**Project 1 Report**

Hanze Hu

**Description**:

The overall project implements a Image Procesing System using Golang.

The png folder contains the definitions for working with images in golang, including loading, saving and updating png images, as well as the code for applying affects using convolutions to individual images. All code in the png folder are sequential.

The editor folder contains editor.go, which is used to parse in configuration data such as type of scheduler scheme to use and number of cores to use if it the scheme can run in parallel.

In the scheduler folder, there are three different implementations for three different types of scheduling schemes: **pipeline**- which implements a Fan-Out Fan-In design using channels for creation of iamge processing tasks and task result aggregation; **bulk synchronous parallel** (bsp)- which implements a bsp model technique to process the images step by step and synchronize between steps; and **sequential**- which processes all input images and effects sequentially, one by one.

**Testing**: You can run the benchmark by running **sbatch benchmark-proj2.sh**, it will launch a sequential python program that runs execute the sequential shcduling scheme, running each size sequentially, the in parallel with different number of threads (2, 4, 6, 8, 12) and different type of scheme. Each size and thread number are ran 5 times and their average is calculated.

**Speedup Graph and Analysis**:

**Pipeline Implementation**: As per the graph below, we can see that the general speed-up trend appears to decrease as we increase the number of threads on the local machine. I think this trend may be due to the fact that since we are spawning numThreads number of goroutines as workers, who are also spawning numThreads number of sub-goroutines, as the number of threads increases, the amount of overhead also increases drastically. The randomness that’s apparent on the Linux Cluster may also be due to having too many goroutines to manage and more randomness in the thread management of the cluster.

Chart, line chart

Description automatically generatedChart, line chart

Description automatically generatedLinux Cluster: Local Machine:

**BSP Implementation**: The graphs below, one ran on the linux cluster and the other on my local machine (m2 macbook air), shows that for all input image sizes, the bsp scheduling scheme is effective in reducing the runtime of overall processing, and can boost the performance of large datasets by up to 2 - 3 times. The dip in speedup for 8 threads on the linux cluster is likely an outlier, as it only appeared for small and mixture input sizes, and the trend did not show up on my local machine. While on the local machine, since it only had 8 cores, speedup was not significantly increased after 8 cores, and even slowed down slightly due to the increased overhead.

Chart, line chart

Description automatically generatedChart, line chart

Description automatically generatedLinux Cluster: Local Machine:

**Bottlenecks and HotSpots**: The hotspots in my sequential program mainly comes from applying the convolution filters to individual image pixels, due to the vast number of pixels in the input images (especially large images), the convolution and image processing step is expected to draw the most amount of computation resources. The bottleneck of my sequential program would come from reading and writing files. Since sequential implementation meant that no more than 1 image could be processed

**Performance Comparison**: In general, the pipeline implementation performed better than the bsp implementation, however, as we increase the number of threads, the speedup improves with the bsp implementation whereas the speedup decreases with the pipeline implementation, to the point where at 12 cores, bsp is performing better than pipeline (or at least at the same level).

**Problem Size**: From the above graphs, we can see that the speedup for all input sizes appear to be roughly the same (especially in the case of bsp implementation), and all input sizes produces very similar general speedup trends, therefore it can be argued that the problem size does not have a huge impact on performance. The reason may be due to the above-mentioned bottlenecks.

**Scheduler Type**: If go uses a N:1 scheduler, the entire program will only have one main thread, which means that only one go routine is effectively running on the hardware, which would produce performance comparable to our sequential implementation, and may even be worse due to overhead.

If go scheduler uses a 1:1 scheduler, spawning and destroying threads will incur much higher costs (not good for our implementation of pipeline), and the cost of switching between threads will also be much higher than the N:M scheduler so overall it should perform better than the N:1 scheduler but worse than N:M scheduler.

**Potential Improvements**:

Under the BSP scheduling scheme, we can potentially improve performance further would be to use separate go routines that are dedicated to reading and writing files, this way, the BSP could potentially start processing images before all images are loaded initially, and can save images as they are processed.