Assignment No 1(B)

Aim : Design and implement Parallel Depth First Search based on existing algorithms using OpenMP. Use a Tree or an undirected graph for 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.top();

s.pop();

if (!visited[curr\_node]) {

visited[curr\_node] = true;

if (visited[curr\_node]) {

cout << curr\_node << " ";

}

#pragma omp parallel for

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

int adj\_node = graph[curr\_node][i];

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; i < m; i++) {

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[i] = false;

}

dfs(start\_node);

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

if (visited[i]) {

cout << i << " ";

}

}\*/

return 0;}

Assignment NO: 1(A)

Aim : Design and implement Parallel Breadth First Search and Depth First Search based on existing algorithms using OpenMP. Use a Tree or an undirected graph for BFS and DFS .

#include<iostream>

#include<stdlib.h>

#include<queue>

using namespace std;

class node

{

public:

node \*left, \*right;

int data;

};

class Breadthfs

{

public:

node \*insert(node \*, int);

void bfs(node \*);

};

node \*insert(node \*root, int data)

// inserts a node in tree

{

if(!root)

{ root=new node;

root->left=NULL;

root->right=NULL;

root->data=data;

return root;

} queue<node \*> q;

q.push(root);

while(!q.empty())

{

node \*temp=q.front();

q.pop();

if(temp->left==NULL)

{

temp->left=new node;

temp->left->left=NULL;

temp->left->right=NULL;

temp->left->data=data;

return root;

}

else

{

q.push(temp->left);

}

if(temp->right==NULL) {

temp->right=new node;

temp->right->left=NULL;

temp->right->right=NULL;

temp->right->data=data;

return root;

}

else

{

q.push(temp->right);

}

}

}

void bfs(node \*head)

{

queue<node\*> q;

q.push(head);

int qSize;

while (!q.empty())

{

qSize = q.size();

#pragma omp parallel for

//creates parallel threads

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

{

node\* currNode;

#pragma omp critical

{

currNode = q.front();

q.pop();

cout<<"\t"<<currNode->data;

}// prints parent node

#pragma omp critical

{

if(currNode->left)// push parent's left node in queue

q.push(currNode->left);

if(currNode->right)

q.push(currNode->right);

}// push parent's right node in queue

}

}

}

int main(){

node \*root=NULL;

int data;

char ans;

do

{

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

cin>>data;

root=insert(root,data);

cout<<"do you want insert one more node?";

cin>>ans;

}while(ans=='y'||ans=='Y');

bfs(root);

return 0;

}

Run Commands:

1. g++ -fopenmp bfs.cpp -o bfs
2. ./bfs

**Assignment 2 (A)**

**Title:** Write a program to implement Parallel Bubble Sort using OpenMP. Use existing algorithms and measure the performance of sequential and parallel algorithms.

#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;

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

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);

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

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);

}

**Assignment 2 (B)**

**Title:** Write a program to implement Parallel Merge Sort using OpenMP. Use existing algorithms and measure the performance of sequential and parallel algorithms.

#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;

}

**Grp A\_Assignment 3**

**Implement Min, Max, Sum and Average operations using Parallel Reduction.**

#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<<"\n enter 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);}

Assignment 4

Aim: Write a CUDA Program for : 1. Addition of two large vectors.

#include<stdio.h>

#include<iostream>

#include<cstdlib>

//\*\*\*\*important to add following library to allow a programmer to use parallel paradigms\*\*\*\*\*

#include<omp.h>

using namespace std;

#define MAX 5

int main()

{

int a[MAX],b[MAX],c[MAX],i;

printf("\n First Vector:\t");

//Instruct a master thread to fork and generate more threads to process following loop structure

#pragma omp parallel for for(i=0;i<MAX;i++)

{

a[i]=rand()%1000;

}

//Discuss issue of this for loop below-if we make it parallel, possibly values that get printed will not be in sequence as we dont have any control on order of threads execution for(i=0;i<MAX;i++)

{

printf("%d\t",a[i]);

}

printf("\n Second Vector:\t"); #pragma omp parallel for for(i=0;i<MAX;i++)

{

b[i]=rand()%1000;

}

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

{

printf("%d\t",b[i]);

}

printf("\n Parallel-Vector Addition:(a,b,c)\t");

#pragma omp parallel for

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

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

}

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

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

}

}

4(B)

%%cu

#include <iostream>

using namespace std;

// CUDA code to multiply matrices

\_\_global\_\_ void multiply(int\* A, int\* B, int\* C, int size) {

// Uses thread indices and block indices to compute each element

int row = blockIdx.y \* blockDim.y + threadIdx.y;

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

if (row < size && col < size) {

int sum = 0;

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

sum += A[row \* size + i] \* B[i \* size + col];

}

C[row \* size + col] = sum;

}

}

void initialize(int\* matrix, int size) {

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

matrix[i] = rand() % 10;

}

}

void print(int\* matrix, int size) {

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

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

cout << matrix[row \* size + col] << " ";

}

cout << '\n';

}

cout << '\n';

}

int main() {

int\* A, \* B, \* C;

int N = 2;

int blockSize = 16;

int matrixSize = N \* N;

size\_t matrixBytes = matrixSize \* sizeof(int);

A = new int[matrixSize];

B = new int[matrixSize];

C = new int[matrixSize];

initialize(A, N);

initialize(B, N);

cout << "Matrix A: \n";

print(A, N);

cout << "Matrix B: \n";

print(B, N);

int\* X, \* Y, \* Z;

// Allocate space

cudaMalloc(&X, matrixBytes);

cudaMalloc(&Y, matrixBytes);

cudaMalloc(&Z, matrixBytes);

// Copy values from A to X

cudaMemcpy(X, A, matrixBytes, cudaMemcpyHostToDevice);

// Copy values from A to X and B to Y

cudaMemcpy(Y, B, matrixBytes, cudaMemcpyHostToDevice);

// Threads per CTA dimension

int THREADS = 2;

// Blocks per grid dimension (assumes THREADS divides N evenly)

int BLOCKS = N / THREADS;

// Use dim3 structs for block and grid dimensions

dim3 threads(THREADS, THREADS);

dim3 blocks(BLOCKS, BLOCKS);

// Launch kernel

multiply<<<blocks, threads>>>(X, Y, Z, N);

cudaMemcpy(C, Z, matrixBytes, cudaMemcpyDeviceToHost);

cout << "Multiplication of matrix A and B: \n";

print(C, N);

delete[] A;

delete[] B;

delete[] C;

cudaFree(X);

cudaFree(Y);

cudaFree(Z);

return 0;

}