Sorted: ";
```

```
    for (int i = 0; i < n; i++)
        cout << par[i] << " ";
    cout << "\nTime taken (parallel): " << t2 - t1 << " seconds\n";

    delete[] seq;
    delete[] par;
    return 0;
}
```

Explanation

1. **Helper Function (`swap`):**
   A simple function to swap two integers.

2. **Sequential Bubble Sort (`bubbleSortSeq`):**
   - Iterates over the array, comparing and swapping adjacent elements if needed.
   - Uses a `swapped` flag to exit early if no swaps occur during an iteration.

3. **Parallel Bubble Sort (`bubbleSortParallel`):**
   - Uses the odd–even transposition method.
   - The starting index for the inner loop alternates between 0 and 1 on each pass.
   - The inner loop is parallelized using OpenMP's `#pragma omp parallel for` directive, so multiple comparisons/swaps occur simultaneously.

4. **`main` Function:**
   - Reads the number of elements and the elements themselves.
   - Duplicates the array to ensure both sorts work on identical data.
   - Measures execution time for both sequential and parallel sorts using `omp_get_wtime()`.

Merge Sort

```cpp
#include <iostream>
#include <omp.h>
using namespace std;

// Merge two sorted subarrays a[left..mid] and a[mid+1..right]
void merge(int a[], int left, int mid, int right) {
    int n = right - left + 1;
    int* temp = new int[n];
    int i = left, j = mid + 1, k = 0;

    while (i <= mid && j <= right) {
        if (a[i] < a[j])
            temp[k++] = a[i++];
        else
            temp[k++] = a[j++];
    }
    while (i <= mid)
        temp[k++] = a[i++];
    while (j <= right)
        temp[k++] = a[j++];

    // Copy back to original array
    for (i = left, k = 0; i <= right; ++i, ++k)
        a[i] = temp[k];

    delete[] temp;
}

// Sequential merge sort
void mergeSortSeq(int a[], int left, int right) {
    if (left < right) {
        int mid = (left + right) / 2;
        mergeSortSeq(a, left, mid);
        mergeSortSeq(a, mid + 1, right);
        merge(a, left, mid, right);
    }
}

// Parallel merge sort using OpenMP sections
void mergeSortPar(int a[], int left, int right) {
    if (left < right) {
        int mid = (left + right) / 2;
        #pragma omp parallel sections
        {
            #pragma omp section
            mergeSortPar(a, left, mid);
            #pragma omp section
            mergeSortPar(a, mid + 1, right);
        }
        merge(a, left, mid, right);
    }
}

int main() {
    int n;
    cout << "Enter number of elements: ";
    cin >> n;

    int* seq = new int[n];
    int* par = new int[n];
```

```
    cout << "Enter elements:\n";
    for (int i = 0; i < n; i++) {
        cin >> seq[i];
        par[i] = seq[i]; // Copy data for parallel sort
    }

    // Sequential merge sort
    double start = omp_get_wtime();
    mergeSortSeq(seq, 0, n - 1);
    double end = omp_get_wtime();
    cout << "\nSequential Time: " << end - start << " seconds\n";

    // Parallel merge sort
    start = omp_get_wtime();
    #pragma omp parallel
    {
        #pragma omp single
        mergeSortPar(par, 0, n - 1);
    }
    end = omp_get_wtime();
    cout << "Parallel Time: " << end - start << " seconds\n";

    cout << "\nSorted Array:\n";
    for (int i = 0; i < n; i++)
        cout << par[i] << " ";
    cout << endl;

    delete[] seq;
    delete[] par;

    return 0;
}
```

Explanation

- **Merge Function:**
  Combines two sorted subarrays into one sorted subarray. A temporary array is allocated dynamically to store the merged result, which is then copied back into the original array.
- **Sequential Merge Sort (`mergeSortSeq`):**
  Recursively divides the array until subarrays have one element, then merges them back together.
- **Parallel Merge Sort (`mergeSortPar`):**
  Uses OpenMP's `#pragma omp parallel sections` to sort the left and right halves concurrently. The merge step is performed after the parallel sections.
- **`main` Function:**
  Reads the input array, creates two copies (one for sequential sorting and one for parallel sorting), measures the execution time for each sort using `omp_get_wtime()`, and prints the sorted array along with the time taken.

Assignment 3

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

```cpp
#include <iostream>
#include <vector>
#include <omp.h>
#include <climits>
#include <algorithm>  // for std::min and std::max

using namespace std;

void compute_reductions(const vector<int>& arr) {
    int min_value = INT_MAX, max_value = INT_MIN, sum = 0;

    #pragma omp parallel for reduction(min:min_value) reduction(max:max_value)
reduction(+:sum)
    for (size_t i = 0; i < arr.size(); ++i) {
        min_value = min(min_value, arr[i]);
        max_value = max(max_value, arr[i]);
        sum += arr[i];
    }

    double average = static_cast<double>(sum) / arr.size();

    cout << "Minimum: " << min_value << "\n"
         << "Maximum: " << max_value << "\n"
         << "Sum: " << sum << "\n"
         << "Average: " << average << "\n";
}

int main() {
    vector<int> arr = {5, 2, 9, 1, 7, 6, 8, 3, 4};

    cout << "Input Array: ";
    for (int val : arr) {
        cout << val << " ";
    }
    cout << "\n\n";

    compute_reductions(arr);

    return 0;
}
```

Explanation

- **Reduction Clauses:**
  The `#pragma omp parallel for` directive splits the loop among threads. The clauses `reduction(min:min_value)`, `reduction(max:max_value)`, and `reduction(+:sum)` create a private copy of each variable for every thread and then combine them after the loop using the specified operations.
- **Using std::min and std::max:**
  Instead of using `if` conditions to update the minimum and maximum values, the code uses `std::min` and `std::max` to make the code cleaner.
- **Computing Average:**
  After obtaining the sum, the average is computed as a double by dividing the sum by the number of elements.

Parallel Reduction – Theory

Parallel Reduction is a technique used in parallel computing to aggregate data (e.g., finding sum, min, max, etc.) from a large array by breaking the task into smaller parts that run concurrently across multiple processors or threads.

How It Works

Instead of performing the operation sequentially on every element (which takes O(n) time), **parallel reduction** performs pairwise operations in **logarithmic time (O(log n))** using multiple threads.

Example Workflow (for Sum):

1. **Start with an array of values**:
   `[a0, a1, a2, a3, ..., an]`
2. **First pass**:
   Each thread adds a pair of elements:
   `[a0 + a1, a2 + a3, ..., an-1 + an]`
3. **Next pass**:
   Threads add results from previous pass:
   `[(a0+a1) + (a2+a3), ...]`
4. Continue until one final result remains.

Operations Explained

1. Sum

Add all elements together using the parallel reduction tree described above.

2. Min

Each thread compares two elements and passes the smaller one. Repeat until the global minimum is found.

3. Max

Same as Min, but pass the **larger** of each pair.

4. Average

First compute the **Sum** using parallel reduction, then divide by total number of elements (can be done in a single thread or as a final parallel step).

Advantages

- Efficient use of CPU/GPU cores
- Reduces time complexity from **O(n)** to **O(log n)**
- Scales well with large datasets

Applications

- Data analytics (mean, median, range)
- Machine learning preprocessing
- Physics and simulation calculations

Assignment 4

Write a CUDA Program for:

1. Vector Addition using CUDA

Theory

CUDA allows you to offload parallel computations onto NVIDIA GPUs. For vector addition:

- **Host Code:** Allocate memory on the host (CPU) and initialize two input vectors.
- **Device Memory Allocation:** Allocate memory on the GPU (device) for the input vectors and the result.
- **Memory Transfer:** Copy the input data from host memory to device memory.
- **Kernel Launch:** Write a CUDA kernel where each thread computes one element of the output vector by adding the corresponding elements from the two input vectors.
- **Result Transfer:** After the kernel finishes, copy the result from the device back to the host.
- **Cleanup:** Free allocated GPU memory.

```
#include <iostream>
#include <cuda.h>
using namespace std;

// CUDA Kernel for vector addition
__global__ void vectorAdd(const float* A, const float* B, float* C, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) {
        C[i] = A[i] + B[i];
    }
}

int main() {
    int n = 1 << 20;  // 1M elements
    size_t size = n * sizeof(float);

    // Allocate host memory
    float* h_A = new float[n];
    float* h_B = new float[n];
    float* h_C = new float[n];

    // Initialize input vectors
    for (int i = 0; i < n; i++) {
        h_A[i] = static_cast<float>(i);
        h_B[i] = static_cast<float>(2 * i);
    }

    // Allocate device memory
    float *d_A, *d_B, *d_C;
    cudaMalloc(&d_A, size);
    cudaMalloc(&d_B, size);
    cudaMalloc(&d_C, size);

    // Copy host data to device
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
```

```
    // Launch kernel
    int threadsPerBlock = 256;
    int blocksPerGrid = (n + threadsPerBlock - 1) / threadsPerBlock;
    vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, n);

    // Copy result back to host
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Verify a few results
    cout << "Sample results:" << endl;
    for (int i = 0; i < 5; i++) {
        cout << h_A[i] << " + " << h_B[i] << " = " << h_C[i] << endl;
    }

    // Clean up
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    delete[] h_A;
    delete[] h_B;
    delete[] h_C;

    return 0;
}
```

2. Matrix Multiplication using CUDA

Theory

For matrix multiplication (computing C = A * B):

- **Host Setup:** Prepare two input matrices (A and B) and an output matrix (C) on the host.
- **Device Memory:** Allocate memory on the device for these matrices.
- **Kernel Design:** Each thread calculates one element of the output matrix C. The thread uses its row and column indices to loop over the shared dimension (number of columns in A or rows in B) to accumulate the dot product.
- **Memory Transfer:** Copy matrices from the host to the device before launching the kernel, and then copy the resulting matrix back.
- **Optimization Note:** While this example uses a straightforward (naive) implementation, real applications often use shared memory tiling for better performance.

```
#include <iostream>
#include <cuda.h>
using namespace std;

#define N 512  // Assuming square matrices of size N x N

// CUDA Kernel for Matrix Multiplication
__global__ void matrixMul(const float* A, const float* B, float* C, int n) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < n && col < n) {
        float sum = 0.0f;
        for (int k = 0; k < n; k++) {
            sum += A[row * n + k] * B[k * n + col];
        }
        C[row * n + col] = sum;
    }
}

int main() {
    int size = N * N * sizeof(float);

    // Allocate host memory for matrices
    float *h_A = new float[N * N];
    float *h_B = new float[N * N];
    float *h_C = new float[N * N];

    // Initialize matrices A and B with some values
    for (int i = 0; i < N * N; i++) {
        h_A[i] = 1.0f;    // You can change these values as needed
        h_B[i] = 2.0f;
    }

    // Allocate device memory
    float *d_A, *d_B, *d_C;
    cudaMalloc(&d_A, size);
    cudaMalloc(&d_B, size);
    cudaMalloc(&d_C, size);
```

```
    // Copy matrices from host to device
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Define block and grid dimensions
    dim3 threadsPerBlock(16, 16);
    dim3 blocksPerGrid((N + threadsPerBlock.x - 1) / threadsPerBlock.x,
                       (N + threadsPerBlock.y - 1) / threadsPerBlock.y);

    // Launch the matrix multiplication kernel
    matrixMul<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

    // Copy the result matrix from device to host
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Display a few elements from the result matrix
    cout << "Sample results from matrix C:" << endl;
    for (int i = 0; i < 5; i++) {
        cout << h_C[i] << " ";  // Print first 5 elements of row 0
    }
    cout << endl;

    // Clean up memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    delete[] h_A;
    delete[] h_B;
    delete[] h_C;

    return 0;
}
```

Summary

- **CUDA Vector Addition:**
  The kernel `vectorAdd` computes the sum for each element. The host code allocates memory, copies data to the GPU, launches the kernel, and then copies the result back.
- **CUDA Matrix Multiplication:**
  The kernel `matrixMul` calculates each element of the output matrix using the dot product of a row from A and a column from B. The host code follows a similar process as in vector addition, setting up device memory, launching the kernel with a 2D grid, and copying the result back.

Compile these examples with a CUDA-capable compiler (like NVCC):

```
nvcc -O2 vector_addition.cu -o vector_addition
nvcc -O2 matrix_mul.cu -o matrix_mul
```