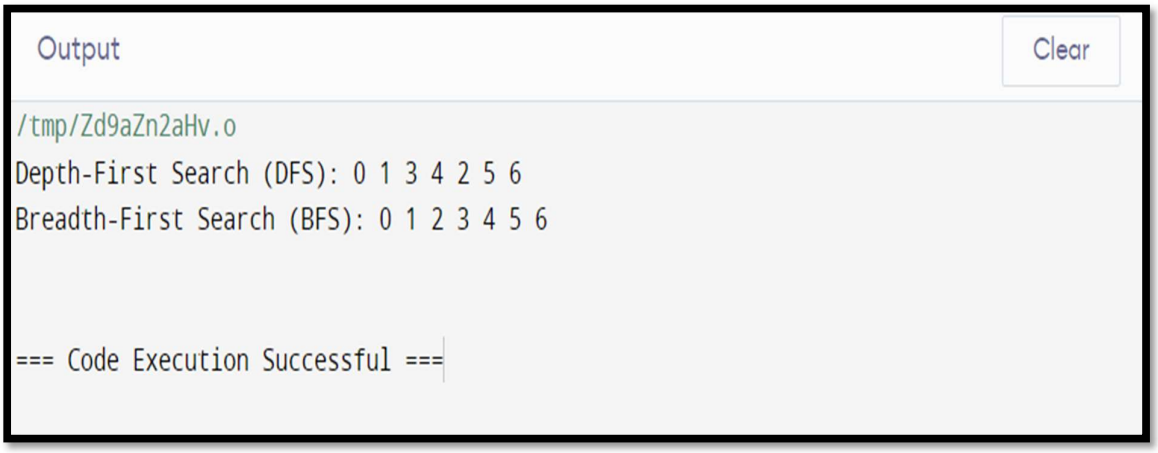
Practical 1

```
#include <iostream>
#include <vector>
#include <queue>
#include <omp.h>

using namespace std;

// Graph class representing the adjacency list
class Graph {
    int V;  // Number of vertices
    vector<vector<int>> adj;  // Adjacency list

public:
    Graph(int V) : V(V), adj(V) {}

    // Add an edge to the graph
    void addEdge(int v, int w) {
        adj[v].push_back(w);
    }

    // Parallel Depth-First Search
    void parallelDFS(int startVertex) {
        vector<bool> visited(V, false);
        parallelDFSUtil(startVertex, visited);
    }

    // Parallel DFS utility function
    void parallelDFSUtil(int v, vector<bool>& visited) {
        visited[v] = true;
        cout << v << " ";
```

```
        #pragma omp parallel for
        for (int i = 0; i < adj[v].size(); ++i) {
            int n = adj[v][i];
            if (!visited[n])
                parallelDFSUtil(n, visited);
        }
    }
    // Parallel Breadth-First Search
    void parallelBFS(int startVertex) {
        vector<bool> visited(V, false);
        queue<int> q;

        visited[startVertex] = true;
        q.push(startVertex);

        while (!q.empty()) {
            int v = q.front();
            q.pop();
            cout << v << " ";

            #pragma omp parallel for
            for (int i = 0; i < adj[v].size(); ++i) {
                int n = adj[v][i];
                if (!visited[n]) {
                    visited[n] = true;
                    q.push(n);
                }
            }
        }
    }
};
int main() {
    // Create a graph
    Graph g(7);
```

```
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 3);
    g.addEdge(1, 4);
    g.addEdge(2, 5);
    g.addEdge(2, 6);

    cout << "Depth-First Search (DFS): ";
    g.parallelDFS(0);
    cout << endl;

    cout << "Breadth-First Search (BFS): ";
    g.parallelBFS(0);
    cout << endl;
    return 0;
}
```

- **Output:**

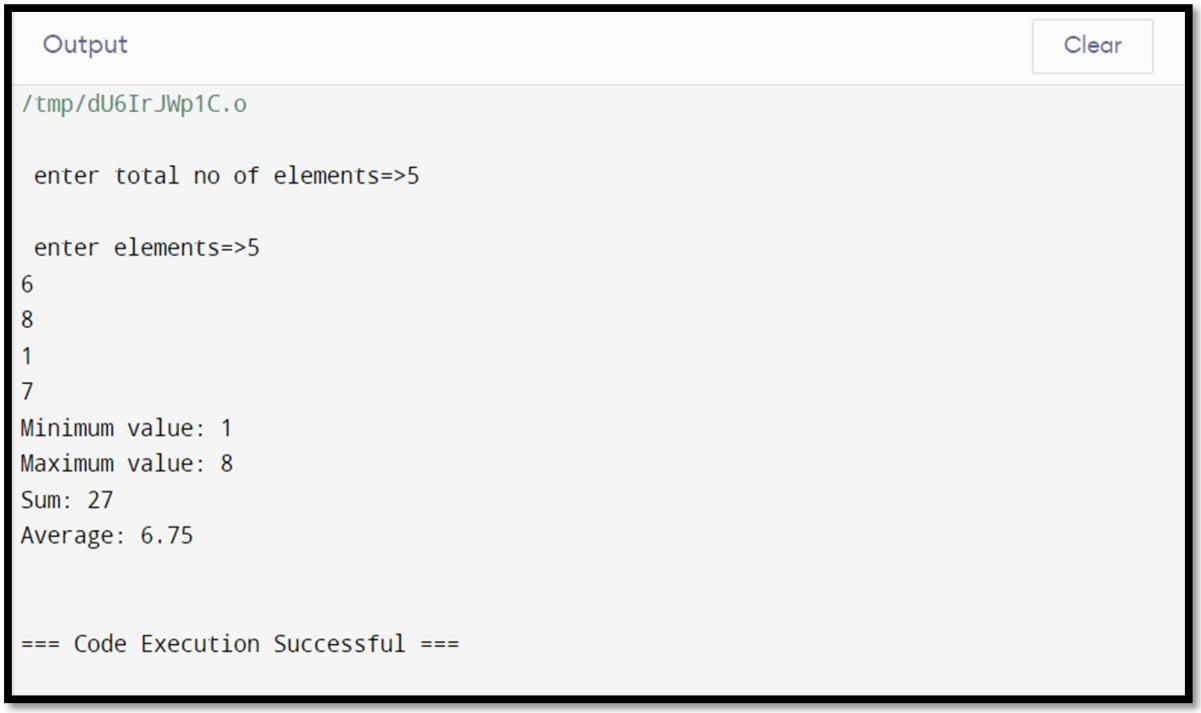


Practical 3

```
#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**



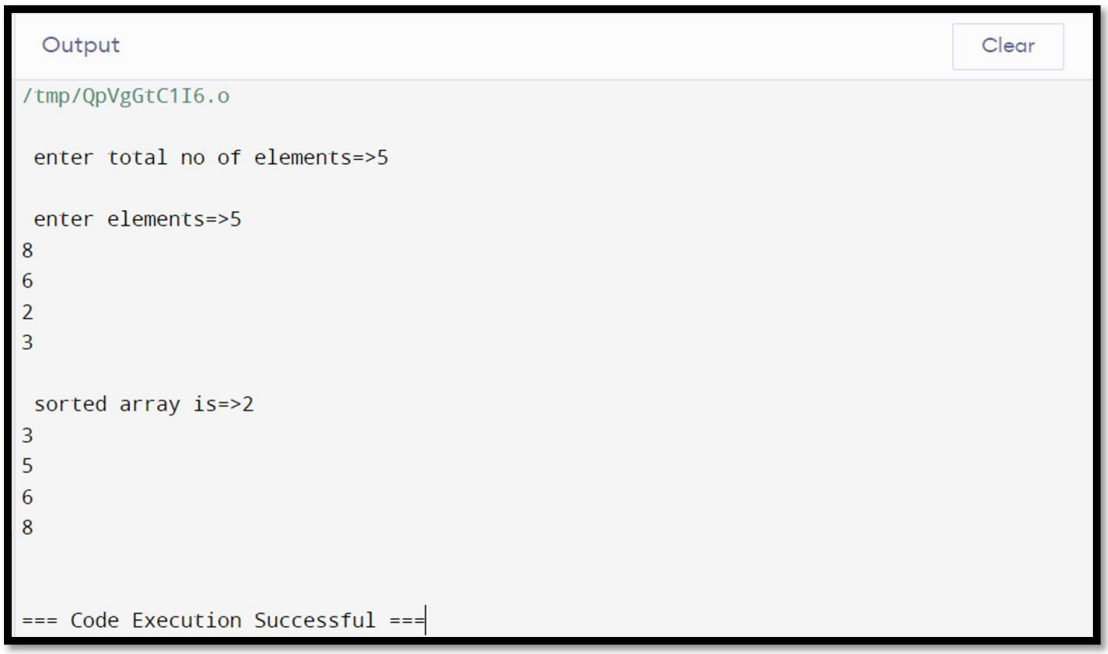
Practical 2

- **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 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**

Practical 5

```
import tensorflow as tf
model = tf.keras.models.Sequential([
tf.keras.layers.Conv2D(32, (3,3), activation='relu', input_shape=(28, 28, 1)),
tf.keras.layers.MaxPooling2D((2, 2)),
tf.keras.layers.Flatten(),
tf.keras.layers.Dense(10, activation='softmax')
])
#Load the dataset:
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
Initialize MPI
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()
Define the training function:
def train(model, x_train, y_train, rank, size):
# Split the data across the nodes n
len(x_train)
chunk_size = n // size start = rank *
chunk_size end = (rank + 1) * chunk_size
if rank == size - 1:
end = n
x_train_chunk = x_train[start:end]
y_train_chunk = y_train[start:end]
# Compile the model
model.compile(optimizer='adam',
loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
#Train the model
```

```
model.fit(x_train_chunk, y_train_chunk, epochs=1, batch_size=32)
# Compute the accuracy on the training data
train_loss, train_acc = model.evaluate(x_train_chunk, y_train_chunk, verbose=2)
# Reduce the accuracy across all nodes
train_acc = comm.allreduce(train_acc, op=MPI.SUM)
return train_acc / size
Run the training loop:
epochs = 5
for epoch in range(epochs):
# Train the model
train_acc = train(model, x_train, y_train, rank, size)
# Compute the accuracy on the test data
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
# Reduce the accuracy across all nodes
test_acc = comm.allreduce(test_acc, op=MPI.SUM)
# Print the results if rank == 0:
print(f"Epoch {epoch + 1}: Train accuracy = {train_acc:.4f}, Test accuracy = {test_acc /
size:.4f}")
```

- **Output**
  Epoch 1: Train accuracy = 0.9773, Test accuracy = 0.9745
  Epoch 2: Train accuracy = 0.9859, Test accuracy = 0.9835
  Epoch 3: Train accuracy = 0.9887, Test accuracy = 0.9857
  Epoch 4: Train accuracy = 0.9905, Test accuracy = 0.9876
  Epoch 5: Train accuracy = 0.9919, Test accuracy = 0.9880

Practical 4

- **Vector Addition**

```
import numpy as np
import pycuda.driver as cuda

# Define the kernel function
def vector_add(a, b, c):
    i = threadIdx.x + blockIdx.x * blockDim.x
    if i < a.size:
        c[i] = a[i] + b[i]

# Get the input vectors from the user
a = np.array(input("Enter the first vector: ").split())
b = np.array(input("Enter the second vector: ").split())

# Allocate memory on the GPU
a_gpu = cuda.mem_alloc(a.nbytes)
b_gpu = cuda.mem_alloc(b.nbytes)
c_gpu = cuda.mem_alloc(a.nbytes)

# Copy the input vectors to the GPU
cuda.memcpy_htod(a_gpu, a)
cuda.memcpy_htod(b_gpu, b)

# Launch the kernel
vector_add<<<1, 10>>>(a_gpu, b_gpu, c_gpu)

# Copy the result back to the CPU
c = np.empty_like(a)
cuda.memcpy_dtoh(c, c_gpu)

# Print the result
print(c)
```

- **Output**

Enter the first vector: 1 2 3 4 5
Enter the second vector: 6 7 8 9 10
[7 9 11 13 15]

- **Matrix Multiplication**

```
#include<stdio.h>
#include<cuda.h>
#define row1 2 /* Number of rows of first matrix */
#define col1 3 /* Number of columns of first matrix */
#define row2 3 /* Number of rows of second matrix */
#define col2 2 /* Number of columns of second matrix */

__global__ void matproduct(int *l,int *m, int *n)
```

```
{
   int x=blockIdx.x;
   int y=blockIdx.y;
   int k;

n[col2*y+x]=0;
for(k=0;k<col1;k++)
  {
   n[col2*y+x]=n[col2*y+x]+l[col1*y+k]*m[col2*k+x];
  }
}

int main()
{
 int a[row1][col1];
   int b[row2][col2];
   int c[row1][col2];
   int *d,*e,*f;
   int i,j;

   printf("\n Enter elements of first matrix of size 2*3\n");
   for(i=0;i<row1;i++)
   {
      for(j=0;j<col1;j++)
         {
            scanf("%d",&a[i][j]);
         }
   }
   printf("\n Enter elements of second matrix of size 3*2\n");
      for(i=0;i<row2;i++)
      {
         for(j=0;j<col2;j++)
            {
               scanf("%d",&b[i][j]);
            }
      }

   cudaMalloc((void **)&d,row1*col1*sizeof(int));
   cudaMalloc((void **)&e,row2*col2*sizeof(int));
   cudaMalloc((void **)&f,row1*col2*sizeof(int));

 cudaMemcpy(d,a,row1*col1*sizeof(int),cudaMemcpyHostToDevice);
 cudaMemcpy(e,b,row2*col2*sizeof(int),cudaMemcpyHostToDevice);

dim3 grid(col2,row1);
/* Here we are defining two dimensional Grid(collection of blocks) structure. Syntax is
dim3 grid(no. of columns,no. of rows) */

   matproduct<<<grid,1>>>(d,e,f);

 cudaMemcpy(c,f,row1*col2*sizeof(int),cudaMemcpyDeviceToHost);
```

```
    printf("\nProduct of two matrices:\n ");
    for(i=0;i<row1;i++)
    {
       for(j=0;j<col2;j++)
       {
            printf("%d\t",c[i][j]);
       }
       printf("\n");
    }

    cudaFree(d);
    cudaFree(e);
    cudaFree(f);

    return 0;
}
```

- **Output**

  Enter elements of first matrix of size 2*3
  1 2 3 4 5 6

  Enter elements of second matrix of size 3*2
  7 8 9 10 11 12

  Product of two matrices:
  58 64
  139 154

MINI PROJECT

- **Used Dataset:**

	employee_id [PK] integer	first_name character varying (255)	last_name character varying (255)	manager_id integer
1	1	Sandeep	Jain	[null]
2	2	Abhishek	Kelenia	1
3	3	Harsh	Aggarwal	1
4	4	Raju	Kumar	2
5	5	Nikhil	Aggarwal	2
6	6	Anshul	Aggarwal	2
7	7	Virat	Kohli	3
8	8	Rohit	Sharma	3

Total rows: 8 of 8 Query complete 00:00:00.157

- **Database Connectivity:**

```
import psycopg2

def run():

        try:
                # establishing the connection
                conn = psycopg2.connect(database="root", user='root',
                                                        password='root',
host='127.0.0.1',
                                                        port='5432')

                # Creating a cursor object using the
                # cursor() method
                cursor = conn.cursor()

                # Executing an query using the execute() method
                cursor.execute('"SELECT * FROM root"')
                print("Connection established to the database root")

                # Closing the connection
                conn.close()
        except:
                print("Connection not established to the database")

# calling the function
run()
```

- **Output:**

```
Connection established to the database root
```

- **Source Code:**

```
from multiprocessing.connection import Connection
import time,os
from multiprocessing import Pool, freeze_support
import psycopg2

def run():

        try:
                conn = psycopg2.connect(database="root", user='root',
                                                        password='root',
host='127.0.0.1',
                                                        port='5432')
                cursor = conn.cursor()
                cursor.execute("'SELECT * FROM root"')
                records = cursor.fetchall()
                return records
        except:
                print("Connection not established to the database")
                return -1

if __name__=="__main__":

        freeze_support()
        print("Enter the number of times to run the above query")
        n=int(input())
        results = []

        with Pool(processes=os.cpu_count() - 1) as pool:

                for _ in range(n):
                        res=pool.apply_async(run)
                        results.append(res)
                        res = [result.get() for result in results]

        print(res)
        pool.close()
        pool.join()
```

- **Output:**

```
PS C:\Users\ritika> & C:/Python312/python.exe "c:/Users/ritika/Downloads/from multiprocessing.py"
Enter the number of times to run the above query
3
[[(1,'Sandeep','Jain',None),(2,'Abhishek','Kelenia',1),(3,'Harsh','Aggarwal',1),(4,'Raju','Kumar',2),
(5,'Nikhil','aggarwal',2),(6,'Anshul','Aggarwal',2),(7,'Virat','Kohli',3),(8,'Rohit','sharma',3),
(1,'Sandeep','Jain',None),(2,'Abhishek','Kelenia',1),(3,'Harsh','Aggarwal',1),(4,'Raju','Kumar',2),
(5,'Nikhil','aggarwal',2),(6,'Anshul','Aggarwal',2),(7,'Virat','Kohli',3),(8,'Rohit','sharma',3),
(1,'Sandeep','Jain',None),(2,'Abhishek','Kelenia',1),(3,'Harsh','Aggarwal',1),(4,'Raju','Kumar',2),
(5,'Nikhil','aggarwal',2),(6,'Anshul','Aggarwal',2),(7,'Virat','Kohli',3),(8,'Rohit','sharma',3)]]
```