OPENMP-BASED PARALLEL IMPLEMENTATION OF A CONTINUOUS SPEECH RECOGNIZER ON A MULTI-CORE SYSTEM

Kisun You, Youngjoon Lee, and Wonyong Sung

School of Electrical Engineering, Seoul National University
San 56-1, Shillim-dong, Kwanak-gu, Seoul 151-744 Korea
{ksyou,yjlee}@dsp.snu.ac.kr, wysung@snu.ac.kr

ABSTRACT

We have implemented a 20,000-word continuous speech recognizer on a multi-core based system. A fine grain parallel processing approach is employed for good scalability, and the OpenMP library is used for enhanced portability. In the emission probability computation, a dynamic workload distribution method is employed for good load balancing. However, the search network involved in the Viterbi beam search is statically partitioned into independent subtrees to reduce memory synchronization overhead. In order to further improve the performance, a workload predictive thread assignment strategy as well as a false cache line sharing prevention method are employed. The test was conducted using WSJ1 20k test and development set. We achieved the speed-up of 3.90 by utilizing four threads parallelization in a four-core system compared to four copies of the baseline single thread speech recognizer running simultaneously. The final recognition system runs about twice the speed of the real-time requirement.

Index Terms— Speech recognition, OpenMP, Parallelization

1. INTRODUCTION

In these days, many multi-core systems appear in various platforms ranging from servers, PC's, and down to embedded systems. A multi-threaded implementation is an attractive approach for developing a real-time large vocabulary speech recognizer (LVCSR).

However, software needs to be carefully designed to make use of the multi-core systems efficiently. Especially, most of the multi-core systems still have a quite limited main memory bandwidth because all of the processor cores are connected to one or two memory systems. Thus, the reduction of memory access contention as well as the load balancing are important in parallel implementation.

There have been various researches about the parallelization of speech recognizers [1] [2]. The AT&T speech recognizer was parallelized for the implementation on a Unix-based symmetric multiprocessor (SMP) workstation [1]. However, the implementation is based on a specific thread library, thus it is not easily portable to other platforms. A parallel LVCSR system for a cellular-phone-oriented multiprocessor platform was developed in [2]. However, their work based on a coarse grain approach does not seem to be easily scalable when the vocabulary size or the number of cores increases.

In this paper, we implemented a multi-thread version of a speech recognizer using OpenMP which enables easy portability. For good scalability with respect to the vocabulary size as well as the number of processors, a fine grain parallel processing approach is employed. In this approach, we need to consider neither the inter-block dependency nor the size balance of each block since the parallelization is usually conducted for each loop. However, we need a prudent approach to reduce the size of sequential parts, evenly distribute the workload, and remove the coherency overhead.

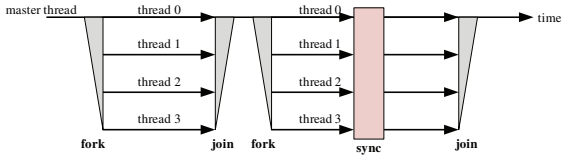

Fig. 1. OpenMP execution diagram.

The rest of this paper is organized as follows. Section 2 describes the OpenMP application programming interface and the baseline implementation of the speech recognizer. The parallelization procedures using OpenMP are shown in Section 3. The optimization techniques to achieve a higher performance gain are presented in Section 4. Section 5 shows the experimental results. Finally, concluding remarks are given in Section 6.

2. BACKGROUND

2.1. OpenMP Parallelization

OpenMP is an application programming interface for a shared memory multiprocessor environment [3]. OpenMP facilitates fork/join parallelism which is illustrated in Fig. 1. The execution starts with a single master thread. When the execution reaches a parallel region, the master thread forks several threads for parallel processing. Since the difference in execution time among the threads leads to stall time for some threads, OpenMP supports several scheduling techniques such as static, dynamic, and guided [3] to balance the workload well.

OpenMP is used for this development since we consider that the speech recognition program can be efficiently parallelized by the fork/join programming model. We employed a fine grain parallel processing approach which basically transforms small program blocks, such as loops, into multiple threads. Although the fine grain parallel processing approach may need more cycles for synchronization, it is advantageous for achieving a good load balancing. Since the fine-grain parallel processing finds out the parallelism inside of an algorithmic block or a function, we do not need to care the dependencies among the algorithmic blocks in speech recognition. Note that the computation or data access workload for each functional block in speech recognition varies much according to several factors such as the vocabulary size and the pruning-level. This also makes the coarse-grain parallel processing hard to be applied.

2.2. Baseline Implementation of a Speech Recognition System

In this paper, we implemented an HMM(Hidden Markov Model)-based continuous speech recognizer. The recognition system consists of three functional blocks - feature extraction, emission probability computation, and Viterbi beam search [4].

We employed a tree-lexicon based search network to reduce the search space and the amount of computation for accumulated likelihoods [5]. Tree copies and language model factoring are applied to support the back-off bigram language model and efficient beam pruning respectively. [6].

In order to reduce the amount of memory required for the search space, we developed a token based search algorithm. The idea is inspired by the HTK token passing algorithm [7], however the implementation is different. For every tree branch in the search space, there are tokens that store the accumulated likelihood and the vocabulary history. Since memory is allocated only when the token is active, we can save dynamically allocated memory space.

The memory access especially in the token processing can be reduced by applying the beam pruning with a predicted threshold [8]. Tokens are updated and newly created at every frame. However, many of the updated and newly created tokens will be pruned out in the beam pruning stage. All the likelihood values of the active tokens are written to the memory, the threshold value for the pruning is determined next, and then the tokens having low values are deleted. These sequential steps need a lot of memory accesses, such as allocating memory for the tokens, writing all the token values, reading them to determine the threshold, and de-allocating memory for the tokens after the pruning. In order to avoid these complex and memory-access intensive operations, we predict the pruning threshold based on the maximum accumulated likelihoods of previous two frames. In this scheme, the tokens with low likelihoods are deleted immediately without memory accesses.

3. PARALLEL IMPLEMENTATION USING OPENMP

We parallelized the baseline recognizer using OpenMP. The feature extraction is omitted in this work because it needs relatively simple computation, mostly FFT (fast Fourier transform), and can be well parallelized. The parallel implementation of the emission probability computation and the Viterbi beam search are described in this section.

3.1. Parallelized Emission Probability Computation

The emission probability computation is the most computation intensive part, and has a regular structure. It computes the likelihood for every active shared HMM state. Since the computation of each HMM state is independent of the others, the emission probability can be easily computed in parallel. The number of active HMM states is counted at every frame to distribute the computation evenly to each thread. We found out that the serial execution time to count the active HMM states is small enough not to affect the overall performance.

3.2. Parallelized Viterbi Beam Search

The Viterbi beam search traverses all the active tree branches in the search network to compute the accumulated likelihood of every path candidate. There are several levels of parallelism we can exploit. First of all, the operations inside any lexicon tree are independent of the ones in the other trees. Note that we have many tree copies for the bigram language model support. Thus, the easiest way to partition the workload is to group the trees and assign each group to different threads. Second, the operations inside the tree branches are independent of the ones in the other tree branches. We can parallelize the workload by assigning different threads to the groups of tree branches. In this work, the tree branch is the unit of workload distribution. The lexical-tree based workload distribution suffers from severe load imbalance since the workload of the lexical tree varies due to the pruning during recognition.

For each thread, we made an independent active branch list. Before processing the active lists in a parallel manner for each frame, we should assign thread numbers to the branches newly activated in the previous frame.

For good load balance, we can distribute the newly activated branches evenly to each thread by dynamic allocation. However, we found out that the dynamic allocation degrades the cache performance due to false cache line sharing. Since many tree branches which reside in contiguous memory region might be modified by different threads, the overhead of the cache coherency protocol becomes larger. We will describe the effect of false cache line sharing further in Section 4.

Instead of dynamic workload allocation, we applied a static workload assignment strategy. The thread number for each tree branch is determined in off-line. Whenever a tree branch is being activated, it is assigned to a designated thread.

Since there is a large number of tree branches in the search network, it is undesirable to store the thread numbers for all the tree branches. Thus, we partitioned the lexical tree into sub-trees. After the partitioning, all the tree branches inside a sub-tree have the same thread number. This approach has two advantages. First, it reduces the overhead of accessing the thread number look-up table as well as the size of the look-up table itself. Once the first-level tree branches have been activated in a thread, all the children tree branches can be processed in the same thread. Second, it helps to balance the workload since the active regions inside a lexical tree are usually in the same level.

After all the tree branches are processed, the likelihood values of the active leaf states should be propagated to the following unigram or bigram trees. Since there are many branches that write their values to the same destination, it will incur much synchronization overhead when parallelized. Thus, this part is not parallelized.

4. OPTIMIZATION OF THE OPENMP PARALLEL SPEECH RECOGNIZER

We applied two optimization techniques to improve the performance of the parallel version developed in Section 3. The optimization techniques are avoiding false cache line sharing and adopting an efficient thread assignment strategy.

4.1. Avoiding False Sharing of Data Cache Lines

Modern processors have long cache lines to compensate for the DRAM latency. Note that no global variable should be duplicated and accessed by different threads to avoid false sharing. For instance, assume that a global variable accessed by multiple threads is defined as follows:

$$int\ nActBranch\ \rightarrow\ int\ nActBranch[4] \qquad (1)$$

Then the four elements in the array will probably reside in the same cache line, incurring the false cache line sharing problem. To allocate these variables in different cache lines, we added appropriate

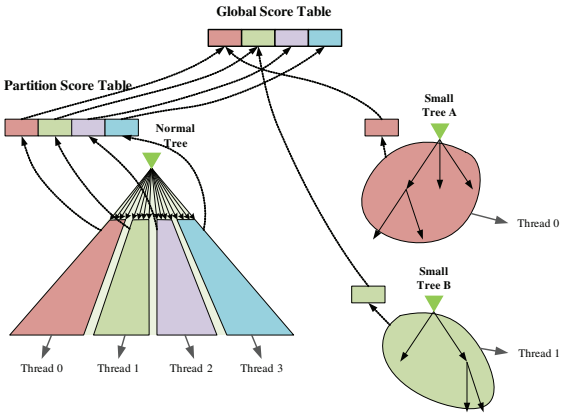

Fig. 2. An illustration of the static load assignment strategy.

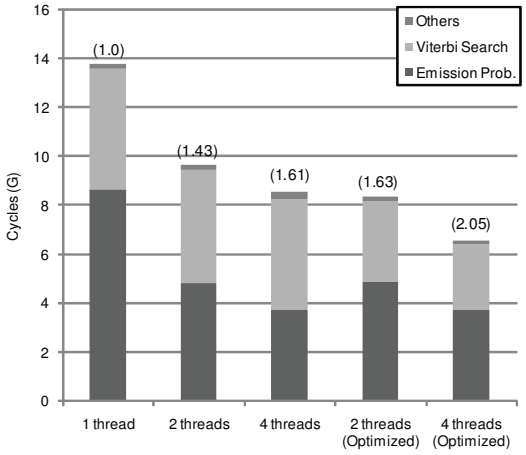

Fig. 3. Execution cycle counts for various implementations (speed-up).

padding to the variables as follows:

$$int\ nActBranch[4] \rightarrow int\ nActBranch[4 << 4] \quad (2)$$

We need to adjust the number of padding variables according to the cache line size. Since the cache line size is 64 bytes, we shifted the index value to left by four.

4.2. Workload Predictive Thread Assignment Strategy

Since the parallelized Viterbi beam search processes the branches in the ancestor-to-descendent way to reduce unnecessary synchronization, sibling branches in each sub-tree are processed continuously in each thread. As a result, there are many consecutive memory accesses for several sibling branches. Thus, we stored the data of the tree branches in the sub-tree in the breadth-first order to help the cache exploit the data locality.

The baseline parallel recognizer assigns the thread numbers to the first level branches using the round-robin strategy. This strategy helps distributing the workload evenly since the tree search network is usually imbalanced. However this strategy is not enough to guarantee good load balance.

To achieve a well-balanced thread assignment, we divided the direct child branches into four consecutive chunks which have different sizes. Note that each chunk has a subtree consisting of all the descending branches which will be processed in each thread.

Since we need to decide the size of the chunks for all the trees, it is difficult to find the optimum solution. In this work, we propose a heuristic method that decides the size of each chunk based on the expected workload of its subtree. Figure 2 illustrates the proposed thread assignment strategy. First, we estimate the workload of a branch with its number of states and its language model probability. The number of states of a branch implies the maximum number of tokens to process, and the language model probability reflects the probability of a branch being activated. After assessing the expected workload of all the descending branches, we can decide the workload score of each subtree by accumulating them. Starting from the unigram tree, the size of the chunk is determined to evenly distribute the workload. Global workload scores of the threads are kept to balance the workload over the trees. For trees with a small number of branches, we assign all the branches to only one thread to avoid unnecessary partitioning and utilize the data locality.

5. EXPERIMENTAL RESULTS

5.1. Experimental Setup

The target system is an Intel Core 2 Quad Q6600 CPU based SMP system. It has four identical processor cores in a chip, and each core has its own L1 instruction (32KB) and L1 data (32KB) caches. Two cores share one unified L2 cache (4MB), and the L2 cache lines are dynamically assigned to the cores. The operating frequency of the processor cores is 2.4GHz.

The acoustic model of the recognizer was trained by HTK, an open-source speech recognition toolkit [7], using the speaker independent training set in WSJ1 corpus, and the test was conducted using WSJ1 20K test and development set [9]. The acoustic beam width was set to 350.0, the language model beam width was set to 100.0, and the language model weight was given to 16.0. The word error rate (WER) was 20.5% for 216 test sentences. Note that the WER can be improved by employing several advanced training techniques, which do not increase the complexity of the recognizer. We verified that the recognition result is the same with the one obtained by the HTK recognizer.

For performance measurements such as cycle counts and load balancing, we used a speech sample of 6.15 seconds long. OProfile [10] which utilizes the hardware performance counters in the CPU was used for profiling.

5.2. Execution Time and Load Balancing

The execution cycle counts of the parallel implementations are shown in Fig. 3. The OpenMP based parallel speech recognizer achieved the speedup of 1.43 and 1.61 for two and four threads implementations, respectively. The further optimized parallel speech recognizer, where the false cache line sharing prevention method and the workload predictive thread assignment strategy are applied, resulted in the speedup of 1.63 and 2.05 for two and four threads, respectively. It takes 2.79 seconds for processing a speech sample of 6.15 seconds long.

The relatively low speedup with the four thread version is due to the limited bus or main memory bandwidth of the multi-core system. The CPI (Cycle Per Instruction) of the optimized four thread version

Table 1. Load balance of the parallelized system

Cycles (M)	Core 0	Core 1	Core 2	Core 3
Emission Prob.	3,700	3,725	3,724	3,677
Viterbi Search	2,425	2,721	2,593	2,657
Others	147	58	33	39
Total	6,272	6,504	6,350	6,373

Table 2. Execution cycle counts comparison of multi-copy and multi-thread versions

Utilization	Multi-copy	Multi-thread	Speedup
Two cores	17,753	8,550	2.08
Four cores	26,155	6,703	3.90

increased from 1.08 to 1.98 when compared with the baseline single thread version. Assuming that the CPI does not change, the speedup of the four thread version will be 3.77. This clearly means that memory access reduction of the speech recognizer is critical to achieve a higher speed-up with a multi-core system.

Table 1 shows the load balance of the optimized 4 threads speech recognizer. Since the emission probability adopted the dynamic workload distribution, it is well balanced. The Viterbi search with the static workload assignment is also moderately balanced even though it has some peak value in Core 1. Note that load imbalance of other functions are mainly due to serial blocks.

5.3. Comparison with Bandwidth Limited Reference

We also compared the two and four thread implementation results with those of the two and four copies of the single thread baseline program running simultaneously on the multi-core system, respectively. Note that, in the two copy version, one copy is allocated to Core 0 and the other copy is mapped to Core 2 to avoid L2 cache sharing. Core 0 and 1 share one L2 cache memory, and Core 2 and 3 share another L2 cache memory in the employed four-core system. In the four copy system, only one copy is allocated to each core, and four duplicated programs share the L2 cache and the bandwidth of the memory system. This comparison seems more reasonable because the original single thread baseline version solely used the bandwidth resource. The results are summarized in Table 2, which indicates that both the two and four thread versions achieve the almost ideal speedup when compared with two and four copy versions. Thus we can find that the memory bandwidth limitation is the most critical bottleneck of low speedup in our multi-thread implementation. The load-balancing and the coherency problems seem solved well by the proposed methods.

6. CONCLUDING REMARKS

We have implemented an OpenMP based multi-thread speech recognizer that is not only portable but also scalable. We developed a workload predictive static scheduling method for Viterbi beam search. With the OpenMP based parallel implementation, we achieved a speedup of 1.43 and 1.61 when two and four threads are employed, respectively. Further optimization of the parallelized versions resulted in the speedup of 1.63 and 2.05 for two and four thread cases, respectively. When compared with the bandwidth limited reference, the speedup increases to 2.08 and 3.90 with a two-core and four-core systems, respectively. The profiling results show that the CPI of the four threads version, 1.98, is quite high when compared with the single thread version, 1.08. This explains that the relatively low speed-up of the four threads version is due to the limited memory bandwidth of the multi-core system, not the imbalance of the parallelized workloads. The implemented four threads version achieves the real-time factor of 0.45.

7. ACKNOWLEDGEMENTS

This work was supported in part by the Brain Korea 21 Project and ETRI SoC Industry Promotion Center Human Resource Development Project for IT SoC Architect.

8. REFERENCES

[1] S. Phillips and A. Rogers, "Parallel speech recognition," *Int. Journal of Parallel Programming*, vol. 27, no. 4, pp. 257–288, 1999.

[2] S. Ishikawa, K. Yamabana, R. Isotani, and A. Okumura, "Parallel LVCSR algorithm for cellphone-oriented multicore processors," *IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, vol. 1, pp. 177–180, May 2006.

[3] M. J. Quinn, *Parallel Programming in C with MPI and OpenMP*, McGraw-Hill, 2003.

[4] X. Huang, A. Acero, and H. W. Hon, *Spoken Language Processing - A Guide to Theory, Algorithm, and System Development*, Prentice Hall PTR, New Jersey, 2001.

[5] H. Ney, R. Haeb-Umbach, B. H. Tran, and M. Oerder, "Improvement in beam search for 10000-words continuous speech recognition," *IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, vol. 1, pp. 9–12, 1992.

[6] M. Federico, M. Cettolo, F. Brugnara, and G. Antoniol, "Language modelling for efficient beam-search," *Computer Speech and Language*, vol. 9, no. 4, pp. 353–379, 1995.

[7] S. Young, G. Evermann, D. Kershaw, G. Moore, J. Odell, D. Ollason, V. Valtchev, and P. Woodland, *The HTK Book Version 3.3*, 2005.

[8] E. Lin, Y. Kai, R. Rutenbar, and T. Chen, "A 1000-word vocabulary, speaker independent, continuous live-mode speech recognizer implemented in a single FPGA," *ACM/SIGDA 15th Int. Symp. on FPGA*, pp. 60–68, 2007.

[9] P. C. Woodland, J. J. Odell, V. Valtchev, and S. J. Young, "Large vocabulary continuous speech recognition using HTK," *IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, vol. 2, pp. 125–128, 1994.

[10] J. Levon, *OProfile - A System Profiler for Linux*, http://oprofile.sourceforge.net.