Parallel Speech Recognition

Steven Phillips and Anne Rogers
AT&T Labs-Research, 180 Park Ave, PO Box 971, Florham Park, NJ 07932-0971
email: {phillips,amr}@research.att.com

Abstract

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 significantly the speed, and therefore the accuracy or vocabulary size, of a speech recognizer. We 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.

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. This paper shows that harnessing the power of shared memory multiprocessors, which are becoming increasingly common, can increase greatly the speed of a speech recognizer. This increase in recognition speed can be used to expand the set of recognition tasks where real time recognition is possible, or to increase recognition accuracy on existing real-time tasks.

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. Shared-memory has achieved widespread acceptance in the marketplace, in part, because it is relatively easy to program. Shared-memory machines are the standard for Unix servers, and PCs with multiple Pentium or Pentium Pro processors are becoming prevalent. We therefore chose a shared-memory multiprocessor for the parallel implementation, specifically a Silicon Graphics Power Challenge XL.

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. [6]. 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.

Previous applications of parallelism to speech recognition have focused on special-purpose hardware (*e.g.* [7]), or on isolated word recognition (*e.g.* [5]), or in the case of 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.* [2]), but are not applicable to large speech recognition tasks, where the graph is defined implicitly.

2 The Sequential Recognizer

The sequential recognizer we used is the Viterbi-based system described in Riley *et al.* [6], which uses the finite state transducer (FSM) library of Pereira *et al.* [4]. This section gives an overview of the sequential system, so that we have a base from which to present the work done in the parallelization.

The sequential recognizer uses the two-level Viterbi search algorithm of Lee and Rabiner [3], which operates at the boundary between the phone model layer (where speech frames are matched against 3-state hidden Markov models of context dependent phones) and an upper layer which represents the mapping of phone sequences to sentences.

The upper layer represents context dependence, the lexicon, and the grammar using weighted finite state transducers, so we refer to it as the *FSM layer.* In order to provide the mapping from phone sequences to sentences to the Viterbi decoder, the FSM layer uses *on-demand composition* of finite state transducers, as described by Pereira et al. [4]. Whenever the Viterbi decoder needs to know the set of transitions out of a state in the FSM layer, the on-demand composition algorithm does only the work needed to determine those transitions. The Viterbi algorithm does not examine much of the graph, and as a result, only a very small fraction of the entire graph is generated.

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 3-state HMM that corresponds to a context dependent phone. For each, frame, the Viterbi algorithm queries the phone model layer to determine the likelihood of observing that frame in the HMM states associated the active arcs. 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.

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 $\text{cost}_1(a) \ldots \text{cost}_3(a)$, 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 an HMM h, let hmmcost($h, i,$ current frame) be the cost of observing the current frame in the i'th state of h. The basic steps involved in processing a single speech frame are the following.

Update the active arc list. For each active state s and for each input symbol α such that there is a transition from s labeled with 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 $\text{cost}_0(a) = \text{cost}(s)$.

Evaluate active arcs using current frame. Set mincost $= \infty$. 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) set mincost $= \min(\text{mincost}, \text{cost}_i(a))$
2. set $\text{cost}_0(a) = \infty$

Produce a new active state list. Set active state list to empty. Then for each active arc a:

1. If for each $i \in \{1 \ldots 3\}$, $\text{cost}_i(a) \geq$ mincost + threshold, then prune a (i.e. delete a from active arc list).
2. Otherwise if $\text{cost}_3(a) <$ mincost + threshold, then do the following. For each FSM transition f in a with destination s, if $\text{cost}_3(a) +$ FSMcost(f) $<$ mincost + threshold then make s active (if it is not already), and set cost(s) $= \min(\text{cost}(s), \text{FSMcost}(f) + \text{cost}_3(a))$. (Initially all states have cost ∞).

ϵ-arcs. 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-arcs can be followed in sequence. The way this is handled is by essentially running Dijkstra's shortest path algorithm[1] on the graph of ϵ-transitions out of the active states at the end of each frame.

3 The Parallel Recognizer

The parallel recognizer is implemented on a shared memory machine that supports coarse-grain parallelism, using library routines that provide user-defined threads and synchronization via locks, barriers, and semaphores.

The Viterbi algorithm is data-centric, so it is natural to allow the sequential data structures to drive the parallelization. The primary data structures are the active arc set and active state set, which are used to track the arcs and states that are currently within the Viterbi search beam. In the sequential recognizer, each set is represented by a linked list. Our approach is to partition these lists into sub-lists, one per thread. This is done via a simple mechanism: state number i (of the FSM layer) is *assigned* to thread i mod p, where p is the number of threads. The active state sub-list for a thread then contains those states that are assigned to it, and all work related those states is done by that thread. The active arc list is partitioned in a similar way: an active arc is assigned to the same thread as its *source* node.

This treatment of the active arc and active state sets determines the structure of the parallel algorithm. Whenever the sequential code performs an operation on the active arc or active state list, the parallel algorithm performs the operation *on each sub-list in parallel.*

In the rest of this section, we present an overview of the parallel implementation, highlighting the most important issues that arose and the points where significant work had to be done to parallelize the algorithms and data structures.

To process a single frame, the Irix library routine `m_fork` is invoked as follows:

```
m_fork(proc_frame, decoder, frame);
```

This forks the threads, each of which is run on the procedure `proc_frame`. The steps in `proc_frame` follow the same outline as the sequential algorithm. To understand the algorithmic changes required in the parallelization, consider the steps performed by a single thread:

Update the active arc 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 the thread reads and writes only its own data.

Evaluate active arcs using the current frame. The thread spins through its active arc sub-list performing the likelihood calculations for each arc. The likelihood calculation may be done simultaneously by other threads, so the code had to be *multi-threaded*.

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 $i \in \{1 \ldots 3\}$. `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 state has been calculated and a vector that contains the computed likelihoods. The changes necessary for multi-threading involved using separate scratch areas for different threads, then writing the computed likelihood *before* writing the bit-vector. The SGI Challenge implements sequential consistency, so any other thread is guaranteed to see the correct likelihood value if it finds the bit set in the bit-vector.

The computation of `mincost` is known as a *reduction*, which is a computation that applies a commutative and associative operation to a collection of values, and is parallelized as follows. While doing the likelihood calculations the thread computes the local `mincost` of its active arcs. Then at the end of this step, we use a barrier to ensure all threads have finished computing, then a single thread computes the global `mincost` from the p local `mincost`s. A second barrier ensures that the global mincost is available before the threads continue to the the next step.

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, above `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 a 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 several bookkeeping data structures need to be updated. 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 transitions determined) concurrently. We discuss the changes to the FSM library below. Second, we postponed the updates to the ϵ-transition heap until a later sequential phase. And finally, we restructured the bookkeeping data structures to allow them to be updated in parallel.

ϵ-transition handling At the end of the frame, a single thread sequentially performs all the delayed heap updates and then does the ϵ-transition handling.

In addition to these algorithmic changes, the parallelization involved restructuring of data structures to avoid contention, with special care being given to allow different threads to allocate and deallocate memory simultaneously.

The multi-threading of the FSM library is centered on the routines for on-demand composition of automata. Two or more automata are combined to produce a composed automaton, whose states correspond to tuples, with a tuple containing one state from each of the input automata. These routines make heavy use of a hash table, which maps from tuples of states to state numbers in the composed automaton. Different threads needing to update the hash table simultaneously formed a point of severe data contention, that required careful synchronization. Locking access to the hash table is an inadequate solution, as too much time would be spent waiting for the lock. We could resolve this conflict using one lock per bucket, as accesses to different buckets do not conflict. Instead we use one lock to manage a collection of buckets, which increases contention slightly, but decreases the number of locks required substantially. Reordering the code to minimize the amount of time any thread holds a bucket lock further reduced contention for the hash table.

Table 1: Running times and speedup of the parallel recognizer

Narrow Beam							
Recognizer	Sequential	Parallel					
Number 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

Wide Beam							
Recognizer	Sequential	Parallel					
Number 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 Results

To evaluate the impact of parallelism on recognition speed, we compared the performance of the parallel recognizer with that of the sequential recognizer for several recognition tasks. The top half of Table 1 shows the average running time over 300 sentences from the 20,000 word ARPA North American business news task on a twenty processor SGI Power Challenge XL. The parallel code 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, and they do not benefit from the extra processors.

The bottom half of Table 1 shows the effect of increasing the beam width from 9.23 to 11.5, which increases the accuracy of the recognizer on this task from 73% to 77% and increases the work by more than a factor of two. The running times shown are again the average over 300 sentences. A parallel speedup of a factor of six is achieved using 12 processors. The parallel efficiency is greater than before, since the amount of work done per speech frame is larger, and hence proportionately less time is spent on synchronization. This indicates that as the sophistication of speech recognition tasks increases, so too will the effectiveness of leveraging shared memory multiprocessors for speech recognition.

Acknowledgements We wish to thank 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] E. W. Dijkstra. "A note on two problems in connection with graphs". *Numerical Mathematics*, 1, 1959.

[2] M. Goudreau, K. Lang, S. Rao, and T. Tsantilas. Towards efficiency and portability: Programming with the BSP model. In *Proceedings of Symposium on Parallel Algorithms and Architectures '96*, 1996.

[3] 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.

[4] F. Pereira M. Riley and E. Chung. Transducer composition: a flexible method for on-demand expansion of context-dependent grammar networks. In *Proceedings of EUROSPEECH-97*, 1997.

[5] H. Noda and M.N. Shirazi. A MRF-based parallel processing algorithm for speech recognition using linear predictive HMM. In *Proceedings of ICASSP '94*, pages I–597 – I–600, 1994.

[6] M.D. Riley, A. Ljolje, D. Hindle, and F. Pereira. The AT&T 60,000 word speech-to-text system. In *Proceedings of EUROSPEECH-95*, pages 207–210, 1995.

[7] 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.