Parallelizing Incremental Bayesian Segmentation (IBS)

Joseph Hastings <jrh@mit.edu>
Siddhartha Sen <sidsen@mit.edu>

May 19, 2003

Abstract

This paper reports on two approaches to parallelizing the IBS algorithm. One approach uses MPI while the other uses Cilk. Both methods were able to achieve a linear speed up when 2 processors were used, but suffered diminishing returns for more than 2. We present a mathematical derivation of IBS and then discuss the two approaches, commenting on the ease of programming as well as various obstacles we encountered along the way. We conclude that neither MPI or Cilk really achieve their goals of making parallel programming easy for the programmer.

Contents

1 Introduction

This paper discusses an attempt to parallelize the IBS (Incremental Bayesian Segmentation)[1] algorithm. The algorithm was recently developed in order to classify temporal patterns in discrete medical data. It is intended to provide an alternative to Hidden Markov Model learning. Currently HMMs are widely used but are computationally expensive to train. This limits their effectiveness for real-time pattern recognition. We discuss two different approaches to parallelizing IBS: an MPI-based application and Cilk-based application. It is hoped that such applications may help to provide researchers with a viable tool for real-time pattern classificaion.

1.1 Organization

Section 2 develops the mathematics behind IBS. Section 3 discusses the implementation of IBS. Section 3.3 discusses the data-sets used to evaluate the performance of the MPI and Cilk versions of IBS. Section 4 gives a basic overview of MPI and Cilk. Section 5 describes the details of the Cilk version of IBS along with its performance on the test sets. Section 6 discusses the MPI implementation and results. Section 7 concludes with an analysis of the resuls and a list of lessons learned from this project.

2 Background

This section is organized as follows: we first discuss Bayesian Segmentation algorithms in general and the problem they attempt to solve. We next discuss Markov processes and the subset of Bayesian algorithms that attempt to learn them. We then discuss an optimal version of IBS as well as the BCD algorithm that it approximates. Finally we discuss the mathematics used in IBS.

2.1 Bayesian Segmentation

The IBS algorithm is part of a family of Bayesian Segmentation Algorithms that aim to describe a time-series of data in terms of generative probability distributions. In general these algorithms classify various pieces of the time-series as being generated by distinct time-invariant distributions. In other words, they view the sequence as being characterized by a set of time-invariant distributions as well as a state variable that picks which distribution is active at any given moment. In general, any number of different distributions may be used to generate the series. In addition, the portions generated by each distribution may be interleaved in an arbitrary manner. Their goal, given a time-series, is to produce a set of distributions as well as the break-points where the driving distribution switched between members of the set.

The time-series of data can be modeled as a set of random variables $X_1, X_2, ...X_N$ where $X_i \in \{1, 2, ..., s\}$. A particular model M_c describes how the X_i's are related. M_c is a set of discrete probability distributions, as well as a mapping from intervals in the time-series to members of the set. To give an example, a time-series with 5000 elements may be modeled as being generated by 5 different distributions that each produced intervals of length 1000. Alternatively, it could be modeled by a set of 2 distributions that alternated producing intervals of length 100. The purpose

of a Bayesian Segmentation Algorithm is to generate a set of candidate models and to evaluate them according to some criteria, in an effort to provide a best estimate of the underlying model.

A model M_c provides a mechanism for determining X_i. We adopt the notation that actual values from the sequence are represented as x_i. We use the shorthand notation $p(x_{tj})$ to mean $Pr\{(X_t = j)\}$. To avoid confusion between the time-series and the random variables representing the time-series, we use the letter S to represent the actual data as a sequence $S = \{S_i\}$ where $S_i \in \{1, 2, ..., s\}$. The variable s represents the number of discrete states present in the time-series. Without loss of generality, the sequence can be encoded using the integers between 1 and s. S_i and x_i both refer to the i^{th} element of the time-series although S_i views this number as a part of the sequence while x_i views the same number as a sample from the random variable X_i.

2.2 The Markov Property

In full generality each X_i could have a different distribution. Furthermore, X_i could depend on all previous values of S_i (x_i). However, in IBS and related algorithms, we limit our models to those that impose the *Markov property* on the X's. The Markov property states that the probability distribution X_t depends only on the previous k values $\{x_{t-1}, x_{t-2}, ..., x_{t-k}\}$ for some value of k. The value k is called the *Markov Order* of the series. The first major division of Bayesian Segmentation Algorithms is between those that consider $k = 1$ and those that consider arbitrary values of k. IBS is part of the subset that only allows for $k = 1$, a condition that is typically called Markovian. When $k = 1$ the resulting distribution is called a Markov chain. Therefore, for the remainder of the paper, the X_i's will be assumed to form a Markov chain. However, as IBS allows for a set of generative distributions, an additional parameter θ is necessary in order to specify which of the chains is used to generate a particular X_t. For IBS, the Markov condition on the X_i's is equivalent to:

$$\forall t > 0, \quad p(X_t = j|(x_0, x_1, ...x_{t-1}), \overrightarrow{\theta}) = p(X_t = j|x_{t-1}, \theta_{t-1}). \tag{1}$$

The vector $\overrightarrow{\theta}$ denotes which member of the set of distributions is used at any point in time. Note that on the right hand side, the probability only depends on θ_{t-1}. The term $t-1$ is used in place of t because this probability is viewed as a state transition probability for time t that is the result of the state of the model at time $t-1$. Rephrasing M_c in terms of the Markov property means that M_c is a set of Markov probability distributions, which can each be represented as a transition probability matrix. For any pair of values i and j, the probability of the Markov chain moving from state i to state j is given by the $[i][j]$ member of the matrix and is time-invariant. We view M_c as an array of matrices indexed by θ. Therefore:

$$p(X_t = j|x_{t-1}, \overrightarrow{\theta}) = M_{\theta_{t-1}}[x_{t-1}][j]. \tag{2}$$

Note that in this interpretation of M_θ the rows sum to one and have non-negative components. For the remainder of the paper we will ignore the subscript of θ and treat the model M_c as if it includes the necessary information encapsulated in $\overrightarrow{\theta}$. We can write the formula above as $p(x_{tj}) = M_c[x_{t-1}][j]$. The role of a Bayesian Segmentation Algorithm is therefore to produce M_c given a time-series S.

2.3 Bayesian Model Selection

Given the constraint on M_c that the X_i's be Markovian, we view the universe as consisting of some set M of Markov processes and an oracle that chooses when and for how long each process is allowed to generate samples. Bayesian segmentation algorithms attempt to find some set of matrices $\hat{M}$ that approximate the true set M, as well as to define the break-points produced by the oracle. From a statistical point of view, M_c is an estimate of the true set of distributions M, and traditional statistical techniques can be employed.

2.3.1 Off-line and On-line Model Selection

A Bayesian approach to this model selection problem compares two possible models M_1 and M_2 in terms of their *likelihood* given the data. We use the informal notation $p(M_c|S)$ to mean the likelihood of the particular set M_c given the sequence of observations S. In general these algorithms can be grouped into *off-line* and *on-line* versions. Off-line variants are allowed to consider the entire sequence S of data while on-line variants attempt to approximate the solution by incrementally considering new data. Both types of analysis will be necessary in order to fully explain IBS. IBS is the on-line approximation to Bayesian Clustering by Dynamics (BCD), which is an off-line Bayesian Clustering Algorithm. The term *clustering* in place of segmentation indicates that multiple segments in the time-series may be mapped to the same underlying probability distribution. Therefore M_c is viewed as a clustering of the time-series, as it describes a set of distributions and the (possibly disjoint) intervals that each distribution was allowed to generate.

We now develop the theory behind IBS by starting from the intractable optimal solution, moving towards BCD, the off-line approximation, and finally, towards IBS as an on-line approximation to BCD.

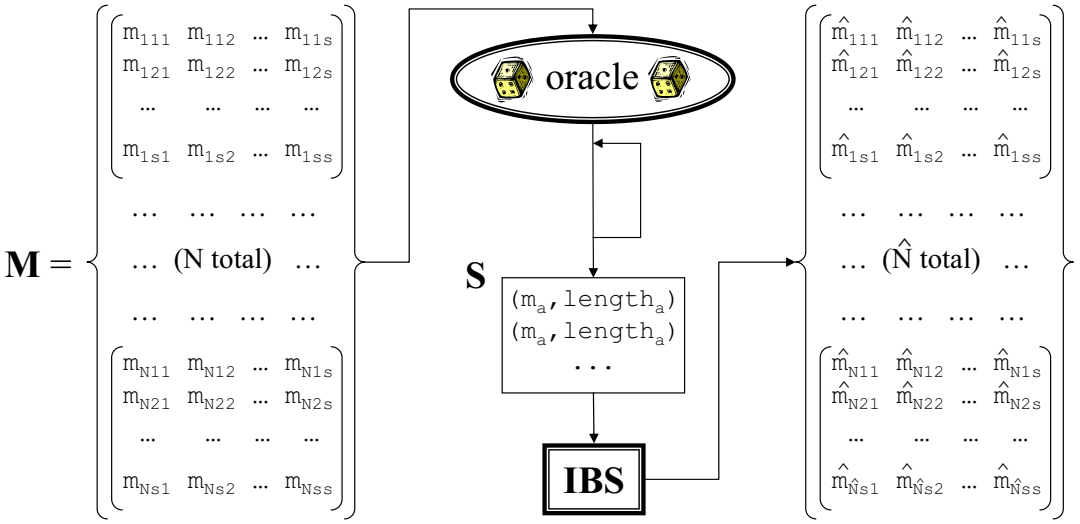

Figure 1: An oracle produces S using the set M of Markov matrices. IBS attempts to segment S and reverse-engineer M.

2.3.2 Partitioning and Clustering

A model M_c (along with the information encoded in $\overrightarrow{\theta}$) defines a partitioning of S. The partitioning of S maps each element to an equivalence class identified by the corresponding element of $\overrightarrow{\theta}$. The vector $\overrightarrow{\theta}$ will have between 1 and N distict elements, depending on how many equivalence classes are used. In general, we use the term partitioning to think of the abstract notion of dividing S into equivalence classes and the term clustering to refer to the process of learning a particular $\overrightarrow{\theta}$ in terms of the break-points generated by the oracle. We call a subsequence of S with identical values of θ an *execution* of the Markov process encoded in the θ^{th} matrix. We use the notation S_θ to indicate the subsequences of S associated with the execution encoded by θ.

2.3.3 Executions and Clusters

We will soon characterize the behavior of an execution in terms of the state transitions that it contains. However, we also need to include the transition between the last member of an execution and the first member of the next execution (if it is not the last member of S). That is, at the point at which the oracle chooses a new matrix, the value of θ changes. However, the transition between the previous state and the new state depends on the old value of θ. Similarly, an execution does not contain a transition into its starting state. We amend the definition of S_θ slightly to include the boundary-case conditions. In order to specify that the transition into a state is included, but not the transition out of that state, S_θ contains a 0 after the first element of the next execution. Also, the subsequences from each of the separate executions with the same θ are concatenated (separated by the 0 indicating not to include the transition into or out of the 0). Therefore S_θ is a sequence that specifies the same set of transitions as all of the executions mapped to θ.

The term clustering was defined above to refer to the information encoded by $\overrightarrow{\theta}$. A *cluster* is defined to be a set of executions related by having the same value of θ. Bayesian Clustering Algorithms are a subset of Bayesian Segmentation algorithms in which the oracle is allowed to return to a previous matrix rather than always generating new matrices for each execution. In the diagram below, $A, B, ..., H$ are executions, and (AB) is also an execution. In this case there are 3 equivalence classes. Vieweing the model M_c as generating S, we call A, B, D, G a cluster and C, E another cluster. $S_{\theta 1}$ would be the sequence containing A then B, followed by the first element of C and then a 0. The next element would be the first element of D. After the first element of E a 0 would follow and the next element would be the start of G. $S_{\theta 1}$ would conclude with the end of G.

2.4 An Optimal Clustering Algorithm

In principle, we could generate all possible partitions of S. There are $g(n)$ of them and each one implies a particular M_c (along with the corresponding $\overrightarrow{\theta}$). We could then select the best model through an exhaustive search. However, the total number of ways to partition S is related to the Stirling numbers of the second kind and is given by:

$$g(n) = \sum_{k=1}^{n} \frac{1}{k!} \sum_{i=0}^{k-1} (-1)^i \binom{k}{i} (k-1)^n \tag{3}$$

This quantity (3) is known to be super-polynomial and BCD is an attempt to greatly reduce the search space. However, before we discuss BCD, we will provide a mathematical framework for comparing two possible models in terms of their likelihood given the sequence. We first need to describe how to generate M_θ given the cluster of executions encoded with θ. Next, we describe how, given two alternative sets of matrices M_1 and M_2, we decide which is a better model for S. An optimal algorithm would apply the equations developed in the next section to every possible partition, and choose the globally optimal set of matrices.

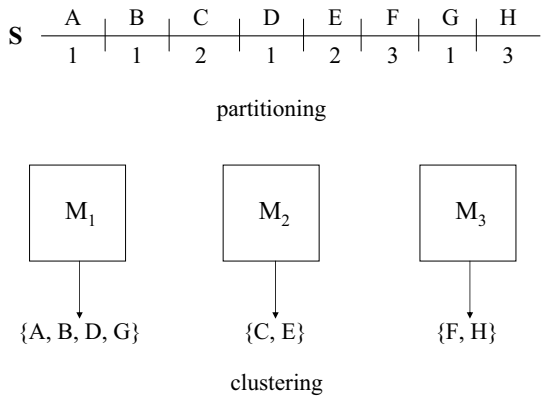

Figure 2: Partitioning of S into three equivalence classes.

2.4.1 Maximum Likelihood Matrix

In order to solve the clustering problem optimally, we first need to solve the problem of generating the optimal matrix in order to best describe the executions that it generated. To rephrase, we need a process for finding the matrix with the maximum likelihood of generating the executions that it has been assigned by a particular partitioning. For example in the example above, we need to generate the M_2 that would be most likely to generate the executions C and E.

Based on the Markovian assumption, the maximum likelihood estimate described above depends only on the transitions that occur in the sequence (along with the edge-cases as described in the definition of an execution). These transitions can be stored in a *count matrix* which is an $s \times s$ matrix N_θ in which n_{ij} is the number of times that $S_{\theta_{t-1}} = i$ and $S_{\theta_t} = j$ in the execution (correctly parsing the 0's artificially inserted as described above). Another implication of the Markov assumption is that the storage required for a cluster depends only on s^2 and not on the number of transitions (assuming that none of the entries would overflow).

It is a well known result that the optimal transition probability matrix for generating a particular count matrix is formed by normalizing each of the rows. Equivalently:

$$\hat{P} = (\hat{p}_{ij}) = \frac{n_{ij}}{\sum_j n_{ij}}. \tag{4}$$

For example, if $s = 5$, $C = 1, 2, 3, 4, 5, 1$ and $E = 5, 1, 3$ then:

$N_{AC} =$

	1	2	3	4	5
1	0	2	1	0	0
2	0	0	1	0	0
3	0	0	0	1	0
4	0	0	0	0	1
5	1	0	0	0	0

$\Rightarrow \hat{P}_{AC} =$

	1	2	3	4	5
1	0	2/3	1/3	0	0
2	0	0	1/1	0	0
3	0	0	0	1/1	0
4	0	0	0	0	1/1
5	1/1	0	0	0	0

2.4.2 Bayesian Paramteter Estimation

We can extend this formula to include prior knowledge of the probabilities for each of the transitions in the traditional Bayesian manner. This will also prevent us from generating 0's in our matrices. Having a probability matrix with 0's is troublesome because we do not know if the oracle will eventually provide a sequence that would be classified using a 0-probability transition. We incorporate prior knowledge by creating a hypothetical time-series of length $\alpha + 1$ in which the α transitions create a hypothetical count matrix with entries α_{ij}. Our count matrix N is now formed by adding α_{ij} to each of the entries. Our new estimate for P is known as the *Bayesian estimate* and has:

$$\hat{P} = (\hat{p}_{ij}) = \frac{\alpha_{ij} + n_{ij}}{\sum_j (\alpha_{ij} + n_{ij})}. \tag{5}$$

Suppose that for our example we use a uniform prior with every $\alpha_{ij} = 1$. Our modified count table would be:

$N_{AC} + N_\alpha =$

	1	2	3	4	5
1	1	3	2	1	1
2	1	1	2	1	1
3	1	1	1	2	1
4	1	1	1	1	2
5	2	1	1	1	1

$\Rightarrow \hat{P}_{AC} =$

	1	2	3	4	5
1	1/8	3/8	2/8	1/8	1/8
2	1/6	1/6	2/6	1/6	1/6
3	1/6	1/6	1/6	2/6	1/6
4	1/6	1/6	1/6	1/6	2/6
5	2/6	1/6	1/6	1/6	1/6

The α's are known as hyper-parameters and they are viewed as free parameters for the algorithm. In practice they would most likely vary with time as they tend to over-damp probability estimates by continually over-estimating low-probability events. If the count matrices ever become completely dense the priors may be abandoned completely. Although decisions regarding the hyper-parameters are important for the machine learning aspects of IBS, they are not relevant for the parallelization analysis and will not be discussed further.

2.4.3 A Bayesian Approach to Model Evaluation

Using the formulas we derived above, we could define an optimal clustering algorithm in which we generate each of the possible clusterings, used the count matrices to generate the transition

probability matrices, and then evaluate the likelihood of the model. We now develop a formula for performing this evaluation. The first step applies Bayes rule:

$$p(M_c|S) = \frac{p(M_c)p(S|M_c)}{p(S)}. \tag{6}$$

The denominator $p(S)$ is constant for each of the perspective models and we therefore find the optimal clustering by maximizing $p(M_c)p(S|M_c)$. $p(M_c)$ is a measure of our *a priori* probability for the particular clustering M_c. In general we have no particular reason to favor one model over another, although separate families of Bayesian algorithms treat this term as variable with various metrics for choosing between models. In our case we can treat this term as a constant for each model and focus on maximizing $p(S|M_c)$. Applying (5) we have that the optimal model maximizes:

$$p(S|M_c, \overrightarrow{\alpha}) = \prod_{\theta} \prod_{t \in t_\theta} \hat{P}_\theta[S_t][S_{t+1}]. \tag{7}$$

While mathematically correct, the above formulation is subject to underflow as the result of multiplying many small numbers. As the result, the sum that is actually maximized is:

$$l(S|M_c, \overrightarrow{\alpha}) = \sum_{\theta} \sum_{t \in t_\theta} log(\hat{P}_\theta[S_t][S_{t+1}]). \tag{8}$$

In principle we could enumerate all possible partitions of S and evaluate (8) for each partition, choosing the best possible M_c. However, as mentioned above, the number of partitions of S grows extremely quickly and this would be intractable. We now discuss the BCD algorithm which is an approximate solution to this optimization problem.

2.5 Bayesian Clustering by Dynamics

BCD operates in two phases. In the first phase a break-point detection algorithm is applied to S. The algorithm itself is not specified by BCD, but a particular variant used in IBS will be discussed in (2.6.3). In practice a number of different methods would be used, perhaps starting with a naive uniform partition and then using results of the previous iteration to guide the break-point detection. This algorithm partitions S in a linear fashion, that is, forming clusters that consist of a single execution. In the language of BCD these initial clusters are called *segments*. The initial break-point detection breaks S into c segments for some $c \ll N$. The segments are each stored in a count matrix, with the possible addition of priors.

The second phase of BCD takes a set of segments and attempts to form an optimal set of clusters by combining sets of segments. In other words, it modifies the $\overrightarrow{\theta}$ vector by taking segments with different values of θ and assigning them the same value. This process is known as subsumption and has the property that the count matrix of the subsumed cluster is the sum of the two count matrices for each of the original clusters. In practice $\overrightarrow{\theta}$ is not stored explicitly as only count matrices are ever needed for the actual computations.

The process of combining segments to form clusters is agglomerative, meaning that segments are never broken apart but can be subsumed together. Finding the optimal agglomeration of segments into clusters forms a much smaller solution space than the set of all models considered in the optimal algorithm, but is again intractable. BCD uses a heuristic search through the space of possible subsumptions to find an approximately optimal solution.

The heuristic search described above has three important components: a metric for evaluating whether a particular subsumption improves the model, a method of generating potential pairs to subsume, and a termination condition. Various choices for these criteria give rise to a family of related algorithms. We discuss the choices made by BCD with an emphasis on applicability to IBS.

2.5.1 Evaluating a clustering

As in the optimal algorithm described above, the parameter being optimized in BCD is $l(S|M_c)$. However, rather than evaluating all possible models, this score is used to incrementally improve the clustering. Given the current set of clusters $M_c = \{M_1, M_2, ..., M_c\}$ the algorithm tests whether some particular pair of clusters M_i and M_j should be combined into M_{ij}. The clusters M_i and M_j are subsumed if:

$$l(S|\underbrace{\{M_1, ..., M_{ij}, ..., M_c\}}_{c-1}) \overset{?}{>} l(S|\underbrace{\{M_1, ..., M_i, ..., M_j, ..., M_c\}}_{c}) \tag{9}$$

The derivation of $l(S|M_c)$ is given in [1] and will be summarized below. First, for clarity, we define the following:

Expression	Meaning
$S \equiv \{S_1, ..., S_k, ..., S_c\}$	having same transitions
$S_k \mapsto N_k \mapsto \hat{P}_k$	cluster produces counts and a Markov matrix
$\alpha_{kij} = \alpha_{ij}$	in cluster k
$n_{kij} = n_{ij}$	in cluster k
$\alpha_{ki} = \sum_j \alpha_{kij}$	row precision in cluster k
$n_{ki} = \sum_j n_{kij}$	occurrences of i in cluster k
$m_k = \sum_i n_{ki} = \lvert S_k \rvert$	length of cluster k
$m = \sum_k m_k = \lvert S \rvert$	length of S
$\alpha = \sum_k \alpha_{ki}$	global precision
$C = \{C_1, ..., C_m\}$, $C_i \in \{1, 2, ..., c\}$	$C_i = j$ if S_i is a member of cluster j

C is a hypothetical vector specifying the cluster membership of each element of S and is the same as the $\overrightarrow{\theta}$ vector from the previous section. The quantity α is called *global precision* and represents the influence of the prior estimates on the probabilities. The remainder of this paper assumes that the α_{kij} follow a Dirichlet distribution, which is discussed in [4]. Equivalently, referring back to (1), we say that $p(X_t = j|x_{t-1})$ is conditionally independent of t given C_{t-1}. In other words, C_{t-1} can be viewed as a state variable, specifying which model to use to generate X_t given x_{t-1}.

This observation encourages us to express $p(S|M_c)$ as a function of S and C, which we can do as follows:

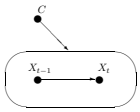

Figure 3: X_{t-1} and X_t are Conditionally Independent Given C_{t-1}.

$$p(S|M_c) = f(S, C)g(S, X_{t-1}, X_t, C). \tag{10}$$

Where:

$$f(S, C) = \frac{\Gamma(\alpha)}{\Gamma(\alpha + m)} \prod_{k=1}^{c} \frac{\Gamma(\alpha_k + m_k)}{\Gamma(\alpha_k)}. \tag{11}$$

This expression comes from the assumption that the lengths of clusters obey a Beta distribution. Note that this expression does not depend on actual values within S but only on the number and relative lengths of the clusters. Treating the α's as a Dirichlet probability distribution, this term gives the *a priori* likelihood that the sequence would be partitioned in k clusters each with length m_k. Alternatively, this can be viewed as a penalty term that discourages S from being overly partitioned. In this light, this term is an application of Occum's razor, causing BCD to favor the smallest number of free parameters that adequately describe the sequence.

$$g(S, X_{t-1}, X_t, C) = \prod_{k=1}^{c} \prod_{i=1}^{s} \frac{\Gamma(\alpha_{ki})}{\Gamma(\alpha_{ki} + n_{ki})} \prod_{j=1}^{s} \frac{\Gamma(\alpha_{kij} + n_{kij})}{\Gamma(\alpha_{kij})}. \tag{12}$$

Here the first product is over the set of cluster, the second is over the rows of the associated count matrices, and the third is over each of the entries in that row. This formula comes from the assumption that the parameters in the c Markov matrices follow Beta distributions and was first constructed in [**?**].

Note that if we set all of the prior estimates to 0, this would reduce to:

$$g(S, X_{t-1}, X_t, C) = \prod_{k=1}^{c} \prod_{i=1}^{s} \prod_{j=1}^{s} \frac{\Gamma(n_{kij})}{\Gamma(\sum_j n_{kij})} \tag{13}$$

This equation looks very similar to the formulation of $\hat{P}_k$ except that the entries are normalized using the Gamma function. This expression comes from the assumption that the transition probabilities within each row obey a Beta distribution.

Now, given two particular models, we can evaluate their likelihood solely on the basis of the priors and the generated count matrices. The equations above are converted into their log form for actual numerical computation. This conversion introduces the log-gamma function, lgf leading to:

$$\mathrm{lf}(S, C) = \frac{\mathrm{lgf}(\alpha)}{\mathrm{lgf}(\alpha + m)} \sum_{k=1}^{c} \frac{\mathrm{lgf}(\alpha_k + m_k)}{\mathrm{lgf}(\alpha_k)}. \tag{14}$$

$$\mathrm{lg}(S, X_{t-1}, X_t, C) = \sum_{k=1}^{c} \sum_{i=1}^{s} \frac{\mathrm{lgf}(\alpha_{ki})}{\mathrm{lgf}(\alpha_{ki} + n_{ki})} \sum_{j=1}^{s} \frac{\mathrm{lgf}(\alpha_{kij} + n_{kij})}{\mathrm{lgf}(\alpha_{kij})}. \tag{15}$$

$$l(S|M_c) = \mathrm{lf}(S, C) + \mathrm{lg}(S, X_{t-1}, X_t, C). \tag{16}$$

2.5.2 Generating a Space of Potential Models

In BCD the calculation of $l(S|M_c)$ is performed on only a subset of all possible models. Given an initial clustering, it calculates a pair-wise distance between each pair of distributions and builds a sorted list of all such pairs. The first two entries have the globally smallest distance, and every pair after the first is the pair of matrices with the next lowest distance. Note that each matrix appears $c - 1$ times in this list. The distance used by BCD is based on the Kullback-Liebler distance:

$$d(p_1, p_2) = \sum_{i=1}^{s} p_{1_i} log \frac{p_{1_i}}{p_{2_i}} \tag{17}$$

This equation is not necessarily symmetric, but can me made into a valid metric by calculating: $\frac{d(p_1,p_2)+d(p_2,p_1)}{2}$. BCD walks along the sorted list and tests whether subsuming the first two matrices would produce a better score according to (16). If the matrices are subsumed, it then re-scans the list removing the original matrices and merging in pairs including the newly created matrix. If the resulting score of the subsumption is lower, it moves to the next pair. BCD terminates when it does not find any pair that can be subsumed to increase the score.

2.5.3 Complexity and Effectiveness of BCD

The BCD algorithm as described involves several important decisions. First, the method of choosing an initial partition is extremely important. Additionally, if a particular number of clusters is desired, this initial partition can be chosen to optimally partition the space. In many cases, if clusters are short, they will be subsumed with nearly any other short cluster, meaning that the order that this space is searched can have a large effect on the final clustering. Finally, the choice of priors and their relative weights can bias the algorithm towards particular ends of the solution space. In [1] the authors point out that uniform priors encourage clusters to be subsumed into a single cluster, despite the intuitive notion that they are uninformative. They also prove that the time complexity of the algorithm is $O(c^4 s^2)$ where s is the number of total unique states and c is the number of initial clusters. However, this time does not include the initial segmenting which is non-trivial in practice. The IBS algorithm is an attempt to approximate the behavior of BCD while operating in an on-line fashion, meaning that the average incremental work required when given a new sample from S is $O(1)$.

2.6 IBS

2.6.1 IBS Motivation

Unlike the optimal clustering algorithm and BCD, the IBS algorithm attempts to operate on an infinite sequence of data. Rather than viewing S as a data-set, it views the sequence as samples from some infinite process. In addition, the samples are generated at a particular time frequency such that IBS must be able to completely process a state transition before receiving the next data point. In principle a buffer could be employed to smooth any temporal variations in processing speed, but this buffer must be finite. While BCD was given the task of explaining S by some agglomerative combination of given clusters, IBS is responsible for both segmenting the series and then grouping the segments into related clusters.

The motivation behind IBS is rooted firmly in a Bayesian view of the world. If the oracle producing the infinite sequence has a finite set of time-invariant Markov processes, and he uses a time-invariant probability distribution to choose the order in which the processes execute, and the execution time itself can be produced according to a time-invariant probability distribution, then maximum likelihood estimates for each of those parameters have a logical interpretation. Unfortunately, it is not feasible to allow all of those parameters to range over arbitrary values. Instead IBS relies on *a priori* estimates for each of the parameters. In particular, they are all assumed to follow some Beta distribution. Taking each of these priors, and assuming that all parameters follow a Beta distribution with Gaussian noise, IBS formulates a Bayesian model selection problem that attempts to find the best explanation for the observed sequence. As discussed in the summary of BCD, even intuitively uninformative priors, such as a uniform distribution over possible values, can strongly bias the predictions made by a Bayesian process. A frequentist objection to this algorithm is that the implicit assumptions hardcoded into the formulations prevent its output from having any particular meaning. However, rather than debating the philosophy of the problem, we move on to its theory and implementation.

2.6.2 Structure of IBS

IBS consists of several tiers of nested loops. At the highest level, it processes samples from the infinite sequence and runs a break-point detection algorithm. Each time it decides that it has reached a break-point, it attempts to classify the segment of transitions between the previous break-point and the current point. This classification must consider subsuming the latest segment into each of the previous clusters. In addition, it must test the likelihood that the segment has come from a previously unobserved process. In theory, if the number of distinct processes were known, or conjectured, the probability that a new segment is novel could be made conditional over all of the finite values for the number of processes. However, in general, it is difficult to paramaterize this distribution. A further complication is whether we believe that the probability of seeing a new cluster is constant or decreases with time. If the oracle is indeed using a time-invariant method of choosing which process to execute then intuitively this should be a decreasing function and we should adjust our likelihood score. Unfortunately these second-order effects are not completely debugged in IBS and are not included in the version that was used to parallelize the code.

In general, without making assumptions about the distinct probability distributions used by the oracle, it is difficult to know whether the distance between two estimated probability distributions

is due to a fundamental difference in the driving process or in sampling variation. The current version of IBS assumes that the set M of matrices is generated at random to uniformly sample the space of all s dimensional matrices. This issue is addressed further in section (3.3).

2.6.3 Break Point Detection

The outer loop of IBS runs a break-point detection algorithm designed to divide the time-series into segments that are then classified into clusters. The break-point algorithm works by accumulating the cumulative probability of the sequence and breaking when it exceeds certain bounds. The bounds are based on the expected value of the cumulative probability, conditional on the assumption that the probabilities being used are correct. If the oracle chooses a new matrix and this assumption is no longer correct, the difference between the assumed (old) distribution and the new distribution will cause the score to quickly exceed its bounds. The speed with which it exceeds those bounds is based on the average distance between matrices in M.

At each transition three variables are updated: the actual cumulative probability, the estimated expected value of this probability, and the estimated variance of this quantity. A break point occurs when the actual cumulative probability deviates from the expected value by more than β standard deviations. This algorithm depends on several parameters and assumptions, which are discussed below.

The first consideration in the break-point detection is how to estimate the probability of a particular transition. This choice is made by building a count matrix for the current segment, including a hypothetical sequence of priors, and using (5). For each transition, the corresponding entry in the $\hat{P}$ matrix is used as an estimate of the probability of that transition occurring. A variable known as *score* is updated by adding the log of $\hat{P}[x_{i-1}][x_i]$. *score* is known as the log-likelihood of the sequence, although the fact that $\hat{P}$ changes at every point makes this a slight deviation from the normal definition. Simultaneously, we calculate the expected value of this parameter, using the basic definition of expectation shown below. Similarly we calculate the variance. These three parameters are defined as follows:

$$c = -log(\hat{P}_{t-1}[x_{t-1}][x_t]) \tag{18}$$

$$E[c] = \sum_j p(c_j)c_j = -1 \cdot \sum_j (\hat{P}_{t-1}[x_{t-1}][j]) \cdot log(\hat{P}_{t-1}[x_{t-1}][j]) \tag{19}$$

$$var(c) = \sum_j p(c_j)(c_j - E[c])^2 = \sum_j (\hat{P}_{t-1}[x_{t-1}][j]) \cdot (-log(\hat{P}_{t-1}[x_{t-1}][j]) - E[c])^2 \tag{20}$$

$$score_t = -1 \cdot \sum_{i=1}^{t} log(\hat{P}_{i-1}[x_{t-1}][x_t]) \tag{21}$$

$$mean_t = -1 \cdot \sum_{i=1}^{t} \sum_{k=1}^{s} \hat{P}_{i-1}[x_{t-1}][k] \cdot log(\hat{P}_{i-1}[x_{t-1}][k]) \tag{22}$$

$$variance_t = \sum_{i=1}^{t} \sum_{k=1}^{s} (-log(\hat{P}_{i-1}[x_{t-1}][k]) - mean_t)^2 \cdot (\hat{P}_{i-1}[x_{t-1}][k]) \tag{23}$$

$$sd_t = \sqrt{variance_t} \tag{24}$$

The break-point criteria is to break if:

$$|score_t - mean_t| > \beta sd_t \tag{25}$$

Each time a break-point is triggered, the count matrix used to generate $\hat{P}$ is then passed to the clustering stage of the algorithm.

2.6.4 IBS Clustering

Unlike in BCD where the entire set of segments is known prior to the clustering phase, IBS must perform segment clustering in an incremental fashion. It keeps a library of existing clusters, and each time a new segment is detected by the break-point phase, it is tested for subsumption with each of them. In addition, a likelihood score is computed for the augmented set of matrices including this segment as a new member of the set. In future versions of IBS this score will be offset by the probability of seeing a novel process which is the only parameter in IBS that varies with time.

Fortunately, this test can be performed fairly efficiently and in parallel. First, note in (12) that the likelihood of the clustering given that the new segment is added as a standalone cluster, is simply the old score plus the double product over the new cluster. Similarly, the score for each of the subsumed variants is equal to the previous score, minus the score for the candidate cluster, plus the score of the double product for the combined count matrix. Each of these scores depends only on a single count matrix, and they can each be evaluated in parallel.

2.6.5 IBS Summary

For the practical application of IBS to a real-world problem, a number of issues need to be considered. In particular, the value β of the break-point detection threshold and the usage of priors are free-parameters. In addition, pre-loading the library of matrices with domain-specific distributions may be necessary if the sequences are too short to fully characterize the underlying probability distribution. Also, the length of sequences needed grows as a function of s because the number of independent parameters in each matrix grows. In general, the longer the length of each execution, the more power IBS has to differentiate between clusters. Also, if executions are short and the information content is dominated by priors, all resulting matrices look similar and will be subsumed together.

3 Code Overview

In this section, we present an overview of the high-level control flow of the IBS program. Then, we suggest different ways of parallelizing the IBS code and discuss the key issues that need to be addressed for this parallelization to work. Finally, we describe the data sets that we generated to evaluate the performance of the parallelized IBS code (both the Cilk and MPI versions).

3.1 High-Level Control Flow

The IBS application consists of a main loop that reads the input file and runs the break-point detection algorithm. Each time a break-point is detected, the count matrix for the previous segment is passed to a function called check_out_process. This function either adds the segment to the library of matrices as a new cluster or subsumes the segment into an existing matrix. Inside check_out_process it enters a loop in which it calls compute_subsumed_marginal_likelihood. This function takes the new segment as well as a candidate cluster from the library and computes the likelihood that would be produced by subsuming the segment and the candidate. The highest score amount all possible candidates and the standalone score indicates how the new segment should be incorporated into the library.

3.2 Parallelizable Computation

In general, we considered several different approaches to parallelizing IBS. First, we considered different techniques for improving the break-point detection algorithm by running it in parallel. However, after some code profiling (in which break-points were detected but ignored by not passing the segment to check_out_process) we discovered that it ran at a rate nearly equal to the rate at which the input file could be read by the operating system. One interesting idea that could be explored would be using the probability distributions to speculatively evaluate the most likely transitions in parallel and then use a communication from the processor that happened to parse the transition first to select from the possibilities. However, given the results of the profiler and the limited time-frame for the project, we decided to focus on parallelizing other portions of the code.

The first step to parallelizing the clustering portion of the code was a mathematical analysis of (12). We realized that the outer product, evaluated over each cluster, could be evaluated in parallel and depended only on the count matrix for a particular cluster. Similarly, when comparing two possible clusterings, the only difference between them would involve a single subsumption. Finally, the encoding of the globally optimal subsumption could be expressed entirely in an integer index into the library of matrices (or -1 if the matrix should be added as a stand-alone cluster).

The next question to consider was how to store the library of matrices. One solution would be to have each processor store some subset of the matrices, and then perform that subset of the subsumptions. However, in this case, calculating the score for adding the new segment as a standalone cluster would require adding scores from each of the processors. We then realized that the library of matrices itself is not particularly large, and we could build it in a replicated manner at each node without any additional communication overhead. Since each node would need to know the new segment in order to compute its subset of subsumptions, as long as it learned the globally optimal index at the end of each classification, each node could build the same resulting library. Note that the memory requirements for the library depend only on the number of clusters and s and not on the number of transitions (assuming that none of the counts overflow). Overflowing is not considered, because in practice, if the count matrices contain enough entries to overflow a particular member, the resulting probabilities are extremely stable with respect to the pertubations caused by an incremental transition. A real-world application of IBS would stop subsuming elements into a matrix when the possible changes in the estimated probabilities were too small to matter.

After we decided to store the library in a replicated fashion, and to perform some subset of the subsumption tests on each node, writing the MPI version of the code was a straightforward applica-

tion of using library functionality. However, writing the Cilk version required a completely different view of the problem as the set of abstractions it provides did not cooperate with this model.

3.3 Data Sets

In order to evaluate the performance of the parallel version of IBS for different numbers of processors, 4 data sets were generated. Each data set was constructed by choosing a certain number of random Markov matrices, with a particular dimension, and then interleaving their execution. Interleaving was performed by choosing a random permutation of the matrices, iterating over the set in that order, and generating a time-series based upon that distribution for a certain number of transitions. This entire process was repeated in a number of cycles. The exact values used in the 4 data sets are listed in the table below.

File	Matrices	States	Transitions	Cycles
50.lisp	16	50	100	20
100.lisp	32	50	100	20
150.lisp	50	50	1000	50
200.lisp	50	99	1000	50

Choosing a random Markov matrix with s states is equivalent to choosing s vectors in s dimensions that have non-negative components that sum to 1. These vectors can also be viewed as points along the $s-1$ dimensional simplex which has the equation: $x_1 + x_2 + ... + x_s = 1$ and each $x_i > 0$. A number of well-known techniques exist for sampling this space with varying levels of bias towards particular types of vectors. The method used picked s pseudo-random numbers on the interval $[0-1]$ and sorted them. The distance between successive pairs of elements, inserting an artificial 0 before the first element to generate the first distance, was normalized to produce a vector that is believed to be an un-biased sampling of this space. The actual behavior of IBS makes no particular assumptions about the character of the matrices it learns, although having a truly uniform sampling of matrices would make it less likely that two matrices appear similar enough to be classified together.

4 Methods of Parallelization

In this section, we provide some background information on the two methods of parallelization used on the IBS C code: MPI and Cilk.

4.1 MPI

MPI, the Message Passing Interface, is a standard library that defines communication protocols for use in parallel computing. It consists of a set of library functions that help programmers write portable parallel code. Unfortunately it has a fairly low-level API meaning that programmers must write a great deal of code in order to parallelize an algorithm. For example, in order to parallelize IBS it was necessary to write marshalling code for transfering data structures. For more information about MPI refer to http://www-unix.mcs.anl.gov/mpi/.

4.2 Cilk

Cilk is a language for multithreaded parallel programming based on ANSI C that is very effective for exploiting highly asynchronous parallelism, which can be difficult to write using message-passing interfaces like MPI [2]. Originally developed by the Supercomputing Technologies Group at the Laboratory for Computer Science at MIT, Cilk differs from other multithreaded programming systems by being algorithmic in nature—that is, it employs a scheduler in its runtime system that allows the performance of programs to be estimated using abstract complexity measures [7]. As a result, the runtime system can guarantee both predictable and efficient performance in its execution of these programs.

The underlying philosophy of Cilk is that a programmer should focus on writing the program to expose its parallelism and exploit locality to the best extent possible. The runtime system, in turn, should be concerned with scheduling the computation to run efficiently on a given platform, taking care of issues such as load balancing, paging, and communication protocols [2]. The runtime system achieves this by implementing a scheduling policy based on randomized *work-stealing*.

When a Cilk program is run, the user has the option of specifying the number of "processors" or "worker threads" to create via a command-line argument (if no number is specified, a default value is used). The runtime system then creates this many worker threads (or *workers*) and schedules the user computation on these workers; it leaves the task of scheudling the threads on the physical processors to the operating system itself. During the execution of a Cilk program, if a worker runs out of work to do, it chooses another worker at random (called the *victim*) and tries to *steal* work from the victim. If the steal is successful, the worker begins working on the stolen piece of work; otherwise, it picks another worker (uniformly at random) to be the victim and continues work-stealing until the steal is successful.

Cilk's work-stealing scheduler executes any Cilk computation in nearly optimal time [2]. If we let T_P be the execution time of a given computation on P processors (or worker threads), then the following two definitions are immediate: 1) T_1 is the total time needed to execute the computation on one processor (this is equivalent to the *work* involved in the computation), and 2) T_∞ is the time needed to execute the computation on an infinite number of processors (this is equivalent to the program's *critical-path length*. Given these definitions, the work-stealing scheduler of Cilk executes a computation on P processors in time

$$T_P \leq T_1/P + O(T_\infty) \tag{26}$$

which is asymptotically optimal [2]. The constant factor hidden in the big O is often close to 1 or 2 [7], so in practice we have that $T_P \approx T_1/P + T_\infty$.

Converting a serial C program into its parallel equivalent in Cilk usually involves no more than the inclusion of the `cilk.h` header file and the addition of a few key words: `cilk`, `spawn`, and `sync`. Section 5.1 describes the modifications made to the IBS C code to convert it into a parallel Cilk program.

5 Cilk Version

In this section, we describe the modifications made to the IBS C code to convert it into a parallel IBS Cilk program, based on the strategy outlined in Section (3.2). We then present some perfor-

mance metrics obtained from running the program on a 32-processor supercomputer owned by the Supercomputing Technologies Group at MIT's Lab for Computer Science.

5.1 Code Modifications

As we stated in Section 4.2, the modifications required to convert a C program into its parallel equivalent in Cilk are minimal. The general procedure is to identify the C methods that will be parallelized (as well as the methods that call them), and convert these methods into Cilk methods. In the case of the IBS C code, our main target for parallelization is the computation performed by the `compute_subsumed_marginal_likelihood()` method (see section 3.2). In order to perform the matrix subsumptions in parallel, we insert the `spawn` keyword before the call to this method:

```
for (i = 0; i < stored_processes; i++) {
  spawn compute_subsumed_marginal_likelihood(proc,
                                             get(processes,i),
                                             copy_process_list(processes));
}

sync;
```

A spawn in Cilk is the parallel analog of a C function call, in that execution proceeds to the child when a procedure is spawned. Unlike a C function call, however, the parent in a spawn can continue to execute in parallel with the child [2]. In the code above, a child is spawned for each of the stored Markov processes (represented internally as matrices) so that the matrix subsumptions can occur in parallel with each other. The `sync` statement ensures that all of the spawned children have completed their execution before the parent continues with its own execution—thus, it acts like a fence across all worker threads of the Cilk program that may be executing these children.

A few additional tricks are required to prevent the worker threads of the IBS Cilk program from stepping on each other's toes. First, instead of passing a pointer to the global list of Markov processes (`processes`) to the spawned child procedures, a shallow copy of the list is made (via the call to `copy_process_list()` above) and this copy is passed. Also, while in the original C code the call to `compute_subsumed_marginal_likelihood()` returns a score that is then compared to the highest score seen so far (replacing the latter value if it is found to be higher), in the Cilk version a global score structure is used for this purpose instead. Since the spawned `compute_subsumed_marginal_likelihood()` procedures directly update the global score structure, we must enforce mutual exclusion between write accesses to this structure. In Cilk, this is achieved by using a special lock of type `Cilk_lockvar`, as seen in the following code fragment:

```
// Declare a global score lock
Cilk_lockvar score_lock;

...
// Initialize the score lock
Cilk_lock_init(score_lock);
...
```

```
void update_global_score(double score, ...)  {
  if(score <= global_score.score) {
    return;
  }

  // Acquire the lock
  Cilk_lock(score_lock);
  ...
  // code to update the global score value
  ...
  // Release the lock
  Cilk_unlock(score_lock);
}
```

5.2 Performance Results

We now present the results obtained from running the IBS Cilk program on different data sets and different numbers of processors (or worker threads). The machine used to run the program is a 32-processor SGI Origin 2000 shared-memory supercomputer supporting sequential consistency; the individual processors run at 195 MHz each [3].

The chart below shows the program's elapsed time for the `50.lisp` and `100.lisp` data sets, described in Section 3.3. For each data set, separate trials were performed using 1, 2, 4, 8, 16, and 32 processors.

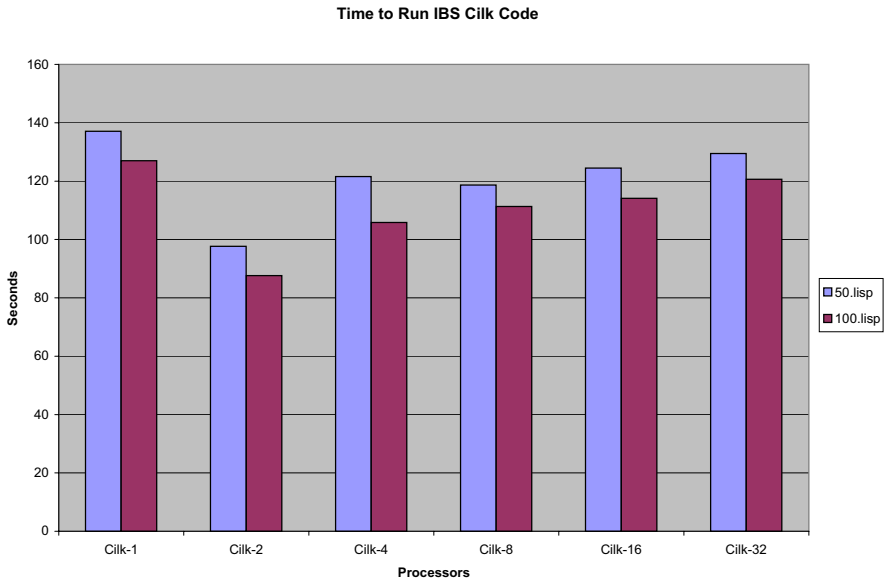

Figure 4: Elapsed time of IBS Cilk code run on `50.lisp` and `100.lisp` data sets.

The results above indicate that the parallelism of the IBS Cilk program is about 2; that is, the parallelism of the program is best exploited by using 2 processors to run it. Beyond 2 processors,

the elapsed time of the program actually increases (but never above T_1, the time taken to run the program on 1 processor). We believe that the reason for this slowdown is the overhead incurred by the Cilk runtime system—the extra cycles needed to manage the created worker threads (recall that the number of worker threads is equivalent to the number of processors specified by the user). It seems as if the individual calls to `compute_subsumed_marginal_likelihood()` do not take as long as we originally thought, and therefore 2 worker threads are able to handle the workload efficiently, the second making only a few steals from the first. When more than 2 worker threads are used, however, the majority of them are unable to find any work to steal, and so they enter a continuous work-stealing loop in which they repeatedly try and fail to steal work from other threads. The end result is that the Cilk runtime system has to spend a significant amount of time handling these work-stealing threads, while the threads themselves are unable to steal (and hence perform) any useful work.

5.3 Adaptive Parallelism

Currently, the user of a Cilk program has to specify the number of processors to run the program on—if no number is specified, a default of 2 processors is used. One way to automate the process of finding an ideal allocation of processors is to incorporate *adaptive parallelism* into the Cilk runtime system. In other words, we would like the runtime system to be able to increase or decrease the number of processors being used by a Cilk job dynamically. The hope is that these dynamic adjustments will allow the runtime system to adapt to the actual parallelism of the program being run. For example, in the case of the IBS Cilk program running on the `50.lisp` or `100.lisp` data sets, the runtime system would ideally settle down to an allocation of 2 processors for the Cilk job.

The task of incorporating adaptive parallelism into Cilk is three-fold; the problem can be divided into the following three subtasks:

1. Implement "static" or "one-time" adaptive parallelism by intelligently determining the initial number of processors to run a Cilk job on if this number is not specified by the programmer. We refer to this number as a Cilk job's *fair share* of processors, defined further below.
2. Implement dynamic adaptive parallelism by increasing the number of worker threads assigned to a Cilk job (without exceeding its fair share) if the *instantaneous parallelism* of the job (defined below) is sufficient to merit the increase.
3. Implement dynamic adaptive parallelism by decreasing the number of worker threads assigned to a Cilk job if the instantaneous parallelism of the job is insufficient to make adequate use of the current thread allocation.

Task 1 above depends directly on our definition of a program's fair share of processors, and tasks 2 and 3 depend on our method of measuring the instantaneous parallelism of a given Cilk job. We now explain the meaning of these terms, based on the results obtained in [5] and [6].

Consider the problem of scheduling J jobs on P processors. At any given time, each job $j = 1, 2, ..., J$ has an *instantaneous parallelism* or *desire* d_j which represents the number of threads that the job currently desires (this is equivalent to the number of threads that are either running or ready to run) [6]. Let m_j be the number of processors allocated by the job scheduler to job j at any given time—this is also known as the job's *allotment*. Then, the vector of allotments $\langle m_1, m_2, ..., m_j \rangle$ is

a legal allocation if $\sum_{j=1}^{J} m_j \leq P$. In addition, an allocation is said to be *efficient* if the following two conditions are held [6]:

- If there exists a job j such that $d_j < m_j$, then $d_i \leq m_i$ for $i = 1, 2, ..., J$.
- If there exists a job j such that $d_j > m_j$, then $\sum_{j=1}^{J} m_j = P$.

Finally, an allocation is said to be *fair* if the following condition hods [6]:

- If there exists a job j such that $d_j > m_j$, then $m_i \leq m_j + 1$ for $i = 1, 2, ..., J$.

Our goal now is to modify the Cilk scheduler so that it dynamically estimates the desires of Cilk jobs and allocates processers to these jobs in a fair and efficient manner. [6] describes such a scheduler, using the "steal rate" of a job j to approximate its desire d_j. In particular, the scheduler in [6] uses the number of consecutive unsuccessful steals to approximate the fraction of time a processor spends stealing. If this fraction is too high, then the scheduler knows that the parallelism of the job is insufficient, and so it decreases the number of processors allocated to the job; if the fraction is too low, then the scheduler knows that the parallelism of the job is greater than the number of allocated processors, so it allows the estimated desire of the job to increase.

The next step from here is to implement the scheduler in [6] and try to improve its algorithm for measuring the desires of Cilk jobs. Unfortunately, due to time constraints, this supplement to our project could not be completed and will have to be postponed for future work.

6 MPI Version

In this section, we describe the modifications made to the IBS C code to convert it into a parallel IBS MPI program, based on the strategy outlined in Section (4.2). We then present some performance metrics obtained from running the program on the test data-sets.

6.1 Code Modifications

Three major code modifications were necessary to write an MPI version of IBS. First, marshalling code was written in order to broadcast a segment from the root node to each of the other nodes. Secondly, the main loop was modified so that the root node parsed the file and performed the break-point detection while each of the other nodes waited to be sent a segment. Finally, compute_subsumed_marginal_likelihood was modified to perform only a subset of the computations based upon the rank of the processor. At the end of this function, MPI_Allreduce was used to calculate and distribute the globally optimal index of the matrix subsumption.

6.1.1 Marshalling

It was necessary to write functions to broadcast a count matrix between the root node and all of the other nodes. Portions of this code are shown below:

```
#define BROADCAST_SIZE 7600
int broadcast_buffer[BROADCAST_SIZE];

typedef struct broadcast {
  int start;
  int end;
  process* proc;
  bool done;
} broadcast;

void broadcast_segment(process* proc,
       const int start,
       const int end,
       bool done) {
  int i,j,k;
  broadcast_buffer[0] = start;
  broadcast_buffer[1] = end;
  broadcast_buffer[2] = proc->index;
  broadcast_buffer[3] = proc->position;
  broadcast_buffer[4] = proc->type;
  ...
  MPI_Bcast(broadcast_buffer,BROADCAST_SIZE,MPI_INT,0,MPI_COMM_WORLD);
}

broadcast* receive_broadcast(int r) {
  int i,j,k;
  process* proc = make_process2(-1,-1);
  if(rank == 0) {
    die(__LINE__,__FILE__);
  }
  MPI_Bcast(broadcast_buffer,BROADCAST_SIZE,MPI_INT,0,MPI_COMM_WORLD);
  bc.start = broadcast_buffer[0];
  bc.end = broadcast_buffer[1];
  bc.proc = proc;
  bc.done = broadcast_buffer[k++];
  return &bc;
}
```

6.1.2 Main Control Loop

The outer main loop was altered slightly such that only the root node began parsing the input file. The other nodes entered an infinite loop where they waited to receive broadcasts in order to do subsumption work. The exited the loop when a flag was set in the broadcast structure.

```
if(rank == 0) {
  db = fopen(argv[1],"r");
  ...
}
else {
   while(true) {
    pBC = receive_broadcast(rank * 17);
    ...
   }
}
```

6.1.3 check_out_process

check_out_process performs the segment classification and was re-written slightly to perform the broadcast and then calculate only a subset of the subsumption scores. MPI_Allreduce was then used to find the global optimum and the relevant changes were made to the library of matrices.

```
broadcast_segment(proc, start, end, done);
local_score.score = compute_marginal_likelihood(processes);
local_score.index = -1;
for(i=0; i<stored_processes; i++) {
  if(i % np != rank)
    continue;
  score = compute_subsumed_marginal_likelihood(proc,
                                               get(processes,i),
                                               &subsumed_result,
                                               processes);
  if(score >= local_score.score) {
    local_score.score = score;
    local_score.index = i;
    best_cluster = subsumed_result;
    candidate = cluster;
  }
}
...
MPI_Allreduce(&local_score,
              &global_score,
              1,
              MPI_DOUBLE_INT,
              MPI_MAXLOC,
              MPI_COMM_WORLD);
if(global_score.index == -1) {
  ret = store_new_process(proc,processes);
}
else {
```

```
    candidate = get(processes,global_score.index);
    ret = store_subsumed_process(proc,
                                 candidate,
                                 best_cluster,
                                 processes);
}
...
```

Similar code appeared in the branch of the main loop executed on the non-root nodes. In this manner the subsumption was performed in parallel after the broadcast of the latest segment. Each node performed the same actions on its local libary of matrices and the result was a distributed replicated set of matrices.

6.2 Performance Results

The MPI version of IBS saw a linear speedup for each of the data-sets when the first additional processor was added. However for more than 2 processors, while there was a slight speedup, the effect was negligible. This is probably due to the fact that the broadcast between the first two processors occureed on the same machine while adding more processors required using TCP/IP. Also the marshalling code was highly un-optimized and required sending 7600 bytes. Given more time to tune the code we would have attempted to greatly reduce this number. Some easy optimization would have been to avoid sending one element per row in each of the matrices and relying on the fact that $s-1$ of them detemine the last value. Furthermore, only the count matrices really needed to be sent as the probability estimates ($s^2 \cdot 8$ bytes per matrix) could be quickly computed at each node based on the integer counts. Finally the counts could have been compressed by using short integers rather than long integers. We estimate that with all of these simple optimizations, the marshalling code would have run at least twice as fast. A graph is shown below illustrating the performance on each of the data-sets.

7 Conclusion

7.1 Summary of Results

Both the IBS and Cilk version of IBS were able to achieve nearly linear speedup over the serial algorithm when 2 processors were used. For MPI the code ran slightly faster on 3 and 4 processors, but inefficient marshalling code prevented the additional computational resources from providing more of a benefit. For Cilk the results were actually worse when more than 2 processors were used as the overhead of the scheduling system caused the performance to suffer.

Overall we feel that the experience of attempting to parallelize a real-world application was a worthwhile endeavor, although it proved to be extremely frustrating at times. Writing the MPI version was extremely difficult both conceptually and pragmatically, as debugging was extremely tedious. Writing the Cilk version was much easier although we were forced to deal with the traditionally difficult issue of synchronization and locking.

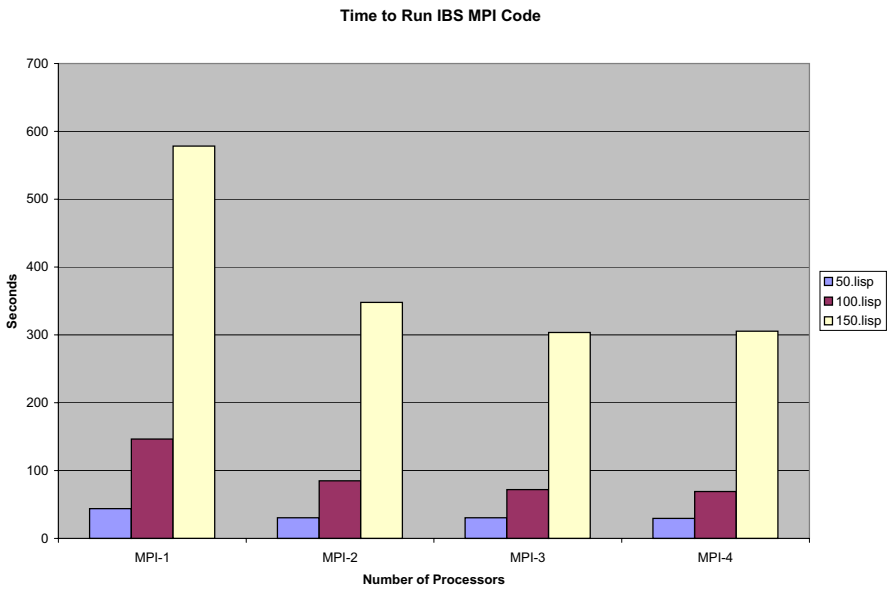

Figure 5: Elapsed time of IBS MPI code run on `50.lisp` and `100.lisp` data sets.

We feel that the process of parallelizing the algorithm involved a series of difficult steps. First, we had to diagnose the bottlenecks in the serial algorithm and find places where parallelization would possibly yield an improvement. Next we were forced to think abstractly about how to best organize the communication between nodes in order to perform the parallel tasks concurrently. Next, and perhaps the most painful portion of the experience, we needed to write MPI code to enact our algorithm. Writing this code produced many unexpected bugs each of which required hours worth of debugging to resolve. Part of this process was made worse by the constant crashing of nodes on beowulf and the instability of PBS. Finally, towards the end of the term, we had to compete with other students for time on the machine to run test batches and measure actual performance.

Writing the Cilk version, while not nearly as difficult as MPI, introduced its own issues. First, we needed to port the code from C++ to C, which was a painful and monotonous task. Next we had to work to get the code to compile in the proper environment with the Cilk compiler. This introduced frustrating library and header file issues but was eventually successful. Finally, the Cilk code seems farily unstable and would produce segmentation faults for seemingly no reason. One of the biggest lessons that we bring from this experience is that parallel programming has a long way to go before it becomes as straightforward as writing and debugging serial code.

References

[1] M. Ramoni — P. Sebastiani — P. Cohen. Bayesian clustering by dynamics. *Machine Learning*, 47(1):91–121, 2002.

[2] Supercomputing Technologies Group. *Cilk 5.3.2 Reference Manual.* MIT Lab for Computer Science, November 2001.

[3] J. Laudon and D. Lenoski. The sgi origin: a ccnuma highly scalable server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[4] P. Sebastiani and M. Ramoni. Incremental bayesian segmentation of categorical temporal data. 2000.

[5] B. Song. Scheduling adaptively parallel jobs. Master's thesis, Massachusetts Institute of Technology, January 1998.

[6] R. D. Blumofe — C. E. Leiserson — B. Song. Automatic processor allocation for work-stealing jobs.

[7] R. D. Blumofe — C. F. Joerg — B. C. Kuszmaul — C. E. Leiserson — K. H. Randall — Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Santa Barbara, California, July 1995.