Parallel Viterbi Search Algorithm for Speech Recognition

by

Rajeev Dujari

Submitted to the Department
of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

Massachusetts Institute of Technology

February, 1992

Author . . . .
Department of Electrical Engineering and Computer Science
January, 1992

Certified by . . . .
Stephen A. Ward
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Campbell L. Searle
Chairman, Departmental Committee on Graduate Students

Parallel Viterbi Search Algorithm for Speech Recognition

by

Rajeev Dujari

Submitted to the Department of Electrical Engineering and Computer Science on January, 1992, in partial fulfillment of the requirements for the degree of

Master of Science

Abstract

The Viterbi search is an important, but computationally expensive, algorithm for speech recognition. Even with the substantial advances expected in processor technology, the massive computational resources required will remain prohibitive for operation of a speech recognition system in real time. This problem motivates the development of a parallel Viterbi search algorithm.

A software implementation of a Viterbi search algorithm was written for NuMesh, a network of programmable communications routers supporting a set of digital signal processors with local memory. Communication between the processors occurs in the logical pattern of a binary tree, embedded in the physical topology of a two-dimensional Cartesian mesh.

Despite the limited architecture of the routers, efficient merging and broadcasting of data were achieved by simple protocols for pipelined communication. Experimental results were collected in evaluation of an analytical model, which projects excellent scaling of performance with the number of processors.

Thesis Supervisor: Stephen A. Ward
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank Prof. Steve Ward, of the Computer Architecture Group, and Dr. Victor Zue, of the Spoken Language Systems Group, for their collaboration in providing me with this unique research opportunity at the MIT Laboratory for Computer Science. Special thanks go to Dr. Gill Pratt, for his speedy turnaround of the new NuMesh DSP board, Mat Hofstetter, for his dedication to developing the CFSM compiler, and all the other members of the Numesh group for their invaluable advice and support. My profound thanks also go to Mike Phillips and Dave Goddeau of the Spoken Language Systems Group for patiently answering my barrage of questions. Lastly, I owe my sanity to Boston's Classic Rocker (WZLX-FM 100.7) for keeping me company during long hours in the lab.

Contents

List of Figures

List of Tables

Chapter 1

Background

1.1 Introduction

The Viterbi search is an important, but computationally expensive, algorithm for speech recognition. The goal of this thesis is the application of parallel processing to support scalable performance of the algorithm in a speech recognition system. This chapter presents background on relevant concepts in parallel processing and speech recognition which frame the problem, followed by an overview of the thesis.

1.2 Parallel Processing

A useful tool for visualizing the structure of a computation is the *dataflow graph.* A dataflow graph is shown in Figure 1.1 for the computation $f = ab + cd$. The arithmetic operations correspond to its nodes, and data values may be visualized as tokens flowing along its arcs. The dataflow graph makes explicit the dependency of the final addition on the results of the two multiplications.

In this example, each operation has two inputs and a single output. When both input values are available to a processor, it can perform the operation to produce the output value. A single processor can perform the operations, one at a time, concluding with the addition. Assuming that each operation requires one unit of execution time, a processor could perform the computation of in three units of time,

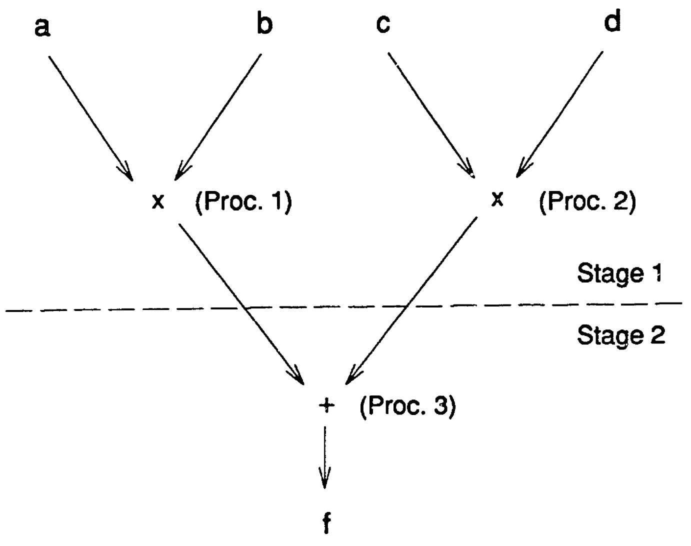

Figure 1.1: Dataflow Graph for $f = ab + cd$

or n such computations in $3n$ units of time.

While simple, this example indicates the potential to exploit *parallelism*, the concurrent execution of operations. As labeled in Figure 1.1, each multiplication operation could be performed simultaneously by a distinct processor. The final addition can be performed by a third processor, which must wait for the completion of the multiplications (a time dependency) and receive their results (a data dependency). At best, n such computations can be performed in only $2n$ units of time.

Additional overhead, however, is incurred in satisfying the data and time dependencies, respectively, by the mechanisms of *communication* and *synchronization*. The cost of this overhead depends greatly on the architecture of the parallel processing system, the structure of the parallel algorithm, and how well they are suited to each other. This overhead ultimately limits how finely the computation can be divided and can degrade how well performance scales with the number of processors.

If a computation is to be performed repeatedly, further parallelism can be exploited by *pipelining*, the concurrent execution of different stages of the computation. Suppose that f is to be computed several times by a parallel system of three processors. The computation could be divided into two stages, as indicated by the dashed line in Figure 1.1.

The two multiplications of the first stage for the next computation of f can be overlapped with the addition of the second stage for the current computation. Neglecting the cost of communication and synchronization, n such computations can be performed in $n + 1$ units of time. Note that while the *latency*, the time for each computation, remains two units, the average time approaches one as n grows large.

A system is said to operate in *real time* if it can process input data at the maximum rate which it arrives. In a pipelined system, each stage must satisfy this criterion, and communication bandwidth between the stages must be sufficiently high. The placement of data buffers between the stages relaxes this criterion, allowing each stage to operate at the average, rather than the maximum, rate of data arrival.

1.3 Speech Recognition

A typical speech recognition system is organized as a series of successive stages. This organization allows each stage to be designed and developed with some degree of isolation. If each stage can process incomplete input from the previous stage to provide incomplete output to the following stage, the system can be pipelined for operation in real time.

A speech recognition system might be just one component of a larger system which interacts with its users through the medium of speech. The goal of speech recognition is to transcribe a spoken utterance to a textual word string. A natural language system, in turn, could attempt to understand the meaning of the utterance. An application, such as an expert system, could act upon commands generated by the natural language system. Finally, the response could be communicated to the user through a speech synthesis system.

To fulfill its purpose, speech recognition must be performed with both speed and accuracy, which depend greatly on the types of algorithms employed. Speech recognition systems can vary widely along several parameters. This research is concerned mainly with speaker-independent, continuous speech recognition. In a speaker-independent system, no attempt is made to adapt to a particular speaker or set of speakers. In continuous speech, words need not be isolated by intervals of silence.

In *lexical access*, the final stage of speech recognition, a dynamic time alignment is performed, giving rise to to a search problem. Many algorithms can be employed for the search, including the Viterbi algorithm. While efficient, the search can require minutes to complete with current desktop computing technology (Sun II), and requires several megabytes of memory, even for a small vocabulary of just a few hundred words.

In contrast, the front-end stages of the speech recognition system can usually be performed in real time by digital signal processors with a modest amount of memory. The lexical access bottleneck prevents the operation of a speech recognition system in real time. Even with technological advances expected in mass-produced processors, the computational expense will remain prohibitive for larger vocabularies. This limitation motivates the development of a parallel Viterbi search algorithm for lexical access search which can operate within a pipelined speech recognition system.

1.4 Overview

This research is the focus of a collaborative effort between two groups at the MIT Laboratory for Computer Science. A parallel Viterbi search algorithm was designed to work with the models used in Summit, a speech recognition system developed by the Spoken Language Systems Group. A software implementation of the algorithm was written for NuMesh, a parallel processing system developed by the Computer Architecture Group.

Chapter 2 describes the Summit speech recognition system with emphasis on the lexical access stage. Chapter 3 discusses search algorithms for lexical access and explains the important role of the Viterbi search. Chapter 4 describes the NuMesh parallel processing system. Chapter 5 presents a parallel Viterbi search algorithm designed for Summit and implemented on NuMesh. Chapter 6 presents an analytical performance model of the implementation. Chapter 7 concludes with projected results and possible extensions of the parallel Viterbi search algorithm.

Chapter 2

Summit: Speech Recognition System

2.1 Introduction

To provide concreteness to the discussion, lexical access will be presented in the context of a particular speech recognition system, Summit. The search problem in the lexical access stage of Summit, however, is similar to that which arises with the hidden Markov model approach [Rabiner '86] predominant in other contemporary speech recognition systems.

Summit has been developed by the Spoken Language Systems Group at the MIT Laboratory for Computer Science. The system can be viewed as a series of successive computational stages [Zue '89] [Zue '90]. Specifically, these stages are spectral analysis, acoustic segmentation, phonetic classification, and the focus of this research, lexical access.

The input to the system is obtained by transducing, filtering, and digitizing the acoustic signal generated by the spoken utterance. Each stage serves to refine this signal representation. This chapter traces this evolution through the stages of Summit, as depicted in Figure 2.1, taken from [Zue '90].

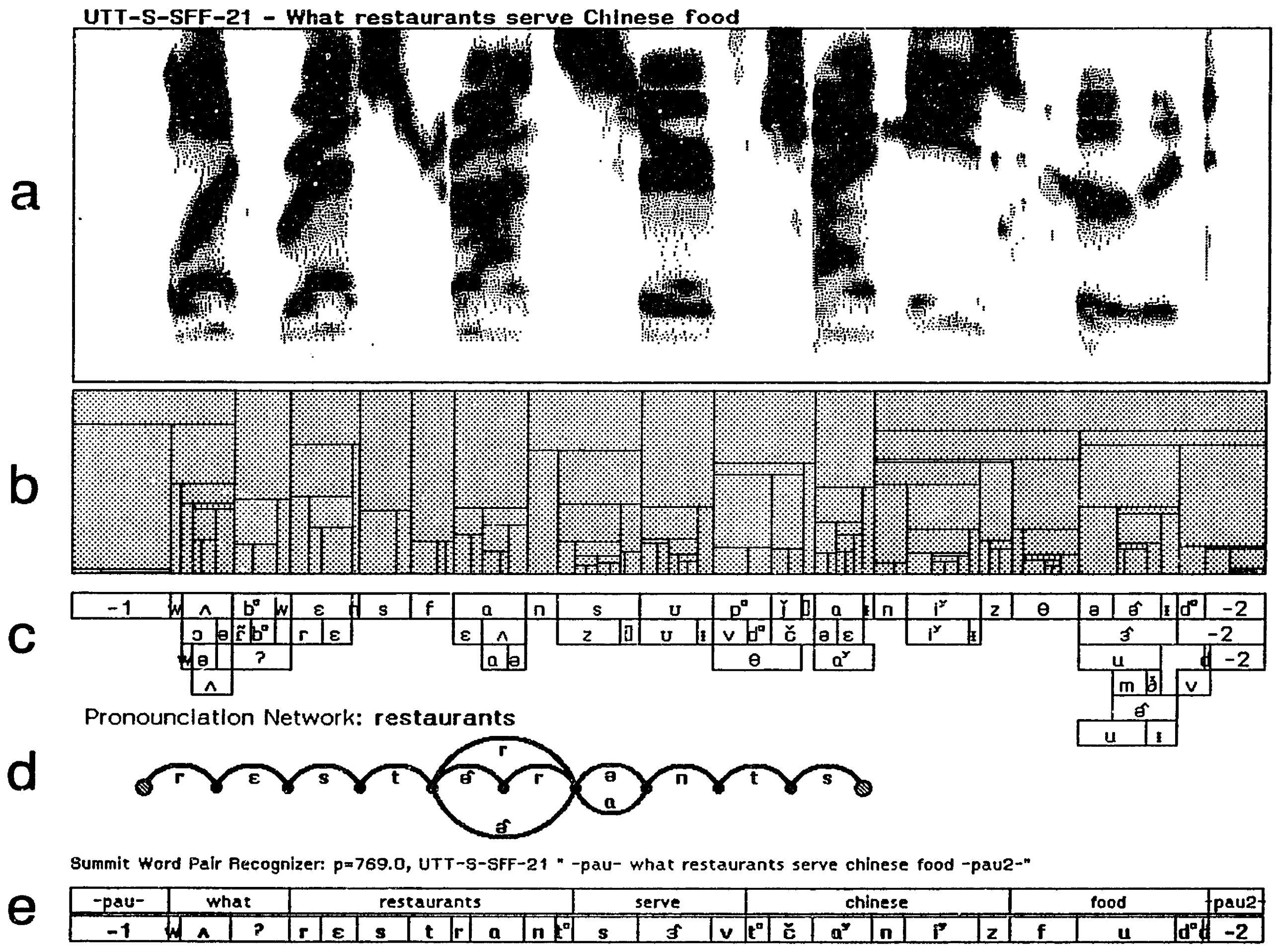
UTT-S-SFF-21 - What restaurants serve Chinese food
a
b
c
Pronounciation Network: restaurants
d
Summit Word Pair Recognizer: p=769.0, UTT-S-SFF-21 " -pau- what restaurants serve chinese food -pau2-"
e
-pau-
what
restaurants
serve
chinese
food
pau2

Figure 2.1: Signal Representations in Summit

2.2 Spectral Analysis

The spectral content of the speech signal is relatively stationary over short intervals of time. The spectral analysis stage makes measurements of the signal, repeatedly, in uniform intervals of time called *frames*. The duration of each frame, which is subject to the classic tradeoff between time and frequency resolution, is five milliseconds. These measurements serve to characterize the signal in each frame.

The mainstay of this stage is a nonlinear model of the human auditory system [Seneff]. Motivated by physiological experiments, the model calculates two sets of energy measurements, mean rate and synchrony response, each over 40 frequency channels. Each set can be displayed as a spectrogram, a density plot like the one in Figure 2.1a, displaying spectral energy versus time.

In addition, signal energy is measured in five broad spectral bands by a windowed fast Fourier transform. Finally, the frequency of the speaker's voicing, if any, is estimated by a pitch period algorithm [Rabiner '75]. Taken together, these measurements form a vector of data which may be compressed by methods such as vector quantization or principal components analysis.

2.3 Acoustic Segmentation

While the speech signal remains stationary during a short frame, it varies over longer intervals of time in order to convey information. The acoustic segmentation stage locates *boundaries* in the signal, which define *regions* of homogeneity. The time basis of the signal is changed from uniform-length frames to variable-length regions.

Many alternative segmentations are suggested to capture both subtle and dramatic changes in the speech signal. These alternatives can be represented in a multi-level data structure of overlapping regions, called the *dendrogram*. A dendrogram is graphically portrayed in Figure 2.1b.

The segmentation algorithm [Glass] identifies acoustic boundaries in the signal by calculating the dissimilarity between the measurement vectors in adjacent frames.

These boundaries define seed regions for the dendrogram. Higher-level regions, which span multiple boundaries, are formed by merging two similar lower-level regions.

2.4 Phonetic Classification

The phonetic classification stage performs pattern matching on each region of the dendrogram to acoustic models of speech called *phones*.[1] Each phone consists of measurement statistics obtained from training data for that phone. These statistics are parametrically modeled as mixtures of Gaussian distributions.

Each region of the dendrogram is scored against all possible phones to produce an *acoustic-phonetic network*, or AP network for short. A simplified AP network, showing only high-scoring phones, is shown in Figure 2.1c. If a region scores poorly against all phones, it may be discarded. If a region scores well against one phone, it may score well against similar phones.

2.5 Lexical Access

The lexical access stage ultimately determines which regions of the AP network to keep by mapping them onto the phones of a *lexical network*. A dynamic time alignment is performed by constructing a *trellis*. A transcription, like the one shown in Figure 2.1e, is produced by finding the path through the trellis with the best score. This section describes the structure of the lexical network, construction of the trellis, and scoring of the paths through the trellis.

2.5.1 Lexical Network

The lexical network contains a pronunciation *model* for each word in the vocabulary of the system. A model consists of discrete states connected by directed arcs, each labeled with a particular phone. A simplified model for the word "restaurant" is

[1]not to be confused with phonemes, which are *perceptual* models. Speakers of English normally perceive many distinct phones as the same phoneme.

shown in Figure 2.1d. A complete model offers several alternative pronunciations for a word.

A model may have multiple start and end states. The end states of a model are connected to *endsets*, which in turn are connected to start states of other models. In the least restrictive model, a single endset could connect all end states to all start states. Continuous speech, however, imposes some constraints.

For example, the final phone of the word "six" and the initial phone of the word "seven" are both /s/. When pronouncing the two words in rapid succession, the two phones are often merged to form a slightly longer /s/. This phenomenon, termed a gemination, is expressed by an endset between all such pairs of words.

2.5.2 Alignment Trellis

The lexical access stage performs a dynamic time alignment by constructing a trellis, a two-dimensional matrix network. The connectivity of the trellis is determined jointly by the structure of the AP network (run-time data) and the lexical network (compile-time data). In the following hypothetical example, the trellis in Figure 2.4 was constructed from the simple AP network in Figure 2.2 and the simple lexical network in Figure 2.3.

The AP network in Figure 2.2 consists of three regions. (Assume higher-level regions were discarded by the phonetic classification stage, as their presence would greatly complicate the illustrations.) Within each region is listed the score against /X/ and /Y/, the only phones in our hypothetical language. Also listed is the penalty for inserting the region into the transcription without mapping it onto a phone.

The lexical network in Figure 2.3 consists of a single pronunciation model with only three states. The start state is S_0 and the end state is S_2. A phone, /X/ or /Y/ is indicated above each arc. Below the arc is listed the penalty for deletion of the phone without the mapping of a corresponding region. Because there is only one word, no endsets are shown for this lexical network.

A node in the trellis is specified by two coordinates. The horizontal index corresponds to an acoustic boundary; the vertical index, lexical state. A path through

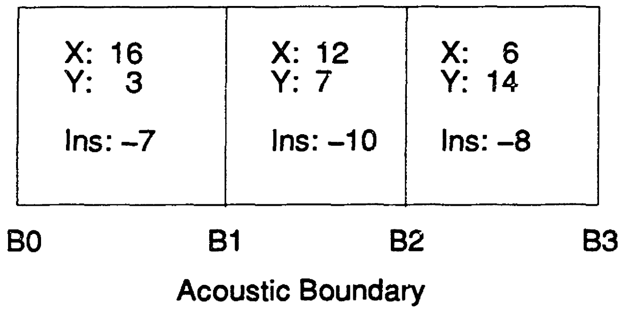

Figure 2.2: Acoustic-Phonetic Network

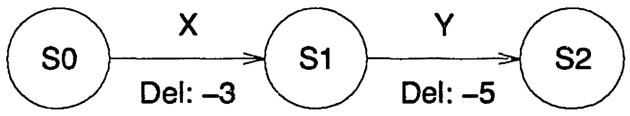

Figure 2.3: Word Pronunciation Model

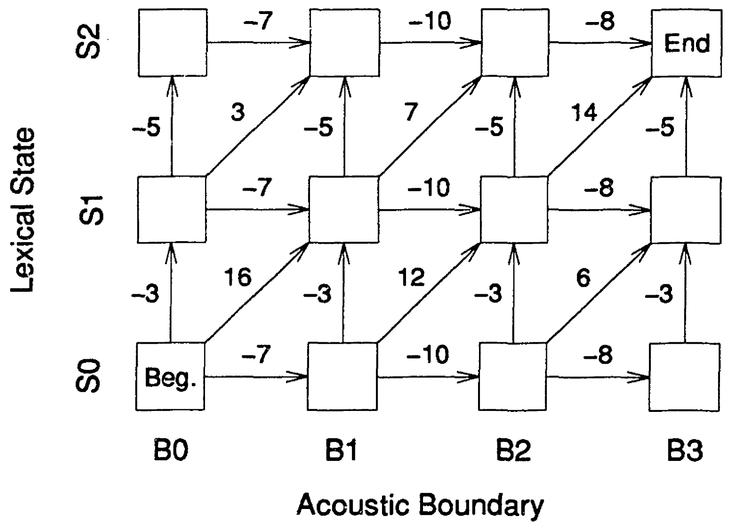

Figure 2.4: Dynamic Alignment Trellis

this trellis must begin at the lower left corner, (B_0, S_0), and end at the upper right corner, (B_3, S_2).

Scoring of Paths

The score of a path is accumulated over the arcs along the path. The four types of arcs that can be placed between the nodes of the trellis are matches, insertions, deletions, and endsets. Matches, insertions, and deletions occur within word models, while endset transitions occur between them.

In a match, a AP network region is mapped onto a lexical phone, resulting in a diagonal transition. Two nodes are connected when a region spans their two boundaries *and* a phonetic arc connects their two states in the lexical network. The segment is labeled with the score of the spanning region against the phone. For example, an connection is made from (B_0, S_0) to (B_1, S_1) with score 16, the score of the region from B_0 to B_1 against /X/, the phone of the arc connecting S_0 and S_1.

In an acoustic insertion, a region is inserted in the transcription without mapping it onto a lexical phone, resulting in a horizontal transition. Two nodes of the same state are connected when a region spans their boundaries. The arc is labeled with the insertion penalty for the spanning region. For example, a connection is made between (B_1, S_1) and (B_2, S_1) with score -10, the insertion penalty for the region from boundary B_1 to B_2.

In a phonetic deletion, a phone is deleted from the transcription without a region, resulting in a vertical transition. Two nodes of the same boundary are connected if their lexical states are connected by a phonetic arc in the lexical network. The arc is labeled with the insertion penalty for the connecting phone. For example, a connection is made from (B_2, S_1) to (B_2, S_2) with score -5, the deletion penalty for /Y/, the phonetic arc connecting state S_1 to S_2 in the lexical network.

In an endset connection, a transition can be made from an end state to a start state at the same boundary, if the two nodes are connected by an endset. A fixed penalty, called a word transition weight, is incurred. Although this type of transition is not illustrated by this example, the connectivity within word models is usually sparse relative to that between word models.

2.5.3 Grammars

The use of grammatical knowledge can aid the search process between word models. A *grammar* may be deterministic, probabilistic, or some combination of both. In a deterministic grammar, the sequence of words in a trellis path is restricted to reduce the search space. In a probabilistic grammar, the scores of paths are modified according to their word sequence to warp the search space.

Deterministic

The most general deterministic grammar is the natural language system which may follow the speech recognition system. Such a system could be taken to *define* grammaticality. A natural language system attempts to identify both the structure (syntax) and meaning (semantics) of the word string in order to understand the utterance.

In practice, a less stringent version of a natural language grammar must be used for two reasons. First, to be useful in a pipelined lexical access stage, the grammar must be able to reject incomplete word strings. Second, many word strings must be tested, so the grammar must not consume substantial computational resources.

A more simplistic deterministic grammar is expressed by word-pair constraints [Zue '91]. Such a grammar restricts which words may follow any particular word. The word-pair grammar is popular because it can be implemented directly in the endset structure of the lexical network and used in conjunction with other grammars.

Probabilistic

The most widespread probabilistic grammars are the unigram, bigram, trigram, or in general, the N-gram, where the N refers to the number of words in the window of the grammar. For each subsequence of N words, the score of the path is modified according to a table of adjustments, based on the statistical frequency the subsequences in training data.

For most entries in a trigram table, training data may be nonexistent or insufficient for statistical significance. A sparse representation of the lookup table can be used for the trigram, with the defaults for missing values obtained from bigram statistics. Similarly, a bigram matrix entry may default to a unigram statistic.

The unigram can be easily incorporated into the lexical access search process by applying the adjustment upon entering the start state of any word. Like the word-pair grammar, the bigram can be incorporated into the endset structure of the lexical network, although a fully associative bigram would greatly increase the connectivity of the trellis by requiring a dedicated endset between each pair of words. A trigram may require additional bookkeeping by the search algorithm.

2.6 Summary

The four stages of the Summit speech recognition system are spectral analysis, acoustic segmentation, phonetic classification, and lexical access. The spectral analysis stage calculates a measurement vector of the signal in each frame, the acoustic segmentation identifies regions of homogeneity, and the phonetic classification stage scores each region against phonetic models to produce an acoustic-phonetic (AP) network.

The lexical access stage maps the regions of the AP network against the phones in a lexical network by constructing a trellis. The scores of the paths through the trellis are determined jointly by the AP network and the lexical network. A transcription is produced by finding the path through the trellis with the highest score. Grammatical knowledge can aid the search process between word models. In the next chapter, two classes of search algorithms for lexical access will be discussed.

Chapter 3

Lexical Access Search Algorithms

3.1 Introduction

As explained in the previous chapter, the lexical access stage performs a dynamic time alignment of the AP network and lexical network, giving rise to the problem of searching for the best path through a trellis. This chapter discusses two classes of search algorithms, the Viterbi decoder and stack decoder, which can be employed for lexical access search.

3.2 Viterbi Decoder

The Viterbi search [Viterbi] was originally introduced as a means of forward error correction coding in communication systems. The algorithm can be applied to similar problems which require dynamic time alignment, including lexical access search in speech recognition. This section discusses the theory, operation, and limitations of the Viterbi decoder.

3.2.1 Theory

The Viterbi algorithm is an example of dynamic programming [Bertsekas]. In this class of algorithms, subproblems that would arise repeatedly from a recursive decomposition are solved once to construct a global solution. Finding the best path through the trellis can be exploited by the dynamic programming technique because the problem of finding the best path can be decomposed into shared subproblems of finding the best subpaths.

In the trellis, any path from node A to node C must pass through a node B such that an arc connects B to node C. Furthermore, the best path from node A to node C *through* node B must contain the best path from node A to node B. Otherwise, a better path from A to B could be substituted to obtain a better path from A to C, resulting in a contradiction.

The Viterbi search algorithm systematically iterates over the nodes in the trellis, maintaining the best path to each node. Each node is labeled with the score d of the best path and a pointer p to the the previous node in that path. Formally,

$$d_j = \max_i \{d_i + a_{ij} \mid (i,j) \in \mathcal{A}\}$$

$$p_j = \arg\max_i \{d_i + a_{ij} \mid (i,j) \in \mathcal{A}\}$$

where a_{ij} is the cost of an arc (i,j) belonging to the set of arcs $\mathcal{A}$ in the trellis. At the conclusion of the utterance, best path can be found by backtracing the pointers from the final node of the trellis.

Suppose there are R regions and B boundaries in the AP network, and N nodes in the lexical network. There are NB nodes in the trellis, so the Viterbi algorithm requires $O(NB)$ computational space for storing scores and pointers. Each node could have, in the worst case, N incoming arcs per region. Thus, the Viterbi search requires $O(N^2R)$ computational time.

The Viterbi algorithm is by far more efficient than direct enumeration of all the paths through the trellis. At each boundary, a path may go through one of N nodes. (For higher-level regions which span more than one boundary, the path can be taken

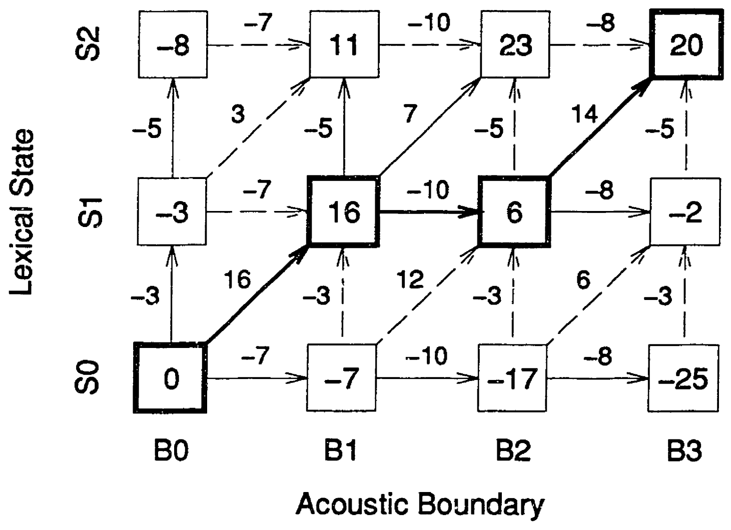

Figure 3.1: Scores from Forward Viterbi Search

to go through an artificial node.) Thus, direction enumeration would require $O(N^B)$ time, which is prohibitively large for realistic values of N and B.

3.2.2 Operation

For the trellis constructed in Figure 2.4, the Viterbi algorithm would produce the labeling in Figure 3.1. The number in each node is the score of the best path to that node. The bold arcs indicate the best path through the trellis, while the dashed arcs indicate paths that were pruned from the search at each node. The operation of the Viterbi search ensures that all of the previous nodes in the trellis have already been labeled.

The structure of the Viterbi algorithm contains many nested loops. The outermost loop iterates over the acoustic boundaries, the columns of the trellis. Each iteration makes three passes over the nodes in each column. In the first, matches and insertions are performed; in the second, deletions; and in the third, endset transitions. The first two passes achieve propagation of scores within word models, while the endset transitions achieve propagation between them.

Matches and insertions for each node are found by two nested loops. The outer loop iterates over the AP network regions ending at the boundary. For each region, an insertion is considered, and an inner loop iterates over the incoming lexical arcs to the node, if any. For each arc, a match is considered. The node is labeled with the score and pointer for the best path found so far.

Deletions must be performed after matches and insertions, but before endset transitions, because the previous node is in the same column of the trellis. Also, two consecutive deletions are disallowed. For each node, a loop iterates over its incoming lexical arcs. If a better path to the node is found, the score and pointer of the node are updated.

Endset transitions are performed by a loop which iterates over the endsets. For each endset, a loop iterates over the end nodes to find the best path for the endset. After the maximum has been found, the word transition penalty is applied. A second loop iterates over the start nodes, updating each node if necessary.

The structure of the endset connections may be viewed as a dataflow graph, subject to many potential compiler optimizations. For example, let a and b be scores of two end nodes attached to an endset, and k be the word transition penalty. The application of the penalty after finding the best score is simply the optimization:

$$\max(a-k, b-k) = \max(a, b) - k$$

Note that the two subtractions on the left side have been reduced to a single subtraction on the right side.

The computation of a best path for each endset can be viewed as an example of common subexpression elimination. This optimization can be carried further by creating intermediate endsets to be shared by the original endsets. For example, let x and y be the score of two endsets which share end nodes a and b.

$$x = \max(a, b, c) - k$$

$$y = \max(a, b, d) - k$$

Each calculation requires two comparisons for a total of four. By introducing an intermediate endset w, which captures the best path from a and b, the number of comparisons is reduced to three:

$$w = \max(a, b)$$

$$x = \max(w, c) - k$$

$$y = \max(w, d) - k$$

3.2.3 Limitations

The Viterbi search can utilize simple grammars, such as the unigram, bigram, or word-pair constraints, which can be incorporated directly into the structure of the endsets. Unfortunately, they tend to fragment the structure of the endsets and reduce the opportunity for optimization. Extending the bigram to a trigram would require a partial backtracing of paths at each endset computation to find the first word in each triplet.

A natural language grammar cannot be utilized because the Viterbi search cannot suggest alternatives to paths rejected by this grammar. This difficulty may prove problematic if the speech recognition system provides input to a natural langauge system, which may reject the best path. Thus, it may be desirable to suggest other good paths.

The Viterbi search algorithm can be extended to maintain the scores of the N best paths to each node of the trellis [Chow]. This extension, however, has a few serious drawbacks. First, the search is slowed because a partial sorting of scores must occur at each node. In addition, the already burdensome memory requirements for storage of scores and labels is scaled by N.

Finally, the value of N must be predetermined. If all N paths are unacceptable, the search must be repeated to extract additional paths. In the next section, a class of algorithms will be described which can extract additional paths from the trellis on demand.

3.3 Stack Decoder

In contrast to the highly iterative structure of the Viterbi search, a stack decoder operates in a recursive fashion. These algorithms selectively enumerate the paths through the trellis, maintaining them in a stack. The stack is usually implemented as a binary heap or priority queue for efficient insertion and removal of paths. The stack decoder can utilize a natural language grammar to filter out ungrammatical paths from the stack. This section describes the basic stack decoder algorithm and an improved version, the A* search.

3.3.1 Basic Algorithm

The stack is initialized with a path consisting of only the start node of the trellis. The best path is popped from the stack and expanded along its outgoing paths, producing several new paths. These paths are inserted back into the stack, ranked according to their scores. Eventually, the complete path with the best score will surface to the top of the stack.

For every good path the stack decoder pursues, it also maintains many similar paths, which includes shorter versions of the path itself. Thus, in practice, stack decoder algorithms use a combination of best-first search between word models and depth-first search within them. This depth-first extension of a path by a complete word can be performed by the Viterbi search. For convenience, a pure best-first strategy can be illustrated by the alignment trellis.

For the trellis given in Figure 2.4, the stack initially consists of the path $\{(B_0, S_0)\}$. The path can be extended by a match, insertion, or deletion, as shown by the solid arcs in Figure 3.2. The previous nodes, the costs of the extensions, and the score of the resulting paths are displayed in Table 3.1. The next path to be expanded will be $\{(B_0, S_0), (B_1, S_1)\}$, because it has the highest score in the stack, 16.

The scores of the arcs in the trellis may be all nonpositive if they are derived from probabilities which range from 0 to 1. To compensate for the length of longer paths, however, both positive and negative scores are used. Unfortunately, this means the

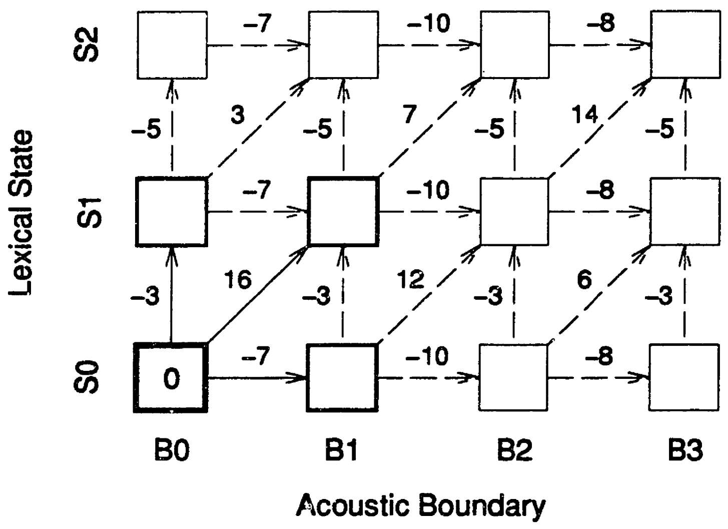

Figure 3.2: Path Expansion in Stack Decoder

Extension	Node	Cost	Score
Match	(B_1, S_1)	16	16
Deletion	(B_0, S_1)	-3	-3
Insertion	(B_1, S_0)	-7	-7

Table 3.1: Path Expansion in Stack Decoder

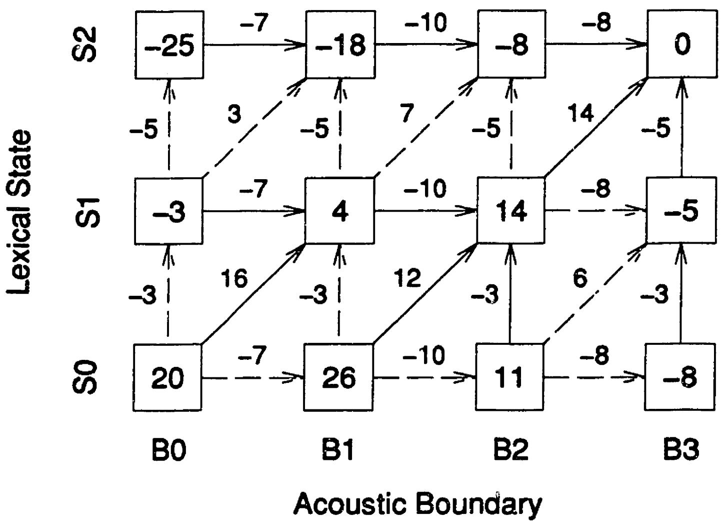

Figure 3.3: Estimates from Backward Viterbi Search

first complete path that surfaces to the top of the stack may be suboptimal.

Because of the enormous number of possible paths, the algorithm must prune paths which appear undesirable. This may result in the correct answer, but is not guaranteed to do so. An improvement to overcome this difficulty and increase efficiency yields the A* algorithm, described next.

3.3.2 A* Search Algorithm

The A* search makes use of estimates of the remaining score for the paths in a stack decoder algorithm. These estimates encourage good paths despite local segments with low scores and discourage bad paths despite local segments with good scores. The paths are ranked in the stack according to the sum of the score and the estimate. The closer the estimate is to the true value, the more efficient is the search.

If the estimate is never less than the actual remaining score, the algorithm can safely prune all paths that have a lower ranking than the first complete path to surface to the top of the stack. In fact, the best possible estimates can be found by performing a Viterbi search *backwards* from the end of the utterance. For the trellis

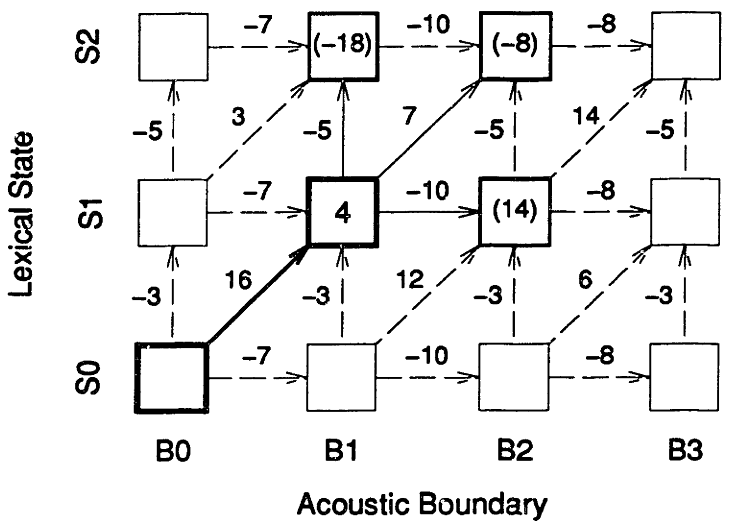

Figure 3.4: Path Expansion in A* Search

Extension	Cost	Score	Node	Estim.	Total
Match	14	23	(B_2, S_2)	-8	15
Deletion	-5	11	(B_1, S_2)	-18	-7
Insertion	-10	6	(B_2, S_1)	14	20

Table 3.2: Path Expansion in A* Search

of Figure 2.4, these estimates are shown in Figure 3.3.

Consider the expansion of the path $\{(B_0, S_0), (B_1, S_1)\}$ in Figure 3.4. The estimates generated from the Viterbi search are given in parentheses at the destination nodes. In Table 3.2, the left side shows the basic stack decoder expansion, and the right side shows the inclusion of the estimate.

Note that the insertion to (B_2, S_1), which is along the best path, is ranked last without the estimate but first with the estimate. The estimate helps the algorithm overlook the match with cost 14 in favor of the insertion with cost –10. With Viterbi estimates, the A* search algorithm will rapidly reconstruct the best path.

Unfortunately, the backwards Viterbi search cannot be initiated until the completion of the utterance. A forward Viterbi search followed by a backward A* search could be used, but natural language grammars used to filter paths from the stack are typically designed to anticipate forwards rather than backwards. In either case, the A* search can not be incorporated into a pipelined lexical access stage.

3.4 Summary

Two major classes of search algorithms can be employed to find the best path through the dynamic alignment trellis. The Viterbi search is an admissible, efficient algorithm which can be incorporated into a pipelined lexical access stage. Furthermore, the algorithm is also valuable for depth-first probing in a stack decoder search and the generation of estimates for an A* search. These features make the Viterbi search the natural choice for a parallel lexical access algorithm.

Chapter 4

NuMesh: Parallel Processing System

4.1 Introduction

NuMesh is a parallel processing system developed by the Computer Architecture Group at the MIT Laboratory for Computer Science. Because it is still in the prototype stage, the system is described primarily by a series of internal memorandums. This chapter explains the motivation behind NuMesh and its architecture.

4.2 Motivation

The traditional substrate for supporting communication and synchronization in parallel processing systems is the backplane bus. The bus derives its popularity from simplicity and modularity. The simplicity of the logical interface allows devices of different types to share the bus. The modularity of the physical interface allows the devices to be flexibly arranged.

The bus, however, does not allow unlimited extensibility. As more device connections are added, the physical length of the bus must be increased, and a long bus must be treated as an analog transmission line. To maintain the physical abstraction, performance is sacrificed.

The time required for each transaction increases for two reasons. First, the time for signal propagation increases as the bus is lengthened. Second, the time for spurious reflections to subside increases because the transmission characteristics are degraded as more device connections are added.

Furthermore, demand for communication bandwidth is likely to increase as the number of devices grows. Contention between processors for the shared bus resource may cause a bottleneck, as communications must be serialized. This limitation renders the bus inadequate for supporting massively parallel processing, motivating the development of a more sophisticated communications substrate.

4.3 Architecture

NuMesh is a communications network designed to support scalable parallel processing while retaining the simplicity and modularity of the bus. Each node of the network has high-bandwidth *ports*, simple logical interfaces, to physically adjacent nodes and a local device. The mechanical packaging of the nodes is modular so the network can be flexibly configured.

Many physical topologies are under consideration for NuMesh. In the current prototype, the nodes can be assembled in a two-dimensional Cartesian mesh. Each node can have up to four immediate neighbors, referred to by the map directions north, east, south, and west. In the planned three-dimensional topology, the number of nodes will grow as $O(n^3)$, where n is the diameter of the network.

4.3.1 Routing

The routing of data through the network is handled by a programmable communications finite state machine (CFSM) at each node. In the current prototype, the CFSM routers are clocked synchronously at 28.5 MHz. A novel global clock synchronization scheme [Pratt] will be incorporated into future prototypes. As with the bus, different types of devices, which may operate asynchronously with respect to each other and the network, can be attached at each node.

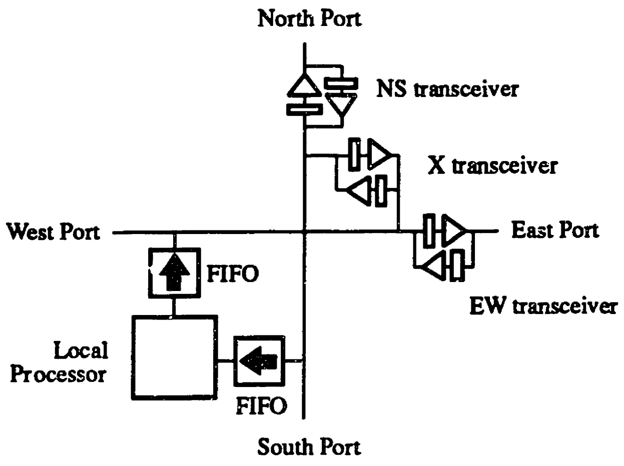

Figure 4.1: CFSM Datapaths

The architecture of the CFSM is minimalistic to allow a fast clock speed. The interface to the local device consists of a pair of first-in, first-out (FIFO) memory buffers, one for each direction of data flow. The FIFO interface serves as a synchronization barrier between the CFSM and the local device, although the device may be clocked synchronously with the CFSM. The ports to neighboring NuMesh nodes consist of a pair of registered transceivers.

The internal datapaths of the CFSM, shown in Figure 4.1 represent a compromise between bandwidth and simplicty. One bus connects the north port, south port, and outgoing FIFO; the other, the east port, west port, and incoming FIFO. The two busses are connected by a pair of registered transceivers to allow data transfers between them. These registers can also serve as temporary storage.

In the current prototype, the CFSM instruction set is very limited. The CFSM does not have the ability to generate a constant onto the datapaths, nor can it examine the value of the data to determine how it should be routed. This architecture is intended to support static routing of data in pre-compiled systolic patterns.

To provide some simple synchronization, status bits are provided for each port. For the transceivers, these signals indicate whether each register has been written but not subsequently read. For the FIFO interface, these signals indicate if the incoming buffer is empty or outgoing buffer is full. The CFSM can utilize these signals to avoid reading invalid or stale input data and over-writing valid output data.

A data transfer may be made safely between two ports, possibly through the cross-bus transceiver, if the data in the source port is available (not empty) and the destination is not blocked (not full). The CFSM may make the transfer without checking these conditions. If it does check them, and finds that either condition is violated, it can either abort the copy or wait until the condition becomes true.

4.3.2 Software

The current processor boards are based on a Texas Instruments digital signal processor [TMS320]. The original DSP board [Abdalla] has only 8K words of memory and runs at 27 MHz. A revised version of this board[1] can address up to one megaword of memory and can support clock speeds of up to 40 MHz. A board board using SPARC processor technology [Tessier] is also planned.

Because the DSP boards lack the file system and console of a desktop computer, a Macintosh IIfx attached to a single node serves as a host. The host computer can determine the configuration of the network by programming NuMesh nodes to explore neighboring nodes, constructing a spanning tree in the process. The host can then use this information to download bootstrap code for each CFSM and DSP without painting itself into a corner.

A CFSM assembler [Nguyen] allows low-level instruction coding. A CFSM compiler[2] provides a higher level of abstraction, concealing some of the peculiarities of the instruction set. It allows the user to write templates for CFSM programs and compile them with parameters.

Creating a software application for NuMesh requires writing a collection of programs which must work in concert. Code must be generated for the host computer, each DSP board, and each CFSM router. The use of automatic build rules (makefiles) eases the development process.

[1]developed by Gill Pratt, of the Computer Architecture Group.

[2]developed by Mat Hofstetter, of the Computer Architecture Group.

While Think C provides a passable debugger for the host computer, debugging tools for NuMesh are spartan. Programs cannot be traced without resorting to an oscilloscope, logic analyzer, or DSP emulator, tools which are usually reserved for debugging hardware.

The DSP boards have a small alphanumeric display and LED indicators for the FIFO interface for feedback to the user, but the interaction between nodes is difficult to observe. A NuMesh simulator [Metcalf] can be used to test programs in a multiprocessing environment.

4.4 Summary

The NuMesh is an alternative communications substrate to the backplane bus. The communication and processing are decoupled in NuMesh. Each node in the NuMesh network contains a CFSM router. The CFSM has minimalistic datapaths and a limited instruction set, allowing efficient communication between processors.

Chapter 5

Parallel Viterbi Search Algorithm

5.1 Introduction

As stated in the previous chapter, a software application for NuMesh requires the creation of a collection of programs to work in concert. A software implementation of a parallel algorithm for the Viterbi search was written for NuMesh. This chapter describes the operation of the DSP programs, host computer program, and CFSM routing, and the communications protocols which govern their interactions.

5.2 Parallel Decomposition

To reduce the complexity of the system, the same program was compiled for each DSP, in a single-program multiple-data (SPMD) approach. The program is divided into two functional modules to facilitate testing. One module is devoted to pure computation, and the other mainly to handling communications with the CFSM. The software can be developed under the MPW environment on the Macintosh or under MS-DOS on a PC. Both modules are listed in Appendix D.

The lexical access trellis can be distributed among the processors by splitting the lexical network among the processors while providing the complete AP network to each processor. A single processor can easily perform the Viterbi computation in real time for a few words. Therefore, there is no need to split word models among different processors, incurring expensive communication and synchronization overhead.

The endset structure, the glue between the word models, can also be distributed among the processors. Each endset is split into local endsets, one for each processor. These local endsets have connections only to the words resident on that processor. For each processor, the score of the best path and a *label* are found. Encoded in fields of the label are a unique identification number for that processor and a pointer to the trellis node, which consists of the acoustic boundary and lexical state coordinates.

The scores and labels of the local endsets compactly encapsulate all the information that needs to be communicated to other processors. Each processor performs the Viterbi search as if were in isolation, with one minor change. For each column of the trellis, the best path to each endset is stamped with a unique identification number for that processor. These values are then *reconciled* with those in the other processors. The values are *merged* in a maximization of the scores to form global endset values. The global values are then *broadcast* to each node, replacing the local values.

5.3 Static Load Balancing

In addition to bootstrapping the NuMesh network, the host computer serves two important functions in the parallel Viterbi algorithm. First, it splits the lexical network, statically allocating the computational load among the processors. Second, it provides file service to the Numesh by downloading data. The program for the host computer was developed under the Think C environment on the Macintosh and is listed in Appendix C.

Because each processor must share its endset data before computation for the next column begins, the processor with the highest computational load will limit execution speed. To avoid bottlenecks, the lexical network should be distributed as smoothly as possible. One approach is to simply allocate the number of word models as evenly as possible. If the load associated with the models can be described by identical, independent distributions, then the weak law of large numbers predicts that the relative variance of their sum tends to be small.

If controlled fine tuning is needed, this approach can be generalized to developing a cost function, which depends linearly on parameters of the lexical network that can be distributed. For example, the function could depend on the number of nodes, arcs, and local endset connections, but not the number of endsets, which is the same for each processor. If the processors are clocked at different speeds, the lexical network can be split so that the cost function is allocated proportionately.

The data structures for the parallel Viterbi search are given by the type definitions listed in Appendix B. The DSP and Macintosh have different floating-point number representations, but fortunately the lexical network and AP network contain only integers, which have the same byte ordering on the DSP and Macintosh. The DSP does not support byte addressing, so character strings must be unpacked.

The use of absolute addresses in the data structures is also avoided because they lose meaning when stored in a file or transmitted to another processor through the network. For example, the arcs of the lexical network are stored in an ordered list. For each node, a table lists the number of incoming arcs. Pointers are not needed because the arcs are always processed in the same order. If this were not the case, the DSP could always construct pointers from the information in the table.

5.4 Tree Configuration

The communication required by the parallel Viterbi search algorithm could be supported by a bus, but at the expense of limited scalability. Unlike the bus, the Numesh parallel processing system makes the physical location of each node an explicit feature of the programming model. For the parallel Viterbi search, a logical configuration was embedded in the physical topology of NuMesh. This abstraction allows each processor to operate without awareness of the physical configuration.

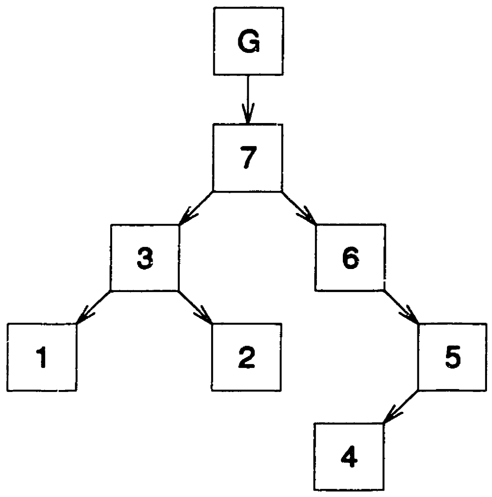

Figure 5.1: Sample Logical Configuration

5.4.1 Logical Configuration

In the logical configuration, the Viterbi search are constrained to communicate in a pattern of a binary tree. A tree need not be balanced and includes the extreme case of a linear chain, where each node (except the one farthest from the root) has a single child. An example of a tree is shown in Figure 5.1. The Viterbi search computation is distributed among the numbered nodes in the tree.

The node labeled G in Figure 5.1, called the *gateway* node, serves two functions. First, it acts as a parent to the root of the tree, allowing that node to draw from the same CFSM templates as the other nodes. Second, it isolates the tree from the host computer or other stages of the system. The gateway serves as a buffer, injecting data for the regions of the AP network as needed.

5.4.2 Physical Configuration

The logical configuration of the Viterbi search must be mapped onto the physical topology of the NuMesh. For the logical configuration shown in Figure 5.1, many physical configurations are possible. One is shown in Figure 5.2, along with a host computer interface.

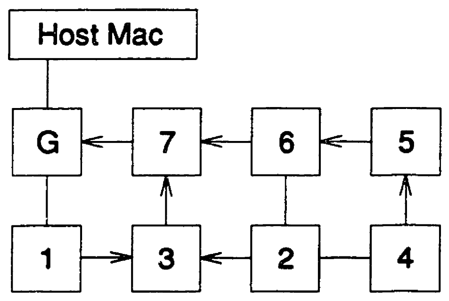

Figure 5.2: Sample Physical Configuration

Node	CFSM Compilation
G	`(gate north east)`
1	`(leaf east)`
2	`(leaf west)`
3	`(fork north west east)`
4	`(leaf north)`
5	`(link west south)`
6	`(link west east)`
7	`(fork west south east)`

Table 5.1: Sample CFSM Template Compilation

Note that not all logical configurations can be embedded in the physical topology conveniently. The number of nodes in a balanced binary tree grows as $O(2^n)$, where n is the depth, while the number of nodes in a two-dimensional mesh grows only as $O(n^2)$, where n is the diameter.

To simplify the compilation of CFSM routing code, each node in the Viterbi search belongs to one of three categories, depending on how many child nodes it has. A node with two children is a *fork* node; one, a *link* node; and none, a *leaf* node. The CFSM code for a node can be compiled by specifying the directions of the parent and child nodes, if any, in the compiler templates given Appendix A. For the configuration shown in Figure 5.2, the CFSM routing code is compiled according to Table 5.1.

5.5 Communication Protocols

To orchestrate the various programs in the parallel Viterbi search, a simple set of communication protocols was developed. These protocols are instrumental for in counting the nodes in the tree configuration, downloading the lexical networks to each node, broadcasting regions of the AP network to all of the nodes, reconciling the endsets, and backtracing the best path. This section details the mechanisms used to support these protocols.

5.5.1 Counting Nodes

The host computer must know the number of nodes in the tree in order to split the lexical network accordingly. Asking the user to provide this information would be inconvenient at compile time and error-prone at run time. Instead, this information can be determined automatically, just after bootstrap, from the configuration of the network.

The computation of the number of nodes in a tree can be expressed recursively. The number of nodes in a tree is equal to the number of nodes in the left subtree, plus the number of nodes in the right subtree, plus one. If a subtree does not exist, the number of nodes is taken to be zero, providing a base case for the recursion.

At bootstrap, a DSP program does not know the number of children the node has. It simply waits for the CFSM to report the number of nodes in the left and right subtrees. After these two values are received, the DSP adds one to their sum. This subtotal is reported to the CFSM for forwarding to the parent node. The root of the tree reports the grand total, through the gateway node, to the host computer.

Because the CFSM cannot generate constants, the host computer must supply two seed zero words. A fork CFSM duplicates both seed words for its children and reports their forwarded subtotals to the DSP. A link CFSM copies both zeros to its child, and reports one zero and the subtotal forwarded from the child to the DSP. A leaf node immediately supplies both zeros to the DSP.

5.5.2 Downloading Lexical Networks

The host computer must download a partial lexical network to each node in the tree. The tree is traversed in post-order, so that a node is visited only after all of its descendant nodes have been visited. The numbering of nodes in Figure 5.1 and Figure 5.2 shows this order for that example. After the DSP has received the lexical network, it sends an acknowledgement word to the CFSM for forwarding to its parent node.

The CFSM cannot examine the data it handles, so it cannot detect the conclusion of a download. Instead, it relies on the acknowledgement word from its child node or its DSP as a cue to change state. The host computer must allow adequate time between downloads for the DSP to acknowledge. Otherwise, a node will receive data that was meant for its parent node.

The lexical networks are preceded by a header word to allow a DSP to distinguish it from an acknowledgement from one of its children. The DSP program can count the number of acknowledgements it receives to determine the number of children it has. This information is useful for satisfying later protocols. Each node is also given a unique identification number at this time for use in creating endset labels.

5.5.3 Broadcasting Regions

The regions of the AP network must be broadcast to each node in the tree. The gateway node stores the regions received from the host computer or previous stages of the speech recognition system. At each boundary, it broadcasts the number of regions and the scores for those regions. The end of the utterance is signalled by sending a zero.

At first glance, it might appear that NuMesh is less efficient at broadcasting data than a bus, where all destination processors might be able to read each data value simultaneously. But while the data takes longer to travel to distant nodes in the NuMesh (latency), the propagation of the data is fully pipelined.

There are typically on the order of 100 phonetic model scores for each region of the

AP network, and at least a few regions ending at each acoustic boundary. Thus, the latency is negligible, and the average time to broadcast a data value approaches the time required to simply send the value to child nodes. This path is effectively a very short bus, which is substantially faster than a bus with many processors attached.

5.5.4 Reconciling Endsets

At each boundary, the endsets with the best scores over all the nodes are merged and broadcast. This mechanism effects transitions between words residing on different nodes. The endset reconciliation process is performed only after all of the scores and labels are computed. The variability in endset computation time tends to even out, as the weak law of large numbers applies here.

Like summation, the maximization can be expressed recursively. The maximum value in a tree is equal to the maximum value over the left subtree, right subtree, and the root of the tree. If no subtree exists, the value is taken to be the most negative integer, providing a base case for the recursion.

Each endset consists of two values, a score and a label. Each node collects the score and label from its children, if any. The endset with the best score is forwarded to the parent node. Note that the fork node must properly sequence these pairs of values from its two children. The global endset values are reported to the gateway node by the root of the tree.

The gateway node stores all of the endset values until the last pair is received, then broadcasts them to the tree. Both the collection and broadcast of data is pipelined, but the two cannot proceed at the same time. The DSP of each node would have difficulty interpreting the data it receives, which appears as a single stream.

This deficiency can be overcome by adding more CFSM hardware. One solution is for the CFSM to tag the data according to its source and the DSP to interpret these tags. Another solution is to support multiple virtual FIFO buffers in a shared memory interface. The location of the data could indicate how it should be interpreted.

5.5.5 Backtracing Path

The best path found by the Viterbi search must be constructed by backtracing the labels distributed among the nodes. When the end of the utterance is reached, a special endset with connection to all of the legitimate end nodes is computed. This endset is reconciled over all of the nodes in the tree to find the best score.

The label is broadcast to all of the nodes. Each DSP extracts the node field of the label to determine whether it was the creator of the label. If the label was not created by the node, the DSP reports a null word index and null label to be reported to the parent node. The root of the tree reports the backtraced label to the gateway node, which broadcasts a new label to be backtraced to the tree.

The gateway node terminates this process when an initial token word, which must begin the utterance, is received. The text for each word is found in a lookup table. The gateway concludes the execution by scrolling the elapsed time, score of the best path, and the textual word string through an 8-character alphanumeric display.

5.6 Summary

In the parallel decomposition of the Viterbi algorithm, the lexical network can be statically distributed among the processors by the host computer program. Each DSP can operate almost as if it were in isolation. Communication between the processors occurs in the logical pattern of a binary tree embedded in the physical topology of the NuMesh. Despite the limitations of the CFSM, a simple set of protocols can support efficient communication.

Chapter 6

Testing, Analysis, and Evaluation

6.1 Introduction

The previous chapter described a parallel Viterbi algorithm designed for Summit and implemented on NuMesh. This chapter describes testing of the implementation. An analytical model is developed, and performance data collected in evaluation of the model is explained.

6.2 Testing

The software implementation of the parallel Viterbi search algorithm was thoroughly tested during the development and debugging process. The software was developed an evolutionary fashion from a sequential to a parallel environment. The validity of the results was verified by direct comparison with results obtained from the Viterbi search module of the Summit system.

6.2.1 Development

The software for the parallel Viterbi search algorithm is divided into modules that can be tested in different combinations. A sequential Viterbi search program was developed under the UNIX environment. The module for the Viterbi search (`viterbi.c`) was linked with modules for handling the data structures of the lexical network (`lexicon.c`) and the AP network (`regions.c`). The correctness of these programs was verified by comparing the results to those produced by Summit for several AP networks on two different lexical networks.

The communications protocols were tested separately by diagnostic procedures in the Viterbi search tree interface module (`tree.c`). The alphanumeric displays of the DSP boards were used to trace the flow of data through NuMesh. A simple NuMesh utility program on the Macintosh host computer was used to read and write data from the NuMesh.

To form the final version, the Viterbi computation module was linked with the tree interface to form the DSP program. A load balancing program was linked with the data structure modules to form the host computer program. Additional procedures were written to transfer the data from the host computer to the NuMesh. A configuration of a single node was tested, then expanded to more complx configurations.

6.2.2 Debugging

Despite the careful development of the software, numerous bugs were encountered. Many of the bugs were difficult to isolate and correct because either they occurred in the development tools themselves or were obscured by the lack of debugging tools. This section describes some of the more notable bugs.

Not surprisingly, some bugs were found in the recently developed and untested CFSM compiler. Two consecutive cycles of access were not provided for the slower FIFO parts found in some of the hardware. Also, status bits were checked while being invalid for two cycles after a copy transaction. These bugs caused invalid data to be processed, and were discovered by writing test code.

A more pernicious bug was found in the interface software of the host computer. The procedure to write data to the network hangs until the transfer register is read. To avoid rebooting the host computer in case the value is never, a timeout is provided. Upon expiration of the timeout, rather than exit with an error, the procedure blithely over-wrote the old value. This bug appeared unpredictably because the data transfer rate depends on the highly variable speed of disk access. Of course, the bug did not appear while single-stepping with the debugger.

On the DSP, a FIFO access is treated as a memory-mapped I/O location. If a value read from the FIFO is never used, the compiler eliminated the reference even with the optimization flag off. If the display was used to verify the correct value was read, the bug disappeared because the value was used. This problem is akin to an analog circuit which works only while observed by an oscilloscope because the probe provides a ground. Declaring the destination location of the FIFO read as `volatile` in C did not completely solve the problem. One command was intended to multiply a value from the FIFO by two. Instead, the compiler added the value to itself to optimize the code, generating two separate references to the FIFO.

The final bug encountered was due to a very simple programming error. While the correct best path was always found, sometimes a random best score was reported because the value was read before it was written. On subsequent test runs, the stale value in memory from the previous run was read, making it appear that the program executed correctly. This error escaped detection for the longest because power cycles were rarely performed.

6.3 Analysis

The execution of the parallel Viterbi search algorithm alternates between two distinct phases. For each column, an array of local endset values is computed by each processor in one phase. In the other phase, the endsets are merged and broadcast in a burst of communication. An analytical model can be developed to predict the execution time for each phase.

6.3.1 Processing

No redundant computation results by allocating the word models of the lexical network among different processors. Assuming that each DSP has at least a few different word models, a static load balancing strategy can distribute the load of computing matches, insertions, and deletions evenly among the processors.

The distribution of the endset structure also does not require more computation. The maximization is simply distributed over the nodes in the tree. As the number of processors employed increases, however, fragmentation of the endset structure tends to reduce the potential impact of some optimizations.

Neglecting this lost opportunity, if one process can perform a Viterbi search for an unoptimized lexical network in P units of time, then n processors should require only P/n units of time. This type of decomposition would allow performance in direct proportion to the number of processors but for the overhead of communication which will be modeled next.

6.3.2 Communication

In the communication phase, the local endset values from the processors must be merged and broadcast to each node. Also, the scores for AP regions ending at the next boundary are broadcast to all of the processors. The links of NuMesh form a pipeline, allowing efficient communication.

Let E be the number of local endsets and C be the time required to merge and broadcast a single endset. If a backplane bus were used to support the parallel Viterbi search, the merging of endsets must be serialized. The time required for communication at each acoustic boundary would be nCE, where n is the number of processors.

Communication in NuMesh, on the other hand, is fully pipelined. The time to processor the endsets is CE, plus a latency, which grows with the number of processors. The depth of a tree grows as $O(\log n)$ at best, for a balanced tree, and $O(n)$ at worst, for a linear chain.

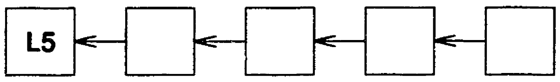

Figure 6.1: Linear Chain of Five Nodes

This range, however, is limited by the physical topology of the network. The latency of a two-dimensional mesh grows as $O(\sqrt{n})$ at best, for a completely populated mesh, and $O(n)$ at worst, for a linear chain. Thus we can model the latency as Ln^{α}, where $0.5 \leq \alpha \leq 1$. Summing the times for communication and processing, we find the total execution time to be:

$$T = Ln^{\alpha} + CE + P/n$$

6.4 Evaluation

A diagnostic procedure was included in the software for the host computer, gateway node, and tree nodes to measure the time required for communication in the absence of processing. The data collected bears out the key features of the model.

6.4.1 Chain Configurations

A special case of a tree is a linear chain. An example is shown in Figure 6.1. Here, $\alpha = 1$, so $T_c = CE + nL$. Using the diagnostic routines, performance data was collected for chains of various lengths in a wide range over the number of endsets.

Note the dramatic increase in communication time from chain L1 to chain L2. This phenomenon occurs because L1 contains only a leaf node, while longer chains must contain a link node. A link node must merge the data from its child node and thus is slower than a leaf node. This bottleneck limits the speed of the pipeline. Otherwise, adding a node to the chain increases the latency by about 4 μs, regardless of the number of endsets. Therefore, we have $L = 4\mu$s.

Communication Time (μs)

Chain	Number of Endsets 25	50	100	200	400
L1	108	169	307	561	1069
L2	171	294	557	1060	2065
L3	175	299	561	1064	2069
L4	179	302	565	1068	2072
L5	183	306	569	1072	2076
L6	187	310	573	1076	2080
L7	190	314	577	1079	2084
L8	194	318	580	1083	2088

Table 6.1: Communication Time for Linear Chains

Communication Time (μs)

Tree	Number of Endsets 25	50	100	200	400
T1	233	419	806	1557	3059
T2	237	423	810	1561	3063
T3	237	423	810	1561	3063
T4	238	423	810	1561	3063
T5	238	423	810	1561	3063
T6	241	426	814	1561	3063

Table 6.2: Communication Time for Various Trees

6.4.2 Tree Configurations

Performance data was also collected for the various trees shown in Figure 6.2. Note that all of the communication time for these configurations are greater than the longest chain, of length 8, listed above in Figure 6.1. The fork node is now the bottleneck which limits the speed of the pipeline. The use of a fork node must be justified by the reduced depth of the tree due to increased parallelization of the structure.

The configurations T2, T3, T4, and T5 have the same performance because they have the same depth. Like having multiple links in a chain, the additional fork of T5

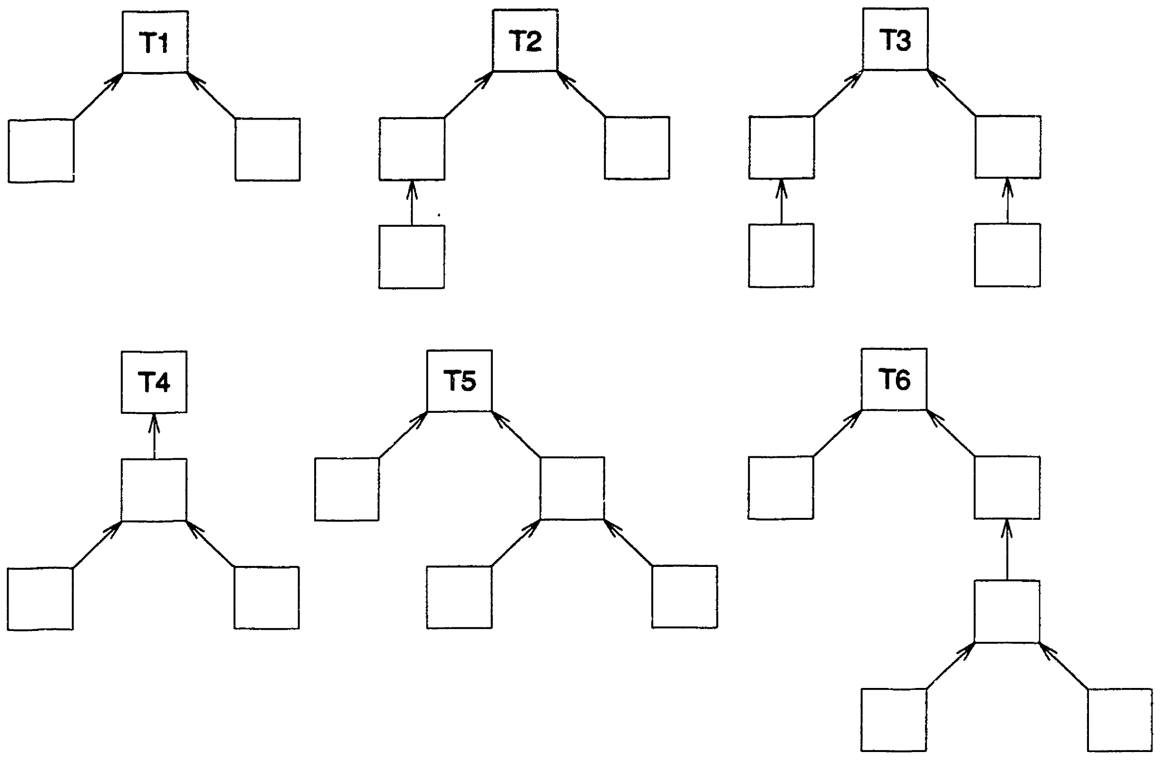

Figure 6.2: Various Tree Configurations

does not degrade performance. Configuration T6 is slightly deeper, resulting in the usual 4 μs increase.

6.5 Summary

The software implementation of the parallel Viterbi search algorithm was thoroughly tested and debugged. The execution alternates between a computation phase and communication phase. An analytical performance model suggests that computation can be divided equally among the processors, while communication costs increase slowly with the number of processors. Key features of the model were borne out by experimental data.

Chapter 7

Results

7.1 Introduction

Based upon the analytical model developed in the previous chapter, this chapter projects the relative performance of the parallel Viterbi search algorithm for a particular speech recognition task. Extensions of the algorithm which could be implemented on NuMesh are also described.

7.2 Projected Performance

Without the substantial memory required, it was not possible to determine the absolute performance of the algorithm on an actual lexical network and AP network. Relative performance, however, can be projected by the analytical model.

If we assume a linear chain configuration is used, then $\alpha = 1$. No forks nodes are needed in a chain, so the parameter C remains constant, with the exception of a single leaf node. If this anomaly is ignored, the relative performance is given by:

$$\frac{T(1)}{T(n)} = \frac{(C+L+P)n}{Cn^2 + Ln + P}$$

For the Airline Travel Information Service (ATIS) speech recognition task, there are 34 endsets. Extrapolating from Table 6.1, we can estimate $C = 215\mu s$. As before, we can set $L = 4\mu s$, which does not depend on the number of endsets.

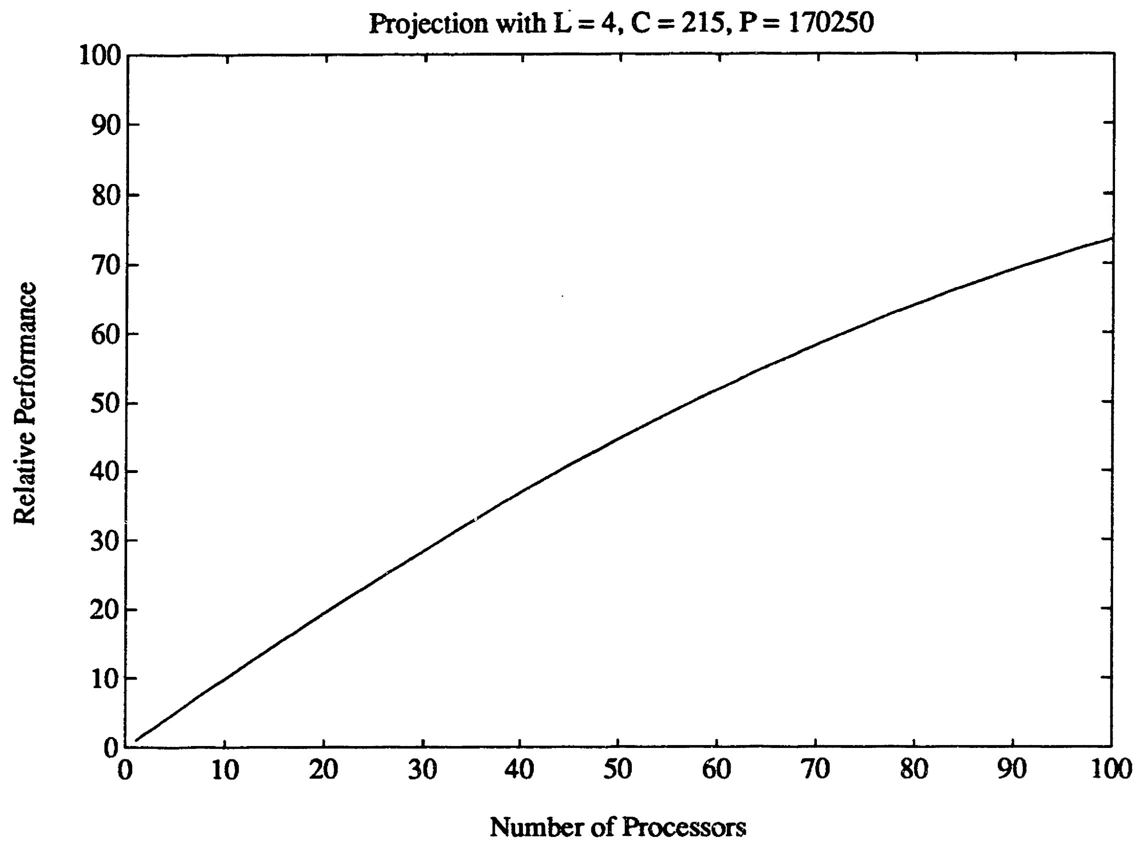

Figure 7.1: Projected Relative Performance

The software could recognize an utterance of 40 boundaries for the Citron lexical network in 227 milliseconds. The lexical network for ATIS is about 30 times as large, so we can estimate $P = 170250$ per boundary. This estimate will be smaller than the actual value because Citron does not require substantial endset computation.

Using these values for the parameters C, L, P, and α, the relative performance can be projected as shown in Figure 7.1. The performance scales well with the number of processors because little communication is performed relative to computation, and the communication overhead grows slowly with the number of processors.

The absolute performance of the algorithm can benefit from the same advances in technology as a uniprocessor program. For example, a custom processor with special hardware to perform the maximizations, similar to the multiply-and-accumulate features in the DSP, could be used to achieve immediate performance gains across the board.

7.3 NuMesh Extensions

Several extensions could be based upon the NuMesh implementation of the parallel Viterbi search algorithm. These include the word spotting recognizer, which performs multiple Viterbi searches, and an A* search which uses the estimates generated from the Viterbi search.

7.3.1 Word Spotting Recognizer

An application of speech recognition in the near future could be to help automate a telephone directory assistance line (411). A caller is first asked to state the city where the desired party is located. A speech recognition system could gather this information before transferring the caller to the operator. The system could have a limited vocabulary of isolated words, requiring little memory or time for processing.

Unfortunately, a substantial fraction of users give this information in the middle of their enquiry. A continuous speech recognition system, however, would require a complete vocabulary. A compromise between continuous word recognition and isolated word recognition is word spotting, recognizing a fragment of speech in the middle of an utterance.

A word-spotting recognizer simply initiates a small Viterbi search at each acoustic boundary in the utterance. If a good match to any word is found, the algorithm terminates. Otherwise, the Viterbi searches expire after a sufficient number of boundaries have passed.

NuMesh can trivially support such a speech recognition system. The nodes could initiate and terminate Viterbi search in a regular cycle. Regions of the AP network could be temporarily stored in a gateway buffer node and automatically routed to each processor.

7.3.2 Parallel A* Search

As described in Chapter 3, the Viterbi search can generate estimates for a stack decoder algorithm in an A* search. For NuMesh, such an A* search could be performed in the same tree configuration as the Viterbi search. Only the scores of the Viterbi search need to be maintained because the pointers are not used by the A* search.

The stack of the A* search could be distributed among the nodes of the tree. The gateway node can poll the processors to find the best path over all of the stacks. The score and best coordinates of this path could be broadcast to each node in the tree, which could extend the path by the the words it contains and insert the results into the local stack.

A complex grammar could filter paths that are reported at the gateway node. A more sophisticated approach is to distribute the grammar over the nodes of the tree to eliminate path extensions or adjust the score. The tree configuration can support many of the communication needs of the A* search with the same protocols described in Chapter 5.

7.4 Conclusion

The parallel Viterbi search algorithm requires little communication between processors relative to the amount of computing performed. Efficient, pipelined communication in NuMesh allows excellent scaling of performance with the number of processors. In addition, the flexibility of Numesh makes it an excellent platform for many extensions of the algorithm.

Bibliography

[Abdalla]	K. Abdalla, "A TMS320C30-Based Digital Signal Processing Board for a Prototype NuMesh Module," MIT Dept. of EECS, Master's Thesis, 1990.
[Bertsekas]	D. Bertsekas, *Dynamic Programming: Deterministic and Stochastic Models*, Prentice-Hall, 1987.
[Chow]	Y. Chow, R. Schwartz, "The N-Best Algorithm: An Efficient Procedure for Finding Top N Sentence Hypotheses," *DARPA Speech and Natural Language Workshop*, Oct. 1989, pp. 199–202.
[Glass]	J. Glass, V. Zue, "Multi-Level Acoustic Segmentation of Continuous Speech," IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing, New York, April 1988.
[Hon]	H. Hon, "A Survey of Hardware Architectures Designed for Speech Recognition," Carnegie Mellon Univ., Tech. Report CMU-CS-91-169, Aug. 1991.
[Honoré]	F. Honoré, "Redesign of a Prototype NuMesh Module", MIT Dept. of Dept. of EECS, Bachelor's Thesis, 1991.
[MacKenzie]	K. MacKenzie, "NuMesh Prototype Hardware Description", MIT Computer Architecture Group, NuMesh Memo 1, August 1990.
[Metcalf]	C. Metcalf, "A NuMesh Simulator," MIT Computer Architecture Group, NuMesh Memo 4, Feb. 1991.

[Nguyen] J. Nguyen, "CFSM Assembler Description," MIT Computer Architecture Group, NuMesh Memo 2, Jan. 1991.

[Pratt] G. Pratt, J. Nguyen, "Synchronization of Hardware Oscillators in a Mesh-Connected Parallel Processor," MIT Computer Architecture Group, NuMesh Memo 0.8, Jan. 1991.

[Rabiner] L. Rabiner, B. Juang, "An Introduction to Hidden Markov Models," *IEEE Acoustics, Speech, and Signal Processing Magazine*, Jan. 1986, pp. 4–16.

[Rabiner '75] L. Rabiner, B. Gold, *Theory and Application of Digital Signal Processing,* Prentice-Hall, 1975, pp. 681–7.

[Seneff] S. Seneff, "A Joint Synchrony/Mean-Rate Model of Auditory Speech Processing," *Journal of Phonetics*, 1988, vol. 16, pp. 55–76.

[Tessier] R. Tessier, "SPARC-Based Processing Element for the NuMesh," MIT Dept. of EECS, Master's Thesis, 1992.

[TMS320] *Third-Generation TMS320 User's Guide*, Texas Instruments Inc., 1991.

[Viterbi] A. Viterbi, "Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm," *IEEE Trans. on Information Theory* 1967, vol. IT-13, pp. 260–9.

[Ward] S. Ward, "The NuMesh: A Scalable, Modular 3D Interconnect", MIT Computer Architecture Group, NuMesh Memo 0.5, January 1991.

[Zue '89] V. Zue, J. Glass, M. Phillips, S. Seneff, "Acoustic Segmentation and and Phonetic Classification in the Summit System", IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing, Glasgow, May 1989.

[Zue '90] V. Zue, J. Glass, D. Goodine, M. Phillips, S. Seneff, "The Summit Speech Recognition System: Phonological Modelling and Lexical Access", IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing, Albuquerque, April 1990.

[Zue '91] V. Zue, "Integration of Speech Recognition and Natural Language Processing in the MIT Voyager System", IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing, Toronto, May 1991.

Appendix A

CFSM Compiler Source

A.1 Gateway Node (gate.nfsm)

```
;;; Gateway node sits between Mac host and root of tree.

(define (gate mac root)
  (repeat 2 (copyww mac root))          ; send seed zero words
  (copyww root mac)                     ; report total to Mac

  (while (not (ifull root))             ; root lex ack?
    (copycc mac root))                  ; download lex
  (copyii root)                         ; discard ack

  (while (not (ifull proc))             ; gateway data ack?
    (copycc mac proc))                  ; download to gate
  (copyii proc)                         ; discard gate ack

  (while true
    (sequence
      (copycc root proc)
      (copycc proc root))) )
```

A.2 Leaf Nodes (leaf.nfsm)

```
;;; A leaf node has no children; it's just a bidirectional pipe!

(define (leaf parent)
  (while true
    (sequence
      (copycc parent proc)
      (copycc proc parent))))
```

A.3 Link Nodes (link.nfsm)

```
;;; A link node has a single child.

(define (link parent child)

; Count number of nodes in tree.

  (copyww parent child proc)               ; propagate seed zero
  (copyww parent child)                    ; propagate seed zero
  (copyww child proc)                      ; count children nodes
  (copyww proc parent)                     ; report count+1 to parent

; Download lexicon to child node.

  (while (not (ifull child))               ; child lexicon ack?
    (copycc parent child))                 ; download to child
  (copyww child proc)                      ; send ack to proc

; Download lexicon to host proc.

  (while (not (ifull proc))                ; proc lexicon ack?
    (copycc parent proc))                  ; download to proc
  (copyww proc parent)                     ; send ack to parent

; Merge endsets from self and children.
; Broadcast new regions and best endsets.

  (while true
    (if (ifull child)
      (sequence                            ; merge endsets
        (repeat 2 (copyww child proc))       ; get child endsets
        (repeat 2 (copyww proc parent)))   ; report best endsets
      (copycw parent proc child))) )       ; else broadcast data
```

A.4 Fork Nodes (fork.nfsm)

```
;;; A fork node has two children.

(define (fork parent left right)

; Count number of nodes in tree.

  (copyww parent left right)            ; propagate seed zero
  (copyww parent left right)            ; propagate seed zero
  (copyww left proc)                    ; count nodes of left
  (copyww right proc)                   ; count nodes of right
  (copyww proc parent)                  ; report left+right+1

; Download lexicon to left node.

  (while (not (ifull left))             ; left lexicon ack?
    (copycc parent left))               ; download to left
  (copyiw left proc)                    ; send ack to proc

; Download lexicon to right node.

  (while (not (ifull right))            ; right lexicon ack?
    (copycc parent right))              ; download to right
  (copyiw right proc)                   ; send ack to proc

; Download lexicon to own proc.

  (while (not (ifull proc))             ; proc lexicon ack?
    (copycc parent proc))               ; download to proc
  (copyiw proc parent)                  ; send ack to parent

; Merge endsets from self and children.
; Broadcast new regions and best endsets.

  (while true
    (if (ifull left)
      (sequence                          ; merge endsets
        (repeat 2 (copyww left proc))         ; get left endsets
        (repeat 2 (copyww right proc))        ; get right endsets
        (repeat 2 (copyww proc parent)))      ; report best endsets
      (copycw parent proc left right))) ; else broadcast
```

Appendix B

Header Files for C Source

B.1 Viterbi Search (viterbi.h)

```
/* Viterbi Search Parameters */

#define DIAG 0
#define FLOOR -32768            /* minimum score value */
#define WTW    350              /* word transition weight */

/* Error Handling */

#if C30
#define error(mesg) Scroll(mesg)
#else
#define error(mesg) {fprintf(stderr,"ERROR! %s\n",mesg); exit(1);}
#endif

/* Endset Label Fields */

#define BOUND_MASK 0xfff00000
#define PNODE_MASK 0x000ff000
#define INDEX_MASK 0x00000fff

#define BOUND_SHIFT 20
#define PNODE_SHIFT 12
#define INDEX_SHIFT  0

#define BOUND_MAX 1000
#define PNODE_MAX (1 << (BOUND_SHIFT - PNODE_SHIFT))
#define INDEX_MAX (1 << (PNODE_SHIFT - INDEX_SHIFT))

/* Message Headers */

#define NEWLEX 256
#define ACKLEX 257
```

B.2 Lexical Network (lexicon.h)

```
/* Lexical Data Structures */

typedef struct
{ long num_nodes;                 /* number of nodes */
  long num_arcs;                  /* number of arcs */
  long text_index;                /* index into text */
} WORD;

typedef struct
{ long node_id;                   /* node index */
  long num_arcs;                  /* number of arcs */
  long word_id;                   /* word index */
} NODE;

typedef struct
{ long phoneme;                   /* phonetic model */
  long lex_score;                 /* lexical score */
  long prev_node;                 /* previous node */
  long del_score;                 /* deletion score */
} INARC;

typedef struct
{ long num_in;                    /* number of in arcs */
  long *nodes_in;                 /* list of nodes in */
  long num_out;                   /* number of out arcs */
  long *nodes_out;                /* list of nodes out */
/* long penalty */
} ENDSET;

typedef struct
{ long num_words;                 /* number of words */
  WORD *words;                    /* list of words */
  long num_nodes;                 /* number of nodes */
  NODE *nodes;                    /* list of nodes */
  long num_arcs;                  /* number of arcs */
  INARC *arcs;                    /* list of arcs */
  long num_endsets;               /* number of endsets */
  ENDSET *endsets;                /* list of endsets */
  long text_size;                 /* size of text*/
  char *word_text;                /* text for words */
} WORDLEX;
```

```
typedef struct
{ long num_nodes;                /* number of local nodes */
  long *arcs_per_node;           /* number arcs per node */
  long *word_for_node;           /* word index for node */
  long num_arcs;                 /* number of arcs */
  INARC *arcs;                   /* list of arcs */
  long num_endsets;              /* number of endsets */
  ENDSET *endsets;               /* list of endsets */
} NODELEX;

/* Procedure Declarations */

WORDLEX *load_lex();
```

B.3 AP Network (regions.h)

```
/* AP Network Parameters */

#define MODELS 106               /* number of phonetic models */
#define REGSIZE (MODELS+2)       /* number of words in packet */
#define REG_MAX 15               /* maximum number of regions */

/* Data Structures */

typedef struct
{ long del_score;
  long left_bound;
  long scores[MODELS];
} REGION;

typedef struct
{
  long num_bounds;
  long *num_regions;
  REGION **regions;
} APNET;

/* Procedure Declarations */

APNET *load_apnet();
APNET *read_apnet();
```

Appendix C

Host Program Source

C.1 Load Balancing (splitlex.c)

```
/* Think C Source for Macintosh Host */

#include <stdio.h>
#include <stdlib.h>
#include "meshio.h"
#include "viterbi.h"
#include "lexicon.h"
#include "regions.h"

main()
{
  WORDLEX *rootlex;
  NODELEX *lex;
  APNET *apnet;
  NODE *nodes;
  INARC *arcs;
  WORD *words;
  long *id_table;
  int num_lex;
  long index;
  int i,j,k;

#if !DIAG

/* Read lexical network and AP network from files. */
/* Stupid Mac doesn't have command line or symlinks, */
/* so duplicate files and rename them appropriately. */

  rootlex = load_lex("citron.lex");
  apnet = load_apnet("citron.net");

#endif
```

```
/* Must run this NuMesh I/O initialization procedure!!! */

  CheckForNumesh();

/* Send two seed zeros to count number of NuMesh nodes. */

  write_numesh(0);
  write_numesh(0);
  num_lex = read_numesh();

#if DIAG
  host_diagnostic(num_lex);
#else

/* Create translation table for splitting lexical network. */

  id_table = (long *) calloc(rootlex->num_nodes, sizeof(long));
  if (!id_table) error("split_lex: calloc failed");

/* Initialize some pointers into the lexical network. */

  index = 0;
  nodes = rootlex->nodes;
  arcs = rootlex->arcs;

/* Download a lexicon to each Viterbi node. */

  for (i=0; i<num_lex; i++)
    {

  /* Create a lexicon and initialize. */

      lex = (NODELEX *) malloc(sizeof(NODELEX));
      if (!lex) error("main: malloc NODELEX failed");

      lex->num_nodes = 0;
      lex->num_arcs = 0;
      lex->num_endsets = rootlex->num_endsets;

  /* Decide how many words to put in each lexicon. */

      while (index < (rootlex->num_words * (i+1)) / num_lex)
       {
         lex->num_nodes += rootlex->words[index].num_nodes;
         lex->num_arcs += rootlex->words[index].num_arcs;
         index++;
       }
  /* Clear translation table for global to local index. */

      for (j=0; j<rootlex->num_nodes; j++)
        id_table[j] = -1;

  /* Flesh out the nodes, arcs, and endsets. */

      add_nodes (lex, nodes, id_table);
      add_arcs (lex, arcs, id_table);
      add_endsets (lex, rootlex->endsets, id_table);

  /* Download the lexicon to current node. */

      write_numesh(NEWLEX);      /* lexicon header */
      write_numesh(i);           /* unique node_id */
      write_lex(lex);            /* lexicon itself */

  /* Update the pointers into the lexical network. */

      nodes += lex->num_nodes;
      arcs += lex->num_arcs;
      free(lex);

  /* Pause a bit to allow node plenty of time to acknowledge. */
  /* Unlike C30, Think C is too stupid to optimize dead loops. */

      for (j=0; j<1000; j++);
    }

/* Download Parameters to Gateway Node */

  /* (1) Number of endsets, sans 2 for init. hack. */

  write_numesh(2 * (rootlex->num_endsets-2));

  /* (2) Translation table for global to local index. */

  write_numesh(rootlex->num_words);
  for (i=0; i<rootlex->num_words; i++)
    write_numesh(rootlex->words[i].text_index);

  /* (3) Chunk of text for words in lexical network */

  write_numesh(rootlex->text_size);
  for (i=0; i<rootlex->text_size; i++)
    write_numesh(rootlex->word_text[i]);
  /* (4) Complete AP network to the Gateway Node */

  write_apnet(apnet);

#endif

}


host_diagnostic(num_lex)
 int num_lex;

{
  int i;

  for (i=0; i<num_lex; i++)
    write_numesh(NEWLEX);
  write_numesh(num_lex);
}
```

C.2 Lexical Network (lexicon.c)

```
/* Utility Procedures for Lexical Networks */

#include <stdio.h>
#include <stdlib.h>
#include "meshio.h"
#include "viterbi.h"
#include "lexicon.h"

#define MAGIC_NUM 5150          /* file header */

#define load_long(x) fread(&x,1,sizeof(long),fp)


/* Load WORDLEX from file */

WORDLEX *load_lex (filename)
     char *filename;
{
  FILE *fp;
  WORDLEX *lex;
  long header;
  long i;
/* Open file, check header */

  fp = fopen(filename, "rb");
  if (!fp) error("load_lex: file not opened");
  load_long(header);
  if (header != MAGIC_NUM) error("Bad magic number in file");

/* Allocate WORDLEX storage */

  lex = (WORDLEX *) malloc(sizeof(WORDLEX));
  if (!lex) error("load_lex: malloc WORDLEX failed");

/* Load words */

  load_long(lex->num_words);
  lex->words = (WORD *) calloc(lex->num_words, sizeof(WORD));
  if (!lex->words) error("load_lex: calloc WORDs failed");
  fread(lex->words, lex->num_words, sizeof(WORD), fp);

/* Load nodes */

  load_long(lex->num_nodes);
  lex->nodes = (NODE *) calloc(lex->num_nodes, sizeof(NODE));
  if (!lex->nodes) error("load_lex: malloc NODEs failed");
  fread(lex->nodes, lex->num_nodes, sizeof(NODE), fp);

/* Load arcs */

  load_long(lex->num_arcs);
  lex->arcs = (INARC *) calloc(lex->num_arcs, sizeof(INARC));
  if (!lex->arcs) error("load_lex: calloc INARCs failed");
  fread(lex->arcs, lex->num_arcs, sizeof(INARC), fp);

/* Load endsets */

  load_endsets(lex, fp);

/* Load text */

  load_long(lex->text_size);
  lex->word_text = (char *) malloc(lex->text_size);
  if (!lex->word_text) error("load_lex: malloc text failed");
  fread(lex->word_text, lex->num_arcs, sizeof(char), fp);

  fclose(fp);
  return(lex);
}
```

```
load_endsets (lex, fp)
     WORDLEX *lex;
     FILE *fp;
{ long i;

  load_long(lex->num_endsets);
  lex->endsets = (ENDSET *) calloc(lex->num_endsets, sizeof(ENDSET));
  if (!lex->endsets) error("Could not calloc ENDSETs");

  for (i=0; i<lex->num_endsets; i++)
    { ENDSET *set = &lex->endsets[i];

      load_long(set->num_in);
      if (set->num_in)
        {
         set->nodes_in = (long *) calloc(set->num_in, sizeof(long));
         if (!set->nodes_in) error("load_endsets: calloc in failed");
         fread(set->nodes_in, set->num_in, sizeof(long),fp);
        }

      load_long(set->num_out);
      if (set->num_out)
        {
         set->nodes_out = (long *) calloc(set->num_out, sizeof(long));
         if (!set->nodes_out) error("load_endsets: calloc out failed");
         fread(set->nodes_out, set->num_out, sizeof(long), fp);
        }

    }
}


/* Produce text dump of WORDLEX */

dump_lex (lex)
     WORDLEX *lex;
{
  NODE *node = lex->nodes;
  INARC *arc = lex->arcs;
  long i,j,k;

#if MAC
  FILE *out = fopen("citron.dump","w");
#else
  FILE *out = stdout;
#endif
```

```
  fprintf(out,"Lexicon has %ld words",lex->num_words);
  fprintf(out," with %ld arcs",lex->num_arcs);
  fprintf(out," and %ld nodes",lex->num_nodes);
  fprintf(out,":\n\n");

  for (i=0; i<lex->num_words; i++)
    {
      WORD *word = &lex->words[i];

      fprintf(out," Word '%s'",&lex->word_text[word->text_index]);
      fprintf(out," has %ld nodes",word->num_nodes);
      fprintf(out," and %ld arcs",word->num_arcs);
      fprintf(out,":\n");

      for (j=0; j<word->num_nodes; j++)
        {
          for (k=0; k<node->num_arcs; k++)
            {
              fprintf(out,"  Arc to %4ld",node->node_id);
              fprintf(out," from %4ld",arc->prev_node);
              fprintf(out," has label %3ld",arc->phoneme);
              fprintf(out," with lex score %4ld",arc->lex_score);
              fprintf(out," and del score %4ld",arc->del_score);
              fprintf(out,".\n"); arc++;
            }
          node++;
        }
      fprintf(out,"\n");
    }

  fprintf(out,"Lexicon has %ld endsets\n",lex->num_endsets);
  for (i=0; i<lex->num_endsets; i++)
    {
      ENDSET *set = &lex->endsets[i];
      fprintf(out,"  Endset %ld",i);
      fprintf(out," has %ld in",set->num_in);
      fprintf(out," and %ld out",set->num_out);
      fprintf(out,"\n IN: ");
      for (j=0; j<set->num_in; j++)
        fprintf(out,"%5ld",set->nodes_in[j]);
      fprintf(out,"\n OUT:");
      for (j=0; j<set->num_out; j++)
        fprintf(out,"%5ld",set->nodes_out[j]);
      fprintf(out,"\n\n");
    }

}
```

```
/* Add nodes to NODELEX */

add_nodes (lex, nodes, table)
     NODELEX *lex;
     NODE *nodes;
     long *table;
{
  long i,j;
  long num_nodes = lex->num_nodes;

  lex->arcs_per_node = (long *) calloc(num_nodes, sizeof(long));
  lex->word_for_node = (long *) calloc(num_nodes, sizeof(long));
  if (!lex->arcs_per_node || !lex->word_for_node)
    error("add_nodes: calloc NODEs failed");

  for (i=0; i<num_nodes; i++)
    {
      lex->arcs_per_node[i] = nodes[i].num_arcs;
      lex->word_for_node[i] = nodes[i].word_id;
      table[nodes[i].node_id] = i;
    }
}


/* Add arcs to NODELEX */

add_arcs (lex, arcs, table)
     NODELEX *lex;
     INARC *arcs;
     long *table;
{
  int num_arcs = lex->num_arcs;
  int i;

  lex->arcs = (INARC *) calloc(num_arcs, sizeof(INARC));
  if (!lex->arcs) error("add_arcs: calloc failed");

  for (i=0; i<num_arcs; i++)
    { INARC *arc = &lex->arcs[i];
      arc->phoneme = arcs[i].phoneme;
      arc->lex_score = arcs[i].lex_score;
      arc->prev_node = table[arcs[i].prev_node];
      arc->del_score = arcs[i].del_score;
    }
}
```

```
/* Add endsets to NODELEX */

add_endsets(lex, sets, table)
     NODELEX *lex;
     ENDSET *sets;
     long *table;
{
  int i,j;

  lex->endsets = (ENDSET *) calloc(lex->num_endsets, sizeof(ENDSET));
  if (!lex->endsets) error("add_endsets: calloc ENDSETs failed");

  for (i=0; i<lex->num_endsets; i++)
    { ENDSET *set = &lex->endsets[i];
      long *nodes;

      set->num_in = 0;
      for (j=0; j<sets[i].num_in; j++)
        if (table[sets[i].nodes_in[j]] != -1)
          set->num_in++;

      set->nodes_in = (long *) calloc(set->num_in, sizeof(long));
      if (!set->nodes_in)
        error("add_endsets: calloc nodes in failed");

      nodes = set->nodes_in;
      for (j=0; j<sets[i].num_in; j++)
        if (table[sets[i].nodes_in[j]] != -1)
          *nodes++ = table[sets[i].nodes_in[j]];

      set->num_out = 0;
      for (j=0; j<sets[i].num_out; j++)
        if (table[sets[i].nodes_out[j]] != -1)
          set->num_out++;

      set->nodes_out = (long *) calloc(set->num_out, sizeof(long));
      if (!set->nodes_out)
        error("add_endsets: calloc nodes out failed");

      nodes = set->nodes_out;
      for (j=0; j<sets[i].num_out; j++)
        if(table[sets[i].nodes_out[j]] != -1)
          *nodes++ = table[sets[i].nodes_out[j]];

    }
}
```

```
write_lex (lex)
     NODELEX *lex;
{
  int i;

  write_block(lex->arcs_per_node, lex->num_nodes);
  write_block(lex->word_for_node, lex->num_nodes);
  write_block(lex->arcs, 4 * lex->num_arcs);
  write_numesh(lex->num_endsets);
  for (i=0; i<lex->num_endsets; i++)
    { ENDSET *set = &lex->endsets[i];
      write_block(set->nodes_in, set->num_in);
      write_block(set->nodes_out, set->num_out);
    }
}


write_block (buf, num)
     long *buf;
     long num;
{
  int i;

  write_numesh(num);
  for (i=0; i<num; i++)
    write_numesh(*buf++);
}
```

C.3 AP Network (regions.c)

```
#include <stdio.h>
#include <stdlib.h>
#include "viterbi.h"
#include "meshio.h"
#include "regions.h"

#define load_long(x) fread(&x,1,sizeof(long),fp)

#define MAGIC_NUM 1984          /* magic number for file */


APNET *load_apnet(filename)
     char *filename;
```

```
{
  FILE *fp;
  APNET *apnet;
  long header;
  int i, j;

/* Open file, check headers */

  fp = fopen(filename,"rb");
  if (!fp) error ("load_apnet: File not opened");
  load_long(header);
  if (header != MAGIC_NUM) error("load_apnet: Bad magic number in file");
  load_long(header);
  if (header != MODELS) error("load_apnet: Wrong number of MODELS");

/* Allocate APNET storage */

  apnet = (APNET *) malloc(sizeof(APNET));
  if (!apnet) error("load_apnet: malloc APNET failed");

/* Allocate boundary storage */

  load_long(apnet->num_bounds);
  apnet->num_regions =  (long *) calloc(apnet->num_bounds, sizeof(long));
  apnet->regions = (REGION **) calloc(apnet->num_bounds, sizeof(REGION *));

  if (!apnet->num_regions || !apnet->regions)
    error("load_apnet: malloc boundaries failed");

/* Fill boundary storage */

  fread(apnet->num_regions, apnet->num_bounds, sizeof(long), fp);

  for (i=0; i<apnet->num_bounds; i++)
    {
      int num = apnet->num_regions[i];
      apnet->regions[i] = (REGION *) calloc(num, sizeof(REGION));
      if (!apnet->regions[i])
        error("load_apnet: calloc REGIONs failed");
      fread(apnet->regions[i], num, sizeof(REGION), fp);
    }
  fclose(fp);
  return(apnet);
}
```

```
dump_apnet(apnet)
     APNET *apnet;
{
  int i,j,k;

  printf("APNET has %d bounds ",apnet->num_bounds);

  for (i=0; i<apnet->num_bounds; i++)
    {
      for (j=0; j<apnet->num_regions[i]; j++)
        {
          REGION *reg = &apnet->regions[i][j];

          printf(" Region %d",j);
          printf(" has left bound %2d",reg->left_bound);
          printf(" and del penalty %d\n",reg->del_score);
        }
    }
}


write_apnet(apnet)
     APNET *apnet;
{
  int i,j, k;
  long wait;

  write_numesh(apnet->num_bounds);

  for (i=0; i<apnet->num_bounds; i++)
    { long *longptr = (long *) apnet->regions[i];

      write_numesh(apnet->num_regions[i]);
      for (j=0; j<apnet->num_regions[i]; j++)
        write_region(&apnet->regions[i][j]);
    }
}


write_region (region)
     REGION *region;
{
  int i,j,k;
  long *longptr = (long *) region;

  for (k=0; k<REGSIZE; k++)
    write_numesh(*longptr++);
}
```

Appendix D

DSP Program Source

D.1 Gateway Node (gateway.c)

```
/* C30 Source for Gateway Node */

#include <stdlib.h>
#include "numesh.h"
#include "viterbi.h"
#include "regions.h"

/* Parameters */

#define WORD_MAX 20                     /* maximum words in string */
#define CLOCK_FREQ (27 * 1000000)       /* DSP has 27.000 MHz clock, */
#define MILLISEC (CLOCK_FREQ/4000)      /* divided by 4 for counter. */

/* Peripheral Time Registers */
/* Refer to C30 User's Guide */

volatile long *Control = (long *) 0x808020;
volatile long *Counter = (long *) 0x808024;
volatile long *Period  = (long *) 0x808028;


main()
{
  APNET *apnet;
  volatile long num_sets;
  volatile long *sets;
  volatile long num_words;
  volatile long *words;
  volatile long size_text;
  volatile char *text;
  volatile long score, index, label;
  long sentence[WORD_MAX];
  char banner[150] = "         ";
  int i, j, count;
  char mesg[10];
  int time;

  WriteFifo(0);                 /* boot word */
  display ("WAITING");

#if DIAG
  gate_diagnostic();
#else

/* Allocate storage for endsets, in fast RAM0 bank if possible. */

  num_sets = ReadFifo();
  if (sets < 512) sets = (long *) 0x809800;
  else sets = (long *) malloc (num_sets * sizeof(int));
  if (!sets) error ("main: malloc set_info failed!");

/* Receive table and text for words. */

  display ("LEXICON");
  num_words = ReadFifo();
  words = (long *) malloc (num_words * sizeof(int));
  if (!words) error("main: malloc word_table failed!");
  for (i=0; i<num_words; i++) words[i] = ReadFifo();

  size_text = ReadFifo();
  text = (char *) malloc (size_text * sizeof(char));
  if (!text) error("main: malloc word_text failed");
  for (i=0; i<size_text; i++) text[i] = ReadFifo();

/* Receive acoustic-phonetic network. */

  display ("A-P NET");
  apnet = read_apnet();

/* Acknowledge data from Mac */

  WriteFifo(0);
  display ("GATEWAY");

/* Reset peripheral timer */

  *Counter = 0;
  *Period = 0x7fffffff;
  *Control = 0x2c1;
```

```
/* Act as parent node of root, inject data as needed. */

  for (i=1; i<apnet->num_bounds; i++)
    {
      volatile long *outptr = (long *) apnet->regions[i];

  /* Broadcast new regions to the tree. */

      WriteFifo (apnet->num_regions[i]);
      for (j=0; j<apnet->num_regions[i] * REGSIZE; j++)
        WriteFifo (*outptr++);

  /* Echo the values for the endsets. */

      for (j=0; j<num_sets; j++) sets[j] = ReadFifo();
      for (j=0; j<num_sets; j++) WriteFifo(sets[j]);
    }

/* Help reconcile final score and endset. */

  WriteFifo(0);
  score = ReadFifo();
  label = ReadFifo();
  writefifo (score);
  WriteFifo (label);

/* Backtrace pointers for reverse path. */

  index = 0;
  count = 0;

  while (index != 1 && count < 20)
    {
      index = ReadFifo();
      if (index < 0 || index > num_words)
        error("main: word out of range");
      sentence[count++] = words[index];
      WriteFifo (index);
      label = ReadFifo();
      WriteFifo (label);
    }


/* Display the results */

  time = *Counter/MILLISEC;
  strcat (banner," Time: ");
  ltoa (time,mesg);
  strcat (banner,mesg);

  strcat (banner," Score: ");
  ltoa (score,mesg);
  strcat (banner,mesg);

  strcat (banner," Path: ");
  while (--count >= 0)
    {
      strcat (banner, &(text[sentence[count]]));
      strcat (banner," ");
    }

  while(1) Scroll (banner);

#endif

}


#if DIAG

gate_diagnostic()
{
  int i, j;
  int num_sets;
  int time;
  char banner[100], mesg[20];
  int *sets = (int *) 0x809800;
  volatile int val;

  display("GATE");

  val = ReadFifo();
  display(val,2000000);
  WriteFifo(0);

  num_sets = 5;
  WriteFifo(num_sets);
  for (j=0; j<2*num_sets; j++)
    sets[j] = ReadFifo();
  for (j=0; j<2*num_sets; j++)
    WriteFifo (sets[j]);

  for (num_sets=25; num_sets<501; num_sets*= 2)
    {
      WriteFifo(num_sets);
      *Counter = 0;
      *Period = 0x7fffffff;
      *Control = 0x2c1;

      for (j=0; j<2*num_sets; j++)
        sets[j] = ReadFifo();
      for (j=0; j<2*num_sets; j++)
        WriteFifo (sets[j]);

      time = *Counter;

      strcpy(banner,"  ");
      strcat(banner," SETS ");
      ltoa(num_sets,mesg);
      strcat(banner,mesg);
      strcat(banner," TIME ");
      ltoa(time,mesg);
      strcat(banner,mesg);

      Scroll(banner);

    }

}

#else


APNET *read_apnet()
{
  APNET *apnet;
  volatile long num_bounds;
  int i, j, k;

  apnet = (APNET *) malloc (sizeof(APNET));
  if (!apnet) error ("read_apnet: malloc APNET failed");

  num_bounds = ReadFifo();
  apnet->num_bounds = num_bounds;

  apnet->num_regions = (long *) calloc (num_bounds, sizeof(long));
  if (!apnet->num_regions) error ("read_apnet: malloc APNET failed");

  for (i=0; i<apnet->num_bounds; i++)
    {
      volatile long *inptr;
      volatile long num_regions;
      num_regions= ReadFifo();
      apnet->num_regions[i] = num_regions;

      apnet->regions[i] = (REGION *) calloc (num_regions, sizeof(REGION));
      if (!apnet->regions[i]) error("read_apnet: calloc REGIONs failed");

      inptr = (long *) apnet->regions[i];
      for (j=0; j < (MODELS+2) * num_regions; j++)
        *inptr++ = ReadFifo();
    }
  return (apnet);
}

#endif
```

D.2 Tree Interface (tree.c)

```
#include <stdlib.h>
#include <string.h>
#include "numesh.h"
#include "viterbi.h"
#include "lexicon.h"
#include "regions.h"

NODELEX *read_lex();
volatile int node_id, num_child;

main()
{
  int count;
  volatile int header;
  volatile int dummy;
  NODELEX *lex;

  WriteFifo(0);                   /* boot word */
  display("WAITING");

/* Help count number of nodes in tree. */

  count = 1;
  dummy = ReadFifo();
  count += dummy;
  dummy = ReadFifo();
  count += dummy;
  WriteFifo(count);
  display_num(count, 0);
```

```
/* Count number of children for node. */

  num_child = 0;
  header = ReadFifo();
  while (header == ACKLEX)
    { num_child++;
      header = ReadFifo();
    }
  display_num(num_child, 0);

#if DIAG
  tree_diagnostic();
#else

  if (header != NEWLEX)
    error("main: bad NEWLEX header");
  display("LEXICON");

  node_id = ReadFifo();
  if (!(node_id < PNODE_MAX))
    error("main: node_id exceeded pnode_max");

  lex = read_lex();
  WriteFifo(ACKLEX);
  if (!(lex->num_nodes < INDEX_MAX))
    error("main: lex->num_nodes exceeded index_max");

  display("VITERBI");
  compute_matrix(lex);

#endif

}


find_best_score (word_id, set, scores, labels, bound)
     long *word_id;
     ENDSET *set;
     int bound;
     long **scores, **labels;
{
  char banner[100];
  char mesg[10];
  volatile long score, label;
  long pnode, index;
  volatile long word;
  int i, j, k;
  score = FLOOR;
  for (i=0; i<set->num_in; i++)
    if (score < scores[bound][set->nodes_in[i]])
      {
                index = set->nodes_in[i];
                score = scores[bound][index];
      }

  if (score == FLOOR) label = -1;
  else label = index + (bound << BOUND_SHIFT) + (node_id << PNODE_SHIFT);

  reconcile(&score, &label, 1);

  while (label != -1)
    {
      volatile long val;

      word = -1;
      pnode = (label & PNODE_MASK) >> PNODE_SHIFT;

      if (node_id == pnode)
        {
          bound = (label & BOUND_MASK) >> BOUND_SHIFT;
          index = (label & INDEX_MASK) >> INDEX_SHIFT;
          label = labels[bound][index];
          word = word_id[index];
        }

      for (j=0; j<num_child; j++)
        {
          val = ReadFifo();
          if (val != -1)
            {
              word = val;
              label = ReadFifo();
            }
          else DiscardFifo();
        }

      WriteFifo(word);
      WriteFifo(label);

      word = ReadFifo();
      label = ReadFifo();
    }

}
```

```
reconcile (score, label, num_sets)
  long *score, *label;
  int num_sets;
{
  int i,j;
  volatile long val;

  for (i=0; i<num_sets; i++)
    {

      for(j=0; j<num_child; j++)
        {
          val = ReadFifo();
          if (score[i] < val)
            {
              score[i] = val;
              label[i] = ReadFifo();
            }
          else val = ReadFifo();
        }

      WriteFifo(score[i]);
      WriteFifo(label[i]);

    }

  for (i=0; i<num_sets; i++)
    {
      score[i] = ReadFifo();
      label[i] = ReadFifo();
    }

}


#if DIAG

tree_diagnostic()
{
  int *score = (int *) 0x809800;
  int *label = (int *) 0x809A00;
  int i,j,k;
  volatile int val;

  WriteFifo(ACKLEX);
  for (i=0; i<512; i++)
    {
      score[i] = i;
      label[i] = i * i;
    }

  display("DIAG");

  while(1)
    {
      val = ReadFifo();
      /* display_num (val,0); */
      reconcile (score, label, val);
    }
}


#else


NODELEX *read_lex()
{
  NODELEX *lex;
  long *longptr;
  int i, j, k;

  lex = (NODELEX *) malloc(sizeof(NODELEX));
  if (!lex) error("read_lex: malloc NODELEX failed");

  lex->num_nodes = ReadFifo();
  lex->arcs_per_node = (long *) calloc(lex->num_nodes, sizeof(long));
  if (!lex->arcs_per_node) error("read_lex: calloc NODEs failed");
  for (i=0; i<lex->num_nodes; i++)
    lex->arcs_per_node[i] = ReadFifo();

  lex->num_nodes = ReadFifo();
  lex->word_for_node = (long *) calloc(lex->num_nodes, sizeof(long));
  if (!lex->word_for_node) error("read_lex: calloc NODEs failed");
  for (i=0; i<lex->num_nodes; i++)
    lex->word_for_node[i] = ReadFifo();

  lex->num_arcs = ReadFifo();
  lex->arcs = (INARC *) calloc(lex->num_arcs, sizeof(long));
  if (!lex->arcs) error("read_lex: calloc ARCs failed");
  longptr = (long *) lex->arcs;
  for (i=0; i<lex->num_arcs; i++)
    *longptr++ = ReadFifo();
  lex->num_arcs /= 4;

  lex->num_endsets = ReadFifo();
  lex->endsets = (ENDSET *) calloc(lex->num_endsets, sizeof(ENDSET));
  if (!lex->endsets) error("read_lex: calloc ENDSETs failed");
  for (i=0; i<lex->num_endsets; i++)
    {
      ENDSET *set = &lex->endsets[i];

      set->num_in = ReadFifo();
      if (set->num_in)
        {
          set->nodes_in = (long *) calloc(set->num_in, sizeof(long));
          if (!set->nodes_in) error("read lex: calloc nodes_in failed");
          for (j=0; j<set->num_in; j++)
            set->nodes_in[j] = ReadFifo();
        }

      set->num_out = ReadFifo();
      if (set->num_out)
        {
          set->nodes_out = (long *) calloc(set->num_out, sizeof(long));
          if (!set->nodes_out) error("read lex: calloc nodes_out failed");
          for (j=0; j<set->num_out; j++)
            set->nodes_out[j] = ReadFifo();
        }
    }

  return(lex);
}



read_regions (regions)
     REGION *regions;
{
  volatile long *inptr;
  volatile int num_regions;
  int i;

  num_regions = ReadFifo();
  if (num_regions > REG_MAX)
    error("read_regions: too many regions");

  inptr = (long *) regions;
  for (i=0; i<num_regions*REGSIZE; i++)
    *inptr++ = ReadFifo();
  return(num_regions);
}


#endif
```

D.3 Viterbi Search (viterbi.c)

```
/* Viterbi Computation Module */

#include <stdlib.h>
#include "numesh.h"
#include "viterbi.h"
#include "lexicon.h"
#include "regions.h"


REGION regbuf[REG_MAX];
long *scores[BOUND_MAX], *labels[BOUND_MAX];
extern int node_id;


compute_matrix (lex)
     NODELEX *lex;
{
  int num_regions;
  REGION *regions = regbuf;
  long *setscore, *setlabel;
  int bound;
  int i, j;

#if C30

/* Allocate endsets in fast RAMO bank of C30 if possible */

  if (lex->num_endsets < 512)
    { setscore = (long *) 0x809800;
      setlabel = (long *) 0x809A00;
    }
  else
    { setscore = (long *) calloc(lex->num_endsets, sizeof(long));
      setlabel = (long *) calloc(lex->num_endsets, sizeof(long));
    }

#else

  setscore = (long *) calloc(lex->num_endsets, sizeof(long));
  setlabel = (long *) calloc(lex->num_endsets, sizeof(long));

#endif

  if (!setscore || !setlabel) error("calloc escores/elabels failed");

  init_column (&lex->endsets[0], scores, labels, lex->num_nodes);

  bound = 1;
  num_regions = read_regions(regions);

  while (num_regions)
    {
      display_num(bound,0);

  /* Allocate storage for column. */

      scores[bound] = (long *) calloc(lex->num_nodes, sizeof(long));
      labels[bound] = (long *) calloc(lex->num_nodes, sizeof(long));
      if (!scores[bound] || !labels[bound])
        error("Could not calloc new column");

  /* Matches */
      update_matches (lex, regions, num_regions, bound, scores, labels);

  /* Insertions */

      update_insertions (lex, regions, num_regions, bound, scores, labels);

  /* Deletions */

      update_deletions (lex, scores[bound], labels[bound]);

  /* Endsets */

      update_endsets (lex, bound, scores, labels, setscore, setlabel);

      if (++bound == BOUND_MAX)
        error("compute_matrix: max bound exceeded");

  /* Get More Regions */

      num_regions = read_regions(regions);
    }

/* Backtrace Path */

  find_best_score (lex->word_for_node, &lex->endsets[1],
                    scores, labels, bound-1);
}



init_column (set, scores, labels, num)
     ENDSET *set;
     int **scores, **labels;
     int num;
{
  int i;

  scores[0] = (int *) calloc(num, sizeof(int));
  labels[0] = (int *) calloc(num, sizeof(int));
  if (!scores[0] || !labels[0]) error("init_column: calloc failed");

  for (i=0; i<num; i++) labels[0][i] = -1;
  for (i=0; i<num; i++) scores[0][i] = FLOOR;
  for (i=0; i<set->num_out; i++)
      scores[0][set->nodes_out[i]] = 0;
}



update_matches (lex, regions, num_regions, bound, scores, labels)
     NODELEX *lex;
     REGION *regions;
     int num_regions;
     int bound;
     int **scores, **labels;
{
  INARC *arc = lex->arcs;
  int i, j, k;

/* Iterate over lexical nodes */

  for (i=0; i<lex->num_nodes; i++)
    {
      int best_bound;
      int best_index;
      int best_score = FLOOR;

      int from_bound, from_index, from_score;

  /* Iterate over arriving arcs */

      for (j=0; j<lex->arcs_per_node[i]; j++)
        {
```

```
    /* Iterate over incoming regions */

          for (k=0; k<num_regions; k++)
            {
              REGION *reg = &regions[k];

      /* Get score of previous node */

              from_bound = reg->left_bound;
              from_index = arc->prev_node;
              from_score = scores[from_bound][from_index];

      /* Add score of arc extension */

              if (from_score == FLOOR) continue;
              from_score += reg->scores[arc->phoneme] + arc->lex_score;

      /* Keep the best score so far */

              if (best_score < from_score)
                {
                  best_bound = from_bound;
                  best_index = from_index;
                  best_score = from_score;
                }
            }
          arc++;
        }

   /* Update node with best score and back pointer */

      scores[bound][i] = best_score;
      if (best_score == FLOOR) labels[bound][i] = -1;
      else labels[bound][i] = labels[best_bound][best_index];

    }
}


update_deletions (lex, score, label)
     NODELEX *lex;
     int *score, *label;
{
  int i, j;
  INARC *arc = lex->arcs;
  for (i=0; i<lex->num_nodes; i++)
    for (j=0; j<lex->arcs_per_node[i]; j++)
      {
        int from_score = score[arc->prev_node];

        if (from_score > FLOOR)
          {
            from_score += arc->del_score;

            if (score[i] < from_score)
              {
                score[i] = from_score;
                label[i] = label[arc->prev_node];
              }
          }
        arc++;
      }
}



update_insertions (lex, regions, num_regions, bound, scores, labels)
     NODELEX *lex;
         REGION *regions;
         int num_regions;
     int bound;
     int **scores, **labels;
{
  int i, j;

  for (i=0; i<num_regions; i++)
    {
      REGION *reg = &regions[i];
      int from_bound = reg->left_bound;

      for (j=0; j<lex->num_nodes; j++)
        {
          int from_score = scores[from_bound][j];

          if (from_score == FLOOR) continue;
          from_score += reg->del_score;

          if (scores[bound][j] < from_score)
            {
              scores[bound][j] = from_score;
              labels[bound][j] = labels[from_bound][j];
            }
        }
    }
}




update_endsets (lex, bound, scores, labels, setscore, setlabel)
     NODELEX *lex;
     int bound;
     long **scores, **labels;
     long *setscore, *setlabel;
{
  int i,j,k;
  long *score = scores[bound];
  long *label = labels[bound];
  int packed = (bound<<BOUND_SHIFT) + (node_id<<PNODE_SHIFT);

/* Collect best scores */

  for (i=2; i<lex->num_endsets; i++)
    {
      ENDSET *set = &lex->endsets[i];
      int best_score = FLOOR;
      int best_index = 0;

      for (j=0; j<set->num_in; j++)
        if (best_score < score[set->nodes_in[j]])
          {
            best_score = score[set->nodes_in[j]];
            best_index = set->nodes_in[j];
          }
      setscore[i] = best_score - WTW;
      setlabel[i] = best_index + packed;
    }

  reconcile(&setscore[2], &setlabel[2], lex->num_endsets - 2);

/* Propagate best scores */

  for (i=2; i<lex->num_endsets; i++)
    {
      ENDSET *set = &lex->endsets[i];
      if (setscore[i] < FLOOR) continue;

      for (j=0; j<set->num_out; j++)
        if (setscore[i] > score[set->nodes_out[j]])
          {
                score[set->nodes_out[j]] = setscore[i];
                label[set->nodes_out[j]] = setlabel[i];
            }
        }
}
```

D.4 Utilities (utils.c)

```
#include "numesh.h"

#if REV==0

void display(mesg)
    char *mesg;
{
  int i = 0;
  int j;

  volatile char *Display0 = (char *) 0x805003;

  while (*mesg && i++<4)
    *Display0-- = *mesg++;
  for (j=i; j<4; j++)
    *Display0-- = ' ';
}

#endif


#if REV==1

void display(mesg)
    char *mesg;
{
  int i = 0;
  int j;

  volatile char *Display1 = (char *) 0x804038;

  while (*mesg && i++<8)
    *Display1++ = *mesg++;
  for (j=i; j<8; j++)
    *Display1++ = ' ';
}

#endif
```

```
display_num(val, time)
  int val, time;
{
  char mesg[20];

  ltoa(val,mesg);
  display(mesg);
  Wait(time);
}



Wait(time)
     long time;
{
  volatile int dummy;
  volatile int i;

  for (i=0; i<time; i++)
    dummy = i;
}


Scroll(mesg)
     char *mesg;
{
  char *window;
  int i, len;

  len = strlen(mesg);

  for (i=0; i<len; i++)
    {
      window = &mesg[i];
      display(window);
      Wait(500000);
    }
}
```