INTERSPEECH 2011

A multithreaded implementation of Viterbi decoding on Recursive Transition Networks

Fabio Brugnara

Human Language Technology Research Unit
FBK-Fondazione Bruno Kessler, Trento, Italy
brugnara@fbk.eu

Abstract

This paper describes the move to a multithreaded implementation of a Recursive Transition Network Viterbi speech decoder, undertaken with the objective of performing low-latency synchronous decoding on live audio streams to support online subtitling. The approach was meant to be independent on any specific hardware, in order to be easily exploitable on common computers, and portable to different operating systems. In the paper, the reference serial algorithm is presented, together with the modifications introduced to distribute most of the load to different threads by means of a dispatcher/collector thread and several worker threads. Results are presented, confirming a performance benefit in accordance with the design goals.

Index Terms: large vocabulary speech recognition, Viterbi decoding, parallel computing, multithreading.

1. Introduction

In the last years, multi-core computer architectures have become widespread, and the challenge to exploit them effectively has naturally emerged in a field like automatic speech recognition, where processing speed is still a major concern. Parallel processing capabilities are present on recent machines both as multi-core CPUs, and as specialized many-core processors, initially devised for graphic computations, but suitable for other compute intensive applications as well. Hence, there have been approaches that exploited the possibilities of the CPU alone, e.g. [1, 2], and others that rely on the availability of a suitable GPU to achieve larger speedups, e.g. [3, 4].

The goal of the work presented in this paper was that of improving the efficiency of an existing decoder to make it better suited for real-time subtitling. The decoder was already working faster that real-time, but in order to ensure low-latency on live audio streams in a real application, a more substantial margin was required.

The design choice was to identify the most demanding portion of the processing and provide an efficient implementation, able to exploit multi-core architectures, while maintaining an easy portability to different environments, thus relying only on native multithreading support as provided by recent operating systems. Beside, particular care has been taken in reducing as much as possible the need for tight synchronization in inter-thread communication.

Some added complexity was due to the fact that the decoder operates on a recursive description of the recognition language, as described in the next section. This capability is important for two reasons: first, it broadens the class of supported grammars to context-free grammars, but above all it allows a convenient and efficient run-time linking of sub-languages into the main recognition language – a feature that is very important in the application fields in which the decoder is employed.

This work was partly supported by *PerVoice* under the grant *Speech Analytics*.

The paper is organized as follows. In Sec. 2 the reference serial implementation of the decoder is described. Sec. 3 describes the modifications made to introduce parallelism, while Sec. 4 presents some benchmark results and describes a real-world application scenario. Finally, Sec. 5 draws some conclusions.

2. The serial implementation

The reference within-word recursive decoder has been entirely developed by FBK, and has been applied to speech recognition with n-gram language models (up to 7-grams), and vocabularies of over a million words. This section introduces its data structures and the overall process flow, which is the basis for the following parallel version.

2.1. Data structures

2.1.1. Static structures

The main data structure is an integrated network, a Weighted Finite State Transducer (WFST) that embodies linguistic and lexical knowledge. Output symbols correspond to words, the target recognition units, while input symbols normally correspond to acoustic units, modeled by HMMs. However, an input symbol can also refer to another WFST, making the overall structure guiding the search a Recursive Transition Network.

The networks are statically loaded in memory, and this choice has benefits and drawbacks. On one end, it avoids overhead due to the need of accessing the language model and dynamically create nodes and arcs, and allows seamless combination of recognition languages of different kinds, such as n-gram language models and rule-based regular grammars. On the other end, it has a typically larger memory footprint, given by the need of keeping in memory a description of the whole network, even if only a portion of it is actually explored during the search, due to the pruning introduced by beam-search. To mitigate the impact of this choice, several techniques are used:

- When dealing with large n-grams models, an LM-pruning procedure is applied, based on the *weighted difference* criterion [5], that is able to significantly reduce the number of n-grams without adversely affecting recognition accuracy.
- The LM is compiled in a tree-like structure with probability factorization and a *shared-tail* topology [6], that removes much of the redundancy.
- The resulting network is further reduced by applying an iterative reduction technique that, unlike the standard optimization procedure, is suboptimal, but can be applied to non-deterministic graphs. The algorithm exploits the equivalence of adjacency lists on nodes, and is typically able to reduce the graph size by around 15%.
- Finally, the network is modified by introducing special "multi-arcs" with a single label to represent sequences of arcs with unique input and output nodes. These "chains" appear

rather frequently in the graphs derived from compilation of LMs, and their substitution with multi-arcs typically reduce the size by more than 30%.

Moreover, when more instances of the decoder run on the same machine with the same LM, the static graphs can be loaded in shared memory, reducing the memory load.

2.1.2. Dynamic structures

All information needed to describe the search status is dynamically allocated and attached to nodes and arcs activated during the search.

Status information for each full arc include an output likelihood, a backpointer referring to the previous step along the best path, and a local trellis that describes the status of each node in that instance of the associated structure, that, as explained before, may be an HMM (see Fig. 1), an HMM chain, or another network . Status information for each network node includes the likelihood and a backpointer to the best incoming arc. Null arcs do not have any associated status information, as the scores and backpointers are directly propagated between their start and end nodes. In the local trellises that correspond to HMM instances, backpointers are only propagated from the upper layer to internal HMM states, and then provided as output for the arc. No record is kept for the details of the paths within the acoustic models.

Backpointers are kept in a tree structure, in which each item contains a reference to its predecessor. This improves efficiency of the backtracking procedure that is carried out several times per second with the purpose of garbage collection: identification of needed backpointers and removal of backpointers that have lost any reference to them due to pruning.

Structures like local trellises and backpointers are subject to very frequent activation and disposal; their allocation is thus made through special structures. Backpointers have fixed size; hence they are located in a paged array that can be incrementally extended, but also has an associated free list for quick reclaim and disposal of already allocated items. References to them are actually indexes in the paged array. Local trellises, instead, have varying sizes, that depend on the object they correspond to. Their allocation is therefore made through a cache that reuses memory from a pool of allocated but unused trellises, applying a best-fit criterion.

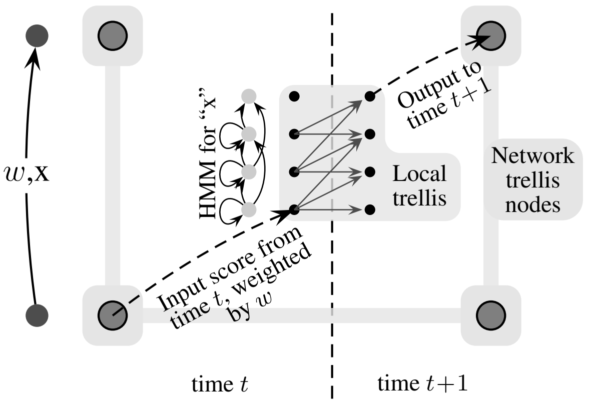

Figure 1: *Score propagation between trellis layers, for a network arc labeled with "x", and weighted by* w.

2.2. Algorithm

The decoding procedure is essentially a time-synchronous Beam-search Viterbi algorithm, at each time instant expanding the hypotheses reaching a set of "active" nodes. This set consists of all nodes whose accumulated likelihood so far differ less than a threshold, the *beam-width*, from that of the node with the highest likelihood. The beam-width can be dynamically adjusted if the number of active nodes exceeds a maximum value given as an option.

Initially, the set of active nodes contains the unique initial node of the main network, and the set of active arcs is empty. Then, for each time instant (each frame) the following operations are performed:

1. *Adjustment of beam threshold*: if the number of active nodes exceeds a predetermined number, a node histogram is computed according to their likelihood, and the beam threshold is adjusted so as to expand, in the following stage, a number of states not larger than the specified value.
2. *Full arcs activation*: all arcs exiting from an active node whose score exceeds the beam threshold are added to the list of active arcs, if they are not already present.
3. *Full arcs expansion*: for each arc in the list of active arcs, an update of the local trellis is performed. This involves carrying network scores and backpointers into the local trellis, weighted by the arc weight, updating the scores for the local trellis for the next time instant, and preparing the arc output score, to be used in the next stage (see Fig. 1). If an arc is referring to another network, instead of to an HMM, a recursive call to the same procedure is made, to advance one time frame in the updating of the sub-network, and to prepare the sub-network output for recombination on the current one. In that case the "local trellis" will be a layered network trellis in itself. An arc may be removed from the active list, if all internal nodes happen to be deactivated by beam pruning. In this stage, a global maximum of likelihood values is also updated, to control node pruning at the next time instant.
4. *Full arcs score recombination*: the list of active nodes is emptied. The output scores of arcs are combined on the network trellis, and a new list of active states is built.
5. *Empty arcs score recombination*: score combination and backpointer propagation is performed along null arcs, that do not involve acoustic probability computations, possibly extending the list of active nodes.
6. *Garbage collection*: at specified intervals, typically few times per second, unneeded backpointers are removed by means of a *mark-and-sweep* procedure.

At the end of the frame sequence, the best path is found as usual by finding the node with highest accumulated score among those marked as "final" nodes, and following the backpointers structure from there.

The decoder can also operate in "streaming mode", in which output is provided progressively, without waiting for the end of the segment. This is achieved by exploiting a known characteristic of Viterbi decoding, especially visible with n-gram language models: at any time in the process, all the backtrack paths of the active hypotheses meet at a single trellis node, typically located less than a second behind the current time. This means that all arc references before that time will be common to all hypotheses, and can therefore be considered fixed. The check for this condition is carried out few times per second, and, if a collapsing point is found, the preceding portion of the path is delivered as output. The process is exact, in the sense that the overall best string is exactly the same as if the decoder performed the usual global backtrack at the end of the audio.

3. The parallel implementation

Among the different stages described in the previous section, there is one that is dominant in terms of computing load, namely the "full arcs expansion". This is the stage in which all acoustic likelihood computations are carried out, together with path

recombination on an often large number of small trellises, corresponding to HMM instances in different contexts. It has been observed that, in a typical setup for real-time live recognition, these operations account for more than 75% of the total computation time. Moreover, each one of those "computing tasks" depends only on local information: it takes the status of the source network arc, possibly updating the input of the local trellis if that node has a likelihood higher than the already present hypothesis, and propagates scores using only the local trellis, the HMM parameters and the current acoustic observation (Fig. 1). These tasks can be executed in parallel without requiring synchronization. Therefore, the basic design choice was to distribute as efficiently as possible the corresponding load to several worker threads, that are started at the beginning of the processing, and execute the following loop:
Token expansion:
1. Wait for a token describing a local trellis to update.
2. Perform the local update.
3. Signal the completion of the task, and go back to 1.

The flow of the processing is governed by the main thread. With respect to the processing stages described in Sec. 2.2, the modification concerns operations 3 to 5, that, as executed in the main thread, become:

3'. *Token dispatching*: traverse the list of active arcs, creating a "task token" for each active arc, and dispatch them to worker threads. This operation deals with a recursive structure but, being simpler than the arc expansion in stage 3 of Sec. 2.2, it can be performed without recursive calls.

4'. *Token output recombination*: as full arc outputs are made available by the worker threads, use the scores to update the network status on the proper network or sub-network trellis. When all active full arcs on a network or sub-network have been processed, also perform empty arcs recombination. If it is a sub-network, provide the output for full arc recombination at the upper level.

Communication between threads (dispatching of tokens and delivery of results) is made by means of "custom" queues. Thread creation and synchronization is based on the `pthreads` library. However, only basic primitives of the library are used, so that porting to other environments can be achieved with limited effort. During development, it was found that the synchronization and data movement needed in accessing the queues could introduce undesired overhead, and hence special care has been taken to minimize it, as described in the following paragraphs.

Token management. A token is a structure that contains:

- Network instance identifier.
- Reference to the arc in the network.
- Maximum value found in the local trellis.
- Reference to the "parent" token.

The first two fields are used in input, as they contain all the necessary information for the worker thread to perform the local trellis update. The third field is set by the worker thread, and is used by the main thread to update the reference for the beam threshold. The fourth field is set by the dispatch procedure, and used by the global recombination procedure to manage the hierarchy of sub-network scores. Tokens have fixed size, and are thus allocated and disposed of by means of an efficient paged array similar to the one used for backpointers. The queues are fed with references to tokens, i.e. 32bit integers, and not with the data themselves. The completion of a token is signaled by putting a reference to it in the output queue.
Dispatch queue. Considering a single token as a dispatch unit has been found to be not convenient; hence tokens are dispatched in packets of configurable size (e.g. 5) so that once a thread gets a task, it has some time to work "full-time" on it. Given the considered architecture, the proper structure would appear to be a single-writer/multiple-reader FIFO, but this requires tight synchronization. To remove the need for synchronization between readers, several thread-specific queues are setup. The main thread, while traversing the list of active arcs, collects tokens up to the desired amount, sends them as a packet on a worker's queue, and prepares to send the next packet to another thread, in a round-robin fashion. The thread-specific queues are therefore single-writer/single-reader. For this kind of queue, there is the possibility of avoiding the need for mutual exclusive access, saving the use of a mutex, and resorting to library synchronization primitives only when a thread must be suspended to wait for the arrival of data.
Output queue. Considerations similar to those of the preceding paragraph can be made with respect to the output queue. This is the queue in which worker threads insert token references as soon as they have completed the related computation, and is read by the main thread to perform global score recombination on the network. In this case, a special kind of multiple-writer/single-reader queue has been implemented: each writer thread has a private buffer, that it can access without mutual exclusion, but all of them use the same binary semaphore to signal the presence of data. In this way, the reading thread needs only to wait for one semaphore to be set, and then examine all the buffers for the presence of data.
Global data structures. As mentioned in Sec. 2.1, local trellises are subject to frequent allocation and deactivation, and are thus managed through a cache that keeps unneeded trellises and reassigns them under request, applying a best-fit strategy according to size. As this kind of allocation is performed exactly in the worker threads, this could lead to a high degree of contention in access to a global structure. To avoid these, separate caches are kept for each thread, even though they lead to an increase in memory usage.

Another structure accessed when performing HMM trellis expansion is a cache of acoustic density values. HMM densities are computed on demand, as they are requested by the expansion of active nodes. Each computed value is stored in an array indexed by density, so that it can be found there instead of being recomputed, if needed again. Empty slots in the cache are marked by a special *nan* value, to which all slots are reset at the beginning of the processing of each frame. Also in this case the choice was that of avoiding synchronization, since the worst thing that may happen, in case of concurrency in cache access, is that the same density is computed more than once in a frame by different threads.

Other global data structures are modified only by the main thread.

4. Results and discussion

4.1. The recognition system

The acoustic models used in the test are 17,260 three-state left-to-right within-word triphone HMMs, composed from a set of 12,510 shared states, plus a few models for spontaneous speech phenomena. State tying is defined by means of a Phonetic Decision Tree. Gaussian tying follows a phonetic tying scheme, that is, all allophones of the same phoneme share, in principle, the same pool of Gaussian components. During training, Gaussians can be detached from mixtures by pruning; hence the mixtures have a variable number of components, with an average of 139 components/mixture, and a total of 52784 shared Gaussians. Acoustic features are 13 MFCC plus first, second and third derivatives, projected in a 39-dimensional feature space by means of an HLDA transformation. Models were trained on 220h of speech, of which 130h coming from broadcast news, and the remaining from automatically transcribed political speeches.

The language model is a "general purpose" 4-gram LM with a vocabulary of 150k words, trained on a corpus of ≈ 1.2 billion words of texts mostly coming from newspapers archives, and compiled in a network with 70M nodes, 74M full arcs and 141M null arcs. In this case, recursion in the network is used for the handling of spontaneous speech phenomena: the "silence" symbol, that can optionally occur between any two adjacent words, refers to a small sub-network of non-speech models that includes the most common filler words, in addition to silence and background noise. This sub-network is essentially a weighted loop of a dozen of symbols, but has more that 12 millions references in the main network. If it was statically linked into it, it would considerably increase its size.

4.2. Results

Table 1 reports timing measurements comparing the serial reference implementation, and the multithreaded implementation, with 2 and 4 threads, at different beam-widths.

The test set consists of three portions of sessions of the Italian Chamber of Deputies, for a total of 2h:36m:35s of actual speech. The Word Error Rate (WER) on this test is 12.35% at the lowest beam-width.

Real time ratios are computed as the ratio between the total elapsed time for processing the audio and the length of the actual speech. The experiments were run on a computer running linux, whith a quad-core Intel i7 870 processor and 8Gb of RAM, that was of course free of other demanding jobs. The peak resident memory usage of the processes was around 3Gb, with an overhead of $\approx 5\%$ for the multithreaded version.

	Time (h:mm:ss)	*Real time ratio*	*Speedup*	*"Amdahl score"*
Beam=10^{-45}, *WER=12.49%*				
Serial	1:30:55	0.58	-	-
2 threads	0:58:15	0.37	1.56	0.72
4 threads	0:41:16	0.26	2.20	0.73
Beam=10^{-50}, *WER=12.41%*				
Serial	1:58:15	0.76	-	-
2 threads	1:15:23	0.48	1.57	0.73
4 threads	0:53:16	0.34	2.22	0.73
Beam=10^{-55}, *WER=12.37%*				
Serial	2:28:56	0.95	-	-
2 threads	1:36:04	0.61	1.55	0.71
4 threads	1:07:57	0.43	2.20	0.73
Beam=10^{-65}, *WER=12.35%*				
Serial	3:41:00	1.41	-	-
2 threads	2:23:36	0.92	1.54	0.70
4 threads	1:43:06	0.66	2.14	0.71

Table 1: *Timing comparison between serial and multithreaded implementation.*

The table also reports an "Amdahl score" derived from Amdhal's law [7], a relation that sets an upper bound on the performance improvements achievable through the parallelization of an algorithm. It is computed as $a = \frac{n(s-1)}{s(n-1)}$, where n is the number of threads, and and s is the speedup. It is interesting to note that, at a beam-width of 10^{-45} and 4 threads, which is the actual configuration used in live transcription, the score is 0.73, corresponding to an "optimal" parallelization of 73% of the total computation load, which is in accordance with the observation reported in Sec. 3, taking into account that it corresponds to a real parallelization of a somewhat larger fraction. Therefore, it appears that the goal of correctly parallelizing the most time-consuming portion of the processing has been achieved. The reported speedups are slightly higher than those reported in [2], in which the reference algorithm is a lexical-tree based search algorithm, but the operating environment is quite similar to the one used here. Better speedups are reported in [1], where the intervention on the code is deeper, and the goal is to parallelize as much as possible of the processing. However, both hardware and software environments are considerably different. In conclusion, the achieved results can be considered satisfactory, but a further improvement is expected by introducing parallelism also in the global recombination stages.

4.3. Deployment

The decoder described here is currently employed in a prototype in use by the FBK spin-off company *PerVoice* (`http://www.pervoice.it`) for supporting the live subtitling of TV news programs and talk shows. The subtitling system relies on a combination of live transcription and respeaking: the broadcast speech is directly fed to the transcription system, whose output is then controlled, corrected and sent "on air" by a human operator. When the audio conditions become prohibitive, as may happen with overlapped or very noisy speech in talk shows, so that the direct output of the transcriber becomes unreliable, then the operator can switch to respeaking, where she dictates the subtitles. The system therefore uses two instances of the decoder, one running on the direct TV audio, with a speaker-independent acoustic model, and one connected to a head-mounted microphone, using a speaker-adapted acoustic model. The decoders both operate in streaming mode, as described in Sec. 2.2, and run on the same 8-core machine, with 4 threads allocated to each one. They share the language model, and thus load the recognition network in shared memory, so as to reduce resource usage.

5. Conclusion and future work

This paper presented the parallelization of a speech decoder, based on the principle of efficiently distributing independent computations on different threads. The serial algorithm and its parallel extension are described, and results are provided to show that the approach, even if not comprehensive, is able to provide appreciable speedups. Further work will be devoted to extend the parallelization to other stages of the algorithm.

6. References

[1] S. Phillips, A. Rogers, *"Parallel Speech Recognition"*, International Journal of Parallel Programming, Vol. 27, no. 4, pp. 257-288, August, 1999

[2] N. Parihar, R. Schlüter, D. Rybach, and E. A. Hansen, "Parallel Lexical-tree Based LVCSR on Multi-core Processors", Proc. INTERSPEECH, pp. 1485-1488, Makuhari, Chiba, Japan, 2010

[3] J. Chong, E. Gonina, Y. Yi, and K. Keutzer, "A Fully Data Parallel WFST-based Large Vocabulary Continuous Speech Recognition on a Graphics Processing Unit", Proc. INTERSPEECH, pp. 1183-1186, Brighton, UK, 2009

[4] P. Cardinal, P. Dumouchel, and G. Boulianne1, "Using Parallel Architectures in Speech Recognition", Proc. INTERSPEECH, pp. 3039-3042, Brighton, UK, 2009

[5] K. Seymore, R. Rosenfeld, "Scalable backoff language models", Proc. ICSLP, Vol. 1, pp. 232-235, Philadelphia, USA, 1996

[6] F. Brugnara, M. Cettolo, "Improvements in Tree based Language Model Representation", In Proc. EUROSPEECH, pp. 1797-1800, Madrid, Spain, 1995.

[7] G. Amdahl, "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities", In Proc. AFIPS, Vol. 30, pp. 483-485., Atlantic City (NJ), 1967