16th European Signal Processing Conference (EUSIPCO 2008), Lausanne, Switzerland, August 25-29, 2008, copyright by EURASIP

A LEXICAL-TREE DIVISION-BASED APPROACH TO PARALLELIZING A CROSS-WORD SPEECH DECODER FOR MULTI-CORE PROCESSORS

Naveen Parihar

Dept. of Electrical and Computer Engineering
Mississippi State University
Mississippi State, MS 39762
email: np1@ece.msstate.edu

Eric A. Hansen

Dept. of Computer Science and Engineering
Mississippi State University
Mississippi State, MS 39762
email: hansen@cse.msstate.edu

ABSTRACT

We present a novel approach to parallelizing a lexical-tree based LVCSR decoding algorithm for multi-core desktop processors. The approach distributes the search space among the cores by dividing the lexical tree in a way that minimizes communication between cores. Synchronization and load balancing schemes for this approach are described. The parallel algorithm is benchmarked on a 5k-word Wall Street Journal task. The context-dependent triphone model baseline system achieves a WER of 8.4%. The algorithm is shown to achieve a speedup of 1.63 on an Intel Core 2 Duo processor, with an average CPU utilization of 86%. The results also show that increasing the width of pruning schemes improves the parallel speedup.

1. INTRODUCTION

After years of exponential growth in general-purpose processor frequencies, physical limits imposed by power density limit further growth. As a result, CPU manufacturers have adopted a different strategy for increasing computational power that involves adding multiple cores to a single chip. Most desktop computers now have dual-core processors; quad-core processors have been recently released; within a few years, processors with 32 or 64 cores are expected to be common. With multi-core architecture becoming increasingly prevalent, applications including speech decoding must be parallelized to exploit the resources of the multiple cores.

Although highly-optimized speech decoders can run in real time on large vocabulary tasks in clean conditions by employing aggressive pruning, aggressive pruning results in performance loss when decoding using the noisy versions of the clean data sets. Less aggressive pruning while decoding in noisy environments can worsen the real-time factor by five to ten times [1]. Hence, there is a need to improve the real-time rate of speech decoders by parallelizing the search.

The multiple acoustic feature sets combination techniques at model level, which are commonly known as probability combination algorithms (both synchronous and asynchronous), improve the performance of a system over the best single feature system [2, 3, 4, 5]. However, the trade-off is the extra computation required for multiple model evaluations. Multiple model evaluations can be done in parallel on multiple cores of a multi-core processor, thereby eliminating any drop in real-time performance of a decoder due to multiple model evaluations. However, the scalability of this approach is limited by the number of available feature streams, and it is unlikely that a large number of cores can be exploited with this approach. Also, the overall real-time rate of such a system would be limited to the slowest model evaluation. A better solution is to keep the flexibility to run the slowest model evaluation on several cores and deliver a better real-time rate. Parallelization of a speech decoder at the search level provides this flexibility.

Surprisingly, over the past two decades, few approaches to parallelizing speech decoder search algorithms on a shared-memory architecture have been proposed. Kimball et al. [6] implemented a parallel version of a simple Viterbi-decoding based recognizer (335 words with no grammar) on the MIMD BBN Butterfly Parallel Processor. RaviShankar [7] created a parallel version of the CMU Sphinx decoder on a PLUS microprocessor using C-threads. Phillips et al. [8] parallelized a Viterbi recognizer on a shared-memory architecture with Challenge processors. However, none of these approaches are for lexical-tree based decoders. A blockwise pipelining approach for a three-core cellphone processor was recently presented [9]. However, the search itself is not parallelized. In this paper, we present a parallel lexical-tree based LVCSR search algorithm for multi-core desktop processors that parallelizes the decoder at the search level. As a result of parallelizing the search, both the acoustic model evaluation and language model lookup are also parallelized. Note that in this research, we are not interested in optimizing acoustic model evaluations, language model lookups or search itself, but are interested in developing generic techniques for parallelizing any lexical-tree based decoder.

2. BASELINE LVCSR SYSTEM

For this research, we employ the public domain Mississippi State decoder [10]. It is a hierarchical dynamic programming based time-synchronous Viterbi search engine that supports N-gram context dependent cross-word triphone decoding. This system has been used on several evaluations including Aurora Evaluations [11]. Because this decoder uses a lexical tree, our approach should be applicable to other lexical-tree based decoders.

2.1 Lexical Tree

Lexical-tree based decoding enables efficient sharing of computations among the active word-beginning triphone models. Instead of creating multiple copies of the lexical-tree, a single copy of the lexical-tree is used to minimize memory requirements. However, the tradeoff is that a N-gram word history has to be maintained for each search path.

2.2 Pruning and Path Merging

The decoder supports three different pruning schemes. A standard **beam pruning** approach is used to maximize search accuracy while minimizing memory requirements. **Active phone model instances pruning** is used to reduce the peak memory requirements. (All the partial paths in the search space that correspond to the same phone model, same lexical node and N-1 word history are considered at the same position in the search space. This position in the search space is represented as a unique phone model instance.) **Word-end pruning** limits the number of active word-level paths, thereby reducing computational and memory load.

Partial paths at all three levels (word, phone, state) in the search hierarchy are merged using the principle of dynamic programming.

2.3 Decoding Algorithm

The portion of the decoder code that is most computationally intensive is abstracted in the pseudocode of Algorithm 1. The detailed design of this decoder, including the search algorithm, is given in [10].

3. PARALLEL SEARCH ALGORITHM

In any approach to parallelizing a search algorithm, it is important to identify independent portions of the search space that can be used to divide the search effort in a way that minimizes interactions among the threads. Typically, interactions among threads take place using mechanisms such as locks and barriers that tend to serialize the code, resulting in loss of concurrency. In our approach to parallelization, we exploit the locality provided by the lexical tree to minimize such interactions. We divide the search space by dividing the lexical tree among threads.

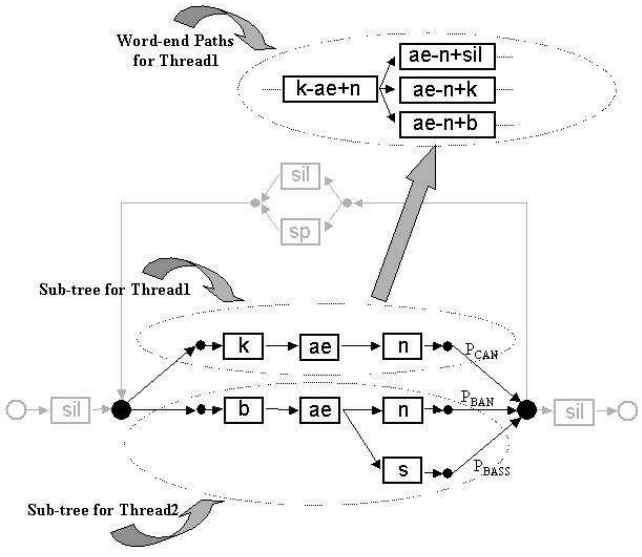

Fig. 1. Division of lexical-tree among threads.

In Figure 1, for example, the search space is distributed among threads by distributing the branches of the lexical tree at the root node, assigning different branches to different threads. Each thread is assigned one or more branches at the root node. Note that each branch at the root node

Algorithm 1 Sequential Decoding Algorithm

```
create a path to start word
for all frames do
    read feature vector
    expand_word_paths_to_states()
    expand_phone_paths_to_states()
    compute phone model instance pruning threshold
    prune active phone model instances based on phone model instance pruning threshold
    project_states()
    compute state-level beam threshold
    prune state-level paths based on state-level beam threshold
    compute phone-level beam threshold
    create_word_paths()
    compute word-level beam threshold
    prune word-level paths based on word-level beam threshold
    compute word-end pruning threshold
end for
backtrace best path using word-level history

expand_word_paths_to_states():
for all active word-level paths do
    apply word-end pruning
    propagate word-level paths to phone model instances
    propagate paths to HMM states in the phone model instance
end for

expand_phone_paths_to_states():
for all active phone-level paths do
    propagate phone-level paths to next phone model instance according to the pronunciation
    propagate paths to HMM states in the phone model instance
end for

project_states():
for all active phone model instances do
    for all states in phone model instance do
        evaluate the state and propagate paths to next states with Viterbi pruning
        if next state is phone model instance last state then
            create phone-level path corresponding to the phone model instance
        end if
    end for
end for

create_word_paths():
for all phone-level paths do
    prune phone-level paths based on phone-level beam threshold
    if phone-level path corresponds to end-phone of a word pronunciation then
        create word-level path corresponding to the word
        merge this path with other word-level paths if appropriate
        add the word to word-level history if appropriate
    end if
end for
```

can also be a tree. We call the group of branches (or trees) at the root node that is assigned to a thread the *thread-tree* corresponding to that thread. In this simple example, there are only two thread-trees in the lexical tree, and we assume that we have two threads available to us. The first thread-tree is composed of the branch that ends at the word *CAN*. The second thread-tree is composed of the part of the lexical tree that has two word-ends: *BAN* and *BASS*. Hence, each thread is assigned one thread-tree. During decoding, each thread searches its own thread-tree(s). While search progresses through a branch, all the triphone models that are constructed dynamically from the monophones in this branch are local to the thread to which this branch is allocated. The states corresponding to these triphone models are also evaluated locally by this specific thread. This approach has the following advantages.

- Acoustic model state evaluation and language model lookup are automatically parallelized,
- Because the branches of the lexical tree are independent, interactions among threads can be significantly reduced,
- The algorithm as well as the changes to the serial code for this approach are simple and minimal because each thread searches its portion of the lexical tree.

Nevertheless, as we will see in the following sections, the need to exchange information among threads and perform load balancing is not entirely eliminated by this approach.

3.1 Synchronization

The time-synchronous property of the Viterbi search must be maintained in any parallelization of the Viterbi algorithm. Even when each thread searches its own thread-tree, all the threads must synchronize with each other at each time frame. In addition to the need to synchronize threads at each time frame, some of the speech decoder operations require synchronization at each time frame. All three pruning schemes require synchronization to compute global maximum scores within the processing of each time frame. The sequence of mandatory synchronization steps in each time frame is collectively called *time synchronization*.

The choice to parallelize the search by using lexical trees to divide the search space results in a second form of synchronization. Whenever any thread reaches one of the leaf nodes of the thread-tree assigned to it, representing word ends, it needs to exchange information with other threads, for two reasons. First, whenever a thread reaches any leaf node of its thread-tree, it needs to broadcast the word-level path representing the word-end at the leaf node to all the other threads. This is necessary because, in the next time frame, this word-level path needs to be expanded in all the thread-trees. In the simple example of Figure 1, at the leaf node of the thread-tree corresponding to thread1, all the word-level paths representing the end of word *CAN* are broadcast to thread2. Second, the path-merging step at word ends requires comparison of all partial paths with the same word history, which may be distributed among various threads. If these paths occur in different threads, the parallel decoder needs to exchange information among threads. This information exchange is called *lexical-tree synchronization*.

3.2 Load Balancing

Due to the statistical nature of the heuristics derived from HMM acoustic models and N-gram language model, the active search-space (active triphone model instances and the corresponding HMM states) at any time cannot be accurately predicted. A trivial way to distribute the lexical tree among the threads is to randomly assign the sub-trees to available threads. However, a static allocation of the active search space among threads, without employing any expert knowledge, may not be efficient from a load-balancing point of view. The number of active triphone models is very large at word-beginnings, and the number of active triphone models drops dramatically as the word-end approaches [12]. If the active triphone models at word-beginnings, which are phonetically similar, are not uniformly distributed among the threads, load imbalance can become a bottleneck. A way to address this load balancing problem is to create groups of phonetically similar monophones based on linguistic knowledge. The lexical tree can then be distributed among threads on the basis of the first monophone in each sub-tree at the root node. Sub-trees with the first monophone which are phonetically similar can be uniformly distributed among the threads.

3.3 Parallel Algorithm

To begin, we separate the sequential algorithm into two parts – parallel and serial. While the parallel part of the algorithm consists of the work that each thread can execute in parallel on child threads, the serial part of the algorithm runs sequentially on a single main thread. For a processor with two cores, we have one main thread and two child threads as shown in Algorithm 2, and Algorithm 3.

The *frame_barrier* is the barrier that provides Viterbi synchronization at each frame. Other barriers in the pseudocode collectively provide *time synchronization*. These barriers define the concurrency among the threads which is obviously very small. It is important that the work is evenly distributed between the threads during the small portions of code between the barriers. The *lexical-tree synchronization* is provided by serializing the code in the function *create_word_paths()* in the main thread. Two locks, not shown in the pseudocode, were used to prevent corruption

Algorithm 2 Main Thread

```
1:  create a path to start word
2:  for all frames do
3:      read feature vector
4:      frame_barrier
5:      models_barrier
6:      compute phone model instance pruning threshold
7:      instance_pruning_threshold_barrier
8:      instance_pruning_barrier
9:      project_states_barrier
10:     compute state-level beam threshold
11:     state-level_beam_pruning_threshold_barrier
12:     state-level_beam_pruning_barrier
13:     compute phone-level beam threshold
14:     create_word_paths()
15:     compute word-level beam threshold
16:     prune word-level paths based on word-level beam
        threshold
17:     compute word-end pruning threshold
18: end for
19: backtrace best path using word-level history
```

Algorithm 3 Child Threads

```
1:  for all frames do
2:      frame_barrier
3:      expand_word_paths_to_states()
4:      expand_phone_paths_to_states()
5:      models_barrier
6:      instance_pruning_threshold_barrier
7:      prune active phone model instances based on phone
        model instance pruning threshold
8:      instance_pruning_barrier
9:      project_states()
10:     project_states_barrier
11:     state-level_beam_pruning_threshold_barrier
12:     prune state-level paths based on state-level beam
        threshold
13:     state-level_beam_pruning_barrier
14: end for
```

of the data structures shared among the threads. While one locks the word history data structure, the second lock is used to lock a common memory manager. Other significant changes involving division of lexical trees among threads are abstracted in the pseudocode.

The parallel search algorithm is implemented using the *P-thread* library on the *Fedora Core 6* Linux operating system. The compiler used is GNU's *gcc verison 4.1.1*.

4. EXPERIMENTAL SETUP AND RESULTS

The parallel decoding algorithm was evaluated on the 5K-word closed-loop WSJ0 task [13]. Phonetic decision-tree based state-tied cross-word speaker-independent triphone acoustic models with 16 Gaussian mixtures per state were trained using the SI-84 training set [14]. The Nov'92 NIST evaluation set, consisting of 330 utterances, was decoded with a standard 5K lexicon and bigram language model. This baseline system achieves a WER of 8.4%.

The parallel algorithm was evaluated by measuring speedup on an Intel Core 2 Duo processor. Speedup is the runtime of the best sequential algorithm divided by the runtime of the parallel algorithm. As shown in Table 1, the initial implementation was slower than the sequential version. However, after identifying the locking of the single common memory manager as the bottleneck, we eliminated this lock by providing a local memory manager for each child thread. This improved the speedup to 1.33. Next, we reduced thread overhead due to thread context switching. The main thread accesses the data structures modified by the child threads while computing each of the pruning thresholds and other serial operations such as *lexical-tree synchronization*. This results in significant processor idling. By removing one child thread and merging all the corresponding code with the main thread, we were able to improve the speedup to 1.49.

Algorithm	Speedup
With Memory-manager Lock	0.34
Without Memory-manager Lock	1.33
With only one child thread	1.49
With load balancing	1.63

Table 1. Parallel Algorithm Speedup on Core 2 Duo.

Beam-width (state, phone, word)	Speedup	WER(%)
200 150 150	1.43	10.4
250 200 200	1.53	8.6
300 250 250	1.63	8.4
325 275 275	1.68	8.4

Table 2. Effect of Beam Pruning on Speedup.

Maximum Active Phone Model Instances	Speedup	WER(%)
6000	1.63	9.1
10000	1.63	8.4
14000	1.71	8.3

Table 3. Effect of Phone Model Instance Pruning on Speedup.

This change enabled the parallel algorithm to run two threads on two cores – one main thread and one child thread. Note that we do not expect this improvement in speedup to be as significant for a very large number of cores where the number of corresponding threads is also very large. Next, we focussed our attention on load balancing, described earlier in this paper. The phonetic groups used for load balancing are vowels, fricatives, nasals, approximants, and stops. Load balancing improved the average processor utilization from 81% to 86%. As a result, the speedup improved to 1.63.

The effect of beam pruning on speedup is shown in Table 2. As the beam-widths are increased, the speedup improves because the concurrency between the *time synchronization* steps improves. Similarly, as shown in Table 3, increasing the threshold for maximum active phone model instances results in improved speedup.

The effect of cache size on speedup is shown in Figure 4. On a Pentium Core Duo, the baseline speedup of 1.63 dropped to 1.48. We believe that a smaller L2 cache size of only 2MB on Pentium Core Duo, as compared to 4MB on Core 2 Duo, results in a larger number of cache misses and a corresponding drop in the speedup.

5. CONCLUSIONS AND FUTURE WORK

We have presented a novel approach to parallelizing a lexical-tree based search algorithm for LVCSR on a multi-core architecture. The speedup obtained for the baseline WSJ0 based system is 1.63 for a two-core processor. Although *time synchronization* is required for Viterbi decoding, concurrency between these synchronization steps can be increased by increasing the various beam widths, which can improve WER. A lexical-tree based approach to load balancing also helps achieve a better distribution of work. In the near future, we plan to evaluate our approach on a quad-core processor, and develop more sophisticated load balancing techniques to increase the CPU utilization. Although we evaluated our approach on the standard 5000 word WSJ0 task, the technique is general and can be applied to larger tasks such as the 20k word WSJ task and Switchboard.

Processor	Cache	Clock Freq.	Speedup
Intel Core 2 Duo	4MB	2.4GHz/core	1.63
Intel Pentium Core Duo	2MB	2.8GHz/core	1.48

Table 4. Effect of Cache Size on Speedup.

REFERENCES

[1] F. Hilger and H. Ney, "Quantile Based Histogram Equalization for Noise Robust Large Vocabulary Speech Recognition," *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 14, no. 3, pp. 845–854, May 2006.

[2] X. Li and R. Stern, "Training of stream weights for the decoding of speech using Parallel feature streams," in *Proc. ICASSP 2003*, Hong Kong, April 2003, pp. 832–835.

[3] H. Tolba, S. A. Selouani, and D. OShaughnessy "Auditory-based acoustic distinctive features and spectral cues for automatic speech recognition using a multi-stream paradigm," in *Proc. ICASSP 2002*, Orlando, FL, May 2002, vol. 1, pp. 837–840.

[4] H. Glotin and F. Berthommier "Test of several external weighting function for multiband full combination ASR," in *Proc. ICSLP 2000*, Beijing, China, October 2000, vol. 1, pp. 333–336.

[5] J. Luettin, G. Potamianos, and C. Neti "Asynchronous stream modeling for large vocabulary audio-visual speech recognition," in *Proc. ICASSP 2001*, Salt Lake City, Utah, May 2001, vol. 1, pp. 169–172.

[6] O. Kimball et al., "Efficient Implementation of Continuous Speech Recognition on a Large Scale Parallel Processor," in *Proc. ICASSP 1987*, Dallas, TX, USA, March 1987, pp. 852-855.

[7] M. K. Ravishankar, *Parallel Implementatation of Fast Beam Search for Speaker-Independent Continuous Speech Recognition*. Tech. Rep., Computer Science & Automation, Indian Institute of Science, Bangalore, India, July 1993.

[8] S. Phillips and A. Rogers, "Parallel Speech Recognition," *International Journal of Parallel Programming*, vol. 27, no. 4, pp. 257-288, January 1999.

[9] S. Ishikawa et al., "Parallel LVCSR Algorithm for Cellphone-oriented Multicore Processors," in *Proc. ICASSP 2006*, Toulouse, France, May 2006, pp. I177-I180.

[10] N. Deshmukh et al., "Hierarchical Search for Large Vocabulary Conversational Speech Recognition," *IEEE Signal Processing Magazine*, vol. 16, no. 5, pp. 84-107, September 1999.

[11] N. Parihar and J. Picone, "An Analysis of the Aurora Large Vocabulary Evaluation," in *Proc. EUROSPEECH 2003*, Geneva, Switzerland, September 2003, pp. 337-340.

[12] J. J. Odell, *The Use of Context in Large Vocabulary Speech Recognition*. Ph.D. thesis, Queens College, University of Cambridge, Cambridge, UK, March 1995.

[13] D. Paul and J. Baker, "The Design of Wall Street Journal-based CSR corpus," in *Proc. ICSLP 1992*, Banff, Alberta, Canada, October 1992, pp. 899-902.

[14] N. Parihar, *Performance Ananlysis of Advanced Front Ends on the Aurora Large Vocabulary Evaluation*. M.S. thesis, Department of Electrical and Computer Engineering, Mississippi State University, Mississippi State, USA, December 2003.