Is Music Audio Processing Embarrassingly Parallel?

Roger B. Dannenberg
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, USA
rbd@cs.cmu.edu

Designing languages and systems for many-core computers is in some respects just like designing for unprocessors. The challenge is, on the one hand, to develop an efficient organization of data, hardware, and software to accomplish the desired computation and, on the other hand, to invent a notation that is expressive and can be translated automatically to executable instructions. With many-core computers, it is far from obvious how to organize computation to run efficiently, but it seems important to think about this aspect of the problem. If nothing else, we need to have some idea of what a running program will look like so that we can begin to imagine how we might express that program in a high-level language. It is also good to know what a near-optimal program looks like in order to evaluate alternatives. Note that we routinely sacrifice execution efficiency to make programming easier. Careful assembly language programming can often yield a 3x to 10x speedup over compiled languages, but we usually forgo the extra performance to avoid the cost of developing and maintaining carefully tuned, hardware-specific implementations. It may be that with many-core processors, we will routinely give up a factor of 100 in performance in order to simplify programming.

It has been suggested that audio processing is “embarrassingly parallel,” meaning that the computation naturally decomposes into many independent tasks, making it ideal for a parallel implementation. In music synthesis, for example, one can imagine synthesizing each instrument in parallel and then mixing the resulting sample streams [1, 2]. This is encouraging, but I believe it is worth a more careful analysis.

Music audio processing is often performed using a synchronous data-flow graph. Audio samples enter the graph from audio input devices and other sources, flow through nodes in the graph, and emerge as outputs to be delivered to an audio output device. Nodes, also called unit generators, take in blocks of samples from upstream nodes and output blocks of samples to downstream nodes. With a uniprocessor, we process the input nodes first, internal nodes second, and the output nodes last. This guarantees that samples flow directly from input to output without buffering or other delay. In more formal terms, the directed acyclic graph establishes a partial order, and a topological sort determines the order of execution for the nodes (unit generators).

There are two ways to extract parallelism from this model. First, the topological sort can tell us where unit generators can be executed in parallel. This might yield a lot of parallelism, but parallel threads must synchronize where their signals are combined, and the slowest path will set the pace. Synchronization and data motion will also slow things down.

The second source of parallelism is pipelining, where the graph is partitioned into stages separated by buffers. While one stage of the pipeline is working on samples for output at time n (measured in samples), the previous stage is working in parallel on samples for output at time n + b, where b is the block length, the number of samples computed at each stage. Unfortunately, adding stages to the pipeline adds latency to the computation because samples are buffered between pipeline stages. For real-time audio processing, we will need to reduce latency by making the block length shorter as the number of pipeline stages increases. This reduces efficiency (the reason we used block-based computation in the first place), and synchronization between stages adds overhead, so there are practical limits and possibly difficult optimization problems inherent in this approach. Consider that adding two samples in the same cache will be much faster than sending the samples to another processor, so there is a point at which spreading work across processors will not provide any speedup.

Finally, music computation is rarely static, so organizing computation to be rapidly reconfigurable will require careful coordination of processors and perhaps additional overhead. Music audio computation can also involve spectral representations, control processing, file I/O, and other additions to the simple uniform graph model, and these must all be considered.

In conclusion, there is great potential for parallel music audio computation. Realizing this potential in practical systems will require a careful investigation of possible run-time organizations. The efficiency of many-core audio processing will depend critically on the ability to allocate and synchronize processors and for processors to exchange data efficiently.

REFERENCES
