1. MIN,MAX

#include <iostream>

#include <vector>

#include <omp.h>

#include <climits>

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);

}

1. MERGE SORT

#include<iostream>

#include<stdlib.h>

#include<omp.h>

#define MAX 100

using namespace std;

void mergesort(int a[], int l, int h);

void merge(int a[], int l, int h, int m);

void mergesort(int a[], int l, int h)

{

int mid;

if (l < h)

{

mid = (l + h) / 2;

#pragma omp parallel sections

{

#pragma omp section

{

mergesort(a, l, mid);

}

#pragma omp section

{

mergesort(a, mid + 1, h);

}

}

merge(a, l, h, mid);

}

}

void merge(int a[], int low, int high, int mid)

{

int size = (high - low) + 1;

int i, j, k, temp[MAX];

i = low; k = 0; j = mid + 1;

// Merge the two parts into temp[]

while (i <= mid && j <= high)

{

if (a[i] < a[j])

{

temp[k] = a[i];

k++;

i++;

}

else

{

temp[k] = a[j];

k++;

j++;

}

}

/\* Insert all the remaining values from i to mid into temp[] \*/

while (i <= mid)

{

temp[k] = a[i];

k++;

i++;

}

/\* Insert all the remaining values from j to high into temp[] \*/

while (j <= high)

{

temp[k] = a[j];

k++;

j++;

}

/\* Assign sorted data stored in temp[] to a[] \*/

for (i = low; i <= high; i++)

{

a[i] = temp[i - low];

}

}

int main()

{

//int\* a, n, i;

int a[20],n,i;

cout << "\n enter total no of elements=>";

cin >> n;

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

{

cin>>a[i];

}

cout << "\n Unsorted Elements=>\n";

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

{

cout << "\t" << a[i];

}

mergesort(a, 0, n - 1);

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

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

{

cout << "\t" << a[i];

}

return 0;

}

1. Bubble Sort

#include<iostream>

#include<stdlib.h>

#include<omp.h>

using namespace std;

void bubble(int \*, int);

void swap(int &, int &);

void bubble(int \*a, int n)

{

int swapped;

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

{

int first = i % 2;

swapped=0;

#pragma omp parallel for shared(a,first)

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

{

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

{

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

swapped=1;

}

}

if(swapped==0)

break;

}

}

void swap(int &a, int &b)

{

int test;

test=a;

a=b;

b=test;

}

int main()

{

int \*a,n;

cout<<"\n enter total no of elements=>";

cin>>n;

a=new int[n];

cout<<"\n enter elements=>";

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

{

cin>>a[i];

}

double start\_time = omp\_get\_wtime(); // start timer for sequential algorithm

bubble(a,n);

double end\_time = omp\_get\_wtime(); // end timer for sequential algorithm

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

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

{

cout<<a[i]<<endl;

}

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

start\_time = omp\_get\_wtime(); // start timer for parallel algorithm

bubble(a,n);

end\_time = omp\_get\_wtime(); // end timer for parallel algorithm

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

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

{

cout<<a[i]<<endl;

}

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

return 0;

}

1. DFS

#include <iostream>

#include <omp.h>

#include <chrono>

using namespace std;

using namespace std::chrono;

class Node {

public:

int data;

Node\* left;

Node\* right;

Node(int value)

{

data = value;

left = NULL;

right = NULL;

}

};

void inorder(Node\* node)

{

if (node == NULL)

{

return;

}

#pragma omp task

inorder(node->left);

cout << node->data << " ";

#pragma omp task

inorder(node->right);

}

int main()

{

Node\* root = new Node(1);

root->left = new Node(2);

root->right = new Node(3);

root->left->left = new Node(4);

root->left->right = new Node(5);

cout << "Inorder Traversal: ";

high\_resolution\_clock::time\_point start\_time = high\_resolution\_clock::now();

#pragma omp parallel

{

#pragma omp single nowait

inorder(root);

}

high\_resolution\_clock::time\_point end\_time = high\_resolution\_clock::now();

auto duration = duration\_cast<microseconds>(end\_time - start\_time).count();

cout << "\nTime taken: " << duration << " microseconds" << endl;

return 0;

}