Project 2

MPCS 52060 – Parallel Programming

Due: August 7th 2020, 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. Make sure to perform a pull upstream inside your repository. This will grab the distribution code for proj2. The command is the following:


```
$ git pull upstream master
```

2. The **proj2** directory should have the following structure (bold items represent a directory):

- **proj2**
  - go.mod
  - **editor**
    - editor.go
  - **png**
    - png.go
    - effects.go
  - report.pdf (or some other readable file)

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. Similar to project 1, The program will read in from `os.Stdin` JSON strings, where each string represents an image along with the effects that should be applied to that image. Each string will have the following format:

```
{
  "inPath": string,
  "outPath": string,
  "effects": [string]
}
```

where each key-value is described in the table below,

Example	Description

Absolute Path:
`"inPath":"/Users/lamont/images/sky.png"`
Relative Path:
`"inPath":"sky.png"`
	The `"inPath"` pairing represents the file path of the image to read in. Images in this assignment will always be PNG files.

Absolute Path:
`"inPath":"/Users/lamont/images/sky\_out.png"`
Relative Path:
`"inPath":"sky\_out.png"`
	The `"outPath"` pairing represents the file path to save the image after applying the effects.

`"effects":["S","B","E"]` `"effects":["G"]`	The `"effects"` pairing represents the image effects to apply to the IMAGE_IN_PATH image. You must apply these in the order they are listed. If no effects are specified (e.g., `[]`) then the out image is the same as the input image. 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.

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=[number of threads]]
    -p=[number of threads] = An optional flag to run the editor in its parallel version.
                             Call and pass the runtime.GOMAXPROCS(...) function the integer
                             specified by [number of threads].
```

Assumptions: No error checking is needed to be done to the strings coming in from `os.Stdin`. You can assume the JSON strings will contain valid values 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 specified by the strings coming in from `os.Stdin`, apply their effects and save the modified images to their 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 and design style more closely when grading this assignment**.

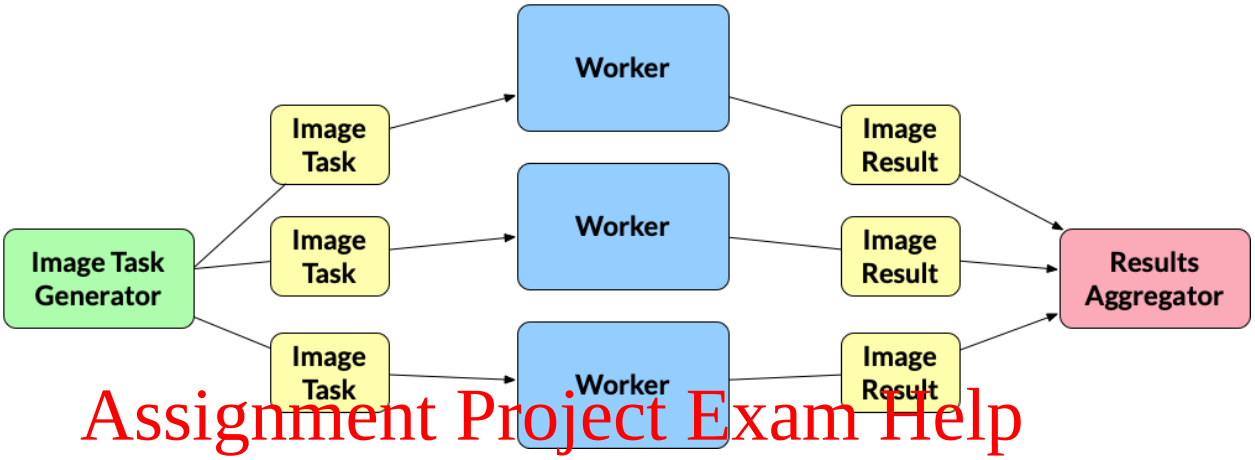
Parallel Version

The parallel version is ran with the `-p` flag. The parallel implementation will be a mixture of using *functional decomposition* and *data decomposition*. The implementation **must** be implemented as follows:

1. The integer given to the `-p` flag must be passed to the `runtime.GOMAXPROCS(numOfThreads)` function. Remember, this function will limit how many physical cores the runtime can use for a given progam. This must be done at the beginning of the program.

2. For this assignment, all synchronization between the goroutines must be done using **channels**. **You cannot use anything from the `sync` package**.

3. You must implement the following *fanin/fanout* scheme:



- **Image Task Generator**: As stated earlier, the program will read in the images to process via `os.Stdin`. Reading from the `os.Stdin` is done by a single generator goroutine. The image task generator will read in the JSON strings and do any preparation needed before applying their effects. What "preparation" means is dependent on your implementation and how you plan to pass the images to the worker goroutines (described below), which handle the filtering. The output from this goroutine will be an `ImageTask` value. How you define the `ImageTask` struct is up to you. The image task generator will write each `ImageTask` to a channel and multiple workers will read from it.

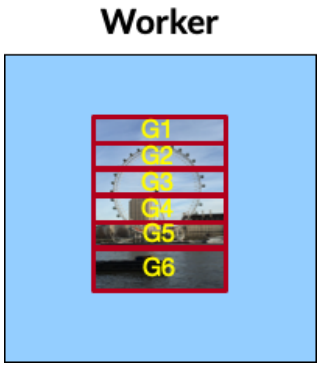
- **ImageTask**: A value that holds everything needed to do filtering for a specific JSON string. Again, its up to you how you define the `ImageTask struct`.

- **Workers**: The workers are the goroutines that are performing the filtering effects on the images. The number of workers is static and is equal to the `numOfThreads` command line argument. A worker must use a pipeline pattern. The "pipeline pattern" can simply be iterating (i.e., using a for-loop) through the image effects. However, you have the freedom to implement this pipeline as you wish but I recommend that each stage be dedicated to a specific filtering effect.

  There is only one requirement for the pipeline component. Each stage of the pipeline, must have a *data decomposition* component, which must do the following:

  - Spawn N number of goroutines, where $N = numOfThreads$.
  - Each spawned goroutine is given a section of the image to work on.
  - Each spawned goroutine will apply the effect for that stage to its assigned section.
  - You should give approximately equal portions to all goroutines.

  Visually the splitting could look something like this if $numOfThreads = 6$:



The output of a worker is an `ImageResult`. This can be same as an `ImageTask`. There is no need to create a a new struct that represents the final image after applying its effects. However, you can define a new `ImageResult` struct that holds that final image. Each worker writes its `ImageResult` to a channel that is read by a Results Aggregator gorountine (described below).

- **ImageResult**: the final image after applying its effects. Again you do not need to define a new `ImageResult` struct but can reuse the same `ImageTask`.
- **Results Aggregator**: The results aggregator gorountine reads from the channel that holds the `ImageResults` and saves the filtered image to its "outpath" file.

4. If all the images have been processed then the main goroutine can exit the program.

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. This code is already inside the **proj2** stater code that you grabbed when performing the `git pull upstream master`.

Test Images & Grading the Output

You can find test images and csv files here:

`https://www.dropbox.com/s/s6sws5w4xcnx94e/proj2_files.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! You will be penalized for any images submitted to your repo!

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. Also, these produced images do not need to look 100% like the provided output. If there are some slight differences based on rounding-error then that's fine for full credit.

Final Note: The provide files (e.g., `file_1, file_2, file_3`) are all the same. The differences between the directories are the image sizes. This is an important note when thinking about the analysis of your program.

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 assign the `-p` flag to (N) threads.

Parameter Amounts:

- $P = \{file_1, file\_2, file\_3\}$
- $N = \{1, 2, 4, 6, 8\}$

Timings

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

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

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

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

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.

Testing script

As with homework 5, you will need to provide a testing script. Please refer back to the homework 5 document for clarification.

Note: As with homework 5, your testing script must be ran on the CS clusters.

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 **20 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.
- Instructions on how to run your testing script.
- 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.