PARALLELIZING WFST SPEECH DECODERS

Charith Mendis *

Massachusetts Institute of Technology

Jasha Droppo, Saeed Maleki
Madanlal Musuvathi
Todd Mytkowicz, Geoffrey Zweig

Microsoft Research

ABSTRACT

The performance-intensive part of a large-vocabulary continuous speech-recognition system is the Viterbi computation that determines the sequence of words that are most likely to generate the acoustic-state scores extracted from an input utterance. This paper presents an efficient parallel algorithm for Viterbi. The key idea is to partition the per-frame computation among threads to minimize inter-thread communication despite traversing a large irregular acoustic and language model graphs. Together with a per-thread beam search, load balancing language-model lookups, and memory optimizations, we achieve a $6.67\times$ speedup over an highly-optimized production-quality WFST-based speech decoder. On a 200,000 word vocabulary and a 59 million ngram model, our decoder runs at $0.27\times$ real time while achieving a word-error rate of 14.81% on 6214 labeled utterances from Voice Search data.

Index Terms— Parallel Viterbi, WFST Decoder, Large vocabulary.

1. INTRODUCTION

Modern-day Large-Vocabulary Continuous Speech-Recognition (LVCSR) systems are based on Hidden Markov Models (HMMs). They break the input utterance into a sequence of *frames*, each typically accounting for 10 ms of speech, and extract suitable acoustic-state scores per frame using Gaussian Mixture Models or, more recently, using Deep Neural Nets [1]. Subsequently, an acoustic model (AM) graph and language model (LM) translate the sequence of acoustic-state scores into the likeliest sequence of words that could have produced the utterance. While the score computation can be efficiently implemented in a GPU [2], efficient parallel algorithms for the rest of the decoding process remains elusive.

This paper describes a parallel Viterbi algorithm for HMM-based LVCSR systems. Its effectiveness arises from the fact that the processing of each frame is *synchronization-free* — using the structure of the AM graph, the algorithm partitions the computation among threads such that two threads do not race when updating a single node or edge. At the same time, the algorithm ensures that work is evenly load balanced among all the threads. Together with techniques for dynamically composing the LM graph without inter-thread communication and careful design of data structures, the parallel algorithm achieves a $5.14\times$ speedup over a state-of-the-art production-scale speech decoder on a machine with 12 cores with *no* loss in accuracy — the parallel implementation produces exactly the same result as the sequential baseline (word-error rate 14.73%). The performance can be further improved by using a per-thread version of the beam-search algorithm that eliminates barrier-synchronization between the frame processing and the pruning of likely candidates. This provides $6.67\times$ speedup with 0.08% loss in accuracy.

A LVCSR implementation can use this additional performance to either increase the decoding speed, especially when catching up with a burst of speech signals delayed by the network, or to improve the accuracy by employing larger models which would otherwise be infeasible. To demonstrate the latter capability, this paper evaluates the algorithm on a large acoustic model with 1.4 million edges and a language model with 59 million ngrams generated from a vocabulary of 200,000 words, and uses a beam search parameters that process roughly 200 thousand edges for each frame. The sequential baseline runs at $1.8\times$ real time, while the parallel version runs at $0.27\times$ real time and achieves an word-error rate of 14.81% on a 6214 labelled utterances from a Voice Search data set.

2. PARALLEL VITERBI ALGORITHM

2.1. WFST-based Speech Decoder

This paper parallelizes a dynamic Weighted-Finite-State-Transducer (WFST) based decoder implementation, but the techniques described generalize to other HMM-based LVCSR systems as well. While a pure WFST decoder would statically compose the AM and the LM graphs, the current implementation dynamically composes them in order to avoid the blowup in the graph size. For each frame, Viterbi maintains *tokens* at each edge in the AM graph. A token represents a potential decode of the input up to the current frame and there can be multiple tokens for each AM edge corresponding to different LM histories. The token processing in a frame proceeds in two phases. The *AM phase* follows *emitting* arcs from each AM edge to generate tokens for the next frame without changing the LM histories. If multiple tokens with the same LM history land on the same edge, the algorithm keeps the one with the best cost. This is called as *token recombination*. From the tokens generated in the AM phase, the *LM phase* follows *non-emitting* arcs, representing word boundaries, to generate more tokens for the next frame. Once again, this phase has to recombine tokens with the same LM history that land on the same AM edge. While doing this, some implementations might choose to keep the best n tokens to provide alternative plausible answers for the same input. The tokens generated from each of these phases is *pruned* based on a beam-search algorithm to only keep tokens that are likely to produce a good answer.

2.2. Parallel Algorithm

In theory, the Viterbi algorithm seems easy to parallelize — all the tokens in a frame phase (AM or LM) can be processed in parallel. The challenge is doing the token recombination efficiently. Figure 1 demonstrates this issue. Nodes A to F are states in the AM graph

*work performed while at Microsoft Research

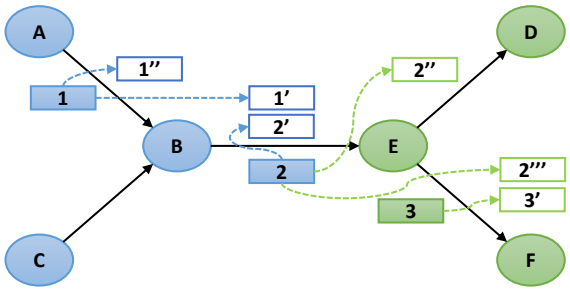

Fig. 1. Synchronization-free partitioning of token computation.

(the colors will be explained shortly) with tokens 1, 2, and 3 residing respectively on AM edges (A, B), (B, E), and (E, F). Assume these tokens have the same LM history. During the AM phase, these tokens will generate new tokens shown in unshaded boxes. In this example, tokens $1'$ and $2'$ land on the same AM edge (B, E). If tokens 1 and 2 are processed by different threads, then these threads have to synchronize, say by using a hardware interlocked operation, to determine the best token for (B, E) [3]. In our experience, any such synchronization in the performance-critical parts of a highly-optimized decoder is sufficient to negate any performance achieved by parallelism.

One way to avoid this problem is to revert to a "pull" model where each AM edge performs the recombination by reading tokens on its predecessor edges. However, in a beam search, only a small percentage of AM edges will have tokens in the current frame. Doing the recombination at every edge is unacceptably expensive. One can still attempt to identify *active* AM edges that will tokens landing on them, but doing so without synchronization or inter-thread communication is a challenge.

The parallel Viterbi algorithm solves this problem by using a *source-node-based* partitioning scheme. This scheme ensures that two threads will never race during recombination. Consider a partitioning of the AM states into two colors: blue (for A,B, and C) and green (for D, E, F) as shown in Figure 1. In the source-node-based partitioning scheme, a token on an AM edge (X, Y) gets the color of the source node X. In this scheme, all tokens that land on the same AM edge will have the same color, as shown in the figure. Now, we allocate the computation such that one thread is responsible for generating all tokens of a particular color. This ensures that there is no synchronization in a frame phase.

While the source-node-based partitioning ensures that each phase is synchronization-free, there can still be inter-thread communication across phases. Consider the edge (A, B) in Figure 1. Both the source and the destination states have the same color. Thus, tokens on this edge are *thread-local* — they are always generated and used by the same thread. In contrast, the source and the destination nodes of (B, E) have different colors. This means that token 2 generated by the blue thread is read in the next frame phase both by the blue thread as well as the green thread.

Such *cross-partition* edges are detrimental to performance for three reasons. First, the inter-thread communication across these edges results in increased traffic between caches/sockets. Second, these edges increase the net tokens read by all the threads — thread-local tokens are read once, while cross-partition tokens are read twice.[1] This increase in the working set of each thread results in poor cache performance. Finally, since outgoing edges of a node are consecutive in the memory, thread-local tokens can be read/written sequentially on a thread-local data structure allowing us to make effective use of the hardware prefetcher.

The next section describes a graph partitioning scheme that reduces the number of cross-partition edges.

2.3. Triphone-based Graph Partitioning

Now we consider the problem of partitioning the AM graph — in other words, assigning colors to AM states. Such a graph partitioning has two conflicting goals. First, as described above, we would like to reduce the number of cross-partition edges. An obvious way is to assign the same color to all the nodes. But, this conflicts with our next goal. To achieve parallel speedups, it is important to balance the load among threads. In other words, we would like to have roughly equal number of AM edges per color.

We use a triphone-based partitioning scheme that leverages the regular structure found in the AM graph. All valid paths through the decoding graph correspond to valid sequences of acoustic model states. In turn, each of these acoustic model states corresponds with either the first, second, or third state of a context-dependent triphone HMM. If a token occupies an arc belonging to the first state of a triphone HMM, we can guarantee that the next two times that token propagates forward, it will land on the second and third state of a related triphone HMM. In particular, all three states will be associated with the same context-*independent* phone that served as the basis of the context-dependent triphones. We utilize this property to devise a partitioning scheme that reduces the number of cross-partition edges.

The partitioning scheme works as follows. For each context-independent phone in the decoding graph, we find the relevant subgraphs containing all its triphones with different left and right contexts by doing a depth first traversal. Each triphone structure is explored until the beginning of the other triphones, and all the nodes and arcs traversed within the structure are grouped together to form the subgraph specific to that triphone. After this step, we have a set of triphone based subgraphs for each of the 42 phones, as well as any whole-word models present in the graph. This ensures that most edges are not cross partition edges.

The amount of parallelism available in multi-core systems is likely to be less than 42. Therefore, we assign a union of these triphone-based subgraphs to threads, guided by runtime profile information from training-set runs, to achieve equitable workloads. This load balancing is further refined dynamically, as described in Section 2.6.

On the Voice Search dataset used in the evaluation (Section 3), only 9% of the AM edges are cross partition with this scheme. In contrast, a naive partitioning approach that colors a AM state based on its node id (modulo total number of partitions desired), results in 75% cross-partition edges. To determine how this translates to runtime performance, we decoded the first 100 utterances from our data set running with 12 threads, and found that 24.66% of tokens are cross-partition at runtime as opposed to 47.68% cross-partition tokens with the naive approach. As cross partition tokens are read by two threads, this translates to a 30.54% reduction in the number tokens processed after accounting for pruning effects.

[1] An advantage of the source-node-based partitioning scheme is that each token is read at most twice per frame phase.

2.4. Token Recombination

With the source-node-based partitioning discussed above, all tokens that land on an AM edge are guaranteed to be processed by a single thread. But there is still a problem of dealing with tokens with different LM histories landing on the same AM edge. A straightforward way to do the recombination is to maintain a data structure per AM edge (say, a hashtable or a linear list) that maintains the current best token for each LM history. However, performing a data structure lookup to perform the recombination is expensive. Instead, our encoder achieves the effect of a word-conditioned decoder [4] by sorting the tokens based on their LM history at the beginning of the AM phase and processes tokens (per thread) in this order. Accordingly, a single pointer per AM edge to remember the best token for the current LM history is sufficient. This technique works for the AM phase as the generated token has the same LM history as the generating token.

However, sorting tokens by LM histories does not work for the LM phase as the generated tokens can have arbitrary LM histories. Fortunately, the LM phase generates far fewer tokens per frame when compared to the AM phase. This allows a different trick. The decoder redundantly generates tokens on an AM edge without recombination. At the end of the LM phase, the encoder sorts all these tokens by LM history and does a linear pass to compute the best token (or the best n tokens) for each AM edge. This has the additional benefit that it gets us one step closer to maintaining the token sortedness for the next AM phase. As tokens generated in the AM phase are already sorted by LM histories, one simply merges these tokens with the sorted tokens from the LM phase. Moreover, the redundant generation of LM-phase tokens means that the LM phase is embarrassingly parallel. In fact, the LM-phase computation in our parallel version is purely thread-local — each thread only traverses the tokens it generated in the previous AM phase. Finally, some of these optimizations speed up the sequential version of our decoder as well. We incorporate any such optimizations in our sequential baseline when calculating parallel speedups.

2.5. Beam Search

The beam search removes tokens that are a *beam-width* worse than the token with the best score. This width is dynamically adjusted to ensure that there is roughly a parameter-specified number of tokens per frame. Our sequential baseline performs a beam-based pruning once after the AM phase and once after the LM phase. In particular, it performs *online* pruning where the phases maintain the current best score and actively avoids generating tokens that will eventually be pruned.

There are two problems in parallelizing such an online pruning algorithm. First, maintaining a global-best score requires expensive synchronization that we avoid. If each thread locally maintains the best score it has seen, its online pruning will be less effective, unless it happens to be the thread that sees the globally-best token. This ineffectiveness results in the parallel version generating up to 3 times more tokens than the sequential version in our experiments. To alleviate this problem, our parallel decoder periodically exchanges the current best score seen by the threads. Such an exchange can be done with a *racy* update to a global best score with no synchronization. Nevertheless, frequent updates can result in expensive cache exchanges between the threads. Our current implementation performs this exchange once per ten tokens generated.

The second issue is that in order to retain the sequential semantics, one needs barriers between the AM phase, LM phase, and the two pruning steps. To eliminate these barriers we use a per-thread pruning algorithm that dynamically sets the beam-width such that there is roughly a parameter-specified number of tokens *per thread* as opposed to globally. This results in slightly less decoding accuracy. Our evaluation uses two parallel versions of decoders that has this optimization turned off or on.

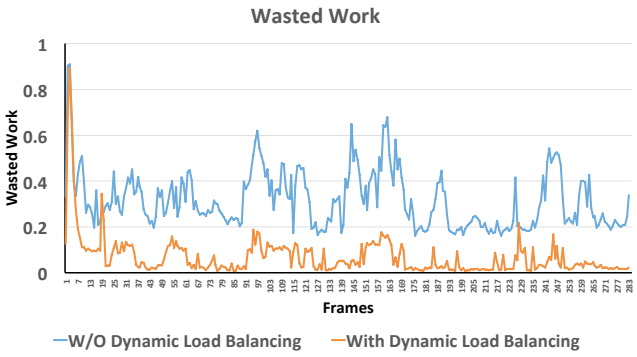

Fig. 2. Wasted work before (blue) and after (red) dynamic load balancing

2.6. Dynamic Load Balancing

While graph partitioning (Section 2.3) balances the number of tokens processed per thread, the difference in the computation per token can result in load imbalance. This is particularly troublesome for the LM phase, where the irregularity of LM graph lookups with variable backoffs can result in large load imbalance.

To overcome this, we implemented a token-stealing scheme, inspired by the idempotent work-stealing implemented in task schedulers [5, 6, 7]. The basic idea is to have the less-loaded threads *steal* tokens from loaded threads. The challenge is in doing so without adding synchronization that could slow down the already loaded thread. This is achieved by stealing from the tail of a token array while the loaded thread processes tokens from the head of the token array. In cases when the token array is almost empty, a token can be both processed by the loaded thread and stolen. This redundancy does not result in correctness issues as token processing is idempotent. However, to minimize performing additional work, our implementation avoids stealing when the token array is smaller than a specific amount.

To measure load imbalance, we introduce a metric called *wasted work*, defined as the ratio of the time threads spend waiting for the slowest thread over the total time spent by all the threads. Figure 2 shows the reduction in the amount of wasted work for each frame in a particular utterance. It is clearly seen that the load imbalance between threads have reduced with dynamic load balancing.

2.7. Memory Optimizations

Viterbi in speech decoding is heavily memory bound as it maintains several large data structures for maintaining the tokens read and written per frame and randomly accessing large AM and LM graphs. This makes it very important to make efficient use of caches and to structure access patterns in strides that trigger hardware prefetching.

We employ several memory optimizations. First, we (manually) performed range analysis to compress fields of data structures. For instance, if a pointer field can only take values in a range of size less than 2^{32}, this pointer can be encoded in a 32 bit field as opposed to

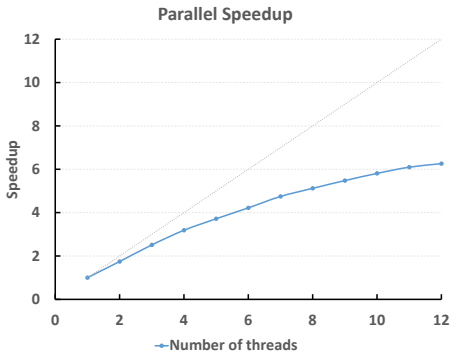

Fig. 3. Speedups for the parallel algorithm over the sequential version (blue).

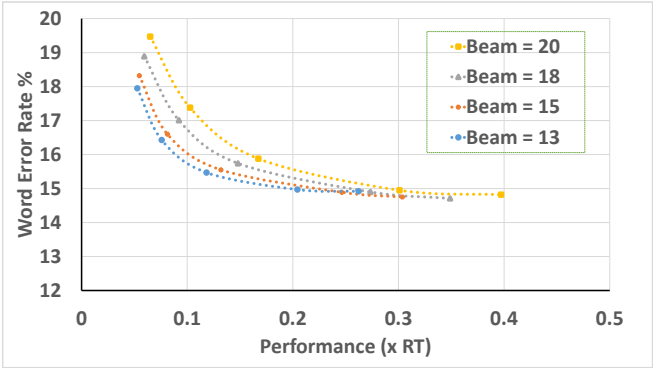

Fig. 4. Performance vs WER curve.

a 64 bit field. Second, we split each struct based on fields that are only read or read-and-written in particular phases. For instance, AM phase only reads some fields in the tokens generated in the previous frame. By separating these fields from those that are not read (such as the predecessor of the token), we are not only reducing the total memory traffic but also improving cache efficiency as now more tokens effectively fit in a cacheline. Together, these optimizations lead to a 26% reduction in memory traffic.

3. EVALUATION

We evaluated our parallel speech decoding algorithm on a large acoustic model with 1.4 million edges and a language model with 59 million ngrams generated from a vocabulary of size 200,000. We used the voice search development dataset which has 6214 utterances for testing. The evaluation was done on a 2.67GHz Intel(R) Xeon(R) X5650 machine with 2 sockets, each socket with 6 cores running Windows 8.1. Each core consists of its own L1 and L2 caches of sizes 32 kB and 256 kB respectively. 12 MB L3 cache is shared within a socket and all cores share 64 GB of RAM. For all our experiments we disabled hyper-threading to achieve consistent performance numbers. We kept beam width at 18k and maximum tokens per frame at 100k for the experiments. The original sequential baseline performance of our system was running at $1.8\times$ real time.

With per-thread beam search (Section 2.5), our parallel implementation runs at $0.27\times$ real time with 12 threads for the entire dataset, leading to a $6.67\times$ performance improvement over our sequential baseline. We incur a 0.08% absolute increase in word-error rate (WER) with per-thread beam search on. Without per-thread beam search, our parallel version runs in $0.35\times$ real time with no loss in accuracy. This is a performance improvement of $5.14\times$.

To evaluate the parallel speedup, we used the parallel decoder running with one thread as the baseline. This baseline runs at $1.56\times$ real time reflecting the sequential improvements of our optimizations. Figure 3 demonstrates the performance as we increase the number of threads. To complete the experiments in a reasonable amount of time, these runs only use the first 100 utterances from our dataset. While not perfectly linear (as shown by the dotted line), the speed up curve demonstrates that the parallel decoder robustly scales to 12 threads with possible scaling beyond 12 threads.

Figure 4 shows the WER vs decoder performance with respect to real time (xRT). Each curve represents a beam width value chosen from 13k, 15k, 18k, 20k, while varying the maximum number of tokens processed per frame. These runs do not use the per-thread beam search. The best WER of 14.73% is achieved with a beam-width set to 18k for the maximum number tokens set to 100k. At this configuration, turning the per-thread beam search increases the WER to 14.81%.

4. RELATED WORK

While designing performant LVCSR systems is an active area of research (see for instance [4]), we focus on prior work that attempts to parallelize the Viterbi computation. The main advantage of the source-node-based partitioning described in this paper is that token recombination is synchronization free. To the best of our knowledge, this is the first paper to do so. Other token recombination approaches either require partially sequential execution or costly atomic operations [3, 8, 9, 10, 11]. In addition, we show how judicious partitioning balances load and minimizes inter-thread communication. While prior work [3, 8, 12] do report 3 to 6 speedups, which is comparable to those reported in this paper, they use a far smaller vocabulary set than ours.

You et al. [3] introduce a parallel speech algorithm that switches between a pull model where each AM edge reads tokens from its predecessors or a push model where a token updates its successors. The push model on large vocabulary uses costly atomic operations to perform token recombination and the pull model requires several synchronizations for a single frame. Ravishankar [8] also does triphone based partitioning, but his scheme involves grouping together triphones which have the same right context for a particular phone, whereas our partitioning works on per phone basis. Also he does not union subgraphs for static load balancing.

Phillips and Rogers [12] proposed another parallel speech decoding algorithm which requires reduction at the end of each frame. With the performance difference between CPU and memory widening since this work was done, it is unclear if this approach scales in modern hardware platforms. You et al. [13] proposed a new multi-core approach but only show $2\times$ speedup with 4 threads with a vocabulary $10\times$ smaller than ours.

There are several other parallel implementations of speech decoding for other platforms such as ARM processors [14] and GPUs [3, 9, 15, 16]. The GPU implementations require several levels of interactions with the host CPU for communication intensive phases for each frame which results in excessive overhead. Since our parallel approach is synchronization-free, we expect that a many-core implementation of it to be faster than others.

5. REFERENCES

[1] Geoffrey Hinton, Li Deng, Dong Yu, George Dahl, Abdel rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara Sainath, and Brian Kingsbury, "Deep neural networks for acoustic modeling in speech recognition," *Signal Processing Magazine*, 2012.

[2] Dong Yu, Adam Eversole, Michael L. Seltzer, Kaisheng Yao, Brian Guenter, Oleksii Kuchaiev, Yu Zhang, Frank Seide, Guoguo Chen, Huaming Wang, Jasha Droppo, Amit Agarwal, Chris Basoglu, Marko Padmilac, Alexey Kamenev, Vladimir Ivanov, Scott Cypher, Hari Parthasarathi, Bhaskar Mitra, Zhiheng Huang, Geoffrey Zweig, Chris Rossbach, Jon Currey, Jie Gao, Avner May, Baolin Peng, Andreas Stolcke, Malcolm Slaney, and Xuedong Huang, "An introduction to computational networks and the computational network toolkit," Tech. Rep. MSR-TR-2014-112, August 2014.

[3] Kisun You, Jike Chong, Youngmin Yi, E. Gonina, C.J. Hughes, Yen-Kuang Chen, Wonyong Sung, and K. Keutzer, "Parallel scalability in speech recognition," *Signal Processing Magazine, IEEE*, vol. 26, no. 6, pp. 124–135, November 2009.

[4] D. Nolden, H. Soltau, and H. Ney, "Progress in dynamic network decoding," in *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, May 2014, pp. 3276–3280.

[5] Robert D. Blumofe and Charles E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, Sept. 1999.

[6] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt, "The design of a task parallel library," in *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, New York, NY, USA, 2009, OOPSLA '09, pp. 227–242, ACM.

[7] Maged M. Michael, Martin T. Vechev, and Vijay A. Saraswat, "Idempotent work stealing," in *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, New York, NY, USA, 2009, PPoPP '09, pp. 45–54, ACM.

[8] Ravishankar Computer Science and M. K. Ravishankar, "Parallel implementation of fast beam search for speaker-independent continuous speech recognition," 1993.

[9] Jike Chong, Youngmin Yi, Arlo Faria, Nadathur Rajagopalan Satish, and Kurt Keutzer, "Data-parallel large vocabulary continuous speech recognition on graphics processors," Tech. Rep. UCB/EECS-2008-69, EECS Department, University of California, Berkeley, May 2008.

[10] Jungsuk Kim and I. Lane, "Accelerating large vocabulary continuous speech recognition on heterogeneous cpu-gpu platforms," in *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, May 2014, pp. 3291–3295.

[11] Jungsuk Kim, Jike Chong, and Ian R. Lane, "Efficient on-the-fly hypothesis rescoring in a hybrid gpu/cpu-based large vocabulary continuous speech recognition engine," in *INTERSPEECH 2012, 13th Annual Conference of the International Speech Communication Association, Portland, Oregon, USA, September 9-13, 2012*, 2012, pp. 1035–1038.

[12] Steven Phillips and Anne Rogers, "Parallel speech recognition," *Int. J. Parallel Program.*, vol. 27, no. 4, pp. 257–288, Aug. 1999.

[13] Kisun You, Youngjoon Lee, and Wonyong Sung, "OpenMP-based parallel implementation of a continuous speech recognizer on a multi-core system," in *Acoustics, Speech and Signal Processing, 2009. ICASSP 2009. IEEE International Conference on*, April 2009, pp. 621–624.

[14] S. Ishikawa, K. Yamabana, R. Isotani, and A. Okumura, "Parallel LVCSR algorithm for cellphone-oriented multicore processors," in *Acoustics, Speech and Signal Processing, 2006. ICASSP 2006 Proceedings. 2006 IEEE International Conference on*, May 2006, vol. 1, pp. I–I.

[15] Paul R. Dixon, Tasuku Oonishi, and Sadaoki Furui, "Fast acoustic computations using graphics processors," in *Proceedings of the 2009 IEEE International Conference on Acoustics, Speech and Signal Processing*, Washington, DC, USA, 2009, ICASSP '09, pp. 4321–4324, IEEE Computer Society.

[16] Patrick Cardinal, Pierre Dumouchel, Gilles Boulianne, and Michel Comeau, "GPU accelerated acoustic likelihood computations," in *In Proc. Interspeech*, 2008, pp. 964–967.