A Parallel Implementation of a Hidden Markov Model with Duration Modeling for Speech Recognition

Carl D. Mitchell, Mary P. Harper, Leah H. Jamieson, and Randall A. Helzerman
School of Electrical Engineering, Purdue University
West Lafayette, IN 47907-1285
{cdm,harper,lhj,helz}@ecn.purdue.edu

Corresponding Author
Leah H. Jamieson
1285 Electrical Engineering
Purdue University
West Lafayette, IN 47907-1285
lhj@ecn.purdue.edu
(phone) 317-494-3653
(fax) 317-494-6440

†This work is supported in part by Purdue Research Foundation, NSF grant number IRI-9011179, and NSF Parallel Infrastructure Grant CDA-9015696.
Abstract

Hidden Markov models (HMMs) are currently the most successful paradigm for speech recognition. Although explicit duration continuous HMMs more accurately model speech than HMMs with implicit duration modeling, the cost of accurate duration modeling is often considered prohibitive. This paper describes a parallel implementation of an HMM with explicit duration modeling for spoken language recognition on the MasPar MP-1. The MP-1 is a fine-grained SIMD architecture with 16384 processing elements (PEs) arranged in a 128x128 mesh. By exploiting the massive parallelism of explicit duration HMMs, development and testing is practical even for large amounts of data. The result of this work is a parallel speech recognizer that can train a phone recognizer in real time. We present several extensions that include context dependent modeling, word recognition, and implicit duration HMMs.

1 Introduction

While hidden Markov models (HMMs) have been a popular and effective method of recognizing spoken language, their complexity has been limited because of computational costs. Despite the fact that explicit duration continuous HMMs more accurately model speech than HMMs with implicit duration modeling, the cost of accurate duration modeling is often considered prohibitive.

Explicit duration modeling has been shown to increase the effectiveness of hidden Markov models in automatic speech recognition [1, 2], but at a large computational cost. If \( D \) is the maximum duration allowed for any state, \( N \) is the number of states, \( K \) is the average number of predecessors per state, and \( T \) is the length (in frames) of the utterance, then a serial implementation of an HMM recognizer has complexity \( O(NDKT) \). Typical values for a phone recognizer are: \( N = 200, K = 1.333, D = 32, \) and \( T = 500 \). The time needed to evaluate several competing models is often prohibitive for a serial machine. This paper presents a parallel implementation of a phone recognition system that has algorithmic complexity \( O((\log N + \log D)KT) \) using \( ND \) PEs. Our parallel implementation accommodates more complex models by utilizing more PEs, so that there is no longer a tradeoff between accurate modeling and the ability to achieve real-time recognition.

The organization of this paper is as follows. A review of HMMs and an example of using HMMs for speech
recognition are given in section 2. Section 3 reviews the algorithms needed for training and recognition. Some related work on parallelizing HMMs is given in section 4. In section 5, we describe our parallel speech recognition system, including implementation details. Complexity is discussed in section 6. In section 7, some extensions of our basic algorithms are pointed out, including the ability to model phones in context. Section 8 lists the key points of our implementation and summarizes our work.

2 Hidden Markov Models for Speech

A Markov model is a finite state machine where the next state depends only on the current state. Associated with each arc of the finite state network (a directed cyclic graph) is the probability of making the given transition. Consider the phone Markov model of spoken language shown in figure 1. A phone is the smallest building block of a language. For example, the word *psychic* has 5 phones: /s/ /a/ /k/ /ih/ /k/.

A hidden Markov model (HMM) is a Markov model where the state of the process (represented by a node in the graph) is not directly observable. However, the process emits an observation (e.g., vector of measurements) each time it arrives in a new state. The observation emitted from a particular state depends on the output distribution of that state.
2.1 Example: Isolated Word Recognition

Suppose that the goal is to recognize a single word that belongs to a small vocabulary. Each word in the vocabulary can be modeled as a succession of phones, where each phone is modeled by a single HMM state as shown in figure 2. We need a measure of the input speech signal that will allow us to compare the utterance to each vocabulary word model. For simplicity, let us use the spectral energy of the signal. Specifically, the speech will be segmented into 10 ms frames, and the energy in each of four spectral bands will be found for each frame. For each segment of speech, the observation vector is modeled by a 4-dimensional Gaussian density. If the input utterance has a duration of 0.7 seconds, then there are 70 frames of speech represented by the set of 70 energy vectors (i.e., the observation sequence). Given the input utterance, our goal is to find the most likely path from the initial state to the final state (and hence the recognized word) that accounts for the observations.
2.2 Explicit Duration Modeling

State duration is the time that the stochastic process stays in one state, which is measured as the number of observations emitted from the process while in the state. In a standard HMM, the process is said to “stay” in state $j$ if a transition is made from $j$ to itself. Let $a_{i,j}$ denote the probability that the next state will be $j$, given that the current state is $i$. If the process has just arrived in state $j$, then the probability that state $j$ will have duration 1 is the probability of leaving at the next time increment, $(1 - a_{j,j})$. The probability of the process staying in state $j$ for exactly two measurements is $a_{j,j}(1 - a_{j,j})$. Thus the distribution of state duration is geometric with parameter $a_{j,j}$. This method of modeling duration is referred to as implicit duration modeling.

Unfortunately, the geometric distribution is often not the best distribution to model the duration of speech events. In an explicit duration HMM [3, 4], the number of observations generated from a state is modeled explicitly with a distribution such as gamma or Gaussian, and self loops are not needed (i.e., $a_{j,j} = 0$). Since it is usually possible to find a parametric distribution that is closer than the geometric distribution to the true distribution of speech events, explicit duration modeling can increase recognition accuracy.

The improved modeling associated with explicit duration modeling comes at the cost of much higher computational costs. If $D$ is the maximum state duration, then both training and recognition time are increased by a factor of $D$.

2.3 Elements of a Speech HMM

The elements of a hidden Markov model can be divided into two categories: those pertaining to the observations being modeled and those that define the structure of the model.

**Observations:** The process being modeled (e.g., the speech signal) is measured at fixed intervals (typically every 10 milliseconds). Each measurement yields a feature vector, where the features commonly represent spectral information. These feature vectors can be modeled directly by continuous distributions, or discretized and modeled by probability mass functions. In a *continuous HMM*, the feature vectors constitute the observations of the HMM. In a *discrete HMM*, the feature vectors are discretized using vector quantization, and the quantized values serve as the HMM observations. Let $M$ denote the number of elements in the feature vector. The number of measurements
Table 1: The elements of a hidden Markov model.

<table>
<thead>
<tr>
<th>Term</th>
<th>Definitions common to implicit or explicit duration HMMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N )</td>
<td>Number of states in the HMM (typically ( \sim 200 ) for context independent phone recognition, and ( \sim 10,000 ) for context dependent phone recognition or word recognition).</td>
</tr>
<tr>
<td>( K )</td>
<td>Average number of predecessors for a given state (typically ( \sim 1.333 ) for context independent phone recognition).</td>
</tr>
<tr>
<td>( a_{i,j} )</td>
<td>Probability that the next state will be ( j ), given that the current state is ( i ).</td>
</tr>
<tr>
<td>( T )</td>
<td>Number of observations in an utterance (typically ( \sim 500 )).</td>
</tr>
<tr>
<td>( O_t^T )</td>
<td>Observation sequence, ( {O_1, \ldots, O_T} ).</td>
</tr>
<tr>
<td>( M )</td>
<td>Dimension of the observation vector (typically ( \sim 30 )).</td>
</tr>
<tr>
<td>( b_j(O_t) )</td>
<td>Probability that observation vector ( O_t ) will be emitted, given the current state is ( j ). We assume ( b_j(O_t) ) is Gaussian with mean ( \mu_j ) and diagonal covariance matrix ( \Sigma_j ).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Term</th>
<th>Definitions for explicit duration HMMs only</th>
</tr>
</thead>
<tbody>
<tr>
<td>( D )</td>
<td>Maximum state duration measured as the number of observations (typically ( \sim 32 )).</td>
</tr>
<tr>
<td>( d_j(\tau) )</td>
<td>The probability that ( \tau ) output vectors will be emitted when the process visits state ( j ).</td>
</tr>
<tr>
<td>( \lambda )</td>
<td>( {{a_{i,j}}, {d_j(\tau)}, {\mu_j}, {\Sigma_j}} ), the complete set of model parameters.</td>
</tr>
</tbody>
</table>

The average number of predecessors, \( K \), is the number of states divided by the number of transitions. In figure 1, \( N = 140 \) and \( K = 185/140 = 1.32 \). The probability of going from state \( i \) to state \( j \) is denoted by \( a_{i,j} \). The probability of observing a particular observation vector, \( O_t \), in state \( j \) is \( b_j(O_t) \). For a continuous HMM, \( b_j(\cdot) \) is a parametric density. For a discrete HMM, \( b_j(\cdot) \) is a set of discrete probabilities. In this paper, we assume that \( b_j(\cdot) \) is Gaussian with mean vector \( \mu_j \) and covariance matrix \( \Sigma_j \). We also assume that the covariance matrix \( \Sigma_j \) is diagonal, which is a common assumption for speech recognition. The elements of the diagonal are denoted by \( \sigma^2_{j,m} \) for \( m = 1, \ldots, M \). In an explicit duration HMM, \( d_j(\tau) \) is the probability that \( \tau \) measurements will be taken while the process remains in state \( j \) (i.e., the duration of state \( j \)). The probability mass function represented by \( d_j(\tau) \) as a function of \( \tau \) is truncated for \( \tau \) greater than \( D \). Each time that state \( j \) is visited, the duration of the visit is chosen according to the state’s duration distribution, \( d_j(\tau) \). Lastly, the set of all HMM parameters, \( \{\{a_{i,j}\}, \{d_j(\tau)\}, \{\mu_j\}, \{\Sigma_j\}\} \) is collectively referred to as \( \lambda \). The elements of an HMM are summarized in table 1.
3 Algorithms for Speech Recognition [5]

Four algorithms form the core of the HMM training and recognition task: Viterbi, forward, backward, and Baum Welch. Once we know the model parameters, $\lambda$, the Viterbi algorithm is all that is needed for recognition. The Viterbi algorithm, an example of dynamic programming, is used to determine the most likely state path that accounts for the observation sequence. Because the next state is dependent only on the current state, the most likely state path can be calculated recursively.

The model parameters are estimated from a large collection of training data that is assumed to be representative of the recognition task. Training the HMM essentially amounts to estimating the probability of an event (e.g., a state transition) by the normalized frequency of the event in the training data. The transition from the initial state to the first state in phone 1 (see figure 1) is an example of an event. The expected number of occurrences of a particular event, which will be referred to as the event count, can be calculated using the partial path probabilities found in the forward and backward algorithms.

The forward algorithm is similar to the Viterbi algorithm, except that the contributions of all partial paths are summed, rather than extending only the single most likely path. Since the sum is needed rather than the max, logarithms are not convenient and scaling must be used to prevent underflow. The output of the forward algorithm is the $N \times T$ array of partial path probabilities. This array is referred to as the alpha trellis or the forward trellis (see figure 3).

The backward algorithm is identical to the forward algorithm except that the sum of partial paths is propagated backwards in time. The output of the backward algorithm is an $N \times T$ array of probabilities for the partial paths that extend to the end of a sentence. This array is referred to as the beta trellis or the backward trellis.

In the Baum-Welch re-estimation algorithm, the count of a particular event is found by summing the probabilities of all paths for which the event occurs and dividing by the sum of all path probabilities. These counts can be expressed efficiently in terms of the forward and backward partial path probabilities. Normalizing the counts yields an improved estimate of the model parameters. This process of refining an estimate of the model parameters is iterated until it converges.
3.1 Training A Hidden Markov Model

The HMM parameters, $\lambda$, are estimated by making an initial guess and then refining that estimate using the characteristics of a set of training data. Training the HMM consists of iterating a 3-step process. First, the probabilities of partial paths beginning at $t = 1$ are found using the forward algorithm on the training sequence, $O^T_1 = \{O_1, ..., O_T\}$. Next, the probabilities of partial sequences that end at time $t = T$ are calculated using the backward algorithm. Finally, the forward and backward probabilities are combined to form a new estimate of the model parameters, $\lambda^{new}$. $P(O^T_1 | \lambda^{new})$ is guaranteed to be greater than or equal to $P(O^T_1 | \lambda)$. This process of improving the model parameters is iterated until the probability of the observation sequence converges to a local maximum. By choosing a good initial estimate for $\lambda$, the local maximum will not differ significantly from the global maximum.

Efficient estimation of $\lambda^{new}$ is possible by utilizing two recursions. A forward recursion finds the probability of a subsequence of observations beginning with $O_1$. Define $a_t(j)$, the forward probability, as follows:

$$a_t(j) = Pr(O_1, O_2, ..., O_t, \text{ state } j \text{ ends at time } t | \lambda)$$

Let state 1 be the initial state and state $N$ be the final state.

$$a_0(1) = 1$$

$$a_0(j) = \sum_{i=1}^{N} a_0(i) a_{i,j} d_j(0) \text{ for } j = 2, ..., N$$

For $t = 1$ to $T$,

$$a_t(j) = \sum_{i=1}^{N} \sum_{\tau=1}^{D} a_{t-\tau}(i) a_{i,j} d_j(\tau) \prod_{s=1}^{\tau} b_j(O_{t-s+1})$$

The two-dimensional array of forward probabilities can be viewed as a trellis, as shown in figure 3. Each entry of the $d^{th}$ column is, in general, dependent on all of the previous $D - 1$ columns. If $D = 1$, then $d_j(\tau) = 1.0$ for $\tau = 1$ and $d_j(\tau) = 0.0$ for $\tau > 1$. In this case, the explicit duration HMM reduces to the standard implicit duration HMM. Figure 3a depicts a trellis for an HMM with duration modeled implicitly and figure 3b shows the Viterbi or forward trellis for an explicit duration HMM.
Figure 3 also reveals the implementational difference between implicit and explicit duration modeling when calculating the Viterbi (or forward) trellis. For the implicit duration case, all predecessors to the node currently being updated are in the previous column. For explicit duration modeling, the predecessors belong to the previous \( D \) columns.

![Diagram of Viterbi and forward trellis](image)

Figure 3: The predecessors necessary to update one node of the Viterbi or forward trellis are shown for a) implicit duration modeling, and b) explicit duration modeling. In the implicit duration model, state duration is accounted for by the self loops in the model. In the explicit duration model, the model parameters \( d_j(\tau) \) reflect the probability of spending time \( \tau \) in state \( j \).

Just as a forward recursion finds the probability of a subsequence of observations beginning with \( O_t \), a backward recursion determines the probability of a subsequence of observations ending with \( O_T \). Define \( \beta_t(i) \), the backward probability, as follows:

\[
\beta_t(i) = Pr(O_{t+1}, O_{t+2}, \ldots, O_T | \text{state } i \text{ ends at time } t)
\]

\[
\beta_T(N) = 1
\]

\[
\beta_T(i) = \sum_{j=1}^{N} a_{i,j} d_j(0) \beta_T(j) \text{ for } i = 1, 2, \ldots, N - 1
\]
Then for $t = T - 1$ to 0,

$$\beta_t(i) = \sum_{j=1}^{N} \sum_{\tau=1}^{D} w_{t,i,j}(\tau) \text{ for } i = 1, \ldots, N$$

where $w_{t,i,j}(\tau) = a_{i,j} d_j(\tau) \beta_{t+\tau}(j) \prod_{s=1}^{\tau} k_{j}(O_{t+s})$

**Figure 4:** Summing the probability of all paths for which state $j$ begins at time $t$ and lasts for 4 observations.

Suppose that we have an estimate, $\lambda$, of the HMM parameters. An improved estimate, $\lambda^{\text{new}}$, can be obtained by taking the ratio of expected counts. For example, to estimate the probability that state $j$ has a duration of four, the “count” for this event is found by summing the probability of all paths for which state $j$ occurs with a duration of four. Figure 4 illustrates how all contributing paths are taken into account for a particular $t$. The operations depicted in figure 4 are repeated for each possible $t$. After all paths are included, the resulting count can be normalized to yield a new estimate of $d_j(4)$. The formulas for finding the counts, which are collectively referred to as the Baum-Welch algorithm [5], can be conveniently expressed in terms of the forward probabilities, the backward probabilities, and the model parameters:

$$a_{\text{count}}(i,j) = \sum_{t=0}^{T-1} \sum_{\tau=1}^{D} a_t(i) w_{t,i,j}(\tau)$$
By normalizing the expected counts, we find an improved estimate of the model parameters:

\[ a_{i,j}^{\text{new}} = \frac{\text{count}(i, j)}{N}, \quad d_j^{\text{new}} = \frac{d_{count}(j, \tau)}{D} \]

\[ \mu_j^{\text{new}} = \frac{\mu_{count}(j)}{\theta_{count}(j)} \quad \sigma_j^{\text{new}} = \frac{\sigma_{count}(j)}{\theta_{count}(j)} - (\mu_j^{\text{new}})^2 \]

### 3.2 Recognition

In the Viterbi algorithm [5], only the most probable partial path is propagated. This allows the operations to be performed in the logarithmic domain, so that scaling is not necessary. Define \( \gamma(t) \) as follows:

\[ \gamma(t) = \log(Pr(O_1, ..., O_t, \text{ state } j \text{ ends at time } t \mid S^{t-1}_1, \lambda)) \]

where \( S^{t}_1 \) is the most likely state path that accounts for \( O_1, ..., O_t \).

\[ \gamma_0(1) = 0 \]

\[ \gamma_0(j) = \max_{i=1}^{N} [\gamma_0(i) + \log(a_{i,j}) + \log(d_j(0))] \], \( j = 2, ..., N \)

For \( t = 1 \) to \( T \),

\[ \gamma_t(j) = \max_{i=1}^{N} \max_{\tau=1}^{D} v_{i,j;\tau} \text{ for } j = 1, ..., N \text{ where} \]

\[ v_{i,j;\tau} = \gamma_{t-\tau}(i) + \log(a_{i,j}) + \log(d_j(\tau)) + \sum_{s=1}^{\tau} \log(b_j(O_{(t-s)+1})) \]

If the argmax is saved for each maximum found during the trellis update loop, the most likely sequence of states can be found by backtracking through through the Viterbi trellis using the saved argmaxs.
4 Parallel Implementations

Much work has been published on reducing the computational burden of HMM speech recognition systems. Since both training and recognition consist of a large number of simple repeated tasks, most parallel implementations have been on SIMD (Single Instruction Multiple Data) or SSIMD (Skewed Single Instruction Multiple Data) architectures. The most common approach has been to speed up the recognition component by dividing the vocabulary into $L$ subsets. Each of $L$ processing elements (PEs) is responsible for finding the best word in its subset. The control unit then selects the best word among the $L$ candidates for each time $t$. The systems that have been applied to speech recognition are typified by a small number of processing elements with low communication overhead. In [6] and [7], the word models were distributed across four transputers. An implementation using this approach on a tree machine with up to 127 PEs is presented in [8, 9]. A 97-node MIMD (Multiple Instruction Multiple Data) butterfly machine that uses this approach is described in [10]. The speedup attainable by this approach compared to a serial implementation can be linear in the number of processors. However, the speedup is often less impressive in practice because much of the search space can be pruned on a serial machine. Each processing node of a multiprocessor implementation can also utilize pruning, but the speedup will only be linear if each of the $L$ search subspaces is pruned equally. Thus the average PE efficiency typically drops off as the number of processors is increased. It is possible to dynamically partition the active vocabulary, but this greatly increases complexity, communication overhead, and memory requirements.

A finer-grain parallel implementation of the recognition component is detailed in [11]. Custom ICs were designed with a large number of very simple PEs and specialized interconnections. Each PE evaluates one candidate in the search space. At any given time, many of the processors evaluate candidates that would be pruned by a serial implementation. An additional disadvantage of this approach is the lack of interconnection flexibility.

The system described in [12] also uses a fine grained approach where each PE updates one state of an HMM. The recognition component is implemented on an associative string processor. While this architecture allows more flexible interconnections than [11], it suffers the same disadvantage of low PE efficiency compared to a serial implementation with pruning.
A different approach to speed up recognition is taken in [13]. The distribution of the output is assumed to be a mixture of $P$ Gaussian distributions. Each mixture is computed on a different node, and the results are combined by the master node. Another node calculates the transition probabilities. This division of labor has the advantage that the computational load is fairly well balanced. However, the amount of parallelism that can be exploited is fixed at the number of mixtures, which is usually less than 20. The implementation in [13] utilizes ten transputers.

Most parallel implementations of speech recognition algorithms offer far less than linear speedup because it is difficult to balance the workload when much of the search space is pruned. During training, pruning is used conservatively or not at all, so that the benefits of parallelism can be more fully exploited. A fine grained implementation of the training algorithms on a SSIMD ring parallel processor is discussed in [14]. The implementation features a balanced distribution of the workload and two schedules so that the forward and backward probabilities (see section 3) can be found simultaneously. Two schedules are advantageous when one sentence is processed at a time, because twice as many PEs can be actively employed. If many sentences are processed at one time, however, the advantage of two schedules is generally neutralized, since most training tasks could utilize far more PEs than are available for the calculation of either the forward or backward probabilities. (For example, a system that trained using 1000 sentences with an average of 35 phones per sentence would require more than 100,000 PE’s if each phone was modeled by three states.)

Training and recognition algorithms for an orthogonal multiprocessor are given in [15]. This architecture is difficult to scale (the authors discuss a recent prototype with 16 64-bit RISC microprocessors) and utilizes a partially shared memory method of communication that greatly complicates software development.

In this paper, we describe both training and recognition algorithms on a SIMD mesh architecture. Our fine grained solution differs from other work in two important ways. First, the HMM that we implement includes a more accurate model of duration that greatly increases the complexity, but also the potential speedup due to parallelization. Secondly, we consider an implementation on a massively parallel architecture, and address the special problems associated with this type of machine.
5 The Parallel Speech Recognizer [16]

We describe a complete HMM system that is currently being used for phone recognition. In addition to the standard model, we have implemented training algorithms that process multiple sentences at a time, a model that assumes a phone transcription is provided with each sentence, and large memory models that are input-output bound. Both discrete and continuous HMMs are considered.

We chose the MasPar MP-1 because our algorithms have several degrees of parallelism that can be exploited, and can efficiently utilize tens of thousands of PEs. In addition, our implementation benefits from fast scan and reduce operations, which are characteristic of the target architecture. Lastly, xnet communication on the MP-1 is efficient for our application since most of the communication is restricted to a small neighborhood of PEs.

The topology of the phone recognizer implemented on the MasPar MP-1 is depicted in figure 1. Each phone is represented by three states, representing the beginning, middle, and end of a phone.

5.1 Architecture of the MasPar MP-1

![Figure 5: Interprocessor communication on the MasPar MP-1.](image)

The MP-1 [17] in the Parallel Processing Laboratory at Purdue University is a massively-parallel computer with 16,384 PEs arranged in a 128-by-128 array. Each PE has a 4-bit ALU and 16 Kbytes of local memory. The MP-1 supports two different networks for interprocessor communication, the xnet and the global router. Communicating
with xnet is fast but a data transfer can take place only between PEs that lie on the same horizontal, vertical, or diagonal row as shown in figure 5. xnet is similar, except that a copy of the transferred variable is left in each of the PEs between the destination and the source. Communications which use the global router are slower, but the router can support any arbitrary PE-to-PE communication.

Our parallel programs are written in MPL [18], which is C extended with a few keywords to support data parallel computation. We use the optimized routines from MasPar’s programming library listed in table 2. In this paper, we summarize the key features of our parallel implementation. Details are available in [19].

<table>
<thead>
<tr>
<th>Routine</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>reduceAdd(variable)</td>
<td>Calculate the sum of each active PE’s variable.</td>
</tr>
<tr>
<td>scanAdd(variable, seg)</td>
<td>Calculate the prefix sum of variable; seg marks scan boundaries. In a</td>
</tr>
<tr>
<td></td>
<td>prefix sum, the PEs are ordered and each PE receives the partial sum</td>
</tr>
<tr>
<td></td>
<td>of variable in all PEs up to and including the current PE.</td>
</tr>
<tr>
<td>scanMaxd(variable, seg)</td>
<td>Calculate the prefix max of variable; seg marks scan boundaries.</td>
</tr>
<tr>
<td>pExp(variable)</td>
<td>Calculate the exponential of variable in each active PE.</td>
</tr>
<tr>
<td>pLog(variable)</td>
<td>Calculate the logarithm of variable in each active PE.</td>
</tr>
<tr>
<td>sendwithAddd(variable, dest)</td>
<td>variable is sent to the PE with index dest. If dest receives two or more values, they are summed.</td>
</tr>
</tbody>
</table>

Table 2: MasPar MP-1 functions used in the parallel speech system.

5.2 Mapping the Forward Algorithm to the MP-1

Consider an implementation of the forward algorithm. At each time $t$, the forward partial path probability, $\alpha_t$, is updated for each HMM state by summing the value of $\alpha_t$ at $t-1$ across all predecessor states. We will assume that the output distribution is Gaussian with a diagonal covariance matrix. This assumption is common for speech recognition. For $t = 1$ to $T$,

$$\alpha_t(j) = \sum_{i=1}^{N} \sum_{\tau=1}^{D} \alpha_{t-1}(i) a_{i,j} d_\tau \prod_{s=1}^{\tau} b_j(O_{t-s+1})$$  \hspace{1cm} (1)

where

$$b_j(O_t) = \left(\frac{2\pi}{M}\right)^{1/2} \exp \left(-\frac{1}{2} \sum_{m=1}^{M} \frac{(O_{t,m} - \mu_{j,m})^2}{\sigma_{j,m}^2}\right)$$

If any predecessors of state $j$ are null states, then $\alpha_t(j)$ cannot be evaluated until all predecessor nodes have been updated for the current time. If all predecessors are null states, then the current state is updated as follows:

$$\alpha_t(j) = \sum_{i=1}^{N} a_t(i) a_{i,j}$$ \hspace{1cm} (2)
Figure 6: Updating the forward probabilities for one state, represented by a circle. a) output probabilities and b) alpha trellis. The dependence on the current state $j$ has been suppressed (e.g., $d(1) = d_j(1)$ and $\alpha_{t-1} = \alpha_{t-1}(j)$)
Each PE calculates the summand of the $\tau$ loop for a particular duration $\tau$ and a particular state $i$, so that $ND$ states can be employed simultaneously. The $D$ PEs allocated to a particular state are aligned contiguously in the same row to reduce communication costs. Note that for each $t$, all but one term used in the product of Gaussian probabilities have been computed prior to time $t$. These probabilities, which are expensive to calculate, are saved in a rotating buffer that is implemented using xnet to shift the probabilities across the $D$ PEs. At time $t$, the partial product is found using three steps: 1) divide the partial product at time $t-1$ by the probability of $O_{t-D}$ in each of the $D$ PEs, 2) shift using xnet, and 3) multiply the first PE by the probability of $O_t$. Figure 6a shows how the output probabilities are calculated for one HMM state, which is represented by the large circle.

The probability of $O_t$ can be efficiently calculated by utilizing most of the $D$ PEs assigned to an HMM state. The mean vector, covariance diagonal, and each observation vector are stored horizontally across the $D$ PEs. This assumes that $M \leq D$, which is a good assumption for this application. For example, $D = 32$ would leave room for the following observation vector components: 14 spectral coefficients, 14 delta spectral coefficients (i.e., the time derivative for each energy band), energy, and delta energy. Staggering the vector components is an efficient utilization of PE memory and greatly speeds up the calculation of the output probabilities.

Each of the $M$ PEs calculates $(O_{t,m} - \mu_{j,m})^2 / \sigma_{j,m}^2$ for a different $m$ and scanAddd() is used to sum the results (see figure 6a). After plural exponentiation with p_exp() and normalization, the output probability is stored in the leftmost PE using xnet.

As is the case with output probabilities, old values of alpha are stored in a rotating buffer implemented with xnet (see figure 6b). Although only the most recent $D$ values of alpha are needed during the propagation of forward probabilities, all $T$ values are needed for the Baum-Welch re-estimation algorithm. To conserve space, each state's alpha array is stored across the $D$ PEs associated with an HMM state. When $t \mod D = 0$, the rotating buffer of forward probabilities is copied to a plural array: alpha_ssave[t/D] = alpha. If $T = 500$ and $D = 32$, then 128 bytes of PE memory are required to save the forward probabilities as double precision numbers.

Since $a_{i,j} = 0$ for all states $i$ that are not connected to state $j$, the outer loop need only sum across the predecessors of state $j$. For the topology shown in figure 1, all states have a single predecessor except for the initial state. Once the summand is accumulated for each state with scanAddd(), the outer sum of equation 1
is found in one of three ways. Because the initial/final state is a null state, the states must be updated in the following order for each $t$: 1) the last two states for a phone sub-model, 2) the initial/final state, and 3) the first state of each phone sub-model.

For the first PE assigned to either the second or third state of a phone sub-model, the outer sum consists of only a single term, which is accessed using the router. If the phones are not split across states (see figure 7), then xnet could be used instead of the router. However, this layout is wasteful of PEs for most choices of $D$.

The first PE of the first state in a phone sub-model has only the initial/final state as a predecessor. However, the initial/final state is a null state, so that the duration is zero with probability one. Hence, the value of alpha found for the current time is propagated to the initial/final state’s successors without delay. The first PE of the first state in a phone sub-model is updated by accessing the current alpha of the initial/final state, and then multiplying by the locally stored transition probability. Note that this is the only time that the transition probability is included since all other transition probabilities are one.

Equation 1 would quickly result in underflow without the use of scaling. Scaling is accomplished by multiplying the $t^{th}$ column of the alpha trellis by a large constant, $c(t)$ [20]. The scaling coefficient is set to the reciprocal of the sum of all elements in the $D$ most recent columns of the alpha trellis (which is guaranteed to be positive). In our layout, the most recent alpha values are spread across $D$ PEs so that $c(t)$ is simply $1$/reduceAddd(alpha). In practice, it is generally sufficient to scale every $T_{scale}$ time increments (i.e., $c(t) = 1.0$ if $t \mod T_{scale} \neq 0$).

After all $T$ observations have been accounted for, the scaling terms can be removed without underflow by subtracting the log of the scaling terms, which are found in parallel using $p\log()$.

5.3 Mapping the Backward and Baum-Welch Algorithms to the MP-1

For efficiency, the Baum-Welch re-estimation algorithm is combined with the backward algorithm. The backward trellis is updated for $t = T - 1$ to 0:

$$\beta_t(i) = \sum_{j=1}^{N} \sum_{\tau=1}^{D} w_{t,i,j}(\tau) \text{ for } i = 1, ..., N$$

(3)

where

$$w_{t,i,j}(\tau) = a_{i,j}d_j(\tau)\beta_{t+\tau}(j) \prod_{s=1}^{\tau} b_j(O_{t+s})$$
Figure 7: One possible HMM layout on MasPar MP-1.

Figure 8: Re-estimating the mean at time $t$. 
In the Baum-Welch algorithm, the mean is estimated using the forward and backward probabilities:

$$
\mu_{\text{count}}(j) = \sum_{t=0}^{T-1} a_t(i) \sum_{j=1}^{N} \sum_{\delta=1}^{D} O_{t+i,j} W_{t,i,j}(\delta)
$$

(4)

where $W_{t,i,j}(\delta) = \sum_{\tau=\delta}^{D} w_{t,i,j}(\tau)$.

The common part of (3) and (4), $u_{t,i,j}(\tau)$, is calculated for each allowed combination of $i$, $j$, and $\tau$ simultaneously. See figure 8. As in the forward algorithm, scanAddd() is used to collect the results for each state duration, and this prefix sum is used to update beta.

Updating $\mu_t$ requires the partial sums from right to left, i.e., a backward scanAddd(). This can be easily obtained by subtracting the shifted forwardly scanned sum from the total sum. Since the forward sum is used to update beta in (3), the additional cost of obtaining the backwardly scanned sum is much smaller than the cost of a second scan operation (assuming that a backward scan were available). However, care must be taken to avoid errors due to finite precision. The scanned sum of non-negative numbers is not guaranteed to be strictly non-decreasing because the order of pair-wise additions is different for each PE. This lack of precision can lead to negative probabilities, which causes instability. Our solution is to set the backwardly scanned sum to zero whenever it is negative.

Notice in equation 4 that there are $D$ values of $t$ for which a given observation vector is multiplied by path probabilities. It is much more efficient to sum all path probabilities and then multiply by each observation vector once. This can be accomplished using xnet. See figure 9. If $\mu_{\text{count}}^{j,t}$ is the variable which accumulates the contribution of $O_t$ to the mean of state $j$, then $\mu_{\text{count}}^{j,t}$ can be shifted across the $D$ PEs that belong to state $j$. When $t = T - D - 1$, the first non-zero sum of path probabilities is multiplied by $O_T$. At the end of the beta update loop (i.e., $t = 0$), all but the first $D - 1$ observation vectors have been included in the new estimate of the mean. A separate loop from 0 to $D - 1$ includes the remaining observation vectors.

The update equation associated with the variance estimate is identical to equation 4, except that $O_t$ is replaced by $O_t^2$. The shifted summers used for the mean are also used for the estimation of the variance. Both the mean and variance estimates require a normalizing term, which is obtained by replacing $O_t$ by 1 in equation 4.

Unlike the alpha trellis, the backward trellis is stored in the predecessor state instead of the state for which it is defined. This allows easier communication during the calculation of the estimation counts.
Figure 9: Vector math can be postponed until the vector weights are collected over D values of \( t \).
5.4 Mapping the Viterbi Algorithm to the MasPar MP-1

Each node of the Viterbi trellis is updated using only the most likely predecessor. By saving the argmax for each trellis node, the best path can be obtained by backtracking.

The best state duration is found by using scanMaxd(). All PEs that have a partial state path probability equal to the max send their duration index to the last PE assigned to the state. This is an implementation of the argmax, i.e., the index of the maximum element. The router can be used to transfer the duration offset to the argmax location. For the case where two or more PEs are equal to the max, it is irrelevant which duration offset is chosen as the argmax. The best predecessor of the initial state is found similarly.

6 Complexity and Results

The architectural complexity of our complete implementation is \(O((N + D)KT)\) due to communication costs. However, measured complexity was close to the algorithmic complexity of \(O((\log N + \log D)KT)\) because the communication costs were dominated by the cost of multiplications and additions.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Serial Complexity</th>
<th>Parallel Algorithmic Complexity</th>
<th>MP-1 Architectural Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address observation vectors.</td>
<td>(O(NDT))</td>
<td>(O(T))</td>
<td>(O(T))</td>
</tr>
<tr>
<td>Update partial products.</td>
<td>(O(NDT))</td>
<td>(O(T))</td>
<td>(O(DT))</td>
</tr>
<tr>
<td>Multiply by partial path probabilities.</td>
<td>(O(NDT))</td>
<td>(O(T))</td>
<td>(O(T))</td>
</tr>
<tr>
<td>Sum partial results.</td>
<td>(O(NDT))</td>
<td>(O(T\log D))</td>
<td>(O(DT))</td>
</tr>
<tr>
<td>Accumulate counts.</td>
<td>(O(NT))</td>
<td>(O(T))</td>
<td>(O(T))</td>
</tr>
<tr>
<td>Update beta of the initial state.</td>
<td>(O(NT))</td>
<td>(O(T\log N))</td>
<td>(O(NT))</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>(O(NDT))</td>
<td>(O((\log N + \log D)T))</td>
<td>(O((N + D)T))</td>
</tr>
</tbody>
</table>

Table 3: Computational complexity of the backward/Baum-Welch algorithm on the MP-1 using \(ND\) PEs, where \(T\) is the number of observations, \(D\) is the maximum state duration, and \(N\) is the number of states in the HMM.

In order to analyze the details of the complexity, the computational cost of the combined backward/Baum-Welch algorithm is summarized in table 3\(^1\). The complexity breakdown is very similar for the other algorithms described in this paper. The architectural complexity of summing the partial results is worse than the algorithmic complexity\(^1\).

\(^1\)Since \(K\) is approximately one for our application, this term is not included in this analysis.
complexity because the MP-1's scanAddd() requires $O(\log D)$ addition steps but the communication cost is $O(D)$. Similarly, updating the partial products requires copying an output probability from the $D^{th}$ PE to the other $D-1$ PEs using xnetc, which has $O(D)$ communication costs. The initial/final state is updated using reduceAddd(), which has a high architectural complexity. However, the reduceAddd() accounted for less than 20% of the total execution time for all combinations of $D$ and $N$ used in our experiments. Our results indicate that communication costs were dominated by the costs of additions and multiplications, so that the measured complexity approximates algorithmic complexity for both training and recognition.

![Graphs showing complexity comparison](image)

**Figure 10:** A comparison between the complexity of the serial and parallel implementations: a) forward algorithm as a function of $D$, b) forward algorithm as a function of the number of phones, c) Viterbi algorithm as a function of $D$, and d) Viterbi algorithm as a function of the number of phones.

Experimental results are shown in figure 10, which compares our parallel implementation to a serial implementation on a Sun 4/280. The improvement of the parallel version over the Sun 4/280 implementation is even more dramatic than shown because the serial times are conservative in that they do not include the cost of system calls used to implement some of the arithmetic.
To determine how well suited our algorithms were to the architecture of the MP-1, we timed the various portions of our algorithms to see how much time they spent computing versus communicating. Our timings indicate that less than one fourth of the execution is spent in communication overhead, so the bandwidth of the xnet and the global router are not limiting factors. In fact, we spend roughly the same amount of time assigning variables and accessing arrays in local PE memory as communicating, in part because every group of 16 PEs in the MP-1 has to share a single data bus. It appears that for our application, the greatest benefit would result from speeding up the time it takes for our PEs to do single or double precision arithmetic: multiplying two doubles with a 4-bit ALU is slow. Hence, our algorithm may be better suited for MasPar’s MP-2 which uses faster PEs.

7 Modifications

In this section, we briefly describe some extensions to the basic phone HMM.

7.1 Multiple Sentences

In order to adequately train the model parameters, the forward-backward counts are generally summed over a large number of training sentences for each iteration of re-estimation. This parallelism can be easily exploited by modeling $S$ sentences simultaneously. $S$ PE partitions are formed (see figure 11), and the HMM algorithms are altered slightly by replacing global sums (using reduce instructions) with partition sums (using scanAddd()).

![Figure 11: An efficient layout for multiple sentences on the MasPar MP-1.](image)
There is an additional overhead if the concurrent sentences are of different lengths. We group sentences of approximately the same length together so that this overhead is negligible. In our work, anywhere between 5 and 16 sentences can be processed simultaneously for $D = 16$.

### 7.2 Models With Given State Sequences

In some cases, the state sequence may be known for each training sentence. Since each sentence has a different topology, a linear model is constructed using the phones provided by the phone transcription. Figure 12 shows the topology for an example sentence. The forward-backward counts are found for each sentence topology. Since each sentence contains a different set of phones, the phone assignment of a given PE will change for each sentence. This means that the probabilities and counts of re-estimation must be maintained in a separate fixed location.

Our solution has been to create two sets of plural variables. For example, $\mu_{\text{count}}$, which stores the expected sum for the phone currently allocated to the PE, is summed into $\text{save}_{\mu_{\text{count}}}$, which resides in a PE that is dedicated to the given phone. The sentence counts are collected into the total counts prior to the next sentence allocation.

Communication costs are much greater for the linear model because the temporary counts must be moved to the permanent counts using the router. However, since all states now have a single predecessor, $\text{scanAddd}()$ can be replaced by the router when updating the initial/final state.

### 7.3 Word Recognition

Although the algorithms presented in this paper have been described in terms of a phone recognizer, they are equally applicable to a word recognizer with the topology shown in figure 2 (with either implicit or explicit duration modeling). Consider estimating the parameters of a word recognizer using supervised training where the sentence model consists of a concatenation of word models. This task is very similar to supervised training of a phone HMM. In this case, the vocabulary size is limited by the amount of PE memory. Each HMM state
requires $D$ duration probabilities, $M$ means, and $M$ variances, all saved as floats. For re-estimation, each state requires $D$ duration counts, $M$ mean counts, and $M$ variance counts, all saved as doubles. Because of supervised training, two copies of each of the above variables are needed; one is dedicated to some word in the vocabulary and the other varies with the word currently allocated. If we assume that each word is modeled by eight states and $M = D = 32$, then each word requires 9216 bytes. This allows a vocabulary on the order of 10,000 words since the memory requirements can be allocated nearly evenly across the 16K PEs. There are enough PEs so that 64 words can be trained simultaneously.

7.4 Context-Dependent HMMs

Thus far, we have considered HMMs where each phone is modeled by three states, one each for the beginning, middle, and end of the phone. Now consider a more detailed model where each phone is modeled in context [21]. The context is typically represented by the preceding phone (left context-dependent), the following phone (right context-dependent), or both. If there are $P$ phones in a language, then there are $P^2$ right context-dependent phones (although some may have zero probability). In general, parallel implementations of these models are characterized by huge memory requirements, communication costs that dominate processing costs, or both. A fair question then is why would we want to implement an algorithm that is known to be input-output bound on any parallel architecture? Admittedly, such an implementation is a poor utilization of the available processing power. However, a serial implementation is also greatly burdened by the huge memory requirements, resulting in excessive paging and swapping. Rather than address scalability, we consider the advantage of a parallel implementation over its serial counterpart for a family of practical input-output bound context-dependent speech recognition problems. Since discrete HMMs require roughly an order of magnitude more output parameters than continuous HMMs, we will consider only the continuous case.

Some differences between context-dependent and context-independent phone modeling are addressed in table 4. The number of PEs and the amount of plural memory is given for a variety of continuous HMMs that were considered in this paper. The numbers in parentheses represent typical values for phone recognition. Model A is the unsupervised context independent phone recognizer discussed in section 2. Model B is also context independent, but assumes that a phone transcription accompanies each training sentence.
Table 4: Number of PEs and memory required for different models. $D$ is the maximum duration of a state, $M$ is the length of an observation vector, $P$ is the number of phones in the language, and $P_{\text{avg}}$ is the average number of phones per sentence.

<table>
<thead>
<tr>
<th>Model</th>
<th>Phones known</th>
<th>Left context</th>
<th>Num PEs per state</th>
<th>Num PEs per sentence</th>
<th>Bytes per PE for output probabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>No</td>
<td>No</td>
<td>D</td>
<td>$3PD (1728)$</td>
<td>$16M + 16M/D (221)$</td>
</tr>
<tr>
<td>B</td>
<td>Yes</td>
<td>No</td>
<td>D</td>
<td>$3P_{\text{avg}}D (2400)$</td>
<td>$16M + 16M/D (221)$</td>
</tr>
<tr>
<td>C</td>
<td>No</td>
<td>Yes</td>
<td>D</td>
<td>$3P^2D (62208)$</td>
<td>$16M + 16M/D (221)$</td>
</tr>
<tr>
<td>D</td>
<td>Yes</td>
<td>Yes</td>
<td>D</td>
<td>$3P_{\text{avg}}D (2400)$</td>
<td>$16M + 16M/D + 48MP^2/16384 (271)$</td>
</tr>
<tr>
<td>E</td>
<td>Yes</td>
<td>Yes</td>
<td>1</td>
<td>$3P_{\text{avg}} (150)$</td>
<td>$32M + 48MP^2/16384 (466)$</td>
</tr>
</tbody>
</table>

Model C, which includes context and does not require a phone transcription, needs more PEs than are available on the MP-1 for any reasonable value of $D$. The fully connected context-dependent model is most useful for recognition where a serial implementation (or perhaps a transputer array) can exploit extensive pruning to achieve reasonably fast performance. Model D, which requires phone transcriptions, uses a small number of PEs for each sentence but still needs to store $O(P^3)$ output distributions. Thus the only difficulty for a context-dependent sentence model is the method of storing and communicating the huge number of model parameters.

Models A, B, C, and D all use $D$ PEs for each HMM state, where each PE calculates the path probability for a different duration of the current state. Neglecting communication costs, a linear speedup of $NS$ can be achieved with $NDS$ PEs. However, since there are far fewer than $NS$ PEs available for most practical problems, an equivalent speedup can be achieved by performing $D$ operations sequentially in each PE. In model E, all possible durations are calculated serially by one PE. The resulting model is much slower per sentence, and has the additional disadvantage that the forward trellis cannot be stored over $D$ PEs. This trellis dominates the total memory requirements. However, the number of sentences that can be processed at a time increases by a factor of $D$. This avoids the use of most of the scan operations and more efficiently utilizes the PEs. Using model $E$, we have trained a continuous HMM phone recognizer with a maximum state duration of 32 and 26 features per observation vector using 10 minutes of speech at 100 frames per second. Each iteration of training took less than 2 minutes, and the parameters converged in the first 5 iterations. Thus the system accomplishes real-time training (neglecting the cost of feature extraction) for a phone recognizer with explicit duration modeling $^2$.

$^2$Model E is also useful for the situation where duration is modeled only implicitly (i.e., $D = 1$).
8 Conclusion

The MP-1 provides an impressive speedup for our HMMs. It provides a large number of PEs, and many of our applications can keep them all productively employed. The key points of our implementation are listed below:

1. The $D$ most recent values of alpha, beta, and the output probabilities are stored in a queue implemented using xnet. This is an efficient allocation of memory and greatly reduces communication.

2. The scaling coefficient is set to the reciprocal of the sum of the most recent $D$ columns of the alpha trellis. This is more robust than summing only the most recent alpha column [20] because it is possible for up to $D - 1$ consecutive column sums to be zero. The distributed layout of alpha makes this implementation of scaling simple.

3. During re-estimation, three degrees of recursion are exploited by utilizing xnet, a backwardly scanned sum, and scanAdd().

4. The $D$ PEs needed for duration modeling are also used to calculate the Gaussian distribution (assuming $M < D$). This also distributes the memory needed for output vectors across PEs.

5. Another queue implemented with xnet stores the summed counts for each of the most recent $D$ observations before multiplying by the vector $O_t$ or $O_t^2$. This reduces the number of multiplications needed to estimate the output probabilities from $2M$ to 2 and the number of additions from $3M$ to 3 for each $t$.

In this paper, we have presented an efficient massively parallel implementation of a complex HMM recognizer. We have also discussed the choice of data structures and suggested novel approaches to implement explicit duration HMM modeling that take advantage of the parallel constructs of the target architecture.

For a serial implementation, training and recognition are $O(NDT)$, since the average number of predecessors, $K$, was about one for our topology. The algorithmic complexity of our implementation of the Viterbi, forward, backward, and Baum-Welch algorithms is $O((\log N + \log D)T)$. On the MasPar MP-1, the architectural complexity is $O((N + D)T)$ using $ND$ PEs. The measured complexity, however, is close to the algorithmic complexity because communication costs are small compared to the cost of additions and multiplications for this application.
Moreover, the log component, which is due to scan and reduce operations, is very small. Thus, the computational costs of both training and recognition are not sensitive to the choice of $D$ or $N$.

This work demonstrates the usefulness of massively parallel high performance computers for speech recognition research. The HMM algorithms can make good use of a large number of processors and the communication overhead has little impact on the execution time. Moreover, we were able to train and use explicit duration HMMs that are significantly more complex than would be practical on conventional workstations.

References


