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:

- Describe your program and the problem it is trying to solve in detail.

My program tries to revisit the Project 1 of the image processing system but by combining the whole project 1 in a pipeline and BSP fashion.

The Pipeline portion is as follows:

There are three pipelines
1) Image Generators -
The pipeline utilizes a single goroutine for reading images to process from effects.txt. This goroutine, known as the image task generator, reads JSON strings representing images and places ImageTask structs into a channel(ImageTask channel) with a capacity equal to the specified thread count.

2) Image Processing (Workers) -
This pipeline is responsible for reading the tasks from the channel(ImageTask channel). The main goroutine spawns workers mentioned as threadcount by the user which tries to take a task from the Image Task channel -> process it with more threads (spawned internally following BSP pattern) -> save the ImageResult to channel ImageResultChan parallelly.
3) Result Aggregator -
This Pipeline is responsible for saving the image results in the data/out directory. The main go routine spawn Result Aggregator which takes image results from imageResultChan and then saves it. If all the images are saved it signals the doneChan to convey that all the tasks are done in order to close the channel exit out of the goroutine. I have tried to implement two versions of the result aggregator when the number of workers in this pipeline is fixed and one when the goroutines vary according to threadcount mentioned by the user.

BSP portion:

As each worker is given image in their bounded queue when it processes the image using popBottom the thread -> spawn more threads equal to config.threadcount which

then breaks up the images into the N portions and implement each effect in a parallel way and wait at the end for all the goroutines to get done and each effect to be applied properly and then move onto next effect and process it the same ways. The barrier here is waiting for all the routines to get done and giving a more range of pixels so that at the end the effect will be neutralized at the end without forming any distortion.

Sequential Version:

The first and foremost step for a parallel programming project is to achieve all the required tasks in sequential order. This not only gives us a standard to compare our parallel programming approaches but also helps us in exploring various avenues of applying parallel programming techniques. This helps us to know the hotspots and bottlenecks present in the program.

A description of why the approach you picked (BSP, map-reduce, pipelining) is the most appropriate. You probably want to discuss things like load balancing, latency/throughput, etc.

I selected a hybrid approach incorporating elements of both pipelining and Bulk Synchronous Parallel (BSP) processing for several compelling reasons.

In my analysis of Project 1, it became evident that as the dataset size increased, bottlenecks emerged, particularly in tasks like image saving that lacked parallelization. While processing images, it became apparent that parallelizing the storage of each image could significantly enhance efficiency. However, utilizing the same thread for this task resulted in hotspots, where each worker had to wait for a lock to access the queue, leading to a sequential bottleneck. This limitation became more pronounced as the number of threads increased. The pipeline approach proved advantageous in this context, allowing the simultaneous execution of all three tasks—processing, saving, and queue management. This not only alleviates sequential bottlenecks but also maximizes concurrency.

To further enhance parallelism and efficiency, I integrated the BSP model for the superstep between each image processing effect. By breaking down images into nearly equal sizes, load balancing became less of an issue. This strategic combination of

pipeline and BSP models ensures faster overall image processing compared to a single-threaded approach.

Additionally, this hybrid strategy outperformed the performance achieved by the parslices and parfiles implementations in Project 1. The synergy between pipeline and BSP not only addresses the bottlenecks observed in the previous project but also capitalizes on the advantages of both models, resulting in a more efficient and scalable solution for image processing.

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?

Coordinating between multiple channels initially posed difficulties. However, as understanding deepened, channels not only served as data-sharing mechanisms but also played a crucial role in signaling task completion or determining when a worker should await exit.

Implementing a lock-free bounded dequeue for work stealing was particularly challenging. The intricacies of masking the stamp and top index using bit manipulation to convert a Double CAS into a single CAS required collaborative efforts with TAs and the professor. The successful implementation of this complex task felt like a significant achievement.

Did the usage of a task queue with work stealing improve performance? Why or why not?

The integration of a task queue with work stealing indeed improved performance. The primary advantage lies in load balancing. With tasks of varying sizes and types, certain workers could complete tasks faster than others. Work stealing effectively redistributed tasks among workers, alleviating the workload on specific threads. The increased speed observed during task stealing further validates its positive impact.

While task stealing instances might be less noticeable with fewer threads, the benefits become more pronounced as the number of threads increases, highlighting its utility in enhancing parallelism.

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?

Hotspots:

- A significant hotspot was identified in the limited number of result savers. As the number of threads increased, image saving emerged as a time-consuming task. Leveraging channels to dynamically adjust the number of result savers mitigated this bottleneck, resulting in improved overall speed.

Bottlenecks:

- Retrieving images from the ImageTask channel became a bottleneck, especially with a higher number of workers compared to the image task generator. This contention among workers for tasks from the channel impacted performance. Addressing this bottleneck could further enhance parallelism.

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.

The limitations in achieving optimal speedup were attributed to various factors:

- The bottleneck in image task retrieval from the channel limited the speedup potential, especially with a higher number of workers.
- The dynamic nature of image task generation, where the generator outpaced the ability of workers to process tasks, created contention and reduced overall performance.

Attached below terminal outputs shows the image task generator is faster as compared to other pipelines and but due to higher number of workers they all at the same time try to get the task from it and to its own queue.

//////////////My terminal outputs

Anshuls-MacBook-Air:editor anshulgupta$ go run editor.go mixture pipeline 6

Image added to taskchannel ../data/in/mixture/IMG_2029.png

adding task ../data/in/mixture/IMG_2029.png

Image added to taskchannel ../data/in/mixture/IMG_2724.png

adding task ../data/in/mixture/IMG_2724.png

Image added to taskchannel ../data/in/mixture/IMG_3695.png

adding task ../data/in/mixture/IMG_3695.png

Image added to taskchannel ../data/in/mixture/IMG_3696.png

Image added to taskchannel ../data/in/mixture/IMG_3996.png

Image added to taskchannel ../data/in/mixture/IMG_4061.png

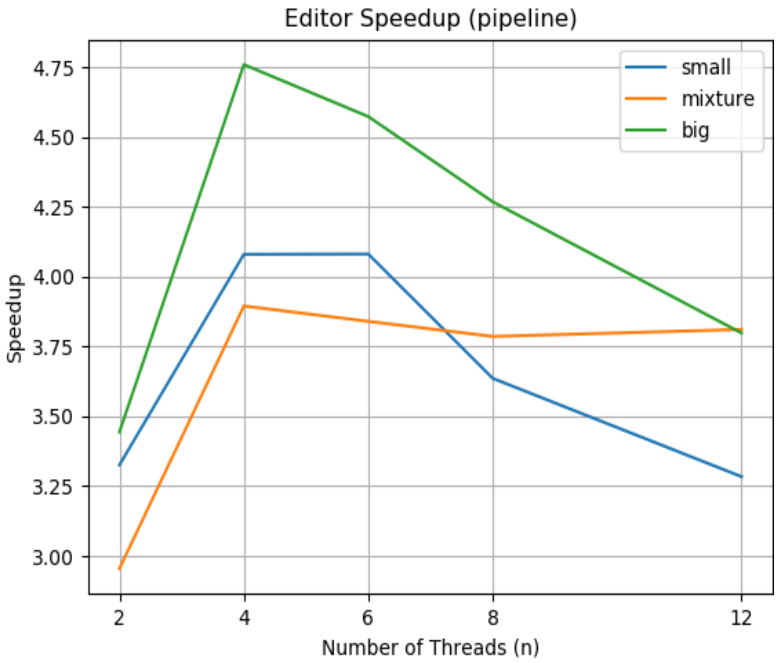
adding task ../data/in/mixture/IMG_3696.png

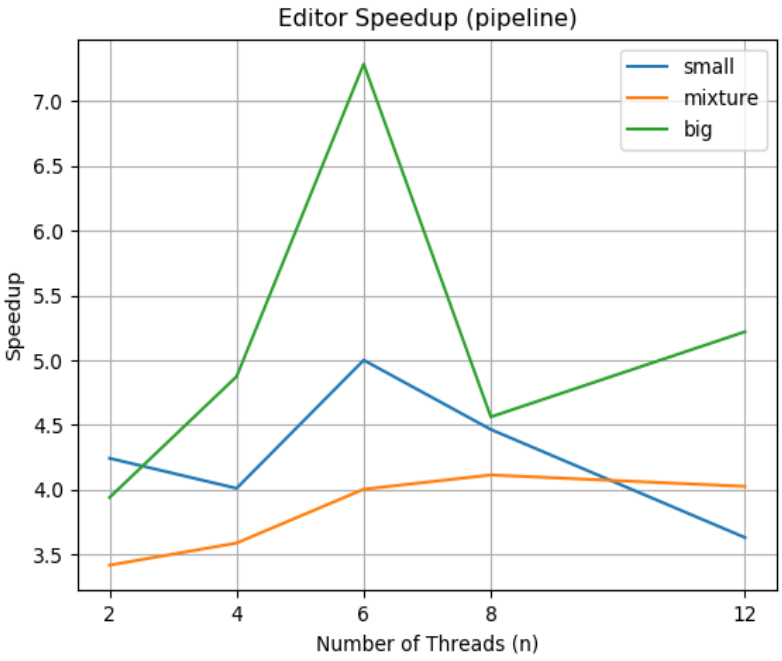
adding task ../data/in/mixture/IMG_3996.png
///////////////

Compare and contrast the two parallel implementations. Are there differences in their speedups?

The two implementations in my project are one without multiple result aggregators and fixed. The former had a higher speed up as compared to the latter one as the image saving is a time consuming task so spawning multiple threads in the pipeline increases its performance and the overall speed up as well. I have discussed the speed up graphs below for both the implementations.

Pipeline without varying results aggregator

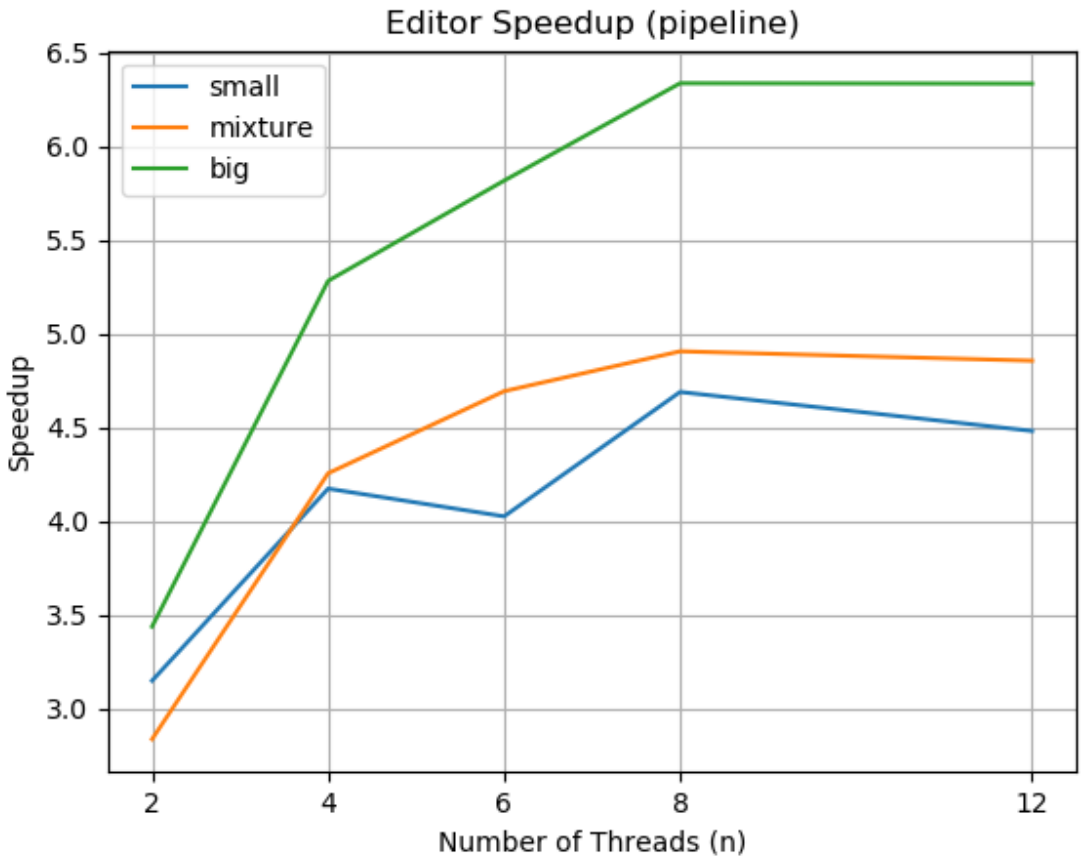




These were the graphs produced on the peanut cluster when 2 fixed result aggregator go routines were spawned. I added both the graphs as in graphs we had higher speedup for 6 threads and then the speedup after 6 threads decreases and is similar in both the cases. However, we have a higher speed up for all the cases in the right graph but there is a sudden decrease in it whereas in the other graph the decrease is relatively gradual.

Another observation is that speedup for mixture is less than the other two data sizes which indicates the need for load balancing in mixture is more required as it contains a mixture of small and big images and it might be the case that a particular thread might be assigned a higher number of big images. Though work stealing is definitely helping in a higher number of threads which is shown by the fact that speedup increases with increasing threads.
Additionally at a time there are only 10 images so it might be possible if we run all the 30 images then the speedup will be better because the effect of stealing will be more profound. In this case if we spawn more than 6 threads roughly each thread will get 2 images and some will get only1 so those threads might try to steal from all the others might be leading to performance issues.

Pipeline with varying results aggregator and workers

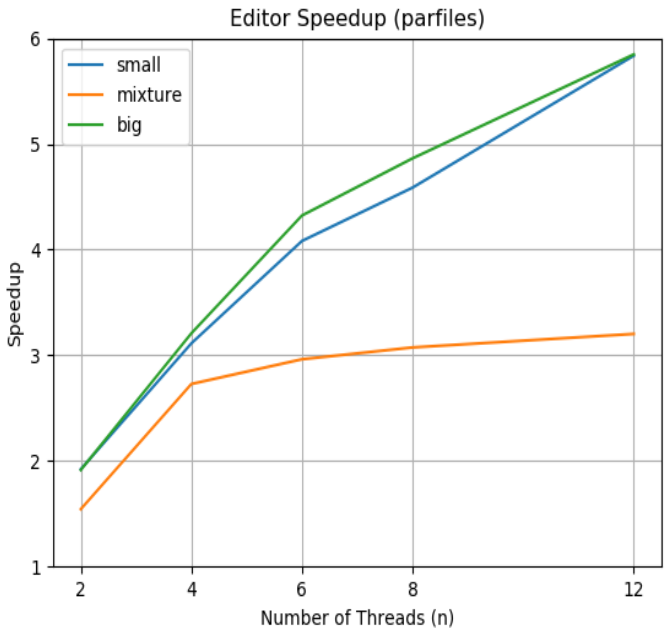


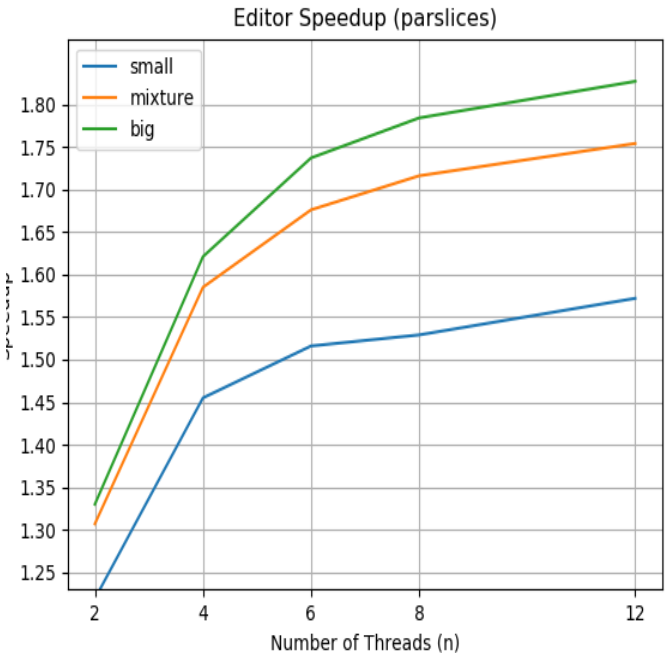
For this implementation I varied both the number of workers and the number of result aggregators. This shows as the number of aggregators increases with the number of threads the speed up increases and flattens. More speedup could be achieved as we plotted graphs only for individual data directories which have 10 images only so after 8 threads the speedup is constant.

The reasoning behind it is that as the pipeline for image processing has a higher throughput which creates a bottleneck in the next pipeline as a lot of results are present for the result aggregator channel to process and hence takes more time. But as we spawn more threads in the result aggregating stage the throughput for the pipeline increases and hence a higher speedup is also achieved.

One more observation is seen that though for small images the speedup is decreasing However, for medium and big the speedup is always increasing which shows a stronger size dependency.

Parslice and parfiles graph for reference.





Comparison between the above implementations to parslices and parfiles

The implementation of a pipeline with varying image savers worked out to be the best in all four cases. Technically because this portion kind of addressed all the shortcomings of all other three implementations. In parfiles where there was sequential bottleneck due to the lock we made it better using channels, and addressed load balancing using work stealing method which produced a direct dependency on size of the task which was missing in parfiles.
For parslices where we found that image saving in sequential manner is not the best approach and we addressed that by giving that task to several other threads so that workers can focus on processing the images only.

One key observation for my pipeline and BSP model combined implementation is that the initial startup at 2 threads is technically very close to the theoretical speedup which shows the effectiveness of this implementation. Moreover the saturation after 8 threads shows that work is being distributed properly because at a time we were giving 10 images. If we give more images the speedup also grows.

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.

To **generate the speed up plots** one can directly run sbatch benchmark-proj1.sh with the specified time so that it doesn't fail because of time run out error.
However there is a catch: the first will be produced when there are fixed (2) result aggregators so You need to do 4 modifications in total .

3 in scheduler/pipeline.go
Line 51,52 and 63 **from 2** in the both for loops and defining the capacity of the result channel **to config.Threadcount.**

1 in benchmark/plot_script.py

Change the name of plot to speed_up_vary2. Which is line 72

.

- **Does the problem size (i.e., the data size) affect performance?**

  Yes the problem size and the performance has a kind of direct relationship. As the problem size increases the time taken becomes higher . However with increasing size we also observe the performance of parallel implementation which can be shown through speed ups. The higher image size the higher the speedup achieved in both the cases with/without varying result aggregators.