Scalable Audio Feature Extraction

Devon Leno Bryant

A Proposal Presented to the Faculty of

Univeristy of Colorado at Colorado Springs

Department of Computer Science

October 2013

Dr. Rory Lewis	Date
Dr. Jugal Kalita	Date
Dr. Terrance Boult	Date

Abstract

This paper describes *Safe*, a proposed system for scalable and efficient audio feature extraction in the Music Information Retrieval domain. The system builds upon previous research in efficient audio feature extraction, taking advantage of both multicore processing and distributed cluster computing to provide a scalable architecture for feature extraction algorithms.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The *Music Information Retrieval* (MIR) domain spans a wide variety of disciplines and applications. MIR techniques are being leveraged by businesses and academics for recommendation systems, classification, identification, automatic transcription, similarity searching, and more. Despite the seemingly different nature of these problems, many of these systems actually leverage similar audio feature extraction algorithms at their core. These feature extraction algorithms can range from low-level digital signal processing techniques to high-level machine learning algorithms. Most of these algorithms are computationally intensive, both in resource usage and running times. While these constraints are generally acceptable for small datasets, many existing MIR libraries and their algorithms are not easily scalable for larger datasets.

Numerous libraries exist for audio feature extraction and many have focused on algorithmic optimizations for efficient extraction, often using C++ for its performance benefits in numerical computing. However, as modern computing continues to shift towards more parallel architectures [2], it is clear that different programming models and techniques are needed to take full advantage of these systems. Often, the imperative techniques used for optimal execution on a single processor are ill-suited or difficult to transition to parallel environments [18].

Additional research has been done in extending these existing libraries to work in distributed computing environments [15, 8, 4]. These systems are all based on a master/worker division of nodes, which suffers from two main drawbacks. The primary drawback is that a master node in a system provides a single point of failure, meaning these systems do not address issues with failure-tolerance. The second drawback is that a master node in a system often enforces bounds on the scalability; introducing an upper bound when throughput is limited by the master node and forcing a lower bound of two nodes from the master/worker separation.

This paper proposes a high-performance concurrent system for audio feature extraction based on modern techniques in parallel computing. Scalability of the system is fault-tolerant and addresses both vertical scalability through multicore processing and horizontal scalability through distributed clustering. While the primary focus of this research is on performance and scalability, there is a strong secondary goal towards providing a general extensible MIR framework for others to re-use and build upon.

1.1 Related Work

There have been many MIR feature extraction libraries developed over the last decade, each with different feature sets and focus. This section outlines some of the most popular or relevant libraries, though many more exist.

One of the earliest libraries to provide audio feature extraction that is still used today is Marsyas [19]. Marsyas is an efficient C++ framework for general dataflow audio processing that has been used for various analysis and synthesis tasks. Marsyas provides both a command line interface and a graphical editor for building data flows. Marsyas has also been extended to perform batch extraction in distributed

environments [4]. The distributed model relies on a custom dispatcher node sending audio clips over TCP to known worker nodes for feature extraction.

Yaafe [14] is a fast and efficient library for audio feature extraction written mostly in C++ with a Python interface. The focus of Yaafe is on efficiency, utilizing dataflow graphs to eliminate redundant calculations between different features. Yaafe is mostly a command line tool and supports output in CSV or HDF5 formats.

jAudio [17] is a Java based audio extraction framework developed primarily for MIR researchers. The framework provides a GUI for selecting features and filling out parameters and can export results to XML or Weka's ARFF [1] format for further experimentation or machine learning. jAudio also provides a mechanism for describing dependencies between features, similar to Yaafe, for the purpose of eliminating duplicate calculations. OMEN [15] is a distributed extraction framework that leverages jAudio as its base extraction engine. The distribution model for OMEN is based on a hierarchy of different node types including a master node providing the overall coordination, library nodes for storing collections of audio data and controlling extraction, and worker nodes for performing the actual feature extraction. Communication in OMEN is handled through SOAP based web services.

VAMP Plugins [2] provides a general C++ framework and API for audio feature extraction algorithms. There are numerous plugin libraries developed by different research groups available for use. The Sonic Visualiser tool provides a helpful user interface for feature exploration with many different graphical displays for visualizing features. The Sonic Annotator tool provides a command line interface for batch feature extraction on large numbers of audio files. Unfortunately, algorithms and results cannot be shared between plugins in the current architecture which leads to some inefficiencies during extraction.

[1] http://www.cs.waikato.ac.nz/ml/weka/arff.html
[2] http://www.vamp-plugins.org/

iSoundMW [8] is a distributed feature extraction framework and similarity search engine. The distribution model uses a single master node that sends small segmented frames of audio data to registered client worker nodes to perform extraction on.

MIRtoolbox [12] is a Matlab based MIR feature extraction library that takes advantage of existing visualizations and algorithms available in the Matlab ecosystem. MIRtoolbox provides some explicit mechanisms for composing features in stages from lower-level or intermediary calculations. However, memory management is an issue since the entire file and all feature calculations are held in memory. MIRtoolbox provides some parallel extraction capabilities, allowing concurrent extraction of separate files.

openSMILE [9] is a C++ based library focused on real-time Speech and MIR feature extraction. Batch extraction capabilities are provided as well as some multi-threaded support for parallel extraction on a single machine. openSMILE is based on a complex shared data memory system which allows single-component writes and multi-component reads of matrix data and leverages ring-buffers to manage memory. Different output formats are provided including CSV, ARFF, LibSVM [3], and HTK [4].

[3]http://www.csie.ntu.edu.tw/ cjlin/libsvm/

[4]http://www.ee.columbia.edu/ln/LabROSA/doc/HTKBook21/HTKBook.html

Chapter 2

Approach

The goal of this research is to provide an efficient framework for audio feature extraction that can easily scale in both vertical and horizontal directions. This is achieved by extending the techniques and algorithms used in other MIR libraries and adapting them to work with functional, asynchronous and reactive programming models. *Safe* is implemented in Scala [1] and leverages the Akka [2] library for parallel and distributed computing.

2.1 Efficient Feature Extraction

The primary abstraction for any extraction library is the description of a *Feature*. Within the *Safe* system, each feature describes a set of required and optional parameters needed to perform its calculations. Features in audio processing are often built on top of other features or share common steps in their calculations. For example, the *Fast Fourier Transform* (FFT) is generally a shared step for any algorithms working in the frequency domain. For that reason, each feature also describes its own dataflow as a sequence of smaller computational steps. Figure 2.1 shows the step sequence for

[1] http://www.scala-lang.org/
[2] http://akka.io/

calculating three different features: *Mel-Frequency Cepstral Coefficients* (MFCC) [6], *Constant-Q Transform* (CQT) [5], and *Spectral Shape* [10].

By breaking down features into smaller steps, efficiencies can be gained by eliminating shared calculations. A set of multiple features in the system is combined to form a *Feature Plan*. When shared calculations between feature sequences are eliminated in a feature extraction plan, the plan naturally forms a directed acyclic dataflow graph. This is a common technique in dataflow analysis that is used by other MIR feature extraction libraries [16, 14] to eliminate redundant calculations, similar to the problem of Common Subexpression Elimination [3] in compiler optimization. Figure 2.2 below shows an example of the optimized dataflow extraction plan for the same three features described earlier.

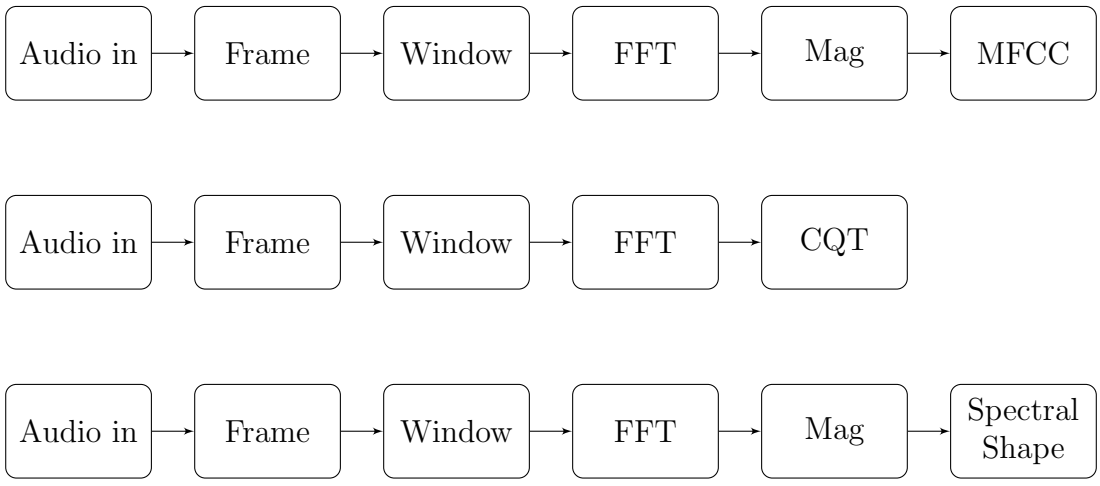

Figure 2.1: Sequential dataflow for MFCC, CQT, and Spectral Shape features

[3] http://en.wikipedia.org/wiki/Common_subexpression_elimination

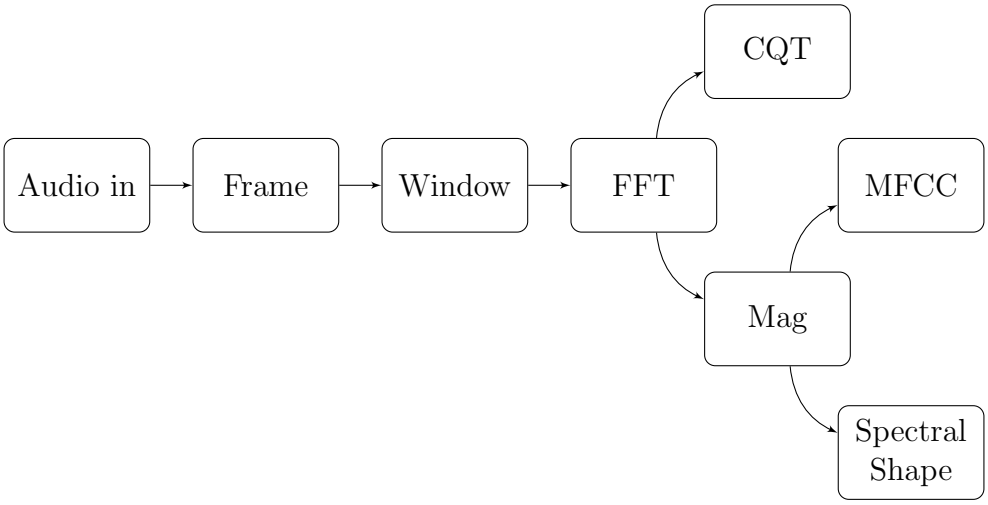

Figure 2.2: Optimal dataflow extraction plan

2.2 Parallel Feature Extraction

With the processing model expressed as a directed acyclic dataflow graph, audio data can be streamed through the pipeline to extract the various features. This approach lends itself to numerous levels of parallelism:

- *File Level* - Extraction can be run concurrently on multiple files.
- *Plan Level* - Any time a plan branches into separate paths, each path can run concurrently on the results of the parent calculation.
- *Feature Level* - When audio data is split into separate chunks (frames), each stage of a feature extraction sequence can concurrently operate on a different section of the data.

File level parallelism is a fairly self-explanatory concept and is the approach used in [12]. Tables 2.1 and 2.2 show examples of plan level and feature level parallelism where each row represents a different processor and each column represents possible computations over time. With plan level parallelism, different steps can run concurrent calculations on the same data outputs of a parent step. For example, table 2.1

shows that the CQT and Magnitude calculations can run concurrently on the output of the FFT. With feature level parallelism, a given calculation may be executing on a section of audio data concurrently with downstream calculations running off previous results. In table 2.2, the example highlights that a Window function can pass its output to the FFT function and immediately begin working on the next frame. *Note*: These tables are used for illustrative purposes of where parallelism can occur and do not represent the actual thread scheduling and division within the system. In the actual running system, some steps will take much longer than others and a single thread may finish multiple steps in the time another thread takes to execute one. Additionally, different combinations of file, plan, and feature level concurrency may occur together.

FFT($window_1$)	Magnitude(fft_1)	MFCC(mag_1)
	CQT(fft_1)	Spectral Shape(mag_1)

Table 2.1: Plan level parallelism

$frame_1$	Window($frame_1$)	FFT($window_1$)	CQT(fft_1)
	$frame_2$	Window($frame_2$)	FFT($window_2$)
		$frame_3$	Window($frame_3$)
			$frame_4$

Table 2.2: Feature level parallelism

Managing these different levels of concurrency with traditional thread scheduling and lock management would be complicated and difficult. Instead, an Actor model [1] for concurrency is used. Actor models, being asynchronous and message-based, provide a natural encoding for stream based dataflow processing. Each computational step in the dataflow graph is represented by an Actor and the intermediary results (messages) between actors are immutable. By encoding the feature extraction dataflow graphs as Actor hierarchies in this fashion, all three levels of parallelism mentioned (file, plan, and feature) are provided with no additional components required. The dynamic nature of Akka's Actor system make it simple to create complex

fault-tolerant feature extraction hierarchies. Within a feature extraction plan, each Actor usually takes on one of four common messaging roles from [11]:

- *Splitter* - Actors that take a single message or result and split the work into smaller pieces for downstream processing. Splitters provide much of the parallelism in feature extraction. File level parallelism is provided by splitting a message with multiple file locations (or a directory) into individual messages for each file. Feature level parallelism is provided by a Frame actor, splitting an individual file into smaller chunks (frames) to perform further computations on.

- *Transform* - The most common actor type in a feature extraction plan is a transform actor. These actors provide simple mapping functions from one type to another and send the result on for further processing. Examples of transforms are Windowing functions, FFT, Power Spectrum, MFCC, Spectral Shape, etc.

- *Resequencer* - Actors that take a stream of potentially unordered messages and send them out in an ordered sequence. Sequencing is important when writing out frame-based extracted features. For example, when writing out to a CSV file the features need to be in the same order as the frames in the source audio file.

- *Aggregator* - Aggregators are used to calculate features that go across frames or across the entire audio file. Some example aggregate functions are Spectral Flux [13] or any max/min/average/total/etc. calculation across a whole audio recording.

By default, a single actor is created for each node in the dataflow graph and the actor hierarchy and messaging mirror the arcs in the graph. For many systems, the concurrency provided by the file, plan, and feature level divisions is sufficient. However, for highly concurrent systems (systems with a large number of processors), the

Splitter and Transform actors can be replaced with actor pools and a round-robin routing strategy. This strategy allows concurrent execution of the same computational step on different frames. For example, multiple FFT calculations could be run concurrently on different frames. Resequencers and Aggregators cannot be placed into pools because they need to maintain some temporary state across multiple messages.

2.3 Distributed Feature Extraction

The distribution model for the system leverages Akka's clustering framework [4]. Akka's cluster system provides a fault-tolerant decentralized membership service based on Amazon's Dynamo [7] store. While previous research in distributed audio feature extraction exists [15, 8, 4], it is all based on a master/worker division of nodes. By leveraging a decentralized and masterless clustering framework, there is no single point of failure or single point bottlenecks in the system. Since the clustering framework is Actor based as well, it is simple to scale the feature extraction framework from a single machine to multiple nodes.

The simplest form of distributed feature extraction starts with a collection of audio files already distributed across the cluster onto the local file systems or through Hadoop HDFS [5]. This configuration is similar to the library nodes outlined in [15]. From there, bulk extraction can be performed across the cluster through a simple broadcast message to all nodes in the cluster. The time taken to extract all features will be limited by the slowest node in the cluster.

When different nodes in a cluster are unbalanced, i.e. some nodes have a larger heap, more CPU, etc., an adaptive load balancing router can be used to distribute feature extraction. This load balancing technique relies on the audio data being

[4] http://doc.akka.io/docs/akka/2.2.1/common/cluster.html
[5] http://hadoop.apache.org/docs/stable/hdfs_design.html

distributed in an HDFS cluster. The load balancer uses a probabilistic router based on weighting from the following metrics:

- *Heap* - Remaining JVM heap capacity.
- *Load* - System load average over the past minute.
- *CPU* - Percentage of CPU utilization.
- *Locality* - Location of data in HDFS.

Chapter 3

Research Plan

3.1 Completed Work

A large portion of this research consists of analyzing MIR feature extraction algorithms and breaking them down into smaller sequential components for dataflow analysis. The following features have been implemented in this system:

1. **Frame** - Produces an iterator of fixed-size, potentially overlapping, frames from an audio input stream.

2. **Window** - A set of common windowing functions, including *Hann*, *Hamming*, *Blackman-Harris*, *Blackman*, and *Bartlett* windows.

3. **Fast Fourier Transform** - A JVM-optmized version of the standard *Cooley-Turkey* FFT algorithm.

4. **Power Spectrum** - A set of spectrum functions including *Power*, *Phase*, and *Magnitude*.

5. **Constant-Q Transform** - Based on the Brown and Puckette [5] CQT algorithm.

6. **Mel-Frequency Cepstral Coefficients** - MFCC based on Davis and Mermelstrin [6] algorithm.

7. **Spectral Shape** - A combination of the spectral *Centroid*, *Spread*, *Skewness*, and *Kurtosis* described in [10].

8. **Spectral Flux** - Based on the Masri [13] algorithm for measuring change in the power spectrum.

9. **Onset Detection** - A spectral onset detection algorithm described in [3].

A framework was developed for describing features and combining multiple features to form extraction plans as optimal dataflow graphs. Plans can be mapped to Actor hierarchies and audio data can be streamed through for parallel extraction. Currently, the system only supports output in a CSV format.

3.2 Remaining Work

The following items still need to be completed:

1. A command line interface needs to be developed, along with an input format for describing feature extraction plans and audio data to be run through the system.

2. The system's performance needs to be evaluated against similar systems for datasets of varying size on a single machine.

3. The cluster messaging components need to be developed.

4. The system needs to be deployed and evaluated in a cluster environment to capture performance metrics.

3.3 Timeline

The following table provides an estimate of the timeline for this research.

Date	**Task**
February 20 2013	Background Research
May 1 2013	Feature Extraction Algorithms
August 19 2013	Parallel Extraction Capabilities
October 1 2013	Thesis Proposal
October 9 2013	Baseline System Evaluation
October 18 2013	Distributed Extraction Capabilities
November 1 2013	Distributed Performance Evaluation
November 13 2013	Thesis Defense

Table 3.1: Research Timeline

References

[1] Gul Abdulnabi Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, 1985.

[2] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[3] Sebastian Böck, Florian Krebs, and Markus Schedl. Evaluating the online capabilities of onset detection methods. In *ISMIR*, pages 49–54, 2012.

[4] Stuart Bray and George Tzanetakis. Distributed audio feature extraction for music. In *ISMIR*, pages 434–437, 2005.

[5] Judith C Brown and Miller S Puckette. An efficient algorithm for the calculation of a constant q transform. *The Journal of the Acoustical Society of America*, 92:2698, 1992.

[6] Steven Davis and Paul Mermelstein. Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 28(4):357–366, 1980.

[7] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *SOSP*, volume 7, pages 205–220, 2007.

[8] François Deliège, Bee Yong Chua, and Torben Bach Pedersen. High-level audio features: Distributed extraction and similarity search. In *ISMIR*, pages 565–570, 2008.

[9] Florian Eyben, Martin Wöllmer, and Björn Schuller. Opensmile: the munich versatile and fast open-source audio feature extractor. In *Proceedings of the international conference on Multimedia*, pages 1459–1462. ACM, 2010.

[10] Olivier Gillet and Gaël Richard. Automatic transcription of drum loops. In *Acoustics, Speech, and Signal Processing, 2004. Proceedings.(ICASSP'04). IEEE International Conference on*, volume 4, pages iv–269. IEEE, 2004.

[11] Gregor Hohpe and Bobby Woolf. *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional, 2004.

[12] Olivier Lartillot, Petri Toiviainen, and Tuomas Eerola. A matlab toolbox for music information retrieval. In *Data analysis, machine learning and applications*, pages 261–268. Springer, 2008.

[13] Paul Masri. *Computer modelling of sound for transformation and synthesis of musical signals*. PhD thesis, University of Bristol, 1996.

[14] Benoit Mathieu, Slim Essid, Thomas Fillon, Jacques Prado, and Gaël Richard. Yaafe, an easy to use and efficient audio feature extraction software. In *ISMIR*, pages 441–446, 2010.

[15] Daniel McEnnis, Cory McKay, and Ichiro Fujinaga. Overview of omen. In *ISMIR*, pages 7–12, 2006.

[16] Daniel McEnnis, Cory McKay, Ichiro Fujinaga, and P Depalle. jaudio: Additions and improvements. In *ISMIR*, pages 385–386, 2006.

[17] Cory McKay, Ichiro Fujinaga, and Philippe Depalle. jaudio: A feature extraction library. In *Proceedings of the International Conference on Music Information Retrieval*, pages 600–3, 2005.

[18] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, September 2005.

[19] George Tzanetakis and Perry Cook. Marsyas: A framework for audio analysis. *Organised sound*, 4(3):169–175, 2000.