International Journal of Parallel Programming, Vol. 27, No. 4, 1999

Parallel Speech Recognition

Steven Phillips[1] and Anne Rogers[1]

Received March 1998; Accepted January 1999

Computer speech recognition has been very successful in limited domains and for isolated word recognition. However, widespread use of large-vocabulary continuous-speech recognizers is limited by the speed of current recognizers, which cannot reach acceptable error rates while running in real time. This paper shows how to harness shared memory multiprocessors, which are becoming increasingly common, to increase the speed significantly, and therefore the accuracy or vocabulary size, of a speech recognizer. To cover the necessary background, we begin with a tutorial on speech recognition. We then describe the parallelization of an existing high-quality speech recognizer, achieving a speedup of a factor of 3, 5, and 6 on 4-, 8-, and 12-processors respectively for the benchmark North American business news (NAB) recognition task.

KEY WORDS: Speech recognition; shared memory; viterbi search; finite state automata.

1. INTRODUCTION

The success of computer speech recognition is limited, in part, by the speed of current speech recognizers. To operate in real time on continuous speech, recognizers must compromise on some or all of vocabulary size, grammar complexity, and recognition accuracy. Consequently, the motive for using parallelism to do real-time recognition has been evident for some time, but only recently has the opportunity become available to use relatively inexpensive and commercially-available multiprocessors to attack this problem. This paper shows that harnessing the power of shared-memory multiprocessors can increase greatly the speed of a speech recognizer. The increase in recognition speed can be used to expand the set

[1] AT&T Labs, Shannon Laboratory, 180 Park Ave, Florham Park, New Jersey 07932-0971. E-mail: {phillips,amr}@research.att.com.

0885-7458/99/0800-0257$16.00/0

of recognition tasks where real-time recognition is possible, or to increase recognition accuracy on existing real-time tasks.

To be of lasting benefit, a parallel recognizer must be closely tied to an existing high-quality sequential recognizer, so that improvements made in sequential speech recognition (phone modeling, likelihood calculations, grammar representation, etc.) can be applied to the parallel recognizer directly. In order to achieve this modularity, we chose to parallelize the sequential recognition system described by Riley *et al.*[1] We chose this particular system as it is powerful and general-purpose, and is used on a wide range of recognition tasks. This generality is in part due to its simple and elegant structure. Our parallelization has maintained these desirable attributes, while adding the option of greatly increased recognition speed on multiprocessor systems.

In keeping with our desire to maintain close ties with a sequential recognizer, we chose to implement our parallel recognizer on a shared-memory machine, specifically a Silicon Graphics Power Challenge XL. Using message-passing would have required a wholesale rewrite of the underlying recognizer.

Previous applications of parallelism to speech recognition have focused on special-purpose hardware [e.g., Wen and Wang[2]], or on isolated word recognition [e.g., Noda and Shirazi[3]], or in the case of the Viterbi algorithm, only on the calculation of observation likelihoods in hidden Markov models. In the latter case, which is closest in scope to our work, the parallelized subroutines contain only about half the sequential work for many large-vocabulary tasks, including the NAB business news task we used as a benchmark, so the parallel speedup cannot exceed a factor of two. Parallel algorithms exist for generic graph search [e.g., Goudreau *et al.*[4]], but are not directly applicable to large speech recognition tasks, where the graph is defined implicitly.

The conceptual focus of the parallelization is the search algorithm, which traverses a graph that is built on-the-fly. In our implementation, the decomposition of work into parallel tasks, the allocation of tasks, and the synchronization among tasks are all driven by a decomposition of the graph. Careful partitioning of the graph, plus several auxiliary data structures yields a clean software architecture that has good performance. The parallel recognizer achieves real-time performance using eight processors on the ARPA North American business news (NAB) recognition task, and speed-ups of 3.1, 5.2, and 6.2 using 4-, 8-, and 16-processors for the same task using more computationally intensive parameters for the search algorithm.

This paper gives a comprehensive review of sequential speech recognition (Section 2), before describing the parallel recognizer (Section 3). Section 4

analyzes the performance of the parallel recognizer, while Section 5 closes with some directions for future work. A shorter version of this paper appeared in Eurospeech.[5]

2. SEQUENTIAL SPEECH RECOGNITION

We describe here a framework for large-vocabulary recognition of continuous speech, namely a statistical approach based on hidden Markov models, finite state transducers, and Viterbi search. We have aimed for as much generality as possible, to try to capture the basic requirements of a speech recognition system. Where specifics are necessary for later discussion of the parallel recognizer (in Section 3), we describe the two-level search algorithm of Lee and Rabiner,[6] as implemented in the Riley *et al.*[1] recognizer. This recognizer uses a (largely independent) software module for manipulating finite state transducers.[7, 8]

This section describes only the search problem in speech recognition: given models of the various levels of speech information and a speech waveform, find the sentence that best matches the waveform. We describe neither the design of the models, nor the training of parameters in the models. For further reading on these and other topics in speech recognition we refer the reader to the text of Rabiner and Juang[9] and the references therein.

2.1. The Basic Model

Human speech contains structure and information at various levels: the raw analog speech signal is interpreted as containing a sequence of discrete phones, which combine to form words, which combine to form sentences. The human brain contains specialized mechanisms to perform the translation from each level to the next [see for example Pinker[10]]. In the same way, a speech recognition system needs to represent each level and to map between them. Here we describe the levels and mappings in some detail.

Audio input. The speech waveform is digitally sampled. Ten millisecond intervals are evaluated using spectral analysis and other forms of signal processing, resulting in a parametric representation as *frames*, or vectors of real numbers.

Phonemes. A *phoneme* is a member of the set of the smallest units of speech that differentiate utterances in a given language or dialect. The *D* of the English *dog* and the *L* of the English *log* are two different phonemes.

Here and elsewhere we use the ARPABET notation[11] for writing phonemes.

Phones and context-dependent units. A phone is best thought of as an acoustic realization of a phoneme, or a model of a possible manifestation of a phoneme. A recognition system may benefit by making finer distinctions that those used in classifying phonemes, so there are typically somewhat more phones than phonemes, though there is a rough correspondence between the two.

A single phone may be pronounced very differently depending on its context. For example the phone *OW* is pronounced differently in *boat* than in *shoal*, due to the influence of the adjacent phones. For this reason speech recognizers use *context-dependent* models of phones. A context-dependent model represents the pronunciation of a particular phone in a particular context, and we refer to the manifestation of a phone in a particular context as a *context-dependent unit*. In our example, there would be (at least) two different phone models (and corresponding context-dependent units) for the phone *OW*, one per distinct context.

The mapping from audio input to context-dependent units is largely independent of the subject of this paper (parallel search), so only a brief outline is appropriate here.

Figure 1 depicts a three-state *Hidden Markov Model*, or HMM. An HMM is a discrete-time first-order Markov chain. There are probability distributions π_1, π_2, and π_3 corresponding to states s_1, s_2, and s_3 respectively. Each π_i is a probability distribution over the space of all possible frames: if f is a frame, then $\pi_i(f)$ is the likelihood of observing frame f while in state i. Considered as a source, the HMM runs through a sequence of states, generated according to the state transition probabilities. At each state i a frame is generated according to π_i.

Three-state HMMs are useful for modeling phones—one state describes the start of the phone, one the middle, and one the end (though

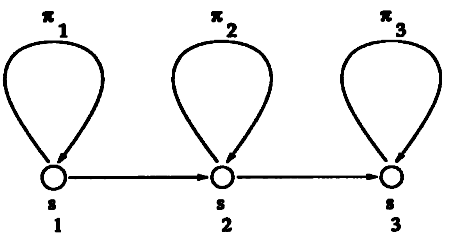

Fig. 1. Example Hidden Markov Model (HMM).

more elaborate HMMs are sometimes used). The probability distributions π_i are typically finite mixtures of Gaussians.

Given a sequence of frames, we cannot determine the state sequence that generated it; instead we must resort to Bayesian reasoning to determine which state sequence is *most likely* to have generated it. Bayesian reasoning is simply the observation that for events p and q,

$$\Pr(p \wedge q) = \Pr(p)\Pr(q \mid p) = \Pr(q)\Pr(p \mid q)$$

and so the conditional probabilities $\Pr(p \mid q)$ and $\Pr(q \mid p)$ satisfy:

$$\Pr(q \mid p) = \Pr(p \mid q)\Pr(q)/\Pr(p)$$

Therefore if we are given p and wish to find the q that maximizes $\Pr(q \mid p)$, we need only maximize $\Pr(p \mid q)\Pr(q)$. In the context of HMMs, say we have observed a sequence $\mathcal{F}$ of frames, and wish to find the state sequence $\mathcal{S}$ that is most likely to have occurred, given $\mathcal{F}$. It suffices that we find the $\mathcal{S}$ that maximizes ($\Pr$(observing $\mathcal{F}$ given $\mathcal{S}$) $\times \Pr(\mathcal{S})$). For example, say an utterance yields the frame sequence $\mathcal{F} = \{f_1, ..., f_4\}$, and we wish to find the sequence of states starting at s_1 and ending at s_3 that best matches $\mathcal{F}$. The possible state sequences and the conditional probabilities $\Pr(\mathcal{F} \mid \mathcal{S})$ shown in Table I.

How do we model the probability $\Pr(\mathcal{S})$ of a state sequence? Different speech recognizers use different methods, for example by recording a probability on the transitions between states, or by using *duration costs*, where each phone has a probability distribution for its duration, or the duration of each of its states. In the underlying recognizer, however, the probabilities of all state sequences are considered equal (only at the HMM level), so we can ignore them. Therefore in our example, the state sequence most likely to have generated the observed frames is just the one with the maximum conditional probability $\Pr(\mathcal{F} \mid \mathcal{S})$ in Table I.

The probability distributions π_i are derived by training using a re-estimation procedure such as the Baum-Welch method, also referred to as the EM (expectation-maximization) method.[12] We will not describe the training process, since it is independent of the recognition process.

Table I

State sequence $\mathcal{S}$	Conditional probability $\Pr(\mathcal{F} \mid \mathcal{S})$
s_1, s_1, s_2, s_3	$\pi_1(f_1) \times \pi_1(f_2) \times \pi_2(f_3) \times \pi_3(f_4)$
s_1, s_2, s_2, s_3	$\pi_1(f_1) \times \pi_2(f_2) \times \pi_2(f_3) \times \pi_3(f_4)$
s_1, s_2, s_3, s_3	$\pi_1(f_1) \times \pi_2(f_2) \times \pi_3(f_3) \times \pi_3(f_4)$

The mapping from context-dependent units to phones is very simple. For example, the sequence {D with right context AA, AA with left context D and right context G, and G with left context AA}translates to the phone sequence {D AA G}, i.e., *dog*.

Words. The collection of pronunciations of all words in the recognizer's vocabulary is known as the *lexicon*. For speaker-independent recognition it helps to allow for multiple pronunciations of some words. For example, the word *data* can be pronounced with the first *a* pronounced as EY (as in *bait*) or as AE (as in *bat*). The alternative pronunciations for *data* can be represented as shown in Fig. 2.

The representation shown here is known as a *weighted finite state transducer*, which we abbreviate as FSM. It defines a mapping from the input alphabet (phones) to the output alphabet (words). This particular FSM maps sequences of phones (corresponding to pronunciations of *data*) into just a single output, the word *data*. An ε symbol denotes that no output symbol is produced on a transition.

Finite state transducers and Hidden Markov Models are really dual representations of the same object. We use HMMs for phone models and FSMs for everything else in this paper, since these are the representations used by the underlying sequential recognizer.

To perform the Viterbi search (described later in Section 2.2) we need to represent the relative likelihoods of the alternative pronunciations of a word. For example, in a population of American English speakers, perhaps half the speakers will pronounce the first *a* of *data* as EY. This can be represented with the weighted FSM (note, the costs in this FSM are made-up) shown in Fig. 2.

The cost assigned to mapping an input sequence to an output sequence is the cost of the minimum path with that label. The cost of a path is determined by the sum of the transition costs, which represent negative log probabilities—a smaller cost corresponds to a higher probability, and summing the costs over a path corresponds to multiplying the probabilities. More formally, the cost is determined using the tropical ({min, + }) semi-ring: the cost of mapping input sequence σ_1 to output sequence σ_2 is the minimum, over paths labeled with σ_1 and σ_2, of the sum

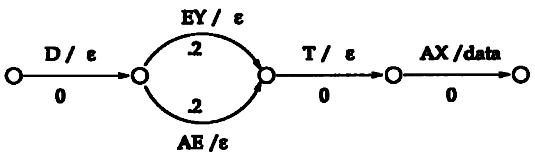

Fig. 2. Alternative pronunciations for *data*.

of the costs on the transitions. As in the case of HMMs, the costs are typically determined by training using a corpus of example pronunciations.

Sentences. A grammar maps sequences of words into sentences. An FSM for a simple word-pair (or bi-gram) grammar is shown in Fig. 3.

States in the word-pair grammar correspond to words. The FSM shown here gives low cost to sentences like *the cat in the hat* (or *the hat in the cat in the hat*), and higher cost to sentences like *in cat the in cat*.

In summary, the major levels of modeling in a speech recognition system are the speech waveform, context-dependent units, phones, words, and sentences. A speech recognizer must be able to map between these levels. The following section describes how this may be done.

2.2. The Search Engine

The task of a speech recognizer is to combine the models at each level of speech representation into a procedure that, given a raw speech waveform, produces a sentence that the speaker is likely to have uttered. The algorithm that is described in this section is the two-level Viterbi search algorithm of Lee and Rabiner,[6] which maps between the context-dependent units and all the higher layers. Mappings between the higher

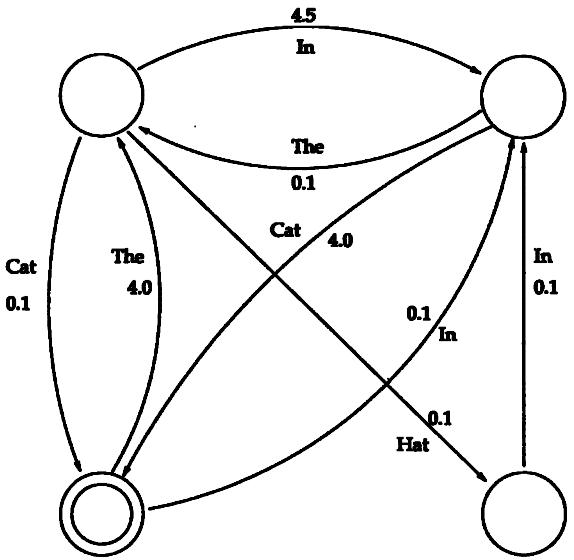

Fig. 3. Example of a simple word-pair grammar.

layers are done using the on-demand FSM composition of Mohri *et al.*,[7] described in the next section.

The levels of speech representation can be partitioned into signal processing levels (audio input and context-dependent units) and combinatorial levels (phones, words, and sentences). This partition produces two layers, which we refer to by the type of representation they use—the HMM layer and the FSM layer. The two-level Viterbi algorithm operates at the boundary between the two layers.

As far as the Viterbi algorithm is concerned, the FSM layer is represented by a single FSM. (How the three levels involved are combined to give this view to the algorithm is the subject of the next section.) An example of this view is shown in Fig. 4. Pictured is an FSM state s_0 with all of its outgoing transitions. Each transition is labeled with an input symbol i_j (which is a context-dependent unit) and a cost c_j, which we call the *FSMCost* for clarity.

The Viterbi algorithm ("the search algorithm") is time-synchronous: it processes each speech frame in turn. It maintains a list of active FSM states, and a list active arcs (each corresponding a collection of related FSM transitions). Each active arc has associated with it a three-state HMM that corresponds to a context dependent unit. For each frame, the Viterbi algorithm queries the HMM layer to determine the likelihood of observing that frame in the HMM states inside the active arcs (this is

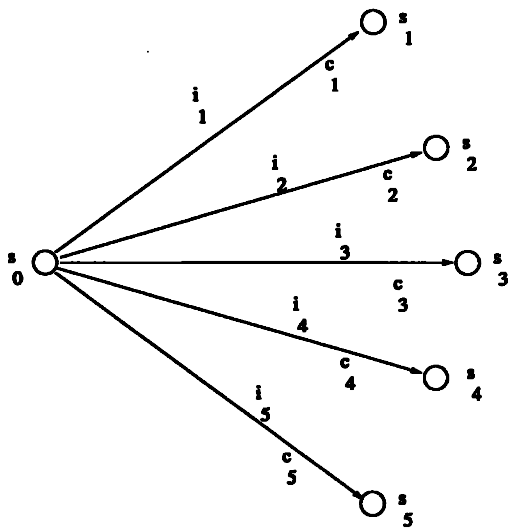

Fig. 4. The transitions emanating from a state of the FSM layer.

called the *likelihood calculation*). These likelihoods are included in the total cost of reaching each HMM state and are then used to determine which states in the FSM layer can be reached at low cost (called the active states). The algorithm then accesses the FSM layer to determine the transitions out of newly active states; these transitions are used to form new active arcs for the next frame. In order to keep the search space small, the algorithm *prunes* states using a *threshold* parameter; during each frame, any state with cost more than the least-cost state plus the threshold is discarded.

The algorithm keeps two lists, an *active state list* and an *active arc list*. The first consists of a subset of the states of the FSM. An active arc corresponds to a *set* of transitions in the FSM, which must all share the same start state and input symbol, but have different end states. For example, if $i_1 = i_2$ in Fig. 4, then the transitions 1 and 2 would share an active arc. (This sharing is an efficiency measure, since some work can be done for both transitions at the same time.) An example active state and active arc are shown in Fig. 5. The *unit cost* of an active arc is just the minimum of the FSMCosts of its corresponding transitions (recorded as an efficiency measure).

For each active state s we keep track of cost(s), the lowest cost path from the start state to s. For each active arc a we keep track of $\text{cost}_1(a) \cdots \text{cost}_3(a)$, the cost of having reached each state of the HMM associated with the active arc. For convenience, we use $\text{cost}_0(a)$ to denote the cost of having reached the source FSM node of a. Initially the active state set consists only of the start state, and the active arc set is empty. For

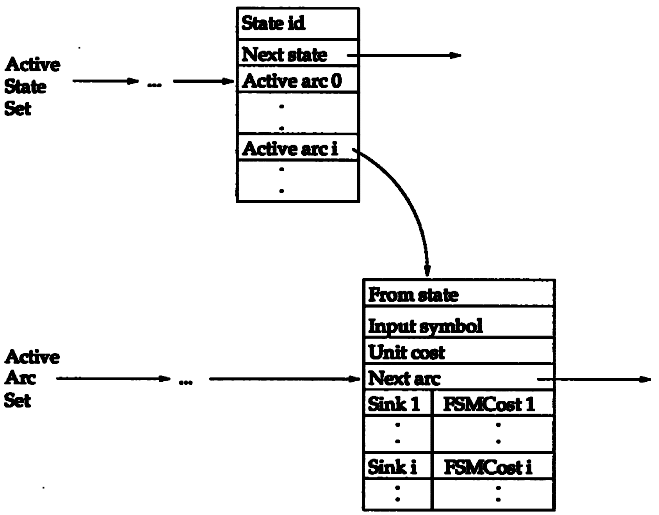

Fig. 5. Active state list and active arc list.

an HMM h, let hmmcost(h, i, current frame) be the cost of observing the current frame in the ith state of h. The basic steps involved in processing a single speech frame are the following.

1. *Update the active arc list using the active state list.* For each active state s and for each input symbol α:
   1. If there is no active arc a corresponding to s and α, create it and add it to the active arc list.
   2. Set $\text{cost}_0(a) = \text{cost}(s)$.
2. *Evaluate active arcs using current frame (likelihood calculation).* Set mincost = ∞. For each active arc a, with corresponding HMM h, do:
   1. For $i = 3$ down to 1:
      (a) set $\text{cost}_i(a) = \min(\text{cost}_i(a), \text{cost}_{i-1}(a))$
      (b) $\text{cost}_i(a) +=$ hmmcost(h, i, current frame)
      (c) mincost = min(mincost, $\text{cost}_i(a)$)
   2. set $\text{cost}_0(a) = \infty$.
3. *From the active arc list, produce a new active state list.* Set active state list to empty. Then for each active arc a:
   1. If for $i \in \{1 \cdots 3\}$, $\text{cost}_i(a) + \text{unitcost}(a) \geqslant$ mincost + threshold, then prune a (i.e., delete a from active arc list).
   2. Otherwise if $\text{cost}_3(a) + \text{unitcost}(a) <$ mincost + threshold, then do the following. For each FSM transition f in a with destination s, if $\text{cost}_3(a) + \text{FSMcost}(f) <$ mincost + threshold then make s active (if it is not already), and set $\text{cost}(s) = \min(\text{cost}(s), \text{FSMcost}(f) + \text{cost}_3(a))$. (Initially all states have cost ∞).

Backtracking. When the last frame has been processed, we need to backtrack to determine the sentence that has been recognized. In order to backtrack, we need to add some links while searching. In particular, we keep two kinds of information: $\text{from}_i(a)$ denotes the last step of the current best path to an HMM state in a given active arc and lattice(s, t) denotes the best path to FSM State s at time t (where time is measured in frames). The following steps are added to the above algorithm to maintain the necessary links:

1. In Step 1.2, when we set $\text{cost}_0(a) = \text{cost}(s)$, also set $\text{from}_0(a) = (s, t-1)$.
2. In Step 2.1, if we set $cost_i(a) = cost_{i-1}(a)$, then also set $\text{from}_i(a) = \text{from}_{i-1}(a)$.

3. In Step 3.2, if we set cost(s) to FSMcost(f) + cost$_3$(a), then also set lattice(s, t) = from$_3$(a).

With these additions, we can trace backward at the end of the recognition and determine the sequence of FSM states that were traversed in order to reach the best final state in a least cost manner.

ε-transitions. Some active arcs have ε as the input symbol, rather than a real symbol from the input alphabet. This means that a transition can be made in the FSM layer *without* consuming an input symbol. This creates a complication, because multiple ϵ-transitions can be followed in sequence. The way this is handled is by essentially running Dijkstra's shortest path algorithm[13] on the graph of ε-transitions out of the active states.

During step 3, each time a state is made active, if it has outgoing ε-transitions, it is inserted into a heap, ordered by state cost. (If the state is in the heap already, and its cost is decreased, then a decrease-key operation is done to maintain the heap property.)

At the end of step 3, the least cost state is deleted from the heap repeatedly. Call the state that is deleted s: for each ε-transition a of s, if cost(s) + FSMcost(a) < mincost + threshold, then the destination state of a is made active (if it is not already) and its cost is updated (as in Step 3.2). If the destination state has ε-transitions, it too is inserted in the heap. This process continues until the heap is empty.

2.3. The FSM Layer and On-demand Composition

Finite state transducers are especially useful when there are multiple layers of information, because they have a natural form of *composition*, which we describe in this section. Consider the two FSMs in Fig. 6a (transition costs have been omitted for clarity).

The top FSM in Fig. 6a converts strings over the alphabet {A, B} into strings over {1, 2}, while the bottom one converts strings over {1, 2} into strings over {W, X, Y, Z}. Say we want to combine the operation of both FSMs, converting strings over {A, B} into strings over {W, X, Y, Z}. This can be done with the *composition* of the two automata, shown in Fig. 6b.

Each state of the composition corresponds to a pair of states, one state from each of the original FSMs. In general, there is a transition $(i, j) \xrightarrow{\alpha/\beta} (k, l)$ in the composition whenever there are transitions $i \xrightarrow{\alpha/\gamma} k$ and $j \xrightarrow{\gamma/\beta} l$, for some γ, in the original FSMs.

Remember that the two-level Viterbi algorithm needs the transformation from context-dependent units to sentences to look like a single FSM. We could achieve this simply by composing the FSMs for translating context-dependent units to phones, phones to words, and words to sentences.

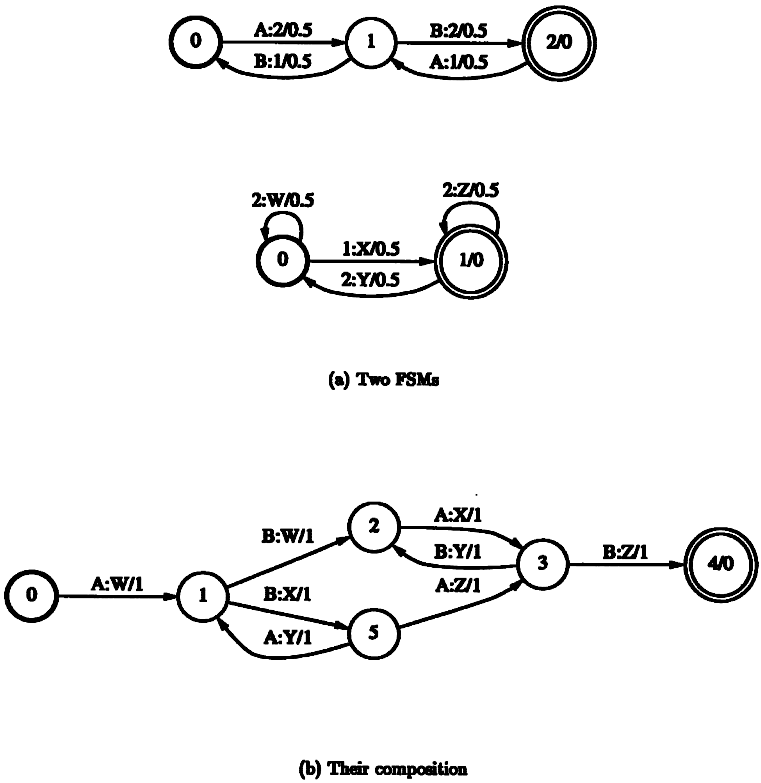

Fig. 6. The composition of two FSMs.

For large vocabulary systems, however, this is infeasible. An FSM for the grammar alone can have millions of transitions, while the lexicon can have many thousands; the composition would be immense. Most of the composition is not needed during recognition, since the pruning done by the Viterbi search ensures that most states and transitions will never be reached. Instead some of the composition can be done "on-demand"—a state of the composition (and its outgoing transitions) needs to be generated only when it is reached during the Viterbi search.

2.4. Bringing It All Together: A Sequential Recognizer

The sequential recognizer was configured with models for the North American business news (NAB) task, summarized in Tables II and III. The largest component is a bi-gram grammar, in which each state corresponds

Table II. Component Sizes in the Sequential Recognizer

Level of Representation	Alphabet size
Context-dependent units	29,666
Phones	53
Words	19,982

to a word in the lexicon, and the transition cost represents the probability of encountering two particular words in sequence. The translation of context-dependent units into phones is done by an FSM (C) that was developed by Pereira and Riley.[13] These models are intended to be examples; the recognizer could be configured with other models, including tri-gram or *n*-gram grammars, without changing the algorithms.

The FSMs for translating phones to words (L) and words into sentences (G) were precomposed, yielding a finite state transducer (M) that translates phones directly into sentences ($L \circ G$). The resulting FSM, which we call the "extended lexicon," was determinized using a determinizing algorithm for weighted transducers[14] (so that each state has at most one outgoing transition for each input symbol) and minimized (finding an equivalent FSM with the minimum number of states) off-line—a time-consuming operation, but one that only needs to be performed once, and not during recognition. After minimization, the extended lexicon had 250,928 states and 9,461,656 transitions. The FSM for translating context-dependent units into phones (C) and the extended lexicon FSM (M) are composed on-demand during recognition.

All parts of the recognition process are heavily data-driven, and the choice of data structures has a big impact on the performance of the recognizer. It also has many implications for parallelization, as we shall see later. The most important data structures of the sequential recognizer are:

Viterbi search. The active arc set, aaset, is a linked list of active arcs, each of which is a structure containing information relevant to that arc, such as the from state, input symbol, list of sinks and associated costs, and

Table III. Sizes of Mappings Between Representations

Mapping	States	Transitions
Context-dependent units → phones (C)	1,558	82,524
Phones → Words (L)	35,000	380,000
Words → Sentences (G)	19,982	5,660,571

current costs of its HMM states. Similarly the active state set, asset, is a linked list of structures.

A vector (or array), called has_existed, holds pointers to active state structures. It is used in Step 3.2, when a state becomes active or has its cost decreased, to find the active state structure corresponding to the state. The has_existed vector grows dynamically —as the number of active states increases, the has_existed vector must be reallocated occasionally.

On-demand FSM composition. The basic operation being performed by the on-demand composition code is: given a state number in the composed FSM compute the set of transitions out of that state. This computation requires the ability to translate a state number n in the composed FSM into the tuple t, the corresponding pair of states from the original FSMs, and vice versa. The forward direction is done using a "vector" n2t —an array that is reallocated occasionally as more states of the composed FSM are generated. The reverse direction is done using a hash table, t2n.

The FSM code also performs caching, so for example when asked for the transitions out of state n a second time, it does not need to recompute them. This caching is done using (among other devices) a vector of state information, called cache→states, containing information on each state, such as whether its transitions have been generated yet. As with n2t, cache→states must be reallocated occasionally as more states are generated.

To summarize, in this section we have presented the basic ideas underlying one speech recognition system and described the two-level Viterbi search algorithm. The next section describes our parallel implementation of this algorithm.

3. THE PARALLEL ALGORITHM

The general form of our parallel implementation of Viterbi search follows the sequential implementation. The sequential algorithm is data-centric, and is designed around two primary data structures: the active arc set and active state set. Our treatment of these data structures, which is described in Section 3.3, determines the structure of the parallel algorithm. Before we discuss the data structures, we review the operations provided by SGIs thread package (Section 3.1) and we explain how to avoid conflicts during memory allocation (Section 3.2). After we describe the implementation of the primary data structures, we present our parallel algorithm in Section 3.4 and then discuss the changes necessary to allow calls to the likelihood calculation (Section 3.5) and the on-demand composition of automata

algorithm from the FSM-library (Section 3.6) to occur in parallel. Finally, we summarize the key ideas used in the parallelization in Section 3.7.

3.1. Thread Package

The SGI Power Challenge provides two ways to write parallel programs. The first uses parallelizing compiler technology and programmer inserted annotations to generate parallel programs automatically, and is best suited to fairly regular programs. The second technique is based on user-defined threads and is better suited to programs, like the recognizer, that use complex data structures.

The thread library, which is built on top of the IRIX[2] system call sproc, supports three synchronization mechanisms: locks, barriers, and semaphores. Synchronization objects are allocated from arenas, which are created by the programmer using a library routine called usinit. The library supplies routines for creating a lock (usnewlock), acquiring a lock (ussetlock), and releasing a lock (usunsetlock). It also supports two types of barriers: m_sync implements a barrier for all threads in a program, whereas barrier implements a barrier for a subset of the threads. barrier takes a barrier object as an argument. Finally, the library provides routines for allocating a semaphore with a given initial value from a specified arena (usenewsema), applying P to a semaphore (uspsema), and for applying V to a semaphore (usvsema).

The m_fork library routine is used to specify that work should be done in parallel. m_fork dynamically forks N threads, where N is set by a call to m_set_nprocs, and does not return until all N threads have finished. These threads call the routine specified as the first argument to m_fork on the remaining arguments. For example, the following code:

```
m_set_procs(NPROCS)
m_fork(p_compute_max, a)
```

forks NPROCS threads, calls the function, p_compute_max(a) in each thread, and then waits until all N have completed before returning. m_get_myid is a library routine that returns the thread's id, which is a number between 0 and $N-1$.

3.2. Memory Allocation

The sequential recognizer uses Vo's vmalloc library,[15] which supports arenas, to do memory management. To allow different threads to

[2] IRIX is a trademark of Silicon Graphics Inc.

allocate memory in parallel we must handle three issues: interference between two requests for more system memory, interference between two threads allocating from the same arena, and deallocation of data that was allocated in parallel.

Each vmalloc arena maintains a chunk of space and hands pieces of it out to the user as necessary. When an arena runs out of space, vmalloc calls a function, specified by the user, to get more memory. In our case, this function calls malloc, and calls to it must be mutually exclusive. Such calls are rare, so we simply use a lock to guarantee mutual exclusion.

Two threads trying to allocate from the same arena will interfere. (Note that allocations from different arenas do not interfere.) We could handle this by associating a lock with each arena, but it is more efficient to expand the number of arenas instead. We replaced each arena from the sequential implementation with a vector of arenas, one per thread, in the parallel implementation. A thread allocates from its local arena rather than a single global arena.

Using this model of allocation causes some technical problems for deallocation. Space must be allocated and deallocated from same arena. In most cases, this does not cause a problem for our implementation because the owner, that is, the thread that allocated space, is responsible for deallocating it as well. In such a situation, the owner of the space simply calls vmfree (the vmalloc library's version free) with the appropriate arena to deallocate the space. Note that two such calls will not interfere, because different threads will deallocate to different arenas. There are, however, two cases in our implementation where space is not deallocated by the allocating thread. The first is related to a hash table used in the FSM library. Hash table entries are deallocated in a piece of sequential code at the very end of the computation. To make it possible to determine the appropriate arena for deallocation, we simply store the identity of the allocating thread in each hash table entry and retrieve it for the call to vmfree. The second is an array that is used as a cache in the FSM library, and it is handled analogously.

3.3. Data Structures

In the sequential recognizer, the active state set is implemented as a linked list. Parallel updates to this list would clearly interfere. One possible solution would be to use a lock to manage access to the list, but the lock would suffer from contention and very likely would be a bottleneck. A better solution is to restructure the list. In place of one shared list, we use a vector of lists, one list per thread, and we make each thread responsible for managing the states on its sub-list. The states, and by extension the original

list, are partitioned based on state numbers: state i is assigned to thread $i \bmod p$, where p is the number of threads. We refer to state i's assigned thread as its *owner*. An active state will appear only on the sub-list of its owner and that thread is responsible for all work related to the state. For example, only a state's owner may add or remove it from the active state set.

The active arc set is also implemented as a linked list in the sequential recognizer. In the parallel implementation, we restructure it in the same way as the active state set: an array of lists with one list per thread. The active arcs are partitioned based on the owner of the source state of the arc. That is, an arc from state i to state j will be assigned to thread $i \bmod p$. As was true for the active states, all work related to an active arc will be done by its owner.

In addition to the two primary data structures, the sequential recognizer maintains some auxiliary data structures: principally, a vector, called has_existed, that holds pointers to the active states and a heap that tracks the costs associated with ε-transitions. The has_existed vector grows dynamically as the number of active states increases. To prevent reads and writes of the vector from interfering with growing the vector, we restructured it into a set of (smaller) vectors (one per thread). Each thread is assigned the portion of the vector that corresponds to its states and grows its portion of the vector as necessary. The representation of the ε-transition heap in the parallel implementation remains unchanged from the sequential implementation.

3.4. Algorithm

In outline, the parallel Viterbi search closely resembles the sequential search described in Section 2.2. The following describes the work done by a single thread:

1. *Update the active arc list using the active state list.* The thread spins through its active state sub-list adding arcs to its active arc sub-list as necessary. No synchronization is required, because each thread reads and writes only its own data.

2. *Evaluate active arcs using the current frame and compute mincost.* The thread spins through its active arc sub-list performing the likelihood calculation for each arc. We discuss the changes necessary to allow calls to the likelihood calculation to occur in parallel in the next section.

The computation of mincost is a reduction: while doing the likelihood calculations each thread computes the local mincost of its active arcs and

then waits until the other threads are finished, finally, a single thread computes the global mincost from the p local mincosts.

3. *From the active arc list, produce new active state list.* This step performs two functions. First, it uses the value of mincost computed in the previous step to prune active arcs with costs that are out of range (that is, before mincost + threshold). Second, it updates the destination states of completed active arcs, adds them to the active state set, and determines their transitions (using on-demand composition of FSMs). An active arc is *completed* if $\text{cost}_3(a)$ is within the accepted range for the search.

The first function is easy to parallelize. Each thread spins through its active arcs, pruning as necessary. The second is more complicated. Recall that all computation related to a state is done by its owner, but that the active arcs are distributed based on their source state not their destination state. This means that the thread that determines that a state should become active is not necessarily its owner. To handle this, we split the computation into two parts, separated by a barrier. In the first part, we record the destinations of completed active arcs in a data structure instead of activating them immediately. The data structure, called pending, is a two dimensional array where each element contains a linked list. A state will be added to the list in element pending[t][s] by thread t, if thread t identifies the state, which is owned by thread s, as active. No synchronization is needed for the pending array, because only thread t will write to row t of the array (pending[t][ ]). The second step does the state updates. Each thread updates the states it owns, that is, thread s updates the states in column s of the pending array (pending[ ][s]).

Making a state active is a fairly complex operation: the FSM library must be called to determine the transitions that originate from the state; a heap used to manage ε-transitions must be updated; and finally, an entry for the state must be added to the has_existed vector. To allow different states to be made active in parallel required changes to the parts of the code that handle each of these actions. First, we multi-threaded the FSM code to allow two FSM states to be expanded (have their outgoing arcs determined) concurrently. We discuss the changes to the FSM library in Section 3.6. Second, we handled updates to the ε-transition heap by simply recording the information necessary to do the update in a local data structure during the parallel phase and then once the parallel phase completes, performing the delayed updates from the local lists sequentially. This method performed better than a lock implementation, because contention for the lock serialized the threads. Finally, as noted earlier, we restructured the has_existed vector to remove the interference caused by the need for it to grow dynamically.

4. *Clean-up and ε-transition handling.* In the final step, which is sequential and performed by a single thread, several data structures from the FSM code are cleaned up and the ε-transitions are handled by performing the delayed heap updates and then processing the heap entries as in the sequential recognizer (see Section 2.2).

Steps 1–3 are managed and synchronized by the function ParPF (for "parallel process frame") shown in Fig. 7, which is invoked for each thread in parallel, once per frame, by the call:

m_fork(ParPF, decoder, frame)

The synchronization is achieved using the m_sync() library routine. Thread 0 processes the local minimum cost vector (proc_Min) to complete the reduction that computes mincost. ParPF calls the following routines:

ParPF1 updates the active arc sub-list using the active state sub-list

ParPF2 evaluates active arcs using the current frame (*likelihood calculation*)

```
void ParPF(RecDecoder_t* decoder, RecFrame_t* frame)
{
  int i, me = m_get_myid();
  RecCost mincost=WORSTCOST;

  ParPF1(decoder, frame);
  ParPF2(decoder, frame);

  m_sync();
  ParPF2reset(decoder);

  if (me == 0) {
    for (i=0, decoder->num_active_arcs=0; i<nprocs; i++) {
      mincost = COLLECT(mincost, decoder->proc_Min[i]);
    }
    decoder->threshold = mincost + decoder->beam_width;
  }

  m_sync();
  ParPF3(decoder->aaset, decoder, frame);

  m_sync();
  ParPF4(decoder);

}
```

Fig. 7. High-level code structure for processing a frame.

ParPF2reset resets several vectors used in the likelihood calculation

ParPF3 prunes the active arc sub-list and determines (from the active arc sub-list) the updates to be made to the active state list, and records them in the pending array.

ParPF4 creates the new active state sub-list by doing the updates recorded in the pending array.

There are two significant parts of the computation that remain sequential: the signal processing front-end that maps the speech signal into a sequence of frames, and the ε-transition handling. Together with other smaller segments of sequential code, they comprise a significant portion of the recognition time when using a large number of processors, up to 25%.

3.5. Multi-Threading the Likelihood Calculation

This section discusses the changes that we made to the likelihood calculation to allow multiple threads to compute likelihoods concurrently.

Computing the likelihood for an active arc involves calculating hmmcost(*h*, *i*, current frame), where *h* is the HMM corresponding to (the input symbol of) this active arc and *i* is a state in the HMM ($1 \leqslant i \leqslant 3$). hmmcost is a memo-ized function, that is, it avoids recomputing likelihoods by remembering the computations it has already done. This memoization is implemented by keeping a pair of vectors: a bit-vector that indicates whether the likelihood for a particular HMM has been calculated and a vector that contains the computed likelihoods.

To allow the threads to evaluate their active arcs in parallel, we modified hmmcost to handle the interference caused by parallel accesses to the memoization vectors. Reads and writes to different elements of the vectors do not interfere, but reads and writes to the same element do. We could have used locks to synchronize accesses to these vectors, but instead we took advantage of two facts. First, as long as each thread uses its own scratch area for intermediate values, then two threads that compute the same likelihood are guaranteed to produce the same value. Second, because the SGI Challenge implements sequential consistency, we know that writes are visible in program order. This means that as long as we write to the result vector before the bit vector, we can guarantee that a reader will get the correct likelihood if it finds corresponding bit set. Together these two facts allow us to ignore the interference caused by parallel accesses to the vectors. At worst, two threads will calculate the same likelihood and the vectors will be written twice.

Recall that the probability distributions computed by the likelihood calculation are a mixture of Gaussians or rotations. The number of rotations

is significantly smaller than the number of HMMs. To save work, hmmcost remembers previously computed rotations as well as previously computed likelihoods. We handle the rotation memoization vectors using the same technique as the likelihood memoization vectors.

3.6. Multi-Threading the FSM Library

As noted earlier, the recognizer represents the mapping from context dependent units to phones using one FSM, and the mapping from phones via words to sentences as another. It uses the "on-demand" composition algorithm provided by the FSM library[8] to combine them into a single FSM. The FSM library uses several data structures to represent the current state of the composition, including a hash table (t2n) that maps tuples of state numbers from the original FSMs into a state number in the composed FSM and a vector (n2t) that does the reverse (that is, maps state numbers in the composed FSM into tuples of state numbers from the original FSM). In addition, the FSM library maintains a cache of previously visited states. These data structures need to grow dynamically to accommodate the growth in the number of states. To allow multiple calls into the FSM library to proceed concurrently, we must coordinate accesses to these data structures carefully. In this section, we describe the changes that we made to achieve this.

3.6.1. The Hash Table

The hash table, called t2n, is represented with an array of linked lists or buckets and the sequential version supports four operations: hashcreate, resize, hashfind, and hashinsert. Hashcreate creates a new hash table, resize grows the hash array and redistributes the hash table entries, hashfind takes a tuple and returns the corresponding state number, if the tuple is in the table, otherwise it returns a flag, and hashinsert adds a tuple to the table.

To understand the synchronization necessary for the hash table, we must understand the contexts in which these operations can occur. The hash table is created once per run during a sequential portion of the computation and as a result creation has no impact on the synchronization requirements of the data structure. Resizes occur dynamically in the sequential implementation, but are allowed to happen only once per frame, in a sequential portion of code, in the parallel implementation. This restriction simplifies the synchronization needed for the hash table. We allow inserts and finds to occur in parallel. There is no conflict when two threads want to access different buckets, but there is a potential conflict when two threads want to access the same bucket. We could resolve this conflict

```
usptr_t *fsmlocks;
HashTable ht;

/* allocate an arena for locks used in fsm code */
fsmlocks = usinit("/dev/zero");

/* allocate array of locks for hash table */
for (i=0; i < NUM_LOCKS; i++) {
  ht->hash_lock[i] = usnewlock(fsmlocks);
  if (ht->hash_lock[i] == NULL) {
    printf("lock allocation failed");
    exit(-1);
  }
}
```

Fig. 8. Lock allocation code.

using one lock per bucket. Instead we use one lock to manage a collection of buckets, which increases contention slightly, but decreases the number of locks required substantially. Figure 8 contains the code for allocating the lock array.

hashinsert and hashfind are called only from the function compstate, shown in Fig. 9, in the sequential recognizer. Compstate takes a tuple as input, and returns the corresponding state number. If the input tuple is not in the hash table, compstate invents a new state number for it (by incrementing a counter) and inserts the tuple and new state number into the table.

The simplest way to multi-thread this code would be to grab a lock for the duration of compstate, including the routines it calls. (We cannot just use a lock inside hashinsert, because another thread could insert the tuple

```
static int compstate(Comp comp, Tuple tuple)
{
  int n;

  /* tuple is the hash key */
  n = hashfind(comp, &tuple);
  if(n == FSMNoState) {
    n = comp->ns++;
    DSTVecSet(&comp->n2t, n, tuple);
    hashinsert(comp, n);
  }
  return n;
}
```

Fig. 9. The sequential function compstate.

```
static int compstate(Comp comp, Tuple tuple)
{
  int bucket, l, n, me;
  HashList hl, oldhl, newhl;
  HashTable t2n;

  /* hashfind - tuple is the hash key */
  bucket = HFUNC(tuple, t2n->tblsz);
  oldhl = t2n->array[bucket];
  for(hl = oldhl; hl != NULL; hl = hl->next) {
    if(EQTUPLE(hl->tuple, tuple)) {
      n = hl->value;
      break;
    }
  }

  /* hashinsert */
  if (hl == NULL) {
    l = bucket % NUM_LOCKS;
    n = FSMNoState;
    me = m_get_myid();
    newhl = (HashList) vmalloc(t2n->area[me],
                               sizeof(struct hash_list_rec));
    newhl->creator = me;
    newhl->tuple = tuple;

    ussetlock(t2n->hash_lock[l]);
    /* second lookup */
    for(hl = t2n->array[bucket]; hl != oldhl; hl = hl->next) {
      if(EQTUPLE(hl->tuple, tuple)) {
        n = hl->value;
        break;
      }
    }

    if (n == FSMNoState) {
      newhl->next = t2n->array[bucket];
      newhl->value = comp->ns[me];
      t2n->array[bucket] = newhl;
      comp->ns[me] = assign_new_state();
    }
    usunsetlock(t2n->hash_lock[l]);

    /* continued in Figure 11 */
```

Fig. 10. Hashinsert and hashfind rolled into compstate.

into the hash table before we grab the lock, and we would end up inserting it a second time.)

Grabbing a lock in compstate, however, would cause a serious contention problem, since it is called frequently. Reducing contention for the bucket locks as much as possible is important. As noted earlier, we can reduce contention by minimizing the amount of time that any one thread holds the lock. We do this by in-lining hashfind and hashinsert into compstate (see Figs. 10 and 11). First, we check whether the desired tuple is already in the table before obtaining the lock. If it is, no update to the table is required and we can avoid acquiring the lock altogether. If the tuple is not found, then we must acquire the appropriate bucket lock and then double check the bucket list, because another thread could have inserted the desired tuple between this thread's initial lookup and its acquisition of the lock. Note that we only need to look at bucket entries that appeared between the initial lookup and the lock acquisition (that is, from $t2n \rightarrow array[bucket]$, the new head of the list, to oldhl, the old head of the list).

```
    /* continued from Figure 10 */

    if (n == FSMNoState)
      n = newhl->value;
    else
    /* tuple was found during the second look-up,
         free the pre-allocated space */
      vmfree(t2n->area[me], newhl);

    /* n2t update code */
    if (n < DSTVecSize(comp->n2t))
      DSTVecSet(&comp->n2t, n, tuple);
    else {
      /* delay the update */
      a = (add_node *) vmalloc(comp->add_node_area[me],
                               sizeof(add_node));
      a->value = n;
      a->tuple = tuple;
      a->next = comp->add_list[me];
      comp->add_list[me] = a;
      comp->add_flag = 1;
    }

  }
  return n;
}
```

Fig. 11. Parallel compstate continued.

To reduce the time spent in the critical region, we allocate space for a new hash table entry speculatively and fill in as many fields as possible before entering the critical region. If the tuple is not found during the second lookup, preallocating space will reduce the amount of time that the thread holds the lock significantly. If the tuple is found during the second lookup, which is unlikely, the preallocated space is simply freed after the lock is released.

In the parallel code, we need to be careful about inventing new state numbers. Rather than using a straightforward counter for the next available state number, which would suffer from severe contention, we hand out state numbers in blocks. We maintain a central counter that holds the next available block of numbers. Each thread starts with a block of unassigned numbers and assigns them as needed. When a thread runs out of unassigned numbers, it requests a new block by grabbing the lock that protects the central counter and updating the counter.

3.6.2. Dynamically Growing Vectors

As noted earlier, the FSM library maintains a vector (called `n2t`, and shown in Fig. 9) that maps state numbers from the composed FSM into tuples of state numbers from the original FSMs. This vector grows dynamically as the number of states increases. We need some mechanism for synchronizing reads, writes, and resizes of this vector. One possible solution is to use a single lock for the entire array, but such a lock would suffer from high contention. Instead, we take advantage of the fact that new states are created only during `ParPF4` (see Fig. 7), and the `n2t` values for states created during the current frame are not needed until a subsequent frame. This allows us to postpone writes to the vector. The code for postponing the writes appears near the bottom of Fig. 10; if a new state number is larger than the current vector size, we store the state and its corresponding tuple on a local list. Following `ParPF`, we grow the `n2t` vector to the appropriate size and store the pending values.

The cache used in the FSM has a similar problem. It too grows dynamically as the number of states increases, but it has the property that we can bound its maximum size in the next frame by the total number of states in the current frame. The reason is that a state can only be expanded (and need its arcs cached) during a frame if the state exists at the start of the frame (except during ε-transition handling, which is done sequentially). As a result, we simply pre-grow the cache at the start of each frame. This, combined with the fact that the cache entry for a state is accessed only by the state's owner (except during ε-transition handling) allows us to read and write the cache without synchronization.

Note in both these cases we are taking advantage of the fact that we know something about how the recognizer calls the FSM library. To write a generally useful multi-threaded version of the FSM library, we would need to replace the dynamically growing vectors with a more parallel-friendly data structure, such as a hash table.

3.7. Parallelization Ideas

The keys ideas underlying our parallelization are:

- Exploit the data-centric nature of the algorithm to do a data-centric work allocation. This choice yields good data locality and allows us to preserve the basic structure of the underlying recognizer in our implementation.
- Exploit different ways to manage data to eliminate interference without having a substantial negative effect on performance. In particular, we restructure data to allow for concurrent access without locking, exploit reductions, and postpone updates to avoid high contention locks.
- Finally, improve performance by reducing lock contention, either by moving unnecessary code out of the critical section or by careful choice of lock granularity.

4. RESULTS

To evaluate the impact of parallelism on recognition speed, we compared the performance of the parallel recognizer[3] with that of the sequential recognizer for several recognition tasks. Here we present the performance results in three ways. First we determine raw speedup on the ARPA North American Business News (NAB) task. Then we analyze the performance in terms of speedup of individual sections of the code. Finally we investigate how consistent the raw speedup and per-code section results remain when recognition task parameters vary.

4.1. Raw Speedup on the NAB Task

Table IV shows the average running time over 300 sentences from the 20,000 word ARPA North American business news task. The parallel code

[3] The results reported here were gathered using an older version of the recognizer that used sbrk in place of malloc to get extra space for a vmalloc arena and used a slightly different method for handing out state numbers in the FSM composition algorithm.

Table IV. Running Times and Speedup of the Parallel Recognizer

Recognizer	Sequential	Parallel					
No. of processors	1	1	2	4	8	12	16
Average running time	35.1	33.7	20.4	12.3	8.4	7.8	7.6
Speedup over sequential	1.0	1.0	1.7	2.8	4.2	4.5	4.6
Relative to real time	3.9	3.7	2.3	1.4	0.9	0.9	0.8

on one processor is slightly faster than the sequential code because of small improvements made to the code structure. The recognition time drops quickly as more processors are used: while the sequential recognizer runs 3.9 times slower than real time, the parallel recognizer runs in real time on eight processors. It is important to note that this degree of speedup is only possible because we parallelized all three major components of the the recognizer: Viterbi search, likelihood calculations, and on-demand FSM composition.

The speedup tails off as the number of processors increases. This is caused, in part, by synchronization at locks on shared data structures and at barriers between phases of the Viterbi algorithm. In addition, we left some small components of the recognizer sequential.

4.2. Efficiency of Parallelization by Component

To understand the performance of the parallel recognizer better, we analyzed the performance of individual sections of code. We used the gettimeofday library routine to time sections of the code. The sections we considered follow the structure of the Viterbi algorithm, as described in Section 3.4, step 4. We recorded the time spent in each of section for each thread for one representative sentence. (Similar results were obtained on other sentences.) From these times we can determine the total work being done, and detect whether extra work is being created by the parallelization.

Table V shows the sum over all frames and all threads, of the time in each section of the code. The decrease in total work that is seen in some code sections is probably due to cache effects. ParPF4 shows a substantial increase in the total work. We have not been able to determine the reason, and this remains a topic for future work.

Table VI shows the sum over all frames of the maximum (over the threads) of the time spent between barriers. (Note that the m_fork routine induces a barrier after ParPF4.) ParPF1 and ParPF2 show reasonable

Table V. Total Time in Each Section of the Parallel Code

Code section	1 processor	2	4	8	12	16
parPF1	1.8	1.7	1.6	1.5	1.5	1.6
parPF2	17.3	20.1	20.1	19.2	18.6	18.3
parPF2reset	0.5	0.6	0.6	0.7	0.7	0.9
parPF3	3.1	2.9	2.4	2.2	2.2	2.2
parPF4	4.3	5.6	6.5	7.6	8.8	11.1
Total	27.1	30.9	31.3	31.2	31.9	34.2

speedup (9.1 on 16 processors). The speedups are not perfect, because of load imbalance in ParPF2 that is caused by the fact that we assign work based on the owner of the data, rather than using a more complicated scheme designed to balance the workload better. ParPF3 shows good speedup (10.3 on 16 processors). And as expected from the total work results, ParPF4 shows poor speed-up (2.9 on 16 processors).

To improve the overall performance of the parallel recognizer, we need to focus our efforts on improving the performance of ParPF4 and to a lesser extent ParPF2.

The observed running time of the recognizer is two to three seconds more than the total sum of maximum times entries in the table, because of synchronization and the sequential work done outside of the ParPF code, chiefly in epsilon-handling, in the signal processing front-end, and in ParPF clean-up code that resizes the hash table, extends the FSM cache arrays, and performs delayed writes into the FSM data structures. The synchronization time is spent in the m_fork that starts the threads running at the beginning of each frame and in the barriers synchronizing after ParPF2, ParPF2reset, and ParPF3.

Table VI. Sum of Maximum Times in Each Section of the Parallel Code

Code section	1 processor	2	4	8	12	16
parPF1 and parPF2	19.1	11.1	5.8	3.2	2.4	2.1
parPF2reset	0.5	0.3	0.2	0.1	0.1	0.1
parPF3	3.1	1.5	0.7	0.4	0.3	0.3
parPF4	4.3	3.2	2.3	1.7	1.5	1.5
Total	27.1	16.1	9.0	5.3	4.3	3.9
Observed running time	27	17	12	8	7	7

Table VII. Wide Beam Running Times and Speedup of the Parallel Recognizer

Recognizer	Sequential	Parallel					
No. of processors	1	1	2	4	8	12	16
Average running time	83.1	80.7	47.0	26.4	16.1	13.9	13.5
Speedup over sequential	1.0	1.0	1.8	3.1	5.2	6.0	6.2
Relative to real time	9.2	8.9	5.2	2.9	1.8	1.5	1.5

4.3. Performance of the Parallel Recognizer under Varying Parameters

The recognizer was run with varying task parameters, to ensure that it gives consistent results. Here we present the results of running the NAB task with a wider beam (11.5). This increases the word accuracy of the recognizer from 73% to 77%, at the cost of increased computation. The average running times on the 300 sentence test suite are shown in Table VII. We do not achieve real-time performance, but we do achieve substantial improvement over the sequential implementation.

The recognizer has better speedup with the wider beam (6.2 versus 4.6). This is because there is more work to be done for each frame, with no extra synchronization, so the percentage of time spent doing useful work increases. This suggests that the effectiveness of the parallel recognizer should increase as it is applied to harder recognition tasks.

As in the narrow-beam case, we analyzed the efficiency of parallelization in each section of the code. Table VIII shows the sum over all frames and all threads of the time in each section of the code, while Table IX shows the sum over all frames of the maximum (over the threads) of the time spent between barriers. The trends are the same as in the baseline case: ParPF4 shows a big increase in total work (and here ParPF2 shows

Table VIII. Total Time in Each Section of the Parallel Code for Wide Beam

Code section	1 processor	2	4	8	12	16
parPF1	5.9	5.7	4.9	4.4	4.4	4.5
parPF2	38.5	42.7	43.1	42.0	42.9	44.6
parPF2reset	0.5	0.6	0.7	0.7	0.8	1.0
parPF3	10.0	9.7	8.2	7.1	6.8	6.7
parPF4	9.3	12.3	14.1	16.7	19.4	23.5
Total	64.3	70.9	70.8	70.8	74.3	80.3

Table IX. Sum of Maximum Times in Each Section of the Parallel Code for Wide Beam

Code section	1 processor	2	4	8	12	16
parPF1 and parPF2	44.4	24.4	12.4	6.3	4.7	4.1
parPF2reset	0.5	0.3	0.2	0.1	0.1	0.1
parPF3	10.0	5.0	2.2	1.1	0.8	0.6
parPF4	9.3	6.7	4.4	3.1	2.7	2.6
Total	64.3	36.5	19.2	10.6	8.2	7.3
Observed running time	66	39	23	16	11	11

a slight increase in total work too); ParPF1 and ParPF3 show a decrease in total work; ParPF1, ParPF2, and ParPF3 show good speedup; and ParPF4 is disappointing with a speedup of only 3.6.

As in the baseline case, the maximum times are two to four seconds less than the observed running time of the recognizer on the timed sentence. The sequential parts of the computation (including ε-transition handling, the signal processing front end, and ParPF cleanup code) and the overhead of synchronization take up the remaining time.

5. CONCLUSIONS

This paper shows that harnessing the power of shared memory multiprocessors can increase greatly the speed of a speech recognizer. The increase in speed can be achieved without changing the structure of the recognition algorithm and without making major changes to the implementation. Our parallel recognizer is closely tied to an existing sequential recognizer, so improvements made in the sequential recognizer (phone modeling, likelihood calculations, grammar representation, etc.) can be applied to the parallel recognizer directly.

Our goal was to achieve real-time recognition, with a reasonable error rate, on the benchmark ARPA North American business news (NAB) task. We achieved that goal, with the parallel recognizer running up to six times faster than the sequential recognizer on the NAB task. The parallel recognizer has performed similarly on other recognition tasks.

Topics for future work include further performance improvements and a Pentium-based implementation. The main place where our parallelization leaves substantial room for improvement is the multi-threaded code for on-demand composition of FSMs. This code has disappointing parallel performance, and a substantial improvement here would increase the parallel efficiency of the recognizer significantly.

We have left some parts of the recognizer sequential that could be parallelized. In particular, the ε-heap could be parallelized, and the signal processing front end could be pipelined with the Viterbi search, so that a new speech frame is ready as the Viterbi algorithm needs it.

Multiprocessors based on Intel's Pentium processor are becoming widespread, so a Pentium-based implementation may be very useful. Multi-Pentium systems offer cheaper CPUs and cheaper memory than UNIX workstations, and it is not clear how the tradeoffs made in building small Pentium multiprocessors will affect parallel performance. Most currently available multi-Pentium systems have at most four processors, rather than the 16 or more available on large UNIX servers. However, our parallel recognizer achieves a factor of three speedup on four Challenge processors; repeating that speedup on a Pentium system would be significant.

ACKNOWLEDGMENTS

We wish to thank Adam Buchsbaum, Emerald Chung, Andrej Ljolje, Mehryar Mohri, Fernando Pereira, and Mike Riley for their help and advice. Power Challenge XL and Irix are trademarks of Silicon Graphics Incorporated.

REFERENCES

1. M. D. Riley, A. Ljolje, D. Hindle, and F. Pereira, The AT&T 60,000 Word Speech-to-text System, *Proc. EUROSPEECH-95*, pp. 207–210 (1995).
2. K. A. Wen and J. F. Wang, Efficient Computing Methods for Parallel Processing: An Implementation of the Viterbi Algorithm, *Computers Math. Applic.*, **17**(12):1511–1521 (1989).
3. H. Noda and M. N. Shirazi, A MRF-Based Parallel Processing Algorithm for Speech Recognition Using Linear Predictive HMM, *Proc. ICASSP '94*, pp. I-597–I-600 (1994).
4. M. Goudreau, K. Lang, S. Rao, and T. Tsantilas. Towards Efficiency and Portability: Programming with the BSP Model, *Proc. Symp. Parallel Algorithms and Architectures* (1996).
5. S. Phillips and A. Rogers, Parallel Speech Recognition, *Proc. EUROSPEECH-97* (1997).
6. C.-H. Lee and L. R. Rabiner, A Frame-Synchronous Network Search Algorithm for Connected Word Recognition, *IEEE Trans. Acoustics, Speech, Signal Proc.*, **37**:1649–1658 (1989).
7. M. Mohri, F. Pereira, and M. Riley, Weighted Automata in Text and Speech Processing, *Proc. ECAI-96 Workshop* , ECAI (1996).
8. M. Mohri, Finite-State Transducers in Language and Speech Processing, *Computational Linguistics* , **23**(2):269–311 (1997).
9. L. Rabiner and B.-H. Juang, *Fundamentals of Speech Recognition*, Prentice Hall, Englewood Cliffs, New Jersey (1993).
10. S. Pinker, *The Language Instinct*, W. Morrow and Co., New York (1994).

11. J. E. Shoup, Phonological Aspects of Speech Recognition, in *Trends in Speech Recognition*, W. A. Lea (ed.), Prentice-Hall, Englewood Cliffs, New Jersey (1980).
12. A. P. Dempster, N. M. Laird, and D. B. Rubin, Maximum Likelihood from Incomplete Data via the EM Algorithm, *J. Roy. Statist. Soc.* **39**(1):1–38 (1977).
13. E. W. Dijkstra, A Note on Two Problems in Connection with Graphs, *Numerical Mathematics*, **1**:269–271 (1959).
14. F. Pereira and M. Riley, Speech Recognition by Composition of Weighted Finite Automata, *Finite-State Language Processing*, MIT Press (1997).
15. Kiem-Phong Vo, Vmalloc: A General and Efficient Memory Allocator, *Software Practice & Experience*, **26**:1–18 (1996).