**Parallel bfs**

#include <iostream>

#include <vector>

#include <queue>

#include <omp.h>

using namespace std;

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

queue<int> q;

q.push(start);

visited[start] = true;

#pragma omp parallel

{

#pragma omp single

{

while (!q.empty()) {

int vertex = q.front();

q.pop();

#pragma omp task firstprivate(vertex)

{

for (int neighbor : graph[vertex]) {

if (!visited[neighbor]) {

q.push(neighbor);

visited[neighbor] = true;

#pragma omp task

bfs(graph, neighbor, visited);

}

}

}

}

}

}

}

void parallel\_bfs(vector<vector<int>>& graph, int start) {

vector<bool> visited(graph.size(), false);

bfs(graph, start, visited);

}

int main() {

// Create an undirected graph with 7 vertices

vector<vector<int>> graph(7);

graph[0] = {1, 2};

graph[1] = {0, 2, 3, 4};

graph[2] = {0, 1, 5, 6};

graph[3] = {1, 4};

graph[4] = {1, 3};

graph[5] = {2};

graph[6] = {2};

// Run parallel BFS on the graph starting from vertex 0

parallel\_bfs(graph, 0);

return 0;

}

**Parallel DFS**

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

stack<int> s;

s.push(start);

visited[start] = true;

#pragma omp parallel

{

#pragma omp single

{

while (!s.empty()) {

int vertex = s.top();

s.pop();

#pragma omp task firstprivate(vertex)

{

for (int neighbor : graph[vertex]) {

if (!visited[neighbor]) {

s.push(neighbor);

visited[neighbor] = true;

#pragma omp task

dfs(graph, neighbor, visited);

}

}

}

}

}

}

}

void parallel\_dfs(vector<vector<int>>& graph, int start) {

vector<bool> visited(graph.size(), false);

dfs(graph, start, visited);

}

**Bubble sort**

#include <iostream>

#include <vector>

#include <omp.h>

using namespace std;

void bubble\_sort\_odd\_even(vector<int>& arr) {

bool isSorted = false;

while (!isSorted) {

isSorted = true;

#pragma omp parallel for

for (int i = 0; i < arr.size() - 1; i += 2) {

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

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

isSorted = false;

}

}

#pragma omp parallel for

for (int i = 1; i < arr.size() - 1; i += 2) {

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

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

isSorted = false;

}

}

}

}

int main() {

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

double start, end;

// Measure performance of parallel bubble sort using odd-even transposition

start = omp\_get\_wtime();

bubble\_sort\_odd\_even(arr);

end = omp\_get\_wtime();

cout << "Parallel bubble sort using odd-even transposition time: " << end - start << endl;

return 0;

}

**Merge sort**

#include <iostream>

#include <vector>

#include <omp.h>

using namespace std;

void merge(vector<int>& arr, int l, int m, int r) {

int i, j, k;

int n1 = m - l + 1;

int n2 = r - m;

vector<int> L(n1), R(n2);

for (i = 0; i < n1; i++)

L[i] = arr[l + i];

for (j = 0; j < n2; j++)

R[j] = arr[m + 1 + j];

i = j = 0;

k = l;

while (i < n1 && j < n2) {

if (L[i] <= R[j])

arr[k++] = L[i++];

else

arr[k++] = R[j++];

}

while (i < n1)

arr[k++] = L[i++];

while (j < n2)

arr[k++] = R[j++];

}

void merge\_sort(vector<int>& arr, int l, int r) {

if (l < r) {

int m = l + (r - l) / 2;

merge\_sort(arr, l, m);

merge\_sort(arr, m + 1, r);

merge(arr, l, m, r);

}

}

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

#pragma omp parallel

{

#pragma omp single

merge\_sort(arr, 0, arr.size() - 1);

}

}

int main() {

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

double start, end;

// Measure performance of sequential merge sort

start = omp\_get\_wtime();

merge\_sort(arr, 0, arr.size() - 1);

end = omp\_get\_wtime();

cout << "Sequential merge sort time: " << end - start << endl;

// Measure performance of parallel merge sort

arr = {5, 2, 9, 1, 7, 6, 8, 3, 4};

start = omp\_get\_wtime();

parallel\_merge\_sort(arr);

end = omp\_get\_wtime();

cout << "Parallel merge sort time: " << end - start << endl;

return 0;

}

**Minmax reduction**

#include <iostream>

#include <vector>

#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);

}

**Cuda (add vector)**

#include <iostream>

#include <cuda\_runtime.h>

\_\_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++) {

std::cout << C[i] << " ";

}

std::cout << std::endl;

// Free memory

cudaFree(dev\_A);

cudaFree(dev\_B);

cudaFree(dev\_C);

cudaFreeHost(A);

cudaFreeHost(B);

cudaFreeHost(C);

return 0;

}

**Cuda (multiplication)**

#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;

}

**Mpi quicksort**

#include <iostream>

#include <algorithm>

#include <mpi.h>

using namespace std;

// Function to partition the array

int partition(int arr[], int low, int high) {

int pivot = arr[high];

int i = (low - 1);

for (int j = low; j <= high - 1; j++) {

if (arr[j] <= pivot) {

i++;

swap(arr[i], arr[j]);

}

}

swap(arr[i + 1], arr[high]);

return (i + 1);

}

// Function to perform quicksort on the partition

void quicksort(int arr[], int low, int high) {

if (low < high) {

int pivot = partition(arr, low, high);

quicksort(arr, low, pivot - 1);

quicksort(arr, pivot + 1, high);

}

}

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

int rank, size;

MPI\_Init(&argc, &argv);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

int n = 100;

int\* arr = new int[n];

int\* recvbuf = new int[n];

int\* sendbuf = new int[n];

// Fill the array with random values

if (rank == 0) {

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

arr[i] = rand() % 100;

}

}

// Divide the array into equal-sized partitions for each process

int sub\_arr\_size = n / size;

int\* sub\_arr = new int[sub\_arr\_size];

MPI\_Scatter(arr, sub\_arr\_size, MPI\_INT, sub\_arr, sub\_arr\_size, MPI\_INT, 0, MPI\_COMM\_WORLD);

// Sort the partition using quicksort

quicksort(sub\_arr, 0, sub\_arr\_size - 1);

// Gather the sorted partitions from each process

MPI\_Gather(sub\_arr, sub\_arr\_size, MPI\_INT, recvbuf, sub\_arr\_size, MPI\_INT, 0, MPI\_COMM\_WORLD);

// Print the sorted array

if (rank == 0) {

cout << "Sorted array: ";

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

cout << recvbuf[i] << " ";

}

cout << endl;

// Measure the execution time of the parallel quicksort algorithm

double start\_time = MPI\_Wtime();

// Perform the above steps again

double end\_time = MPI\_Wtime();

double parallel\_execution\_time = end\_time - start\_time;

// Measure the execution time of the sequential quicksort algorithm

start\_time = MPI\_Wtime();

quicksort(arr, 0, n - 1);

end\_time = MPI\_Wtime();

double sequential\_execution\_time = end\_time - start\_time;

// Calculate speedup and efficiency

double speedup = sequential\_execution\_time / parallel\_execution\_time;

double efficiency = speedup / size;

cout << "Sequential execution time: " << sequential\_execution\_time << endl;

cout << "Parallel execution time: " << parallel\_execution\_time << endl;

cout << "Speedup: " << speedup << endl;

cout << "Efficiency: " << efficiency << endl;

}

MPI\_Finalize();

return 0;

}

**Huffman**

#include <iostream>

#include <cuda\_runtime.h>

\_\_global\_\_ void buildHuffmanTree(int\* frequencies, int\* tree, int n) {

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

if (i < n) {

// Find the two lowest frequency nodes

int min1 = INT\_MAX, min2 = INT\_MAX;

int minIndex1, minIndex2;

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

if (frequencies[j] != 0 && frequencies[j] < min1) {

min2 = min1;

minIndex2 = minIndex1;

min1 = frequencies[j];

minIndex1 = j;

} else if (frequencies[j] != 0 && frequencies[j] < min2) {

min2 = frequencies[j];

minIndex2 = j;

}

}

// Combine the two lowest frequency nodes into a new node

int newNodeIndex = n + i;

frequencies[newNodeIndex] = min1 + min2;

tree[newNodeIndex] = 0;

tree[newNodeIndex + n] = 0;

if (minIndex1 < minIndex2) {

tree[newNodeIndex] = minIndex1;

tree[newNodeIndex + n] = minIndex2;

} else {

tree[newNodeIndex] = minIndex2;

tree[newNodeIndex + n] = minIndex1;

}

}

}

int main() {

int n = 256;

int\* frequencies;

int\* tree;

cudaMalloc(&frequencies, n \* sizeof(int));

cudaMalloc(&tree, 2 \* n \* sizeof(int));

// Initialize frequencies

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

frequencies[i] = i + 1;

}

int numBlocks = (n + 255) / 256;

buildHuffmanTree<<<numBlocks, 256>>>(frequencies, tree, n);

// Encode the data using the Huffman tree

// ...

cudaFree(frequencies);

cudaFree(tree);

return 0;

}