Parallel Lexical-tree Based LVCSR on Multi-core Processors

Naveen Parihar\(^1\), Ralf Schlüter \(^2\), David Rybach\(^2\), and Eric A. Hansen\(^3\)

\(^1\)Dept. of Electrical and Computer Engineering, Mississippi State University, USA.
\(^2\)Human Lang. and Pattern Recognition, Comp. Sc. Dept., RWTH Aachen University, Germany.
\(^3\)Dept. of Computer Science and Engineering, Mississippi State University, USA.

naveenparihar@ieee.org, {schluter,rybach}@cs.rwth-aachen.de, hansen@cse.msstate.edu

Abstract

Exploiting the computational power of multi-core processors for large vocabulary continuous speech recognition (LVCSR) requires changes in the recognizer architecture. In this paper, we consider how to parallelize the search component of a lexical-tree based speech recognizer. We introduce a hybrid-parallel method for dynamically dividing the lexical-tree copies among the cores at each frame. Each core is responsible for graph traversal in the lexical-tree copies allocated to it. This approach is compared to a previously-introduced static method that divides the lexical tree itself, so that each core is responsible for a different subtree of each of the lexical-tree copies. The new method outperforms the previous one when applied to the RWTH TC-STAR EPPS English LVCSR system running on four cores of an Intel Core-i7 processor with varying pruning-beam width settings.

Index Terms: parallel speech recognition

1. Introduction

Personal computers, servers and embedded devices have dual-core, tri-core or quad-core processors. Speech decoding must be parallelized to exploit the increased processor throughput.

The two most time-consuming components of single-pass LVCSR decoding are: (1) Gaussian-mixture likelihood computation, and (2) Search management (graph traversal and pruning). Likelihood computation takes 30%-70% of total time, mostly for arithmetic operations which means it is computation-intensive. Data-parallel model based methods for speeding up likelihood computation on multi-core processors are described in [1]. It is also possible to use a GPU to speed up likelihood computation [2, 3].

The search management component consumes most of the remaining time. At large beam widths, the time for search management dominates overall runtime. Model mismatch conditions such as channel and noise mismatch, common in practical applications, also increase search space size for given beam-width settings [4]. Since the search management component must manipulate the active search space in main memory, it is communication-intensive. Parallelizing search management is challenging for several reasons; multi-core processors have limited main-memory bandwidth, the presence of some inherently serial steps limits speedup, multiple fine-grained parallel steps require multiple synchronizations, and synchronization of multiple steps with small time granularity results in load imbalance that accumulates over time.

Previous work on parallelizing search algorithms can be classified based on the type of search (single/multiple-pass), the parallel-algorithm model, and the shared-memory hardware.

(a) Single-pass, data-parallel, and multi-processor: Early work on parallelization includes a Viterbi recognizer (335 words) on the MIMD BBN Butterfly Processors [5], the Sphinx recognizer on a PLUS microprocessors using C-threads [6], and the AT&T recognizer on Challenge processors [7]. None of these approaches are for lexical-tree based recognizers.

(b) Two-pass, pipeline, and multi-core processor for embedded devices: A blockwise pipelining approach for a three-core cellphone processor was recently presented [8]. The two passes of the search are pipelined and mapped to two cores but none of the individual search passes is parallelized.

(c) Single-pass, data-parallel, and GPU: A WFST-based recognizer was parallelized in [3].

(d) Single-pass, data-parallel, and multi-core processor: A WFST-based search algorithm is parallelized in [9]. A lexical-tree division-based (LTD) method is presented [10]. A similar LTD approach with a different static load-balancing technique was introduced in [11]. However, the LTD methods suffer from load imbalance due to static load-balancing techniques.

This paper presents a novel method for parallelizing search management for a lexical-tree based single-pass LVCSR recognizer. It is based on a hybrid parallelization model that involves both data-parallel and work-pool models. The hybrid approach, called Lexical-tree Copies Decomposition (LTCD), exploits the locality provided by the lexical tree in a different fashion that avoids the main drawbacks of the LTD method. Instead of dividing the lexical tree itself, we dynamically distribute the active copies of the lexical tree among the cores during the state (HMM) expansion step using a work-pool model. The work-pool model provides a dynamic and a complete load balancing of the state expansion step. It also improves the load balancing of the other parallel steps that are parallelized using a data-parallel model. A comparison of the new hybrid-parallel LTCD method with the data-parallel LTD method is presented.

2. Background

We briefly review the search algorithm of the recognizer we parallelize as well as our previous approach to parallelization.

Word-Conditioned Tree Search: The RWTH recognizer uses a word-conditioned lexical-tree search algorithm with across-word context-dependent triphone models [12]. High-level pseudocode for the algorithm is given in Figure 1. Standard beam and histogram pruning schemes are applied to the state hypotheses and word-end hypotheses [13]. Early application of language model (LM) knowledge is incorporated using LM lookahead trees [14].

Parallelization Based on Lexical-tree Division: The data-parallel LTD method [10, 11] parallelizes search management by exploiting the locality provided by the lexical tree to divide
is parallelized using an independent data-parallel decision-tree traversal is parallelized using a work-pool model, while the search management. The HMM state expansion step of graph Our new LTCD method, is a hybrid approach to parallelize based approach (PFLC) [1]. The LM lookahead trees for newly activated lexical-tree copies are created in serial because it takes only a small percentage of overall runtime. However, application of the LM lookahead scores to the active search hypotheses is achieved using a data-parallel model on multiple cores. The scalability of the LTCD method with the number of cores depends on the average number of active lexical-tree copies at any time frame, which usually varies from a few tens to a few hundreds depending on the beam width for a large-vocabulary task.

The state expansion step for within-word graph traversal is parallelized as follows. During processing of each time frame, work done while expanding the active state hypotheses corresponding to the active lexical-tree copies is considered a work pool. Each lexical-tree copy is represented by a set of active lexical-tree copies allocated to the cores in the previous time frame processing, and a list of newly activated but unallocated lexical-tree copies in the current time frame. Each newly activated lexical-tree copy is represented by a set of state hypotheses corresponding to the entry states of lexical tree. Thus, for $N_c$ cores, the work pool consists of $(N_c + 1)$ lists.

The work pool is dynamically distributed among the cores using an asynchronous round robin (ARR) load balancing scheme as shown in Figure 3. The list $i$ represents a set of lexical-tree copies allocated to core $i$ during state expansion in the previous time step. The list $N_c$ represents the newly activated lexical-tree copies. Each core, core $i$, begins state expansion in parallel by expanding state hypotheses corresponding to the list of lexical-tree copies in list $i$. This order is shown in Figure 3 by a priority number (Lower number means higher priority), priority $i$, for core $i$. In particular, the core $i$ allocates a block of lexical-tree copies from list $i$. The corresponding block of state hypotheses is stored in consecutive memory locations representing efficient coalesced memory read accesses. Core $i$ runs out of work when all the work in list $i$ is completed. Then a block of unexpanded lexical-tree copies is transferred to core $i$ from the right neighbor of list $i$ defined as list $(i + 1) \mod N_c$. If

1: create a path to start word
2: for all frames do
3: read feature vector
4: activate new lexical-tree copies
5: expand HMM states by creating state hypotheses
6: create LM lookahead trees for newly activated lex-tree copies
7: apply LM lookahead scores to state hypotheses
8: compute threshold, $f_{beam \text{-state}}$ for beam pre-pruning
9: prune state hypotheses based on $f_{beam \text{-state}}$
10: likelihood computation of state hypotheses
11: compute threshold, $f_{beam \text{state}}$ for beam pruning
12: prune state hypotheses based on $f_{beam \text{state}}$
13: compute threshold, $f_{hist \text{-state}}$, for histogram pruning
14: prune state hypotheses based on $f_{hist \text{-state}}$
15: create word-end hypotheses for all state hypotheses that reach the leaf nodes of the lexical tree
16: compute threshold, $f_{beam \text{-word-end}}$, for beam pruning
17: prune word-level hypotheses based on $f_{beam \text{-word-end}}$
18: create word backpointers for traceback
19: recombine word-end hypotheses
20: compute threshold, $f_{hist \text{-word-end}}$, for histogram pruning
21: prune word-end hypotheses based on $f_{hist \text{-word-end}}$
22: end for
23: backtrace best-path using word backpointers

Figure 1: High-level pseudo-code of the single-pass search. the search effort in a way that minimizes interactions among the cores. The lexical tree is divided into subtrees at the root nodes, and the subtrees are allocated to different cores. Figure 2 demonstrates the division of search space by dividing the lexical-tree copies assuming a bigram LM and a three-word vocabulary. Our implementation keeps only active-search hypotheses corresponding to the active LM history.

Groups of phonetically similar monophones are created based on linguistic knowledge [10]. The triphone arcs originating at the root nodes of the lexical tree are called first-generation arcs. Each first-generation arc represents a tree which are distributed among the cores on the middle phone of the triphone on the arc. Arcs with phonetically similar monophones are distributed among the cores to create subtrees. At runtime, each core traverses its subtree. One of the drawbacks of this method is increased load imbalance as the number of cores increases. Also, the maximum number of cores that can be exploited is limited by the number of distinct monophones that occur as the middle phone on the first-generation arcs.

3. Lexical-tree Copies Decomposition

Our new LTCD method, is a hybrid approach to parallelize search management. The HMM state expansion step of graph traversal is parallelized using a work-pool model, while the other parallel steps of graph traversal and pruning are parallelized using a data-parallel model. Likelihood computation is parallelized using an independent data-parallel decision-tree based approach (PFLC) [1]. The LM lookahead trees for newly activated lexical-tree copies are created in serial because it takes only a small percentage of overall runtime. However, application of the LM lookahead scores to the active search hypotheses is achieved using a data-parallel model on multiple cores. The scalability of the LTCD method with the number of cores depends on the average number of active lexical-tree copies at any time frame, which usually varies from a few tens to a few hundreds depending on the beam width for a large-vocabulary task.

The state expansion step for within-word graph traversal is parallelized as follows. During processing of each time frame, work done while expanding the active state hypotheses corresponding to the active lexical-tree copies is considered a work pool. Each lexical-tree copy is represented by a set of active lexical-tree copies. The work pool is organized as a lists of lexical-tree copies allocated to the cores in the previous time frame processing, and a list of newly activated but unallocated lexical-tree copies in the current time frame. Each newly activated lexical-tree copy is represented by a set of state hypotheses corresponding to the entry states of lexical tree. Thus, for $N_c$ cores, the work pool consists of $(N_c + 1)$ lists.

The work pool is dynamically distributed among the cores using an asynchronous round robin (ARR) load balancing scheme as shown in Figure 3. The list $i$ represents a set of lexical-tree copies allocated to core $i$ during state expansion in the previous time step. The list $N_c$ represents the newly activated lexical-tree copies. Each core, core $i$, begins state expansion in parallel by expanding state hypotheses corresponding to the list of lexical-tree copies in list $i$. This order is shown in Figure 3 by a priority number (Lower number means higher priority), priority $i$, for core $i$. In particular, the core $i$ allocates a block of lexical-tree copies from list $i$. The corresponding block of state hypotheses is stored in consecutive memory locations representing efficient coalesced memory read accesses. Core $i$ runs out of work when all the work in list $i$ is completed. Then a block of unexpanded lexical-tree copies is transferred to core $i$ from the right neighbor of list $i$ defined as list $(i + 1) \mod N_c$. If
the work in right neighbor, list \((i+1)modN_c\), is also exhausted, the transfer is initiated from list \((i+2)modN_c\). This process continues until there is no list with any work. Figure 4 shows the decomposition among two cores assuming a three word vocabulary and a bigram LM.

Division of the active lexical-tree copies among the cores has the additional effect of partly parallelizing across-word graph traversal. The first part of across-word graph traversal, which consists of creating word-end hypotheses, is parallelized because each core is responsible for creating the word-end hypotheses at the leaf nodes of its lexical-tree copies. However, the second part involves recombination of the word-end hypotheses and is serial. Assuming a trigram LM, consider recombination at the root nodes, \(s_0(r)\), of a lexical tree copy with a specific word history \((v, w)\), where \(r\) represents a specific across-word right context. Maximization of all the word-end hypotheses that ended with word \(w\) and context \(r\) is performed over all the lexical-tree copies with a word history \((u, v)\) where \(u\) is a variable. Hence, recombination occurs across multiple lexical-tree copies, which may be distributed among the cores.

**Frame-Processing Flow, Synchronization, and Pruning:**
Parallel frame-processing is composed of a specific but mixed sequence of parallel and serial steps. Each step depends on completion of the previous step, and hence, this sequence can be modeled as a *task-dependency graph* that requires synchronization among the cores. Figure 5 illustrates the parallel frame-processing flow, and the synchronization points. The different types of synchronization are summarized below.

(a) **Time Synchronization due to time-synchronous Viterbi.**
(b) **Synchronization in Parallel Pruning:** Parallel beam pruning requires two synchronizations. The first synchronization is applied after the first step, which involves finding a local best score at each core. The second step computes the global best score from local best scores. The second synchronization is needed at the end of second step because the third step involves parallel pruning on each core using the global best score.

Histogram pruning also requires two synchronizations. The first synchronization is needed before the first step, which computes the threshold using a histogram in serial. The second synchronization is applied after the first step, which involves finding a local best score at each core. The second step computes the global best score from local best scores.

**Overall Optimizations:**
Search management is communication-intensive because the working data size exceeds the size of all the cache levels and the data must be read/written from main memory. The serial recognizer is already optimized to maximize the coalesced memory accesses by placing the data in buffers. Effectively, the bandwidth reduces by half for two-core parallelization. Hence, we designed the data structures to maximize preservation of the data locality of the serial recognizer. This included placing core specific data into separate buffers. False sharing among the cores is avoided by making sure that data for multiple cores does not reside in the same cache line. Overhead due to the processor coherency protocol is avoided to a large extent by making sure that whenever a core modifies some data in memory, it is not referenced by another core at that time.

**4. EXPERIMENTATION**

**4.1. Recognition System**

Experiments were run on an Intel Core-i7 Quad processor (Model # 920). The parallel versions were implemented using the P-thread library, and compiled using GNU gcc v4.3.2.

The experiments were performed using the RWTH 2007 TC-STAR English LVCSR baseline system 1 [16]. A one-pass four-gram lexical-tree based Viterbi decoding was performed. Phonetic decision tree based state tying resulted in a single tree with 4,501 generalized triphone states consisting of a total of 880,244 density components. Ten speech segments totaling 121.7 secs were randomly selected from the 2007 eval set to create Test-set-10 for profiling purposes. All the timing measurements are averaged over five runs. The Test-set-10 achieves a WER of 8.8% for various beam widths. The distribution of the elapsed runtime and key search statistics (average number of active lexical-tree copies / frame and average number of active state hypotheses active / frame) for the serial recognizer are shown in Table 1. Standard deviation in runtime was between 0.2-0.5 secs for various beam widths. We also randomly selected one recording with 165 segments totaling 2,370.61 secs from the 2007 eval set to create Test-set-165 that is used to benchmark the performance.

<table>
<thead>
<tr>
<th>Block Size</th>
<th>1</th>
<th>5</th>
<th>25</th>
<th>100</th>
<th>500</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Time (s)</td>
<td>374.84</td>
<td>374.53</td>
<td>374.35</td>
<td>374.71</td>
<td>380.31</td>
</tr>
</tbody>
</table>

**Table 2:** With a beam width of 400, elapsed runtime of Test-set-10 due to varying block size for PLCMD method on 2-cores.

---

1The width of sections between the synchronization points does not reflect the actual processing time.
Although the speedup is slightly less as compared to/Test-set-10, processors have a limited bandwidth. However, the best speedups are achieved by LTCD in the range 1.47-1.30 which is slightly but consistently better than the speedup achieved by LTD in the range 2.09-1.74, without any Werner degradation, and it outperforms the previous parallelization method by 11-17%. It is likely to improve performance even further on future multi-core processors that have more numbers of cores and a better memory architecture.

### 6. References


[9] N. Parihar and E. Hansen, “A Lexical-tree Division-based Approach to parallelizing the search management component of a lexical-tree based recognizer, called lexical-tree copies decomposition. Instead of dividing the lexical tree into subtrees that are statically distributed among the cores, as in previous work, it dynamically distributes the active copies of the lexical tree among the cores. Dynamic load balancing leads to improved performance. When evaluated on an Intel Core-i7 Quad processor in solving a large search space problem, it achieves parallel speedup in the range 2.09-1.74, without any Werner degradation, and it outperforms the previous parallelization method by 11-17%. It is likely to improve performance even further on future multi-core processors that have more numbers of cores and a better memory architecture.

### 5. Conclusion

We have described a hybrid-parallel approach to parallelizing the search management component of a lexical-tree based recognizer, called lexical-tree copies decomposition.