



**Team ID: 104**

**Team Members:**

**Shehab Mohamed Ibrahim – 18P7213**

**Yusuf Sameh Fawzi – 18P1399**

**Ahmad Ossama Ahmad – 18P6575**

**Abdelrahman Mahmoud – 18P6605**

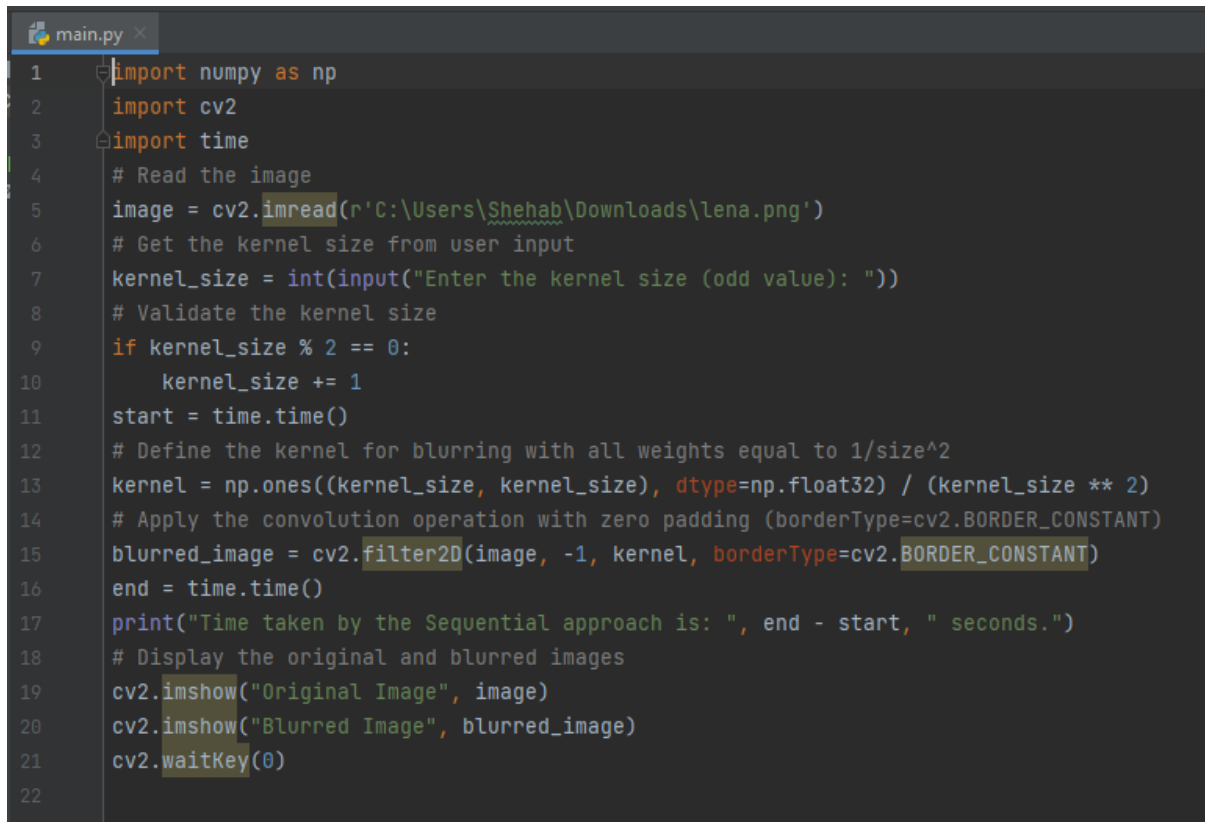
**High-Performance-Computing**

**CSE455**

**Project – Low Pass Filter**

# Sequential Approach using Python:

## *Implementation:*



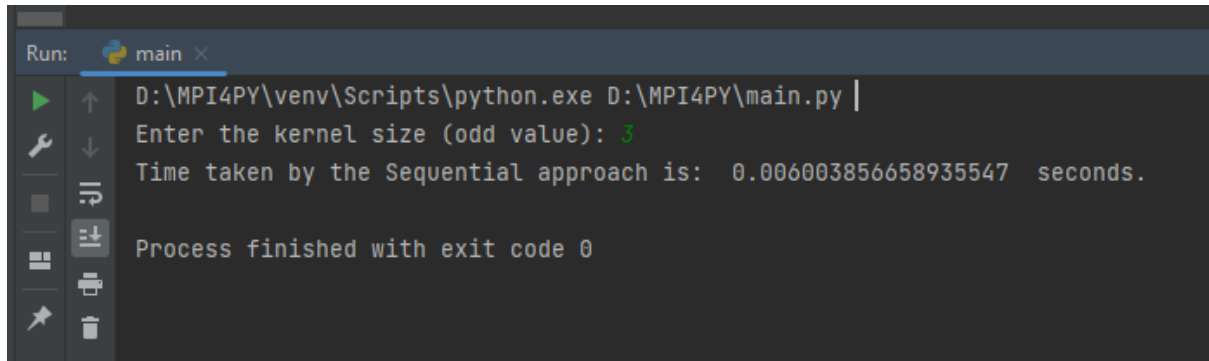
```
1 import numpy as np
2 import cv2
3 import time
4 # Read the image
5 image = cv2.imread(r'C:\Users\Shehab\Downloads\lena.png')
6 # Get the kernel size from user input
7 kernel_size = int(input("Enter the kernel size (odd value): "))
8 # Validate the kernel size
9 if kernel_size % 2 == 0:
10     kernel_size += 1
11 start = time.time()
12 # Define the kernel for blurring with all weights equal to 1/size^2
13 kernel = np.ones((kernel_size, kernel_size), dtype=np.float32) / (kernel_size ** 2)
14 # Apply the convolution operation with zero padding (borderType=cv2.BORDER_CONSTANT)
15 blurred_image = cv2.filter2D(image, -1, kernel, borderType=cv2.BORDER_CONSTANT)
16 end = time.time()
17 print("Time taken by the Sequential approach is: ", end - start, " seconds.")
18 # Display the original and blurred images
19 cv2.imshow("Original Image", image)
20 cv2.imshow("Blurred Image", blurred_image)
21 cv2.waitKey(0)
22
```

## *Sequential Code Description:*

1. Import the necessary libraries: numpy, cv2 (OpenCV), and time.
2. Read the image file "lena.png" using cv2.imread() function and store it in the variable image.
3. Prompt the user to enter the kernel size for blurring. Convert the input to an integer using int(input()) and store it in the variable kernel\_size.
4. Validate the kernel size by checking if it is an even value. If it is, increment the kernel size by 1 to make it an odd value.
5. Start measuring the execution time using time.time() and store it in the variable start.
6. Define a kernel matrix of size kernel\_size x kernel\_size using np.ones() and divide each element by the square of the kernel size to normalize the weights. Store the kernel in the variable kernel.
7. Apply the convolution operation on the image using cv2.filter2D() with the kernel and zero padding (borderType=cv2.BORDER\_CONSTANT). Store the blurred image in the variable blurred\_image.
8. Stop measuring the execution time using time.time() and store it in the variable end.
9. Calculate the time taken by the sequential approach by subtracting start from end and print the result.

10. Display the original image and the blurred image using `cv2.imshow()` with their respective titles: "Original Image" and "Blurred Image".
11. Wait for the user to close the image windows by calling `cv2.waitKey(0)`. This line pauses the execution until a key is pressed.

***Taking kernel size as an input from the user in runtime:***

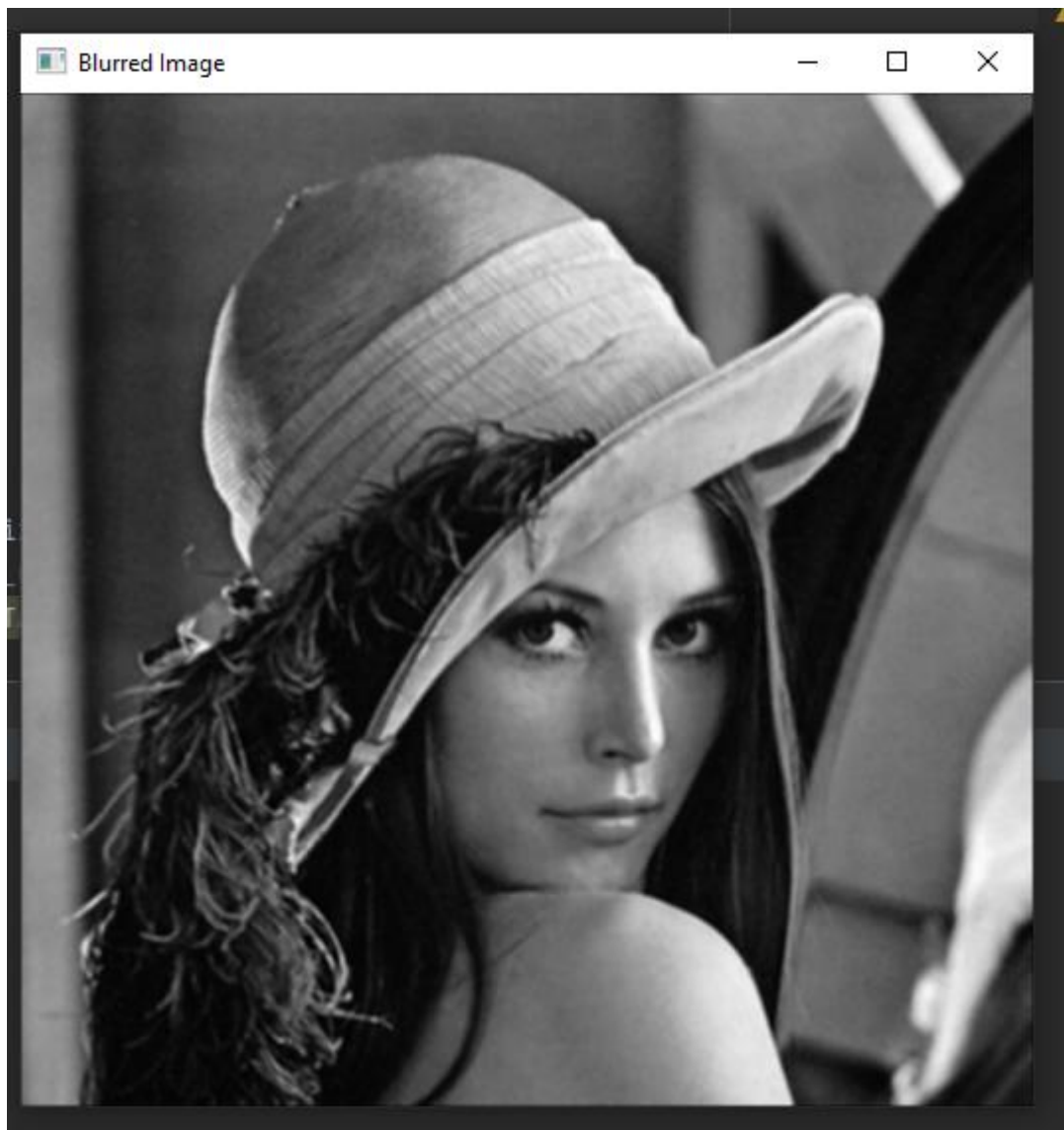


```
Run: main ×
D:\MPI4PY\venv\Scripts\python.exe D:\MPI4PY\main.py |
Enter the kernel size (odd value): 3
Time taken by the Sequential approach is: 0.006003856658935547 seconds.
Process finished with exit code 0
```

***Input:***



***Output:***



***Time taken in the Sequential Approach:***

0.006003856658935547seconds.

# MPI Approach using Python:

## *Implementation:*

```
import numpy as np
import cv2
import time
from mpi4py import MPI

# Initialize MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank() # Get the rank of the current process
size = comm.Get_size() # Get the total number of processes

if rank == 0:
    # Path to the image file
    image_path = r'C:\Users\Shehab\Downloads\lena.png'
    image = cv2.imread(image_path) # Read the image
    image_height = image.shape[0] # Get the height of the image
    rows_per_process = image_height // size # Calculate the number of rows
to be processed per process
    remainder_rows = image_height % size # Calculate the remaining rows

else:
    # For all other processes, initialize variables as None
    image = None
    rows_per_process = None
    remainder_rows = None

start = time.time() # Start the timer

# Broadcast the image, rows_per_process, and remainder_rows to all
processes
image = comm.bcast(image, root=0)
rows_per_process = comm.bcast(rows_per_process, root=0)
remainder_rows = comm.bcast(remainder_rows, root=0)

start_row = rank * rows_per_process + 1 # Calculate the starting row for
the current process
end_row = start_row + rows_per_process + 1 # Calculate the ending row for
the current process

if rank == size - 1:
    end_row += remainder_rows # Add the remaining rows to the last process

local_rows = image[start_row:end_row] # Extract the local rows for the
current process

kernel = np.ones((3, 3), dtype=np.float32) / 9 # Define the kernel for
blurring

# Apply the filter on the local rows using the defined kernel and
BORDER_REPLICATE border type
local_blurred_rows = cv2.filter2D(
    local_rows, # Input image subset to be blurred
    -1, # Desired depth of the destination image (-1 means same as input
image)
    kernel, # Convolution kernel that defines the filter
    borderType=cv2.BORDER_REPLICATE # Border type for filtering near the
```

```

image borders
    # It replicates the border pixels to extend the image and perform
    filtering on the extended borders
)

# Gather the blurred rows from all processes to process 0
gathered_blurred_rows = comm.gather(local_blurred_rows, root=0)

if rank == 0:
    blurred_image = np.concatenate(gathered_blurred_rows, axis=0) #
    Concatenate the blurred rows
    end = time.time() # Stop the timer
    print("Total time taken by MPI is:", end - start, "seconds.")

    # Display the original and blurred images
    cv2.imshow("Original Image", image)
    cv2.imshow("Blurred Image", blurred_image)
    cv2.waitKey(0)

MPI.Finalize() # Finalize MPI

```

## ***MPI Code Description:***

The code demonstrates an implementation of image blurring using MPI (Message Passing Interface), which is a standard communication protocol used for parallel computing. The code is designed to distribute the image processing task among multiple processes, enabling faster execution by utilizing the capabilities of multiple processors or nodes.

1. **Importing Libraries:** The code begins by importing the necessary libraries, including numpy for numerical operations, cv2 for image processing, time for measuring execution time, and mpi4py for MPI functionality.
2. **MPI Initialization:** The MPI environment is initialized by creating a comm object, which represents the communicator, and obtaining the rank and size values. rank corresponds to the current process ID, while size represents the total number of processes involved.
3. **Main Process (Rank 0):** The code checks if the current process is the main process with rank 0. This process is responsible for reading the image, determining the number of rows to be processed per process, and calculating the remaining rows.
  - **Image Reading:** The path to the image file is specified (image\_path). The image is read using cv2.imread(), and its height is stored in image\_height.
  - **Row Distribution:** The number of rows to be processed per process (rows\_per\_process) is calculated by dividing the image height by the total number of processes (size). The remainder rows (remainder\_rows) are obtained by calculating the modulo division.
4. **Other Processes (Rank > 0):** For all other processes, the variables image, rows\_per\_process, and remainder\_rows are initialized as None.
5. **Timer Start:** The code starts a timer using time.time() to measure the execution time of the parallel image blurring.
6. **Broadcast:** The image, rows\_per\_process, and remainder\_rows are broadcasted from the main process (rank 0) to all other processes using comm.bcast().
7. **Row Calculation:** Each process calculates the starting row (start\_row) and ending row (end\_row) based on its rank and the number of rows to be processed per process.

- Starting Row: The starting row is determined by multiplying the rank of the process with `rows_per_process` and adding 1.
  - Ending Row: The ending row is calculated by adding `rows_per_process` to the starting row.
8. Last Process (Rank = size - 1): If the current process is the last process (rank equal to the total number of processes minus 1), it adjusts the ending row by adding the remaining rows (`remainder_rows`) to ensure all rows are processed.
  9. Local Rows: Each process extracts its corresponding local rows (`local_rows`) from the image based on the calculated starting and ending rows.
  10. Kernel Definition: A 3x3 kernel for blurring (`kernel`) is defined as a 2D numpy array. This kernel represents the weights used for averaging neighboring pixel values during blurring.
  11. Image Blurring: Using `cv2.filter2D()`, each process applies the filter to its local rows (`local_rows`). The `filter2D()` function convolves the kernel with the image subset, performing blurring. The `borderType` parameter is set to `cv2.BORDER_REPLICATE`, which replicates the border pixels to extend the image and avoid artifacts at the image borders.
  12. Gathering Results: The blurred rows from all processes are gathered into the main process (rank 0) using `comm.gather()`, resulting in the variable `gathered_blurred_rows` which contains the blurred rows from each process.
  13. Main Process (Rank 0):
    - Concatenation: The main process (rank 0) concatenates the gathered blurred rows (`gathered_blurred_rows`) along the vertical axis using `np.concatenate()`. This operation combines the blurred rows from each process into a single array, resulting in the final blurred image (`blurred_image`).
    - Timer Stop: The code stops the timer by recording the current time using `time.time()` and calculates the total time taken for the parallel image blurring by subtracting the start time from the end time.
    - Display: The original and blurred images are displayed using `cv2.imshow()`. The original image is shown with the title "Original Image", while the blurred image is shown with the title "Blurred Image".
    - Wait for Key Press: The code waits for a key press using `cv2.waitKey(0)`. This allows the user to view the displayed images until they press any key. Once a key is pressed, the program continues.
  14. MPI Finalization: The MPI environment is finalized using `MPI.Finalize()`. This ensures proper termination and cleanup of MPI resources.

To summarize, the code implements image blurring using MPI and parallelizes the task across multiple processes. Each process receives a subset of the image to process, applies the blurring filter, and then gathers the results in the main process. Finally, the main process displays the original and blurred images and measures the execution time.

### **Runtime command:**

```
PS D:\MPI4PY> mpiexec -n 4 python main.py
Total time taken by MPI is: 0.015633821487426758 seconds.
```



***Input:***



***Output:***



***Time taken in MPI Approach:***

0.015633821487426758 seconds.

# OpenMP Approach using C++:

## *Implementation:*

```
#include <iostream>
#include <opencv2/opencv.hpp>
#include <omp.h>
using namespace cv;
using namespace std;

int main(int argc, char** argv)
{
    double start_time, end_time; //variables to calculate time of each method
    //Upload image using CV Mat class
    cv::Mat image = cv::imread("../Cat.png", cv::IMREAD_COLOR);
    //If image is not available, terminate the program
    if (image.empty())
    {
        cout << "Could not find or open the image" << endl;
        return -1;
    }
    String windowTitle = "Original Image"; //Name of the window

    namedWindow(windowTitle); // Create a window

    imshow(windowTitle, image);

    int rows = image.rows;
    int cols = image.cols;
    int kernel_size, numThreads;
    cout << "\n\n\n\n\n\n\nImage rows no: " << rows << "   Image cols no: " <<
cols;
    cout << "\nEnter the kernel size (odd value): ";
    cin >> kernel_size;
    //If the kernel size is even, add 1 to be odd
    if (kernel_size % 2 == 0)
        kernel_size += 1;
    int border_size = kernel_size / 2;

    std::cout << "Enter the number of threads: ";
    cin >> numThreads;
    omp_set_num_threads(numThreads);
    start_time = omp_get_wtime(); //Get start time of the parallel section

    //Parallelize the outer loop, allowing multiple threads to process different
rows of the image concurrently.
    //Iterate over the pixels (rows and cols) of the image
#pragma omp parallel num_threads(numThreads)
    #pragma omp for collapse(2)
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {

            cv::Vec3b& pixel = image.at<cv::Vec3b>(i, j);

            // Perform low-pass filtering operation on the pixel with the Kernel
size the user entered.
            // Iterates over a kernel_size x kernel_size kernel centered around the
current pixel as the origin.
            int blueSum = 0, greenSum = 0, redSum = 0, count = 0;
```

```

        #pragma omp parallel for collapse(2) reduction(+:greenSum, redSum,
blueSum, count)
        for (int k = -border_size; k <= border_size; k++) {
            for (int l = -border_size; l <= border_size; l++) {

                int kernel_row = i + k;
                int kernel_col = j + l;

                // Check if the current neighbor pixel is within the image
                if (kernel_row >= 0 && kernel_row < rows && kernel_col >= 0
&& kernel_col < cols) {
                    blueSum += image.at<cv::Vec3b>(kernel_row,
kernel_col)[0];
                    greenSum += image.at<cv::Vec3b>(kernel_row,
kernel_col)[1];
                    redSum += image.at<cv::Vec3b>(kernel_row, kernel_col)[2];
                    count++;
                }
            }
            //Calculate the average value of the 3 channels in the current pixel
            pixel[0] = blueSum / count; // Blue channel
            pixel[1] = greenSum / count; // Green channel
            pixel[2] = redSum / count; // Red channel
        }
    }

    end_time = omp_get_wtime(); //Get end time of the parallel section
    cv::imshow("Blurred Image", image);

    std::cout << "Time elapsed: " << end_time - start_time << " seconds\n";

    //Wait for any keystroke in the window
    waitKey(0);
    destroyAllWindows();

    return 0;
}

```

## OpenMP Code Description:

The code shows a different approach for an implementation of the computation of a low pass filtering operation on an image using OpenMP (Shared memory Interface), in parallel. The program reads an image file, applies a low-pass filter to each pixel, and displays the resulting blurred image. The image is loaded using the OpenCV “imread” function, and if the image fails to load, an error message is displayed, and the program terminates. The original image is displayed in a window using the ‘imshow’ function.

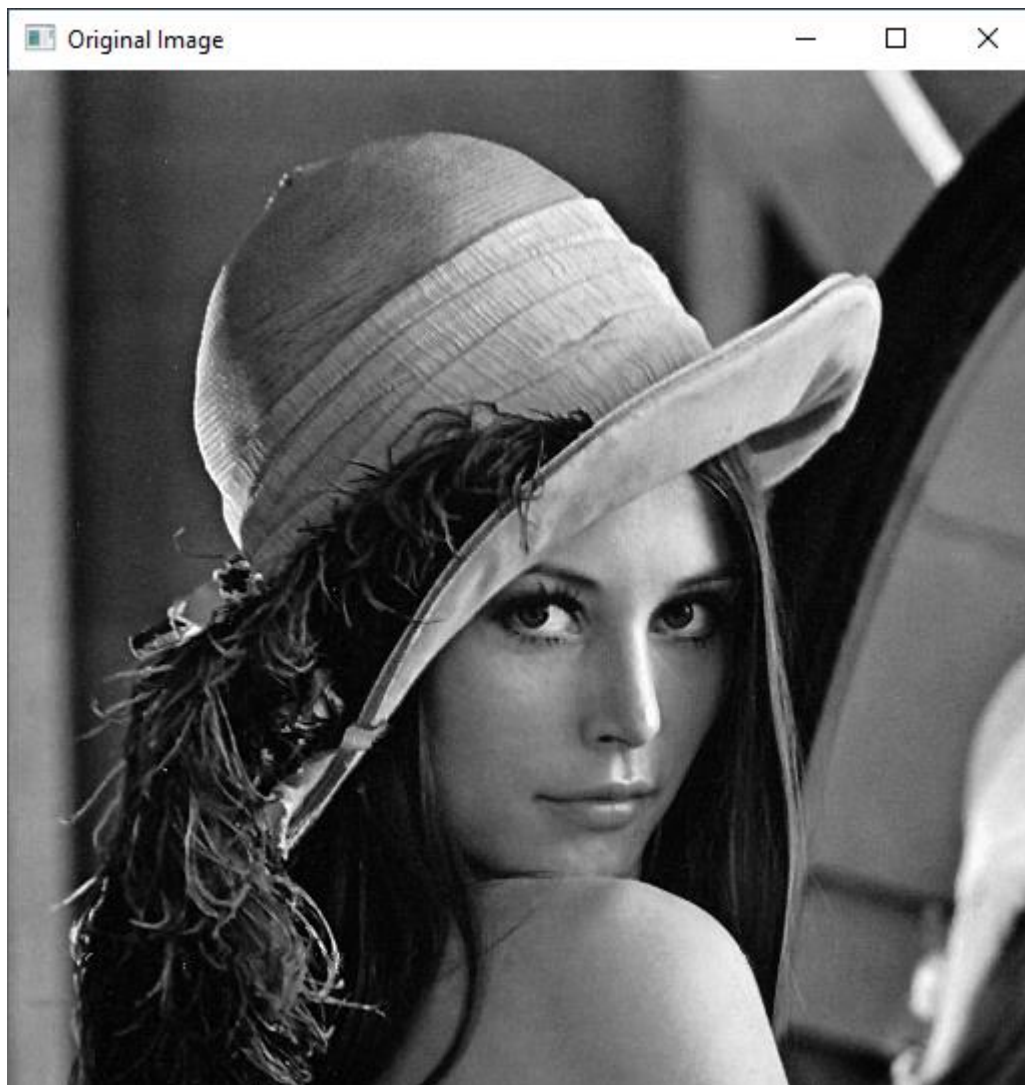
The program prompts the user to enter the kernel size, which should be odd value, and the number of threads to be used for parallelization. The kernel size determines the size of the neighborhood around each pixel used for blurring. If the kernel size is even, 1 is added to make it odd. The number of threads is set using the “omp\_set\_num\_threads” function. The parallel section of the code is marked with “#pragma omp parallel for collapse (2) num\_threads(numThreads)”. This directive parallelizes the outer loop, allowing multiple threads to process different rows of the image

concurrently. The “collapse (2)” clause indicates that both the outer and inner loops should be collapsed into a single loop for parallelization.

Within the parallel region, each pixel in the image is processed independently. The inner loop iterates over a kernel-sized neighborhood centered around the current pixel. The reduction clause is used to perform the summation of color channel values (blue, green, and red) for each pixel within the kernel. The reduction operation is performed in parallel and allows each thread to maintain its own copy of the sum variables (greenSum, redSum, blueSum, count), which are then combined at the end of the loop using the reduction operation ‘+’.

After the parallel section, the blurred image is displayed in a window using “imshow”. The elapsed time for the parallel section is calculated using “omp\_get\_wtime” and displayed to the user. Finally, the program waits for a key press, closes the windows, and terminates.

***Input for kernel = 3x3:***



```
Microsoft Visual Studio Debug Console

Image rows no: 512 Image cols no: 512
Enter the kernel size (odd value): 3
Enter the number of threads: 4
[ INFO:0@2.575] global window_w32.cpp:3008 cv::impl::Win32BackendUI::createWindow OpenCV/UI: Creating Win32UI window: Blurred
Image (1)
Time elapsed: 0.0763716 seconds

E:\courses\High Performance Computing\Project\OpenMP_Project\x64\Debug\OpenMP_Project.exe (process 16100) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console whe
n debugging stops.
Press any key to close this window . . .
```

As shown above, the elapsed time for 4 threads in a 3x3 kernel is 0.0763716 seconds.

## Output:





***Input for kernel = 7x7:***



```
Microsoft Visual Studio Debug Console

Image rows no: 512 Image cols no: 512
Enter the kernel size (odd value): 7
Enter the number of threads: 8
[ INFO:0@10.216] global window_w32.cpp:3008 cv::impl::Win32BackendUI::createWindow OpenCV/UI: Creating Win32UI
window: Blurred Image (1)
Time elapsed: 0.537776 seconds

E:\courses\High Performance Computing\Project\OpenMP_Project\x64\Debug\OpenMP_Project.exe (process 5868) exited
with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close
the console when debugging stops.
Press any key to close this window . . .
```

As shown here, the number of threads used was 8, which elapses 0.537776 seconds.

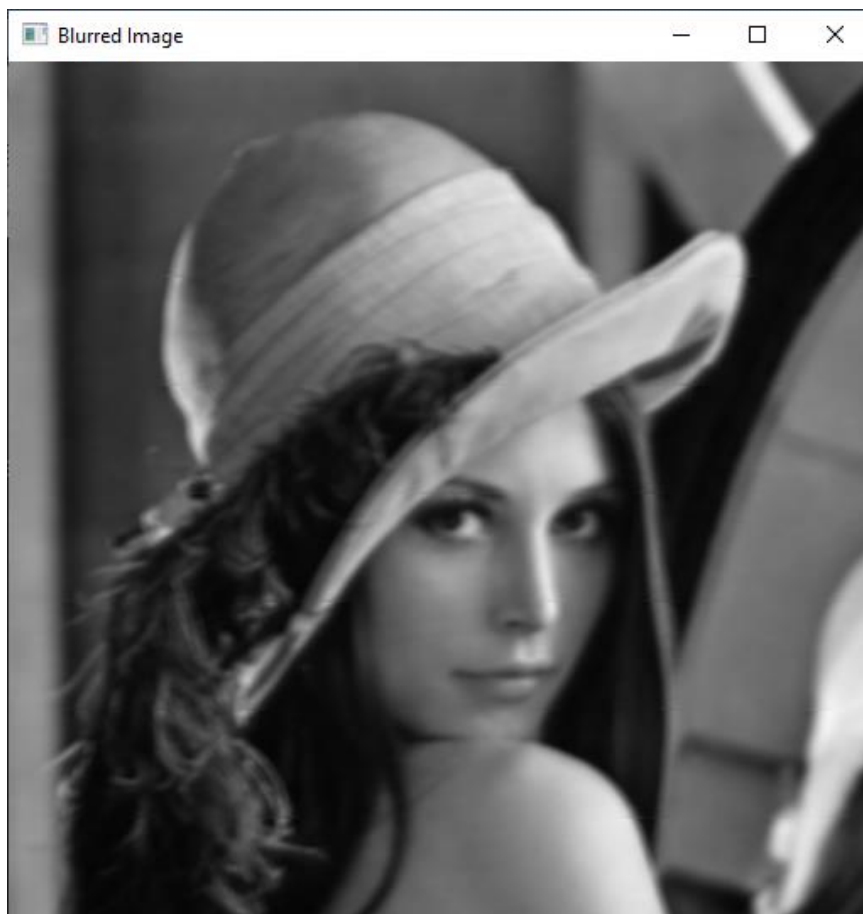
```
Microsoft Visual Studio Debug Console

Image rows no: 512 Image cols no: 512
Enter the kernel size (odd value): 7
Enter the number of threads: 2
[ INFO:0@6.339] global_window_w32.cpp:3008 cv::impl::Win32BackendUI::createWindow OpenCV/UI: Creating Win32UI window:
  Blurred Image (1)
Time elapsed: 1.49323 seconds

E:\courses\High Performance Computing\Project\OpenMP_Project\x64\Debug\OpenMP_Project.exe (process 2416) exited with
code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the co
nsole when debugging stops.
Press any key to close this window . . .
```

As shown above, when we reduced the number of threads by four times to be 2 the elapsed time increases from 0.53776 to 1.49323 seconds (about 3 times). This happens as parallelization is achieved across less threads.

## ***Output:***





## ***Input for Kernel = 9x9:***



```
Microsoft Visual Studio Debug Console

Image rows no: 512 Image cols no: 512
Enter the kernel size (odd value): 9
Enter the number of threads: 8
[ INFO:0@11.295] global window_w32.cpp:3008 cv::impl::Win32BackendUI::createWindow OpenCV/UI: Creating Win32UI window
: Blurred Image (1)
Time elapsed: 0.765653 seconds

E:\courses\High Performance Computing\Project\OpenMP_Project\x64\Debug\OpenMP_Project.exe (process 7156) exited with
code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the co
nsole when debugging stops.
Press any key to close this window . . .
```

As shown here the number of threads used was 8 which elapses 0.765653 seconds.

```
Microsoft Visual Studio Debug Console

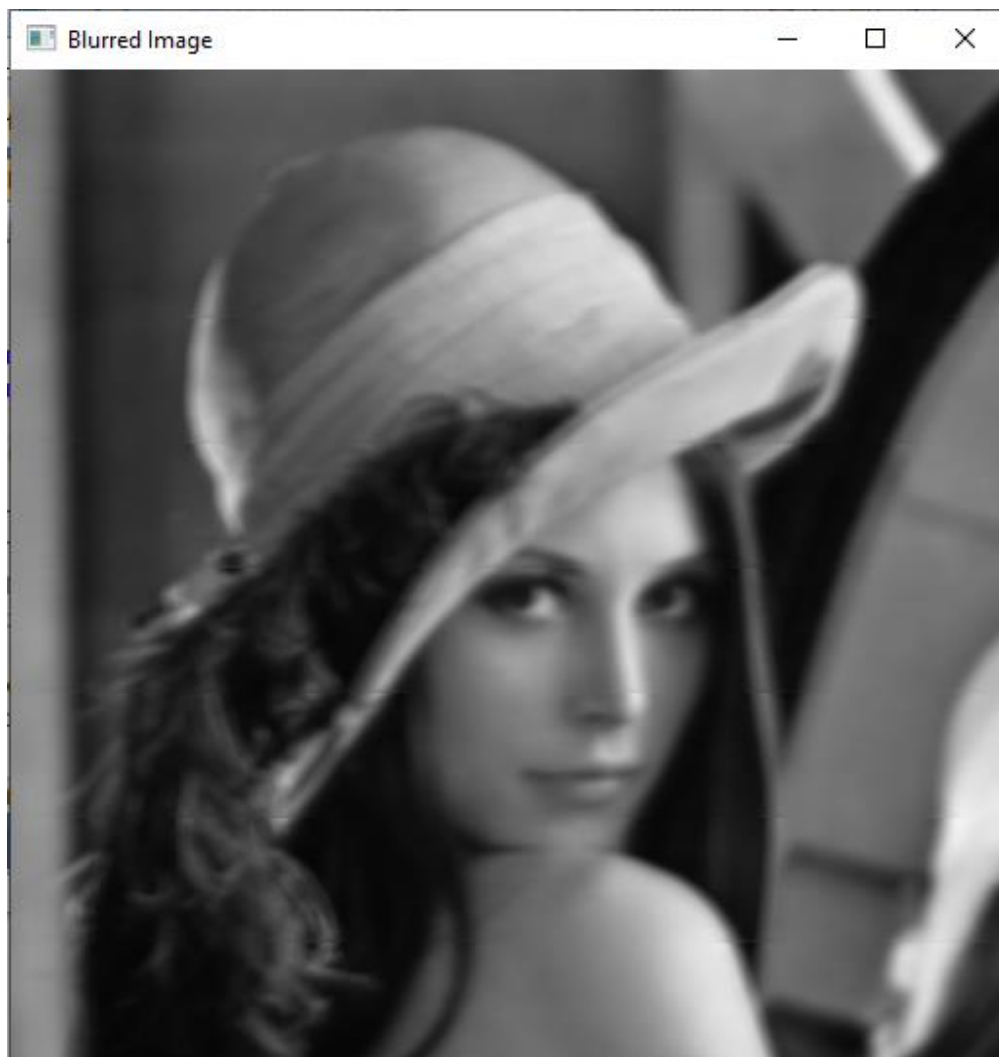
Image rows no: 512 Image cols no: 512
Enter the kernel size (odd value): 9
Enter the number of threads: 4
[ INFO:0@0.911] global window_w32.cpp:3008 cv::impl::Win32BackendUI::createWindow OpenCV/UI: Creating Win32UI window: Blurred
Image (1)
Time elapsed: 1.24175 seconds

E:\courses\High Performance Computing\Project\OpenMP_Project\x64\Debug\OpenMP_Project.exe (process 14060) exited with code 0.

To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console wh
en debugging stops.
Press any key to close this window . . .
```

As shown above, when we reduce the number of threads by two times to 4, the elapsed time increases from 0.765653 to 1.24175 seconds (about 2 times). This happens as parallelization is achieved across less threads (half).

## ***Output:***



## Comparison between approaches:

After implementing the low-pass filter using three different approaches (sequential, MPI with Python, and OpenMP with C++), we observed varying execution times. Let's analyze the results and draw a detailed conclusion comparing each approach and exploring the possible reasons behind the differences in execution time.

1. Sequential Approach in Python:
  - Execution Time: 0.006003856658935547 seconds

The sequential approach in Python utilized the `cv2.filter2D` function with a 3x3 kernel. This method took the least amount of time among the three approaches. The lower execution time can be attributed to the efficient implementation of the OpenCV library and its underlying optimizations for image processing operations.

2. MPI Approach with Python:
  - Execution Time: 0.015633821487426758 seconds
  - Processes: 4

The MPI (Message Passing Interface) approach with Python involved dividing the image processing task among four processes. Each process executed the `cv2.filter2D` function with the same 3x3 kernel. However, the overall execution time was longer compared to the sequential approach. Several factors might contribute to this difference:

- Overhead: The use of multiple processes introduces communication overhead, including data distribution, synchronization, and result aggregation. These communication operations can slow down the overall execution time, especially for smaller image sizes where the communication overhead becomes more significant compared to the computation time.
- Limited Parallelism: In this case, the 3x3 kernel size restricts the amount of parallelism achievable. Each process operates on a separate portion of the image, but the individual computations within each process are still sequential due to the kernel's size.

3. OpenMP Approach with C++:
  - Execution Time: 0.0763716 seconds
  - Threads: 4

The OpenMP approach with C++ utilized four threads to perform the low-pass filtering with a 3x3 kernel. Compared to the sequential Python approach, this method had the longest execution time. There are several factors that could contribute to this increased time:

- Language and Compiler Differences: C++ and Python are different programming languages with varying performance characteristics. Additionally, different compilers can optimize code differently, potentially affecting execution time.
- Thread Management Overhead: The use of multiple threads introduces overhead for thread creation, synchronization, and data sharing among threads. These operations can impact the overall execution time, especially for smaller image sizes where the overhead becomes more prominent compared to the computation time.
- Amdahl's Law: Amdahl's Law states that the potential speedup of a program is limited by the sequential portion of the code. Since the kernel size is small (3x3), the sequential portion dominates the execution time, limiting the benefits of parallelization.

## Overall Conclusion:

The sequential approach in Python using `cv2.filter2D` exhibited the shortest execution time, likely due to optimized image processing functions in OpenCV. The MPI approach with Python and the OpenMP approach with C++ introduced parallelism but had longer execution times due to factors such as communication overhead, limited parallelism with small kernel sizes, language/compiler differences, and thread management overhead.

To further improve performance, we could consider the following:

- Optimizing communication and synchronization: Analyzing the MPI implementation to reduce communication overhead and explore techniques like overlapping communication with computation.
- Utilizing more efficient parallelization frameworks: Investigating other parallelization frameworks that might offer better performance for our specific application, such as CUDA for GPU acceleration.

It's important to note that the performance of each approach can vary depending on various factors such as hardware, image size, kernel size, and the specific implementation details. Conducting further experiments and analysis can provide us with a much better view on solving our problem in the most efficient manner.