Contents

CAB401

Report of High Performance and Parallel Computing Project

Name: Daniel Pham

ID: N10640754

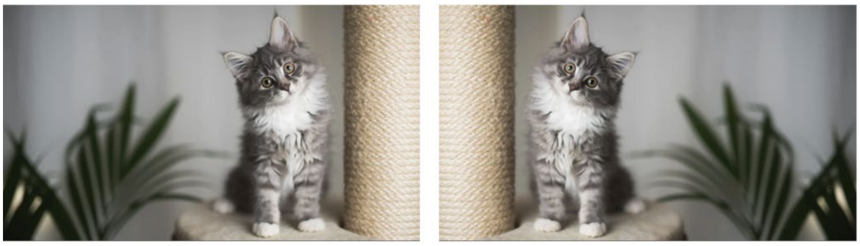
1. Analysis of original application

Function:

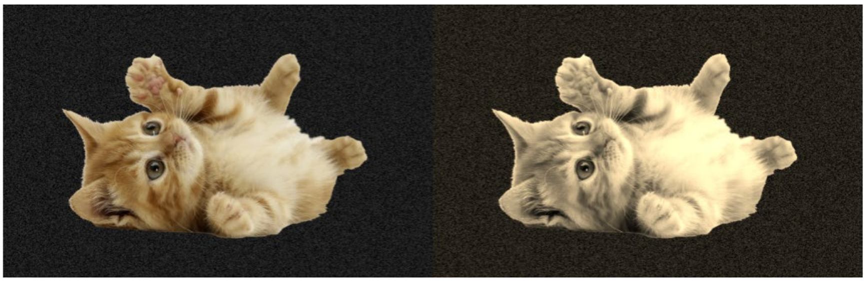
This application (named by me: **FilterPipeApp**) implements 1 class called Pipeline to process the console's print output, and file saving locations and 7 classes (called **Node Class**) to demonstrate 7 types of function that transform the image (each Node Class contains a "process()" function to readjust the image's appearance):

- **Grayscale**: Apply grayscale effect to the given image

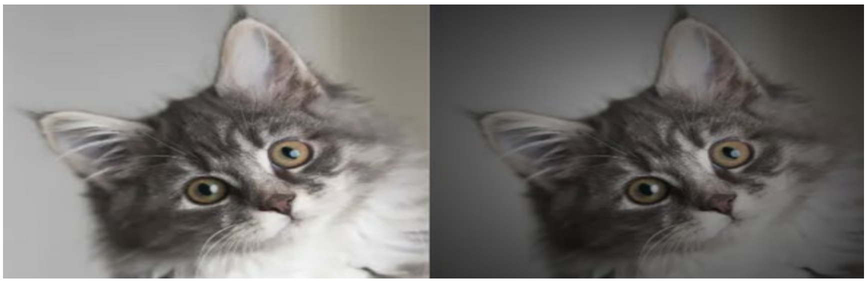
- **Flip**: Flips a picture horizontally or vertically



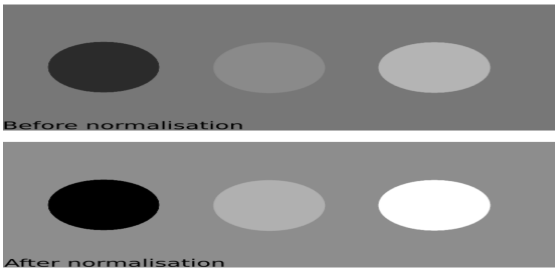
- **Sepia**: Apply sepia filter to the given image



- **Vignette**: Apply an effect which makes pixel getting darker based on the distance between the pixel and the image centre (the further the darker)

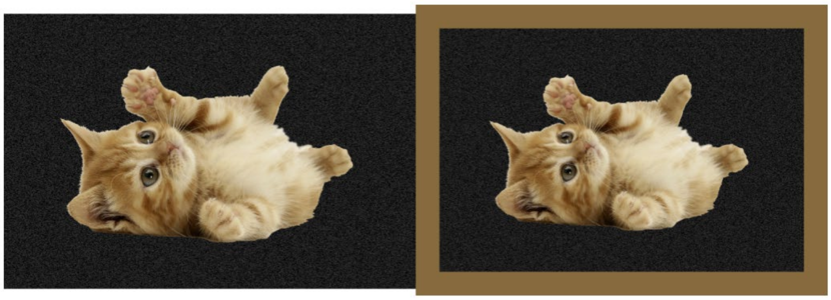


- **Normalise**: change the range of pixel intensity values



- **Resize**: Adjust the size of an image

- **Add border**: Create borders with customed size and colour to the image



To run the software, a pipe.txt file is given as an input which is filled with instructions to run each Nodes in a desired sequence. Below is an example pipe.txt file with written node instructions.

```
node=resize newSize=4000x2200
node=grayscale
node=add_border borderSize=100 borderColor=100,40,30
node=sepia
node=vignette
node=normalise
node=flip direction=[vertical]
```

Example image processing according to above pipe.txt file: (Appendix)

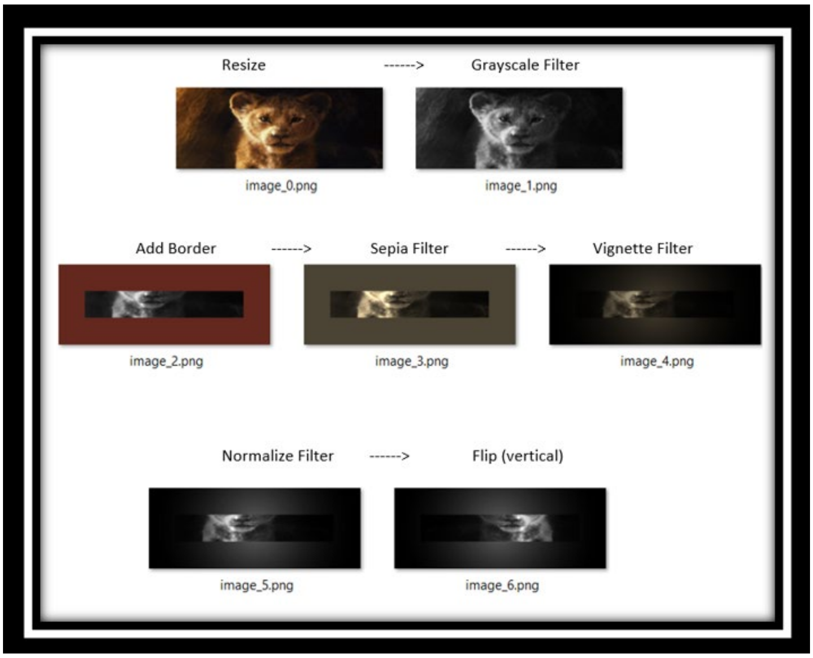
INPUT: 3840 X 2160 image

grow.png

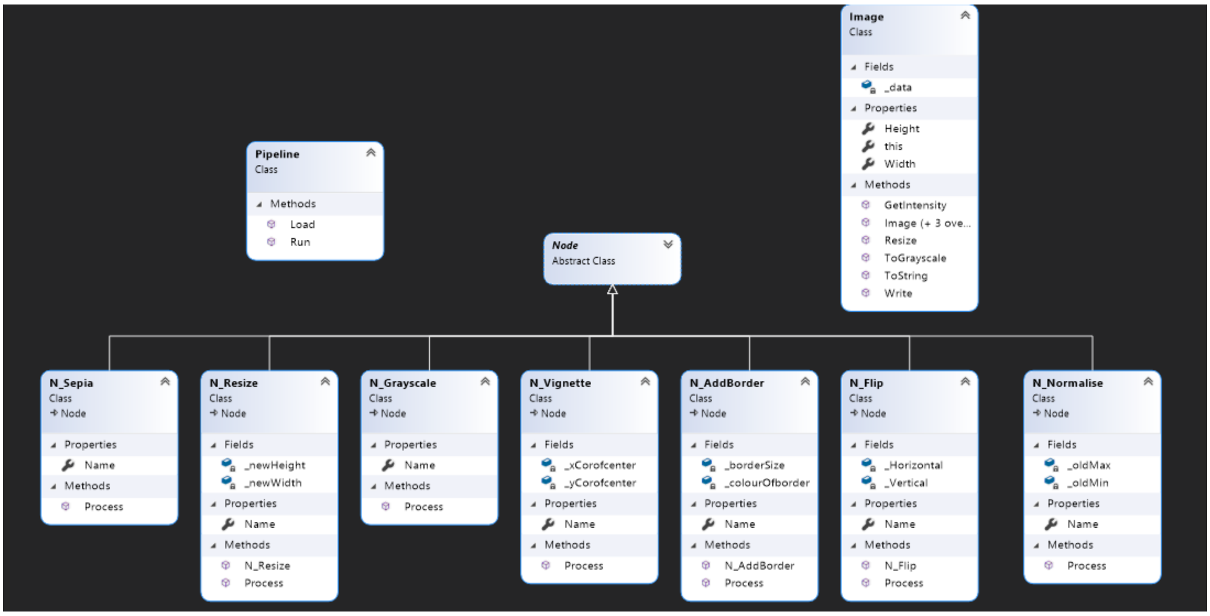
FINAL OUTPUT:

image_6.png

PROCESSING:



An overall view of applications' Architecture (in Class Diagram):



Pipeline Class: when running will read through the instructions given from pipe.txt file then sequentially choose out and toggle suitable Node Class that described in the pipe.txt file to make changes to the input image.

Image Class: Illustrated the image input and its' statics

Potential parallelism analysis:

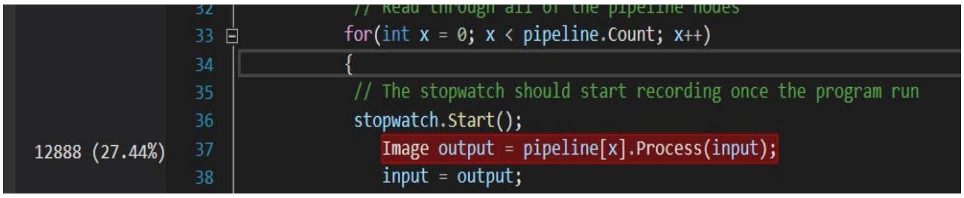
Current Scalable parallelism discussion:

- The application’s problem size is possibly expandable/scalable due to the variety of image input and given Node classes. With 7 types of photo-adjust features we can create enormous types of pipe.txt file with unique combinations to run on many kinds of image input.

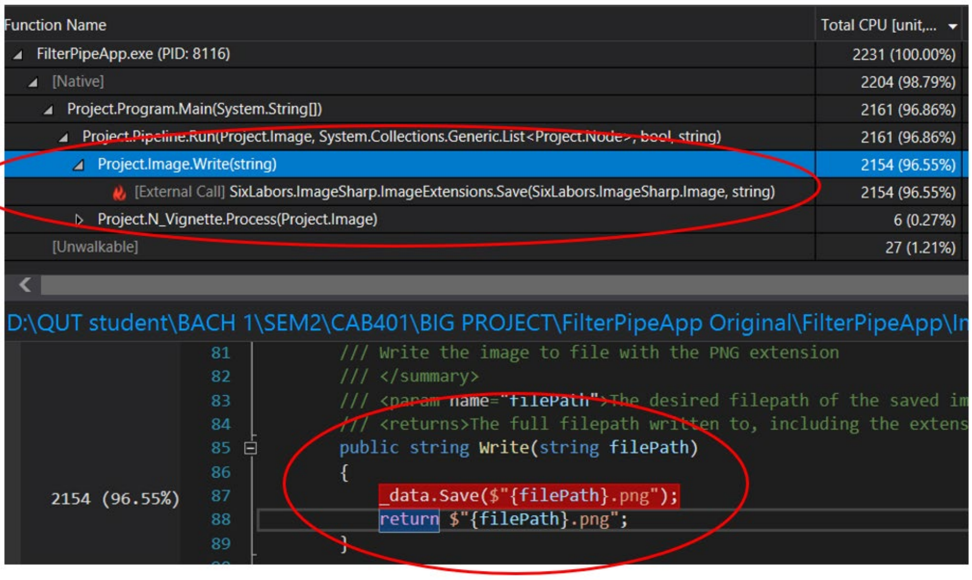
- The scalable parallelism of this application is existed due to many for loops inside **Process()** functions in **each Node Class** and the **Run()** function from **Pipeline Class.**

Areas that parallelism might exist (based on High CPU usage**):**

- 1st Area: The **Run Function** from Pipeline Class has a For Loop to go through then process each type of unique node from the pipe.txt file in sequential order.



By using Performance Profiler from Visual Studio, I have noticed that this function creates a high CPU usage by running “**SixLabors.ImageSharp.ImageExtensions.Save() –** code reference“ an external Function which is inside the **Write()** function from **Image Class.**



’s Write() function is implemented in Pipeline Class to write/save the image output into a given directory, and this function has slowed down the whole processing sequential (other node instructions need to wait for previous changes saved then be able to continue)

```
D:\QUT student\BACH 1\SEM2\CAB401\BIG PROJECT\FilterPipeApp Original\FilterPipeApp\Pipeline.cs:26
                              if(saveDir != "")
                              {
                                  // Create a folder to store processed nodes
         1 (0.02%)                System.IO.Directory.CreateDirectory(saveDir);
                                  // Create sub-file
      3144 (75.76%)               output.Write($"{saveDir}/image_{x}");
                              }
                          }
```

Due to the logic of the Run() in Pipeline class which is saving every single change of the image into a "out folder", that means if we got **N** node instruction lines found inside pipe.txt the app will need to save **N** transformation version of the image. Therefore, that's the reason why the CPU usage so high in this part of code. The parallelism might be implemented in this function to parallelly save all changes in an out folder.

- 2nd Area: Nodes such as **Normalise, Vignette, Grayscale, Sepia, AddBorder, and Flip,** each contains a **Process**() function which includes one or more For Loop iterations, these loops/nested For Loop can create high CPU usage (Totally 22.24% CPU usage from all above-mentioned Nodes).

Project.N_Normalise.Process(Project.Image)	2459 (7.37%)
Project.N_Vignette.Process(Project.Image)	2064 (6.19%)
Project.N_Grayscale.Process(Project.Image)	1101 (3.30%)
Project.N_Sepia.Process(Project.Image)	704 (2.11%)
Project.N_AddBorder.Process(Project.Image)	572 (1.72%)
Project.N_Flip.Process(Project.Image)	518 (1.55%)

But according to current logic of the For Loops in the application, it only sequentially goes through each pixel step by step on one core (No parallelism applied in the current state). For example, the Flip Node: (the logic of stepping through pixel in other nodes are pretty the same)

```
// Go through each image's pixels
for (int x = 0; x < input.Width; x++)
{
    for (int y = 0; y <= input.Height - 1; y++)
    {
        // Exchange each pixels at the closer y-position with the respectively pixels at further y-position
        Rgba32 oldPixel = input[x, input.Height - 1 - y];
        Rgba32 newPixel = new Rgba32(
            r: oldPixel.R,
            g: oldPixel.G,
            b: oldPixel.B,
            a: oldPixel.A
        );
        output[x, y] = newPixel;
    }
}
```

therefore, a parallel programming technique can be implemented in this situation to create parallel pixel processing effect.

Safe spots discussion:

1st Area:

The For Loop inside Run() function has a form like this

```
Image b = input image;  // "b" value need to be preserved

Pipe [ N ];  //N is the number of lines in text file

For (i = 0, i < N, i++)

{

        Image a = Pipe[ i ].Process( b );

        b = a;       }
```

When i = 0, 'b' will be read then be assigned with the 'a' value (Pipe[0].Process(b))

When i = 1, 'b' will be read then be assigned with the 'a' value (Pipe[1].Process(b)) and it will continue until the control dependency ended (i > N), therefore the value b is read before it got overwritten by a later statement. The logic behind this is to record the changes happened from 1st line of node instruction and from the record changes the application will continue apply more changes from later instruction to that record, the data dependency from each iteration is expectedly high ("a" value will be written in each iteration) => The data is anti-dependence (R -> W) => **This loop is unsafe to parallel.**

2nd Area:

- **N_Grayscale:**

Algorithm:

```
For(x=0, x < Image_Width, x++)

        For(y=0, y < Image_Height, y++)

                {

                //Read  RGBA value of input image's pixel in location x, y then assigned to "a"

                Rgba A = Input_Image.GetIntensity(x,y) ; //this variable doesn't need to be preserved

                //Write RGBA value to output image's pixel in location x, y

                  Output_Image[x,y] = A ;

                }

Return Output_Image;
```

When x=0, y=0 **read** RGBA of Input_Image's pixel in [0,0] then **read** again (through "A"-dummy variable) in next statement

When x=0; y=1 **read** RGBA of Input_Image's pixel in [0,1] then **read** again (through "A") in next statement

- ⇨ During the control dependencies (x < width and y < Height), the RGBA value of input/output image's pixel location [x,y] will only be read once and never got overwritten(no need to be preserved), therefore, the data dependency of this value is input dependence. Furthermore, the order of processing each pixel does not matter, the tasks can be done individually => the for loop is safe to parallel in both inner and outer.
- ⇨ This analysis can apply **to Nodes: Vignette, Flip, Add_Border and Sepia** since they basically use the same logic for their loops and use of variables, therefore, **they are all safe to parallel.**

- **N_Normalise:**
  This class's process() function got 2 parts of nested loop that might be parallelable.

Part 1

```
// Go through each pixel of image
for (int x = 0; x < input.Width - 1; x++)
{
    for (int y = 0; y < input.Height - 1; y++)
    {
        // Check the old minimum value of pixel image
        if ((int)(output[x, y].R) < _oldMin)
        {
            _oldMin = (int)(output[x, y].R);
        }
        // Check the old maximum value of pixel image
        if ((int)(output[x, y].R) > _oldMax)
        {
            _oldMax = (int)(output[x, y].R);
        }

    }
}
```

Part 2

```
// Go through each pixel of image
for (int x = 0; x < input.Width - 1; x++)
{
    for (int y = 0; y < input.Height - 1; y++)
    {
        // Normalise number of the old image based on the intensity of image
        double _numOld = output.GetIntensity(x, y);
        Rgba32 oldPixel = input[x, y];
        // Formula to find a new normalise number for the pixel image
        double relavtiveBrightness = Math.Abs((_numOld - _oldMin) / (_oldMax - _oldMin));
        double numNew = Math.Abs(_newMin + relavtiveBrightness * (_newMax - _newMin));
        // Apply new normalise number to each pixel red, green and blue colors
        Rgba32 newPixel = new Rgba32(
            r: (byte)Math.Min(255, Math.Max(0, numNew)),
            g: (byte)Math.Min(255, Math.Max(0, numNew)),
            b: (byte)Math.Min(255, Math.Max(0, numNew)),
            a: oldPixel.A
        );
        output[x, y] = newPixel;
    }
}
return output;
```

  - Part 2 got the same logic of the data dependency from N_Grayscale (Due to the tendency of visiting each pixel once to read value then stores in respectively output pixel and all variables inside the loop don't need to be preserved) => this part is safe to parallel
  - Meanwhile Part 1 needs to Read then Write Min/Max RGBA pixel values quite often, therefore, there are many read/write activities happened (to find smallest/largest value) on these 2 variables => the data is anti-dependence => Part 1 is not very safe for parallelism.

2. Use of tools and techniques

Possible strategies to parallel the overall app:

- For 1st Area the block of code would need a restructure logic and rewrite the algorithm to make it work parallelly, as mentioned above the Write() function is a bottleneck, so the strategy would be temporarily saving all Image changes into a memory then parallelly permanently save it later in a sequent (inside **a separate For Loop - peeling**).
- For 2nd Area: Loops from this part can be transformed into an explicitly parallel form. Instead of reconstructing the whole working style of the app we should focus more on parallelising the process() function from each node, which could achieve effective speedup in scalable parallelism. If individual process can speedup, so does the whole application.
- About part 1 of N_Normalise, beside the unsafety of Min/Max values we can use multiple individual min/max value investigations in parallel and finally compare results all together from each thread. (But this strategy is not worthy as those previous 2 to deploy since the CPU burnt mostly on them)

Technologies used

Hardware: 8 cores for better data investigation

System Information

Current Date/Time: Saturday, 23 October 2021, 10:58:14 AM
Computer Name: MSI
Operating System: Windows 10 Home 64-bit (10.0, Build 19043)
Language: English (Regional Setting: English)
System Manufacturer: Micro-Star International Co., Ltd.
System Model: GV62 8RE
BIOS: E16JEIMS.105
Processor: Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz (8 CPUs), ~2.3GHz
Memory: 8192MB RAM
Page file: 11421MB used, 5322MB available
DirectX Version: DirectX 12

Software:

For better performance and analysis capability **Visual Studio 2019** has been chosen to deploy my project (mainly for its' Performance Profiler tools to analyse code's highest CPU usage areas). **C#** is main language for this project, due to the assistance of "`System.Threading.Tasks library`" has made my parallel processing job much simpler and safer. The design of function: **Parallel.For()** from above library give flexible capability of turning a sequential loop to paralleled version and an option of changing cores number for testing (using `new ParallelOptions {MaxDegreeOfParallelism = Core_Num } parameter`).

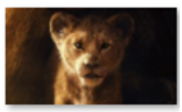
Check Result:

To make sure the results are the same in each change we can simply check the output images from "Out Folder". Since each method return a unique output from given input and was detailly recorded **step-by-step.**

Input: (pipe.txt file + input image**)**

```
node=resize newSize=4000x2200
node=grayscale
node=add_border borderSize=100 borderColor=100,40,30
node=sepia
node=vignette
node=normalise
node=flip direction=[vertical]
```

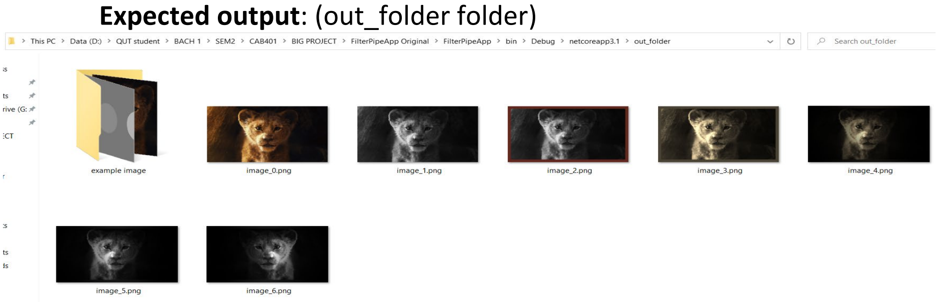
+



grow.png

Image	
Dimensions	3840 x 2160
Width	3840 pixels
Height	2160 pixels
Bit depth	24

Expected output: (out_folder folder)



3. Overcoming Barriers

1ST Barrier: Safe Loops in each Node class's process() method.

By simply convert the sequential For Loop to Parallel for loop version using Parallel.For I have successfully created a scalable parallelism effect in each method. At first I have tried to parallel both inner and outer loop (this action didn't damage the result from original), it worked well until the 4th core increments and the speedup is constant until the end. Then I have tried to transform the outer loop only, this technique is safer than creating a nested Parallel.For loop because it creates more space for each thread to work individually and avoiding very small-rate of sharing data in inner-loop.

And the outcomes seemed like method of changing only outer loop is more effective (due to shorter processing time).

Code Snippets:

Flip Method:

Before

```
    // Go through each image's pixels
    for (int x = 0; x <= input.Width - 1; x++)
    {
        for (int y = 0; y < input.Height; y++)
        {
            // Exchange each pixels at the closer x-position with
            Rgba32 oldPixel = input[input.Width - 1 - x, y];
            Rgba32 newPixel = new Rgba32(
                r: oldPixel.R,
                g: oldPixel.G,
                b: oldPixel.B,
                a: oldPixel.A
            );
            output[x, y] = newPixel;
        }
    }
}
// Horizontal flip
if (_Horizontal == true)
{
    // Go through each image's pixels
    for (int x = 0; x < input.Width; x++)
    {
        for (int y = 0; y <= input.Height - 1; y++)
        {
            // Exchange each pixels at the closer y-position with
            Rgba32 oldPixel = input[x, input.Height - 1 - y];
            Rgba32 newPixel = new Rgba32(
                r: oldPixel.R,
                g: oldPixel.G,
                b: oldPixel.B,
                a: oldPixel.A
            );
            output[x, y] = newPixel;
        }
    }
```

After

```
    // Go through each image's pixels
    Parallel.For(0, input.Width - 1, new ParallelOptions { MaxDegreeOfParallelism = Node.processor_Num },
        x => {
            for (int y = 0; y < input.Height; y++)
            {
                // Exchange each pixels at the closer x-position with the respectively pixels at further x
                Rgba32 oldPixel = input[input.Width - 1 - x, y];
                Rgba32 newPixel = new Rgba32(
                    r: oldPixel.R,
                    g: oldPixel.G,
                    b: oldPixel.B,
                    a: oldPixel.A
                );
                output[x, y] = newPixel;
            }
        });
}
// Horizontal flip
if (_Horizontal == true)
{
    // Go through each image's pixels
    Parallel.For(0, input.Width - 1, new ParallelOptions { MaxDegreeOfParallelism = Node.processor_Num },
        x => {
            for (int y = 0; y <= input.Height - 1; y++)
            {
                // Exchange each pixels at the closer y-position with the respectively pixels at further y
                Rgba32 oldPixel = input[x, input.Height - 1 - y];
                Rgba32 newPixel = new Rgba32(
                    r: oldPixel.R,
                    g: oldPixel.G,
                    b: oldPixel.B,
                    a: oldPixel.A
                );
                output[x, y] = newPixel;
            }
        });
```

Grayscale method:

Before

```
public static Image ToGrayscale(Image image)
{
    Image result = new Image(image.Width, image.Height);
    for (int x = 0; x < result.Width; x++)
    {
        for (int y = 0; y < result.Height; y++)
        {
            byte intensity = (byte)(image.GetIntensity(x, y));
            result[x,y] = new Rgba32(intensity, intensity, intensity, image[x,y].A);
        }
    }
    return result;
}
```

After

```
2 references
public static Image ToGrayscale(Image image)
{
    var parallelOption = new ParallelOptions() {
        MaxDegreeOfParallelism = Node.processor_Num
    };
    Image result = new Image(image.Width, image.Height);
    Parallel.For(0, result.Width, parallelOption, x => {
        for (int y = 0; y < result.Height; y++)
        {
            byte intensity = (byte)(image.GetIntensity(x, y));
            result[x, y] = new Rgba32(intensity, intensity, intensity, image[x, y].A);
        }
    });
```

Sepia method:

Before

```
// Go through each image's pixels
for (int x = 0; x < input.Width; x++)
{
    for (int y = 0; y < input.Height; y++)
    {
        // Apply new r,g,b values to each pixels to make sepia effect
        Rgba32 oldPixel = input[x, y];
        Rgba32 newPixel = new Rgba32(
            r: (byte)Math.Min(255, Math.Max(0, 0.393 * oldPixel.R + 0.769 * oldPixel.G + 0.189 * oldPixel.B)),
            g: (byte)Math.Min(255, Math.Max(0, 0.349 * oldPixel.R + 0.686 * oldPixel.G + 0.168 * oldPixel.B)),
            b: (byte)Math.Min(255, Math.Max(0, 0.272 * oldPixel.R + 0.534 * oldPixel.G + 0.131 * oldPixel.B)),
            a: oldPixel.A
        );
        output[x, y] = newPixel;
    }
}
```

After

```
Parallel.For(0, input.Width - 1, new ParallelOptions { MaxDegreeOfParallelism = Node.processor_Num },
        x =>
        {
            for (int y = 0; y < input.Height; y++)
            {
                // Apply new r,g,b values to each pixels to make sepia effect
                Rgba32 oldPixel = input[x, y];
                Rgba32 newPixel = new Rgba32(
                    r: (byte)Math.Min(255, Math.Max(0, 0.393 * oldPixel.R + 0.769 * oldPixel.G + 0.189 * oldPixel.B)),
                    g: (byte)Math.Min(255, Math.Max(0, 0.349 * oldPixel.R + 0.686 * oldPixel.G + 0.168 * oldPixel.B)),
                    b: (byte)Math.Min(255, Math.Max(0, 0.272 * oldPixel.R + 0.534 * oldPixel.G + 0.131 * oldPixel.B)),
                    a: oldPixel.A
                );
                output[x, y] = newPixel;
            }
        });
return output;
```

Vignette Method:

Before

```
for (int x = 0; x < input.Width; x++)
{
    for (int y = 0; y < input.Height; y++)
    {
        Rgba32 oldPixel = input[x, y];
        double xDistance = Math.Abs(_xCorofcenter - x);
        double yDistance = Math.Abs(_yCorofcenter - y);
        double distance = Math.Round(Math.Sqrt(Math.Pow(xDistance, 2) + Math.Pow(yDistance, 2)));
        // Follow the formula to calculate the brightness of each pixel
        double brightness = Math.Pow((maxDist - distance) / maxDist, 2);
        // Apply brightness intensity to each pixel r,g,b colors
        Rgba32 newPixel = new Rgba32(
            r: (byte)Math.Min(255, Math.Max(0, oldPixel.R * brightness)),
            g: (byte)Math.Min(255, Math.Max(0, oldPixel.G * brightness)),
            b: (byte)Math.Min(255, Math.Max(0, oldPixel.B * brightness)),
            a: oldPixel.A
        );
        output[x, y] = newPixel;
    }
```

After

```
Parallel.For(0, input.Width - 1, new ParallelOptions { MaxDegreeOfParallelism = Node.processor_Num },
        x =>
        {
            for (int y = 0; y < input.Height; y++)
            {
                Rgba32 oldPixel = input[x, y];
                double xDistance = Math.Abs(_xCorofcenter - x);
                double yDistance = Math.Abs(_yCorofcenter - y);
                double distance = Math.Round(Math.Sqrt(Math.Pow(xDistance, 2) + Math.Pow(yDistance, 2)));
                // Follow the formula to calculate the brightness of each pixel
                double brightness = Math.Pow((maxDist - distance) / maxDist, 2);
                // Apply brightness intensity to each pixel r,g,b colors
                Rgba32 newPixel = new Rgba32(
                    r: (byte)Math.Min(255, Math.Max(0, oldPixel.R * brightness)),
                    g: (byte)Math.Min(255, Math.Max(0, oldPixel.G * brightness)),
                    b: (byte)Math.Min(255, Math.Max(0, oldPixel.B * brightness)),
                    a: oldPixel.A
                );
                output[x, y] = newPixel;
            }
        });
```

Normalise Method:

Before

```
for (int x = 0; x < input.Width - 1; x++)
{
    for (int y = 0; y < input.Height - 1; y++)
    {
        // Normalise number of the old image based on the intensity of image
        double _numOld = output.GetIntensity(x, y);
        Rgba32 oldPixel = input[x, y];
        // Formula to find a new normalise number for the pixel image
        double relavtiveBrightness = Math.Abs((_numOld - _oldMin) / (_oldMax - _oldMin));
        double numNew = Math.Abs(_newMin + relavtiveBrightness * (_newMax - _newMin));
        // Apply new normalise number to each pixel red, green and blue colors
        Rgba32 newPixel = new Rgba32(
            r: (byte)Math.Min(255, Math.Max(0, numNew)),
            g: (byte)Math.Min(255, Math.Max(0, numNew)),
            b: (byte)Math.Min(255, Math.Max(0, numNew)),
            a: oldPixel.A
        );
        output[x, y] = newPixel;
    }
```

After

```
// Go through each pixel of image
Parallel.For(0, input.Width - 1, new ParallelOptions { MaxDegreeOfParallelism = Node.processor_Num },
        x =>
        {
            for (int y = 0; y < input.Height - 1; y++)
            {
                // Normalise number of the old image based on the intensity of image
                double _numOld = output.GetIntensity(x, y);
                Rgba32 oldPixel = input[x, y];
                // Formula to find a new normalise number for the pixel image
                double relavtiveBrightness = Math.Abs((_numOld - _oldMin) / (_oldMax - _oldMin));
                double numNew = Math.Abs(_newMin + relavtiveBrightness * (_newMax - _newMin));
                // Apply new normalise number to each pixel red, green and blue colors
                Rgba32 newPixel = new Rgba32(
                    r: (byte)Math.Min(255, Math.Max(0, numNew)),
                    g: (byte)Math.Min(255, Math.Max(0, numNew)),
                    b: (byte)Math.Min(255, Math.Max(0, numNew)),
                    a: oldPixel.A
                );
                output[x, y] = newPixel;
            }
        });
```

Addborder Method:

Before

```
for (int x = 0; x < _borderSize; x++)
{
    for (int y = 0; y < input.Height; y++)
    {
        // Apply color to the right height of border
        Rgba32 newPixel = _colourOfborder;
        output[x, y] = newPixel;
    }
}
// Check and draw the top width of border
for (int x = 0; x < resizedImage.Width; x++)
{
    for (int y = 0; y < _borderSize; y++)
    {
        // Apply color to the top width of border
        Rgba32 newPixel = _colourOfborder;
        output[x, y] = newPixel;
    }
}
// Check and draw the left height of border
for (int x = input.Width - _borderSize; x < input.Width; x++)
{
    for (int y = 0; y < input.Height; y++)
    {
        // Apply color to the left height of border
        Rgba32 newPixel = _colourOfborder;
        output[x, y] = newPixel;
    }
}
// Check and draw the bottom width of border
for (int x = 0; x < resizedImage.Width; x++)
{
    for (int y = input.Height - _borderSize; y < input.Height; y++)
    {
        // Apply color to the bottom width of border
        Rgba32 newPixel = _colourOfborder;
        output[x, y] = newPixel;
    }
}
// Check the size of border to insert the image
for (int x = _borderSize; x < input.Width - _borderSize; x++)
{
    for (int y = _borderSize; y < input.Height - _borderSize; y++)
    {
        // Insert the image based on the size of border
        output[x, y] = resizedImage[x, y];
    }
}
```

After

```
// Check and draw the right height of border
Parallel.For(0, _borderSize, x => {
    for (int y = 0; y < input.Height; y++)
    {
        // Apply color to the right height of border
        Rgba32 newPixel = _colourOfborder;
        output[x, y] = newPixel;
    }
});

// Check and draw the top width of border
Parallel.For(0, resizedImage.Width, x => {
    for (int y = 0; y < _borderSize; y++)
    {
        // Apply color to the top width of border
        Rgba32 newPixel = _colourOfborder;
        output[x, y] = newPixel;
    }
});

// Check and draw the left height of border
Parallel.For(input.Width - _borderSize, input.Width, x => {
    for (int y = 0; y < input.Height; y++)
    {
        // Apply color to the left height of border
        Rgba32 newPixel = _colourOfborder;
        output[x, y] = newPixel;
    }
});

// Check and draw the bottom width of border
Parallel.For(0, resizedImage.Width, x =>
{
    for (int y = input.Height - _borderSize; y < input.Height; y++)
    {
        // Apply color to the bottom width of border
        Rgba32 newPixel = _colourOfborder;
        output[x, y] = newPixel;
    }
});
// Check the size of border to insert the image
Parallel.For(_borderSize, input.Width - _borderSize, x => {
    for (int y = _borderSize; y < input.Height - _borderSize; y++)
    {
        // Insert the image based on the size of border
        output[x, y] = resizedImage[x, y];
    }
});
```

2ND Barrier: the Write() – Save Image function inside Pipeline Class- Run() function.

As said before on section 2, this function create a bottle neck for the whole application, for each increment of instruction lines the system need to stop and save the changes from previous initial instruction which creates a long wait-time between instructions. Therefore I have used a list and array (converting purposes) to temporarily store all Image output values

```
public static Image Run(Image input, List<Node> pipeline, bool logging, string saveDir)
{
    List<Image> stepList = new List<Image>();
    Image[] stepArray = new Image[pipeline.Count];
```

, then use a separately Parallel.For Loop to randomly map each thread into given items inside the Array. This action creates a parallel effect of processing each item in the array => solved the bottleneck of the whole software.

Code Snippet:

Before

```
for(int x = 0; x < pipeline.Count; x++)
{
 // The stopwatch should start recording once the program run
 stopwatch.Start();
   Image output = pipeline[x].Process(input);
   input = output;
 // The stopwatch should stop recording once the program successfully complete running and print out required stats

   if(logging == true)
   {
    Console.WriteLine($"Node:{pipeline[x].Name}");
    Console.WriteLine($"Input: Image ({input.Width.ToString()} X {input.Height.ToString()})");
    Console.WriteLine("     Processing...");
       // Create a folder if the folder doesn't exists
       if (saveDir != "")
       {
           // Create a folder to store processed nodes
           System.IO.Directory.CreateDirectory(saveDir);
           // Create sub-file
           output.Write($"{saveDir}/image_{x}");
       }
       stopwatch.Stop();
       Console.WriteLine("Took: {0}", stopwatch.Elapsed);
    total_processing_time += (double)stopwatch.Elapsed.TotalSeconds;
    Console.WriteLine($"Output: Image ({output.Width.ToString()} X {output.Height.ToString()})\n");
       stopwatch.Reset();
   }
}
//print out total processing time
    Console.WriteLine("Total processing time: " + total_processing_time + " Seconds");
// Print out the current pipeline nodes that has been stored at a specific folder

    Console.WriteLine($"Saved at: {Path.GetFullPath(saveDir)}");
Image output2 = input;
```

After

```
for(int x = 0; x < pipeline.Count; x++)
{
 // The stopwatch should start recording once the program run
 stopwatch.Start();
   Image output = pipeline[x].Process(input);
   input = output;

   if(logging == true)
   {
    Console.WriteLine($"Node:{pipeline[x].Name}");
    Console.WriteLine($"Input: Image ({input.Width.ToString()} X {input.Height.ToString()})");
    Console.WriteLine("     Processing...");
       //save the processed step
       stepList.Add(output);
       // The stopwatch should stop recording once the program successfully complete running and print out
       stopwatch.Stop();
       Console.WriteLine("Took: {0}", stopwatch.Elapsed);
       total_nodes_time += (double)stopwatch.Elapsed.TotalSeconds;
       Console.WriteLine($"Output: Image ({output.Width.ToString()} X {output.Height.ToString()})\n");
       stopwatch.Reset();
   }
}


 Stopwatch stopwatch1 = new Stopwatch();
 // Create a folder if the folder doesn't exists
 if (saveDir != "")
 {
     stepArray = stepList.ToArray();
     stopwatch1.Start();
     Parallel.For(0, stepArray.Length, new ParallelOptions {MaxDegreeOfParallelism = Node.processor_Num },
         i => {
         // Create a folder to store processed nodes
         System.IO.Directory.CreateDirectory(saveDir);
         // Create sub-file
         stepArray[i].Write($"{saveDir}/image_{i}");
     });
     stopwatch1.Stop();
 }
```

4. Optimal Speedup

From above analysis we can see **"Saving image into folder"** and **"Applying filter to image"** are two main parts of application's processing job. Therefore, I will focus on investigating/timing the speedup of these two areas.

Timing Method:

To test the processing time of the overall application I have used **StopWatch**() function (from `using System.Diagnostics;` library**)**. By placing the watch in the core processing/highest CPU usage areas of the application, which is the for loop inside **PipeLine class's Run() Function**.

Sequential Version:

```
double total_processing_time = 0;
double total_nodes_time = 0;
double total_saving_time = 0;
// Print out running pipeline on image with width and height of the image
Console.WriteLine($"Running pipeline on Image ( {input.Width} X {input.Height} ) \n" );
// A build in function to record the times taken for processing each pipeline nodes
Stopwatch stopwatch = new Stopwatch();
Stopwatch stopwatch1 = new Stopwatch();
// Read through all of the pipeline nodes
for (int x = 0; x < pipeline.Count; x++)
{
// The stopwatch should start recording once the program run
stopwatch.Start();
    Image output = pipeline[x].Process(input);
    input = output;
// The stopwatch should stop recording once the program successfully complete running and print out required stats

    if(logging == true)
    {
     Console.WriteLine($"Node:{pipeline[x].Name}");
     Console.WriteLine($"Input: Image ({input.Width.ToString()} X {input.Height.ToString()})");
     Console.WriteLine("     Processing...");
        // Create a folder if the folder doesn't exists
        if (saveDir != "")
        {
            // Create a folder to store processed nodes
            System.IO.Directory.CreateDirectory(saveDir);
            // Create sub-file
            stopwatch1.Start();
            output.Write($"{saveDir}/image_{x}");
            stopwatch1.Stop();
            total_saving_time += (double)stopwatch1.Elapsed.TotalSeconds;

        }
        stopwatch.Stop();
        //calculate the applying filter time only (doesnt count saving time).
        Console.WriteLine("Took: {0}", (stopwatch.Elapsed - stopwatch1.Elapsed));

     total_processing_time += (double)stopwatch.Elapsed.TotalSeconds;
     Console.WriteLine($"Output: Image ({output.Width.ToString()} X {output.Height.ToString()})\n");

        stopwatch.Reset();
        stopwatch1.Reset();
     }
}
```

Then print the result on Console

```
//print out processing time
total_nodes_time = total_processing_time - total_saving_time;
Console.WriteLine("All Node processing time: " + total_nodes_time + " Seconds");
Console.WriteLine("Save File processing time: " + total_saving_time + " Seconds");
Console.WriteLine("Total processing time: " + total_processing_time + " Seconds");
```

The output would be like this:

```
Node:Normalise
Input: Image (4000 X 2200)
     Processing...
Took: 00:00:02.6316320
Output: Image (4000 X 2200)

Node:Flip (Horizontal:False, Vertical:True)
Input: Image (4000 X 2200)
     Processing...
Took: 00:00:00.5780300
Output: Image (4000 X 2200)

All Node processing time: 7.682383599999998 Seconds
Save File processing time: 18.0382389 Seconds
Total processing time: 25.720622499999998 Seconds
```

Paralleled Version:

```
double total_processing_time = 0;
double total_nodes_time = 0;
// Print out running pipeline on image with width and height of the image
Console.WriteLine($"Running pipeline on Image ( {input.Width} X {input.Height} ) \n" );
// A build in function to record the times taken for processing each pipeline nodes
Stopwatch stopwatch = new Stopwatch();
// Read through all of the pipeline nodes
for(int x = 0; x < pipeline.Count; x++)
{
// The stopwatch should start recording once the program run
stopwatch.Start();
    Image output = pipeline[x].Process(input);
    input = output;

    if(logging == true)
    {
     Console.WriteLine($"Node:{pipeline[x].Name}");
     Console.WriteLine($"Input: Image ({input.Width.ToString()} X {input.Height.ToString()})");
     Console.WriteLine("     Processing...");
        //save the processed step
        stepList.Add(output);
        // The stopwatch should stop recording once the program successfully complete running and print out required stats
        stopwatch.Stop();
        Console.WriteLine("Took: {0}", stopwatch.Elapsed);
        total_nodes_time += (double)stopwatch.Elapsed.TotalSeconds;
        Console.WriteLine($"Output: Image ({output.Width.ToString()} X {output.Height.ToString()})\n");
        stopwatch.Reset();
    }
}


Stopwatch stopwatch1 = new Stopwatch();
// Create a folder if the folder doesn't exists
if (saveDir != "")
{
    stepArray = stepList.ToArray();
    stopwatch1.Start();
    Parallel.For(0, stepArray.Length, new ParallelOptions {MaxDegreeOfParallelism = Node.processor_Num },
        i => {
        // Create a folder to store processed nodes
        System.IO.Directory.CreateDirectory(saveDir);
        // Create sub-file
        stepArray[i].Write($"{saveDir}/image_{i}");
    });
    stopwatch1.Stop();
}
total_processing_time = (double) stopwatch1.Elapsed.TotalSeconds + total_nodes_time;
```

Print out result

```
//print out total processing time
Console.WriteLine("All Node processing time: " + total_nodes_time + " Seconds" + " On " + Node.processor_Num + " Core(s)");
Console.WriteLine("Save File processing time: " + (double)stopwatch1.Elapsed.TotalSeconds + " Seconds" + " On " + Node.processor_Num + " Core(s)")
Console.WriteLine("Total processing time: " + total_processing_time + " Seconds" + " On " + Node.processor_Num+ " Core(s)");
```

Console output

```
Node:Normalise
Input: Image (4000 X 2200)
     Processing...
Took: 00:00:01.6833677
Output: Image (4000 X 2200)

Node:Flip (Horizontal:False, Vertical:True)
Input: Image (4000 X 2200)
     Processing...
Took: 00:00:00.3182006
Output: Image (4000 X 2200)

All Node processing time: 4.9433150999999995 Seconds On 2 Core(s)
Save File processing time: 9.9146242 Seconds On 2 Core(s)
Total processing time: 14.8579393 Seconds On 2 Core(s)
```

And Finally for flexible changing between number of cores when timing in paralleled version I have create a global variable in Node class.

```
abstract class Node
{
    public const int processor_Num = 2;
```

to change parameter of core nums in each Parallel.ForLoop()

```
// Go through each image's pixels
Parallel.For(0, input.Width - 1, new ParallelOptions { MaxDegreeOfParallelism = Node.processor_Num },
```

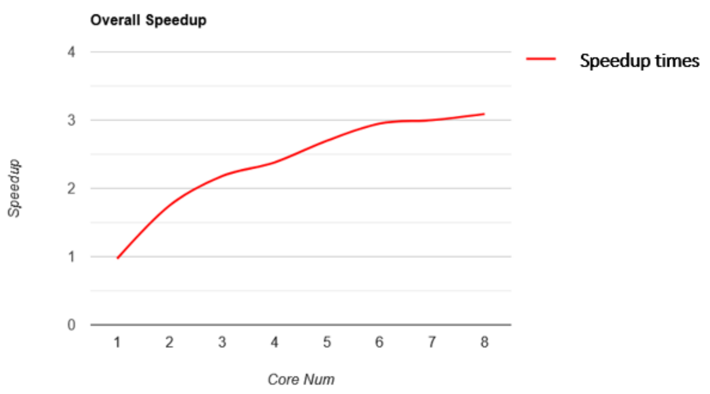
Tasks	**Sequential Version**(seconds)	**Paralleled Version**	
		Number of Core	**Time**(seconds)
Overall Image Processing Time (All Nodes)	7.9	1	8.0
		2	5.0
		3	4.1
		4	3.99
		5	3.5
		6	3.6
		7	3.2
		8	3.2

Tasks	**Sequential Version**(seconds)	**Paralleled Version**	
		Number of Core	**Time**(seconds)
Overall Image Saving Time (All steps)	18.04	1	18.7
		2	9.8
		3	7.75
		4	6.9
		5	6.1
		6	5.2
		7	5.4
		8	5.2

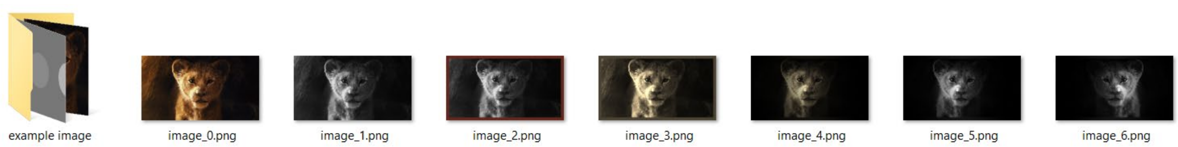
3

Tasks	**Sequential Version**(seconds)	**Paralleled Version**	
		Number of Core	**Time**(seconds)
Overall App Processing Time (Total of 2 times above)	25.99	1	26.7
		2	14.8
		3	11.89
		4	10.9
		5	9.6
		6	8.8
		7	8.69
		8	8.4

We can see that the overall app processing time is nearly **N** time faster when increase **N** number of Core until the 3rd core increment the speedup still gaining but with a really small speedup (~0.1 speedup) => the speedup is nearly "typical" sub-linear speedup.



Since the speedup has been achieved the output data is the same.



5. Conclusion

My attempt to parallel this application is fine but not really success as I expected ("Perfect" linear speedup). The project might exploit more speedup performance if I can split the input image into multiple smaller images according to number of cores, then assign each core to process appropriate down-scale part. For illustration:

https://www.youtube.com/watch?v=LIAMr-dfg1Y&t=975s

But IMO, the Parallel.For already implemented a similar idea by randomly choosing each thread to process coming pixels in the whole image. Therefore, to obtain more potential speedup I need to investigate/improve the source codes from SixLabors library (especially the "**SixLabors.ImageSharp.ImageExtensions.Save() –** code reference") and other image's processing method.