**Practical 1 : DFS & BFS**

#include <iostream>

#include <vector>

#include <queue>

#include <omp.h>

using namespace std;

// Graph class representing the adjacency list

class Graph {

int V; // Number of vertices

vector<vector<int>> adj; // Adjacency list

public:

Graph(int V) : V(V), adj(V) {}

// Add an edge to the graph

void addEdge(int v, int w) {

adj[v].push\_back(w);

}

// Parallel Depth-First Search

void parallelDFS(int startVertex) {

vector<bool> visited(V, false);

parallelDFSUtil(startVertex, visited);

}

// Parallel DFS utility function

void parallelDFSUtil(int v, vector<bool>& visited) {

visited[v] = true;

cout << v << " ";

#pragma omp parallel for

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

int n = adj[v][i];

if (!visited[n])

parallelDFSUtil(n, visited);

}

}

// Parallel Breadth-First Search

void parallelBFS(int startVertex) {

vector<bool> visited(V, false);

queue<int> q;

visited[startVertex] = true;

q.push(startVertex);

while (!q.empty()) {

int v = q.front();

q.pop();

cout << v << " ";

#pragma omp parallel for

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

int n = adj[v][i];

if (!visited[n]) {

visited[n] = true;

q.push(n);

}

}

}

}

};

int main() {

// Create a graph

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);

/\*

0 -------->1

| / \

| / \

| / \

v v v

2 ----> 3 4

| |

| |

v v

5 6

\*/

cout << "Depth-First Search (DFS): ";

g.parallelDFS(0);

cout << endl;

cout << "Breadth-First Search (BFS): ";

g.parallelBFS(0);

cout << endl;

return 0;

}

**Practical 2: Merge & Bubble Sort**

Compile & run => g++ sort.cpp -lgomp -o sort

./sort

**BUBBLE SORT :**

#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){

//Sort odd indexed numbers

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

#pragma omp for

for (int j = 1; j < n; j += 2){

if (array[j] < array[j-1])

{

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

}

}

// Synchronize

#pragma omp barrier

//Sort even indexed numbers

#pragma omp for

for (int j = 2; j < n; j += 2){

if (array[j] < array[j-1])

{

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

}

}

}

}

void printArray(int arr[], int n){

for(int i = 0; i < n; i++) cout << arr[i] << " ";

cout << "\n";

}

int main(){

// Set up variables

int n = 10;

int arr[n];

int brr[n];

double start\_time, end\_time;

// Create an array with numbers starting from n to 1

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

// Sequential time

start\_time = omp\_get\_wtime();

bubble(arr, n);

end\_time = omp\_get\_wtime();

cout << "Sequential Bubble Sort took : " << end\_time - start\_time << " seconds.\n";

printArray(arr, n);

// Reset the array

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

// Parallel time

start\_time = omp\_get\_wtime();

pBubble(arr, n);

end\_time = omp\_get\_wtime();

cout << "Parallel Bubble Sort took : " << end\_time - start\_time << " seconds.\n";

printArray(arr, n);

}

**MERGE SORT 1 :**

#include <iostream>

#include <omp.h>

using namespace std;

void merge(int arr[], int low, int mid, int high) {

// Create arrays of left and right partititons

int n1 = mid - low + 1;

int n2 = high - mid;

int left[n1];

int right[n2];

// Copy all left elements

for (int i = 0; i < n1; i++) left[i] = arr[low + i];

// Copy all right elements

for (int j = 0; j < n2; j++) right[j] = arr[mid + 1 + j];

// Compare and place elements

int i = 0, j = 0, k = low;

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

if (left[i] <= right[j]){

arr[k] = left[i];

i++;

}

else{

arr[k] = right[j];

j++;

}

k++;

}

// If any elements are left out

while (i < n1) {

arr[k] = left[i];

i++;

k++;

}

while (j < n2) {

arr[k] = right[j];

j++;

k++;

}

}

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

if (low < high) {

int mid = (low + high) / 2;

#pragma omp parallel sections

{

#pragma omp section

{

parallelMergeSort(arr, low, mid);

}

#pragma omp section

{

parallelMergeSort(arr, mid + 1, high);

}

}

merge(arr, low, mid, high);

}

}

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

if (low < high) {

int mid = (low + high) / 2;

mergeSort(arr, low, mid);

mergeSort(arr, mid + 1, high);

merge(arr, low, mid, high);

}

}

int main() {

int n = 10;

int arr[n];

double start\_time, end\_time;

// Create an array with numbers starting from n to 1.

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

// Measure Sequential Time

start\_time = omp\_get\_wtime();

mergeSort(arr, 0, n - 1);

end\_time = omp\_get\_wtime();

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

// Reset the array

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

//Measure Parallel time

start\_time = omp\_get\_wtime();

parallelMergeSort(arr, 0, n - 1);

end\_time = omp\_get\_wtime();

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

return 0;

}

**MERGE SORT 2 :**

#include <omp.h>

#include <stdlib.h>

#include <array>

#include <chrono>

#include <functional>

#include <iostream>

#include <string>

#include <vector>

using std::chrono::duration\_cast;

using std::chrono::high\_resolution\_clock;

using std::chrono::milliseconds;

using namespace std;

void p\_mergesort(int \*a, int i, int j);

void s\_mergesort(int \*a, int i, int j);

void merge(int \*a, int i1, int j1, int i2, int j2);

void p\_mergesort(int \*a, int i, int j) {

int mid;

if (i < j) {

if ((j - i) > 1000) {

mid = (i + j) / 2;

#pragma omp task firstprivate(a, i, mid)

p\_mergesort(a, i, mid);

#pragma omp task firstprivate(a, mid, j)

p\_mergesort(a, mid + 1, j);

#pragma omp taskwait

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

} else {

s\_mergesort(a, i, j);

}

}

}

void parallel\_mergesort(int \*a, int i, int j) {

#pragma omp parallel num\_threads(16)

{

#pragma omp single

p\_mergesort(a, i, j);

}

}

void s\_mergesort(int \*a, int i, int j) {

int mid;

if (i < j) {

mid = (i + j) / 2;

s\_mergesort(a, i, mid);

s\_mergesort(a, mid + 1, j);

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

}

}

void merge(int \*a, int i1, int j1, int i2, int j2) {

int temp[2000000];

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];

}

}

std::string bench\_traverse(std::function<void()> traverse\_fn) {

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

traverse\_fn();

auto stop = high\_resolution\_clock::now();

// Subtract stop and start timepoints and cast it to required unit.

// Predefined units are nanoseconds, microseconds, milliseconds, seconds,

// minutes, hours. Use duration\_cast() function.

auto duration = duration\_cast<milliseconds>(stop - start);

// To get the value of duration use the count() member function on the

// duration object

return std::to\_string(duration.count());

}

int main(int argc, const char \*\*argv) {

int \*a, n, rand\_max;

n = 100;

rand\_max = 200;

a = new int[n];

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

a[i] = rand() % rand\_max;

}

int \*b = new int[n];

copy(a, a + n, b);

cout << "Generated random array of length " << n << " with elements between 0 to " << rand\_max

<< "\n\n";

std::cout << "Sequential Merge sort: " << bench\_traverse([&] { s\_mergesort(a, 0, n - 1); })

<< "ms\n";

cout << "Sorted array is =>\n";

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

cout << a[i] << ", ";

}

cout << "\n\n";

omp\_set\_num\_threads(16);

std::cout << "Parallel (16) Merge sort: "

<< bench\_traverse([&] { parallel\_mergesort(b, 0, n - 1); }) << "ms\n";

cout << "Sorted array is =>\n";

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

cout << b[i] << ", ";

}

return 0;

}

/\*

OUTPUT:

Generated random array of length 100 with elements between 0 to 200

Sequential Merge sort: 0ms

Sorted array is =>

2, 3, 8, 11, 11, 12, 13, 14, 21, 21, 22, 26, 26, 27, 29, 29, 34, 42, 43, 46, 49, 51, 56, 57, 58, 59,

60, 62, 62, 67, 69, 73, 76, 76, 81, 84, 86, 87, 90, 91, 92, 94, 95, 105, 105, 113, 115, 115, 119,

123, 124, 124, 125, 126, 126, 127, 129, 129, 130, 132, 135, 135, 136, 136, 137, 139, 139, 140, 145,

150, 154, 156, 162, 163, 164, 167, 167, 167, 168, 168, 170, 170, 172, 173, 177, 178, 180, 182, 182,

183, 184, 184, 186, 186, 188, 193, 193, 196, 198, 199,

Parallel (16) Merge sort: 1ms

Sorted array is =>

2, 3, 8, 11, 11, 12, 13, 14, 21, 21, 22, 26, 26, 27, 29, 29, 34, 42, 43, 46, 49, 51, 56, 57, 58, 59,

60, 62, 62, 67, 69, 73, 76, 76, 81, 84, 86, 87, 90, 91, 92, 94, 95, 105, 105, 113, 115, 115, 119,

123, 124, 124, 125, 126, 126, 127, 129, 129, 130, 132, 135, 135, 136, 136, 137, 139, 139, 140, 145,

150, 154, 156, 162, 163, 164, 167, 167, 167, 168, 168, 170, 170, 172, 173, 177, 178, 180, 182, 182,

183, 184, 184, 186, 186, 188, 193, 193, 196, 198, 199,

OUTPUT:

Generated random array of length 1000000 with elements between 0 to 1000000

Sequential Merge sort: 165ms

Parallel (16) Merge sort: 42ms

\*/

**Practical 3 : min,max,sum,avg**

#include<iostream>

#include<omp.h>

using namespace std;

int minval(int arr[], int n){

int minval = arr[0];

#pragma omp parallel for reduction(min : minval)

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

if(arr[i] < minval) minval = arr[i];

}

return minval;

}

int maxval(int arr[], int n){

int maxval = arr[0];

#pragma omp parallel for reduction(max : maxval)

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

if(arr[i] > maxval) maxval = arr[i];

}

return maxval;

}

int sum(int arr[], int n){

int sum = 0;

#pragma omp parallel for reduction(+ : sum)

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

sum += arr[i];

}

return sum;

}

int average(int arr[], int n){

return (double)sum(arr, n) / n;

}

int main(){

int n = 5;

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

cout << "The minimum value is: " << minval(arr, n) << '\n';

cout << "The maximum value is: " << maxval(arr, n) << '\n';

cout << "The summation is: " << sum(arr, n) << '\n';

cout << "The average is: " << average(arr, n) << '\n';

return 0;

}

Practical 4 :