Project 2

MPCS 52060 – Parallel Programming

Due: June 1st 2018, by 11:59pm

Preliminaries

As I talked about in class, many algorithms in image processing benefit from parallelization. In this assignment, you will create an image processing system that reads in a series of images and applies certain effects to them using image convolution. If you are unfamiliar with image convolution then you should read over the following sources before beginning the assignment:

- `http://www.songho.ca/dsp/convolution/convolution2d_example.html`
- `https://en.wikipedia.org/wiki/Kernel_(image_processing)`

Initial Setup

1. Create a directory called **proj2** inside your repository. It must be called **proj2** with all lowercase letters.
2. The **proj2** directory should have at least the following structure (bold items represent a directory):

    - **proj2**
        - report.pdf (or some other readable file)
        - **src**
            - editor
                - editor.go

Again, the name of the file must match exactly as specified. You may add any additional supporting libraries and code that you implement to the src directory.

Assignment: Image Processing System

For this assignment, you will create image editor that will apply image effects on series of images using 2D image convolution. Please make sure to read over the articles presented in the Preliminaries section before beginning the assignment. The program will take in a comma-separated file (not technically the standard CSV format), where each line represents an image along with the effects that should be applied to that image. Each line will have the following format:

```
IMAGE_IN_NAME, IMAGE_OUT_NAME, EFFECT_1, EFFECT_2, ..., EFFECT_N
```

where each field is described in the table below,

Data Field	Description
IMAGE_IN_PATH	The IMAGE_IN_PATH field represents the file path of the image to read in. Images in this assignment will always be PNG files. Example: "/Users/lamont/images/sky.png" or specifying a relative location "sky.png".
IMAGE_OUT_PATH	The IMAGE_OUT_PATH field represents the file path to save the image after applying the effects. Example: "/Users/lamont/images/sky_out.png" or specifying a relative location "sky_out.png".
EFFECT_1, EFFECT_2, ..., EFFECT_N	The effect fields (i.e., EFFECT_1, EFFECT_2, ..., EFFECT_N) represents the image effects to apply to the IMAGE_IN_PATH image. You must apply these in the order they are listed. Each effect is separated by a comma. Example: "S,E,G" or "G,B,S". What image effect each letter stands for is described in the *Image Effects* section.

The program will read in the images, apply the effects associated with an image, and save the images to their specified output file paths. How the program processes this file is described in the *Program Specifications* section.

Note: Again, the file is inot the standard CSV formatted file where each line has the same number of columns; therefore, using a CSV library may not be appropriate for this assignment.

Image Effects

The sharpen, edge-detection, and blur image effects are required to use image convolution to apply their effects to the input image. Again, you can read about how to perform image convolution here:

`http://www.songho.ca/dsp/convolution/convolution2d_example.html`

As stated in the above article, the size of the input and output image are fixed (i.e., they are the same). Thus, results around the border pixels will not be fully accurate because you will need to pad zeros where inputs are not defined. You are required to use the a zero-padding when working with pixels that are not defined. **You may not use a library or external source to perform the convolution for you. You must implement the convolution code yourself**. The grayscale effect uses a simple algorithm defined below that does not require convolution.

Image Effect	Effect Description
S	Performs a sharpen effect with the following kernel: $\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$
E	Performs a edge-detection effect with the following kernel: $\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$
B	Performs a blur effect with the following kernel: $\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$
G	Performs a grayscale effect on the image. This is done by averaging the values of all three color numbers for a pixel, the red, green and blue, and then replacing them all by that average. So if the three colors were 25, 75 and 250, the average would be 116, and all three numbers would become 116.

Program Specifications

You will implement two versions of the `editor` program: a sequential version and a parallel version. The program has the following usage statement:

```
Usage: editor [-p=[num of threads]]  <csv file>
    <csv file>  = The file that contains the batch of images that need to be processed.
    -p=[num of threads] = An optional flag to run the editor in its parallel version.
                          You also have the option of specifying the number of threads
    [num of threads] = the number of workers in the program (including the main thread)
```

Assumptions: No error checking is needed. The `<csv file>` file provided will always be valid and provided in the format described above.

Sequential Version

The sequential version is **ran by default** when executing the `editor` program. The user must provide the `-p` flag to specify that they want to run the program's parallel version. The sequential program is relatively straightforward. As described above, this version should run through the images inside the `<csv file>`, apply the specified effects for an image and save the modified images to their specified output files.

Note: You should implement the sequential version first. Make sure your code is **modular** enough such that you can potentially reuse functions/data structures later in your parallel version. Think about what libraries should be created (e.g., feed and lock libraries you created for project 1). **We will be looking at code-style more closely when grading this assignment**.

Parallel Version

The parallel version is ran when the user specifies the `-p` flag. By default, the number of worker threads running in the program is equal to value returned by calling `runtime.NumCPU()`; otherwise, it is equal to

the user-specified amount (`[num of threads]`). This amount does not include the main goroutine as one of the workers. There is no specific way I'm requiring you to implement the parallel version. However, you must **at least** adhere to/include the following:

1. All worker threads other than the main thread must be spawned before any real work begins. Thus, `[num of threads]` must be spawned before beginning to process the `<csv file>`. Additionally, those threads are the only threads to be spawned in the program. You cannot spawn additional threads at later points in time. For this assignment, we are working under the assumption that thread resources are unlimited.

2. A *functional decomposition* partitioning must be implemented somewhere in your code. I recommend using the Bulk synchronous parallel (BSP) model I described in class:

   `https://en.wikipedia.org/wiki/Bulk_synchronous_parallel`

   Along with two queues, one queue to handle pushing out work to the group of threads, and the other queue for storing images that are fully completed (i.e., all effects have be performed on the image) and need to be saved out to a file. How the BSP model and queues interact is up to you to figure out. For this assignment, I'm forcing you to think heavily about the design and how modules interact with each other instead of directly telling you.

   Again, you do no need to use this model. You have the full freedom to functionally decompose the work as you wish.

3. A *data decomposition* partitioning must be implemented somewhere in your code. I recommend that if you are using the BSP model that during a super-step that each thread in the group is assigned a section of the data to work on. You can also think of each super-step as applying one effect to the entire image.

   As I stated for functional decomposition, you have the freedom to data decompose the work as you wish. You are not required to use this technique.

4. You cannot use Go channels in this assignment.

Working with Images and Startup Code

As part of the Go standard library, an `image`(`https://golang.org/pkg/image/`) package is provided that makes it easy to load,read,and save PNG images. I recommend looking at the examples from these links:

- Go PNG docs: `https://golang.org/pkg/image/png/`
- Helpful tutorial: `https://www.devdungeon.com/content/working-images-go` (Make sure to cite this website, if you are going to use a similar structure to the code provided.)
  - The developer directly accesses the `Pix` buffer. I would recommend you use the `At()` and `Set()` methods as specified by the Go PNG documentation.

Note: The image package only allows you to read an image data and not modify it in-place. You will need to create a separate out buffer to represent the modified pixels.

To help you get started, I provide code for loading, saving, performing the grayscale effect on a png image. You are not required to use this code and you can modify it as you wish. The zip file with this code is located on the "Resources" column on the homework page.

Test Images & Grading Output

You can find test images and csv files here:

`https://www.dropbox.com/s/cxyvd9f6qr5nep4/csvs.zip?dl=0`

I couldn't load them on to the course website because the images are too big. Please let me know if you have problems downloading the images or the csv files.

DO NOT SUBMIT THESE FILES TO YOUR REPOSITORY!

The grading and testing of the assignment will only be done using these files so as long as your program works with these then the I/O portion will be counted as correct.

Assignment Writeup

You will run timing measurements on both the sequential and parallel versions of `editor.go`. Use the test-data I provided as problem sizes for your measurements. The problem sizes (i.e., the total number of operations being processed for one program execution), P, will remain the same for both versions. For the parallel version, you will spawn a certain number of worker threads (N).

Parameter Amounts:

- $P = \{cs\_file\_1, csv\_file\_2, csv\_file\_3\}$
- $N = \{1, 2, 4, 6, 8\}$

Timings

For the sequential version, you will produce 3 timing results based on P:

- Timing 1: $P = csv\_file\_1$
- Timing 2: $P = csv\_file\_2$
- Timing 3: $P = csv\_file\_3$

For the parallel version, you will produce a timing result per number of worker threads with a fixed problem size and block size. Thus, you will have 15 timing results:

- Timing 5: $P = csv\_file\_1, N = 1$
- Timing 6: $P = csv\_file\_1, N = 2$
- Timing 7: $P = csv\_file\_1, N = 4$
- Timing 8: $P = csv\_file\_1, N = 6$
- Timing 9: $P = csv\_file\_1, N = 8$
- Timing 10: $P = csv\_file\_2, N = 1$
- Timing 11: $P = csv\_file\_2, N = 2$
- Timing 12: $P = csv\_file\_2, N = 4$
- Timing 13: $P = csv\_file\_2, N = 6$
- Timing 14: $P = csv\_file\_2, N = 8$
- ...

Using those timings, produce speedup graphs per problem size (i.e., you'll have 3 speedup graphs) or I will allow them to be on the same graph. The Y-Axis will list the speedup measurement and the X-Axis will list the number of worker threads. Similar to the graph shown below. For each graph, make sure to title it, and label each axis. Make sure to adjust your Y-axis range so that we can accurately see the values. That is, if most of your values fall between a range of [0,1] then don't make your speedup range [0,14] as shown in the diagram below.

Calculating Timings

As with the prior homework assignment, running and timing a program once can give misleading timings because:

- A process may create a cache on its first execution; therefore, running faster subsequently on additional executions
- Other processes may cause the command to be starved of CPU or I/O time
- There might be random interrupt that causes the timing to be an outlier

Thus, for each timing indicated above you will actually run that timings **5 times** and take the average of those timings. Also, **make sure you close down all applications when performing your timing tests!** You want to make sure you are getting accurate results.

Performance Analysis

Please submit a report (pdf document, text file, etc.) summarizing your results from the experiments and the conclusions you draw from them. Your report should include your plot(s) as specified above and a self-contained report. That is, somebody should be able to read the report alone and understand what code you developed, what experiments you ran and how the data supports the conclusions you draw. **You must analyze your graphs and explain the outcome of your experiments!**. Some of you did not do this for homework 4. The report **must** also include the following (along with your analysis of your experiments):

- A description of how you implemented your solution. This should include a description on any additional libraries you used/implemented.
- Make sure to describe in detail, your functional and data decomposition approaches and why you choose them.
- Specifications of the testing machine you ran your experiments on (i.e. Core Architecture (Intel/AMD), Number of cores, operating system, memory amount, etc.)
- Answers to the following questions:
  - What are the **hotspots** and **bottlenecks** in your sequential program? Were you able to parallelize the hotspots and/or remove the bottlenecks in the parallel version?
  - Describe the **granularity** in your implementation. Are you using a coarse-grain or fine-grain granularity? Explain.
  - Does the image size being processed have any effect on the performance? For example, processing csv files with the same number of images but one file has very large image sizes, whereas the other has smaller image sizes.