MINI-PROJECT

Title: Colour Detection using Pandas & OpenCV

Problem Statement: Colour Detection using Pandas & OpenCV by implementing an efficient deep learning algorithm.

Aim: Write a python program to Colour Detection using Pandas & OpenCV.

Objectives

- To understand the concept of deep learning algorithms.
- To understand the concept of colour detection using pandas and openCV.

Outcome:

After completing the implementation, the students will be able to:

- Display result for deep learning based on opencv library
- Efficiently Colour Detection using Pandas & OpenCV

Theory:

Colour detection is the process of detecting the name of any colour. For humans this is an extremely easy task but for computers, it is not straightforward. Human eyes and brains work together to translate light into colour. Light receptors that are present in our eyes transmit the signal to the brain. Our brain then recognizes the colour. Since childhood, we have mapped certain lights with their colour names. We will be using the somewhat same strategy to detect colour names.

Colour detection is the process of detecting and recognizing colours in an image or video. It involves identifying the RGB (Red, Green, Blue) values or other colour models of pixels in an image or video stream and categorizing them as specific colours.

Colour detection can be used in a variety of applications, such as image processing, computer vision, robotics, and more. It is commonly used in industries such as printing, automotive, and textiles to ensure colour consistency and accuracy.

Colour detection can be performed using various techniques, including thresholding, colour segmentation, and machine learning algorithms. These techniques can help identify and isolate specific colours in an image or video stream, enabling the development of applications such as colour-based object tracking, colour sorting, and colour recognition.

The concept of colour detection is, as the name suggests, a part of image processing that involves differentiation between objects based on their colour. For example if we are processing an image with a number of coloured objects, and we want to process only the ones of a particular colour, then colour detection methods basically return a binary image where only the portions with relevant colour are white, while the rest is black. This reduces the

information of the image to only the relevant portions, which make it easier to process for various operations. There are multiple ways in which to carry out colour detection.

Library use for colour detection:

1. **OpenCV:** OpenCV (Open Source Computer Vision Library) is a powerful open-source computer vision and machine learning software library. It provides a comprehensive set of functions and tools for image and video processing, including object detection, feature extraction, camera calibration, image enhancement, and more.

   OpenCV was initially developed by Intel in 1999, and since then, it has become one of the most popular and widely used computer vision libraries in the world. It is written in C++, but it has bindings for various programming languages, including Python, Java, and MATLAB.

   The library is designed to be cross-platform, and it runs on various operating systems, including Windows, Linux, macOS, Android, and iOS. It is free to use and distribute under the BSD license, making it an attractive choice for both academic and commercial applications.

   OpenCV is a versatile library that can be used for a wide range of applications, from simple image processing tasks such as filtering and thresholding to complex computer vision tasks such as object detection and tracking, facial recognition, and 3D reconstruction. Its ease of use and powerful features make it a go-to choice for many computer vision and machine learning developers.

2. **Pandas:** Pandas is an open-source Python library that provides high-performance, easy-to-use data structures and data analysis tools for handling structured data. It is built on top of the NumPy library and provides additional data manipulation and analysis functionalities.

   Pandas provides two primary data structures: Series and DataFrame. A Series is a one-dimensional array-like object that can hold any data type, while a DataFrame is a two-dimensional table-like data structure that consists of rows and columns of data.

   Pandas provides a wide range of data manipulation and analysis tools, including data filtering, grouping, aggregation, merging, pivoting, and reshaping. It also provides functions for data cleaning, missing data handling, and data transformation.

   Pandas is widely used in data analysis and data science applications, including finance, economics, social sciences, and engineering. Its ease of use, flexibility, and performance make it a popular choice for handling and analyzing structured data in Python.

3. **Numpy:** NumPy is an open-source Python library for scientific computing that provides support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.

NumPy provides a fast and efficient array-processing interface to Python, which is significantly faster than Python's built-in list structure. It also provides a range of mathematical functions for linear algebra, Fourier analysis, random number generation, and more.

The core of NumPy is the nd array (n-dimensional array) object, which is a fast, flexible, and memory-efficient array structure that can hold large amounts of data. NumPy arrays can be created from a range of sources, including lists, tuples, and other arrays. They can be manipulated using a wide range of array operations, such as slicing, indexing, reshaping, and broadcasting.

NumPy is widely used in scientific computing, data analysis, and machine learning applications. It is a fundamental building block of many popular Python libraries, including Pandas, Matplotlib, and SciPy. Its speed and efficiency make it a popular choice for working with large datasets, and its ability to interface with C and Fortran code makes it a valuable tool for scientific computing applications.

About the Project:

In this colour detection Python project, we are going to build an application through which you can automatically get the name of the colour by clicking on them. So for this, we will have a data file that contains the colour name and its values. Then we will calculate the distance from each colour and find the shortest one.

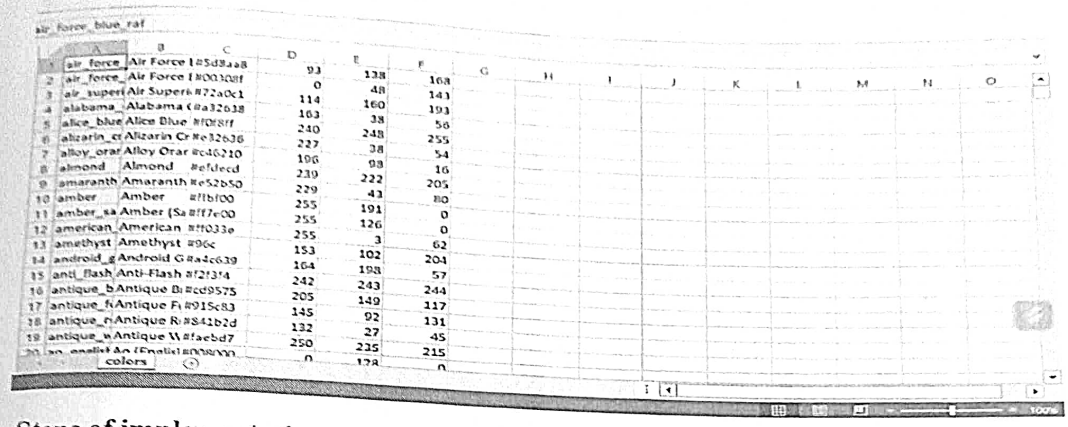
The algorithm used in this code is a simple distance function that calculates the minimum distance between the selected colour and all the colours in the CSV file. The colour with the smallest distance is considered the closest match, and its name is returned. This is a brute-force approach, and more sophisticated methods such as k-means clustering or neural networks can be used for colour detection. However, this simple distance-based approach is fast and accurate for small datasets like the one used in this code

Prerequisites

Before starting with this Python project with source code, you should be familiar with the computer vision library of Python that is *OpenCV* and *Pandas*.

The Dataset:

Colours are made up of 3 primary colours; red, green, and blue. In computers, we define each colour value within a range of 0 to 255. So in how many ways we can define a colour? The answer is 256*256*256 = 16,581,375. There are approximately 16.5 million different ways to represent a colour. In our dataset, we need to map each colour's values with their corresponding names. But don't worry, we don't need to map all the values. We will be using a dataset that contains RGB values with their corresponding names.



	A	B	C	D	E	F
1	air_force	Air Force	#5d8aa8	93	138	168
2	air_force_	Air Force	#000000	0	48	143
3	air_superi	Air Superi	#72a0c1	114	160	193
4	alabama_	Alabama	#a32638	163	38	56
5	alice_blue	Alice Blue	#f0f8ff	240	248	255
6	alizarin_cr	Alizarin Cr	#e32636	227	38	54
7	alloy_orar	Alloy Orar	#c46210	196	98	16
8	almond	Almond	#efdecd	239	222	205
9	amaranth	Amaranth	#e52b50	229	43	80
10	amber	Amber	#ffbf00	255	191	0
11	amber_sa	Amber (Sa	#ff7e00	255	126	0
12	american_	American	#ff033e	255	3	62
13	amethyst	Amethyst	#96c	153	102	204
14	android_g	Android G	#a4c639	164	198	57
15	anti_flash	Anti-Flash	#f2f3f4	242	243	244
16	antique_b	Antique Bı	#cd9575	205	149	117
17	antique_fı	Antique Fı	#915c83	145	92	131
18	antique_r	Antique R	#841b2d	132	27	45
19	antique_w	Antique W	#faebd7	250	235	215
20	ao_(englis	Ao (Englis	#008000	0	128	0

colors

Steps of implementation of colour detection:

To colour detection, the following steps can be taken:

Step1:Import necessary libraries: cv2, numpy, pandas, and argparse.

Step 2:Create an argument parser to take the image path from the command line.

Step 3:Read the image using OpenCV.

Step 4: Define global variables for later use.

Step 5:Read a CSV file containing colour names and their RGB values using pandas.

Step 6:Define a function getColorName() to calculate the minimum distance from all colors and get the most matching colour.

Step 7:Define a function draw_function() to get x, y coordinates of mouse double click.

Step 8:Create a window to display the image and set mouse callback function.

Step 9: Inside an infinite loop:

a. Display the image.

b. If the user double-clicks the mouse, update the global variables with the RGB values of the pixel clicked, and draw a rectangle on the image with the colour of the clicked pixel.

c. Get the colour name corresponding to the clicked pixel's RGB values using the getColourName() function.

d. Display the colour name and RGB values on the image.

e. If the colour is very light, display the text in black colour.

f. Break the loop if the user presses the Esc key.

Step 10: Destroy all windows when done.

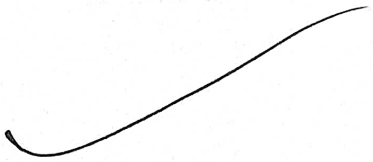
Program For the color detection project:

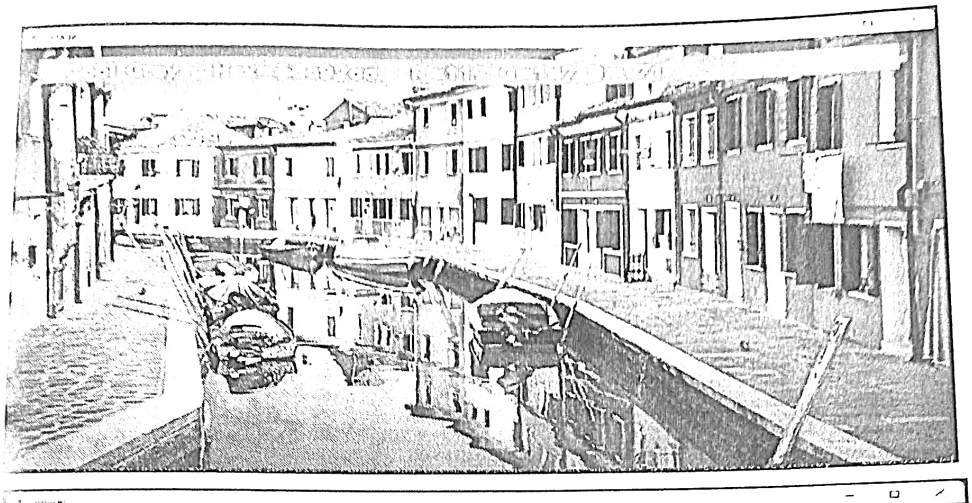
```
import cv2
import numpy as np
import pandas as pd
import argparse

#Creating argument parser to take image path from command line
ap = argparse.ArgumentParser()
ap.add_argument('-i', '--image', required=True, help="Image Path")
args = vars(ap.parse_args())
img_path = args['image']

#Reading the image with opencv
img = cv2.imread(img_path)

#declaring global variables (are used later on)
clicked = False
r = g = b = xpos = ypos = 0

#Reading csv file with pandas and giving names to each column
index=["color","color_name","hex","R","G","B"]
csv = pd.read_csv('colors.csv', names=index, header=None)

#function to calculate minimum distance from all colors and get the most matching color
def getColorName(R,G,B):
    minimum = 10000
    for i in range(len(csv)):
        d = abs(R- int(csv.loc[i,"R"])) + abs(G- int(csv.loc[i,"G"]))+ abs(B- int(csv.loc[i,"B"]))
        if(d<=minimum):
            minimum = d
```

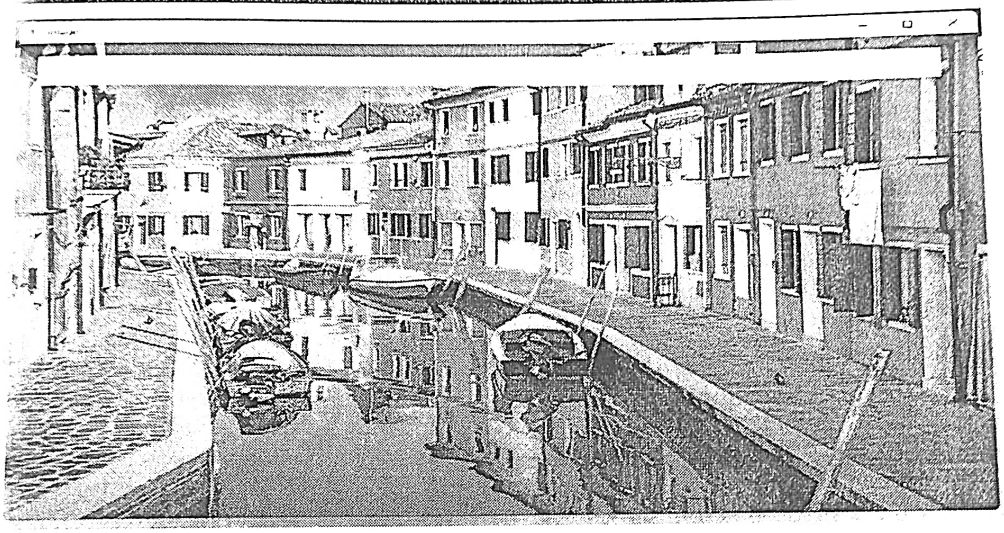
```
        cname = csv.loc[i,"color_name"]
    return cname

#function to get x,y coordinates of mouse double click
def draw_function(event, x,y,flags,param):
    if event == cv2.EVENT_LBUTTONDBLCLK:
        global b,g,r,xpos,ypos, clicked
        clicked = True
        xpos = x
        ypos = y
        b,g,r = img[y,x]
        b = int(b)
        g = int(g)
        r = int(r)

cv2.namedWindow('image')
cv2.setMouseCallback('image',draw_function)

while(1):

    cv2.imshow("image",img)
    if (clicked):

        #cv2.rectangle(image, startpoint, endpoint, color, thickness)-1 fills entire rectangle
        cv2.rectangle(img,(20,20), (750,60), (b,g,r), -1)

        #Creating text string to display( Color name and RGB values )
        text = getColorName(r,g,b) + ' R='+ str(r) +  ' G='+ str(g) +  ' B='+ str(b)

        #cv2.putText(img,text,start,font(0-7),fontScale,color,thickness,lineType )
```

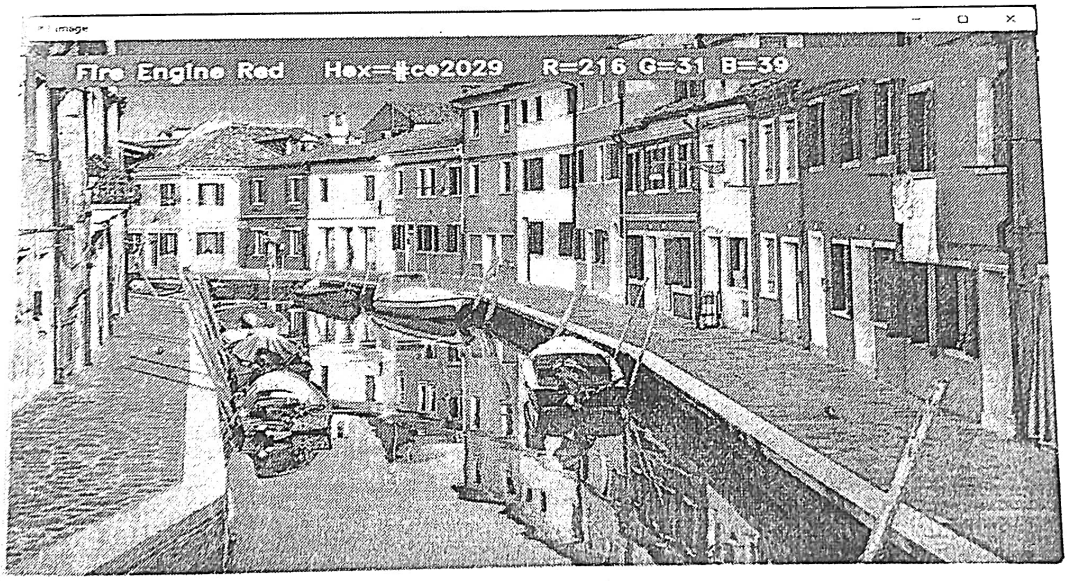
```
        cv2.putText(img, text,(50,50),2,0.8,(255,255,255),2,cv2.LINE_AA)

        #For very light colours we will display text in black colour
        if(r+g+b>=600):
            cv2.putText(img, text,(50,50),2,0.8,(0,0,0),2,cv2.LINE_AA)

        clicked=False

    #Break the loop when user hits 'esc' key
    if cv2.waitKey(20) & 0xFF ==27:
        break

cv2.destroyAllWindows()
```

Output For the :









Conclusion:

Thus, we will be able to implement deep learning techniques to create colour detection using pandas and OpenCV

MINI-PROJECT

Parallel Quicksort Algorithm

Title: Performance enhancement of Parallel Quicksort Algorithm using MPI

Problem Statement: To evaluate performance enhancement of parallel Quicksort Algorithm using MPI.

Aim: Write a program to enhance the performance of parallel quicksort algorithm using MPI

Prerequisites:

- Concepts of existing sequential algorithms
- Concepts of High Performance Computing

Objectives:

- To understand the concept of quicksort algorithm based on sequential approach.
- To understand the concept of parallel algorithm.
- To achieve performance enhancement of parallel quicksort algorithm using MPI

Outcome:

After completing the implementation, the students will be able to:

- Display result for parallel quicksort algorithm
- Analyze and enhance the performance of parallel quicksort algorithm

Theory:

Parallel computing has become increasingly popular in recent years, as it enables the development of efficient and scalable algorithms that can take advantage of the computational power of multiple processors. One of the most widely used parallel algorithms is the Quicksort algorithm, which is a popular sorting algorithm that is based on the divide and conquer paradigm. In parallel Quicksort, the input data is partitioned into multiple sub-arrays, each of which is sorted independently on a different processor. The results are then combined to produce the final sorted array.

MPI (Message Passing Interface) is a popular parallel computing library that enables the development of parallel applications that can run on multiple processors or nodes. MPI provides a set of communication primitives that allow processes to exchange data and synchronize their execution. The parallel Quicksort algorithm can be implemented using MPI by partitioning the input array into multiple sub-arrays and distributing them across the processors. Each processor sorts its sub-array independently and exchanges data with other processors to merge the sorted sub-arrays.

One of the key advantages of using parallel Quicksort with MPI is that it can significantly improve the performance of the algorithm. This is because the algorithm can take advantage of the parallel processing power of multiple processors to sort the input array much faster than a single processor. The performance improvement can be measured in terms of the speedup and efficiency of the algorithm.

Speedup is defined as the ratio of the time taken to execute the algorithm on a single processor to the time taken to execute the algorithm on multiple processors. Efficiency is defined as the ratio of the speedup achieved to the number of processors used. A perfectly efficient algorithm would have an efficiency of 1.0, indicating that the performance scales linearly with the number of processors used.

There are several factors that can affect the performance of parallel Quicksort with MPI. One of the key factors is the size of the input array. As the size of the input array increases, the performance of the algorithm may degrade due to the increased communication overhead between the processors. Another factor is the distribution of the input data among the processors. If the data is unevenly distributed, some processors may finish their work earlier than others, leading to idle time and reduced efficiency.

Message Passing Interface (MPI):

MPI (Message Passing Interface) is a popular parallel computing library that provides a standardized way to develop parallel applications that can run on multiple processors or nodes. It enables processes to communicate and synchronize their execution using a set of communication primitives, such as send and receive operations. MPI is widely used in scientific computing, high-performance computing, and other fields that require parallel processing of large datasets.

Parallel Quicksort is one of the many algorithms that can be implemented using MPI to take advantage of the parallel processing power of multiple processors. MPI enables the distribution of the input data across the processors and the parallel execution of the Quicksort algorithm on each processor. Each processor sorts its sub-array independently, and then the results are combined by exchanging data with other processors to merge the sorted sub-arrays.

MPI is particularly useful for performance enhancement of parallel Quicksort for several reasons. First, it enables efficient communication between the processors, which is critical for effective load balancing and minimizing idle time. Load balancing is essential to ensure that each processor has an equal amount of work to perform, and idle time is minimized to ensure that the algorithm runs as efficiently as possible.

Second, MPI provides a flexible and scalable communication model that can be adapted to different types of parallel architectures and configurations. This allows developers to optimize the performance of the algorithm for specific hardware configurations and to take advantage of the unique features of different parallel architectures.

Third, MPI provides a high-level abstraction of the parallel hardware, which makes it easier for developers to write parallel applications without having to worry about low-level details of the hardware. This abstraction simplifies the development process and enables developers to focus on the algorithmic aspects of their applications.

Finally, MPI is widely used in the scientific computing community, and there is a large community of users and developers who provide support and share best practices. This community is an excellent resource for developers who are new to MPI or who need help optimizing their applications.

Speedup is defined as the ratio of the time taken to execute the algorithm on a single processor to the time taken to execute the algorithm on multiple processors. Efficiency is defined as the ratio of the speedup achieved to the number of processors used. A perfectly efficient algorithm would have an efficiency of 1.0, indicating that the performance scales linearly with the number of processors used.

There are several factors that can affect the performance of parallel Quicksort with MPI. One of the key factors is the size of the input array. As the size of the input array increases, the performance of the algorithm may degrade due to the increased communication overhead between the processors. Another factor is the distribution of the input data among the processors. If the data is unevenly distributed, some processors may finish their work earlier than others, leading to idle time and reduced efficiency.

Message Passing Interface (MPI):

MPI (Message Passing Interface) is a popular parallel computing library that provides a standardized way to develop parallel applications that can run on multiple processors or nodes. It enables processes to communicate and synchronize their execution using a set of communication primitives, such as send and receive operations. MPI is widely used in scientific computing, high-performance computing, and other fields that require parallel processing of large datasets.

Parallel Quicksort is one of the many algorithms that can be implemented using MPI to take advantage of the parallel processing power of multiple processors. MPI enables the distribution of the input data across the processors and the parallel execution of the Quicksort algorithm on each processor. Each processor sorts its sub-array independently, and then the results are combined by exchanging data with other processors to merge the sorted sub-arrays.

MPI is particularly useful for performance enhancement of parallel Quicksort for several reasons. First, it enables efficient communication between the processors, which is critical for effective load balancing and minimizing idle time. Load balancing is essential to ensure that each processor has an equal amount of work to perform, and idle time is minimized to ensure that the algorithm runs as efficiently as possible.

Second, MPI provides a flexible and scalable communication model that can be adapted to different types of parallel architectures and configurations. This allows developers to optimize the performance of the algorithm for specific hardware configurations and to take advantage of the unique features of different parallel architectures.

Third, MPI provides a high-level abstraction of the parallel hardware, which makes it easier for developers to write parallel applications without having to worry about low-level details of the hardware. This abstraction simplifies the development process and enables developers to focus on the algorithmic aspects of their applications.

Finally, MPI is widely used in the scientific computing community, and there is a large community of users and developers who provide support and share best practices. This community is an excellent resource for developers who are new to MPI or who need help optimizing their applications.

Thus, MPI is a powerful parallel computing library that can be used to implement parallel Quicksort and other algorithms efficiently. It provides a

standardized way to develop parallel applications and enables efficient communication and load balancing between processors. By using MPI, developers can take advantage of the parallel processing power of multiple processors and achieve significant performance enhancements compared to sequential algorithms.

Parallel Quicksort Algorithm:

Parallel Quicksort is a popular sorting algorithm that takes advantage of the parallel processing power of multiple processors to sort large datasets efficiently. The algorithm is based on the divide and conquer paradigm, which involves dividing the input data into smaller sub-problems that can be solved independently and then combining the results to obtain the final solution.

In the case of parallel Quicksort, the input data is partitioned into multiple sub-arrays, each of which is sorted independently on a different processor. The sub-arrays are partitioned based on a pivot element, which is selected from the input array. The elements in each sub-array are compared with the pivot element, and those that are smaller are moved to the left of the pivot, while those that are larger are moved to the right.

After each processor has sorted its sub-array, the results are combined by exchanging data with other processors to merge the sorted sub-arrays. The merging process involves comparing the elements in the sorted sub-arrays and selecting the smallest element to add to the final sorted array. This process continues until all the elements have been merged into a single sorted array.
Parallel Quicksort can be implemented using various parallel computing libraries, such as MPI (Message Passing Interface) or OpenMP (Open Multi-Processing). The performance of the algorithm can be measured in terms of the speedup and efficiency achieved, which depend on the number of processors used and the size and distribution of the input data.

Overall, parallel Quicksort is a powerful technique for sorting large datasets efficiently by taking advantage of the parallel processing power of multiple processors. With careful implementation and optimization, the algorithm can provide significant performance enhancements compared to sequential Quicksort algorithms.

Algorithm Steps of Parallel Quicksort:

The high-level steps for implementing parallel Quicksort using MPI for performance enhancement:

1. Initialize MPI and obtain the rank and size of the processes.
2. Distribute the input data among the processes using MPI_Send and MPI_Recv operations.
3. Each process sorts its sub-array using the sequential Quicksort algorithm.

4. Exchange data between the processes using MPI_Send and MPI_Recv operations to merge the sorted sub-arrays.
5. Repeat steps 3 and 4 recursively until all the elements are sorted.
6. Gather the sorted data using the MPI_Gather operation.
7. Finalize MPI and output the sorted data.

It is important to note that there are different variations of this algorithm that can be optimized for specific hardware configurations and input data distributions. For example, load balancing can be optimized by adjusting the size of the sub-arrays or by using dynamic load balancing techniques. Additionally, the algorithm can be optimized for different types of parallel architectures, such as shared-memory or distributed-memory architectures.

Code:

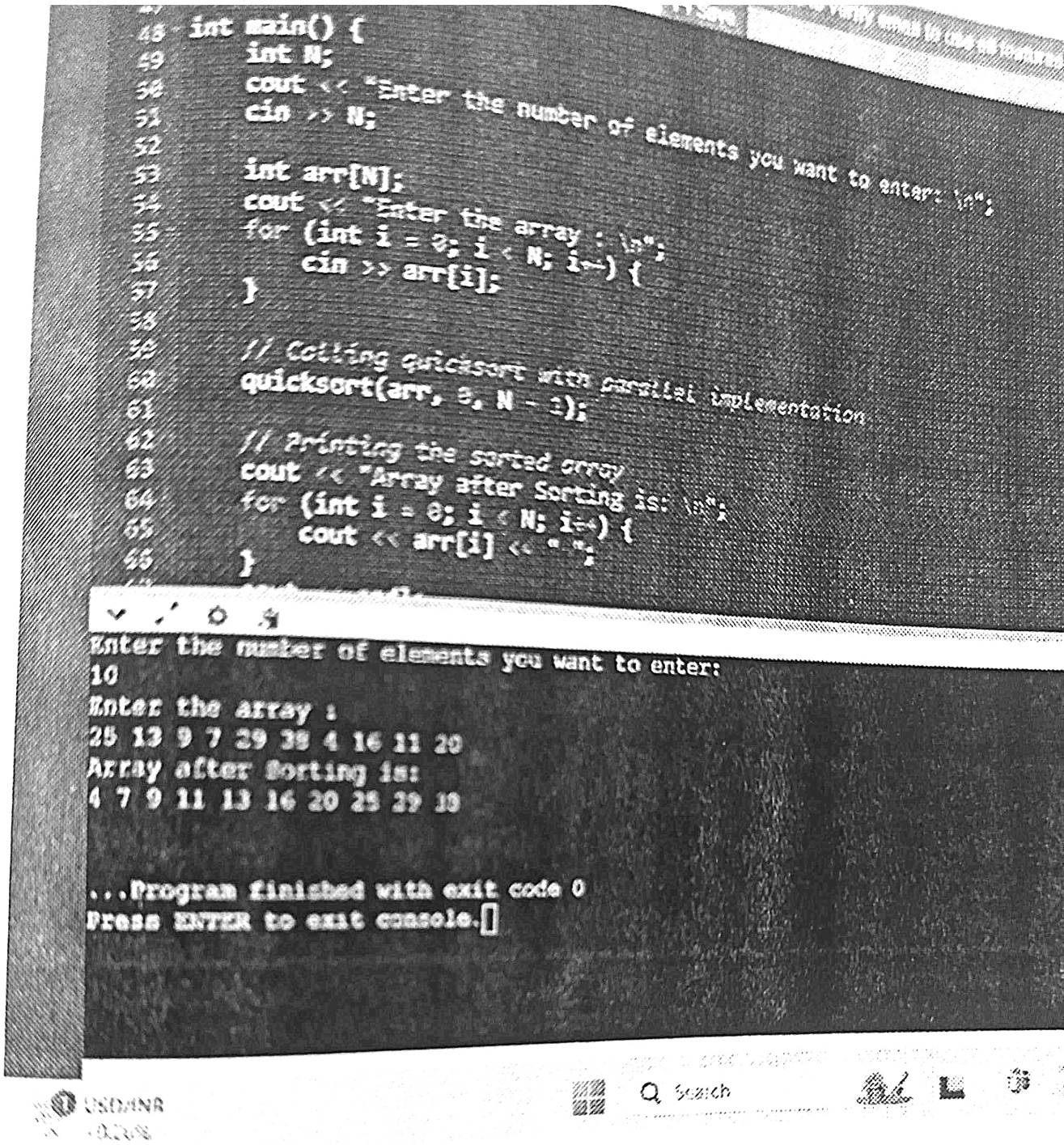
```
// C++ program to implement the Quick Sort
// using OMI
#include <bits/stdc++.h>
#include <omp.h>
using namespace std;

// Function to swap two numbers a and b
void swap(int* a, int* b)
{
        int t = *a;
        *a = *b;
        *b = t;
}

// Function to perform the partitioning
// of array arr[]
int partition(int arr[], int start, int end)
{
        // Declaration
```

```
    int pivot = arr[end];
    int i = (start - 1);

    // Rearranging the array
    for (int j = start; j <= end - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[end]);

    // Returning the respective index
    return (i + 1);
}

// Function to perform QuickSort Algorithm
// using openmp
void quicksort(int arr[], int start, int end)
{
    // Declaration
    int index;

    if (start < end) {

        // Getting the index of pivot
        // by partitioning
        index = partition(arr, start, end);

// Parallel sections
```

```
#pragma omp parallel sections
            {
#pragma omp section
                    {
                            // Evaluating the left half
                            quicksort(arr, start, index - 1);
                    }
#pragma omp section
                    {
                            // Evaluating the right half
                            quicksort(arr, index + 1, end);
                    }
            }
        }
}

// Driver Code
int main()
{
        // Declaration
        int N;

        // Taking input the number of
        // elements we wants
        cout << "Enter the number of elements"
                << " you want to Enter\n";
        cin >> N;

        // Declaration of array
        int arr[N];
```

```
    cout << "Enter the array: \n";

    // Taking input that array
    for (int i = 0; i < N; i++) {
        cin >> arr[i];
    }
    // Calling quicksort having parallel
    // code implementation
    quicksort(arr, 0, N - 1);

    // Printing the sorted array
    cout << "Array after Sorting is: \n";

    for (int i = 0; i < N; i++) {
        cout << arr[i] << " ";
    }

    return 0;
}
```

Output:



Conclusion:

Thus, we will be able evaluate performance e
algorithm using MPI.