Nichada Wongrassamee

Proj 1 Image Convolution System

Please submit a report (pdf document, text file, etc.) summarizing your results from the experiments and the conclusions you draw from them. Your report should also include the graphs as specified above and an analysis of the graphs. That is, somebody should be able to read the report alone and understand what code you developed, what experiments you ran and how the data supports the conclusions you draw. The report must also include the following:

Description

- Convolution Image processing on 10 images. Per each image we can apply multiple effects like Blur, B/W then Edges.
- For convolutional calculation, we'll be using kernel weights on the pixel and its 8 surrounding pixels. The sum of the effects will become the new image pixel.
- We're splitting the strategies into 3 types : sequential, parallel by files, parallel by slices.
- **Sequential** is working with each image and each effect step by step.
- **Parallel by files**
  - Implementation:
    - Spawning **n** go routines
    - Each routine is individually accessing the queue of images, pop from the queue and work until complete all the effects on the image.
    - I've implemented a TAS lock to prevent the race condition, which is when two go routines approaches the queue at the same time and preventing them from working on the same image.
    - When the queue becomes empty, the job is done.
- **Parallel by slices**, we split the images into x chucks. Each go routine is applying effect on each chunk parallelly.
  - Implementation:
    - Calculate the split area : how many rows in the image should one go routine to be working.
    - Pop the image from the queue one by one.

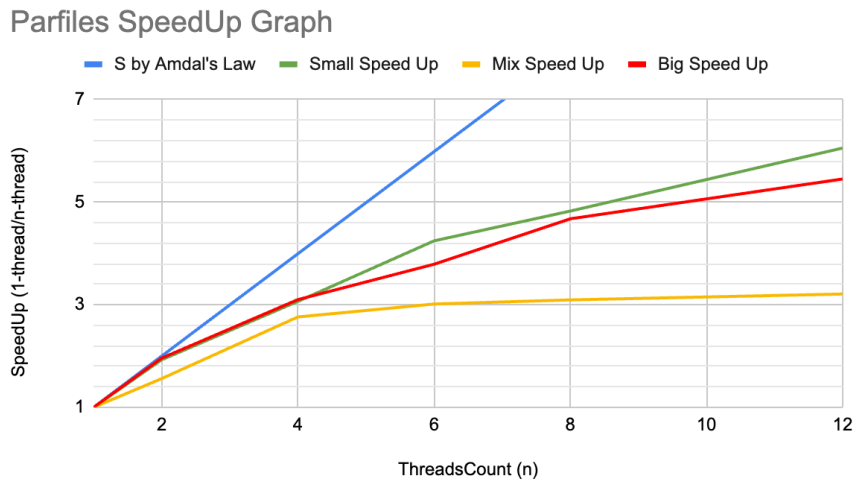
        - Per each effect, sprawl n go routines. Each go routine apply effect on its slice parallelly. Wait for everyone to finish one effect before moving to the next effect.
    - I tested the image on 3 images: big/ small/ and mixture/. Record the running time for {1,2,4,8,12} threads, compare the SpeedUp time when using different strategies.

Tests

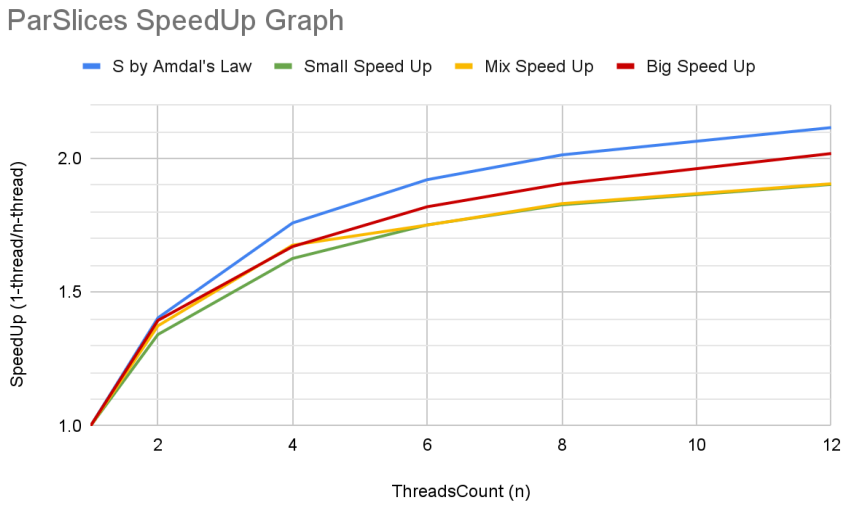
- Instructions on how to run your testing script. We should be able to just say `sbatch benchmark-proj1.sh`; however, if we need to do another step then please let us know in the report.
    - `sbatch benchmark-proj1.sh`
    - The log of times will be in the `benchmark/times/` folder. In the time_summary.txt, I recorded the minimum time after 5 experiments per each test. This is measured using the `time` unix command. I.e. `Par: parfiles, Size: mixture, Threads: 1, Min Time: 1m26.018s`
    - Log.txt is reporting the stdout, which shows how much time the program spent on the parallelized section. This time is measured using “time” package in go.
    - I recorded the data in this spreadsheet for graph plotting

Evaluation

- As stated previously, you need to explain the results of your graph. Based on your implementation why are you getting those results? Answers the following questions:



-



-
-
- What are the hotspots and bottlenecks in your sequential program?
    - Applying kernel calculation per each pixel consumes a lot of CPU resources.
    - Loading images to struct
    - Reading and writing pixels into img.in and img.out takes time.
- Which parallel implementation is performing better? Why do you think it is?
    - The parfiles are performing better because 99% of the program is parallelized, meanwhile parslices has only 50% of the program parallelized.
- Does the problem size (i.e., the data size) affect performance?
    - Yes, larger file takse more time.
    - The mixed file result in less SpeedUp performance due to the non-uniform distribution of task.
- How close are your programs to the expected speedup computed with Amdahl's law? If they are not close, why?
    - Amdahl’s law is assuming that in the parallelize portion, if we add n threads, the program will linearly n times faster.
    - Parslices is closer to Amdahl law overall. And among the Parfiles, small/ big/ folders are closer to Amdahl law.
      Here’s the analysis:
    - Parfiles(small/ big/) to Amdahl’s law:
        - Each parallel part is taking approximately equal time. This is evidenced in **small/** and **big/** results. These folders have similar sized image, therefore the computation time per each thread (each image) must be similar.

        - In the contrary, the **mixture/** folder SpeedUp line is not close to Amdahl's prediction. This is the result of non uniform work-distribution, and hence adding more threads does not significantly improve the speedup. Because the thread that is working on big file will still be working while those that finished the small files are waiting in idle.
    - Parslices is close to Amdahl's law:
        - Parslices has 57% of parallelize work. Within that portion, images are slices into pieces and n threads are working on the slice of the image on the same effect. This creates a very uniform load distribution per thread.
        - Therefore, it resonates with Amdahl's law which assumes the parallel work will increase the speedup linearly.
        - The graph does converge to 2, because of the 0.43 sequential task. Hence, it's approaching 2 as the sequential becomes dominant as n increases. S_max as the parallize portion is approaching 0 is 1/0.43 = 2.33.

Performance Improvement

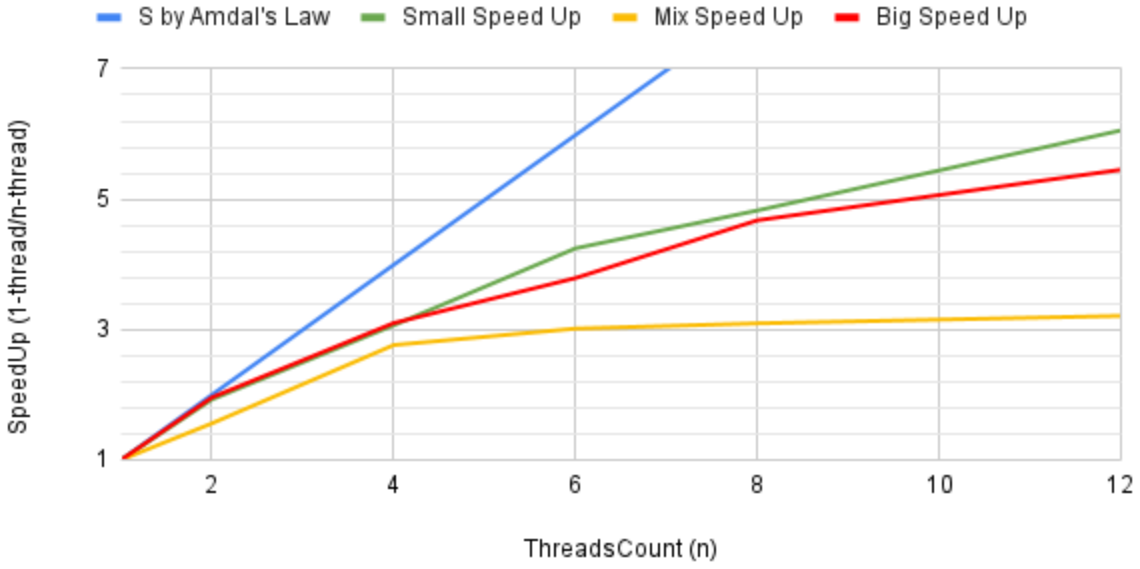
Based on the topics we discussed in class, identify the areas in your implementation that could hypothetically see increases in performance (if any). Explain why you would see those increases.

- **Parfiles**
    - Parfiles has an overhead when modifying the same shared queue resource. Since the work is mostly uniformed, the threads have contentions acquiring lock in order to pop from the queue at the same time. There'll be overhead here when threads are waiting for lock.
    - To improve, pre-allocate tasks before running go routines. We can assign x number of tasks for each routine if we assume that each task takes similar time to complete. We can avoid the contention for lock by this method.
    - Implement a queuing system for the threads and change the image queue into an array. Implement queueNumber on an atomic sharedCounter, each thread its number and read from Image task array in parallel. Getting the queue number atomically is faster than locking the entire queue and pop from it.
    - Alternatively, can use a concurrent data structure with lazy synchronization or lock free. This allows us to only lock the node

instead of the whole queue. The effect might not be significant because there's contention at the node.

- **Parslices** has almost 50% of sequential time. This sequential work includes : read input to queues, load image, loop through effects before it spawn go tasks. It also works on an image one by one. I will improve the performance by increase the parallelize work
- **Parallelize Image Loading**:
  - Sprawling the go routines to load its own chunk of image. Each go routine will load their own chunk -> work on E1 (effect1) -> synchronize with another routine and save the E1img1 -> work on E2. However, we still pop the image out sequentially.
- To improve further, I'll combine the parfiles and parslice approaches. Increasing throughput by allowing multiple images to be processed parallely and also speeding up when applying effects by slices.

Parfiles SpeedUp Graph



ParSlices SpeedUp Graph