Parallel Discrete Hidden Markov Model (DHMM)

Aashima Bhatia (CS13M001) Arjun Lal B (CS13M006)
Parth Joshi (CS09B051) Sudharshan GK (CS13M050)

April 18th, 2014

1 Introduction

Hidden Markov Models (HMMs) find several applications in bioinformatics, speech and video processing (sequential pattern recognition), protein structure modelling, sequence alignment etc. They have been used widely for sequential patterns of lengths in the range of millions or billions. There are aspects of HMMs that readily lend themselves to parallelization and we have attempted to use the ideas presented in [1] to come up with our own parallel implementation of two key problems associated with HMM training, namely the evaluation and the alignment problems.

2 Description

In order to be able to describe the basic outline of what we have done as part of this project and our parallel implementation, we will define very briefly the basic notation used to describe a Hidden Markov Model and its associated algorithms. An HMM is parametrized as $\lambda = (\pi, A, B)$ where,

$A(N \times N)$ is the transition probability matrix; a_{ij} corresponds to the probability of transitioning from state i to state j.

$B_o(N \times N)$ is a partial emission probability matrix; it is a diagonal matrix with b_{kk} corresponding to the probability of emitting symbol o in state k.

$\pi(1 \times N)$ is the initial state probability vector; π_i corresponds to the probability that the observation sequence begins in state i.

In the design of an HMM model there are three primary issues that must be resolved.

1. **The Evaluation Problem -** Given a model and a sequence of observations, how do we compute the probability that the observed sequence was produced by the model.
   Consider the observation sequence $O = o_1, o_2, o_3 \ldots, o_T$. The most efficient method to compute this probability is in terms of the so-called *forward variable vector* $\alpha_\mathbf{t}$ or *backward variable vector* $\beta_\mathbf{t}$.
   The recursive formulation for $\alpha_\mathbf{t}$ is given as

$$\begin{aligned} \alpha_\mathbf{t} &= \alpha_{\mathbf{t-1}} \cdot A \cdot B_{o_t} \forall t = 2, \ldots, T \\ \alpha_\mathbf{1} &= \pi \cdot B_{o_1} \end{aligned}$$

We can see that this sequential computation can be unfolded to remove the dependence of $\alpha_{\mathbf{t}}$ on $\alpha_{\mathbf{t-1}}$. Define the transient probabilities $C_t(N \times N)$ as

$$\begin{aligned} C_t &= A \cdot B_{o_t} \forall t = 2, \ldots, T \\ C_1 &= \pi \cdot B_{o_1} \end{aligned}$$

Then

$$\alpha_{\mathbf{T}} = C_T \cdot C_{T-1} \cdots C_2 \cdot C_1$$

The primary idea we are planning to exploit is the associativity of matrix multiplication to parallelize the above matrix multiplication. However, it should be noted that the α_t's are vectors while the C_t's except for C_1 are matrices. Therefore the complexity of the sequential algorithm is $O(N^2T)$ while the parallel one is $O(N^3T)$. One advantage we have taken advantage of for the parallel approach is that since the number of observation symbols M is finite and usually much smaller than the sequence length T, the C_t's can be pre-computed.

2. **The Sequence Alignment Problem -** The solution to optimal state sequence alignment is given by the well-known Viterbi algorithm. Although the Viterbi algorithm is a dynamic programming (DP) algorithm, the manner in which we have attempted to parallelize it is different from the regular parallelization of DPs, as taught to us in class. We have made use of the special structure of the problem in our case, where the DP algorithm can be unfolded in terms of associative operations over the set of matrices, in the manner used for the evaluation problem. Here, the goal is to determine the state sequence that, in terms of probability, best explains the given observation sequence. It is interesting to note that the idea of using the associativity of matrix multiplication carries forward to this problem but with a different associative operator max-multiplication $\cdot_m$. If matrix multiplication $\cdot$ is defined as

$$(X \cdot Y)_{ij} = \sum_k X_{ik} \times Y_{kj}$$

then matrix max-multiplication can be defined as

$$(X \cdot_m Y)_{ij} = \max_k (X_{ik} \times Y_{kj})$$

Implementing the computation of the *cost table* in the DP can then be reduced to repeated application of the above operation on the C_t matrices. Once the optimal cost i.e. the maximum probability is known, we can do the usual backtracking (with a little variation) to determine the actual state sequence that produced the highest probability.

3. **The Parameter Estimation Problem -** This can be done using the Baum Welch re-estimation procedure which is an Expectation Maximization algorithm. We had listed its implementation as one of our "maybes" but were unable to complete it in the timeframe we had.

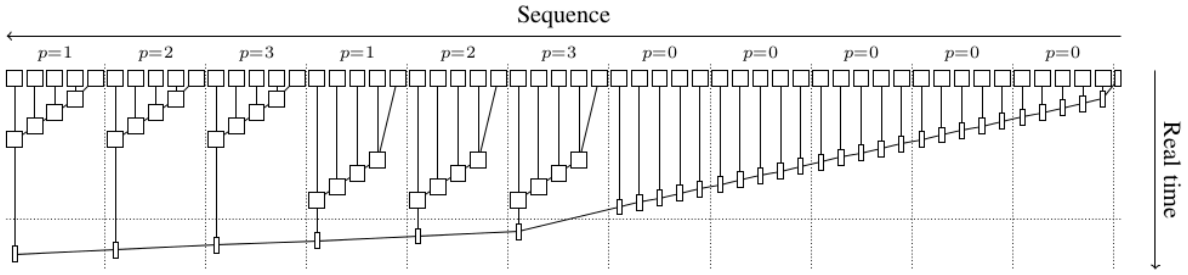
3 Implementation

We have used PThreads for our implementation with the code being written in C++.

3.1 Forward Evalutation

Figure 1 shows the schema for our implemetation of the forward evaluation algorithm. As was mentioned earlier, the vector-matrix multiplications require $O(N)$ more computations than the matrix-matrix multiplications. This fact makes the sequential algorithm faster than a naive parallel algorithm. We take advantage of this fact in our implementation as follows.

Figure 1: The schema for the forward evaluation computation. The squares indicate matrices and the vertical rectangles are vectors with lines indicating data dependecies. [1]



The set of C_t matrices as indicated by the top-most row in the figure are divided up into chunks or blocks that are maintained in a doubly ended queue. The size of each block (the number of C_t matrices in it) is taken to be the square root of the length of the sequence T. One thread picks off blocks from the head of the queue while the others all pick blocks from the tail. This ensures that the first thread which is going to be faster gets a larger share of the computation load. Locks have been used for synchronization since we have used the C++-98 standard which does not provide compare-and-swap or get-and-set like operations for lock-free synchronization. When the queue becomes empty, all the blocks have been processed so that we have the result matrices from each block available. The results are then collected by the main thread to obtain the final evaluation probability.

3.2 Viterbi Algorithm

Figure 2 shows the schema for our implemetation of the Viterbi algorithm. The primary difference here is that the operation being performed on the vectors and matrices is max-multiplication instead of multiplication. Further, once the first phase which is similar to the evaluation problem is over, we must backtrack to compute the optimal state sequence. This is done as follows.

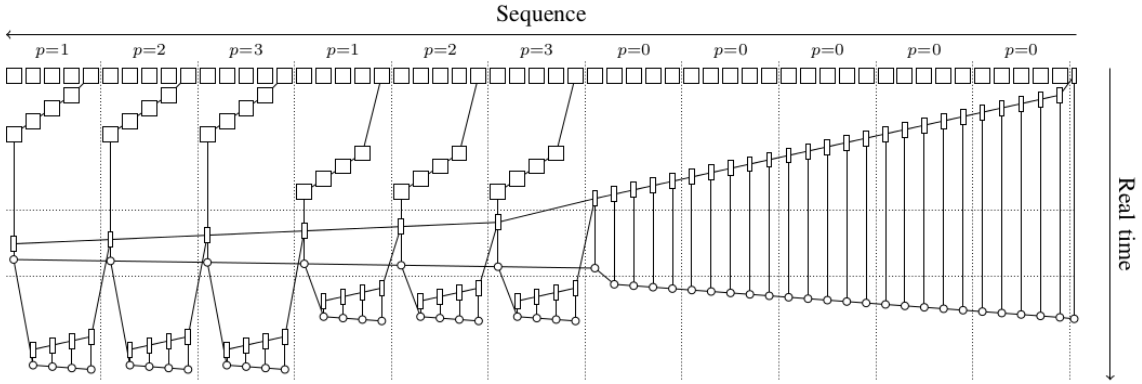
After phase I, we have the probabilities of the best paths upto the times t that form the block boundaries. We then backtrack at the block level to determine the states that are part of the optimal state sequence at the times t. This is depicted in the figure by the first layer of circles. Once we are done with the block level computation, we can again process the blocks independently and proceed to phase II. We re-initialize the queue and again assign blocks to each thread in the same manner as before. Each thread now performs only vector-matrix max-multiplications to obtain the cost table and then backtracks to get the partial optimal state sequence within its block. At the end of this process, the entire optimal state sequence has been determined.

4 Experiment

We are using the TIDIGIT database, which is a standard digit database used for speech recognition models for digits. The database comprises recordings (.wav files) from a large number of speakers corresponding to each digit. Some amount of preprocessing to extract the appropriate features from the recordings that is required is done by separate scripts. The output of the above scripts along with a configuration file (for the initial transition and emission probability) is taken as input.

The experiment is done with variable length observations ,variable state sizes and different symbols.

Figure 2: The schema for the Viterbi computation. The squares indicate matrices and the vertical rectangles are vectors with lines indicating data dependecies. The circles denote the states in the optimal state sequence for the respective symbols. [1]



5 Results

5.1 Forward Evalutaion

Figure 3 shows the results we obtained for different sequence lengths and different number of threads for the evaluation problem. It can be seen that the run times show a weak scaling behaviour with the performance of the algorithm improving as the sequence length increases. Further, the performance improves as more threads are employed in the computation.

5.2 Viterbi Algorithm

Figure 4 shows the results we obtained for different sequence lengths and different number of threads for the Viterbi algorithm. It can be seen that the run times show a weak scaling behaviour with the performance of the algorithm improving as the sequence length increases. However, the performance improvement is not that good, particularly when we use more than two threads. This can be attributed to thread creation and destruction overhead when we use a larger number of threads because in our current implementation, we use one set of threads for phase I of the algorithm and then create a new set of threads for phase II. This is responsible for a large overhead particularly when block sizes are small. A solution for this problem would have been the use of barriers to synchronize rather than joining threads. This would allow the same threads to continue the computation where they left off and also avoid some redundant computation that is being performed in the current implementation while at the same time avoiding much of the overhead of thread creation and destruction.

6 Correctness

We are comparing our results with existing sequential code using the same initial parameters for training. We have knowledge of a program written by Richard Myers and Jim Whitson. However, it does not do Viterbi alignment as part of its training. There are a number of HMM libraries that implement the sequential versions of the above algorithms but with minor variations and we considered the most suitable one for our purposes. Moreover, we used our own sequential implementation to compare results.

The output is accurate and is same as that of sequential execution output.

Figure 3: Comparison of the running times of the evaluation algorithm for different sequence lengths and different number of threads

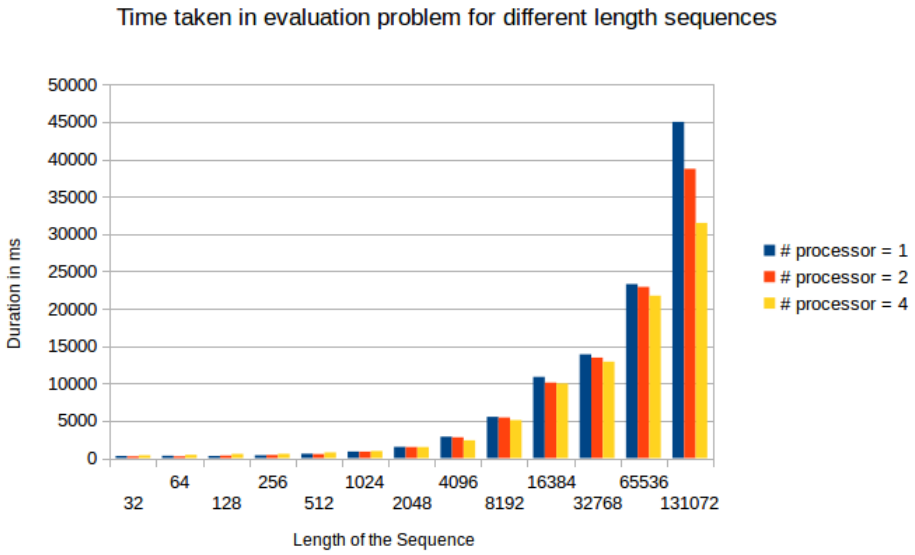
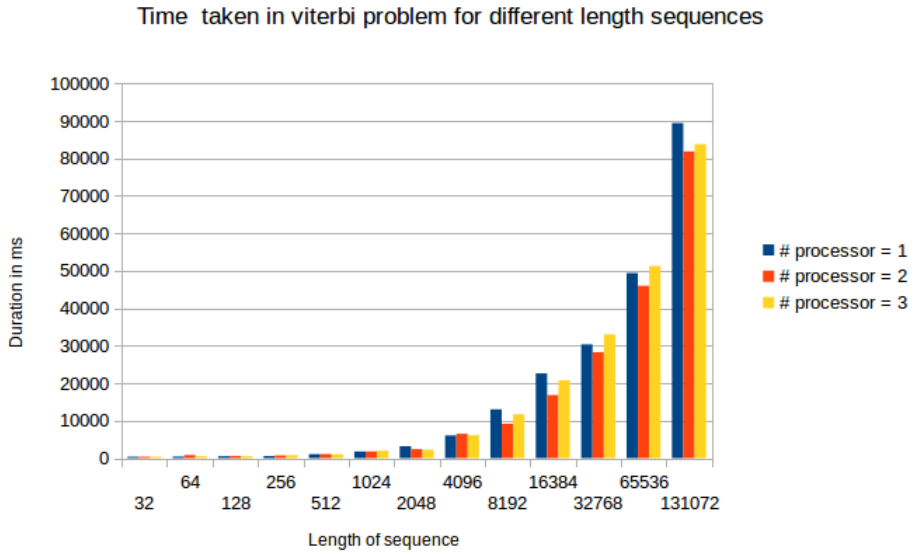


Figure 4: Comparison of the running times of the evaluation algorithm for different sequence lengths and different number of threads



References

[1] Algorithms for a parallel implementation of Hidden Markov Model with a small state space, Jesper Nielsen, Andreas Sand, *IPDPS* 2011. http://www.hicomb.org/papers/HICOMB2011-06.pdf

[2] Unidirectional and Parallel Baum–Welch Algorithms, William Turin, *IEEE TRANSACTIONS on Speech Audio Processing*, 1998.

[3] A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition, Lawrence E. Rabiner, *Proceedings Of The IEEE, Vol. 77 No. 2*, 1989.