Project Description:

For this project I implemented a parallel system which applies effects (like Grayscale, Sharpen, Blur) to images. My system takes in input from users and can run in different modes: sequential, pipeline and bsp. The system takes in a JSON containing a set of input file paths, output file paths and list of effects to be applied to each image. The system reads in images from the input path, applies the effects to the image and then writes to the output path. Many of the effects which the system handles utilize 2d convolution, a kernel based approach described here: http://www.songho.ca/dsp/convolution/convolution2d_example.html

The sequential version of the system simply reads files in one by one, without using any parallelism. The pipeline approach spawns a number of different threads, which all synchronously handle different parts of the process. One thread reads in tasks from the json input file, multiple threads apply effects to the image passed in, and another thread writes images to output once they have been processed. I used golang's "channel" paradigm to pass data and signals between threads allowing them to coordinate and exit once they have finished their work. The final approach I used was bulky synchronous processing. In this approach, a number of threads work together on each step of the process ("superstep") and wait for their companion threads to complete a step before moving on to the next superstep ("global sync")

To run testing script: sbatch benchmark-proj2.sh

Performance analysis

The most significant bottleneck / hotspot in my sequential program occurs when effects are applied to the images. There are a very small number of images so reading / writing is trial but 2d convolution is a compute intensive process, especially on the large image files. Since my sequential process only allows one image to be processed at a time, this creates a significant bottleneck. My BSP implementation is running significantly faster than my pipeline. A large part of this is likely due to the fact that I failed to implement a system using mini-workers, so I'm not taking full advantage of parallelism. I also think that this task lends itself extremely well to a BSP approach. It has 3 clear "supersteps" (read, edit, write) which are all extremely parallelizable. The synchronization cost tied to barrier use is relatively low compared to the cost of using dozens of channels. Problem size had a moderate but not significant impact in both cases, especially at the larger end of number of threads. This relationship was not monotonic though, on the pipeline implementation some middle thread numbers performed worse than in the smaller samples On both my pipeline and BSP implementations, the gains from parallelization are probably more evident on the larger directories. An N:1 schedule would likely remove all performance gains and instead would probably make both BSP and pipeline worse than sequential. In an N:1 schedule, we would be incurring the overhead cost of spinning up threads without getting any parallelization benefits, since our system would be running sequentially. A 1:1 scheduler would likely have identical or even greater performance gains than N:M. To determine the exact impact of that, I would need to dig deeper to determine exactly what "M" is in this case.

I could improve performance by successfully implementing mini-workers. I also believe that my BSP approach would benefit from a more precise division of work among each thread. A more optimal way to divide work would take into account the number of effects on each image, and not simply base it on the number of images each thread has to handle. I'm also curious about lockfree or waitfree approaches that would remove the need to use a barrier in my BSP

Graphs:

Legend:

Blue - Small

Orange - Mixed

Green - Big