A PARALLEL IMPLEMENTATION OF THE A*-VITERBI ALGORITHM FOR SPEECH RECOGNITION

by
M. Ravi Shanker

School of Computer Science
McGill University, Montreal

June 1993

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE

Parallel A*-Viterbi for Speech Recognition

Abstract

The problem of speech recognition is one that lends itself to parallelization. A common method used for speech recognition is the Viterbi algorithm. Unfortunately, this method is computationally expensive for large vocabularies. A new two pass method has been proposed, using the Viterbi algorithm as the first pass and the A* algorithm as the second, making use of the results of the Viterbi algorithm. Both these algorithms can be made faster by parallelizing them.

This thesis report describes the design and implementation of a parallel version of these algorithms on a BBN Butterfly multi-processor machine, and it also presents the outcome of the parallelization. It was observed that the parallel version of the Viterbi algorithm ran 8 times faster than the sequential version. This was observed for the recognition of both short words and long words. The A* algorithm, however, displayed different behaviour for short words as compared to long words. With a short word, the parallel version of the A* algorithm ran slightly slower than the sequential version; for a long word, it ran considerably faster than the sequential version - it was observed to run as much as 150 times faster.

This thesis comments on the results thus obtained, and attempts to explain the behaviour of the different parallel implementations.

Résumé

Le problème de la reconnaissance de la parole peut facilement être résolu en parallèlisant. L'algorithme le plus fréquemment utilisé est celui de Viterbi mais, malheureusement, cette méthode prend trop de temps pour un vocabulaire étalé. Pour y remédier, la méthode à "double passe" a été proposé. A son premier passage, l'algorithme Viterbi est utilisé et au deuxième tour, l'algorithme A* utilise les résultats obtenus au passage précédent. Les temps d'exécution des deux processus peut être considérablement réduits s'ils sont parallélisés.

Cette thèse décrit la conception et l'implentation d'une version parallèle de ces algorithmes sur la machine BBN Butterfly tout en présentant les résultats de la parallélisation. Les résultats obtenus ont montré que la version parallèle de l'algorithme Viterbi est 8 fois plus rapide, tant pour les mots longs ou courts, que sa version séquentielle. Par contre, la version parallèle de l'algorithme A* est légèrement plus lente pour les mots courts que sa version séquentielle tandis que pour un mot long, la version parallèle est 150 fois plus rapide.

Le but de cette thèse est d'expliquer les divers comportements des différentes exécutions parallèles.

Acknowledgments

First, and foremost, I wish to thank Prof. Guang R. Gao for having accepted me as his student, and for having given me the opportunity to work in this very interesting field of parallel programming. I would also like to thank IRIS-PRECARN for funding this research topic.

I am very grateful to Dr. Patrick Kenny, Dr. Matthew Lennig, and INRS Telecommunications, Montreal, for allowing me to use their software in my thesis.

My love and affection go to my parents, S.G.V. Mani and Lahtha, and to my sister, Indu, for all they have given me.

I would also like to extend my love, and my gratitude to my uncle, S. G. Lokanathan, my aunt, Kausalya, and my cousins, Ramesh, Vanita and Raghu for making me feel so much at home in Montreal.

I would like to thank the Systems Staff – Luc Boulianne, Bill Heelan and Matthew Sams for giving me free access to the labs, and for fixing all the problems that I faced. Special thanks are due to Kent Tse for helping me with 'biernat'.

My thanks go to the staff at the School of Computer Science – Lorraine, Lise, Franca, Vicki, Vera and Mirriam for all the help that they extended to me over the three years that I spent in the department.

I have to extend a very special thanks to Holly Avery for keeping me supplied with story books to dispel my blues (of which there were many).

Last, but by no means least, thanks to my colleagues and friends at McGill and elsewhere – Sumithra and François, Govind and Bhama, Sreedhar and Lakshmi, Maryam, Shashank, Chandrika, Martin, Azzedine, Sadeka and all the others, too numerous to be named here, for making life interesting during these three years.

And to all the above, for putting up with my jokes.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the last two decades, attempts have been made to automate the recognition of human speech. The term "speech recognition" is one that covers many different approaches to the problem of recognizing human speech. It ranges from isolated word recognition to continuous speech recognition, from speaker-dependent recognition to speaker-independent recognition, and from a small vocabulary to a large vocabulary. The simplest scenario is a speaker-dependent, isolated word recognition on a small vocabulary and the most complex is a speaker-independent, continuous speech recognition on a large vocabulary. In any case, the speech recognition problem, as developed over the years, is a highly computation-intensive problem; it requires fast processors, and large amounts of memory. Many attempts have, therefore, been made to try and speed up the process using various techniques. In this thesis, we attempt to run a speech recognition algorithm in parallel on a multiprocessor machine, and we present the results of the parallelization of the algorithm.

1.1 Speech Recognition

In the recent past, many speech recognition strategies have been proposed and implemented [BJM83, LRS83]. These strategies span many sciences, including signal processing, pattern recognition, artificial intelligence, statistics, information theory, probability theory, computer algorithms, psychology, linguistics, and biology [Lee89]. Of these strategies, the probability theory method using hidden Markov models (HMMs)

is most widely used. An HMM is a parametric model that is particularly suitable for describing speech events. HMMs (see section 2.1) have two stochastic processes which enable the modeling of acoustic phenomena as well as time scale distortions. Furthermore, efficient algorithms exist for accurate estimation of HMM parameters. finally, HMMs are a succinct representation of speech events and therefore require less storage than many other strategies.

Isolated word recognition using HMMs is usually formulated as one of finding the path in an HMM whose posterior probability (given the acoustic observations) is maximal [KHG+91]. The easiest way of doing this is by means of the Viterbi algorithm (see section 2.2.2). This algorithm is a time synchronous search algorithm that completely processes time t before going on to time $t+1$. For time t, each state of the HMM is updated by the best score from states at time $t-1$. From this, the *most probable state sequence* can be recovered at the end of the search.

A full Viterbi search is quite efficient for moderate tasks; however, for large tasks, it can be very time consuming. Another drawback of the Viterbi algorithm is the fact that it reports only the best recognition hypothesis, whereas, in many cases, we would like to investigate the N-best hypotheses. These drawbacks have been addressed by means of a new approach in which the Viterbi algorithm is coupled with the A* search.

1.1.1 The A* Algorithm

The A* algorithm was developed in an artificial intelligence environment, and it belongs to a class of algorithms known as graph search algorithms (see Chapter 3), which are widely used in AI applications. These are algorithms that find a path through a graph from a start node to a (set of) goal node(s). Graph search strategies may be of two types: *uninformed* and *informed*. Uninformed search strategies include the *depth-first* and *breadth-first* searches. These are exhaustive methods for finding paths to a goal node. For many tasks, it is possible to use task-dependent information to help reduce the search [Nil80]. Information of this sort is usually called *heuristic information*, and search procedures using it are called *heuristic search methods*. The A* algorithm is one such heuristic search method. It makes use of two functions: the *cost function* and the *heuristic function* to aid it in searching the graph for the goal nodes.

1.1.2 The A*-Viterbi Algorithm

The A*-Viterbi algorithm needs a lexical tree G and its corresponding quotient graph G^*, described in section 4.1. The lexical tree contains nodes, which are labeled by phonemes, and arcs (or transitions) between these nodes. Each word in the lexicon may be extracted from the tree by traversing the transitions between nodes, and noting the phonemes that appear along the path from the root of the graph to the leaf. The nodes of the quotient graph have an equivalence relationship with those of the lexical tree, and as a result, the quotient graph is smaller than the lexical tree (see section 4.1).

The algorithm consists of two passes, the first one uses the quotient graph, and the second the lexical tree. It is described in detail in Section 4.3. A brief description of the algorithm follows :

Pass I The Viterbi traversal of the quotient graph G^*. This pass gives us the backward probabilities, or β-values (described in section 2.2.1), and the forward probabilities, or α-values (described in section 2.2.1) for the complete observation sequence $Y_1, \ldots, Y_T$.

Pass II The α- and β-values calculated in the previous pass are used by the A*-algorithm as it searches the lexical tree, looking for a word that best matches the given observation sequence.

1.2 Speeding Up the Algorithm

The ultimate goal of speech recognition is to achieve real time recognition of continuous human speech. In this thesis, we study how the parallelization of the A*-Viterbi algorithm helps in speeding up the process of speech recognition.

In every program, there are some parts that do not depend on the execution of other parts; in other words, there is no *dependency* between these different parts of the program. Such portions of the code which show no dependency may be executed in parallel, allowing the program to run faster as a whole. In the case of the A*-Viterbi algorithm for isolated word recognition, there are large portions of the program that show such a lack of dependency, and which may therefore be run in parallel. Hence,

there is a potential for the program to run significantly faster when it is parallelized than when it is purely sequential.

1.2.1 Parallel Computer Architecture

Under the classification proposed by Flynn (see [Fly66, HP90]), computers may be categorized as :

1. **SISD** – Single Instruction, Single Data stream.
2. **SIMD** – Single Instruction, Multiple Data stream.
3. **MIMD** – Multiple Instruction, Multiple Data stream.

These categories are discussed in greater detail in section 5.1. In this thesis, we are interested in parallel processing, since we wish to design and implement a parallel version of the A*-Viterbi algorithm. All parallel processing machines belong to the MIMD category because each processor may work on a different portion of the code, and may operate on different data. The results of the various processors have to be collected at some point in time and decisions must be made on the results thus obtained. Such an event, where the various processors cease their work and share their results with other processors, is referred to as *synchronization*.

The two important issues in a parallel processor system are :

- Memory latency – The time taken between the issue of a memory fetch, and return of the value to the processing unit.
- Synchronization – The process by which the various processing units cease to work on the code, and share their results with other processing units

These two issues are tightly coupled in an inverse relation. Reduction of memory latency increases the cost of synchronization, and reduction in the cost of synchronization increases memory latency [AI87].

Parallel processing machines may be of two types : *shared-memory* machines (or *multiprocessors*), and *message-passing* machines (or *multicomputers*). This distinction is created on the basis of a difference in the method of synchronization. In a

multiprocessor, the various processing units communicate with each other by means of setting variables in a common pool of memory referred to as the *shared memory* of the machine. On the other hand, in a multicomputer, the various processing units communicate by sending messages to one another.

The machine that was used for this thesis was the BBN-Butterfly machine. It is a multiprocessor with 32 processing units, in which synchronization between the various processing units is achieved by means of a shared memory.

1.2.2 Parallelization of the Algorithm

Both, the Viterbi and the A* algorithms have the capacity to be parallelized, and made to run faster on a multi-processor machine [KCSK87, KRR88]. In the Viterbi portion of the code, the calculation of the transition scores for the β-values is highly parallelizable, and we should expect to get a speedup proportional to the number of processors working on the calculation. Finding the best path can also be parallelized, but we can expect less speedup here, because of synchronization, and some sequentialization of the code. Calculation of α-values is parallelizable fairly easily, with each processor working on a different phoneme. The A* algorithm is also parallelizable and we could expect linear to super-linear speedup depending on the word to be recognized. There is one caveat in the parallelization of the A* algorithm, however, and that is the parallel A* algorithm could give non-optimal answers. We should take care to discard any erroneous answers from A*.

In the parallelization of the Viterbi portion, we come up against the fact that the BBN Butterfly machine is a distributed shared memory machine. We therefore have to distribute the phonetic graph over all the processors in order to balance the amount of computing done by each processor. We will also have to consider the fact that we must keep the number of remote memory accesses to a minimum if we wish to get good speedup results.

In the parallelization of the A* algorithm, we may follow different strategies :

- **centralized strategy** - good for large granularity problems, simple to implement.
- **distributed strategy** - good for small granularity problems, more complicated than the centralized strategy.

In our case, we use the centralized strategy because of its ease of implementation and because our problem has large granularity.

1.3 Main Results and Interpretation

The results of our parallelization are very informative. The parallelization of the Viterbi portion of the code gave us a speedup of about 8 times with roughly 13 processors. With a larger number of processors, the speedup did not increase appreciably This is due to the following reasons :

- The quotient graph was designed in such a way as to be used efficiently by a sequential machine. This generated constraints on how easily it could be distributed among the various processors of a parallel machine.
- As a result of the design of the quotient graph, there were a large number of remote memory accesses. These could not be reduced without redesigning the quotient graph.

If the graph (and the software) had been designed for optimal use by the parallel machine in question, we could have achieved far greater speedup than what we observed.

The parallel A* code gave us very large speedup results for a large word, but very little speedup for a small word. This is understandable, because for small words, the A* algorithm takes very little time to process, and parallelization in such cases does not help.

The most important conclusion that was drawn from this thesis is as follows : *to get the best possible performance from a parallel machine, the code has to be designed to take full advantage of the capabilities of that machine.* In this case, the results obtained were good, but they could have been much better had the code been designed for the parallel machine starting from the design specifications onward.

1.4 Related Work

Fettweis and Meyr discuss hardware implementations of a parallel Viterbi decoder in [FM89] and [FM91]. The central unit of a Viterbi decoder is a data-dependent

feedback loop which performs an add-compare-select (ACS) operation. This nonlinear recursion is a bottleneck for a high-speed parallel implementation. Their paper presents a solution to implement the Viterbi algorithm by parallel hardware for high data rates.

Kimball and associates discuss the parallel implementation of the Viterbi algorithm for continuous speech recognition in [KCSK87]. The algorithm was developed for the BBN Butterfly machine, and used context dependent HMMs to achieve high recognition accuracy.

K. A. Wen and J. Y Lee discuss the parallel implementation of the Viterbi algorithm in [WL88]. They present a dual-dimensional parallelization for the Viterbi algorithm, i.e., parallelization of the decoding procedures within each stage of the trellis and parallelization of the decoding procedures over consecutive stages.

Y. F. Zhang and P. Csillag discuss a parallel architecture for Viterbi decoding in [ZC89].

Austin, Schwartz and Placeway discuss a new technique to speed up time-synchronous beam searches in [ASP91]. They call it the Forward-Backward Search, and it is mathematically related to the Baum-Welch forward-backward training algorithm [BE67].

Black and Meng discuss a parallel Viterbi decoding scheme in which the required hardware complexity approaches the speedup factor independent of the number of states in [BM90]. Theirs is a block based parallel implementation of the Viterbi algorithm in which concurrent decoding of independent blocks is achieved by using the self synchronizing property of the Viterbi algorithm.

Kumar and Rao discuss the parallelization of the A* algorithm in [RK87, KR87]. Kumar, Rao and Ramesh discuss different parallel formulations of the A* algorithm and the results of these formulations for the Traveling Salesman Problem, the Vertex-Cover Problem, and the 16-puzzle in [KRR88].

1.5 Outline of Thesis

The report is organized as follows. Chapter 2 introduces the hidden Markov model (HMM), and how it is applied to speech recognition. The Viterbi algorithm is explained in chapter 2.2.2; this section will describe the forward and backward Viterbi

algorithms (section 2.2.1), and also the block Viterbi algorithm (section 1.2). Chapter 3 discusses some aspects of artificial intelligence related with the A* algorithm. Chapter 4 discusses the A*-Viterbi algorithm. Chapter 5 will describe the BBN Butterfly multiprocessor machine, and how we can use it to parallelize the code. Chapter 6 will describe how the algorithms are parallelized, with section 6.2 describing the parallelization of the backward Viterbi and the block Viterbi algorithms, and section 6.3 describing the parallelization of the A* algorithm. Chapter 7 interprets the results of the parallelization and chapter 8 draws the final conclusions.

Chapter 2

Hidden Markov Models and the Viterbi Algorithm

In the recent past, hidden Markov models (HMMs) have been used extensively in automatic speech recognition. In this chapter, we will first define hidden Markov models and present algorithms for evaluating, and decoding, with HMMs.

2.1 Definition of the Hidden Markov Model

A hidden Markov model is a collection of states connected by transitions. Each transition carries two sets of probabilities : a transition probability, which provides the probability for taking this transition, and an output probability density function (pdf), which defines the conditional probability of emitting each output symbol from a finite alphabet given that a transition is taken [Lee89].

A hidden Markov model is defined by :

- $\{s\}$ A set of states including an initial state S_I and a final state S_F.
- $\{a_{ij}\}$ - A set of transitions where a_{ij} is the probability of taking a transition from state i to state j.
- $\{b_{ij}(k)\}$ -- The output probability matrix : the probability of emitting symbol k when taking a transition from state i to state j.

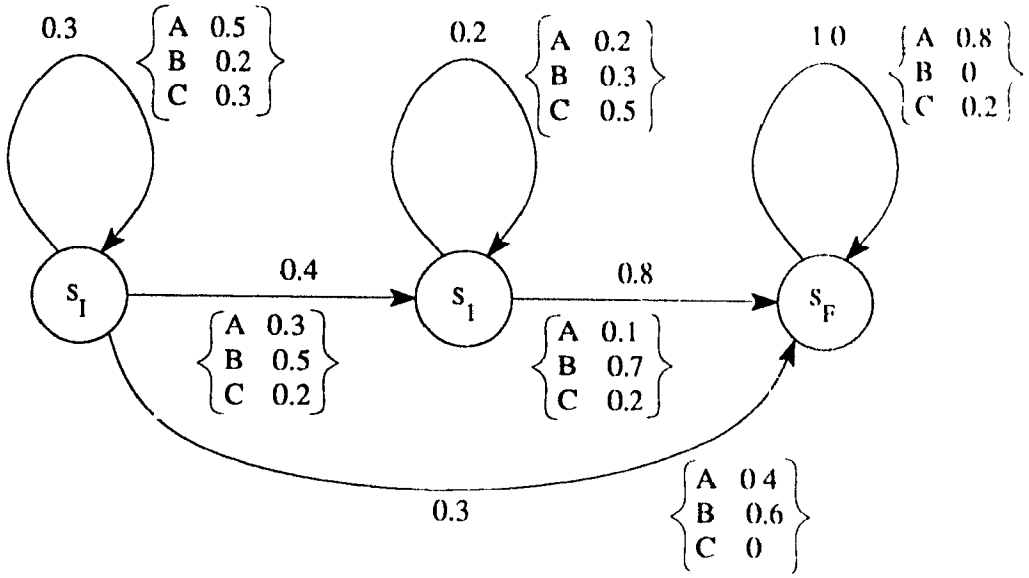

Figure 2.1: The Hidden Markov Model

Figure 2.1 shows an example of a hidden Markov model with three output symbols, A, B and C. A sequence of output symbols is generated by a corresponding transition from one HMM state to another. For instance, to generate the sequence ABCA, the following transitions would have to be performed :$S_I \rightarrow S_I$, $S_I \rightarrow S_F$, $S_F \rightarrow S_F$, $S_F \rightarrow S_F$.

From the definition, since both a and b are probabilistic, they must satisfy the following properties [Lee89] :

$$a_{ij} \geq 0, \quad b_{ij}(k) \geq 0, \quad \forall i, j, k \tag{2.1}$$

$$\sum_j a_{ij} = 1, \quad \forall i \tag{2.2}$$

$$\sum_k b_{ij}(k) = 1, \quad \forall i, j \tag{2.3}$$

a and b can be written as :

$$a_{ij} = P(X_{t+1} = j | X_t = i) \tag{2.4}$$

$$b_{ij}(k) = P(Y_t = k | X_t = i, X_{t+1} = j) \tag{2.5}$$

where $X_t = j$ means the Markov chain was in state j at time t, and $Y_t = k$ means the output symbol at time t was k. We will use the random variable Y to represent the

probabilistic function of a stationary Markov chain X. Both X and Y are generated by a hidden Markov model; however, Y, the output sequence, is directly observed, while X, the state sequence, is *hidden*.

In a first-order HMM, there are two assumptions. The first is the *Markov assumption* :

$$P(X_{t+1} = x_{t+1}|X_1^t = x_1^t) \quad = \quad P(X_{t+1} = x_{t+1}|X_t = x_t) \tag{2.6}$$

where X_i^j represents the state sequence $X_i, X_{i+1}, \ldots, X_j$. Equation 2.6 states that the probability that the Markov chain is in a particular state at time $t+1$ depends only on the state of the Markov chain at time t, and is conditionally independent of the past.

The second assumption is the *output-independence assumption* :

$$P(Y_t = y_t|Y_1^{t-1} = y_1^{t-1}, X_1^{t+1} = x_1^{t+1}) = P(Y_t = y_t|X_t = x_t, X_{t+1} = x_{t+1}) \tag{2.7}$$

where Y_i^j represents the output sequence $Y_i, Y_{i+1}, \ldots, Y_j$. Equation 2.7 states that the probability that a particular symbol will be emitted at time t depends only on the transition taken at that time (from state x_t to state x_{t+1}), and is conditionally independent of the past.

2.2 The HMM Problems

Given the definition of hidden Markov models, there are three problems of interest [Lee89] :

1. **The Evaluation Problem** – Given a model and a sequence of observations, what is the probability that the model generated the observations?

2. **The Decoding Problem** Given a model and a sequence of observations, what is the most likely state sequence in the model that produced the observations?

3. **The Learning Problem** – Given a model and a set of observations, what should the model's parameters be so that it has a high probability of generating the observations?

If we could solve the *evaluation* problem, we would have a way of scoring the match between a model and an observation sequence, which could be used for isolated word recognition.

If we could solve the *decoding* problem, we could find the best matching state sequence given an observation sequence, which could be used for continuous speech recognition.

If we could solve the *learning* problem, we would have the means to automatically learn the parameters given a set of training data.

In this thesis, we are only interested in the first two problems, we will concentrate therefore, on studying the evaluation and decoding problems in this chapter.

2.2.1 The Evaluation Problem : The Forward Algorithm

The evaluation problem can be stated as : given a model, M, with parameters $\{s\}, \{a\}, \{b\}$, compute the probability that it will generate a sequence y_1^l. This involves summing the probabilities of all paths of length T :

$$P(Y_1^T = y_1^T) = \sum_{x_1^{T+1}} P(X_1^{T+1} = x_1^{T+1}) P(Y_1^T = y_1^T | X_1^{T+1} = x_1^{T+1}) \qquad (2.8)$$

In other words, to compute the probability of the sequence y_1^l, we enumerate all paths x_1^{T+1} of length T that generate y_1^T, and sum all their probabilities. The probability of each path x_1^{T+1} is the product of the transition probabilities and the output probabilities of each step in the path.

The first factor (transition probability) in equation 2.8 can be rewritten by applying the Markov assumption :

$$P(X_1^{T+1} = x_1^{T+1}) = \prod_{t=1}^{T} P(X_{t+1} = x_{t+1} | X_t = x_t) \qquad (2.9)$$

The second factor (output probability) in equation 2.8 can be rewritten by applying the output-independence assumption :

$$P(Y_1^T = y_1^T | X_1^{T+1} = x_1^{T+1}) = \prod_{t=1}^{T} P(Y_t = y_t | X_t = x_t, X_{t+1} = x_{t+1}) \qquad (2.10)$$

Substituting eqns. 2.9 and 2.10 into eqn. 2.8, we have :

$$P(Y_1^T = y_1^T) = \sum_{x_1^{T+1}} \prod_{t=1}^{T} P(X_{t+1} = x_{t+1}|X_t = x_t) P(Y_t = y_t|X_t = x_t, X_{t+1} = x_{t+1}) \tag{2.11}$$

We can use the HMM parameters a and b to evaluate eqn. 2.11. This method is exponential since it requires enumeration of all paths with length T.

However, since the probability of each individual quantity depends only on y_t, x_t, and x_{t+1}, it is possible to compute $P(Y_1^T = y_1^T)$ with recursion on t. Therefore, let us define :

$$\alpha_i(t) = \begin{cases} 0 & t = 0 \quad \& \quad i \neq S_I \\ 1 & t = 0 \quad \& \quad i = S_I \\ \sum_j \alpha_j(t-1) a_{ji} b_{ji}(y_t) & t > 0 \end{cases} \tag{2.12}$$

$\alpha_i(t)$ is the probability that the Markov process is in state i having generated y_1^t. Clearly then,

$$P(Y_1^T = y_1^T) \quad = \quad \alpha_{S_F}(T) \tag{2.13}$$

Figure 2.2 illustrates the computation of α as a sweep through a *trellis* using the HMM in Fig. 2.1 and the observation sequence A C B A. Each cell indicates the cumulative probability at a particular state (row) and time (column). An arrow in Fig. 2.2 indicates that a transition from its origin state to its destination state is legal. Consequently, arrows are only allowed between adjacent columns. As eqn. 2.12 indicates, the computation begins by assigning 1.0 to the initial state and 0.0 to all other states at time 0. The other cells are computed *time-synchronously* from left to right. Each column of states for time t is completely computed before going on to time $t+1$, the next column. When the states in the last column have been swept, the final state in the final column contains the probability of generating the observation sequence. In this case, the probability of generating the sequence ACBA is 0.014.

This algorithm is called the *forward algorithm*. It enables us to evaluate the probability that an observation sequence was generated by an HMM, M, or $P(y|M)$. However, in speech recognition, we need to find $P(M|y)$. By Bayes rule, we have

$$P(M|y) \quad = \quad \frac{P(y|M)P(M)}{P(y)} \tag{2.14}$$

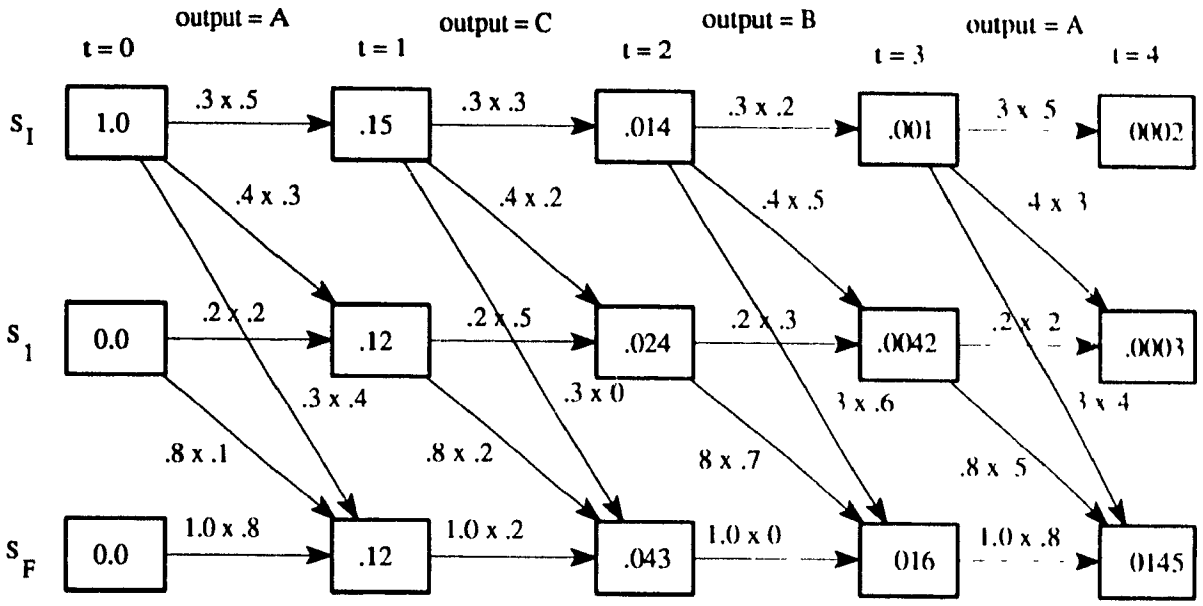

Figure 2.2: A Trellis in the Forward Computation

Since $P(y)$ is constant for a given input, the task of recognizing an observation sequence involves finding the model that maximizes $P(y|M)P(M)$. $P(y|M)$ can be evaluated by the forward algorithm, and $P(M)$ is a probability assigned by the *language model*. In the case of a language model where all words are equally likely (the $P(m)$s are equal for all m), only the first factor need be considered.

The Backward Algorithm

In eqn. 2.12 of the preceding section, we defined $\alpha_i(t)$, or the probability that an HMM **M** has generated y_1^t and is in state i. We could equivalently define its counterpart, $\beta_i(t)$, or the probability that **M** is in state i, and will generate y_{t+1}^T. Like α, β can be recursively computed as follows :

$$\beta_i(t) = \begin{cases} 0 & i \neq S_F \ \& \ t = T \\ 1 & i = S_F \ \& \ t = T \\ \sum_j a_{ij} b_{ij}(y_{t+1}) \beta_j(t+1) & 0 \leq t \leq T \end{cases} \tag{2.15}$$

Figure 2.3 illustrates the computation of βas a sweep through the trellis using the HMM in Fig. 2.1. As we can see, the α values and the β values give the same final

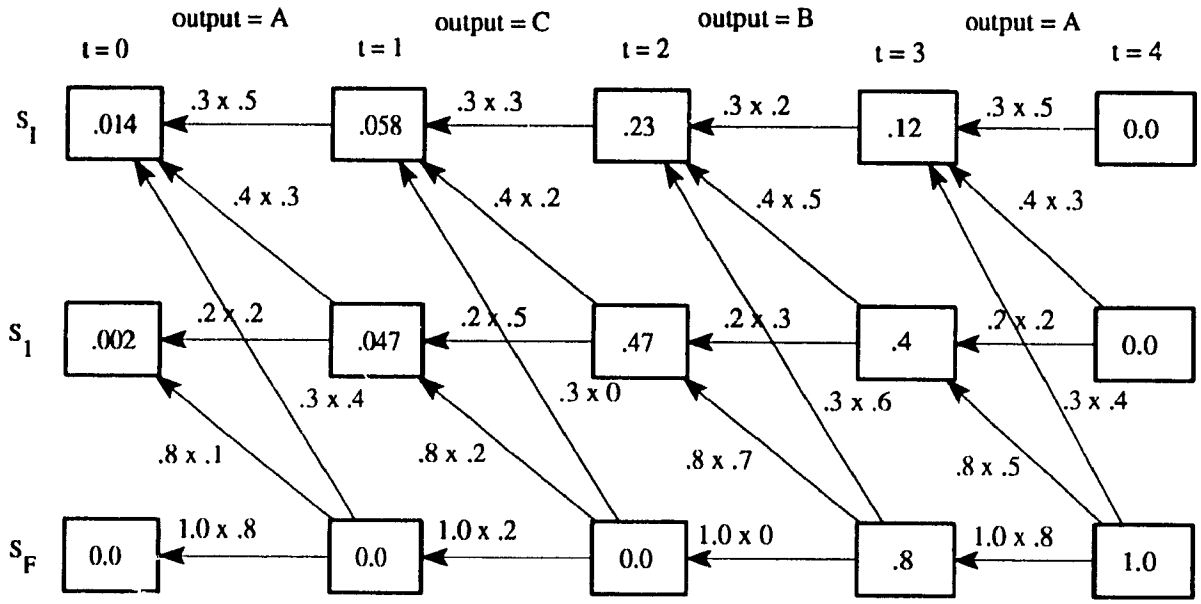

Figure 2.3: A Trellis in the Backward Computation

results for the same observation, i.e., the probability of generating the observation sequence ACBA is 0.014.

2.2.2 The Decoding Problem : The Viterbi Algorithm

While the forward algorithm computes the probability that an HMM generated an observation sequence, it does not provide a state sequence. In many applications, it may be desirable to have such a sequence.

Unfortunately, by definition, the state sequence is *hidden* in an HMM. The best we can do is produce the *state sequence that has the highest probability of being taken while generating the observation sequence*. To do that, we need only modify the forward pass slightly. In the forward pass, we summed the probabilities that came together. Now, we need to choose and remember the maximum. Eqn. 2.12 can therefore be rewritten as :

$$\alpha_i(t) = \begin{cases} 0 & t = 0 \ \& \ i \neq S_I \\ 1 & t = 0 \ \& \ i = S_I \\ \max_j \alpha_j(t-1) a_{j,i} b_{j,i}(y_t) & t > 0 \end{cases} \tag{2.16}$$

To uncover the most likely state sequence, we must remember the best path to each cell, which is the concatenation of the best path to its predecessor state and the best step to the cell.

This algorithm is known as the *Viterbi algorithm* [For73, Vit67, LRS83, Lee89]. To be more specific, we could refer to this as the *forward Viterbi algorithm*. If $\mathbf{M}$ is a hidden Markov model, then the Viterbi *score* of $\mathbf{M}$, given a sequence of observations $y_1, \ldots, y_T$ is defined as follows : $V(y_1, \ldots, y_T|M) = \max_s \alpha_T(s)$ where s ranges over all the sink states.

We have, equivalently, a *backward Viterbi algorithm* which we obtain by modifying eqn. 2.15. The backward Viterbi algorithm may be defined as :

$$\beta_i(t) = \begin{cases} 0 & t = T \ \& \ i \neq S_F \\ 1 & t = T \ \& \ i = S_F \\ \max_j a_{ij} b_{ij}(y_{t+1}) \beta_j(t+1) & 0 \leq t < T \end{cases} \tag{2.17}$$

The Viterbi *score* is this case is given by : $V(y_1, \ldots, y_T|M) = \max_s \beta_0(s)$ where s ranges over all the start states.

2.3 Using HMMs for Speech Recognition

In this section, we will examine how HMMs can be used for speech recognition. We will discuss how to construct models to represent units of speech, and how to recognize speech with HMMs [Lee89].

Hidden Markov models are a natural representation of speech. The output distribution models the parametric distribution of speech events, and the transition distribution models the duration of these events. HMMs can be used to represent any unit of speech.

The most natural unit of speech is the word. But, while words are what we want to recognize, they are not a practical choice for large-vocabulary recognition because the amount of training and storage is enormous. Instead, some subword unit should be used. The subword unit of speech that is most commonly used is the *phoneme*. A phoneme is the fundamental unit of speech. It is the basis on which all sounds are made. There are approximately 40 distinct sounds in the English language, and

hence, 40 different phonemes. All the words in the English language are made up of varying combinations of these 40 phonemes. Each word could then be represented as a network of phonemes which encodes every way the word could be pronounced. We could then instantiate each instance of a phoneme with its hidden Markov model. Then we have a large HMM that encodes all the legal words.

By placing all the knowledge in the data structures of the HMMs, it is possible to perform a global search that takes all the knowledge into account at every step.

Using such a network (or graph) or HMMs, it is easy to implement both isolated word and continuous speech recognition. For isolated word recognition, we could use the forward pass to score the input word against each of the models. Assuming no language model, the model with the highest probability is chosen as the recognized word. We could also use the Viterbi algorithm for recognition. If subword units are used, then they would be concatenated into words first.

2.4 Viterbi-based Searches

The Viterbi search [Vit67, For73] has been discussed as a solution to one of the three HMM problems in section 2.2.2. To briefly reiterate, the Viterbi search is a time synchronous search algorithm that completely processes time t before going on to time $t+1$. For time t, each state is updated by the best score from state at time $t-1$. From this, the *most probable state sequence* can be recovered at the end of the search [Lee89].

A full Viterbi search is quite efficient for moderate tasks. However, for large tasks, it can be very time consuming. In our case, we would like to perform isolated word recognition on a large vocabulary. This would entail an enormous amount of computation if we were to use the Viterbi search. Therefore, we make use of a new algorithm – the A*-Viterbi algorithm (see Chapter 4) – which allows us to speed up the computation quite considerably [Ken90, KHG+91]. To understand the A*-Viterbi algorithm, we will first need to study the A* algorithm, which we will discuss in the following chapter.

Chapter 3

Artificial Intelligence and the A* Algorithm

The A* algorithm is one of the tools developed to search graphs in the artificial intelligence (AI) environment. Most AI systems display a fairly rigid separation between the standard computational components of *data*, *operations*, and *control*[Nil80]. Hence, the major elements of an AI *production system* are a *global database*, a set of *production rules*, and a *control system*.

The global database is the central data structure used by an AI production system. This database varies depending on the application; in our case (speech recognition), it is a tree structure.

The production rules operate on the global database. Each rule has a *precondition* that is either satisfied or not by the global database. If the precondition is satisfied, the rule can be *applied*. Application of the rule changes the database.

The control system chooses which applicable rule should be applied and ceases computation when a *termination condition* on the global database is satisfied.

Let us consider a simple example of an AI production system — the 8-puzzle problem.

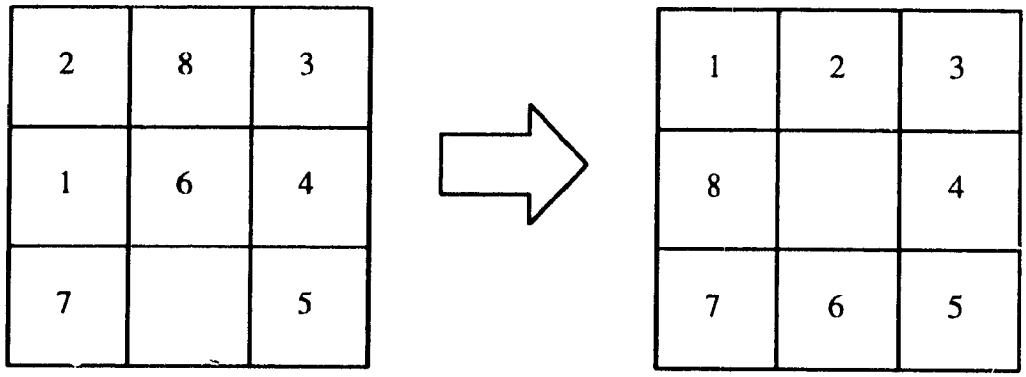

Figure 3.1. Initial and goal configurations for the 8-puzzle

3.1 The 8-Puzzle

The *8-puzzle* is a simple example of the AI production system, and it will enable us to understand an AI problem better. The 8-puzzle consists of eight numbered, movable tiles set in a 3×3 frame [Nil80]. One cell of the frame is always empty, thus making it possible to move an adjacent numbered tile into the empty cell. Two configurations of tiles are given in Fig. 3.1. The problem is to change the initial configuration into the goal configuration. The solution to the problem would be an appropriate sequence of moves, such as "move tile 2 to the right, move tile 4 down, ..., etc.".

To solve a problem using a production system, we must specify the global database, the rules, and the control strategy. For the 8-puzzle, we can easily identify elements of the problem that correspond to these three components. These elements are the problem *states*, the *moves*, and the *goal state*. In the 8-puzzle, each tile configuration is a problem state. The set of all possible configurations is the *space* of problem states, or the *problem space*. These may be represented as a 3×3 array or matrix of numbers.

A move transforms one problem state into another state. In the 8-puzzle, we have the following four moves : move the empty space (blank) up, move the blank down, move the blank to the right, and move the blank to the left. These moves are modeled by production rules that operate on the state descriptions in the appropriate manner. Each rule has preconditions that must be satisfied by a state description in order for

them to be applicable. Thus, the precondition for the rule "move blank to the left" is that the blank must not already be at the leftmost column of the matrix

In the 8-puzzle, we want to reach the goal configuration from a given initial configuration. The problem goal condition forms the basis for the termination condition of the production system. The control strategy repeatedly applies rules to state descriptions until a description of a goal state is produced. It also keeps track of the rules that have been applied so that it can compose them into the sequence representing the problem solution.

3.2 Control Strategies

Selecting rules to be applied on the problem state, and keeping track of those sequences of rules already tried and the databases they produced constitute what we call the *control strategy* for production systems[Nil80]. The operation of AI production systems can most accurately be characterized as a *search process* in which rules are tried until some sequence of them is found that produces a database satisfying the termination condition. Efficient control strategies require enough knowledge about the problem being solved so that the rules selected by the control strategy have a good chance of being the most appropriate under the circumstances.

There are two major kinds of control strategies :

- *Irrevocable* : In an irrevocable control regime, an applicable rule is selected and applied without possibility of a later change.
- *Tentative* : In the tentative control regime, an applicable rule is selected (arbitrarily or otherwise), the rule is applied, but provision is made to return later to this point in the computation to apply some other rule

  Tentative control regimes are again distinguished into :

  - **Backtracking** : In this method, a *backtracking point* is established when a rule is selected. If the subsequent computation encounters difficulty in producing a solution, the state of the computation reverts to the previous backtracking point, where another rule is applied instead, and the process continues.

- **Graph-search** : In this method, provision is made to keep track of the effects of several sequences of rules simultaneously. Various kinds of graph structures and graph searching procedures are used in this type of control. The A* algorithm belongs to this class of control strategies.

3.2.1 Graph-Search Strategies

We can think of a graph-search control strategy as a means of finding a path in a graph from a node representing the initial database to one representing a database that satisfies the termination condition of the production system[Nil80]. Before discussing such strategies, let us first review some graph-theory terminology.

A *graph* consists of a set of *nodes* (not necessarily finite). Certain pairs of nodes are connected by *arcs*, and these arcs are *directed* from one member of the pair to the other. Such a graph is called a *directed graph*. For our purposes, the nodes are labeled by databases, and the arcs are labeled by rules. If an arc is directed from node n_i to node n_j, then the node n_j is said to be a *successor* of node n_i, and node n_i is said to be a *parent* of node n_j. In our case (for the speech recognition problem), a node can have only a finite number of successors.

A *tree* is a special case of a graph in which each node has at most one parent. A node in the tree having no parent is called a *root node*. A node in the tree having no successors is called a *tip node*. We say that the root node is of *depth* zero. The depth of any other node in the tree is defined to be the depth of its parent plus 1.

A sequence of nodes $(n_{i_1}, n_{i_2}, \ldots, n_{i_k})$, with each n_{i_j} a successor $n_{i_{j-1}}$ for $j = 2, \ldots, k$ is called a *path* from node n_{i_1} to n_{i_k} with *length* k. If a path exists from node n_i to node n_j, then node n_j is said to be *accessible* from node n_i. Node n_j is then a *descendent* of node n_i, and node n_i is an *ancestor* of node n_j. The problem of finding a sequence of rules transforming one database into another is thus equivalent to the problem of finding a path in a graph.

Often it is convenient to assign positive *costs* to arcs, to represent the cost of applying the corresponding rule. It is assumed that these costs are all greater than some arbitrarily small positive number, ϵ. The cost of a path between two nodes is then the sum of the costs of all the arcs connecting the nodes on the path. In the speech recognition problem, we will want to find that path having *minimal* cost between two nodes.

In the simplest type of problem, we desire to find a path (perhaps having minimal cost) between a given node s, representing the initial database and another given node t, representing some other database. The more usual situation, however, involves finding a path between a node s and *any* member of a set of nodes $\{t_i\}$ that represent databases satisfying the termination condition. We call the set $\{t_i\}$ the *goal set*, and each node t in $\{t_i\}$ is a *goal node*.

In our application, the control strategy generates part of an implicitly specified graph. This implicit specification is given by the *start* node s, representing the initial database, and the rules that alter databases. It is convenient, at this point, to introduce the notion of a *successor operator* that is applied to a node to give *all* of the successors of that node (and the costs of the associated arcs). We call this process of applying the successor operator to a node, *expanding* the node.

3.2.2 A General Graph-Searching Procedure

The process of explicitly generating part of an implicitly defined graph can be defined as follows :

Procedure **GRAPHSEARCH**

1. Create a *search graph*, G, consisting solely of the start node, s. Put s on a list called **Open**.
2. Create a list called **Closed** that is initially empty.
3. *LOOP* : if **Open** is empty, exit with failure.
4. Select the first node on **Open**, remove it from **Open**, and put it in **Closed**. Call this node n.
5. If n is a goal node, exit successfully with the solution obtained by tracing a path along the pointers from n to s in G. (Pointers are established in step 7).
6. Expand node n, generating the set, M, of its successors that are not ancestors of n. Install these members of M as successors of n in G.

7. Establish a pointer to n from those members of M that were not already in G (ie., not already in **Open** or **Closed** add these members of M to **Open**.(See text)
   For each member of M that was already on **Open** or **Closed**, decide whether or not to redirect its pointer to n(See text).
   For each member of M already in **Closed**, decide for each of its descendants in G whether or not to redirect its pointer.

8. Reorder the list in **Open**, either according to some arbitrary scheme, or according to heuristic merit.

9. Goto LOOP.

This procedure is sufficiently general to encompass a wide variety of special graph-searching algorithms. The procedure generates an explicit graph, G, called the *search graph* and a subset, T, of G, called the *search tree*. Each node in G is also in T. The search tree is defined by the pointers that are set up in step 7. Each node (except s) in G has a pointer directed to just one of its parents in G, which defines its unique parent in T. The search graph forms a partial ordering because no node in G is one of its own ancestors (step 6). Each possible path to a node discovered by the node is defined by T. Roughly speaking, the nodes on **Open** are the tip nodes of the search tree, and the nodes on **Closed** are the nontip nodes. More precisely, at step 3 of the procedure, the nodes on **Open** are those (tip) nodes of the search tree that have not yet been selected for expansion. The nodes on **Closed** are either tip nodes selected for expansion that generated no successors in the search graph or nontip nodes of the search tree.

The procedure orders the nodes on **Open** in step 8 so that the "best" of these is selected for expansion in step 4. Whenever the node selected for expansion is a goal node, the process terminates successfully. The successful path from start node to goal node can then be recovered (in reverse) by tracing the pointers back from the goal node to s. The process terminates unsuccessfully whenever the search tree has no remaining tip nodes that have not yet been selected for expansion. In the case of unsuccessful termination, the goal node(s) must have been inaccessible from the start node.

Step 7 requires some additional explanation[Nil80]. If the implicit graph being searched was a tree, we could be sure that none of the successors generated in step 6

had been generated previously. Every node (except the root node) of a tree is the successor of only one node and thus is generated once only when its unique parent is expanded. Thus, in this special case, the members of M in steps 6 and 7 are not already on either **Open** or **Closed**. In this case, each member of M is added to **Open** and is installed in the search tree as a successor of n. The search graph is the search tree throughout the execution of the algorithm, and there is no need to change parents of nodes in T.

If the implicit graph being searched is not a tree, it is possible that some of the members of M have already been generated, i.e., they may already be in **Open** or **Closed**. The problem of determining whether a newly generated database is identical to one generated before can be computationally expensive. For this reason, some search processes avoid making this test, with the result that the search tree may contain several redundant successor computations. Hence, there is a tradeoff between the computational cost of generating a larger search tree (containing multiple nodes labeled by identical databases). In steps 6 and 7 of procedure **GRAPHSEARCH**, we are assuming that it is worthwhile to test for identical nodes.

Procedure **GRAPHSEARCH** may be "informed" or "uninformed", depending on Step 8. If the reordering is done randomly, then the graph search is "uninformed". If it is done according to some heuristic function, then the search is "informed". The A* is one such "informed" search, which makes use of an heuristic function to expand the nodes.

3.3 Heuristic Graph-Search Procedures

The uninformed search methods are exhaustive methods for finding paths to a goal node[Nil80]. In principle, these methods provide a solution to the path-finding problem, but they are often infeasible to use to control AI production systems because the search expands too many nodes before a path is found.

For many tasks, it is possible to use task-dependent information to help reduce search. Information of this sort is usually called *heuristic information*, and search procedures using it are called *heuristic search methods*. It is often possible to specify heuristics that reduce search effort without sacrificing the guarantee of finding a minimal length path.

Heuristic information can be used to order the nodes on **Open** in step 8 of **GRAPHSEARCH** so that the search expands along those sectors of the graph that are considered most promising. In order to apply such an ordering procedure, we need a method for computing the "promise" of a node. One such method makes use of an *evaluation* function over the nodes. Suppose we denote the evaluation function by the symbol f. Then $f(n)$ gives the value of the function at node n.

We use the function f to order the nodes on **Open** in step 8 of **GRAPHSEARCH** By convention, the nodes on **Open** are ordered in increasing order of their f values. Ties among f values are ordered arbitrarily, but always in favour of goal nodes.

The way in which **GRAPHSEARCH** uses an evaluation function to order nodes can be illustrated by considering again our 8-puzzle example. We use the simple evaluation function :

$$f(n) = d(n) + W(n)$$

where $d(n)$ is the depth of node n in the search tree and $W(n)$ counts the number of misplaced tiles in that database associated with node n. Thus the start node configuration

2	8	3
1	6	4
7		5

has an f value equal to $0 + 4 = 4$.

The results of applying **GRAPHSEARCH** to the 8-puzzle using this evaluation function are shown in Fig. 3.2. The value of f for each node is circled; the uncircled numbers show the order in which the nodes are expanded.

The choice of evaluation function critically determines search results. The use of an evaluation function that fails to recognize the true promise of some nodes can result in nonminimal cost paths; whereas, the use of an evaluation function that overestimates the promise of all nodes results in expansion of too many nodes.

In general, we can define the evaluation function f so that its value, $f(n)$, at any node n is given by

$$f(n) = g(n) + h(n),$$

where

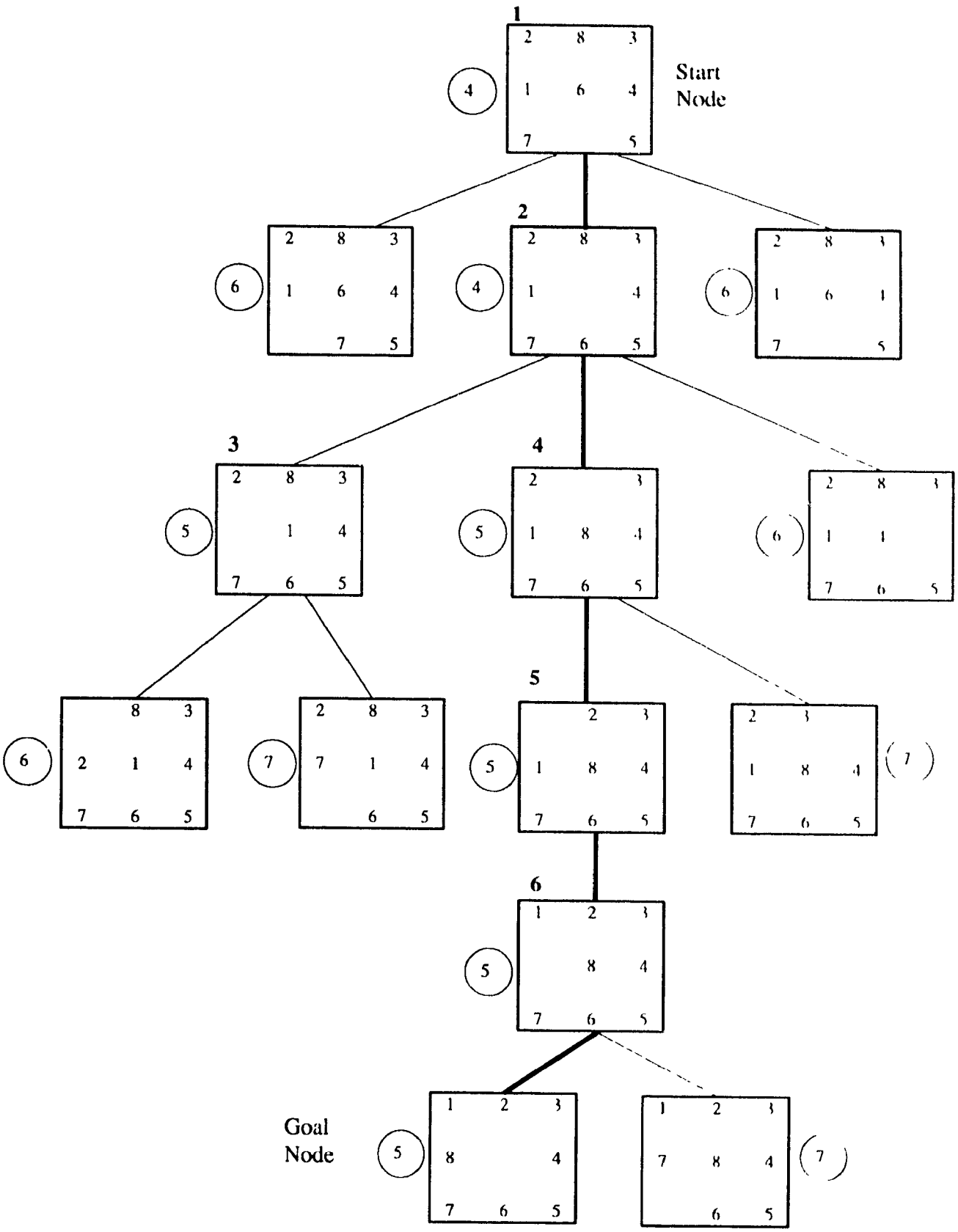

Figure 3.2: A search tree using an evaluation function

$g(n)$ = the cost of the minimal cost path from the start node s to node n, and

$h(n)$ = the heuristic estimate of the cost of a minimal cost path from node n to a goal node

That is, $f(n)$ is an estimate of the cost of a minimal cost path *constrained to go through node* n. That node on **Open** having the smallest f value is then the node estimated to impose the least severe constraint; hence it is appropriate that it be expanded next.

The **GRAPHSEARCH** algorithm using this evaluation function for ordering nodes is called the **A* algorithm**.

Chapter 4

The A*-Viterbi Algorithm

Isolated word recognition using HMMs is formulated as one of finding the path in an HMM whose posterior probability (given the acoustic observations) is maximal [Ken90, KHG+91]. The easiest way of doing it is by means of the Viterbi algorithm [For73, Vit67, Lee89, LRS83]. To recapitulate, the Viterbi algorithm is a time synchronous search algorithm that completely processes time t before going on to time $t+1$.

A full Viterbi search is quite efficient for moderate tasks; however, for large tasks, it an be very time consuming. To speed up the process for large vocabulary isolated word recognition, a new two-pass algorithm, called the A*-Viterbi algorithm was developed.

The A*-Viterbi algorithm needs a phonetic graph, G, and its quotient graph, G^*, in the following section.

4.1 Phonetic Graph and Quotient Graph

A phonetic graph G [KHG+91] consists of a list of nodes and a list of branches (see Fig. 4.1). Each branch consists of two nodes and the arc that connects them. Consider the nodes n_1 and n_3, and the arc between them in Fig. 4.1. The triple (n_1, f_1, n_3) is a branch of the phonetic graph, G. n_1 is referred to as the *starting node*, and n_3 as the *target node*. f_1 is the phoneme that allows the transition from node n_1 to

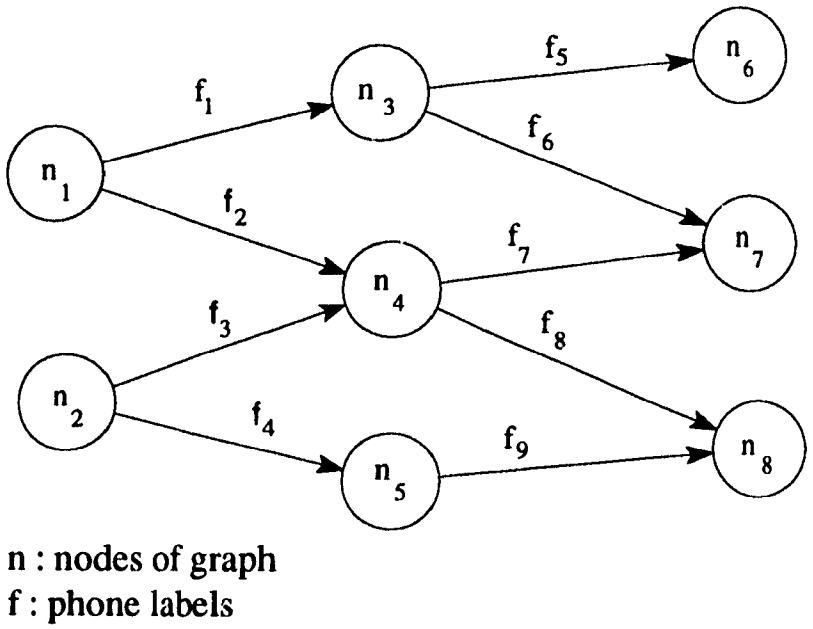

Figure 4.1: The Phonetic Graph

n_3. Similarly, (n_2, f_3, n_4) and (n_3, f_5, n_6) are also branches of the phonetic graph in Fig. 4.1. Each phone label is modeled by an HMM as discussed in Chapter 2.

n_1 and n_2 are referred to as the *initial nodes* of the phonetic graph because they are not the target nodes of any branch. n_6, n_7 and n_8 are referred to as the *final nodes* of the graph since they are not the starting nodes of any branch. A *path* in G can then be defined as a sequence of branches $b_1, \ldots, b_K$, having the property that the target node for each branch is the starting node for the succeeding branch. The path is said to be *complete* if the starting node of b_1 is an initial node of G, and the target node of b_K is a final node of G.

If we are given a phonetic graph G, and a lexicon L, we say that G generates L if every complete path in G corresponds to a transcription in L, and every transcription in L corresponds to a complete path in G. When the lexicon is large, the corresponding phonetic graph is also large. Running the Viterbi algorithm on such a large phonetic graph is very inefficient, and would take too long to run. We therefore create a smaller graph, G^*, called the *quotient* of G.

Given a phonetic graph G and an equivalence relation $\approx$ on the nodes of G, we can construct a new graph G^*, called the *quotient* of G by $\approx$. The nodes of G^* are equivalence classes of the nodes of G. A triple (n_1^*, f, n_2^*) is a branch in G^* if there

is a branch (n_1, f, n_2) in G such that n_1^* is the equivalence class of n_1 and n_2^* is the equivalence class of n_2. Note that for every path in G there is a corresponding path in G^* whose branches carry the same phone labels. In particular, if G generates a lexicon L, every complete transcription in L corresponds to a complete path in G^*. Since the nodes in G^* are equivalence classes of the nodes in G, G^* has fewer branches than G and hence is smaller than G. The Viterbi algorithm can be run on G^*, and the α and β values can be calculated

4.2 The Block Viterbi Algorithm

In this section we discuss how to calculate the forward probabilities (or α values) using the forward block Viterbi algorithm [KHG+91] Consider a branch (n_1, f, n_2) of the phonetic graph G. The algorithm enters node n_1 at time t, and enters node n_2 at time t'. In the period $t \rightarrow t'$, the algorithm traverses the HMM that models the phone f which labels the transition from n_1 to n_2, the algorithm, therefore, is at the start state of the HMM at time $t+1$, and leaves the sink state of the HMM at time t'. We would like to calculate the forward probability (or α value) for node n_2 at time t', given the α value of node n_1 at time t. This can be done in the following manner :

$$\alpha_t(n_2) = \max_{t<t'} \max_{f} \max_{n_1} \alpha_t(n_1) V(y_{t+1}, \ldots, y_{t'}|f) \tag{4.1}$$

where $V(y_{t+1}, \ldots, y_{t'}|f)$ is the Viterbi score for phoneme f,

f ranges over all phones, and

for each f, n_1 ranges over all nodes such that (n_1, f, n_2) is a branch in G. The paper by Kenny et al [KHG+91] gives a more detailed description of the block Viterbi algorithm, and the A*-Viterbi algorithm in general.

4.3 The A*-Viterbi Algorithm

The algorithm has two passes :

Pass I The Viterbi pass : This pass is done on the quotient graph. This computation has two distinct phases :

Phase I In this phase, the backward probabilities, or β-values (see page 16) are calculated for the entire observation sequence $Y_1, \ldots, Y_T$. Consider eqn. 2.17 for definition of the β-value calculation :

$$\beta_t(i) = \begin{cases} 0 & t = T \ \& \ i \neq S_F \\ 1 & t = T \ \& \ i = S_F \\ \max_j a_{ij} b_{ij}(y_{t+1}) \beta_j(t+1) & 0 \leq t < T \end{cases} \quad (4.2)$$

This is split into two parts :

1. In the first part, we compute the *transition scores*, $a_{ij} \times b_{ij}(y_t)$, for all values of i, j and t, given the observation sequence $y_1, \ldots, y_T$. These scor re then stored in a large enough data structure for future use.
2. We find the maximum of the scores calculated in the previous step, subject to $\beta_j(t+1)$ for $0 \leq t < T$.

Phase II The block Viterbi algorithm (discussed in section 4.2 is then used to calculate the forward probabilities, or α-values.

Pass II The A* pass : This is then done on the phonetic graph corresponding to the lexical tree L, using the α and β values calculated in Pass I as the cost function and the heuristic function respectively, in order to recognize the word. The A* search of the lexicon proceeds as follows. At each iteration of the algorithm, there is a sorted list (or *stack*) of partial transcriptions, each with a heuristic score. The forward and backward probabilities calculated in the previous pass are associated with the stack entries as described below.

Let T be the lexical tree corresponding to L that is generated by the A* algorithm. Each stack entry can be thought of as a node in T. The *score* of a partial transcription $\mathbf{f}$ is defined as : $h(\mathbf{f}) = \max_{0 \leq t \leq T} \alpha_t(\mathbf{f}) \beta_t(\mathbf{f})$ The partial transcription with the highest heuristic score is expanded, meaning that for each of the one phone extensions permitted by the lexicon, the heuristic score of the extended transcription is calculated and the extended transcription is inserted into the stack at the appropriate position. The algorithm terminates when a complete transcription appears at the top of the stack [KHG+91].

This algorithm can be speeded up further by parallelizing each pass. We have implemented this parallelization on the BBN Butterfly multiprocessor machine (which

is discussed in chapter 5), and we discuss the parallelization of the algorithm in chapter 6.

Chapter 5

The BBN Butterfly Machine

The aim of this thesis is to implement the A*-Viterbi algorithm discussed in Chapter 4 on a multiprocessor machine and to study the speedup results that we can obtain on it. To that end, we have implemented a parallel version of the A*-Viterbi algorithm on the BBN Butterfly multiprocessor machine.

Multiprocessor machines belong to the growing number of computers that come under the category of *high-performance computers*. These computers include high-speed vector computers such as the CRAY, dataflow computers, superscalar machines such as the R6000, massively parallel machines such as the Connection Machine, etc.

5.1 Computer Architecture

In 1966, Flynn proposed a simple model of categorizing all computer systems [Fly66, HP90]. This categorization looks at the parallelism in the instruction and data streams, and placed all computers in one of four categories :

1. **SISD** Single Instruction, Single Data stream. Examples of these machines are abundant the DEC VAX, the IBM 360, the Intel 8086 microprocessor, etc. This category contains the uniprocessor systems made to-date.

2. **SIMD** - Single Instruction, Multiple Data stream. Under this scenario, n data streams would simultaneously go to n processing units to do n operations

of the same type (such as addition). Vector computers such as the CRAY machines could be classified under this heading, but in their case, there is only one processing unit which performs the operation on n data streams. Unlike vector machines, massively parallel SIMD machines rely on interconnection or communication networks to exchange data between processing units. Examples of SIMD machines would be the Connection Machine 2, and the ILLIAC IV.

3. **MIMD** – Multiple Instruction, Multiple Data stream. In this category, we have a network of processing elements working on different data at the same time in order to speed up the overall processing time. We will discuss this category in greater detail in the following section because the BBN Butterfly machine belongs to this category of computers.

5.1.1 MIMD Architectures

The greatest goal of computer architecture is to compose a powerful computer by connecting many existing smaller ones. To understand the MIMD philosophy, let us get familiar with a few standard terms that are used with MIMD machines [HP90].

MIMD machines may be broadly divided into:

- *shared memory* machines, or *multiprocessor* machines, such as the BBN Butterfly and the Sequent Balance, and
- *message passing* machines, or *multicomputers*, such as the Hypercube and Transputer based machines.

In a *multiprocessor* machine, the machine offers the programmer a single memory address that all processors can access. Processes communicate through shared variables in memory, with any processor being able to write to any portion of the shared memory. In a *multicomputer*, processes communicate by sending messages to one another [HP90]. We shall limit our discussion to multiprocessors since the machine on which this thesis was implemented – the BBN Butterfly – was a multiprocessor.

Two common architectures for multiprocessor machines can be seen in Fig. 5.1. In Fig. 5.1(a), all the memories are at the same distance from any processor – it would take the same amount of time for a processor to access any of the memory units;

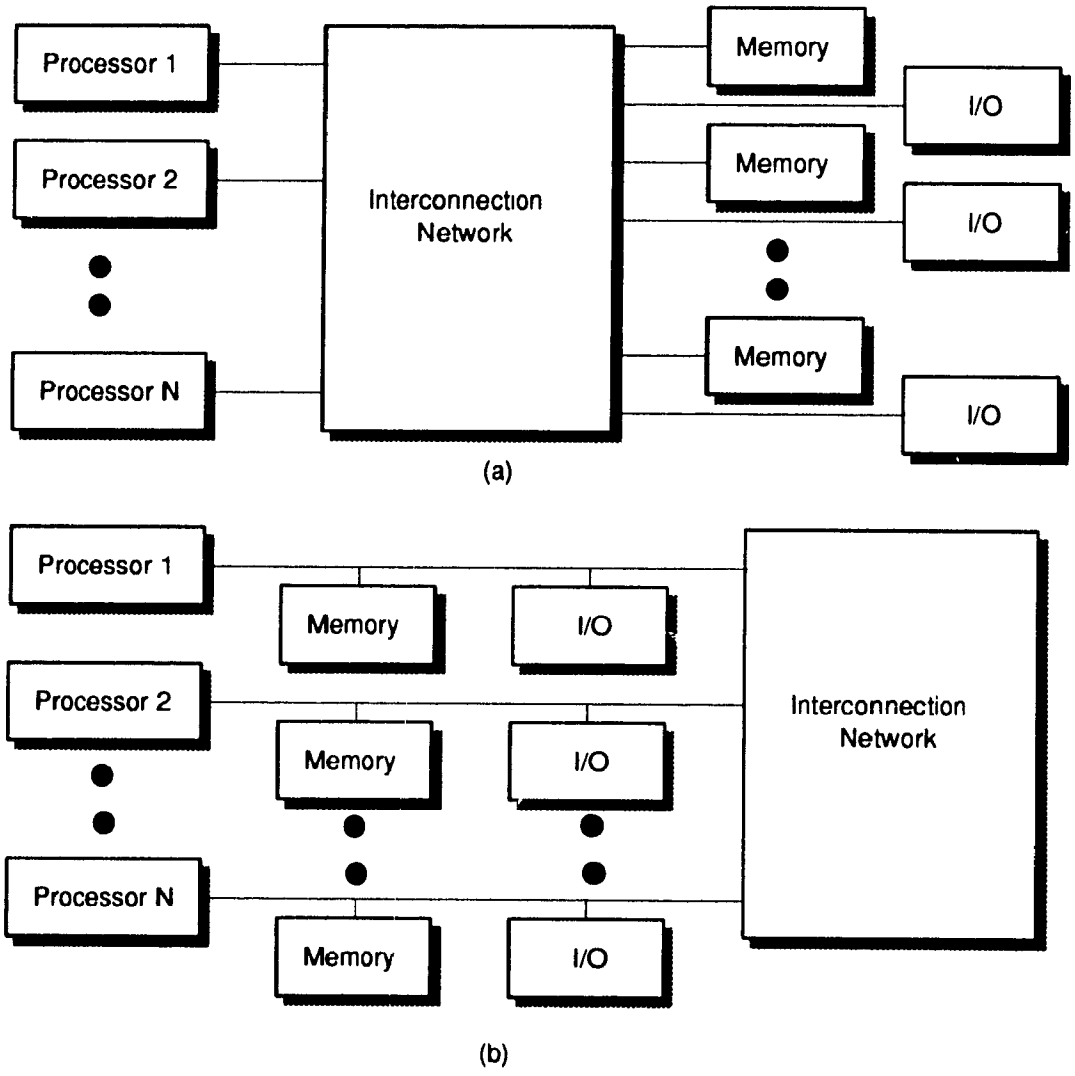

Figure 5.1: Architectures for Multiprocessor Machines

this organization of memory is known as a *centralized* shared memory. On the other hand, Fig. 5.1(b) is an example of a *distributed* shared memory organization. In such an organization, each processor has some memory which is physically close to it, and other memory which is physically far away. The memory which is close to a processor is referred to as the *local* memory, and the memory which is farther away is referred to as the *remote* memory. A processor can access local memory much faster than it can access remote memory. The BBN Butterfly is an example of one such distributed shared memory multiprocessor. In this machine, it takes approximately 5 to 8 times longer to access remote memory as compared to local memory.

The interconnection network that connects the various processor and memory elements is a very important part of an MIMD machine (see Fig. 5.1). It may be

a simple bus, or a cross-bar switch, or a multi-stage network (such as the butterfly network), etc. [Sto90].

5.2 The BBN Butterfly Machine

The BBN Butterfly parallel processor system is a distributed shared-memory MIMD machine [BBN87, GAG88, Mon89]. It has 32 nodes, with each node containing a processing element, a memory management unit and a communications processor. Collectively, the memories of all the nodes form the *shared memory* of the machine. All inter-processor communication is done using shared memory

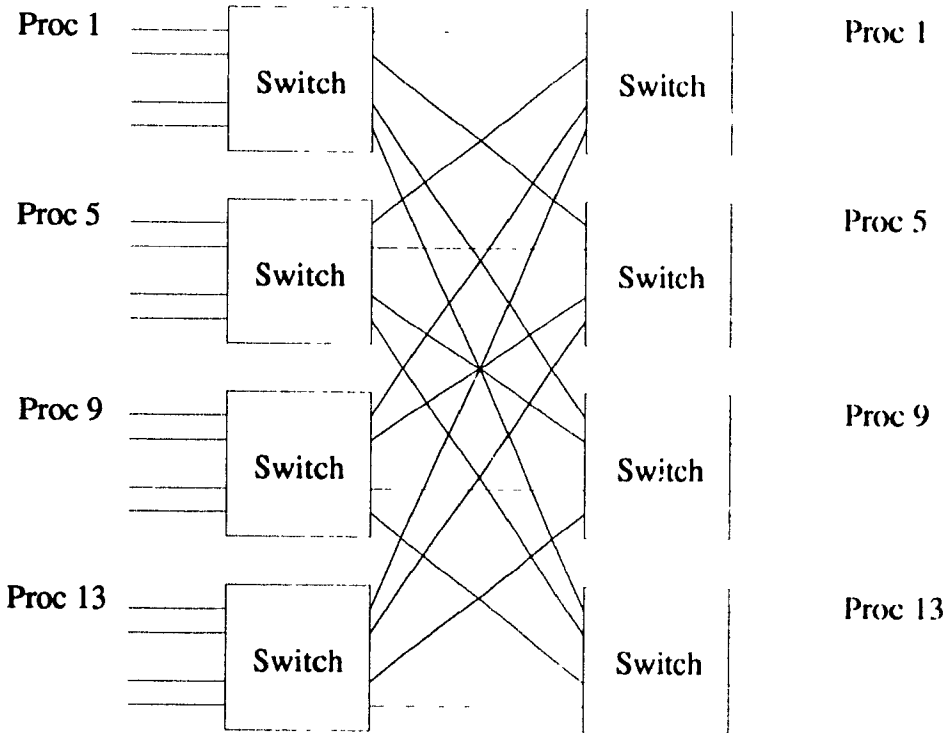

Figure 5.2: The Butterfly Interconnection Network for a 16 Processor System

These 32 nodes are connected by a multi-stage interconnection network known as the *butterfly switch* (see Fig. 5.2 for an example of the butterfly interconnection network for a 16 processor machine). Such a multi-stage connection organization makes it possible to keep all nodes equidistant from each other with regard to memory

accesses, even though they may physically be at different distances from one another. Thus, all remote memory accesses take the same amount of time (of course, the local memory access is much faster than remote memory access).

The BBN Butterfly comes with a library of subroutines which may be used by the programmer to initiate and control parallel programs in C and FORTRAN. This library of subroutines is known as the *Uniform System* . Let us discuss the Uniform System with reference to the parallel matrix multiplication program (see Fig. 5.3).

The Uniform System has subroutines to :

- set up the machine for parallel programming - *InitializeUs* (). This indicates to the operating system that the program will be invoking the Uniform System routines later on in order to be able to run some functions in parallel.

- initiate parallel programming by "generating" tasks *GenOnI* (). Multiple tasks are initiated by means of "generators" . Let us assume that there are P processors available for work, and T is a task that must be done on N ($N > P$) index values. Then, *GenOnI (T, N)* initiates the tasks on P processors, with each processor getting a unique index value ($0 \ldots (P-1)$) to work on. As each processor finishes its task with the given index, the generator re-invokes it on the same task with a new index. This is continued till some processor processes task T with index value $N-1$, at which time the processing comes to an end.

  In the matrix multiplication program, the generator used is *GenOnA* (), in the function *Body* *GenOnA* calls function *DotProduct* () with parameters i and j, which refer to the row of matrix b and column of matrix c, respectively. When *DotProduct* finishes computing the dot product of the two vectors, *GenOnA* gives it new values of i and j to compute the dot product of two new vectors. This process continues till the matrix multiplication is complete.

- share values among various processors - *Share* (). *Share* allows all the processors to share the value in a variable. This variable should always be a non-static global variable. In the matrix multiplication program, the values of variables *Size1* and *Size2* are shared among all the processors.

- allocate memory in either local memory or shared memory - *UsAlloc* (), etc. This family of subroutines allows the user to explicitly control the allocation of memory. *UsAlloc* allocates memory on any processor that has the most memory

```
/* Matrix multiply - unoptimized example program */

#include <us.h>

int Size1, Size2, junk;
float ** a, ** b, ** c;

InitProblemOnce ()
{
  int i, j;

  a = (float * *) UsAllocScatterMatrix (Size1, Size1,
                sizeof (float));
  b = (float * *) UsAllocScatterMatrix (Size1, Size2,
                sizeof (float));
  /*
   * We store the matrix C in its transposed form
   * therefore, C is actually a Size2 x Size1 matrix,
   * though it is stored as a Size1 x Size2 matrix.
   */
  c = (float * *) UsAllocScatterMatrix (Size1, Size2,
                sizeof (float));

  ShareScatterMatrix (& a, Size1);
  Share (& b);
  Share (& c);

  for (i = 0; i < Size1; i++)
  {
    for (j = 0; j < Size2; j++)
    {
      b [i][j] = (i == j) ? 3.0 : 0.0;
      c [i][j] = Size1 * i + j;
    }
  }
}

InitPerRun ()
{
  int i, j;

  for (i = 0; i < Size1; i++)
    for (j = 0; j    Size1; j++)
      a [i][j] = 0.0;
}

DotProduct (dummy, i, j)
int dummy, i, j;
{
  int k;
  float temp;
  float * bb = b [i], * cc    c [j];

  temp = 0.0;
  for (k = 0; k < Size2; k++)
    temp += *bb++ * *cc++;
  a [i][j] = temp;
}

Body ()
{
  GenOnA (DotProduct, Size1, Size1);
}

PrintAnswer (time, procs, speedup)
int time, procs;
float speedup;
{
  int i, j;
  int x, y;

  x = (Size1 < 6) ? Size1   6;
  y = (Size2 < 6) ? Size2   6;

  for (i = 0; i < x; i++)
  {
    printf ("\na [%d]  - ", i);
    for (j = 0; j < y; j++)
      printf ("%d. ", (int) a [i][j]);
  }
  printf ("\n");
  TimeTestPrint (time, procs, speedup);
}

main ()
{
  InitializeUs ();
  printf ("\nStarting Matrix Multiply\nMatrix size    ");
  scanf ("%d %d", & Size1, & Size2);
  printf ("Size 1 - %d, Size 2    %d\n", Size1, Size2);
  Share (& Size1);
  Share (& Size2);
  InitProblemOnce ();
  TimeTest (InitPerRun, Body, PrintAnswer);
}
```

Figure 5.3: A parallelized matrix multiplication program

available. *UsAllocLocal* allocates memory from local memory. *UsAllocOnUsProc* allocates memory on the designated processor. All the above routines allocate memory in such a way that it can be shared among all the processors. The standard C memory allocator *malloc*, on the other hand, allocates memory from local memory, but this cannot be shared with other processors.

In the matrix multiplication program, *UsScatterMatrix* is used to allocate each row of the matrix on a different processor in order to reduce memory contention.

- lock portions of code to ensure mutual exclusion - *UsLock* () and *UsUnlock* ().
- perform atomic operations to add and subtract values from a shared global variable *Atomic_add* ().

Chapter 6

Parallelizing the Algorithm

In this chapter, we shall discuss the issues faced in the parallelization of the A* Viterbi algorithm. Let us consider the various parts of the algorithm and see how they may be parallelized.

1. The Viterbi algorithm. Consider the equation :

$$\beta_i(t) = \max_j a_{ij} b_{ij}(y_{t+1}) \beta_j(t+1)$$

   This may be split into two parts.

   (a) The first part is the computation of the transition scores $a_{ij} * b_{ij}(y_{t+1})$ This can be parallelized in such a way that each processor computes the transition score for a different part of the observation sequence. In other words, each processor works on time 't' ($1 \leq t \leq T$)

   (b) Calculating the best path through the trellis. In this part, we make each processor work on N/P nodes of the phonetic graph, where N is the total number of nodes in the graph, and P is the number of processors being used.

2. Calculating the point scores. Consider the equation 4.1 (page 30)

$$\alpha_t(n) = \max_{t'<t} \max_f \max_{n'} \alpha_{t'}(n') V(Y_{t'+1}, \ldots, Y_t | f)$$

   Here, it is very easy to see that calculation of the scores for each phoneme 'f' can be done in parallel, and the maximum can then be calculated

3. The A* search. Here, parallelization can be achieved by allowing different processors to search disjoint sub-trees of the search space graph.

There are a few important implementation criteria to be considered here :

1. Contention for memory. The nodes of the phonetic graph will be accessed very frequently during the Viterbi phase of the algorithm. If all the nodes are allocated on a single memory module of the BBN Butterfly machine, then we would have problems with traffic being directed to one module of the machine. This would create *hot spots*, which would reduce the efficiency of the parallel algorithm.

   This problem is solved by statically distributing the nodes of the phonetic graph over the processors involved in the parallelization. Each processor will work on N/P nodes of the phonetic graph, as mentioned above; and the nodes that each processor works on will be available in that processor's local memory. This method of distribution of the nodes thus has the added advantage of reducing the number of remote memory accesses performed by the program.

2. Availability of memory. Each memory module has a limited amount of memory; if we were to attempt to put the entire lexical tree on one module, we would surely fail to do so. This problem is solved in a manner similar to the first -- we distribute the nodes of the lexical tree among all the processors which take part in the parallelization of the algorithm.

3. Remote memory accesses. On a distributed shared-memory machine, the penalty for accessing remote memory as compared to local memory is very high. On the BBN Butterfly machine, it is approximately 5 to 8 times costlier (in terms of time) to access remote memory as compared to local memory.

   This problem is reduced my making local copies of as much of the data as possible; for instance, each processor has a local copy of all the phoneme models, and the transition probabilities.

In this thesis, we will be working on isolated word recognition. To this end, we will be given each word as a sequence of acoustic observations, $Y_1, \ldots, Y_T$, and the program would attempt to map this observation sequence to a word in the lexical tree. Therefore, before we begin to calculate the β values during the Viterbi phase

of the algorithm, we would have the entire observation sequence for the word to be recognized. For the remainder of this chapter, we will assume that there are N nodes in the phonetic graph, and that there are P processors working in parallel on the algorithm.

6.1 Starting the Parallel Processing

Parallel processing is initiated by means of the *GenOnI*() generator routine (discussed in Section 5.2). This routine starts off all the P processors at the same time. Each processor is assigned a unique number $0 \ldots (P-1)$ by means of the *UsProc*() routine. Processor 0 acts as the controlling processor, it starts the parallel processing when needed, and collects the data after all the processors are done. After processor 0 does some initializations, it instructs processor 1 to read the phonetic graph, and processor 2 to read the lexical tree. The phonetic graph and the lexical tree are distributed among all the processors by means of the *UsAllocOnUsProc*() routine. After the initializations are done, all the processors other than processor 0 become idle till processor 0 gives them the signal to start parallel work on one part of the program or the other.

6.2 Phase I : The Parallel Viterbi Algorithm

In this section we shall discuss the parallelization of the β value calculation, and the parallelization of the α value calculation using the block Viterbi algorithm discussed in Section 4.2.

6.2.1 Part 1 : The Parallel β Value Calculation

Let us now consider the equation for the calculation of the β values :

$$\beta_i(t) = \max_j a_{ij} b_{ij}(Y_{t+1}) \beta_j(t+1)$$

where i and j are states in the phonetic graph, and $i \rightarrow j$ is a transition from i to j in the graph.

As mentioned earlier, we can split this up into two parts :

1. In the first part, we could calculate the transition scores,

$$T_{ij}(t+1) = a_{ij} b_{ij}(Y_{t+1})$$

in a parallel fashion. Each processor performs the calculation for a different value of 't' ($0 \leq t < T$). It is quite obvious that there is no dependence between a processor working on observation Y_t, and a processor working on observation $Y_{t'}$ ($t \neq t'$). In other words, we can expect to see parallelization for this part of the code rise linearly with the number of processors working on the code.

Both portions of the calculation, a_{ij}, and $b_{ij}(Y_{t+1})$ are obtained from tables which are stored in local memory, which allows us to reduce the time taken for the calculation by avoiding a costly remote memory access. The values could be stored in logarithmic form, which would then allow us to perform an addition instead of a floating point multiplication; this would also help in reducing the time taken for the calculation.

The $T_{ij}(t)$ calculated are then stored in a large matrix, with $N \times T$ values, for later use.

2. In the second part, we need to do $\beta_i(t) = \max_j \beta_j(t+1) T_{ij}(t+1)$. This is a straight forward table look-up (from the previous calculation), followed by a compare and select. The table is set up in such a way that the calculation of the β value for each node s in the phonetic graph can proceed in parallel as described below.

The phonetic graph has been statically distributed among the P processors in such a way that N/P states of the graph are on the local memory of each processor, and no two processors contain the same state in their local memory. At each time instant t ($0 \leq t < T$), each processor works on its set of N/P states. For each of its states i, it calculates

$$\max_j T_{ij}(t+1) \beta_i(t+1)$$

where j is the state linked to i by a transition $i \rightarrow j$. State j need not be on the local memory of the processor working on state i; in such a case, the processor would have to go to remote memory to fetch the values related to j. This plays a large part in restricting the amount of speed-up that is obtained in parallelizing this portion of the code.

6.2.2 Part 2 : The Parallel α Value Calculation

Let us consider the equation for the calculation of the α values :

$$\alpha_t(n) = \max_{t'<t} \max_{f} \max_{n'} \alpha_{t'}(n') V(Y_{t'+1}, \ldots, Y_t | f)$$

In this equation, all the processors work on time t ($1 \leq t \leq T$), with each processor working on a different f. After all the processors have finished finding

$$\max_{t'<t} \max_{n'} \alpha_{t'}(n') V(Y_{t'+1}, \ldots, Y_t | f)$$

for their particular phoneme(s) f, the results are collected by one processor, which then proceeds to find $\max_f$ of the values thus collected. As may be expected, this parallelization will not give us the speedup results that we experienced during the calculation of the transition scores (see section 6.2.1) because node n and node n' may be on different memory modules, forcing the processors to go to remote memory.

6.3 Phase II : The Parallel A* Algorithm

Parallelizing the A* and related algorithms has come under a lot of study in recent years [KRR88, KR87, RK87, RKR87, NV89]. The heuristic domain knowledge available for the A* algorithm can be used to avoid searching some parts of the search space which may be unpromising. This means that parallel processors following a simple strategy, such as divide the search space statically into disjoint parts and let each one be searched by a different processor, may end up doing a lot more work than a sequential processor. This would tend to reduce the speedup that can be obtained by parallel processing. In the following section, we discuss the different methods used to parallelize the A* algorithm, both for shared memory multi-processors, and for distributed memory multi-processors. These parallel formulations mainly differ in the data structures used to implement the **Open** list of the A* algorithm. The **Open** list may be implemented as a heap to allow $O(\log N)$ access time (where N is the number of entries in the list).

6.3.1 The Centralized Strategy

In this strategy, we use a global **Open** list, and let each parallel processor work on one of the current best nodes in the **Open** list. There is one important point to be

kept in mind with this approach :

Since **Open** will be accessed by all the processors frequently (each processor will access it when it has expanded the best node in the list, and has to expand the next best node), it will have to be maintained in a shared memory that is easily accessible to all the processors. In the BBN Butterfly parallel processor, there are Uniform System commands that allow the user to specify various portions of memory as shared, and these portions of memory can be accessed by all the processors. In addition, locks are provided so that this memory can be modified with mutual exclusion guaranteed. The deletions and insertions into the queue must be done concurrently and so, algorithms must be designed carefully in order to keep the locked portion of the **Open** list to a minimum while ensuring data integrity [RK88]. Even with all this, contention for **Open** will limit the performance of the machine.

6.3.2 The Distributed Strategy

One way to avoid the contention due to a centralized **Open** list is to let each processor have its own local **Open** list[1]. Initially, the search space is divided statically and given to different processors; this may be done by expanding some nodes and distributing them to the local **Open** lists of different processors. Now, all the processors select and expand nodes simultaneously without causing contention on a shared **Open** list as before.

However, in the absence of any communication between individual processors, it is possible that some processors may work on a good part of the search space, while others may work on bad parts that would have been pruned by the sequential search. This would lead to a high redundancy factor and poor speed-up. Some possible ways to get around this obstacle are listed below :

1. **The Blackboard Communication Strategy**

   In this strategy, there is a shared **Blackboard** through which the nodes are switched among processors as follows :

[1]These **Open** lists may still be implemented as heaps to allow $O(\log N)$ access time

After selecting a least f-value node from its local **Open** list, the processor proceeds with its expansion only if it is within a "tolerable" limit of the best node in the Blackboard.

If the selected node is much better than the best node in the Blackboard, then the processor transfers some of its good nodes to the Blackboard

If the selected node is much worse than the best node in the Blackboard, then the processor transfers some good nodes from the Blackboard to its local **Open** list.

In either case, a node is reselected for expansion from the local **Open** list. In my opinion, this Blackboard could be maintained as a heap as the only operations performed on it are deletions of the best node, and insertions of good nodes. The concurrent insertions/deletions could be performed on the list as suggested in [RK88]. It could be stored in shared memory (ie., distributed over all the processors) so that there is no contention at one processor for the Blackboard

The choice of the "tolerable" limit is important, as it affects the number of nodes expanded as well as the amount of node switching between local **Open** lists and the Blackboard. If the tolerance level is low, then nodes will be switched very frequently between the Blackboard and the local **Open** lists unless all the **Open** lists have the same value for their best nodes. If the tolerance level is high, then the node switching will occur less frequently, but then a processor could possibly expand nodes that are inferior to the nodes waiting to be expanded in other processors.

2. **The Random Communication Strategy**

In this strategy, each processor periodically puts the newly generated successors of the selected node into the **Open** list of a randomly selected processor. This ensures that if some processor has a good part of the search space, then others get a part of it. If the frequency of transfer is high, then the redundancy factor can be small; otherwise it can be very large. The choice of frequency of transfer is effectively determined by the cost of communication. If this cost is low (such as on the BBN Butterfly) then it would be best to perform communication after every node expansion

3. **The Ring Communication Strategy**

In this strategy, different processors are assumed to be connected in a virtual ring. Each processor periodically puts the newly generated successors of the

selected node into the **Open** list of one of its neighbours in the ring. This allows transfer of good work from one processor to another. As in the previous scheme, the cost of communication determines the choice of frequency of transfer.

From the studies done in [KRR88], it is seen that the distributed strategy with the **Blackboard** scheme performs very well for problems such as the Traveling Salesman Problem and the Vertex Cover Problem. It is superior to the centralized strategy, mainly because the centralized strategy is not good for problems with small granularity, and it is superior to the other distributed schemes discussed above, because it has a redundancy factor that can be easily controlled by the programmer.

6.3.3 Implementation of the Parallel A* Algorithm

We have used the centralized strategy in parallelizing the A* algorithm due to its simplicity. The parallel algorithm utilizes one **Open** list, and every processor goes to it to

- get a new node to expand; and
- insert the nodes generated.

Processor 0 expands a few nodes of the tree before it allows the other processors to work in parallel. This is done to ensure that there is no traffic jam among the processors at the beginning of the A* search. After the parallel search is invoked, each processor expands a node and puts the children of that node in its local shared memory space, while maintaining the links with the rest of the list on the other processors; in this way, the **Open** list is distributed among all the memories of the parallel machine. Insertion of nodes into and deletion of nodes from the **Open** list are atomic operations, and there is a lock to ensure that only one processor can access the **Open** list at any given time. When any processor detects the goal node, it sets a flag that causes all other processors to abort their searches and return control to processor 0. Processor 0 then deletes the **Open** list that was built during the A* search. This is done to ensure that no random insertions or deletions are done in the intervening period when the unsuccessful processors are still expanding their own nodes.

Chapter 7

Results and Interpretation

The Viterbi algorithm was parallelized as described in Chapter 6, and the results are given below. We performed the speed-up tests on two words – a short word ('he'), and a long word ('sabotage'). The rationale for this was that we could expect a fairly uniform speed-up for the parallel Viterbi algorithm for all words, but the A* algorithm would perform differently for words of differing length. We expect the A* algorithm to be speeded up quite significantly for long words, and not very much for short words.

During parallelization of the Viterbi algorithm, we postulated that the total number of remote memory accesses would increase with the number of processors being used in the parallelization. If we have P processors, then we expect at most $1/P$ local memory accesses, and $(P-1)/P$ remote memory accesses. This was found to hold true (see Fig. 7.1) during the test runs in which we measured the remote memory accesses for upto 10 processors.

The implication of this observation is quite obvious. Speed up will not increase infinitely as the number of processors increases. There will be an initial gain of speedup with number of processors, which will then drop when more processors are added. This reduction in speedup can be explained by the increased number of remote memory accesses. We have mentioned earlier that it is 5 – 8 times as costly (in terms of time) to access remote memory as it is to access local memory (see page 41). Thus, when a large number of processors are made to work on the problem, more time is spent in reading from and writing to remote memory than in performing useful work.

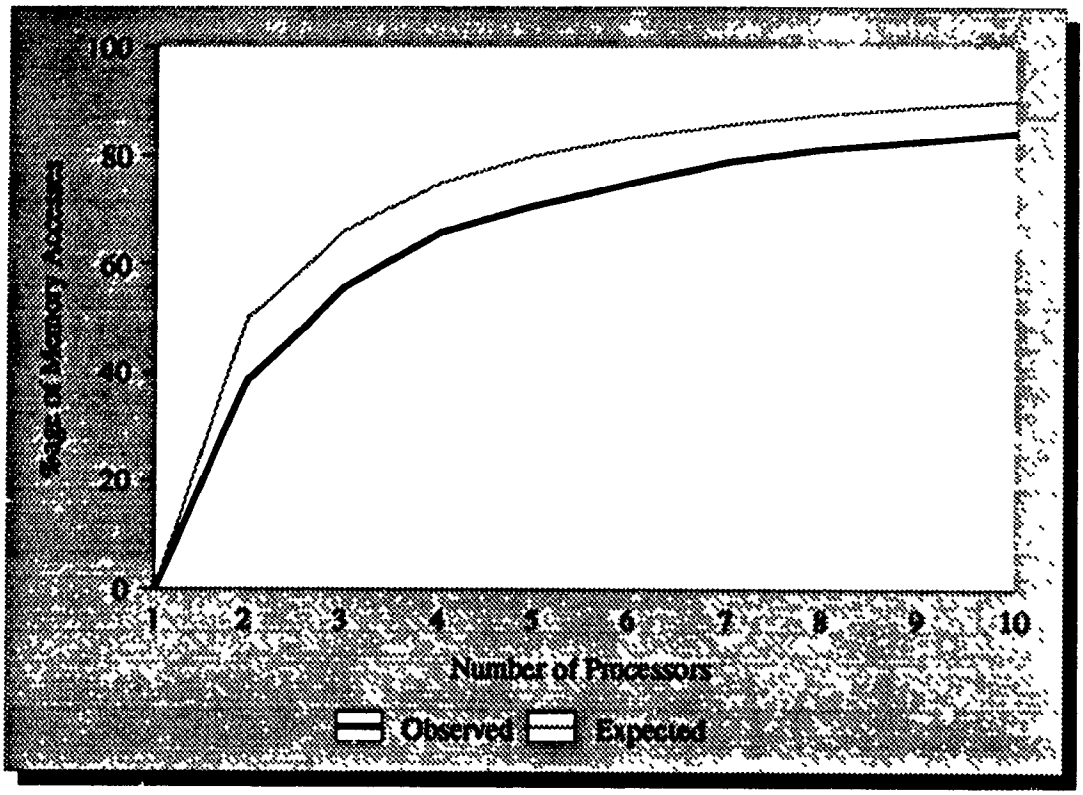

Figure 7.1: Growth of remote memory accesses with number of processors

This drop in speedup due to increased remote memory accesses pertains only to the Viterbi phase of the algorithm, and not to the A* phase. This is because the Viterbi phase involves a fixed number of computations, regardless of the number of processors involved. The A* phase, on the other hand, is a tree search, and the search is terminated when the goal node is found by any one of the processors involved in the search.

In the process of measuring the number of remote memory accesses during the parallel Viterbi algorithm, we observed a very significant fact namely, the phonetic graph is not a uniformly distributed graph. Different nodes of the graph have differing numbers of children. As a result of such a skewed structure, some processors do significantly more work than others. On referring to Table 7.1, we see that processor 0 makes $\approx$ 95000 remote memory accesses whereas processor 9 makes only $\approx$ 30000 remote memory accesses. Processor 0 thus performs about 3 times the number of remote memory accesses that processor 9 does, and so processor 9 would have finished its work, and would be idling while other processors may yet have more than half their work remaining.

This above problem could be fixed if we redesigned the phonetic graph to make it more suitable to be used in the distributed shared-memory multiprocessor system

Proc #	Remote	Local	Total
0	95264	30600	125864
1	94836	23772	118608
2	88704	19614	108318
3	82194	14532	96726
4	74424	11886	86310
5	61656	8694	70360
6	47628	5418	53046
7	38178	3318	41496
8	28350	1386	29736
9	30143	797	30940
Total	641377	120017	761394

Table 7.1: Memory Accesses Per Processor for 10 Processors

that we were using. Unfortunately, the redesigning of the graph to spread the work evenly among the processors is beyond the scope of this thesis. It is the opinion of the author that the graph, and indeed, the entire algorithm must be developed from the beginning with a multiprocessor architecture in mind. Significant gains in speed up can yet be possible with this approach to speech recognition

We will now present the results of the parallelization of the A*-Viterbi algorithm on the BBN Butterfly machine. The measurements were made in a very simple manner. Each of the three phases was implemented in the form of a function. The software was so implemented that there was one processor that controlled the parallel execution; this processor would invoke all the other processors to begin parallel processing, and it would perform a barrier synchronization to get the values from the other processors. With such a setup, it was easy to make calls to timing routines just before and just after each of the three functions was called. The timings were calculated in terms of microseconds in order to have accuracy. Each graph in the following two sections has a line graph drawn in black, and a smooth curve drawn in grey. The line graph is the actual data, and the grey line is a the result of a curve fitting algorithm on the actual data.

	Under each category, the time taken is displayed in microseconds					
	Beta Value		Impulse Response		A* Search	
	Time taken	Speed Up	Time taken	Speed Up	Time taken	Speed Up
1	119758	1.000	15324	1.000	5416	1.000
2	110183	1.087	14049	1.091	8058	0.672
3	58666	2.041	7733	1.982	6916	0.783
4	51195	2.339	6087	2.518	7187	0.754
5	46470	2.577	6808	2.251	7953	0.681
6	33374	3 588	5795	2.644	7704	0.703
7	30696	3.901	5362	2.858	7054	0.768
8	28720	4.170	3983	3.848	6837	0.792
9	24212	4 946	5004	3.062	7491	0.723
10	23066	5.192	5129	2.988	7429	0.729
11	20775	5.765	4291	3.571	7179	0.754
12	20816	5.753	4275	3.585	7270	0.745
13	19529	6.132	4170	3.675	7162	0.756
14	17504	6.842	3975	3.855	7508	0.721
15	18520	6.466	2850	5 377	7504	0.722
16	18737	6.392	3587	4.272	7692	0.704
17	18841	6.356	3033	5.053	7670	0.706
18	18662	6.417	2996	5.115	8092	0.669
19	18312	6.540	3166	4.840	8450	0.641
20	17966	6.666	3366	4.553	8766	0.618

Table 7.2: Table of results for word 'he'

7.1 Results for a Small Word

The following data and graphs show the timing and speedup results for a small word 'he'. We could come to the following conclusions from Table 7.2 :

- From Table 7.2, we see that when only one processor is working on the recognition, it takes almost 2 minutes (119 seconds) to calculate the β values. This time is reduced steadily as the number of processors working on the problem is increased, till it reaches about 19 seconds with 13 processors, a total speedup of about 6.13 times that of a single processor. Beyond 13 processors, the increase in speedup is negligible, reaching just 6.66 with 20 processors (see Fig. 7.2).

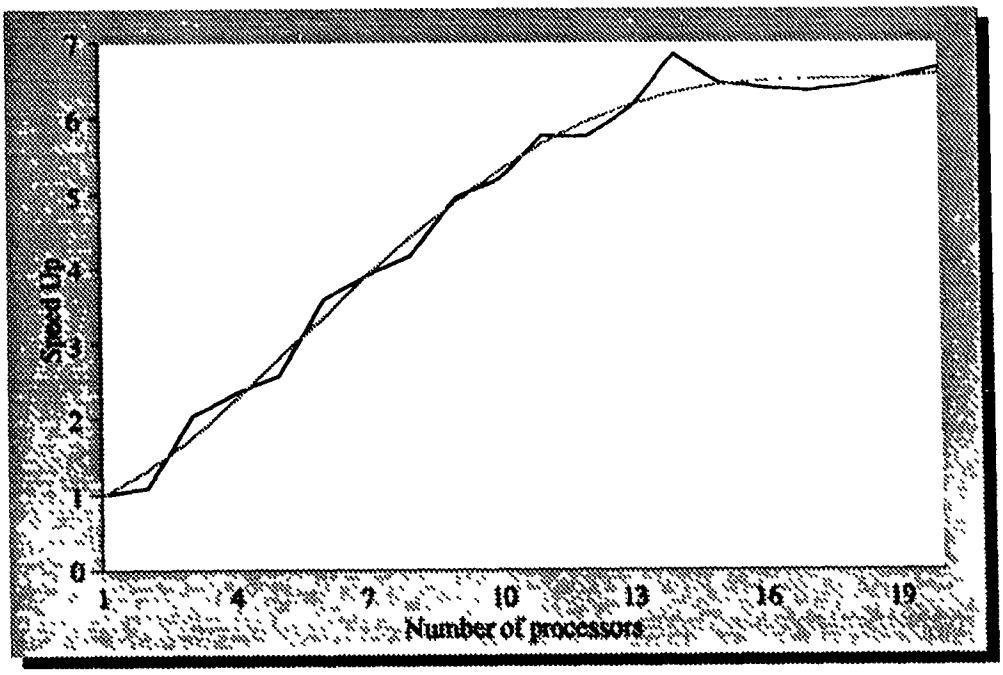

Figure 7.2: Speed up for calculating the β values

This phenomenon can be explained by considering the equation :

$$\beta_i(t) = \max_j a_{ij} b_{ij}(y_{t+1}) \beta_j(t+1)$$

When a large number of processors works on the calculation, the number of remote memory accesses increases (see Table 7.1 and Fig. 7.1). As we have mentioned on earlier occasions, remote memory accesses are very slow compared to local memory accesses, and this is a large part of the reason why the speedup does not increase beyond a certain limit.

- From Table 7.2, we see that with just one processor working on the α value calculation, it takes about 15 seconds to calculate all the α values. This is a fairly fast time to start with, and so we cannot expect very large speedup figures when parallel processing is initiated.

  Speedup for the α value calculation seems to rise almost continuously with number of processors involved, rising to about 3 with 10 processors and 4.5 with 20 processors (see Fig. 7.3). This is well below the rate of approximately 6 that we observed for the β value calculation. This is partly because α value calculation involves quite a bit of remote memory accesses that cannot at this time be resolved satisfactorily. A major factor in the lower speedup is the

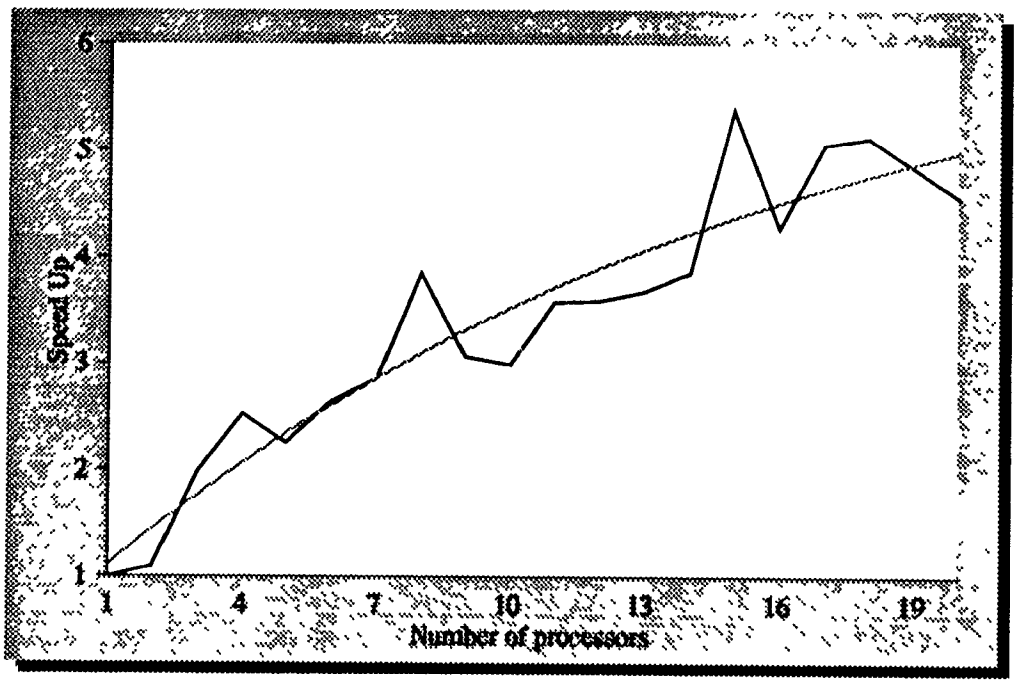

Figure 7.3: Speedup for calculating α values

equation itself :

$$\alpha_t(n) = \max_{t'<t} \max_{f} \max_{n'} \alpha_{t'}(n') V(Y_{t'+1}, \ldots, Y_t | f)$$

Each processor is calculating $\max_{t'<t} \max_{n'} \alpha_{t'}(n') V(Y_{t'+1}, \ldots, Y_t | f)$ for a different phoneme f at the same time instant t. After each processor finds the maxima over $(t' < t)$ and n', there is a barrier synchronization at which the maximum over all the phonemes f is calculated. Thus, though there are three maxima in the equation, two of them are "hidden" in the parallel processing, and only one of them $\max_f$ is a sequential bottle neck. The calculation of the α values is further helped by the fact that it is largely a table look-up from the data that have been calculated during the β value calculation.

- From Table 7.2, we see that in the uniprocessor case, the A* search was completed in about 5.1 seconds. This is a fairly fast search, and we cannot expect the parallel A* to do very much better. In fact, we should expect a probable degradation of performance, and in fact, that is what we do see. For 13 processors, it takes about 7.16 seconds to complete the A* search (see Fig. 7.1). This is because the one processor version of the algorithm expanded a very small number of nodes; this was calculated during the test runs, and it was found to be 9, i.e., the goal node was found in the ninth expansion of the search node.

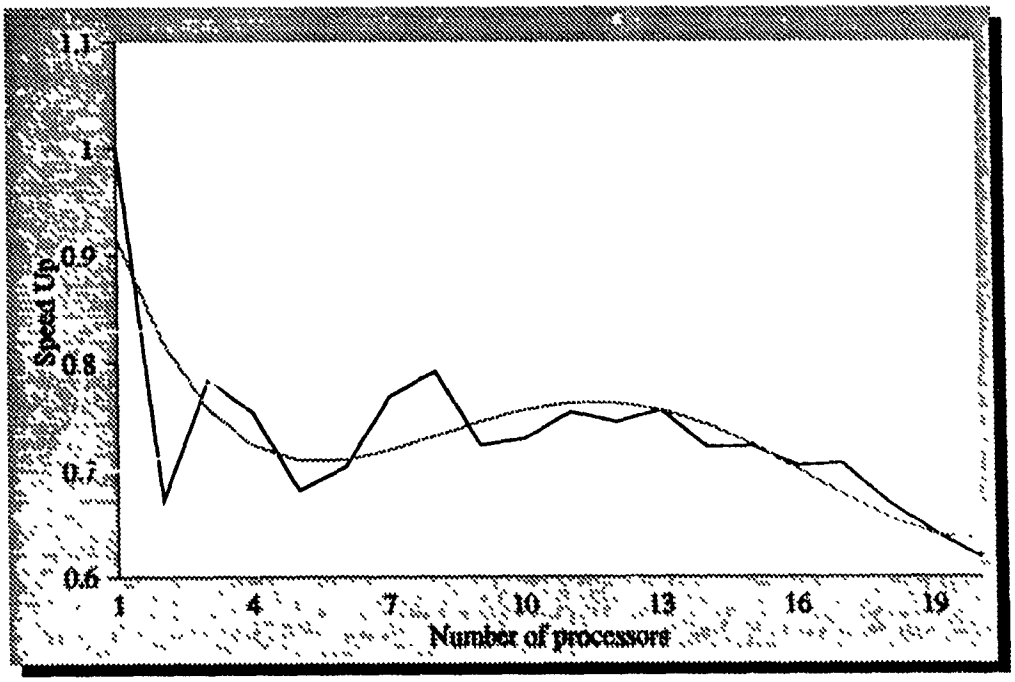

Figure 7.4: Speedup for doing the A* search

This is indeed a very small number of expansions, and it does not serve to parallelize this particular A* search because any parallelization would only increase the time spent in the searching. What we observe in the graph depicts exactly that scenario when we attempt to parallelize an A* search which expands very few nodes in the uniprocessor system. We can expect to see the effects of parallelization in the A* search only when a substantial number of expansions are performed in the uniprocessor system. As an empirical figure, the author would suggest that parallelization should not be attempted for fewer than 50 or 100 expansions. This figure is subject to variation depending on the multiprocessor system used, and the speed of the individual processor

	Under each category, the time taken is displayed in microseconds					
	Beta value		Impulse response		A* Search	
	Time taken	Speed Up	Time taken	Speed Up	Time taken	Speed Up
1	239287	1 000	60991	1.000	3447779	1.000
2	204116	1.172	43987	1.387	71595	48.157
3	104075	2.299	29300	2.082	45592	75.622
4	79812	2.998	21545	2.831	22062	156.277
5	67904	3 524	17770	3.432	23712	145.402
6	58116	4.117	14237	4.284	19420	177.538
7	52808	4.531	14833	4.112	15387	224.071
8	48816	4.902	12899	4.728	21412	161.021
9	48983	4.885	13474	4.527	22391	153.981
10	45458	5.264	12929	4.717	22745	151.584
11	39304	6.088	12716	4.796	20841	165.433
12	37008	6.466	12374	4.929	20375	169.216
13	32062	7.463	10700	5.700	25054	137.614
14	31670	7.556	12758	4.781	26096	132.119
15	31158	7.680	10912	5 589	19325	178.410
16	32012	7.475	8292	7.355	27412	125.776
17	30358	7.882	11916	5.118	27583	124.997
18	30354	7.883	14137	4.314	30075	114.639
19	30850	7.756	9604	6.351	33933	101.605
20	30816	7.765	10395	5.867	28883	119.371

Table 7.3: Table of results for word 'sabotage'

7.2 Results for a large word

The following table and figures show the timing and speedup results for a large word 'sabotage'. We can come to the following conclusions from the Table 7.3 :

- From Table 7.3, we see that in the one processor case, it takes almost 4 minutes ($\approx$ 239 seconds) to calculate the β values, indicating that this is a longer word than the one we discussed earlier. This time is reduced with parallel processing to $\approx$ 32 seconds with 13 processors, corresponding to a speedup of about 7.46. And again, as we observed earlier, this speedup does not increase much after 13

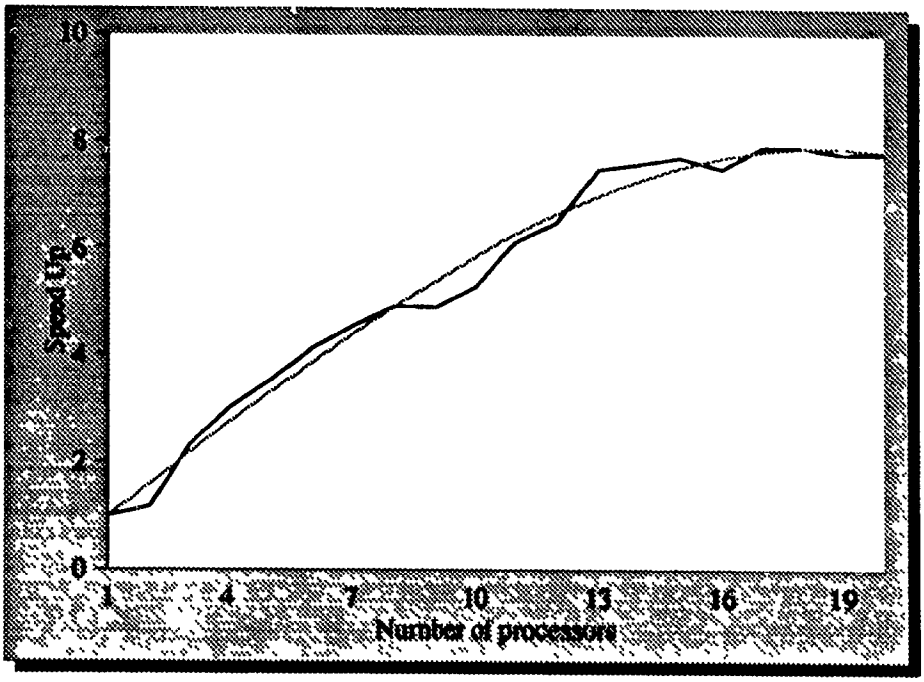

Figure 7.5: Speed up for calculating the β values

processors, and reaches a value of 7.76 with 20 processors (see Fig. 7.5). The reasons for this are the same as those discussed in the previous section.

- From Table 7.3, we see that it takes about a minute (60 seconds) to calculate the α values with the block Viterbi calculation. We may hope to gain better speedup values for this α value calculation than we obtained in the earlier case, because this case is much slower than the earlier one. And, from the table, we see that with 13 processors we have a speedup of 5.7, whereas earlier we had a speedup of 3.675. Beyond 13 processors, we don't get much increase in speedup, with a speedup of 6.35 with 19 processors (see Fig 7.6) The reasons for this are the same as the ones discussed in the previous chapter

- A* search is really helped by parallelization in this case because of the time it takes to do it in the uniprocessor case. From table 7.3, we see that it took almost an hour (3447 seconds) to get the goal node in the single processor case, but with more processors working, this was cut down to as little as 25 seconds with 13 processors. In fact, with just 7 processors, it took as little as 15 seconds to get the solution to the A* part, giving us a speedup of about 224 And this was achieved with a fairly simple parallelization strategy (the **centralized** strategy). This gives us a very good speedup curve (see Fig 7.7).

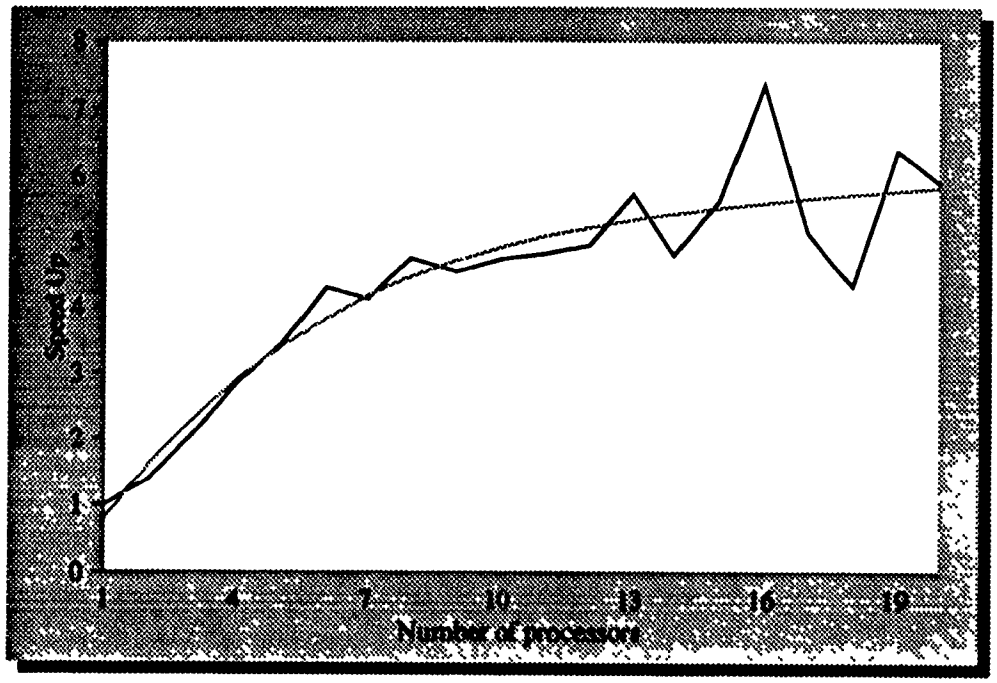

Figure 7.6 Speedup for calculating α value

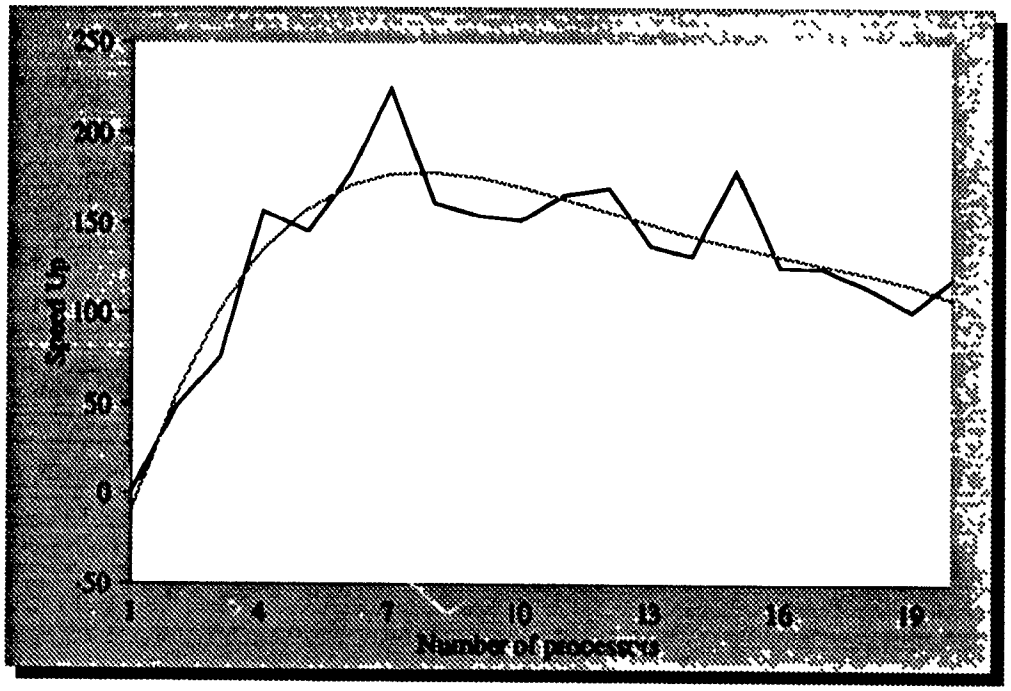

Figure 7.7: Speedup for doing the A* search

7.3 A Summary of the Results

The results we have observed and presented in the previous sections may be summarized as follows :

- The speedup for the β value calculation is about 6 or 7 with 13 processors, and it does not increase appreciably beyond that. We have seen that the phonetic graph is not uniformly structured, and this forces some processors to continue processing while other processors remain idle. The redesigning of the phonetic graph is a major undertaking which is beyond the scope of this thesis. If it is undertaken, and its transitions ordered in a different manner, conducive to the multiprocessor architecture that we have used, we should be able to see larger speedups than we have achieved

- The speedup for the α value calculation varies from about 3 for a small word to about 6 for a large word with 13 processors. A large portion of this algorithm involves looking up data from a structure that was filled as a side effect during the β value calculation. It was not possible to localize the accesses to this data structure on each processor, and so a large number of remote memory accesses were being performed during this phase. A thorough redesigning of this part of the algorithm is needed, with good modularity of data structures in order to allow the data structure to be allocated on different memory modules. This would reduce the number of remote memory accesses, and allow us to achieve greater speedup.

- We can expect very good speedup for the A* algorithm, provided the uniprocessor system expands a sufficiently large number of nodes before it finds the goal node. In the case where a small number of nodes are expanded before the goal node is discovered, parallelization does not give us any speedup, as we have noticed; it actually detracts from the performance of the algorithm, and should not be used in such cases. In the case where a large number of nodes are expanded, parallelization is very useful. As observed with the large word, we could get a speedup of 224 with 7 processors.

  It is the author's suggestion that parallelization of the A* be attempted only after 50 or 100 nodes are expanded during the A* search, depending on the multiprocessor machine used, and the speed of the individual processors. If the individual processors are slow, parallelization should be initiated after a fewer

number of nodes are expanded; if the individual processors are fast, parallelization should be deferred. The above number of 50 or 100 would be good for the BBN Butterfly machine.

Chapter 8

Conclusions and Future Work

In conclusion, the A*-Viterbi algorithm can be parallelized fairly easily, but with a distributed shared memory machine such as the BBN Butterfly, speedup can be limited by the amount of remote memory accesses that need to be done.

8.1 Comments on the Parallelization

The distribution of the phonetic graph was done in a very naive manner. If this distribution is done in a better fashion, we could reduce the number of remote memory accesses, and thus speed up the processing of the β value calculations.

During the calculation of the block Viterbi algorithm, the program again does a lot of remote memory accesses for values which could be copied into local memory. If this is done, the calculation of the block Viterbi algorithm could also be speeded up substantially.

The software that was used for this thesis was optimized for use in a sequential machine. This proved to be a major drawback in its parallelization. In the opinion of the author, parallelization of code can best be done as follows .

- Keep the code simple; do not make any optimizations by hand.
- Use simple data structures; keep in mind that these data structures may have to be shared.

- Have a knowledge of the basic architecture of the machine on which the parallelization is to be performed. The code will have to be customized differently for different parallel architectures.
- Have a global view of the code; identify portions of the code that may be parallelized, and move such parts into distinct modules.

8.2 Suggestions for Further Work

One very interesting project would be to implement this algorithm on a message-passing multiprocessor system and study the speedup results. There are portions of the code that can be fairly easily converted into pipelines, and a message passing multiprocessor system could implement pipelines more easily than a shared-memory system would. The entire approach to this problem would have to be different from the approach followed here. Static division of the phonetic graph as implemented in this thesis might not be possible, and so other avenues might have to be explored. Data sharing among the processors will not be as easy as going to remote memory. There will be no concept of a global memory from which each processor could access data. And there are a whole host of other issues that are unique to the message passing world that will not appear in the shared-memory world.

A second project would be to take the current implementation, and work on redesigning the phonetic graph to speedup the β value calculation part. One may also study as to whether the α value and β value calculations may be merged.

A third project would be to convert the A* search into an IDA* search. The IDA* is the Iteratively Deepening A* search which was developed to address one of the major problems with the A*. The A* search involves enormous amounts of memory – the search tree expands exponentially, and so does the memory it occupies. The IDA* search, on the other hand, uses memory in a linear fashion. The algorithm is a recursive one, and the memory used depends on the level of recursion at the given time. A second factor to note in the IDA* search is that we are guaranteed a linear speedup, and the goal node detected is guaranteed to be the most optimal one. And, unlike in the A*, the termination criteria need not be modified to detect the optimal goal node.

Appendix A

Code Used in Parallelization

On the BBN Butterfly machine, in order to run a program on more than one processor under the *Uniform System*, it has to be run under a *cluster*. for instance, if we wish to run the program *a.out* with 5 processors, we have to give the command line

```
% cluster 5 a.out
```

where the % sign is the normal UNIX prompt. This runs *a.out* with 5 processors, which may be used for parallel processing in the program. The function *main*() is executed by one of the 5 processors (this is decided by the scheduler at run time); we will refer to this processor as the *main* processor. The other 4 processors are inactive, waiting for commands from the main processor; these processors may be activated by means of the *generator* routines that are available in the Uniform System. These and other issues will be discussed later in this chapter.

In this chapter, we shall discuss the way in which the algorithm was parallelized, and how the parallel processing was controlled. Each section will discuss a different facet of the parallelization, and examples will be given in the form of pseudo-code. In some cases, where relevant, the actual code may be used to demonstrate the techniques used.

A.1 Starting the Processors

Parallel processing is initiated by the main program by means of the *generator* routine *GenOnI*().

```
int GenId = GenOnI (par_main, tot_procs);
```

The parameter *par_main* is the function that all the processors have to execute, and the parameter *tot_procs* is the number of processors that are going to work on the function *par_main*. *GenOnI* is a *synchronous* generator, i.e., it forces *tot_procs* number of processors, including the processor that is making the generator call, to work on the function indicated (in this case, *par_main*). When all the processors have finished working on the function, control then returns to the original processor which called *GenOnI*.

A.2 Allocation of Processor Numbers

Each processor is allocated a unique number in order to control the parallelization process. This is done by means of the *UsProc_Node* call. This call assigns a unique Uniform System processor number to the processor that calls it. This number may then be used to allocated memory on the processor, to force the processor to perform certain parallel tasks, etc. The usage is as follows :

```
int my_num = UsProc_Node;
```

The variable *my_num* may then be used throughout the program to specify the processor.

A.3 Starting and Stopping Parallel Execution

The functions *start_parallel* and *stop_parallel* are used to initiate and terminate parallel execution of the program. Each processor has the responsibility to trigger

num_to_wake other processors. This ensures a smooth transition from single processor execution to multi-processor execution of the code. The processor which does the triggering is referred to as the *parent* processor, and the processor which is triggered is the *child*. The following section of code sets the processor numbers of the child processors for the current processor. *to_wake* is an array that holds the identities of the child processors. *to_sleep* is the identity of the parent processor of the current processor. NUM_TO_WAKE is the maximum number of children that a processor can have.

```
to_wake = (short *) malloc (sizeof (short) * NUM_TO_WAKE);
num_to_wake = 0;

for (i = 0; i < NUM_TO_WAKE; i++)
{
    to_wake [i] = my_num * NUM_TO_WAKE + i + 1;
    if (to_wake [i] < procs_to_use) num_to_wake ++;
}

/*
 * The main processor (with my_num == 0) has no parents, therefore
 * its 'to_sleep' is 0
 */
if (!my_num)
    to_sleep = 0;
else
{
    to_sleep = my_num / NUM_TO_WAKE;
    if (!(my_num % NUM_TO_WAKE)) to_sleep--;
}
```

On completion of a task, if a processor has any child processors, it waits for a signal from its children indicating that they have finished their work and then signals its parent; if a processor has no child processors, it signals its parent immediately upon finishing its work. This method of signalling ensures that there is reduced contention when a large number of processors are involved in the parallel execution of the code.

```
/*
 * Trigger the other processors to do parallel processing
 */
start_parallel ()
{
    if there are children to wake
    {
        wake each child by setting the corresponding 'my_flag' variable

        set current 'my_flag' to be the number of children
    }
}

/*
 * Signal to the parent processor that the current processor
 * has finished execution of the code.
 */
stop_parallel ()
{
    if current processor is processor 0, then
        wait till all the children have signalled completion
    else
        if there are any children, then
            wait till all the children have signalled completion
        else
            set 'my_flag' to 0

    signal completion to parent processor
}
```

A.4 The Busy Wait Loop

All the processors other than processor 0 perform this infinite loop. Each of them waits on the pointer *my_flag*, the contents of which are set by the *parent* processor in order to initiate parallel execution of the code.

```
while (1)
{
    /*
     * Wait till the parent processor wakes you up
     */
    while (!(*my_flag)) UsWait (50);

    start_parallel (my_flag, num_to_wake, my_num);

    RefreshLocalShareValues ();

    switch (what_to_do)
    {
        case PARCOMPUTE :
            /* Computes the transition scores */
            ParCompute ();
            break;
        case PARGRAPHPROC :
            /* Calculates the max beta values */
            ParGraphProcDo(my_ud->GrProc, (*ParGraphIndex));
            break;
        case HMMPOINT :
            /* Calculates the alpha values with the block Viterbi algorithm */
            Atomic_add (doHMMPointFlag, 1);
            HMMPointCalculate ();
            Atomic_add (doHMMPointFlag, -1);
            break;
        case PARSEARCH :
            /* Does the A* search */
            ParLexiconSearch ();
            break;
    }

    stop_parallel (my_flag, num_to_wake, my_num);

    if (what_to_do == EXIT) break;
}
```

When *my_flag* has been set by the parent processor, the current processor then proceeds to wake its child processors, if any. It then calls the function *RefreshLocalShareValues*, which updates any shared variables whose values may have been set in the interim. It then checks the variable *what_to_do* in order to find out which function must be run in parallel. The switch statement helps to decide which of the four parts of the algorithm has to be computed in parallel. When the computation has been finished, each processor then calls the function *stop_parallel* to indicate to its parent that it has completed its task, and is awaiting further directions. The processor exits when the variable *what_to_do* directs it to.

A.5 Code for Parallel Computation of Transition Scores

The following code was used to parallelize the computation of the transition scores. We present it in the form of pseudo-code in order to explain it better.

```
ParCompute ()
{
    Get the time instant 't'
    while t < T
    {
        Set the data pointer to store the output values
        Get the observation vector

        for every phoneme model (each is an HMM)
        {
            Get the model
            Set the data pointer to input the data

            If the model is not NULL, compute the transition scores
                for the model at the given time
        }

        Get the next time instant
    }
}
```

It may be noticed that we have used a *while* loop to compute the values for the time. This is deliberately done in order to make parallelization possible. With the *while* loop, getting the next time instant is done atomically, by means of locks. This allows each processor to get a unique time instant for it to work on. This might not have been accomplished as easily if we had used a *for* loop.

A.6 Code for Calculating the β Values

To calculate the β values, we need to calculate the max over all the states j in the phonetic graph for each time instant t. Therefore, each processor works on one part of the graph for a particular time instant, and the results are collated by processor 0 before going on to the next time instant.

```
ParGraph ()
{
    for each time instant 't' (0 <= t < T)
    {
        start the parallel execution
        calculate the max beta value
        stop the parallel execution
    }
}
```

A.6.1 Calculating the Max for the β Values

To calculate the max over all the states j in the phonetic graph, each processor works on just the states in its local memory, thus reducing the total time spent in this section of code. The speedup here cannot be linear with the number of processors because while getting the best value from all the states connected to the current state, a processor may have to go to remote memory to get the concerned values. In fact, with a larger number of processors, the loop starts performing very badly, because the number of remote memory accesses becomes quite large.

```
ParGraphProcDo ()
{
    for each state on the current processor
    {
        Get the phoneme corresponding to the state
        and set the data pointers

        Get the best value from all the states connected to
        the current state
    }
}
```

A.7 Code for Calculating the α Values

Each of the processors executes the following piece of code, thus performing the calculation for one of the phoneme models.

```
Get the phoneme model index

while there are phoneme models to be done
{
    Get the phoneme model

    if the model is not NULL
    {
        for the time period t to t'
            Calculate the best alpha value
    }
    Advance the phoneme model index
}
```

A.8 The Parallel A* Code

Every processor executes the following piece of code. Each processor accesses the **Open** list to get the best node in the list. Once the processor has the best node, it

deletes the node from the list. This process of getting the node and deleting it from the list is done after locking the list so that no other processor changes the list at the same time. The processor then checks the node to see if it is the desired goal node. If so, it signals the other processors to cease searching, and it prints the top 25 contenders from the **Open** list, and returns SUCCESS to processor 0. If the node is not the desired goal node, the processor expands the node and puts its children on the *Open* list for later searching.

```
doLexiconSearch ()
{
    do infinitely
    {
        Check the abort flag
        If some other processor has found the
            goal node return ABORT

        Lock the OPEN list
        Get the top node
        Delete the top node from the OPEN list
        Unlock the OPEN list

        If the node just obtained is the best node then
        {
           set the flag to abort the processing in the other processors.
           lock the OPEN list
           display the topmost entry, which is the goal
           unlock the OPEN list
           return SUCCESS
        }

        Expand the node just obtained, and add its children
        to the search space
    }
}
```

Appendix B

The Uniform System Subroutines

The BBN Butterfly multiprocessor machine has a library of subroutines that may be used to parallelize a program. This library of subroutines is known as the *Uniform System*. In this section, we shall study the major Uniform System library calls, which involve processor control, memory management, synchronization, and atomic operations. These four categories are very essential for managing a parallel processing system.

B.1 Generators

Generators are part of the processor management routines of the Uniform System. They control the starting and finishing of a *task*. In general, a task should be kept fairly small so that the system can respond to changing task scenarios. The Uniform System supports two generator control disciplines [BBN89] :

- **Synchronous** generators return to the caller after all of the generated tasks have been processed. Furthermore, the processor that calls a synchronous generator always works on the tasks that are generated.
- **Asynchronous** generators return to the caller as soon as the generator has been activated. This enables the calling process to do other work. The calling process can later work on generated tasks if it so chooses.

The Uniform System supports several "families" of generators :

1. *Index* family. Given an integer range, generators in the index family generate a task for each value (index) within the range. The call may be of the form :

```
code = GenOnI (worker, range);
```

or

```
code = AsyncGenOnI (worker, range);
```

This generates tasks of the form

```
worker (0, index);
```

where the parameter *index* ranges from $0 \ldots (range - 1)$; in this case, *range* must be less than 2^31. *GenOnI* is the synchronous generator, and *AsyncGenOnI* is the asynchronous generator.

2. *Array* family. Given two integer ranges (which can be thought of as array dimensions), generators in the array family generate a task for each pair of values (which can be thought of as row and column indices) within the ranges. The call may be of the form :

```
code = GenOnA (worker, range1, range2);
```

or

```
code = AsyncGenOnA (worker, range1, range2);
```

This generates tasks of the form

```
worker (0, index1, index2);
```

where *index1* ranges from $0 \ldots (range1 - 1)$, and *index2* takes values $0 \ldots (range2 - 1)$.

B.2 Memory Allocators

The Uniform System provides a variety of memory allocators that allocate storage in globally shared memory. The normal allocator, *malloc*, can be used with Uniform System programs to allocate storage in process private memory; such memory cannot be shared among processors.

The various memory allocators are :

- *UsAlloc (SizeInBytes)* will allocate a block of storage on a processor whose memory is least used.
- *UsAllocLocal (SizeInBytes)* will allocate globally shared storage on the local processor.
- *UsAllocOnUsProc (Processor, SizeInBytes)* will allocate globally shared storage on the specified processor. In this case, *Processor* is a Uniform System virtual processor number. This virtual processor number is obtained from the variable *UsProc_Node*, and it ranges from $0 \ldots (P-1)$ where P is the number of processor available to the program.

B.3 Synchronization and Atomic Operations

Sometimes two processors need to work on the same data at the same time. If the order of work does not matter, for instance, in the incrementing of a counter, the principal concern is that the processors should not interfere with one another. The Uniform System supports atomic operations for 16-bit and 32-bit quantities. The following functions will do atomic operations on addition, 'and'ing, and 'or'ing :

- **atomadd** and **atomadd32** perform atomic additions on 16 and 32 bit quantities respectively. The usage is

  ```
  int x = atomadd32 (intptr, increment);
  ```

  In this case, *intptr* must be a pointer to type **long** (or **int** where the implementation allocates 4 bytes for **int**). After the atomic addition, **intptr* will have been incremented by the value *increment* and x will have the older value of **intptr*. *atomadd* should be used while working with quantities of type **short**.
- **atomand** and **atomand32** perform atomic 'and'ing operations on 16 and 32 bit quantities respectively.
- **atomior** and **atomior32** perform atomic 'or'ing operations on 16 and 32 bit quantities respectively.

Some cases may require more than a simple atomic operation. In these cases, it may be necessary to construct a *lock* around the code, as follows :

```
lock;
    operations that must be atomic
unlock;
```

The Uniform System provides the following lock and unlock operations :

```
UsLock (lock, n);
UsUnlock (lock);
```

In this case, *lock* is a pointer to a **short**, and is stored in globally shared memory, and should be accessible by all the processors involved in the parallel processing. n is an integer that specifies the time to wait in tens of microseconds between attempts to set the lock. Using 0 for n forces the use of a default value, which is about one millisecond.

Bibliography

[AI87] Arvind and R. A. Iannucci. Two Fundamental Issues in Multiprocessing. Computation Structures Group Memo 226, MIT Laboratory for Computer Science, 1987.

[ASP91] S. Austin, R. Schwartz, and P. Placeway. The Forward-Backward Search Algorithm. In *IEEE International Conference of Acoustics, Speech and Signal Processing, Toronto*, pages 697 – 700, 1991.

[BBN87] BBN Advanced Computers Inc. *Butterfly Product Overview*, 1987.

[BBN89] BBN Advanced Computers Inc. *Programming with the Uniform System*, 1989.

[BE67] L. E. Baum and J. A. Eagon. An Inequality with Applicatons to Statistical Estimation for Probabilistic Functions of Markov Processes and to a Model of Ecology. *American Math Society Bulletin*, 73:360 - 362, 1967.

[BJM83] L. R. Bahl, F. Jelinek, and R. L. Mercer. A Maximum Likelihood Approach to Continuous Speech Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-5(2):179 – 190, 1983.

[BM90] P. J. Black and T. H.-Y. Meng. A Hardware Efficient Parallel Viterbi Algorithm. In *IEEE International Conference of Acoustics, Speech and Signal Processing, Vol 2*, pages 893 – 896, 1990.

[Fly66] M. J. Flynn. Very High-Speed Computing Systems. *Proceedings of the IEEE*, 54:1901 - 1909, December 1966.

[FM89] G. Fettweis and H. Meyr. Parallel Viterbi Algorithm Implementation : Breaking the ACS-Bottleneck. *IEEE Transactions on Communications*, 37(8):785 – 790, August 1989.

[FM91] G. Fettweis and H. Meyr. High-Speed Parallel Viterbi Decoding : Algorithm and VLSI-Architecture. *IEEE Communications Magazine*, pages 46 – 55, May 1991.

[For73] G. D. Forney. The Viterbi Algorithm. *Proceedings of the IEEE*, 61(3):268 278, March 1973.

[GAG88] E. F. Gehringer, J. Abullarade, and M. H. Gulyn. A Survey of Commercial Parallel Processors. *Computer Architecture News*, 16(4):75 107, September 1988.

[HP90] J. L. Hennessy and D. A. Patterson. *Computer Architectur : A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.

[KCSK87] O. Kimball, L. Cosell, R. Schwartz, and M. Krasner. Efficient Implementation of Continuous Speech Recognition on a Large Scale Parallel Processor In *International Conference of Acoustics, Speech and Signal Processing, Dallas*, pages 77 - 80, 1987.

[Ken90] P. Kenny. The A*-Viterbi Algorithm. *Personal Communication*, 1990.

[KHG+91] P. Kenny, R. Hollan, V. Gupta, M. Lennig, P. Mermelstein, and D. O'Shaughnessy. A* - Admissible Heuristics for Rapid Lexical Access. In *IEEE Proceedings of the ICASSP*, 1991.

[KR87] V. Kumar and V. N Rao. Parallel Depth First Search Part 2. Analysis. *International Journal of Parallel Programming*, 16(6):501 519, 1987

[KRR88] V. Kumar, K. Ramesh, and V. N. Rao. Parallel Best-First Search of State Space Graphs : A Summary of Results. In *Proceedings of the National Conference on Artificial Intelligence (AAAI 88)*, pages 122 127, 1988.

[Lee89] Kai-Fu Lee. *Automatic Speech Recognition : The Development of the SPHINX System*. Kluwer Academic Publishers, 1989.

[LRS83] S. E. Levinson, L. R. Rabiner, and M. M. Sondhi An Introduction to the Application of the Theory of Probabilistic Functions of a Markov Process to Automatic Speech Recognition. *The Bell System Technical Journal*, 62(4):1035 - 1074, April 1983.

[Mon89] J.-M. Monti. Evaluation of a GP1000 Butterfly Computer. ACAPS Technical Memo 14, School of Computer Science, McGill University, Montreal, Que., October 1989.

[Nil80] N. J. Nilsson. *Principles of Artificial Intelligence*. Tioga Press, 1980.

[NV89] K. S. Natarajan and Sarkar V. Processor Scheduling Algorithms for Constraint - Satisfaction Search Problems. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 140 149, August 1989.

[RK87] V. N. Rao and V. Kumar. Parallel Depth First Search. Part 1. Implementation. *International Journal of Parallel Programming*, 16(6):479 - 500, 1987.

[RK88] V. N. Rao and V. Kumar. Concurrent Insertions and Deletions in a Priority Queue. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 207 - 211, August 1988.

[RKR87] V. N. Rao, V. Kumar, and K. Ramesh. A Parallel Implementation of Iterative - Deepening A*. In *Proceedings of the National Conference of Artificial Intelligence (AAAI - 87)*, pages 178 - 182, 1987.

[Sto90] H. S. Stone. *High-Performance Computer Architecture, 2nd Edn.* Addison-Wesley Publishing Co., 1990.

[Vit67] A. J. Viterbi. Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm. *IEEE Transaction on Information Theory*, IT-13:260 - 269, April 1967.

[WL88] Kuei Ann Wen and Jau Yien Lee. Parallel Processing for Viterbi Algorithm. *Electronics Letters*, 24(17):1098 - 1099, 18th August 1988.

[ZC89] Y. F. Zhang and P. Csillag. Parallel Architecture for High-Speed Viterbi Decoding of Convolutional Codes. *Electronics Letters*, 25(14):887 - 888, 6th July 1989.

Index