Practical 1 :

Breadth First search Program Code :

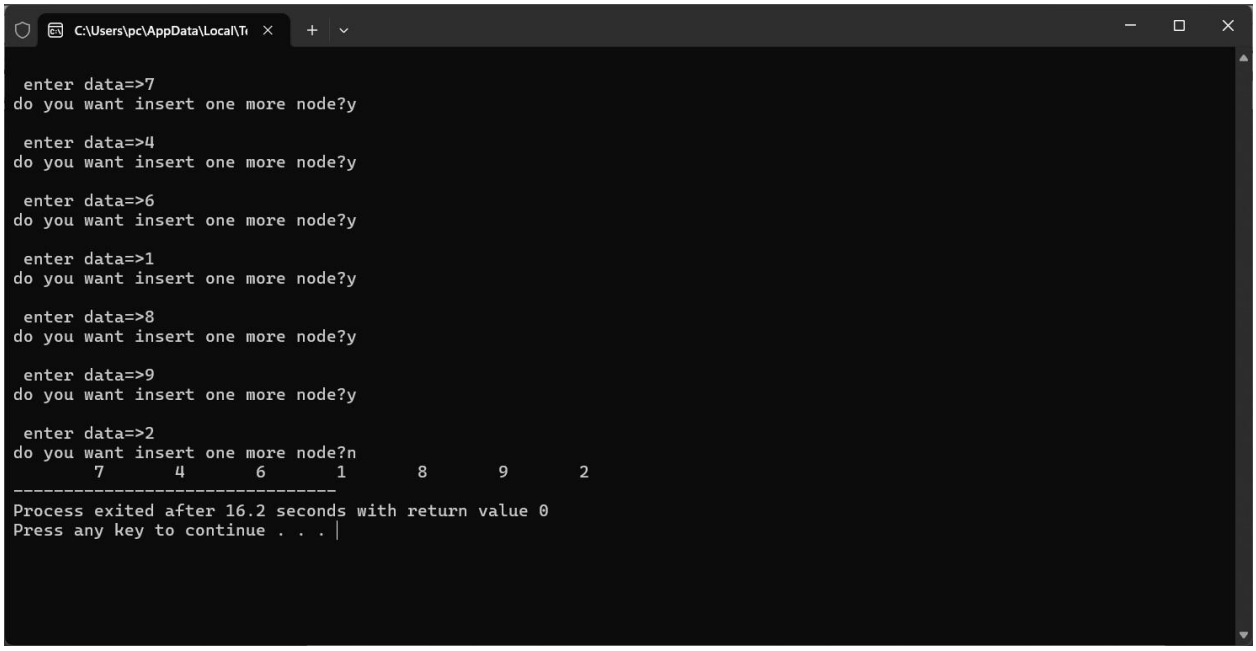
```
#include<iostream>
#include<stdlib.h>
#include<queue> using
namespace std;
 class node
{
public:
 node *left, *right;         int
data;
 };
 class Breadthfs
{
 public:
node *insert(node *, int);  void
bfs(node *);
};
node *insert(node *root, int data)
// inserts a node in tree
{ if(!root)
{
 root=new 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;
}
```

Output :

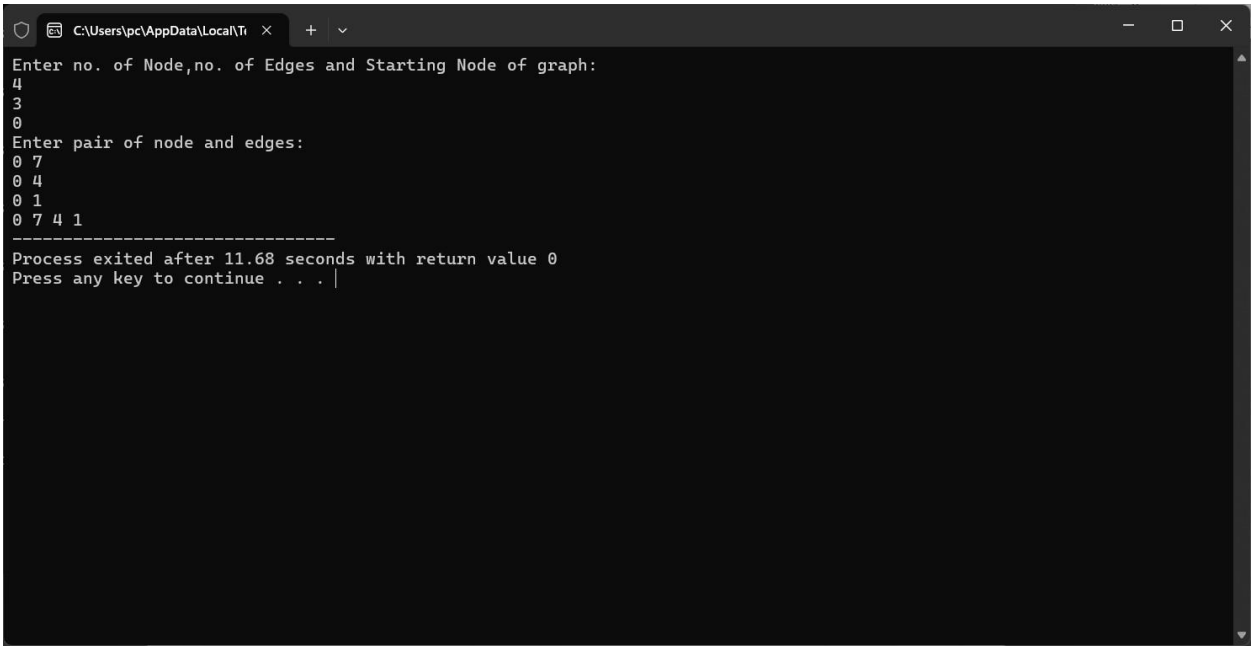

C:\Users\pc\AppData\Local\T
enter data=>7
do you want insert one more node?y
enter data=>4
do you want insert one more node?y
enter data=>6
do you want insert one more node?y
enter data=>1
do you want insert one more node?y
enter data=>8
do you want insert one more node?y
enter data=>9
do you want insert one more node?y
enter data=>2
do you want insert one more node?n
7 4 6 1 8 9 2
Process exited after 16.2 seconds with return value 0
Press any key to continue . . .

Program For Depth First Search :

```
#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();

        if (!visited[curr_node]) {
visited[curr_node] = true;
                        s.pop();
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,no. of Edges and Starting Node of graph:\n";
cin >> n >> m >> start_node;
        //n: node,m:edges        cout<<"Enter pair
of node and edges:\n";

            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);
return 0;
}
```

Output :-


C:\Users\pc\AppData\Local\T
Enter no. of Node,no. of Edges and Starting Node of graph:
4
3
0
Enter pair of node and edges:
0 7
0 4
0 1
0 7 4 1

Process exited after 11.68 seconds with return value 0
Press any key to continue . . .

Practical 2 :

Program Code For 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;
}
```

Output For Bubble Sort :

```
 sorted array is=>12
12
23
24
42
51
54
56
75
84
Time taken by sequential algorithm: 0.0149999 seconds

 sorted array is=>12
12
23
24
42
51
54
56
75
84
Time taken by parallel algorithm: 0 seconds

--------------------------------
Process exited after 10.57 seconds with return value 0
Press any key to continue . . .
```

Program for 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;
    double start_time, end_time, seq_time, par_time;
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];
    }

    // Sequential algorithm     start_time =
omp_get_wtime();    mergesort(a, 0, n-1);    end_time
= omp_get_wtime();    seq_time = end_time -
```

```
start_time;     cout << "\nSequential Time: " <<
seq_time << endl;

   // Parallel algorithm
start_time = omp_get_wtime();
   #pragma omp parallel
   {
      #pragma omp single
      {
         mergesort(a, 0, n-1);
      }
   }
   end_time = omp_get_wtime();     par_time =
end_time - start_time;     cout << "\nParallel Time:
" << par_time << endl;

   cout<<"\n sorted array is=>";
for(i=0;i<n;i++)
   {
      cout<<"\n"<<a[i];
   }
   return 0;
}
```

Output for Merge Sort :

```
 enter total no of elements=>15

 enter elements=>21
52
58
564
48456
6548
1234
457
12
548
456
774
548
654
245

Sequential Time: 0

Parallel Time: 0

 sorted array is=>
12
21
52
58
245
456
457
548
548
564
654
774
1234
6548
48456
--------------------------------
Process exited after 22.38 seconds with return value 0
```

Practical 3 :

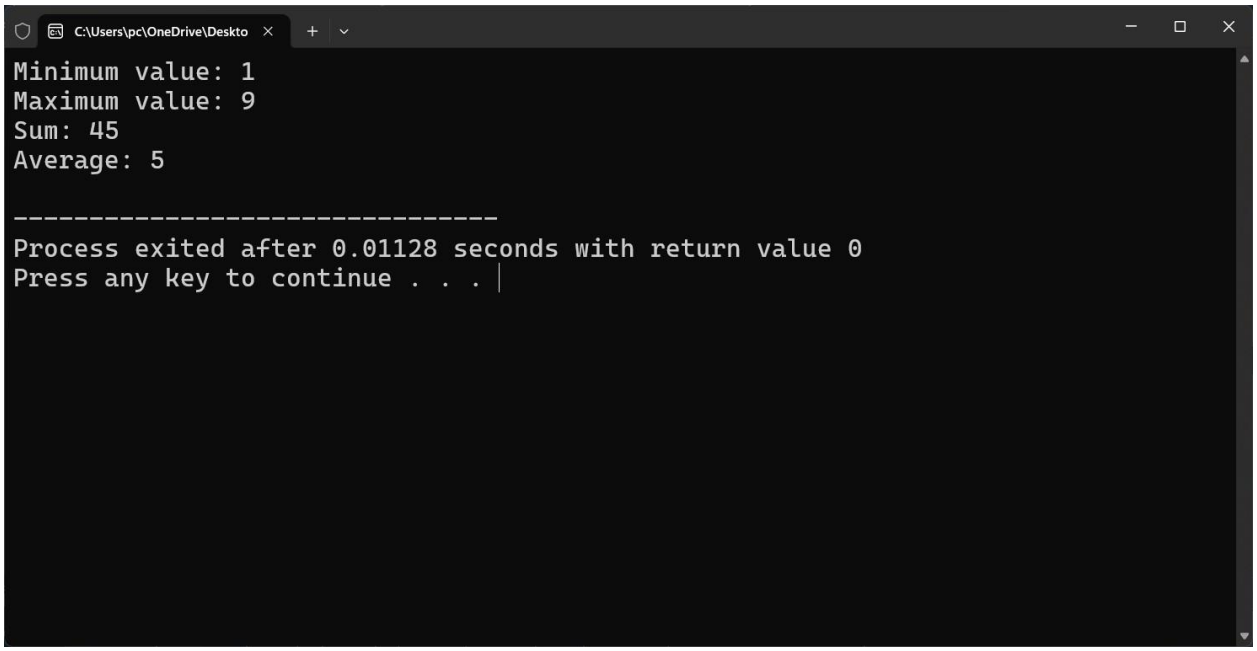
Program for Parallel Reduction :

```
#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;
arr.push_back(5);
arr.push_back(2);
arr.push_back(9);
arr.push_back(1);
arr.push_back(7);
arr.push_back(6);
arr.push_back(8);
```

```
arr.push_back(3);

arr.push_back(4);


  min_reduction(arr);   max_reduction(arr);

sum_reduction(arr);

average_reduction(arr);

}
```

Output :



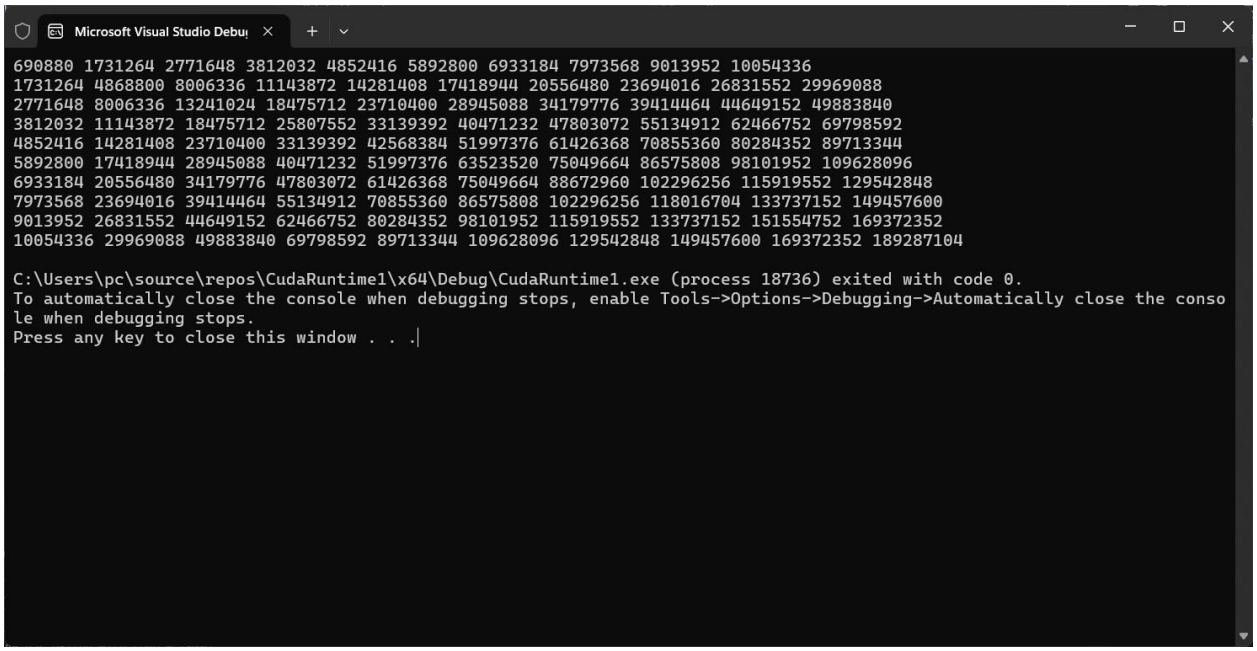
Practical 04 :

Program for Matrix multiplication using CUDA :

```
#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 = 128;
    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, size, cudaMemcpyDeviceToHost);

    // 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;
}
```

Output :



Program for Vector Addition using CUDA :

```
#include <iostream>
#include <cuda_runtime.h>

using namespace std;

__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;
    addVectors << <numBlocks, blockSize >> > (dev_A, dev_B, dev_C, n);

    // 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;
}
```

Output :