Cut-And-Stitch: Efficient Parallel Learning of Linear Dynamical Systems on SMPs

Lei Li, Wenjie Fu, Fan Guo, Todd C. Mowry, Christos Faloutsos
Carnegie Mellon University
{leili,wenjief,fanguo,tcm,christos}@cs.cmu.edu

ABSTRACT

Multi-core processors with ever increasing number of cores per chip are becoming prevalent in modern parallel computing. Our goal is to make use of the multi-core as well as multi-processor architectures to speed up data mining algorithms. Specifically, we present a parallel algorithm for approximate learning of Linear Dynamical Systems (LDS), also known as Kalman Filters (KF). LDSs are widely used in time series analysis such as motion capture modeling and visual tracking etc. We propose *Cut-And-Stitch* (CAS), a novel method to handle the data dependencies due to the chain structure of hidden variables in LDS, so as to parallelize the EM-based parameter learning algorithm. We implement the algorithm using OpenMP on both a supercomputer 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 serial version. In addition, *Cut-And-Stitch* can be generalized to other models with similar linear structures such as Hidden Markov Models (HMM) and Switching Kalman Filters (SKF).

Categories and Subject Descriptors: I.2.6 Artificial Intelligence: Learning - parameter learning D.1.3 Programming Techniques: Concurrent Programming - parallel programming G.3 Probability and Statistics: Time series analysis

General Terms: Algorithms; Experimentation; Performance.

Keywords: Linear Dynamical Systems; Kalman Filters; OpenMP; Expectation Maximization (EM); Optimization; Multi-core.

1. INTRODUCTION

Time series appear in numerous applications, including motion capture [11], visual tracking, speech recognition, quantitative studies of financial markets, network intrusion detection, forecasting, etc. Mining and forecasting are popular operations relevant to time series analysis. Two typical statistical models for such problems are hidden Markov models (HMM) and linear dynamical systems (LDS, also known as Kalman filters). Both assume linear transitions on hidden (i.e. 'latent') variables which are considered discrete for HMM and continuous for LDS. The hidden states or variables in both models can be inferred through a forward-backward procedure involving dynamic programming. However, the maximum likelihood estimation of model parameters is difficult, requiring the well-known Expectation-Maximization (EM) method [1]. The EM algorithm for learning of LDS/HMM iterates between computing conditional expectations of hidden variables through the forward-backward procedure (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 cubic in the dimensionality of observations, 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 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 for LDS 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 [6] benefit little to the straightforward EM algorithm due to the nontrivial data dependencies in LDS. 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 [10]. 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 [9]. Other articles use Google's map-reduce techniques [8] 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 [4, 12]. 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. [1]

[1]Or exactly, models with large diameters. The diameter of a model is the length of longest acyclic path in its graphical representation. For example, the diameter of the LDS in Figure 1 is N.

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, $m \times H$

Table 1: Symbol table

In this paper, we propose the *Cut-And-Stitch* method (CAS), which avoids the data-dependency problems. We show that CAS can quickly and accurately learn an LDS in parallel, as demonstrated on two popular architectures for high performance computing. The basic idea of our algorithm 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 would 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 chain structure of LDS (also HMM), 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

Here we give a brief introduction to Linear Dynamical Systems (LDS), including its formalization, its learning algorithm and its connections to hidden Markov models (HMM).

Consider a multi-dimensional sequence $\mathcal{Y} = \mathbf{y}_1, \ldots, \mathbf{y}_\mathrm{N}$ of a length N. For example, $\mathcal{Y}$ could be a sequence of marker position vectors captured by video cameras, where each vector $\mathbf{y}_i$ is of dimensionality m. Suppose the evolution of the observation is driven by a hidden Markov process. For example, in motion capture modeling, hidden variables may correspond to a sinusoid moving pattern, while the observed motion could be periodic walking cycles. In LDS, both the transitions among the hidden variables as well as their projections to the observations are described as linear Gaussian models (Eq (2-2)). 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:

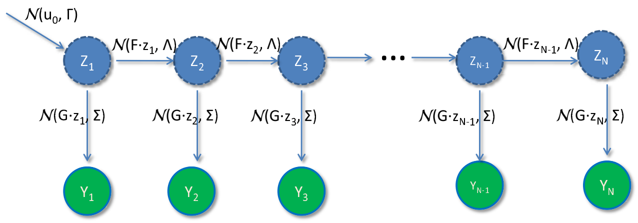
$$\mathbf{z}_1 = z_0 + \omega_0 \quad (1)$$

$$\mathbf{z}_{n+1} = \mathbf{F}\mathbf{z}_n + \omega_n \quad (2)$$

$$\mathbf{y}_n = \mathbf{G}\mathbf{z}_n + \epsilon_n \quad (3)$$

where z_0 is the initial state of the whole system, and ω_0, ω_i and $\epsilon_i (i = 1 \ldots \mathrm{M})$ are multivariate Gaussian noises:

$$\omega_0 \sim \mathcal{N}(0, \Gamma) \quad \omega_i \sim \mathcal{N}(0, \Lambda) \quad \epsilon_j \sim \mathcal{N}(0, \Sigma)$$



$\mathcal{N}$(u$_0$, Γ)
$\mathcal{N}$(F·z$_1$, Λ)
$\mathcal{N}$(F·z$_2$, Λ)
$\mathcal{N}$(F·z$_{N-1}$, Λ)
Z$_1$
Z$_2$
Z$_3$
Z$_{N-1}$
Z$_N$
$\mathcal{N}$(G·z$_1$, Σ)
$\mathcal{N}$(G·z$_2$, Σ)
$\mathcal{N}$(G·z$_3$, Σ)
$\mathcal{N}$(G·z$_{N-1}$, Σ)
$\mathcal{N}$(G·z$_N$, Σ)
Y$_1$
Y$_2$
Y$_3$
Y$_{N-1}$
Y$_N$

Figure 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.

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 [1], 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.

Although we will focus on LDS in the rest of this paper, our *Cut-And-Stitch* method could also be adapted to HMMs with a careful replacement of context, because their graphical models are very similar. Figure 1 shows the structure of the graphical representation of an LDS; notice that the structure remains the same for hidden Markov models, with the only differences that the hidden (and possibly observed) variables are discrete and that the conditional distributions should be replaced by multinomial distributions. Accordingly, the forward-backward algorithm of HMM is still tractable and can be implemented in a similar manner, and the M-step in the learning algorithm can be modified as well.

3. CUT-AND-STITCH: PROPOSED METHOD

In the standard EM learning algorithm described in Section 2, the chain structure of the LDS 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 ideas on overcoming such dependencies and describe the details of *Cut-And-Stitch* parallel learning algorithm.

3.1 Intuition and Preliminaries

Our guiding principle to reduce the data dependencies is to divide LDS 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 parameters, say $\theta_1, \ldots, \theta_k$, 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

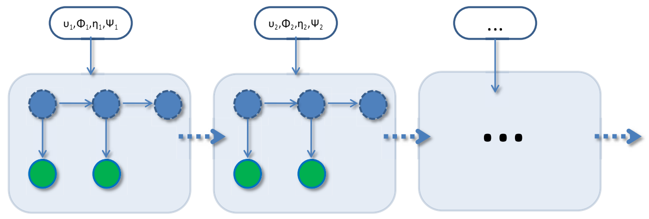


Figure 2: Graphical illustration of dividing LDS into blocks in the *Cut* step. Note *Cut* introduces additional parameters for each block.

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.

Given a sequence of observed values $\mathcal{Y}$ with length of N, the learning goal is to best fit the parameters $\theta = (\mu_0, \Gamma, F, \Lambda, G, \Sigma)$. 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 LDS 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 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):

$$\begin{aligned} P(\mathbf{z}_{i,1}) &= \mathcal{N}(\upsilon_i, \Phi_i) && (4)\\ P(\mathbf{z}_{i,j+1}|\mathbf{z}_{i,j}) &= \mathcal{N}(\mathbf{F}\mathbf{z}_{i,j}, \Lambda) && (5)\\ P(\mathbf{z}'_{i,T}|\mathbf{z}_{i,T}) &= \mathcal{N}(\mathbf{F}\mathbf{z}_{i,T}, \Lambda) && (6)\\ P(\mathbf{y}_{i,j}|\mathbf{z}_{i,j}) &= \mathcal{N}(\mathbf{G}\mathbf{z}_{i,j}, \Sigma) && (7) \end{aligned}$$

where the block size $T = \frac{\mathrm{N}}{k}$ and $j = 1 \ldots T$ 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 $\mathbf{z}'_{i,T}$.

$$P(\mathbf{z}'_{i,T}) = \mathcal{N}(\eta_i, \Psi_i) \qquad (8)$$

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:

$$\begin{aligned} P(\mathbf{z}_{i,1}|\mathbf{y}_1, \ldots, \mathbf{y}_{i-1,T}) &= \mathcal{N}(\upsilon_i, \Phi_i) && (9)\\ P(\mathbf{z}_{i+1,1}|\mathbf{y}_1, \ldots, \mathbf{y}_\mathrm{N}) &= \mathcal{N}(\eta_i, \Psi_i) && (10) \end{aligned}$$

Therefore, we could derive a local belief propagation algorithm to compute the marginal posterior $P(\mathbf{z}_{i,j}|\mathbf{y}_{i,1} \ldots \mathbf{y}_{i,T}; \upsilon_i, \Phi_i, \eta_i, \Psi_i, \theta)$. 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 $P(\mathbf{z}_{i,j}|\mathbf{y}_{i,1} \ldots \mathbf{y}_{i,j})$, while the local backward pass calculates the whole posterior $P(\mathbf{z}_{i,j}|\mathbf{y}_{i,1} \ldots \mathbf{y}_{i,T})$ (to save space, we omit the parameters). Using the properties of linear Gaussian conditional distribution and Markov properties (Chap.2 &8 in [1]), one can easily infer that both posteriors are Gaussian distributions, denoted as:

$$\begin{aligned} P(\mathbf{z}_{i,j}|\mathbf{y}_{i,1} \ldots \mathbf{y}_{i,j}) &= \mathcal{N}(\mu_{i,j}, \mathbf{V}_{i,j}) && (11)\\ P(\mathbf{z}_{i,j}|\mathbf{y}_{i,1} \ldots \mathbf{y}_{i,T}) &= \mathcal{N}(\hat{\mu}_{i,j}, \hat{\mathbf{V}}_{i,j}) && (12) \end{aligned}$$

We can obtain the following forward-backward propagation equations from Eq (4-8) by substituting Eq (9-12) and expanding.

$$\begin{aligned} \mathbf{P}_{i,j-1} &= \mathbf{F}\mathbf{V}_{i,j-1}\mathbf{F}^T + \Lambda && (13)\\ \mathbf{K}_{i,j} &= \mathbf{P}_{i,j-1}\mathbf{G}^T(\mathbf{G}\mathbf{P}_{i,j-1}\mathbf{G}^T + \Sigma)^{-1} && (14)\\ \mu_{i,j} &= \mathbf{F}\mu_{i,j-1} + \mathbf{K}_{i,j}(\mathbf{y}_{i,j} - \mathbf{G}\mathbf{F}\mu_{i,j-1}) && (15)\\ \mathbf{V}_{i,j} &= (\mathbf{I} - \mathbf{K}_{i,j})\mathbf{P}_{i,j-1} && (16) \end{aligned}$$

The initial values are given by:

$$\begin{aligned} \mathbf{K}_{i,1} &= \mathbf{\Phi}_i\mathbf{G}^T(\mathbf{G}\mathbf{\Phi}_i\mathbf{G}^T + \Sigma)^{-1} && (17)\\ \mu_{i,1} &= \upsilon_i + \mathbf{K}_{i,1}(\mathbf{y}_{i,1} - \mathbf{G}\upsilon_i) && (18)\\ \mathbf{V}_{i,1} &= (\mathbf{I} - \mathbf{K}_{i,1})\mathbf{\Phi}_i && (19) \end{aligned}$$

The backward passing equations are:

$$\begin{aligned} \mathbf{J}_{i,j} &= \mathbf{V}_{i,j}\mathbf{F}^T(\mathbf{P}_{i,j})^{-1} && (20)\\ \hat{\mu}_{i,j} &= \mu_{i,j} + \mathbf{J}_{i,j}(\hat{\mu}_{i,j+1} - \mathbf{F}\mu_{i,j}) && (21)\\ \hat{\mathbf{V}}_{i,j} &= \mathbf{V}_{i,j} + \mathbf{J}_{i,j}(\hat{\mathbf{V}}_{i,j+1} - \mathbf{P}_{i,j})\mathbf{J}_{i,j}^T && (22) \end{aligned}$$

The initial values are given by:

$$\begin{aligned} \mathbf{J}_{i,T} &= \mathbf{V}_{i,T}\mathbf{F}^T(\mathbf{F}\mathbf{V}_{i,T}\mathbf{F}^T + \Lambda)^{-1} && (23)\\ \hat{\mu}_{i,T} &= \mu_{i,T} + \mathbf{J}_{i,T}(\eta_i - \mathbf{F}\mu_{i,T}) && (24)\\ \hat{\mathbf{V}}_{i,T} &= \mathbf{V}_{i,T} + \mathbf{J}_{i,T}(\Psi_i - \mathbf{F}\mathbf{V}_{i,T}\mathbf{F}^T - \Lambda)\mathbf{J}_{i,T}^T && (25) \end{aligned}$$

Except for the last block:

$$\hat{\mu}_{k,T} = \mu_{i,T} \qquad \hat{\mathbf{V}}_{k,T} = \mathbf{V}_{i,T} \qquad (26)$$

3.3 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:

$$Q(\theta^{new}, \theta^{old}) = \mathbb{E}_{\theta^{old}}[\log p(\mathbf{y}_1 \ldots \mathbf{y}_N, \mathbf{z}_1 \ldots \mathbf{z}_N|\theta^{new})] \qquad (27)$$

Now taking the derivatives of Eq 27 and zeroing out give the updating equations (Eq (34-39)). 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*.

$$\mathbb{E}[\mathbf{z}_{i,j}] = \hat{\mu}_{i,j} \tag{28}$$
$$\mathbb{E}[\mathbf{z}_{i,j}\mathbf{z}_{i,j-1}^T] = \mathbf{J}_{i,j-1}\hat{\mathbf{V}}_{i,j} + \hat{\mu}_{i,j}\hat{\mu}_{i,j-1}^T \tag{29}$$
$$\mathbb{E}[\mathbf{z}_{i,j}\mathbf{z}_{i,j}^T] = \hat{\mathbf{V}}_{i,j} + \hat{\mu}_{i,j}\hat{\mu}_{i,j}^T \tag{30}$$

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

$$\tau_i = \sum_{j=1}^{T} y_{i,j}\mathbb{E}[\mathbf{z}_{i,j}^T] \tag{31}$$
$$\xi_i = \mathbb{E}[\mathbf{z}_{i,1}\mathbf{z}_{i-1,T}^T] + \sum_{j=2}^{T}\mathbb{E}[\mathbf{z}_{i,j}\mathbf{z}_{i,j-1}^T] \tag{32}$$
$$\zeta_i = \sum_{j=1}^{T}\mathbb{E}[\mathbf{z}_{i,j}\mathbf{z}_{i,j}^T] \tag{33}$$

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,

$$\mu_0^{new} = \hat{\mu}_{1,1} \tag{34}$$
$$\mathbf{\Gamma}_0^{new} = \hat{\mathbf{V}}_{1,1} \tag{35}$$
$$\mathbf{F}^{new} = \left(\sum_{i=1}^{k}\xi_i\right)\left(\sum_{i=1}^{k}\zeta_i - \mathbb{E}[\mathbf{z}_N\mathbf{z}_N^T]\right)^{-1} \tag{36}$$
$$\begin{aligned}\mathbf{\Lambda}^{new} &= \frac{1}{N-1}\Bigg(\sum_{i=1}^{k}(\zeta_i - \mathbf{F}^{new}\xi_i^T - \xi_i(\mathbf{F}^{new})^T) \\ &\quad + \mathbf{F}^{new}(\sum_{i=1}^{k}\zeta_i - \mathbb{E}[\mathbf{z}_N\mathbf{z}_N^T])(\mathbf{F}^{new})^T - \mathbb{E}[\mathbf{z}_{1,1}\mathbf{z}_{1,1}^T]\Bigg)\end{aligned} \tag{37}$$
$$\mathbf{G}^{new} = \left(\sum_{i=1}^{k}\tau_i\right)\left(\sum_{i=1}^{k}\zeta_i\right)^{-1} \tag{38}$$
$$\begin{aligned}\mathbf{\Sigma}^{new} &= \frac{1}{N}\Bigg(Cov(\mathcal{Y}) + \sum_{i=1}^{k}(-\mathbf{G}^{new}\tau_i^T \\ &\quad -\tau_i(\mathbf{G}^{new})^T + \mathbf{G}^{new}\zeta_i(\mathbf{G}^{new})^T)\Bigg)\end{aligned} \tag{39}$$

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 (9-10), we could approximately estimate the block parameters with the following equations.

$$\upsilon_i = \mathbf{F}\mu_{i-1,T} \tag{40}$$
$$\Phi_i = \mathbf{F}\mathbf{V}_{i,T}\mathbf{F}^T + \Lambda \tag{41}$$
$$\eta_i = \hat{\mu}_{i+1,1} \tag{42}$$
$$\Psi_i = \hat{\mathbf{V}}_{i+1,1} \tag{43}$$

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

$$\upsilon_1 = \mu_0 \qquad \Phi_1 = \Gamma \tag{44}$$

In summary, the parallel learning algorithm works 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 (28-30), which includes *forward* and *backward* propagation of beliefs.

Stitch estimates the parameters through *collecting* (C) local statistics of hidden variables in each block Eq (31-33), taking the *maximization* (M) of the expected log-likelihood over the parameters Eq (34-39), and connecting the blocks by *re-estimate* (R) the block parameters Eq (40-44).

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 (31-33) is done in parallel on k processors. Until all local statistics are computed, we use one processor to calculate the parameter using Eq (34-39). Upon the completion of computing the model parameters, every processor computes its own block parameters in Eq (40-44). 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.

3.4 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.

4. IMPLEMENTATION

We will first discuss properties of our proposed *Cut-And-Stitch* method and what it implies for the requirements of the computer architecture:

- **Symmetric**: The *Cut* step creates a set of equally-sized blocks assigned to each processor. Since the amount of computation depends on the size of the block, our method achieves 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.

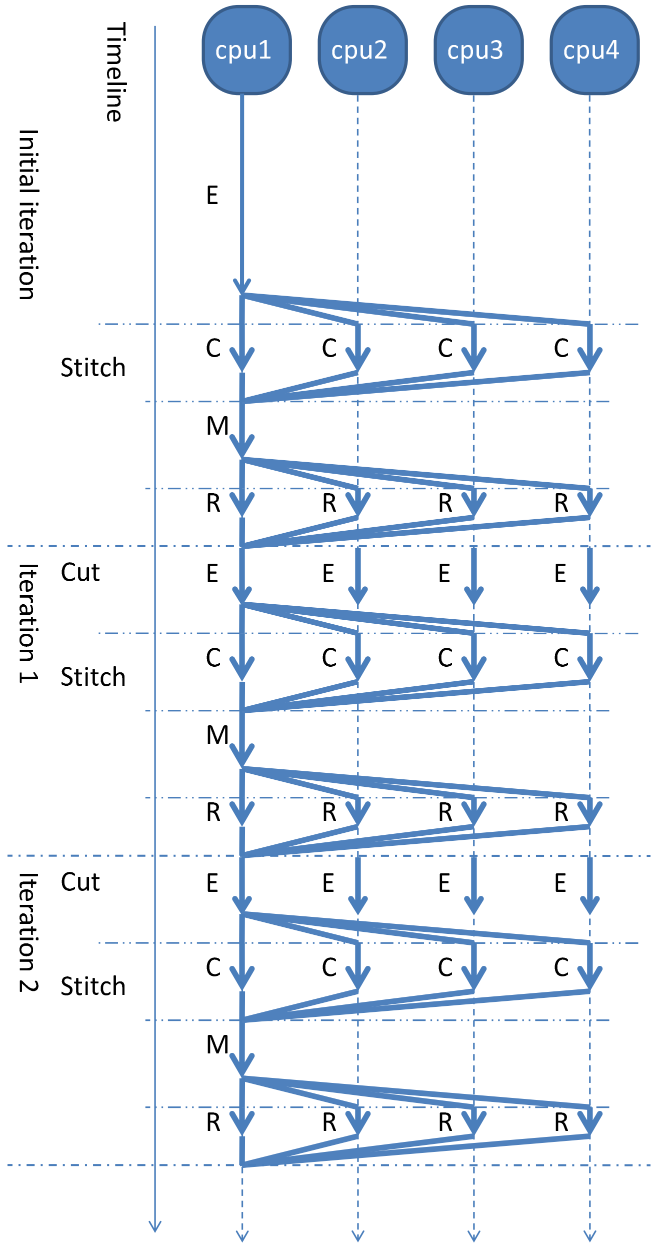


Figure 3: Graphical illustration of *Cut-And-Stitch* algorithm on 4 CPUs. Arrows indicates the computation on each CPU. Tilting lines indicate the necessary synchronization and data transfer between the CPUs and main memory. Tasks labeled with "E" indicate the (parallel) *estimation* of the posterior 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.

- **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 parallel learning algorithm for LDS using OpenMP, a multi-programming interface that supports shared memory on many architectures, including both commercial desktops and supercomputer clusters. Our choice of the multi-processor API is based on the fact that OpenMP is flexible and fast, while the code generation for the parallel version is decoupled from the details of the learning algorithm. 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).

The algorithm is implemented in C++. Several issues on configuring OpenMP for the learning algorithm are listed as follows:

- **Variable Sharing** Conditional expectation in the E-step 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? Usually there are 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, in our implementation, we improve the M-step by using four processors to calculate model parameters in Eq (34-39): two for Eq (34-35), one for Eq (36-37) and one for Eq (38-39).

- **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 and did experiments on real data. Our goal is to answer the following questions:

- Speedup: how would the performance change as the number of processors/cores increase?

- Quality: how accurate is the parallel algorithm, compared to serial one?

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

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

5.1 Dataset and Experimental Setup

We run the experiments on a supercomputer as well as on a commercial desktop, both of which are typical SMPs.

- The supercomputer is an SGI Altix system[2], 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.

- 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 used a 17MB motion dataset from CMU Motion Capture Database [3]. 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 method to learn the transition dynamics and projection matrix of each motion, using H=15 hidden dimensions.

5.2 Speedup

We did experiment on all of the 58 motions with various number of processors on both machine. 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 our algorithm by counting the basic arithmetic operations. 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 3 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 the supercomputer with a maximum of 128 processors. Figure 5 shows the wall clock time and speedup on the multi-core desktop (maximum 4 cores). We also include the theoretical limit from Amdahl's law. Table 2 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.

5.3 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, 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, 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 from the experiments is very tiny, with a maximum 0.3% and mean 0.17%, and no clear evidence of increasing error with more processors. In some cases, the parallel algorithm even found higher (0.074%) likelihood than the serial EM. 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

In order to show the visual quality of the parallel learning algorithm, we observe a case study 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

[2]cobalt.ncsa.uiuc.edu

[3]http://mocap.cs.cmu.edu/

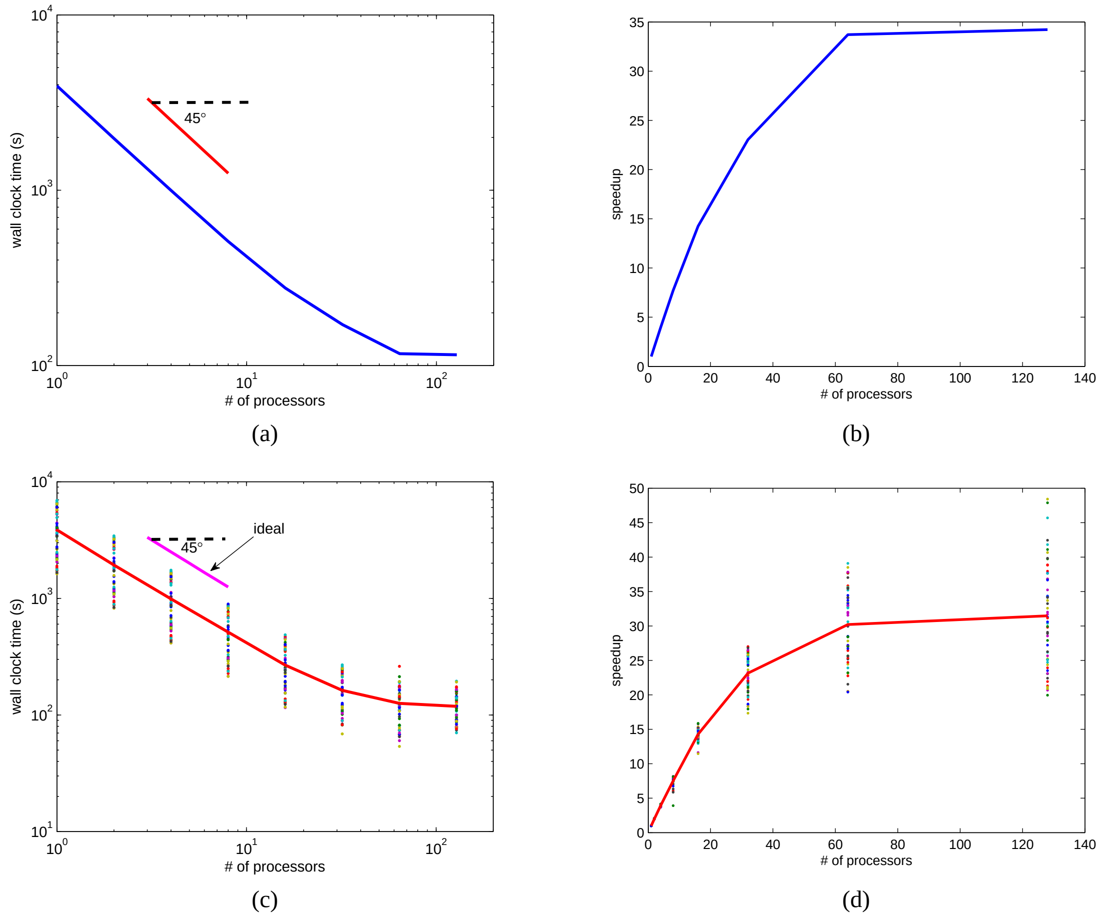


Figure 4: Performance of *Cut-And-Stitch* on multi-processor supercomputer, running on the 58 motions. The Sequential version is on one processor, identical to the EM algorithm. (a) Running time for a sample motion (subject 16 #22, walking, 307 frames) in log-log scales; (b) Speedup for walking motion(subject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

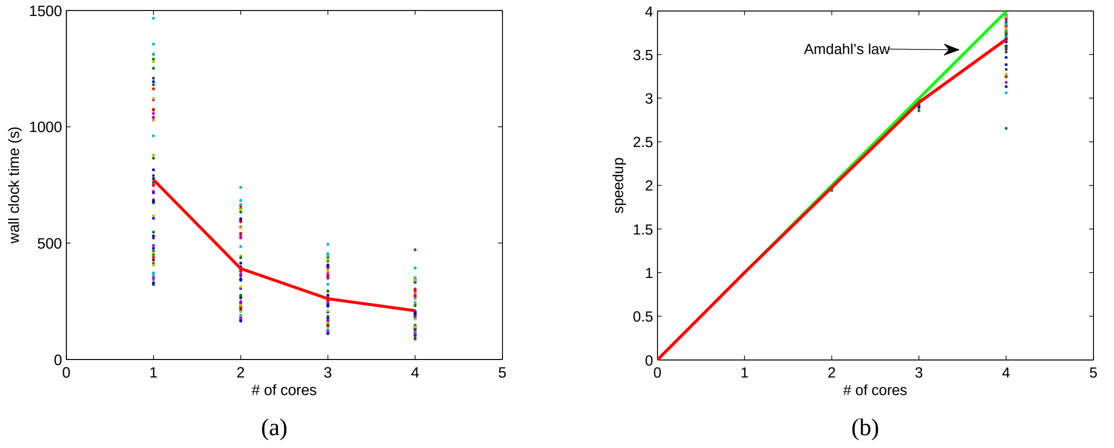


Figure 5: Performance of *Cut-And-Stitch* on multi-core desktop, running on the 58 motions. The Sequential version is on one processor, identical to the EM algorithm. (a) running time for all motions in log-log scales; (b) average speedup for the 58 motions, versus number of cores k.

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

Table 4: Normalized Reconstruction Error

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 reconstructed with model parameters from 4-core computation. Table 4 shows the reconstruction error: both parallel and serial achieve very small error and are similar to each other. Figure 6 and Figure 7 show the reconstructed sequences of the feet coordinates. Note our reconstruction (red lines) is very close to the original signal (blue lines).

6. RELATED WORK

Data mining and parallel programming receives increasing interest. Parthasarathy et al. [2] 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 [13] used OpenMP to parallelize the discovery of frequent patterns in large graphs, showing excellent speedup of up to 30 processors.

Cong et al. [7] 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. [9] developed a parallel algorithm to learn SVM through cascade SVM. Collobert et al. [5] 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. [3] 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 [8] to parallelize a set of learning algorithm such as naïve-Bayes, PCA, linear regression and other similar algorithms [4,12]. 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

In this paper, we explore the problem of parallelizing the learning algorithm for LDS models on symmetric multiprocessor architectures. The main contributions are as follows:

- We propose an approximate parallel learning algorithm for Linear Dynamic Systems, and implement it using the OpenMP API on shared memory machines.
- We performed experiments on a large collection of 58$\times$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 super computer). We showed that our reconstruction error is almost identical to the serial algorithm.

Future work could extend our *Cut-And-Stitch* method to models with similar chain structure such as HMMs. Another direction is to extend *Cut-And-Stitch* for switching Kalman filters (SKF).

Acknowledgements. This material is based upon work supported by the National Science Foundation under Grants No.IIS-0326322. The data used in this project was obtained from mocap.cs.cmu.edu supported by NSF EIA-0196217. This work is also partially supported by the Pennsylvania Infrastructure Technology Alliance (PITA), an IBM Faculty Award, a Yahoo Research Alliance Gift, and a SPRINT gift. 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, Wanhong Xu for his help on processing data, and Erika Chin and David Zhu for their valuable suggestions.

8. REFERENCES

[1] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[2] G. Buehrer, S. Parthasarathy, S. Tatikonda, T. Kurc, and J. Saltz. Toward terabyte pattern mining: an architecture-conscious solution. In *Proc. of the 12th ACM SIGPLAN*, pages 2–12. ACM, 2007.

[3] E. Chang, K. Zhu, H. Wang, H. Bai, J. Li, Z. Qiu, and H. Cui. Parallelizing support vector machines on distributed computers. In J. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *NIPS 20*, pages 257–264. MIT Press, 2007.

[4] 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.

[5] R. Collobert, S. Bengio, and Y. Bengio. A Parallel Mixture of SVMs for Very Large Scale Problems. In T. G. Dietterich, S. Becker, and Z. Ghahramani, editors, *NIPS*. MIT Press, 2002.

[6] C. B. Colohan, A. Ailamaki, J. G. Steffan, and T. C. Mowry. Tolerating dependences between large speculative threads via sub-threads. In *ISCA '06*, pages 216–226. IEEE Computer Society, 2006.

[7] S. Cong, J. Han, and D. Padua. Parallel mining of closed sequential patterns. In *Proc. of the 11th ACM SIGKDD*, pages 562–567. ACM, 2005.

[8] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, 2004.

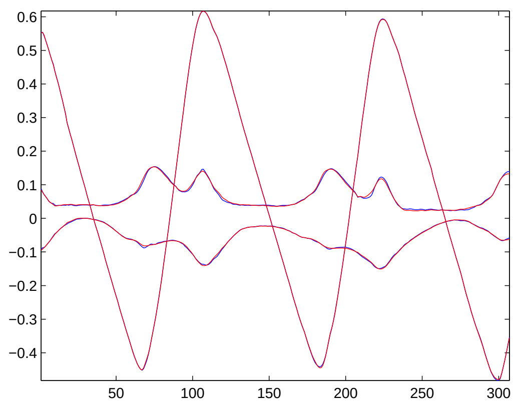
[9] H. P. Graf, E. Cosatto, L. Bottou, I. Dourdanovic, and V. Vapnik. Parallel support vector machines: The cascade SVM. In *NIPS 17*, 2004.

[10] Intel. Intel research advances 'era of tera': www.intel.com/pressroom/archive/releases/20070204comp.htm.

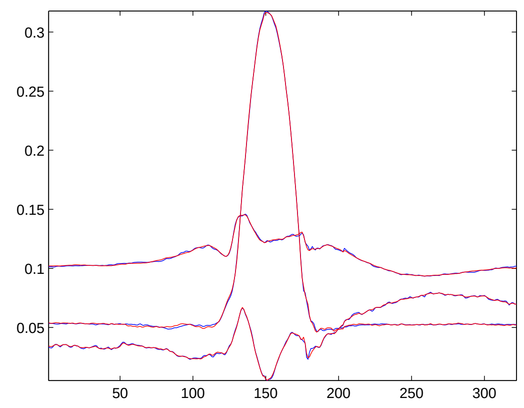
[11] L. Li, J. McCann, C. Faloutsos, and N. Pollard. Laziness is a virtue: Motion stitching using effort minimization. In *Short Papers Proceedings of EUROGRAPHICS*, 2008.

[12] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *HPCA '07*, pages 13–24. IEEE Computer Society, 2007.

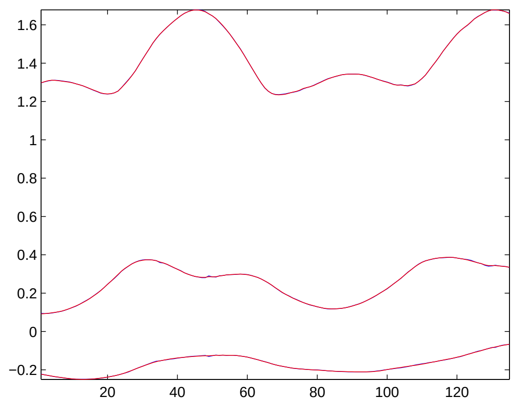
[13] S. Reinhardt and G. Karypis. A multi-level parallel implementation of a program for finding frequent patterns in a large sparse graph. In *IPDPS*, pages 1–8. IEEE, 2007.



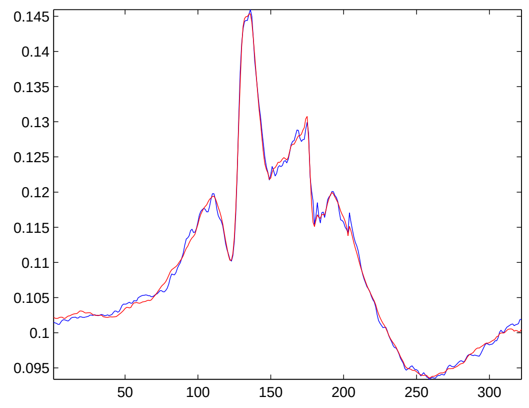
(a) right foot (16#22)



(b) left foot (16#1)

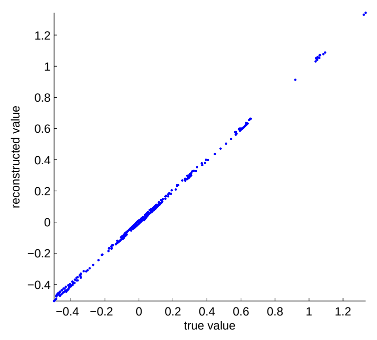


(c) right foot (16#45)

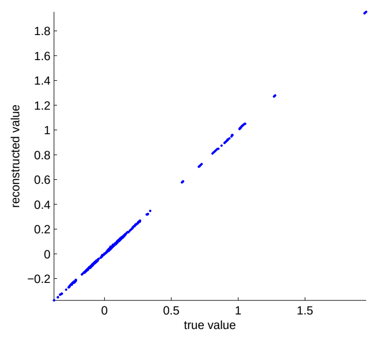


(d) left foot, x (16#1)

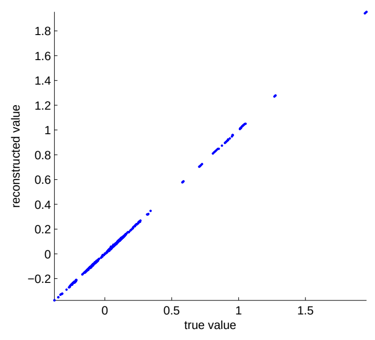
Figure 6: Visual effects: the reconstructed x, y, z coordinates using learned parameters on 4 cores. Horizontal axis is frame index (time tick). (a) right foot coordinates (x,y,z) for the walking motion (subject 16 #22). (b) left foot coordinates for the jumping motion (subject 16 #1). (c) right foot coordinates for the running motion (subject 16 #45). (d) magnification of the x coordinate (the upper curve in (b)). Note that the reconstructed sequences (red lines) are so close to the original signals (blue lines), that the plots looks like a set of purple lines; this illustrates the high accuracy of *Cut-And-Stitch*.



(a)



(b)



(c)

Figure 7: 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).