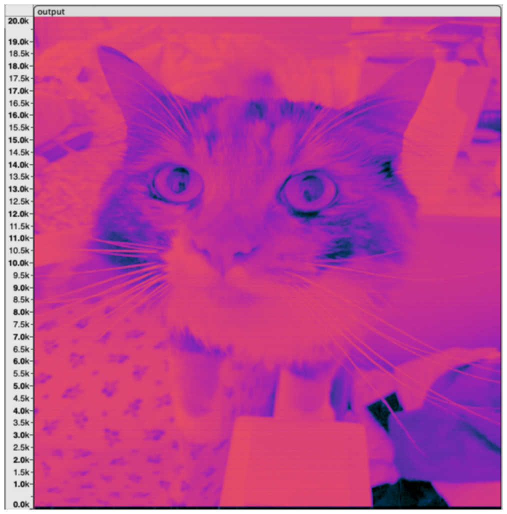
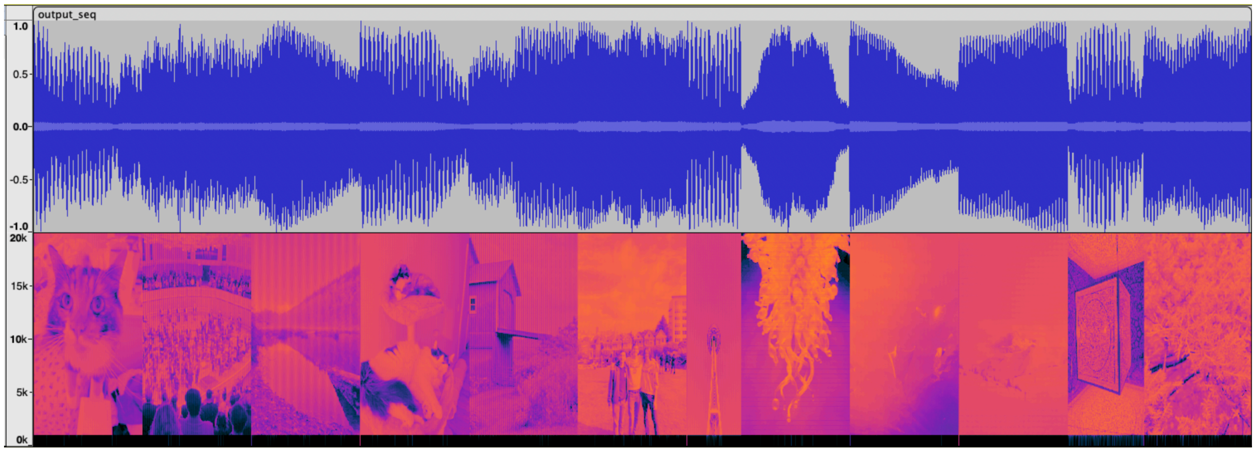
Project 3 Write Up
Cory Turnbaugh



A single image as a spectrogram (frequency on Y axis, time on X axis, color saturation indicates amplitude of that frequency at that time, brighter is greater amplitude). The output can look pretty good with the right parameters (though will always sound terrible).



A series of 12 images in a single waveform (expected output of large.json input) as both a waveform (top, amplitude on Y axis, time on X axis) and spectrogram (bottom).

Problem

Preface:

My project takes in png images as input and exports an audio file where the spectrogram of that file (a kind of 3d graph representation of audio with time on x axis, frequency on y axis, amplitude as color intensity) looks like the input images. Originally, my project was going to be parallelizing the process of converting a single image to a spectrogram, but I realized this did not mesh well with the project criteria. Without going into too much detail, the largest problem was there wasn't any performance improvement with the dequeue as each task (a single row of pixels) was the same amount of work. I therefore adjusted my original problem to a slightly more contrived one: drawing multiple images to the same audio file to produce a panoramic view. Rather than parallelizing each row of a single image, I parallelized the processing of multiple different. I can't quite think of a use case for this, but honestly my original idea wasn't too practical either.

Project Description:

A JSON file input provides the program with information on what images to use and parameters for the output:

```
[
   {
      "ImgPath": "images/img1.png",
      "Duration": 20,
      "SampleRate": 44100,
      "MinFreq": 100,
      "MaxFreq": 20000,
      "Height": 900,
      "NumTones": 2,
      "Contrast": 1
   },
   …
]
```

Duration specifies the length in seconds that image should cover in the audio file, the sample rate is the sample rate of the output audio file (my program only really supports 44100hz at the moment), min and max freq specifies the frequency range the image should cover in the output, height scales the image to a certain number of pixels, which is helpful particularly for large images, numTones specifies how many different frequencies should be used for a single pixel row (more often than not more tones introduces strange phase issues that reduces the image clarity, but this is very good for testing as for a value *t* = numTones, the runtime is multiplied by *t*), and contrast increases the difference between the darkest and brightest pixels which is sometimes helpful for image clarity but does not impact runtime.

The order of these images in the JSON file is important as it determines the order in which the images will appear in the output audio file.

Duration, Frequency, Height, and NumTones are the parameters that I manipulate to greatly influence how long processing a given image takes a thread.

Each image is scaled and converted to grayscale. For every row of the image, the y value of that row is mapped to a frequency based on the input parameters and the intensity of each pixel is used to get the amplitude of the wave at that point. Each pixel is mapped to a certain number of samples determined by the desired duration, the sample rate, and how many pixels are in the resulting scaled image. Each sample of each row is added together and normalized, then the program repeats for all images, concatenating their audio buffer results then exporting the final file.

I use the free software Audacity to view the audio files as a spectrogram and I did not think at the time about how this might be annoying to grade. There are a few tools online that allow for viewing spectrograms such as this one (https://www.dcode.fr/spectral-analysis). If you use this site or one like it I would suggest unchecking "Logarithmic Scale" and also turning down your volume.

Execution

Navigate to the src/testing directory and execute

```
./benchmark.sh
```

to run all tests on slurm and generate graphs. This will use the example JSON files I created (small.json, mix.json, large.json). By default this will run all tests 3 times and get the average. If you want to save time and run fewer tests, adjust the NUM_RUNS global constant in the benchmark.py file.

You can also create your own input by writing a custom JSON file following parameters specified above and running (in the same testing directory):

```
go run testingio.go FILE_PATH.json seq
```

OR

```
go run testingio.go FILE_PATH.json map NUM_THREADS
```

OR

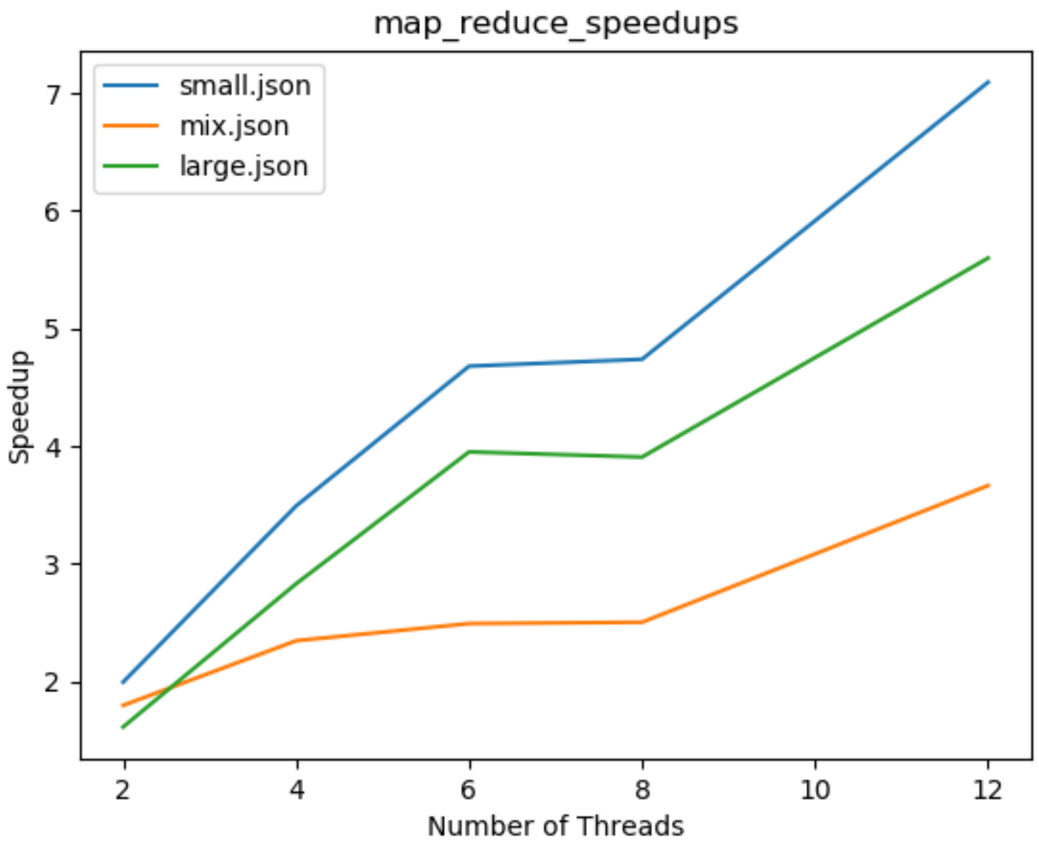
```
go run testingio.go FILE_PATH.json steal NUM_THREADS
```

where NUM_THREADS is an integer.

Parallel Implementations

(Graphs produced by calculating speedup relative to sequential implementation. Input sets include “small.json”, a series of 12 images scaled to be low pixel densities, low duration, and only ever 1 frequency per pixel; “large.json” consists of the same 12 images but much higher pixel density, longer durations, and 1-2 frequencies per pixel; “mix.json” is a combination of these inputs with a few more small images than large to ensure we would see work stealing occur, still 12 images total. Each datapoint for the graphs is the average of 3 runs. I had initially attempted 5 but was consistently timing out)

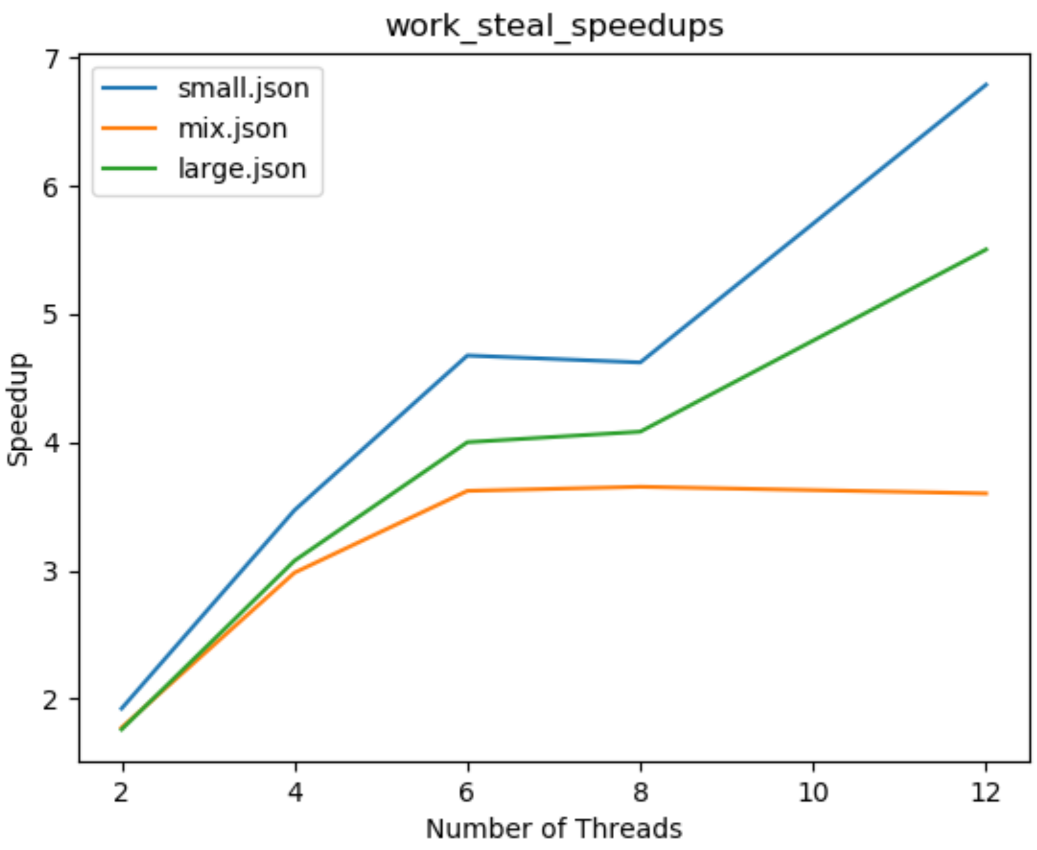
1)



For the first parallel implementation I used a map-reduce paradigm, as this fit quite naturally with the problem: if the input was *N* images, it made sense for each of the *T* threads to process approximately *N/T* images—the mapping phase.

To produce a single output audio file, the audio data for each group of images processed by each of the threads (essentially a large slice of integer values where each index is a single sample (44100 samples per second) and the value of that index is the amplitude of the sound wave) needed to be combined into a single slice with length the sum of all audio data produced by each thread. Also, since I had decided I wanted the images to retain their JSON input order, I needed each thread to wait until others were finished to ensure each thread would insert its data in the correct slice position, given the final duration of images produced by each thread (If we know the duration from the input, why can't we just place the data without waiting for other threads to finish? Because the final number of samples is unlikely to match the input duration exactly due to rounding, and since we are dealing with exact slice indices, we do need the exact values).

2)



The work stealing implementation similarly splits images amongst the threads to be processed in parallel, with the key difference that each thread has its own lock-free double-ended queue where it first pushes each task to the bottom when initially spawned, then pops from the bottom as it completes tasks. If any thread finishes processing all its assigned images, it attempts to steal from other threads' dequeues by popping from the top of their queue. If this fails, the thread exits immediately or, if it is the last thread to finish, it waits if other threads are still running, or if it is the last thread to finish it creates a final audio buffer and calculates where each image should go in this buffer then broadcasts all threads to wake. The threads then rush to add all images to the final buffer, incrementing an atomic integer representing the index of the image that thread claims, and the last thread to finish that then exports the audio to disk.

The work stealing implementation shines when using a series of input images with vastly different workloads. For example, certain images in "mix.json" take only a few (~5) seconds to run while others can take 30-60 seconds. I didn't try to bias the tests like this, but if it were the case that the first few images all took 60+ seconds and the rest took only 5 seconds, the 1st parallel approach would always be considerably bottlenecked by the first thread launched, while this work stealing implementation would be able to balance the load by distributing these larger files among the idle threads.

Looking at mix.json data on the graphs confirms this intuition. Parallel implementation 1) is clearly bottlenecked for 2-8 threads, and it is only at 10-12 threads where speedup resumes because since I tested with only 12 images, each thread is taking on average only 1 image. If I added substantially more images, we could expect the bottleneck to continue. Parallel implementation 2) does bottleneck at 2-6 threads (in the same way) as a single or few threads are no longer stuck processing all of the large images in the set. We do see a bottleneck from 8-12 threads, but this is because we reach the minimum runtime in which a single thread can process a large image. I talk about this bottleneck and ideas for how to address it in the **Hotspots and Bottlenecks** section below.

For the small.json and large.json input sets, however, where each image consists of approximately the same amount of work, we don't get the work stealing advantage and actually see less speedup on average than parallel implementation 1). The additional overhead of the dequeue is to blame for this, as rather than simply iterating through its own local slice, each thread must push and pop to a more complex thread-safe data structure, suffering the runtime cost of atomic operations.

Hotspots and Bottlenecks

Bottlenecks

Granularity: If I had more time to work on this project, I would want to try further decomposing each task. As we can see in the speedup graphs, once we reach 6 threads, the speedup for the 2nd parallel implementation plateaus. While the work stealing has done what it can to distribute the work, the program must still wait for these large images to be processed, while any thread that has finished its work and has nothing to steal closes. As discussed in the preface on the first page, processing each image is an embarrassingly parallel operation, though as I was first-and-foremost focused on satisfying the project requirements, I did not have time to further divide the problem. Doing so may introduce more overhead, but datasets such as mix.json would likely see continued speedup beyond 6 threads.

IO: Writing the wave file to disk is bottlenecked as a single thread must make the final system call.

For both parallel implementations, there is a certain amount of parallel work required to determine where in the final slice each image should go to maintain the proper ordering. This involves summing the lengths of all audio buffers produced by all threads.

Hotspots

Granularity: The bottleneck mentioned above is a great opportunity for further parallelizing the problem and taking advantage of its embarrassingly parallel nature.

Reading multiple images: Since this program can work with any number of images, it made a lot of sense to split the images among the threads rather than process each sequentially.

Reducing: A key step of both implementations is taking the work performed individually by threads and merging them into a single audio buffer. This is another place where I take advantage of parallel processing. Since an audio buffer consists of hundreds of thousands of integer values, I could worry less about cache invalidation.

Challenges

Maintaining the image order was a challenge because the two parallel implementations required different considerations. While parallel implementation 1) ensured that, at least within a single thread, images would be in order, and as long as I appended the audio data in the same order as threads were launched, the final output would also be in order. With parallel implementation 2, suddenly all bets were off. Work stealing made it such that images could be randomly distributed across threads, and I needed to add an additional orderNum parameter and calculate for each image (rather than for each thread's block of images) what sample it should appear at in the final slice.