Assign 1 hpc :

#include<iostream>

#include<omp.h>

// #include<bits/stdc++.h>

#include<stack>

#include<queue>

#include<chrono>

using namespace std;

class Graph{

    public:

        // vector<vector<int>> graph;

        // vector<bool> visited;

        // int vertices = 0;

        // int edges = 0;

        int vertices = 6;

        int edges = 5;

        vector<vector<int>> graph = {{1},{0,2,3},{1,4,5},{1,4},{2,3},{2}};

        vector<bool> visited;

        // Graph(){

        //     cout << "Enter number of nodes: ";

        //     cin >> vertices;

        //     cout << "Enter number of edges: ";

        //     cin >> edges;

        //     graph.assign(vertices,vector<int>());

        //     for(int i = 0 ; i < edges;i++){

        //         int a,b;

        //         cout << "Enter adjacent nodes: ";

        //         cin >> a >> b;

        //         addEdge(a,b);

        //     }

        // }

        void addEdge(int a, int b){

            graph[a].push\_back(b);

            graph[b].push\_back(a);

        }

        void printGraph(){

            for(int i = 0; i < vertices; i++){

                cout << i << " -> ";

                for(auto j = graph[i].begin(); j != graph[i].end();j++){

                    cout << \*j << " ";

                }

                cout << endl;

            }

        }

        void initialize\_visited(){

            visited.assign(vertices,false);

        }

        void dfs(int i){

            stack<int> s;

            s.push(i);

            visited[i] = true;

            while(s.empty() != true){

                int current = s.top();

                cout << current << " ";

                s.pop();

                for(auto j = graph[current].begin(); j != graph[current].end();j++){

                    if(visited[\*j] == false){

                        s.push(\*j);

                        visited[\*j] = true;

                    }

                }

            }

        }

        void parallel\_dfs(int i){

            stack<int> s;

            s.push(i);

            visited[i] = true;

            while(s.empty() != true){

                int current = s.top();

                cout << current << " ";

                #pragma omp critical

                    s.pop();

                #pragma omp parallel for

                    for(auto j = graph[current].begin(); j != graph[current].end();j++){

                        if(visited[\*j] == false){

                            #pragma omp critical

                            {

                                s.push(\*j);

                                visited[\*j] = true;

                            }

                        }

                    }

            }

        }

        void bfs(int i){

            queue<int> q;

            q.push(i);

            visited[i] = true;

            while(q.empty() != true){

                int current = q.front();

                q.pop();

                cout << current << " ";

                for(auto j = graph[current].begin(); j != graph[current].end();j++){

                    if(visited[\*j] == false){

                        q.push(\*j);

                        visited[\*j] = true;

                    }

                }

            }

        }

        void parallel\_bfs(int i){

            queue<int> q;

            q.push(i);

            visited[i] = true;

            while(q.empty() != true){

                    int current = q.front();

                    cout << current << " ";

                    #pragma omp critical

                        q.pop();

                #pragma omp parallel for

                    for(auto j = graph[current].begin(); j != graph[current].end();j++){

                        if(visited[\*j] == false){

                            #pragma omp critical

                                q.push(\*j);

                                visited[\*j] = true;

                        }

                    }

            }

        }

};

int main(int argc, char const \*argv[])

{

    Graph g;

    cout << "Adjacency List:\n";

    g.printGraph();

    g.initialize\_visited();

    cout << "Depth First Search: \n";

    auto start = chrono::high\_resolution\_clock::now();

    g.dfs(0);

    cout << endl;

    auto end = chrono::high\_resolution\_clock::now();

    cout << "Time taken: " << chrono::duration\_cast<chrono::microseconds>(end - start).count() << " microseconds" << endl;

    cout << "Parallel Depth First Search: \n";

    g.initialize\_visited();

    start = chrono::high\_resolution\_clock::now();

    g.parallel\_dfs(0);

    cout << endl;

    end = chrono::high\_resolution\_clock::now();

    cout << "Time taken: "<< chrono::duration\_cast<chrono::microseconds>(end - start).count() << " microseconds" << endl;

    start = chrono::high\_resolution\_clock::now();

    cout << "Breadth First Search: \n";

    g.initialize\_visited();

    g.bfs(0);

    cout << endl;

    end = chrono::high\_resolution\_clock::now();

    cout << "Time taken: "<< chrono::duration\_cast<chrono::microseconds>(end - start).count() << " microseconds" << endl;

    start = chrono::high\_resolution\_clock::now();

    cout << "Parallel Breadth First Search: \n";

    g.initialize\_visited();

    g.parallel\_bfs(0);

    cout << endl;

    end = chrono::high\_resolution\_clock::now();

    cout << "Time taken: " << chrono::duration\_cast<chrono::microseconds>(end - start).count() << " microseconds" << endl;

    return 0;

}

A2:

#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){

    int i,j;

    #pragma omp parallel for shared(array, n) private(i, j)

    for (i = 0; i < n - 1; i++) {

        for (j = 0; j < n - i - 1; j++) {

            if (array[j] > array[j + 1])

            swap(array[j], array[j + 1]);

        }

    }

}

int main(){

    int n = 10000;

    int a[n];

    double start\_time, end\_time;

    // Create an array of n numbers, with numbers from n to 1

    for(int i = 0, j = n; i < n; i++, j--) a[i] = j;

    // Create a copy

    int b[n];

    for(int i = 0; i < n; i++) b[i] = a[i];

    // Measure Sequential Time

    start\_time = omp\_get\_wtime();

    bubble(a,n);

    end\_time = omp\_get\_wtime();

    cout << "Time taken by sequential algorithm: " << end\_time - start\_time << " seconds\n";

    //Measure Parallel time

    start\_time = omp\_get\_wtime();

    pBubble(b,n);

    end\_time = omp\_get\_wtime();

    cout << "Time taken by parallel algorithm: " << end\_time - start\_time << " seconds";

    /\*

    The parallel bubble sort algorithm does not do better than the sequential algorithm. In fact, it is actually slower in most cases. This is because the bubble sort algorithm is a very simple algorithm that does not lend itself well to parallelism. The overhead of synchronizing the threads and dividing the array into smaller parts outweighs the benefits of parallel execution.

    \*/

    return 0;

}

#include<iostream>

#include<omp.h>

using namespace std;

void merge(int a[], int i1, int j1, int i2, int j2, int n){

    int temp[n];

    int i, j, k;

    i = i1;

    j = i2;

    k = 0;

    while(i <= j1 && j <= j2){

        if(a[i] < a[j]) temp[k++] = a[i++];

        else temp[k++] = a[j++];

    }

    while(i <= j1) temp[k++] = a[i++];

    while(j <= j2) temp[k++] = a[j++];

    for(i = i1, j = 0; i <= j2; i++, j++) a[i] = temp[j];

}

void mergesort(int a[],int i,int j, int n){

    int mid;

    if(i < j){

        mid = (i + j) / 2;

        mergesort(a, i, mid, n);

        mergesort(a, mid + 1, j, n);

        merge(a, i, mid, mid + 1, j, n);

    }

}

void pMergesort(int a[],int i,int j, int n){

    int mid;

    if(i < j){

        mid = (i + j) / 2;

        #pragma omp task firstprivate(a, i, j, n)

        mergesort(a, i, mid, n);

        #pragma omp task firstprivate(a, i, j, n)

        mergesort(a, mid + 1, j, n);

        #pragma omp taskwait

        merge(a, i, mid, mid + 1, j, n);

    }

}

int main()

{

    int n = 10;

    int a[n];

    double start\_time, end\_time;

    // Create an array of n numbers, with digits from n to 1 in descending order

    for(int i = 0, j = n; i < n; i++, j--) a[i] = j;

    // Create a copy

    int b[n];

    for(int i = 0; i < n; i++) b[i] = a[i];

    // Measure Sequential Time

    start\_time = omp\_get\_wtime();

    mergesort(a, 0, n - 1, n);

    end\_time = omp\_get\_wtime();

    cout << "Time taken by sequential algorithm: " << end\_time - start\_time << " seconds\n";

     //Measure Parallel time

    start\_time = omp\_get\_wtime();

    #pragma omp parallel

    {

        #pragma omp single

        {

            mergesort(b, 0, n - 1, n);

        }

    }

    end\_time = omp\_get\_wtime();

    cout << "\nTime taken by parallel algorithm: " << end\_time - start\_time << " seconds";

    // Unfortunately parallel algorithms only do well on large scales.

    // In our case sequential may always do better than parallel.

    // This is because parallel algorithms have the overhead of creating threads.

    return 0;

}

A3:

#include <iostream>

#include <omp.h>

#include <climits>

/\*

The min\_reduction function finds the minimum value in the input array using the #pragma omp parallel for

reduction(min: min\_value) directive, which creates a parallel region and divides the loop iterations among the

available threads. Each thread performs the comparison operation in parallel and updates the min\_value

variable if a smaller value is found.

1. Similarly, the max\_reduction function finds the maximum value in the array, sum\_reduction function finds the

sum of the elements of array and average\_reduction function finds the average of the elements of array by

dividing the sum by the size of the array.

1. The reduction clause is used to combine the results of multiple threads into a single value, which is then

returned by the function. The min and max operators are used for the min\_reduction and max\_reduction

functions, respectively, and the + operator is used for the sum\_reduction and average\_reduction functions. In

the main function, it creates a vector and calls the functions min\_reduction, max\_reduction, sum\_reduction, and

average\_reduction to compute the values of min, max, sum and average respectively.

\*/

using namespace std;

void min\_reduction(int arr[], int n) {

    int min\_value = INT\_MAX;

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

    for (int i = 0; i < n; i++) {

        if (arr[i] < min\_value) {

            min\_value = arr[i];

        }

    }

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

}

void max\_reduction(int arr[], int n) {

    int max\_value = INT\_MIN;

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

    for (int i = 0; i < n; i++) {

        if (arr[i] > max\_value) {

            max\_value = arr[i];

        }

    }

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

}

void sum\_reduction(int arr[], int n) {

    int sum = 0;

    #pragma omp parallel for reduction(+: sum)

    for (int i = 0; i < n; i++) {

        sum += arr[i];

    }

    cout << "Sum: " << sum << endl;

}

void average\_reduction(int arr[], int n) {

    int sum = 0;

    #pragma omp parallel for reduction(+: sum)

    for (int i = 0; i < n; i++) {

        sum += arr[i];

    }

    cout << "Average: " << (double)sum / n << endl;

}

int main() {

     int arr[] = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3};

    int n = sizeof(arr) / sizeof(arr[0]);

    min\_reduction(arr, n);

    max\_reduction(arr, n);

    sum\_reduction(arr, n);

    average\_reduction(arr, n);

    return 0;

}

A4:

#include <iostream>

#include <cuda\_runtime.h>

using namespace std;

\_\_global\_\_ void addVectors(int\* A, int\* B, int\* C, int n)

{

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

    if (i < n)

    {

        C[i] = A[i] + B[i];

    }

}

int main()

{

    int n = 1000000;

    int\* A, \* B, \* C;

    int size = n \* sizeof(int);

    // Allocate memory on the host

    cudaMallocHost(&A, size);

    cudaMallocHost(&B, size);

    cudaMallocHost(&C, size);

    // Initialize the vectors

    for (int i = 0; i < n; i++)

    {

        A[i] = i;

        B[i] = i \* 2;

    }

    // Allocate memory on the device

    int\* dev\_A, \* dev\_B, \* dev\_C;

    cudaMalloc(&dev\_A, size);

    cudaMalloc(&dev\_B, size);

    cudaMalloc(&dev\_C, size);

    // Copy data from host to device

    cudaMemcpy(dev\_A, A, size, cudaMemcpyHostToDevice);

    cudaMemcpy(dev\_B, B, size, cudaMemcpyHostToDevice);

    // Launch the kernel

    int blockSize = 256;

    int numBlocks = (n + blockSize - 1) / blockSize;

    addVectors<<<numBlocks, blockSize>>>(dev\_A, dev\_B, dev\_C, n);

    // Copy data from device to host

    cudaMemcpy(C, dev\_C, size, cudaMemcpyDeviceToHost);

    // Print the results

    for (int i = 0; i < 10; i++)

    {

        cout << C[i] << " ";

    }

    cout << endl;

    // Free memory

    cudaFree(dev\_A);

    cudaFree(dev\_B);

    cudaFree(dev\_C);

    cudaFreeHost(A);

    cudaFreeHost(B);

    cudaFreeHost(C);

    return 0;

}

#include <cuda\_runtime.h>

#include <iostream>

\_\_global\_\_ void matmul(int\* A, int\* B, int\* C, int N) {

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

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

    if (Row < N && Col < N) {

        int Pvalue = 0;

        for (int k = 0; k < N; k++) {

            Pvalue += A[Row\*N+k] \* B[k\*N+Col];

        }

        C[Row\*N+Col] = Pvalue;

    }

}

int main() {

    int N = 512;

    int size = N \* N \* sizeof(int);

    int\* A, \* B, \* C;

    int\* dev\_A, \* dev\_B, \* dev\_C;

    cudaMallocHost(&A, size);

    cudaMallocHost(&B, size);

    cudaMallocHost(&C, size);

    cudaMalloc(&dev\_A, size);

    cudaMalloc(&dev\_B, size);

    cudaMalloc(&dev\_C, size);

    // Initialize matrices A and B

    for (int i = 0; i < N; i++) {

        for (int j = 0; j < N; j++) {

            A[i\*N+j] = i\*N+j;

            B[i\*N+j] = j\*N+i;

        }

    }

    cudaMemcpy(dev\_A, A, size, cudaMemcpyHostToDevice);

    cudaMemcpy(dev\_B, B, size, cudaMemcpyHostToDevice);

    dim3 dimBlock(16, 16);

    dim3 dimGrid(N/dimBlock.x, N/dimBlock.y);

    matmul<<<dimGrid, dimBlock>>>(dev\_A, dev\_B, dev\_C, N);

    cudaMemcpy(C, dev\_C, size, cudaMemcpyDeviceToHost);

    // Print the result

    for (int i = 0; i < 10; i++) {

        for (int j = 0; j < 10; j++) {

            std::cout << C[i\*N+j] << " ";

        }

        std::cout << std::endl;

    }

    // Free memory

    cudaFree(dev\_A);

    cudaFree(dev\_B);

    cudaFree(dev\_C);

    cudaFreeHost(A);

    cudaFreeHost(B);

    cudaFreeHost(C);

    return 0;

}