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 .

itle: Parallel Depth First Search based on existing algorithms using OpenMP

#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;

}

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

**Title: Write a program to implement Parallel Bubble Sort using OpenMP**

#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 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];

    }

    bubble(a,n);

    cout<<"\n sorted array is=>";

    for(int i=0;i<n;i++)

    {

    cout<<a[i]<<endl;

    }

return 0;

}

**Merge Sort:**

**Title: Write a program to implement Parallel Bubble Sort using OpenMP**

#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 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];

    }

    bubble(a,n);

    cout<<"\n sorted array is=>";

    for(int i=0;i<n;i++)

    {

    cout<<a[i]<<endl;

    }

return 0;

}

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

**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);

}

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

**2. Matrix Multiplication using CUDA C**

**1.**

#include <iostream>

#include <cuda\_runtime.h>

#include /usr/local/cuda/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;

// Copy data from device to host

cudaMemcpy(C, dev\_C, size, cudaMemcpyDeviceToHost);

// Print the results

for (int i = 0; i < 10; i++)

{

     cout << C[i] << " ";

}

cout << endl;

// Free memory

cudaFree(dev\_A);

cudaFree(dev\_B);

cudaFree(dev\_C);

cudaFreeHost(A);

cudaFreeHost(B);

cudaFreeHost(C);

return 0;

}

**2.**

#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

// 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;