ASSIGNMENT NO: 01

BFS

#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;

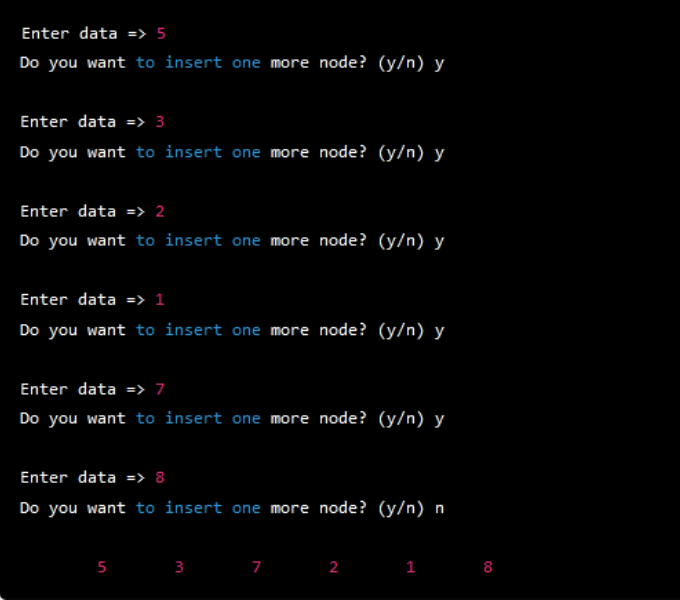
}

 Run Commands:

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

1. ./bfs

Output:



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 << " ";

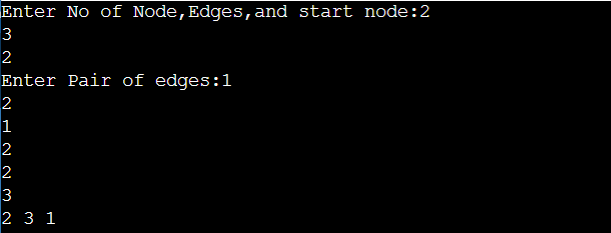
     }

}\*/

return 0;

}

OUTPUT:



ASSIGNMENT NO :03

**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;

}

OUTPUT:



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,mid+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;

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];

}

}

int main()

{

int \*a,n,i;

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

cin>>n;

a= new int[n];

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

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

{

     cin>>a[i];

}

   // start=.......

//#pragma omp…..

mergesort(a, 0, n-1);

//          stop…….

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

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

{

     cout<<"\n"<<a[i];

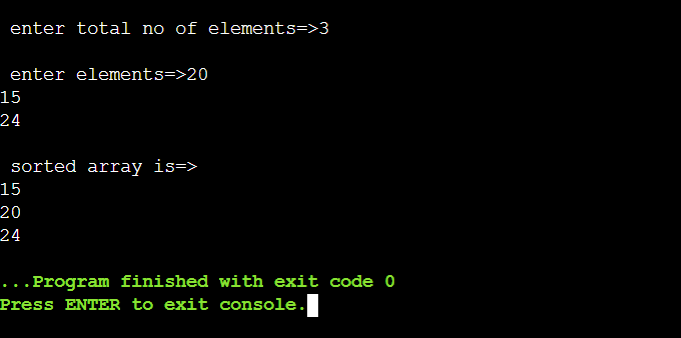
}

   // Cout<<Stop-Start

return 0;

}

OUTPUT:



**ASSIGNMENT No :03**

#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);

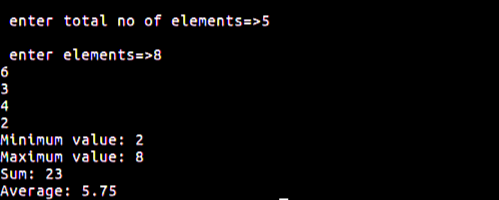
  max\_reduction(arr, n);

  sum\_reduction(arr, n);

  average\_reduction(arr, n);

}

Output:



ASSIGNMENT NO: 04

 Addition of two large vectors

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

// CUDA kernel. Each thread takes care of one element of c

\_\_global\_\_ void vecAdd(double \*a, double \*b, double \*c, int n)

{

    // Get our global thread ID

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

    // Make sure we do not go out of bounds

    if (id < n)

        c[id] = a[id] + b[id];

}

int main( int argc, char\* argv[] )

{

    // Size of vectors

    int n = 100000;

    // Host input vectors

    double \*h\_a;

    double \*h\_b;

    //Host output vector

    double \*h\_c;

    // Device input vectors

    double \*d\_a;

    double \*d\_b;

    //Device output vector

    double \*d\_c;

    // Size, in bytes, of each vector

    size\_t bytes = n\*sizeof(double);

    // Allocate memory for each vector on host

    h\_a = (double\*)malloc(bytes);

    h\_b = (double\*)malloc(bytes);

    h\_c = (double\*)malloc(bytes);

    // Allocate memory for each vector on GPU

    cudaMalloc(&d\_a, bytes);

    cudaMalloc(&d\_b, bytes);

    cudaMalloc(&d\_c, bytes);

    int i;

    // Initialize vectors on host

    for( i = 0; i < n; i++ ) {

        h\_a[i] = sin(i)\*sin(i);

        h\_b[i] = cos(i)\*cos(i);

    }

    // Copy host vectors to device

    cudaMemcpy( d\_a, h\_a, bytes, cudaMemcpyHostToDevice);

    cudaMemcpy( d\_b, h\_b, bytes, cudaMemcpyHostToDevice);

    int blockSize, gridSize;

    // Number of threads in each thread block

    blockSize = 1024;

    // Number of thread blocks in grid

    gridSize = (int)ceil((float)n/blockSize);

    // Execute the kernel

    vecAdd<<<gridSize, blockSize>>>(d\_a, d\_b, d\_c, n);

    // Copy array back to host

    cudaMemcpy( h\_c, d\_c, bytes, cudaMemcpyDeviceToHost );

    // Sum up vector c and print result divided by n, this should equal 1 within error

    double sum = 0;

    for(i=0; i<n; i++)

        sum += h\_c[i];

    printf("final result: %f\n", sum/n);

    // Release device memory

    cudaFree(d\_a);

    cudaFree(d\_b);

    cudaFree(d\_c);

    // Release host memory

    free(h\_a);

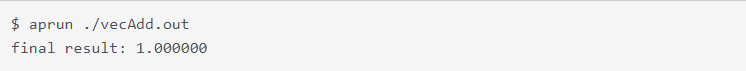
    free(h\_b);

    free(h\_c);

    return 0;

}

OUTPUT:



Matrix Multiplication using CUDA C

dev\_array.h

#ifndef \_DEV\_ARRAY\_H\_

#define \_DEV\_ARRAY\_H\_

#include <stdexcept>

#include <algorithm>

#include <cuda\_runtime.h>

template <class T>

class dev\_array

{

// public functions

public:

explicit dev\_array()

: start\_(0),

end\_(0)

{}

// constructor

explicit dev\_array(size\_t size)

{

allocate(size);

}

// destructor

~dev\_array()

{

free();

}

// resize the vector

void resize(size\_t size)

{

free();

allocate(size);

}

// get the size of the array

size\_t getSize() const

{

return end\_ - start\_;

}

// get data

const T\* getData() const

{

return start\_;

}

T\* getData()

{

return start\_;

}

// set

void set(const T\* src, size\_t size)

{

size\_t min = std::min(size, getSize());

cudaError\_t result = cudaMemcpy(start\_, src, min \* sizeof(T), cudaMemcpyHostToDevice);

if (result != cudaSuccess)

{

throw std::runtime\_error("failed to copy to device memory");

}

}

// get

void get(T\* dest, size\_t size)

{

size\_t min = std::min(size, getSize());

cudaError\_t result = cudaMemcpy(dest, start\_, min \* sizeof(T), cudaMemcpyDeviceToHost);

if (result != cudaSuccess)

{

throw std::runtime\_error("failed to copy to host memory");

}

}

// private functions

private:

// allocate memory on the device

void allocate(size\_t size)

{

cudaError\_t result = cudaMalloc((void\*\*)&start\_, size \* sizeof(T));

if (result != cudaSuccess)

{

start\_ = end\_ = 0;

throw std::runtime\_error("failed to allocate device memory");

}

end\_ = start\_ + size;

}

// free memory on the device

void free()

{

if (start\_ != 0)

{

cudaFree(start\_);

start\_ = end\_ = 0;

}

}

T\* start\_;

T\* end\_;

};

#endif

matrixmul.cu

#include <iostream>

#include <vector>

#include <stdlib.h>

#include <time.h>

#include <cuda\_runtime.h>

#include "kernel.h"

#include "kernel.cu"

#include "dev\_array.h"

#include <math.h>

using namespace std;

int main()

{

// Perform matrix multiplication C = A\*B

// where A, B and C are NxN matrices

int N = 16;

int SIZE = N\*N;

// Allocate memory on the host

vector<float> h\_A(SIZE);

vector<float> h\_B(SIZE);

vector<float> h\_C(SIZE);

// Initialize matrices on the host

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

for (int j=0; j<N; j++){

h\_A[i\*N+j] = sin(i);

h\_B[i\*N+j] = cos(j);

}

}

// Allocate memory on the device

dev\_array<float> d\_A(SIZE);

dev\_array<float> d\_B(SIZE);

dev\_array<float> d\_C(SIZE);

d\_A.set(&h\_A[0], SIZE);

d\_B.set(&h\_B[0], SIZE);

matrixMultiplication(d\_A.getData(), d\_B.getData(), d\_C.getData(), N);

cudaDeviceSynchronize();

d\_C.get(&h\_C[0], SIZE);

cudaDeviceSynchronize();

float \*cpu\_C;

cpu\_C=new float[SIZE];

// Now do the matrix multiplication on the CPU

float sum;

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

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

sum = 0.f;

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

sum += h\_A[row\*N+n]\*h\_B[n\*N+col];

}

cpu\_C[row\*N+col] = sum;

}

}

double err = 0;

// Check the result and make sure it is correct

for (int ROW=0; ROW < N; ROW++){

for (int COL=0; COL < N; COL++){

err += cpu\_C[ROW \* N + COL] - h\_C[ROW \* N + COL];

}

}

cout << "Error: " << err << endl;

return 0;

}

kernel.h

#ifndef KERNEL\_CUH\_

#define KERNEL\_CUH\_

void matrixMultiplication(float \*A, float \*B, float \*C, int N);

#endif

kernel.cu

#include <math.h>

#include <iostream>

#include "cuda\_runtime.h"

#include "kernel.h"

#include <stdlib.h>

using namespace std;

\_\_global\_\_ void matrixMultiplicationKernel(float\* A, float\* B, float\* C, int N) {

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

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

float tmpSum = 0;

if (ROW < N && COL < N) {

// each thread computes one element of the block sub-matrix

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

tmpSum += A[ROW \* N + i] \* B[i \* N + COL];

}

}

C[ROW \* N + COL] = tmpSum;

}

void matrixMultiplication(float \*A, float \*B, float \*C, int N){

// declare the number of blocks per grid and the number of threads per block

// use 1 to 512 threads per block

dim3 threadsPerBlock(N, N);

dim3 blocksPerGrid(1, 1);

if (N\*N > 512){

threadsPerBlock.x = 512;

threadsPerBlock.y = 512;

blocksPerGrid.x = ceil(double(N)/double(threadsPerBlock.x));

blocksPerGrid.y = ceil(double(N)/double(threadsPerBlock.y));

}

matrixMultiplicationKernel<<<blocksPerGrid,threadsPerBlock>>>(A, B, C, N);

}