Distributed audio feature extraction for music

Stuart Bray
Computer Science Department
University of Victoria
3800 Finnerty Rd
Victoria BC, Canada
`sbray@csc.uvic.ca`

George Tzanetakis
Computer Science Department (also in Music)
University of Victoria
3800 Finnerty Rd
Victoria BC, Canada
`gtzan@cs.uvic.ca`

ABSTRACT

One of the important challenges facing music information retrieval (MIR) of audio signals is scaling analysis algorithms to large collections. Typically, analysis of audio signals utilizes sophisticated signal processing and machine learning techniques that require significant computational resources. Therefore, audio MIR is an area were computational resources are a significant bottleneck. For example, the number of pieces utilized in the majority of existing work in audio MIR is at most a few thousand files. Computing audio features over thousands files can sometimes take days of processing. In this paper, we describe how *Marsyas-0.2*, a free software framework for audio analysis and synthesis can be used to rapidly implement efficient distributed audio analysis algorithms. The framework is based on a dataflow architecture which facilitates partitioning of audio computations over multiple computers. Experimental results demonstrating the effectiveness of the proposed approach are presented.

Keywords: distributed processing, dataflow networks, large-scale music information retrieval

1 INTRODUCTION

Advances in storage capacity, network speed, and audio compression have made possible the storage of large collections of audio and music on personal computers and portable devices. Projecting these trends it is likely that in the near future all of recorded music in human history will be available digitally. In the last few years, there has been a number of publications exploring ways of analyzing audio signals for music information retrieval (MIR) applications.

Developing audio and music analysis systems is challenging. Existing algorithms for extracting content information from music tend to use sophisticated signal processing and machine learning techniques which require signficant computational resources. Audio signal processing for music signals utilizes computationally intensive time-frequency analysis techniques such as the short time Fourier transform, wavelets and auditory filterbanks. Moreover, machine learning algorithms such as Support Vector Machines (SVMs) and Artificial Neural Networks (ANNs) require large amounts of training data in order to build reliable classifiers that generalize well. Currently developers of audio analysis applications are faced with a hard dilemma. They can use an interpreted programming environment such as Matlab that provides a lot of the necessary components for building audio analysis applications but is not as efficient as compiled code, or write the code from scratch which requires a significant investment of time just to build the necessary infrastructure. Distributed computation can be used to speed up computations and deal with the large data sets required for the analysis of audio signals. Existing mechanisms for parallelizing computer programs are designed for general programming tasks and tend to be complicated and hard to use. Therefore in audio analysis typically distributed computing is not used at all or when it is used it is tailored to very specific projects.

In this paper, we describe how *Marsyas-0.2*, a dataflow-based audio analysis and synthesis framework can be used for rapid prototyping and developing of distributed audio analysis systems. Although audio analysis is a challenging application domain, it provides some domain-specific constraints that can inform the design of an effective framework. Using *Marsyas-0.2*, music analysis systems are assembled from component using a dataflow-based approach. An important advantage of the dataflow approach proposed in this work is that it simplifies distributing audio and music analysis computation among various computers. The dataflow model emphasizes the parallel computational structure of a particular problem. Using a declarative dataflow specification approach the programmer can distribute audio analysis algorithms with minimum effort. In contrast, distributing traditional sequential programs places a significant burden to the programmer who has to decide which parts of the code can be parallelized and deal with load distribution, scheduling, synchronization and communication.

2 RELATED WORK

Dataflow programming has a long history. The original (and still valid) motivation for reasearch into dataflow was to take advantage of parallelism. Motivated by criticisms of the classical von Neumann hardware architecture dataflow architectures for hardware were proposed as an alternative in the 1970s and 1980s. During the same period a number of textual dataflow languages such as Lucid (Wadge and Ashcroft, 1985) were proposed. During the 1990s there was a new direction of growth in the field of dataflow visual programming languages especially in specific application domains. Succesful commercial examples include Labview [1] and SimuLink [2]. A recent comprehensive review of the history of dataflow programming languages can be found by Johnston et al. (2004). A recent trend is to view dataflow computation as a software engineering methodology for building systems using existing languages (Manolescu, 1997). A small scale case study of distributed music analysis using the Geddei processing framework is described by Wood and Keefe (2004). The description of a large scale evaluation of different acoustic measures of similarity is provided by Berenzeig et al. (2003). A similar dataflow approach to audio processing is utilized in CLAM (Amatriain, 2005). More recently the M2K/D2K dataflow framework (Downie et al., 2004) has been used for evaluation of MIR systems. The Max/MSP [3] dataflow visual programming language has also been used extensively in computer music.

3 DATAFLOW ARCHITECTURE

Marsyas-0.2 [4] is a software framework, written in C++, for rapid prototyping and experimentation with audio analysis and synthesis with specific emphasis on processing music signals. A variety of existing building blocks that form the basis of many published algorithms are provided as dataflow components that can be composed to form more complicated algorithms (black-box functionality). In addition, it is straightforward to extend the framework with new building blocks (white-box functionality).

In *Marsyas* terminology the processing nodes of the dataflow network are called *MarSystems* and provide the basic building blocks out of which more complicated systems are built. Existing components include: I/O (audio files, Matlab, audio playback and recording), feature extraction (short-time Fourier transform, wavelets, mel-frequency cepstral coefficients), synthesis (wavetable, phasevocoder) and machine learning classifiers. clustering). Similarly to CLAM (Amatriain, 2005), *Marsyas-0.2* makes a clear distinction between data-flow which is synchronous and control-flow which is asynchronous. Because *Marsyas-0.2* has synchronous dataflow (i.e at every "tick" a specific data slice is propagated across the entire network) there is no need for queues between nodes and shared buffers can be used for better performance (similarly to Univ pipes).

[1]`http://www.ni.com/labview/`
[2]`http://www.mathworks.com/products/simulink/`
[3]`http://www.cycling74.com`
[4]`http://marsyas.sourceforge.net`

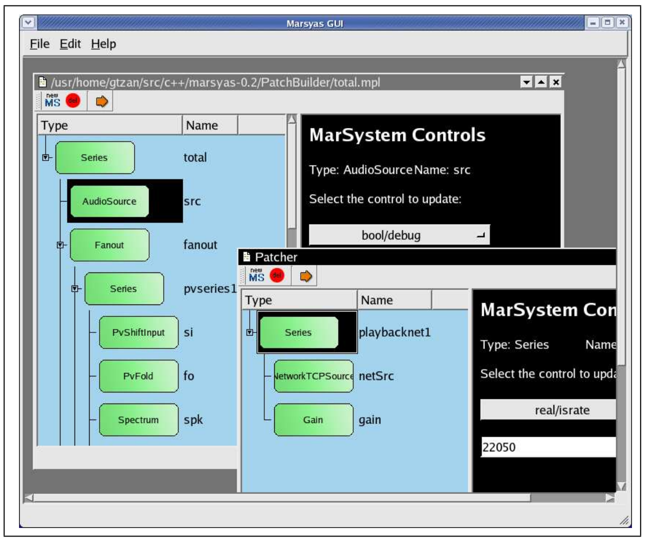

Figure 1: Marsyas-0.2 Visual Patch Builder

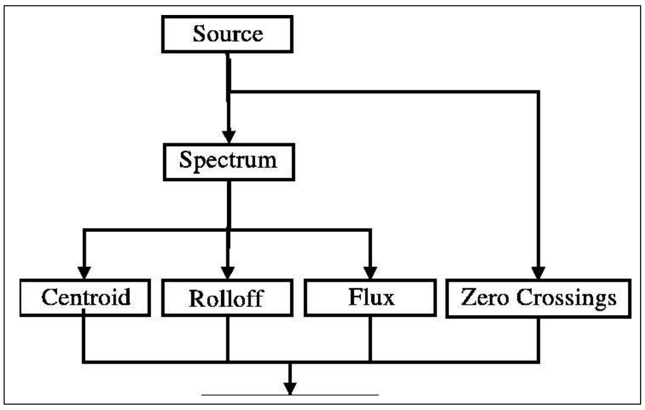

Figure 2: Feature Extraction Network for real-time Music/Speech classification

MarSystems can be instantiated at run-time. Therefore any complicated audio computation expressed as a dataflow network can be instantiated at run-time. For example multiple instances of any complicated network can be created as easily as the basic primitive *MarSystems*. This is accomplished by using the *Prototype* and *Composite* design patterns (Gamma et al., 1995). Currently there are three ways to build audio analysis and applications in *Marsyas-0.2*. The first is the traditional method of writing directly C++ code and compiling an executable. The second is based on a simple scripting language that provides syntactic constructs for building the dataflow network, setting appropriately the controls and moving sound through the network. The third way is to use a visual patch builder which uses the scripting language "under the hood".

Figure 1 shows the visual patchbuilder that can be used for specifying dataflow networks and controls. Figure 2 shows a dataflow network for extracting audio features for real-time music/speech classification. The entire network can be created at run-time without requiring any code compilation. The feature extraction front-end of Tzanetakis and Cook (2002), has been implemented as a dataflow network in *Marsyas-0.2*.

3.1 Distributed audio computation

There are two standard data communication protocols used on the Internet: transmission control protocol (TCP), and user datagram protocol (UDP). TCP provides reliability mechanisms to ensure that all packets are received exactly as they are sent; on the other hand, UDP provides no such mechanisms but is sigificantly faster due to less overhead. UDP is therefore the protocol of choice for real-time streaming applications where data is time critical.

Marsyas-0.2 supports both the UDP and TCP protocols. In order to send data to another machine, a "network sink" object is simply inserted somewhere in the flow of a *MarSystem*. In order to receive data, a "network source" object is inserted. Control flow and data flow are managed seperately so that controls can be changed from the sender and propagate through the system. The idea is that a user can operate several worker machines and the view of the distributed system is abstracted as one large composite *MarSystem*.

4 EXPERIMENTS

Audio features can be calculated either in realtime using UDP, or non-real time using TCP. One of the main goals of our experimentation was to demonstrate the cost benefits of parallelization accross multiple computers; thus, we used TCP and non-realtime feature extraction. For all the experiments a feature vector consisting of the means and variances of smoothed Mel-frequency Cepstral Coefficients as well as STFT-based features such as spectral centroid and rolloff was used. The data consists of 30-second audio clips and a 35-dimensional feature vector is calculated for each clip. The actual audio waveform samples are transmitted over the network.

4.1 Optimal Vector Size

File transfer throughput using TCP is highly affected by the size of the packet generated at the application layer. Requests that are larger than the TCP Maximum Segment Size (MSS) get buffered and partitioned in the transport layer; this can be costly. Additional overhead can come from several sources, some of which are the handling of out-of-order segments, retransmissions, and checksum calculations. A detailed analysis of the segmentation cost is beyond the scope of this paper, but is well documented and available from other sources.

Our experimentation was done on a 100Base-T Ethernet local area network of Apple G5 computers, where the MSS is 1460 bytes. Although the details of TCP are largely implementation specific, typically segments that are smaller than the MSS will be sent immediately. Consequently, we found that the optimal vector size to use would be as close to the MSS as possible, as shown in figure 3. From the graph you will also note the significant measured cost increase of over-stepping the boundary of the MSS. The cost increase when smaller packets are used is likely due to sending more packets than necessary, involving extra processing in all layers of TCP/IP, and saturation on the network.

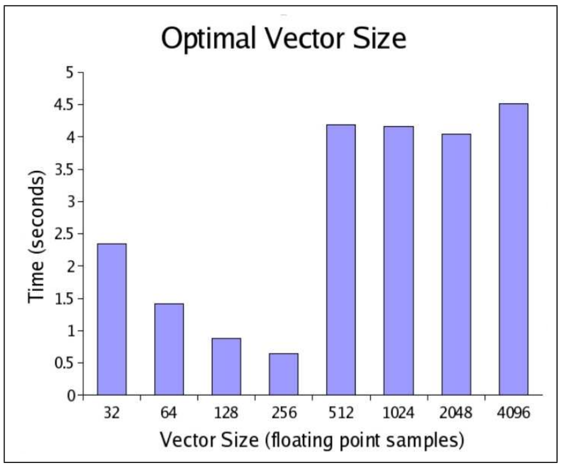

Figure 3: Optimal Vector Sizes for Maximum Throughput

4.2 Parallelization of Feature Extraction

In this section we demonstrate our results from parallelizing audio computation across multiple computers using the data-flow architecture of Marsyas 0.2. The model involves a dispatcher node that sends seperate clips from an audio collection to each worker node in the network. The job of a worker node is to simply calculate features for each file it receives and then send the results to a collector process (possibly running on the same machine as the dispatcher) that gathers the results. In our first experiment, the audio collection was partitioned into sub-collections which were sent to each worker. All the audio clips are stored on the dispatcher. We found that the optimal number of worker nodes in this model was three, after which there was no time benefit of using extra machines. In fact, it was costly to add any more than five worker nodes due to the network capacity of the dispatcher / collector. Later in this section it is shown that collection partitioning across multiple dispatchers can improve results. Table 1 shows results of using the collection dispatcher model, using up to five worker nodes and audio collections of up to 10,000 files. The format is hours:minutes:seconds.

Table 1: Parallelization results for Collection Dispatcher

	10	**100**	**1000**	**10000**
Local	00:05	00:58	09:39	1:36:49
1W	00:07	01:10	11:48	1:58:49
2W	00:03	00:38	06:01	1:10:46
3W	00:04	00:34	05:49	59:46
4W	00:03	00:34	05:52	1:04:56
5W	00:04	00:36	05:54	1:08:36

The problem with the first approach is that some nodes may complete processing the features of their respective sub-collection before others, sitting idle. Thus the time it takes to process the entire audio collection is dependent on the slowest node in the system. In order to alleviate this problem and make use of idle nodes, an adaptive approach is used where the dispatcher sends data as necessary to each worker. That way, each node in the system is work-

Table 2: Parallelization results for Adaptive Dispatcher

	10	**100**	**1000**	**10000**
Local	00:05	00:58	09:39	1:36:49
1W	00:07	01:10	11:48	1:58:49
2W	00:04	00:40	06:21	1:02
3W	00:03	00:33	05:33	57:10
4W	00:03	00:31	05:24	54:20
5W	00:03	00:31	05:25	54:15

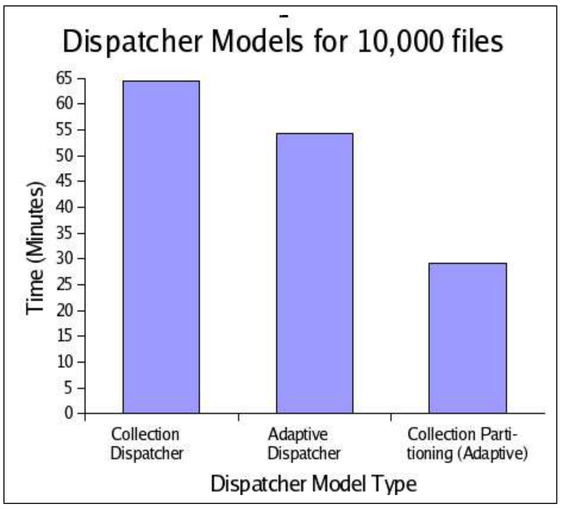

Figure 4: Comparison of each Dispatcher Model

ing until the dispatcher has finished processing the files in the collection. Table 2 shows the increase in performance based on this approach.

4.3 Data Partitioning and Multiple Dispatchers

When using one dispatcher the bottleneck is the the network capacity of the Ethernet card on the dispatcher. To demonstrate this fact we decided to partition the audio collection across two dispatchers, essentially doubling the network capacity. In this model four worker nodes are still used, as well as a seperate collector node whose only job was to gather the feature results from each worker. As expected, the time it took to process the same size audio collection was cut in half. In figure 4 we show the results of processing features on an audio collection of 10,000 files, using four worker nodes for each type of dispatcher. Theoretically this model can be further improved by partitioning the data across even more dispatchers in an hierarchical fashion.

4.4 Large Scale Experimentation

Typically feature extraction tests run on audio collections of around 10,000 files. Based on our results we expect a linear trend as collection sizes increase. To test that hypothesis a large-scale test using the data-partitioning model with the adaptive dispatcher was conducted on 100,000 files. As expected, it took approximately ten times the amount of time to complete the 100,000 file test as it took to complete the 10,000 file test (5:00:44).

5 CONCLUSIONS-FUTURE WORK

Distributed audio feature extraction using the dataflow approach exemplified by *Marsyas-0.2* can result in significant savings in computation time without significantly burdening the programmer. Experimental results show that using 5 computers we can perform audio feature extraction for 100,000 30-second clips in 5 hours.

There are many directions of future work. More computationally-intensive feature front-ends such as auditory filterbanks can also be distributed. In many applications audio feature extraction is followed by machine learning. We are conducting experiments in distributed machine learning and it's integration with feature extraction. The ability to scale to large datasets in reasonable amounts of time will enable us to perform detailed adjusting of parameters without risking overfitting. We are also exploring the use of multi-threaded collectors and dispatchers to take advantage of mutiple-cpu and multicore workstations.

ACKNOWLEDGEMENTS

Our thanks to Yvonne Coady and Dan Hoffman for their help with the networking and system aspects of this work.

References

X. Amatriain. *An Object-Oriented Metamodel for Digital Signal Processing with a focus on Audio and Music*. PhD thesis, Univ. of Pompeu Fabra, 2005.

A. Berenzeig, B. Logan, D. Ellis, and B. Whitman. A large scale evaluation of acoustic and subjective music similarity measures. In *Proc. Int. Conf. on Music Informaiton Retrieval (ISMIR)*, Baltimore, USA, 2003.

J.S. Downie, J. Futrelle, and D. Tcheng. The int. music information retrieval system evaluation laboratory: Governance, access, and security. In *Int. Conf. on Music Information Retrieval (ISMIR)*, Spain, 2004.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

W. Johnston, J.R. Paul Hanna, and R.J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36(1):1–34, March 2004.

D. Manolescu. A data flow pattern language. In *Proceedings of the 4th Pattern Languages of Programming*, Monticello, Illinois, September 1997.

G. Tzanetakis and P. Cook. Musical Genre Classification of Audio Signals. *IEEE Trans. on Speech and Audio Processing*, 10(5), July 2002.

W. Wadge and E.A. Ashcroft. *Lucid, the dataflow programming language*. APIC Studies in Data Processing. Academic Press, New York, NY, 1985.

G. Wood and S. Keefe. A case study of distributed music analysis using the geddei processing framework. In *Proc. Int. Conf. on Music Information Retrieval (ISMIR)*, pages 275–276, Barcolona, Spain, 2004.