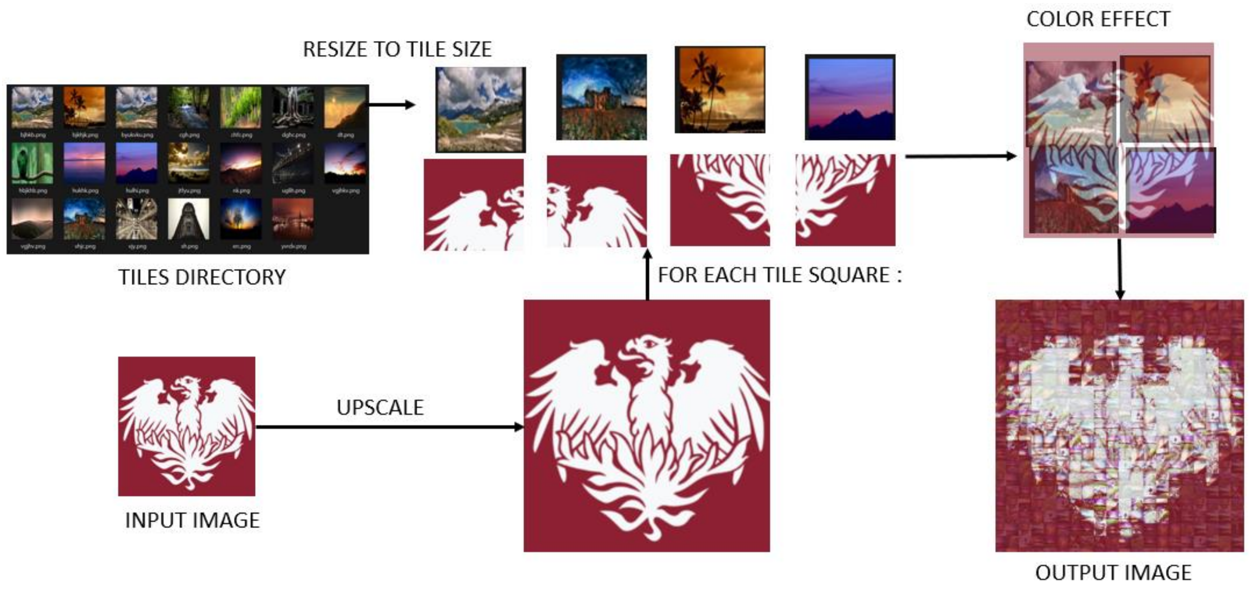
Project 3 Report- Mosaic Collage Generator

Muhammad Aulia Firmansyah
mauliafirmansyah@uchicago.edu

System Description



General Program Implementation

This program is an implementation of mosaic collage image generator using Go language. There are two main parts of the process of this program.

In first part, this program accepts an input image (-i flag) and a directory of images (-d flag) which will be used as tile images covering the input image. The input image is upscaled based on value of -U flag. The images from the directory will be loaded and resized to a square of tile with configured size (-s flag). This part is the first part where the parallelization is implemented in the parallel versions. In the “parallel” version (-M p flag), the directory entries are loaded into a channel, which is then consumed by goroutines, which then sends the resized tile images to another channel, to be added into an array. In the “work stealing” version (-M w flag), the directory entries are uniformly assigned to a deque for each goroutine. Then, the goroutines consume their deques, run, and send the result into a channel to be added into an array. Therefore, this program has created an upscaled input image and array of square shaped tile images.

In second part, for each tile sized squares in the upscaled input image:

- A tile image is selected randomly from the tile images array.
- The reference image is extracted from input image cropped at tile position.
- A color transfer using histogram matching is applied to the tile image based on the reference image color histogram.
- The colored tile image is blended in with the original tile image with adjusted weight value from the -B flag.

- The result of the blend-in is then combined with the reference image with adjusted intensity weight value from the -I flag.

This loop is the second part of the program where the parallelization is implemented in the parallel versions. Similar to the first part, in the “parallel” version (-M p flag), the tile positions are loaded into a channel, which is then consumed by goroutines, which apply the color effect to the upscaled input image, then send a Boolean value to another channel, to mark the progress of task completion. In the “work stealing” version (-M w flag), the tile positions are uniformly assigned to a deque for each goroutine. Then, the goroutines consume their deques, apply the color effect to the upscaled input image, then send a Boolean value to another channel, to mark the progress of task completion. Finally, the combined image with the tile images is then saved to the output file. (-o flag)

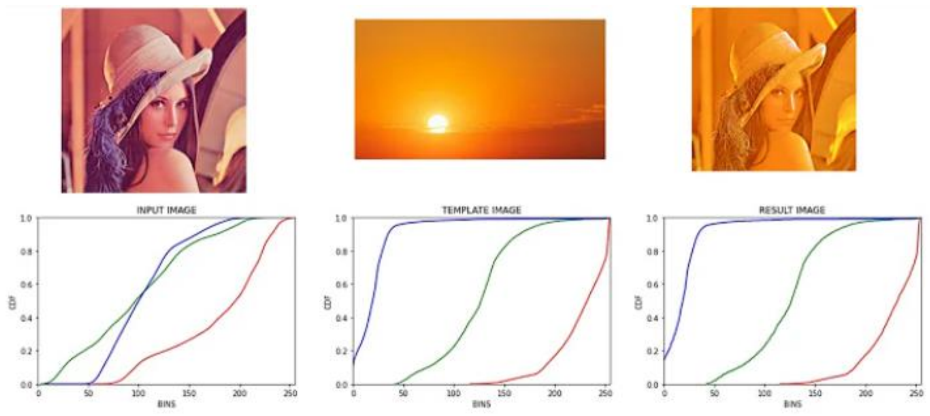
Here is the printed usage information of this program:

```
FLAGS:
  -B float
        intensity of tile images color blend-in in float (0.0 - 1.0). 1.0 for
full blend in, 0.0 for tiles image only (default 0.8)
  -I float
        intensity of mosaic images in float (0.0 - 1.0). 1.0 for full mosaic
images, 0.0 for input image only (default 0.8)
  -M string
        running mode: s=sequential(default), p=parallel, w=parallel with work
steal (default "s")
  -T int
        Number of goroutines. ignored if sequential. Must be positive
(default 1)
  -U int
        Input image upscaling in integer. Must be positive (default 1)
  -d string
        Path to the mosaic tiles directory
  -i string
        Path to the input image
  -o string
        Path to the output image
  -s int
        Size of mosaic tiles in pixels. Must be positive
```

There are several Go program files implemented in this project:

- The deque folder with both deque.go and index.go defines the double ended queue (deque) used in the work stealing version of the program.
- The utils.go represents miscellaneous functions used in the program, such as interpolate and array range generation.

- The png folder contains the structure of Image struct used in the program. It is divided into file.go which handles image file operation and effect.go which handle image effect implementation. There are several effects implemented in this file:
    - Resize, which scales an image into specified width and height by deciding input pixel location for each output pixel.
    - Subsize, which extract an portion of image based on specified rectangle position by deciding input pixel location for each output pixel.
    - ColorBlend, which combines a source color and destination color with a specified weight.
    - ColorTransfer, which transfer a color profile from a reference image into an input image. This program implements a simple version of color transfer, which is histogram matching (https://en.wikipedia.org/wiki/Histogram_matching). More specifically, this program translates this Python implementation of histogram matching into the Go version. (https://www.youtube.com/watch?v=JiQ5lzggUfU&t=151s&pp=ygUSaGlzdG9ncmFtIG1hdGNoaW5n)



Histogram Matching

- The scheduler folder is where the main execution of the program is located. There is a file for each implementation of the program (sequential.go, parallel.go, worksteal.go), and there is scheduler.go file which defines program configuration and forward them based on its version.
- Finally, there is the mosaic.go which handles the input flags of the program to make the program configurable.

How To Run

To run this project, go to “proj3” folder, then run “python3 benchmark/benchmark-proj3.py”. To ensure that the script run without problem, Python version 3 should be installed, with matplotlib package included. Before running the script, the dataset should be put into its respective folder. The dataset can be accessed using this link: proj3-muhauliaf-extra

Result

After running the script file, there should be several outputs produced, which are the followings:

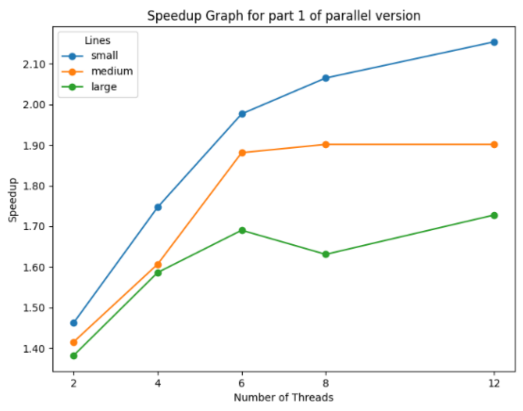
- Image output, which are the generated images from mosaic collage generator. There is one image output for each test size (small, medium, large)

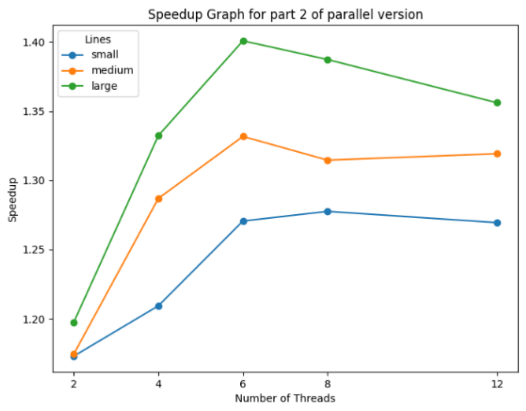
- The benchmark-proj3.json file, which contains runtimes for both parts, all versions (sequential, parallel, work stealing), all sizes (small, medium, large), and all threads for parallel versions.
- Graph images, which shows speedups of both parallel versions (parallel, work stealing) compared to sequential version for both parts.

Moreover, when running the proj3/mosaic.go program, assuming the input is valid, it will output two number, which is the runtime of first part and the second part of the program, respectively.

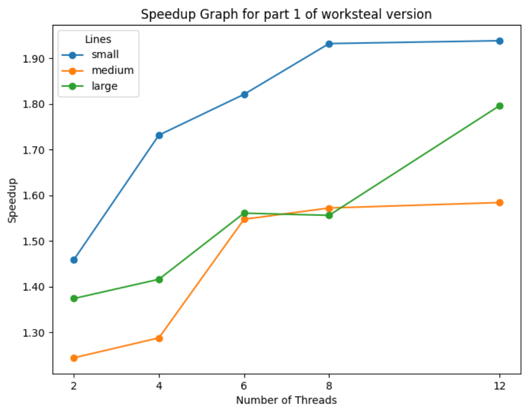
Here are the speedup graph after running the script in a laptop with this specification:

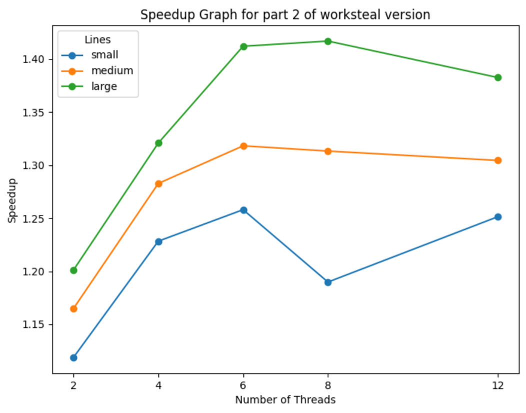
- Model: HP EliteBook x360 830 G7 Notebook PC
- Processor: Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz (8 CPUs), ~2.1 GHz
- Memory: 32768MB RAM
- Operation System: Windows 11 Pro 64-bit (10.0, Build 22631)





Speedup Graph of Part 1 and Part 2 of Parallel version





Speedup Graph of Part 1 and Part 2 of Work stealing version

Analysis

There are several insights that can be observed from those results.

In terms of performance, based on the speedup graph, the usage of a task deque with work stealing noticeably improves performance compared to sequential version. From my understanding, this work stealing version is basically a simple fork join parallel version, with enhancement where an empty thread will attempt to steal task from other thread's deque. Therefore, this should minimize idle threads from waiting for other threads to finish.

The main different of first part and second part of this program is that in first part, there are 20 tile images designed to be different in sizes, grouped into small difference (small), uniform difference (medium), and extreme difference (large). While in the second part, all the tiles have been resized into identical sized square tiles, thus guaranteeing uniformity.

Interestingly, this is reflected in the speedup result. In the part 1 graph, generally the parallel version performs better than the work stealing version. In this part, the inputs are designed to be varying in sizes, thus creating more stealing attempts. Because of overhead cost of stealing, and frequent yielding, the performance suffers a bit compared to parallel version. On the flip side, in the part 2 graph, the performance of work stealing thread is almost identical to the parallel version. This is because the inputs are guaranteed to be identical, making less to no chance of work stealing.

Another fascinating insight is that as shown the graph, the speedup of part 1 is generally higher than part 2. One possible reason for this is the part 1 consists of loading files and has smaller CPU operations than part 1, which is mostly CPU bound process. Therefore, the Go program can parallelize both I/O and CPU effectively in part 1, while in part 2, the speedup is limited by capability of CPU.

In this program, there are quite a lot of possible hotspots, where most of them have been parallelize, which are the tile image generations, and image effect application. This is because the tiles are independent of each other, minimizing critical sections of this program. As for bottlenecks, theoretically, the most significant ones are loading the image files. This is because this process depends on I/O operations which is significantly slower than CPU operations. However, I also parallelize this part, so I think the bottlenecks performance should be improved significantly, as shown in the part 1 graph. This may depends on how Go language manages the I/O tasks parallelization.

The most challenging part I faced while creating this program is the color transfer implementation, because I have to learn the concept and understand it fully to know what and how to implement this algorithm. Surprisingly, the channel implementation is quite easy once I know the tricks about how to use it. In my opinion, channel is one of the greatest benefits of using Go language, other than goroutine. On the other hand, the work stealing version is more difficult than channel, because I have to make sure that there are no race condition and ABA problem in this implementation. Another insight that I found when doing research for this project is that Golang is superior in terms of parallel CPU programming but maybe outshined by other languages which uses GPU computation, such as C/C++.