Performance Analysis

The objective of the project was all about diving into the world of parallel processing in Go to speed up image processing tasks like sharpening, edge detection, and blurring (to perform these effects I have written a universal code in **png/neweffects.go** which can take the starting and endingY positions of an image and perform these effects to it). I used two parallel processing methods: handling multiple files at once (**parfiles**) and dealing with slices of each image concurrently (**parslices**). I used these techniques on 3 kinds of image dataset (small, mixture, and big). Then, I dug into the performance data to see how these parallel methods stacked up against the traditional sequential approach.

**Instructions for running the script**

The benchmark script for the Peanut cluster can be executed by running the **sbatch benchmark-proj1.sh** command. I have introduced two more python scripts called **benchmark.py** and **graph.py.**

Benchmark.py helps in running all possible combinations of parallel methods and image datasets **5 times.** It then gets the minimum time for a particular combination and at the ends stores the pandas dataframe into an excel file **results.xlsx**.

Graph.py reads the results.xlsx file and then calculates the speed up using the formula of time taken by 1 thread divided by time taken by the nth thread. It then stores the speed up values into a new sheet in results.xlsx called **Speedup**. After that, it generate the graphs from both the parallel methods for further analysis.

**Sequential Analysis**

The main issue in the sequential program comes from the convolution operations needed for sharpening, edge detection, and blurring effects. These are big on the computation front. The snags happen because each image and each pixel within the images are processed one after the other, making the processing time shoot up linearly with more images or bigger image sizes.

Getting into the details of these bottlenecks:

1. **Convolution Operations:**
   * The heart of the issue lies in the convolution operations needed for sharpening, edge detection, and blurring. Each convolution is a math-heavy affair, iterating through each pixel of the image and its neighbors. The more pixels, the longer it takes to get through the pleasantries.
2. **Single-Threaded Execution:**
   * The sequential program is a lone ranger, processing one image at a time with a single thread. This method stands in its own way when it comes to scaling up. More images leads to longer wait.
3. **File I/O:**
   * File operations are also a bottleneck. Reading from and writing to disk are necessary, but they’re not speedy.
4. **No Concurrent Execution:**
   * The lack of concurrency is a big bottleneck. While one image is being processed, the others are just sitting there. It’s a lot of wasted potential and resources.

**Parallel Implementation**

When I compared the **parfiles** and **parslices** methods, it was clear that **parslices** had a bit of an edge, especially with the 'big' data directory. This is likely because**:**

1. **Better Utilization of Resources:**
   * The **parslices** method divides each image into slices and processes these slices concurrently. This way, it's not just handling multiple images at once, but also breaking down larger images into more manageable pieces.
2. **Balanced Workload:**
   * The **parfiles** method was like having each worker on a different task, but if one work is way messier (read: bigger image), that worker is stuck working longer. On the other hand, **parslices** is like having all your workers work together on each task, one image at a time, making sure no one gets bogged down with a tougher task.
3. **Handling Bigger Images:**
   * When it came to the big data directory, **parslices** really shone. Larger images are a tough nut to crack, but by slicing them up, **parslices** made sure the workload was spread out evenly. It’s like breaking down a huge, intimidating project into smaller, more manageable tasks.
4. **Less Waiting Around:**
   * In **parfiles**, while one thread is slogging through a big image, others might be twiddling their thumbs after breezing through smaller ones. But with **parslices**, all threads are kept busy most of the time, making the whole process more efficient.

Here are the graphs comparing the results from **parfiles** and **parslices**

A graph with lines and numbers

Description automatically generatedA graph with different colored lines

Description automatically generated

**SpeedUp Analysis**

The speedup I saw didn't quite match up with the theoretical speedup Amdahl's Law predicted, especially when more threads joined the party.

Amdahl's Law provides an ideal framework to anticipate the potential speedup when parts of a program are parallelized. Applying this law, expectations were set for the speedup achievable as the number of threads increased from 2 to 12.

Upon reviewing the actual speedup garnered from the parallel implementations, it's noticeable that the real-world speedup doesn't exactly mirror the optimistic projections of Amdahl's Law. Several factors contribute to this deviation:

1. **Overhead of Parallelism:**
   * The overhead incurred from managing multiple threads, synchronizing data, and other parallel coordination efforts isn’t accounted for in Amdahl’s simplistic model. These overheads impede the attainment of the theoretical speedup projected by Amdahl's Law.
2. **Non-Uniform Work Distribution:**
   * The workload distribution among threads wasn’t always uniform, especially in the **parfiles** implementation. Amdahl's Law assumes a perfect division of labor which wasn't the case here, hence the actual speedup trailed the theoretical one.
3. **Sequential Portions:**
   * Amdahl's Law hinges on the premise that a significant portion of the program is parallelizable. However, there are inevitable sequential segments in the code which limit the speedup, aligning with Amdahl’s assertion that the sequential part of a program becomes a bottleneck as the number of processors increases.
4. **Memory Contention:**
   * With multiple threads attempting to access shared resources or memory, contention occurs. This contention can stall threads, again moving the actual performance away from the idyllic scenario depicted by Amdahl's Law.
5. **Variability in Data Sizes:**
   * The varied sizes of data in the 'big', 'mixture', and 'small' directories also played a role in the disparity between actual and expected speedup. Amdahl's Law doesn’t factor in such data variability which could significantly impact performance.

In light of these factors, the actual speedup achieved leaned towards being more modest compared to the theoretical speedup computed through Amdahl’s Law. This analysis reinforces the understanding that while Amdahl's Law gives a high-level insight into potential speedup, the real-world intricacies and overheads of parallel computing often lead to a lower-than-expected speedup.