Task 1:

This part of the assignment is aimed to analyze the performance of serial and parallel version of a program. The program deals with image processing where gray-scaling and flipping of image is done using C++ programming language. The program uses the libppm.h as header file for execution.

The following table shows the total time taken by the program to gray-scale and flip a .ppm image of following dimension:
Height = 6400
Weight = 6300

In this task, the program is parallelized using OpenMP library where two for loops each on to_grayscale() method and flip() method is tagged with following command
#pragma omp parallel for

<u>To run the serial version of program following steps needs to be followed:</u>
Compile: g++ -o task_1_ppm_serial_processing task_1_ppm_serial_processing.cc
Run: ./task_1_ppm_serial_processing ttu_tile.ppm ttu_tile_grayed_flipped.ppm

<u>To run the parallel version of program following steps needs to be followed:</u>
Compile: g++ -fopenmp -o task_1_ppm_parallel_processing_openmp task_1_ppm_parallel_processing_openmp.cc
Run: ./task_1_ppm_parallel_processing_openmp ttu_tile.ppm ttu_tile_grayed_flipped.ppm

The following execution times are recorded by running each program for 20 times.

Serial Version	**Parallel Version**
Total time: 1.34418 Total time: 1.33777 Total time: 1.33721 Total time: 1.33531 Total time: 1.33537 Total time: 1.33642 Total time: 1.33633 Total time: 1.33579 Total time: 1.33499 Total time: 1.33483 Total time: 1.33659 Total time: 1.33763 Total time: 1.3397 Total time: 1.33855 Total time: 1.33874 Total time: 1.33908	Total time: 2.45968 Total time: 2.45476 Total time: 2.46145 Total time: 2.45346 Total time: 2.48379 Total time: 2.46787 Total time: 2.47627 Total time: 2.47113 Total time: 2.47557 Total time: 2.47673 Total time: 2.45976 Total time: 2.44199 Total time: 2.46265 Total time: 2.44853 Total time: 2.4433 Total time: 2.46013

Total time: 1.33988 Total time: 1.33757 Total time: 1.33636 Total time: 1.33602	Total time: 2.44969 Total time: 2.44046 Total time: 2.43152 Total time: 2.42831
Average Time: 1.337416	**Average Time: 2.4573525**

Answers for the questions:

1. What is the average time for twenty runs of the serial version of the code (i.e. with pragmas commented)?

=> Average time for twenty runs of the serial version of the code is *1.337416.*

2. What is the average time for twenty runs of the parallel version of the code?

=> Average time for twenty runs of the parallel version of the code is *2.4573525.*

3. Calculate the speedup of the parallel version. Is the parallel code significantly faster?

=> After analyzing the data collected above in the table, parallel version of the code seems slower than the serial version of the code as average time for parallel version, *2.4573525* is greater than average time for serial version, *1.337416*.

The parallel version could not speedup because of the communication overhead among the processors. The program is ran on the system which has 4 processors (in the provided VM). The openMP library assigned task for the 4 processes by partitioning the input image into 4 portions (row-wise) so that each processor process the assigned portion parallely. But, the execution for parallel version took more time because those 4 processes needs to communication with each other during the execution. And this cause the parallel version to take more time.

4. The Methodology section above described how you decompose the image processing routines to parallelize them. Obviously, OpenMP did all the work for you. How many rows do you think OpenMP assigned to each processor?

Hint: have your code print the image's height, and also have you code print out the number of threads in the computation (the function omp_get_thread_num() returns the number of threads).

=> For ttu_tile.ppm image: omp_get_thread_num() returned 4 threads. And the height of the image is 6400.

So, 6400/4 = 1600 rows are assigned for each thread or processor.

Task 2:

This part of the assignment specifically deals with the same problem statement as described in the Task 1 but in different way. This part is designed specially to implement the concept of Pipelining. Pipelining means passing the result of first operation to the second operation so that the second operation can perform its desired task on the input whereas the first operation can perform its desired task on the other input at the same time. So, that both of the operation executes in parallel. The program uses the libppm.h as header file for execution.

For this part, two methods; enqueue and dequeue are created to perform pipelining. Where enqueue method puts the row which is result of to_grayscale method (method to convert RGB into grayscale) and dequeue method is called form the flip method so that the dequeue method will provide the grayscaled row from the flip method.

The following table shows the total time taken by the program to gray-scale and flip a .ppm image of following dimension:
Height = 6400
Weight = 6300

In this task, the program is parallelized using OpenMP library where two for loops each on to_grayscale() method and flip() method is tagged with following command
#pragma omp parallel for

<u>To run the serial version of program following steps needs to be followed:</u>
Compile: g++ -o task_2_ppm_serial_processing_pipelining task_2_ppm_serial_processing_pipelining.cc
Run: ./task_2_ppm_serial_processing ttu_tile.ppm ttu_tile_grayed_flipped.ppm

<u>To run the parallel version of program following steps needs to be followed:</u>
Compile: g++ -fopenmp -o task_2_ppm_parallel_processing_pipelining task_2_ppm_parallel_processing_pipelining.cc
Run: ./task_2_ppm_parallel_processing_pipelining ttu_tile.ppm ttu_tile_grayed_flipped.ppm

The following execution times are recorded by running each program for 20 times.

Serial Version	**Parallel Version**
Total time: 1.50507 Total time: 1.49095 Total time: 1.48991 Total time: 1.48964 Total time: 1.49064 Total time: 1.49125 Total time: 1.49014 Total time: 1.49046 Total time: 1.51582	Total time: 1.18926 Total time: 1.18968 Total time: 1.1901 Total time: 1.19066 Total time: 1.18947 Total time: 1.19701 Total time: 1.19004 Total time: 1.18907 Total time: 1.19031

Total time: 1.49435 Total time: 1.4929 Total time: 1.49506 Total time: 1.50956 Total time: 1.49304 Total time: 1.49147 Total time: 1.49051 Total time: 1.49168 Total time: 1.49153 Total time: 1.49155 Total time: 1.49249	Total time: 1.27347 Total time: 1.19128 Total time: 1.18917 Total time: 1.19024 Total time: 1.18983 Total time: 1.18975 Total time: 1.18959 Total time: 1.18945 Total time: 1.18994 Total time: 1.18957 Total time: 1.19228
Average Time: 1.494401	**Average Time: 1.1945085**

Answers for the Questions:

1. What is the average time for twenty runs of the serial version of the code (i.e. with pragmas commented)?

=> Average time for twenty runs of the serial version of the code is *1.494401.*

2. What is the average time for twenty runs of the parallel version of the code?

=> Average time for twenty runs of the parallel version of the code is *1.1945085.*

3. Calculate the speedup of the parallel version. Is the parallel code significantly faster?

=> After analyzing the total time taken by each version of the programs for twenty times, it is seen that parallel version of the program is approximately 30% faster than the serial version.

Calculation:

= *(1.494401 - 1.1945085) / 1.494401*

= 0.2998925 * 100

= 29.98925

= 30% approx

4. In the Methodology section above, pipelining, which is a method of functional decomposition, is described. Describe how you would implement the solution using only domain decomposition.

=> In domain decomposition, problem should be decomposed according to the input data. So, I would like to use following steps for the decomposition.

First, decision about how input data elements should be divided among cores should be made. In general, when dividing the rows among processors, we should divide the work equally. So, if the image consists of n rows, and there are p processors available, then each processor should get roughly n/p rows. At the end each image will be divided into chunks where each chunk consists of n/p rows of pixels. Then we assign each chunk of rows to a processor.

Second, decide which tasks each core should be doing. Each processor then assigned to perform grayscaling on the assigned chunks and the result will be passed to the for flipping to the another processor where the first processor can do grayscaling for the another chunk.