Parallel Viterbi Estimation & Gradient Ascent

Final Project for CS378 – Parallel Scientific Computing

Mark Kedzierski
Senior of Computer Science
University of Texas at Austin
mark@mark-kedzie.com

Abstract

Polyphonic music transcription can be solved effectively by stating it as a problem of Maximum a Posteriori state trajectory estimation (Viterbi[1]*) over a Switching Kalman Filter Model (Cemgil, 2003). Unfortunately, exact Viterbi calculation is generally intractable over SKFM's because of the exponential increase in number of mixture components which are required to accurately represent a Posterior density at each time-step. This is due to the requirement of keeping track of every possible combination of switch variables across all time steps; we gain a mixture component for every possible value of the switch variable, at each time step. Though pruning the propagated message does greatly increase performance; the calculations are still time-consuming due to the large transition matrices we multiply in the standard Kalman Filtering equations.*

Fortunately, the large cross-products can be computed much faster by parallel computers. I present two parallel implementations of polyphonic music transcription and compare their performance. The resultant four algorithms presented here can be applied more generally as Viterbi Estimation and Gradient-Ascent a.k.a. Hill-Climbing.

[1] For a complete source on Viterbi and Bayesian inference, see (Murphy)

Introduction

In this report I will compare two parallel algorithms for solving the problem of polyphonic music transcription based on Bayesian inference. Specifically, transcribing a digital recording of a single guitar player. The applications of this capability include music education, composition, and recording. Specifically I am focusing on the recording of a single guitar playing a polyphonic melody. First, we define a state space model based on (Cemgil, 2004) for generating audio signals not unlike those of stringed instruments. We then use our generative model to infer the most likely piano roll, i.e. a musical score, which would generate the audio waveform. The piano roll describes the musical performance which caused the digital audio data. This is done by generating waveforms for every possible piano roll and comparing them to the source. The closest waveform would give us the correct piano roll. In practice, (because we can't realistically make all those calculations), we do this in real-time as a Viterbi MAP state trajectory estimation problem using Switching Kalman Filters (Murphy, 2002). This means we propagate the most likely sequence of states which led to the current audio sample.

But even with estimation algorithms, the computations are still intractable, as they grow exponentially at

each time-step. In order for them to be useful, we need to take advantage of parallel processors.

Generative Model

The generation of a digitally sampled damped sinusoidal of frequency ω can be stated as a linear dynamical system given by (Cemgil, 2003). We assume that our observations of the system are digital samples taken at a constant rate Fs.

Observations - $y_{1:T}$

At each time slice the observation represents a single audio sample. The value of y_t represents the amplitude of the wave at time t. It has only a single component.

State Vector - $s_{1:T}$

The waves can be generated by a two-dimensional *oscillator* vector, s_t, which rotates around the origin with an angular frequency ω. The length of vector, $|s_t|$, corresponds to the initial amplitude of the sinusoidal and decreases over time at rate constant ρ. This is called the *damping coefficient*. Because the variables ω and ρ stay constant for a given model they are called parameters.

The observations are generated by projecting the 2-dimensional state vectors onto a 1-dimensional plane (over time). This is depicted in the figure below (Cemgil, 2003).

A damped oscillator in state space form. Left: At each time step the state vector, s_t, rotates by ω and its length becomes shorter. Right: The actual waveform is a one dimensional projection from the two dimensional state vector.

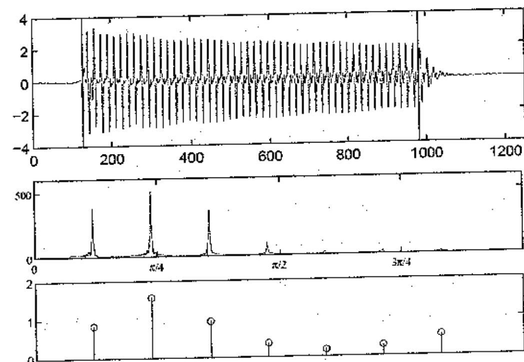
Figure 1

Initial state estimate - π

The initial state estimate, $\pi \equiv p(s_0)$, is drawn from a zero-mean Gaussian with covariance matrix S.

$$\pi \sim \mathcal{N}(0,S)$$

The diagonal sum of S, Tr(S), is proportional to the initial amplitude of sound wave. The covariance structure defines how the 'energy' is distributed along the harmonics (Cemgil, 2003). It is initially estimated with a strong 1st and 2nd harmonics. The later overtones have increasingly less energy. This pattern can be shown by a Fourier transform of the audio signal.



Top: Audio signal of electric bass. Originally at 22 kHz, down-sampled by factor D=20. Middle: Frequency Spectrum (DFT) of audio signal Bottom: Diagonal entries of S for H=7 correspond to overtone amplitudes.

Figure 2

Transition Matrix - A

The transition matrix, A, is responsible for rotating the state vector around the origin by ω and decreasing it's length by ρ (i.e. projecting $s_t \rightarrow s_{t-1}$). This is defined with Given's rotation matrix, $B(\omega)$, which rotates a two dimensional vector by ω radians:

$$A \equiv \begin{pmatrix} \rho\text{Cos}[\omega] & -\rho\text{Sin}[\omega] \\ \rho\text{Sin}[\omega] & \rho\text{Cos}[\omega] \end{pmatrix}$$

Observation Matrix - C

The observations are simply projections of the state vector. We

define the observation matrix as 1×2H dimensional projection matrix:

$$C = [0 \quad 1 \quad .... \quad 0 \quad 1]$$

Polyphonic Transcription

Polyphonic Music Transcription is a combination of Viterbi MAP trajectory estimation and iterative gradient ascent performed in windows. Viterbi estimation is used to track note onsets in each window, while the gradient ascent algorithm calculates the most likely chord which sounds for the entire window. We present both algorithms as well as parallel implementations.

Viterbi Estimation

Given an audio waveform, our goal is to infer the most likely sequence of piano roll indicators, $r_{1:T}$, which could give rise to the observed audio samples. This is, in general, a Viterbi estimation problem. The value we are seeking is defined as the *Maximum A Posteriori* trajectory:

$$r^*_{1:T} \equiv \underset{r_{1:T}}{\text{argmax}}\ p(r_{1:T} \mid y_{1:T})$$

It represents the most likely sequence of hidden variables to cause an observed output. This is different from the posterior distribution because it is just a point estimate. Calculation is the same as for filtering, except for replacing the summation over r_{t-2} with the maximization (MAP). It is important to note that, in Viterbi estimation, we propagate a *filtering potential,* $\delta_{1:T}$, as opposed to a *filtering density,* $\alpha_{1:T}$. The term potential used to indicate that this value is not normalized. This is sufficient because we only care about the best piano roll, (i.e. configuration $r_{t:t+1}$ with highest likelihood), we can save calculations by using a point estimate.

$$p(r_{1:T} \mid y_{1:T}) \propto \int_{s_t} p(y_{1:t} \mid s_t, r_{1:t})\, p(s_{1:t} \mid r_{1:t})\, p(r_{1:t})$$

Inference is more difficult in the switching state-space model because we have to keep track of every possible enumeration of $r_{1:T}$ and return the sequence with the highest likelihood. The representation of the filtering potential is a Mixture of Gaussians (MoG) with a single component for each enumeration of $r_{1:T}$,(we use H=1 to simplify the examples):

$$\delta_{t|t} \equiv p(y_{1:t}, s_t, r_t, r_{t-1}) \equiv \begin{Bmatrix} \delta_t(1,1) & \delta_t(1,2) \\ \delta_t(2,1) & \delta_t(2,2) \end{Bmatrix}$$

where each,

$$\begin{aligned} \delta_{t|t}(i, j) &\equiv p(y_{1:t}, s_t, r_t = i, r_{t-1} = j) \\ &\equiv p(y_{1:t}, s_t, r_t = i, r_{t-1} = j) \end{aligned}$$

is also a MoG.

At each time step we make separate estimates for every configuration of piano roll indicators $\{r_t, r_{t-1}\}$. This corresponds to the prediction step. It can also be considered an *expand* step. As shown by the left side of the next figure.

Kalman Filter Equations

To propagate our posterior state estimate across time slices we use the Kalman Filter[2] equations.

$P_k^- = A P_{k-1} A^T + Q$

$\hat{s}_k^- = A s_{k-1}$

$K_k = P_k^- C^T (C P_k^- C^T + R)^{-1}$

$\hat{x}_k = \hat{x}_k^- + K_k (y_k - C \hat{x}_k^-)$

$P_k = (I - K_k C) P_k^-$

At each time iteration, we compute 10 cross products for each

[2] For a great introduction to Kalman Filtering, refer to (Welch, 2001)

piano roll configuration for each mixture component. We discuss parallelization in a later section.

Polyphonic Chord Transcription - Gradient Ascent

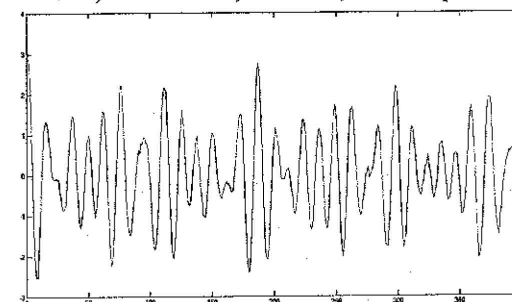
In this section we use the Viterbi estimation techniques learned in the last section use them to transcribe polyphonic chords. The number of notes in the model is represented by M. The difference in this case is that we infer the variables $r_{1:M}$, rather than $r_{1:M,1:T}$. We assume that the configuration stays constant for the entire window, t=0:T.

The algorithm, given in (Cemgil, 2003), is a simple greedy search. We start with at an initial estimate of the chord configuration (zero's or drawn from prior), and uniform prior $p(r_{1:M})$. A more informative prior can be very helpful in finding the correct configuration, $r_{1:M}$. We then calculate the likelihood of our state estimate, δ, and all neighboring[3] configurations differing from our by a single note. If δ is the most likely, we stop (at the local maximum). Otherwise we continue until convergence.

The following section shows some of my results from chord transcription experiments. The best chord configuration at each iteration is represented equally as a Boolean array of length M[4]. A listing of actual note names which make up the chord (chord tones) are included after the Boolean array. Asterisks indicate a correct estimate. The system parameters used are listed as well.

[3] Neighboring configurations are those which differ by one note. (Cemgil)
[4] M represents the number of notes in our system model

12 notes, 1 Harmonic, 4096 Hz, 400 samples



Actual chord configuration	chord tones
1 2 2 1 2 2 2 1 2 2 2 2	[A, C, E]

	Chord Config	-Log Likelihood
i=0	2 2 2 2 2 2 2 2 2 2 2 2	-156933.7858
i=1	2 2 2 2 1 2 2 2 2 2 2 2	-18761.5701
i=2	2 2 2 2 1 2 2 1 2 2 2 2	-6361.4038
i = 3	1 2 2 2 1 2 2 1 2 2 2 2	-332.5287
i = 4	1 2 2 1 1 2 2 1 2 2 2 2	273.0216
i = 5	1 2 2 1 2 2 2 1 2 2 2 2	287.7244 **

(Kedzierski, 2005)

Figure 3

Because the algorithm does stop at a local maximum, it can get stuck before finding the real maximal likelihood. It is best to run the algorithm several times from different initial chord configurations[5] drawn from the prior and take the best result.[6]

To run the gradient algorithm we need to calculate M *filtering potentials* over the entire span of the window. Note that because we are dealing with potentials, as opposed to densities, we do not need to filter all chord configurations. We just calculate a point estimate of the likelihood in the posterior state distribution.

This can be accomplished using the Kalman Filtering Equations but requires 10 matrix multiplications for each note on each time-slice. Realistic

[5] Initial chord configurations are drawn from a prior distribution over chords. Prior defined in (Kedzierski, 2005)
[6] Best result refers the the chord with the highest likelihood.

window size would be 400 samples[7], which results in 4,000 matrix multiplications for *each* note in the model at *each* time slice. realistic model would have approximately 50 notes. Parallelization could greatly benefit the performance of the algorithm by speeding up the matrix multiplications.

Parallel Implementations

Next we implement and benchmark two different parallel implementations of the Viterbi and Gradient algorithms.

Parallel Implementation I - Cannon Matrix Multiplication

The first method uses the same filtering algorithms as above but calculates each Matrix cross-product on 2HM processors using Cannon's algorithm. This effectively reduces the time of the calculation to a single dot product.

Here the source code for a serial matrix multiply:

```
for(i=0;i<rows;i++)
  for(j=0;j<b->cols;j++) {
    float sum = 0;
    for(k=0;k< cols;k++)
      sum += data[i][k]*
             b->data[k][j];
      c->data[i][j] = sum;   }
```

It is apparent from the loop structure that we need to calculate 2*H*M dot products and combine them. which takes n^3 dot products of dimension N. We use the Cannon algorithm to lower that to just a single dot product on 2HM processors. In the code for cannons algorithm we assign each index in the resultant state vector/matrix to a single processor.

[7] From 8,000Hz signal downsampled by a factor of 2

```
int row = my_rank/N;
int col = my_rank%N;

for(i=0;i<N;i++) {
      //Dividing the work
      local_x[i] = a[row][i];
      local_y[i] = b[i][col];}
int local_dot =
Serial_dot(local_x, local_y, N);
```

The result matrix is the same size as in the inputs (2HMx2HM). The processors are synchronized before calculating each cross product when the host sends each of them their assigned vectors using MPI_Send. The root processor then performs its share of work and waits to receive confirmation from each of the processes indicating they have completed their calculations. Once they are all returned the root combines the data.

This method has the trade-off of frequent synchronization too lightning fast matrix multiplication. It can be further sped up by more processors by splitting up the dot products. However, My intuition says that the frequent synchronizations should slow execution of the parallel algorithm.

The first method applies in the same way to both Viterbi and Gradient algorithms so there is no need to distinguish between them when discussing it. However, when exploring the next parallelization we will consider each algorithm separately.

Parallel Implementation II - Viterbi MAP Estimation

In the second method we use a more creative way of separating the workload among the processors. We assign each processor to a single transition matrix and send it a message. This results in having two processors performing calculations for each note in

the model. One performs the note-on transition function, the other the mute transition function to each. The cross products are performed in serial.

In order to do this effectively we have to divide all the components depending on the combination of sound/mute states for the last 2 time slices. This is accomplished by storing the mixture of Gaussians message in the following structure of the transition function:

$$f(r_{t,j} \mid r_{t-1,j-1}) = \begin{pmatrix} f_{1,1} & \square & \square & \square & f_{2,1}/M & \cdots & \cdots & f_{2,1}/M \\ \square & \ddots & \square & \square & \vdots & \ddots & \iddots & \vdots \\ \square & \square & \ddots & \square & \vdots & \iddots & \ddots & \vdots \\ \square & \square & \square & f_{1,1} & f_{2,1}/M & \cdots & \cdots & f_{2,1}/M \\ f_{2,1} & \square & \square & \square & f_{2,2} & \square & \square & \square \\ \square & \ddots & \square & \square & \square & \ddots & \square & \square \\ \square & \square & \ddots & \square & \square & \square & \ddots & \square \\ \square & \square & \square & f_{2,1} & \square & \square & \square & f_{2,2} \end{pmatrix}$$

Structure of polyphonic transition function

Figure 4

The matrix is organized into 4 regions which represent the possible combinations of the last two time slices. The individual components will move from region to region when they change states. On a note onset the component is initialized to region (1,2). When they continue ringing they remain in the (1,1) region. When the note is released, it moves to region 2,1. The region 2,2 is for silent notes.

The component, (1,2) is different because it represents a note onset. It's important to note that the piano roll configuration before the onset, r1:onset , doesn't effect the likelihood of future indicators after it. This enables us to we can replace messages from xt-1 mute with the maximum (scalar) likelihood estimate among them. We introduce this scalar value, Z_{t-1} mute, as a prior for the next onset and tag the message with $r_{1:t-1}$*. This replacement enables the algorithm to be tractable by allowing pruning of the (2,2) region without losing accuracy. We can then actually put a flat limit on the number of components to the number of processors available. To be effective, we need about a 100 processors.

The processors are forced, however, to synchronize after each time step to compare the likelihoods for each the possible chord configurations, so pruning can be performed at the end of each iteration. This could be modified so that the sync is only done after several time-steps. The trade-off This is done on the first 'master' processor by waiting for messages from all the processes using MPI_Recv. Each process sends a N length vector which the host process appends to form the new matrix. and then the new state vector is distributed among the processors using MPI_Bcast. the synchronization is again done by the host process receiving messages from all the processes.

Parallel Implementation II - Gradient Ascent Estimation

When we perform gradient ascent we need to filter a single chord configuration for each note in the model. This means that we do M passes in each iteration. I propose splitting the work among M processors and have each processor compute one configuration. The processors have to be synchronized at each iteration of the algorithm to identify the highest log likelihood[8].

[8] We use the log likelihood to avoid common underflow issues

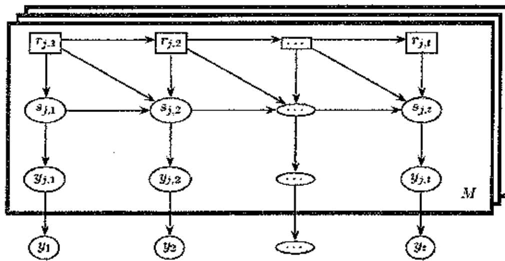


Figure adapted from (Cemgil, 2003), shows M copies of the generative model stacked as 'plates'. In the parallel algorithm, each plate would be assigned a processor.

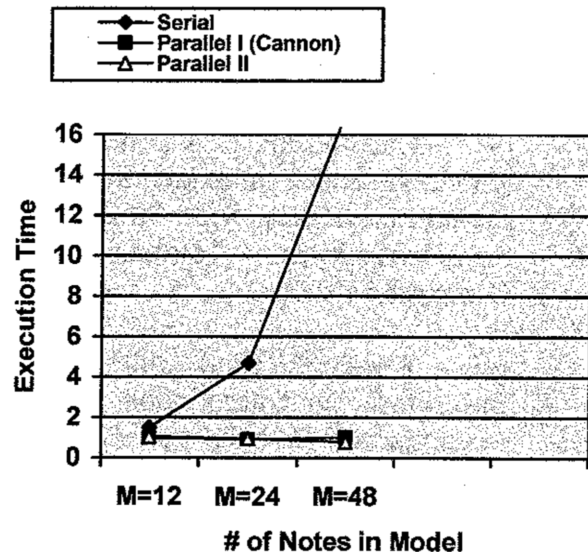
Figure 5

Performance Benchmark

The generative model for the following experiments was trained on my particular guitar using the EM[9] algorithm for 5 iterations. The audio segments where down-sampled from 8,000 Hz by a factor of 20 and this was accounted for in the model by adjusting the angular frequencies of the oscillator bank.

The results show the obvious extreme performance increase when going to parallel. The serial algorithm's execution time grows exponentially as the size of model increases while the parallel algorithms stay the same. Suprisingly, there wasn't as much difference in the running times of the two parallel algorithms in the experiments I ran, however.
We show 3 separate runs using the serial and both parallel algorithms on models with 12, 24, and 48 notes. H = 5 for all models. The samples audio data contained 400 samples.

[9] Expectation Maximization. See (Ghahramani,1996)



Future Work

The next step is to modify the parallel algorithm to syncronize less often; I believe improvements in performance could still be made.

I also plan to research OpenMPI solutions which could be useful for professional and home recording studios running processor clusters. A VST[10] serial plug-in implementing polyphonic transcription in serial mode is already in the works.

References

Sorted Alphabetically

[1] Cemgil, A. T., Kappen, H. J., & Barber, D. (2003). Generative model based polyphonic music transcription. In Proc. of IEEE WASPAA, New Paltz, NY. IEEE Workshop on Applications of

[10] Virtual Studio Technology. See http://www.steinberg.net

Signal Processing to Audio and Acoustics.

[2] Cemgil, A. T. Bayesian Music Transcription. PhD thesis, Radboud University of Nijmegen, 2004.

[3] Ghahramani, Z., & Hinton, G. E. (1996). Parameter estimation for linear dynamical systems. (crg-tr-96-2). Tech. rep., University of Totronto. Dept. of Computer Science.

[4] Kedzierski, Mark (2005). Bayesian Guitar Transcription. Undergraduate research rep, University of Texas at Austin, Deparment of Computer Science.

[5] Murphy, K. P. (2002). Dynamic Bayesian Networks: Representation, Inference and Learning. Ph.D. thesis, University of California, Berkeley.

[6] Welch, G., & Bishop, G. (2001). An Introduction to the Kalman Filter. SIGGRAPH course 8, University of North Carolina at Chapel Hill.