Parallel Programming - Project 2

Overview

This program implements a parallel approach to image processing. The sequential, pipeline, and BSP modes operate in the following manner:

Sequential

The RunSequential function initializes a json.Decoder, and runs in a for loop while there are still entries to be read. The json entries are decoded into a Request struct that holds the name of the image, the name out the out path, and the effects to apply. In addition the program reads in the directories to process from the instance of the config struct it receives from the scheduler. For each directory listed:

the sequential program loads the image from the directory, calls the convolute function

1. RunSequential loads the image from the directory
2. Applies each effect in sequence - calling the Convolute method
3. Updates the In attribute on the png.ImageTask struct to the memory address of the “Out” image
4. Saves the image once all effects have been applied

Pipeline

The pipeline version relies on four functions (RunPipeline, ImageLoader, Worker, MiniRoutine) and seven pipelines (imgStream, imgSliceStream, push, signal, stopGo, workerReturn). The program flows like so:

1. RunPipeline launches numThreads worker routines and sets up the workerReturn channel. It also calls the ImageLoader and passes the imgStream to the worker routines. RunPipeline then waits to receive numThreads signals on the workerReturn pipeline before returning
2. Image Loader sets up the image stream pipeline, and reads in requests from the decoder. It also initializes a series of required attributes in the ImageTask struct: directories to process, effects to apply, the name to save the file as. The ImageTasks are sent to the worker routines via the imgStream channel
3. Worker reads in ImageTasks from the imgStream and partitions the image into sections for each of the MiniWorkers it launches per ImageTask. It then sends the ImageTasks to the MiniWorkers via the imgSlice channel. Worker then coordinates applications of effects via the push and signal channel. Worker sends the effect to apply via the push channel, and then waits for signals from every MiniWorker before switching the In and Out pointers on the Image Task struct and sending the next effect to apply. Once all effects have been applied, Worker sends a signal on the stopGo channel to cause each MiniWorker to return. Worker returns when the imgStream is closed.
4. MiniRouinte takes in a single from the imgSliceStream and parameters for the section of the image to operate on. MiniRoutine waits for the Worker routine to send which effect to apply, calls Convolute, and sends a signal to Worker upon completion. MiniRoutine returns when it receives a signal on the StopGo Channel

Bulk Synchronized Parallel

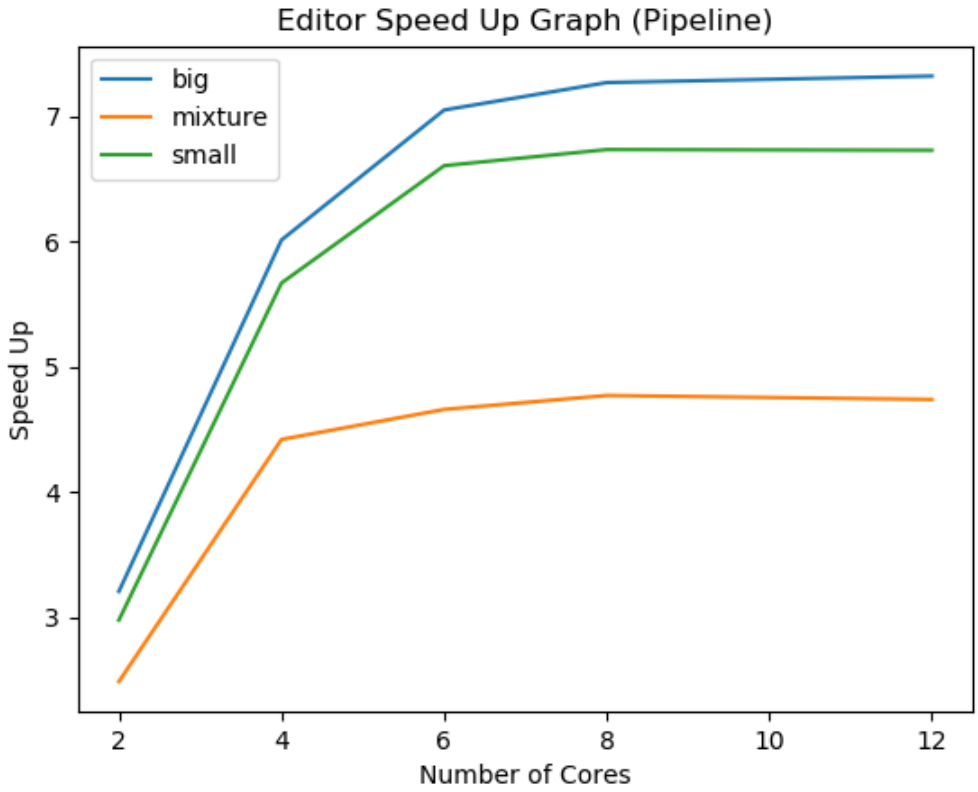
The BSP implementation uses applications of effects as its super step. Each thread is given a section of the image to work on based on its thread id, and the thread with the highest id is termed the lead thread and given the image “stub.” Threads apply the required effect to their section and wait for all other threads to complete (using a condition variable). The last thread to finish updates the EffectCount variable to point to the next effect, or updates the DirectoryIdx variable to point to the next directory to load the image from. The last thread to arrive at the top of the loop is also responsible for checking whether a new image needs to be loaded or if another request should be read in from the json decoder.

Instructions for Running Testing Script

The testing script can be launched via the following command: `sbatch benchmark-proj2.sh`.

Speed Up Analysis

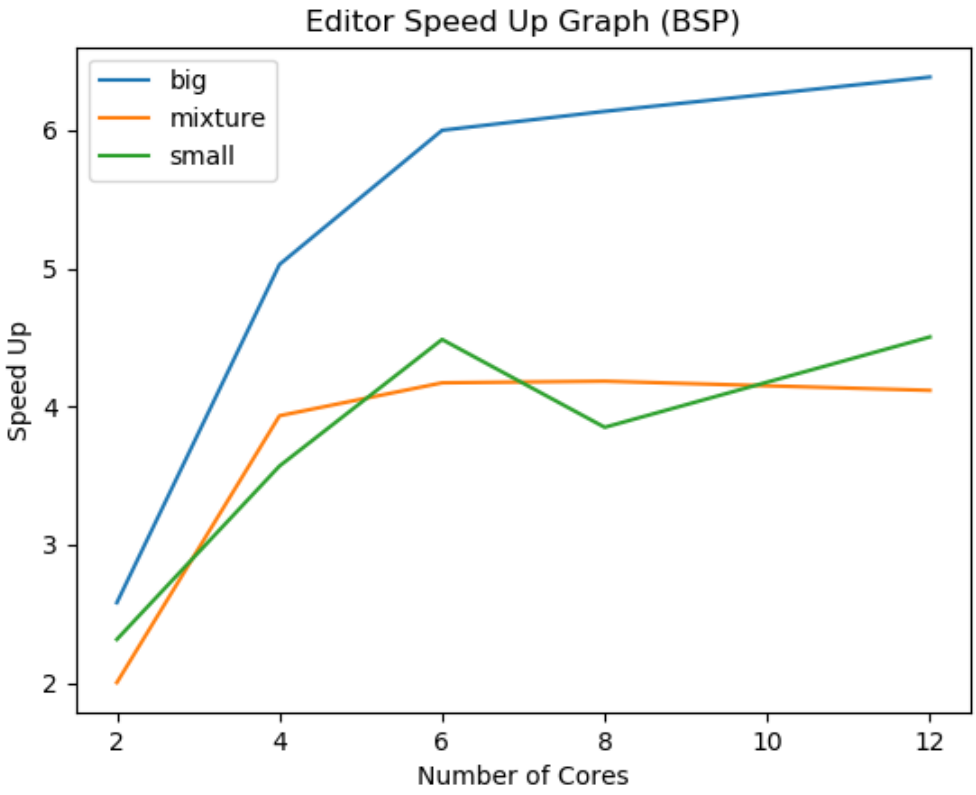
Pipeline



The speed up graph for pipeline shows what is expected. As the “problem” size increases, we see greater benefits from parallelism. This follows from the nature of the computations, each pixel transformation is independent of other pixels in the output file and operations of other threads. Of course, these benefits start to plateau as there are many sequential elements in the program (e.g., loading the requests from decoder). The plateau also occurs because each test only has 10 tasks. Therefore, two threads in the 12-core run will sit idle, limiting performance improvement over the 8-core test.

While the appearance of higher speedup in the small directory vs. for the mixture directory feels counterintuitive it makes sense given the “waiting” time some threads will experience once there are no more images to processes. Some threads will finish with a small image as their last task, and will have to wait for threads working on big images. This limits the benefits of parallelism significantly. However, in the tests for small and big, threads will complete their work at comparable rates because of similarities in task sizes, limiting idle time.

Bulk Synchronized Parallel



The speed up graph for BSP is close to what is expected. We see the highest speed up for the big directory, with lower speed up for small and mixture. Interestingly, the lines for the big and small directories suggests the program may run even faster if we add more cores. This makes sense given the even distribution of work across threads and the suitability of matrix multiplication for parallelism.

Lower speed up for the mixture directory vs. the small directory is puzzling because BSP is not subject to the same waiting issue as pipeline. Theoretically, the speed up for mixture should be the weighted average of the speedup for the big and small directories. It is unclear why this phenomenon is occuring.

The kink in the speedup for the small directory can be attributed to variation is system traffic, which processes like loading images are highly sensitive to. Previous tests showed monotonic increases for the small directory.

Short Answer Questions

- **What are the hotspots and bottlenecks in your sequential program?** The sequential program has hotspot for the convolution method as it requires millions of calculations to process a single small image. It is difficult to say whether the sequential program has any bottlenecks as it is always proceeding one task at a time
- **Which parallel implementation is performing better? Why do you think it is?** Pipeline is performing better than BSP because it has higher throughput of tasks. Pipeline processes N tasks for N threads, with N^2 threads working in total. Whereas, BSP applies N threads to a single task. There is also some inter-thread dependency in BSP because every thread moves in lock-step. While this exists for the MiniRoutines in pipeline, there is greater independence among worker level routines. BSP also suffers from lock contention, as every thread must access the lock twice per cycle - though it is possible to simplify to one access per cycle. Lock contention scales linearly with thread count.
- **Does the problem size (i.e., the data size) affect performance?** Problem size significantly affects performance. The larger the image, the greater the benefits of parallelism because overhead is amortized over a larger set of computations
- **The Go runtime scheduler uses an `N:M` scheduler. However, how would the performance measurements be different if it used a `1:1` or `N:1` scheduler?** This depends on the pattern. BSP would benefit from a 1:1 structure because every thread lives for the lifetime of the program. Pipeline would suffer in this set up because it spawns N threads per task per thread, and spinning up kernel level threads is expensive. Both patterns would suffer in an `N:1` scheme because there is no parallel execution

Scope for Performance Improvement

Implementing work stealing is the best opportunity for driving performance improvement in this program. Generally, threads in BSP or pipeline will idle at some point. This is especially true since image sizes can vary ~2x within big and small directories, and up to 20x in the mixture directory. Since the pixel transformations are independent, it would be easy for idle threads to steal work from other threads and continue operating.