**HPC1**

#include <iostream>

#include <vector>

#include <queue>

#include <stack>

#include <omp.h>

using namespace std;

// Define the graph structure

class Graph {

public:

    vector<vector<int>> adj\_list;

    Graph(int num\_nodes) {

        adj\_list.resize(num\_nodes);

    }

    void add\_edge(int u, int v) {

        adj\_list[u].push\_back(v);

        adj\_list[v].push\_back(u);

    }

};

// Breadth First Search (BFS) algorithm

void bfs(Graph& graph, int start\_node, vector<bool>& visited) {

    queue<int> q;

    visited[start\_node] = true;

    q.push(start\_node);

    while (!q.empty()) {

        int u = q.front();

        q.pop();

        cout << u << " ";

        #pragma omp parallel for

        for (int i = 0; i < graph.adj\_list[u].size(); i++) {

            int v = graph.adj\_list[u][i];

            if (!visited[v]) {

                visited[v] = true;

                q.push(v);

            }

        }

    }

}

// Depth First Search (DFS) algorithm

void dfs(Graph& graph, int start\_node, vector<bool>& visited) {

    stack<int> s;

    visited[start\_node] = true;

    s.push(start\_node);

    while (!s.empty()) {

        int u = s.top();

        s.pop();

        cout << u << " ";

        #pragma omp parallel for

        for (int i = 0; i < graph.adj\_list[u].size(); i++) {

            int v = graph.adj\_list[u][i];

            if (!visited[v]) {

                visited[v] = true;

                s.push(v);

            }

        }

    }

}

int main() {

    int num\_nodes = 7;

    Graph graph(num\_nodes);

    // Add edges to the graph

    graph.add\_edge(0, 1);

    graph.add\_edge(0, 2);

    graph.add\_edge(1, 3);

    graph.add\_edge(1, 4);

    graph.add\_edge(2, 5);

    graph.add\_edge(2, 6);

    vector<bool> visited(num\_nodes, false);

    // Run BFS algorithm in parallel

    #pragma omp parallel

    {

        #pragma omp single nowait

        bfs(graph, 0, visited);

    }

    cout << endl;

    visited.assign(num\_nodes, false);

    // Run DFS algorithm in parallel

    #pragma omp parallel

    {

        #pragma omp single nowait

        dfs(graph, 0, visited);

    }

    return 0;

}

Output:-  
  
0 1 2 3 4 5 6

0 2 6 5 1 4 3

**HPC2**

#include <iostream>

#include <vector>

#include <algorithm>

#include <omp.h>

void parallel\_bubble\_sort(std::vector<int>& arr) {

    bool swapped = true;

    int n = arr.size();

    #pragma omp parallel shared(arr, n, swapped)

    {

        while (swapped) {

            swapped = false;

            #pragma omp for nowait

            for (int i = 1; i < n; i++) {

                if (arr[i - 1] > arr[i]) {

                    std::swap(arr[i - 1], arr[i]);

                    swapped = true;

                }

            }

        }

    }

}

void parallel\_merge\_sort(std::vector<int>& arr) {

    if (arr.size() > 1) {

        int mid = arr.size() / 2;

        std::vector<int> left(arr.begin(), arr.begin() + mid);

        std::vector<int> right(arr.begin() + mid, arr.end());

        #pragma omp parallel sections

        {

            #pragma omp section

            parallel\_merge\_sort(left);

            #pragma omp section

            parallel\_merge\_sort(right);

        }

        std::merge(left.begin(), left.end(),

        right.begin(), right.end(), arr.begin());

    }

}

int main() {

    std::vector<int> arr = {5, 2, 9, 1, 7, 6, 8, 3, 4};

    std::cout << "Original array: ";

    for (auto i : arr) {

        std::cout << i << " ";

    }

    std::cout << std::endl;

    // Parallel bubble sort

    parallel\_bubble\_sort(arr);

    std::cout << "After parallel bubble sort: ";

    for (auto i : arr) {

        std::cout << i << " ";

    }

    std::cout << std::endl;

    // Parallel merge sort

    parallel\_merge\_sort(arr);

    std::cout << "After parallel merge sort: ";

    for (auto i : arr) {

        std::cout << i << " ";

    }

    std::cout << std::endl;

    return 0;

}

Output:-  
  
Original Array: 5 2 9 1 7 6 8 3 4

After parallel bubble sort: 1 2 3 4 5 6 7 8 9

After parallel merge sort: 1 2 3 4 5 6 7 8 9

**HPC3**

#include <iostream>

#include <vector>

#include <limits.h>

#include <omp.h>

using namespace std;

void min\_reduction(vector<int>& arr) {

    int min\_value = INT\_MAX;

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

    for (int i = 0; i < arr.size(); i++) {

        if (arr[i] < min\_value) {

            min\_value = arr[i];

        }

    }

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

}

void max\_reduction(vector<int>& arr) {

    int max\_value = INT\_MIN;

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

    for (int i = 0; i < arr.size(); i++) {

        if (arr[i] > max\_value) {

            max\_value = arr[i];

        }

    }

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

}

void sum\_reduction(vector<int>& arr) {

    int sum = 0;

    #pragma omp parallel for reduction(+: sum)

    for (int i = 0; i < arr.size(); i++) {

        sum += arr[i];

    }

    cout << "Sum: " << sum << endl;

}

void average\_reduction(vector<int>& arr) {

    int sum = 0;

    #pragma omp parallel for reduction(+: sum)

    for (int i = 0; i < arr.size(); i++) {

        sum += arr[i];

    }

    cout << "Average: " << (double)sum / arr.size() << endl;

}

int main() {

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

    min\_reduction(arr);

    max\_reduction(arr);

    sum\_reduction(arr);

    average\_reduction(arr);

    return 0;

}

Output:-  
  
Minimum value: 1

Maximum value: 9

Sum: 45

Average: 5

**HPC4**

**1) Vector Addition**

#include <cuda\_runtime.h>

#include <iostream>

#define N 1000000

\_\_global\_\_ void add(int\* a, int\* b, int\* c) {

    int i = blockIdx.x \* blockDim.x + threadIdx.x;

    if (i < N) {

        c[i] = a[i] + b[i];

    }

}

int main() {

    int \*a, \*b, \*c; // host vectors

    int \*dev\_a, \*dev\_b, \*dev\_c; // device vectors

    int size = N \* sizeof(int);

    // Allocate memory on host

    a = (int\*)malloc(size);

    b = (int\*)malloc(size);

    c = (int\*)malloc(size);

    // Allocate memory on device

    cudaMalloc((void\*\*)&dev\_a, size);

    cudaMalloc((void\*\*)&dev\_b, size);

    cudaMalloc((void\*\*)&dev\_c, size);

    // Initialize host vectors

    for (int i = 0; i < N; i++) {

        a[i] = i;

        b[i] = i \* 2;

    }

    // Copy host vectors to device

    cudaMemcpy(dev\_a, a, size, cudaMemcpyHostToDevice);

    cudaMemcpy(dev\_b, b, size, cudaMemcpyHostToDevice);

    // Launch kernel with 1 block and 1024 threads per block

    add<<<1, 1024>>>(dev\_a, dev\_b, dev\_c);

    // Copy result from device to host

    cudaMemcpy(c, dev\_c, size, cudaMemcpyDeviceToHost);

    // Print the first 10 values of the result

    for (int i = 0; i < 10; i++) {

        std::cout << c[i] << " ";

    }

    // Free memory

    cudaFree(dev\_a);

    cudaFree(dev\_b);

    cudaFree(dev\_c);

    free(a);

    free(b);

    free(c);

    return 0;

}

**Output:-**

0 3 6 9 12 15 18 21 24 27

**2) Matrix Multiplication**

#include <cuda\_runtime.h>

#include <iostream>

#define N 1000

\_\_global\_\_ void matrixMul(int\* a, int\* b, int\* c) {

    int row = blockIdx.y \* blockDim.y + threadIdx.y;

    int col = blockIdx.x \* blockDim.x + threadIdx.x;

    if (row < N && col < N) {

        int sum = 0;

        for (int k = 0; k < N; k++) {

            sum += a[row \* N + k] \* b[k \* N + col];

        }

        c[row \* N + col] = sum;

    }

}

int main() {

    int \*a, \*b, \*c; // host matrices

    int \*dev\_a, \*dev\_b, \*dev\_c; // device matrices

    int size = N \* N \* sizeof(int);

    // Allocate memory on host

    a = (int\*)malloc(size);

    b = (int\*)malloc(size);

    c = (int\*)malloc(size);

    // Allocate memory on device

    cudaMalloc((void\*\*)&dev\_a, size);

    cudaMalloc((void\*\*)&dev\_b, size);

    cudaMalloc((void\*\*)&dev\_c, size);

    // Initialize host matrices

    for (int i = 0; i < N; i++) {

        for (int j = 0; j < N; j++) {

            a[i \* N + j] = i;

            b[i \* N + j] = j;

        }

    }

    // Copy host matrices to device

    cudaMemcpy(dev\_a, a, size, cudaMemcpyHostToDevice);

    cudaMemcpy(dev\_b, b, size, cudaMemcpyHostToDevice);

    // Launch kernel with 1 block and 1024 threads per block

    dim3 blockDim(32, 32);

    dim3 gridDim((N + blockDim.x - 1) / blockDim.x,

                (N + blockDim.y - 1) / blockDim.y);

    matrixMul<<<gridDim, blockDim>>>(dev\_a, dev\_b, dev\_c);

    // Copy result from device to host

    cudaMemcpy(c, dev\_c, size, cudaMemcpyDeviceToHost);

    // Print the first 10 values of the result

    for (int i = 0; i < 10; i++) {

        std::cout << c[i] << " ";

    }

    // Free memory

    cudaFree(dev\_a);

    cudaFree(dev\_b);

    cudaFree(dev\_c);

    free(a);

    free(b);

    free(c);

    return 0;

}

**Output:-**

**0 495000 990000 1485000 1980000 2475000**

**2970000 3465000 3960000 4455000**