Hearing Program Behavior with TSAL

Joel C. Adams, PhD
Department of Computer Science
Calvin University
Grand Rapids, MI USA 49546
adams@calvin.edu

Mark C. Wissink
Department of Computer Science
Calvin University
Grand Rapids, MI USA 49546
mcw33@students.calvin.edu

***Abstract*—Much work has been done in the area of real-time algorithm visualization, in which a program produces a graphical representation of its behavior as it executes. In this lightning talk, we examine the relatively uncharted territory of real-time *audialization*, in which a program produces a sonic representation of its behavior as it executes. Such work seems apt to be beneficial for auditory learners, especially those with visual disabilities. To support this exploration, we have created the Thread Safe Audio Library (TSAL), a platform-independent, object-oriented C++ library that provides thread-safe classes for mixing, synthesizing, and playing sounds. Using TSAL, we can create an audialization by taking a working program and adding library calls that generate behavior-representing sounds in real time. If a program is multithreaded, each thread can play distinct sounds, allowing us to hear the multithreaded behavior. This lightning talk provides an overview of TSAL and demonstrates several audializations, including the Producers-Consumers Problem, Parallel MergeSort, and others.**

Keywords—audialization, audio, concurrent, hearing, multicore, multithreading, parallel, sound, synchronization, threads

I. Introduction

Much has been done to create visualizations for common sequential algorithms (e.g., [7] provides a literature survey), and some have worked on concurrent/parallel visualizations, such as [1, 2, 4]. Visualizations provide a means of *seeing* program behavior, and since many students are visual learners, visualization makes sense as a starting point for sensory depictions of program behavior.

However, not everyone is a visual learner. Some students are auditory learners who learn best by hearing; others are tactile learners who learn best by touch and manipulation, and so on. In particular, visualizations offer limited benefits for students with visual disabilities, which suggests we explore other sensory means of representing a program's behavior. In this work, we focus on the sense of hearing and the use of sound.

When a typical laptop runs a compute-intensive multithreaded program, the extra heat generated by the active cores causes the laptop's fan to start. The resulting white noise provides a crude sonic indicator that the program is doing something atypical. This is an unintentional sonic side effect of the hardware engineering.

We instead propose to intentionally add sound-generating calls to a program in a way that lets us *hear* its behavior. We describe the resulting sonic effect as an **audialization**—a sonic representation of the program's behavior—similar to a visualization, but for hearing instead of seeing.

We have been unable to find any published research related to *hearing program behavior*. Some user-interface researchers have published work on the use of *ear-cons*—the sonic equivalent of icons—interface widgets that emit distinctive sounds as one moves the mouse over them, for people with visual impairments (e.g., [5]). CS education researchers have also published work on using sound-file processing to motivate novice programmers (e.g., [6]), but we have been unable to find any published work on intentionally incorporating sound into a program in order to audibly represent the program's behavior.

However, there is a precedent for such work. In their biography of Claude Shannon titled *A Mind at Play*, Jimmy Soni and Rob Goodman relate the following story from when Shannon visited Alan Turing in London in 1950:

> "Even decades later, Shannon would recall one of Turing's inventions:
>
> *So I asked him what he was doing. And he said he was trying to find a way to get better feedback from a computer so he would know what was going on inside the computer. And he'd invented this wonderful command. See in those days, they were working with individual commands. And the idea was to discover good commands.*
>
> *And I said, what is the command? And he said, the command is to put a pulse to the hooter, put a pulse to the hooter. Now let me translate that. A hooter … in England is a loudspeaker…*
>
> *Now what good is this crazy command? Well the good of this command is that if you're in a loop, you can have this command in that loop and every time it goes around the loop it will put a pulse in and you will hear a frequency equal to how long it takes to go around that loop. And then you can put another one in some bigger loop and so on. And so you'll hear all of this coming on and you'll hear this 'boo boo boo boo boo boo boo' and his concept was that you'd soon learn to listen to that and know when it got hung up in a loop or something else or what it was doing all the time, which he'd never been able to tell before*." [8]

Thus, years before the development of the compiler, debugger, graphics, or any of the other modern programming conventions, Turing had the idea of creating a sound-pulse machine instruction he could use to *hear* a program executing. The result would be a new sonic language that he could use to profile correct programs and debug incorrect ones.

This idea seems to have been lost in the nearly 70 years since Turing had it. We propose to revive this practice.

This work funded by NSF DUE #1822486.

II. Some Example Audializations

It is challenging to describe audializations using words, because it is difficult to describe dynamic sound without hearing it. This lightning talk will include live-demos of the following:

A. The Producers-Consumers Problem

The Producers-Consumers problem is often solved by implementing a bounded buffer as a monitor that provides mutually-exclusive put() and get() operations. This simplifies the problem because, from the perspective of an item:

1. A producer creates an item (1a), and then uses put() to deposit the item into the buffer (1b);
2. the item sits in the buffer, until…
3. a consumer uses get() to retrieve the item from the buffer (3a), and then consumes that item (3b).

To audialize this problem, we use sound-frequency to indicate the state of an item. More precisely, when a producer produces an item, it generates a unique, low-frequency tone for that item (step 1a). When the producer acquires the buffer's lock and the item is put into the buffer, this tone increases to a mid-frequency pitch (step 1b). This middling tone is sustained so long as the item remains in the buffer (step 2). When a consumer acquires the buffer's lock and gets the item from the buffer, the tone again increases to a high-frequency pitch (step 3a), which stops when the consumer finishes consuming the item (step 3b).

Since each widget is associated with a unique tone when it is created and multiple producers are creating widgets at the same time, there are multiple, distinct low-frequency tones playing at the same time, providing an audial representation of the concurrency. However the get() operation is mutually exclusive, so only one tone at a time is ever increasing from the low-to-mid frequency, thus letting students hear the mutually exclusive nature of the operation. When multiple items are in the buffer at the same time, there are multiple mid-frequency tones playing simultaneously, letting students hear the buffer storing the items. The get() operation is also mutually exclusive, so only one tone at a time is ever increasing from the mid-to-high-frequency, letting students hear the mutually exclusive nature of that operation. Finally, the buffer is a monitor, so only one tone at a time will ever be increasing through the low-to-mid frequencies or the mid-to-high-frequencies, allowing students to hear the mutually exclusive nature of monitor operations.

B. MergeSort (Sequential and Parallel)

The MergeSort algorithm is sufficiently easy to parallelize that at least one popular CS2/*Data Structures* textbook includes a parallel version of the algorithm [3].

In general, a sorting audialization might represent the magnitude of the item currently being accessed using a sound's frequency—the larger the value of the item being accessed, the higher the tone. Given a sequence of random items, the audialization of a sequential algorithm thus initially sounds like a random jumble of tones, with one tone sounding at a time as the algorithm compares the items. By contrast, the audialization of a multithreaded/parallel algorithm initially sounds much more chaotic than the sequential algorithm, as the different threads access different items simultaneously. However, if the sequence becomes more ordered as the algorithm runs, then the sounds being played will correspondingly become more ordered.

The MergeSort algorithm recursively divides the sequence without accessing any of its items until a subsequence is trivially sorted. Most of the work is done in a bottom-up fashion, when two sorted subsequences are merged into a single sorted sequence as the recursion unwinds. Initially, the audialization sounds random, but as unsorted subsequences become ordered, the tones become increasingly ordered, and each time two sorted subsequences are merged into a sorted sequence, the audialization produces an ascending sequence of tones.

An audialization of the sequential MergeSort algorithm does all of this work with a single thread, so one tone is played at a time. The overall sequence of tones oscillates between chaos and order as the algorithm oscillates between recursively dividing the sequence and merging sorted subsequences. As the algorithm runs, these sorted subsequences become longer, gradually increasing the order. One can hear a clear delineation between when the first half the original sequence is sorted, and when sorting begins on the second half of the sequence. After the second half is sorted, one hears a mostly ascending sequence of tones as the two sorted halves are merged.

By contrast, an audialization of the parallel MergeSort algorithm spawns off new threads during the winding phase of the recursion, so that multiple threads are simultaneously merging the sorted subsequences into sorted sequence, causing multiple tones to be played simultaneously. For example, if eight threads are being used, there will initially be eight different ascending tones being played simultaneously as those eight threads merge sixteen sorted subsequences. Half of those threads then terminate, leaving four threads to merge the remaining eight subsequences, during which four ascending tones will be heard at once. Half of those threads then terminate, leaving two threads to merge the four sorted subsequences, during which two ascending tones will be heard. When half of those threads terminate, there will be a single thread merging the final two sorted subsequences, producing a single sequence of ascending tones. In short, a student can hear the difference in the sequential vs the parallel algorithms. Since the parallel version also provides a significant speedup, an audialization also lets a student hear how much faster the parallel version solves the sorting problem compared to the sequential version.

III. Conclusion

In this lightning talk, we present audializations—programs that provide sonic representations of their behavior—to provide a different way of experiencing program execution. These audializations provide a means for auditory learners (or people with visual impairments) to hear:

- the difference between sequential vs parallel execution.
- concurrent abstractions such as mutual exclusion, synchronization, blocking behavior, and so on.
- parallel topics such as speedup and recursive decomposition.
- other topics that are still in development.

Our audializations are created using the Thread Safe Audio Library (TSAL), a new tool that allows multiple threads to simultaneously play sounds in real time. By letting us hear a program (parallel or sequential) executing, audialization provides a modern-day realization of and extension to Turing's "*put a pulse to the hooter*" mechanism. TSAL thus creates a new language mechanism by which programs can "speak" to us.

References

[1] J. Adams, P. Crain, C. Dilley, C. Hazlett, E. Koning, S. Nelesen, J. Unger. TSGL: A Tool for Visualizing Multithreaded Behavior, *Journal of Parallel and Distributed Computing*, Volume 118, Issue P1, Aug 2018, pp. 233-246.

[2] S. Carr, J. Mayo, and C.K. Shene. ThreadMentor: a Pedagogical Tool for Multithreaded Programming, Journal on Educational Resources in Computing (JERIC), 3(1), March 2003, Article 1.

[3] N. Dale, C. Weems and T. Richards. *C++ Plus Data Structures* (6/e). Jones & Bartlett, 2018, pp. XX.

[4] A. Danner, T. Newhall, K. Webb. ParaVis: A Library for Visualizing and Debugging Parallel Applications, *Proc. of 9th NSF/TCPP Workshop on Parallel and Distributed Computing Education* (EduPar-19), in conjunction with IEEE IPDPS'19, Rio de Janeiro, Brazil May 2019.

[5] S. Garzonis, S. Jones, T. Jay, E. O'Neill. Auditory icon and earcon mobile service notifications: intuitiveness, learnability, memorability and preference. Proc. SIGCHI Conf. on Human Factors in Computing Systems, April 2009. pp. 1513-1522.

[6] M. Guzdial, D. Ranum, B. Miller, B. Simon, B. Ericson, S. Rebelsky, J. Davis, K. Deepak, D. Blank. Variations on a theme: role of media in motivating computing education. *Proc. of the 41st ACM SIGCSE Symposium on CS Education*, March 2010, pp. 66-67.

[7] C. Shaffer, M. Cooper, A. Alon, M. Akbar, M. Stewart, S. Ponce, S. Edwards. Algorithm Visualization: The State of the Field. *ACM Transactions on Computing Education*, 10(3), August 2010. Article 9.

[8] J. Soni and R. Goodman, *A Mind At Play*, Simon & Schuster, 2017, pp. 108-109.