Parallel Video Processing

Neal Wadhwa

December 17, 2012

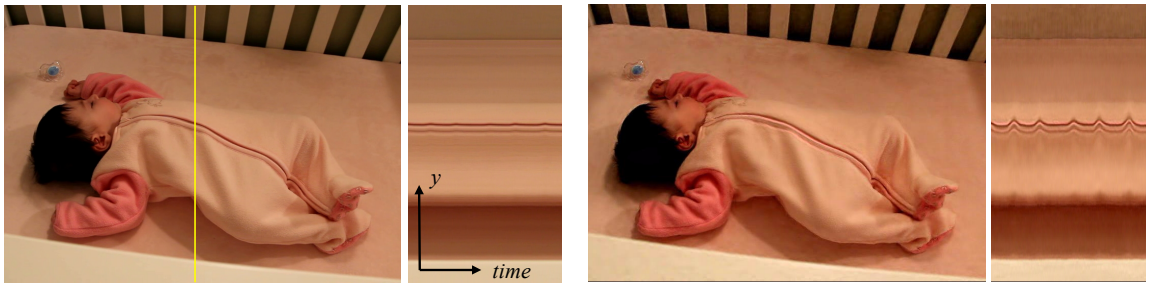


Figure 1: A video processing algorithm that may be easy to parallelize. Motion magnification of a baby breathing. (a) A frame from the input sequence (*baby*), with a spatiotemporal YT slice taken at the location marked yellow. (b) our result with spatiotemporal slices as in (a), and a zoom-in on a patch around the baby's chest in the insets.

1 Parallel Video Processing

Uncompressed modern videos are often shot at high frame rates and high spatial resolutions. A single full HD video contains over 2.5 Gigabytes of information per second. A video processing algorithm might convert the video into a transform domain in which different spatial frequency bands are separated out. Depending on the representation, the resulting transformed video signal might have 10 GB to 1.5 TB of information per second. While processing 1.5TB of information per second may be impractical in practice, it is still useful in a research setting.

Many video algorithms are easy to parallelize because the computation occurs locally in space or time. I will try to parallelize a video processing algorithm called motion magnification. The processing occurs in three stages. Each stage is easy to parallelize, but it is unclear how to store the data to parallelize all three stages.

1. Transform each frame using a steerable pyramid. Each frame could be stored on a different processor.

2. Apply temporal filtering to each pixel through time in the steerable pyramid representation. Each pixel through time could be stored on a different processor. The temporal processing requires all time values for each pixel.

3. Inverse transform the resulting frames. Each frame could be stored on a different processor.

We will refer to these three stages in the results section.

In this project, I ported Matlab code that implemented this algorithm to Julia. I compared their performance serially and in parallel. I also attempted to parallelize this algorithm in both Matlab and Julia. I used Matlab's parfor function and compared it to hand tuning the parallelization using Julia's parallel computing functionality.

2 Brief Description of Motion Magnification Algorithm

The heart of the motion magnification algorithm lies in the steerable pyramid representation. After each frame is turned into a steerable pyramid, the phases are temporally filtered and magnified. Each frame's representation is then inverse transformed.

2.1 Steerable Pyramid

Steerable pyramids are overcomplete representations of images in which information about different frequency bands of an image are separated out. Both scale and orientation are relegated to different bands. The steerable pyramid is filter bank in which each filter corresponds to a different scale and orientation. Each filter "masks" out the frequency and orientation of interest. Furthermore, the sum of the squares of the filters is equal to unity, which means that the bands can be added together to reconstruct the original image.

If the filters are specified by $\Psi_{\omega,\theta}$ where ω corresponds to scale and θ corresponds to frequency and the image is specified by I, then the steerable pyramid is given by

$$S_{\omega,\theta} = \mathcal{F}^{-1}\{\mathcal{F}\{I\} \times \Psi_{\omega,\theta}\} \tag{1}$$

The image can be reconstructed by multiplying each band by its corresponding filter and then summing. That is,

$$\tilde{I}_R = \sum S_{\omega,\theta}\Psi_{\omega,\theta} = \sum \tilde{I}\Psi^2_{\omega,\theta} \tag{2}$$

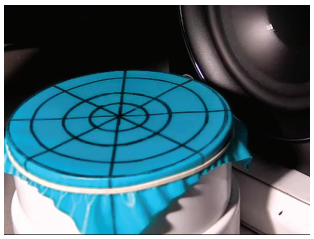
where the sum is over all the scales and orientations.

It is possible to compute a Steerable pyramid corresponding to cos and one corresponding to sin. By combining them, it is possible to come up with a complex steerable pyramid representation. The phase of a such a pyramid corresponds to motion and it is that we are manipulating temporally.

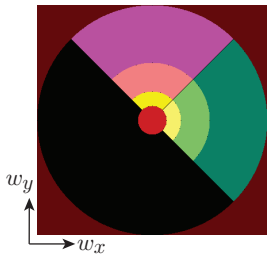
For each frame, a steerable pyramid is computed.

2.2 Temporal Processing

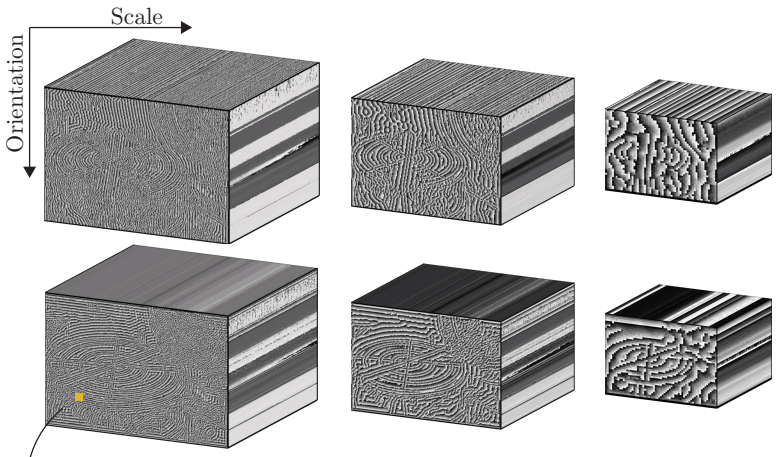
For each pixel in time, the phases are bandpassed and then multiplied by an amplification factor α. This amplified phase signal is then added to the original signal. Then the video is collapsed and reconstructed. The process is described in the following figure.



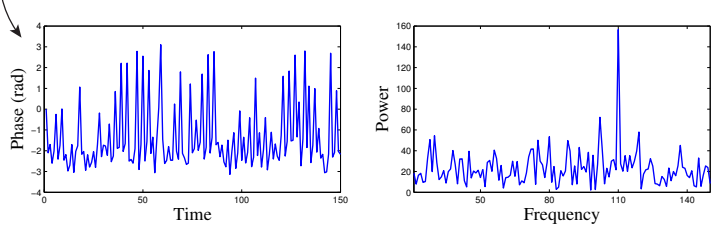
(a) Input frame (*membrane*)



(b) Ideal transfer functions



(c) Phase videos for different scales and orientations in (b)



(d) Phase data

Figure 2: Our approach manipulates phases in the video to modulate motions. (a) A frame from a video showing a membrane oscillating to 110Hz sound wave. (b) Ideal transfer functions of different filters in a complex steerable pyramid with three scales and two orientation bands, drawn on the spatial frequency plane (high and low pass residuals in red; black pixels are not covered by the transfer functions). (c) Phase videos corresponding to the different scales and orientations in (b). (d) Observing phase over time for a single location, scale and orientation (left; marked on the phase video in (c)), reveals motions at the frequency of interest (right; 110Hz). By temporally filtering and amplifying the phase data, we magnify the motions in a required frequency range.

3 Results

I ran several tests comparing different tweaks of the algorithm. All the tests were run on julia.mit.edu. I restricted myself to 12 cores or less because Matlab has a limitation in which it cannot run in parallel on

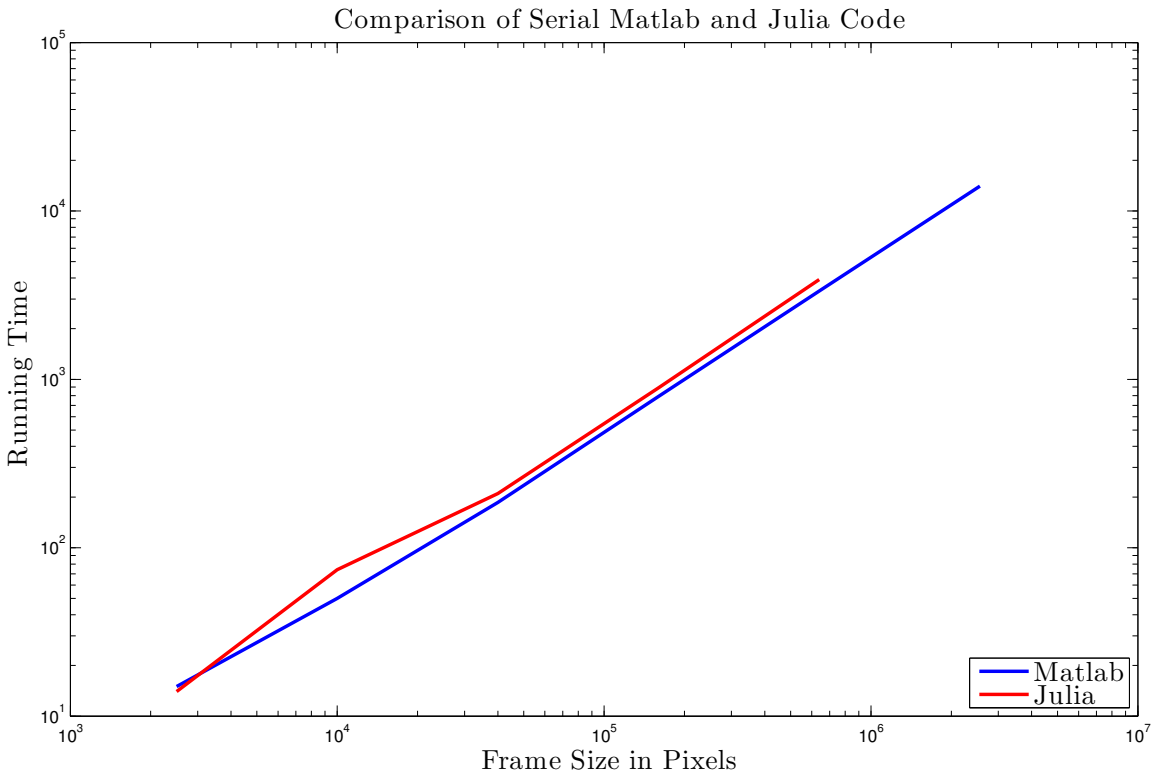


Figure 3: Comparison of Julia and Matlab on a single core. The test was run on julia.mit.edu. The speeds are about the same. Julia is roughly 10% slower.

more than 12 cores without a Distributed Computing Server.

As a first baseline, I first ported the code from Matlab to Julia. The resulting code is slightly slower in Julia, but it is very close. It is not surprising that the languages are so close since most of the processing is FFTs which are done via the libfftw library. See figure 3.

As a second baseline, I used Matlab's parfor to parallelize the first and third stages of the algorithm. I used 12 cores on the machine julia.mit.edu. Surprisingly, using parfor only gives a factor of two performance boost on this machine despite there being 12 times as much computing power. Also surprising is that this boost is roughly constant for all problem sizes. See figure 4.

To test out Julia's parallel processing, I constructed a distributed array to store the steerable pyramids of each frame. Frames that were near each other time were store on the same processor. I then used Julia's spawnat to construct the pyramid for each frame in the first stage of the algorithm. This was very fast and I got a more than 12 times speed up (17 times), which was very surprising. See table 1 and figure 5.

However, for the temporal processing in the second stage, I had to retrieve data from each of these processors and this resulted in massive slowdowns. This ate away at almost all the gains in the previous stage. As a result, the algorithm was about twice as slow as serial code for small problem sizes and slightly faster for large problem size. My version of Julia (11-14-2012) would segfault when I tried to create an array of size 1600x1600x3x300. As a result I wasn't able to test very large problem sizes. I wonder if this code would have gotten substantially faster than serial code.

Since a large portion of the algorithm's running time was spent in the second stage due to the fact that

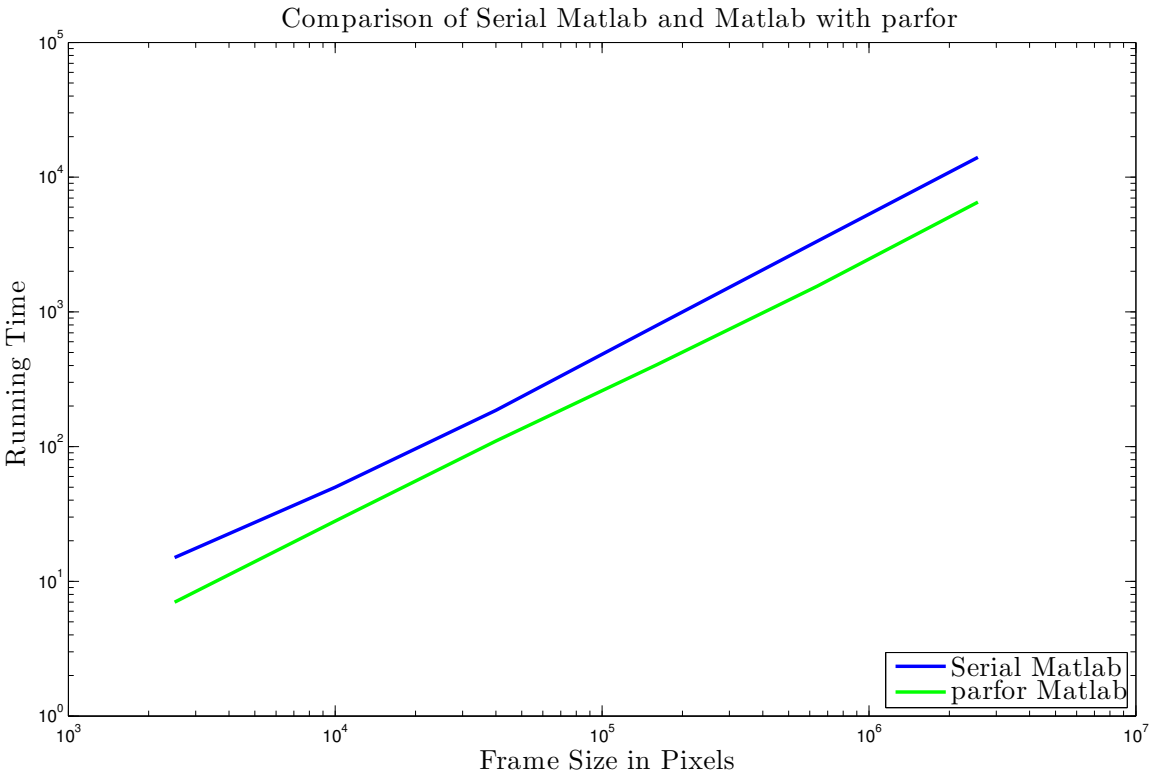


Figure 4: Comparison of Serial and Parallel Matlab. The test was run on 12 cores on julia.mit.edu. The speedup is roughly two times.

Serial code running times for each stage					
Problem Size	50x50	100x100	200x200	400x400	800x800
Stage 1 Time:	5	23	65	305	1343
Stage 2 Time:	2	13	22	174	561
Stage 3 Time:	7	38	123	412	2010
Parallel code running times for each stage					
Problem Size	50x50	100x100	200x200	400x400	800x800
Stage 1 Time:	0.3	1	3.5	14	56
Stage 2 Time:	26	47	125	436	1828
Stage 3 Time:	21	35	108	831	3611

Table 1: Running times for each stage for parallel vs. serial Julia code

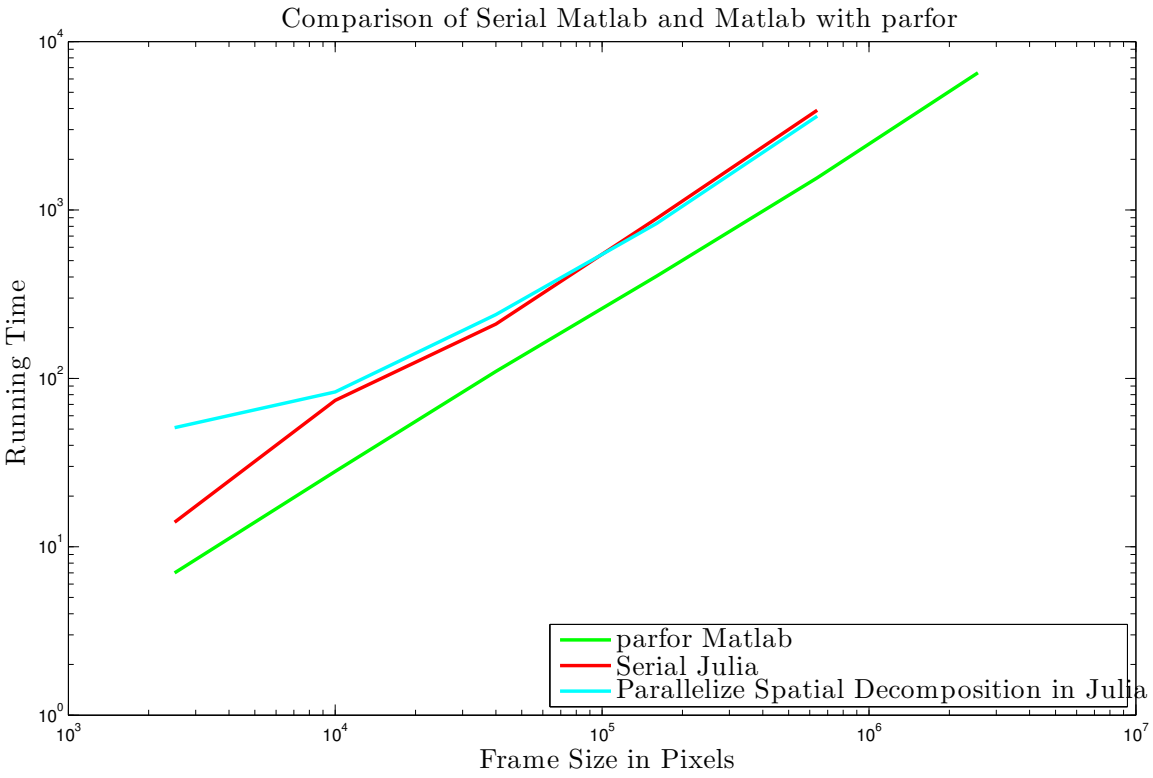


Figure 5: A comparison of a simple parallel version of the code in which the first and third stage are parallelized in Julia using spawnat. The code is much slower for small problem sizes, but slightly faster for large problem sizes. Most of the time is actually spent moving data between processors.

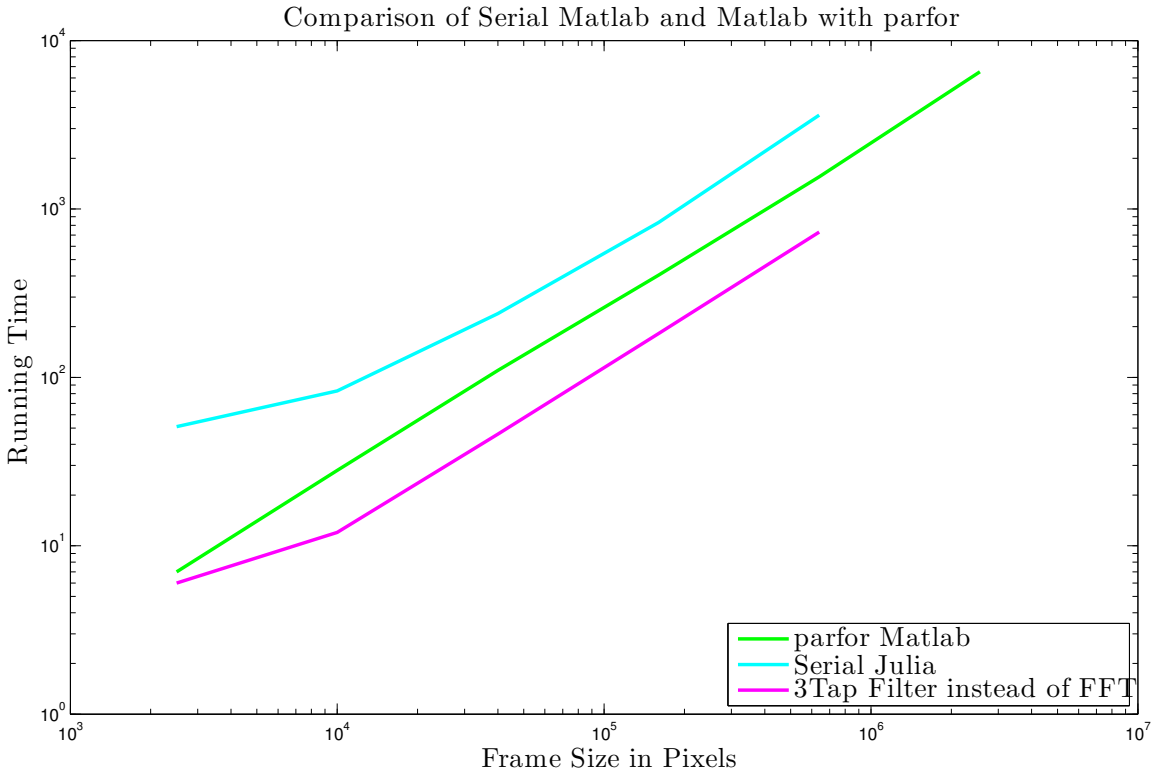


Figure 6: The algorithm was modified to use a local temporal filter rather than an ideal filter. This resulted in a huge speed up. The transparency of Julia's parallel computing suite made this possible. Matlab's parfor failed to capitalize on this.

the data was spread out across different cores, I modified the algorithm slightly to only need local temporal data. I achieved this by using a three tap filter in time instead of an ideal bandpass filter. This was much faster than the serial algorithm and much faster than the Matlab version of the algorithm. It was 2.5x faster than Matlab with parfor and 5x faster than serial Julia. I was surprised that Matlab failed to capitalize on the locality in time. See figure 6.

4 Conclusion

Parallelizing this algorithm was more challenging than I thought it would be. I thought the independence of the computations in each stage would lead to a very easy close to 12x speed up. However, it was more challenging to get that than I thought. I was also pleased with how easy it was to use Julia's parallel computing tools.