Multicore technologies in Jack and Faust

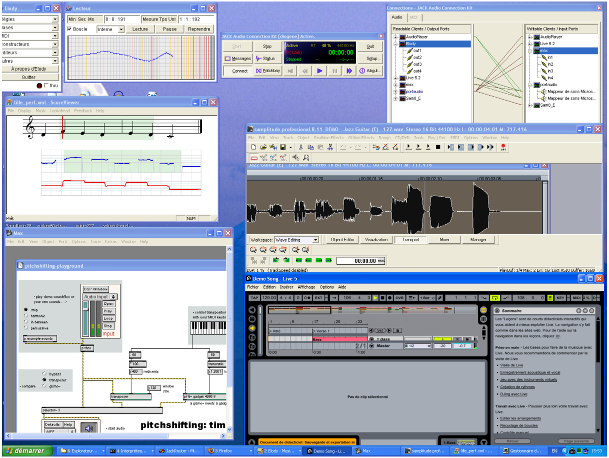
Y. Orlarey, S. Letz, D. Fober
Grame

ICMC'08, Belfast, August 2008

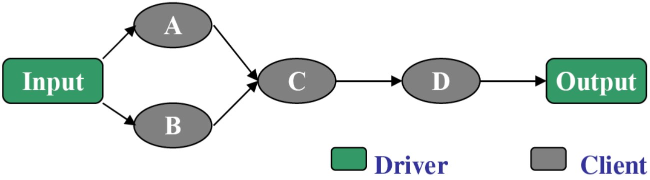
Outline

Jack Audio Server

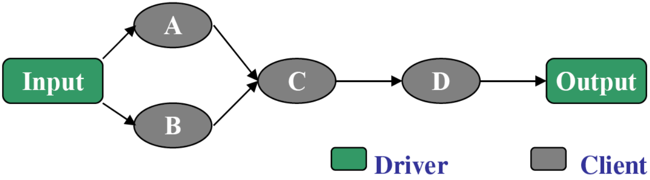
Jack is a low latency audio server that runs on Linux, Macosx and Windows



Original Jack Activation Model

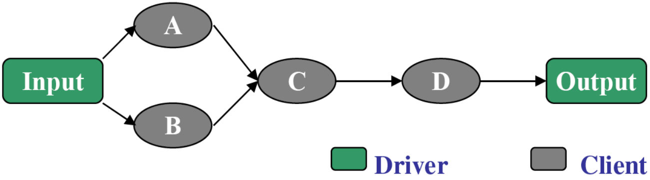


Original Jack Activation Model



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 machines.

Original Jack Activation Model



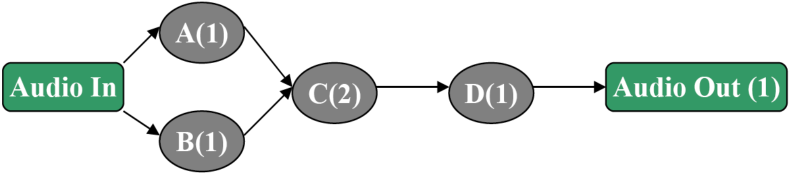
The first versions of Jack were based on a sequential activation mechanism finely tuned for mono-core machines, but unable to take advantage of modern multi-core machines.
A “topological sort” was used to find an activation order (A, B, C, D or B, A, C, D here)

New Semi-Dataflow Activation Model

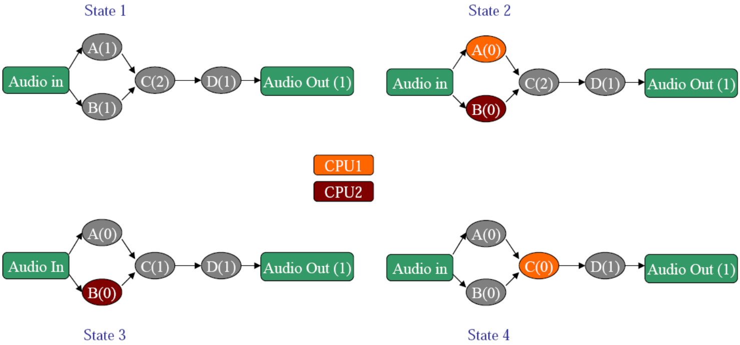
In the new semi-dataflow model an application in the graph becomes *runnable* when all inputs are available. Each client uses an *activation counter* to count the number of input clients which it depends on.

New Semi-Dataflow Activation Model

In the new semi-dataflow model an application in the 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.



Semi-Dataflow Activation Model in action



Various types of activation

Jack proposes various types of activations

Various types of activation

Jack proposes various types of activations

1. *Synchronous*

Various types of activation

Jack proposes various types of activations

1. *Synchronous*
2. *Asynchronous*

Various types of activation

Jack proposes various types of activations

1. *Synchronous*
2. *Asynchronous*
3. *Free-wheel*

Various types of activation

Jack proposes various types of activations

1. *Synchronous*
2. *Asynchronous*
3. *Free-wheel*
4. *Pipelined*

Outline

Outline

1. Jack
2. Faust
   - Overview
   - Parallel code generation
   - Performances
3. Conclusion

FAUST : Functional AUdio Stream

A programming language for realtime signal processing

Design Principles :

FAUST : Functional AUdio Stream

A programming language for realtime signal processing

Design Principles :

1. *Functional approach* : A purely functional programming language for real-time signal processing

FAUST : Functional AUdio Stream

A programming language for realtime signal processing

Design Principles :

1. *Functional approach* : A purely functional programming language for real-time signal processing
2. *Strong formal basis* : A language with a well defined formal semantic

FAUST : Functional AUdio Stream

A programming language for realtime signal processing

Design Principles :

1. *Functional approach* : A purely functional programming language for real-time signal processing
2. *Strong formal basis* : A language with a well defined formal semantic
3. *Efficient compiled code* : The generated C++ code should compete with hand-written code

FAUST : Functional AUdio Stream

A programming language for realtime signal processing

Design Principles :

1. *Functional approach* : A purely functional programming language for real-time signal processing
2. *Strong formal basis* : A language with a well defined formal semantic
3. *Efficient compiled code* : The generated C++ code should compete with hand-written code
4. *Easy deployment* : Multiple native implementations from a single Faust program

Very Simple Example

A Faust program describes a *signal processor*, a mathematical function that maps input signals to output signals.

Example

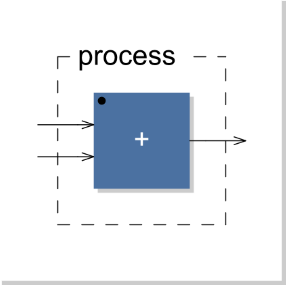
```
process = +;
```

Very Simple Example

A Faust program describes a *signal processor*, a mathematical function that maps input signals to output signals.

Example

```
process = +;
```



Stereo Pan

Faust syntax is based on a *block diagram algebra* :

`(A:B), (A,B), (A<:B), (A:>B), (A~B)`

Stereo Pan

```
p = hslider("pan", 0.5, 0, 1, 0.01);
process = _ <: *(sqrt(1 - p)), *(sqrt(p));
```

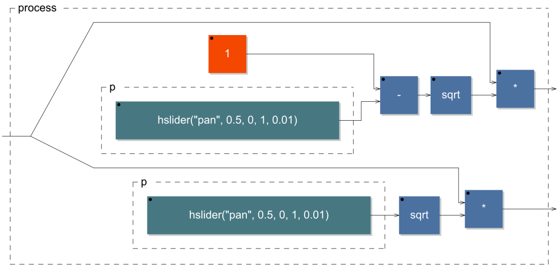
Stereo Pan

Faust syntax is based on a *block diagram algebra* :

(A:B), (A,B), (A<:B), (A:>B), (A~B)

Stereo Pan

```
p = hslider("pan", 0.5, 0, 1, 0.01);
process = _ <: *(sqrt(1 - p)), *(sqrt(p));
```



Easy Deployement

Several audio plateforms are supported

Thanks to specific *architecture files* the same Faust code can be used to generate a variety of applications or plugins :

Easy Deployement

Several audio plateforms are supported

Thanks to specific *architecture files* the same Faust code can be used to generate a variety of applications or plugins :

1. LADSPA
2. Max/MSP
3. Puredata
4. Q
5. SuperCollider
6. VST
7. Jack
8. Alsa
9. OSS

Some environments have Faust embedded

Some environments have Faust embedded

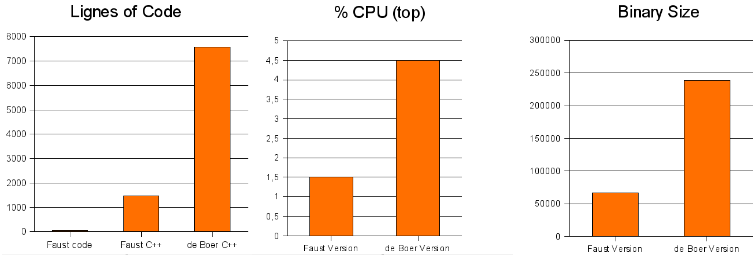
1. Snd-Rt : http://www.notam02.no/arkiv/doc/snd-rt/
(see Kjetil Matheussen poster, August 27 - session 2)

Some environments have Faust embedded

1. Snd-Rt : http://www.notam02.no/arkiv/doc/snd-rt/
   (see Kjetil Matheussen poster, August 27 - session 2)
2. CLAM : http://clam.iua.upf.edu/

Efficient code generation (monoprocessor)

Comparing Marteen de Boer's Tapiir with the equivalent Faust Tapiir



Outline

two 1-pole filters in parallel connected to an adder

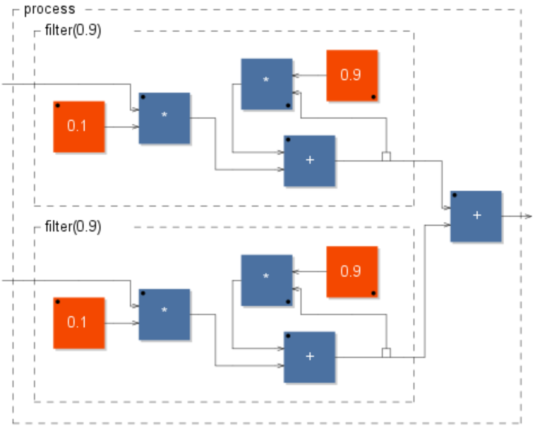
As an example we will use a very simple Faust program : :

two 1-pole filters in parallel connected to an adder

```
filter(c) = *(1-c) : + ~ *(c);
process = filter(0.9), filter(0.8) : +;
```

two 1-pole filters in parallel connected to an adder

Block-diagram representation automatically generated by Faust compiler using −svg option



The generated C++ code

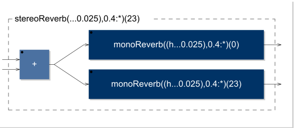
The Faust compiler can produce 3 types of C++ code :

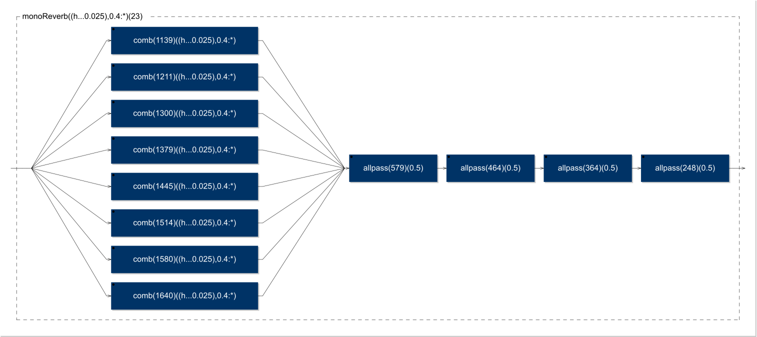
1. scalar code (default mode) see
2. vector code (`-vec` option) see
3. parallel code (`-omp` option) see

Outline

1. Jack
2. Faust
   - Overview
   - Parallel code generation
   - Performances
3. Conclusion

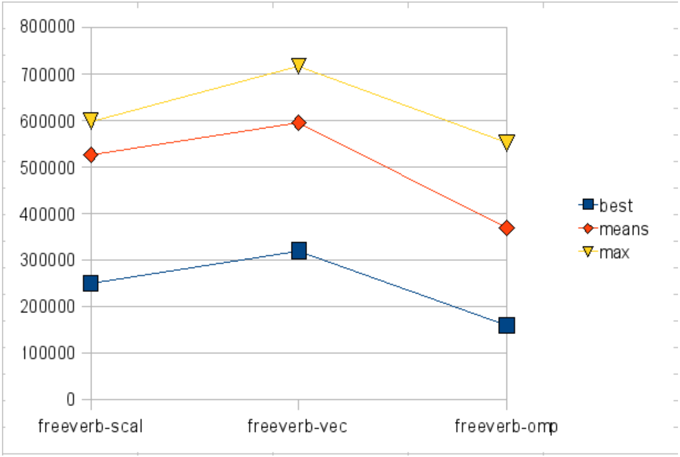
Freeverb


stereoReverb(...0.025),0.4:*)(23)
+
monoReverb((h...0.025),0.4:*)(0)
monoReverb((h...0.025),0.4:*)(23)


monoReverb((h...0.025),0.4:*)(23)
comb(1139)((h...0.025),0.4:*)
comb(1211)((h...0.025),0.4:*)
comb(1300)((h...0.025),0.4:*)
comb(1379)((h...0.025),0.4:*)
comb(1445)((h...0.025),0.4:*)
comb(1514)((h...0.025),0.4:*)
comb(1580)((h...0.025),0.4:*)
comb(1640)((h...0.025),0.4:*)
allpass(579)(0.5)
allpass(464)(0.5)
allpass(364)(0.5)
allpass(248)(0.5)

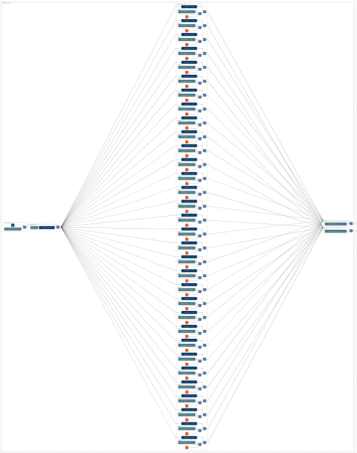
Freeverb on Vaio VGN-SZ3VP (2 cores)

Best speedup for the parallel version: 2, average: 1.62



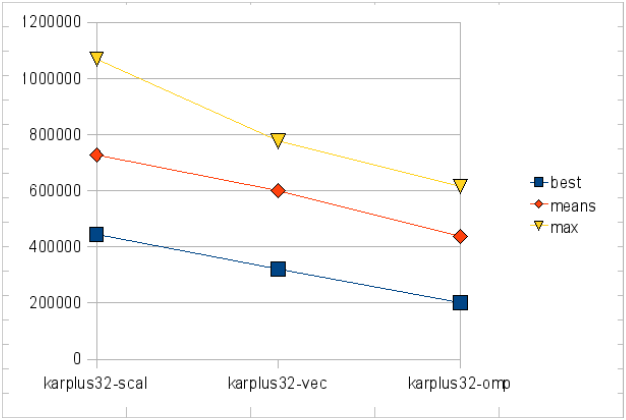
Karplus 32

32 sligtly detuned Karplus-strong strings mixed on a stereo bus.



Karplus 32 on Vaio VGN-SZ3VP (2 cores)

Best speedup for the parallel version: 1.59, average: 1.37

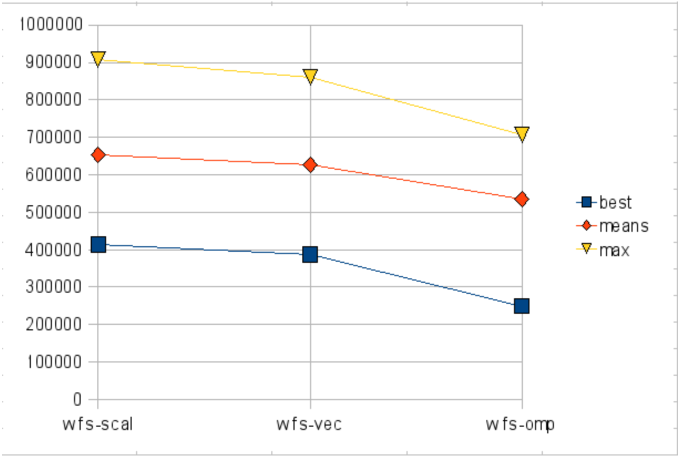


Wave Field Synthesis

Simple 8 channels Wave Field Synthesis.

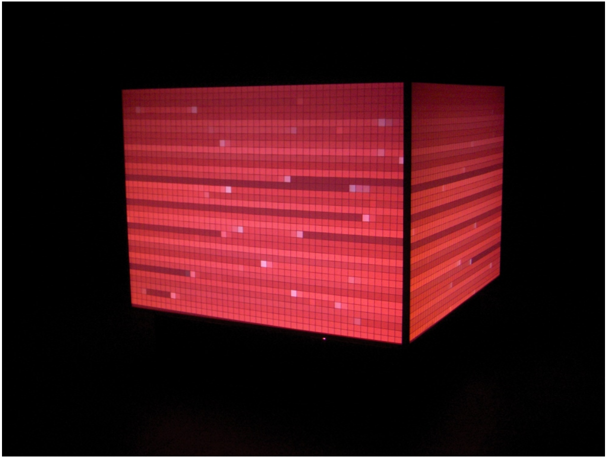
Wave Field Synthesis on Vaio VGN-SZ3VP (2 cores)

Best speedup for the parallel version: 1.55, average: 1.17



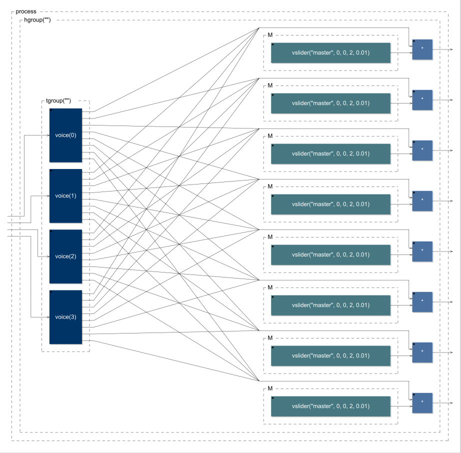
Sonik Cube

Sound and Visual Installation (Trafik/GRAME, 2006) :
3mx3mx3m cube reacting to sounds in an audio feedback space



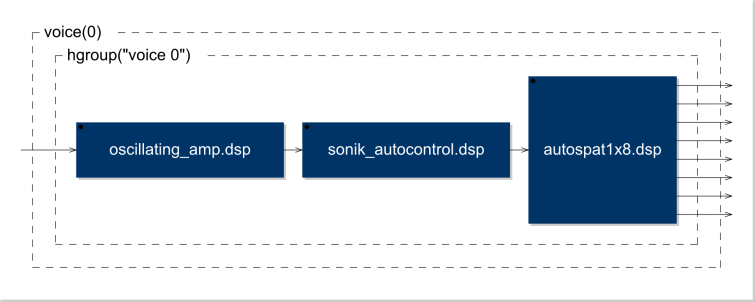
Ethersonik

Toplevel block-diagram of Ethersonik, the audio software of Sonik Cube



Ethersonik

Voice block-diagram



Ethersonik

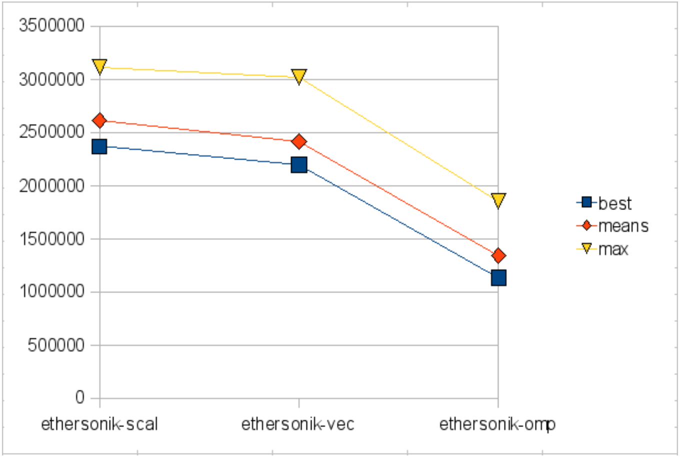
Ethersonik source code

source code

```
voice (v) = hgroup("voice %v",
                    component("oscillating_amp.dsp")
                  : component("sonik_autocontrol.dsp")
                  : component("autospat1x8.dsp")
              );

M = vslider ("master",0,0,2,0.01);

process = hgroup("",
                 tgroup("", par(i,4,voice(i)))
              :> par(i,8,*(M)));

```

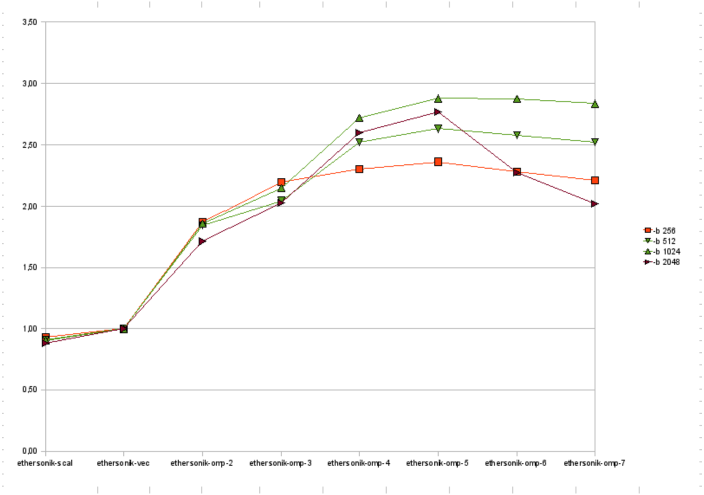
Ethersonik on Vaio VGN-SZ3VP (2 cores)

Best speedup for the parallel version: 1.94, average: 1.79



Ethersonik on Macpro (8 cores)

Best speedup (cores: 5, vectors: 1024): 3,09, average: 2.88



Outline

To Sum Up

To Sum Up

1. There is a lot of *task* + *data* parallellisms to exploit in audio applications

To Sum Up

1. There is a lot of *task* + *data* parallellisms to exploit in audio applications
2. Signal languages with a *simple and well defined formal semantic* are easy to parallelise. It's the way to go.

To Sum Up

1. There is a lot of *task* + *data* parallellisms to exploit in audio applications
2. Signal languages with a *simple and well defined formal semantic* are easy to parallelise. It's the way to go.
3. OpenMP is a simple and effective solution for multicore machines

To Sum Up

1. There is a lot of *task* + *data* parallellisms to exploit in audio applications
2. Signal languages with a *simple and well defined formal semantic* are easy to parallelise. It's the way to go.
3. OpenMP is a simple and effective solution for multicore machines
4. But efficient parallelisation is not *that* easy to achieve

To Sum Up

1. There is a lot of *task* + *data* parallellisms to exploit in audio applications
2. Signal languages with a *simple and well defined formal semantic* are easy to parallelise. It's the way to go.
3. OpenMP is a simple and effective solution for multicore machines
4. But efficient parallelisation is not *that* easy to achieve
5. Memory bandwith is a major limitation in SMP machine

Ressources

1. Jack http://jackaudio.org
2. Jackdmp http://www.grame.fr/~letz/jackdmp.html
3. Faust http://faust.grame.fr
4. OpenMP http://openmp.org/wp/
5. Snd-Rt http://www.notam02.no/arkiv/doc/snd-rt/
6. CLAM http://clam.iua.upf.edu/