ABSTRACT

Two ongoing research projects at Grame aim at providing efficient solution for audio applications on many/multi core systems. The first one is Jackdmp, a C++ version of the Jack low-latency audio server for multi-core machines. The second is Faust, a compiled DSP language with auto parallelization features. This paper will briefly describe these two projects focusing on multi-core related questions.

1. JACK

JACK is a multi-platform low-latency audio server. It can connect a number of different applications to an audio device, as well as allowing them to share audio between themselves. The first versions of Jack were based on a sequential activation mechanism finely tuned for mono-core machines, but unable to take advantage of modern multi-core machine. The recent Jackdmp version aims at removing these limitations. The activation system has been changed for a data flow model and lock-free programming techniques for graph access have been used.

1.1. Natural parallelism

In a Jack server like system, there is a natural source of parallelism when Jack clients depend of the same input and can be executed on different processor at the same time. The main requirement is then to have an activation model that allows the scheduler to correctly activate parallel runnable clients.

![Client graph](image)

**Figure 1.** Client graph: Client A and B could be executed at the same time, C must wait for A and B end, D must wait for C end.

1.2. Data-flow activation

A graph of Jack clients typically contains sequential and parallel sub-parts (Fig 1). When parallel sub-graph exist, clients can be executed on different processors at the same time. A data-flow model can be used to describe this kind of system: a node in a data-flow graph becomes runnable when all inputs are available. Each client uses an activation counter to count the number of input clients which it depends on. The state of client connections is updated each time a connection between ports is done or removed.

Activation will be transferred from client to client during each server cycle as they are executed: a suspended client will be resumed, executes itself, propagates activation to the output clients, go back to sleep, until all clients have been activated.

1.3. Pipelining for sequential graphs

Sequential graphs can also take benefit of multi-core machines using pipelining techniques. The idea is to divide the audio buffer of size D into N sub parts and run the entire graph with N buffers of D/N size. For example taking a driver buffer size of 1024 frames, with N = 4, the graph is processed with buffers of 1024/4 = 256 frames.


Running the graph with smaller buffers means more context switches with their associated cost. Thus there is a balance to find between the size of the sub buffer and the number of available processors. The overall benefit is to lower the worst-case execution ending date in a given cycle, thus possibly executing more CPU hungry graph and still avoiding audio xruns.

2. FAUST

Faust is a compiled language for real-time signal processing and synthesis that targets high-performance audio applications and plugins. The programming model of Faust combines a strong mathematical semantic with a very expressive block-diagram syntax. Faust programs are translated by the Faust compiler into C++ programs. The compiler try to synthesize the most efficient C++ implementa-

---

1 The data-flow model still works on mono-processor machines and will correctly guaranty a minimum global number of context switches like the "pre-sorting step" model.
tion thanks to various symbolic techniques. The resulting code works at sample level, is self-contained and doesn’t depend on any DSP runtime library. Moreover it can usually compete with, and sometimes outperform, hand-written C/C++ code.

We recently started working on an extension of the compiler to generate parallel programs. In the following paragraphs we briefly illustrate this extension.

2.1. A very simple example

As example we will use a very simple Faust program: two 1-pole filters in parallel connected to an adder:

\[
\begin{align*}
\text{filter}(c) &= *(1-c) + \neg *(c); \\
\text{process} &= \text{filter}(0.9), \text{filter}(0.9) : +;
\end{align*}
\]

![Block-diagram representation automatically generated by Faust compiler using -svg option](image)

2.2. Generation of C++ code

The Faust compiler starts by computing the mathematical equations of the output signals. These equations are then typed, simplified and normalized. After a phase of common sub expression discovery, the C++ code is generated. In our case we obtain the following code:

```cpp
for (int i=0; i<count; i++) {
    fRec0[i] = 0.1f*input1[i] + 0.9f*fRec0[i-1];
}
for (int i=0; i<count; i++) {
    fRec1[i] = 0.1f*input0[i] + 0.9f*fRec1[i-1];
}
for (int i=0; i<count; i++) {
    output0[i] = fRec1[i] + fRec0[i];
// post processing
    fRec1[i] = fRec1[i];
    fRec0[i] = fRec0[i];
}
```

2.3. Parallelized Code using OpenMP

With the new -omp option, the Faust compiler reorganizes the code to exhibit all the potential parallelism (basically a topological sort on the dependencies graph) and places OpenMP pragmas that indicate the C++ compiler how to parallelize the code.

```cpp
#pragma omp sections
{
    #pragma omp section
    for (int i=0; i<count; i++) {
        fRec0[i] = 0.1f*input1[i] + 0.9f*fRec0[i-1];
    }
    #pragma omp section
    for (int i=0; i<count; i++) {
        fRec1[i] = 0.1f*input0[i] + 0.9f*fRec1[i-1];
    }
    #pragma omp for
    for (int i=0; i<count; i++) {
        output0[i] = fRec1[i] + fRec0[i];
    }
}
```

As you can see in the generated code the two filters are computed in parallel and added in a parallel loop.

2.3.1. Efficiency

Thanks to its semantic, parallelization of Faust code is easy. What is much more difficult is the generation of efficient parallel code. If in some cases the performances on a core 2 duo can nearly double, in some other parallelization can have a negative impact. Too fine grained parallelism can increase the memory bandwidth demand (a crucial resource in SMP architectures) as well as the multi-threading overheads. The challenge is to find the optimal balance between potential parallelism and these overheads.

3. RESOURCES


---

2 OpenMP is an Application Program Interface (API), jointly defined by a group of major computer hardware and software vendors that provides a portable, scalable model for developers of shared memory parallel applications