Parallelizing a Facial-blurring Program

David Bai

Abstract

My original idea for the project was simple: given some kind of video, anonymize any people featured in the film by blurring their face and distorting their voice. The voice-distortion feature was never implemented, but face-blurring was through the use of frame data retrieval/video processing, a facial recognizer, a blurring filter, and more video processing to create the final image. For parallelization purposes, converting the video to frames and getting the frames' filenames were all part of pre-processing, while the stitching of (potentially-blurred) images was post-processing. The parallel work laid in going through the extracted frames and detecting faces in there. Given the post-pre-processing data organization, i.e. in discrete files, a parallel-for was implemented using both Cilk and OpenMP to scan multiple files for faces at a time. Given the lengthy run-time of the serial version (Over 9 minutes to process a <30 second clip at 24fps), the speed-up exhibited by both parallel versions translated to a noticeable/significant performance increase. Based on timing tests for the same clip at 2, 12, and 24 frames per second, the speed-up does seem to increase as the problem size increases which illustrates the idea of weak scaling. This seems to plateau however at around 3x speed up.

Introduction

Video processing is not an easy process to do quickly. A single image can be most simply represented as a two-dimensional array of pixels/three-ples representing pixels, but the amount of data needed quickly blows up as framerate and resolution do too. Even ignoring the audio portion of the video, at 24fps a 30 second video becomes 720 images. If the frames are high quality images, they can easily be over a megabyte each, for a total of something in the order of a full gigabyte. With video compression and encoding techniques this is somewhat ameliorated but videos are still taxing for even modern hardware to process, unlike just audio or static image input.

The ‘difficulty’ of video processing means there is some room for increased performance via parallelization, so I wanted to do something in that problem space. More importantly, I wanted to do something at least nominally useful, so I eventually decided to make an anonymizer. With an ever-increasing amount of internet-connected devices in the world, concerns about security and privacy are rising as well. This in mind, I wanted to write a program that would take an ordinary .mp4 file as input and output that same file with a key difference – blurring faces past recognition. What I ended up coding is not perfectly anonymizing, but it does serve as a good proof-of-concept of the grander scheme.

Design

The first part of the program design was the blurring function. I used the blurring filter from class as a starting point but I went on to make it first indiscriminately copy the image, then only blur a specified region. This way, I can blur just faces and not the entire image. Later on I added a code-snippet (and corresponding global variable) to highlight the blurred region. This is more for debugging and demonstration purposes as the larger goal is not to make it clear where the faces are but to blur them instead.

Pre-processing involved using FFMPEG to convert the input .mp4 file to a series of .png images. The number of total frames was determined by the framerate. To generate different sized inputs, I used two, twelve, and twenty-four frames per second options for the same approx. twenty-one second video. Also part of pre-processing was creating an array of the image filenames. This is done in two iterations of the directory – the first one simply counts the number of .png files to initialize an array of the right size and the second one actually populates the array. Also initialized in an array of bools (ints in C), one for each frame, which represent whether a face was found or not.

The main work/the parallel region/timed region was a for-loop that iterated through the array of filenames. Each iteration read in an image from the filename array and checked if it was a valid image or not. Then the read image was used to create an array of pixel structs which was copied into a second array to represent the output pixels. The face detector object took the image and

a declared vector of Rect objects and did the actual detection. If at least one face was found, then the corresponding bool in the array of bools was set to 1 and afore-mentioned the blur function was called for each face to blur that particular one. Then the pixel arrays were used to populate a Mat object so that openCV could write the image back out.

The parallelization that was implemented was initially a simple matter of making the for-loop iterating through the filenames array a parallel for-loop. For a while, however, there was a somewhat hidden race condition that caused the program to occasionally crash for large enough inputs (AKA high frame-rate extraction) or generally be inconsistent in output videos. This was somewhat hidden as FFMPEG conversion to video does not seem to be completely consistent/deterministic, so diff was not very helpful. Some difference could be seen in the output videos, but the videos themselves seemed OK if a little incorrect. It turns out that the race condition came from the CascadeClassifier object declared right outside of the for-loop for the serial version. In the parallel versions, multiple threads can attempt to access the classifier at the same time resulting in wonky facial blurring that is all over the place. By moving the object declaration inside the for-loop for the parallelized versions, this problem was eliminated though at the expense of increased overhead from each thread instantiating a copy of the same classifier. I believe the program is fully functional in all versions, though one possible issue is whether or not the openCV library functions are thread-safe. If not, there could be still more hidden race conditions.

Post-processing was simply using FFMPEG to turn the (potentially-blurred) frame data back into a movie. The output was much lower quality due to conversion to and from frame data, which contributes to the less than 100% accuracy of the blurring.

Performance Analysis

Performance was measured in terms of real run-time as it is both simple to measure and highly useful/relevant information. The timer starts right before and ends right after the for-loop region that goes through the array of filenames. I took four trials of each implementation (serial,

openMP, Cilk) for each input size (using two, twelve, and twenty-four frames/sec) for a total of 36 recorded runs of the program.

2 fps	1	2	3	4	Avg
Ser	47.82	47.878	47.825	47.907	47.8575
OMP	17.736	16.689	16.236	17.252	16.97825
Cilk	16.556	16.456	17.166	16.983	16.79025

12 fps	1	2	3	4	Avg
Ser	285.485	285.477	285.975	273.526	282.6158
OMP	92.391	93.584	93.789	92.819	93.14575
Cilk	92.029	91.839	92.847	92.175	92.2225

24 fps	1	2	3	4	Avg
Ser	569.295	568.491	568.779	540.591	561.789
OMP	184.247	180.821	179.392	183.303	181.9408
Cilk	183.561	184.12	182.293	184.841	183.7038

Fig 1: Recorded runs (time in seconds)

The times are fairly consistent, except for trial #4 for the serial implementation from 12 and 24 frame/sec generated data. That may be as trial #4 was ran on a different day so different amount of people were sharing the cluster. That being said, the parallel implementations did not really change, or at least not in the same direction. Perhaps parallelizing the work to multiple threads makes the runtime more consistent in the face of minor perturbations.

	2	12	24
Serial Avg	47.8575	282.6158	561.789
OMP Avg	16.97825	93.14575	181.9408
Cilk Avg	16.79025	92.2225	183.7038
Speedup	2.834446	3.049236	3.07287

Fig 2: Speedups

Though there are not many points of data, the three different problem sizes suggest that the speedup does indeed scale with problem size but also plateaus after not too long. That the speedup is so low implies that there is a lot of overhead and/or some strangled scaling. These may come from the library functions like imwrite() which could be a bottleneck if multiple

images cannot be written/saved to memory at once. Still 3x faster is not insignificant, especially with such long run-times.

Conclusion

From working on this project for the last week or so, I can say that parallelization of a program does not have to be obscenely difficult while still giving good performance increase. Though quite a ways from processing a frame quickly enough to anonymize a stream in real-time, being able to process three times amount of video seems like a lot to me. In terms of what features I would like to add in the future, the audio part would be interesting. For this I would have to chunk up the audio track so that in the same frames where a face was detected and blurred, the audio would be distorted. This could have problems if a face is detected in, say, every other frame as the audio might be choppy. To address this, perhaps if a face is detected in a frame then the audio is distorted for a half-second regardless of whether a face is detected in that period or not. Given more time, it'd be interesting to see how performance scales with even larger input and if it does indeed plateau or if it continues to grow. Lastly, it'd be interesting to do a sort of stencil/lattice parallelization pattern regarding the actual face-detection. Perhaps instead of parallelizing the for-loop that deals with the filenames array, multiple threads could look at (overlapping) smaller subsections of a single frame. This could have different performance due to different locality.