15-740 Computer Architecture, Fall 2009
Project Final Report

Bin Fu (binf@cs.cmu.edu) 1st Dec. 2009

1. Introduction

In this course project I try to parallelize the well-known Hidden Markov Model in machine learning area. This work is motivated by one previous course project in this class, [Li, et al. 08], where the authors parallelize another similar application (Linear Dynamic Systems).

More generally, I try to deal with a specific problem in Hidden Markov Model: the Baum-Welch algorithm (http://en.wikipedia.org/wiki/Baum–Welch_algorithm), which is strongly related to the better known Forward-Backward Algorithm. Due to the data dependency nature inside the Baum-Welch algorithm, the parallelization is an approximate procedure, although the experiments show that our parallel algorithm achieves same level of data likelihood as the original sequential algorithm. The parallel algorithm also shows very promising speedup using a small number of processors. I envision this method can be further applied to other machine learning frameworks as well, including the Conditional Random Field (CRF, http://en.wikipedia.org/wiki/Conditional_random_field).

Actually, during recent years people have been working on the parallel machine learning algorithms a lot. Many algorithms have been looked at, and almost linear speedup is achieved to k-means, logistic regression, SVM, PCA and EM etc. ([Chu, et al. 06]).

The following parts of the report are organized as follows: in Section2 I will introduce the details on both the Baum-Welch algorithm and our parallelization ideas. Experiments and related analysis will be given in Section3, and some tips during the project will be presented in Section4. I will finalize the project in Section5 at last.

2. Parallel Hidden Markov Model

In this section I will first introduce the general Hidden Markov Model (section 2.1) and the specific problem I am trying to parallelize (Baum-Welch algorithm, at section 2.2), then present our parallel idea (section 2.3).

(In this section I have to copy some formula figures from Latex which I derived earlier, so the resolution of those figures might not be very clear. I'm sorry for the inconvenience in advance.)

2.1 Hidden Markov Model

Hidden Markov Model is a sequential machine learning framework, which is presented in Figure1. Hidden Markov Model is widely used in temporal applications such as speech systems, handwriting systems and Bioinformatics field.

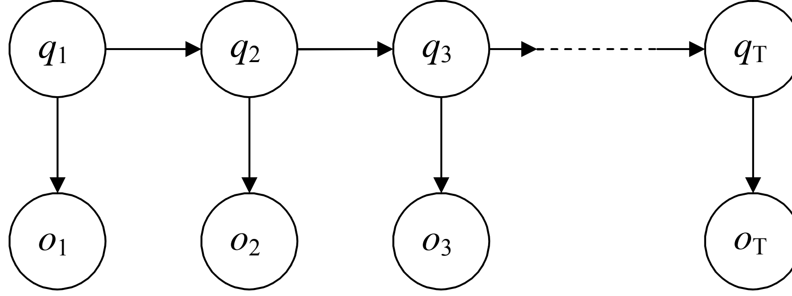

Figure1: Hidden Markov Model

In the Hidden Markov Model, we have a series of Observations $O = (o_1, o_2 \ldots o_T)$. Each observation has M possible values ($o_i = v_1, v_2 \ldots v_M$). The observations are directly "controlled" by a set of hidden variables which we cannot see $X = (q_1, q_2 \ldots q_T)$, where each hidden variable has N possible values ($q_i = s_1, s_2 \ldots s_N$).

The main property of Hidden Markov Model (actually Markov Process) is the Markov property, which is that the hidden variable q_i is only depended to its previous hidden variable q_{i-1}. All hidden variables before q_{i-1} have no effect in q_i.

We use a set of parameters $\lambda = \{A, B, \Pi\}$ to represent a Hidden Markov Model where:

Transition Matrix $A = \{a_{i,j}\}$ s.t. $a_{i,j} = P(q_t = s_j \mid q_{t-1} = s_i)$

Observation Matrix $B = \{b_i(v_k)\}$ s.t. $b_i(v_k) = P(o_t = v_k \mid q_t = s_i)$

Initialization Vector $\Pi = \{\pi_i\}$ s.t. $\pi_i = P(q_1 = s_i)$

Please notice that in Hidden Markov Model transition matrix A and observation matrix B are constant for all time series.

One problem inside Hidden Markov Model is the Training problem: given all the observations $O = (o_1, o_2 \ldots o_T)$, which set of parameters λ can best "explain" the observations. More formally, we want to find the parameters λ such that:

$$\lambda = \arg\max_{\lambda} P(o_1, o_2, ..., o_T|\lambda)$$

However, there is no exact tractable algorithm that can solve this problem. Baum-Welch algorithm and Baldi-Chauvin algorithm on the other side can iteratively achieve the local maximum likelihood. So in this project we look at the Baum-Welch algorithm, an EM-like algorithm to parallelize.

2.2 Baum-Welch algorithm

In order to solve the Training problem in the Hidden Markov Model, we iteratedly update the parameter λ. EM algorithm promises that the data likelihood will not decrease over time. So we keep updating the parameters, until the difference of data likelihood between two iterations is smaller enough, when we stop the algorithm.

At each iteration, we calculate the following values:

$$\alpha_t(i) = P(o_{1 \ldots} o_t, q_t = s_i \mid \lambda) \quad i = 1,2,\ldots N$$

$$\beta_t(i) = P(o_{t+1\ldots}o_T \mid q_t = s_i, \lambda) \quad i = 1,2,\ldots N$$

The calculation of $\alpha_t(i)$ is conducted from the first variable (q_1) to the last (q_T) (**Forward**, (1)), while the calculation of $\beta_t(i)$ is conducted from the last variable (q_T) back to the first (q_1) (**Backward**, (2)). That's why this algorithm is also known as the **Forward-Backward** algorithm:

$$\alpha_1(i) = \pi_i b_i(o_1)$$

$$\alpha_{t+1}(i) = b_i(o_{t+1})\Sigma_{j=0}^{N-1}\alpha_t(j)a_{ji} \tag{1}$$

$$\beta_T(i) = 1$$

$$\beta_t(i) = \Sigma_{j=0}^{N-1}\beta_{t+1}(j)a_{ij}b_j(o_{t+1}) \tag{2}$$

Next, we need a couple of auxiliary variables $\gamma_t(i)$ and $\xi_t(i, j)$ to help us easier updating the model parameters, and their calculations only depend on $\alpha_t(i)$ and $\beta_t(i)$:

$$\gamma_t(i) = P(q_t = s_i \mid O, \lambda) \quad i = 1,2,\ldots N$$

$$\xi_t(i, j) = P(q_t = s_i, q_{t+1} = s_j \mid O, \lambda) \quad i,j = 1,2,\ldots N$$

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{\Sigma_{j=0}^{N-1}\alpha_t(j)\beta_t(j)} \tag{3}$$

$$\xi_t(i,j) = \frac{\gamma_t(i)a_{ij}b_j(o_{t+1})\beta_{t+1}(j)}{\beta_t(i)} \tag{4}$$

Finally, we use the generated values to update the model parameters:

$$\pi_i^{'} = \gamma_1(i)$$

$$a_{ij}^{'} = \frac{\Sigma_{t=1}^{T-1}\xi_t(i,j)}{\Sigma_{j=0}^{N-1}\Sigma_{t=1}^{T-1}\xi_t(i,j)} \tag{5}$$

$$b_i(v_k)^{'} = \frac{\Sigma_{t=1,o_t=v_k}^{T}\gamma_t(i)}{\Sigma_{t=1}^{T}\gamma_t(i)} \tag{6}$$

For each iteration, the computation complexity of Baum-Welch algorithm is O(NT(N+M)). Although this complexity is not very high itself, the algorithm sometimes needs to be run a lot of iterations before convergence. I also find it time-consuming for large N,M,T (please refer to Section3).

2.3 Parallel idea of Baum-Welch algorithm

As I just mention, we still need to find a way to accelerate the process. The idea is very similar to the one proposed in [Li, et al. 08].

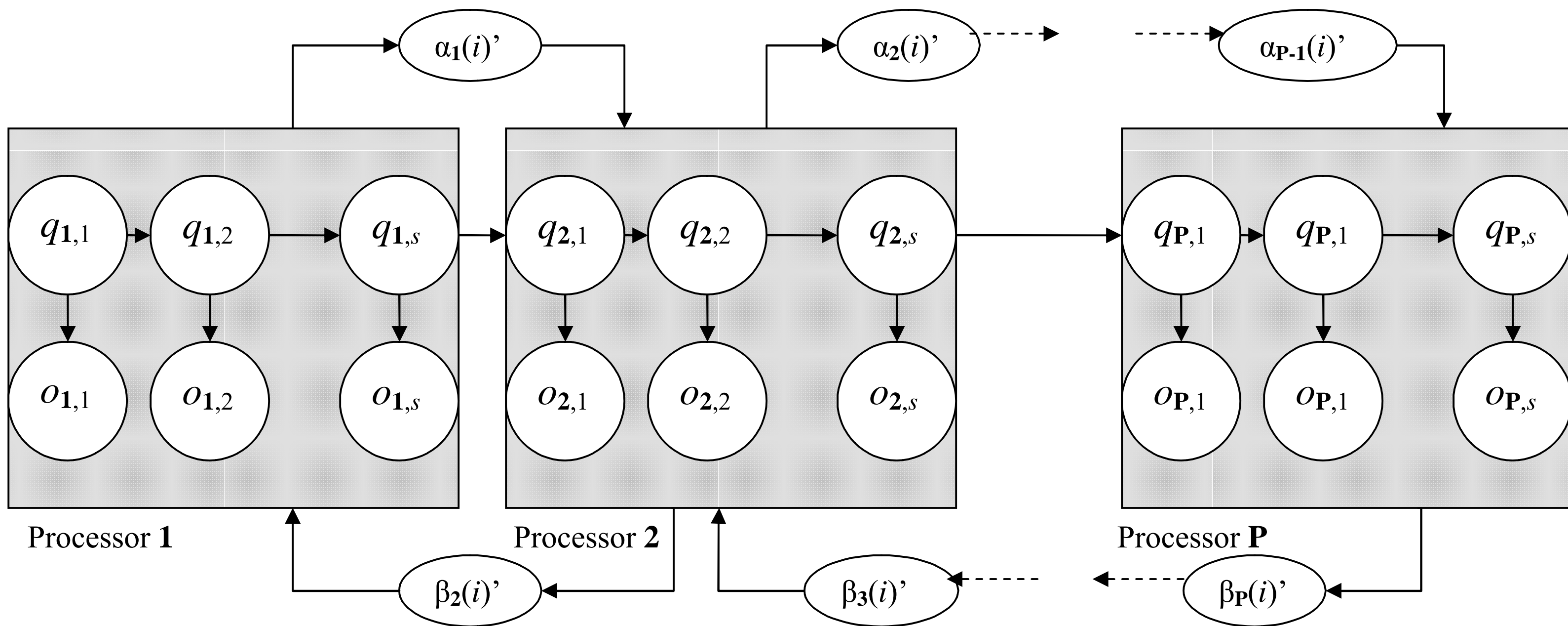

Figure2: Parallel idea of Hidden Markov Model

We first "cut" the original Hidden Markov Chain (with length T) to several blocks. Assume we will use P processors to run the parallel Baum-Welch algorithm, and we have **T = P * s**, where s is the length of sub-sequence that each processor need to deal with (Figure2).

Then each processor can simultaneously run the Forward-Backward algorithm for its sub-sequence. For example, processor1 will deal with $q_{\mathbf{1},1}$, $q_{\mathbf{1},2}$,…, $q_{\mathbf{1},s}$. Two things then need to be solved: How to communicate between adjacent blocks when calculating $\alpha_t(i)$ and $\beta_t(i)$, and how do the processors work together to update the model parameters.

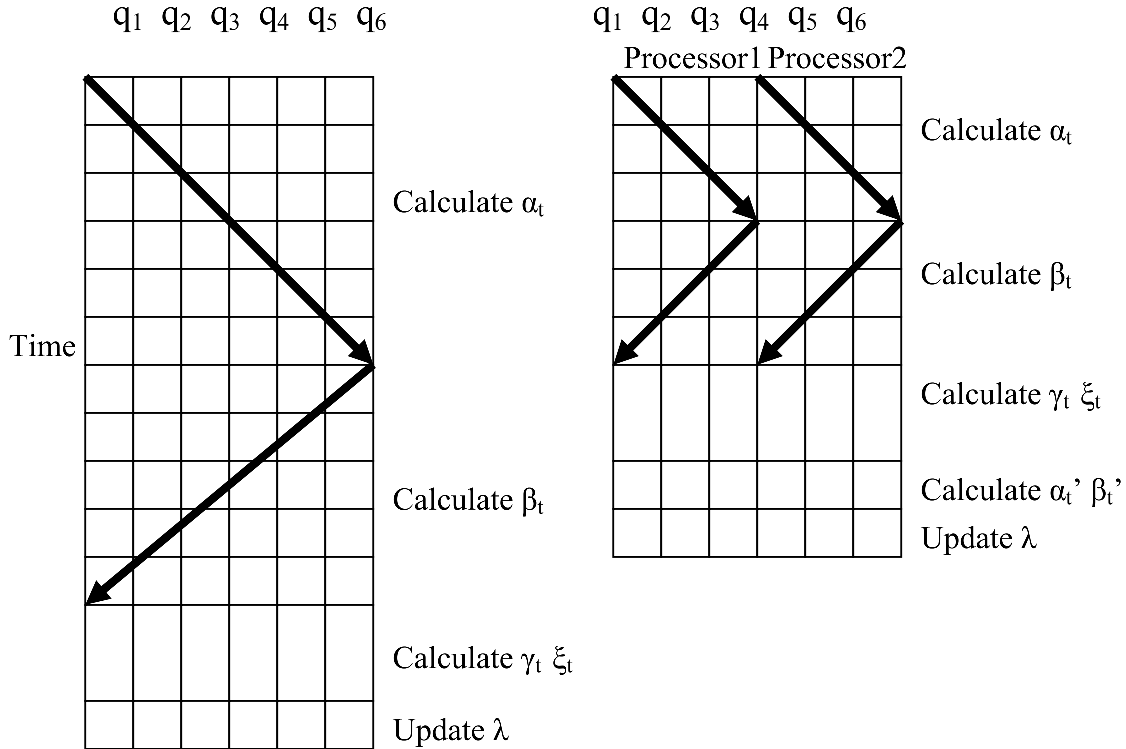

Figure 3: Flowchart of the mth iteration of Sequential Baum-Welch (left) and Parallel Baum-Welch (right), with T = 6 and P = 2. For the parallel algorithm here, Processor1 will use the $\beta_{\mathbf{2},0}(i)$ calculated at $(m\text{-}1)^{th}$ iteration; Processor2 will use the $\alpha_{\mathbf{1},3}$ calculated at $(m\text{-}1)^{th}$ iteration.

2.3.1 Communicate between adjacent blocks

Figure3 shows a simple example of how the sequential and parallel Baum-Welch algorithm performs. In the example, the length of sequence is 6 and we use 2 processors to parallelize it. At the mth iteration, sequential Baum-Welch algorithm needs to first calculate $\alpha_t(i)$ from q_1 to q_6, then $\beta_t(i)$ back from q_6 to q_1. After both $\alpha_t(i)$ and $\beta_t(i)$ are available, it then use them (and other auxiliary parameters) to update the model parameter λ.

For the parallel Baum-Welch algorithm, since two processors can simultaneously calculate its own $\alpha_{p,t}(i)$ and $\beta_{p,t}(i)$ (p is the index of processor), a large proportion of time will be saved. However, Processor1 cannot wait for the $\beta_{\mathbf{2},0}$ to start the backward calculation, while Processor2 cannot wait for $\alpha_{\mathbf{1},3}$ for its forward calculation. So instead, we use the value of such $\alpha_{p,t}(i)$ and $\beta_{p,t}(i)$ values of their **previous iteration** instead. For example, at this m^{th} iteration, we use the $\alpha_{p,t}(i)$ and $\beta_{p,t}(i)$ values that calculated already at $(m\text{-}1)^{th}$ iteration. Then, we calculate the new $\alpha_{p,t}(i)$ and $\beta_{p,t}(i)$ values for the $(m+1)^{th}$ iteration to use later.

2.3.2 Update of model parameters

For the parallel step, we need to update $\lambda = \{A, B, \Pi\}$ at each iteration. Π is trivial since we can just let the first processor to update it; For A and B, it is easy to replace (5) and (6)

with (5') and (6'), where $\gamma_{p,t}(i)$ $\xi_{p,t}(i,j)$ are the respective local parameters within the pth processor:

$$a_{ij}^{'} = \frac{\Sigma_{p=1}^{P}\Sigma_{t=1}^{T-1}\xi_{p,t}(i,j)}{\Sigma_{p=1}^{P}\Sigma_{j=0}^{N-1}\Sigma_{t=1}^{T-1}\xi_{p,t}(i,j)} \quad (5')$$

$$b_i(v_k)' = \frac{\Sigma_{p=1}^{P}\Sigma_{t=1,o_t=v_k}^{T}\gamma_{p,t}(i)}{\Sigma_{p=1}^{P}\Sigma_{t=1}^{T}\gamma_{p,t}(i)} \quad (6')$$

So now we have introduced the idea and implementation details of the parallel Baum-Welch algorithm. Table 1 lists the times of computation of important steps introduced above. Generally, parallel algorithm should reduce the time complexity from O(NT(N+M)) to O(NT(N+M)/P) where P is the number of processors.

One interesting thing I discover in the following experiment is that (4) takes much longer time (much more than theoretically 4x) than other operations. I think it might because of that $\xi_t(i,j)$ is stored using a 3d-array in my implementation, where others ($\alpha_t(i)$, $\beta_t(i)$ and $\gamma_t(i)$) use 2d-array. Higher dimension array should have worse locality thus run longer.

Table 1: Times of Computations of Sequential and Parallel Baum-Welch

	(1)	(2)	(3)	(4)	(5)	(6)
Sequential	$2(T-1)N^2$	$N+3(T-1)N^2$	N^2T	$4N^2T$	$2N^2T$	NMT
Parallel	$2(T-1)N^2/P$	$(N+3(T-1)N^2)/P$	N^2T/P	$4N^2T/P$	$2N^2T/P$	NMT/P

Another important point is, since the parallel algorithm is an approximation to the sequential algorithm, there is no guarantee that the data likelihood will not decrease over time. We will see some "counterexamples" in the following experiments.

3. Experiments

In this Experiment section I want to testify the effectiveness of the parallel algorithm. Two main questions are: how fast is the parallel algorithm (will be covered in Section 3.1), and how good result it can achieve, in this application, the data likelihood (Section 3.2).

I conduct all the experiments on the OpenCloud cluster in Parallel Data Lab in Carnegie Mellon University (http://www.pdl.cmu.edu/DISC/index.shtml)

OpenCloud: each node consists of 8 processors, each an Intel® Xeon® E5440 2.83GHz CPU. Each node is also equipped with 16GB RAM, 32KB L1 DCache, 32KB L1 ICache and 6144KB L2 cache. I use the openMP inside GCC 4.3.2 compiler to run the parallel algorithm.

Since I don't have real-world data for testing, I randomly generate some datasets for the experiments. For each synthetic dataset, I first set N, M and T, then randomly generate Observations $O = (o_1, o_2\ldots o_T)$ and also the initial value of parameters $\lambda = \{A, B, \Pi\}$. I make sure that the parameter is reasonable, e.g. π_i should add up to 1.0.

I generate two sets of input data, a **small** one with N=20, M=10 and T=256 and a **large** one with N=100, M=50 and T=1536.

3.1 Speedup of the parallel algorithm

In this subsection we discuss the running time of the parallel Baum-Welch algorithm.

Log-log scale in Figure4 indicates that, for small input data ("Actual 256" and "Ideal 256"), the speedup flats out after using 4 processors. This is because the running time on this small dataset is too small (several seconds for 250 iterations), so the overhead of using multiple threads cannot be neglected. Actually, for this small scale of data we might not even be interested in parallelize it.

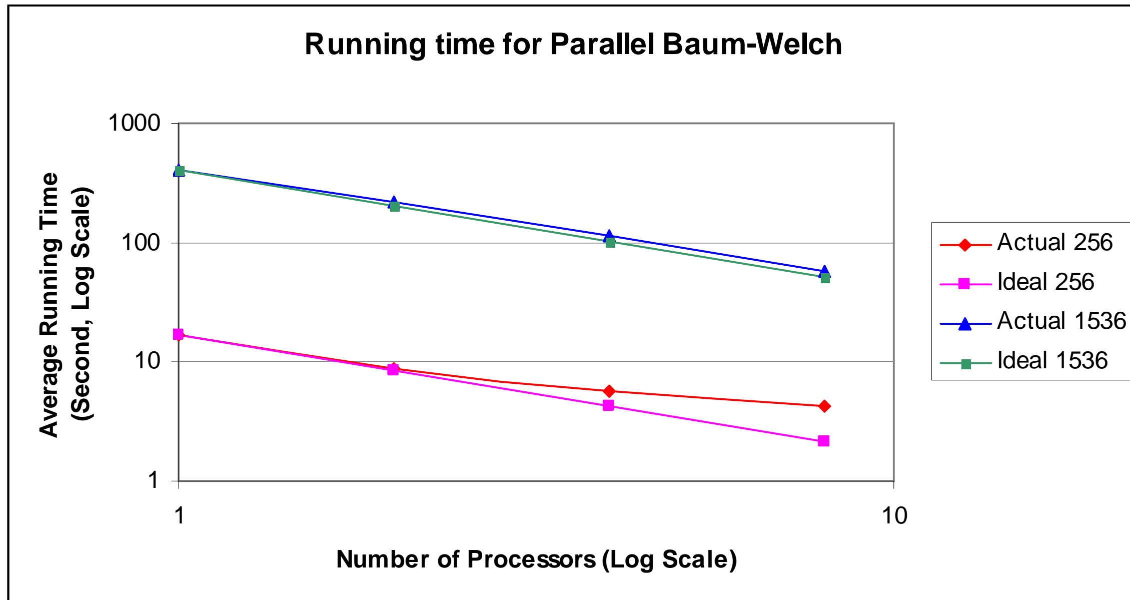

Figure4: Running time of the parallel Baum-Welch algorithm, varying the number of processors. This log-log graph shows fairly good speedup, especially for larger input data.

On the other side, for large input data ("Actual 1536" and "Ideal 1536"), the speedup is very promising even after using 8 processors. Since this input data is not very big (only takes around 400 seconds to finish 250 iterations), we envision the speedup will be very good for larger input data and more processors as well.

Another very interesting discovery is that, while looking at the speedup over processors (2 processors 1.84x, 4 processors 3.62x, 8 processors 7.20x for large input data), we see the speedup from 1 processor to 2 processors is **less** than from 2-4 and from 4-8. This is somehow counter-intuitive, since more processors usually indicate more communication conflict/overhead. But for our application, the communication is relatively small, so the overhead to start threads (1-2) stands out more.

3.2 Data Likelihood of the parallel algorithm

In order to evaluate the quality of the parallel algorithm, we use the Data Likelihood $P(o_1,o_2,\ldots,o_T|\ \lambda)$ since both the sequential and parallel algorithm try to maximize it. We may use some other metrics (e.g. the prediction accuracy of HMM) as well.

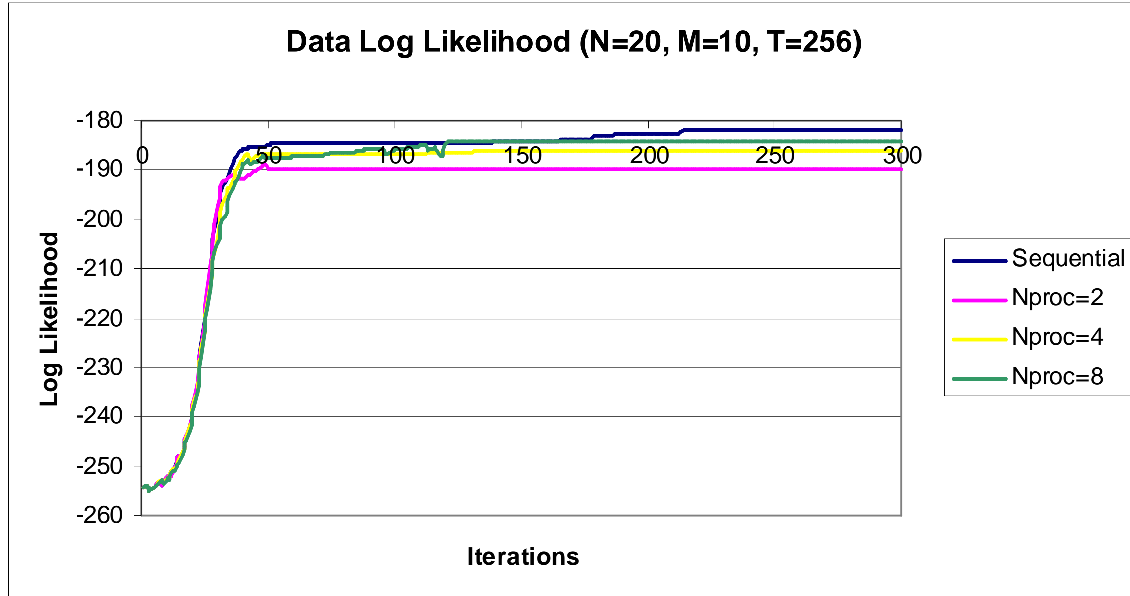

Figure5: Data likelihood of the small input data.

Figure5 and Figure 6 give the data log likelihood for two small datasets (with same M, N and T, different observations and initial model parameters). They behave somewhat different: In Figure5, sequential algorithm achieves the highest Log likelihood, and the parallel algorithms are 2%-5% worse; In Figure6, however, one parallel algorithm (4 processors), although using an approximation algorithm, achieves even higher data log likelihood. I'm not totally convinced of what's happening here, but it seems to me that since sequential algorithm will lead to a local maximum, it's possible that the sequential algorithm falls to a "worse" local maximum while the parallel algorithm falls to a "better" local maximum.

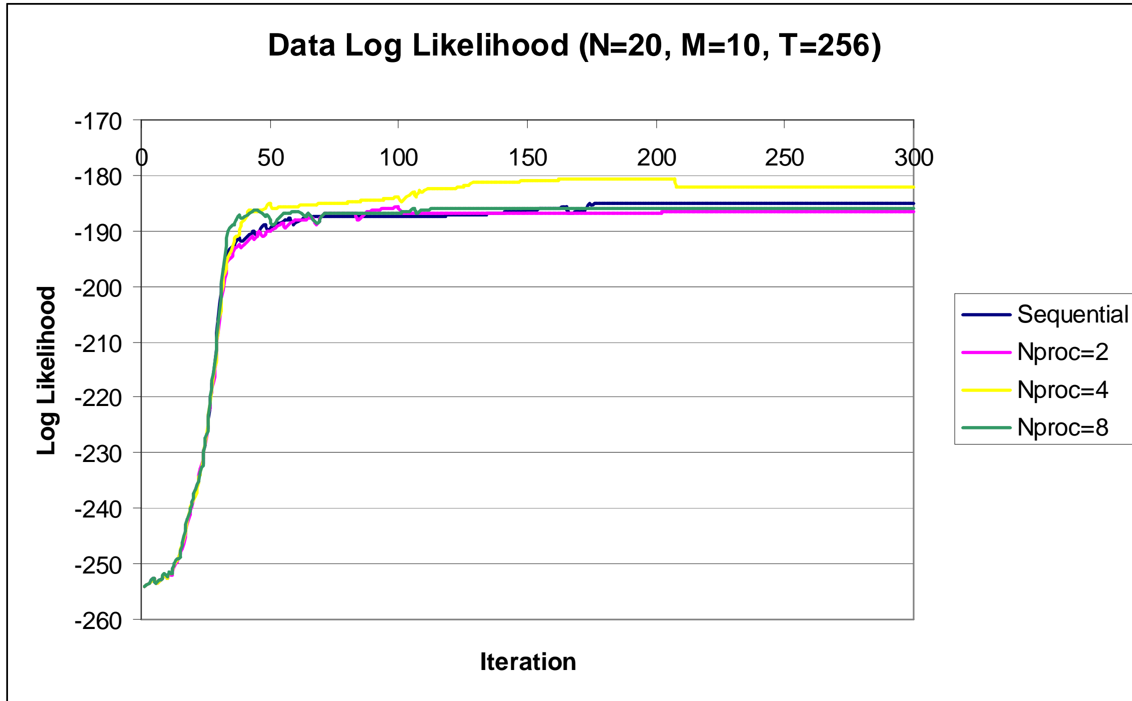

Figure6: Data likelihood of another small input data.

Another property is that sequential algorithm, since using an EM-like procedure, will have non-decreasing likelihood over time. Parallel algorithm however sometimes drops since it's an approximation to the sequential algorithm and has no such non-decreasing property.

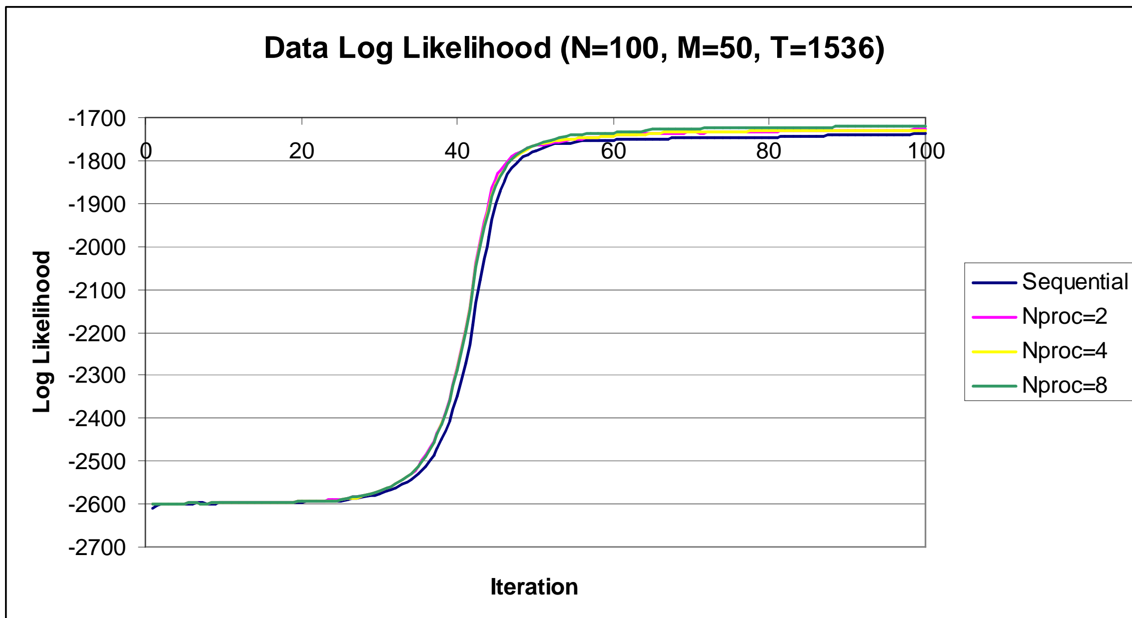

Figure7: Data likelihood of the large input data.

Figure7 shows the Data Log Likelihood for the large dataset. We see clearly that for this case, likelihood results using different number of processors are more closed together. The difference of log likelihood is smaller than 0.5% for all parallel algorithm results. Sequential algorithm doesn't achieve the highest likelihood again, but within a very small margin.

To conclude, experiments show that the parallel Baum-Welch algorithm has promising speedup to its sequential version, especially for large input data set. Although the parallel algorithm is just an approximation, it achieves similar (sometimes even higher) likelihood to the sequential algorithm.

4. Lessons learnt in the project

In this experiment I better understand how openMP works. Since previously I had some experience using Hadoop (Map/Reduce), now I think I can make some comparisons between them. I think openMP is easier to program, (should be) faster and more flexible. But Hadoop's advantage includes its fault tolerance, scalable when using many processors and optional out-of-core processing when necessary.

I also develop some habits which help me to write faster (rather than the most intuitive) programs. For example, I used ***t* = 1 to T** in sequential algorithm to indicate an index value. When I tried to convert the algorithm to a parallel one, since **T = P * s**, within the pth processor I wrote ***t* = 1 to s** and then use **(*p* * s + *t*)** to represent the same index. However, since each time an additional multiplication and addition are involved, the above operation takes around 20% of the total time. Finally I discovered the problem and improve to ***t* = *p* * s to *p* * s + s**.

Another example is the use of multi-dimensional array. Locality of the array is very important, which has been discussed in Table 1.

5. Conclusion and Future Works

In this project I parallelize the Baum-Welch algorithm of the Hidden Markov Model. I use a similar method as to the work at [Li, et al. 08]. Comparing to the sequential algorithm, same level of Data Likelihood is achieved using up to 8 threads. The parallel algorithm also exhibit very good speedup as well.

As [Li, et al. 08] pointing out at last, this similar parallel idea can be applied to other sequential machine learning models as well. If there is similar Forward-Backward iteration framework, this parallel idea can be even applied naturally, like Conditional Random Field (http://en.wikipedia.org/wiki/Conditional_random_field).

In terms of the future work, I plan to further complete this work, including more thorough test on real data and using more processors to speedup. I have looked at the CRF problems and have come up with the parallel ideas as well. I also hope to work on other machine learning models later.

Distribution of Total Credit: I did most of the work myself. I learnt a lot from the course slides from Roni Rosenfeld (at his Machine Learning class) at Language Technology Institute, Carnegie Mellon University. I also want to thank for Lei Li and Kai Ren in Computer Science Department, Carnegie Mellon University for their insights when I discussed the ideas about this project with them.

Reference

[Li, et al. 08] Lei Li, Wenjie Fu, Fan Guo, Todd C. Mowry, Christos Faloutsos. Cut-and-stitch: efficient parallel learning of linear dynamical systems on SMPs. KDD '08, Las Vegas, NV.

[Chu, et al. 06] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In B. Schölkopf, J. C. Platt, and T. Hoffman, editors, NIPS 19, pages 281–288. MIT Press, 2006.