Project 1: An Image Processing System

Sonia Sharapova

November 2024

1 Part 1: Sequential Implementation

In the sequential implementation, the program iterates over the provided images, applies their specified effects, and saves the new, edited images to their output files.

The program handles the convolution for the specified effects using a 3x3 kernel. Zero-padding is implemented for border pixels. For the greyscale effect, the program converts each pixel to grayscale by averaging the RGB channels and creates a new image with the modified values.

2 Part 2: Multiple Images in Parallel

Process multiple images in parallel, where each individual image is handled by only one thread.

Explanation of the Code

- RunParallelFiles reads each JSON entry, creates a Task, and enqueues it into TaskQueue.
- The number of goroutines is set to the minimum of the specified thread count and the number of tasks.
- Each goroutine (worker) locks the TAS lock, dequeues a task, unlocks, and then processes the task.
- The sync.WaitGroup ensures the main program waits for all goroutines to complete before exiting.
- Each task is processed by the processImage function, which applies effects in sequence and saves the result.

Timing: The parallelized sections of the code were timed.

3 Part 3: Parallelize Each Image

Parallelize the processing of individual images by dividing each image into 'slices'.

Explanation of the Code

- Iterate over the same queue as done for parallelizing image files
- Divide each image into slices, each assigned to a goroutine. The goroutines apply effects to their own slice of the image. Apply each effect across all slices.
- Use waitgroups to ensure that the processing of one image is completed before moving on to the next.
- Create overlapping slices to avoid boundary issues.

Boundary Handling: To handle overlapping boundaries within slices, the program checks if the next y coordinate falls within an overlapping areas with the current space. To ensure synchronization, this means that every goroutine has access to neighboring pixel data, meaning it has access to overlapping rows needed for convolution.

4 Part 4: Performance Measurements and Speedup Graphs

To following tables summarize the performances for running the three programs on the Linux clusters:

Performances for Sequential Implementation

Total Running Time for Sequential Implementation		
Small Images	Big Images	Mixed Images
20.33 s	227.01 s	87.29 s

Amdahl's Law:

S(N) = $\frac{1}{(1-P)+\frac{P}{N}}$

- $S(N)$ is the speedup with N threads.
- P is the parallelizable portion of the code.
- $1 - P$ is the sequential portion of the code.

Sequential Execution Time:

- Small Images: 20.33 s
- Big Images: 227.01 s
- Mixed Images: 87.29 s

$$\text{P} = 1 - \frac{\text{Parallel Execution Time with 1 Thread}}{\text{Sequential Execution Time}}$$

Performance Measurements for File Parallelization:

Execution Time for Images in Parallel (in seconds)			
Number of Threads	Small Images	Big Images	Mixed Images
1	19.63	217.97	92.58
2	8.3	112.45	60.33
4	5.44	86.38	34.59
6	4.12	72.23	36.9
8	4.24	67.75	32.65
12	3.69	50.92	34.2

Parallel Execution Time with 1 Thread:

- execution time for small images: 19.63 s
- execution time for big images: 217.97 s
- execution time for mixed images: 92.58 s

P Calculations

- Small Images: $1 - \frac{19.63}{20.33} = 0.0344$
- Big Images: $1 - \frac{217.97}{227.01} = 0.0398$
- Mixed Images: $1 - \frac{92.58}{87.29} = -0.0606$

Expected Speedup Using Amdahl's Law

Small Images (P=0.0344)

- 2 Threads: $\frac{1}{(1-0.0344)\frac{0.0344}{2}} = \frac{1}{0.9654+0.01713} = 1.015$
- 4 Threads: $\frac{1}{0.9654+\frac{0.0344}{4}} = \frac{1}{0.9654+0.0087} = 1.023$
- 6 Threads: $\frac{1}{0.9654+\frac{0.0344}{6}} = \frac{1}{0.9654+0.00577} = 1.029$

- 8 Threads: $\frac{1}{0.9654+\frac{0.0344}{8}} = \frac{1}{0.9654+0.004325} = 1.032$
- 12 Threads: $\frac{1}{0.9654+\frac{0.0344}{12}} = \frac{1}{0.9654+0.00288} = 1.034$

Big Images (P=0.0398)

- 2 Threads: $\frac{1}{(1-0.0398)+\frac{0.0398}{2}} = \frac{1}{0.9602+0.0199} = 1.022$
- 4 Threads: $\frac{1}{(1-0.0398)+\frac{0.0398}{4}} = \frac{1}{0.9602+0.00995} = 1.032$
- 6 Threads: $\frac{1}{(1-0.0398)+\frac{0.0398}{6}} = \frac{1}{0.9602+0.00663} = 1.032$
- 8 Threads: $\frac{1}{(1-0.0398)+\frac{0.0398}{8}} = \frac{1}{0.9602+0.00498} = 1.038$
- 12 Threads: $\frac{1}{(1-0.0398)+\frac{0.0398}{12}} = \frac{1}{0.9602+0.00332} = 1.040$

Mixed Images (P=0.4553)

Since P is negative, we treat it as zero (parallelization doesn't improve performance)

Calculating Speedup for Infinite Threads (Theoretical Limit)

Amdahl's Law simplifies to: $S(\infty) = \frac{1}{1-P}$

Inf. Threads Speedup

- Small Images (P = 0.0346):
  $S(\infty) = \frac{1}{1-0.0346} = 1.036$
- Big Images (P = 0.0398):
  $S(\infty) = \frac{1}{1-0.0398} = 1.041$
- Mixed Images (P = 0):
  $S(\infty) = \frac{1}{1-0} = 1$

File Parallelization: Results

The expected speedups for the Parallel Image Files method based on Amdahl's Law are minimal due to the low values of P calculated above. The theoretical speedups for even a high number of threads (up to 12) are close to 1 for all datasets, with a maximum of around 1.04 for the small and big images datasets and no significant improvement for the mixed images. In theory, this indicates that the image file processing task is largely sequential with only a small portion that benefits from parallelization. In reality, however, speedup with increased thread count is observed. Based on the results in Figure 1, a higher number

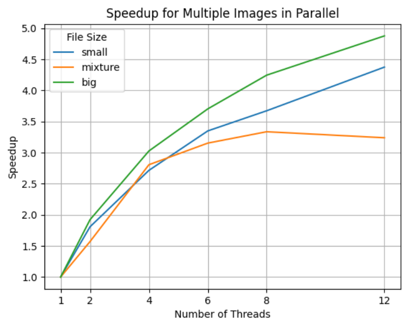

Figure 1: Speedup graph for parallelizing image files.

of threads leads to speedup in all datasets, with the big dataset showing the greatest speedup. This shows that in a practical scenario, the parallel image file processing improves the image processing tasks. The low/no speedup estimated with Amdahl's law may have been due to overhead when running on 1 thread.

Performance Measurements for Parallelizing Across Slices:

Execution Time for Parallelizing Img Slices (in seconds)			
Number of Threads	Small Images	Big Images	Mixed Images
1	6.66	110.96	47.54
2	6.19	85.9	40.03
4	5.63	85.05	38.46
6	5.58	81.27	38.04
8	6.39	78.41	35.15
12	6.38	78.16	34.76

Parallel Execution Time with 1 Thread:

- execution time for small images: 6.66 s
- execution time for big images: 110.96 s
- execution time for mixed images: 47.54 s

P Calculations

- Small Images: $1 - \frac{6.66}{20.33} = 0.6724$
- Big Images: $1 - \frac{110.96}{227.01} = 0.5112$
- Mixed Images: $1 - \frac{47.54}{87.29} = 0.4553$

Expected Speedup Using Amdahl's Law

Small Images (P=0.6724)

- 2 Threads: $\frac{1}{(1-0.6724)\frac{0.6724}{2}} = \frac{1}{0.3276+0.3362} = 1.5234$
- 4 Threads: $\frac{1}{0.3276+\frac{0.6724}{4}} = \frac{1}{0.3276+0.0.1681} = 1.8669$
- 6 Threads: $\frac{1}{0.3276+\frac{0.6724}{6}} = \frac{1}{0.3276+0.1121} = 2.1177$
- 8 Threads: $\frac{1}{0.3276+\frac{0.6724}{8}} = \frac{1}{0.3276+0.0841} = 2.3276$
- 12 Threads: $\frac{1}{0.3276+\frac{0.6724}{12}} = \frac{1}{0.3276+0.0560} = 2.6644$

Big Images (P=0.5112)

- 2 Threads: $\frac{1}{(1-0.5112)+\frac{0.5112}{2}} = \frac{1}{0.4888+0.2556} = 1.5974$
- 4 Threads: $\frac{1}{(1-0.5112)+\frac{0.5112}{4}} = \frac{1}{0.4888+0.0.1278} = 1.7651$
- 6 Threads: $\frac{1}{(1-0.5112)+\frac{0.5112}{6}} = \frac{1}{0.4888+0.0852} = 1.9087$
- 8 Threads: $\frac{1}{(1-0.5112)+\frac{0.5112}{8}} = \frac{1}{0.4888+0.0639} = 2.0024$
- 12 Threads: $\frac{1}{(1-0.5112)+\frac{0.5112}{12}} = \frac{1}{0.4888+0.0426} = 2.0436$

Mixed Images (P=0.4553)

- 2 Threads: $\frac{1}{(1-0.4553)+\frac{0.4553}{2}} = \frac{1}{0.5447+0.2276} = 1.5266$
- 4 Threads: $\frac{1}{(1-0.4553)+\frac{0.4553}{4}} = \frac{1}{0.5447+0.0.1138} = 1.6765$
- 6 Threads: $\frac{1}{(1-0.4553)+\frac{0.4553}{6}} = \frac{1}{0.5447+0.0759} = 1.7603$
- 8 Threads: $\frac{1}{(1-0.4553)+\frac{0.4553}{8}} = \frac{1}{0.5447+0.0569} = 1.8236$
- 12 Threads: $\frac{1}{(1-0.4553)+\frac{0.4553}{12}} = \frac{1}{0.5447+0.0379} = 1.8179$

Inf. Threads Speedup

- Small Images (P = 0.6724): $S(\infty) = \frac{1}{1-0.6724} = \frac{1}{0.3276} = 3.0524$
- Big Images (P = 0.5112): $S(\infty) = \frac{1}{1-0.5112} = \frac{1}{0.4888} = 2.00459$
- Mixed Images (P = 0.4553): $S(\infty) = \frac{1}{1-0.4553} = \frac{1}{0.5447} = 1.8364$

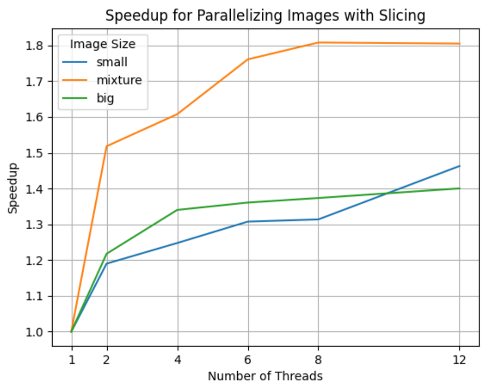

Figure 2: Speedup graph for parallelizing image slices.

Results

The expected values align more closely with Amdahl's Law predictions when parallelizing over slices vs. parallelizing the files. As the number of threads increases, the program has a higher speedup and the code executes faster.

5 Part 5: Performance Analysis

Project Summary

This project involved implementing sequential and parallel versions of a program used to apply image processing effects onto specified images. These effects were applied using convolutions that would grayscale, blur, or sharpen the images. In the sequential implementation, the images were processed one at a time. For the parallelized analysis, two techniques were applied: in the first, multiple images were processed in parallel with each image handled by a separate thread, and in the second, each image was divided into slices and the slices were processed in parallel within a single image.

The program's performance was measured across three datasets of varying sizes (small, big, mixed), and their respective speedups were calculated to evaluate the efficiency of the parallelization strategies. To interpret the results, the execution times were compared to theoretical predictions from Amdahl's Law which gave insight into the effectiveness and scalability for each parallel implementation.

Program Usage

Within the 'benchmark' directory, run:
$ sbatch benchmark-proj1.sh

This script will execute two python scripts for file and slice parallelization, respectively. The python code runs the effect application program with different thread counts for each dataset and record the execution times. After this analysis, it plots the speedups for the respective programs.

Results

Parallelizing Across Image Files:

The speedup increases with the number of threads for all datasets, with the 'big' dataset achieving the highest speedup, reaching nearly 5x with 12 threads. This behavior aligns with the expected benefit of parallelization when processing independent images. The more images there are, the more effective parallelization becomes because each thread can work on a separate image without interference. The big dataset benefits the most due to the larger number of images and higher individual processing times, which allows for better load distribution across threads.

Parallelizing Across Image Slices:

The speedup is much lower compared to the Parallel Image Files method, with a maximum of around 1.8x for the 'mixture' dataset and limited gains for the other datasets. The limited speedup is due to the overhead associated with managing inter-slice dependencies in each image. The need for synchronization between slices to ensure effects are applied in the correct order introduces bottlenecks, reducing the efficiency of this parallelization approach. Furthermore, small images may not have enough data per slice to fully utilize each thread, leading to overhead outweighing performance gains. Overall, the execution times across multiple threads were pretty, regardless of how many mult. threads were introduced.

In terms of speedup, parallelization across image files performed better than the slice parallelization. Parallelizing at the image level allows each thread to work independently on a separate image, minimizing the need for inter-thread communication and synchronization. In contrast, the slicing approach has inherent synchronization overhead due to dependencies across slices, which reduces its efficiency.

Hotspots and Bottlenecks: The main hotspot in the sequential program is the image processing loop, where each effect is applied to every pixel of each image. For large images or datasets with many images, this section dominates the execution time. The bottlenecks arise from portions of the program that cannot be parallelized, such as the initial setup, reading from the effects, and preparing images for processing. Additionally, the parallel versions have portions that remains effectively sequential, which limits the potential speedup that can be achieved as these tasks do not scale with the addition of more threads.

Impact of Problem Size: The size of the dataset significantly affects performance. Larger datasets (like the big dataset) achieve higher speedup with more threads because there is more work to distribute, reducing the relative impact of synchronization and other overheads. Smaller datasets or individual images with fewer pixels do not benefit as much from parallelization due to limited work per thread and higher relative overhead.

Comparison with Amdahl's Law: The expected speedups using Amdahl's Law were computed using the total running time with with parallelization and not just the parallel regions, so this may have led to less accurate estimations. The speedups for parallelizing the program varied from the expected results for both parallelization strategies. In parallelizing across files, there was very little speedup expected with an increased number of threads, but I observed a speedup of up to 5x as the file size and number of threads increased. The discrepancies between the expected and observed results may be due to error in calculation, or from overhead in the program's execution. My results for parallelizing across image slices more closely resembled predictions from Amdahl's Law, but still weren't a perfect match. For the 'mixture' dataset, my observed results matched the expected with a speedup of 1.8x for 12 threads, but my observed speedups were a bit lower than expected for the other two datasets.

Potential Performance Improvements: Since the thread communication handling overlap in slice parallelization introduced overhead in the program execution, reducing the need for synchronization between slices or employing a different technique for this could improve performance. Reducing the dependency between slices would allow more independent work per thread, potentially increasing the effective speedup. Combining both parallelization approaches (processing multiple images in parallel and processing slices in parallel within each image) could also offer benefits for very large datasets. For instance, using multiple threads to process different images, while applying the slicing method to each image, could balance the workload more effectively across large datasets.

Conclusion

This experiment demonstrates that parallelization can significantly improve image processing performance, particularly when the workload is divided across multiple images. This approach scales well with the number of threads and data size, closely matching the theoretical speedup predicted by Amdahl's Law. However, the Parallel Image Slices approach shows limited speedup due to synchronization overhead.

Overall, this analysis highlights the importance of choosing the appropriate parallelization strategy based on data size and workload characteristics, as well as the challenges posed by inter-thread dependencies in slice-based parallelization.

References:

Links to resources used:

RGBA to Greyscale: Effects in Golang
Using Goroutines on Images: Image Processing
Parallelizing Image Manipulation: Applying Effects
Parallelizing over File Lines: File parallelization
Calculating Speedup with Amdahl's Law: Stack Exchange
Matplotlib (used as a reference): Documentation
Python Regular Expressions: Documentation

+ Lecture notes and examples from class and golang