**Parallel BFS:**

#include <iostream>

#include <queue>

#include <vector>

#include <omp.h>

using namespace std;

void parallelBFS(int start, vector<vector<int>>& adj, int numNodes) {

vector<bool> visited(numNodes, false);

queue<int> q;

visited[start] = true;

q.push(start);

// Parallel BFS using OpenMP

while (!q.empty()) {

int node = q.front();

q.pop();

cout << node << " ";

#pragma omp parallel for

for (int i = 0; i < adj[node].size(); i++) {

int neighbor = adj[node][i];

if (!visited[neighbor]) {

#pragma omp critical

{

visited[neighbor] = true;

q.push(neighbor);

}

}

}

}

cout << endl;

}

int main() {

int numNodes = 6;

vector<vector<int>> adj(numNodes);

adj[0].push\_back(1);

adj[0].push\_back(2);

adj[1].push\_back(0);

adj[1].push\_back(3);

adj[2].push\_back(0);

adj[2].push\_back(4);

adj[3].push\_back(1);

adj[3].push\_back(5);

adj[4].push\_back(2);

adj[4].push\_back(5);

adj[5].push\_back(3);

adj[5].push\_back(4);

int startNode = 0;

parallelBFS(startNode, adj, numNodes);

return 0;

}

**Parallel DFS:**

#include <iostream>

#include <vector>

#include <stack>

#include <omp.h>

using namespace std;

void parallelDFS(int start, vector<vector<int>>& adj, int numNodes) {

vector<bool> visited(numNodes, false);

stack<int> s;

s.push(start);

visited[start] = true;

// Parallel DFS using OpenMP

while (!s.empty()) {

int node = s.top();

s.pop();

cout << node << " ";

#pragma omp parallel for

for (int i = 0; i < adj[node].size(); i++) {

int neighbor = adj[node][i];

if (!visited[neighbor]) {

#pragma omp critical

{

visited[neighbor] = true;

s.push(neighbor);

}

}

}

}

cout << endl;

}

int main() {

int numNodes = 6;

vector<vector<int>> adj(numNodes);

adj[0].push\_back(1);

adj[0].push\_back(2);

adj[1].push\_back(0);

adj[1].push\_back(3);

adj[2].push\_back(0);

adj[2].push\_back(4);

adj[3].push\_back(1);

adj[3].push\_back(5);

adj[4].push\_back(2);

adj[4].push\_back(5);

adj[5].push\_back(3);

adj[5].push\_back(4);

int startNode = 0;

parallelDFS(startNode, adj, numNodes);

return 0;

}

**Bubble Sort:**

#include <iostream>

#include <vector>

#include <cstdlib>

#include <ctime>

#include <omp.h>

using namespace std;

void sequentialBubbleSort(vector<int>& arr) {

int n = arr.size();

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

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

if (arr[j] > arr[j+1]) {

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

}

}

}

}

void parallelBubbleSort(vector<int>& arr) {

int n = arr.size();

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

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() {

const int size = 1000; // Change this size as needed

vector<int> data(size), dataCopy;

srand(time(0));

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

data[i] = rand() % 1000;

}

dataCopy = data;

// Sequential

double startSeq = omp\_get\_wtime();

sequentialBubbleSort(data);

double endSeq = omp\_get\_wtime();

// Parallel

double startPar = omp\_get\_wtime();

parallelBubbleSort(dataCopy);

double endPar = omp\_get\_wtime();

cout << "Sequential Time: " << (endSeq - startSeq) << " seconds\n";

cout << "Parallel Time: " << (endPar - startPar) << " seconds\n";

return 0;

}

**Merge Sort:**

#include <iostream>

#include <vector>

#include <cstdlib>

#include <ctime>

#include <omp.h>

using namespace std;

void merge(vector<int>& arr, int left, int mid, int right) {

vector<int> temp(right - left + 1);

int i = left, j = mid + 1, k = 0;

while(i <= mid && j <= right) {

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

else temp[k++] = arr[j++];

}

while(i <= mid) temp[k++] = arr[i++];

while(j <= right) temp[k++] = arr[j++];

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

arr[left + i] = temp[i];

}

void sequentialMergeSort(vector<int>& arr, int left, int right) {

if(left < right) {

int mid = (left + right) / 2;

sequentialMergeSort(arr, left, mid);

sequentialMergeSort(arr, mid + 1, right);

merge(arr, left, mid, right);

}

}

void parallelMergeSort(vector<int>& arr, int left, int right, int depth = 0) {

if(left < right) {

int mid = (left + right) / 2;

if (depth <= 3) { // Limit depth to avoid too many threads

#pragma omp parallel sections

{

#pragma omp section

parallelMergeSort(arr, left, mid, depth + 1);

#pragma omp section

parallelMergeSort(arr, mid + 1, right, depth + 1);

}

} else {

sequentialMergeSort(arr, left, mid);

sequentialMergeSort(arr, mid + 1, right);

}

merge(arr, left, mid, right);

}

}

int main() {

const int size = 100000;

vector<int> arr(size), arrCopy;

srand(time(0));

for (int i = 0; i < size; i++)

arr[i] = rand() % 100000;

arrCopy = arr;

// Sequential Merge Sort

double startSeq = omp\_get\_wtime();

sequentialMergeSort(arr, 0, size - 1);

double endSeq = omp\_get\_wtime();

// Parallel Merge Sort

double startPar = omp\_get\_wtime();

parallelMergeSort(arrCopy, 0, size - 1);

double endPar = omp\_get\_wtime();

cout << "Sequential Time: " << (endSeq - startSeq) << " seconds\n";

cout << "Parallel Time: " << (endPar - startPar) << " seconds\n";

return 0;

}

**Reduction Min , Max , Sum, Average:**

#include <iostream>

#include <vector>

#include <omp.h>

#include <climits>

using namespace std;

int main() {

int n = 1000000;

vector<int> arr(n);

// Initialize with some values

for (int i = 0; i < n; i++)

arr[i] = rand() % 1000;

int max\_val = INT\_MIN;

int min\_val = INT\_MAX;

long long sum = 0;

double avg = 0;

// Parallel Reduction

#pragma omp parallel for reduction(max:max\_val) reduction(min:min\_val) reduction(+:sum)

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

if (arr[i] > max\_val)

max\_val = arr[i];

if (arr[i] < min\_val)

min\_val = arr[i];

sum += arr[i];

}

avg = (double)sum / n;

// Output results

cout << "Parallel Reduction Results:" << endl;

cout << "Max: " << max\_val << endl;

cout << "Min: " << min\_val << endl;

cout << "Sum: " << sum << endl;

cout << "Average: " << avg << endl;

return 0;

}

**CUDA vector addition:**#include <iostream>

#include <cuda\_runtime.h>

using namespace std;

// GPU Kernel for vector addition

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

int i = threadIdx.x;

if (i < n) {

c[i] = a[i] + b[i];

}

}

int main() {

const int N = 5;

int a[N] = {1, 2, 3, 4, 5};

int b[N] = {10, 20, 30, 40, 50};

int c[N]; // Result array

int \*d\_a, \*d\_b, \*d\_c;

// Step 1: Allocate memory on GPU

cudaMalloc((void \*\*)&d\_a, N \* sizeof(int));

cudaMalloc((void \*\*)&d\_b, N \* sizeof(int));

cudaMalloc((void \*\*)&d\_c, N \* sizeof(int));

// Step 2: Copy input data to GPU

cudaMemcpy(d\_a, a, N \* sizeof(int), cudaMemcpyHostToDevice);

cudaMemcpy(d\_b, b, N \* sizeof(int), cudaMemcpyHostToDevice);

// Step 3: Launch kernel with N threads

addVectors<<<1, N>>>(d\_a, d\_b, d\_c, N);

// Step 4: Copy result back to CPU

cudaMemcpy(c, d\_c, N \* sizeof(int), cudaMemcpyDeviceToHost);

// Step 5: Print result

cout << "Result: ";

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

cout << c[i] << " ";

}

// Step 6: Free GPU memory

cudaFree(d\_a);

cudaFree(d\_b);

cudaFree(d\_c);

return 0;

}

**Cuda matrix multiplication:**

#include <iostream>

#include <cuda\_runtime.h>

using namespace std;

#define N 3

// CUDA Kernel for Matrix Multiplication

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

int row = threadIdx.y; // 0,1,2

int col = threadIdx.x; // 0,1,2

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[N\*N] = {

1, 2, 3,

4, 5, 6,

7, 8, 9

};

int b[N\*N] = {

9, 8, 7,

6, 5, 4,

3, 2, 1

};

int c[N\*N]; // Result matrix

int \*d\_a, \*d\_b, \*d\_c;

// Allocate memory on GPU

cudaMalloc((void\*\*)&d\_a, N\*N\*sizeof(int));

cudaMalloc((void\*\*)&d\_b, N\*N\*sizeof(int));

cudaMalloc((void\*\*)&d\_c, N\*N\*sizeof(int));

// Copy data from CPU to GPU

cudaMemcpy(d\_a, a, N\*N\*sizeof(int), cudaMemcpyHostToDevice);

cudaMemcpy(d\_b, b, N\*N\*sizeof(int), cudaMemcpyHostToDevice);

// Launch kernel with 2D block of threads (3x3 = 9 threads)

dim3 threadsPerBlock(N, N);

matrixMul<<<1, threadsPerBlock>>>(d\_a, d\_b, d\_c);

// Copy result back to CPU

cudaMemcpy(c, d\_c, N\*N\*sizeof(int), cudaMemcpyDeviceToHost);

// Print result

cout << "Result Matrix:\n";

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

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

cout << c[i \* N + j] << " ";

}

cout << endl;

}

// Free GPU memory

cudaFree(d\_a);

cudaFree(d\_b);

cudaFree(d\_c);

return 0;

}