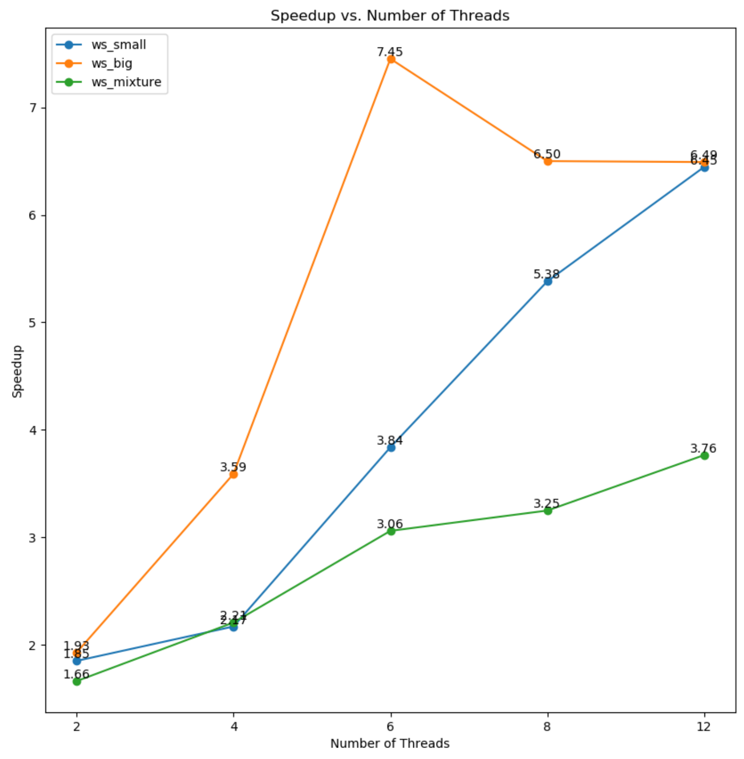
Project 3: Image Processing System + Work Stealing and Map Reduce

Project Description and Parallel Solution Description

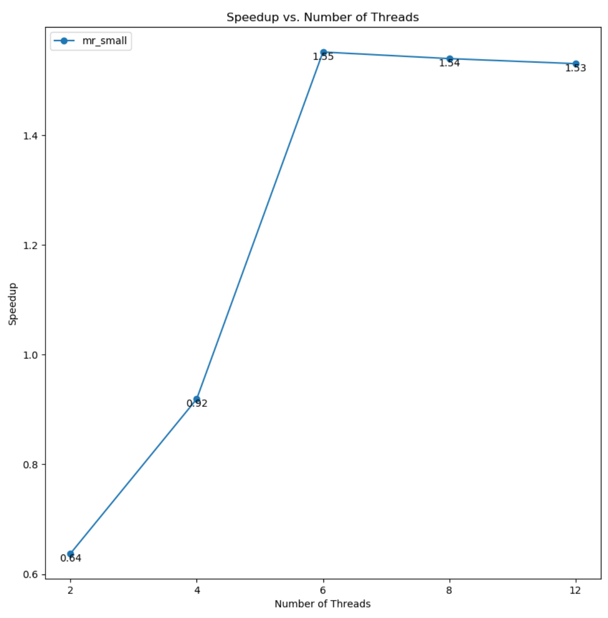
- This project uses golang to serially or parallelly apply grayscale, sharpening, blurring and edge detection effects on a set of images.
- The user of this project can run serial, or 2 types of parallel implementation for a given set of effects (eg. G, E, S - grayscale, followed by edge detection, which is then followed by sharpening of the image) on one or more images. In both these parallelization techniques, each thread gets whole images to process.
- **Lock Free Bounded Deque** and **Work Stealing Paradigm** - A double ended queue made using an array and large enough capacity, which is the Data Structure used to store the local goroutine queues. This is useful in implementing the Work Stealing Parallelization. The global goroutine queue is created using effects.txt and local goroutines are distributed to each thread by spinning them with "go" statement. Once a thread is finished with its task, it pops from the top of other processor's queues. To add to tasks however, it only pushes from the bottom.
- **Map Reduce -** Mappers read data from separate files, and emit intermediate results to a channel.The shuffler combines the intermediate results by grouping them based on the Region field. Each thread reduces images of a particular region using goroutines. Map Reduce could combine similar effects together, thereby improving load balancing for the program and making processing faster.
- **How to run your testing script -** `sbatch benchmark.sh`
    - The results files appear in the editor directory.
    - go run editor.go small ws 12
    - go run editor.go big mr 12
- **Describe the challenges you faced while implementing the system. What aspects of the system might make it difficult to parallelize? In other words, what did you hope to learn by doing this assignment?**
    - I was hoping to learn how to handle load balancing through this assignment. The challenges I faced were that despite the work stealing being implemented, there are other variables involved for speedup to improve such as the granularity of the tasks. For smaller tasks, work stealing gives nearly linear improvement as there is better load balancing.
- **Did the usage of a task queue with work stealing improve performance? Why or why not?**
    - It improved the performance when compared to no work stealing as implemented in assignment 1. (6.5 vs 6 in small and big tasks as well) I

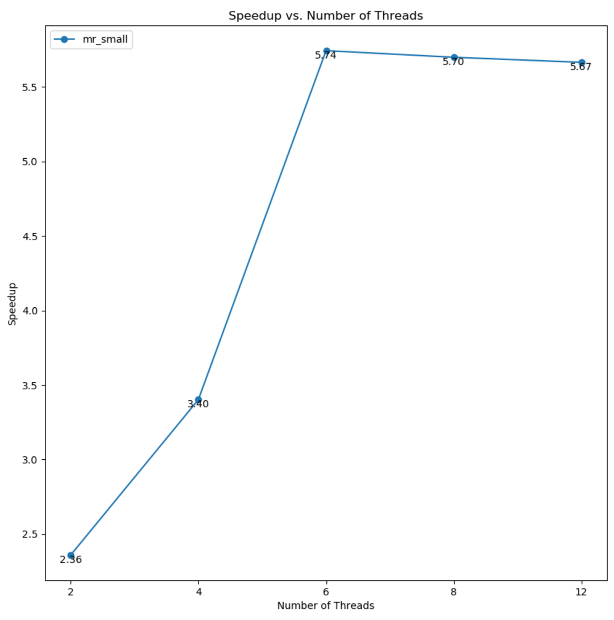
think the improvement is attributed to better load balancing especially for the mixture of images, where task distribution might be variable.

Results and Discussion



ws represents Work Stealing and mr represents Map Reduce





The first image is a speedup plot of small images processed using Map Reduce by using the sequential small image processing as a reference. The second image uses execution time of the Map Reduce code with 1 thread as reference. In my code, the mapping part could not be parallelized and that is a huge bottleneck as can be seen in the differences between speedups.

- **What are the hotspots (i.e., places where you can parallelize the algorithm) and bottlenecks (i.e., places where there is sequential code that cannot be parallelized) in your sequential program? Were you able to parallelize the hotspots and/or remove the bottlenecks in the parallel version?**
    - Bottleneck:
        - For the Work Stealing Parallelization, the generator can be considered the bottleneck since that hasn't been parallelized.
        - For the Map Reduce Parallelization, the map and shuffle were serial and the reduce part was parallelized. So the map and shuffle parts were the bottleneck.
    - Hotspots:
        - In the Work Stealing Paradigm, the hotspots include the idle workers trying to popTop on the same memory location thereby slowing down the working processor.

        - In the Map Reduce Parallelization, images from the same region were not the same in number. So one thread could end up processing 5 images whereas another could process just 1. The granularity of the task is variable and the load is not balanced.

- **What limited your speedup? Is it a lack of parallelism? (dependencies) Communication or synchronization overhead? As you try and answer these questions, we strongly prefer that you provide data and measurements to support your conclusions.**
    - In the Work Stealing Paradigm, it is possible that the optimal number of processors would be the number of images to process. The idle threads would only slow down the progress of others. That might be the reason there is a peak in my plot (at number of threads = 6) rather than a linear increase.
    - In the Map Reduce, there was a load imbalance and that greatly affected the speedups. The overhead associated with mapping, shuffling and transferring intermediate data between mappers and reducers became a limiting factor.

- **Compare and contrast the two parallel implementations. Are there differences in their speedups?**
    - There was a difference between speedups in the 2 parallel implementations.
    - In Work Stealing, mixture images benefited most consistently, which I believe is because of the evenness of work distribution due to work stealing. There might have earlier been some threads with small images which were idle.
    - In Map Reduce, I ran the code for only small images, and I believe that although the speedup improved a lot with increase in thread count, this will plateau since there are limited number of regions and each region is processed by a single thread.