Parallel Audio Quick Search on Shared-Memory Multiprocessor Systems

Yurong Chen, Wei Wei, Yimin Zhang

Intel China Research Center
8/F, Raycom Infotech Park A, No.2, Kexueyuan South Road, Zhong Guan Cun,
Haidian District, Beijing 100080, China
{yurong.chen}@intel.com

Abstract

Audio search plays an important role in analyzing audio data and retrieving useful audio information. In this paper, a Partially Overlapping Block-Parallel Active Search method (POBPAS) is proposed to perform audio quick search on shared-memory multiprocessor systems (SMPs). This method uses a proper data segmentation to achieve parallelism and performs a high level of parallelism with little additional work. Several techniques including I/O optimization, proper data partition and dynamic scheduling are also introduced to maximize its scalability performance.

In addition, we conduct a detailed performance characterization analysis of the parallel implementation of the POBPAS for three data sets on two Intel Xeon SMPs. Experimental results indicate that there are no obvious parallel limiting factors in the implementation except memory bandwidth. As a result, it can achieve 11.3X speedup for a larger data set (searching a 15 seconds' clip in a 27 hours' audio stream) on the 16-way processor system.

1. Introduction

Audio data from radio, television, databases, or on the Internet has been a rich resource of information. Audio search (e.g., searching a large audio stream for an audio clip, even if the large audio stream is corrupted/distorted) is an effective way to analyze audio data and retrieve useful audio information. It has many applications including analysis of broadcast music/commercials, copyright management over the Internet, or finding metadata for unlabeled audio clips, and etc [1,2,3,4]. A typical audio search system is serial and designed for single processor systems [5]. It normally takes a long time for such a search system to search for a target audio clip in a large audio stream. In many cases, however, an audio search system is required to work efficiently on large audio streams, e.g., to search large streams in a very short time (e.g., close to real-time). Additionally, an audio stream may be partially or entirely distorted, corrupted, and/or compressed. This requires that an audio search system should be robust enough to identify those audio segments that are the same as the target audio clip, even if those segments may be distorted, corrupted, and/or compressed [2,3]. Thus, it is desirable to have an audio search system which can quickly and robustly search large audio databases for a target audio clip.

In this paper, a Partially Overlapping Block-Parallel Active Search method (POBPAS) is proposed to efficiently employ the multiple processors and speed up audio searching for large audio streams on Shared-memory Multi-Processor systems (SMPs). This method also could be potentially used as an underlying audio processing technique for emerging personal media mining applications on future large-scale multi-core systems. Furthermore, several techniques including I/O optimization, proper data partition and dynamic scheduling are introduced to maximize its scalability performance. A parallel audio quick search system with the POBPAS method is implemented on two Intel Xeon SMPs. This paper also characterizes the performance of the parallel audio search system in detail.

The reminder of this paper is organized as follows. Section 2 gives an introduction to the framework of the audio quick search. Section 3 describes the POBPAS method and some optimization techniques in its implementation. The experimental results and application characterizations are reported in Section 4. Finally, we conclude in Section 5.

2. Framework of Audio Quick Search

Figure 1 outlines the serial audio searching system. First, the feature vector sequences are extracted from both the audio clip and the audio stream. Then, windows of the same length as the clip are applied to both feature sequences. The feature vectors within a window are used to estimate a CCGMM (Common Component Gaussian Mixture Model, [4]). Then the distance is calculated between the CCGMMs estimated from the clip and the windowed stream. If the distance is below a threshold, the audio clip is considered to be detected and located in the audio stream. Finally, the window on the audio stream is shifted forward in time and the search proceeds. Below are details for each stage of the system.

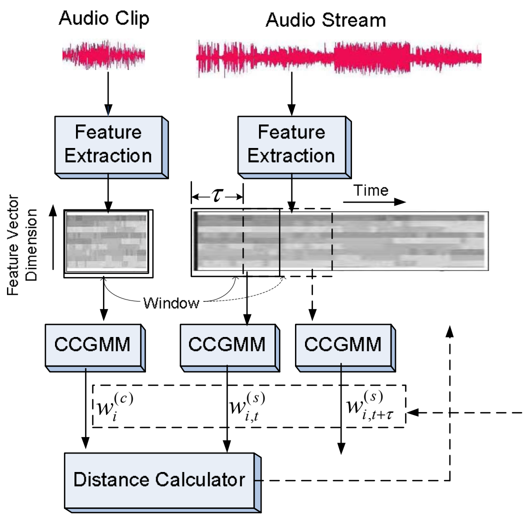

Figure 1. Overview of the Serial Audio Searching System

Feature extraction stage: The feature vector sequences are extracted from both the audio clip and the audio stream. Then, windows of the same length as the clip are applied to both feature sequences. A general approach is to separate the audio into frames. Then, features may be computed for each frame. Some of the features include Fourier coefficients, mel-frequency cepstral coefficients (MFCCs), spectral flatness and derivatives, means, variances.

Modeling stage: Feature vectors within a window are used to estimate a CCGMM model. In this approach, audio segments are modeled by a GMM (Gaussian Mixture Model, [4]), which consists of M Gaussian components with component weights $W_i^{(k)}$, means $\mu_i^{(k)}$, and covariance $\sum_i^{(k)}$, $i = 1,2,\cdots,M$:

$$P^{(k)}(x) = \sum_{i=1}^{M} w_i^{(k)} \mathrm{N}\left(x \mid \mu_i^{(k)}, \Sigma_i^{(k)}\ \right) \qquad (1)$$

To avoid poor estimation and simplify Kullback Leibler distance computation, we further assume that the GMMs for all the audio segments share a common set of Gaussion components. They are pre-trained and stored as a universal GMM:

$$p^{(u)}(x) = \sum_{i=1}^{M} W_i^{(u)} N\left(x \mid \mu_i^{(u)}, \Sigma_i^{(u)}\ \right).$$

Thus Equation (1) becomes (with $\mu_i^{(k)}, \Sigma_i^{(k)}$ being constrained to $\mu_i^{(u)}, \Sigma_i^{(u)}$ respectively):

$$p^{(k)}(x) = \sum_{i=1}^{M} w_i^{(k)} \mathrm{N}\left(x \mid \mu_i^{(u)}, \Sigma_i^{(u)}\ \right) \qquad (2)$$

Now we need only estimate the component weights to specify a CCGMM for an audio segment. Given T feature vectors $x_t (t = 1,2,\cdots,T)$ for an audio segment k , the weights are estimated by the following formula:

$$w_i^{(k)} = \frac{1}{T} \sum_{t=1}^{T} \frac{w_i^{(u)} \mathrm{N}\left(x_t \mid \mu_i^{(u)}, \Sigma_i^{(u)}\ \right)}{\sum_{j=1}^{M} w_j^{(u)} \mathrm{N}\left(x_t \mid \mu_j^{(u)}, \Sigma_j^{(u)}\ \right)} \qquad (3)$$

Active search stage: The distance is calculated between the CCGMMs estimated from the clip and the windowed stream. If the distance is below a threshold, the audio clip is considered to be detected and located in the audio stream. In our approach, Kullback Leibler (KLMAX) distance between two CCGMMs can be approximated as follows:

$$d_{KLMAX} = \max_{i=1,2,..,M} (w_i^{(1)} - w_i^{(2)}) \log \frac{w_i^{(1)}}{w_i^{(2)}} \qquad (4)$$

As the window applied over the audio stream shifts forward in time, we can skip some unnecessary hypothesis matches, while mathematically guaranteeing no false dismissals.

3. Parallel Audio Quick Search Method

In this section, we propose a Partially Overlapping Block-Parallel Active Search method (POBPAS) to efficiently and quickly perform audio searching for large audio streams on SMPs. This method uses a proper data segmentation to achieve parallelism and performs a high level of parallelism with little additional work. Furthermore, several techniques including proper data partition, dynamic scheduling and I/O optimization are also introduced to improve the parallel performance in practice.

Figure 2 outlines the parallel audio searching system using the POBPAS method. First, the large audio stream is partitioned into partially overlapped blocks (multiple smaller substreams) with equal length for simplification. Then, the feature vector sequence is extracted from the audio clip, and a CCGMM is estimated from the feature vector on a processor. After that, these small audio streams can be dynamically assigned to different processors, and each processor extracts the feature vector sequences and estimates the CCGMMs from the audio stream, and detects the correct matches using the KLMAX distance between the CCGMMs estimated from the clip and the windowed small stream. When a processor completes the audio searching on a small audio stream, it can receive another uncalculated small stream to continue until there is no available stream anymore.

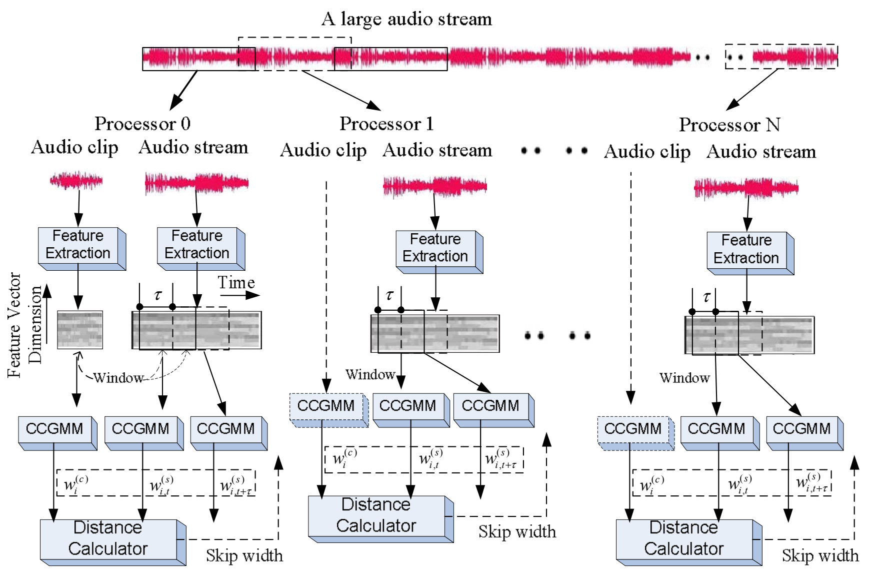

Figure 2. Overview of the Parallel Audio Searching System Using the POBPAS Method

In the implementation, the minimal length of the overlap equals $FNClip-1$, where $FNClip$ is the frame number of the clip. These overlaps ensure that we cannot miss any correct detection in the parallel search process. On the other hand, the length of each small stream should be greater than or equal to the length of the clip. Note that smaller block size may result in large overlapped computation, while larger block size may result in considerable load imbalance in parallel processing by multiple processors. So an appropriate block size should be chosen to reduce the overlapped computation and load imbalance.

In practice, the block length is usually several tens (or hundreds) of times the length of the clip, and the number of blocks is much larger than the number of processors. We use dynamic scheduling for parallel processing these small audio streams to reduce the load imbalance overhead, since the computation cost of time-serials active search [5] for the same clip is much different from different audio streams even with equal length.

Finally, in order to overlap I/O and computation effectively, the proper length of audio stream to be loaded into memory buffer from the disk by each processor is chosen to reduce the I/O contention.

Figure 3 is the pseudo code illustrating an example of POBPAS for performing robust and parallel audio search on a SMP system. At line 02, audio search module may be initialized, e.g., target audio clip file and large audio stream file may be opened, and global parameters may be initialized. At line 04, a large audio stream may be partitioned into NG smaller partially overlapped streams. At line 06, a model (e.g., CCGMM) may be established for the target audio clip. At line 08, NG audio streams may be dynamically scheduled to available processors and parallel processing of the scheduled groups may be started. Line 08 uses one example instruction with OpenMP directives [7] that sets up parallel implementation and other parallel implementation instructions may also be used.

Lines 10 through 34 illustrate how each of NG streams is processed and searched for the target in parallel by a processor in the multiprocessor system. It is worth noting that for illustration purpose, process in lines 12 to 34 is shown as iteration from the first stream until the last stream. In practice, if there are several processors available, several groups are processed in parallel by these available processors. At line 14, some or all of audio streams in each stream may be further partitioned into NS partially overlapped segments if such streams are longer in time than the target audio clip. Line 16 starts iterative process for each segment of the group, shown in lines 18 through 32. At line 20, a feature vector sequence (frame by frame) may be extracted from the segment. At line 22, a model (e.g., CCGMM as shown in Equations (1) to (3)) may be established for the segment. At line 24, distance (e.g., KL-max distance as shown in Equation (4)) between the segment model and the target audio clip model may be computed. At line 26, whether the segment matches the target audio clip or not may be determined based on the distance calculated in line 24 and a predetermined threshold #1. If the distance is less than the threshold #1, the segment matches the target audio clip. At line 28, whether a number of following segments (e.g., M segments) in the same audio (sub)stream may be skipped from searching may be determined based on the distance calculated in line 24 and a predetermined threshold #2. If the distance is larger than the threshold #2, M segments may be skipped from searching. At line 42, search results from local arrays from all of the processors may be summarized and outputted to a user.

```
02: Initialization;
04: Partition a large audio stream into smaller NS partially
    overlapped blocks (smaller audio streams);
06: Establish a model for target audio clip;
08: #pragma omp parallel for schedule(dynamic,1),
    num_threads(NumOfThread);
    /* dynamically schedule smaller streams to available processors
    and start parallel processing of the scheduled streams by multiple
    processors */
10: For streamid = 0 to NG-1
12: {
14:         Partition current stream into NS partially overlapped
            segments, if necessary;
16:         For segmentid = 0 to NS-1
18:         {
20:                 Extract a feature vector sequence;
22:                 Establish a model for the segment;
24:                 Compute distance between the model of each
                    segment and the target audio clip model;
26:                 If Distance < threshold #1, Match!
28:                 else if Distance > threshold #2,
                            Skip M segments in the same audio (sub)stream;
30:                 Store results into an local array for the stream;
32:         }
34: }
42: Output search results of local arrays from each processor ;
```

Figure 3. The Pseudo-code of the POBPAS Method

4. Performance Characterization and Analysis

We implemented the parallel audio search system based on the POBPAS method by using OpenMP programming model with C language on two Intel Xeon shared-memory multiprocessor systems. The first is a 4-way SMP system (Xeon MP 2.8GHz, L3 cache size 2MB, FSB Speed 400MHz), and the second is a Unisys-Es7000 system which consists of 16 processors (Xeon MP 3.0GHz, L3 cache size 4MB, FSB speed 400MHz, L4 cache size 32MB shared by each 4 processors, 4x4 Crossbar interconnection). The key characteristics of these two systems show the Unisys system has larger memory access latency because the memory transactions need to go through L4 cache before visiting the memory. For software configuration, on both platforms, we use Intel C/C++ Compiler Version 9.0 to compile the program under Windows 2003 Sever OS with full compiler optimization. We collected the performance data with some Intel performance analysis tools such as Vtune Analyzer and Thread Profiler [6].

In our experiments, the audio signals were framed with 20ms length and 10 ms shift, and 14 MFCCs were then extracted from each frame as feature vectors. The 1997 Mandarin Broadcast News (MBN) corpus (Hub4-NE) was used, which consisted of about 30 hours of recorded broadcasts from different sources, such as CCTV and VOA. All together there were 58 files, where 53 files were selected to train the universal GMM with 128 diagonal Gaussian components via EM algorithm. The components of the universal GMM were taken as the common components of CCGMMs. For the audio database, we constructed three real data sets by recording the TV program. Their sizes are 500MB (9hr), 1GB (18hr) and 1.5GB (27hr), respectively. The length of audio search clip is 15 seconds and its size is about 160KB.

4.1. Performance Optimization

As mentioned in the previous section, the proper size of audio block should be chosen to reduce the overlapped computation and load imbalance. In order to obtain the proper size of block, we did some experiments on changing the block size in the implementation. For simplification, we set the block size as a linear function of the clip size. The experiment shows it has better performance to let the block size equal to 25 times the clip size, which is about 8MB. In the following experiments, we used this block size as the default. Figure 4 shows the dynamic scheduling improve the performance for the 1.5GB data set on the 16-way system by 3% and 7% in 8P and 16P cases, respectively.

Thread scheduler, an essential component in OS, plays an important role in driving the application to a high performance. There are already a lot of studies on thread scheduling, for instance, [8] gives a comprehensive study on different scheduling techniques and concludes that the decision made by the OS has a significant impact on performance with a relatively large processor count. Table 1 reports the performance benefit when enabling the thread affinity mechanism by binding threads to specific processors, to minimize the threads migration and context switches among processors. In addition, it improves the data locality performance and mitigates the impact of maintaining the cache coherency among all the processors. With thread affinity, the L3 cache miss rate and FSB (Front Side Bus) coherency ratio are reduced by 7.5% and 49% respectively. Here, the cache coherency ratio is defined as the ratio of bus requests

	Time (s)	L1 cache miss rate	L2 cache miss rate	L3 cache miss rate	FSB Bandwidth (MB/s)	FSB Coherency Ratio
Non-Affinity	88.0	3.72%	4.51%	60.33%	260	25.28%
Affinity	85.1	3.54%	4.55%	55.83%	115	12.89%

Table 1. Performance Comparison of POBPAS with and without Thread Affinity on QP-machine for 1.5GB Data Set (4 threads)

satisfied by another cache over the total number of read requests sent on the bus. Furthermore, the FSB bandwidth utilization is also reduced by 56%.

Based on the version with thread affinity, the I/O optimization is utilized to overlap the I/O operation and computation. The approach is that we did not let the processor read any data from the disk at the beginning and just define the data range for each thread (and create a buffer with the audio block size for each thread). This is because the MFCC feature vector is extracted frame by frame by each thread in the feature exaction stage. We let each thread read a 160KB audio frame in its own audio block from the disk each time as the feature extraction stage begins. In this way we can reduce the I/O contention obviously. Figure 4 shows the performance benefit when using dynamic scheduling and I/O optimization on the 16-way Unisys machine for the 1.5GB data set. From this figure, we can see that the I/O optimization improve the performance in 8P and 16P cases by 15% and 20%, respectively.

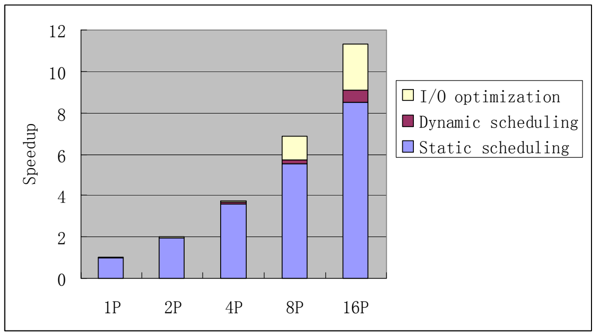

Figure 4. Optimization for the POBPAS Method on the 16-way System for the 1.5GB Data Set (speedups are normalized to the baseline on single processor)

4.2. Scalability Performance Study

Figure 5 shows the speedup of the parallel audio search system based on the POBPAS method for three different audio data sets on the 16-way SMP. From this figure, we can see the system scale well with the increasing number of processors for these three data sets. It can achieve linear speedup on 2 or 4 processors, 6.2~6.7X speedup on 8 processors and 8.8~11.3X on 16 processors.

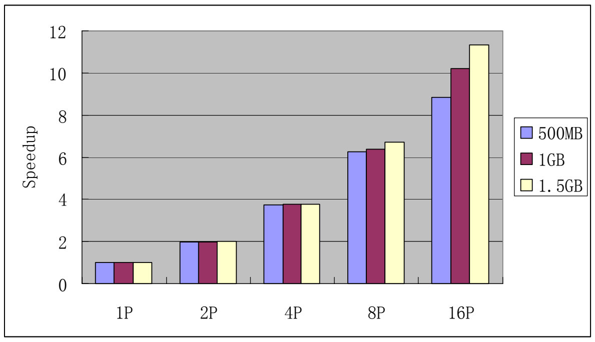

Figure 5. Speedup of the Parallel Audio Search System on the 16-way System for Three Data Sets

To deeply understand the scaling limiting factors, we characterize the parallel performance from the high level general parallel overheads, e.g., synchronizations penalties, load imbalance, and sequential sections, to the detailed memory hierarchy behavior, e.g., cache miss rates and FSB bandwidth.

Fig 6 gives the general parallel profiling metrics for the parallel audio search system, where “Parallel” means running time inside the parallel region, and “Imbalance” represents time spent waiting for other threads to reach the end of a parallel region. The profiling information indicates that the parallel system has very low synchronization and parallel overhead. The sequential area and load imbalance increases steadily with the increasing number of processors but remains at a relatively low percentage for larger data sets (less than 1.7% for the 1.5GB data set in 16P case). Overall speaking, the general parallel limiting factors are insignificant for larger data sets, especially for the 1.5GB data set, and will not hurt the scalability performance for the parallel system on the 16-way system.

Besides the general scalability performance factors, memory subsystem also plays an important role in identifying the scaling performance bottlenecks. Figure

7 shows the L3 level cache miss rate, cache coherency ratio and the memory bus utilization rate.

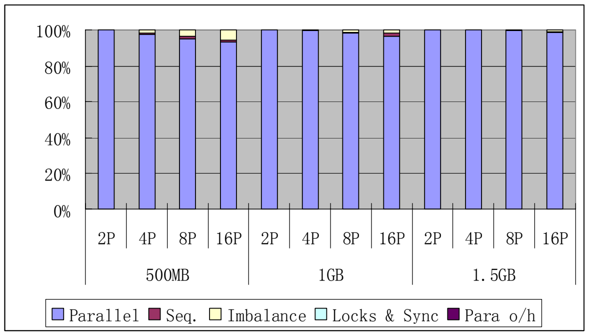

Figure 6. Impact of the General Parallel Limiting Factors for Three Data Sets

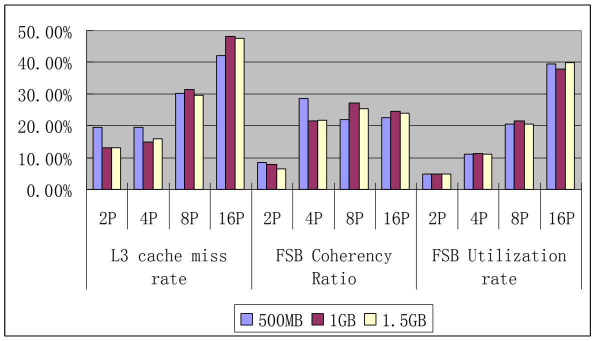

Figure 7. Memory Characterization of the Parallel Audio Search System on the 16-way System for Three Data Sets

From Figure 7, we can see that the L3 cache miss rate increases with increased numbers of processors. However, the L1 cache miss rate and L2 cache miss rate are very low (about 4%), the aggregate cache miss rate (L1 * L2 *L3) ranges from 0.02% to 0.07% with different data sets and different processor numbers. Figure 7 also shows the cache coherency ratio changes little in cases we used more than 4 processors. These two characteristics reveal that they are not scaling performance bottlenecks for the parallel system. At the same time, we can see the FSB utilization rate increase linearly with the increase of processor numbers for all data sets from Figure 7. The parallel audio search system utilized about 40% of FSB bandwidth of the 16-way system with 16 processors for the three data sets. This result shows the memory bandwidth should be a limiting factor for the scalability performance of the parallel audio search system.

5. Conclusions

In this paper, we implemented and analyzed a parallel quick audio search system with a proposed Partially Overlapping Block-Parallel Active Search method on two Intel Xeon shared-memory multiprocessor systems. This method uses a proper data segmentation to achieve parallelism and performs a high level of parallelism with little additional work. It also could be potentially used as an underlying audio processing technique for emerging personal media mining applications on future large-scale multi-core systems. In the implementation, several techniques including I/O optimization, proper data partition, dynamic scheduling, etc., are utilized to maximize its scalability performance. Our results show that for a 27 hours' audio stream (15 seconds' clip) it can achieved a speedup of 11.3X on the 16-way processor system.

At the same time, we performed a detailed performance characterization of the parallel quick audio search system for three data sets. The thread profiling analysis shows that it has little load imbalance and extremely low percentage of sequential region, even on up to 16 processors. Memory hierarchy performance analysis indicates it has very low cache misses and has a high memory bus bandwidth requirement. This result reveals that this parallel audio quick search system is constrained by the bandwidth and the larger bandwidth may further improve its scalability performance.

References

[1] P. Cano, E. Batlle, T. Kalker, and J. Haitsma, A review of algorithms for audio fingerprinting, Proceedings of 2002 *International Workshop on Multimedia Signal Processing*, pages 169-173, 2002.

[2] C. J. Burges, J. C. Platt, and S. Jana, Distortion discriminant analysis for audio fingerprinting, *IEEE Trans. on Speech and Audio Processing*, 11(3):165-174, May. 2003.

[3] S. Sukittanon and L. E. Atlas, Modulation frequency features for audio fingerprinting, Proceedings of *ICASSP'02*, pages 1773-1776, 2002.

[4] Y. Wang and C. Huang, Speaker-and-environment change detection in broadcast news using the common component gmm-based divergence measure, In *INTERSPEECH-2004*, pages 1069-1072, 2004.

[5] K. Kashino, T. Kurozumi, and H. Murase, A quick search method for audio and video signals based on histogram pruning, *IEEE Trans. on Multimedia*, 5(3):348-357, Sep. 2003.

[6] Intel Corp. VTune performance analyzer. Available at http://www.intel.com/software/products/vtune

[7] OpenMP Architecture Review Board: OpenMP C and C++ Application Program Interface, Version 2.0. http://www.openmp.org

[8] R. C. Kunz. PhD dissertation, Performance bottlenecks on large-scale shared-memory multiprocessors. 2004.