Content-based Image Search

Parallelization using the OpenMP library

Group members:

Faridreza Momtaz Zandi

Mahya EhsaniMehr

Ali Mojahed

Spring 2023

::General description::

In this project, which was written in Python, we wanted to be able to find similar images using an image query by having a dataset of random and unrelated images. For this, there are several important factors that we must apply to the images and use each of them to finally make the best choice for the final images. These factors allow us to remove noises, obtain a histogram, and apply appropriate features to obtain a feature vector and finally find the similarity between two images by finding the Euclidean distance.

The first problem we encountered was not the possibility of using OpenMP commands in Python, and only one library supported it called Numba, which was very troublesome and could only work in very special and isolated conditions, by checking libraries Like pymp, we concluded that Python is not a suitable language for working with OpenMP and it is better to use languages that support this library natively, so we decided to rewrite the program in our own C+ Plus language.

::Code Review::

Together, we will review the program code and the use of each section:

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <dirent.h>
#include <opencv2/opencv.hpp>
#include <chrono>
#include <omp.h>
```

The code includes several libraries for I/O operations, image processing, scheduling, and parallelization using OpenMP.

```cpp
// Function to resize an image to a given size
cv::Mat resizeImage(const cv::Mat& image, int width, int height)
{
    cv::Mat resizedImage;
    cv::resize(image, resizedImage, cv::Size(width, height));
    return resizedImage;
}
```

resizeImage: resizes an image to the specified width and length.

```cpp
// Function to compute the mean value of an image
double computeMean(const cv::Mat& image)
{
    cv::Scalar meanVal = cv::mean(image);
    return meanVal[0];
}

```

computeMean: calculates the mean value of an image.

```cpp
// Function to compute the median value of an image
double computeMedian(const cv::Mat& image)
{
    cv::Mat sortedImage;
    cv::sort(image, sortedImage, cv::SORT_EVERY_COLUMN | cv::SORT_ASCENDING);
    int totalPixels = image.rows * image.cols;

    double medianValue;
    if (totalPixels % 2 == 0)
    {
        int index1 = (totalPixels / 2) - 1;
        int index2 = index1 + 1;
        medianValue = (sortedImage.at<uchar>(index1) + sortedImage.at<uchar>(index2)) / 2.0;
    }
    else
    {
        int index = (totalPixels / 2);
        medianValue = sortedImage.at<uchar>(index);
    }

    return medianValue;
}

```

computeMedian: calculates the median value of an image.

```cpp
// Function to compute the histogram of an image
cv::Mat computeHistogram(const cv::Mat& image)
{
    int numBins = 256; // Number of bins for the histogram
    int histSize[] = { numBins };
    float range[] = { 0, 256 };
    const float* ranges[] = { range };
    int channels[] = { 0 }; // Compute histogram only for the first channel (grayscale image)

    cv::Mat histogram;
    cv::calcHist(&image, 1, channels, cv::Mat(), histogram, 1, histSize, ranges);

    return histogram;
}

```

computeHistogram: calculates the histogram of an image.

```cpp
// Function to compute the standard deviation of an image
double computeStandardDeviation(const cv::Mat& image)
{
    double mean = computeMean(image);
    cv::Scalar meanSquaredDiff = cv::mean((image - mean).mul(image - mean));
    return std::sqrt(meanSquaredDiff[0]);
}

```

computeStandardDeviation: Calculates the standard deviation of an image.

```cpp
// Function to compute the Hu Moments of an image
cv::Mat computeHuMoments(const cv::Mat& image)
{
    cv::Moments moments = cv::moments(image);
    cv::Mat huMoments;
    cv::HuMoments(moments, huMoments);
    return huMoments;
}

```

computeHuMoments: Computes the Hu moments of an image.

```cpp
// Function to compute the Euclidean distance between two feature vectors
double computeDistance(const cv::Mat& queryFeatureVector, const cv::Mat& datasetFeatureVector)
{
    cv::Mat diff;
    cv::absdiff(queryFeatureVector, datasetFeatureVector, diff);
    cv::Mat squaredDiff = diff.mul(diff);
    cv::Scalar sum = cv::sum(squaredDiff);
    return std::sqrt(sum[0]);
}
```

computeDistance: calculates the Euclidean distance between two feature vectors.

```cpp
std::vector<std::string> getFilesInDirectory(const std::string& dirPath)
{
    std::vector<std::string> fileNames;
    DIR* directory;
    struct dirent* entry;

    directory = opendir(dirPath.c_str());
    if (directory != nullptr)
    {
        while ((entry = readdir(directory)) != nullptr)
        {
            std::string fileName = entry->d_name;
            if (fileName != "." && fileName != "..")
            {
                fileNames.push_back(dirPath + fileName);
            }
        }
        closedir(directory);
    }

    return fileNames;
}
```

The getFilesInDirectory function takes a directory path as input and returns a vector of file names in that directory. It uses the dirent.h library to read the directory contents.

```cpp
int main()
{
    auto start = high_resolution_clock::now();
    std::string datasetPath = "./dataset/images/"; // Path to the dataset images
    std::string queryImagePath = "000000124442.jpg"; // Path to the query image

    std::vector<std::string> imagePaths = getFilesInDirectory(datasetPath);

    // Load and resize the query image
    cv::Mat queryImage = cv::imread(queryImagePath, cv::IMREAD_GRAYSCALE);
    queryImage = resizeImage(queryImage, 500, 500);

    // Apply noise reduction techniques to the query image
    cv::Mat queryImageSmallBlur;
    cv::Mat queryImageLargeBlur;
    cv::Mat queryImageGaussianBlur;

    cv::blur(queryImage, queryImageSmallBlur, cv::Size(3, 3));
    cv::blur(queryImage, queryImageLargeBlur, cv::Size(9, 9));
    cv::GaussianBlur(queryImage, queryImageGaussianBlur, cv::Size(9, 9), 0);

    // Compute features for the query image
    double queryMean = computeMean(queryImage);
    double queryMedian = computeMedian(queryImage);
    double queryStdDev = computeStandardDeviation(queryImage);
    cv::Mat queryHuMoments = computeHuMoments(queryImage);
    cv::Mat queryHistogram = computeHistogram(queryImage);

    // Compute features for all dataset images
    std::vector<std::string> bestImagePaths;
    std::vector<double> bestImageDistances(20, std::numeric_limits<double>::max());
```

The main function is the entry point of the program. Performs the following steps:

- Initializes the start time to measure the duration of the run.
- Defines the path of dataset images and query image.
- It calls the function getFilesInDirectory to get the paths of all the images in the dataset.
- Loads the query image and resizes it.
- It applies noise reduction techniques (small blurs, large blurs and Gaussian blurs) to the query image.
- Calculates various features (mean, median, standard deviation, Ho-moments and histogram) for the query image.

- Launches containers to store the best image paths and their corresponding distances.

```cpp
    omp_set_num_threads(3); // Set the number of threads to 3

    #pragma omp parallel for
    for (int i = 0; i < imagePaths.size(); ++i)
    {
        // Load and resize the dataset image
        cv::Mat datasetImage = cv::imread(imagePaths[i], cv::IMREAD_GRAYSCALE);
        datasetImage = resizeImage(datasetImage, 500, 500);

        // Apply noise reduction techniques to the dataset image
        cv::Mat datasetImageSmallBlur;
        cv::Mat datasetImageLargeBlur;
        cv::Mat datasetImageGaussianBlur;

        cv::blur(datasetImage, datasetImageSmallBlur, cv::Size(3, 3));
        cv::blur(datasetImage, datasetImageLargeBlur, cv::Size(9, 9));
        cv::GaussianBlur(datasetImage, datasetImageGaussianBlur, cv::Size(9, 9), 0);

        // Compute features for the dataset image
        double datasetMean = computeMean(datasetImage);
        double datasetMedian = computeMedian(datasetImage);
        double datasetStdDev = computeStandardDeviation(datasetImage);
        cv::Mat datasetHuMoments = computeHuMoments(datasetImage);
        cv::Mat datasetHistogram = computeHistogram(datasetImage);

        // Compute similarity using Euclidean distance
        double distance = computeDistance(cv::Mat(1, 1, CV_64F, &datasetMean), cv::Mat(1, 1, CV_64F, &queryMean))
            + computeDistance(cv::Mat(1, 1, CV_64F, &datasetMedian), cv::Mat(1, 1, CV_64F, &queryMedian))
            + computeDistance(cv::Mat(1, 1, CV_64F, &datasetStdDev), cv::Mat(1, 1, CV_64F, &queryStdDev))
            + computeDistance(datasetHuMoments, queryHuMoments)
            + computeDistance(datasetHistogram, queryHistogram);
```

It sets the number of threads to 3 using the OpenMP omp_set_num_threads function.

It uses OpenMP parallelization with a parallel omp pragma for instructions to process dataset images concurrently.

In the parallel region, each dataset image is loaded, resized, and processed similarly to the request image.

The similarity between the query image and each dataset image is calculated using the Euclidean distance formula.

The best image paths and distances are updated based on the calculated similarity values.

```cpp
        // Update the best image paths if the current distance is smaller than the current best distances
        #pragma omp critical
        {
            for (int j = 0; j < bestImageDistances.size(); ++j)
            {
                if (distance < bestImageDistances[j])
                {
                    bestImagePaths.insert(bestImagePaths.begin() + j, imagePaths[i]);
                    bestImagePaths.resize(20);
                    bestImageDistances.insert(bestImageDistances.begin() + j, distance);
                    bestImageDistances.resize(20);
                    break;
                }
            }
        }
    }

    // Save the names of the best 20 images to a file
    std::ofstream outputFile("best_images.txt");
    if (outputFile.is_open())
    {
        for (const auto& imagePath : bestImagePaths)
        {
            outputFile << imagePath << std::endl;
        }
        outputFile.close();
    }
    else
    {
        std::cerr << "Failed to open the output file." << std::endl;
        return 1;
    }
    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<microseconds>(stop - start);
    std::cout << "\nTime taken by function: " << duration.count() << " microseconds";

    return 0;
}
```

After the parallel region, the names of the top 20 images are stored in a text file named "best_images.txt".

If the file cannot be opened, an error message is printed.

The end time is recorded and the running time is calculated.

The execution time is printed on the console.

::presentation of results::

First, we executed the above program with a smaller dataset and in four different forms, one to four fragments, and the results of this execution are shown in the diagram below.

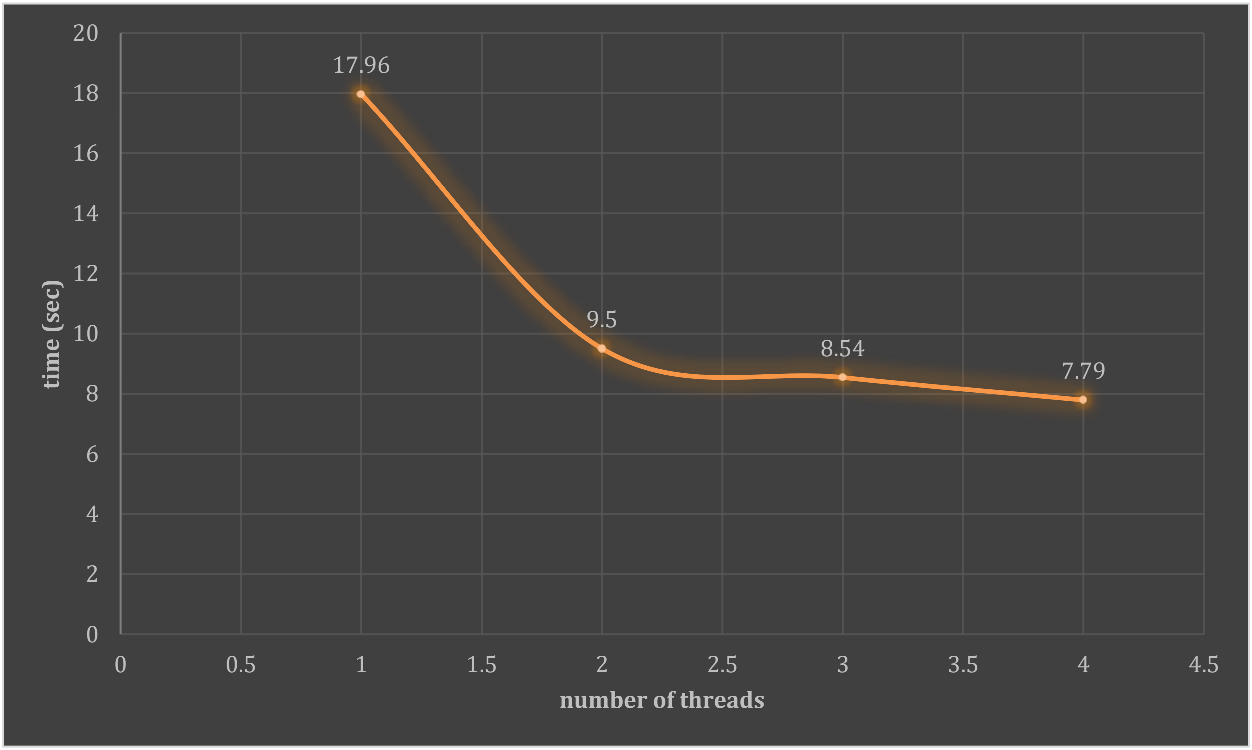

chart 1: executing code on 1215 images with 1 to 4 threads

Now we run the program with a larger dataset and in the form of one to eight chunks this time and record all the states, as we can see in the diagram below:

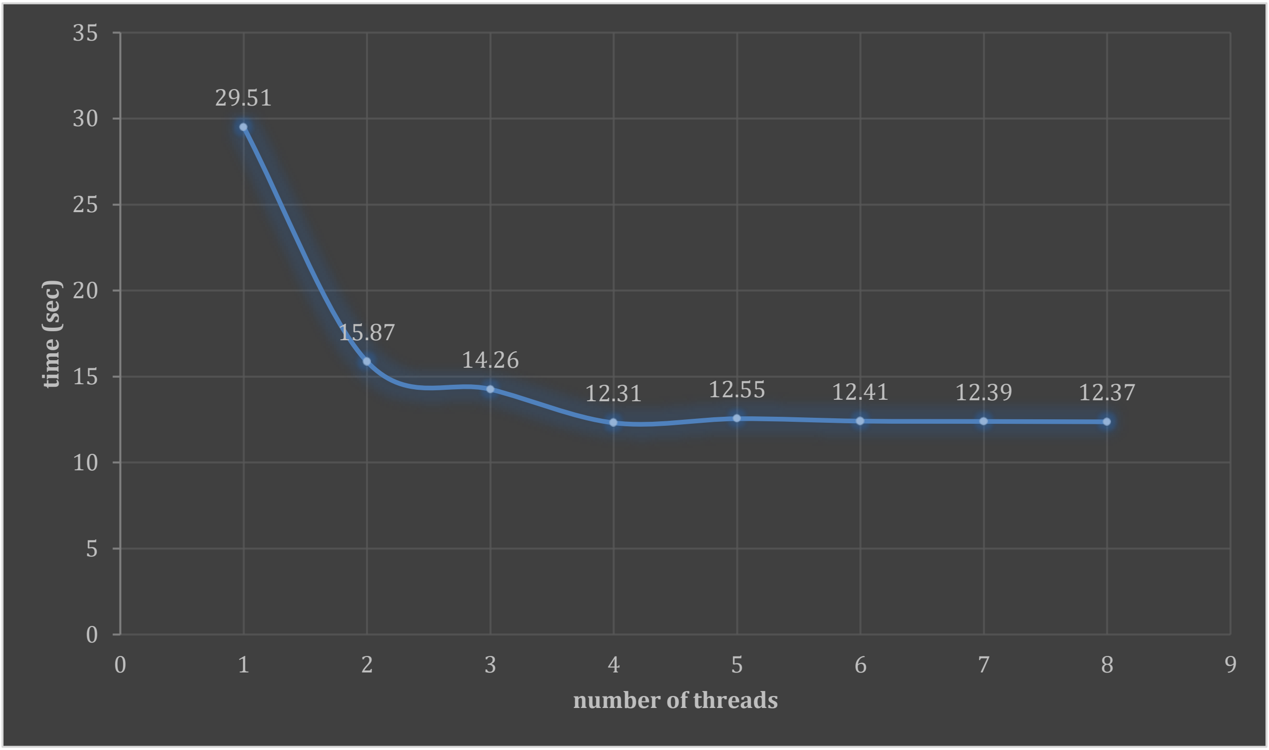

chart 2: executing code on 2015 images with 1 to 8 threads

Now, in the following images, we can see some examples of program execution in different conditions:

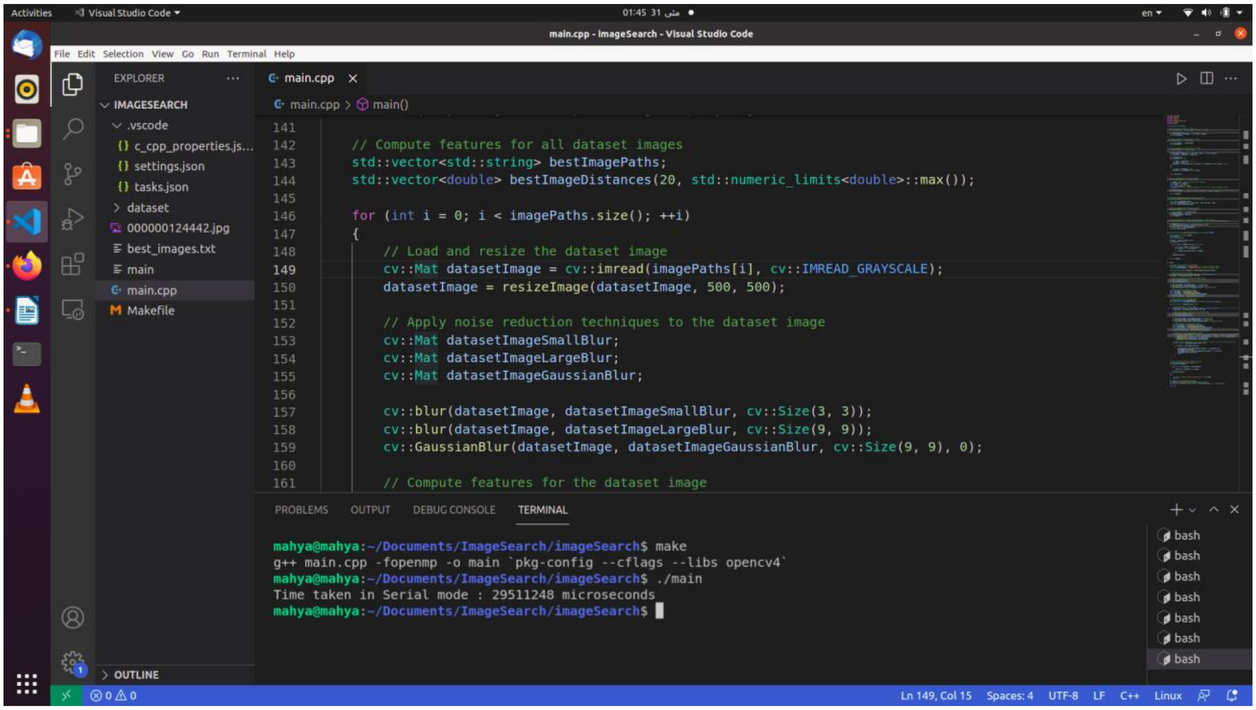

Figure1 execution in serial mode

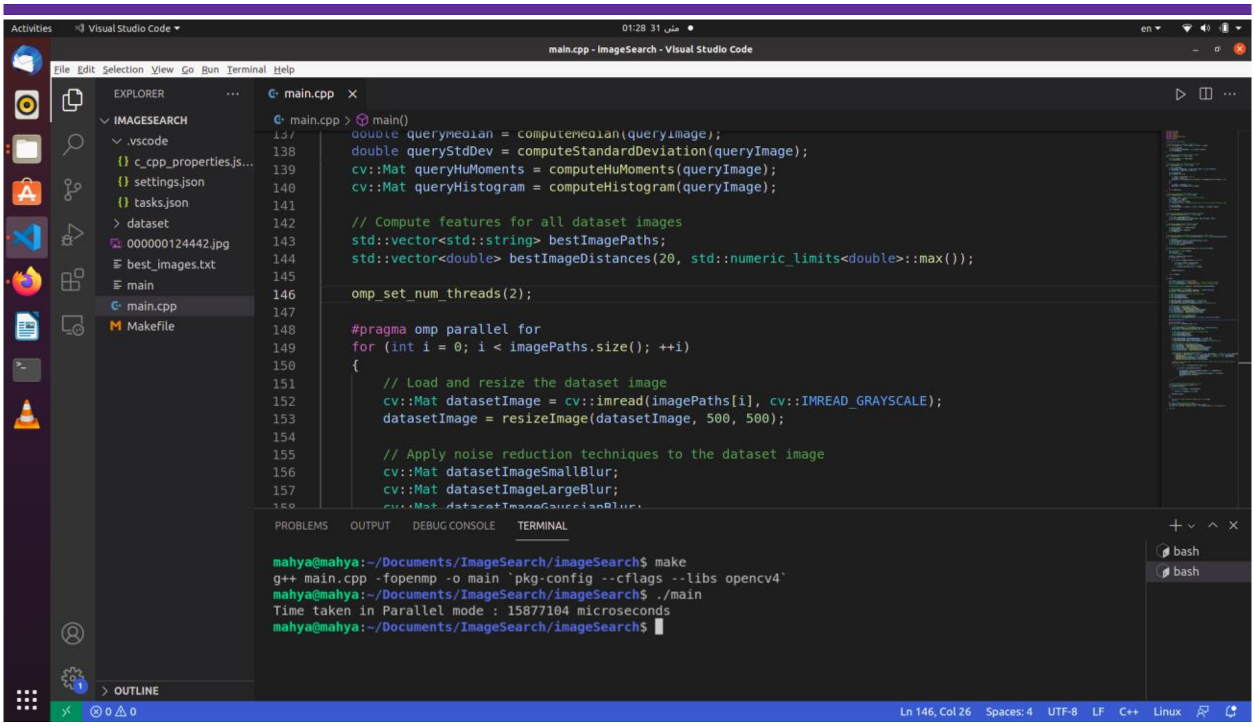

Figure 2 excution in parallel and two thread mode

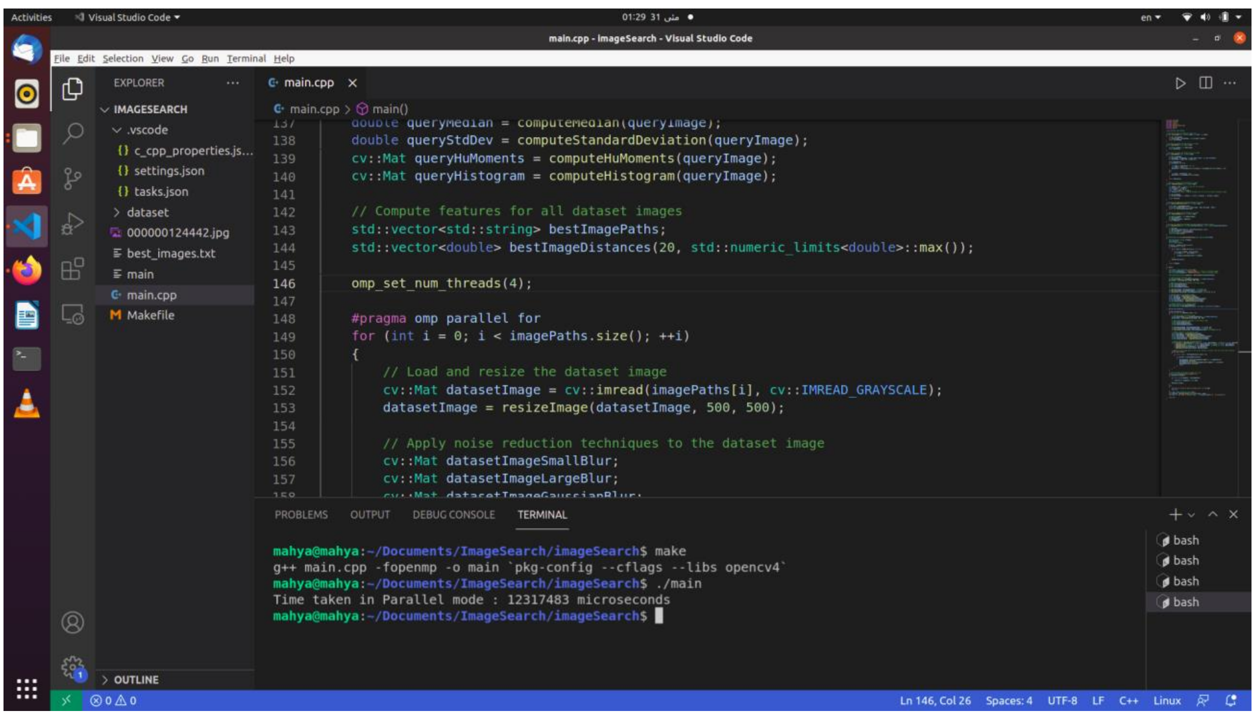

Figure 3 execution in parallel and four thread which is the best

::Hardware specifications::

The machine on which this code was executed and the results obtained on Ubuntu 20.04 operating system and hardware are as follows:

Machine model: Lenovo IdeaPad 500-ISK

Processor: Intel core™ i5-6200U CPU @ 2.30GHz 2.4 GHz

Physical cores: 2 pcs

Supported chips: 4 pcs

RAM memory: 8 GB