hpc1

**dfs**

#include <iostream>

#include <vector>

#include <stack>>

#include <omp.h>

using namespace std;

const int MAX = 100000;

vector<int> graph [MAX];

bool visited[MAX];

void dfs(int node) {

stack<int> s;

s.push(node);

while (!s.empty()) {

int curr\_node = s s.top();

pop ()

if (!visited[curr\_node]) {

visited[curr\_node] = true;

if (visited[curr\_node]) {

cout << curr\_node << "";

#pragma omp parallel for

for (int i = 0 1 graph[curr\_node].size(); i++) {

int adj\_node graph[curr\_node] [1];

if (!visited[adj\_node]) {

s.push(adj\_node);

}

}

}

}

}

int main() {

int n, m, start\_node;

cout << "Enter No of Node, Edges, and start node:" ;

cin >>n>m>>start\ node

//n: node,m: edges

cout << "Enter Pair of edges:";

for (int i = 0 1 < m 1 ++) {

int u, v;

cin >>u>>v

//u and v: Pair of edges

graph[u].push\_back(v);

graph[v].push\_back(u); }

#pragma omp parallel for

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

visited[1] = false;

}

dfs(start\_node);

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

if (visited[1]) {

cout<<i<< “”;

}

}\*/

return 0;

}

cmd:

6 7 0

0 1

02

13

24

25

45

53

**Hpc 2:**

**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)

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

{ int first= i% \* 2;

#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] );

}

}

}

}

void swap(int &a, int &b)

{

int temp;

temp=a;

a=b;

b=temp;

}

int main(){

int \*a,n;

cout<<"\nEnter size of Array: ";

cin>>n;

a=new int[n];

cout<<"\nEnter elements: \n";

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

{

cin>>a[i];

}

bubble(a,n);

cout<<"\nSorted array is: \n";

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

{ cout<<a[i]<<endl;

}

return 0;

}

G++ -fopenmp bubble.cpp

./a.out

**Merge Sort**

#include<iostream>

#include<stdlib.h>

#include<omp.h>

using namespace std;

void mergesort(int a[], int i, int j);

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

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

int mid;

if (i < j)

{

mid = (i + j) / 2;

#pragma omp parallel sections {

#pragma omp section {

mergesort(a, i, mid);

}

#pragma omp section

{

mergesort(a, m+1,j);

}

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

}

}

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

{

int temp[1000];

int i, j, k;

i =i1;

j =i2;

k = 0;

cout << "\nMerging: ";

for (int x = i 1; x <=j1; x++)

{ cout << a[x] <<"”;

}

cout <<” and";

for ( int x =i2; x <=j2; j 2;x++)

{ cout<< a[x] <<"";

}

cout << endl;

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 <= j 2;i++,j++) {

a[i] =temp[j]; }

cout << "Result after merging: ";

for (int x =i1 ; x <=j2;x++) {

cout <<a[x]<< " ";

}

cout << endl;

}

int main() {

int \*a, n, i;

cout << "\nEnter size of Array: ";

cin >> n;

a= new int[n];

cout << "\nEnter elements: \n";

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

{

cin >>a[i];

}

mergesort t(a, 0, n - 1) ;

cout << "\nSorted array is : ";

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

{

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

}

return 0;

}

**Min max**

#include <iostream>

//#include <vector>

#include <omp.h>

#include <climits>

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 - 1) << endl;

}

int main() {

int \*arr, n;

cout<<”\nenter total no of elements=>";

cin>>n;

arr=new int [n] ;

cout<<"\n enter elements=>”;

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

{

cin>>arr[i];

}

// int arr[]= \{5, 2, 9, 1, 7, 6, 8, 3, 4\} ; int n = size(arr);

min\_reduction(arr, n);

max\_reduction(arr, n);

sum\_reduction(arr, n);

average\_reduction(arr, n);

}

g++ -fopenmp ass.cpp -o ac

./ac

**cuda**

!nvcc-version

nvcc: NVIDIA (R) Cuda compiler driver

Copyright (c) 2005-2022 NVIDIA Corporation Built on Wed\_Sep\_21\_10:33:58 PDT 2022

Cuda compilation tools, release 11.8, V11.8.89 Build cuda\_11.8.r11.8/compiler.31833905\_0

!pip install git+https://github.com/andreinechaev/nvcc4jupyter.git

Looking in indexes: https://pypi.org/simple, https://us-

python.pkg.dev/colab-wheels/public/simple/ Collecting git+https://github.com/andreinechaev/nvcc4jupyter.git Cloning https://github.com/andreinechaev/nvcc4jupyter.git to

/tmp/pip-req-build-czotn\_qr Running command git clone-filter=blob:none --quiet https://github.com/andreinechaev/nvcc4jupyter.git/tmp/pip-req-build-

czotn\_qr om/andreinechaev/nvcc4jupyter.git to commit

Preparing metadata (setup.py) e=NVCCPlugin-0.0.2-py3-none-

any.whl size=4287

sha256-194a071cdce43ec0da1bcd2a72836d9047a5ac7365efa00d8deb8b1bd157a2d

7

Stored in directory:

/tmp/pip-ephem-wheel-cache-00\_2ab5x/wheels/a8/b9/18/23f8ef71ceb8f63297 dd1903aedd067e6243a68ea756d6feea

Successfully built NVCCPlugin

Installing collected packages: NVCCPlugin Successfully installed NVCCPlugin-0.0.2

%load\_ext nvcc\_plugin

created output directory at /content/src

Out bin/content/result.out

VECTOR ADDITION

%%CU

#include <stdio.h>

// CUDA kernel for vector addition

\_global\_ void vectorAdd(int\* a, int b, int\* c, int size)

{

int tid blockIdx.x blockDim.x + threadIdx.x;

if (tid size) {

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

}

}

int main() {

int size = 100 // Size of the vectors int a, b, c; int\* dev\_a, dev\_b, // Host vectors dev\_c; // Device vectors

// Allocate memory for host vectors

a= (int\*)malloc(size sizeof(int)); b= (int\*)malloc(size sizeof(int));

c = (int\*)malloc(size

sizeof(int));

// Initialize host vectors for (int i = 0 i < size; i++) {

a[i] = i;

b[i] = 2i

}

// Allocate memory on the device for device vectors cudaMalloc((void\*\*)&dev\_a, size sizeof(int)); cudaMalloc((void\*\*)&dev\_b, size sizeof(int)); cudaMalloc((void\*\*)&dev\_c, size sizeof(int));

// Copy host vectors to device cudaMemcpy(dev\_a, a, size sizeof(int), cudaMemcpyHostToDevice); cudaMemcpy(dev\_b, b, size sizeof(int), cudaMemcpyHostToDevice);

// Launch kernel for vector addition

int blockSize = 256;

int gridSize = (size + blockSize - 1) / blockSize; vectorAdd<<<gridSize, blockSize>>>(dev\_a, dev\_b, dev\_c, size);

// Copy result from device to host cudaMemcpy(c, dev\_c, size sizeof(int), cudaMemcpyDeviceToHost);

// Print result

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

}

printf("%d + d = \d\n", a[i], b[i] , c[i] ) ;

// Free device memory

cudaFree (dev\_a);

cudaFree (dev\_b);

cudaFree (dev\_c);

// Free host memory

free(a);

free(b);

free(c);

return 0;

}