Parallel Implementation of Fast Beam Search for Speaker-Independent Continuous Speech Recognition

M.K. Ravishankar
Computer Science & Automation
Indian Institute of Science
Bangalore – 560-012
INDIA

July 16, 1993

Contents

1 Introduction

This report describes a parallel, multi-threaded implementation of the Fast Beam Search (FBS) algorithm for continuous, speaker-independent speech recognition. The original, sequential version of this algorithm, called SPHINX, has been implemented by K.F.Lee and others at Carnegie Mellon University ([7]).

The main goal of this work is to explore ways of speeding up SPHINX via concurrency. The version of SPHINX we are working with[1] runs about 15 times slower than real-time on a DECStation 5000 to recognize spoken utterances from a 1000-word vocabulary, and about 40 times slower with a 5000-word vocabulary. (In this report, we shall mainly concentrate on experiments from the 1000-word vocabulary.) Although it is easy to show the existence of a high degree of parallelism, the situation is complicated by the enormous memory bandwidth required to approach real-time recognition speeds. Any parallelization scheme must also be able to sustain this bandwidth requirement on today's parallel processing machines.

The scheme outlined in this document demonstrates that one can indeed parallelize FBS very efficiently on shared-memory multiprocessors. It does this by statically partitioning data and computation among multiple *threads*. Individual threads are assigned to separate processors, while the presence of multiple memory modules in the machine provides the necessary memory bandwidth. However, even state-of-the-art shared-memory architectures can be rendered helpless by excessive inter-thread communication or remote memory accesses, which swamps the memory interconnection network. The static partitioning employed in parallel-FBS is specifically intended to minimize such inter-thread communication. At the same time, it attains a very even computational load balance among threads. One such implementation has been carried out on the PLUS multiprocessor ([5]) at Carnegie Mellon University using the *C-threads* facility ([6]). We include performance figures based on measurements carried out on this implemenatation.

In order to keep this document relatively self-contained, we begin with an overview of the sequential FBS implementation in Section 2, including a brief, abstract development of the speaker-independent, continuous speech recognition algorithm. We also examine how the computational load is distributed among various steps in the sequential algorithm. In Section 3 we describe the main features of the parallel implementation of FBS, and finally, in Section 4 we give details of performance statistics gathered from simulations as well as the implementation on the PLUS multiprocessor. It should be pointed out that the parallelization has been carried out entirely by hand; automatic parallelization of FBS is one of our major tasks for future research.

2 Overview of Sequential FBS

We begin with an overview of the original, sequential implementation of FBS. We concentrate only on the essential features of FBS relevant to this document. For further details please refer to [8] and [7]. In the following sections we introduce the representation of speech components in FBS, and an abstract description of the speech recognition problem, before moving on to the specific description of the sequential FBS implementation. We also take a look at the distribution of computational load for various steps in the algorithm.

[1] SPHINX is constantly being changed for obtaining better accuracy and for handling larger voculaburies. The parallel version of the FBS algorithm reported here corresponds to the SPHINX version of summer '92. We also plan to adapt the work reported here to future versions of SPHINX.

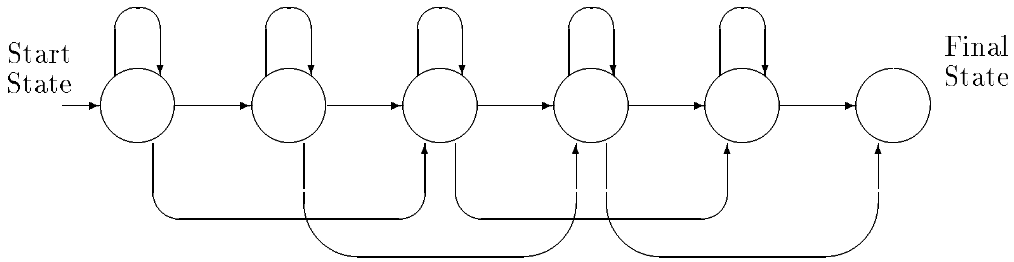

Figure 1: Topology of a Generic Hidden Markov Model

2.1 Representation

The raw, continuous speech signal is processed to extract a number of relevant *features*, such as signal power and frequency spectra, in consecutive 10msec segments or *frames*. There are 4 features in the current FBS implementation; the exact physical significance of these features and the algorithms for extracting them are irrelevant and beyond the scope of this report. The point to note here is that the computational effort required to convert the raw, analog speech signal into discrete feature values every 10msec is not significant, and it can be readily overlapped with the recognition algorithm. Henceforth, we shall simply look at the speech input as a sequence of 10msec frames, each of which contains a number of (8-bit) values, one for each feature. The speech recognition problem is concerned with the determination of the sequence of words most likely to produce the observed speech input.

In FBS, each word is described as a sequence of context-independent phonemes or *phones*, denoting the phonetic content of spoken words. Phones represent something smaller than syllables. (For example, the word "available" is represented by a sequence of seven phones, denoted as: AX-V-EY-L-AX-B-L.) This allows the entire dictionary to be represented by a relatively small number of phones; only 48 phones are needed to make up all the words in the current, 997-word dictionary.

Each phone is, in turn, represented by a hidden Markov model (HMM), consisting of 6 states and 14 state transitions. (We shall use the terms "model" and "HMM" interchangeably in this report.) There is one start state and one final state among the 6 states of an HMM. Each transition in a HMM accounts for a 10msec frame of speech. Associated with each transition is a static or *intrinsic* transition probability, as well as a dynamic or *acoustic* transition probability. The latter represents the probability of the transition producing any given feature value. The acoustic probability of a transition is, therefore, actually a set of probability distributions, one for each feature in the speech signal. Since our features are an 8-bit quantities, each probability distribution is a table of 256 discrete probability values. Figure 1 shows the generic HMM used for all phones[2].

A word can be represented by linearly chaining the HMM's corresponding to its phones, with an ϵ-transition[3] from the final state of each HMM to the start state of its successor; the start state for the word is simply that of the first HMM, and the final state that of the last HMM in the sequence. Clearly, one can trivially extend this scheme to a string of words. Also, it is straightforward to eliminate ϵ-transitions in such composite sequences by

[2]The HMMs are generated automatically by various learning and training algorithms which are beyond the scope of this report, and in any case, irrelevant to our topic.

[3]The ϵ-transitions do not consume an input frame.

adding appropriate non-ϵ-transitions, in a manner analogous to eliminating ϵ-transitions in finite state automata.

The above scheme does not, however, yield good results for speaker-independent, continuous speech recognition, partly because the transition from one phone to the next is too abrupt (in our model). To improve continuous speech recognition, more complex models are constructed by considering each context-independent phone in conjunction with all left- and right-context phones occurring in the language. Each such triple is called a *triphone*. Like context-independent phones, triphone models have corresponding HMMs[4].

The performance is further improved by adopting a *language model* consisting of two components. A *bigram* table contains a list of successor words for each word in the dictionary, together with the probabilities of such word pair occurrences in the language. A *unigram* table lists the stand-alone probability of occurrence of each word in the language. These two probabilities can be represented in the HMM-sequence for a string of words by assigning an appropriate intrinsic probability to an ϵ-transition leading from the final state of a word to the start state of the next word.

2.2 Abstract Speech Recognition Algorithm

From an abstract viewpoint, speech recognition consists of simply matching all possible word sequences against the speech input, and selecting the best possible match. That is, given a speech input of N frames, we search for the most likely path (i.e., sequence of transitions) through the representations of all possible sequences of words[5], as follows:

```
for (each possible word sequence W) {
    construct the corresponding sequence of triphones;
    eliminate epsilon-transitions from it;
    find all possible paths, with exactly N transitions,
        from the start state to the final state in this sequence;
    for (each such path T)
        compute probability(T), given the speech input; /* see below */
    probability(W) = max(probability(T)) over all paths T in W;
}
recognized speech = word sequence W with highest probability(W);
```

Computing probability(T), where T is some path through a sequence of HMMs, is straightforward. Let us use S_n to denote the state we are in after traversing n transitions, and $P(S)$ to denote the probability of any state S. Initially, S_0 is the start state for the sequence, $P(S_0) = 1$, and the probability of all other states=0. Also, S_N must be the final state of the HMM sequence. Let t_n denote the n-th transition in path T; it consumes the n-th speech input frame and takes us from state S_{n-1} to S_n. Let $P_i(t)$ denote the intrinsic or speech-independent probability of any transition t, and $P_a(t)$ the acoustic or speech-dependent probability of transition t for a given speech frame (i.e., the probability that transition t will produce the feature values observed during the given frame). The latter is obtained by looking up t's acoustic probability distribution tables for the observed feature values. Then,

[4] Although there are 48x48x48 potential triphones, only a small subset is needed in practice, given the actual occurrences of left- and right- contexts for each phone. Our language has 7597 triphone and phone HMMs.

[5] The set of all possible word sequences is infinite, since there is no limit to a sequence's length. However, we can confine ourselves to the finite set of word sequences whose shortest path does does not exceed N.

$$P(S_n) = P(S_{n-1})P_i(t_n)P_a(t_n).$$

Probability(T) is simply $P(S_N)$, the probability of the final state of the HMM sequence after traversing the entire path T (and consuming all N input frames).

Clearly, the above formulation is extremely time-consuming—it is an exponential process, as the list of possible word sequences and the number of paths to be searched grows exponentially with length. Fortunately, the same algorithm can be recast as a dynamic programming problem, based on the observation that in the optimal (highest probability) path from S_0 to S_N, any *initial sub-path* from S_0 to an intermediate state S_n must also be optimal. Looking at it another way, if we can identify an optimal initial sub-path to a given state S after n transitions, all other initial sub-paths leading to S after n transitions can be dropped from further investigation, since they can never lead to an optimal path. Thus, we are interested only in the *highest* probability value that any state of any word could possibly have at any given moment or frame. (Of course, in order to identify the spoken words we should also be able to trace the sub-path that led to this optimum value for each state.) This leads directly to the FBS implementation.

2.3 Outline of FBS Implementation

The key data structure in the FBS implementation is the instantiation of each word in the dictionary as a sequence of its constituent triphone models. The computation is primarily concerned with maintaining the (best) probability for each state in this data structure at each frame. That is, at the end of each successive frame, one knows the highest probability of finding oneself in any given state of any given word.

Figure 2 depicts the data structure. The triphone HMM instances are referred to as *channels*, and the probability computation for each state in each frame is termed *channel evaluation*. The entire data structure is called the *WordChannel* array.

Note the channel fanout at the end of each word in the *WordChannel* array. It indicates that the last channel in a word can be derived from several possible triphone HMMs. Recall that a triphone is made up of a central context-independent phone, in conjunction with a left- and a right-context phone. The right-context phone for the last triphone in a word must, therefore, be the first phone of the next uttered word, which can be anything in the language. Hence, one must consider all possible right-context phones for evaluation at the end of each word. In the current vocabulary, there are, on the average, several tens of possible right contexts for each word, whereas there are fewer than 10 phones per word. A similar consideration exists for the first channel of each word, but its discussion is deferred to the end of this section.

The FBS algorithm is a standard, time-synchronous Viterbi beam search (see [3]). This algorithm processes the input speech time-synchronously, completely updating all accessible channel states for frame n, before moving on to frame $n+1$. Let $P(S,n)$ represent the probability of any state S after n frames are consumed. Initially, $P(S,0) = 1$ if S is the start state[6], and 0 if S is any other state. Also, let $P_i(t)$ be the intrinsic probability of any transition t, and $P_a(t,n)$ the acoustic probability of transition t, given the input speech feature values in the n-th frame. (Note the small change in notation from Section 2.2.) The update for frame n has two stages:

[6] A spoken utterance is always assumed to begin with a unique *silence*-word. Thus, the start state is the first state of the HMM sequence representing the silence-word.

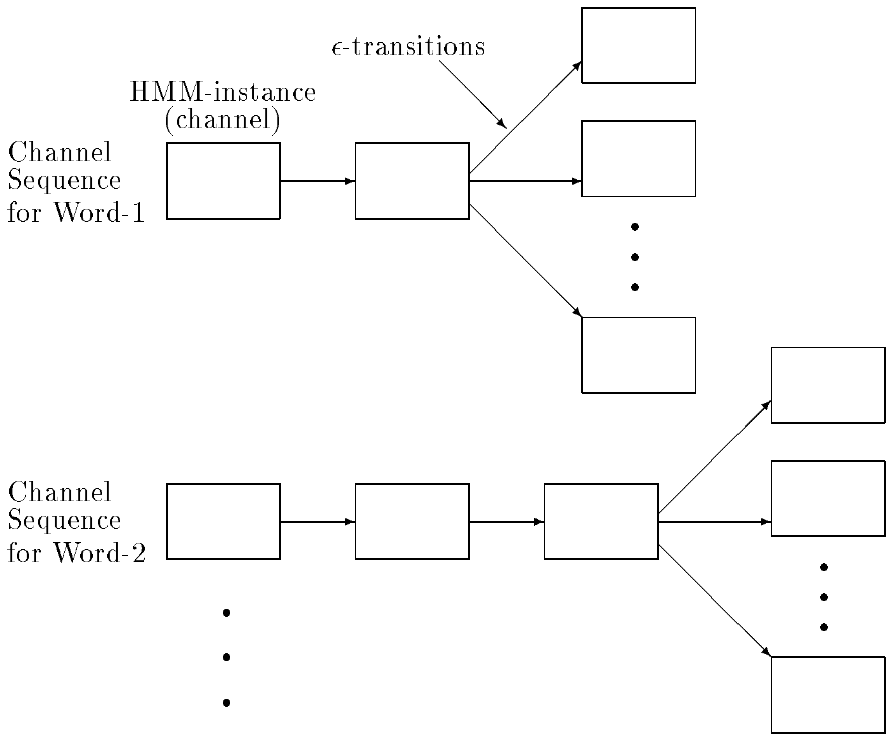

Figure 2: The *WordChannel* Array Data Structure

1. For each state S in each word, the following is computed:

$$P(S,n) = max(P(R,n-1)P_i(RS)P_a(RS,n)),$$

   over all predecessor states R of S. RS represents the transition from R to S. (For the sake of simplicity of discussion, assume, for the moment, that ϵ-transitions have been eliminated from HMM sequences for each word, as mentioned earlier.)

2. From the probabilities for the final state(s) of each word, all potential successor words (specifically, the probabilities of the start states of potential successor words) are updated. (This corresponds to the inter-word ϵ-transitions.)

See [8], [7] for details.

Since there are many thousands of channels (over 15000 in our database), it is extremely time-consuming to evaluate every one of them for each frame, which is only 10msec long. Therefore, we reduce the computational load by computing only the most likely or *active* channels in each frame; the rest are pruned out. This is accomplished in several steps, as follows.

1. At the end of each frame, those channels that end up with a relatively high probability[7] or *score* are retained for further evaluation in the next frame[8].

[7]Note that channels are instances of HMMs, and contain several states. We are, therefore, talking loosely when we refer to the probability of a channel; it actually pertains to the best or highest probability state in that channel.

[8]The range of allowable probabilities, relative to the highest probability attained by any channel in a

2. If a channel has an acceptable probability (i.e., score within the beam-width) in its final state, its successor channels in the *WordChannel* array within the same word, if any, are also activated for the next frame, and their probabilities updated accordingly. This process is called intra-word channel expansion or simply *intra-word expansion*, and represents the handling of intra-word ϵ-transitions.

3. If a last channel of a word has an acceptable probability in its final state, we have a potential word recognition, whose utterance ends on the current frame. There may be many such words. Of course, only one of them belongs to the most likely word sequence, but we do not yet know which one. At the moment, all of them are candidates for further exploration, called *inter-word expansion*. This step essentially hypothesizes potential new words beginning on the next frame, given the set of candidate words that ended on the current frame. It accomplishes this based on the language model—the *bigram* and *unigram* tables—and activates the first channel of each word hypothesized. This step represents the handling of inter-word ϵ-transitions.

The active channels during each frame are maintained in a variable called *ActiveChannelList*.

The beam search procedure begins with the start state of the distinguished silence-word having probability 1, and all other states at 0. It then processes each frame, performing channel-evaluation for each active channel during that frame and updating the active channel list for the next frame. When the final frame is consumed, we can determine the probability associated with the most likely utterance by looking for the word with the highest probability in one of its final states.

Figure 3 depicts the dynamic programming aspect of this algorithm. It represents a 2-D array of probability values for each state in the *WordChannel* array after each frame. At a given frame, for each state S, the backward (parent) pointers indicate the state in the previous frame which led to the best probability for S. (The pointers in the figure are for our conceptual understanding of the development of various search paths as frames are processed; we have not yet incorporated them into our algorithm.)

Although we know the *probability* of the most likely word sequence at this point, we have no mechanism for tracing the word sequence itself. This requires an additional data structure, which we call the *BackPointer* table. It essentially maintains the backward pointers shown in Figure 3, but at a word-level rather than at the state-level.

Recall that at each frame we could identify a number of potential words recognized at the end of that frame. The *BackPointer* table maintains a list of such candidate words ending at each frame. Furthermore, whenever such a word activates a new successor word (by an inter-word expansion) in the *WordChannel* array, the start state of the successor word is initialized with a *parent* pointer to the parent word's *BackPointer* table entry. During channel evaluation and intra-word expansion in succeeding frames, the *parent* pointer is propagated along any path in the *WordChannel* array that leads successfully from that start state, until the end of the word is reached. Thus, by observing the *parent* pointer associated with any state in the *WordChannel* array, we can identify (from the *BackPointer* table) the previous word that led to this point. When a potential word recognition occurs, not only is the word entered into the *BackPointer* table, but the *parent* pointer of the final state of the word is also recorded in that entry. Thus, for any word in the *BackPointer*

frame, is called the *beam-width* (hence the name Fast Beam Search). We shall not be concerned with other aspects of the beam-width (such as the mechanics of obtaining an optimal value for it) beyond noting its existence.

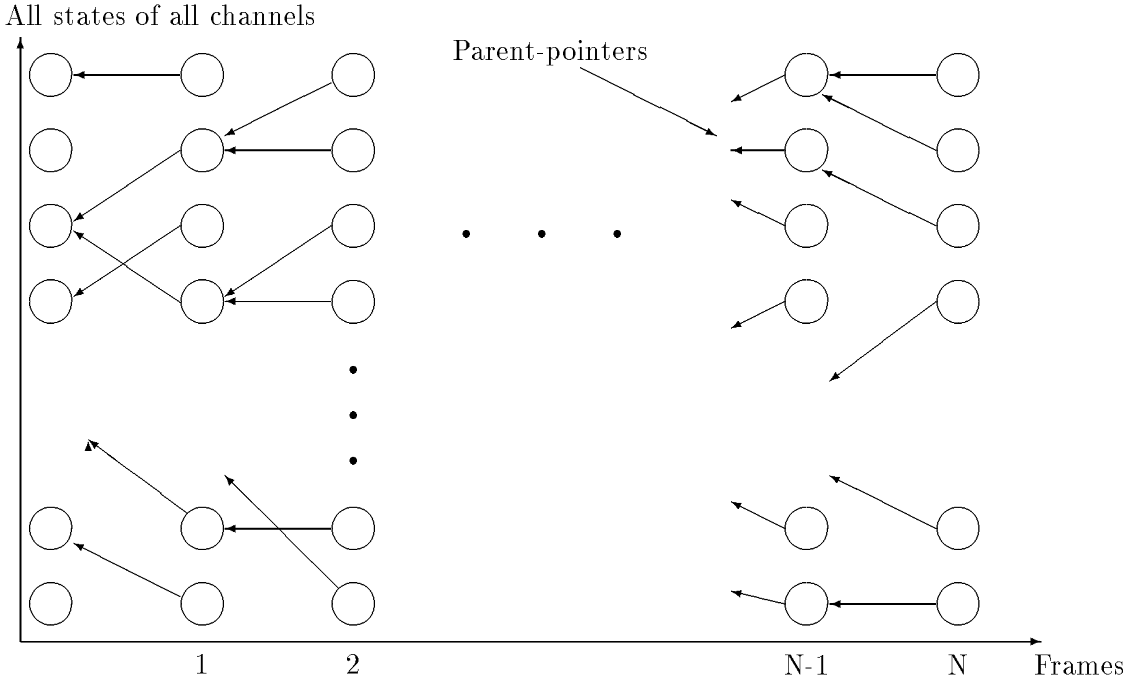

Figure 3: Dynamic Programming Aspect of the FBS Algorithm

table, one can trace the best sequence of words that led to it (in reverse), by following its *parent* pointer entries. Finding the best possible word sequence after all the frames are processed is, therefore, straightforward.

Putting it all together, the overall FBS algorithm is captured by the following pseudo-code:

```
ActiveChannelList = first channel of the special silence-word;
for (each frame) {
    /*** Evaluate active channels ***/
    for (each channel C in ActiveChannelList) {
        for (each state S of C)
            evaluate best P(S), as described earlier;
        prob(C) = max(P(S)), over all states S of C;
    }
    bestprob = max(prob(C)) over all channels C;

    /*** Identify potential words recognized on current frame, and
         update ActiveChannelList for the next frame ***/
    NextActiveChannelList = empty;
    for (each channel C in ActiveChannelList) {
        /*** prune current ActiveChannelList ***/
        if (prob(C) within beam-width of bestprob)
            add C to NextActiveChannelList;

        /*** Intra-word expansion (activate successor channels) ***/
        if (prob(final state of C) within beam-width of bestprob)
            activate successor channels to C, add to NextActiveChannelList;
```

```
        /*** Identify potential word recognitions ***/
        if (C is the last channel in a word W, and
          prob(final state of C) is within beam-width of bestprob) {
            add W (and its parent-pointer value) to BackPointer table;
        }
    }

    /*** Inter-word expansion (based on bigrams) ***/
    for (each word W added to BackPointer table in current frame) {
        for (each successor word W1 to W in the bigram table)
            /* bigram-prob(W-W1) is the probability of word pair <W,W1>,
               obtained from the bigram table */
            if (bigram-prob(W-W1) * prob(W) is within beam width) {
                activate the first channel of W1 (with parent pointer to
                    BackPointer table entry for W),
                adding it to NextActiveChannelList;
            }
    }
    /*** Inter-word expansion (based on unigrams) ***/
    identify the best word W added to BackPointer table in current frame;
    if (at least one word was successfully recognized on this frame) {
        for (each word W1 in the unigram table) {
            activate the first channel of W1 (with parent pointer to
                BackPointer table entry for W),
            adding it to NextActiveChannelList;
        }
    }
    ActiveChannelList = NextActiveChannelList;
}
find best word ended on the final frame and trace back through
    BackPointer table to extract the most likely word sequence;
```

Some of the details, such as propagation of parent pointers, have been omitted from the above code. Also, note that if intra-word or inter-word expansion activates an already active channel, it is effective only if the new probability of the start state of the channel is better than its existing probability. Likewise, if the same word is attempted to be entered into the *BackPointer* table many times during a given frame, only that entry with the best score is entered.

We have already seen the need for maintaining multiple channels at the end of each word in the *WordChannel* array (Figure 2). Likewise, the first channel of each word has a left-context that depends on the *previous* word in the spoken (i.e., recognized) utterance. However, there is no explicit fanout as in the case of the last triphone, since we are only interested in the *one* left-context giving the highest probability for the first channel of the new word. On the other hand, the parent HMM for the first channel of any given word may vary from frame to frame, depending on which previous word (left-context) leads to the highest score for the start state of the channel (after inter-word expansion). Thus, the first channel for each word must also keep track of its parent HMM for each state from frame to frame.

2.4 More on FBS Implementation

We now look at some optimizations of the computation described in the previous section in greater detail, beginning with the computation of the probability of each state in the *WordChannel* array.

As defined above in Section 2.3, the computation of the probability of a state S at the end of frame n, $P(S,n)$, is given by: $P(S,n) = max(P(R,n-1)P_i(RS)P_a(RS,n))$, over all predecessors R of S, where, $P_i(RS)$ is the intrinsic probability, and $P_a(RS,n)$ the acoustic probability of transition RS for frame n.

However, there are a couple of optimizations. First, the acoustic probability P_a of a transition depends on the underlying HMM. There may be several active instances (i.e., channels) of a given parent HMM during the frame; one need not calculate all the P_a's for each one again and again. And, second, the acoustic probability distribution tables for all the HMMs are not unique; the same table may be common to several transitions of a number of HMMs, or even within a single HMM. One need only compute P_a once for each unique transition type. Generally, HMMs for triphone models derived from the same context-independent base phone share common distribution tables. All the unique distributions are maintained separately, and the state transitions simply point to the appropriate ones. There are 4740 distinct transition types in our speech database; each type has its own set of probability distribution tables, one table for each feature type in the speech input.

As a result of these optimizations, channel evaluation in the n-th frame can be broken up into three separate steps, *Computing Acoustic Scores* for individual transition types, *Model Evaluation*, and *Channel Evaluation*:

```
/*** Compute Acoustic Scores for each transition type ***/
for (each unique HMM transition type)
    compute its acoustic score based on the current features values;

/*** Evaluate HMM transition probabilities ***/
for (each triphone model H)
    if (ActiveChannelList contains a channel derived from H)
        for (each transition T of H)
            prob(T) = intrinsic prob(T) * acoustic score(T);

/*** Evaluate Channels in ActiveChannelList ***/
for (each channel C in ActiveChannelList)
    for (each state S in C) {
        P(S,n) = max(P(R,n-1) * prob(RS))
            for all predecessor states R of S;
        /* where, prob(RS) is the probability of transition RS
           computed above for the underlying HMM */
    }
```

The details of computing the acoustic score for each transition type have been omitted; the goal here is to obtain a single scalar score value via a straightforward combination of the acoustic probabilities of the 4 different feature values for the frame. The computation is identical for each of the 4740 transition types.

2.5 Sequential FBS Statistics

Table 1 summarizes the relative time spent in the various steps of the sequential FBS algorithm, obtained by averaging its performance on 180 different spoken sentences from

Table 1: Relative Computation Times

Computation Step	Relative Time(%)	Std.Dev
Computing Acoustic Scores for HMM Transition Types	56.05	3.7
Model Evaluation	13.23	0.7
Channel Evaluation	15.56	1.6
Updating ActiveChannelList & BackPointer table	15.08	1.5

the 1000 word vocabulary. The standard deviations are also included. Each sentence ranges from about 1.5sec to 6sec (about 150–600 frames).

We can see that over 99.9% of the overall computation time is accounted for by these steps (and the variance is extremely small). Furthermore, preliminary statistics from handling a vocabulary system of about 5000 words indicate that the cost of acoustic scores computation drops somewhat with larger vocabularies, and the other three steps increase correspondingly. At the same time, the recognition time increases to about 40 times real-time on a DECStation 5000 (from about 15 for the 1000-word vocabulary). Clearly, one must look for substantial amounts of parallelism to approach real-time recognition speed. Equally clearly, none of the steps can be ignored in a parallel implementation as being insignificant.

From Section 2.4, one can see that each of the acoustic scores can be computed independently, and in parallel. Likewise, once the acoustic scores are available, each HMM can be evaluated independently and in parallel. And, subsequently, so can the active channels. With the current 1000-word database, one has to evaluate all 4740 acoustic scores, and, on the average, about 500-800 active HMMs and 1000-1500 active channels during each frame. Hence, there obviously is a significant amount of inherent parallelism in these steps of the FBS algorithm.

In Section 3.1.4 below, we shall show that the additional step of pruning and computing the active channel list for the following frame can also be parallelized substantially. Finally, it is also possible to identify the list of candidate words recognized upon each frame in parallel. However, in practice, there are very few such occurrences per frame—about 0–15 words per frame. Hence, it is justifiable to maintain a single, centralized *BackPointer* table, and update it sequentially.

The granularity of each concurrently executable task, however, is quite small. Table 2 summarizes the time spent in computing a single acoustic score (for a single HMM transition type), evaluating one HMM, evaluating one channel, and executing the inter-word-expansion task for one bigram entry and one unigram entry. (Furthermore, these times are more or less independent of the speech database; i.e., the vocabulary.) The small granularity indicates that one must exercise care to avoid overhead in distributing these tasks to individual processors.

Table 2: Granularity of Individual Tasks

Computation Step	Time(usec)
Acoustic Score	16
Model Evaluation	12
Channel Evaluation	18
Word-expansion (Bigram)	7
Word-Expansion (Unigram)	12

3 Parallel FBS

We now describe our multi-threaded implementation of parallel-FBS. In parallelizing FBS, there are two broad options: dynamic load balancing using a central task queue model, or static load balancing based on some suitable static data partitioning. The former presumably achieves the best (most even) load balance among all threads, at the cost of contention for the single task queue and a potentially high memory interconnection bandwidth requirement for access to physically distributed data whose locations are not predictable. For example, a look at Table 2 shows that the granularity of each individual task that can be computed in parallel is very small. Thus, contention for a central task queue is likely to be high.

A preliminary study ([4]) indicated that FBS inherently has a very high memory bandwidth requirement, resulting from the amount of computation and data access needed in each 10msec frame. The memory bandwidth could be met by employing multiple memory modules (as in many state-of-the-art multiprocessors). However, it was felt that if the memory accesses were unpredictably distributed among the modules, the resulting communication load would swamp any interconnection network. Therefore, we chose to concentrate on a simpler, static data partitioning approach, which also allows the computation to be distributed such that the need for data movement between memory modules is minimized.

We have implemented a multi-threaded version of FBS on the PLUS multiprocessor and a few variations on a thread simulator. In retrospect, we find that a judiciously chosen strategy of static partitioning results in a reasonably even load balance. Furthermore, the partitioning scheme, to be discussed below, imposes only minimal communication between the threads. Thus, it should be possible to map this implementation, with minor modifications, even onto fast message-passing machines (i.e., parallel-processors without shared-memory). If one does wish to explore the benefits of a dynamic load balancing scheme on shared-memory machines such as the Stanford DASH-1 ([1], [2]), or a multiprocessor DEC Alpha system, it may be worthwhile to re-examine some of the partitioning and mapping strategies described in this document. Even on such NUMA architecture machines with hardware-based caching, however, it seems intuitively apparent that static partitioning can improve performance by minimizing cache invalidations and misses.

The basic strategy is to partition the triphone HMMs among a fixed number of threads statically and as uniformly as possible, and to partition the computation among threads accordingly. For example, each thread is responsible for evaluating the models assigned to it, and evaluating the channels derived from those models. There is no attempt to balance the computational load among threads by dynamically moving pending computation to an idle thread. The parallelization is discussed in detail in the following section.

3.1 Mapping Data and Computation to Threads

Some of the main data structures that we have seen in sequential FBS are: the HMM array, the *WordChannel* array, the acoustic transition probability distribution tables (each of which may be shared among multiple HMMs), the acoustic *scores* array (see Section 2.4), and the *BackPointer* table. In addition, there is also the *ActiveChannelList* array, the language model, consisting of *bigram* and *unigram* tables, and the language dictionary, consisting of words and their constituent phone and triphone lists. Some of the above are read-only data structures, and can be fully replicated among all threads, if necessary. Others are read-write structures, and are typically partitioned among threads. (On PLUS, however, they can also be replicated, with hardware maintaining consistency among all copies.) A partitioned assignment of data structures to threads implies a corresponding partitioned placement on physical nodes hosting the threads (assuming threads do not migrate from one node to another).

In the following sections we go into the details of the distribution of data structures and each computational step among threads. (To facilitate the discussion, we consider the steps in a different order than their appearance in the computation.) During this discussion, we shall often need to refer to that partition or subset of a data structure that is local to a given thread. We shall do this by prefixing the data structure name with the keyword "*thread*". For example, within a code fragment, *thread.ActiveChannelList* refers to the subset of *ActiveChannelList* that is local to the thread executing that code.

3.1.1 Model Evaluation

Triphone models and channels can be statically partitioned among threads in many ways. However, the following two criteria virtually dictate a preferred partition.

First, the static partition must achieve an even load balance across the threads. It has been found, in practice, that only a few tens of words are actively searched in any given frame. That is, the channels (and hence the parent HMMs) of very few words are active, the rest having been pruned out by the beam-search algorithm. As described in Section 2.3, the majority of the channels in the *WordChannel* array are at the end of each word, owing to the right-context fanout. It is, therefore, important to spread such channels (and their parent HMMs) evenly among the threads.

Second, notice from Section 2.4 that channel evaluation requires transition probabilities computed for the parent HMMs. It is desirable to minimize communication between threads by evaluating channels on the same threads as their parent HMMs. However, the parent HMMs of the first channel of each word are not known *a priori*. As described in Section 2.3, they can have different parent HMMs in different frames, depending on the possible left-context words at each frame. Therefore, in a static partition it is necessary to place such HMMs and derivative channels on the same thread, in order to avoid inter-thread traffic.

Taken together, these two observations lead to the following partition for triphone HMMs: For each context-independent or *base* phone, its derivative triphones are distributed among threads by the right-context, but aggregated on the same thread by the left-context. That is, for a given base phone p, all triphones $p_l p p_r$ with left context phone p_l and right context phone p_r are partitioned into separate groups based on pp_r, and each group is assigned to an individual thread. The total number of groups are evenly spread among threads, as far as possible.

Accordingly, the model evaluation step is also statically partitioned, with each thread responsible for only those HMMs assigned to it. Of course, only the active HMMs within each group need be evaluated. Note that during each frame the model evaluation step requires the result of the previous step: computation of acoustic score for each HMM transition type. This is treated in Section 3.1.3 below.

The following pseudo-code fragment captures the computation performed by each thread during model evaluation. Note that each thread must know which HMMs are local to it; this can be ascertained statically, during initialization.

```
/*** Evaluate HMM transition probabilities, in parallel ***/
for (each triphone model H local to this thread)
    if (thread.ActiveChannelList contains a channel derived from H) {
        for (each transition T in H)
            prob(T) = intrinsic prob(T) * acoustic score(T) in this frame;
    }
```

3.1.2 Channel Evaluation

The *WordChannel* array is partitioned simply by assigning each channel to the same thread that contains the channel's parent HMM (or group of parent HMMs, in the case of the first channel of a word). Each thread also maintains a local list of active channels during each frame; i.e., a partitioned version of *ActiveChannelList*, designated *thread.ActiveChannelList* according to our convention. During the channel evaluation step, each thread evaluates only those channels in its private *thread.ActiveChannelList*. Although channel evaluation requires the results of the model evaluation step, they are available locally; inter-thread communication is not necessary. However, during channel evaluation we also need to compute the highest probability among all channels, so that we can later prune the active channel list for the next frame. This requires a small amount of communication and synchronization among threads.

Each thread executes the following pseudo-code during channel evaluation (the initialization of variable *global_bestprob* during each frame is not shown):

```
/*** Evaluate Channels in thread.ActiveChannelList, in parallel ***/
for (each channel C in thread.ActiveChannelList) {
    for (each state S in channel C) {
        P(S,n) = max(P(R,n-1) * prob(RS)) for all
            predecessor states R of S;  /* n = current frame no. */
        /* where, prob(RS) is the probability of transition RS
           computed for the underlying HMM */
    }
    /* compute best channel prob. within thread.ActiveChannelList */
    C.bestprob = max(probability of all states in C);
}
thread.bestprob = max(C.bestprob) over channels C in thread.ActiveChannelList;

/* compute global best channel probability */
LOCK global_bestprob;
global_bestprob = max(global_bestprob, thread.bestprob);
UNLOCK global_bestprob;
```

3.1.3 Acoustic Scores Computation

As outlined in Section 2.4, there are 4740 distinct types of transitions in the database. Each type is distinguished by a unique set of probability distributions, one distribution for each feature type in the speech input. During each frame, we have to compute a *score* for each transition type, by combining the probabilities of the 4 feature values in the current frame. The score is subsequently used to evaluate the HMM transition and channel probabilities.

The score computation is identical and independent for each transition type, and can, therefore, be perfectly partitioned. The probability distribution tables for the 4740 transition types are partitioned evenly across the threads, in any arbitrary order, and each thread computes the scores for its local set. For this, the speech input signal feature values must be broadcast to all the threads during each frame. However, this is a minuscule volume of communication.

The resulting 4740 scores values are used during the model evaluation step. Since a particular transition type is shared between any number of triphone models derived from a single base or context-independent phone, the corresponding score computed for that transition type is needed in all such models. And, since we distribute the triphones for each base phone among all the threads, each score is potentially read by more than one thread during model evaluation. Thus, each score value must be multicast to some set of threads before model evaluation.

In the implementation on PLUS, the entire scores array is simply broadcast to *all* the threads during each frame. This is accomplished by *replicating* the acoustic scores array on all threads[9]. The PLUS coherency protocol automatically updates all copies whenever the data structure is written, thus effecting the broadcast. In machines that support only message-passing, explicit broadcast or multicast would be necessary.

The following pseudo-code is executed by each thread in computing the acoustic scores in each frame (details of the actual computation have been omitted; they involve combining the probabilities of the observed features values into a single scalar quantity, the *acoustic score*):

```
/*** Evaluate Acoustic Scores, in parallel ***/
read (or receive) feature values in current frame;
for (each unique HMM transition type assigned to this thread) {
    compute its acoustic score based on the current features values;
    broadcast this score to all other threads;
}
```

This is the sole step in our parallel decomposition scheme that requires a significant volume of data communication between threads in each frame. With the current database, 4740 words (acoustic scores) are broadcast to each node in each 10msec frame, implying a minimum bandwidth requirement of about 1.9 MBytes/sec per node. While this is by no means infeasible, larger vocabulary systems will necessarily increase the load. It may become imperative to limit the bandwidth requirement by restricting the placement of triphones on threads in some ways, which was the source of the problem in the first place. We shall see the effect of this in the case of the PLUS implementation in Section 4.

[9] More accurately, on all nodes which host at least one thread.

3.1.4 Next Active Channel List Creation

In this section we discuss the parallelization of the remaining steps of computation in each frame; i.e., computing *thread.ActiveChannelList* for the next frame, and identifying the candidate words recognized on each frame, together with updating the *BackPointer* table. The former is accomplished in several stages: by pruning the *current thread.ActiveChannelList*, by intra-word expansion, and by inter-word expansion (see Section 2.3). The potential word recognitions at the end of the current frame are identified by scanning the current set of *thread.ActiveChannelLists*, and the *BackPointer* table is simultaneously updated.

The *BackPointer* table cannot be partitioned among threads in any obvious manner, since any of its entries (in the current frame) may be read by any thread during inter-word expansion (see Section 2.3). However, there are very few updates to the table during each frame (see Section 2.5). This allows us to maintain a single, central copy of the *BackPointer* table, without its becoming a serialization bottleneck and without incurring excessive inter-thread communication. Updates to the table require locking and serialization, and new entries to the table must eventually be broadcast to all the threads before inter-word expansion.

We first turn to channel pruning and intra-word expansion. Clearly, given the best channel probability in the current frame and the beam width, each thread can prune its local *ActiveChannelList* independently and in parallel. However, a thread may need to activate, for the next frame, new channels that reside on other threads. This requires inter-thread communication. In our implementation, each thread maintains a request queue *ActivateReqQueue*, into which other threads can enter channel activation requests, along with relevant information such as the start state probability and the *parent* pointer into the *BackPointer* table. Each queue only holds requests for channels local to the corresponding thread. (Therefore, the channel placement for the entire *WordChannel* array must be made known to all threads during initialization).

The following pseudo-code fragment, executed by each thread, summarizes the above steps—channel pruning, intra-word expansion and identification of potential word recognitions (including *BackPointer* table update):

```
read global_bestprob; /* computed during channel evaluation */
thread.NextActiveChannelList = empty;
for (each channel C in thread.ActiveChannelList) {
    /*** prune currently active channels for next frame ***/
    if (prob(C) within beam-width of global_bestprob)
        activate C and add to thread.NextActiveChannelList;

    /*** do intra-word expansion (activate successor channels) ***/
    if (prob(final state of C) is within beam-width of global_bestprob) {
        for (each successor channel C1 of C in the WordChannel array) {
            if (C1 is assigned to this thread)
                activate C1 and add to thread.NextActiveChannelList;
            else /* C1 assigned to some remote thread RmtThread */
                enqueue C1 on ActivateReqQueue of RmtThread, together
                    with new probability for C1 and parent pointer;
        }
    }

    /*** identify candidate words recognized, and
         update BackPointer table ***/
```

```
    if (C is the last channel in a word W, and
      prob(final state of C) is within beam-width of bestprob) {
        LOCK BackPointer table;
        add W (and its parent pointer) to BackPointer table;
        UNLOCK BackPointer table;
    }
}
```

After the above steps, and before inter-word expansion, the channel activation requests enqueued in the *ActivateReqQueue* structures are processed by each thread, as follows:

```
for (each entry in thread.ActivateReqQueue) {
    activate the channel specified by this entry,
        and add to thread.NextActiveChannelList;
}
clear thread.ActivateReqQueue;
```

Inter-word expansion can be parallelized quite easily, with minimal inter-thread communication. This step is invoked only if at least one word was recognized with an acceptable probability at the end of the current frame, and it consists of the potential activation of the first channel of each word in the dictionary (see Section 2.3). The home thread of each such channel is known statically during initialization, (see Sections 3.1.1 and 3.1.2). Therefore, the words themselves can be statically partitioned among threads for inter-word expansion. Specifically, this requires the *bigram* and *unigram* tables to be partitioned into several *thread.bigram* and *thread.unigram* subtables, one for each thread. For each thread, *thread.bigram* consists of a list of successor words for each word in the dictionary; exactly those successor words are included whose first channel has been assigned to that thread. The *unigram* table is partitioned on the same basis. With such a partition, each thread can, independently and in parallel, carry out inter-word expansion based on its local subset of bigrams and unigrams.

Multi-threaded inter-word expansion, based on bigrams, is shown by the following pseudo-code:

```
/*** Inter-word expansion (based on bigrams) ***/
receive updates to BackPointer table during current frame;
for (each word W added to BackPointer table in current frame) {
    for (each successor word W1 to W in the thread.bigram table)
        if (bigram-prob(W-W1) * prob(W) is within beam width) {
            activate the first channel of W1 (with parent pointer to
                BackPointer table entry for W),
            adding it to thread.NextActiveChannelList;
        }
}
/* the following is needed for inter-word expansion based on unigrams;
   the details are somewhat complicated and have been omitted */
identify the best word W added to BackPointer table in current frame;
```

Finally, we perform inter-word expansion based on unigrams; a straightforward parallelization of the sequential version:

```
/*** Inter-word expansion (based on unigrams) ***/
if (at least one word was successfully recognized on this frame) {
```

```
    for (each word W1 in the thread.unigram table) {
        /* Note: W = best word added to BackPointer table in
           this frame */
        activate the first channel of W1 (with parent pointer to
            BackPointer table entry for W),
        and add W1 to thread.NextActiveChannelList;
    }
}
```

3.2 Putting It All Together

Parallel-FBS begins by partitioning HMMs, channels, acoustic probability distribution tables, and unigram and bigram tables among threads, as discussed in the earlier sections. For example, it creates a local list of HMMs for each thread, creates *thread.ActiveChannelList*, *thread.NextActiveChannelList* tables, and creates *thread.bigram* and *thread.unigram* subtables. It also creates various locks for mutual exclusion, e.g., for updating the *BackPointer* table, as well as initializes barrier variables (see below).

The overall structure of parallel-FBS in the core, search computation is very similar to that of the sequential FBS, and is shown below. Each thread executes this code:

```
for (each speech input frame) {
    receive speech signal feature values for current frame;

    compute acoustic scores for the local HMM transition types,
        and broadcast results to all threads;
    BARRIER;

    evaluate transition probabilities of (active) local HMMs;
    evaluate channel probabilities of channels in
        thread.ActiveChannelList, and find the best channel probability;
    BARRIER;

    prune active channels, perform intra-word expansion,
        and update BackPointer table for current frame;
    BARRIER;

    process inter-thread channel activation requests
        queued during the intra-word expansion step;

    perform inter-word expansion based on bigrams;
    BARRIER;
    perform inter-word expansion based on unigrams;

    thread.ActiveChannelList = thread.NextActiveChannelList;
}
```

4 Performance Evaluation

The parallelization scheme described in the earlier section was implemented and tested using the *C-threads* facility ([6]) and several measurements were obtained, which are summarized in this section. The experiments were carried out with the number of threads varying between 1 and 5 for each of 180 different spoken utterances from a 1000-word vocabulary.

Table 3: Computational Load Per Thread Per Frame

No. of threads:	1	2	3	4	5
Models Evaluated	584	291	194	145	115
Channels Evaluated	1150	575	383	286	229
Remote Channel Activation Requests	0	58	52	43	37
Inter-word Expansions (based on bigrams)	510	254	169	126	101
Inter-word Expansions (based on unigrams)	799	399	265	199	159

We first examine the actual load distribution achieved by the static partitioning scheme. Table 3 lists the load per thread for each major computational step. The load is measured in terms of the number of HMMs (or channels, bigrams, etc.) evaluated per thread per frame, and is averaged over all frames of all 180 utterances. The number of channels evaluated is the same as the number examined during pruning, recognition of words ended on the current frame, and intra-word expansion. Hence, figures for the latter are not shown explicitly. Furthermore, the acoustic scores computation step is perfectly balanced among the threads and is also omitted from the table.

We should stress that the table shows figures averaged over several hundreds of frames per speech utterance. If the load distribution within individual frames were highly uneven, we would be in trouble, owing to the barrier synchronizations within each frame. This factor is considered later in this section. For now, we note that measurements show that the load distribution within individual frames is not skewed too significantly. (Furthermore, even the average load distribution is itself of interest, since it may be possible to avoid strict barrier synchronization in each frame, and resort to some speculative execution in future implementations.)

To get a clearer picture of the load distribution, the above table is reproduced in Table 4, where the load per thread is shown as a percentage of the corresponding load when there is only one thread. We also show the load on the most and least heavily loaded thread (the *Max* and *Min* values, respectively) in each case.

As we can expect, there is no increase in the overall load with the number of threads. The max. and min. values indicate the extent of unevenness in the average load per thread. There does not seem to be a straightforward correlation between this and the number of threads. For example, there is more unevenness with 4 threads than with either 3 or 5. The inter-word expansion steps exhibit the greatest load unevenness among threads. We should note that the step of processing remote, inter-thread channel activation requests (see Section 3.1.4) has been omitted from the table. The reason is that the table lists load per thread as a percentage of the load in the single-thread case; however, the number of inter-thread channel activation requests with only a single thread is 0! We note that the time spent in this step is quite small, as we can predict from Table 3.

In summary, the load distribution exhibited by our partitioning scheme appears to be largely even for each of the computational steps (for the number of threads we have experimented with). Variations with increasing number of threads, however, appear to be

Table 4: Computational Load Per Thread Per Frame (Percentage of Load on a Single Thread)

No. of threads:		2	3	4	5
Models Evaluated	(Avg)	49.9	33.2	24.9	19.9
	(Max)	50.2	33.7	26.5	20.6
	(Min)	49.6	32.7	23.6	19.0
Channels Evaluated	(Avg)	49.9	33.3	24.9	19.9
	(Max)	53.1	34.1	30.0	21.7
	(Min)	46.8	32.5	20.3	18.3
Inter-word Expansions	(Avg)	49.9	33.2	24.9	19.8
(based on bigrams)	(Max)	54.6	33.8	31.7	23.6
	(Min)	45.2	32.9	20.4	16.9
Inter-word Expansions	(Avg)	50.0	33.2	24.9	19.9
(based on unigrams)	(Max)	55.2	34.2	32.8	22.2
	(Min)	44.8	31.9	18.2	18.5

somewhat unpredictable.

A multi-threaded implementation of parallel-FBS has been carried out on the PLUS multiprocessor, from which actual computation times (wall-clock times) are available for each thread, for each step of the algorithm. (Each thread was placed on a separate processor.) These figures give us an idea of the actual speedup possible in each computational component of the parallel-FBS algorithm. For example, even though computing the acoustic scores is perfectly parallelizable, the actual measured times show less than linear speedup with increasing number of threads. This is owing to the need to broadcast the computed acoustic scores to all threads (as described in Section 3.1.3), which increases the load on the memory system.

Table 5 shows the relative computation time on PLUS per thread per frame, for each step of the algorithm. These are, once again, percentage values relative to the single-thread case. Also, they are specific to the PLUS system; other machines would probably exhibit somewhat different characteristics.

Clearly, only two steps in Table 5 show less than linear speedup with number of threads. Computing acoustic scores, as we have just described, requires propagating the computed values to each thread, incurring extra memory, bus, and network overhead on the PLUS multiprocessor. Thus, there is loss of efficiency even though the computation can be distributed perfectly evenly among threads. It is something to be concerned about, especially if more concurrent threads are employed, since this step accounts for over 50% of the total computation time (see Table 1, and further discussion below).

The other step in Table 5 with sub-linear speedup—channel pruning, etc.—requires mutual exclusion for updating common data structures, as well as non-local access to data assigned to remote threads. Specifically, accesses to the global *BackPointer* table and the channel activation request queues on other threads (see Section 3.1.4) have to be serialized; the latter also requires accesses to the queues which are remote data structures. Both these factors combine to reduce per-thread efficiency. However, it is less of a concern since this step accounts for a relatively small fraction of the overall computation time—about 5%.

Table 5: Computational Time Per Thread Per Frame (Percentage of Time for the Single Thread Case)

No. of threads:	2	3	4	5
Acoustic Scores Computation	55.0	38.8	30.5	25.8
Model Evaluation	50.4	33.8	25.4	20.4
Channel Evaluation	50.0	33.3	24.9	19.9
Channel Pruning, Intra-word Expansion, & Backpointer Table Update	66.6	49.6	39.7	33.7
Inter-word Expansion (based on bigrams)	50.2	33.3	24.9	20.0
Inter-word Expansion (based on unigrams)	50.1	33.5	25.5	20.0

As mentioned earlier, we note that all this while we have only looked at the *average* load per frame in evaluating load-balance. However, a speech utterance consists of several hundreds of frames with barrier synchronizations within each frame. We could end up with highly skewed load per thread when we observe each individual frame in isolation, although the average load over the entire utterance appears to be evenly distributed. This would nullify all our efforts, since the barrier operations cause all threads to progress at the rate of the slowest during each frame.

Measurements have shown that the load distribution per frame, while not as even as the average indicated in the above tables, is not dramatically uneven. We can judge the load balance by looking at the computation time per thread for each of the major steps as a fraction of the *total* execution time (i.e., the wall-clock time from start to finish, including the time taken to execute the barrier synchronizations etc.). In other words, we are interested in thread utilization figures—the fraction of total elapsed time that is spent usefully in each of our computational steps. The rest is overhead in carrying out the barrier synchronizations as well as caused by load imbalance (which shows up as wasted time waiting at barriers).

Table 6 summarizes this information for the PLUS implementation, averaged over all utterances. It shows that the thread utilization remains well above 90% for all thread configurations. There is a gradual decline in thread utilization with increasing number of threads, but it is not monotonic.

Finally, Table 7 summarizes the speedup with increasing number of threads in our implementation on PLUS. (The standard deviation over 180 utterances is under 0.05.) Much of the inefficiency seen with larger number of threads is, in fact, attributable to the sub-linear speedup in the acoustic computation step (Table 5), which is actually distributed perfectly evenly. If this step were perfectly scaleable on PLUS, the expected speedup with 5 threads would be at least 4.5, instead of the present 3.85. The residual inefficiency is accounted for by overheads caused by various barrier synchronizations and other serializations, as well as load imbalances within individual frames.

We also experimented with other schemes of static load distribution, none of which approached the performance indicated above. For example, the HMM models were dis-

Table 6: Thread Utilization (Percentage of Total Execution Time)

No. of threads:	1	2	3	4	5	st.dev
Acoustic Scores Computation	56.0	55.0	55.8	53.8	56.0	3.8
Model Evaluation	13.2	11.9	11.4	10.6	10.4	0.6
Channel Evaluation	15.6	13.8	13.3	12.2	11.9	1.4
Channel Pruning, Intra-word Expansion, & Backpointer Table Update	5.2	6.3	6.7	6.6	6.8	1.5
Channel Activation Reqs. From Other Threads	0.0	0.3	0.4	0.4	0.4	0.1
Inter-word Expansion (based on bigrams)	2.4	2.2	2.0	1.9	1.8	0.5
Inter-word Expansion (based on unigrams)	7.4	6.6	6.3	5.9	5.7	0.6
TOTAL	99.8	96.1	95.9	91.4	93.0	

Table 7: Speedup

No. of threads:	2	3	4	5
Speedup over single thread:	1.79	2.56	3.14	3.85

tributed among threads based only on their context-independent base phone (rather than base-phone–right-context-phone combination). All other data structures and computations were partitioned accordingly. The advantage of such a mapping is that there is no need to broadcast acoustic scores data among threads, since there is no sharing of acoustic probability distribution tables among HMMs with different base phones. This, as we noted above, is a major source of loss of performance on PLUS. However, this scheme failed owing to highly non-uniform load distribution; reasons for this have been indicated in Section 3.1.1.

5 Summary

We have described a simple, multi-threaded parallel implementation of the FBS algorithm for speaker-independent, continuous speech recognition. The parallelization is based on a static data and computation partitioning method. The original, sequential algorithm consists of about half-a-dozen steps within a loop body which is repeated for each 10msec frame of the speech input. Each step must be parallelized in order to attain a reasonable speedup. We have shown that this is indeed possible. The parallel FBS system has been simulated and also implemented on the PLUS multiprocessor at Carnegie Mellon University. Measurements have shown that even under static partitioning, the average load distribution is remarkably even across threads. Actual performance on PLUS has shown a speedup of about 3.85 with 5 threads (one thread per processor). Some of the inefficiency is accounted for by the need to broadcast acoustic scores information to all threads in each frame, which imposes additional load on the processor buses and memory network. We believe that if this overhead is eliminated, the speedup can be improved to about 4.5 with 5 threads. However, experiments to determine load balance and performance with much larger numbers of threads remain to be conducted.

A number of other questions remain open. First, how well does static partitioning perform on architectures other than PLUS? Second, our concurrent implementation is entirely hand-crafted; is it possible to automate or semi-automate the process so that a similar parallelization can be carried out on future versions of the sequential FBS algorithm? Third, is it possible to search several frames in parallel, speculatively? This would open up another source of parallelism, and also possibly overcome load imbalances within each frame. Finally, how is performance affected by dynamic load balancing among threads? Dynamic load balancing eliminates uneven load distribution (and hence idle thread times), but incurs greater network, memory and processor bus loads. This is already a problem for the single case of broadcasting acoustic scores, as seen from the PLUS experience.

Acknowledgements

I would like to thank Prof. R. Bisiani and Fil Alleva at Carnegie Mellon University, Pittsburgh, for helping me understand the original FBS algorithm. I would also like to thank Prof. N. Viswanadham and Dr. K. Gopinath of the Indian Institute of Science, Bangalore, India, for providing me with resources to carry out some of the work.

References

[1] Lenoski, D. *et al*, "The Directory-Based Cache Coherence Protocol for the Dash Multiprocessor", *Proc. 17th Intl. Symp. on Comp. Arch.*, May. '90, pp 148-159.

[2] Lenoski, D., "The Design and Analysis of DASH: A Scalable Directory-Based Multiprocessor", *Tech. Rept. No. CSL-TR-92-507*, Computer Systems Laboratory, Stanford University, Feb. '92.

[3] Bahl, L.R. *et al*, "A Maximum-likelihood Approach to Continuous Speech Recognition", *IEEE Trans. Patt. Anal. Mach. Intell.*, vol. PAMI-5, Mar. '83, pp 179-190.

[4] Bisiani, R.B. and Ravishankar, M.K., "Time-Space Characteristics of FBS", Speech-group Internal Document, School of Comp. Sci., Carnegie Mellon University, Jul. '92.

[5] Bisiani, R.B. and Ravishankar, M.K., "PLUS: A Distributed Shared-Memory System", *Proc. 17th Intl. Symp. on Comp. Arch.*, May. '90, pp 115-124.

[6] Cooper, E.C. and Draves, R.P., "C Threads", Tech. Rept., School of Comp. Sci., Carnegie Mellon University, Nov. '90.

[7] Lee, K., "Large-Vocabulary Speaker-Independent Continuous Speech Recognition: The SPHINX System", Comp. Sci. Dept., Carnegie-Mellon University, Apr. '88.

[8] Lee, K., Hon, H., and Reddy, R., "An Overview of the SPHINX Speech Recognition System", *IEEE Trans. on ASSP*, Jan '90, pp. 599-609.