Parallelization Strategies for a Dynamic Lexical Tree Decoder

Matthias Vogelgesang and Florian Metze

CMU-LTI-01-010

Language Technologies Institute
School of Computer Science
Carnegie Mellon University
5000 Forbes Ave., Pittsburgh, PA 15213
http://www.lti.cs.cmu.edu/

Parallelization Strategies for a Dynamic Lexical Tree Decoder

Matthias Vogelgesang and Florian Metze
`matthias.vogelgesang@gmail.com, fmetze@cs.cmu.edu`

Abstract

Increasingly, physical limitations lead to a shift from high clocked single core processors to CPUs with up to eight, or more, independent but slower processing cores, and multi-core or even multi-CPU computers. In order to retain performance gains in the future, the speech decoding process has to be re-organized to employ a certain amount of thread-level parallelism on those CPUs. In this work, we compare two common approaches for dynamic prefix tree decoders: Parallel Score Computation and Parallel Search, and a combination of both. Both have already been studied intensively, however it is shown here, that the latter suffers from hardware cache effects which limit absolute speed-ups and scalability in general. We propose a cache efficient variation of the Parallel Score Computation which is more scalable and faster than any other parallel strategy we compared it with.

Index Terms: speech recognition, parallel processing

1 Introduction

Parallelizing decoders of large vocabulary continuous speech recognition (LVCSR) systems has seen great interest recently, due to the fact that speed gains cannot easily be obtained by buying faster hardware. In current and upcoming general purpose hardware, parallelization has to be exploited by rewriting the software to run in a thread-level concurrent fashion.

Some early ideas on how to organize parallel speech decoders were published by Ravishankar [1] and Phillips et al. [2], although Ravishankar's implementation is limited to the rare PLUS microprocessor and Phillips et al. parallelized a WFST-type speech decoder for Challenge processors.

Ishikawa et al. ported their decoder to a three-core handheld ARM CPU architecture [3] using a blockwise pipelining approach. Their embedded hardware imposed tight restrictions on the design of the software architecture. Therefore, it is not very scalable nor applicable to most general LVCSR systems. GPU or FPGA implementations which offload acoustic model score computations to dedicated processing units showed very good speed-ups [4, 5] but these specialized units are not necessarily available in a general systems either. Furthermore programming them for legacy systems is sometimes impossible.

Parihar et al. ported the Mississippi State Decoder to a parallel tree-division architecture [6, 7, 8] and achieved modest speed-ups. The interesting novelty is the statistically driven root node assignment during the tree division. Similar to Parihar, You et al. [9] implemented a parallel speech decoder using OpenMP. Unlike the former approach, they created new copies of the prefix tree which were then assigned to different threads.

A common low-level parallelization scheme exploits the instruction level parallelism of vector instructions in modern microprocessors such as Intels MMX and SIMD Streaming Extensions (SSE) [10, 11] to concurrently calculate arithmetic operations during score evaluation.

While speed-ups greater than a factor of 10 have been achieved on some tasks, for full LVCSR and complex language models, no speed-up larger than 2 has been reported, with relatively low efficiencies (using a setup similar to ours, [8] for example achieves a speed-up slightly below 2, with efficiencies around 0.5).

In this paper, we will present our implementation of parallelization based on lexical tree division, and present results that speed-ups are achieved overwhelmingly due to speed-ups in acoustic score computation, and cacheing effects.

Overall, parallelization is still a viable strategy, but the pruning required particularly for large systems makes it difficult to efficiently parallelize the actual search phase. Splitting the calculations into two distinct phases of score computation (computed almost as part of pre-processing) and decoding (which processes different utterances on different cores) presents itself as the most general strategy, particularly if the score computation can be performed on GPUs.

2 Parallelization Strategies

2.1 Baseline: Pronunciation Prefix Tree Decoder

The "Ibis" decoder [12] uses a time-synchronous search based on Hiden Markov Models (HMMs) and a single copy of the pronunciation prefix tree (PPT) with dynamically allocated instances of nodes and early path recombination using the full language model information.

In each node of the PPT, we keep a list of linguistic morphed instances. Each instance stores its own backpointer and scores for each state of the underlying HMM with respect to the linguistic state of this instance. Since the linguistic state is known, the complete language model information can be applied for all possible successor words for that node in the PPT.

The advantage of this search space organization is that beam and topN pruning can be applied very easily and path recombination (which is usually done at the word ends) can be performed as soon as the word becomes unique, which is usually a few phones before reaching the leaf.

Using a division of of the search space into sub-trees, starting from different root nodes, all these operations can naturally be performed in parallel, and using local information only, which facilitates memory access in parallel architectures and reduces the need for synchronization between threads. Work can be scheduled with a simple round-robin strategy.

2.2 Parallel Score Computation

Computing the likelihoods for all necessary acoustic models in a certain frame has a regular structure and can be easily parallelized [13, 9]. Generally, speech decoders request acoustic models on a per-frame basis and calculate the corresponding scores in each step. Since it is very likely that mixtures for an acoustic model which are requested for a given frame are also needed in the near future, the score can also be computed for the next k frames and stored in a sliding window cache. Let $t_a(n)$ be the time needed to calculate the acoustic model scores for n acoustic models, then the sequential time can be approximated by $t_s \approx t_a(n_f \cdot k) + t_v(n_f)$, where n_f is the needed number of acoustic models in a given frame f and t_v is the time for the Viterbi search.

The parallel score evaluation or *Cache Parallelism* with n_t threads can be approximated by $t_p \approx t_a\left(\frac{n_f \cdot k}{n_t}\right) + t_v(n_f) + t_o(k)$, where t_o is the threading overhead depending on the number of threads. Clearly, we achieve a higher speed-up $\frac{t_s}{t_p}$, when the overhead is low. Otherwise, Amdahl's law comes into play [14] which limits the maximum speed-up to $1/t_v(n_f) + t_o(k)$. The total execution times for decoding N frames can be approximated by

$$T_1 = \sum_{f=1}^{N} t_a\left(\frac{n_f \cdot k}{n_t}\right) + t_v(n_f) + t_o(n_t). \tag{1}$$

As far as we know, this technique is implemented in many current system, even if it is not mentioned explicitely. In our experiments, a cache width of 8 frames proved optimal.

We can reduce the threading overhead of $N \cdot t_o(n_t)$ time units and presumably improve the hardware cache hit-rate by computing all acoustic model scores for all frames upfront. The new approximated total execution time for our proposed cache-friendly parallelization strategy, which we call *Full Pre-computation* from here on, is

$$T_2 = t_a\left(\frac{N \cdot n_a}{n_t}\right) + t_o(n_t) + \sum_{f=1}^{N} t_v(n_f), \tag{2}$$

where n_a is the total number of acoustic models of the speech system. As one can easily tell, systems with small beams or compute-intensive language model (LM) lookups do not benefit very well from this kind of parallelization, since t_v could span a relatively large sequential fraction of the decoding loop. In the next section we sketch a common parallelization approach that deals with this problem.

2.3 Parallel Search

As shown in the preceding section, it is desirable to parallelize the whole decoding loop iteration instead of only the score computation. The most promising approach for dynamic tree decoders splits the pronunciation search tree at its meta-root and assigns word roots and the appropriate sub-trees to different threads [9, 6]. Different scheduling policies can be used, to decide which thread is assigned the next expanded root such as static round-robin dynamic load-balancing or assignment according to phonetic similarity as proposed in [6].

Similar to Equation 1, we can estimate the total execution time by

$$T_3 = \sum_{f=1}^{N} \underbrace{t_a\left(\frac{n_f \cdot k}{n_t}\right) + \frac{t_p}{n_t}}_{\text{parallel}} + \underbrace{t_o(n_t) + t_c(n_f)}_{\text{sequential}}. \tag{3}$$

However, we now have code paths that need to be synchronized with critical sections or barriers. Therefore, it is crucial to keep these parts as short as possible. The decoder has to execute these parts sequentially for a total time of t_c.

The following advantages of *Parallel Search* [7] are also true for our case of a single PPT search space organization: (a) the acoustic model score computation is implicitly parallelized, (b) thread communication can be reduced because lexical tree branches are inherently independent from each other which minimizes communication between the corresponding threads, and (c) the changes compared to an unmodified serial decoder are simple and minimal. In Section 3, we will show that these assumptions are true, but will not necessarily lead to a better performing parallel decoder.

2.4 Combination

The two previous strategies can be *Combined*, by first pre-computing all acoustic scores and storing them in the cache, and then running the search in parallel, replacing the parallel score computation with a simple table look-up.

3 Experiments

We conducted experiments on the presented approaches using an Intel Core i5 750 with four identical cores and a processor base clock frequency of 2.67 GHz. Each core has an independent 16 KiB instruction and 16 KiB data L1 cache, as well as a 256 KB large general purpose L2 cache. All cores share a single, dynamically allocated 8 MB L3 cache.

A second set of experiments was run on an 8-core SMP Dell computer with two Intel Xeon E5320 processors, featuring a base clock frequency of 1.86 GHz and 4 MB L3 cache. Ibis was compiled with gcc 4.4.3 and optimized flags like -03 and x86-64 architecture specific flags to enable SSE3 support.

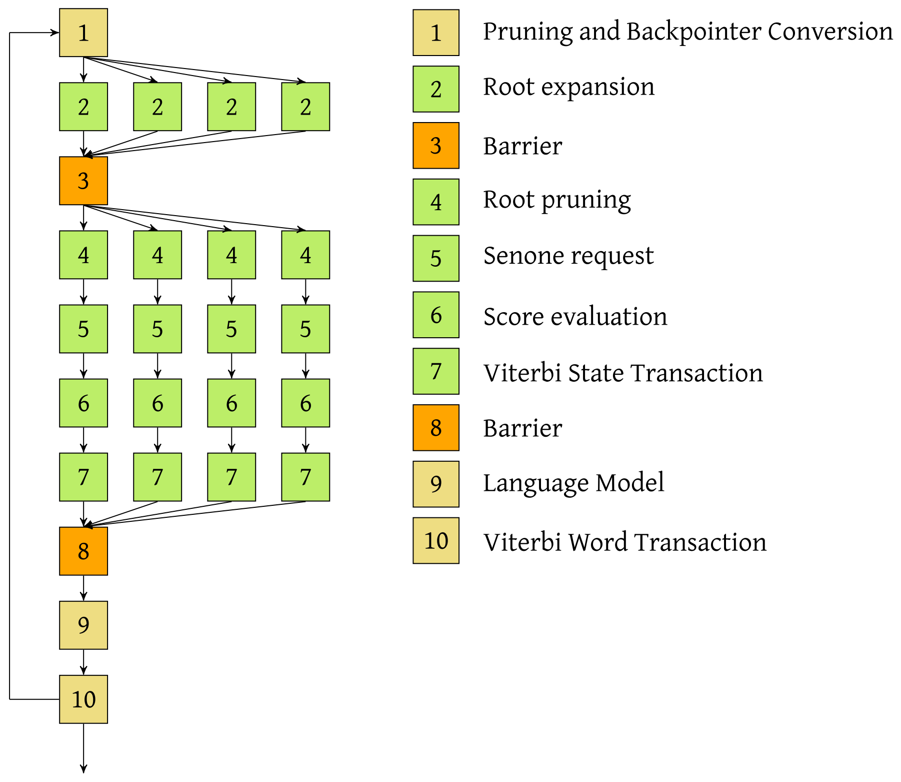

Figure 1: The OpenMP "fork-join" programming model applied to the Ibis decoder. Relatively few synchronization points are required compared to related work.

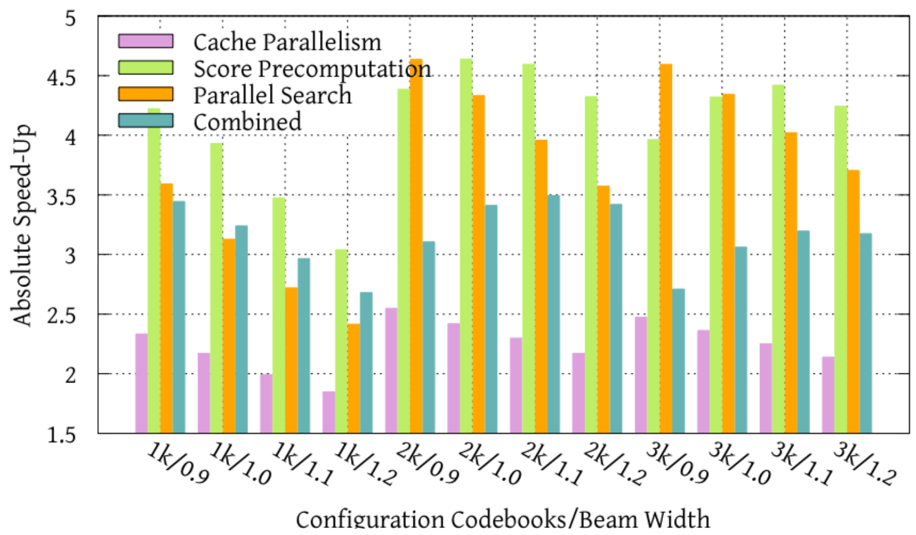

Figure 2: Speed-up on 4-core i5 machine using different speech system parameters and four threads.

We modified the source of the Ibis speech decoder [12] from the Janus Speech Recognition Toolkit to implement the strategies given in Section 2 using the OpenMP API[1] and its fork-join programming model as shown in Figure 1.

Language model access could be parallelized as well, but the implementation is complicated by the use of a cache structure, which involves read and write access to memory at a fine granularity, so this has not been attempted so far.

We analyzed the execution on a variety of environment configurations. The independent variables of our experiment were number of acoustic models 1000, 2000 and 3000, the number of threads ranging from 1 to 4, four different beam settings and the actual strategy. The test database consisted of 10 minutes of clean German WSJ-type speech. The acoustic model was trained on 14 h of matching audio, the vocabulary was 5 k words, and a 3-gram LM was used.

3.1 Speed-Up

We calculated the maximum attainable speed-up by dividing the run-time measured with the sequential base decoder by the run-time measured with four threads and for all parameter combinations. All reported timings are averages over 5 runs. In Figure 2, the speed-up for all combinations are shown. We can see that the Score Precomputation's speed-up excels in most situations and even generates a super-linear speed-up. With bigger speech systems the Parallel Search reaches the speed-ups of the Pre-Computation and exceeds them in some cases. The Cache Parallelism is good, yet not very effective. The com-

[1]`http://www.openmp.org/`

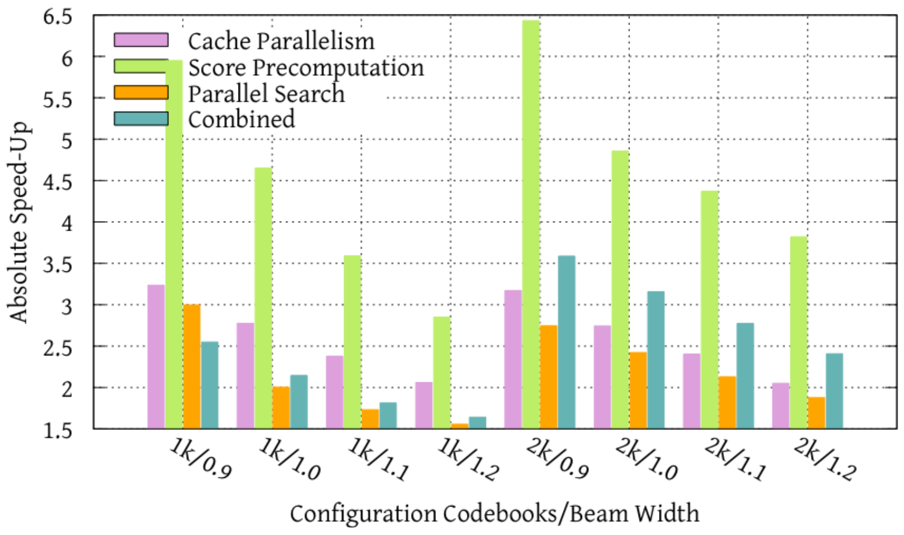

Figure 3: Speed-up on dual-processor 8-core machine using different speech system parameters and eight threads.

Table 1: Comparison of different approaches using 4 threads on a Intel i5 Quad-core and a sliding window cache with a width of 8 frames.

Approach	Time (s)	Speed-Up	
Single score w/o cache	759	(1)	-
Single score w cache	675	1.12	(1)
Cache Parallelism	272	2.79	2.48
Score Precomputation	125	6.07	5.40
Parallel Search	140	5.42	4.82
Combined	134	5.66	5.03

bined approach of Pre-Computation and Parallel Search is, except for the 1k system, worse than either of those strategies. Figure 3 shows the corresponding speed-ups for the 8-core machine. The 3k codebook case was not tried for time constraints.

3.2 Scalability

Table 1 summarizes our results. As shown in Figure 4, all strategies scale to a certain degree, regardless of the configuration. For two threads, the Parallel Search is the best approach, which is not obvious when comparing the graphs with the maximum speed-ups from Section 3.1. This observation is useful under the premise that on a shared machine with mixed processes not every core might be available all the time and therefore the Parallel Search may be more

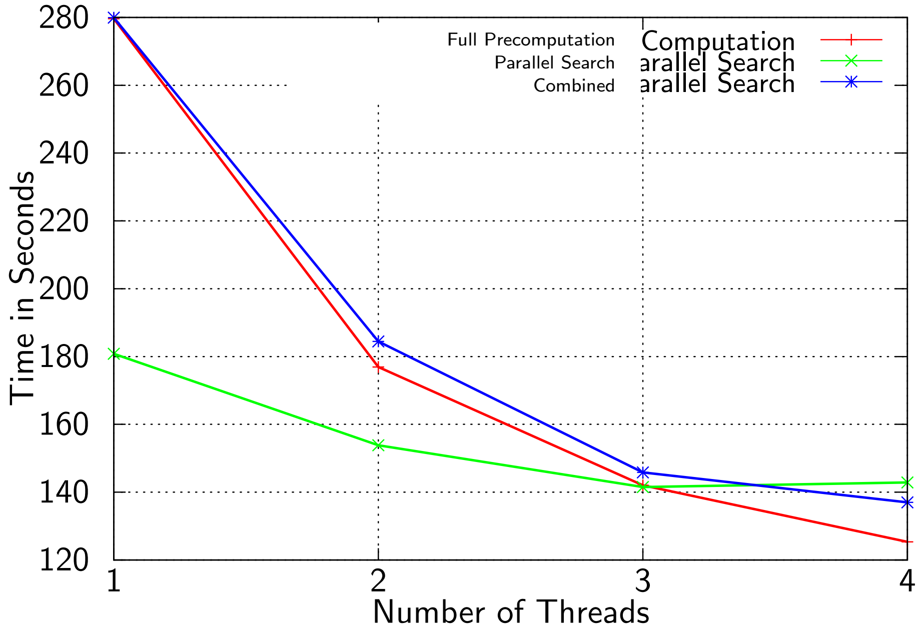

Figure 4: Scalability of the approaches. For more than 3 threads, *Full Precomputation* outperforms the other approaches. While *Parallel Search* shows nice gains, the *Combined* strategy being behind *Full Precomputation* shows that the gains are in fact due to improved score computation, not parallelized search.

Table 2: Mean overhead in milliseconds. The numbers correspond to the blocks in Figure 1.

Block	Construct	Time (ms)
3	Barrier	11.1
8	Barrier	21.5
9	Critical Section (LM)	48.0
10	Critical Section (Viterbi Word)	11.2

beneficial.

3.3 Discussion

In this work, we present and discuss results which we achieved on a typical “small” speech recognizer, which could be handled comfortably on a single machine dedicated to these experiments only. We indicate the changes we observed for smaller or larger systems. As in comparable work, we were able to achieve speed-ups up to a factor of about two, with relatively low efficiency. Our main findings are that: (a) the sub-tree division approach parallelizes the single-tree based search quite well when compared to other, similar work using tree copies; (b) The comparison of three approaches shows that the speedups are overwhelmingly due to the speedup of the score computation using a frame cache for acoustic score computations; and (c) However, it is possible to achieve even super-linear speed-ups under certain conditions, due to cache effects.

Unless low latency (i.e. continuous generation of hypothesis with as little delay as possible) is a requirement, the best speed-up (and the best efficiency) can therefore be achieved by computing all acoustic model scores in advance, maybe even on a GPU, and parallelizing the search conventionally, by decoding individual utterances in individual threads or processes, using one core per utterance.

Table 2 shows the parallelization overhead. At this point, the barriers show comparable overhead to the largest remaining critical sections, which is the reason why further speed-ups are hard to achieve in a parallel search based approach. In this work, we implemented and compared the load-balancing mechanisms describes in [7]. We achieved our best results with a static round-robin assignment strategy. *Parallel Search* does not perform as well as *Full Pre-Computation* because too many parts of a loop iteration are still executed sequentially due to synchronization and thread management, although the latter is negligible compared to the synchronization [15]. We measured the total number of CPU cycles for the distinct phases of serial and parallel execution using the PAPI[2] performance counter framework. In a typical 4-core setting on our test configuration, a considerable cumulative amount of 26 % of the code is still sequentially executed, which limits the maximum speed-up.

[2]`http://icl.cs.utk.edu/papi/`

4 Conclusion

Our experiments (more details can be found in [16]) show that the sub-tree division approach works well for a lexical tree based decoder based not on multiple tree copies, but on a linguistic polymorphism of active nodes.

We can draw the following conclusions from our experimental results: *Score Precomputation* shows the best speed-ups in almost all cases, and should use the maximum number of cores available. When using more acoustic models and wider beams, the *Parallel Search* exceeds the performance of the *Precomputation* based approaches. In general, the number of acoustic models correlates positively with the performance whereas the beam width correlates negatively with the performance.

The last point sound contradictory. However, an increase of the beam width not only increases the amount of acoustic models to be computed but also the number of LM lookups and word recombinations whereas an increase of the codebook size does not affect aforementioned sequential code paths. Therefore, an integration of run-time algorithms that decide which strategy to pursue in order to optimize actual speed-ups, might be useful.

Therefore, many more factors than only the parallelization strategy influence the performance of a parallel decoder: The size of the speech system in terms of codebook size and beam width, software architecture of a legacy decoder, hardware cache sizes and so on. In our experience, in order to construct efficient parallel speech decoders, hand-optimization or an auto-tuning scheme is crucial.

5 Acknowledgements

The first author of this work was funded by *Landesstiftung Baden-Württemberg* (*InterACT*) and the Cisco Research Center while at Carnegie Mellon University.

References

[1] M. K. Ravishankar, "Parallel implementation of fast beam search for speaker-independent continuous speech recognition," Indian Institute of Science, Bangalore, Tech. Rep., July 1993.

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

[3] S. Ishikawa, K. Yamabana, R. Isotani, and A. Okumura, "Parallel LVCSR algorithm for cellphone-oriented multicore processors." in *ICASSP '06: Proceedings of the 2006 IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 1. Toulouse; France: IEEE, 2006, pp. 177–180.

[4] J. Chong, E. Gonina, Y. Yi, and K. Keutzer, "A fully data parallel WFST-based large vocabulary continous speech recognition on a graphics process-

ing unit," in *ICASSP '09: Proceedings of the 2009 IEEE International Conference on Acoustics, Speech and Signal Processing.* Taipei; Taiwan: IEEE, 2009.

[5] T. Fujinaga, K. Miura, H. Noguchi, H. Kawaguchi, and M. Yoshimoto, "Parallelized viterbi processor for 5000-word large-vocabulary real-time continuous speech recognition FPGA system," in *Proc. INTERSPEECH.* ISCA, 2009, pp. 1483–1486.

[6] N. Parihar and E. Hansen, "A lexical-tree division-based approach to parallelizing a cross-word speech decoder for multi-core processors," in *EUSIPCO 2008: Proceedings of the 16th European Signal Processing Conference*, August 2008.

[7] N. Parihar and E. A. Hansen, "Analysis of a parallel lexical-tree-based speech decoder for multi-core processors," in *EUSIPCO 2009: Proceedings of the 17th European Signal Processing Conference*, 2009.

[8] N. Parihar, R. Schlüter, D. Rybach, and E. A. Hansen, "Parallel lexical-tree based LVCSR on multi-core processors," in *Proc. INTERSPEECH.* ISCA, 2010, pp. 1485–1489.

[9] K. You, Y. Lee, and W. Sung, "OpenMP-based parallel implementation of a continuous speech recognizer on a multi-core system," in *ICASSP '09: Proceedings of the 2009 IEEE International Conference on Acoustics, Speech and Signal Processing.* Taipei; Taiwan: IEEE Computer Society, 2009, pp. 621–624.

[10] S. Kanthak, K. Schütz, and H. Ney, "Using SIMD instructions for fast likelihood calculation in LVCSR," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing ICASSP*, April 2000, pp. 1531–1534.

[11] N. Parihar, R. Schlüter, D. Rybach, and E. A. Hansen, "Parallel fast likelihood computation for LVCSR using mixture decomposition," in *Proc. INTERSPEECH.* ISCA, 2009, pp. 3047–3050.

[12] H. Soltau, F. Metze, C. Fügen, and A. Waibel, "A one-pass decoder based on polymorphic linguistic context assignment," in *Automatic Speech Recognition and Understanding, 2001. ASRU '01. IEEE Workshop on*, 2001, pp. 214–217.

[13] P. Cardinal, P. Dumouchel, and G. Boulianne, "Using parallel architectures in speech recognition," in *Proc. INTERSPEECH.* ISCA, 2009.

[14] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proc. Spring Joint Computer Conference (AFIPS '67).* New York, NY, USA: ACM, Apr. 1967, pp. 483–485.

[15] J. M. Bull, "Measuring synchronisation and scheduling overheads in OpenMP," in *In Proceedings of First European Workshop on OpenMP*, 1999, pp. 99–105.

[16] M. Vogelgesang, "Parallelization strategies for the janus speech decoder," Master's thesis, Faculty of Computer Science, Karlsruhe Institute of Technology, Karlsruhe; Germany, Sep. 2010.