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 include your plot(s) as specified above and a self-contained report. 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. You must analyze your graphs and explain the outcome of your experiments!

Analysis

The report must also include the following (along with your analysis of your experiments):

- **A description of how you implemented your solution. This should include a description on any additional libraries you used/implemented.**

The overall problem was to implement a series of filters (or none or one) to an image. The general methodology is to apply a convolution kernel to each pixel of an image (or a grayscale effect) and save the final filtered image after all filter effects are implemented on all pixels of the image. I use three libraries to solve this problem:

1. png - The png library houses the Image data structure which consists of an in image (image.Image) field and an out image (*image.RGBA64) field. The in image field stores the image/pixel data from the original image read from standard input (initially). The out image field is a blank canvas the size of the in image to save the filtered image to after the effect is applied. The Load(filePath string) method creates the Image data structure by reading in the inPath of the original image, and then sets the in and out image fields accordingly, as described above. The Save(filePath string, noChange bool) method saves the filtered image in the out field in the Image data structure to a given file. If there is more than one filter, the UpdateInImg() method replaces the in image field with the out image field so that the filtering effects can be compounded for the next effect. The clamp method ensures that the rgb value of a pixel of an image is never greater than 65,535 or less than 0.

   In this package are also the three kernel used for "S," "E," and "B" convolutions and an ApplyConvolution(effect string) method that sends the kernel associated with the effect to the Convolute function in the effects.go file (also in the png package). There is also the GrabChunk(yMin int, yMax int) method that takes a chunk of an image by grabbing the subsection of the full image from the yMin through yMax pixels (and all x-pixels in that range) and creating a new Image out of that, where the in image is the portion of the original image and the out image is a blank canvas the size of the image portion. There is also a NewImage method to create a blank canvas for recombining image chunks in the parallel implementation, and a ReAddChunk method to add the filtered image chunk to the blank image canvas created with NewImage.

   In the png package is also a program called effects.go. This program applies the convolution to the in image of the Image data structure or simply applies the grayscale effect (that does not require convolution) and saves the filtered result to the out image field of the Image data structure using the Set method.

2. imagetask - The imagetask library houses the ImageTask data structure which consists of the unmarshaled json data read from stdin (i.e. inPath, outPath, and the effects slice). It also has an Image data structure loaded in to the Img field, which is created by calling the Load(inPath) method in the png library. It also contains YPixelStart, YPixelEnd, and ChunkPart fields for use in the parallel implementation to help with splitting up the image and putting it back together. In the imagetask library are all of the methods that manipulate the ImageTask data structure. This includes a CreateImageTask(imageTask string) method that takes in the stdin json data and returns an ImageTask with the inPath, outPath, and effects fields filled in from the json data and using png.Load(inPath) to create the Image data structure to house the image's pixels. In also contains SaveImageTaskOut(), which save's the ImageTask's Image out field to the directory by calling the png's package Save() function. Finally, there is a SplitImage(threads) method that splits up an ImageTask's Image in to approximately even slices based on the number of threads and number of y-pixels. What is returned from here is an array of ImageTask's that are chunks from the original ImageTask's Image. The inPath, outPath, and effects are the same for these new ImageTasks. The Img field is now an Image data structure housing the pixels of the chunk from the original Image (the in image field), which is gotten by calling GrabChunk(yMin int, yMax int) in the png library, and an empty canvas the size of the chunk for saving the filtered Image (the out image field) later. It also contains the YPixelStart, YPixelEnd, and ChunkPart fields to identify what chunk of the original image this portion of the image is for recombining the full image, later.

3. editor - The editor library implements the sequential and parallel versions of the program, utilizing the png and image task libraries, explained above.

Sequential Version

For the sequential version, the main goroutine reads in the json data and saves the data in to an ImageTask data structure by calling imagetask.CreateImageTask. See above for how CreateImageTask works. After we have the ImageTask, the program loops through the effects of the ImageTask and applies those to the ImageTask's Image's in image field. It does this by calling ApplyConvolution(effect string), which resides in the png package as described above, on the ImageTask's Image field if the effect is "E," "B," or "S." When this happens, ApplyConvolution(effect string) sends the appropriate convolution kernel and the Image to the Convolute method in editor.go. Convolute will apply the convolution to the in image field and save the filtered result to the out image field. Otherwise, Grayscale() in editor.go will be called if the "G" filter is given. Grayscale(), similarly, applies the grayscale filter to the in image and saves the filtered image to the out image field. If there is more than one effect to apply, the UpdateImg() method is called to replace the in image with the recently created out image so that the effects can be compounded. After all effects are applied, the out image is saved to the project directory. One image must complete all of these steps before another image can start.

Parallel Version

For the parallel version, first the main goroutine sets GOMAXPROCS equal to the numOfThreads variable and creates a few channels for use later in the program. tThe main goroutine then spawns a goroutine to read the json data from standard input, which creates an ImageTask for each task read and sends that ImageTask to an imageTasks channel for consumption by the worker goroutines. The main goroutine then spawns numOfThreads number of goroutines to act as workers to consume ImageTasks from the imageTasks channel as the ImageTasks come in. In a given worker goroutine, the goroutine grabs an ImageTask from the imageTasks channel and splits the image in to even chunks by calling the SplitImage function in the imagetask library (see above) based on y-pixels and numOfThreads, creating a slice of ImageTasks that are made up of the chunks of the original Image. Those chunks are then put in a chunkStream channel. These chunks in the chunkStream channel are then consumed by numOfThreads goroutines for each filter that needs to be applied. That is, each filtering goroutine grabs one of these image chunks, applies the appropriate filter using the same methodology as explained in the sequential version (just for the chunk instead of the entire image), and then puts the chunk back in the chunkStream channel for consumption by the next filter. Note that one filter must be applied to all image chunks of a given image before the next one can be applied because there is inherent data dependence in convolution where convoluting the edges of the image chunks requires pixels from the other image chunks. After all effects have been applied, the image is put back together using the NewImage and ReAddChunk methods in the png library (explained above). Once the image is put back together, the completely filtered image is written to the imageResults channel, which is consumed by another goroutine. That is, the main goroutine, after spawning the numOfThread workers, spawns another goroutine to read in filtered images from the imageResults channels and saves those images exactly as in the sequential version. Back in the worker function, the goroutine will check to see if there are more ImageTasks to consume from the imageTask channel and if there are not any more it will exit. If there are more ImageTasks, the worker will grab another ImageTask and execute all of the worker steps for the new ImageTask, again. The main goroutine waits for all workers to exit and all results to be read and saved from the imageResults channel before it exits.

- **Make sure to describe in detail, your functional and data decomposition approaches and why you choose them.**

Functional Decomposition

Functional decomposition occurs where code sections of the sequential version can be run independent of each other. The first functional decomposition occurs by running the generator function to read in image tasks from standard input and pushing those tasks to a channel for workers to consume. The standard input can be read independently of each other and do not require the knowledge of previous standard input to produce deterministic results, so this is an ideal place for functional decomposition. The second functional decomposition is spawning numOfThreads goroutines to act as workers to read in ImageTasks from the image tasks channel. Each goroutine grabs one ImageTask at a time to process but up to numOfThreads workers can be processing different ImageTasks

at once. Processing each image is completely independent of the others, so this is a perfect place for functional decomposition. Then, within each worker, numOfThreads goroutines are given a chunk of the image and do the filtering for the chunk. Each image chunk, for the most part, can be filtered independently of the other chunks which is why numOfThreads goroutines can be applying filters to all the chunks, at once. However, one filter must be applied to all chunks of an image before the next filter can be applied. This is because there is some data dependence between the chunks with the convolution algorithm. We wouldn't want the next filter applied to be working with "old" pixels, so we need to wait. This data dependence causes an implicit synchronization between the numOfThreads filtering goroutines for each filter. When all the filters are applied to all of the chunks and the image is put back together, the image is sent to an imageResults channel that is solely responsible for saving the completely filtered images. In this way, we can pass on parts of the problem to the next stage without having to wait for all stages to be complete, which is the exact definition of functional decomposition.

Data Decomposition

Data decomposition happens within each png image. I split up each image in to numOfThreads chunks by y-pixels. For instance, if there are 816 y-pixels and 2 threads, then the image is split in to two image chunks - one with y-pixels 0-407 and one with y-pixels 408-816. This is the BLOCK, * distribution of data to threads discussed in class. I actually give the chunks slightly more y-pixels than the even splits because convolution requires application of the kernel on the surrounding pixels. In the last row of an image chunk, for instance, if there were no buffer with a few more rows of pixels, then the convolution on the last row of pixels would be incorrect. We need multiple rows and not just one because convolution can have compounding effects that need multiple rows to get the correct final, filtered image. The data was decomposed this way because convolution of separate chunks of the image simultaneously creates load balancing that increases performance.

- **Instructions on how to run your testing script.**

Running Locally

Navigate to the "./hw6/proj2/editor" directory. **Put the unzipped proj2_files directory in the editor folder.** The python program copies the images from file1, file2, and file3 and the corresponding .txt files in the editor folder when the given problem size is being run.

Run the command "python3 timer-and-plot.py [numOfTrials]". The [numOfTrials] is an optional argument that allows you to specify how many times a timing event should be run. It defaults to 20 (as described in the project description) if not supplied. The reason for this optional argument is that the program was taking an eternity on my laptop to run 20 iterations of the file2 and file3 problems.

Inside of the python program, editor.go is being called with the specified number of threads for each problem. Each one of these iterations is executed 20 times (by default unless specified by the optional command line argument). Each time an iteration completes, a message prints such that the user knows where they are in the program execution. For instance, when the 18th iteration of problem size = file1 and threads = 2 completes, the message "DONE: problem: file1 , number of threads: 2 , iteration: 17" will print to the command line. The iterations are indexed to 0 which is why iteration 17 is actually iteration 18.

After all timing events have completed, the program will print a dataframe of the average runtimes for the parallel programs followed by an array of the average times for the sequential programs where the first array position is the average sequential file1 problem size time and the last index position is the average sequential file3 problem size time. Then a data frame of the speedups (calculated from the first dataframe and sequential time array) will be printed and a graph of those speedups will be saved to benchmark.png in the editor directory.

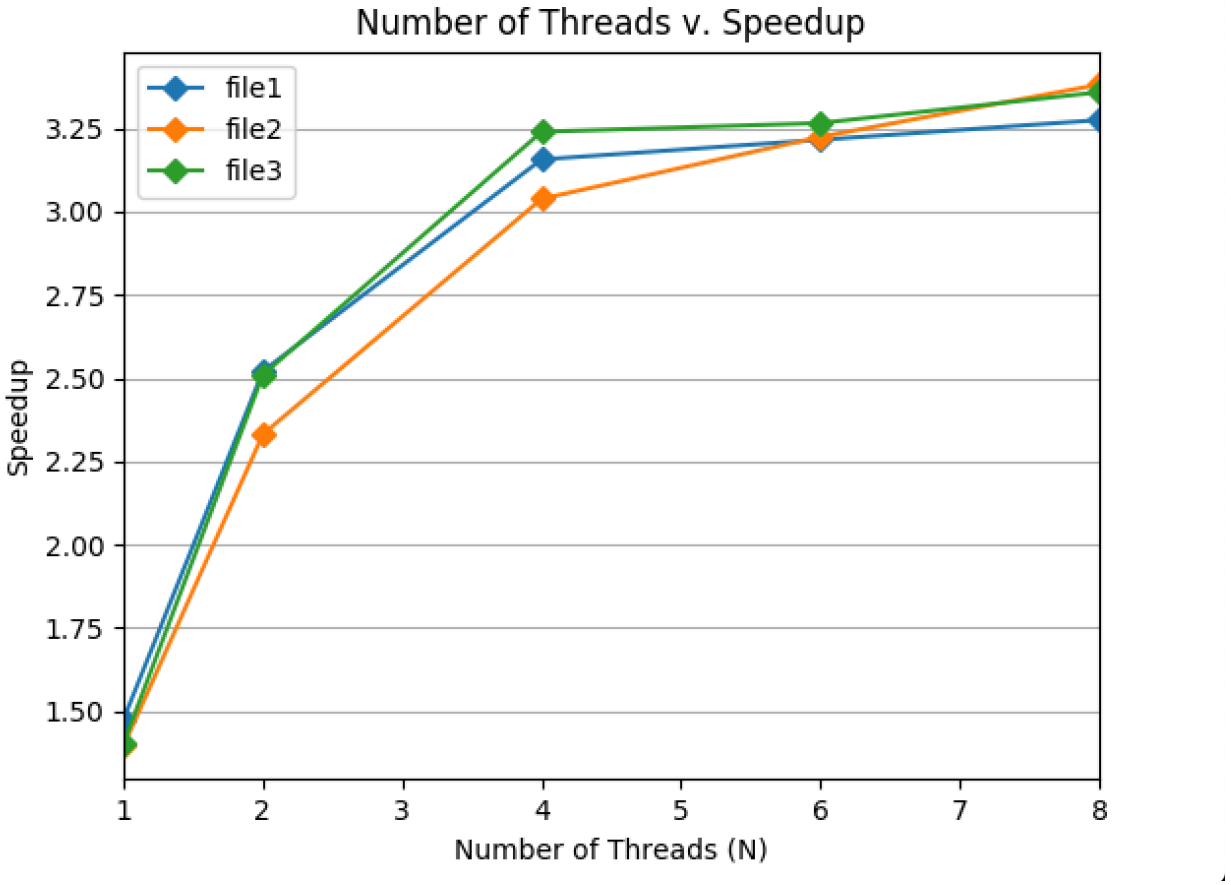
Running on the Cluster

Navigate to the "./hw6/proj2/editor" directory and run "sbatch proj2-slurm-job.job" with the proj2_files directory in the "./hw6/proj2/editor" directory. Running this command will automatically run the 20 iterations for each timing event. This should take about 7 hrs to run. The output graph (benchmark.png) will be located in "./hw6/proj2/editor" as before. The output will be the same as the locally run version except it will be saved to the "slurm/out" directory.

Note that you'll need to change the paths in "proj2-slurm-job.job" to your directory paths if you want this to work correctly on your machine. For instance, change "/home/erbaclaire/slurm" to "/home/YOUR CNET ID/slurm".

Because it takes so long to run, if you only want to run one iteration then go in to "proj2-slurm-job.job" and change the last line to "python3 timer-and-plot.py 1". You can change 1 to any number if you want more iterations.

- **Analysis of results.**



In this experiment, we had 3 problems of sizes file1, file2, and file3 with the same images in all of them but file1 < file2 < file3 in terms of image size. There were 5 thread amounts (goroutines spawned): 1, 2, 4, 6, and 8. For each problem size, the editor.go program was run such that the number of threads is equal to 1, 2, 4, 6, or 8. There are 15 such timing operations (3 problem sizes * 5 thread amounts = 15 timing operations). The sequential version of the program is run for each problem size (3 problem sizes = 3 timing operations), as well. Speedup is calculated as the sequential version time / parallel version time. So, for instance, if the file2 images took 1, 0.9, 0.6, 0.4, and 0.3 seconds to run for 1, 2, 4, 6, and 8 thread amounts, respectively, and the sequential version took 1.5 second to run, then the plotted speedup would be (1, 1.5), (2, 1.67), (4, 2.5), (6, 3.75), (8, 5) for the file2 problem size line on the output speedup graph. For each problem size and number of threads, editor.go is timed 20 times and the average of those 20 times is plotted in order to avoid misleading timings caused from caching of processes on the first execution, other applications running, or other random interrupts. Therefore, overall, 360 timing results are calculated ((15 parallel timing events + 3 sequential timing events) * 20 iterations for each event = 360 calls of editor.go).

The above graph is the result of running 20 iterations of each timing event on the general cluster.

As can be seen, we get speedups in the range of 1.5-3.4 depending on number of threads and file sizes. Note that for all of the problem sizes, the 1 thread version does slightly better than the sequential version. This is because even though there is only one thread, which means the convolution is still being applied to the entire image all at once instead of chunks of the image (i.e. no real data decomposition), the program still benefits from the functional decomposition of having the reading in of tasks, processing of images, and saving of images being done asynchronously from one another.

The speedup graph indicates that about 80% of my program is parallelized, based on Amhdal's Law. This makes sense when examining the code. There is significant parallelization with the functional and data decomposition explained earlier in the report, which allows for tasks to run in parallel. This can be observed readily by the fact that when there are more threads (i.e. there is even more data decomposition and load balancing by having smaller chunks of images spread out over more cores to lessen the hotspot), there is generally more speedup for all problem sizes. However, there are necessary points in the code that are sequential. For instance, there is only one channel saving the completed ImageTasks, which means only one ImageTask can be saved at a time and all other ready ImageTasks must wait. Additionally, all of the image chunks of a given image need to have the first filter applied before any of the image chunks can go on to have the next filter applied, due to the semi-dependent nature of the convolution algorithm. Most notably sequential in the program is that the program uses channels to coordinate between the functionally decomposed portions of the code. For instance, the generator goroutine sends ImageTasks read in from standard input to the imageTasks channel, which are read by the workers that split up the ImageTask's image in to chunks and sends those to an imageChunks channel within the worker for processing. The image chunks are put back together after all filtering and then sent to the imageResults channel for saving. Channels are communicating sequential processes (CSP), which is a method for describing patterns of interaction in concurrent systems. The program specifically uses unbuffered channels which perform synchronous communication between goroutines. These unbuffered channels guarantee that an exchange between two goroutines is performed at the instant the send and receive takes place. If the goroutines are not ready at the same instant, the channel makes the goroutine wait (i.e. synchronization). As explained below, I use coarse grained granularity so these moments of synchronous communication are spread out. That is, many calculations are being performed before the synchronization occurs between the channels, which allows for more load balancing, opposing the sequential effect.

Finally, it is interesting to note how the problem sizes compare with each other. First, file1 has slightly worse speedup than file3 for most of the thread amounts. File2 then has slightly worse speedup than file1 until 6 threads where it becomes even and then 8 threads where it becomes even with file3. I explore why this is in the answer to the last question in this report. Please reference that for details.

- **Answers to the following questions:**

  - **What are the hotspots and bottlenecks in your sequential program?**

Hotspots are the areas within the program that are doing the most work. In my sequential program most of the work is happening during the convolution stage. Each convolution of an image has to convolute each pixel in the image. Convoluting the entire image involves ranging over all (x, y) pixels and for each one of those pixels doing a total of 9 computations (multiplying the 3x3 kernel spaces by the corresponding pixels in the image). This is where the CPU is doing the most work in the sequential program.

Bottlenecks are areas of a program that cause a slow down in performance. In the sequential program, the bottleneck is happening where each image has to be read in, filtered, and then saved before another image can go through that same process. Therefore, each time an image is processed all other images need to wait and make no progress. Reading from standard input and the saving the image take many cycles to complete. So on top of waiting for the large computations implemented in the convolution, there is waiting that occurs by the other images when an image is read in and saved.

  - **Were you able to parallelize the hotspots and/or remove the bottlenecks in the parallel version?**

I was able to parallelize the hotspots and partially remove the bottlenecks in the parallel version. The filtering with the convolution algorithm on the pixels is now spread out over multiple cores instead of just one core processing all pixels, sequentially. There is still some slowness and bottlenecks from saving the images. That is, because there was only one go channel to save images, only one could be saved at a time. Especially for the larger images, this produced a slow down. Additionally, originally I would have liked to just send image chunks directly to the next

filter as soon as they were done. Because of the convolution algorithm, however, I had to make sure all chunks of a given image went through one filter before passing in to another. This caused a bottleneck with chunks waiting for other chunks in the image to finish before they could move on to the next filter all together.

- **Describe the granularity in your implementation. Are you using a coarse-grain or fine-grain granularity? Explain.**

For each worker, overall, I implement coarse grained granularity. There are long periods of computation between synchronization/communication events. For each worker, an image is completely split up, filtered through all filters, and put back together before the final image is sent to the imageResults channel for saving. All computations for a given image are happening within the worker and it is not until the image is completely done that the communication between channels (i.e. fan in) is implemented. For the filtering within the workers, however, this is more fine-grained granularity. All image chunks in an image need to be put through one filter before going on to the next because of the data dependencies described above.

- **Does the image size being processed have any effect on the performance? For example, processing csv files with the same number of images but one file has very large image sizes, whereas the other has smaller image sizes.**

In both the parallel and the sequential versions, larger images should take longer to process than smaller images because the amount of convolution calculations that need to be performed on the larger images (because they have more pixels) is greater.

In theory, larger image sizes would benefit more from parallelization of the program because larger image sizes have more pronounced hotspots in their sequential versions. That is, the weight of computation being put on one core is more exaggerated. For instance, an image with 120 pixels and an image with 100,000 pixels in their sequential versions both have all the computation put on one core. However, the 100,000 pixel file has many more computations to do. Imagine a second thread was introduced to process each image. Then two cores could each work on 50,000 pixels for the 100,00 pixel image while the 120 pixel image would have two cores processing 60 pixels each. The magnitude of relief on the 100,000 pixel job is much greater than the 120 pixel job. Therefore, we would expect to see the 100,000 2-thread version of the program to process much quicker than its sequential version. In the 120 pixel job it is not clear that the 2-thread version would be much better than the sequential because the pixel amounts are so negligible. Therefore, it is likely the speedup for the 100,000 pixel job would be more pronounced than the 120 pixel job.

In our implementation, the size file3 > the size of file2 > the size of file1. Or that is what we were told. If you look at the actual files in the different folders, only some of the files are larger, but not all of them. For instance, in file2 and file3, IMG_4061 is the same size (i.e. has the same number of pixels). Not every picture in file1 is smaller than its file2 counterpart and same for file2 compared to file3. Additionally, the pixel amount differences that do occur between the files are on the magnitude of 4-6, which is not that larger.

In my implementation, the larger file size (i.e., file3) has only slightly better speedups than smaller file sizes (file1 and file2). We see higher speedups for file3 and this is because the abundance of pixels in the larger file sizes benefits more from the elimination of the hotspot (i.e. the convolution on each pixel). That is, when having to wait for each pixel in a full, large image, the sequential bottleneck is more pronounced because there are so many pixels waiting. When load balancing and convoluting equal chunks of the image, however, this bottleneck is somewhat removed and has a greater impact than the removal of the bottleneck in the smaller image sizes. The actual difference between the file1 and file3 speedup is not very pronounced, though, and this is because the actual number of pixel difference in not absurdly large. Therefore, it is not surprising that the speedups are relatively close.

File2 does not follow this pattern. The images in file2 have more pixels than those in file1 (in theory) but have worse speedups than file1 until 6 threads, where the speedup becomes even with file1, and then 8 threads where the speedup of file2 does slightly better than that of file3 even though file3 has larger images than file2. These speedup differences, to be fair, are small (< 0.1) so they could just be seen as all three problem sizes having about the same speedups. This may be explained, again, by the fact that the actual difference in size between file1 and file2 is not extremely pronounced, so it makes sense that each file size has about the same “relief” from the sequential bottleneck and hotspots as the others when parallelizing.