Audio Engineering Society

Convention Paper

Presented at the 129th Convention
2010 November 4–7 San Francisco, CA, USA

Automatic Parallelism for Dataflow Graphs

Ramy Sadek

University of Southern California Institute for Creative Technologies, 12015 Waterfront Drive, Playa Vista, CA, 90094, USA
Sadek@ict.usc.edu

ABSTRACT

This paper presents a novel algorithm to automate high-level parallelization from graph-based data structures representing data flow. This automatic optimization yields large performance improvements for multi-core machines running host-based applications. Results of these advances are shown through their incorporation into the audio processing engine Application Rendering Immersive Audio (ARIA) presented at AES 117. Although the ARIA system is the target framework, the contributions presented in this paper are generic and therefore applicable in a variety of software such as Pure Data and Max/MSP, game audio engines, non-linear editors and related systems. Additionally, the parallel execution paths extracted are shown to give effectively optimal cache performance, yielding significant speedup for such host-based applications.

1. BACKGROUND AND MOTIVATION

Graph-based data structures have become popular representations within audio processing software, especially in visual dataflow programming systems such as Pure Data [1], Max/MSP [2] and third-party APIs and libraries like Sonic Flow [3]. This paper presents an algorithm for efficient automated extraction of parallelism from dataflow graphs. As the trend toward multiple-core processors replaces the former trend toward ever-greater processor clock rates, computational efficiency increasingly relies on parallel processing. This is especially true for host-based processing software common in application areas ranging from game audio to non-linear editing, and from effects processing to room correction software. Audio software stands to benefit significantly from parallelism provided by algorithms that leverage multiple cores. Yet parallel programming presents many challenges and pitfalls for the programmer. These pitfalls are detrimental to efficiency. Developers with expertise in audio development are rarely the same ones with expertise in concurrent programming, and vice versa.

This paper presents an algorithm to automate parallelization such that the parallel code need only be written once, allowing concurrency to be abstracted away from the audio programmer. Thus, developers with differing expertise (i.e. systems and concurrent programming vs. DSP and filter design) focus on their areas of specialty without conflating these separate issues.

This division of labor is especially useful in software such as game systems, APIs, nonlinear editors, and end-user applications where high-level tasks can be run in parallel (tracks, effects, etc.). These sorts of applications commonly use buffer sizes ranging from as many as 2048 samples per buffer, to as few as 6 samples in low latency applications. With these relatively small buffer sizes, the cost of dividing buffers and reassembling the solution often outweighs the potential speedup of parallelization. Instead, these applications use very large numbers of buffers internally that are processed separately as atomic chunks.

This computation model holds for interactive applications such as game audio, digital audio workstations, immersive audio systems, and many computer music applications, to name a few. Therefore our focus is on these types of applications. Our approach remains valid for large parallel problems that can be modeled in a dataflow graph.

1.1. Graph Representation

Formally, a graph is a set of *nodes* that are connected by *edges*. See Figure 1 for a simple example. Note that the edges in the figure have arrows. These arrows indicate that the graph is *directed*. That is, there is a direction associated with each connection between nodes. For example, in Figure 1 there is an edge that goes from Node A to Node C, but there is no edge from Node C to Node A.

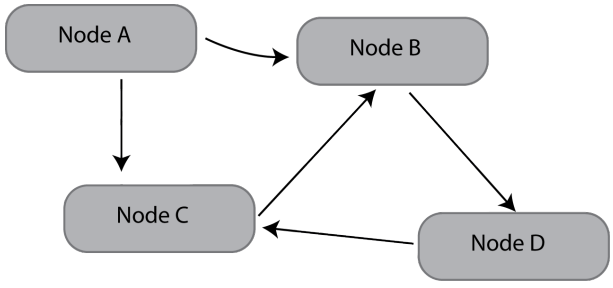

Figure 1 A simple directed graph containing a cycle.

A *path* through the graph is a sequential series of connected nodes. A path that returns to its starting point is called a *cycle*. The graph in Figure 1 contains a cycle formed by the edges connecting nodes B, C and D. Graphs that contain no cycles are *acyclic*. Directed graphs with no cycles are called Directed Acyclic Graphs (DAGs).

In the context of audio processing, graph nodes represent processes while edges represent signal routing or dataflow. As such, the simple example in Figure 2 implements high-pass filtered noise.

This paradigm is known as the dataflow programming model because paths through the graph represent the flow of data through processing steps.

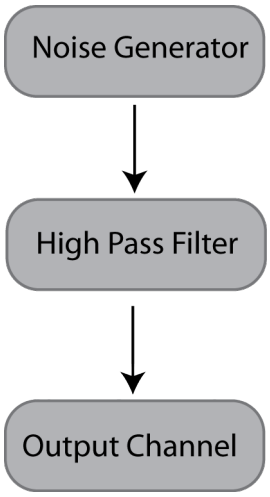

Figure 2 A simple dataflow example of high-pass filtered noise.

2. ALGORITHM

2.1. Overview

Our algorithm recursively searches for dependency paths through the graph. By dependency we mean a path upon which other paths depend to execute. For example in Figure 2 the high pass filter must wait for the output of the noise generator before it can filter that output. In this sense a dependency path is a set of nodes and edges whose corresponding outputs are required as inputs to other nodes.

Starting with source nodes (those with input valence 0), each source node n_i is stored and appended to the execution list of its parent (if it exists). Nodes with multiple parents are added to the execution list of the final parent to be processed, ensuring that nodes are only executed after all their inputs are available. Finally, the source nodes and their associated output edges are removed and the algorithm repeats (recursively) until there are no nodes remaining.

The output is a set of N execution lists that can be processed in parallel, where N is the number of source nodes without dependencies found during the search step. These lists can be concatenated if N is greater than the number of processors to avoid thread contention and improve performance. Below is a synopsized pseudocode of the algorithm:

Let G be an input directed acyclic graph.
Let n_i be the *i*th node in G
Let V_i be the input valence of n_i

Flow(G)
  //find all source nodes at current recursion level
  $\{n_i\}$ = *all Nodes in* G *with* $V_i = 0$
  //append n_i *to appropriate execution list*
  Process($\{n_i\}$)
  //store data paths
  Copy Outputs($\{n_i\}$) *to Inputs*(*Children*($\{n_i\}$))
  //remove sources and recurse
  Return Flow(G' = *Remove*($\{n_i\}$ *from* G))

This psuedocode assumes existence of a few simple functions:

- *Outputs*(*n*) takes as input a node and returns a set of output edges that correspond to the node's outputs.
- *Inputs*(*n*) takes in a node and returns its input edges.
- *Process*(*n*) checks to see if the input node has a parent in an existing execution list. If not, it creates a new execution list for that node. If it has more than one parent, it attaches it to the last parent.

The notation $\{n_i\}$ is shorthand to indicate iteration over a set of nodes here when used in an assignment or function call.

Once this algorithm completes, the graph G has been converted into a set of execution lists, ordered by dependency, so that sources complete before the nodes that require their output. It is worth noting that the algorithm is similar to the depth-first-search and topological sort algorithms [4] in computer science; however, those algorithms do not solve the problem of extracting parallel paths posed here.

2.2. Handling Details

As specified thus far, the algorithm is guaranteed to give correct output, but not optimal output. The ordering of the graph traversal is important. Fortunately, ensuring optimal behavior requires only two sorting steps.

In a given recursion level when we find a set of source nodes, certain orderings may be preferable to others. For example, it makes sense to process the node with the largest number of dependencies first, so that it does not delay other execution lists.

In the case of multiple source nodes with the same number of dependencies, we sort again by length of execution list, to maintain cache coherence early in the computations, so that branching dependencies are handled optimally.

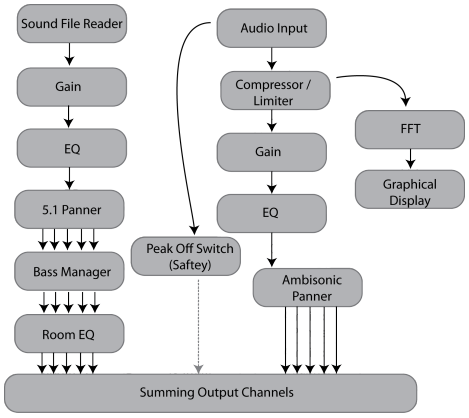

Figure 3 Example of complex dataflow program running multiple spatial audio systems with safety off switch (dotted line).

The algorithm assumes a directed acyclic graph. However, to implement a feedback process generally

requires graph cycles. We can extend the algorithm to allow feedback with some user input. If the edges that create feedback cycles are marked as such, the algorithm can ignore them during graph traversal. Once the execution lists are constructed, adding appropriate copies of input and output buffers to the nodes of the execution lists reinstates the feedback. This step also ensures that feedback happens only once per buffer, and no infinite loops form as a result.

The algorithm allows for node processes to be defined as graphs themselves, requiring only in-place expansion at runtime to ensure fully parallelized output.

3. CACHE PERFORMANCE

3.1. A little about CPU cache

Cache performance of the CPU is often the key limiting performance factor in host-based audio processing. While a complete discussion of CPU cache is beyond the scope of this paper, a brief and overview of the subject follows. For more information on CPU cache see [4].

A CPU cache is the small amount of extremely high speed memory located near the main processing units. This type of memory is much faster than main memory. Ideally, if we could fit our entire software program and its associated data into this high-speed memory, our programs would run very fast. However, such high-speed memory is extremely expensive, so CPUs generally use only a small amount of high-speed memory for the cache.

Data is read from main memory and stored in the cache before being used in CPU calculations. Results are also stored in the cache and written back to main memory. Operations to read or write main memory are very slow compared to CPU calculations and cache operations. Minimizing main memory reads and writes is a major component of software speed optimization. Therefore, an algorithm with good cache behavior is crucial for high performance.

The cache is used to store recently used data on the assumption that it will be used again shortly. Some caching schemes pull in memory from areas nearby requested locations in the hopes of pre-loading data that will be requested soon, improving cache performance and speed. This is called prefetching.

3.2. Algorithm Cache behavior

Because each buffer travels immediately from parent to child, our algorithm affords excellent cache *locality* by processing each buffer as far down the execution path as possible. Cache locality improves performance by reducing the number of fetches from memory and often allows vector instructions to process multiple samples simultaneously. Streaming data represents the worst case for cache locality because as data is used the stream replaces it with the next chunk. Thus, the best case scenario for an audio buffer is to have processes applied consecutively, rather than applying each process to consecutive buffers. This is exactly the ordering that our algorithm produces.

In the case of buffered audio applications, it is the number of internal voices, or total buffers processed per unit time that defines performance. Given this metric we can show that our algorithm offers *effectively* optimal performance[1]. We examine the efficacy of our cache performance by comparing it with perfect performance. Perfect performance assumes that whenever a datum is needed, it is already in cache. We can then compare this efficacy with the improvement offered by perfect performance to see if it affords another internal voice. If it does not, the effective difference between optimal performance and our algorithm difference is zero.

Since in practice the cache is always much smaller than the audio data set, we know that the complete audio data will not fit into cache. We also expect data pre-fetching to consistently load buffers into cache before they are requested because of the random-access nature of audio applications (looping, midi controllers, sequencing, etc.). In the practical sense, optimal cache performance will mean that each buffer is loaded fetched from memory the minimum possible number of times.

The best case for our algorithm is a set of independent execution lists that do not send data to each another. It is easy to see that this achieves optimal cache behavior: each buffer is fetched once, fits into cache through the duration of processing until it is output. But in practice most interesting dataflow graphs will involve some dependencies between execution paths.

[1] The general cache performance optimization problem is NP-complete, which means finding a globally optimal is not feasible. However, the simpler problem of optimality for buffered audio is solvable with this simpler performance metric.

Dependencies create two problems. First, when a thread reaches a node that does not have all its inputs ready, it has to wait on those inputs. The second problem is that the thread computing those inputs may be working at optimal cache efficiency, but on a different core from the first thread. So the cached data will have to be written to main memory or a (slower) shared cache, possibly thus undermining the performance benefits of our algorithms excellent cache performance up to that point.

Unfortunately there is no silver bullet to solve this problem; we can always find a pathological case. Consider the (unlikely in practice) case of a node that takes hundreds of inputs. This node's input data set will not fit into cache, so processing the node requires many memory reads. Optimizing the order of reads reduces to the intractable cache optimization problem.

The good news is that in practice the pathological cases are almost as unlikely as the optimal case above. The dependency sorting discussed in section 2.2 ameliorates the problem. Sorting paths by number and length of dependencies (*i.e.* effectively the number of nodes waiting on each source) means that paths in great demand execute early, minimizing the wait time of other threads.

With only a few execution paths, it is easy to create pathological cases where all worker threads are waiting on a single slow input or where the needed buffer is always in the wrong cache. But as the number of execution paths grows large (the very reason we are exploring automated parallelism) these cases become less likely and less detrimental to performance.

4. FINDINGS

We tested this algorithm for performance improvement on an 8-core x86 processor system. Our standard test subgraph is shown in Figure 4.

To test performance, we repeatedly add this subgraph to the global graph[2] until ARIA detects buffer under-run. The 10-Channel panner uses the Speaker-Placement Correction Amplitude Panning algorithm [6]. This node always outputs ten channels from each mono input, so each addition of the subgraph to the global graph adds 10 internal voices. We tested the algorithm on a 2007 8-core x86 family CPU. Performance peaked at 511 subgraphs (roughly 5100 internal voices) with only five worker threads. Although the CPU had eight cores, increasing the number of worker threads beyond five created a bottleneck at the mixer, which increased thread contention and decreased efficiency

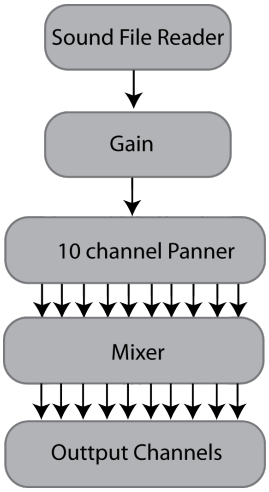

Figure 4 Our performance testing subgraph.

In an attempt to alleviate the bottleneck, we implemented an asynchronous multithreaded mixer. This mixer used a predetermined number of worker threads to sum its inputs, computed mixed its own output even while the worker threads generated input for it. This alleviated thread contention, but memory consumption at 500 subgraphs was so large that the cache performance suffered, the mixer had poor locality, and performance remained largely unchanged.

This outcome exemplifies the unintuitive performance tradeoffs inherent in concurrent programming and demonstrates the need to abstract (or automate) issues of concurrency from those of audio-specific programming.

5. CONCLUSIONS AND FUTURE WORK

We have presented an algorithm for automatic extraction of parallelism in dataflow graphs. This automation allows for the separation of software development tasks between audio and concurrent programming specialists, and abstracts details of concurrency by leveraging dataflow programming. In

[2] Note that the mixer node and the output channels are global, new instances are not created with each addition of the subgraph, but they are included in the figure for clarity.

our tests the algorithm exhibited excellent parallelism and cache performance.

We hope in the near future to stress test this algorithm by eliminating the bottlenecks at the mixing and output stage described above. We believe peak performance will occur when the number of worker threads is at or near the number of available cores. It will be interesting to gather empirical data as the number of cores available on a single system grows from a handful to dozens.

Similarly we hope to explore the questions of efficient concurrency and cache performance on architectures other than x86. For example CPUs such as the Itanium, Power6 intended for mainframes and servers generally offer advanced vector instruction support and significantly larger caches than their consumer-oriented counterparts. Though these CPUs generally offer lower clock rates than the x86 familiar to most users, it may well be the case that they offer the highest performance for audio processing in the future.

Finally we plan to investigate annotating graphs with timing information. We hope to explore whether knowledge of nodes' prior execution times will allow for improved scheduling.

6. ACKNOWLEDGEMENTS

The project or effort described here has been sponsored by the U.S. Army Research, Development, and Engineering Command (RDECOM). Statements and opinions expressed do not necessarily reflect the position n or the policy of the United States Government, and no official endorsement should be inferred.

7. REFERENCES

[1] http://puredata.info

[2] http://cycling74.com

[3] http://sonicflow.sourceforge.net

[4] Hennessy, J. L., D. A. Patterson, et al. (2007). Computer architecture : a quantitative approach. Boston, Morgan Kaufmann.

[5] Sadek, R. "A Host-Based Real-Time Multichannel Immersive Sound Playback and Processing System". AES 117th Convention, New York, 2004.

[6] Sadek, R and Kyriakakis, C. "A Novel Multichannel Panning Method for Standard and Arbitrary Loudspeaker Configurations". AES 117th Convention, New York, 2004.

[7] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. 2009 Introduction to Algorithms, Third Edition. The MIT Press.