PAPER

Efficient Parallel Learning of Hidden Markov Chain Models on SMPs

Lei LI[†], **Bin FU**[†], ***and*** **Christos FALOUTSOS**[†], ***Nonmembers***

SUMMARY Quad-core cpus have been a common desktop configuration for today's office. The increasing number of processors on a single chip opens new opportunity for parallel computing. Our goal is to make use of the multi-core as well as multi-processor architectures to speed up large-scale data mining algorithms. In this paper, we present a general parallel learning framework, *Cut-And-Stitch*, for training hidden Markov chain models. Particularly, we propose two model-specific variants, CAS-LDS for learning linear dynamical systems (LDS) and CAS-HMM for learning hidden Markov models (HMM). Our main contribution is a novel method to handle the data dependencies due to the chain structure of hidden variables, so as to parallelize the EM-based parameter learning algorithm. We implement CAS-LDS and CAS-HMM using OpenMP on two supercomputers and a quad-core commercial desktop. The experimental results show that parallel algorithms using *Cut-And-Stitch* achieve comparable accuracy and almost linear speedups over the traditional serial version.

key words: *Linear Dynamical Systems; Hidden Markov Models; OpenMP; Expectation Maximization (EM); Optimization; Multi-core.*

1. Introduction

Time series, no matter categorical or continuous valued, appear in numerous applications, including motion capture [1], visual tracking, speech recognition, quantitative studies of financial markets, network intrusion detection, bio-informatics etc. Mining and forecasting are popular operations relevant to time series analysis. Hidden Markov chain models, typically example including hidden Markov models (HMM) and linear dynamical systems (LDS, also known as Kalman filters), are often used to model those sequences and their generation process. Both assume linear transitions on hidden (i.e. 'latent') variables which are considered discrete for HMM and continuous for LDS. In this paper, we will focus on the parallelizing of learning algorithm for hidden Markov chain models. We will propose a general scheme and two model-specific variants CAS-LDS for the Linear Dynamical System, and CAS-HMM for Hidden Markov Model. We will focus on the message passing algorithm(computation of posterior marginal), which is the basic inference algorithm for both Bayesian Network and Markov Random Fields. Traditionally, learning those model parameters is difficult, requiring the well-known Expectation-Maximization (EM) method [2]. The EM algorithm for learning of LDS/HMM iterates between computing conditional expectations of hidden variables through the forward-backward belief propagation (E-step) and updating model parameters to maximize its likelihood (M-step). Although EM algorithm generally produces good results, the EM iterations may take long to converge. Meanwhile, the computation time of E-step is linear in the length of the time series but super linear in the dimensionality of observations and hidden variable, which results in poor scaling on high dimensional data. For example, our experimental results show that on a 93-dimensional dataset of length over 300, the EM algorithm for LDS would take over one second to compute each iteration and over ten minutes to converge on a high-end multi-core commercial computer. Such capacity may not be able to fit modern computation-intensive applications with large amounts of data or real-time constraints. While there are efforts to speed up the forward-backward procedure with moderate assumptions such as sparsity or existence of low-dimensional approximation, we will focus on taking advantage of the quickly developing parallel processing technologies to achieve dramatic speedup.

Traditionally, the EM algorithm running on a multi-core computer only takes up a single core with limited processing power, and the current state-of-the-art dynamic parallelization techniques such as speculative execution [3] benefit little to the straightforward EM algorithm due to the nontrivial data dependencies in Markov chain models. As the number of cores on a single chip keeps increasing, soon we may be able to build machines with even a thousand cores, e.g. an energy efficient, 80-core chip not much larger than the size of a finger nail was released by Intel researchers in early 2007 [4]. Earlier this year (2010), Intel developed another chip that consists of 48 Pentium-class IA-32 cores which supports fast intercore communication [5]. This paper is along the line to investigate the following question: how much speed up could we obtain for machine learning algorithms on multi-core? There are already several papers on distributed computation for data mining operations. For example, "cascade SVMs" were proposed to parallelize Support Vector Machines [6]. Other articles use Google's map-reduce techniques [7] on multi-core machines to design efficient parallel learning algorithms for a set of standard machine learning algorithms/models such as naïve Bayes and PCA, achieving almost linear speedup [8, 9]. However, these methods do not apply to HMM or LDS directly. In essence, their techniques are similar to dot-product-like parallelism, by using divide-and-conquer on independent sub models; these do not work for models with complicated data dependencies such as HMM and LDS. *

[†]The authors are with Computer Science Department of Carnegie Mellon University.

DOI: 10.1587/trans.E0.??.1

*Or exactly, models with large diameters. The diameter of a

Table 1 Symbols and annotations.

(a) LDS

Symbol	Definition
$\mathbf{Y}$	a multi-dimensional observation sequence
$\mathbf{Z}$	the hidden variables ($= \{\mathbf{z}_1, \ldots, \mathbf{z}_\mathrm{N}\}$)
m	the dimension of the observation sequence
H	the dimension of hidden variables
N	the duration of the observation
$\mathbf{F}$	the transition matrix, $H \times H$
$\mathbf{G}$	the project matrix from hidden to observation, $m \times H$

(b) HMM

Symbol	Definition
$\mathbf{Y}$	observation sequence ($= \{\mathbf{y}_1, \ldots, \mathbf{y}_\mathrm{N}\}$)
$\mathbf{Z}$	the hidden variables ($= \{\mathbf{z}_1, \ldots, \mathbf{z}_\mathrm{N}\}$)
N	the duration of the observation
$\mathbf{M}$	number of discrete values of observation variables
$\mathbf{V}$	possible values for observation variables ($= \{\mathbf{v}_1, \ldots, \mathbf{v}_M\}$)
$\mathbf{K}$	number of discrete values of hidden variables
$\mathbf{S}$	possible values for hidden variables ($= \{\mathbf{s}_1, \ldots, \mathbf{s}_K\}$)
$\mathbf{A}$	the transition matrix, $K \times K$
$\mathbf{B}$	the project matrix from hidden to observation, $K \times M$
Π	the initialization vector, ($= \{\pi_1, \ldots, \pi_K\}$)

In this paper, we extend our earlier work [10] on the *Cut-And-Stitch* method (CAS), which avoids the data-dependency problems. We propose algorithms for quickly and accurately learning LDS and HMM in parallel, and demonstrate on two popular architectures for high performance computing. The basic idea of our algorithms is to (a) *Cut* both the chain of hidden variables as well as the observed variables into smaller blocks, (b) perform intra-block computation, and (c) *Stitch* the local results seamlessly by summarizing sufficient statistics and updating model parameters and an additional set of block-specific parameters. The algorithm (CAS-LDS and CAS-HMM) will iterate over 4 steps, where the most time-consuming E-step in EM as well as the two newly introduced steps could be parallelized with little synchronization overhead. Furthermore, this approximation of global models by local sub-models sacrifices only a little accuracy, due to the structure of hidden Markov chain models, as shown in our experiments, which was our first goal. On the other hand, it yields almost linear speedup, which was our second main goal.

The rest of the paper is organized as follows. We first describe the Linear Dynamical System in Section 2 and present our proposed *Cut-And-Stitch* method in Section 3. Then we describe the programming interface and implementation issues in Section 4. We present experimental results Section 5, the related work in Section 6, and our conclusions in Section 7.

2. Background

Markov chain models are often used in capturing the temporal behavior of a system. Hidden Markov chain models are

model is the length of longest acyclic path in its graphical representation. For example, the diameter of the LDS in Figure 1 is N.

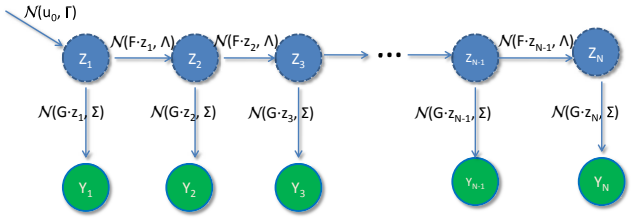


Fig. 1 A Graphical Representation of the Linear Dynamical System: $\mathbf{z}_1, \ldots, \mathbf{z}_\mathrm{N}$ indicate hidden variables; $\mathbf{y}_1, \ldots, \mathbf{y}_\mathrm{N}$ indicate observation. Arrows indicate Linear Gaussian conditional probabilistic distributions.

those with random variables unobserved in Markov chains. Both linear dynamical systems (LDS) and hidden Markov models (HMM) fall into this framework. The chain of hidden variables, for example, could represent the (unknown) functions of a genetic sequences modeled by HMM, or velocities and accelerations of rockets by Kalman filters. In the following, we will first introduce the general framework of hidden Markov chain models, and describe traditional algorithms for learning those models respectively. Table 1 lists the symbols and annotations used in both LDS and HMM.

In hidden Markov chain models, a sequence of observations $\mathbf{Y}$ ($= \vec{y}_1, \ldots, \mathbf{y}_\mathrm{N}$) are drawn from an emission probability distribution $P(\mathbf{y}_n|\mathbf{z}_n)$, and hidden variables $\mathbf{z}_n$ from a Markov chain with the transition probability distribution $P(\mathbf{z}_{n+1})|\mathbf{z}_n$. The joint pdf of the model is as follows:

$$P(\mathbf{Y}, \mathbf{Z}) = P(\mathbf{z}_1) \prod_{n=2}^{\mathrm{N}} P(\mathbf{z}_{n+1}|\mathbf{z}_n) \prod_{n=1} P(\mathbf{y}_n|\mathbf{z}_n) \quad (1)$$

2.1 Linear Dynamical Systems

In LDS, both the transitions among the hidden variables as well as their projections to the observations are described as linear Gaussian models (Eq (3-3)). We denote them as a matrix $\mathbf{F}$ for the transition ($H \times H$) with noises $\{\omega_n\}$; and a matrix $\mathbf{G}$ ($m \times H$) for the projection with the noises $\{\epsilon_n\}$ at each time-tick n. Figure 1 provides the graphical representation of following equations defining an LDS:

$$P(\mathbf{z}_1) = \mathcal{N}(z_0, \Gamma) \quad (2)$$

$$P(\mathbf{z}_{n+1}|\mathbf{z}_n) = \mathcal{N}(\mathbf{F} \cdot \mathbf{z}_n, \Lambda) \quad (3)$$

$$P(\mathbf{y}_n|\mathbf{z}_n) = \mathcal{N}(\mathbf{G} \cdot \mathbf{z}_n, \Sigma) \quad (4)$$

where z_0 is the initial state of the whole system.

Given the observation sequence, the goal of the learning algorithm is to compute the optimal parameter set $\theta = (\mu_0, \Gamma, F, \Lambda, G, \Sigma)$. The optimum is obtained by maximizing the log-likelihood $l(\mathcal{Y}; \theta)$ over the parameter set θ. As mentioned in Section 1, the typical learning method for LDS is the EM algorithm [2], which iteratively maximizes the expected complete log-likelihood in a coordinate-ascent manner:

$$Q(\theta^{new}, \theta^{old}) = \mathbb{E}_{\theta^{old}}[\log p(\mathbf{y}_1 \ldots \mathbf{y}_N, \mathbf{z}_1 \ldots \mathbf{z}_N|\theta^{new})]$$

In brief, the algorithm first guesses an initial set of model

parameters θ_0. Then, at each iteration, it uses a forward-backward algorithm to compute expectations of the hidden variables $\hat{\mathbf{z}}_n = \mathbb{E}[\mathbf{z}_n \mid \mathcal{Y}; \theta_0]$ $(n = 1, \ldots, \mathrm{N})$ as well as the second moments and covariance terms, which is the E-step. In the M-step, it maximizes the expected complete log-likelihood of $\mathbb{E}[L(\mathcal{Y}, \mathbf{z}_{1\ldots\mathrm{N}})]$ with respect to the model parameters. Since the computation of $\mathbb{E}[\mathbf{z}_n \mid \mathcal{Y}]$ depends on $\mathbb{E}[\mathbf{z}_{n-1} \mid \mathcal{Y}]$ and $\mathbb{E}[\mathbf{z}_{n+1} \mid \mathcal{Y}]$, the straightforward implementation of the EM algorithm can not exploit much instruction level parallelism.

2.2 Hidden Markov Model

Hidden Markov Model (HMM) shares the same graphical model as LDS. However, hidden variables $\mathcal{Z}$ for HMM are discrete and the transitions between them follow the multinomial distributions. The observation $\mathcal{Y}$ can be either discrete or continuous. We will describe the discrete case, however, the learning algorithm is similar.

Assume each observation variable $\mathbf{y}_n$ of a Hidden Markov Model has M possible values $(v_1, v_2, ..., v_M)$, each hidden variable $\mathbf{z}_n$ has K possible values $(s_1, s_2, ..., s_K)$. Then parameter set of the HMM λ includes the transition matrix $\mathbf{A}_{pq}$ $(K \times K)$, observation matrix $\mathbf{B}_p(v_r)$ $(K \times M)$ and initialization vector π_i $(K \times 1)$. The data in the model flows according to the subsequent equations:

$$P(\mathbf{z}_1 = s_p) = \pi_p \tag{5}$$

$$P(\mathbf{z}_n = s_q | \mathbf{z}_{n-1} = s_p) = \mathbf{A}_{pq} \tag{6}$$

$$P(\mathbf{y}_n = v_r | \mathbf{z}_n = s_p) = \mathbf{B}_p(v_r) \tag{7}$$

The training problem for HMM is as follows: given observations $\mathcal{Y}$, find an optimal λ that maximize the data likelihood. With no tractable direct solution, Training problem can be solved by an EM algorithm as well, which specifically is known as the Baum-Welch algorithm [11].

3. Cut-And-Stitch: Proposed Method

The traditional learning algorithms for both LDS and HMM are through expectation-maximization, where in the E-step, the algorithms run forward and backward to obtain estimation of the posterior distribution of the hidden variables ($\mathbf{z}$'s). The chain structure of both model enforces the data dependencies in both the forward computation from $\mathbf{z}_n$ (e.g. $\mathbb{E}[\mathbf{z}_n \mid \mathcal{Y}; \theta]$) to $\mathbf{z}_{n+1}$ and the backward computation from $\mathbf{z}_{n+1}$ to $\mathbf{z}_n$ In this section, we will present general ideas on overcoming such dependencies and describe the details of *Cut-And-Stitch* parallel learning algorithm, for both LDS and HMM respectively.

3.1 Intuition and General Scheme

Our guiding principle to reduce the data dependencies is to divide the hidden chain of variables into smaller, independent parts. Given a data sequence $\mathcal{Y}$ and k processors with shared memory, we could cut the sequence into k subsequences of equal sizes, and then assign one processor to each subsequence. Each processor will learn the individual parameters, $\theta_1, \ldots, \theta_k$ in the case of LDS, associated with its subsequence, using the basic, sequential EM algorithm. In order to obtain a consistent set of parameters for the whole sequence, we use a non-trivial method to summarize all the sub-models rather than simply averaging. Since each subsequence is treated independently, our algorithm will obtain near k-fold speedup. The main design challenges are: (a) how to minimize the overhead in synchronization and summarization, and (b) how to retain the accuracy of the learning algorithm. Our *Cut-And-Stitch* method (or CAS) is targeting both challenges, in two different models.

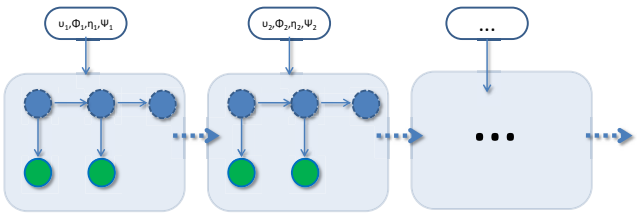


Fig. 2 Graphical illustration of dividing LDS into blocks in the *Cut* step. Note *Cut* introduces additional parameters for each block.

Given a sequence of observed values $\mathcal{Y}$ with length of N, the learning goal is to best fit the parameters: $\theta = (\mu_0, \Gamma, F, \Lambda, G, \Sigma)$ for LDS and $\lambda = (\pi, \mathbf{A}, \mathbf{B})$ for HMM. In general, the Cut-And-Stitch (CAS) algorithm consists of two alternating steps: the *Cut* step and the *Stitch* step. In the *Cut* step, the Markov chain of hidden variables and corresponding observations are divided into smaller blocks, and each processor performs the local computation for each block. More importantly, it computes the initial beliefs (marginal expectation of hidden variables) for its block, based on the neighboring blocks, and then it computes the improved beliefs for its block, independently. In the *Stitch* step, each processor computes summary statistics for its block, and then the parameters of the model are updated globally to maximize the EM learning objective function (also known as the *expected complete log-likelihood*). Besides, local parameters for each block are also updated to reflect changes in the global model. The CAS algorithm iterates between *Cut* and *Stitch* until convergence.

3.2 CAS-LDS

3.2.1 Cut step

The objective of *Cut* step is to compute the marginal posterior distribution of $\mathbf{z}_n$, conditioned on the observations $\mathbf{y}_1, \ldots, \mathbf{y}_\mathrm{N}$ given the current estimated parameter θ: $P(\mathbf{z}_n | \mathbf{y}_1, \ldots, \mathbf{y}_\mathrm{N}; \theta)$. Given the number of processors k and the observation sequence, we first divide the hidden Markov chain into k blocks: $B_1, \ldots, B_k$, with each block containing the hidden variables $\mathbf{z}$, the observations $\mathbf{y}$, and four extra parameters υ, Φ, η, Ψ. The sub-model for i-th block B_i is described as follows (see Figure 2):

$$P(\mathbf{z}_{i,1}) = \mathcal{N}(\upsilon_i, \Phi_i) \tag{8}$$

$$P(\mathbf{z}_{i,j+1}|\mathbf{z}_{i,j}) = \mathcal{N}(\mathbf{F}\mathbf{z}_{i,j}, \Lambda) \quad (9)$$
$$P(\mathbf{z}'_{i,T}|\mathbf{z}_{i,T}) = \mathcal{N}(\mathbf{F}\mathbf{z}_{i,T}, \Lambda) \quad (10)$$
$$P(\mathbf{y}_{i,j}|\mathbf{z}_{i,j}) = \mathcal{N}(\mathbf{G}\mathbf{z}_{i,j}, \Sigma) \quad (11)$$

where the block size $T = \frac{\mathrm{N}}{k}$ and $j = 1 \dots T$ indicating j-th variables in i-th block ($\mathbf{z}_{i,j} = \mathbf{z}_{(i-1)*T+j}$ and $\mathbf{y}_{i,j} = \mathbf{y}_{(i-1)*T+j}$). η_i, Ψ_i could be viewed as messages passed from next block, through the introduction of an extra hidden variable $\mathbf{z}'_{i,T}$.

$$P(\mathbf{z}'_{i,T}) = \mathcal{N}(\eta_i, \Psi_i) \quad (12)$$

Intuitively, the *Cut* tries to approximate the global LDS model by local sub-models, and then compute the marginal posterior with the sub-models. The blocks are both logical and computational, meaning that most computation about each logical block resides on one processor. In order to simultaneously and accurately compute all blocks on each processor, the block parameters should be well chosen with respect to the other blocks. We will describe the parameter estimation later but here we first describe the criteria. From the Markov properties of the LDS model, the marginal posterior of $\mathbf{z}_{i,j}$ conditioned on $\mathcal{Y}$ is independent of any observed $\mathbf{y}$ outside the block B_i, as long as the block parameters satisfy:

$$P(\mathbf{z}_{i,1}|\mathbf{y}_1, \dots, \mathbf{y}_{i-1,T}) = \mathcal{N}(\upsilon_i, \Phi_i) \quad (13)$$
$$P(\mathbf{z}_{i+1,1}|\mathbf{y}_1, \dots, \mathbf{y}_{\mathrm{N}}) = \mathcal{N}(\eta_i, \Psi_i) \quad (14)$$

Therefore, we could derive a local belief propagation algorithm to compute the marginal posterior $P(\mathbf{z}_{i,j}|\mathbf{y}_{i,1} \dots \mathbf{y}_{i,T}; \upsilon_i, \Phi_i, \eta_i, \Psi_i, \theta)$. Both computation for the forward passing and the backward passing can reside in one processor without interfering with other processors except possibly in the beginning. The local forward pass computes the posterior up to current time tick within one block $P(\mathbf{z}_{i,j}|\mathbf{y}_{i,1} \dots \mathbf{y}_{i,j})$, while the local backward pass calculates the whole posterior $P(\mathbf{z}_{i,j}|\mathbf{y}_{i,1} \dots \mathbf{y}_{i,T})$ (to save space, we omit the parameters). Using the properties of linear Gaussian conditional distribution and Markov properties (Chap.2 &8 in [2]), one can easily infer that both posteriors are Gaussian distributions, denoted as:

$$P(\mathbf{z}_{i,j}|\mathbf{y}_{i,1} \dots \mathbf{y}_{i,j}) = \mathcal{N}(\mu_{i,j}, \mathbf{V}_{i,j}) \quad (15)$$
$$P(\mathbf{z}_{i,j}|\mathbf{y}_{i,1} \dots \mathbf{y}_{i,T}) = \mathcal{N}(\hat{\mu}_{i,j}, \hat{\mathbf{V}}_{i,j}) \quad (16)$$

We can obtain the following forward-backward propagation equations from Eq (8-12) by substituting Eq (13-16) and expanding.

$$\mathbf{P}_{i,j-1} = \mathbf{F}\mathbf{V}_{i,j-1}\mathbf{F}^T + \Lambda \quad (17)$$
$$\mathbf{K}_{i,j} = \mathbf{P}_{i,j-1}\mathbf{G}^T(\mathbf{G}\mathbf{P}_{i,j-1}\mathbf{G}^T + \Sigma)^{-1} \quad (18)$$
$$\mu_{i,j} = \mathbf{F}\mu_{i,j-1} + \mathbf{K}_{i,j}(\mathbf{y}_{i,j} - \mathbf{G}\mathbf{F}\mu_{i,j-1}) \quad (19)$$
$$\mathbf{V}_{i,j} = (\mathbf{I} - \mathbf{K}_{i,j})\mathbf{P}_{i,j-1} \quad (20)$$

The initial values are given by:

$$\mathbf{K}_{i,1} = \mathbf{\Phi}_i\mathbf{G}^T(\mathbf{G}\mathbf{\Phi}_i\mathbf{G}^T + \Sigma)^{-1} \quad (21)$$
$$\mu_{i,1} = \upsilon_i + \mathbf{K}_{i,1}(\mathbf{y}_{i,1} - \mathbf{G}\upsilon_i) \quad (22)$$
$$\mathbf{V}_{i,1} = (\mathbf{I} - \mathbf{K}_{i,1})\mathbf{\Phi}_i \quad (23)$$

The backward passing equations are:

$$\mathbf{J}_{i,j} = \mathbf{V}_{i,j}\mathbf{F}^T(\mathbf{P}_{i,j})^{-1} \quad (24)$$
$$\hat{\mu}_{i,j} = \mu_{i,j} + \mathbf{J}_{i,j}(\hat{\mu}_{i,j+1} - \mathbf{F}\mu_{i,j}) \quad (25)$$
$$\hat{\mathbf{V}}_{i,j} = \mathbf{V}_{i,j} + \mathbf{J}_{i,j}(\hat{\mathbf{V}}_{i,j+1} - \mathbf{P}_{i,j})\mathbf{J}_{i,j}^T \quad (26)$$

The initial values are given by:

$$\mathbf{J}_{i,T} = \mathbf{V}_{i,T}\mathbf{F}^T(\mathbf{F}\mathbf{V}_{i,T}\mathbf{F}^T + \Lambda)^{-1} \quad (27)$$
$$\hat{\mu}_{i,T} = \mu_{i,T} + \mathbf{J}_{i,T}(\eta_i - \mathbf{F}\mu_{i,T}) \quad (28)$$
$$\hat{\mathbf{V}}_{i,T} = \mathbf{V}_{i,T} + \mathbf{J}_{i,T}(\Psi_i - \mathbf{F}\mathbf{V}_{i,T}\mathbf{F}^T - \Lambda)\mathbf{J}_{i,T}^T \quad (29)$$

Except for the last block:

$$\hat{\mu}_{k,T} = \mu_{i,T} \qquad \hat{\mathbf{V}}_{k,T} = \mathbf{V}_{i,T} \quad (30)$$

3.2.2 Stitch step

In the *Stitch* step, we estimate the block parameters, collect the statistics and compute the most suitable LDS parameters for the whole sequence. The parameters $\theta = (\mu_0, \Gamma, F, \Lambda, G, \Sigma)$ is updated by maximizing over the expected complete log-likelihood function:

$$Q(\theta^{new}, \theta^{old}) = \mathbb{E}_{\theta^{old}}[\log p(\mathbf{y}_1 \dots \mathbf{y}_N, \mathbf{z}_1 \dots \mathbf{z}_N|\theta^{new})] \quad (31)$$

Now taking the derivatives of Eq 31 and zeroing out give the updating equations (Eq (38-43)). The maximization is similar to the M-step in EM algorithm of LDS, except that it should be computed in a distributed manner with the available k processors. The solution depends on the statistics over the hidden variables, which are easy to compute from the forward-backward propagation described in *Cut*.

$$\mathbb{E}[\mathbf{z}_{i,j}] = \hat{\mu}_{i,j} \quad (32)$$
$$\mathbb{E}[\mathbf{z}_{i,j}\mathbf{z}_{i,j-1}^T] = \mathbf{J}_{i,j-1}\hat{\mathbf{V}}_{i,j} + \hat{\mu}_{i,j}\hat{\mu}_{i,j-1}^T \quad (33)$$
$$\mathbb{E}[\mathbf{z}_{i,j}\mathbf{z}_{i,j}^T] = \hat{\mathbf{V}}_{i,j} + \hat{\mu}_{i,j}\hat{\mu}_{i,j}^T \quad (34)$$

where the expectations are taken over the posterior marginal distribution $p(\mathbf{z}_n|\mathbf{y}_1, \dots, \mathbf{y}_N)$. The next step is to collect the sufficient statistics of each block on every processor.

$$\tau_i = \sum_{j=1}^{T} y_{i,j}\mathbb{E}[\mathbf{z}_{i,j}^T] \quad (35)$$

$$\xi_i = \mathbb{E}[\mathbf{z}_{i,1}\mathbf{z}_{i-1,T}^T] + \sum_{j=2}^{T} \mathbb{E}[\mathbf{z}_{i,j}\mathbf{z}_{i,j-1}^T] \quad (36)$$

$$\zeta_i = \sum_{j=1}^{T} \mathbb{E}[\mathbf{z}_{i,j}\mathbf{z}_{i,j}^T] \quad (37)$$

To ensure its correct execution, statistics collecting should be run after all of the processors finish their *Cut* step, enabled through the *synchronization* among processors. With

the local statistics for each block,

$$\mu_0^{new} = \hat{\mu}_{1,1} \quad (38)$$

$$\mathbf{\Gamma}_0^{new} = \hat{\mathbf{V}}_{1,1} \quad (39)$$

$$\mathbf{F}^{new} = \left(\sum_{i=1}^{k}\xi_i\right)\left(\sum_{i=1}^{k}\zeta_i - \mathbb{E}[\mathbf{z}_N\mathbf{z}_N^T]\right)^{-1} \quad (40)$$

$$\mathbf{\Lambda}^{new} = \frac{1}{N-1}\Bigg(\sum_{i=1}^{k}(\zeta_i - \mathbf{F}^{new}\xi_i^T - \xi_i(\mathbf{F}^{new})^T) + \mathbf{F}^{new}(\sum_{i=1}^{k}\zeta_i - \mathbb{E}[\mathbf{z}_N\mathbf{z}_N^T])(\mathbf{F}^{new})^T - \mathbb{E}[\mathbf{z}_{1,1}\mathbf{z}_{1,1}^T]\Bigg) \quad (41)$$

$$\mathbf{G}^{new} = \left(\sum_{i=1}^{k}\tau_i\right)\left(\sum_{i=1}^{k}\zeta_i\right)^{-1} \quad (42)$$

$$\mathbf{\Sigma}^{new} = \frac{1}{N}\Bigg(Cov(\mathcal{Y}) + \sum_{i=1}^{k}(-\mathbf{G}^{new}\tau_i^T - \tau_i(\mathbf{G}^{new})^T + \mathbf{G}^{new}\zeta_i(\mathbf{G}^{new})^T)\Bigg) \quad (43)$$

where $Cov(\mathcal{Y})$ is the covariance of the observation sequences and could be precomputed.

$$Cov(\mathcal{Y}) = \sum_{n=1}^{N}\mathbf{y}_n\mathbf{y}_n^T$$

As we estimate the block parameters with the messages from the neighboring blocks, we could reconnect the blocks. Recall the conditions in Eq (13-14), we could approximately estimate the block parameters with the following equations.

$$\upsilon_i = \mathbf{F}\mu_{i-1,T} \quad (44)$$

$$\Phi_i = \mathbf{F}\mathbf{V}_{i,T}\mathbf{F}^T + \Lambda \quad (45)$$

$$\eta_i = \hat{\mu}_{i+1,1} \quad (46)$$

$$\Psi_i = \hat{\mathbf{V}}_{i+1,1} \quad (47)$$

Except for the first block (no need to compute η_k and Ψ_k for the last block):

$$\upsilon_1 = \mu_0 \qquad \Phi_1 = \Gamma \quad (48)$$

To extract the most parallelism, any of the above equations independent of each other could be computed in parallel. Computation of the local statistics in Eq (35-37) is done in parallel on k processors. Until all local statistics are computed, we use one processor to calculate the parameter using Eq (38-43). Upon the completion of computing the model parameters, every processor computes its own block parameters in Eq (44-48). To ensure the correct execution, *Stitch* step should run after all of the processors finish their *Cut* step, which is enabled through the synchronization among processors. Furthermore, we also use synchronization to ensure *Maximization* part after *Collecting* and *Re-estimate* after *Maximization*. An interesting finding is that our method includes the sequential version of the learning algorithm as a special case. Note if the number of processors is 1, the *Cut-And-Stitch* algorithm falls back to the conventional EM algorithm sequentially running on single processor.

3.3 CAS-HMM

In order to solve the training problem in Hidden Markov Model, we used EM algorithm to iteratively update the parameter set λ. At each iteration, we calculate $\alpha_n(p)$ from $\mathbf{z}_1$ to $\mathbf{z}_N$ (forward computation), and $\beta_n(p)$ from $\mathbf{z}_N$ back to $\mathbf{z}_1$ (backward computation). We also define auxiliary variables $\gamma_n(p)$ and $\xi_n(p,q)$ to help us update the HMM model. The definitions of α, β, γ and ξ are shown from Eq (49-52).

$$\alpha_n(p) = P(\mathbf{y}_1, ...\mathbf{y}_n, \mathbf{z}_n = s_p|\lambda) \quad (49)$$

$$\beta_n(p) = P(\mathbf{y}_{n+1}, ..., \mathbf{y}_N|\mathbf{z}_n = s_p, \lambda) \quad (50)$$

$$\gamma_n(p) = P(\mathbf{z}_n = s_p|\mathbf{y}_1, ..., \mathbf{y}_N, \lambda) \quad (51)$$

$$\xi_n(p,q) = P(\mathbf{z}_n = s_p, \mathbf{z}_{n+1} = s_q|\mathbf{y}_1, ..., \mathbf{y}_N, \lambda) \quad (52)$$

In the *Cut* step with k processors, we again split the Hidden Markov Model into k blocks: $B_1, ..., B_k$. We still use the notation $\mathbf{z}_{i,j}$ and $\mathbf{y}_{i,j}$ to indicate j-th variable in the i-th block. Same notations also apply to other intermediate variables like $\alpha_{i,j}(p)$. In order to propagate information between adjacent blocks, we define two set of parameters $\delta_i(p)$ and $\kappa_i(p)$ for each block B_i, where:

$$\delta_i(p) = \alpha_{i-1,T}(p) \quad (53)$$

$$\kappa_i(p) = \beta_{i+1,1}(p) \quad (54)$$

Then local HMM blocks can update themselves according to Eq (55-63).

$$\alpha_{1,1}(p) = \pi_p\mathbf{B}_p(\mathbf{y}_{1,1}) \quad (55)$$

$$\alpha_{i,1}(p) = \mathbf{B}_p(\mathbf{y}_{i,1})\Sigma_{q=1}^{K}\delta_i(q)\mathbf{A}_{qp} \quad (56)$$

$$\alpha_{i,j}(p) = \mathbf{B}_p(\mathbf{y}_{i,j})\Sigma_{q=1}^{K}\alpha_{i,j-1}(q)\mathbf{A}_{qp} \quad (57)$$

$$\beta_{k,T}(p) = 1 \quad (58)$$

$$\beta_{i,T}(p) = \Sigma_{q=1}^{K}\kappa_i(q)\mathbf{A}_{pq}\mathbf{B}_q(\mathbf{y}_{i+1,1}) \quad (59)$$

$$\beta_{i,j}(p) = \Sigma_{q=1}^{K}\beta_{i,j+1}(q)\mathbf{A}_{pq}\mathbf{B}_q(\mathbf{y}_{i,j+1}) \quad (60)$$

$$\gamma_{i,j}(p) = \frac{\alpha_{i,j}(p)\beta_{i,j}(p)}{\Sigma_{q=1}^{K}\alpha_{i,j}(q)\beta_{i,j}(q)} \quad (61)$$

$$\xi_{i,j}(p,q) = \frac{\gamma_{i,j}(p)\mathbf{A}_{pq}\mathbf{B}_q(\mathbf{y}_{i,j+1})\beta_{i,j+1}(q)}{\beta_{i,j}(p)} \quad (62)$$

$$\xi_{i,T}(p,q) = \frac{\gamma_{i,T}(p)\mathbf{A}_{pq}\mathbf{B}_q(\mathbf{y}_{i+1,1})\kappa_i(q)}{\beta_{i,T}(p)} \quad (63)$$

In the *Stitch* step, each block $\mathbf{B}_i$ first collect necessary statistics:

$$\tau_i(p,q) = \Sigma_{j=1}^{T}\xi_{i,j}(p,q) \quad (64)$$

$$\zeta_i(p,q) = \Sigma_{l=1}^{K}\Sigma_{j=1}^{T}\xi_{i,j}(p,l) \quad (65)$$

$$\eta_i(p,v_r) = \Sigma_{j=1,\mathbf{y}_{i,j}=v_r}^{T}\gamma_{i,j}(p) \quad (66)$$

$$\varphi_i(p,v_r) = \Sigma_{j=1}^{T}\gamma_{i,j}(p) \quad (67)$$

Except for the last block:

$$\tau_k(p,q) = \Sigma_{j=1}^{T-1}\xi_{k,j}(p,q) \quad (68)$$

$$\zeta_k(p,q) = \Sigma_{l=1}^{K}\Sigma_{j=1}^{T-1}\xi_{k,j}(p,l) \quad (69)$$

Subsequently, all blocks work together to update HMM model λ at Eq (70-72). δ_i and κ_i are also updated here according to Eq (53-54).

$$\pi_p^{new} = \gamma_{1,1}(p) \quad (70)$$

$$\mathbf{A}_{pq}^{new} = \frac{\Sigma_{i=1}^{k}\tau_i(p,q)}{\Sigma_{i=1}^{k}\zeta_i(p,q)} \quad (71)$$

$$\mathbf{B}_p(v_r)^{new} = \frac{\Sigma_{i=1}^{k}\eta_i(p,v_r)}{\Sigma_{i=1}^{k}\varphi_i(p,v_r)} \quad (72)$$

3.4 Warm-Up Step

In the first iteration of the algorithm, there are undefined initial values of block parameters υ,Φ,η and Ψ, needed by the forward and backward propagations in *Cut*. A simple approach would be to assign random initial values, but this may lead to poor performance. We propose and use an alternative method: we run a sequential forward-backward pass on the whole observation, estimate parameters, i.e. we execute the *Cut* step with one processor, and the *Stitch* step with k processors. After that, we begin normal iterations of *Cut-And-Stitch* with k processors. We refer to this step as the *warm-up* step. Although we sacrifice some speedup, the resulting method converges faster and is more accurate. Figure 3 illustrates the time line of the whole algorithm on four CPUs.

In summary, the *Cut-And-Stitch* algorithms (CAS-LDS and CAS-HMM) work in the following two steps, which could be further divided into four sub-steps:

Cut divides and builds small sub-models (blocks), and then each processor *estimate* (E) in parallel posterior marginal distribution in Eq (32-34) and Eq (55-63), which includes *forward* and *backward* propagation of beliefs.

Stitch estimates the parameters through *collecting* (C) local statistics of hidden variables in each block Eq (35-37)and Eq (64-69), taking the *maximization* (M) of the expected log-likelihood over the parameters Eq (38-43) and Eq (70-72), and connecting the blocks by *re-estimate* (R) the block parameters Eq (44-48) and Eq (53-54).

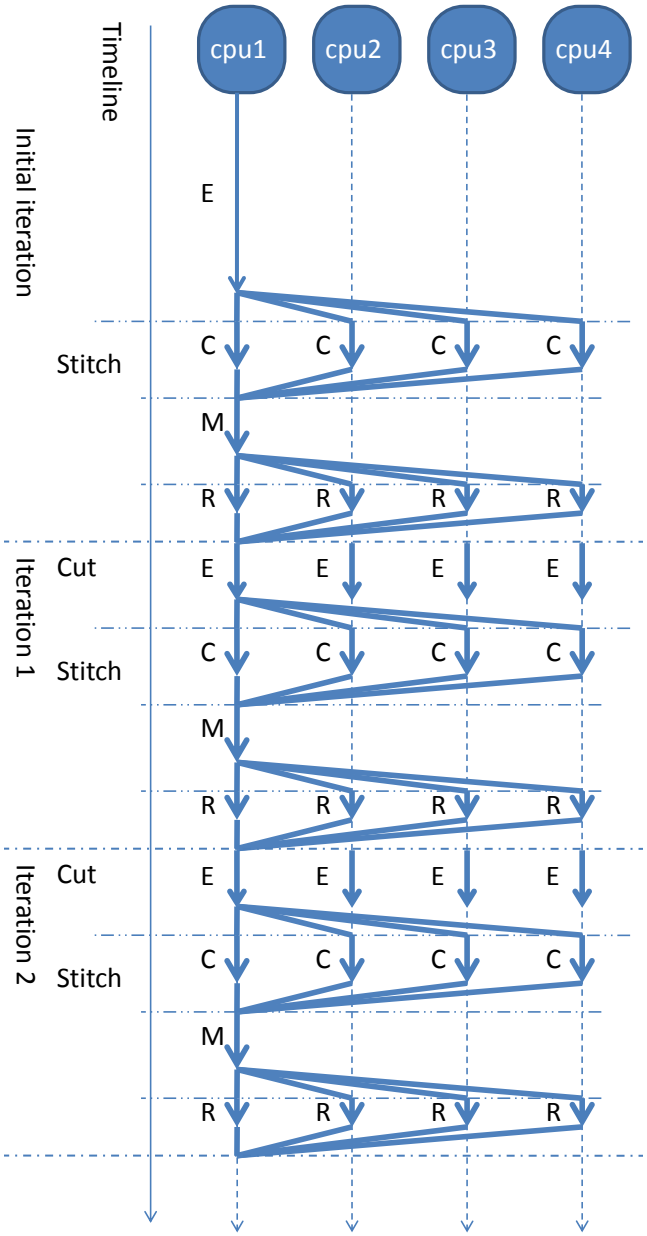


Fig. 3 Graphical illustration of *Cut-And-Stitch* algorithm on 4 CPUs. Arrows indicates the computation on each CPU. Tilting lines indicate the necessary synchronization and data transfer between the CPUs and main memory. Tasks labeled with "E" indicate the (parallel) *estimation* of the posterior marginal distribution, including the forward-backward propagation of beliefs within each block as shown in Figure 2. (C) indicates the *collection* of local statistics of the hidden variables in each block; (M) indicates the *maximization* of the expected log-likelihood over the parameters, and then it *re-estimates* (R) the block parameters.

4. Implementation

We will first discuss properties of our proposed *Cut-And-Stitch* for both LDS and HMM and what it implies for the

requirements of the computer architecture:

- **Symmetric**: The *Cut* step creates a set of equally-sized blocks, no matter whether discrete hidden variables or continuous, assigned to each processor. Since the amount of computation depends on the size of the block, both algorithms expect good load balancing on symmetric processors.
- **Shared Memory**: The *Stitch* step involves summarizing sufficient statistics collected from each processor. This step can be done more efficiently in shared memory, rather than in distributed memory.
- **Local Cache**: In order to reduce the impact of the bottleneck of processor-to-memory communication, local caches are necessary to keep data for each block.

The current Symmetric MultiProcessing (SMP) technologies provide opportunities to match all of these assumptions. We implement our CAS-LDS and CAS-HMM using OpenMP, a multi-programming interface that supports shared memory on many architectures, including both commercial desktops and supercomputer clusters. Alternatively, it can also be implemented in MPI without significant modification. We use the OpenMP to create multiple threads, share the workload and synchronize the threads among different processors. Note that OpenMP needs compiler support to translate parallel directives to run-time multi-threading. And it also includes its own library routines (e.g. timing) and environment variables (e.g. the number of running processors).

There are several issues on configuring OpenMP for both parallel learning algorithm as follows:

- **Variable Sharing** Posterior expectations computed in the E-step (Eq. (32-34) and Eq. (51-52)) are stored in global variables of OpenMP, visible to every processor. There are also several intermediate matrices and vectors results for which only local copies need to be kept; they are temporary variables that belong to only one processor. This also saves the computational cost by preserving locality and reducing cache miss rate.
- **Dynamic or Static Scheduling** What is a good strategy to assign blocks to processors? OpenMP provides two choices: static and dynamic. Static scheduling will fix processor to always operate on the same codes while dynamic scheduling takes an on-demand approach. We pick the static scheduling approach (i.e. fix the block-processor mapping), for the following reasons: (a) the computation is logically block-wise and in a regular fashion and (b) we have performance gains by exploiting the temporal locality when we always associate the same processor with the same block. Furthermore, we maximize any parallelizable computation in each of the E,C,M,R steps. For example, in our implementation of CAS-LDS, we improve the M-step by using four processors to calculate model parameters in Eq (38-43): two for Eq (38-39), one for Eq (40-41) and one for Eq (42-43).
- **Synchronization** As described earlier, the *Stitch* step of the learning algorithm should happen only after the *Cut* step has completed, and the order of stages inside *Stitch* should be *collecting*, *maximization* and *re-estimate*. We put barriers after each step/stage to synchronize the threads and keep them in the same pace. Each iteration would include four barriers, as shown in Figure 3.

5. Experiments

To evaluate the effectiveness and usefulness of our proposed *Cut-And-Stitch* method in practical applications, we tested our implementation on SMPs. Our goal is to answer the following questions:

- Speedup: how would the performance change as the number of processors/cores increase?
- Quality: while the parallel algorithm is faster than serial algorithm, are we giving up any precision on derived model?

We will first describe the experimental setup and the dataset we used.

5.1 Dataset and Experimental Setup

We run the experiments on a variety of typical SMPs, two supercomputers and a commercial desktop.

M1 The first supercomputer is an SGI Altix system[†], at National Center for Supercomputing Applications (NCSA). The cluster consists of 512 1.6GHz Itanium2 processors, 3TB of total memory and 9MB of L3 cache per processor. It is configured with an Intel C++ compiler supporting OpenMP. We use this supercomputer to test our LDS algorithm.

M2 The second supercomputer is an SGI Altix system[††], at Pittsburgh Supercomputing Center (PSC). The cluster consists of 384 1.66GHz Itanium2 Montvale 9130M dual-core processors (a total of 768 cores), 1.5TB of total memory and 8MB of L3 cache per processor. It is configured with SuSE Linux and Intel compiler. We use this supercomputer to test our HMM algorithm.

M3 The test desktop machine has two Intel Xeon dual-core 3.0GHz CPUs (a total of four cores), 16G memory, running Linux (Fedora Core 7) and GCC 4.1.2 (supporting OpenMP). We use this supercomputer to test both of our LDS and HMM algorithm.

To test our LDS algorithm, we used a 17MB motion dataset from CMU Motion Capture Database [†††]. It consists of 58 walking, running and jumping motions, each with 93 bone positions in body local coordinates. The motions span several hundred frames long (100∼500). We use our

[†] cobalt.ncsa.uiuc.edu
[††] http://www.psc.edu/machines/sgi/altix/pople.php
[†††] http://mocap.cs.cmu.edu/

Table 2 Rough estimation of the number of arithmetic operations $(+, -, \times, /)$ in E, C, M, R sub steps of *Cut-And-Stitch* in LDS. Each type of operation is equally weighted, and only the largest portions in each step are kept.

	#of operation of LDS
E	$N \cdot (m^3 + H \cdot m^2 + m \cdot H^2 + 8H^3)$
C	$N \cdot H^3$
M	$2k \cdot H^2 + 4H^3 + k \cdot m \cdot H + 2m \cdot H^2 + m^2 \cdot H$
R	$2k \cdot H^3$

Table 3 Wall-clock time for the case of Walking Motion (#22) on multi-processor/multi-core (in seconds), and the average of normalized running time on 58 motions (serial time= 1).

# of Procs	time (sec.)	avg. of norm. time
1(serial)	3942	1
2	1974	0.5
4	998	0.256
8	510	0.134
16	277	0.0703
32	171	0.0438
64	117	0.0342
128	115	0.0335

Table 4 Count of arithmetic operations $(+, -, \times, /)$ in E, C, M, R sub steps of *Cut-And-Stitch* in HMM. Each type of operation is equally weighted, and only the largest portions in each step are kept.

	#of operations of HMM
E	$9 \cdot N \cdot K^2$
C	$2K \cdot N \cdot (K + M)$
M	$2k \cdot K \cdot (K + M)$
R	$4k \cdot K^2$

method to learn the transition dynamics and projection matrix of each motion, using H=15 hidden dimensions.

For HMM, we used a synthetic dataset, with the observation sequences randomly generated. The data has $K = 100$ hidden states and $R = 50$ different observation values. The duration of the sequence is N=1536.

5.2 Speedup

The speedup for k processors is defined as

$$S_k = \frac{\text{running time with a single processor}}{\text{running time with } k \text{ processors}}$$

According to Amdahl's law, the theoretical limit of speedup is

$$S_k \leq \frac{1}{(1-p) + \frac{p}{k}} < k$$

where p is the proportion of the part that could run in parallel, and $(1 - p)$ is the part remains serial. To determine the speedup limit, we provide an analysis of the complexity of both algorithms by counting the basic arithmetic operations.

5.2.1 Speedup for LDS

We did experiment on all of the 58 motions with various number of processors on both machines (M1 and M3). Assume that the matrix multiplication takes cubic time, the inverse uses Gaussian elimination, there is no overhead in synchronization, and there is no memory contention. Table 2 lists a rough estimate of the number of basic arithmetic operations in the *Cut* and *Stitch* steps with E, C, M, and R sub steps. As we mentioned in Section 3, the E,C,R sub steps can run on k processors in parallel, while the M step in principle, has to be performed serially on a single processor (or up to four processors with a finer breakdown of the computation).

In our experiment, N is around 100-500, $m = 93$, $H = 15$, thus p is approximately $99.81\% \sim 99.96\%$.

Figure 4 shows the wall clock time and speedup on M1 with a maximum of 128 processors. Figure 5 shows the wall clock time and speedup on M3 (maximum 4 cores). We also include the theoretical limit from Amdahl's law. Table 3 lists the running time on the motion set. In order to compute the average running time, we normalized the wall clock time relative to the serial one, defined as

$$t_{norm} = \frac{t_k}{t_1} = \frac{1}{S_k}$$

where t_k is wall clock time with k processors.

The performance results show almost linear speedup as we increase the number of processors, which is very promising. Taking a closer look, it is near linear speedup up to 64 processors. The speedup for 128 processors is slightly below linear. A possible explanation is that we may hit the bus bandwidth between processors and memory, and the synchronization overhead increases dramatically with a hundred processors.

5.2.2 Speedup for HMM

The algorithmic complexity of our parallel HMM implementation is shown in Table 4. Figure 6 and Figure 7 show the wall clock time and speedup on multi-core desktop and PSC supercomputer respectively. Comparing to LDS with similar model size, each iteration of HMM implementation would take much less time, so the overhead of parallel framework stands out earlier: the speedup for HMM starts to getting less impressive when we use about 16 processors.

However, for some Hidden Markov Model applications such as bioinformatics, sequence length (N) could be much larger: for example, each DNA sequence might contain thousands, millions or even more *b*ase pairs(pairs of nucleotides A,T,G,C). We envision that our *Cut-And-Stitch* method would exhibit better speedup towards those problems.

5.3 Quality

In order to evaluate the quality of our parallel algorithm, we run our algorithm on a different number of processors and compare the error against the serial version (EM algorithm on single processor). Due to the non-identifiability problem,

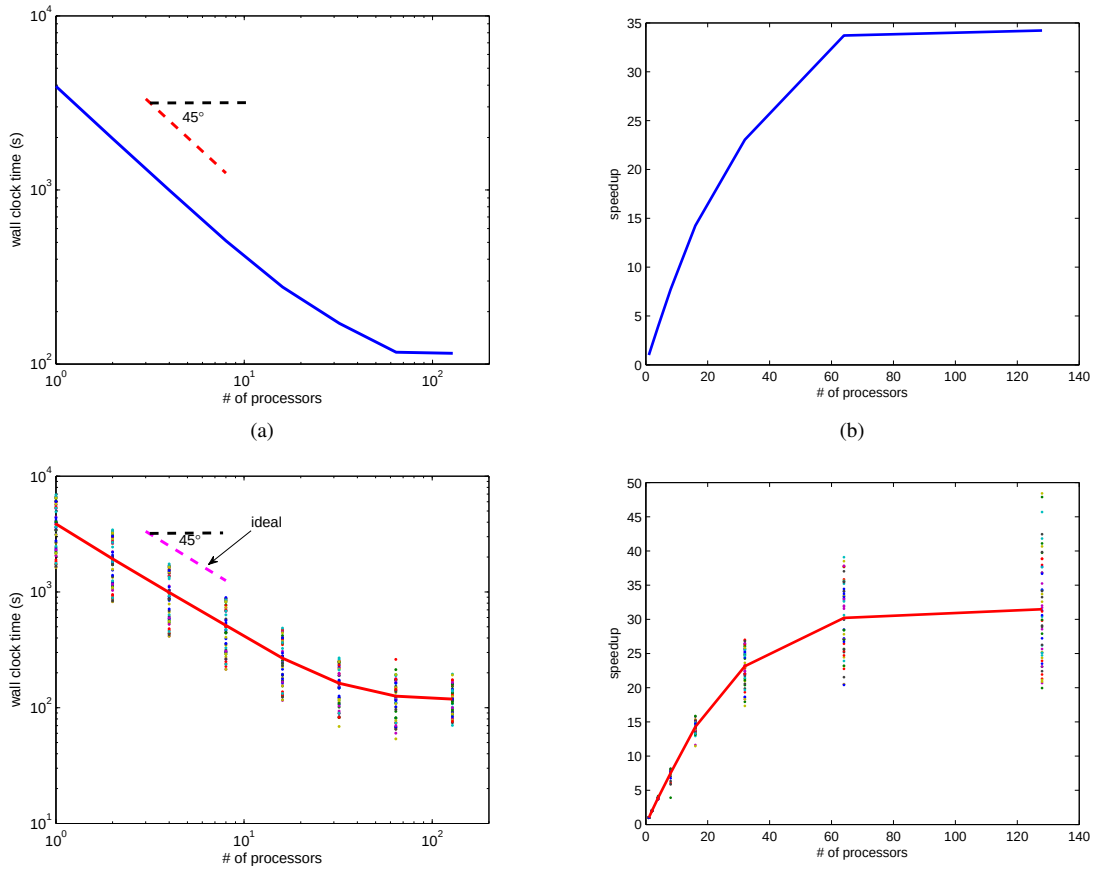


Fig. 4 Performance of *Cut-And-Stitch* LDS on NCSA supercomputer, running on the 58 motions. The Sequential version is on one processor, identical to the EM algorithm. (a) Running time for a sample motion (subject 16 #22, walking, 307 frames) in log-log scales; (b) Speedup for walking motion(subject 16 #22) compared with the sequential algorithm; (c) Average running time (solid line) for all motions in log-log scales. (d) Average speedup for all motions, versus number of processors k.

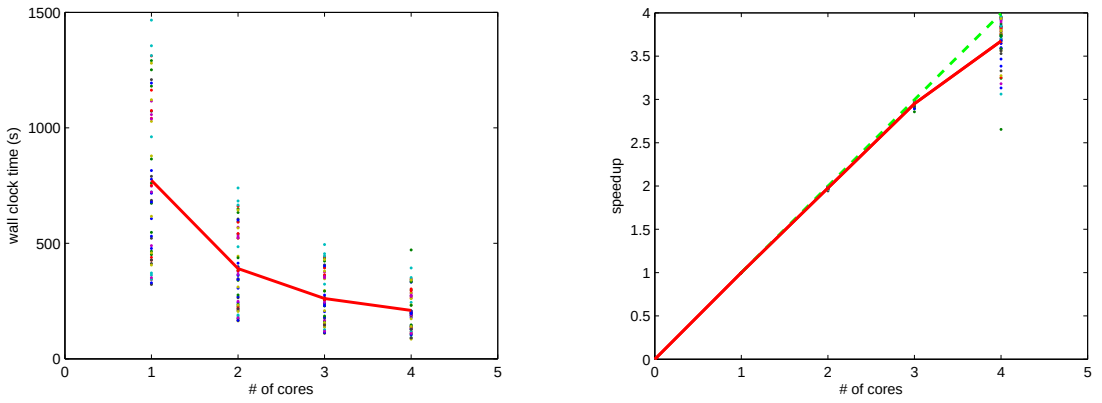
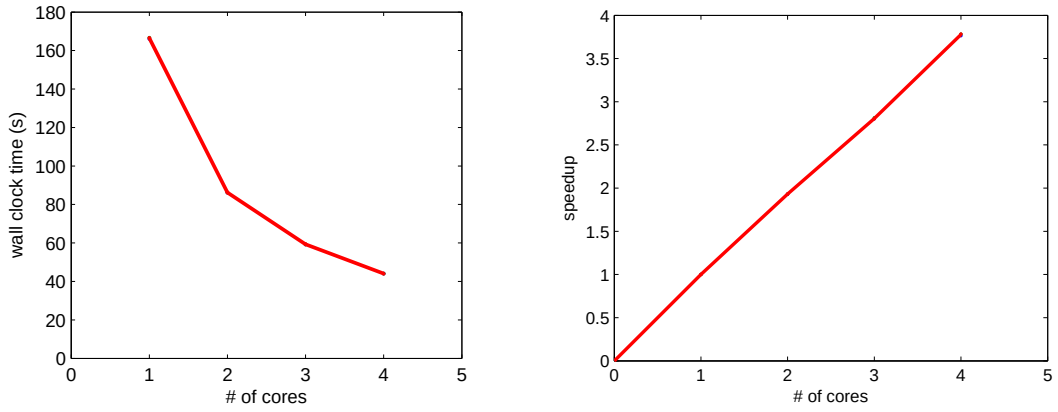


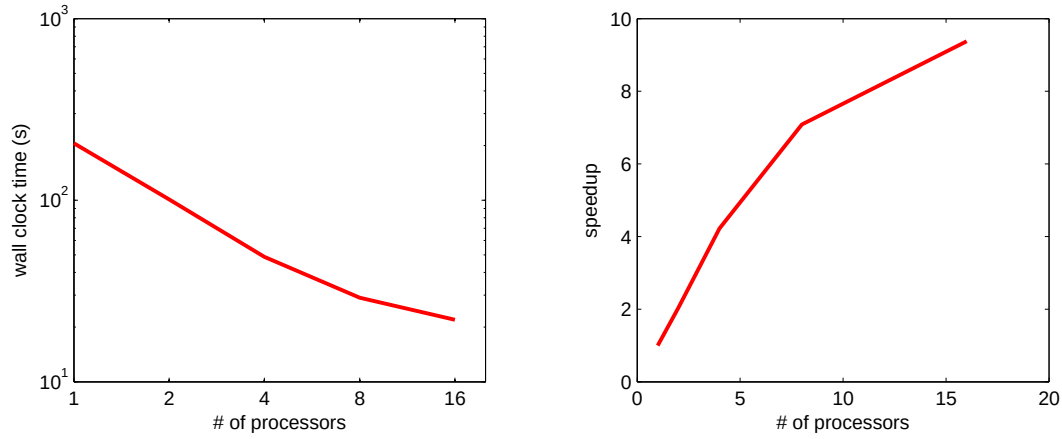
Fig. 5 Performance of *Cut-And-Stitch* LDS on multi-core desktop, running on the 58 motions. The Sequential version is on one processor, identical to the EM algorithm. (a) running time for all motions; (b) average speedup for the 58 motions, versus number of cores k.

the model parameters for different run might be different, thus we could not directly compute the error on the model parameters. Since both the serial EM learning algorithm and the parallel one tries to maximize the data log-likelihood,



(a) (b)

Fig. 6 Performance of *Cut-And-Stitch* HMM on multi-core desktop, (a) running time versus number of cores k; (b) speedup versus number of cores k.



(a) (b)

Fig. 7 Performance of *Cut-And-Stitch* HMM on PSC supercomputer, (a) running time versus number of cores k in log-log scale; (b) speedup versus number of cores k.

we define the error as the relative difference between log-likelihood of the two, where data log-likelihood is computed from the E step of the EM algorithm.

$$error_k = \frac{l(\mathcal{Y};\hat{\theta}_1) - l(\mathcal{Y};\hat{\theta}_k)}{l(\mathcal{Y};\hat{\theta}_1)} \times 100\%$$

where $\mathcal{Y}$ is the motion data sequence, $\hat{\theta}_k$ are parameters learned with k processors and $l(\cdot)$ is the log-likelihood function. The error to both of our LDS and HMM experiments are very tiny: error of LDS algorithm has a maximum of 0.5% and mean of 0.17%; error of HMM algorithm has a maximum of 1.7% and mean of 1.2%. Furthermore, there is no clear possitive correlation between error and the number of processors. In some cases, the parallel algorithm even found higher (0.074%) likelihood than the serial algorithm. Note there are limitations of the log-likelihood criteria, namely higher likelihood does not necessarily indicate better fitting, since it might get over-fitting. The error curve shows the quality of parallel is almost identical to the serial one.

5.4 Case study for LDS

In order to show the visual quality of the parallel learning algorithm, we observe a case study of our parallel LDS implementation on two different sample motions: walking motion (Subject 16 #22, with 307 frames), jumping motion (Subject 16 #1, with 322 frames), and running motion (Subject 16 #45, with 135 frames). We run the CAS algorithm with 4 cores to learn model parameters on the multi-core machine, and then use these parameters to estimate the hidden states and reconstruct the original motion sequence. The test criteria is the reconstruction error (NRE) normalized to the variance, defined as

$$NRE = \sqrt{\frac{\sum_{i=1}^{N} ||y_i - \hat{y}_i||^2}{\sum_{i=1}^{N} ||y_i - \sum_{j=1}^{N} y_j/N||^2}} \times 100\%$$

where y_i is the observation for i-th frame and $\hat{y}_i$ is the recon-

structed with model parameters from 4-core computation. Table 5 shows the reconstruction error: both parallel and serial achieve very small error and are similar to each other. Figure 8 and Figure 9 show the reconstructed sequences of the feet coordinates. Note our reconstruction (dotted lines) is very close to the original signal (solid lines).

Table 5 Normalized Reconstruction Error

method	Walking	Jumping	Running
Serial	1.929%	1.139%	0.988%
Parallel(4-core)	1.926%	1.140%	0.985%

6. Related Work

Hidden Markov Model is first introduced by Leonard E. Baum et al. [12] and has been widely applied in many temporal applications such as speech technology [13], handwriting system, part-of-speech tagging and bioinformatics [14].

Data mining and parallel programming receives increasing interest. Parthasarathy et al. [15] develop parallel algorithms for mining terabytes of data for frequent itemsets, demonstrating a near-linear scale-up on up to 48 nodes.

Reinhardt and Karypis [16] used OpenMP to parallelize the discovery of frequent patterns in large graphs, showing excellent speedup of up to 30 processors.

Cong et al. [17] develop the Par-CSP algorithm that detects closed sequential patterns on a distributed memory system, and report good scale-up on a 64-node Linux cluster.

Graf et al. [6] developed a parallel algorithm to learn SVM through cascade SVM. Collobert et al. [18] proposed a method to learn a mixture of SVM in parallel. Both of them adopted the idea of splitting dataset into small subsets, training SVM on each, and then combining those SVMs. Chang et al. [19] proposed PSVM to train SVMs on distributed computers through approximate factorization of the kernel matrix.

There is an attempt to use Google's Map-Reduce [7] to parallelize a set of learning algorithm such as naïve-Bayes, PCA, linear regression and other similar algorithms [8, 9]. Their framework requires the summation form (like dot-product) in the learning algorithm, and hence could distribute independent calculations to many processors and then summarize them together. Therefore the same techniques could hardly be used to learn long sequential graphical models such as Hidden Markov Models and Linear Dynamical Systems.

7. Conclusions

Hidden Markov chain models are important tools in analyzing time series data. In this paper, we explore the problem of parallelizing the learning algorithm on symmetric multiprocessor architectures, for two typical such models, linear dynamical systems (LDS) and hidden Markov models (HMM) The main contributions are as follows:

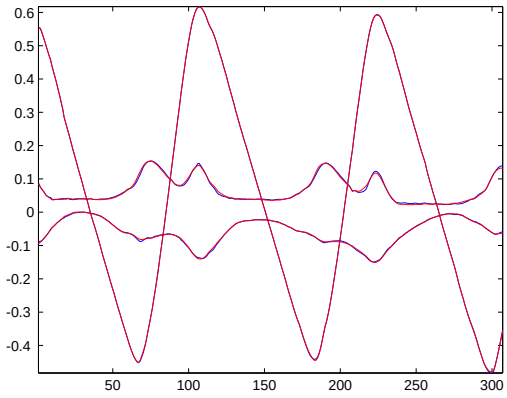
- We propose approximate parallel learning algorithms CAS-LDS for Linear Dynamic System and CAS-HMM Hidden Markov Model, and implement them using the OpenMP API on shared memory machines.
- We performed experiments for LDS on a large collection 58×93 real motion capture sequences spanning 17 MB. *Cut-And-Stitch* showed near-linear speedup on typical settings (a commercial multi-core desktop, as well as a supercomputer). We showed that our reconstruction error is almost identical to the serial algorithm.
- Experiments for HMM on a synthetic dataset exhibit also ideal speedup on multiple processors. Quality of model generated from parallel *Cut-And-Stitch* is as good as from serial algorithm.

Acknowledgments

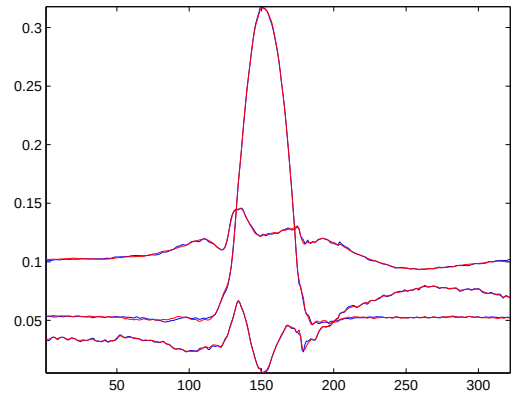
This material is based upon work supported by the National Science Foundation under Grants No.DBI-0640543. The data used in this project was obtained from mocap.cs.cmu.edu supported by NSF EIA-0196217. Also, under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344 (LLNL-CONF-404625), subcontracts B579447, B580840. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding parties. We would like to thank Pittsburgh Supercomputing Center and National Center for Supercomputing Applications for providing machines and support.

References

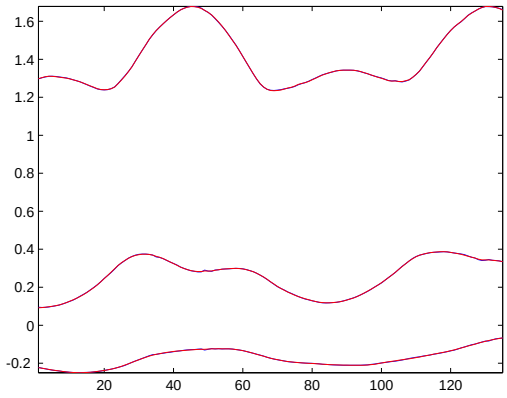
[1] L. Li, J. McCann, C. Faloutsos, and N. Pollard, "Laziness is a virtue: Motion stitching using effort minimization," Short Papers Proceedings of EUROGRAPHICS, 2008.
[2] C.M. Bishop, Pattern Recognition and Machine Learning, Springer, 2006.
[3] C.B. Colohan, A. Ailamaki, J.G. Steffan, and T.C. Mowry, "Tolerating dependences between large speculative threads via sub-threads," ISCA '06, pp.216–226, IEEE Computer Society, 2006.
[4] Intel, "Intel research advances 'era of tera': http://www.intel.com/pressroom/archive/releases/2007/20070204comp.htm.
[5] Intel, "Intel lifts the hood on its 'single-chip cloud computer': http://spectrum.ieee.org/semiconductors/processors/intel-lifts-the-hood-on-its-singlechip-cloud-computer."
[6] H.P. Graf, E. Cosatto, L. Bottou, I. Dourdanovic, and V. Vapnik, "Parallel support vector machines: The cascade SVM," NIPS 17, 2004.
[7] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," OSDI'04: Sixth Symposium on Operating System Design and Implementation, 2004.
[8] 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," NIPS 19, ed. B. Schölkopf, J.C. Platt, and T. Hoffman, pp.281–288, MIT Press, 2006.
[9] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and



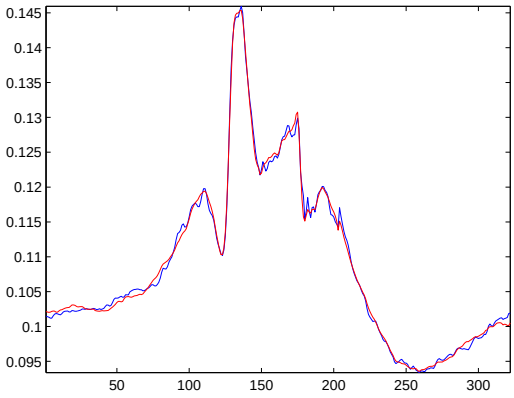
(a) right foot (16#22)



(b) left foot (16#1)

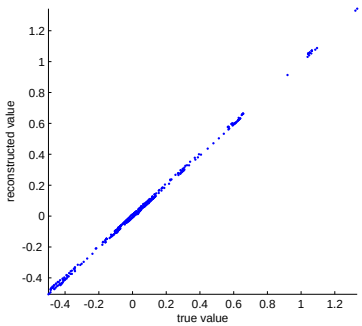


(c) right foot (16#45)

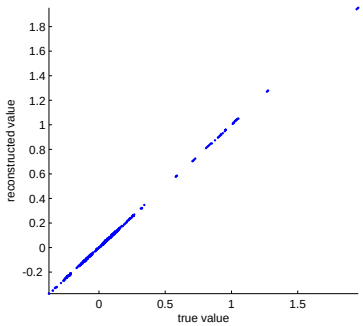


(d) left foot, x (16#1)

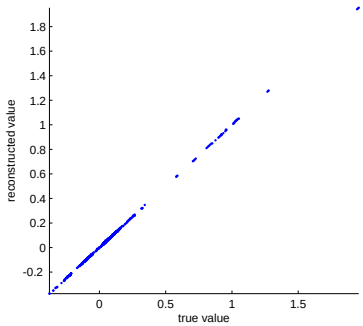
Fig. 8 Visual effects: the reconstructed x, y, z coordinates using learned parameters on 4 cores. Horizontal axis is frame index (time tick). (a) right foot coordinates (x,y,z) for the walking motion (subject 16 #22). (b) left foot coordinates for the jumping motion (subject 16 #1). (c) right foot coordinates for the running motion (subject 16 #45). (d) magnification of the x coordinate (the upper curve in (b)). Note that the reconstructed sequences (dotted lines) are so close to the original signals (solid lines), that the plots looks like a set of overlapping lines; this illustrates the high accuracy of *Cut-And-Stitch*.



(a)



(b)



(c)

Fig. 9 Scatter plot: reconstructed value versus true value. For clarity, we only show the 500 *worst* reconstructions - even then, the points are very close on the 'ideal', 45 degree line. (a) walking motion (subject 16 #22). (b) jumping motion (subject 16 #1). (c) running motion (subject 16 #45).

C. Kozyrakis, "Evaluating mapreduce for multi-core and multiprocessor systems," HPCA '07, pp.13–24, IEEE Computer Society, 2007.

[10] L. Li, J. McCann, N. Pollard, and C. Faloutsos, "Dynammo: Mining

and summarization of coevolving sequences with missing values," Proc. of 15th SIGKDD, New York, NY, USA, ACM, 2009.
[11] L.E. Baum, T. Petrie, G. Soules, and N. Weiss, "A maximization technique occurring in the statistical analysis of probabilistic functions of markov chains," The Annals of Mathematical Statistics, vol.41, no.1, pp.164–171, Feb. 1970.
[12] L.E. Baum and T. Petrie, "Statistical inference for probabilistic functions of finite state markov chains," The Annals of Mathematical Statistics, vol.37, pp.1554–1563, 1966.
[13] L.R. Rabiner, "A tutorial on Hidden Markov Models and selected applications in speech recognition," Proc. IEEE, vol.77, no.1, pp.257–285, Feb. 1989.
[14] S.R. Eddy, "Profile Hidden Markov Models," Bioinformatics, vol.14, pp.755–763, Feb. 1998.
[15] G. Buehrer, S. Parthasarathy, S. Tatikonda, T. Kurc, and J. Saltz, "Toward terabyte pattern mining: an architecture-conscious solution," Proc. of the 12th ACM SIGPLAN, pp.2–12, ACM, 2007.
[16] S. Reinhardt and G. Karypis, "A multi-level parallel implementation of a program for finding frequent patterns in a large sparse graph," IPDPS, pp.1–8, IEEE, 2007.
[17] S. Cong, J. Han, and D. Padua, "Parallel mining of closed sequential patterns," Proc. of the 11th ACM SIGKDD, pp.562–567, ACM, 2005.
[18] R. Collobert, S. Bengio, and Y. Bengio, "A parallel mixture of SVMs for very large scale problems," NIPS, ed. T.G. Dietterich, S. Becker, and Z. Ghahramani, MIT Press, 2002.
[19] E. Chang, K. Zhu, H. Wang, H. Bai, J. Li, Z. Qiu, and H. Cui, "Parallelizing support vector machines on distributed computers," NIPS 20, ed. J. Platt, D. Koller, Y. Singer, and S. Roweis, pp.257–264, MIT Press, 2007.

Lei Li received his B.E. degree from the Department of Computer Science and Engineering, Shanghai Jiao Tong University. Lei is currently a Ph.D. candidate at Computer Science Department, Carnegie Mellon University. His research interests include machine learning, data mining, and time series analysis.

Bin Fu received his B.E. degree from the Department of Computer Science and Technology, Tsinghua University. Bin is currently a Ph.D. candidate at Computer Science Department, Carnegie Mellon University. His research interests include artificial intelligence.

Christos Faloutsos is a Professor at Carnegie Mellon University.
He has received the Presidential Young Investigator Award by the National Science Foundation (1989), the Research Contributions Award in ICDM 2006, thirteen "best paper" awards, and several teaching awards. He has served as a member of the executive committee of SIGKDD; he has published over 200 refereed articles, 11 book chapters and one monograph. He holds five patents and he has given over 30 tutorials and over 10 invited distinguished lectures. His research interests include data mining for graphs and streams, fractals, database performance, and indexing for multimedia and bio-informatics data.